GENERALIZATIONS OF MULTI-AGENT PATH FINDING PROBLEM FOR INCREMENTAL ENVIRONMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FATİH SEMİZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JULY 2022

Approval of the thesis:

GENERALIZATIONS OF MULTI-AGENT PATH FINDING PROBLEM FOR INCREMENTAL ENVIRONMENTS

submitted by **FATİH SEMİZ** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering, Çankaya University _____

Prof. Dr. Faruk Polat
Computer Engineering, METU _____

Prof. Dr. Kemal Leblebicioğlu
Electrical and Electronics Engineering, METU _____

Assoc. Prof. Dr. Erol Şahin
Computer Engineering, METU _____

Assoc. Prof. Dr. Mehmet Tan
Computer Engineering, TOBB ETU _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Fatih Semiz

Signature :

ABSTRACT

GENERALIZATIONS OF MULTI-AGENT PATH FINDING PROBLEM FOR INCREMENTAL ENVIRONMENTS

Semiz, Fatih
Ph.D., Department of Computer Engineering
Supervisor: Prof. Dr. Faruk Polat

July 2022, 100 pages

Multi-Agent Path Finding problem (MAPF) is finding a path for multiple agents from a list of starting locations to a list of goal locations in such a way that the agents' routes do not pass through the same location at the same time. The problem occurs in real-world during the transporting packages in warehouse environments with robots moving on rails, cleaning closed areas with cleaning robots, and protecting areas with multiple robots etc. It is usually sufficient to use discrete maps to express these problems. However, unlike the standard MAPF setting, in these problems there may be a need to replan agent paths while the movement of the agents continues. The need to replan agent paths may be due to the following reasons: packages falling on the road, passing of external vehicles, or new tasks added to the problem while the problem continues. Such situations can be better expressed with an incremental MAPF problem structure. In this thesis, we describe a MAPF variation where certain nodes on the map become temporarily impassable. We have created methods that effectively solve this problem definition and have proven through many experiments that they are effective solutions. We also defined a MAPF problem variation in which agents have

multiple destinations in the lifelong MAPF problem structure. We have developed a new algorithm that solves the MAPF problem involving multiple destinations. For the task allocation problem, we created heuristic methods to minimize the total amount of travelled paths, and we tested these methods with many experiments and analyzed their performance.

Keywords: Multi Agent Path Finding, Incremental Planning, Task Allocation, Heuristic Search, Lifelong Planning

ÖZ

ÇOK ETMENLİ YOL BULMA PROBLEMİNİN ARTIMLI ORTAMLAR İÇİN GENELLEŞTİRMELERİ

Semiz, Fatih
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Faruk Polat

Temmuz 2022 , 100 sayfa

Çoklu Etmenler için Yol Bulma problemi (MAPF) gerçek dünyada da örnekleri olan ve bilgisayar bilimleri alanında sıklıkla çalışılan bir problemdir. Amacı birden çok etmen için onların başlangıç noktalarından bitiş noktalarına etmenlerin rotaları aynı anda aynı lokasyondan geçmeyecek şekilde yol bulmaktır. Gerçek dünya problemlerine örnek olarak ray üzerinde hareket eden robotlar ile depo ortamlarında paket taşınması, kapalı alanların temizlik robotları ile temizlenmesi, alanların çoklu robotlar ile korunması gibi problemler verilebilir. Bu problemleri ifade etmek için sayısal haritalar kullanmak genelde yeterli olabilmektedir. Ancak bu problemler standart MAPF senaryolarının aksine yola paketlerin düşmesi, harici araçların geçmesi veya problem devam ederken probleme yeni işler eklenmesi gibi durumlardan dolayı etmenlerin hareketi devam ederken yeniden planlama yapmaya ihtiyaç duyabilir. Bu gibi durumlar artımlı bir MAPF problem yapısı ile daha iyi ifade edilebilir. Bu tez çalışmasında haritadaki belli düğümlerin geçici bir süre geçilemez hale geldiği bir MAPF varyasyonu tanımladık. Bu problem tanımını etkili bir şekilde çözen yöntemler oluşturduk ve onların etkili çözümler olduklarını birçok deney ile kanıtladık. Ayrıca hayat

boyu MAPF problem yapısında etmenlerin birden fazla hedef noktasına sahip olduğu bir MAPF problemi tanımladık. Birden fazla hedef noktasına sahip etmenler içeren MAPF problemini çözen yeni bir algoritma geliştirdik. İş dağıtımı problemi için de toplam gezilen yol miktarını minimize etmeye yönelik sezgisel yöntemler oluşturduk ve bu yöntemleri birçok deney ile test ederek performanslarını analiz ettik.

Anahtar Kelimeler: Çok Etmenli Yol Bulma Problemi, Artımlı Planlama, İş Dağıtımı, Sezgisel Arama, Hayat Boyu Planlama

Dedicated to my dear family

ACKNOWLEDGMENTS

I would like to thank to my supervisor Prof. Dr. Faruk Polat with whom I started from my undergraduate years and continued to work together in graduate and doctorate studies for always approaching me positively on this long journey and for the countless experiences he taught me. Working with him has greatly contributed to improving myself as a person as well as develop myself in academic world.

I would also like to thank my family, who always supported me during my ups and downs during my doctoral adventure, and who were always there for me. Without their support, I wouldn't be here.

Finally, I would like to thank my wife, Ece, for her continuous support on this path and for being with me through all these difficulties. She motivated me in my doctoral studies that I am trying to carry out together with my business life. I owe my endless gratitude to my wife, who brought me back from time to time when I thought about quitting and when I got close to giving up from time to time. She gave me the strength to continue.

This thesis is partially supported by TÜBİTAK 1001 project under grant no 120E504.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ACAA	Add to Closest Average Agent
ACA	Add to Closest End Agent
ACASP	Add to Closest Average Start End Points
ACP	Add to Closest Point
ACSA	Add to Closest Start Agent
ASP	Answer Set Programming
BPA	Best Possible Adding
CBM	Conflict-based Min-cost Flow
CBS	Conflict Based Search
CBS-D*-Lite	Conflict Based Search with D*-lite
CT	Constraint Tree
GCC	GNU Compiler Collection
HCA*	Hierarchical Cooperative A*
ICT	Increasing Cost Tree
ICTS	Increasing Cost Tree Search
IDA*	Iterative Deepening A*
I-MAPF	Incremental Multi Agent Path Finding
LPA*	Lifelong Planning A*
MAPD	Multi Agent Pickup and Delivery
MAPF	Multi Agent Path Finding
MAPF-MD	Lifelong Multi-Agent Path Finding with Multiple Delivery Locations
MAPF-TA	MAPF Task Assignment

MDD	Multivalued Decision Diagram
MD-DCBS	Multiple Delivery Conflict Based Search with D*-lite
NASA	National Aeronautics and Space Administration
NP	Nondeterministic Polynomial Time
PERR	Package Exchange Robot Routing
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories
TAPF	Target Assignment and Path Finding
TP	Token Passing
UAV	Unmanned Aerial Vehicle
W-HCA*	Windowed HCA*

CHAPTER 1

INTRODUCTION

In Multi-Agent Path Finding (MAPF) problem, the main aim is to find conflict-free paths for more than one agent. Given a graph, the aim is to determine a path from an initial vertex to a target vertex for each agent such that no two agents can be at the same vertex at the same time. An example of a MAPF problem with three agents is shown in Figure 1.1. Many variations of this problem are being studied in the MAPF community. Generally, in these variations, an objective function is chosen and the quality of the solution is determined by the degree to which that objective function is satisfied. This problem, MAPF, is an Artificial Intelligence (AI) planning

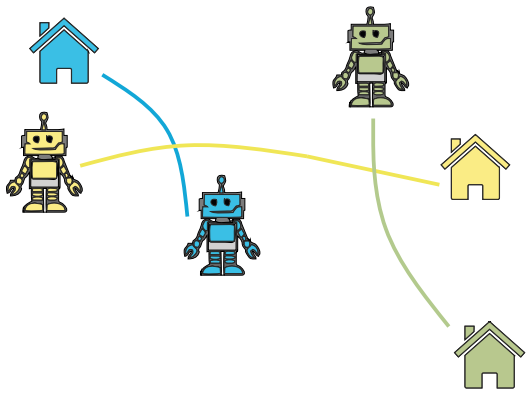


Figure 1.1: An example MAPF problem.

problem that is different from conventional robotic route planning problems. MAPF problems generally focus on how agents (such as robots) will move on a given map, without considering the physical constraints of the robots [2]. This simplification in the problem complies with the definition of the problem since the applications of this problem are mostly grid-based and include narrow corridors. Moreover, the problems generally contain more agents and target points than the classical robot problems

[3, 4]. MAPF has real-world applications in warehouse management (Amazon Kiva robots, Alibaba Quicktrone robots, Karis robots) [5, 6, 7] and autonomous tug robots (NASA - Ames project) [8]. This is a difficult problem since the state space grows exponentially with the number of agents [9, 10].

1.1 Motivation and Problem Definition

Ma et al. [11], showed that MAPF is inadequate for representing real-life problems, and provided four possible new directions for expanding MAPF. They emphasized the importance of addressing the generalizations of MAPF to real-world scenarios as opposed to developing faster methods for the standard formulation of the MAPF problem.

In this thesis, we wanted to focus on some of the generalizations of MAPF that address real-world scenarios. In many real-life problems, changes may occur during execution of the plan that requires contingency planning [12, 13, 14]. For example, in a cargo distribution problem, packages have to be delivered to goal locations by vehicles. During delivery, some parts of the planned paths may become unavailable because of a traffic jam, maintenance work, or some other reason. Under such circumstances, some agents need to modify their paths in such a way that the total cost of re-planning is minimized. As another example, in a warehouse, external effects (such as people or cargo falling off a shelf) can block the road temporarily or permanently. Therefore, a broader problem definition is needed to solve real-world problems.

In this thesis, we introduce a variant of MAPF where the environment can change while the agents are moving from their initial locations to target locations. In these problems, agents have to adapt their paths in accordance with new obstacles (temporary). We name this variant the Incremental Multi-Agent Path Finding problem (I-MAPF) [15].

Another MAPF generalization is lifelong MAPF, where new jobs are constantly added to the system and need to be assigned to agents. This real-world problem is a common problem especially in automated warehouse environments, where it is valuable to find effective solutions [16, 14, 17, 18].

Another MAPF generalization is the MAPF problems involving agents with multiple destinations. Although there are pick-up and delivery variations of this problem in the literature [19, 20, 21], problems involving agents that schedule multiple jobs and do those jobs one by one will also be valuable in warehouse like environments.

The following sub-sections refer to terms that are frequently used in the MAPF problem:

1.1.1 Conflicts

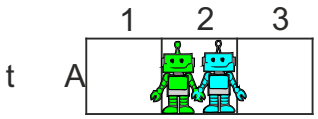


Figure 1.2: An example of an vertex conflict where the green agent and the blue agent collide in the node A2 at time t.

Although the term conflict varies from study to study, it can generally be defined as a situation that prevents a MAPF solution from being valid.

One of the commonly used conflict definitions is vertex conflicts. An illustration explaining vertex conflicts is provided in Figure 1.2. Vertex conflicts refer to the situation of two agents a_i and a_j being on the same node v at the same time step t in a grid-based world.

Another widely used definition of conflict is edge conflict. In this definition, two agents a_i and a_j cannot cross the same edge e at opposite direction at the same time t. An illustration explaining edge conflicts is provided in Figure 1.3.

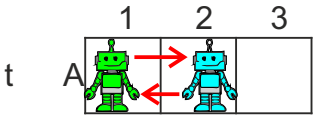


Figure 1.3: An example of an edge conflict where the green agent and the blue agent try to be in the same edge at opposite direction (between A1 and A2) at time t.

Other conflict types used are the following conflict and cycle conflict. Following conflict prevents agents from passing through the same node at one-second intervals. At a time step t, a cycle occurs if all the agents involved in the problem move to a node where an agent was previously present. Cycle conflict prevents cycles in a solution.

1.1.2 MAPF Solutions

The solution to the MAPF problem includes the routes that all agents reach from their starting point to their ending point. For the solution to be valid, conflict situations should not occur between the agent paths. In Figure 1.4 an example MAPF problem solution is presented. In this MAPF problem, two agents operate in a 4x4 word. Black cells represent obstacles. The solution routes are presented on the right side of the picture. Also, the paths that the agents follow are presented on the map. This solution is a valid MAPF solution because the paths do not have any conflicts.

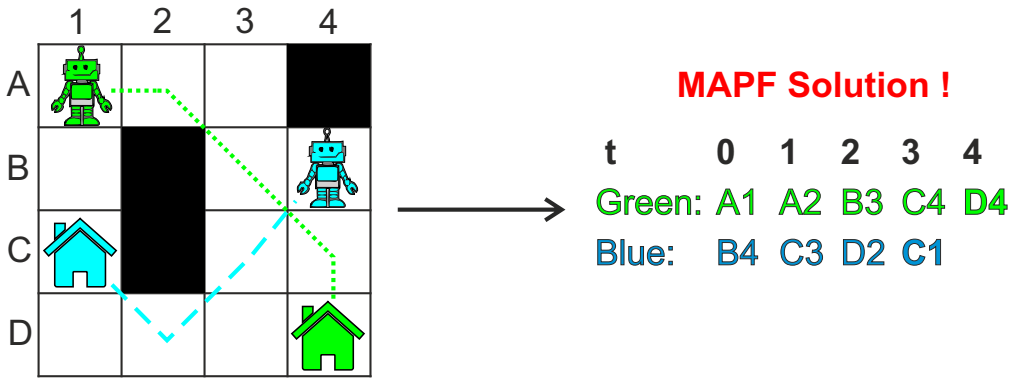


Figure 1.4: An example of MAPF solution is presented where the green agent and the blue agent reaches their goals without having any conflicts (in this example vertex conflicts are used).

1.1.3 Assumptions on Agent Behaviors After Reaching Targets

In MAPF solutions, agents can arrive at their targets at different times. In such cases, different assumptions can be made about the agents reaching the target. The most

common assumption is that the agent continues to stay or disappear after reaching the target. Both assumptions can be used in MAPF problems.

1.1.4 Actions Allowed by Agents in the MAPF Problem on Discrete Maps

In this section, we talked about the possible actions that agents can take at each time step in discrete maps. Allowed action sets can vary from problem to problem. Normally 9 different actions are allowed on a grid map. These actions are shown in Figure 1.5: move one cell to the right, left, top, bottom, top right, bottom right, top left, bottom left, and wait in the same cell. However, in most MAPF studies, diago-

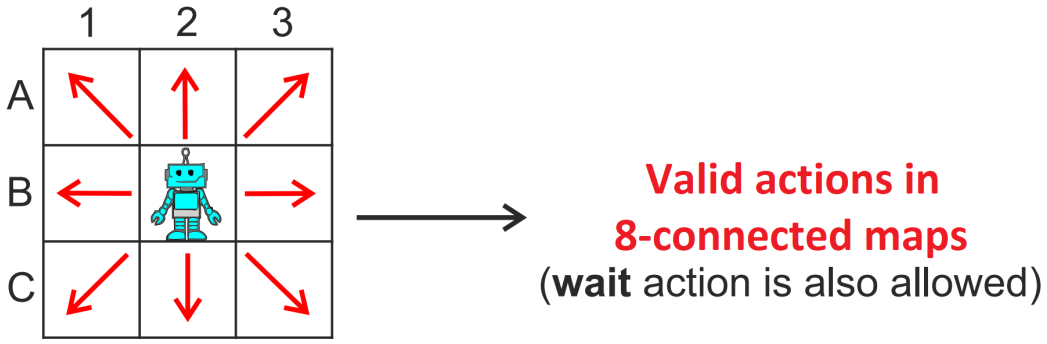


Figure 1.5: Actions that an agent can take in a time step in a MAPF problem in a 8-connected grid environment.

nal moves are not allowed. In such problem settings, the number of actions the agent can take decreases to 5. The reason for this is that robots generally have to move on rails or linear paths in environments such as warehouses. The allowed actions in such problems are also shown in Figure 1.6.

When there is an obstacle in one or more of the neighboring cells around the agent, these cells should be removed from the list of cells that the agent can move into.

1.1.5 Objective Functions

In order to determine which MAPF solution is better than the other, a metric that measures the quality of the solution must be determined. Two common metrics are

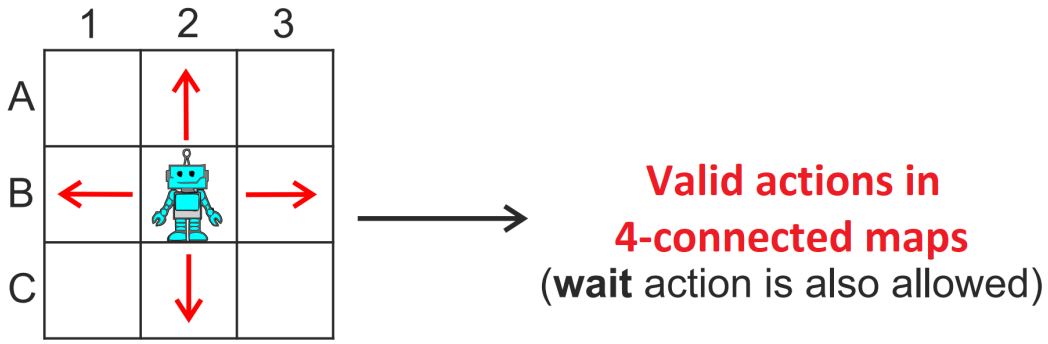


Figure 1.6: Actions that an agent can take in a time step in a MAPF problem in a 4-connected grid environment.

the sum of costs and makespan.

The sum of costs metric is calculated as the sum of the total times that all agents in the problem reach their destination. The formulation is provided below where k is the agent count, i is the agent number that is being processed and C_i is the cost of i_{th} agent:

$$\sum_{i=1}^{k} C_i \tag{11}$$

The makespan metric is calculated as the maximum time to reach the target for all agents in the problem. The formulation is provided below where k is the agent count, i is the agent number that is being processed and C_i is the cost of i_{th} agent:

$$max(C_i) \qquad where \qquad 1 \leq i \leq k \tag{12}$$

1.1.6 Incremental Multi-Agent Path Finding (I-MAPF)

In this study, in addition to the standard MAPF problem, we have defined a MAPF variation in which some nodes on the map become unavailable for a time period while the problem continues, and if at least one of these nodes is in the route of at least one agent, the routes must be updated for those agents. We call this variant the Incremental Multi-Agent Path Finding problem (I-MAPF). In Figure 1.7, an environmental change that occurs in the second time step and disappears in the fourth time step is

defined as an example of an I-MAPF problem with three agents. It is possible to see the routes that the agents planned while there was no environmental change in the upper left. Since a location in their plan became unavailable from the second time step, the agents updated their initial plans from that time step.

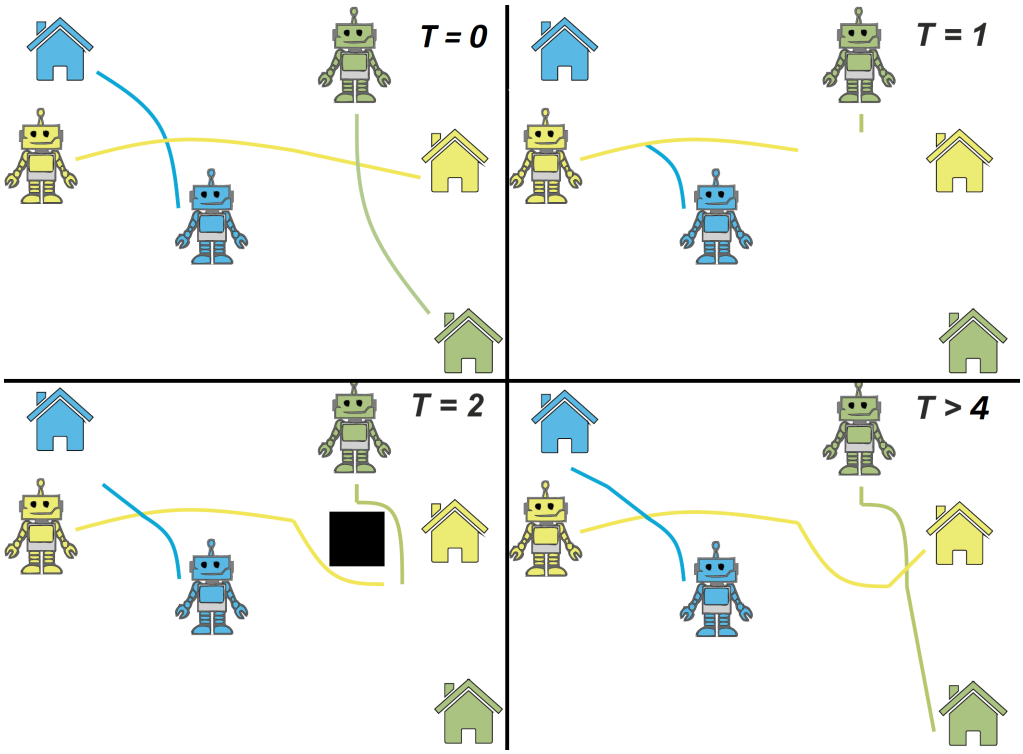


Figure 1.7: An example of I-MAPF problem where an environmental change occurs from time-step two to time-step four. Agents updated their plans accordingly.

In an I-MAPF problem, there are n agents $A = \{a_1, a_2 \cdots a_n\}$ on an undirected graph $G = (V, E)$ trying to go from the start point determined for it s_i to the end point determined for it f_i. In this setting, V represents the nodes on which agents can stand, and E represents the edges between those nodes. Each agent has a specific start point s_i and an specific endpoint f_i where s_i and $f_i \in V$. In this problem vertex conflicts and edge conflicts are used. There are two actions the agent can take at each time step. The first of these is to move to a new node, and the second is to wait at the current node. We can represent current state of an agent as (a_i, v_j, t) where a_i is the agent, v_j is the node it stays and t is the time step. The next step of the same agent can either be $(a_i, v_k, t + 1)$ or $(a_i, v_j, t + 1)$ where $k \neq j$. Agents can start from the same starting point but must leave their start points at different time steps. It is assumed

that agents disappear when they arrive at their goal location. We evaluated the actions of moving to a neighboring node or waiting on the same node as equally weighted. In the I-MAPF problem, we defined the term environmental change as a node $v_i \in V$ becoming unreachable for a period of time. The number of environmental changes and how long those environmental changes will take is given as inputs to the problem.

Agents are informed that a node is unavailable, only one-time step in advance. Only the information that that node cannot be reached is shared in this information, and the information on how long this situation will last is not provided to the agents by the system. The system notifies the agents that a node is available again one-time step before that node is available. The formal definition of environmental changes can be explained as follows: EC^n is a dynamic list with n ordered entries. Each input in the EC^n consists of triples as in the $\{v, t, \Delta_t\}$ example. In this example, v is the vertex that will become unavailable, the t value represents the time when the node v will become inaccessible, and Δ_t represents the length of the unavailability period. The EC list is ordered according to the t values of the triples in it. At the start of the problem, the EC list is initialized as empty $EC = \emptyset$. When each environmental change occurs, n is incremented by one, and a node v is selected from the V list and added to the EC list with the time information it will occur (t) and the duration it will last (Δ_t). The notation EC_i is used to represent the i^{th} element of the EC list, and the notation $|EC|$ is used to represent the size of EC list.

Let Δ_i be the period during which the node v_i (which is the first element of the triple EC_i, shortly $EC_i(1)$) will stay blocked. Then, for time period Δ_i, the vertex $EC_i(1)$ will be inaccessible in V, we call this updated version of vertex list as V'. During Δ_i, agents are only allowed to visit vertices from V'. After Δ_i expires, the vertex that was inaccessible is included in the vertex list again (more than one change can occur simultaneously and these vertex changes update V incrementally for the time steps they occur). If an environmental change blocks a vertex v_i for time-period t to t' (where $t' = t + \Delta_i$), then any agent planning to visit v_i during time-period t to t' must update its plan to avoid visiting v_i during time-period t to t'. We assumed that environmental changes do not occur at the start locations of the agents. However, they can occur at the goal locations. In that case, the agent should arrive at the vertex after the goal node becomes available again. The goal of each agent is to find a path (a sequence

of actions) from its start vertex to its goal vertex without any conflicts. A solution to this problem is the set of all agent paths without any conflicts. Also, agent paths must adapt to environmental changes. The output of the I-MAPF problem is a set of paths $\{P_0...P_k\}$ where P_i is a sequence of vertexes with time information without conflict or block. Formally, $P_i = \{(v_0, 0), (v_1, 1), \cdots, (v_i, i), (v_{i+1}, t_{i+1}), \cdots, (v_m, m)\}$ where v_0 is the start vertex of a_i and v_m is the goal vertex of a_i (i.e., $v_0 = s_i$ and v_m = f_i). The optimization criterion (objective function) used in this problem is the sum of the costs of the paths of the agents.

1.1.7 Multi-Agent Path Finding Problem with Multiple Delivery (MAPF-MD)

We worked on an lifelong variation in which agents can have more than one ordered destination. New destinations can be inserted into the system anytime after the initial job-assignment has been made, and these new destinations must also be assigned to agents, and the time of visiting the new destination must also be determined. We called this Lifelong Multi-Agent Path Finding with Multiple Delivery Locations (MAPF-MD). For example, problems in which some sub-domains of a certain area are assigned to agents and covered by them simultaneously can be given as an example of this type of problem (i.e, cleaning an area with autonomous robots). In such problems, it may be necessary to cover certain sub-areas repeatedly, and the decision to assign these tasks to which agent may need to be made in order to minimize the total energy consumed.

An example MAPF-MD problem is shown in Figure 1.8. In this example, there are three agents and each agent has two destinations. In addition, we can see a newly added job in the system as marked in red. The problem should also include the order in which this job will be added to which agent's destinations. What is mentioned here is not a capacity but a job scheduling sequence.

Similar to the I-MAPF problem described in the previous Section (1.1.6), an instance of MAPF-MD also consists an agent set $A = \{a_1, a_2, \cdots, a_k\}$ and an undirected graph $G = (V, E)$. Similarly, the starting vertexes where agents can start $S = \{s_1, s_2, \cdots, s_k\}$ and the ending vertexes they can finish $F = \{f_1, f_2, \cdots, f_k\}$ are determined. Any agent a_i has a specific start vertex s_i and a set of goal vertices

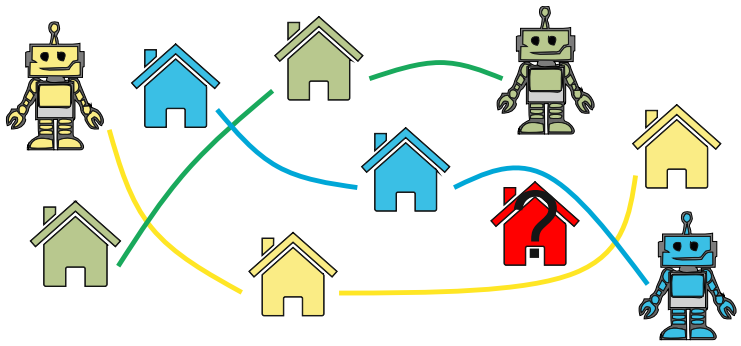


Figure 1.8: An example MAPF-MD problem is represented where three agents exists and all of them have two destination locations. A new job (shown in red) is need to be assigned to one of the agents and the order in which that work is to be fulfilled must be decided.

F_i. F_i is the ordered set of goal vertices that agent a_i must visit sequentially, and it is a subset of all goal vertices: $F_i \subset F$. For each agent a_i the length of F_i can be different and Since F_i is a dynamic list, F_i may change as time steps progress. F_x is the list of goal vertices that are added to the system after the job has started. It is initially empty: $F_x = \emptyset$. As time progresses, goal locations in F_x are distributed to agents, and these goal locations are added to the agents' own destination lists. There are two actions the agent can take at each time step. The first of these is to move to a new node, and the second is to wait at the current node. These actions both have the same cost in this setting. We used the similar problem settings used in in the previous Section (1.1.6) on conflicts (vertex and edge conflicts are used), agent disappears assumptions (agents disappear after they reach their final destinations) and cost calculations (objective function is the sum of costs of the paths of the agents). At any time a new goal vertex f_x can be added to system (a new destination to be visited) such that $f_x \notin F$ and $f_x \in V$. After this addition F becomes $F = F \cup \{f_x\}$, F_x becomes $F_x = F_x \cup \{f_x\}$ and the destination list of the agent a_i that the new job f_x is assigned to becomes $F_i = F_i \cup \{f_x\}$. Here f_x does not have to be inserted to the end of the list F_i, it can be inserted anywhere in F_i. The goal of each agent is to find a path (a sequence of actions) from its start vertex to its last goal vertex, by visiting all of its goal vertexes in an ordered fashion, without any conflicts. A solution to this problem is the set of all agent paths without any conflicts.

1.2 Proposed Methods

In this thesis, we worked on generalizations of the MAPF problem to represent the real word needs better, and we proposed methods to solve these variations efficiently. The methods that we offer are as follows:

- In Chapter 3, we first solve the I-MAPF problem with CBS (one of the standard MAPF solutions). For this, we proposed the CBS-replanner algorithm. Then we used D*-lite as a low-level solver to make this solution more suitable for the incremental domain, and we made a number of changes in the outline of the algorithm to make it more effective and complete. With the results, we showed that the newly proposed algorithm quickly updates its path according to environmental changes and is much more suitable for replanning,
- In Chapter 4, we developed an algorithm that solves a variation of MAPF consisting of agents with multiple delivery locations. To solve this problem, we ran a low-level search for each delivery location pair. We chose to use an incremental low-level solver to take advantage of its ability to cache previous search information as there are many replanning actions in the known environment. Then we used the CBS algorithm by combining the paths found for these delivery location pairs. Thus, we found a solution to this problem by finding optimal distance paths. We also solved the problem of constantly adding new jobs to the problem (a.k.a. Lifelong MAPF) by creating a set of heuristic job assignment algorithms. Our primary goal here was to minimize the total path cost by using these heuristics.

1.3 Contributions and Novelties

Throughout this thesis, we have proposed several solutions to problems that extend MAPF in a way that brings it closer to real-world problems. We focused on solutions that increase the replanning speed for the incremental domain. We have defined a novel algorithm that plans optimal paths for agents with multiple locations. In addition, while planning continues, we focused on path cost instead of makespan and

worked on cost optimization.

Our contributions are as follows:

- We provided a new generalization of MAPF and present a new problem I-MAPF which is more appropriate for real-life scenarios. This new generalization includes adding dynamic node changes (availability/inavailability) to the MAPF problem and requiring agents to adapt to these changes quickly.
- We provided two novel algorithms to solve I-MAPF problem:
  - The CBS-replanner is the incremental version of the CBS algorithm. With each new environmental change, it updates the agent starting points and calls a new CBS algorithm to update the solution. This approach produces solutions that adapt to environmental changes but are, unfortunately too expensive to be used in real-life problems. Furthermore, the A* algorithm is not suitable for dynamical environments because it cannot cache previous searches. Therefore, it has to recalculate all the paths from scratch.
  - We proposed the CBS-D*-lite algorithm to use the advantages of D*-lite in dynamical environments. Furthermore, we made a couple of changes in the overall structure of the CBS algorithm. The algorithm generally works in three steps. Whichever of these three steps it finds a solution, it stops there. Since the aim is to search as little as possible, a wider search is made in the solution area at each step. In this way, both fast and complete solutions that are not far from optimal are produced.
- We provided a new generalization of MAPF in which agents can have multiple destinations. We combined this problem with lifelong MAPF to create a new realistic problem.
- We provide a new optimal solution to solve MAPF instances having multiple delivery locations.
- We provided several new heuristic algorithms to assign jobs to agents and decide the order of these new destinations in agents' destination lists. Our heuristics use sum cost minimization.

1.4 The Outline of the Thesis

The remainder of the thesis is organized as follows: In Chapter 2, the previous studies in this field and the details of the studies in the field that form the basis of this study are explained. In Chapter 3, the algorithms of CBS-planner and CBS-D*-lite from I-MAPF solutions are explained in detail. The analyzes made about them and the results of the experiments are shared. In Chapter 4, the multiple delivery conflict-based search algorithm (MD-DCBS) algorithm for solving the MAPF problem involving multiple delivery locations for agents is described. In addition, five new heuristic algorithms that decide the job distribution and when to do the new job are introduced, and the results of the experiment are shown with a solution in which these two algorithms are used jointly. Finally, Chapter 5 summarizes all these described studies, emphasizes their importance in the field, and conveys the inferences made from the experimental results.

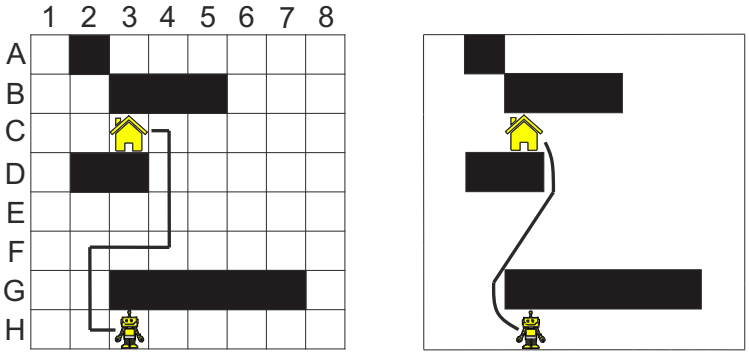
CHAPTER 2

BACKGROUND AND RELATED WORK

In this section, the preliminary information necessary to better explain the studies described in the thesis is presented. In addition, summaries of current studies in the fields related to the studies in this thesis are also presented in this section. This chapter discusses the evolution of path finding algorithms, a survey on multi-agent path finding solutions, examples of generalized multi-agent path finding problems, and studies attacking them.

2.1 Single Agent Path Finding

The single agent path planning problem is the problem of creating a path by choosing one of the actions allowed for each time step of an agent from a certain starting point to the endpoint in the graph and avoiding the obstacles on the map.



(a) Discrete Map (b) Continuous Map

Figure 2.1: An example showing single agent path finding in discrete and continuous maps.

There are many studies to solve the single agent path finding problem. Some of these are methods that work on discrete maps, while others are methods that work on continuous maps. Figure 2.1 shows an example problem presented in a discrete and a continuous map. To give an example of studies on path finding for continuous space: studies on visibility graphs [22, 23], voronoi diagrams [24], cell decomposition [25, 26], potential fields [24] and sensor-based path finding [27, 28] can be cited as examples. Since the focus of this thesis is mostly on discrete maps, the rest of this section will focus on studies working on discrete maps.

Studies that solve the path finding problem generally focus on the solution with heuristic-based algorithms [29]. Path finding problem is studied in many different areas, the unmanned aerial vehicles (UAV) [30, 31], route planning for robots [32], and studies on transportation [33] can be given as examples.

2.1.1 A* Search Algorithm

A* search algorithm [34] is an optimal and complete graph traversal algorithm that is frequently used in path finding problems. It avoids expanding paths that are already expensive. It performs a best-first search to find the optimal solution where the evaluation function is $f(n) = g(n) + h(n)$:

- $g(n)$ = cost so far to reach the node n.
- $h(n)$ = estimated cost from node n to goal.
- $f(n)$ = estimated total cost (via an admissible heuristic) of path through node n to goal.

The determining factor in the performance of the A* search algorithm is the chosen heuristic function. The selected heuristic will determine the number of nodes the A* algorithm will traverse. If $h(n)$ is admissible and consistent, the A* search algorithm will operate optimally.

In the A* algorithm, $c(x, y)$ represents the one-step cost of switching between nodes x and y. The algorithm starts by assigning the starting node to the open list and

equating the value function to infinity. It then starts repeating the next steps until the open list is empty. It selects the node with the lowest f-function value from the open list, marks it as the best node, and adds n_{best} to the closed list. If this node is equal to the destination node, the algorithm terminates and returns the path found through the closed list. If this node is not equal to the target node, the next steps are executed. The next steps are performed for all neighbors of the n_{best} node. For each neighbor, its distance from its target node is calculated by heuristic functions. The value function (f) is calculated by summing the heuristic value (h) with the total cost spent up to this node (g). If this node is not in the open list, it is added to the open list, if it is already in the open list, the new g value is compared with the old g value. If the new value is smaller, this value is updated. A pseudocode of the A* algorithm is provided at Algorithm 1.

Algorithm 1 A* Search Algorithm

Input: A graph and a root node to start

Add root node to $OpenList$.
$f(n) = \infty$
while $OpenList$ NOT empty **do**
    Pick n_{best} from $OpenList$ such that $f(n_{\text{best}}) <= f(n)$
    Remove n_{best} from $OpenList$ and add it to $ClosedList$
    if $n_{\text{best}} == goal$ **then**
        return $ClosedList$ ▷ *Return the found path*
    else
        Expand $\forall x$ that are neighbors of n_{best} and $\notin ClosedList$
        for Each successor node x of n_{best} **do**
            $h(x) =$ heuristically estimate distance to $goal$
            $g(x) = g(best) + c(best, x)$
            $f(x) = g(x) + h(x)$
            if x is NOT in $OpenList$ **then**
                Add x to $OpenList$
            else
                if $(g(n_{\text{best}}) + c(n_{\text{best}}, x) < g(x))$ Update x ▷ *Compare new g(x) with the previous g(x)*

2.1.2 Incremental Single Agent Path Finding

In incremental search algorithms, if there are changes in the world in which the agents are planning, they must update their plan according to these changes [35]. There are many studies focusing on incremental single-agent path finding in the literature. Lumelsky and Stepanov [36], Pinadeh and Snyder [37], Korf [38] and Zelinsky [39] tried to use existing information to re-plan the path dynamically. Stentz tried to solve the problem by generalizing the A* algorithm by making it responsive to environmental changes to use in dynamic environments. He developed two algorithms, D* [40] which partially changes the f and g values of the vertexes when environmental change occurs, and Focused D*, which is the improved version of the D* [41] that focuses on updating the costs to reduce the number of expanded states. Later, Ramalingam and Reps came up with an incremental algorithm called DynamicSWSF-FP [42] to solve a similar problem which is a grammar problem. DynamicSWSF-FP is capable of handling the arbitrary number of edge insertion, edge deletion, and cost changes. Koenig et al. [43] combined A* and DynamicSWSF-FP in Lifelong Planning A*. Koenig and Likhachev also proposed D*-Lite, which behaves similarly to D* but algorithmically different. It includes fewer conditional branches, which makes it easier to implement and extend [44].

2.1.3 D*-Lite

This subsection provides the general structure of the D*-lite [44] algorithm. D*-lite considers a goal-directed robot navigation task inside an unknown terrain. The robot starts its navigation from the start cell and tries to reach the goal cell. The terrain is unknown but the robot can observe its adjacent cells. It moves to one of the adjacent and accessible cells. D*-lite first computes the shortest path from the robot's current cell to the goal cell under the assumption that all of the unobserved cells are reachable. The robot follows this path until it reaches its goal cell or until it observes an inaccessible cell on its path. If the robot observes an inaccessible cell on its path (robot observes a new obstacle), then the robot re-computes the shortest path from its current location to the goal cell. D*-lite re-calculates goal distances only for those cells that can be used in the shortest path. It continues this process until it reaches

goal coordinates or determines that the goal location is unreachable.

The main difference of this algorithm is that for each node it keeps two estimated values of that node. One of them is g, which is the estimation of the objective function value based on the available information. The second is rhs. Rhs is a one-step forward estimation of the objective function value. The system uses these two estimated values to define consistency. If they are equal, the node is marked as consistent. Otherwise, the node is marked as inconsistent. Inconsistent nodes are added to the open list for processing. D*-lite does not store any pointers in memory. If there is a directional edge defined from node a to node b, a is called the successor of b, and b is called the predecessor of a. Also, the rhs value of a node is calculated with the help of the g values of its successors and the cost of passing the edges to that successor nodes.

For case u, rhs is calculated as: $rhs(u) = min_{s' \in succ(u)}(c(u, s') + g(s'))$. The priority of a node is determined by the heuristic h algorithm, along with the minimum g and rhs values. The primary key is calculated as follows: $min(g(s), rhs(s)) + h(s_{start}, s)$, the secondary key is calculated as: $min(g(s), rhs(s))$.

One of the most important features of the D*-lite algorithm for fast replanning is that it searches backward from the target node first (so that only the relevant nodes are expanded). In this way, it is quite easy to update the starting point of the agent in this algorithm. However, since the search is backward, it is difficult to understand which node was created in which time step when the nodes were first created. A pseudocode of the D*-lite algorithm is provided at Algorithm 2.

2.1.4 LPA*

Lifelong planning A* (LPA*) [43] is developed to generalize A* for dynamic environments where edge costs may change during the search. Unlike traditional A*, LPA* is capable of handling changes by just expanding the nodes that are affected by the change. It checks whether the shortest path from the start node to the current node is subject to the edge cost change. The algorithm does that by taking a minimum of the paths adding edge costs to the parents of the node. If the shortest path is changed,

Algorithm 2 D*-Lite Search Algorithm

Input: A graph and a root node to start

1: **while** robot did not reach the goal **do**
2: $initialize()$
3: $compute\_shortest\_path()$
4: **while** $s_{\text{start}} != s_{\text{goal}}$ **do**
5: $s_{\text{start}} = argmin_{s' \in \text{Succ}(s_{\text{start}})}(c(s_{\text{start}}, s') + g(s'))$
6: Move to s_{start}
7: Scan graph for changed edge costs
8: **if** any edge costs changed **then**
9: **for** all directed edges (u, v) with changed edge cost **do**
10: Update the edge cost $c(u, v)$
11: $update\_vertex(u)$
12: **for** all $s \in U$ **do**
13: $u.update(s, calculate\_key(s))$
14: $update\_vertex(u)$
15: $compute\_shortest\_path()$

LPA* updates the node's g value and then marks its children as their shortest path might also be changed. Child nodes are pushed to the open list to be expanded later. LPA* is complete and optimal if the heuristic is consistent and non-negative.

The LPA* algorithm does a forward search, unlike the D*-lite algorithm. D*-lite works faster by searching backward. However, if it is desired to have information such as which node was generated in which time step while resolving conflicts, it will be much easier to add the time information of each generated node, since LPA* uses a forward search. This will facilitate conflict resolution processes. A pseudocode of the LPA* algorithm is provided at Algorithm 3 and Algorithm 4.

2.2 Studies Working on Multi Agent Path Finding (MAPF)

In this section, we provided the survey we made about the studies that solved the Multi Agent Path Finding problem, under the headings we made according to the

Algorithm 3 LPA* Algorithm - Main

Input: A graph and a root node to start

```
1: function PROCEDURE CALCULATEKEY(s)
2:     Initialize()
3:     while do
4:         ComputeShortestPath()
5:         Wait for changes in edge costs
6:         for all directed edges (u, v) with changed edge costs do
7:             Update the edge cost c(u, v)
8:             UpdateVertex(v)
```

categories of the studies.

2.2.1 Optimal MAPF Approaches

Optimal MAPF approaches aim to find the most optimal solution in the solution space. They usually find the best solution, but sometimes it can take a long time for them to find a solution because of the high time complexity of large problems. The general aim of studies in this field is to develop methods to find the best solution without searching the entire space. Since the solution space size of the MAPF problem grows exponentially depending on the number of agents, it is possible to gain an exponential acceleration by reducing the number of agents in the problem. In this context, Standley [45] introduced the independence detection framework (ID). According to this study, if the solutions of the two agent groups do not conflict with each other, these two agent groups are independent of each other. The purpose of ID is to reduce the problem to independent groups of agents and solve the problem for them. Coupled methods can be given as another example of optimal MAPF approaches. These methods perceive agents as multidimensional composite systems. Increasing cost tree search (ICTS) [46] can be given as an example of optimal MAPF solvers. ICTS is a two-level algorithm, the top-level ICTS looks for the increasing cost tree (ICT). Each node in the ICT contains a list of individual route costs with as many elements as the number of agents. In this study, all possible outcomes at a cost c for each agent are stored in data structures called multi-value decision diagram

Algorithm 4 LPA* Algorithm

Input: A graph and a root node to start

1: **function** PROCEDURE CALCULATEKEY(s)
2: return $[min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))]$
3: **function** PROCEDURE INITIALIZE
4: $U = \emptyset$
5: for all S $rhs(s) = g(s) = \infty$
6: (rhs(s_{start}) = 0
7: U.Insert(s_{start},[$h(s_{start})$;0]
8: **function** PROCEDURE UPDATEVERTEX(u)
9: if ($u \neq s_{start}$) $rhs(u) = min_{s' \in pred(u)}(g(s') + c(s', u))$
10: if ($u \in U$) U.Remove(u)
11: if ($g(u) \neq rhs(u)$) U.Insert(u,CalculateKey(u))
12: **function** PROCEDURE COMPUTESHORTESTPATH
13: **while** U.TopKey() $<$ CalculateKey(s_{goal}) OR rhs(s_{goal}) $\neq$ g(s_{goal}) **do**
14: u = U.Pop()
15: **if** g(u) $>$ rhs(u) **then**
16: g(u) = rhs(u)
17: for all $s \in$ succ(u) UpdateVertex(s)
18: **else**
19: g(u) = ∞
20: for all $s \in succ(u) \bigcup \{u\}$ UpdateVertex(s)

(MDD) [47]. MDDs are compact data structures that are a special case of directed acyclic graphs. ICTS tests whether nodes at the lower level contain optimal results, by looking for k non-conflicting routes for k different agents in the cross product of MDDs. If a set of non-conflicting agents can be found in this way, the lower level result returns true and the search results, otherwise the lower level returns false and the search continues with another node at the higher level. Another example of a paired method is the CBS algorithm [48]. CBS is a two-level algorithm just like ICST. CBS performs its search on a tree called a constraint tree (CT). Unlike ICTS in CBS, at each node of the tree, a solution set of agent plans maintains a list of agent plans, total cost, and agent-plan conflicts. A new node on the CT is created by removing a

conflict in the previous node's conflict list. At the lower level, the agent routes are updated according to the newly defined constraints, while at the upper level, the CT tree is visited and the optimal solution is tried to be found. Another optimal solution example is SMT-CBS. SMT stands for satisfiability modulo theories (SMT). SMT-CBS is a study that brings together compilation-based and search-based studies [49]. SMT-CBS outperforms CBS and ICTS and performs closely but generates less space than SAT-based approaches (we provide details about reduction-based approaches in the Subsection 2.2.1.1). Boyarski et. al., 2020, proposed an optimal algorithm that solves the MAPF problem with an iterative deepening version of CBS. In this study, the authors created a memory-efficient CBS version by applying the logic applied in the iterative deepening A* (IDA*) study [50] to the CBS algorithm. In addition, this new approach is an optimal and complete approach like the CBS algorithm. The main difference between iterative deepening approaches and the best first search approaches is that iterative deepening approaches use a limit-based depth-first search approach instead of best-first search. Memory savings are generally achieved by this change. Also, this study uses LPA* in low-level search, using the idea that incremental approaches make use of prior knowledge [51]. This study is similar to our study in terms of using an incremental single-agent planner as a low-level solver. While D*-lite was preferred in our study, LPA* algorithm was preferred in this study. It is seen that the tests performed are aimed at solving the standard MAPF problem. In our study, the I-MAPF problem, which is a MAPF generalization, is solved. Also, in our study, we continued to build the CBS tree from where it left off and searched for a sub-optimal but very fast replanning version. Their version, on the other hand, follows the perspective of accelerating sub-steps without interfering with the main working structure of the CBS algorithm.

2.2.1.1 Reduction-based MAPF Approaches

Reduction-based solvers can also be given as examples of optimal MAPF solvents. We wanted to examine the studies within this scope under a separate subsection. The main purpose of reduction-based solvers is to reduce the MAPF problem to well-known problems in computer science and solve that reduced problem. Examples of such methods include the studies that reduce MAPF problem to boolean satisfiability

(SAT) [52, 53] problem, integer linear programming [9, 54] and answer set programming [55, 56, 57]. As a disadvantage of such methods, it can be shown that the conversion of the problem to another problem requires extra operations in addition to the solution of the problem and can increase the solution time [48].

2.2.1.2 Conflict Based Search

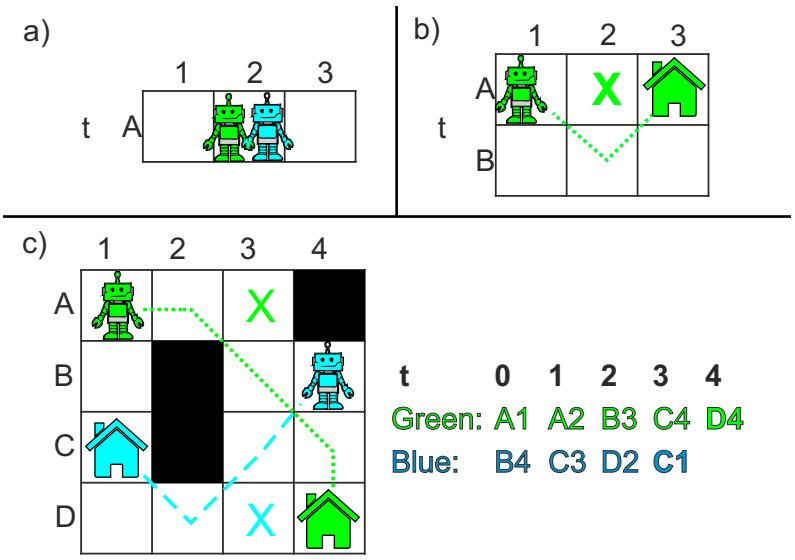


Figure 2.2: Some of the CBS definition illustrations; a) presents a conflict at time t, b) presents a constraint for the green agent and its path after it adapted its path according to that constraint, c) presents a valid solution containing consistent paths for each of the agents (to their constraints) and no conflicts (agents do not visit the same nodes at the same time)

CBS [48] is an offline search-based algorithm to find optimal paths for the agents in the MAPF problem. The optimization criterion used in CBS is the sum of costs of the paths of the agents. CBS represents all possible solutions to the MAPF problem as a state space. It represents every possible solution in state space as a node. It generates a special tree from these nodes, this tree is called the constraint tree (CT). CBS searches this tree to find the optimal solution. CBS starts its operation by finding paths for each of the agents from their starting locations to goal locations ignoring other agents. The root of three (CT) contains these single-agent paths. The set of k

paths for k agents forms a *solution* to a MAPF task. Afterward, the algorithm checks whether there is a conflict between these agent paths. They define a *conflict* as a situation in which two agents a_i and a_j occupy the same node v at the same time-step t (Figure 2.2-a). Each conflict is formalized to be between 2 agents if more than 2 agents try to occupy the same node at the same time, this is expressed by more than one conflict involving 2 agents each. New nodes in the state space (new CT nodes) are created by resolving conflicts. To resolve a conflict, one of the agents in that conflict should be told not to go through that node at that time step, and the restriction for this single agent is called a constraint. A *constraint* specifies a restriction that a particular agent a_i cannot occupy a node v at a specific time t (Figure 2.2-b). Since CBS does not know which of the agent paths will bring the optimal solution later, it creates a constraint for both agents. CBS creates 2 new nodes to CT by adding these two constraints to the previous CT node (one constraint is added to the right child node and one constraint is added to the left child node). When new nodes are created, agents that constraints are added to plan their routes again (satisfying the constraints added), and conflicts are recalculated according to the new paths. The tree continues to expand until it finds a node without conflicts. They call the solutions that conform to the given constraints and do not contain any conflicts as *valid solutions* (Figure 2.2-c). The algorithm performs a best-first search on the CT where nodes are ordered by their costs (sum of total costs of all agent paths). An example CT is provided in Figure 2.3. The best-first search maintains two lists during the search which are open-list and closed-list. The open list is the list of all generated nodes. These nodes are the states that the algorithm generated, which is a subset of all possible states. They include the states that are created but are not chosen by the algorithm later to be expanded. The closed list is the list of all expanded nodes. Ties are broken by using a conflict avoidance table (CAT) [45]. This policy suggests that when two states with the same cost are placed on the open-list, choosing the states having a lower number of conflicts with other agent paths is favorable. In the CBS algorithm, the process of searching the constraint tree and creating new nodes is called high-level-search. On the other hand, planning individual paths that satisfy the given constraints is called a low-level search. CBS uses the A* algorithm to make a single-agent search in low-level search.

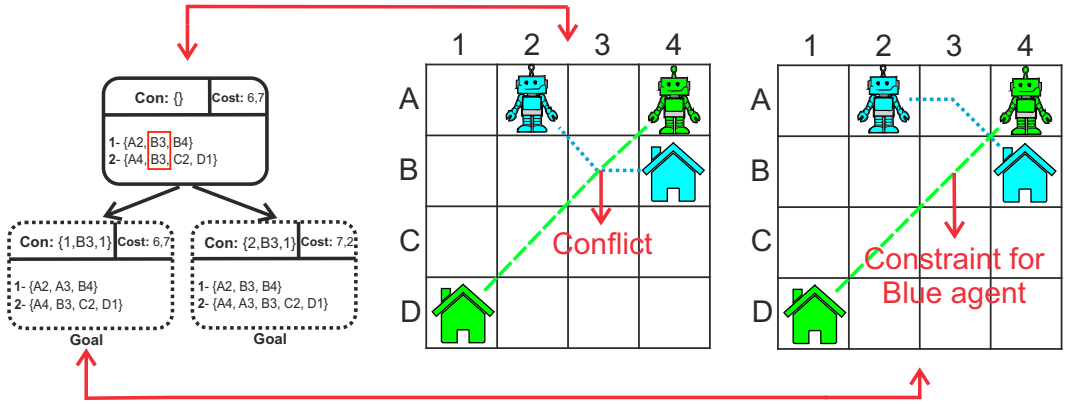


Figure 2.3: An example constraint tree (CT) is shown. Two of its nodes are graphically represented.

2.2.2 Sub-optimal MAPF Approaches

Since the methods used to find optimal results are costly in terms of the amount of computation, there has also been a trend towards sub-optimal approaches. Sub-optimal approaches focus on finding good enough results, rather than focusing on finding the best solution in the entire solution space. Some sub-optimal studies guarantee that the difference between the solutions they find and the optimal solution is below a certain limit, these studies are called bounded sub-optimal studies. One of the examples of bounded sub-optimal solvers is enhanced partial expansion A* (EPEA*) [58]. EPEA* is the enhanced version partial expansion A* (PEA*) [59]. PEA* is an A*-based search algorithm that ensures that nodes that are generated and not used in the A* algorithm are not expanded. EPEA* does not even generate the surplus nodes with the help of prior domain and heuristic-specific knowledge. Another bounded sub-optimal example of MAPF solvers is M* (A Complete Multirobot Path Planning Algorithm with Performance Bounds) [60]. This algorithm aims to minimize the dimensionality of the search space by coupling the robots that are found to interact. They dynamically generate low-dimensional search spaces inside the full configuration space. Another sub-optimal MAPF solver example is hierarchical cooperative A* (Eng. hierarchical cooperative A* - HCA*) [61]. In this method, agent routes are planned one by one in a predetermined order. When the first agent finds a route

to reach the endpoint, it records it in a global reservation table. Then, the planning agents plan their routes so that they do not conflict with the routes in that table and add their routes to the reservation table. As another variation of the HCA* method, the windowed HCA* method (W-HCA*) [61] has been proposed. One disadvantage of the HCA* method and its variations is that deadlocks can occur in multi-agent environments and therefore the HCA* algorithm is not complete. In addition, HCA* cannot guarantee that the solution will be above a certain quality and may produce solutions that are far from optimal. Another subset of sub-optimal approaches specific to MAPF is rule-based algorithms. Rule-based approaches define specific action sets for agents, and the problem is tried to be solved by making agents use these action sets where appropriate [62, 63, 64, 65]. In such approaches, the aim is to reach a solution quickly by searching a small part of the solution space. Such approaches may compromise the quality of the solution to reach quick solutions.

2.2.3 Studies on the Generalized MAPF Problem

Recent studies have revealed that MAPF problem modeling is insufficient to reflect real-life scenarios [66, 67]. Researchers have begun to focus on solving extended MAPF problems in order to meet real-life requirements. One of these extended MAPF problems is the combined target assignment and path finding problem (TAPF) [68]. In this study, the problem of distributing tasks to agents is discussed, the solution must include a set of agents, and the tasks are distributed first to the teams and then to the agents within the teams themselves. In most real-life scenarios, agents can swap their payloads. Ma et al. [68] formulated this scenario and introduced the package exchange robot routing (PERR) problem and, inspired by MAPF solutions, presented optimal and semi-optimal solutions for solving PERR. In another study, Cohen and Koenig [69] presented a new constrained semi-optimal algorithm that takes advantage of the problem infrastructure and calculates the agent route using only edges (either by the user or automatically) from the supplied edge set. Cohen and Koenig named this supplied set of edges the highway. Wan et al., in 2018, presented a MAPF approach that solves a life-long study of MAPF in dynamic environments. In the presented article, dynamism is defined by adding new agents to the environment. The lifelong definition of the problem comes from the fact that new agents can be added to

the problem all the time. In this problem, the newly added agents are added to the system with their targets assigned. After adding agents to the system no job assignment is made to the agents. In addition, after adding new agents, the previous solution in the CBS algorithm is modified and the new agents are run again from where they left off in a way to calculate their routes. This study differs from the problem definition of our project because the definition of dynamism is different and it does not allocate targets [70]. Bogatarkan et al., 2018, proposed a solution to the dynamic MAPF problem. The dynamism in this study is defined as the introduction of new obstacles to the map or the displacement of obstacles. The proposed solution method makes use of the answer set programming method. In this study, minimization of task completion times was determined for each agent as a minimization criterion [71].

2.2.3.1 Multi Agent Pick Up and Delivery

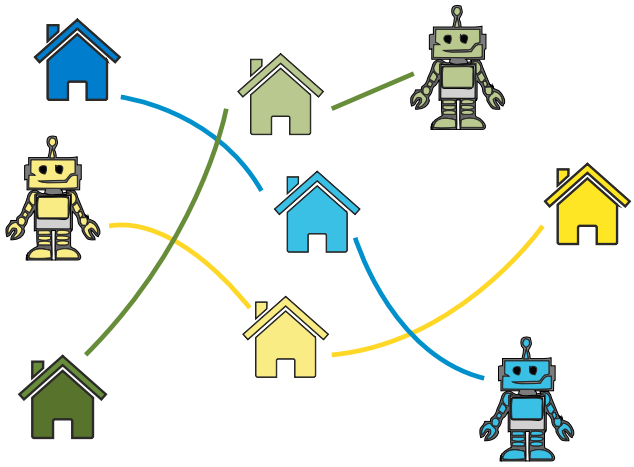


Figure 2.4: An example problem setting of Multi Agent Pickup and Delivery (MAPD) problem. Light-colored houses represent the pickup locations and dark-colored houses represent the delivery locations.

The pickup and delivery problem is a problem where mobile agents have to pick up an object from one place and leave it in another place. Figure 2.4 shows an example multi-agent pickup and delivery problem. This problem has been studied extensively before and its complexity has proven to be NP-hard [72, 73]. Recently, this prob-

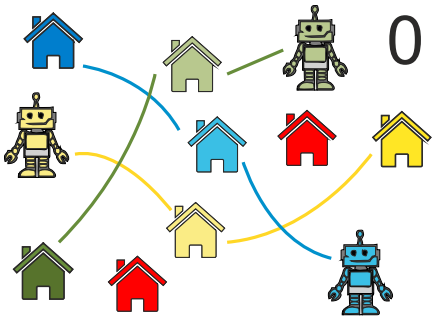
lem has been combined with the multi-agent path finding problem and has started to work in this domain. This problem is a frequently encountered problem, especially in warehouse environments. Problems, where multiple mobile robots take the packages placed on the shelves without colliding with each other and move on a grid structure and leave the package on another shelf, are an example of the real-world application of this problem. This problem is often worked together with the problem of allocating work to agents [74, 75]. Such studies will be discussed in more detail in Section 2.2.4.

2.2.3.2 Lifelong Multi-Agent Path Finding

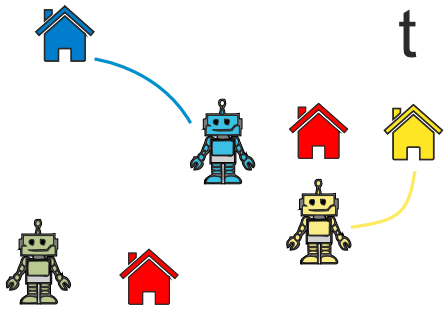
Tasks can be added to the system at any time in the Lifelong Multi-Agent Path Finding problem. Agents are expected to be assigned to these added jobs and the Multi-Agent Path Finding problem is expected to be resolved after these assignments are made [75].

2.2.4 Studies on Combined Task Assignment and MAPF Problem

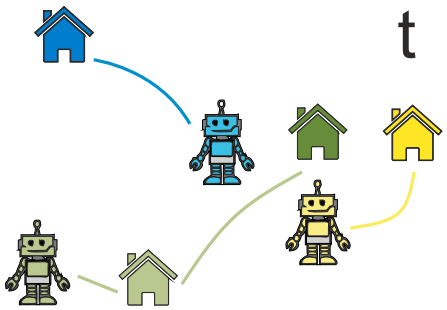
Examples combining MAPF and target assignment problems can be found in the literature recently [76, 68, 77]. Ma and Koenig presented the conflict-based min-cost-flow (CBM), a hierarchical algorithm, to solve the TAPF problem. In 2017, Nguyen expanded the TAPF problem to include an unequal number of targets and agents, setting deadlines for targets, queuing grouped targets, and targets that could consist of a string of checkpoints, and solved the problem using solution set programming (ASP). Hönig et al. in 2019, introduced a new optimal and complete solution method called MAPF-TA, and introduced an algorithm that solves the MAPF and target allocation problems together. In this study, the search process was carried out on a forest structure instead of a tree. The biggest difference between these types of studies and our present study is that in these studies, target allocation is done at the beginning of the problem and new targets cannot be added later. In our study, on the other hand, assuming that the initial target allocation has already been made, we plan to allow the addition of targets to the system later and focus on the allocation of those targets to



(a) Initial planning



(b) Simulation time t



(c) Simulation time t after task assignment

Figure 2.5: An example problem setting that combines multi-agent pickup and delivery and task assignment problems. Light-colored houses represent the pickup locations and the dark-colored houses represent the delivery locations. a) shows the initial planning, b) the red houses show the newly added pickup and delivery locations at time t. c) the new job is assigned to the green agent because it is the only free agent.

the agents, and we aim to reduce the cost of the solutions that result from these new additions. Another study that does the target allocation process in a more similar way to the method we want to do is Ma et al. by 2017 [75]. In this study, they studied a lifelong version of the MAPF problem with pick-up and delivery locations. In the lifelong MAPF problem, new targets can be added to the system continuously. To match objectives with agents, Ma et al. used a token passing mechanism. Each agent that finishes its job requests a token to be able to select a new job from the job list, and when the token is given to it, it chooses the appropriate job from the list. Wu et. al. worked on a version of the multi-agent pick-up and delivery problem in 2021, where each task must be completed within a certain deadline. They solved the problem by integrating task assignments and path planning processes. They decided which task would be selected and assigned to the agents with a metric called flexibility [78]. In 2021, Chen and colleagues worked on a variation of the multi-agent pick-up and delivery problem where agents can have capacities. This study proposes an approach that solves task assignment and path planning together. In this approach, marginal-cost assignment heuristic and meta-heuristic improvement strategies are used [79]. An example setting combining MAPD and task assignment is provided in Figure 2.5.

CHAPTER 3

INCREMENTAL MULTI-AGENT PATH FINDING WITH CBS-D*-LITE

Existing multi-agent path finding (MAPF) algorithms are offline methods that aim at finding conflict-free paths for more than one agent. In many real-life applications, it is possible that a multi-agent plan cannot be fully executed due to some changes in the environment (represented as a graph), or in missions in which the agents are involved. Even in the case of a minor change, the offline planning algorithm must be re-started from scratch to generate a new plan, and this often requires a substantial amount of time. Motivated by this real-life requirement, we introduced the Incremental Multi-Agent Path Finding (I-MAPF) problem (Section 1.1.6). In this problem, any location (node) in the initial environment (graph) can become unavailable for some time and then become available again. Agents can be informed about these changes before they occur and some agents have to update their plans if they planned to use that location. The Conflict Based Search (CBS) is one of most the successful algorithms in solving MAPF problems. To our best knowledge, there are no currently existing studies that attempt at solving the I-MAPF problem. In this chapter, we propose a new method to solve the I-MAPF problem, called CBS-D*-lite. CBS-D*-lite is built upon CBS and avoids re-planing for agents that are not affected by the environmental changes. To achieve this, CBS-D*-lite employs D*-lite, an incremental single-agent path-finding algorithm as the lower-level search method in CBS. Empirically, we show that the CBS-D*-lite provided faster results than regular CBS, and the total cost provided by CBS-D*-lite is generally close to the total cost values provided by the regular CBS when there are environmental changes.

In this section, we propose a solution method [15] for the I-MAPF problem. The first step of this method involves using the D*-lite algorithm as the lower-level planner.

We talked about how the D*-lite algorithm needs to go through a series of updates to do this, and how these updates can be done. Secondly, we made a few changes in the structure of the CBS algorithm that could increase the replanning speed and presented them.

When the availability status of cells in the map can change dynamically, some agents need to modify their paths in such a way that the total cost of re-planning is minimized. CBS is not a suitable approach for this because it employs A* in the low-level search and A* is an off-line algorithm, not suitable for dynamic environments. Because CBS uses A*, CBS has to re-compute all of the agent paths and re-generate the CBS tree from scratch to provide a new optimal plan for the agents. Consequently, using CBS in I-MAPF will produce an optimal strategy, but it will significantly increase the running time. Motivated by this inefficiency, we proposed a modified version of CBS to minimize re-planning overheads. We employed D*-lite [44], an incremental search algorithm, as a low-level search instead of A*. At the beginning of the task, we create a D*-lite instance for each agent. As CBS generates new constraints, D*-lite only modifies the relevant parts of the paths. When a new environmental change occurs, only the affected agents change their paths. If this change causes new conflicts with other agents paths, we add new nodes to the CBS tree. CBS-D*-lite does not create the CBS tree from scratch. It expands the CBS tree from where it left off. By dint of this expansion strategy, CBS-D*-lite avoids a significant amount of the re-planning. Thereby, CBS-D*-lite rapidly adapts the agent paths to environmental changes. Since there is no algorithm solving I-MAPF problem is presented before, we developed an incremental version of regular CBS, called the CBS-replanner, to evaluate the performance of our proposal. The CBS-replanner basically updates its map after every environmental change and runs the original CBS algorithm from scratch.

We show that the CBS-replanner is complete and optimal, but it is too slow to be used. On the other hand, CBS-D*-lite is complete but not optimal. It is faster than the CBS-replanner in dynamical environments. We tested and compared our algorithm on different planning problems. First we compared the two approaches on hand-crafted inputs. Then we randomly generated 8x8 graphs and made tests with a varying number of agents. After that we compared the effect of various types of environmental changes (hand crafted, randomly created and randomly created changes that forced

to occur at one of the agents path) to the overall running time, total path cost, and success rates of the algorithms. Finally, we tested both approaches on benchmark maps from [1]. Our approach returned better results in both stationary and dynamical environments. We observe that the difference between the performance of regular the CBS-replanner and CBS-D*-lite becomes more recognizable as the number of changes increases. In addition, in another study that is not within the scope of this thesis, we solved the problem we defined in this thesis with a CBS variation that uses LPA* as a low-level planner with a similar structure defined in this study [80]. We compared that variation with CBS-D*-lite. In the results, we observed that the CBS-D*-lite version found better results in terms of path cost. In terms of replanning speed, when the number of environmental changes are high, incremental CBS variation using LPA* is also seemed to be a good alternative to solve this problem.

3.1 Method

In this section, we present a novel approach, CBS-D*-lite, which can solve I-MAPF problem instances efficiently. Our method is based on the CBS algorithm [48] and it extends CBS in a couple of ways. First of all, regular CBS cannot solve incremental problems. To address this, we started with a naïve approach called the CBS-replanner. This naïve approach produces solutions that adapt to environmental changes but is, unfortunately too expensive to be used in real-life problems. The main reason why CBS is not suitable for incremental searches is that it uses A* as its low-level search. The A* algorithm cannot cache previous searches. Therefore, it has to recalculate all the paths from scratch. To solve this problem efficiently, we needed a mechanism to make use of previously calculated data. D*-lite [44] is shown to be a fast and efficient approach for solving single agent incremental path planning problems. So we changed the low-level planner of the CBS-replanner from A* to D*-lite. This novel approach is called CBS-D*-lite. It has the ability to cache previous searches. It only calculates the part of the paths that are affected by the environmental changes.

Our method of deciding when to call both algorithms is the same, so we found it appropriate to describe this method here. The system notifies every new vertex accessibility/inaccessibility one time-step before the change, but it does not provide

information about how long they will last. For this reason, for each time step, if there is a change (a new change or a previous change that $\Delta_i > 1$) and that change causes a conflict with the current solution in that time step, we call our algorithms in that time step and update the result by resolving the conflict. If there is more than one vertex unavailability that causes conflict at the same time step, we solve them one by one (we resolve the first conflict by calling our algorithm, then after finding the solution we resolve the second, etc). In these situations, we resolve the conflicts by first come first served strategy. If both occurred at the same time we randomly select one of them. Also at the time of replanning, if one or more vertices that were inaccessible become accessible again we add them back to the vertex list and then execute the algorithm.

Both algorithms provided below use the same input-output convention. They take a set of agents (A) with start (S) and finish locations (F), set of edges (E) and vertices (V) to represent a map and the current vertex accessibility/inaccessibility and its time (EC_i and EC_times_i). EC is a dynamically allocated list and updated after sensing every new change. The output of the algorithms is a set of non-conflicting agent paths and the total cost of that solution. The algorithms return this information inside the goal node of the created constraint tree.

In the following subsections, we describe the CBS-replanner, CBS-D*-lite and their theoretical analysis, respectively. After that, we explain the modifications we made on D*-lite to adapt it to our algorithm. Lastly, we provide a running example to explain better the working logic of the algorithms.

3.1.1 The CBS-replanner

The CBS-replanner is the incremental version of the CBS algorithm. In Algorithm 5, we provide a pseudo code of the CBS-replanner algorithm. At the beginning we provide the function that reads the number of environmental changes, the time-steps at which they will occur at and their durations (lines 1-11). These initialization jobs are given under the initialize_variables() procedure. Here we used randomly generated data, but when there is real data this section can be updated. Before any environmental changes, we make an initial plan without considering environmental changes

($CBS(A, E, V, S, F)$). This first plan works exactly the same as the regular CBS. When CBS-replanner is called it first reads the current changes and the last time step that CBS-replanner called where i is the number of change and t is the current time step (line 12). Then, it reads the previous solution (line 13). Then, CBS-replanner prepares the agents to make them ready for the new plan. Preparations of the agents include clearing the previously assigned constraints, saving the traversed part of the paths and updating the start locations of the agents. We cannot change parts of the agent paths that are already traversed. Hence, agents save traversed agent paths. We call these traversed paths realized-paths. Realized-paths contain physically visited locations (not plans). Each time the CBS-replanner is called, it appends recently traversed paths to realized-paths (line 15). In the end, the solution will contain the realized-paths of all agents. Each time the CBS is re-run it solves the problem for the remaining parts of the agent paths, so we need to update the start location of the agents with their current locations (line 16). It is important to notice that agents' current locations are the last locations of their realized-paths. The CBS-replanner clears the previously assigned constraints before the new run to find the optimal path under these circumstances (line 17). Of course, this causes re-planning of previously resolved conflicts and results in creating many surplus states at the high-level. Here we update the start time of EC_i, in this way we will process EC_i until its Δ_i finishes (lines 18,19). After the preparations are completed, the vertex list is updated by making the blocked vertex inaccessible which is the current element of the EC. The updated vertex list is called V' (line 21). Since we do not know how long the change will take, we act as if it will last forever and adjust our solution accordingly. If there is a change again in the future, we will adjust our path again in the future calls of this function. Then, we check the previous Δ_i values of the previous unavailabilities and if any of the vertices becomes accessible we make it accessible in V' (line 22). CBS is re-run with the new temporary vertex list V' and agent paths are adapted to environmental changes (line 23). Actually, here CT is created from scratch and the problem is solved from scratch with the new starting points of the agents and a temporary vertex list V'. The solution returned is a constraint tree (CT) node which contains the optimal solution under the given knowledge of the environment. After the last environmental change has occurred, remaining parts of the planned paths will be appended to realized-paths. The overall work-flow of the CBS-replanner is

Algorithm 5 CBS-replanner algorithm

Require: I-MAPF instance $= \{A, E, V, S, F, EC_i\}$

Ensure: realized-paths

procedure INITIALIZE_VARIABLES( )
    $n \leftarrow rand(1,$MAX_NUMBER_OF_CHANGES$)$
    $t = 0$
    $EC \leftarrow \emptyset$ ▷ *Initialize all elements to* $\emptyset$
    for n times **do** ▷ *Randomly determine values*
        $EC_i(2) \leftarrow rand(1,MAX_TIME_STEP)$ ▷ *Initialize* t *value*
        $EC_i(3) \leftarrow rand(1,$MAX_CHANGE_DURATION$)$ ▷ *Initialize* Δ_i *value*
    sort(EC) ▷ *Sort according to* $EC(2)$
    for each env. change occurrence time $EC_i(2)$ **do**
        $EC_i(1) = rand(1, |V|)$
    return EC,t
$EC_i, t \leftarrow$ initialize_variables() ▷ *Read current change*
solution $\leftarrow$ previous MAPF solution
for each agent a_j in A **do**
    realized-paths$_j$ = realized-paths$_j \cup$ solution.planned paths$_j[t : EC_i(2)]$
    $s_j \leftarrow v_j$ where state a_j is (a_j, v_j, t)
    a_j.clear_constraints()
if $EC_i(2) <= EC_i(3)$ **then**
    $EC_i(2) = EC_i(2) + 1$ ▷ *Prepare* EC_i *for next time step*
$t \leftarrow EC_i(2)$
$V' \leftarrow$ make_temporal_unavaliable_vertex_inaccessible($v_{EC_i(1)}$)
$V' \leftarrow$ apply_vertex_changes(V') ▷ *Add accessible vertices to list*
solution $\leftarrow CBS(A, E, V', S, F)$

summarized in Figure 3.1.

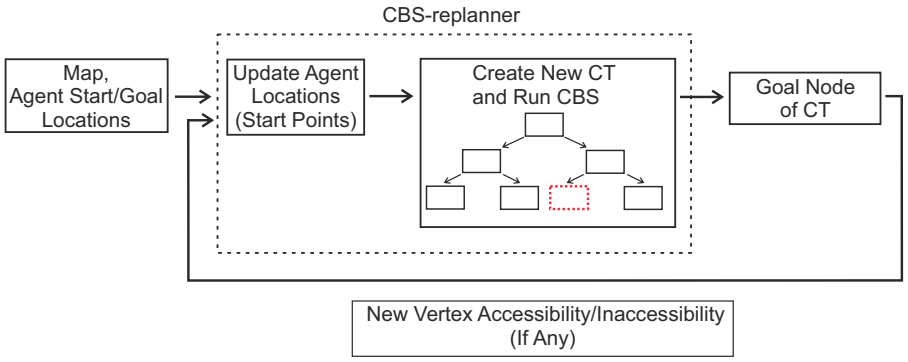


Figure 3.1: Work-flow of the CBS-replanner algorithm. When there are no environment changes, the algorithm directly creates the new CT and runs CBS.

3.1.2 CBS-D*-Lite

We developed the CBS-D*-lite method to use the advantages of D*-lite in dynamical environments. In Algorithm 6, we give the pseudo code of the CBS-D*-lite algorithm. Before any environment change occurs a solution is calculated by running classical CBS that used D*-lite as low-level-search ($DCBS(A, E, V, S, F)$). Here the dynamical conflict based search (DCBS) function is called. DCBS works similar to the CBS algorithm but there are small differences. The first difference is that it uses the D*-lite search algorithm at the low-level search. Furthermore, a few design constraints used in the CBS algorithm are modified. As A* does not cache any information and does not use any previous information from earlier searches, there was no need to make individual A* copies for the agents in CBS. However, this situation has changed in DCBS. To cache previous searches and make a faster replanning, each agent has its own D*-lite instance in DCBS. The algorithm starts by reading the current environmental change (line 1). This initialization process is exactly the same as the initialize_variables() procedure provided in Algorithm 5 and called at the beginning of the problem. Hence, we used the same procedure and did not here repeat the same procedure definition. Then, it reads the previous solution (line 2). Each time the CBS-D*-lite called, it appends recently traversed paths to realized-paths (line 4).

Algorithm 6 CBS-D*-lite algorithm

Require: I-MAPF instance $= \{A, E, V, S, F, EC_i\}$

Ensure: realized-paths

$EC_i, t \leftarrow$ initialize_variables() ▷ *Read current change*

solution $\leftarrow$ previous MAPF solution

for each agent a_j in A **do**

    realized-paths$_j$ = realized-paths$_j \cup$ solution.planned paths$_j[t : EC_i(2)]$

for each agent a_j in A **do** ▷ *Stage-1 starts*

    if conflicts(a_j, $EC_i(1)$) at $EC_i(2)$ **then**

        a_j.addConstraint($EC_i(1), EC_i(2)$)

        $s_j \leftarrow v_j$ where state a_j is (a_j, v_j, t)

        $solution.paths_j \leftarrow$ low_level_search()

        break

if solution is NOT valid **then** ▷ *Stage-2 starts*

    add Constraint($EC_i(1), EC_i(2)$) to solution

    solution $\leftarrow DCBS(solution, A, E, V, S, F)$

    $V' \leftarrow$ make_temporal_unavaliable_vertex_inaccessible($v_{EC_i(1)}$) ▷ *Stage-3 starts*

    $V' \leftarrow$ apply_vertex_changes(V') ▷ *Add accessible vertices to list*

    if solution is NOT valid **then**

        for each agent a_j in A **do**

            $s_j \leftarrow v_j$ where state a_j is (a_j, v_j, t)

            a_j.clear_constraints()

        solution $\leftarrow DCBS(A, E, V', S, F)$

if $EC_i(2) <= EC_i(3)$ **then**

    $EC_i(2) = EC_i(2) + 1$ ▷ *Prepare EC_i for next time step*

$t \leftarrow EC_i(2)$

After this point the CBS-D*-lite algorithm has three stages:

- **Stage-1:** It finds the agent that its path conflicts with the current vertex unavailability. Then, it modifies only the affected agent's path. If the solution is valid after this process, the algorithm is done for the current environmental change.
- **Stage-2:** If stage-1 is unsuccessful (if the solution is not valid) then CBS-D*-lite starts a new CBS search in which the root of the CT is the best node (solution) of the previous CBS run's CT. There will not be any conflict in this new root as it is the previous solution. The constraint that was found in stage-1 will be added to the new root's constraint list. The DCBS will then generate new nodes to find a solution.
- **Stage-3:** If the second stage is also unsuccessful then CBS-D*-lite clears all of the previous constraints and starts a new search from scratch (just like the CBS-replanner).

Then, the algorithm starts its first stage (lines 5-10). In this stage, the algorithm uses an important observation, as we already have a non-conflicting solution set the new environmental change (one time step of it) can only conflict with the path of one agent. By environmental change we mean the portion of it to be adapted, which is one time-step long. The other parts of the environmental change will be adapted in the following calls to CBS-D*-lite if needed. There may be more than one change at the same time, but as we adapt to them one by one, this observation still holds. So we find that agent (lines 5-6) and replan its path with its D*-lite instance without considering the optimality of the overall solution (line 9). If the agent can find a way to modify its path without conflicting with the environmental change and the other agent's paths then this means that we find a valid solution in a very fast fashion. If this generated solution is not valid, we move to stage-2 of the algorithm (lines 11-13). Stage-2 starts its operation by adding the constraint that found on stage-1 to the goal node of the previous solution (line 12). Then, it makes a call to DCBS (line 13), but this time it also provides the previous solution (after the constraint is added) as a parameter. We overloaded the DCBS function to use the previous solution in the search tree. DCBS takes this solution and makes it the root of the CT. Then it calculates and adds the current constraint to the constraint list and performs a conflict-based search. The first

thing to notice here is that the thing that enables us to search from where we left is the usage of D*-lite instances. The second important aspect is that this call to CBS starts to search a CT from its middle. By saying middle, we mean that we make the goal node of the previous tree as the root node of the new tree (which means we carry the constraints created for the agents with us), and some of these constraints would have changed if we would make a CBS search from the scratch for the new vertex set. This means that it is not guaranteed to always find a solution if it exists; it is only a faster attempt to modify the solution to an environmental change. If this attempt also does not yield a valid solution, then we move to stage-3 which is identical to what the CBS-replanner does (lines 14-20). Then, we prepare EC_i for next time step (lines 21-23). We actually add this stage to the algorithm to make CBS-D*-lite a complete algorithm. After the last environmental change has occurred, the remaining parts of the planned paths will be appended to the realized-paths. The overall work-flow of the CBS-D*-Lite is summarized in Figure 3.2.

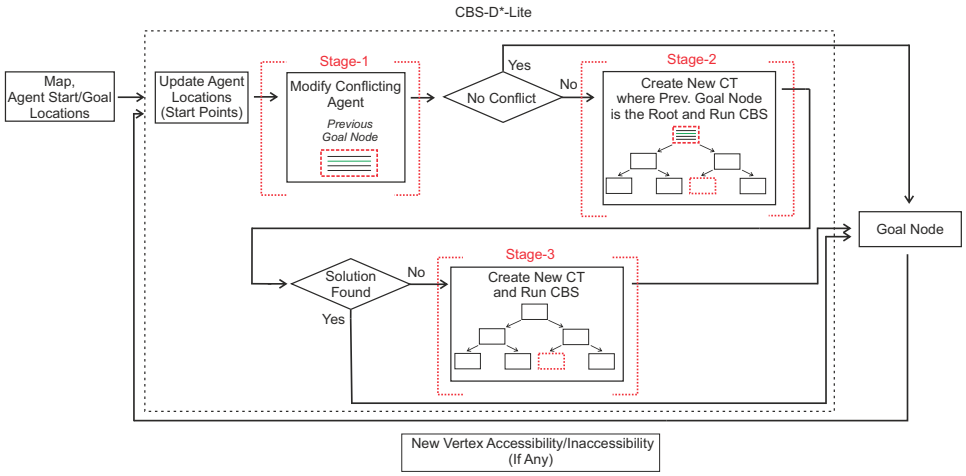


Figure 3.2: Work-flow of the CBS-D*-Lite algorithm. When there are no vertex accessibilities/inaccesibilities the algorithm directly goes to stage-3. After any vertex changes occur, to make replanning, the algorithm follows stage-1, stage-2, stage-3 path to find a solution.

3.1.3 Theoretical Analysis

We examine theoretical analysis under 5 subsections. We first provide a discussion about our optimality assumptions. We then introduce some subsections on the optimality and completeness of both algorithms provided. Lastly, we provide subsections to analyze the running times of the provided algorithms.

3.1.3.1 Optimality Assumptions

An optimal action that an agent performed before an environmental change can result in an overall solution that is non-optimal. In this study we assumed that an optimal solution in the dynamical domain is generated by always behaving optimally under the current conditions. We assumed that, when an environmental change occurs, current conditions change and agents must calculate new optimal plans to maintain optimality (for that time step). Furthermore, the agents don't know how long an environmental change will last, so when they encounter an environmental change they presume that it will last forever. They can only correct their paths after they learn that the vertex inaccessibility (environmental change) has ended.

3.1.3.2 Optimality and Completeness of CBS-replanner

Lemma 3.1.1. *CBS returns an optimal solution and it returns a solution if one does exists (complete) [48].*

Corollary 3.1.1.1. *CBS-replanner provides the optimal solution for non-proactive myopic agents and it is complete when there are no environmental changes.*

Corollary 3.1.1.1 holds because CBS-replanner calls CBS when there are no environmental changes and it is proven to be optimal and complete in Lemma 3.1.1.

Corollary 3.1.1.2. *At each call, CBS-replanner provides the optimal solution and it is complete.*

Corollary 3.1.1.2 holds because, at each call to the CBS-replanner, it sets up a new MAPF problem from where agents left off. Then it runs a new CBS instance for

that problem. This is exactly the same as running a new CBS for a MAPF problem (Lemma 3.1.1).

Theorem 3.1.2. *CBS-replanner provides optimal (the optimality defined in Section 3.1.3.1) solutions to I-MAPF problem and it is complete.*

Proof. We prove it by induction. The base case consists of the first call of CBS-replanner where I-MAPF problem does not have any environmental changes. It is shown to provide optimal and complete solutions in 3.1.1.1. Now, assume CBS-replanner provided the optimal solution after the first $i-1$ calls. During the i_{th} call CBS-replanner will provide the optimal solution if there exists one which is shown in 3.1.1.2. Thus, this completes the induction, CBS-replanner provides the optimal solution for the I-MAPF problem if a solution exists. □

3.1.3.3 Running time of each call of CBS-replanner

The complexity of each call of the CBS-replanner is $O(CBS)$. Here, $O(CBS)$ is the worst-case complexity of the CBS algorithm. At each call to CBS-replanner, agents perform some actions and get closer to their goals. In a worst case, the total distance to the solution can stay the same (this comes from the nature of a conflict-based search). Therefore, the complexity of the new plan can be at most equal to $O(CBS)$. As a result, this guarantees that the CBS-replanner can take at most $O(CBS)$ time for a single call. The downside of the CBS-replanner is that it usually works close to its worst-case running time. Hence, it is too slow to be used for a real-world problem.

3.1.3.4 Optimality and Completeness of CBS-D*-lite

Lemma 3.1.3. *D*-lite is an optimal and complete single agent search algorithm which can work on static and dynamical environments [44].*

Theorem 3.1.4. *CBS that uses D*-lite in its low-level-search is complete and optimal for solving MAPF problems.*

Proof. Lemma 3.1.1 shows that CBS is optimal and complete for MAPF problems.

When we change the low-level-solver of CBS from A* to D*-lite, D*-lite will still solve all of the single agent searches optimally from Lemma 3.1.3. The high-level-search will work exactly same as the original CBS. As the low-level-search function will return the same results, the CBS that uses D*-lite (note that this is different than CBS-D*-lite we provided, this function only used for the planning before any environmental changes occurred) will work exactly same with CBS which is shown to be optimal and complete for MAPF problems in Lemma 3.1.1. □

Corollary 3.1.4.1. *CBS-D*-lite provides the optimal solution and it is complete when there are no environmental changes.*

Corollary 3.1.4.1 holds because CBS-D*-lite calls CBS that uses D*-lite (these two are different) when there are no environmental changes and it is proven to be optimal and complete in Theorem 3.1.4.

Theorem 3.1.5. *At each call, CBS-D*-lite finds a solution to the current MAPF problem if there exists one but the solution is not guaranteed to be optimal.*

Proof. This proof has two parts, the completeness and optimality parts. For the completeness, CBS-D*-lite algorithm either finds a solutions in Stage-1 or Stage-2, or it moves to Stage-3 which provides a similar behavior to CBS-replanner. CBS-replanner is proved to be complete, hence CBS-D*-lite is also guaranteed to find the solution.

For the optimality, the CBS-D*-lite is not guaranteed to be optimal. When it finds a solution at Stage-1 or Stage-2, that solutions may not be optimal. However, any solution generated by the algorithm through Stage-3 is optimal. Hence, CBS-D*-lite is not guaranteed to find the optimal solution. □

Theorem 3.1.6. *CBS-D*-lite returns a solution for I-MAPF if one exist, and it is not guaranteed to be optimal.*

Proof. Completeness of CBS-D*-lite can be proved by following a similar approach shown in Theorem 3.1.2. The base case is again the first call and it is shown to be complete in Corollary 3.1.4.1. Assume that it finds a solution if it existed in first $i-1$ calls, and it is shown that CBS-D*-lite is complete for each call in Theorem 3.1.5.

So, the i^{th} call is also guaranteed to find a solution if there exists a solution. Hence the induction holds for the completeness of CBS-D*-lite.

For the optimality we show that it is not guaranteed to be optimal for each call, hence it is not guaranteed to be optimal for the overall solution. □

3.1.3.5 Running time of each call of CBS-D*-lite

At Stage-1, CBS-D*-lite only makes replanning for a single agent. At Stage-2, it runs CBS algorithm with a smaller state space because it has already resolved conflicts recently. At Stage-3, it makes exactly the same thing as CBS-replanner does; but for most of the cases the algorithm does not reach Stage-3 (as can be seen in the experiments). So its worst time complexity for a single call is $O(CBS)$ again, but in practice the running time is significantly shorter.

3.1.3.6 Analysis on the Number of Replanning Actions

We call both algorithms for each time-step that the environmental state causes a conflict with the current solution (when $\Delta_i > 1$), and we solve them by making a replanning at each of these time-steps. Furthermore, when there is more than one environmental change at each time step we make a replanning for them sequentially. This will increase the number of replannings that need to be done as the Δ_i grows and the number of environmental changes increases.

3.1.4 Modifying D*-Lite

We used Neufeld and Sredzki's D*-lite implementation [81] and modified it to make it suitable for our work. Two versions of this algorithm are available on the internet. In this study, we used the old version, that is, the version without new extra optimizations.

It seems that 3 basic questions need to be answered in order to integrate the D*-lite algorithm into the incremental algorithm (a modified version of CBS) we have

designed:

1. Adding the time steps to the lower level planners: this step is necessary because every time there is a new environmental change, the paths of the agents up to that time must be recorded and the starting points updated, and conflicts with other agents can be avoided.

2. Updating the agent starting points as the simulation progresses: this step is not simply updating the agent starting points as coordinates, but recording the agent's route up to that time when the time steps progress, deleting the conflict information sent to the agent's lower-level planner so far that no longer needs to be used, and updating the state space in the agent's algorithm.

3. Adding conflict information: this step is to ensure that nodes that are already unreachable due to conflict or environmental change at the time of planning of the solver algorithm are not expanded. The steps to be taken to achieve this will be examined under this heading.

3.1.4.1 Adding time steps:

Usually, D*-lite is designed to be a 2D path-planning algorithm for single agent path planning. In MAPF, the time of use of a particular vertex is also essential (because it is a multi-agent problem). A vertex that is busy at a time step can be available at another step. Hence, we needed a 3rd dimension which is time. To introduce time, we inserted a third variable to a vertex which is t. So, for every vertex $v_i \in V$, we have the tuple (x, y, t) where x and y represent the coordinates of the vertex and t represents the visit time of that vertex. Accordingly, since our problem works on a discrete world and on 4-connected maps, each step added to the agent route will actually correspond to a time step (the waiting process will add a value to the path file by increasing the g value by 1 from the same node. Since, it will not create a different situation than going right, left, up or down). Therefore, the g values held for each node in the algorithms will actually correspond to the time step values t. For this reason, time step values will be determined by using g values in operations (such as conflict resolution).

3.1.4.2 Updating the agent starting points as the simulation progresses:

This is pretty easy in this algorithm as D*-lite does a backward search. The D*-lite algorithm updates the state space from target location to start location and expands the minimum number of states to form a solution space. Then, it creates the agent route by choosing the ones with the minimum g value from these nodes. Since the state space is already updated, when the starting point of the agent is updated to be on its route or its neighbors, this situation can be solved by simply updating the coordinates without any action, since those states are already updated with g values in the state space. If the starting point of the agent is updated to a location that is not in or around the calculated route, since these nodes are not created, a route is planned again to create these situations. However, in our case, the starting point can be updated very quickly before this second mentioned situation occurs, as the agent will progress by following the plan it has prepared. Figure 3.3.

3.1.4.3 Adding conflict information:

This process can be shown as perhaps the biggest problem we encountered in the process of integrating the D*-lite algorithm. Since the states to be added to the solution space are determined and created by backward search and no new states are added to the solution space, we need to change the content of the algorithm here. This is because when a new conflict is added, it can make a node on this route impassable. For every vertex, D*-lite stores the predecessor and successor vertices in lists such that $(v_i \notin PR)$ & $(v_i \notin SU)$ where PR is predecessor list and SU is successor list. We exclude constrainted nodes from the successor list. When this change affects a single node, you can switch to the next node and continue without taking any action. However, if multiple changes are concentrated in the same region (an example of this situation is given with red environmental changes in Figure 3.4), then there may not be a neighboring node in a suitable state to go to. In order to avoid this situation, if a node becomes unsuitable due to an environmental change or a conflict, its lowest cost neighbor should be added to the route (hence the neighbor states should be expanded).

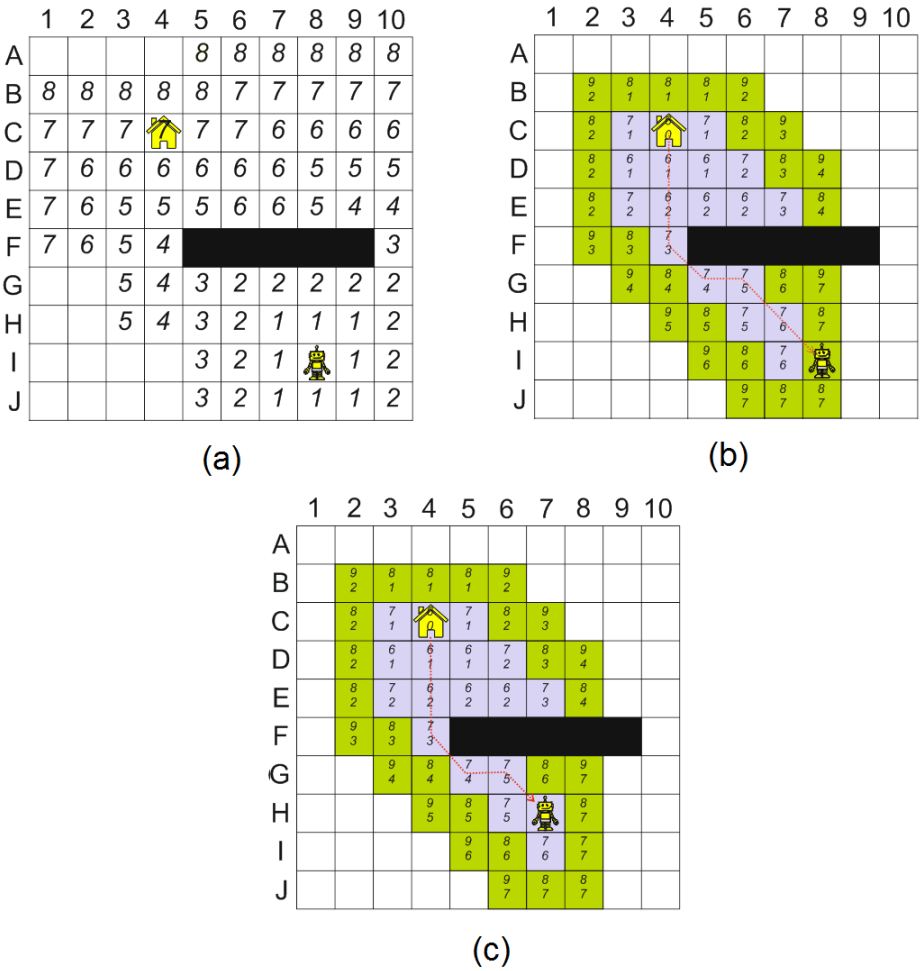


Figure 3.3: An image describing the working logic of the D*-lite algorithm and updating the agent starting points. (a) A grid structure showing the distances to the starting point that D*-lite has planned with Dijkstra algorithm. (b) Shows the process of finding the starting point by searching backwards. After this process is done, starting from the starting point, the cheapest nodes are selected and the solution is found. The top numbers are calculated as $min(g(s), rhs(s)) + h(s)$, the bottom numbers are calculated as $min(g(s), rhs(s))$. Here, g is the cost to date, rhs is one step lookahead, and h is the heuristic calculation result. (c) Shows updating a point on the path. Since the necessary calculations are already made for those nodes, a new route can be planned by selecting the minimum nodes.

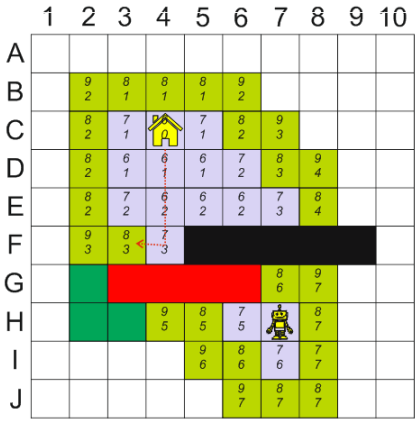


Figure 3.4: In this figure, an environmental change has occurred in the problem illustrated in Figure 3.3 (the resulting environmental change is indicated by red cells). The nodes whose value must be calculated so that the new route of the agent can be planned are shown in dark green.

3.1.5 Running Example

We demonstrate the flow of the CBS-D*-lite with a simple running example of two agents in a grid-world environment of size 4 x 4 (see Figure 3.5). In this scenario, $S1$ and $S2$ represent the starting locations, and $G1$ and $G2$ represent the goal locations of the agents. Robot drawings with the agent ids show the current locations of the agents. Agent-1 wants to reach location $D1$ starting from $A4$, and agent-2 wants to reach location $B4$ starting from $A2$ (Figure 3.5-a). In the scenario, we have two different environmental changes. One of them occurs at time-step $t = 1$ on location $B3$, and the second one occurs at time-step $t = 3$ on location $C2$. At these time-steps, locations $B3$ and $C2$ become unavailable for one time-step and become available again. Agents are informed about these changes one time-step before they occur. That is to say, the first change is announced to agents at $t = 0$ and the second one is announced at $t = 2$. Between each environmental change, calculations that are made to update agent paths are visualized in Figure 3.6. This representation presents the contents of the CT nodes, for each run of CBS. We present the current constraints used with **con** and current environmental change that is announced with **env**. We placed these on the upper left-hand corner of the CT node. At the upper right-hand corner we show the total **cost** of the solution of the current CT node. On the lower part

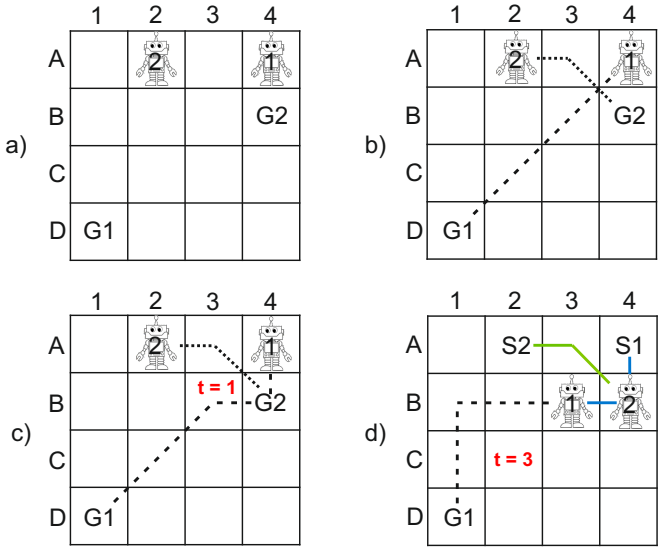


Figure 3.5: Running example; a) presents the initial agent configurations, b) presents the initial plans of the agents (t=0), c) presents the agent plans after the first environmental change ($t = 0$), d) presents the agent plans and the realized-paths of the agents after the second environmental change ($t = 2$)

of the CT we provide the current solution. Here, first the agent-ID and then the set of visited locations are provided as an ordered list. This list shows the locations of the specified agents at each time-step. If two agents' plans conflict with each other, or an environmental change conflicts with one of the agents' plan, then we represent these conflicts by drawing red rectangles around them. The planned actions are colored black and the locations that are already visited by the agents are colored green in the agent plans. The CT nodes that contain valid solutions are provided with a dotted outline and labeled as goal nodes.

CBS-D*-lite starts to solve the problem by running a CBS and producing a plan before any environmental change occurs. This provides a non-conflicting plan for the two agents. Figure 3.6-a represents the CT of the initial plan. Here, at the root node, agents plan their paths without considering each other. Agent-1 planned the route $\langle A2, B3, B4\rangle$ and agent-2 planned the route $\langle A4, B3, C2, D1\rangle$. Both agents planned to use the location $B3$ at time-step 1, and this caused a conflict. So CBS created two new nodes by adding $B3$ as the constraint to each conflicting agents.

These new nodes do not contain any conflicts, hence they are labeled as goal nodes. Initial plans of the agents returned from the CBS (goal node) can be seen at Figure 3.5-b.

After the initial plan, agents are informed about the first environmental change that will occur at $t = 1$. Because of this announcement, agents have to modify their paths in such a way that they do not use $B3$ at $t = 1$. Figure 3.6-b, shows the calculations that CBS-D*-lite does to modify the agent paths without running CBS from scratch. Agent-2's previously planned path includes visiting $B3$ at $t = 1$, this conflicts with the environmental change. CBS-D*-lite first adds a new constraint to agent-2 and runs its low-level-search to adapt its path (stage-1 in CBS-D*-lite). This clears the conflict with the environmental change, but this time agent-1 and agent-2's paths conflict at location $A3$ at time-step 1. So, CBS-D*-lite moves to stage-2 and runs the CBS by using this new node as the root node and it does not change the previous constraints. This new run of the CBS finds a goal node after adding $(A3, 1)$ to agent-2's constraint list. Thus the CBS-D*-lite returns the solution after running two stages. If this tree search were unsuccessful then CBS-D*-lite would move to stage-3 which runs a new CBS from scratch, but this time it was not needed. The agent plans after this environmental change can be seen in Figure 3.5-c. Here the red text at $B3$ states that that node is not available at time-step 1.

Until the second environmental change occurs, agents follow the previously planned routes. At time-step 2, agents are informed about the second environmental change. Figure 3.6-c provides the calculations to adapt the solution to this change. This time, CBS-D*-lite returns a solution at stage-1 just by changing the path of agent-2. The returned agent paths and plans are provided in Figure 3.5-d, where The green line shows the realized-path of agent-2, the blue path shows the realized-path of the agent-1 and the dotted line is the planned path of agent-1 after the environmental change. As there are no further environmental changes, agent-1 will simply follow the dotted line after this point.

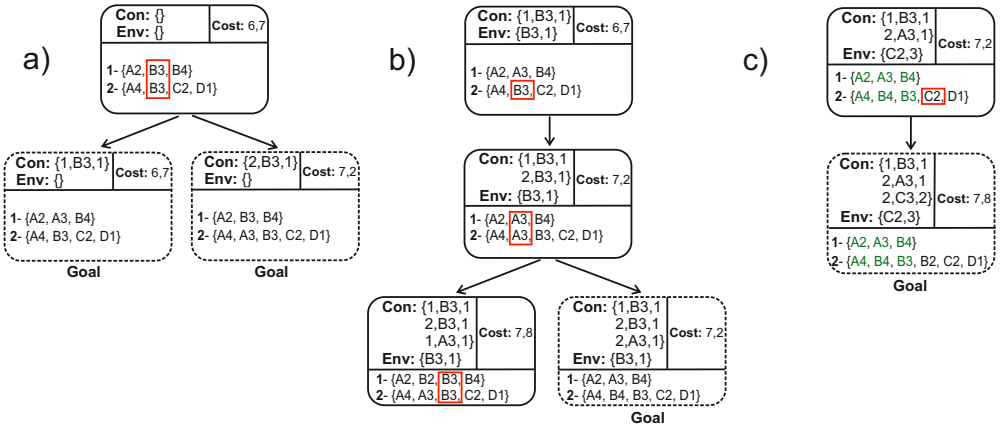


Figure 3.6: CBS-D*-lite execution after each environmental change; a) presents the initial CBS run (t=0), b) presents the calculations after the first environmental change (t=0), c) presents the calculations after the second environmental change (t=2).

3.2 Experimental Study

CBS algorithm is shown to provide optimal solutions for plain MAPF problem and it is fast. It has been already compared with the majority of existing MAPF solvers [48]. There are no studies solving I-MAPF problem we introduced in this paper. Existing algorithms do not have extensions to handle I-MAPF problem and we think that they are not suitable for comparison as they are not shown to be better than CBS [48]. That's why we base our experimental study on CBS and its extensions, CBS-replanner and CBS-D*-lite proposed in this study.

Since there are no previous attempts that use a CBS variant to solve the I-MAPF problem, we compared the performance of the CBS-replanner and the CBS-D*-lite algorithms in this section. We examined the performances of the algorithms with experiments involving environmental changes in different numbers and lengths.

The length of the environmental changes was randomly changed between 1 and 3 ($1 \leq \Delta_i \leq 3$). For both algorithms, we made replanning for every time step where any change caused a conflict. When more than one change causes conflict at the same time-step, we make replanning for them sequentially. This situation may cause the

solution to be slow when Δ_i values increase. For this reason, in this study we worked on small Δ_i values ($1 \leq \Delta_i \leq 3$). Note that this information is not shared with agents. Furthermore, larger Δ_i values may cause agents to proceed longer on paths that later turn out to be unfeasible (they cannot anticipate) which can raise the total cost. It will be on our agenda to make this algorithm suitable for large Δ_i values and we also plan to provide a version where the duration of the failure is provided to agents.

We examined the impact of environmental changes on the running time and the total path costs of the approaches. We tested the algorithms on two different datasets. The first data set consists of 8x8 grids with a varying number of agents. The second data set consists of two maps from the Dragon Age: Origins (DAO) game. These maps and some scenarios are reachable at the benchmark set provided by Nathan Sturtevant [1]. We introduced a time limit for the runs. We stop the experiment if it takes more than three hundred seconds. If the algorithms fail to return a solution within this time period, then we mark those experiments as unsuccessful experiments. We included a test case in the average only if it was successful for both of the algorithms otherwise we discarded that test case. We provide the test environment, details of the data sets, the test results and the conclusions from the test results in the following subsections.

3.2.1 Test Environment

We developed the project on a PC with a 64 bit 3.40 GHz Intel i7 processor. We used the Ubuntu 14.04.2 LTS operating system and, C++ programming language for implementation.

3.2.2 Data Sets

In the first data set, we started by creating a handcrafted map and manually generating a scenario for it. We also developed a random graph generator to generate grids with varying number of obstacles that are randomly placed. We also used randomly generated scenarios for the agents. We perform the experiments with 3 to 16 agents. For each agent count, we generate 100 grids with randomly generated obstacles in it. We report the average of all these 100 tests. In each test case, we applied five

environmental changes. In each time-step, we chose one of the nodes in the graph randomly and made it unavailable. Another different experiment is the case where we made unavailable only the nodes from the agent plans. In this scenario, each environmental change is guaranteed to effect at least one agent's path. We call these critical node failures.

In the second data set, we worked with den520d and brc202d maps from DAO game. These two maps have different topologies. The den520d map has many open spaces and has no bottlenecks. In contrast, the brc202d has no open spaces and many bottlenecks. We selected some of the scenarios presented in benchmark data sets [1]. These scenarios use ten agents for MAPF. These scenarios determine the starting and goal locations of the agents. In each scenario, we applied five critical node failures as environmental changes.

3.2.3 Test Results

In the following subsections, we provide different experiments on handcrafted and benchmark maps. In the following subsection, we reported the test results on handcrafted scenarios. Then in the following three subsections we reported the elapsed times of both algorithms and compared their performances after each replanning action on different maps and scenarios. Lastly, we provide a comparison of both algorithms in terms of total-path-cost and success rates.

3.2.3.1 Hand Crafted Tests

We first examined the performances of the algorithms on a single handcrafted grid. The hand-crafted grid is generated to analyze the behavior of the algorithms when there are numerous conflicts. Figure 3.7, shows the input used for this experiment. It is an 8x8 grid with some obstacles in it. For both of the algorithms, we prepared five different environmental changes. We chose each environmental change to occur on one of the agent paths. For example, in the first environmental change node, $B6$ becomes unavailable for the 1st time-step. So, agents have to adapt their plans in such a way that they do not use $B6$ at time step 1. Each agent's starting point is provided

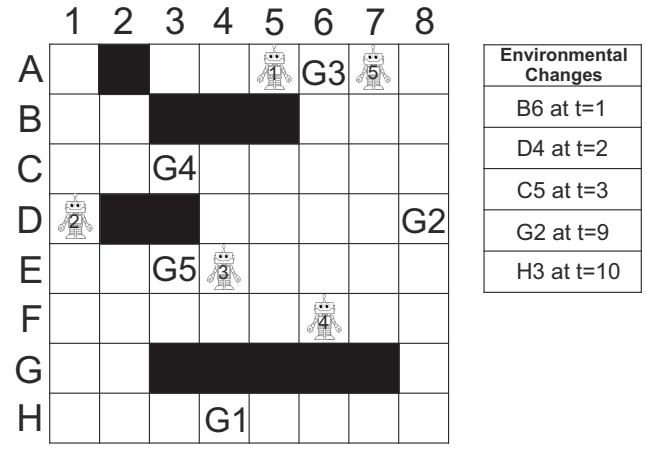


Figure 3.7: 8x8 hand crafted grid example with 5 agents and 5 environmental changes where black cells represent obstacles.

with a robot picture with the referred agent's id on it. The goal locations of the agents are provided with G signs. $G1$, for example, is the goal location of the first agent. The goal location of agent-5 is ($E3$) and the start point of the agent-3 is $E4$.

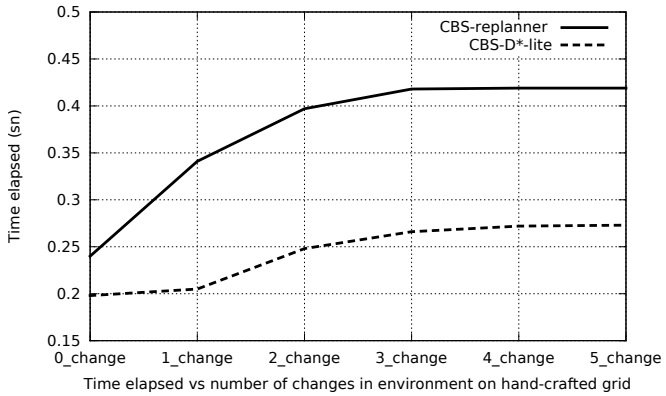


Figure 3.8: Performance overview for 8x8 hand crafted grid with 5 agents.

Results of both algorithms are given in Figure 3.8. Here, the x-axis contains environmental changes and the y-axis contains the total time passed since the start. In Figure 3.8, CBS-D*-lite is shown to provide better results than the CBS-replanner. With each environmental change, CBS-D*-lite finds the solution by only changing a few numbers of agents' paths, which takes much less time compared to computing

all agent paths from scratch and resolving conflicts. This happens because for most of the cases CBS-D*-lite can find a solution by just going into stage-1 and stage-2. In a more complex map, CBS-D*-lite would enter stage-3 in some cases, and the results in those cases would be closer to CBS-replanner. As time passes, agent paths get smaller; smaller paths cause fewer conflicts. With each environmental change, this causes a decrease in the increase rate of the total time for the CBS-replanner. CBS-D*-lite resolves single environmental changes by modifying a few numbers of agent paths (because it enters stage-1). The amount of time it spends for this is very small compared to running CBS from scratch. For this reason, its curve looks flatter compared to the CBS-replanner. Table 3.1 provides the solutions generated by

Table 3.1: Solutions provided by CBS-replanner vs CBS-D*-lite.

CBS-replanner
{**1** - A5, A5, B6, C6, D5, E4, F3, G2, H3, H4}
{**2** - D1, C2, C3, C4, C5, D6, D7, D8}
{**3** - E3, D4, C5, B6, A6}
{**4** - F6, E5, D5, D4, C3}
{**5** - A7, B7, C6, D5, E4, E3}
Total cost = 35,63

CBS-D*-lite
{**1** - A5, A5, A6, A6, B6, C5, D5, E4, F3, G2, H3, H4}
{**2** - D1, C2, C3, C4, C5, D6, D7, D8}
{**3** - E3, D4, C4, C5, C6, B6, A6}
{**4** - F6, E5, D5, D4, C3}
{**5** - A7, B7, C6, D5, E4, E3}
Total cost = 38,39

both algorithms. Both solutions are valid and do not conflict with the environmental changes. The total path-cost of the CBS-replanner is smaller than the cost of CBS-D*-lite because it provides optimal solutions whereas CBS-D*-lite does not. For our study, making fast replanning is more valuable than finding optimal solutions. Since this amount of difference in the total path-cost is acceptable for us.

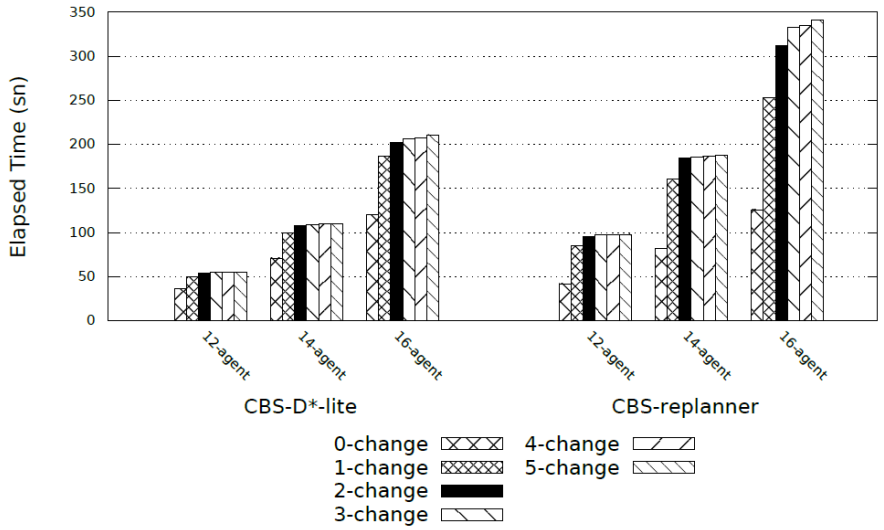
3.2.3.2 Randomly Created Data with Random Changes

Table 3.2: CBS-replanner vs CBS-D*-lite with randomly created environmental-changes. 3 to 16 agents are used, and for each case 5 environmental-changes are included in the environment.

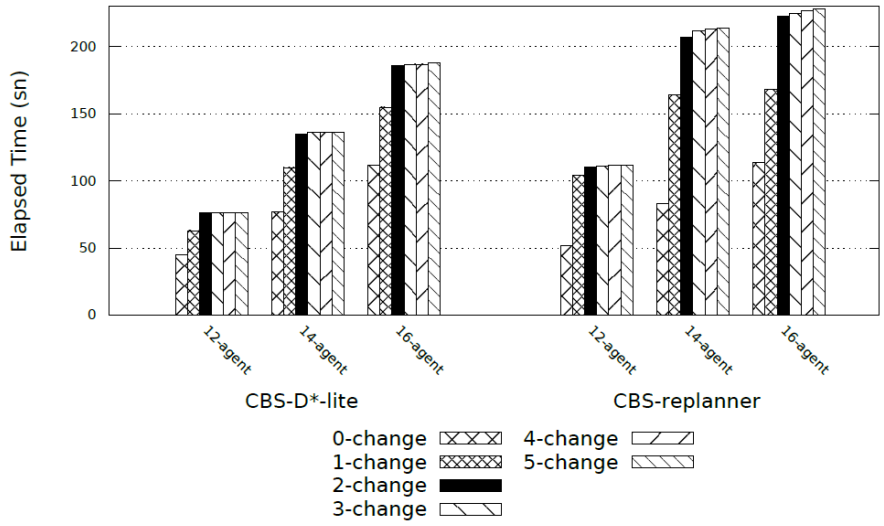
	CBS-replanner Elapsed Time (Sn)						CBS-D*-lite Elapsed Time (Sn)					
Agent No	Environmental Change						Environmental Change					
	0	1	2	3	4	5	0	1	2	3	4	5
3	0.007	0.013	0.016	0.016	0.016	0.016	0.006	0.009	0.009	0.009	0.009	0.009
4	0.028	0.056	0.062	0.064	0.064	0.065	0.025	0.035	0.038	0.038	0.038	0.039
5	0.032	0.064	0.071	0.073	0.074	0.074	0.029	0.041	0.044	0.044	0.045	0.045
6	0.132	0.262	0.294	0.301	0.303	0.304	0.122	0.172	0.184	0.187	0.188	0.188
7	0.416	0.827	0.927	0.949	0.955	0.958	0.386	0.544	0.581	0.591	0.594	0.594
8	2.151	4.274	4.796	4.906	4.938	4.954	1.955	2.757	2.941	2.994	3.006	3.011
9	4.028	8.004	8.980	9.187	9.246	9.277	3.565	5.027	5.363	5.460	5.482	5.490
10	8.516	16.92	18.99	19.42	19.55	19.61	7.604	10.72	11.44	11.65	11.69	11.71
11	18.22	36.20	40.62	41.55	41.82	41.96	15.71	22.15	23.64	24.06	24.16	24.19
12	42.97	85.38	95.80	98.00	98.64	98.96	36.42	51.35	54.79	55.78	56.00	56.09
13	53.81	106.9	120.0	122.7	123.5	123.9	48.92	68.98	73.60	74.92	75.22	75.34
14	76.91	152.8	171.5	175.4	176.6	177.1	71.88	101.4	108.1	110.1	110.5	110.7
15	118.2	234.9	263.5	269.6	271.3	272.2	113.7	160.3	171.1	174.1	174.8	175.1
16	123.2	244.8	274.7	281.0	282.8	283.7	120.8	170.3	181.7	185.0	185.8	186.0

In Table 3.2 we provide the results of performance with randomly generated failures on 8x8 randomly generated grids. In this experiment, we create five environmental changes. These environmental changes are created by, randomly making one of the nodes in the graph unavailable at each time-step. As the graphs are dense, for a low number of agents, they create less conflicts. Hence, both of the algorithms provided fast solutions for them. For the experiments including a high number of agents, performance differences become more explicit. For some of the cases, a randomly selected node did not cause any conflict on any agent's path. For cases consisting of a low number of agents, some agents already reached their destinations before the environmental change occurred. This situation caused faster solutions because finding a solution to a low number of agents is easier. In most of the cases, CBS-D*-lite was able to find a solution by just going into stage-1 and stage-2. The time difference before and after an environmental change occurred because of the cases that made CBS-D*-lite go into stage-3. CBS-replanners running time value is similar to CBS-D*-lite's stage-3 performance, but since CBS-replanner always performs

with this performance its average running times are worse than the ones provided by CBS-D*-lite.



(a) Performances on randomly created data with random changes.



(b) Performances on randomly created data with changes occur on path.

Figure 3.9: Performance overview on randomly created 8x8 dense graphs.

In every experiment the first environmental change causes the largest time increase, and each subsequent causes a smaller increase in time than the previous one. The reason for this is that the problem has nearly the same size as the zero environmental change scenario. With each succeeding environmental change, the size of the problem

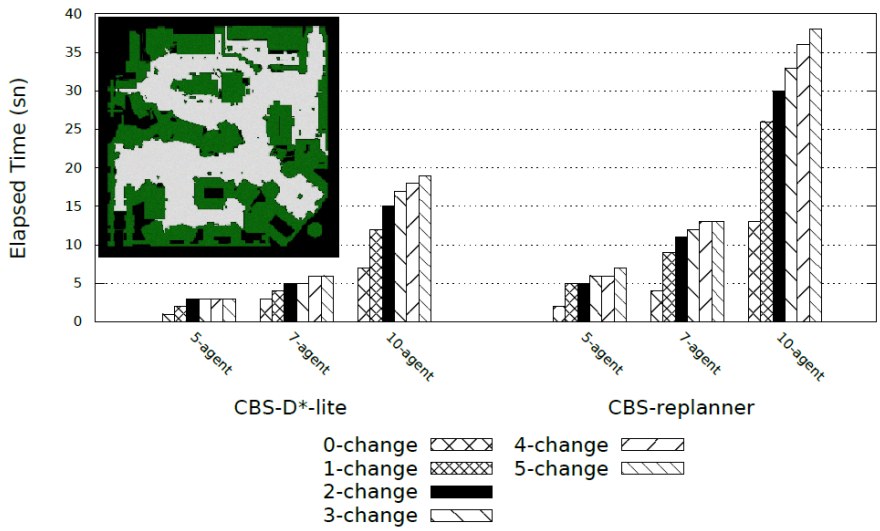
to be solved decreases. This situation causes a decrease in spent time. It is important to notice that we reported the total time that elapsed after each environmental change, not the time spent on that environmental change. To see the difference between the two methods, we generated a bar chart for the same example 3.9. In Figure 3.9a we provide only the results for a high number of agents. The trend of the total solution time is more explicit in this figure. CBS-D*-lite total spent time increase is slower than the CBS-replanner's increase-rate. The difference between the two algorithms arose from situations where CBS-D*-lite can solve the problem without entering stage-3. The largest difference occurs after the first change. Then, as the solution size decreases, the algorithms spend less time to find the solution.

3.2.3.3 Randomly Created Data with Changes that Occur on a Path

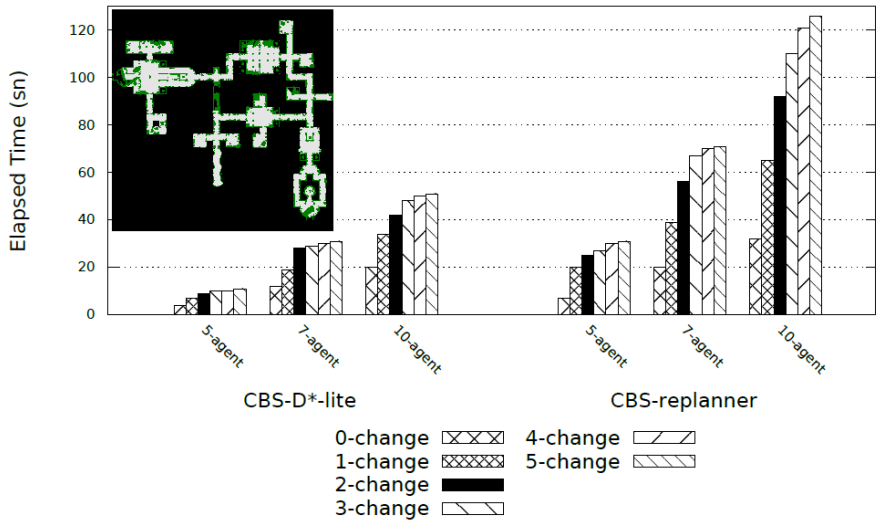
We constructed this experiment to confirm that environmental change affects the agent paths. The results are similar to the random change scenario. As these two experiments contain two different inputs, it is not possible to compare them head-to-head. The main difference between Figure 3.9a and Figure 3.9b occurs because of the problem size. In this experiment, problem size did not shrink as fast as in the previous experiment. The reason for this is that each environmental change affects at least one agent's path. This situation keeps the problem large. For both of the algorithms, there are jumps after the first, second and third environmental changes, but for the experiment that randomly created environmental changes are used, these jumps have occurred with smaller sizes.

3.2.3.4 Benchmark maps

Figure 3.10, reports the results of the experiments on den520d and brc202d maps. In these results, our A* implementation (used as a low-level-solver in CBS-replanner) worked slower than our D*-lite implementation (used as a low-level-solver in CBS-D*-lite). We can recognize this difference by examining the times presented in the zero-change bar for both of the implementations. Apart from this difference, the ratio of replanning time to overall solving time is significant. For the den520d map, each



(a) Performance overview in den520d map.



(b) Performance overview in brc202d map.

Figure 3.10: Performances of CBS-replanner and CBS-D*-lite with the different number of agents and changing environments.

replanning for the CBS-D*-lite implementation took ten to twenty percent of the time to solve the case from scratch. Again the main reason for the changes in the CBS-D*-lite bar are the cases that the algorithm entered into stage-3. After the first change, the CBS-replanner implementation took nearly the same amount of time to solve the case from scratch. After the second change, this ratio decreased from fifty percent to seventy percent. As the number of changes increased, some of the agents finished

their job, causing a decrease in the time of replanning for both algorithms. As the replanning is more costly for the CBS-replanner it caused a greater decrease of the replanning time on it with respect to earlier environmental changes. For the brc202d map, both algorithms took four to five times longer to solve the initial problem than the den520d case. In the brc202d map experiments, we observed a larger ratio of replanning times according to the complex nature of the map.

3.2.3.5 Comparison on Total-Path-Cost Values and Success Rates

In this section, we compared the path-cost values provided by the CBS-D*-lite and CBS-replanner. In Table 3.3, we presented the results on the three different maps. We plotted the average path-cost values of 100 randomly created 10-agent experiments. In the den520d map, there are many open areas. In open areas, agents have lots of low-cost alternative ways to follow when one node in their path becomes unavailable. Because of the open areas, CBS-D*-lite and CBS-replanner produced similar path-cost values on the majority of the test cases. The average path cost difference is around 1 to 1.5 units. In the brc202d map, the difference is a little bit larger. The reason for the difference is brc202d has some narrow streets which provide fewer low-cost alternative paths. So, for the cases including narrow streets, CBS-D*-lite found higher value path-costs compared to CBS-replanner. The average path cost difference is 2 to 7 units. 8x8 map is a small, dense environment where there is a limited number of alternative paths to follow. Hence, on the 8x8 map, the path cost value differences are more recognizable. For some cases CBS-D*-lite's stage-1 and stage-2 returned results far from optimal on this map. This caused a difference of around 1 to 3 units. The average path costs on this map are much smaller compared to den520d or brc202d. Therefore, the path cost difference of 3 units has a much larger impact compared to brc202d or den520d maps.

During the same experiment, the success rate values of the algorithms do not change much after the environmental changes. Therefore, we found it appropriate to set only one success rate value for each experiment. Both algorithms generally able to find fast solutions on 8×8 map experiments, and hence their success rate values are high on these experiments. On the experiments conducted on the den520d map,

Table 3.3: CBS-replanner vs CBS-D*-lite with randomly created environmental-changes that occur on the paths of the agents. 10 agents are used, and for each case 5 environmental-changes are included in the environment.

Map		CBS-replanner (Total Path Cost)						CBS-D*-lite (Total Path Cost)					
		Environmental Change						Environmental Change					
		0	1	2	3	4	5	0	1	2	3	4	5
8x8	average	42.09	43.03	44.04	44.82	45.86	46.69	42.09	44.07	45.63	47.08	48.58	49.85
	min	11.24	11.24	11.94	12.07	12.73	13.07	11.24	11.24	11.94	12.07	12.73	13.07
	max	70.21	70.21	70.21	72.46	72.87	73.46	70.21	73.63	75.04	76.04	77.46	78.46
	success rate	92.91%						96.4%					
den520d	average	1420.4	1421.9	1423.5	1424.6	1425.8	1427.3	1420.4	1422.6	1424.7	1426.4	1427.9	1429.8
	min	26.73	27.73	28.15	26.73	30.98	31.98	26.73	27.73	28.15	26.73	30.98	31.98
	max	2176.7	2179.7	2181.0	2183.4	2184.4	2188.6	2176.7	2182.3	2184.3	2187.2	2188.2	2193.9
	success rate	72.43%						91.01%					
brc202d	average	2959.96	2961.58	2964.04	2965.41	2966.54	2967.59	2959.96	2963.17	2967.00	2969.57	2971.47	2973.53
	min	32.80	33.63	34.21	34.21	35.04	35.04	32.80	33.63	34.21	34.21	35.04	35.04
	max	6175.38	6179.79	6184.03	6187.03	6189.44	6192.85	6175.38	6181.2	6188.28	6192.03	6194.44	6198.27
	success rate	66.72%						86.29%					

the CBS-replanner generally worked slowly and provided a low success rate on these experiments. On the other hand, CBS-D*-lite generally provided fast results, but in some cases, stage-1 of CBS-D*-lite was unable to find a solution. Therefore, the algorithm has passed to stage-2 and stage-3. In these situations, the algorithm worked slowly and exceeded the time limit. In the brc202d map, the situation is similar; the CBS-D*-lite provided a better success rate value than the CBS-replanner. The success rate values of both algorithms are a little bit smaller because of the complexity of the brc202d map.

3.2.4 Conclusions From Experiments

This experimental study showed that CBS-D*-lite, which is an incremental algorithm can find the results by rapid replanning (especially when a new solution can be generated by modifying one or two agents paths). On average CBS-D*-lite provided much faster replanning than CBS-replanner. The solutions provided are sometimes optimal and sometimes sub-optimal (not far from optimal). The path cost difference of CBS-D*-lite and CBS-replanner was small on average in all of the maps. The largest cost difference occurred in dense environments having narrow streets where there are not

many low-cost path alternatives after an environmental change.

CHAPTER 4

LIFELONG MULTI-AGENT PATH FINDING PROBLEM WITH MULTIPLE DELIVERY LOCATIONS

In lifelong MAPF problems, new jobs can be added to the problem during the simulation period. The problem of allocating these new jobs to agents needs to be resolved. Another problem we are trying to solve here is to find solutions for agents that include more than one destination in their plans. While designing this problem, it is aimed to distribute the new jobs to the agents while they continue their work. Thus, it is ensured that the agents continue with their new work just as they have finished their current work. This situation can be advantageous in terms of minimizing the total distance spent in problems such as continuous monitoring of areas with multi-robots, cleaning of certain areas with multi-robots and so on. We called this Lifelong Multi-Agent Path Finding with Multiple Delivery Locations (MAPF-MD). To solve this problem we introduced the Multiple Delivery Conflict-Based Search algorithm (MD-DCBS). To handle multiple delivery locations we define multiple low-level solver instances for each agent. The aggregations of all of the paths produced by the low-level solver instances constitute the path of that agent. After that we run CBS on aggregated paths. We have shown that this version solves MAPF-MD instances correctly. We also proposed multiple job-assignment heuristics to generate low-total-cost solutions and determined the best performing method amongst them. In this variation, agents do not carry a load, unlike the capacitated MAPF problem. Instead, they should visit more than one place.

A representation of a MAPF-MD problem is provided in Figure 4.1. In this figure, there are 3 agents and each of them has 2 ordered destinations. The robots represent the starting points of the agents and the houses represent the delivery locations of the

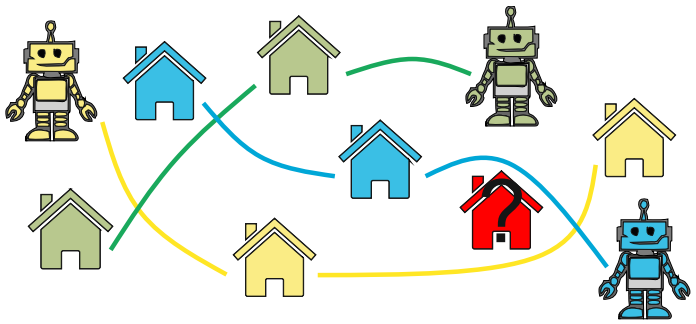


Figure 4.1: An example MAPF-MD problem representation.

agents. Agents and their delivery locations are color-matched. The red house is a new job that needs to be assigned. The aim is to assign it to one of the agents and visit it in such a time that will cause minimal cost change in the current multi-agent plan.

In this study we worked on heuristic algorithms that assign destinations to agents in such a way that the resulting multiple delivery MAPF problem generates low-cost solutions. In this problem, we used the sum of the total path lengths of the agents in the problem for cost measurement. We generated many heuristics to assign destinations and determine when to visit those destinations. Then we decided which heuristic strategy to use according to the results of various experiments. After assigning the destination we solved the resulting MAPF-MD problem with a modified version of the CBS algorithm [48] in which we used the D*-lite search algorithm [44] as the low-level search. As there are many replanning actions on the known environment we used the D*-lite search strategy to cache previous search information instead of using A* [82] in the low-level search. For each destination pair (source and the destination) we generated a D*-lite object, and for each agent we aggregated the paths generated from that D*-lite object and ran the CBS on those aggregated paths.

We tested our solution method on 8×8 handcrafted grids and on real-world scenarios like the benchmark maps provided by Nathan Sturtevant [1]. Moreover, we generated experiments with several different job-assignment strategies on the same maps and scenarios, and revealed the best performing of the experimented heuristics. We also integrated another incremental MAPF solver we generated before [80] to this idea and compared their results by using the best performing heuristic amongst the ones we

created. In the results, we observed that the D*-lite powered version of MAPF-MD solutions produces faster results as the number of target points added to the problem increases. Our contributions to this field were the new strategy (MD-DCBS) to solve MAPF-MD problem, and providing a heuristic for the job-assignment problem to find low-cost solutions.

4.1 Method

4.1.1 MD-DCBS

Require: Agent and path-ID $= \{a_i, k\}$

1: **if** No input provided **then** ▷ *Calculate path for all agents*
2: **for** each agent a_i in C **do**
3: **for** each destination f_j in F_i **do**
4: a_i.dstarList $\leftarrow$ a_i.dstarList + dstar-lite(f_{j-1},f_j) ▷ f_{j-1} *is* s_i *for the first destination*
5: a_i.pathList $\leftarrow$ a_i.pathList + dstar-lite(f_{j-1},f_j).plan()
6: a_i.path = aggregate(a_i.pathList)
7: **else** ▷ *Update a part of the path*
8: a_i.path = a_i.dstarList[k].plan()
9: a_i.path = aggregate(a_i.pathList)

Figure 4.2: low-level-search-MD function

The multiple delivery DCBS (MD-DCBS) algorithm has a flow similar to that of the CBS algorithm, with a couple of differences. The first difference is that it uses D*-lite instead of A* search in the low-level search. Adding new destinations to the system and introducing new constraints actually gives the system dynamical behavior. Due to its ability to cache previous search information, D*-lite is a better match than A* for dynamic environments, which is why we decided to use D*-lite instead of A*. We call the version of the CBS algorithm using D*-lite as the low-level search DCBS. Another big difference between the MD-DCBS and the CBS concerns the number of operations to handle multiple delivery locations of the agents. In MD-

DCBS we used more than one D*-lite instances per agent. Each agent has n locations to visit, 1 start location and $n-1$ destination locations. We need to plan a path for each successive destination, and for this we defined a D*-lite instance for each successive destination pair. We planned paths for each of the D*-lite instances and then aggregated all of the paths to generate an agents' *aggregated path*. The CBS is run on the aggregated paths of the agents. When a conflict occurs, the corresponding constraint is added to the concerned part of the aggregated path. Then the constraint is added to the corresponding D*-lite instance and the conflict is resolved by replanning that path. This new updated part of the path is then updated on the aggregated path and CBS will continue its process as normal. In terms of CBS nothing changes but we handled an agent's path as a combination of many smaller paths each of which is calculated by the D*-lite instances created for that agent. Figure 4.3 presents the MD-DCBS algorithm. After each new destination is added to the system the MD-DCBS algorithm is called again. F_x is the list where the destinations introduced to the system after the agents started their jobs are kept. We use the notation F_{x_i} to specify that ith element of the list F_x. We also use $|F_x|$ to denote the size of F_x. If a new destination is added to the system, then the job-assignment-heuristic decides which agent to assign that job to and when to visit that destination (lines 1-3). Then the assigned job is deleted from the F_x list (line 4) (the details of the job-assignment heuristic are provided in section 5.2 below). After that, the root node is initialized in the same way as the CBS. In MD-DCBS the low-level search works differently from that in the CBS, hence we named it low-level-search-MD. Figure 4.2 presents the structure of the low-level-search-MD function. The low-level-search-MD is the place where we define and plan paths with D*-lite instances. If the parameter set is empty it defines and calculates paths for each successive destination pairs and then aggregates (lines 1-6 - *low-level-search-MD*). This part is used at the beginning of the MD-DCBS. If an agent and the path-ID are provided, then the function only plans a path for that part and then aggregates the paths to update the overall path (lines 7-9 - *low-level-search-MD*). A graphic representation of this function is provided in Figure 4.4. In this representation, an agent and its destinations are provided above where the start point of the agent is the robot and the destinations are the houses. In the lower part, a D*-lite object is defined for each of the successive destination pairs ($S-D1$, $D1-D2$ and $D2-D3$). The aggregated path is the aggregations of

Require: MAPF-MD Instance $= \{A, E, V, S, F, F_x\}$

1: **if** $|F_x| > 0$ **then**
2: AgentID,order = Job-assignment-heuristic ($F_{x|F_x|}$)
3: $F_{AgentID}$.insert($F_{x|F_x|}$,order)
4: F_x = $F_x / F_{x|F_x|}$
5: root.constraints = $\emptyset$
6: root.solution = low-level-search-MD()
7: root.cost = $\sum_{root.solution}$
8: Insert root to Open-List
9: **while** Open-List NOT empty **do**
10: $P \leftarrow$ Best Node from Open-List
11: Validate paths in P until a conflict occurs
12: **if** P has no conflict **then**
13: **return** P.solution
14: $C \leftarrow$ first conflict (a_i, a_j, v, t) in P
15: **for** each agent a_i in C **do**
16: $A \leftarrow$ new-CT-node()
17: A.constraints $\leftarrow$ P.constraints
18: k = find-the-part-of-the-agent-path-conflicting(A,(a_i, v, t))
19: A.constraints[k] $\leftarrow$ P.constraints[k] + (a_i, v, t)
20: A.solution $\leftarrow$ P.solution
21: A.solution $\leftarrow$ low-level-search-MD(a_i,k) ▷ *Update solution*
22: A.cost = $\sum_{A.solution}$
23: **if** A.solution $< \infty$ **then** ▷ *Solution was found*
24: Insert A to Open-List

Figure 4.3: The MD-CBS algorithm

these paths with time-steps adjusted. When a new conflict occurs, the constraint is added to the part where it occurs and only that part is repaired. After this point, until the constraints are added to the agents (line 17) MD-DCBS works similarly to CBS. MD-DCBS determines which D*-lite objects path should be updated, and calls the low-level-search-MD with that information (lines 18-21). The remaining part of the

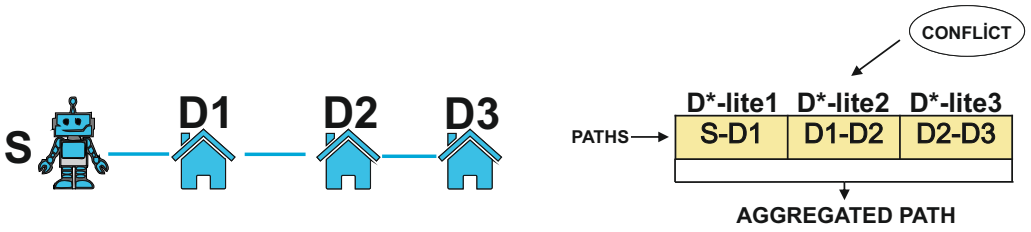


Figure 4.4: A graphical overview of the low-level-search-MD which is the low-level search mechanism of the MD-DCBS.

algorithm works identically to CBS.

4.1.2 Job-Assignment Heuristics

For this problem, the aim is to calculate the distance of the newly added target to the plans of the agents and to allocate it to the closest agent. In addition, after allocating to the target agent, the order of visit is also decided to be before or after the target point to which it is closest. At this stage, we conducted an experimental study. As new heuristics were created, we tested their results and selected the best performing algorithm. Since the objective function we used for the problem here is total path cost, we compared the created heuristic algorithms firstly according to the difference of the result they return from the optimal total path cost and then according to their calculation speed. The important point here is to calculate how far the chosen heuristic algorithms are from the optimal result. Although it is not possible to find the optimal solution for such incremental problems, it is possible to determine a realistic lower limit by trying all possible additions. Since trying all possible places where the new target can be added will cause an exponential problem, it will only be possible to use this metric for small-scale problems. While creating the heuristic algorithms in this study, we were inspired by the closeness centrality method in the graph theory literature [83, 84].

According to this method, in a connected graph, the closeness centrality (or closeness) of a node is a measure of centrality in a network calculated as the sum of the lengths of the shortest paths between the node and all other nodes in the graph. Thus we can

infer that, the more central a node is, the closer it is to all other nodes. Calculation of the closeness centrality is provided by the following equation:

$$C(x) = \frac{1}{\sum_y d(y,x)} \tag{41}$$

In the following sub-sections, we will talk about the heuristic methods we have developed, respectively. The path we followed while developing these methods is to apply the closeness centrality logic to this problem. Also our aim was to create simplified versions of this method that generate results by applying fewer operations. Each subsection will describe a job-assignment heuristic. For each approach, we have shared a graphical representation of that approach and its explanations. In these pictures, the circles represent the agent starting points, the squares represent the agent ending points, the pentagons represent the newly added target point to the system and the d_i values represent the distance value between the two nodes.

In the heuristic algorithms we present in the following sub-sections we defined t as the arrival time of each new job. The paths traversed by each agent until the moment t are recorded as the paths traveled, and the starting point of the agent is updated as the current location at the time t. Agents destinations are also updated (traversed destinations are deleted). Then the new job is added with the heuristics to this updated destination list. In this section, the proposed heuristics and their strategies are provided.

4.1.2.1 Add to Closest Start Agent

The Add to Closest Start Agent (ACSA) heuristic, makes a distance calculation for each agent. In this distance calculation, for each agent, the distance from agent's start point to the target point is calculated. Then, the agent with the minimum value of these calculated values is selected to add the target point. Then the target point is added to the selected agent plans to be visited just after the starting point of the selected agent. An illustration of this working mechanism is presented in Figure (Figure 4.5) where t represents the time step that the new target is added to the system. The

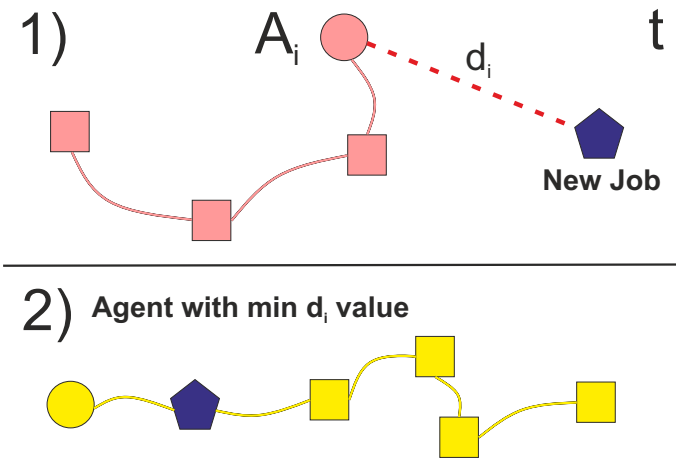


Figure 4.5: The Add to Closest Start Agent (ACSA) heuristic: 1) For each agent distance of its start location to new job location is calculated. 2) The new job is added after the start location of the agent with the minimum d_i distance.

sub-picture indicated by the number-1 (on the top) describe the distance calculation made for each agent. The sub-picture expressed with number-2 (below) describe how that target is added to the agent.

4.1.2.2 Add to Closest End Agent

The Add to Closest End Agent (ACA) heuristic, works in a similar way to the ACSA heuristic. As a difference, this heuristic calculates distances from the ending point to the target point instead of the start point. Other than that, the working mechanism of ACA is the same as ACSA. An illustration of this heuristic is represented in Figure 4.6.

4.1.2.3 Add to Closest Average Start End Points

The Add to Closest Average Start end Points (ACASP) heuristic works with the following mechanism. For all agents, the distance between the newly added target and the starting point of the agent is calculated. Then, the distance between the newly added target and the agent's endpoint is calculated. By taking the average of these

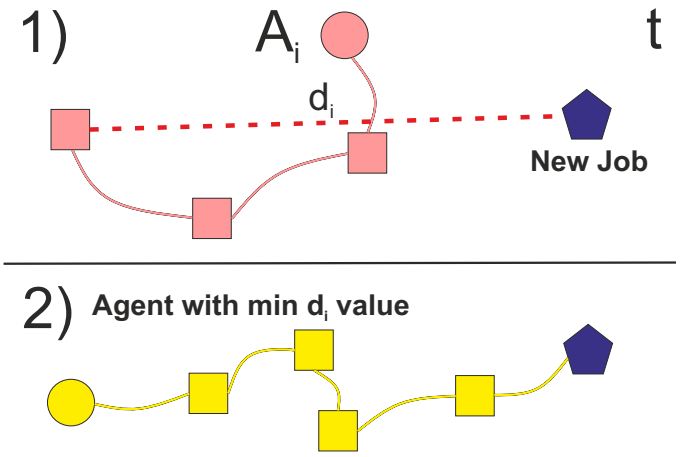


Figure 4.6: The Add to Closest End Agent (ACA) heuristic: 1) For each agent the distance of its last destination location to new job location is calculated. 2) The new job is added after the last destination location of the agent with the minimum d_i distance.

two distances, the value of that agent is calculated. The agent with the minimum value among all agents is selected for adding the new incoming target. In this algorithm, when the new incoming target will be visited between the target points of the agent is also decided differently from the first two heuristics. The distance of all target points of the selected agent to the new target point is calculated one by one. The target list of the agent is updated by adding the new incoming target after the target point where the shortest distance is calculated. An illustration of this heuristic is represented in Figure 4.7.

4.1.2.4 Add to Closest Point

The Add to Closest Point (ACP) heuristic is the closest approach to the closeness centrality approach among the developed heuristic algorithms. This could also be called the way we have adapted the closeness centrality approach to this problem. In this approach, we calculated the distance from all destination points of each agent to the newly added target point. The minimum of these distances is recorded as the calculated value for that agent. The agent with the minimum value among all agents

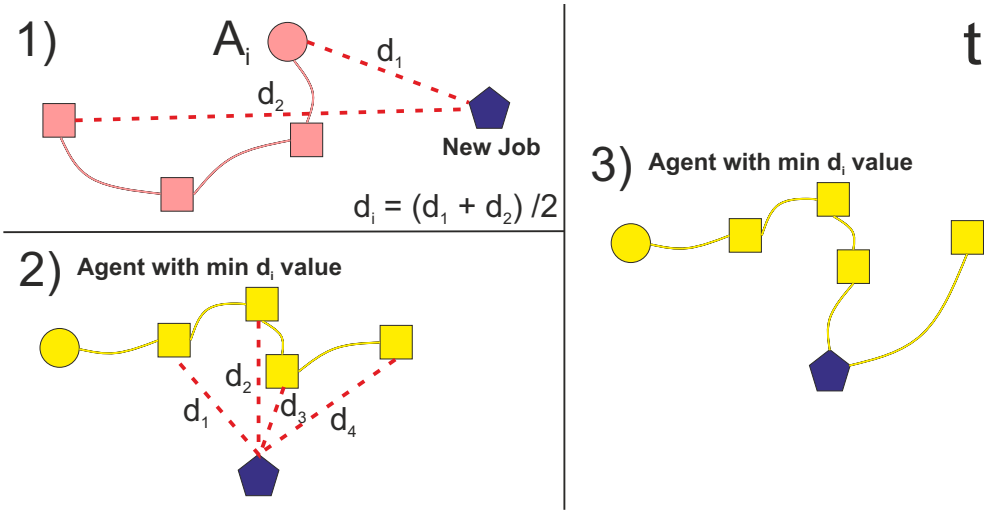


Figure 4.7: The Add to Closest Average Start End Points (ACASP) heuristic: 1) For each agent the distance of it from the new job location is calculated by averaging the distances of its start location and last destination location from the new job location. 2) The new job's distance from each of the destinations is calculated for the agent with the minimum d_i distance 3) The new job is added after the closest destination of the agent with the minimum d_i distance.

is selected for adding the new incoming target. We decided with the same method used in Section 4.1.2.3 how to add the newly added target point to the chosen agent. Accordingly, the new destination is added to the destination list of the agent after the nearest destination of the agent. An illustration of this heuristic is represented in Figure 4.8.

4.1.2.5 Add to closest average agent

The Add to Closest Average Agent (ACAA) heuristic calculates the distance of the newly added target for each agent from all the destinations of that agent. Then, it calculates the value determined for that agent by calculating the average of these distances. The agent for which this value is minimum is selected to add the new target point. The order in which the new target will be visited is decided by the method described in Section 4.1.2.3. An illustration of this heuristic is represented in Figure 4.9.

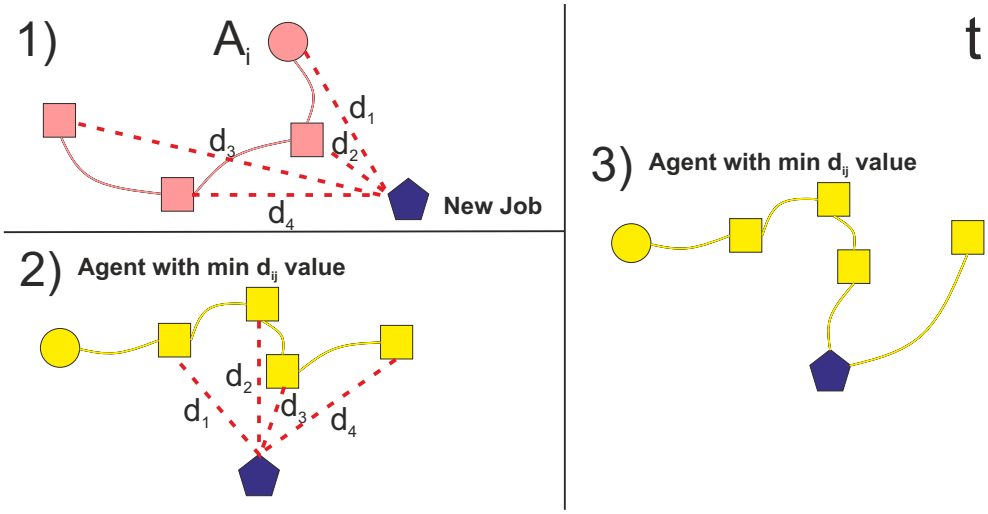


Figure 4.8: The Add to Closest Point (ACP) heuristic: 1) For each agent the distance from all destination points of that agent to the newly added target point is calculated. The minimum of these distances is recorded as the calculated value for that agent. 2) The new jobs distance from each of the destinations is calculated for the agent with the minimum d_i distance 3) The new job is added after the closest destination of the agent with the minimum d_i distance.

4.1.2.6 Best Possible Adding

The Best Possible Adding (BPA) heuristic is an algorithm we have developed to calculate the distances of other heuristics to a near-optimal lower limit. For this, for each agent, we have added the newly added target point to the destination list of that agent, with all possible orderings (trying one at a time). We found the total path cost by solving the total problem for each case with the modified CBS solver we developed. Then, the new target is added to the list of destinations of that agent in the order that gives the lowest total result from all these possibilities. An illustration of this heuristic is represented in Figure 4.10.

4.1.3 Theoretical Analysis

After each new job is added to the system, the optimal solution changes. We assumed that behaving optimally during each job-assignment process will provide an optimal

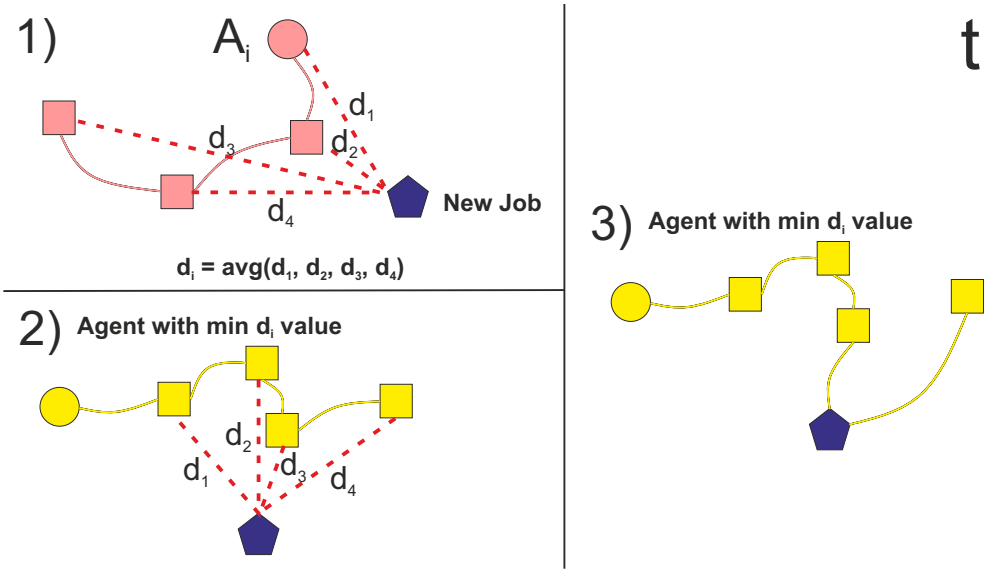


Figure 4.9: The Add to Closest Average Agent (ACAA) heuristic: 1) For each agent the distance of it from the new job location is calculated by first finding the distance of the newly added target for each agent from all the destinations of that agent. Then, it calculates the value determined for that agent by calculating the average of these distances. 2) The new job's distance from each of the destinations is calculated for the agent with the minimum d_i distance 3) The new job is added after the closest destination of the agent with the minimum d_i distance.

solution. The overall solution provided by this strategy does not have to be optimal every time. However, as we do not know in advance which new job will be added to the system, this is a realistic approach.

The only difference between DCBS and CBS is the use of D*-lite instead of A*. Both D*-lite and A* are optimal and complete approaches [82, 44]. [48] proved that CBS returns an *optimal* solution and it returns a solution if one exists (*complete*). DCBS uses the exact same tree search mechanism that CBS uses. Hence, DCBS is also an optimal and complete approach. In terms of running time, running DCBS multiple times after each new job assignment is faster than running CBS multiple times because DCBS uses D*-lite which can cache previous search information. MD-DCBS is also a complete and optimal approach apart from the job-assignment strategy. As we made job assignments via heuristics the overall algorithm is not optimal. After job

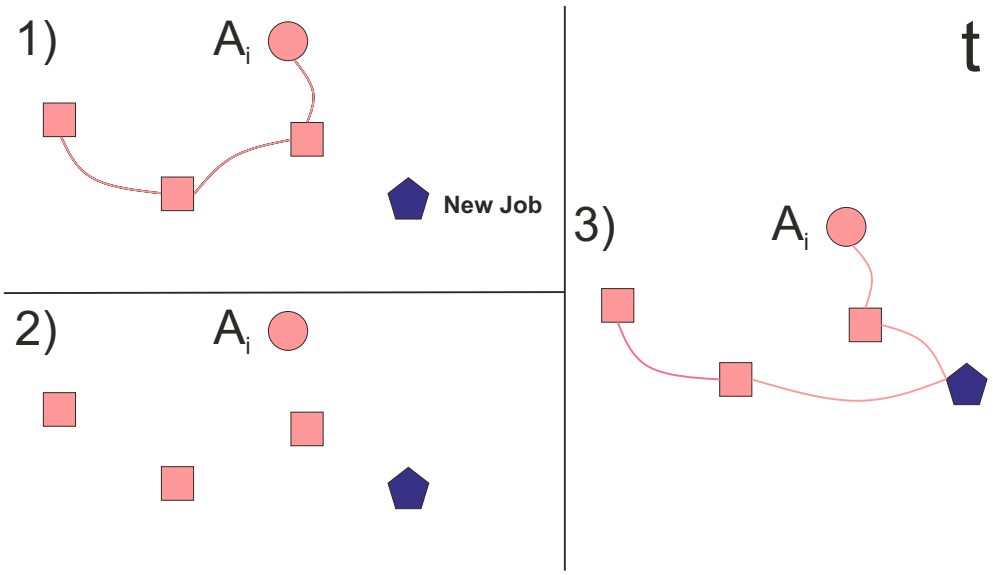


Figure 4.10: The Best Possible Adding (BPA) heuristic: 1) For each agent, the MAPF problem created by adding the newly added target point to that agent should be solved. 2) For each agent, the newly added destination point is added before and after all the elements of that agent's destination list, and the MAPF problem that occurs with that scenario is solved. 3) Out of all these solved MAPF problems, the assignment method that gives the minimum result is chosen as the solution.

assignment and aggregation, MD-DCBS provides the same input to the CBS search, and hence that part of the MD-DCBS is optimal and complete. If we assume that there are n destinations for each of the agents. This means that there will be n different path planning jobs for each agent when replanning is needed for that agent's path. This was 1 for each agent in the CBS algorithm. Apart from that, there will be a cost of aggregating paths. Each time a low-level search is called for an agent, an aggregation job is also performed (at the start of the MD-DCBS aggregation is performed for all agents). So the running time is roughly |CT nodes|$\times$($n\times$D*-lite search + aggregation cost). MD-DCBS is called again after each new job assignment. The overall solution is not optimal but it behaves optimally between each new job assignment.

We generated an all-pairs shortest path table before running these heuristics, so we did not recalculate distances when we call the heuristics. Apart from BPA, all heuristics are called MD-DCBS once to calculate the total cost. The distances are retrieved from

the table, hence they took $O(1)$ time. If there are n agents and k destinations for each agent, BPA calls MD-DCBS $n \times (k+2)$ times.

4.1.4 Running Example

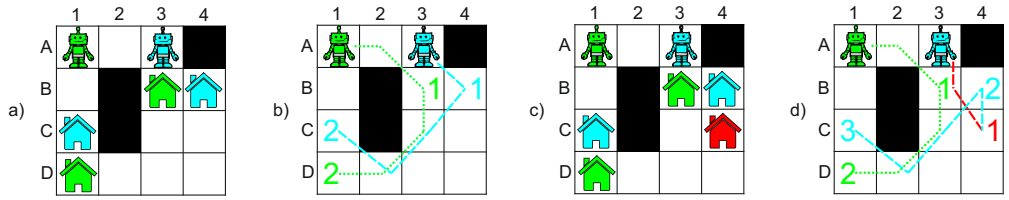


Figure 4.11: Running example; a) represents an initial multiple delivery problem, b) represents the initial agent plans generated by MD-DCBS, c) represents the new destination to be assigned, d) represents the updated paths after the destination is assigned and visited by closest start point adding strategy.

In Figure 4.11, a toy MAPF-MD problem is presented with two agents and two destinations to be visited for each agent. Agents are represented by robots and the destinations are represented by houses. Each agent and its destination is given the same color. After the agents start their jobs a new destination (shown in red) is added to the system. The initial system configuration is presented in Figure 4.11-a. The results after the MD-DCBS run are given in Table 4.1 where the graphical representation of MAPF-MD is depicted in Figure 4.11-b. A new destination is then added to the system (represented by a red house in Figure 4.11-c). Next, we used the ACSA heuristic to assign the new destination. The coordinate of the new destination is $C4$. The start point of the green agent is $A1$ and the start point of the blue agent is $A3$. Distance from $C4$ to $A1$ is $1.414 \times 3 + 1 = 5.242$ (there are 3 diagonal moves and a horizontal move) and the distance from $C4$ to $A3$ is $1.414 \times 1 + 1 = 2.414$. As the second distance is smaller, the new destination is added before the start point of the blue agent. In Figure 4.11-d, the resulting paths of the agents are presented. The path going to the new destination is presented with a red color. The paths generated after the new destination are given in Table 4.1.

Table 4.1: Multiple delivery MAPF solution before and after the new package.

MAPF-MD	**MAPF-MD after new job added**
{**1** - A1, A2, **B3**, C3, D2, **D1**}	{**1** - A1, A2, **B3**, C3, D2, **D1**}
{**2** - A3, **B4**, C3, D2, **C1**}	{**2** - A3, B3, **C4**, **B4**, C3, D2, **C1**}
Total cost = 11.4853	Total cost = 13.4853

4.2 Experimental Study

To our best knowledge, there are no previous attempts that focus on minimizing total cost after new job-assignment. For this reason, we compared the effect of the several job-assignment strategies we suggested (presented in the method section) on the total cost. We used the MD-DCBS algorithm to solve the multiple delivery MAPF problems.

As a second type of experiment, we integrated the proposed method in this study with a different low-level incremental planner and examined the effect of the lower-level planner on the outcome. For this, we integrated the incremental MAPF solver using LPA*, which we proposed in 2022 into the algorithm provided in this study [80]. The reason for this preference is that these two algorithms have similar infrastructures and the integration process can be performed more quickly. Using the best heuristic selected by these two studies, we solved the MAPF-MD problem and compared the performance of these algorithms.

We developed the project on a PC with a 64 bit 3.40 GHz Intel i7 processor. We used the Ubuntu 14.04.2 LTS operating system and C++ programming language for implementation.

4.2.1 Datasets

In this section, we talked about the datasets we created to test the heuristic algorithms developed for job assignment and the datasets we created to test the different methods we developed to solve the MAPF-MD problem, under separate headings.

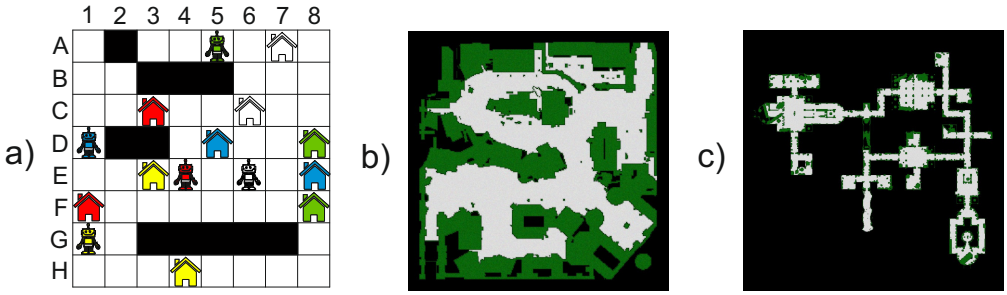


Figure 4.12: A graphical overview hand crafted map (a) and the benchmark maps den520d (b), brc202d (c) [1]

We used 3 different maps for the tests in both parts. The first of these maps is the 8×8 map. This map is a grid map that we created ourselves, which contains some obstacles. We thought it was a good option for examining the results of algorithms in small and cramped environments and for testing their accuracy in hand-crafted tests. The created 8×8 map is shared in Figure 4.12-a. The other two maps we used in the tests consist of maps from the benchmark set provided by Nathan Sturtevant [1]. We used the den520d and brc202d maps from the Dragon Age: Origins (DAO) game. The den520d and the brc202d maps are grid-like environments and have sizes of 257×256 and 481×530 respectively.

In our experiments, we randomly created the starting and goal points of the agents in a way that they do not coincide with the obstacles on the maps. Similarly, we randomly created the destinations to be added to the agents. The new destination locations were guaranteed to not collide with the previous destinations and the obstacles on the maps.

4.2.1.1 Datasets for Testing Heuristics

In the experiments with the 8×8 grid, we used five randomly generated agents with two destination locations and then added one new destination to these multi-agent plans. We generate results in 100 different scenarios. For the experiments on the den520d and brc202d maps, we used 10 randomly generated agents with two destination locations and then added one new destination to these multi-agent plans. We generated results in 100 different scenarios.

4.2.1.2 Datasets for Testing MAPF-MD Solvers

In the experiments with the 8×8 grid, we used five randomly generated agents with two destination locations and then added eight new destinations to these multi-agent plans. We generate results in 330 different scenarios. For the experiments on the den520d, we have two different settings. In the first one, we used five randomly generated agents with two destination locations and then added eight new destinations to the problem. We generated 260 in different scenarios for this setting. In the second setting, we wanted to test the algorithms with a higher number of agents and a higher number of destinations added. Hence, we used ten randomly generated agents with two destination locations and added twenty-four new destinations to the problem. We generated 156 in different scenarios for this case. Lastly, for the experiments on the brc202d map, we used five randomly generated agents with two destination locations and then added eight new destinations to the problem. We generated 166 in different scenarios for this case.

4.2.2 Comparison of the Job-Assignment Heuristics According to Total-Path Costs and Total Time Spent They Provide

In Table 4.2, we present the total-path-cost (left) and total elapsed time values (right) produced on three different maps by the heuristic algorithms. We calculated the average, minimum, and maximum of these values by using different job-assignment heuristics. These total cost and the elapsed-time values are calculated after the new destination is added to the system. The minimum values of the minimum, maximum, and average values are shown in bold in the tables.

In the results produced in 8×8 grid, for all three cases, the ACP method is closest to the BPA method which is the result of all possible combinations. ACSA is able to find a minimum solution but on average it provided worse results than the ACP. A similar situation occurred on maximum values with the ACAA and the ACA heuristics. The ranking according to average cost values is ACP $>$ ACAA $>$ ACASP $>$ ACA $>$ ACSA (from best to worst, excluding BPA). Throughout the experiments on the den520d map, again the ACP method provided the closest results to BPA. The al-

Table 4.2: Total path cost values (left) and total time spent (right) by running the several job-assignment heuristics we presented. 5 agents are used in 8×8 map, and for the den520d and brc202d maps 10 agents are used.

Map		Total Path Cost Values						Total Time Spent (sec)					
		ACSA	ACA	ACASP	ACP	ACAA	BPA	ACSA	ACA	ACASP	ACP	ACAA	BPA
8x8	Min.	**27.49**	29.49	29.49	**27.49**	30.90	27.49	**0.008**	**0.008**	0.011	0.012	0.011	0.181
	Max.	77.08	**76.01**	77.08	**76.01**	**76.01**	74.25	1.214	**1.200**	1.205	1.234	1.267	2.590
	Avg.	51.55	51.10	50.78	**50.19**	50.66	49.51	**0.029**	**0.029**	0.037	0.038	0.039	0.512
den520d	Min.	1032.3	929.28	1027.9	**922.6**	**922.6**	922.0	1.543	**1.444**	2.003	2.075	2.269	32.466
	Max.	2278.3	**2260.7**	**2260.7**	**2260.7**	**2260.7**	2243.5	5.740	**4.591**	7.411	6.930	7.681	63.211
	Avg.	1548.1	1544.2	1531.1	**1513.6**	1523.8	1500.1	2.876	**2.212**	3.490	3.413	4.647	41.148
brc202d	Min.	1243.3	1121.0	**1117.5**	**1117.5**	1193.8	1117.5	8.726	**8.060**	11.37	10.70	11.98	64.40
	Max.	3151.7	3352.9	3356.3	**3004.9**	3108.8	2982.9	25.82	**24.00**	26.92	27.16	32.44	131.1
	Avg.	1998.5	2015.6	1956.8	**1895.9**	1947.8	1852.4	11.79	**10.83**	15.25	14.30	18.87	85.72

gorithms which provided the minimum and the maximum total averages are changed. ACP and ACAA provided the minimum total cost and ACP, ACAA, ACASP, and ACA provided the maximum total cost. The ranking according to average cost values is identical to 8×8 grid experiments. This time the differences between the total cost values are larger in these results. In the experiments on the brc202d map, ACP is still the best-performing heuristic. ACP and ACASP provided the minimum path cost values. On the maximum path cost values, ACP is the one that is closest to BPA. On the average values, the ranking is similar to the previous experiments but this time the total path cost values provided by ACSA are smaller than the values provided by ACA.

In the results produced in the 8×8 grid, ACA is the algorithm that provided the results in the shortest amount of time on average. ACSA provided similar results to ACA. ACASP, ACP, and ACAAs results are close, and the ranking is ACASP, ACP, and ACAA. BPA worked clearly slower than the other heuristics as expected. ACSA provided the fastest result, and ACA is the one that spent the smallest amount of time among the maximum values provided by the heuristics.

4.2.3 Comparison of the MAPF-MD Solvers with Different Low-Level Solvers

In this section, we compared MAPF-MD solvers with different low-level solvers. The method proposed in this article, MD-DCBS, uses the low-level D*-lite algorithm, while the MD-DCBS with LIMP algorithm uses the low-level LPA* algorithm.

In Table 4.3, we shared the total-path-cost results provided by the algorithms in cases with 8 destinations added. The 0 state indicates the initial case. The agents have one start and two goal points at the beginning. After each destination is added, the job is assigned to one of the agents and the problem is resolved, and the total path cost for each addition is included in the table. Since the case of adding 24 destinations is too large to include in this table, we created a second table with a summary of the total-path-cost results. We shared a summary of the total path cost results of the algorithms in Table 4.4. In this summary, the average total path costs of the algorithms, the average total path cost differences of the algorithms, and the total-path-cost value for the case where the total-path cost difference is maximum are included for all datasets (including the 24 destination experiment). When we examine the results provided in Table 4.3 and Table 4.4 we can see that the two algorithms show close performances in terms of total-path cost.

Table 4.3: Comparison of total path cost results of MAPF-MD solvers with different low-level solvers on different maps.

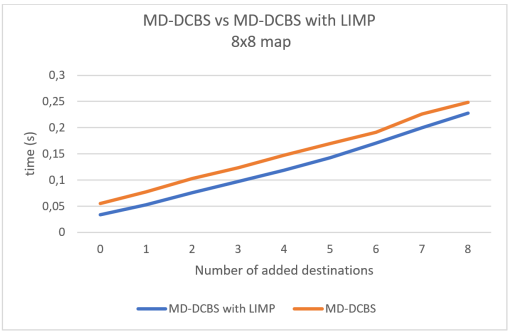
		Destinations Added								
		0	1	2	3	4	5	6	7	8
8x8 - 8	MD-DCBS	56,991	59,091	61,342	63,336	65,418	67,530	69,803	72,767	75,794
	MD-DCBS with LIMP	56,400	58,442	60,636	62,615	64,570	66,633	68,830	71,694	74,533
den520d - 8	MD-DCBS	1616,5	1673,0	1727,1	1781,2	1841,9	1903,6	1972,6	2033,3	2105,2
	MD-DCBS with LIMP	1620,8	1678,5	1732,1	1786,4	1847,1	1908,7	1978,5	2038,9	2112,2
brc202d - 8	MD-DCBS	2218,4	2302,1	2390,5	2484,4	2561,3	2654,9	2741,0	2842,7	2949,1
	MD-DCBS with LIMP	2218,2	2301,9	2390,3	2484,2	2561,1	2654,8	2740,9	2842,5	2948,9

A comparison of the running times of the algorithms is presented in Figure 4.13. According to these results, it is seen that the algorithms work similarly in terms of speed on the 8×8 map. However, as the map gets larger, it is seen that the difference in operating speed between the two algorithms increases in favor of MD-DCBS. In addition, in the tests performed by increasing the number of added destinations, it

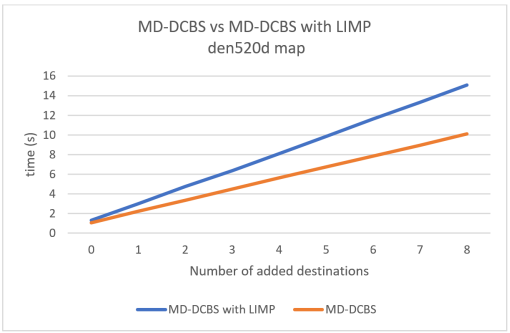
Table 4.4: Summary of total path cost results on different maps of MAPF-MD solvers with different low-level solvers.

	8x8 - 8			den520d - 8			brc202d - 8			den520d - 24		
Solver	avg-dif	max-dif	avg-cost	avg-dif	max-dif	avg-cost	avg-dif	max-dif	avg-cost	avg-dif	max-dif	avg-cost
MD-DCBS	0,8576	1,2606	65,786	-5,4235	-6,9885	1850,5	0,1647	0,1988	2571,6	0,3321	0,5513	3771,75
MD-DCBS with LIMP			64,928			1855,9			2571,4			3771,4

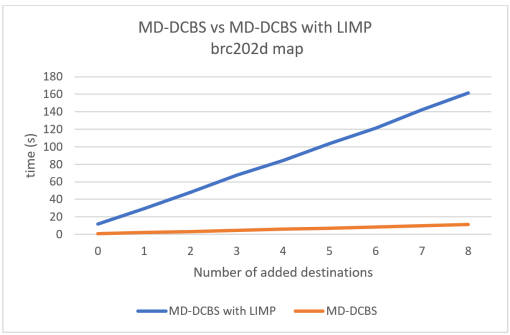
is seen that the algorithms produce similar results with the situations with a small number of destinations, but as the number of destinations added increases we see that the difference in the running time of these algorithms increased.



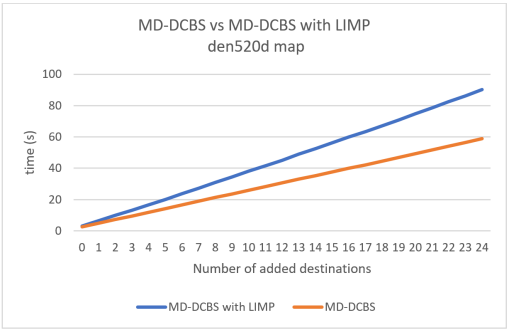
(a) 8×8 map - 8 destinations added



(b) den520d map - 8 destinations added



(c) brc202d map - 8 destinations added



(d) den520d map - 24 destinations added

Figure 4.13: Comparison of MAPF-MD solvers in terms of running time with different low-level solver integrations on different maps and different destination numbers.

CHAPTER 5

CONCLUSION

This section includes the problems we attacked in this thesis, the solutions we produced for those problems, and discussions on how to proceed in the future.

5.1 Summary

Since the CBS algorithm is not designed for incremental environments, it cannot cache previous information when the need for replanning arises. In this context, using an incremental single-agent search algorithm that can cache previous information in the lower-level search can speed up the low-level replanning process. Speeding up low-level planning is valuable, but not sufficient. When the working logic of CBS is examined, it can be observed that the main effect on the time spent during the operation of CBS is the number of nodes produced during the high-level search. Minimizing the total number of low-level planners run by reducing the number of nodes produced by the high-level search will further increase the replanning speed of the algorithm. That's why we decided on making modifications to the main structure of CBS to reduce the number of nodes that need to be produced in high-level search. Here, it is of great importance to do replanning quickly to adapt quickly to changes in incremental environments. For this reason, it seems appropriate to the nature of the problem to renounce the optimality and find fast and near-optimal quality results.

Another topic we were working on was generating a MAPF solver for multi-destination agents. The solution method we created here was to place a middle layer between the low-level search and the high-level search and thus produce optimal solutions after that the middle layer sends its computed paths to a high-level search. This would con-

serve standard CBS working logic. In the middle layer, the idea of considering the destinations as pairs and planning the paths between them with low-level planners assigned to them was our idea. Here we solved the incoming constraints in that planner and matched the time steps of different low-level planners to each other. Finally, we combined the created paths as a single path and send that single path to the high-level search. The only question mark that could arise here was the memory usage that the separately kept low-level planners would spend to keep the maps.

Lastly, in the lifelong MAPF problem, we wanted to plan a job distribution in such a way as to optimize the path total cost. Here it was logical to use heuristic methods to make a quick distribution when new jobs arrived. However, a limiting value was needed to measure the path cost quality of these created methods. For this, we have developed another algorithm that considers all possible job-distribution options, which are too slow to use under normal conditions, but which can help to understand the differences in the values of the developed methods from the optimal.

In this thesis, we created algorithms that can make fast replanning in incremental environments by basing on the CBS algorithm and modifying it. We designed an optimal approach for the MAPF problem involving multi-destination agents, and we created several heuristic algorithms for the job distribution problem.

Chapter 3 proposes two new algorithms to solve the I-MAPF problem. One of them is the CBS-replanner algorithm, which uses the CBS algorithm as it is and enables it to work incrementally. This algorithm guarantees to find the optimal result for each of the MAPF problems created after each environmental change. Since we cannot know the changes in advance, it does not seem possible to reach a global optimum for this problem. However, although it produces optimal results for the problems created after each change, it is considered too slow to use in real life, since it solves a new MAPF problem from scratch for each new change. The second developed algorithm is the CBS-D*-lite algorithm. In this algorithm, the D*-lite algorithm is used as a low-level planner. The D*-lite algorithm has been modified so that D*-lite can be integrated with CBS. Afterward, the CBS algorithm was also modified. Here, 3 stages are defined for the algorithm when replanning will be done. CBS-D*-lite follows a 3-stage strategy when replanning. The aim here is to achieve the result by

producing as few nodes as possible. With each new stage, slightly more nodes are generated. Stage-3 of this algorithm is exactly the same as what CBS-replanner does. But the main goal is to reach the conclusion in the first two stages in most scenarios. When the results are examined, it is seen that CBS-D*-lite performs replanning much faster than CBS-replanner and the path cost results are close to CBS-replanner.

Lastly, in Chapter 4, for the MAPF problem including multi-destination agents, we kept a separate D*-lite object for each destination pair. We replanned the route for the affected part of the path. This is achieved by adding the incoming constraint to the D*-lite object of the corresponding part of the agent path. Afterward, we combined this newly planned route with the routes previously planned by other D*-lite objects and made time adjustments. Finally, we ran the CBS algorithm with the aggregated path of all D*-lite objects. This resulting algorithm does not violate the optimal and complete structure of the CBS algorithm. However, keeping multiple D*-lite objects can consume a lot of memory in problems where the map is too large or when agents have too many destinations. In addition, we developed heuristic job distribution algorithms to solve the job distribution problem by optimizing the total path cost. Among these developed heuristic algorithms ACP, which was the algorithm whose working logic was the most similar to the closeness centrality method, gave the most balanced performance in terms of both working speed and total path cost performance for us. While the path costs it found were close to BPA, which we can see as a brute force approach, ACP gave a very fast performance compared to BPA in terms of working speed. Also, to explore the effect of using different low-level solvers on the MD-DCBS algorithm, we created a version of MD-DCBS that uses the LPA* algorithm as a low-level solver and compared these two algorithms in various situations. In the results, we observed that these two algorithms work closely in terms of total-path-cost, but in cases where the number of added destinations increases, the MD-DCBS algorithm produces faster results in running time.

5.2 Future Work

The studies presented within the scope of this thesis are considered to be developable in many respects.

Firstly, it has been observed that the solutions presented for the I-MAPF problem work at small delta values when considering the duration of an environmental change. It has been evaluated that the amount of replanning should be reduced in order for the solutions to be suitable for environmental changes with higher delta values. For this reason, the solution presented can be designed to group the environmental changes. By this way, making replanning for the groups of environmental changes can lead to less replanning.

Another idea is to combine the existing incremental problem ideas (such as adding agents to the problem at any time) with the I-MAPF problem, thus producing a solution to a problem that more accurately reflects real-life problems.

Finally, for the job distribution algorithms being developed to optimize the total path cost, it is evaluated that the jobs added to the system can be grouped with a clustering mechanism and assignments can be made to the agents in groups.

REFERENCES

[1] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[2] H. Ma and S. Koenig, "Ai buzzwords explained: Multi-agent path finding (mapf)," *AI Matters*, vol. 3, no. 3, p. 15–19, 2017.

[3] R. Stern, "Multi-agent path finding - an overview," in *RAAI Summer School*, 2019.

[4] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Barták, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *SOCS*, 2019.

[5] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Persistent and robust execution of mapf schedules in warehouses," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1125–1131, 2019.

[6] P. Dasler and D. M. Mount, "Online Algorithms for Warehouse Management," in *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, vol. 149 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 56:1–56:21, 2019.

[7] J. Stenzel and D. Luensch, "Concept of decentralized cooperative path conflict resolution for heterogeneous mobile robots," in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 715–720, 2016.

[8] R. Morris, C. Pasareanu, K. Luckow, W. Malik, H. Ma, T. K. S. Kumar, and S. Koenig, "Planning, scheduling and monitoring for airport surface operations," in *AAAI-16 Workshop on Planning for Hybrid Systems (PlanHS)*, 2016.

[9] J. Yu and S. M. LaValle, "Planning optimal paths for multiple robots on graphs," 2013.

[10] 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 *Twenty-Sixth International Conference on Automated Planning and Scheduling*, ICAPS'16, p. 477–485, 2016.

[11] H. Ma, S. Koenig, N. Ayanian, L. Cohen, W. Hönig, T. K. S. Kumar, T. Uras, H. Xu, C. A. Tovey, and G. Sharon, "Overview: Generalizations of multi-agent path finding to real-world scenarios," *CoRR*, vol. abs/1702.05515, 2017.

[12] A. Murano, G. Perelli, and S. Rubin, "Multi-agent path planning in known dynamic environments," in *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, pp. 218–231, 2015.

[13] B. Atiq, V. Patoglu, and E. Erdem, "Dynamic multi-agent path finding based on conflict resolution using answer set programming," in *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020*, vol. 325, pp. 223–229, 2020.

[14] Q. Wan, C. Gu, S. Sun, M. Chen, H. Huang, and X. Jia, "Lifelong multi-agent path finding in a dynamic environment," in *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pp. 875–882, 2018.

[15] F. Semiz and F. Polat, "Incremental multi-agent path finding," *Future Generation Computer Systems*, vol. 116, pp. 220–233, 2021.

[16] H. Ma, J. Li, T. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," AAMAS '17, p. 837–845, 2017.

[17] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 11272–11281, May 2021.

[18] H. Ma, W. Hönig, T. K. S. Kumar, N. Ayanian, and S. Koenig, "Lifelong path planning with kinematic constraints for multi-agent pickup and delivery," *CoRR*, vol. abs/1812.06355, 2018.

[19] M. Liu, H. Ma, J. Li, and S. Koenig, "Task and path planning for multi-agent pickup and delivery," in *AAMAS*, 2019.

[20] Z. Chen, J. Alonso-Mora, X. Bai, D. D. Harabor, and P. J. Stuckey, "Integrated task assignment and path planning for capacitated multi-agent pickup and delivery," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5816–5823, 2021.

[21] O. Salzman and R. Stern, "Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems," AAMAS '20, p. 1711–1715, 2020.

[22] K. Yu, "Finding a natural-looking path by using generalized visibility graphs," in *PRICAI 2006: Trends in Artificial Intelligence, 9th Pacific Rim International Conference on Artificial Intelligence, Proceedings* (Q. Yang and G. I. Webb, eds.), vol. 4099 of *Lecture Notes in Computer Science*, pp. 170–179, 2006.

[23] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, "Visibility graphs," in *Computational geometry*, pp. 307–317, Springer, 2000.

[24] E. J. Gómez, F. M. Martínez Santa, and F. H. M. Sarmiento, "A comparative study of geometric path planning methods for a mobile robot: Potential field and voronoi diagrams," in *2013 II International Congress of Engineering Mechatronics and Automation (CIIMA)*, pp. 1–6, 2013.

[25] F. Lingelbach, "Path planning using probabilistic cell decomposition," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 1, pp. 467–472, IEEE, 2004.

[26] R. Gonzalez, M. Kloetzer, and C. Mahulea, "Comparative study of trajectories resulted from cell decomposition path planning approaches," in *2017 21st International Conference on System Theory, Control and Computing (ICSTCC)*, pp. 49–54, IEEE, 2017.

[27] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, and W. Burgard, *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

[28] K. N. McGuire, G. C. de Croon, and K. Tuyls, "A comparative study of bug algorithms for robot navigation," *Robotics and Autonomous Systems*, vol. 121, p. 103261, 2019.

[29] D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristicbased path planning," in *in: Proceedings of the Workshop on Planning under Uncertainty for Autonomous Systems at The International Conference on Automated Planning and Scheduling (ICAPS*, 2005.

[30] S. Aggarwal and N. Kumar, "Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges," *Computer Communications*, vol. 149, pp. 270–299, 2020.

[31] F. Semiz and F. Polat, "Solving the area coverage problem with uavs: A vehicle routing with time windows variation," *Robotics and Autonomous Systems*, vol. 126, p. 103435, 2020.

[32] B. Fu, L. Chen, Y. Zhou, D. Zheng, Z. Wei, J. Dai, and H. Pan, "An improved a* algorithm for the industrial robot path planning with high success rate and short length," *Robotics and Autonomous Systems*, vol. 106, pp. 26–37, 2018.

[33] G. Jagadeesh, T. Srikanthan, and K. Quek, "Heuristic techniques for accelerating hierarchical routing on road networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 3, no. 4, pp. 301–309, 2002.

[34] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.

[35] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy, "Incremental heuristic search in ai," *AI Magazine*, vol. 25, no. 2, pp. 99–99, 2004.

[36] V. Lumelsky and A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *IEEE Transactions on Automatic Control*, vol. 31, no. 11, pp. 1058–1063, 1986.

[37] A. Pirzadeh and W. Snyder, "A unified solution to coverage and search in explored and unexplored terrains using indirect control," in *Proceedings., IEEE International Conference on Robotics and Automation*, pp. 2113–2119 vol.3, 1990.

[38] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2, pp. 189–211, 1990.

[39] A. Zelinsky, "A mobile robot exploration algorithm," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 6, pp. 707–717, 1992.

[40] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *INTERNATIONAL JOURNAL OF ROBOTICS AND AUTOMATION*, vol. 10, pp. 89–100, 1993.

[41] A. Stentz, "The focussed d* algorithm for real-time replanning," in *In Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1652–1659, 1995.

[42] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-path problem," *Journal of Algorithms*, vol. 21, no. 2, pp. 267–305, 1996.

[43] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artificial Intelligence*, vol. 155, no. 1, pp. 93–146, 2004.

[44] S. Koenig and M. Likhachev, "Dlite," in *Eighteenth National Conference on Artificial Intelligence*, p. 476–483, 2002.

[45] S. T. Scott, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pp. 173–178, AAAI Press, 2010.

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

[47] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton, "Algorithms for discrete function manipulation," in *1990 IEEE international conference on computer-aided design*, pp. 92–93, IEEE Computer Society, 1990.

[48] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artif. Intell.*, vol. 219, pp. 40–66, Feb. 2015.

[49] P. Surynek, "Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories," in *Proceedings of the*

Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, pp. 1177–1183, International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[50] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[51] E. Boyarski, A. Felner, D. Harabor, P. J. Stuckey, L. Cohen, J. Li, and S. Koenig, "Iterative-deepening conflict-based search," in *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pp. 4084–4090, 2021.

[52] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives," in *Ninth Annual Symposium on Combinatorial Search*, 2016.

[53] A. Andreychuk, K. Yakovlev, P. Surynek, D. Atzmon, and R. Stern, "Multi-agent pathfinding with continuous time," *Artificial Intelligence*, p. 103662, 2022.

[54] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding.," in *IJCAI*, pp. 1289–1296, 2019.

[55] R. N. Gómez, C. Hernández, and J. A. Baier, "Solving sum-of-costs multi-agent pathfinding with answer-set programming," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 9867–9874, 2020.

[56] A. Bogatarkan and E. Erdem, "Explanation generation for multi-modal multi-agent path finding with optimal resource utilization using answer set programming," *Theory and Practice of Logic Programming*, vol. 20, no. 6, pp. 974–989, 2020.

[57] E. Erdem, D. G. Kisa, U. Oztok, and P. Schüller, "A general formal framework for pathfinding problems with multiple agents," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[58] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, "Enhanced partial expansion a," *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014.

[59] T. Yoshizumi, T. Miura, and T. Ishida, "A* with partial expansion for large branching factor problems.," in *AAAI/IAAI*, pp. 923–929, 2000.

[60] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *2011 IEEE/RSJ international conference on intelligent robots and systems*, pp. 3260–3267, IEEE, 2011.

[61] D. Silver, "Cooperative pathfinding.," *Aiide*, vol. 1, pp. 117–122, 2005.

[62] M. M. Khorshid, R. C. Holte, and N. R. Sturtevant, "A polynomial-time algorithm for non-optimal multi-agent pathfinding," in *Fourth Annual Symposium on Combinatorial Search*, 2011.

[63] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3268–3275, IEEE, 2011.

[64] Q. Sajid, R. Luna, and K. E. Bekris, "Multi-agent pathfinding with simultaneous execution of single-agent primitives.," in *SoCS*, 2012.

[65] B. de Wilde, A. W. ter Mors, and C. Witteveen, "Push and rotate: cooperative multi-agent path planning," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pp. 87–94, 2013.

[66] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI magazine*, vol. 29, no. 1, pp. 9–9, 2008.

[67] R. Morris, C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. S. Kumar, and S. Koenig, "Planning, scheduling and monitoring for airport surface operations," in *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[68] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," *arXiv preprint arXiv:1612.05693*, 2016.

[69] L. Cohen and S. Koenig, "Bounded suboptimal multi-agent path finding using highways.," in *IJCAI*, pp. 3978–3979, 2016.

[70] Q. Wan, C. Gu, S. Sun, M. Chen, H. Huang, and X. Jia, "Lifelong multi-agent path finding in a dynamic environment," in *2018 15th International Conference*

on Control, Automation, Robotics and Vision (ICARCV), pp. 875–882, IEEE, 2018.

[71] A. Bogatarkan, V. Patoglu, and E. Erdem, "A declarative method for dynamic multi-agent path finding.," in *GCAI*, pp. 54–67, 2019.

[72] B. Coltin, "Multi-agent pickup and delivery planning with transfers," 2014.

[73] G. Berbeglia, J.-F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," *European journal of operational research*, vol. 202, no. 1, pp. 8–15, 2010.

[74] M. Liu, H. Ma, J. Li, and S. Koenig, "Task and path planning for multi-agent pickup and delivery," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2019.

[75] H. Ma, J. Li, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," *arXiv preprint arXiv:1705.10868*, 2017.

[76] W. Hönig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2018.

[77] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[78] X. Wu, Y. Liu, X. Tang, W. Cai, F. Bai, G. Khonstantine, and G. Zhao, "Multi-agent pickup and delivery with task deadlines," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 12, pp. 206–208, 2021.

[79] Z. Chen, J. Alonso-Mora, X. Bai, D. D. Harabor, and P. J. Stuckey, "Integrated task assignment and path planning for capacitated multi-agent pickup and delivery," *IEEE Robotics and Automation Letters*, 2021.

[80] M. A. Yorgancı., F. Semiz., and F. Polat., "Limp: Incremental multi-agent path planning with lpa," in *Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART,*, pp. 208–215, SciTePress, 2022.

[81] J. Neufeld and A. Sredzki, "dstar-lite." https://github.com/ArekSredzki/dstar-lite/blob/master/Dstar.cpp, 2015.

[82] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[83] A. Bavelas, "Communication patterns in task-oriented groups," *The journal of the acoustical society of America*, vol. 22, no. 6, pp. 725–730, 1950.

[84] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, no. 4, pp. 581–603, 1966.

CURRICULUM VITAE

Personal Information

Surname, Name	:	Semiz, Fatih
Nationality	:	Turkish

Education

Degree	Institution	Year of Graduation
M.Sc.	METU Computer Engineering	2015
B.Sc.	METU Computer Engineering	2012

Work Experience

Year	Place	Enrollment
2021-Present	ASELSAN Inc.	Software Engineer
2012-2021	METU - Computer Eng. Dep.	Research, Teaching Assistant
2011 June-Jan.	KOVAN Research Lab.	Intern, Part Time Researcher
2010 June-Dec.	MODSIMMER	Intern, Part Time Researcher

Foreign Languages

Advanced English

Publications

1. Semiz Fatih, and Polat Faruk. "Incremental multi-agent path finding." Future Generation Computer Systems 116: 220-233, 2021.

2. Yorgancı Mucahit, Semiz Fatih, and Polat Faruk. "LIMP: Incremental Multi-agent Path Planning with LPA". Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART) - Volume 1: 208-215, 2022.

3. Semiz Fatih, and Polat Faruk. "Solving the area coverage problem with UAVs: A vehicle routing with time windows variation." Robotics and Autonomous Systems 126: 103435, 2020.

4. Bender Bahar, Atasoy Mehmet Emre, and Semiz Fatih. "Deep Learning-Based Human and Vehicle Detection in Drone Videos." 2021 6th International Conference on Computer Science and Engineering (UBMK). IEEE, 2021.

5. Seylan Çağlar, Semiz Fatih, and Bican Özgür Saygın. "İnsansız araçlarla düzlemsel olmayan alanların taranması." Savunma Bilimleri Dergisi 11.1: 107-117, 2012.

Interests and Hobbies

Basketball, Photography, Movies