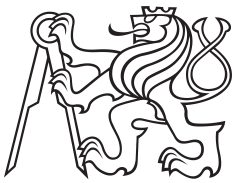
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Robust Plan Execution in Multi-Agent Systems

Josef Weis

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Weis Josef** Personal ID number: **483701**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Robust Plan Execution in Multi-Agent Systems

Bachelor’s thesis title in Czech:

Robustní realizace plánů v multi-agentních systémech

Guidelines:

Multi-Agent Path Finding aims to find the optimal collision-free trajectory for a team of mobile agents (robots) from their starting positions to given destinations. However, unexpected events may occur during plan execution that prevent the plan from being realized as designed. It is thus necessary to monitor whether such events have occurred and subsequently correct the plan. The student's task will be:
1) Get acquainted with the simulator for multi-agent planning (https://github.com/Kei18/mapf-IR).
2) Implement a tool that will make the correct plan invalid and modify the simulator so that it can work with such plans.
3) Implement the algorithms from article [1] and incorporate it into the simulator.
4) Design and implement a method which monitors plan execution and detects possible collisions.
5) Perform experimental verification of functionality and properties of the implemented algorithms and document these.

Bibliography / sources:

[1] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham and N. Ayanian, "Persistent and Robust Execution of MAPF Schedules in Warehouses," in IEEE Robotics and Automation Letters, vol. 4, no. 2, pp. 1125-1131, April 2019, doi: 10.1109/LRA.2019.2894217.
[2] K. Okumura, Y. Tamura and X. Défago, "Iterative Refinement for Real-Time Multi-Robot Path Planning," 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021, pp. 9690-9697, doi: 10.1109/IROS51168.2021.9636071.
[3] Hönig, W., Kumar, T. S., Cohen, L., Ma, H., Xu, H., Ayanian, N., & Koenig, S. (2016, March). Multi-agent path finding with kinematic constraints. In Twenty-Sixth International Conference on Automated Planning and Scheduling.

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **16.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I would like to express my thanks to the supervisor RNDr. Miroslav Kulich, Ph.D., for providing me constructive and supportive guidance on a regular basis and eagerly discussing any problem or question I had.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 20. May 2022

.....

Abstract

Multi-Agent Path Finding (MAPF) is a problem of finding a collision-free plan consisting of paths for multiple agents in which the agents do not collide with each other. The realization of these paths may not be as planned for various reasons, resulting in collisions. The most common issue is a delay of an agent. The robustness in the MAPF refers to a guarantee that the agents will not collide during plan execution. The approach used in this thesis is the Action Dependency Graph (ADG), which captures the action-precedence relationships of the originally created plan. Using this graph while monitoring plan execution, it is possible to synchronize agents' movements to avoid collisions. The thesis discusses the properties of this approach and its use in replanning during plan execution, and lays a solid foundation for experimental verification of the approach's functionality and properties, in the form of a modified simulator and supporting algorithms.

Keywords: ADG, Multi-Agent Path Finding, Robust plan execution

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Intelligent and Mobile Robotics, CIIRC
Jugoslávských partyzánů 1580/3
160 00 Praha 6

Abstrakt

Multi-agentní plánování cest (Multi-Agent Path Finding) je problém spočívající v hledání bezkolizního plánu skládajícího se z cest pro větší počet agentů, ve kterém se agenti mezi sebou nekolidují. Realizace těchto cest nemusí být přesně taková, jak byla původně naplánována z různých důvodů. Nejčastějším problémem je zpoždění agenta. Robusností se v MAPF rozumí garancí, že agenti nebudou v realizaci plánu kolidovat. Přístup, který je použit v této práci je Graf akčních závislostí (ADG), který zachycuje vztahy akcí z hlediska přednosti v původně vytvořeném plánu. Používáním tohoto grafu při monitorování exekuce plánu je možné synchronizovat pohyby agentů a předejít tak kolizím. Tato práce se zabývá vlastnostmi tohoto přístupu, jeho použitím při přeplánování během exekuce plánu a pokládá pevný základ pro experimentální ověření vlastností a funkčnosti přístupu ve formě modifikovaného simulátoru a vedlejších algoritmů.

Klíčová slova: ADG, Multi-agentní plánování, Robustní realizace plánu

Překlad názvu: Robustní realizace plánů v multi-agentních systémech

Contents

Chapter 1

Introduction

Multi-Agent Pathfinding (MAPF) is a problem of planning paths for multiple agents so that the agents will be able to follow these paths concurrently without colliding with each other [SSF+19]. Its contemporary applications include situations where multiple agents need to synchronize their actions to finish the tasks without collision, such as automated warehouses and autonomous vehicles.

Most MAPF solvers provide a collision-free plan but with the assumption that agents always move synchronously. In practice, this may not be the case, and for various reasons, an agent can be delayed in the execution of its plan, which can cause undesirable collisions. Multiple approaches have been introduced to address this issue, guaranteeing that agents will not collide despite delays - this guarantee is referred to as robustness.

This thesis studies one of the state-of-the-art approaches, the Action Dependency Graph (ADG) framework. The main objective is to implement the ADG, incorporate it into a simulator, and verify its functionality and properties. This is related to another task - implementing a tool that will make the correct plan invalid and modifying the provided simulator so that it can work with such plans. Simulation of invalid plans may result in a possible collision; therefore, implementing a method of monitoring plan execution and detecting possible collisions is desirable. Furthermore, the advantages of replanning in the middle of plan execution are discussed, and related algorithms are tested for functionality and properties.

In chapter 2, related work is summarized to provide insight into the MAPF problem-solving. The problem and notation used throughout the thesis are defined in chapter 3. Chapter 4 describes in detail the ADG approach. Chapter 5 presents a way of introducing delays to an existing plan that is later used to demonstrate the ADG functionality. Chapter 6 discusses the plausibility of replanning during plan execution and connects it to the ideas stated in the previous chapters. Chapter 7 shows multiple experiments using the ADG in simulations; Chapter 8 shortly describes the implementation, and Chapter 9 concludes the thesis.

Chapter 2

Related work

2.1 MAPF solvers

This section introduces some of the state-of-the-art solvers, their properties, and their principles. There are many other methods solving MAPF problem not listed in here as it is not the aim of this thesis to encompass them in detail. The mentioned methods are related to grid pathfinding and they all present different approaches to the MAPF problem.

Solvers are divided by their method of finding a solution into three categories [FSS+17], [OMDT19]: (1) *Search-based* - uses space-searching algorithms such as modified A* or tree-search to obtain results. (2) *Reduction-based* - reduces the MAPF problem to other known problems such as Boolean satisfiability problem (SAT), integer linear programming or Constraint satisfaction problem (CSP). (3) *Rule-based* - makes agent move step-by-step following ad-hoc rules. Hybrid solvers make use of more than one method listed above.

Conflict-based search (CBS) [SSFS21] is an optimal and complete search-based solver. It tests possible solutions and generates constraints to resolve conflicts. A constraint is a tuple (a_i, v, t) that prohibits agent a_i from occupying vertex v at time step t. The solution with the lowest cost, path respecting imposed constraints, and no inter-agent conflicts is selected [FSS+17].

Enhanced CBS (ECBS) [BSSF14] is a complete and suboptimal search-based solver. It uses focal search in both high-level and low-level searches. The purpose of the low-level search is to find single-agent paths. In contrast, the high-level search is used to find the most promising solution, validating it with respect to being collision-free and creating constraints should any collision be found. The focal search narrows the search only to nodes of particular interest. The suboptimal factor w determines the range of quality of the returned solution. SOC of the solution will never be higher than the SOC of the optimal solution multiplied by w.

Hierarchical Cooperative A* (HCA*) [Sil21] is an incomplete and suboptimal search-based solver. Agents are sorted according to predefined order, such as the likelihood of a collision. Paths are planned for agents with respect to the order of agents and then added to the global reservation table. Any agent may not occupy a specific location at specific times if other agents already reserve it [FSS+17].

Push and Swap (PS) [LB11] uses two primitives - `push`, which moves an agent toward its goal, and `swap`, which swaps two agents' positions, using empty tiles in the process. PS is complete if the graph contains at least two empty tiles and is suboptimal. It falls into the rule-based solvers' category [OTD21].

Revised Prioritized Planning (RPP) [CNKS15] assigns priorities to agents, and plans paths for agents one by one so that it avoids start positions of lower priority agents and avoid conflicts with higher priority agents. A solution is reached if each agent has a satisfying (goal-reaching) path that avoids all lower priority agents' starting positions and all goal positions of higher priority agents. RPP is an incomplete and suboptimal search-based solver.

Priority Inheritance with Backtracking (PIBT) [OMDT19] is an incomplete, suboptimal rule-based solver. PIBT plans every time step the location of all agents for the next time step until a solution is reached. It ensures that all agents reach their goals eventually, but they might not be at their goals simultaneously, making it incomplete [OTD21].

2.2 Improving robustness of existing plan

k-robust solvers [ASF+18] are MAPF solvers which have as an output a plan with a guarantee of being collision-free even considering at most k time step

delays for each agent. In other words, the plan does not have k-delay conflict that occurs if and only if $\pi_i(t) = \pi_j(t + \Delta)$ for $\Delta \in [0, k]$, where π is a plan, t is a time step, i and j denote different agents. k-robust solvers may be created as a variant of existing non-robust MAPF solvers.

MAPF-POST [HKC+16] is a post-processing step that takes kinematic constraints and edge lengths into account and constructs a simple temporal network based on the precedence relation of MAPF plan schedule. Using this simple temporal network, agents maintain a safe distance so that they will not collide.

RMTRACK [GCF04] addresses the delay possibility as a control law. An agent will not cross an intersection area before another robot that was supposed to cross the area first [HKT+19].

Chapter 3

Problem setting

3.1 Problem formulation

This section defines important MAPF terms commonly used in related literature and this thesis. They were mostly taken from papers [SSF+19] and [FSS+17], which offer a much broader overview of the MAPF problem than it is stated here. Any other used sources are properly cited in the text.

The classical MAPF problem with R agents is a tuple $<G,\ s,\ t>$, where $G = (V,\ E)$ is an undirected graph, V is a set of vertices and E a set of edges. s:[1, ..., R] $\rightarrow V$ maps an agent to source vertex (also referred to start position), and t: [1, ..., R] maps an agent to a target vertex (also referred to goal position).

An action is a function a:$V \rightarrow V$ such that $a(v) = v$', where v and v' are vertices. Performing action a on the agent in vertex v will result in the agent being in vertex v' next time step - a discretized time unit. Two types of actions are available for an agent. The first one is `move`, in which the agent proceeds to change the position from vertex v to another vertex v' for which an edge $(v, v') \in E$ exists. The second action is `wait`, where the agent waits at its current vertex position. There are two assumptions in the Classical MAPF regarding time: (1) time is discretized into time steps, (2) every action takes exactly one time step.

A sequence of actions $\pi = (a_1, ..., a_n)$ is called a single-agent plan or path, if it leads from starting position to goal position for a particular agent. A solution is a set of k single-agent plans, one for each agent. A solution is valid if it is collision-free.

MAPF solvers produce collision-free plans, by taking into consideration the notion of conflicts during planning. There are two main types of conflicts that MAPF solvers generally consider - vertex conflict and swapping conflict. Vertex conflicts occurs if two agent are planned to occupy the same vertex at the same time step. Formally, this can be written as follows:

$$\pi_i[t] = \pi_j[t]$$

where π is a single agent plan, i and j denote different agents and t is a time step. Swapping conflict occurs when two agents are planned to swap positions in a single time step. Formally, this can be written as follows:

$$\pi_i[t+1] = \pi_j[t] \; and \; \pi_i[t+1] = \pi_j[t]$$

where π is a single agent plan, i and j denote different agents and t is a time step.

The MAPF variant addressed in the thesis is MAPF in 4-neighbor grids. In this variant, every vertex represents a cell in a two-dimensional grid, and the move action of an agent is limited to 4 neighboring cells, i.e. moving to the right, left, lower, or upper cell. If an agent's cell is next to the border of the grid or next to an obstacle, its number of neighboring cells is lower, and so is the number of possible move actions.

The two most commonly used functions for evaluating solution quality are Sum of costs (SOC) and Makespan. The cost is the sum of time steps in a single-agent plan from the start position to the goal position, where the agent stays for the rest of the plan. The SOC is a sum of all single-agent plans costs in the solution. For a MAPF solution $\pi = (\pi_1, ..., \pi_R)$, the SOC of π is defined as

$$SOC = \sum_{1 \leq i \leq R} |\pi_i|$$

Makespan is the first time step in which all agents stay at their goal positions. For a MAPF solution $\pi = (\pi_1, ..., \pi_R)$, the makespan of π is defined as

$$M = max_{1 \leq i \leq R} |\pi_i|$$

Finding an optimal MAPF solution is an NP-hard problem [YL13]. With the increasing number of agents, finding a solution becomes more computationally demanding as state-space grows exponentially. MAPF solver is an

algorithm that returns a solution to the MAPF problem. Optimal MAPF solvers focus only on finding optimal solutions, while suboptimal MAPF solvers sacrifice optimality for faster runtime. A solver is complete if it always finds a valid solution to the MAPF problem if there is one.

Realization of the plan must take into account that the assumption of executing every action exactly in one time step is unrealistic in practice, as robots do not always move at the same velocity due to being subjected to higher-degree dynamic constraints and various other slowdowns. When a robot is slowed down in plan execution, other agents may perform an action leading to the same vertex as the one occupied by the slowed agent, thus creating a collision. Realization of the plan should address this issue and prevent collisions from happening.

Chapter 4

Agent delay

As physical robots are subject to dynamic constraints, control inaccuracies, and unforeseen slowdowns [HKT+19], they may be delayed in the execution of planned movements, creating the possibility of collisions. A simulation environment that does not consider these limitations should execute agents' actions exactly as planned. In such an environment, it is desirable to introduce delays in the initially collision-free plan to verify the functionality of robust-increasing frameworks and test their properties. This chapter presents an algorithm that easily introduces an arbitrary number of delays to an arbitrary number of agents in any time step.

Before executing the delaying algorithm, it is necessary to obtain information about the plan and intended delays. Delay can be characterized by three integers - ID of delayed agent dA, starting time step of delay sT, and the number of time step units to be delayed d. As for the plan, the essential features are the number of agents R, makespan M, the goal of the delayed agent g, and a positions matrix pos encoding the positions of all agents with shape $R \times M$. Examples of position matrices are shown in Tables 4.1-4.2. Because multiple delays can be intended, some of the information above should be stored in an array: let a delay be an array of tuples (dA, sT, d). Then, the array index should correspond to a particular intended delay.

The algorithm for delaying one agent is presented in Algorithm 1 and is structured into three phases. In the first phase, a position of a delayed agent at the start of delay is d times inserted into the pos, with the first dimension index being the delayed agent ID and the second starting time step of delay. This prolongs the path of the delayed agent by d time steps while other

paths remain unchanged. In the second phase, there is a cycle in which the penultimate position in the delayed agents' path is checked whether it is the goal position. If the statement is true, it would also mean the last position is the goal position, and that last position can be removed from the path. If the statement is false, prolonging the agents' path also increases makespan, and the increment is stored in the *timeStepsToAdd* variable. In the third phase, all other agent's paths are prolonged to match the new makespan by adding goal positions at the end of the respective agent's path. The algorithm's output is a new makespan and a new position matrix with one delayed agent, which can be later converted to the original plan format.

Algorithm 1 Delay Agent

Input: $delay, R, M, pos$
Output: $delayedPositions, newM$

```
    /* duplicate the delayed position */
 1: for i ← 0 to delay.d do
 2:     insert(pos[delay.dA], delay.sT, pos[delay.dA][delay.sT])
    /* remove unnecessary array elements */
 3: timeStepsToAdd ← 0
 4: for (i ← length(pos[delay.dA]) − 1; i > M; i − −) do
 5:     if pos[delay.dA][i − 1] = g then
 6:         pos[delay.dA].remove(i)
 7:     else
 8:         timeStepsToAdd ± length(pos[delay.dA]) - 1 - M
    /* add elements to arrays of other agents*/
 9: for i ← 0 to R do
10:     if i ≠ delay.dA then
11:         for y ← 0 to timeStepsToAdd do
12:             pos[i].append(pos[i][timeSteps])
13: delayedPositions ← pos
14: newM ← M + timeStepsToAdd
15: return delayedPositions, newM
```

A model situation with two agents is shown in Figure 4.1, with its optimal and collision-free plan in form of position matrix shown in Table 4.1. Let the red agent (labeled as an agent 0) be delayed by two time step units at time step 1. The path changes are shown in Table 4.1.

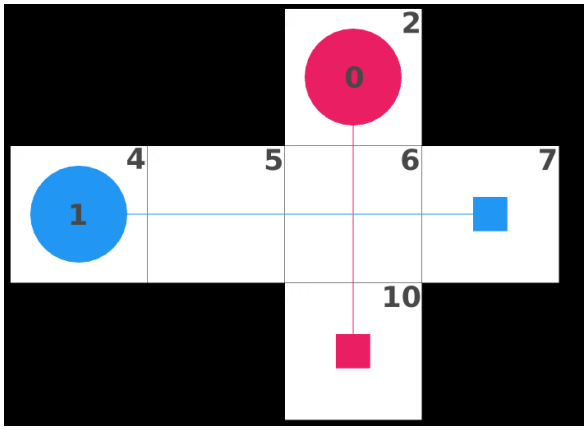


Figure 4.1: Model situation with two agents

Agent \ Time	0	1	2	3
0	2	6	10	10
1	4	5	6	7

(a) : Position matrix of original valid plan

Agent \ Time	0	1	2	3	4	5
0	2	6	6	6	10	10
1	4	5	6	7		

(b) : Delayed path of agent 0

Agent \ Time	0	1	2	3	4
0	2	6	6	6	10
1	4	5	6	7	

(c) : Reduced redundant goal positions in delayed path

Agent \ Time	0	1	2	3	4
0	2	6	6	6	10
1	4	5	6	7	7

(d) : Added goal position to other path to align path lengths

Table 4.1: Path changes while executing Algorithm 1

Introducing more delays to the plan is easily done by repeating the described algorithm with previously delayed positions and the new makespan as input. This is shown in Algorithm 2, but with a few extra steps in lines two and three. Different results may arise depending on the order of applying intended delays for the same agent. For example, let red agent on the previous model situation be delayed by one time step unit at time step 0 and also by two time step units at time step 1. A desirable result would be red agent staying at node 2 for two time steps and then staying at node 6 for three time steps. However, by first introducing the one time step delay, the positions change in such a way that the next intended two step delay at time step 0 will only prolong the first delay. This is shown in Table 4.2a. To prevent such situations, the order of delays must be in descending order according to the starting time step of delay. By reversing the order of delays, the paths will be delayed as intended (Table 4.2b).

Agent \ Time	0	1	2	3	4	5
0	2	2	2	2	6	10
1	4	5	6	7	7	7

(a) : Position matrix of delayed plan with applying delays in the ascending order according to the starting time step

Agent \ Time	0	1	2	3	4	5
0	2	2	6	6	6	10
1	4	5	6	7	7	7

(b) : Position matrix of delayed plan with applying delays in the descending order according to the starting time step

Table 4.2: Different paths are returned depending on the order of applying delays

Algorithm 2 Delay Multiple Agents

Input: $delay, R, M, pos$
Output: $delayedPositions$

```
for i ← 0 to length(arr_d) do
    sortIndices ← getSortIndices(delay.sT)
    ix ← sortIndices[i]
    pos, M ← delayAgent(delay[ix], R, M, pos)
delayedPositions ← positions
return delayedPositions
```

Chapter 5

Action Dependency Graph

5.1 The ideas and construction of the ADG

Action Dependency Graph (ADG) [HKT+19] is a graph that captures action-precedence relationships of the MAPF solution. It is constructed as a set of vertices representing the position of an agent at a particular time and a set of edges representing action dependency. The edges are distinguished as Type 1 edge and Type 2 edge. The former one represents dependencies of single agent's actions, which creates a constraint that all agent actions must be done in the exact sequence as planned. Type 2 edges represent action dependencies of two agents planning to move to the same location at different times. The dependency creates a constraint that an agent cannot move to a location unless all agents who have an earlier planned arrival time in that location finish their action.

Let R be the number of agents, i index denoting a particular agent, t time step, s agent's position before executing an action, g agent's position after executing the action, n number of time steps until a goal is reached, P MAPF plan structure, containing s, g, t and possibly other relevant variables. Algorithm 3 constructs the ADG with time complexity $O(R^2M^2)$, where M is makespan. In lines 2 to 8, the algorithm constructs Type 1 edges that link the plan's information associated with single agents' action - the origin of the edge is P_t^i and the destination is P_{t+1}^i. In lines 10 to 17, Type 2 edges are constructed if a condition that s of one some agent i is equal to g of other agent i' and the planned time step t of s^i is less or equal to the planned time

Algorithm 3 ADG creation

Input: Plan P_i for each robot.
Output: G_{ADG}

```
1:  /* create vertices and Type 1 edges */
2:  for i ← 0 to R do
3:      Add vertex P_0^i to V_ADG
4:      P ← P_0^i
5:      for t ← 1 to n^i do
6:          Add vertex P_t^i to V_ADG
7:          Add edge (p, P_t^i) to E_ADG
8:          P ← P_t^i
9:  /* create Type 2 edges */
10: for i ← 0 to R do
11:     for t ← 0 to n^i do
12:         for i' ← 0 to R do
13:             if i ≠ i' then
14:                 for t' ← 0 to n^i' do
15:                     if s_t^i = g_t'^i' and t ≤ t' then
16:                         Add edge (P_t^i, P_t'^i') to E_ADG
17:                         break
```

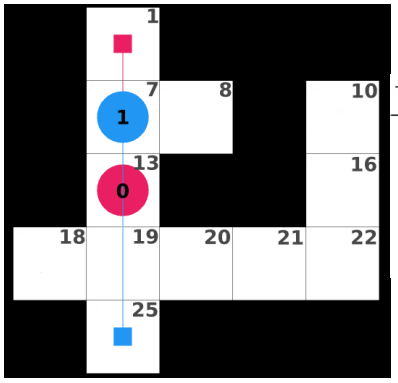
step t of $g^{i'}$ holds.

A model scenario is shown in Figure 5.1a with two agents. The optimal MAPF plan is as follows: The blue agent has to move to the right cell to allow the red agent to reach its goal destination. The next time step, the blue agent returns to its starting cell and moves down to reach its own goal destination. The actions that correspond to the described optimal plan are shown in Figure 5.1b. This plan, however, has action dependencies that can cause a collision if no safety measurements are taken. If the blue agent is delayed in the first time step for one time step unit, the red agent would move directly into the same cell as the blue agent and collide. Creating the ADG for this plan and applying it in the simulation, the action dependency of the red agent on the blue agent would be recognized, and the red agent would stall its actions until this dependency no longer exists. When an agent finishes its action, the corresponding vertex can be marked as no longer active, removing all dependencies (edges) associated with this vertex. The constructed ADG for the plan of the model scenario is shown in Figure 5.1b.

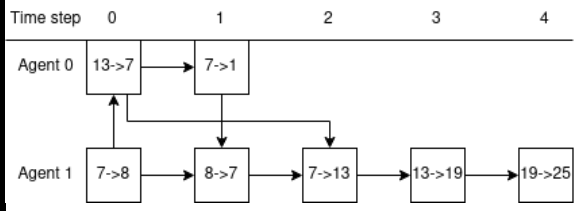
An agent cannot execute its action if there are active vertices on which the action is dependent. In practice, this means that the vertex corresponding to the agents' action is a destination of some existing edge with still active origin vertex. For example, in Figure 5.1b, at the start of the simulation, the only possible action is $7 \rightarrow 8$ of the blue agent, since it is not a destination of

any existing edge. After executing this action, all dependencies associated with this action are removed, and now 13 → 7 of the red agent is the only possible action. This procedure continues until all agents reach their goal destinations.

Although this method is robust to any delay of an agent, it comes at the expense of a possible increase of SOC or makespan. The optimal plan for the scenario in Figure 5.1b has SOC 7 and makespan 5; after simulating the ADG disregarding any delays, the SOC increased to 10, and the makespan increased to 7.



(a) : Model scenario



(b) : The optimal plan for the model scenario, including the ADG edges represented as arrows

Figure 5.1: Model scenario and visualized dependencies on the optimal plan

5.2 Plan monitoring

The ADG guarantees robustness when it is used during monitoring plan execution. When an agent is ready to execute an action, it must first be checked if any dependencies prevent the agent from carrying out the action. If there are any such dependencies, the agent waits until the dependency no longer exists. For that reason, when an agent finishes an action, it is necessary to remove all dependencies connected with that action so that other agents would not be vainly restrained in their movements. Therefore, the ADG must be continuously updated during the plan execution. The detailed algorithm of feasible plan monitoring is shown in Algorithm 4.

Algorithm 4 Plan monitoring with ADG

Input: $G_{ADG}, P, realPositions$

```
1: t_g ← 0
2: SOC ← 0
3: for i ← 0 to R do
4:     finished[i] ← false
5:     t_a[i] ← 0
6: while finished.count() ≠ R do
7:     doActions(G_ADG, t_a, P, realPositions, finished)
8:     if increaseTimestep() = true then
9:         t_g ← t_g + 1;
10:        updateADG(G_ADG, t_a, P, realPositions, finished, t_g, SOC)
11: function doActions(G_ADG, t_a, P, realPositions, finished)
12:    for i ← 0 to R do
13:        t ← t_a[i]
14:        if finished[i] = false and P_t^i = realPositions[i] then
15:            if isDependent(G_ADG, i) = false then
16:                commenceAction(i)
17:        if P_{t+1}^i = realPositions[i] then
18:            P_t^i.finished ← true
19:            stopAction(i)
20: function updateADG(G_ADG, t_a, P, realPositions, finished, t_g, SOC)
21:    for i ← 0 to R do
22:        t ← t_a[i]
23:        if P_t^i.finished = true then
24:            for all (P_t^i, v) ∈ E_ADG do
25:                E_ADG.erase(P_t^i, v)
26:            t_a[i] ← t + 1
27:        if isAgentFinished(P_t^i) = true then
28:            SOC ← t_g + SOC
29:            finished[i] ← true
```

The above algorithm may look complex, but the ideas it represents are simple. The input is an ADG graph G_{ADG}, plan structure P, and *realPositions* array, which should correspond to the real positions of agents in plan execution. The other notation used in the algorithm is described next. *finished* is a boolean array that stores information on whether an agent is in the goal position and does not have any more actions planned. R denotes the number of agents. t_g is a time step of the plan realization, independent of the individual agents. t_a is an array of each agent's time step. For example, if the agent i is delayed for 7 time step units at the start of the simulation, after 7 time steps, t_g would be 7, and $t_a[i]$ would be 0. Keeping track of individual agent's time steps is vital to knowing what actions they should take

next. The following three functions are domain-specific: `increaseTimestep` is connected to the real-time clock, and determines whether a time step should be increased; `commenceAction` commands an agent to execute its next action; `stopAction` commands an agent to stop its current action. The `isAgentFinished` function checks whether an agent is in the goal position and has no other actions planned. `isDepended` checks whether the next action of an agent is not dependent on other actions. The remaining non-described notation has the same meaning as in the previous algorithms.

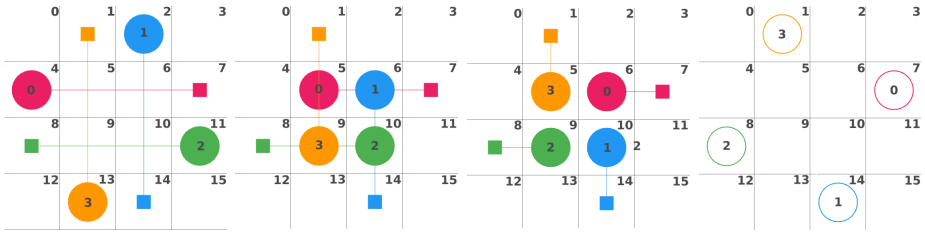
In lines 1 to 7, multiple variables and arrays are initialized. In lines 8 to 14, the while cycle is constructed with the terminating condition that all agents reached are in their goal positions. First, the function `doActions` is called, which commands agents to execute their actions. The agent commences executing its next action if the following is true:

- The agent's plan is not finished
- The agent resides at the currently planned position
- The agent is not dependent on any other agent's actions

If an agent is at the target position of action, it stops executing the action and marks the vertex associated with the stopped action as finished. This function is repeatedly called until a time step is increased. Then, the `updateADG` function is called. All associated edges in the ADG with the finished vertices are removed. It is also checked whether the agent is at the goal position and does not have any other actions planned, as it is a terminating condition for the while cycle.

5.3 Cycles in ADG and their detection

ADG is supposed to be an acyclic graph, but the ADG construction algorithm does not completely disallow the possibility of cycle creation. The following example shows the behavior of ADG when cycles are present.



(a) : timestep 0 **(b) :** timestep 1 **(c) :** timestep 2 **(d) :** timestep 3

Figure 5.2: An example of simulated MAPF plan, where four agents change their positions among themselves

Figure 5.2 shows an example of four agents changing their positions in a circular movement. This would be an expected output of MAPF planners since they do not take the possibility of delays into account. However, this situation would require precise synchronous execution to avoid collisions. In ADG, this would be detectable as a cycle. Execution with ADG containing a cycle would result in a deadlock of agents involved in the cycle. The dependencies of the described situation are shown in Figure 5.3.

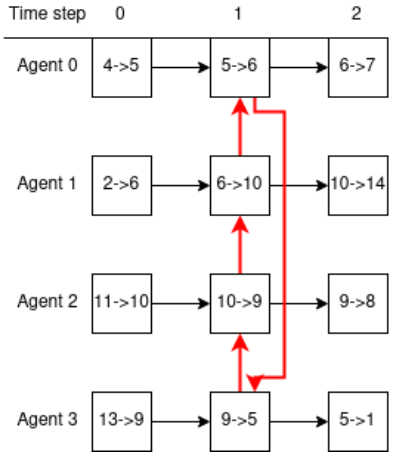


Figure 5.3: ADG dependencies of the optimal plan for the scenario on Figure 5.2a. The cycle is highlighted in red.

In the ADG of the plan in Figure 5.3, there is a cycle in nodes in time step t_1. Upon execution, this will result in a deadlock situation, where neither agent can start an action leading to a target position in time step t_2 node because the depending actions will never be finished.

In paper [HKT+19], it is proposed to prevent cycles occurring during planning, for example, by disallowing an agent to move out of a particular position in the perpendicular direction of another robot moving into that same position. If no such precaution is done beforehand, it is necessary to check whether there are any cycles in the ADG upon its creation.

Detecting cycles in a graph is usually handled by depth-first search. However, it is possible to simplify the search by knowing that a cycle can only occur across one particular time step. There are two types of edges in the ADG, and neither of them allows a vertex to be dependent on another vertex planned later. Type 1 edges are a sequence of one agent's moves, therefore a vertex can be dependent only on vertices planned with earlier time. Type 2 edges introduce the property that a vertex can be only dependent on vertices of other agents with earlier time (see Algorithm 3, line 15). A possibility of a vertex being dependent on another vertex at a later time does not exist.

The simplified search for cycles is shown in Algorithm 5. The search starts with an initial time step variable t (line 1). First, all vertices with the planned execution time t are marked as not visited (lines 2 to 3). Suppose there is such Type 2 edge that vertex v_t is an origin. In that case, vertex v'_t is a destination, and both vertices are planned to be executed in time step t, the vertices are marked as visited, and the destination vertex is passed to the recursive function (lines 5 to 8). In the recursive function, for each Type 2 edge where the origin is the passed vertex and the destination is another vertex planned to be executed in time step t, the destination vertex is checked if it has been visited. If true, the cycle is successfully detected; otherwise, the destination vertex is passed to the recursive function. Should the recursive function fail to detect a cycle, the search continues with different initial origin vertex (lines 4 to 5). Should no cycle be detected for any initial origin vertex with planned execution time t, the time step variable t is incremented by 1.

Algorithm 5 Searching for cycles in ADG

Input: G_{ADG}
Output: true if cycle is detected, otherwise false

```
1:  for t ← 0 to M do
2:      for all v_t do
3:          v_t.visited ← false
4:      for all v_t do
5:          for (v_t, v'_t) ∈ E_ADG do
6:              v_t.visited ← true
7:              v'_t.visited ← true
8:              if CycleDetectRec(G_ADG, v'_t) then
9:                  return true
10:             v_t.visited ← false
11:             v'_t.visited ← false
12: return false
13: function CycleDetectRec(G_ADG, v_t)
14:     for (v_t, v'_t) ∈ E_ADG do
15:         if v'_t.visited = true or CycleDetectRec(G_ADG, v'_t) then
16:             return true
17:     return false
```

Chapter 6

Replanning

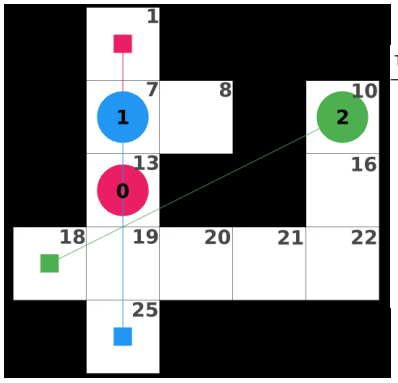
In certain situations, it is necessary to create a new plan during the execution of an already existing one. These include dynamic scenarios, where agents' goals change when reached or when an unforeseen obstacle is detected and may collide with the agent's trajectories. Furthermore, replanning may also be advantageous to create a new plan with better-expected properties. This chapter links the concept of replanning to ADG, discussing the possibilities and benefits the replanning has in ADG-controlled plan execution.

The following sections describe the situations when replanning is advantageous or necessary in further detail, present algorithms that would efficiently replan based on ADG, and show a feasible way of connecting replan algorithms with a simulator.

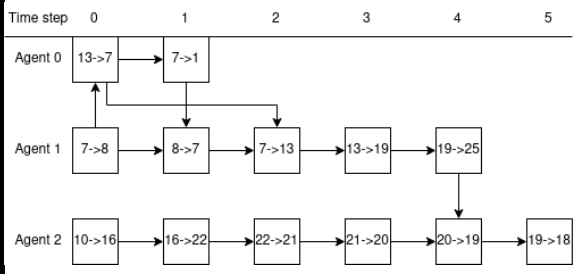
6.1 Replanning to reduce SOC

Suppose the plan realization is controlled by the ADG. An agent a_i cannot execute an action until all agents whose agent a_i is dependent on finish their actions, stalling it for an unforeseeable amount of time steps. This property of the ADG is helpful as it guarantees robustness but at the cost of an increase in SOC. When a_i is stalled, at the same time, other agents may be dependent on a_i and therefore stalled, which further increases total SOC. To put it concisely, ADG guarantees robustness to delays but also introduces additional delays in plan execution as a result.

A model scenario in Figure 6.1a is an example of the above situation. It is similar to the one discussed in the previous chapter (Figure 5.1), but this one includes an extra green agent (labeled as an agent 2). The optimal plan with ADG dependencies is shown in Figure 6.1b.



(a) : Model scenario



(b) : The optimal plan for the model scenario, including the ADG dependencies

Figure 6.1: Model scenario and visualized dependencies on the optimal plan

There is one key dependency that links the blue agent's (labeled as an agent 1) and the green agent's action in time step 4. The green agent can perform the action planned in time step 4 only when the blue agent completes its action in time step 4; however, the blue agent must also deal with dependencies connected to the red agent (labeled as an agent 0) beforehand. If the plan execution is controlled by ADG and no additional delays are considered (agents, therefore, move synchronously), the green agent would be stalled while waiting for the blue agent to finish its planned action on time step 4. The resulting paths are shown in Table 6.1b, and the paths of the optimal plan without ADG controlling are shown in Table 6.1a.

Agent \ Time	0	1	2	3	4	5	6
0	13	7	1				
1	7	8	7	13	19	25	
2	10	16	22	21	20	19	18

(a) : Paths of the optimal plan

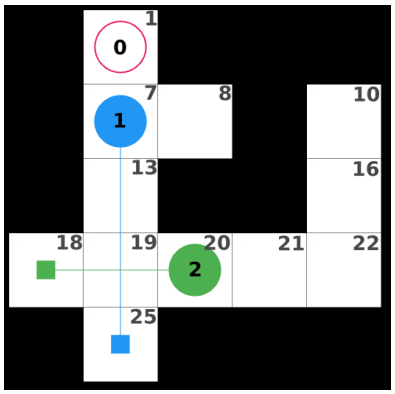
Agent \ Time	0	1	2	3	4	5	6	7	8	9
0	13	13	7	1						
1	7	8	8	8	7	13	19	25		
2	10	16	22	21	20	19	19	19	19	18

(b) : Paths after ADG-controlled simulation

Table 6.1: The differences in paths after ADG-controlled simulation and the optimal plan

The SOC of the original plan is 13, and the makespan is 6, whereas the SOC of the ADG-controlled plan is 19 and the makespan is 9. Even though the assumption of synchronous agent moving was declared (and no delay was induced as a consequence), the SOC and makespan increased considerably in the ADG simulation. The green agent attributed the most to the increase with 3 more actions than the projected number in the optimal plan; however, this was caused as a consequence of the blue agent being delayed before executing the action planned at time step 4. This shows that not only does ADG introduce delays during plan execution, but the delays also may cumulate.

Taking a closer look at the situation when the green agent was stalled, it is noticeable that the stalling could have been prevented by replanning.



(a) : Stalling of the green agent

(b) : The blue agent was supposed to cross the intersection first but it was delayed before the situation arose. Red agent is therefore stalled futilely

Figure 6.2: Situations where replanning would improve SOC

In figure 6.2a, the agents are in the middle of the execution of the plan, precisely at time step 4 (see Table 6.1b). The red agent is in the goal position, and the green agent is waiting for the blue agent to execute its actions as it is dependent on it (see Figure 6.1b). If a new plan is created now, with starting positions the agents currently occupy, it would be possible to create a new ADG.

Agent \ Time	0	1	2	3
0	1			
1	7	13	19	25
2	20	19	18	

(a) : Paths of the optimal plan after replanning

Agent \ Time	0	1	2	3	4
0	1				
1	7	13	13	19	25
2	20	19	18		

(b) : Paths after replanning in ADG-controlled simulation

Table 6.2: Different paths are returned depending on the order of applying delays

Suppose the agents immediately follow the new plan and are controlled by

the new ADG. The paths of a new optimal plan for the situation in Figure 6.2a are shown in Table 6.2a. In contrast to the plan in Table 6.1a, the green agent is planned to cross tile number 19 sooner than the blue agent. This change of priority would also mean that according to the new ADG, the blue agent would now be dependent on the green agent and not the other way around. When the simulation finishes, the overall SOC before and after replanning is 17, and makespan is 8. This is an improvement to the simulation without replanning discussed before, where SOC was 19 and makespan 9. Note that the simulation with replanning assumed that the new plan was created immediately, which is generally not the case.

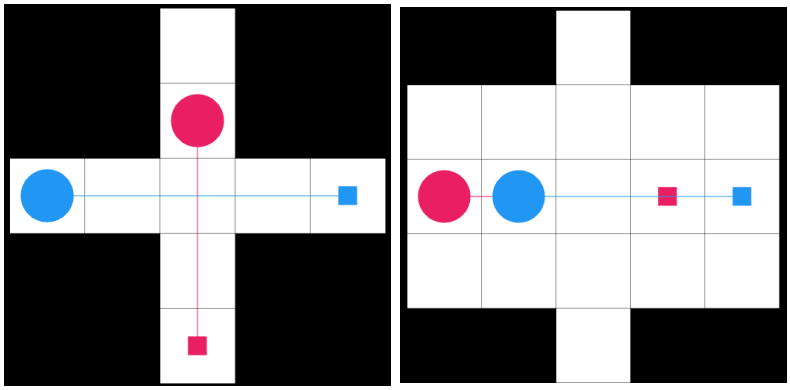
Another scenario where replanning is beneficial is shown in Figure 6.2b. In this scenario, the advantages of replanning should be much more evident. Suppose the agents are in the middle of plan execution, and according to the original plan, the blue agent would cross the intersection sooner than agent the red agent. Therefore, the ADG derived from this plan created a dependency of the red agent on the blue agent such that the red agent cannot cross the intersection until agent the blue agent does. However, since the blue agent is far from the intersection, it was likely delayed before the situation in the figure arose. The red agent must wait futilely until the blue agent manages to cross the intersection. If replanned, the new plan should consider that the red agent is closer to the intersection and let it cross the intersection first, significantly decreasing overall SOC.

All of these leads to a conclusion that replanning is beneficial in ADG-controlled plan execution; the crucial question is when it is plausible to replan. For minor delays, replanning could be meaningless as the agents may resolve the dependencies sooner than the new plan is found. Therefore, a good indicator is when a dependent agent's stalling is projected to be considerably high, such as the situation in Figure 6.2b.

6.2 Replanning to resolve agent's malfunction

In this thesis, an agent is malfunctioning if it cannot execute any more actions. This could happen by running out of batteries or stopping receiving a signal from the control program. The controlling program would either detect this immediately or at a later time because the agent would be stalled for a significant amount of time steps in one position unreasonably.

In Figure 6.3, two distinctive situations with malfunctioning agents are



(a) : Red agent is waiting for the malfunctioning blue agent, the situation can be fixed by re-planning

(b) : Red agent is waiting for the malfunctioning blue agent, the blue agent has to be perceived as an obstacle

Figure 6.3: Scenarios, where the red agent is waiting for the malfunctioning blue agent

presented. The malfunctioning agent is the blue one in both cases. In Figure 6.3a, suppose the red agent is dependent on the blue agent. The red agent would be stalled indefinitely as the blue agent can no longer execute actions that would eliminate the dependency. This problem can be solved by simply replanning. The dependency would be reversed, and the red agent would reach its goal position.

In figure 6.3b, the agents are planned to reach goals in train-like motion. Since the blue agent is malfunctioning, this cannot happen. As opposed to the previous situation, the replanning is not so simple. The MAPF solvers would perceive the train-like motion still as the best plan as it comes with the lowest SOC and makespan. A legitimate approach is to consider the blue agent to be an obstacle and replan the path only for the functioning red agent.

6.3 Commit cut

The commit cut algorithm [HKT+19] was designed to efficiently replan in dynamic scenarios, where agents' goals change when reached and the simulation finds and executes new plans continuously. Since testing persistence is not the aim of this thesis, the focus in this section will be shifted to another declared property of the algorithm - robustness to the newly appeared obstacles. It will also be discussed whether the approach is feasible to reduce the overall SOC.

Algorithm 6 Commit cut

Input: G_{ADG}
Output: commit cut $c^i \in V_{ADG}$ for $i = 0, ..., R-1$
1: $\{d^0, ..., d^{R-1}\} \leftarrow ComputeDesiredSet(G_{ADG})$
2: $G'_{ADG} \leftarrow (V_{ADG}, E'_{ADG})$ where
3: $E'_{ADG} = \{(v, v') | (v', v) \in E_{ADG}\}$
4: $reachable \leftarrow \emptyset$
5: $q \leftarrow Queue(\{d^0, ..., d^{R-1}\})$
6: **while** q *not empty* **do**
7: $\quad P_t^i \leftarrow Dequeue(q)$
8: $\quad reachable \leftarrow reachable \cup \{P_t^i\}$
9: $\quad$ **for** $(P_t^i, v) \in E'_{ADG}$ **do**
10: $\quad\quad$ **if** $v \notin reachable$ **then**
11: $\quad\quad\quad Enqueue(q, v)$
12: **for** $j \leftarrow 0$ **to** R **do**
13: $\quad c^j \leftarrow argmax_t\{P_t^i | P_t^i \in reachable \wedge i = j\}$

The purpose of Algorithm 6 is to find such a set of vertices (commit cut), one for each agent, where the positions after executing those vertices would be used as a starting position for the MAPF solver. Since there is a time lag until a new plan is found, the algorithm first finds a set of vertices (desired set) such that the expected execution time of actions is larger than the expected planning time (line 1). Next, the reversed graph of ADG is computed, where all edge's directions are reversed (lines 2 to 3). The vertices in the desired set are enqueued (line 5), a new set is created (reachable set, line 4), and an exhaustive search is executed on the reversed graph (lines 6 to 11). A vertex P_t^i is dequeued and added to the reachable set during the search. All vertices in edges, where P_t^i is the destination, are checked if they are already in the reachable set; if not, the vertices are enqueued, and the search continues. The reachable set after the search ends is called a set of committed vertices. The last actions for each agent in the set of committed vertices are added to the commit cut set (lines 12 to 13).

The starting positions are sent to a planner, and the agents execute actions in the reachable set until the new plan is created. If an agent executes the action in the commit cut set, it stops at the current position and refrains from executing any more actions from the old plan. The new plan is combined with the vertices from the reachable set that were not executed. The first vertices from the new plan are synchronized in time so that there are no dependencies from the new plan to the old plan.

According to the original paper [HKT+19], the algorithm is supposed to be robust to newly appeared obstacles; when an agent detects an obstacle, it notifies a planner of the obstacle and refrains from doing any action until

replanned. It is not clear how it would deal with situations as shown in Figure 6.3b, where, should the blue agent be stopped, the red agent cannot execute any set of actions as the blue agent blocks the way. Furthermore, the algorithm cannot be efficiently used in situations where it would be beneficial to replan to reduce overall SOC, as was discussed in Section 6.1, particularly the scenario in Figure 6.2b. In this scenario, the red agent waits on the blue agent to cross the intersection due to imposed dependency. After replanning, it would be desirable that the dependency would not exist to let the red agent cross the intersection first. However, at minimum, the commit cut algorithm commits all blue agent's actions up to the one that crosses the intersection, and the red agent would still need to wait for the blue agent before transitioning to the new plan.

6.4 Modified commit cut

The commit cut algorithm has properties that make it efficient in real-time dynamic scenarios but may not be the best option when the intention is to decrease overall SOC in plan execution. The presented Algorithm 7 is a modified version of the commit cut algorithm and it is a novel approach introduced in this thesis.

The main idea is to resolve futile stalling that may be present in ADG-controlled plan execution by setting an arbitrary number of time steps k that would correspond to the maximum number of actions the agent can execute in the old plan before transitioning to the new plan.

The realization of the idea is to go iteratively through the ADG and add to the reachable set only the vertices that can be reached in k time steps from current position. A vertex P_t^i is added to the reachable set if it does not have any Type 2 edges, where P_t^i is a destination and the origin vertex v is not in the reachable set (lines 9 to 18). The `while` cycle ends when no more vertices are added during one iteration (line 19)

The situation in Figure 6.2b can now be successfully resolved using this algorithm. Let $k = 3$, for example. The blue agent adds all three next vertices to the reachable set, as it is not dependent on the red agent. The red agent does not add any vertices, as the first vertex in its plan sequence depends on the vertex of the blue agent that has not been added to the set. During planning phase, the blue agent executes $k' \leq k$ actions. When the new plan is created, it is combined with $k - k'$ blue agent's next actions in the old plan.

Algorithm 7 Modified commit cut

Input: G_{ADG}
Output: commit cut $c^i \in V_{ADG}$ for $i = 1, ..., R$

$\{0^0, ..., t_0^{R-1}\} \leftarrow GetCurrentTimes(G_{ADG})$
$\{t_k^0, ..., t_k^{R-1}\} \leftarrow ComputeDesiredTimes(G_{ADG})$
$reachable \leftarrow \emptyset$
$finished \leftarrow false$
while $finished = false$ **do**
    $finished \leftarrow true$
    for $i \leftarrow 0$ **to** R **do**
        for $t \leftarrow t_0^i$ **to** t_k^i **do**
            $addToReachable \leftarrow true$
            for $(v, P_t^i) \in E_{ADG}$ **do**
                if $v \notin reachable$ **then**
                    $addToReachable \leftarrow false$
                    break
            if $addToReachable = false$ **then**
                break
            else
                $reachable \leftarrow P_t^i$
                $t_0^i \leftarrow t_1^i$
                $finished = false$
for $j \leftarrow 0$ **to** R **do**
    $c^j \leftarrow argmax_t\{P_t^i | P_t^i \in reachable \wedge i = j\}$

Since the red agent is closer to crossing the intersection, the optimal plan would let the red agent cross the intersection first; therefore, the red agent will not depend on the blue agent in the newly created ADG.

6.5 Incorporating replanning into simulator

This section shows a feasible way to connect replanning algorithms with a simulator. This method provides several features - first, a new plan is created independently on the simulator so that the simulation would not be halted. The agents can still execute actions during the finding of a new plan. Second, the new plan is safely combined with the actions of the old plan that were supposed to be executed before transitioning to the new plan. Third, if any delays were introduced before the simulation began for testing purposes, the new plan will still adhere to the arranged delays so the test will not be flawed.

The inter-process communication was created using the nanomsg library

[nana] with the C++ binding [nanb]. The plan visualizer and planning algorithms were implemented by Keisuke Okumura [OTD21] [map] and the plan visualizer uses the openFrameworks library [oF]. The plan visualizer shows the continuous movement of the agents while executing actions from the plan. Delaying the plan (Algorithm 1 and 2) was done in python script; all of the rest were implemented in C++. Several modifications were made to the plan visualizer for this thesis's purposes, making it a plausible simulator for testing robustness. The planner and the simulator are independent programs, which is why inter-process communication is implemented.

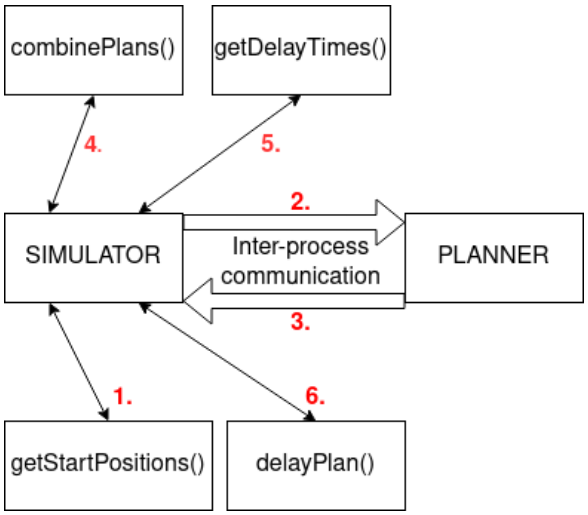


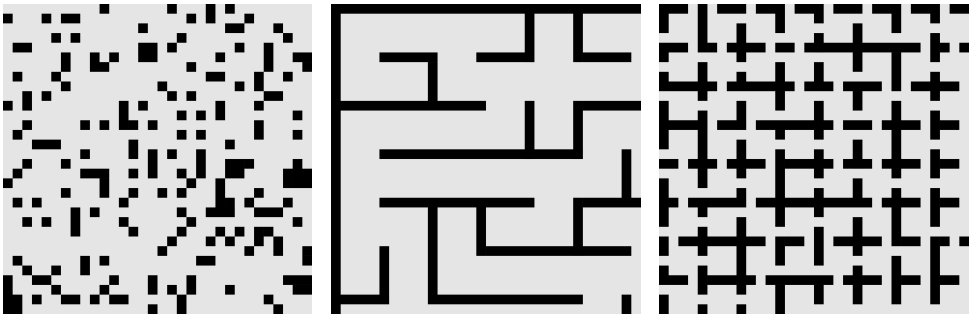
Figure 6.4: Process diagram with the designated workflow order shown in red numbers.

The workflow depicted in Figure 6.4. starts with the function `getStartPositions` , which returns computed intended start positions for creating the new plan, for example, by using the commit cut algorithm (1.). The simulator passes all necessary information to create the new plan to the planner(2.). The planner starts searching for a solution given the new start positions. When the planner find the solution, it is passed back to the simulator(3.). The actions of the old plan that were supposed to be executed before transitioning to the new plan are combined with the new plan in the `combinePlans` function (4.). At this point, the combined plan is already usable in simulation, but if there were designated delays in the old plan for testing purposes, the new plan erased these delays, and it would be sensible to put the previously designated delays back. In the `getDelayTimes` function, the scheduled delay times and duration are extracted (5.). In the `delayPlan` function, the delays are added to the plan using Algorithms 1 and 2 from Chapter 4 (5.).

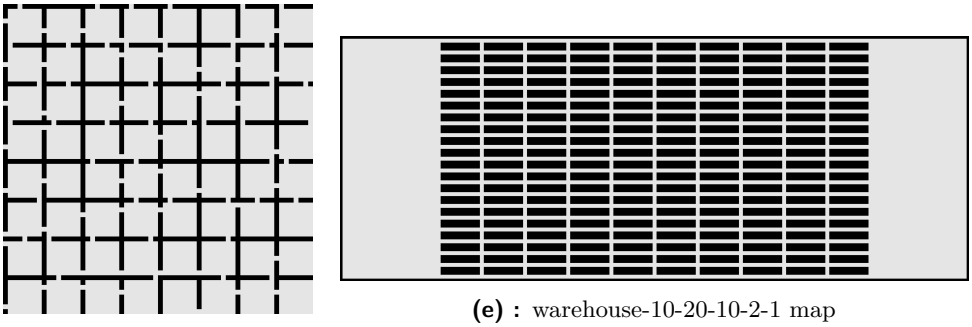
Chapter 7

Experiments

This section shows multiple experiments that verify the functionality and properties of the implemented algorithms, specifically the ADG, the commit cut, and the modified commit cut. The experiments ran on computer with CPU i5-1135G7 2.4GHz, 15 GB RAM, and Ubuntu 20.04.3 64-bit OS. The used maps are from the MovingAI dataset [SSF+19] (Figure 7.1).



(a) : random-32-32-20 map **(b) :** maze-32-32-4 map **(c) :** room-32-32-4 map



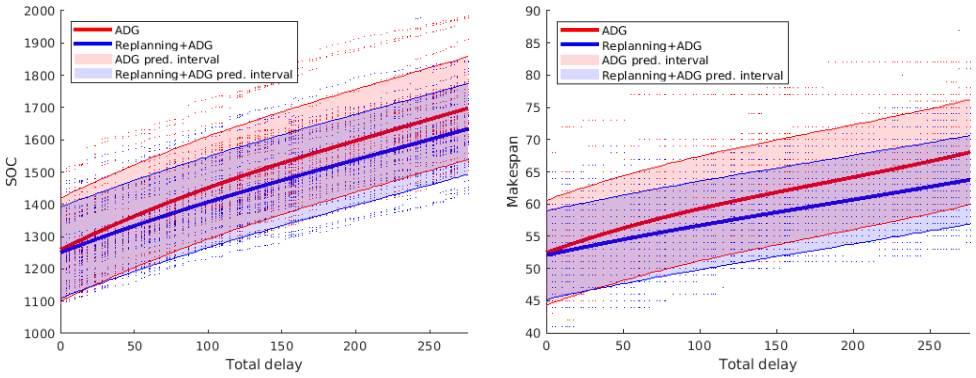
(d) : room-64-64-8 map **(e) :** warehouse-10-20-10-2-1 map

Figure 7.1: Maps used in the experiments

7.1 ADG experiments

In this type of experiment, two different approaches are compared: The ADG-controlled plan realization and ADG-controlled plan realization with a periodical replanning. The approaches are evaluated through SOC and makespan. The number of agents in each map was chosen so that the cycles do not happen often. The motivation is to test whether replanning is effective when combined with an ADG-controlled plan. As discussed in the previous chapters, using ADG can create situations where agents wait futilely. Replanning should sufficiently resolve these situations.

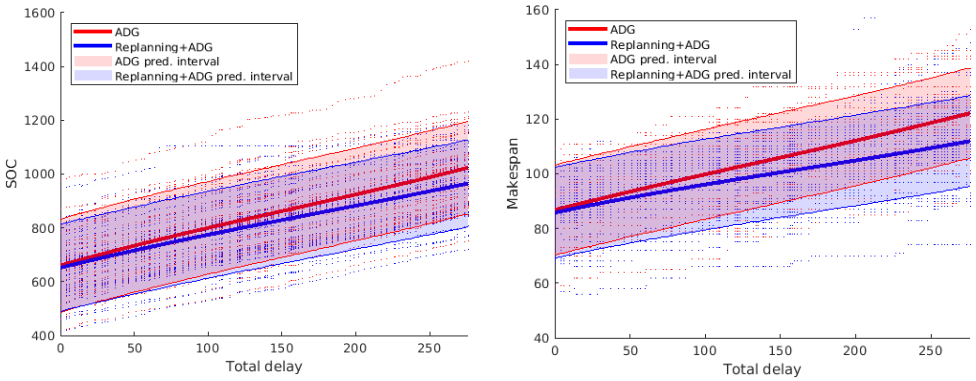
The procedure is as follows: For 5 different maps, 30 instances (different starts and goals of the agents for the same map), and 100 sets of delays, the SOC and makespan are calculated. The procedure is further explained below.
1) The delays are determined before the start of the simulation, and the delay is defined as a tuple <*agent*, *start*, *duration*>. The values of *agent*, *start* (starting time step of delay), and *duration* (in time steps) are chosen randomly with uniform distribution. For *agent*, the range is [0, number of agents]. For *start*, the range is [0, time step of path end of the chosen agent], the path end is the time step when an agent is at a goal and does not move away from it. For *duration*, the range is [1, 5]. All number generators are seeded with the same number through the experiments.
2) The number of sets of delays is 100. The first set of delays is empty, i.e., no delays are introduced to the original plan. The k-th delay set contains values of delay that are obtained in step 1) $k-1$ times.
3) The replanning is done instantly, i.e., the agents immediately transition from the old plan to the new one.
4) All of the planning were made by the MAPF solver ECBS with suboptimality factor $w = 1.1$, including the original plan
5) Should the original plan be invalid, or the creation of the path takes longer than 10 seconds, or a cycle is detected in the ADG before the start of the simulation, the experiment on the current instance is aborted, and a new instance is generated instead.
6) If a cycle is detected after replanning, the agents stick to the original plan.
7) The replanning is done periodically - specifically, every 10th time step.



(a) : Comparison of the two methods in SOC

(b) : Comparison of the two methods in makespan

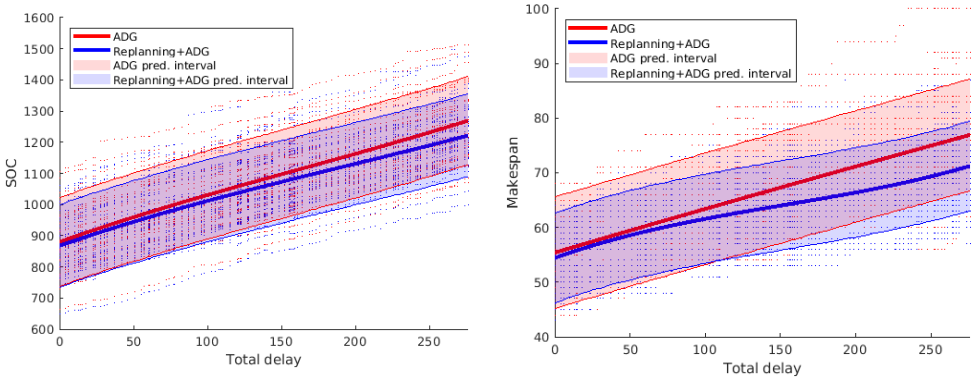
Figure 7.2: The experiments for random-32-32-20 map with 50 agents



(a) : Comparison of the two methods in SOC

(b) : Comparison of the two methods in makespan

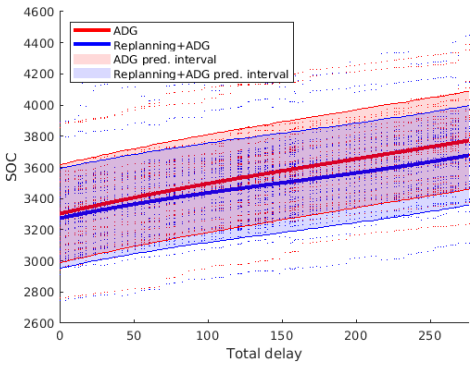
Figure 7.3: The experiments for maze-32-32-4 map with 15 agents



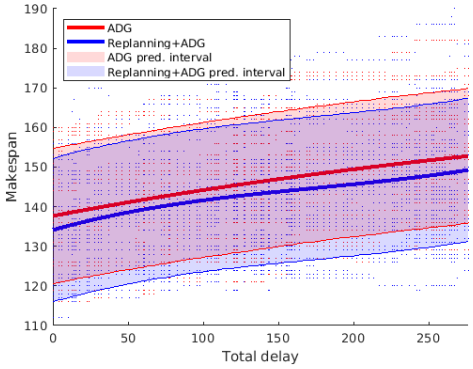
(a) : Comparison of the two methods in SOC

(b) : Comparison of the two methods in makespan

Figure 7.4: The experiments for room-32-32-4 map with 30 agents

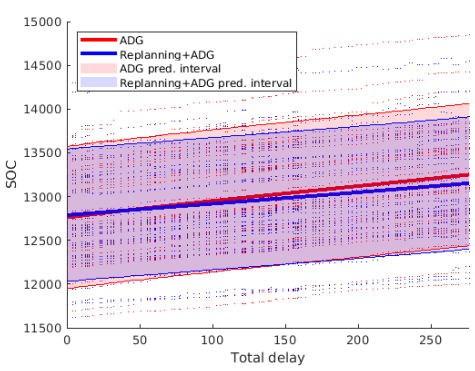


(a) : Comparison of the two methods in SOC

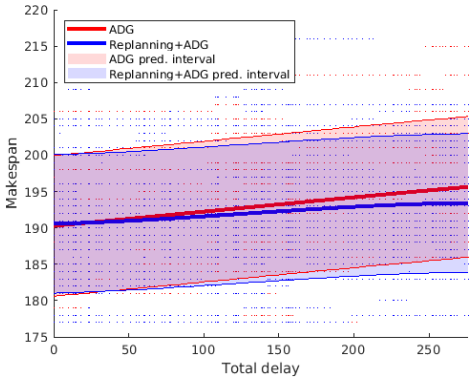


(b) : Comparison of the two methods in makespan

Figure 7.5: The experiments for room-64-64-8 map with 50 agents



(a) : Comparison of the two methods in SOC



(b) : Comparison of the two methods in makespan

Figure 7.6: The experiments for warehouse-10-20-10-2-1 map with 150 agents

In the figures 7.2-7.6, the horizontal axis is the total amount of delays in time steps and the vertical axis is a quality evaluating function - makespan or SOC. The red line fits the ADG results to a third-degree polynomial; the blue line fits the results of the ADG with a periodical replanning to a third-degree polynomial. The highlighted region represents 80% probability that the data is in that region.

Map \ Total delay	0	69	138	207	276
random-32-32-20	0.69	2.58	3.41	3.70	3.79
maze-32-32-4	1.27	2.73	3.70	4.62	5.73
room-32-32-8	1.41	1.67	2.13	2.83	3.76
room-64-64-8	0.89	1.40	2.04	2.51	2.50
warehouse-10-20-10-2	-0.22	0.13	0.37	0.56	0.74

Table 7.1: The improvement in SOC of ADG-controlled plan realization with a periodical replanning compared to the ADG-controlled plan realization. The improvement is shown in percent.

Map \ Total delay	0	67	138	207	276
random-32-32-20	0.70	3.68	4.91	5.50	6.26
maze-32-32-4	1.11	2.88	4.78	6.66	8.38
room-32-32-4	1.77	1.85	4.41	6.87	7.38
room-64-64-8	2.58	1.73	2.07	2.56	2.31
warehouse-10-20-10-2	-0.15	0.23	0.46	0.70	1.13

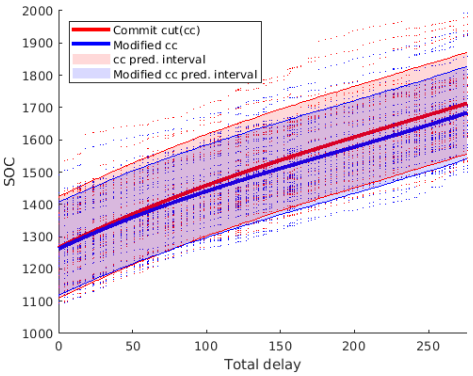
Table 7.2: The improvement in makespan of ADG-controlled plan realization with a periodical replanning compared to the ADG-controlled plan realization. The improvement is shown in percent.

The Figures 7.2-7.6 and Tables 7.1-7.2 show that the approach of ADG-controlled plan realization with a periodical replanning has proven to be more efficient in terms of SOC and makespan in all of the presented experiments. Furthermore, the effect becomes more significant with the increasing number of delays. It has been observed that the cycles in the initial plans have occurred in units of percents. It was also verified that both approaches are robust, as no collisions have happened through the experiments.

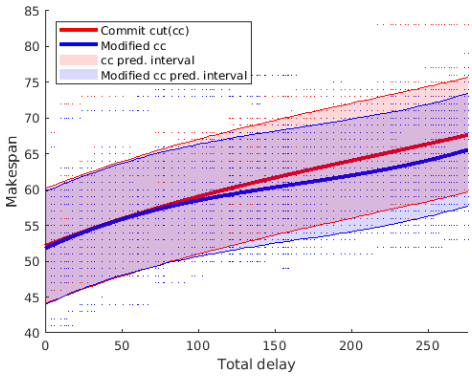
7.2 Commit cut experiments

In this type of experiment, the two compared approaches are the Commit cut algorithm and the Modified commit cut algorithm, a new method devised in this thesis. The approaches are evaluated through SOC and makespan. Both the approaches use the ADG to provide a fluid transition from the old plan to the new plan. However, the Commit cut algorithm does not repair situations where agents wait futilely for another delayed agent. The experiments do not only test Commit cut properties but also show whether the Modified commit cut can be a plausible alternative.

The procedure is as follows: The SOC and makespan are calculated for one map, 30 instances with different starts and goals of the agents, and 100 sets of delays. The procedure is further explained below.
1) The delays are generated the same way as in the ADG experiments in the previous section.
2) The replanning is always done only at the start of the 15th timestep.
3) The predicted number of time steps it takes to obtain a new plan is set to 5. The agents still execute actions according to the old plan as defined in the algorithms. 4) The actual number of time steps it takes to obtain a new plan is set to 3. This number would not be set in real-time simulations, and it would vary depending on the complexity of the map. The number is set to a fixed value to ensure that simulation of both algorithms performs under the same conditions. 5) All of the planning were made by MAPF solver ECBS with suboptimality factor $w = 1.1$, including the original plan
6) Should the original plan be invalid, or the creation of the path takes longer than 10 seconds, or a cycle is detected in the ADG, the experiment on the current instance is aborted, and a new instance is generated instead.
/



(a) : Comparison of the two methods in SOC



(b) : Comparison of the two methods in makespan

Figure 7.7: The comparison of applying the commit cut algorithm and the modified commit cut algorithm. The horizontal axis is the total amount of delays in time steps; the vertical axis is a quality evaluating function - makespan or SOC. The red line fits the commit cut results to third-degree polynomial; the blue line fits the modified commit cut results to third-degree polynomial. The highlighted region represents 80% probability that the data is in that region

Map \ Total delay	0	69	138	207	276
random-32-32-20	0.35	0.94	1.58	1.92	1.69

Table 7.3: The improvement in SOC of Modified commit cut algorithm compared to the Commit cut algorithm. The improvement is shown in percent

Map \ Total delay	0	67	138	207	276
random-32-32-20	0.66	0.32	1.87	3.32	3.08

Table 7.4: The improvement in makespan of Modified commit cut algorithm compared to the Commit cut algorithm. The improvement is shown in percent

The Figure 7.7 and Tables 7.3-7.4 show that the approach of Modified commit cut has proven to be more efficient in terms of SOC and makespan in the presented experiments. The effectiveness also becomes more significant with the increasing number of delays. The fitted curve of the Commit cut algorithm is very similar to the one of ADG in Figure 7.2a and Figure 7.2b. This was expected, as the commit cut algorithm is based on the ADG and does not fix its drawbacks. On that note, the Modified commit did not manage to improve the properties as much as the ADG with a periodical replanning approach. Nonetheless, it also offers the property of fluid transition from the new plan to the old plan, which the ADG with a periodical replanning approach does not have.

Chapter 8

Conclusion

This thesis revolves around the robustness in the plan execution. Most MAPF solvers provide a collision-free plan but with the assumption that agents move synchronously and as planned. This is an unrealistic assumption as physical robots are subjected to many defects, such as higher-order dynamic constraints or control inaccuracies. Without further control, executing plans created under mentioned assumption may result in collisions. This is the motivation for implementing a framework that guarantees robustness in the plan execution. The implemented framework in this thesis is the ADG - Action Dependency Graph, which captures the action-precedence relationship in the original MAPF plan. In the ADG-controlled plan execution, an agent cannot execute an action that would move it to a position where another agent may reside unless the agent has already moved out of the position.

Many tasks had to be completed to verify the functionality and properties of this framework experimentally. First, a tool for introducing delays was devised and implemented. Its purpose is to make the original plan invalid, which would result in possible collisions and pose a challenge for the ADG to provide robustness. The ADG had to be implemented and incorporated into the simulator. The simulator had to be modified so that it would simulate the invalid plan and monitor when the execution of the invalid plan diverges from the original plan, so the movements of all agents can be correctly adjusted according to the ADG. Completing these tasks laid a solid foundation for performing multiple experiments of the ADG. Another implemented algorithm is the commit cut algorithm. It is based on the ADG, and it searches for the starting positions of agents for replanning. It was also intended to be experimentally verified for its functionality and properties. The algorithm did not resolve one of the ADG's main drawbacks - the stalling for a long

time of an agent until the action dependencies are solved. Therefore, a new replanning algorithm was devised and implemented with the intention of reducing the overall SOC during ADG-controlled plan execution.

As the experiments show, the ADG is robust to an arbitrary number of delays but at the expense of a futile stalling. Using ADG with predefined replanning times proved to be a more efficient approach. This shows that ADG can be further optimized for better results.

The commit cut algorithm functioned correctly but did not fix the stalling the ADG creates. A new approach was devised - it is based on similar ideas to the commit cut algorithm, but it performs way more effectively in the measured evaluation functions SOC and makespan.

The experiments were possible to be carried out because a functional simulator was created beforehand. It was specifically modified for testing robustness. This simulator can be used to test the implemented algorithms further or to test other robustness-increasing algorithms if the simulator is slightly modified accordingly.

The subsequent development in the robust plan execution can be based on improving the properties of the ADG. Particularly, the ADG can be further optimized, as shown in the experiments, so the agents do not wait futilely for the delayed agent or fix cycles' occurrence. The direction that can lead to the correct solution to these problems may be by cleverly assigning priorities to agents or actions so that the dependencies may change during the plan execution.

´

Appendix A

Bibliography

[ASF+18] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou, *Robust multi-agent path finding*, SOCS, 2018.

[BSSF14] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner, *Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem*, SOCS, 2014.

[CNKS15] Michal Cáp, Peter Novák, Alexander Kleiner, and Martin Selecký, *Prioritized planning algorithms for trajectory coordination of multiple mobile robots*, IEEE Transactions on Automation Science and Engineering **12** (2015), no. 3, 835–849.

[FSS+17] Ariel Felner, Roni Stern, S. E. Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R Sturtevant, Glenn Wagner, and Pavel Surynek, *Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges*, SOCS, 2017.

[GCF04] Jean Gregoire, Michal Cáp, and Emilio Frazzoli, *Locally-optimal multi-robot navigation under delaying disturbances using homotopy constraints*, Autonomous Robots **42** (2018-04), no. 4, 895 – 907 (en).

[HKC+16] Wolfgang Hönig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig, *Multi-agent path finding with kinematic constraints*, Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'16, AAAI Press, 2016, p. 477–485.

[HKT+19] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian, *Persistent and robust execution of mapf schedules in warehouses*, IEEE Robotics and Automation Letters **4** (2019), 1125–1131.

[LB11] Ryan Luna and Kostas E. Bekris, *Push and swap: Fast cooperative path-finding with completeness guarantees*, IJCAI, 2011.

[map] *mapf-ir*, `https://github.com/Kei18/mapf-IR`, Accessed: 2022-04-21.

[nana] *nanomsg*, `https://nanomsg.org`, Accessed: 2022-03-14.

[nanb] *nanomsgxx*, `https://github.com/achille-roussel/nanomsgxx`, Accessed: 2022-03-14.

[oF] *openframeworks*, `https://openframeworks.cc`, Accessed: 2022-04-21.

[OMDT19] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura, *Priority inheritance with backtracking for iterative multi-agent path finding*, Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 535–542.

[OTD21] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago, *Iterative refinement for real-time multi-robot path planning*, CoRR **abs/2102.12331** (2021).

[Sil21] David Silver, *Cooperative pathfinding*, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment **1** (2021), no. 1, 117–122.

[SSF+19] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Barták, *Multi-agent pathfinding: Definitions, variants, and benchmarks*, CoRR **abs/1906.08291** (2019).

[SSFS21] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant, *Conflict-based search for optimal multi-agent path finding*, Proceedings of the AAAI Conference on Artificial Intelligence **26** (2021), no. 1, 563–569.

[YL13] Jingjin Yu and Steven M. LaValle, *Structure and intractability of optimal multi-robot path planning on graphs*, Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, AAAI'13, AAAI Press, 2013, p. 1443–1449.

Appendix B

Additional files

Only the essential directories are mentioned. The rest are supporting files.

File/Directory	Description
`F3-BP-2022-Weis-Josef.pdf`	The PDF file of this thesis.
`thesis`	The directory containing the Latex source files and used images for the creation of this thesis document.
`visualizer`	The directory that contains most of the work created for the purposes of this thesis.
`mapf`	The directory of planner with implemented MAPF solvers by Keisuke Okumura.