Conflict-based Plan Merging for Multi-agent Pathfinding

Master Thesis

Aurélien SIMON

Matrikel-Nr: 806567

Supervisors

Klaus Strauch

Etienne Tignon

Torsten Schaub

Potsdam University

Master Cognitive Systems: Language, Learning and Reasoning

November 21, 2023

Abstract

Multi-Agent Pathfinding (MAPF) is an artificial intelligence problem with wide-ranging applications including GPS, video games, and traffic control. MAPF involves orchestrating multiple agents to navigate from initial positions to specific destinations while avoiding collisions. In this work, we introduce a "Plan Merging" approach, a three-step approach based on path conflicts designed to tackle MAPF problems. The three steps are designated Individual Path Finding, Path Selection and Solving. We show that in certain cases, our proposed approaches can outperform classical MAPF methods. However, it also highlights the inherent limitations of our Plan Merging techniques, particularly in scenarios where obtaining a complete solution is challenging or not feasible.

Multi-Agent Pathfinding (MAPF) ist ein Problem der künstlichen Intelligenz mit weitreichenden Anwendungen wie GPS, Videospiele und Verkehrskontrolle. Bei MAPF geht es darum, mehrere Agenten so zu koordinieren, dass sie von ihren Ausgangspositionen zu bestimmten Zielen navigieren und dabei Kollisionen vermeiden. In dieser Arbeit stellen wir einen "Plan Merging"-Ansatz vor, einen dreistufigen Ansatz, der auf Pfadkonflikten basiert und zur Lösung von MAPF-Problemen entwickelt wurde. Die drei Schritte werden als Individual Path Finding, Path Selection und Solving bezeichnet. Wir zeigen, dass die von uns vorgeschlagenen Ansätze in bestimmten Fällen die klassischen MAPF-Methoden übertreffen können. Es werden jedoch auch die inhärenten Grenzen unserer Techniken zur Zusammenführung von Plänen hervorgehoben, insbesondere in Szenarien, in denen der Erhalt einer vollständigen Lösung schwierig oder nicht möglich ist.

Declaration of originality

I confirm that the master's thesis I have submitted is my own original work. I have not utilized any external sources or aids except for the ones explicitly mentioned. In cases where I have incorporated verbatim passages from other works, I have duly acknowledged the source through proper citations. This work has not been previously submitted for any course or examination, nor has it been presented to any other authority for approval.

Eigenständigkeitserklärung

Ich versichere, dass ich die eingereichte Masterarbeit selbstständig verfasst habe. Ich habe keine externen Quellen oder Hilfsmittel verwendet, außer denen, die ausdrücklich genannt sind. In den Fällen, in denen ich wörtliche Passagen aus anderen Arbeiten übernommen habe, habe ich die Quelle durch ordnungsgemäße Zitate ordnungsgemäß angegeben. Diese Arbeit wurde weder für einen Kurs oder eine Prüfung eingereicht, noch wurde sie einer anderen Behörde zur Genehmigung vorgelegt.

Signature/Unterschrift

Place/Ort, Date/Datum

Contents

List of Figures

Listings

Chapter 1

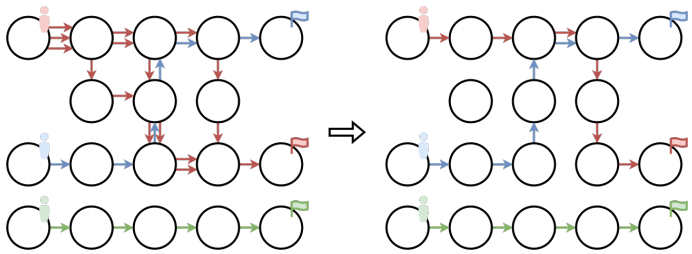
Introduction & Background

1.1 Introduction

Multi-Agent Pathfinding (MAPF) [16–18], is an artificial Intelligence problem with diverse real-world applications such as warehouse management [21], video games [13], routing, planning, and robotics [20]. In essence, MAPF involves multiple agents moving within an environment, aiming to navigate from initial positions to goal locations. The primary challenge in MAPF is to find individual paths for each agent, ensuring that they do not collide. Various techniques have been developed to tackle this problem, including search algorithms like CBS [15] (Conflict-Based Search) and reduction-based-solving methods [4].

In this Master Thesis, we focus on the "Plan Merging" approach, which aims to solve MAPF problems through a three-steps process. The first step, denoted as Individual Path Finding, involves computing paths for each agent independently, which corresponds to classic pathfinding or single-agent pathfinding [6]. The second step denoted Path Selection focuses on finding a collision-free solution using the previously computed path where selection occurs using conflict represented in two ways; potential conflict and likelihood of presence using heatmaps. The third and last step of Plan Merging is to find a solution, by 1), reducing the size of the problem using paths selected in previous step to delimitate a subgraph. Or by 2), using the conflict-free paths issued from previous step as a baseline for a MAPF algorithm. The appeal of this approach lies in the lower complexity of pathfinding compared to the overall MAPF problem [14]. The Plan Merging approach anticipates saving computation time at the cost of a possible loss of optimality. For instance, given some robots in a warehouse moving from a shelf to another, we first computes some paths for them regardless of their conflict among each other. We then eliminate paths that are "bad" through a heuristic, which can be in our case, through heatmap. A heatmap being a representation of potential conflict among paths achieved by allocating a value from 0 to 1 for each vertex. The value representing the likelihood of having a conflict. If a complete conflict-free solution cannot be found among the

remaining paths, we use the paths that are not conflicting as a preprocessing step for a classical MAPF approach. The following figure shows an example with three agents. In this example, a solution is directly found by eliminating the right paths.



The approach in its definition is inspired and close to CBS's definition, however, the planning part of CBS stops if a conflict occurs and compute another time the planning part considering the conflict previously encountered, which is not the case for the Individual Path Finding; conflicts are handled after Individual Path Finding.

Furthermore, our study shows that in certain cases, especially for large instances, our proposed approaches can outperform classical MAPF methods in terms of computation time. On the other hand, the results outline the limits of the Plan Merging techniques we have developed. In certain scenarios, obtaining a complete solution might prove challenging or not feasible with the approaches we introduced, indicating the need to recognize the inherent limitations of the Plan Merging strategies.

1.2 Background

1.2.1 MAPF

The following definitions of Multi-Agent Path Finding (MAPF) follow the ones in [10]. MAPF is a triple (V, E, A) where V, E denotes a connected graph, V being a set of vertices and E a set of edges connecting them, and A being a set of agents. For each agent $a = (s, g) \in A$, $s \in V$ denotes the starting location and $g \in V$ denoting the goal location. Every starting position and every goal position are disjoint. For each discrete time step $t \in \mathbb{N}_0$, an agent can either, wait at its current vertex or move to a neighbouring one.

The output for MAPF problems is a plan Π. A plan is a collection $(\pi_a)_{a \in A}$ of finite sequences in (V, E) where each sequence π_a is represented by a finite sequence of adjacent or identical vertices in V from s to g for agent $a = (s, g)$. We use $\pi_a(t) = v$ to denote that agent a is located at vertex v at time step t. For each $a = (s, g) \in A$, we have $\pi_a(0) = s$ and $\pi_a(|\pi_a| - 1) = g$, where $|\pi_a|$ gives the length of sequence π_a. Generally, for any $a = (s, g)$ and any $0 \leq t \leq |\pi_a| - 1$, we have $\pi_a(t) \in V$. In addition, we also have $(\pi_a(t), \pi_a(t + 1)) \in E$ with $0 \leq t < |\pi_a| - 1$

A plan is considered *valid* if, taken pair wisely, sequences are collision-free. A vertex conflict occurs whenever two different agents occupy the same vertex at the same time step. Formally, we have $conflict(a, a', t)$ if given any $a, a' \in A$ and $t \in \mathbb{N}_0$, we have $\pi_a(t) = \pi_{a'}(t)$. An edge conflict (or swapping conflict) occurs whenever two agents exchange their position or are using the same vertex at the same time. We have $conflict(a, a', t)$ if given any $a, a' \notin inA$ and $t \in \mathbb{N}_0$, we have $\pi_a(t-1) = \pi_{a'}(t)$ and $\pi_a(t) = \pi_{a'}(t-1)$.

A plan $(\pi_a)_{a \in A}$ has a conflict, if a conflict (a, a', t) occurs in $(\pi_a)_{a \in A}$ for some pair $a, a' \in A$ of agents and a time step $t \in \mathbb{N}_0$.

Sum-of-costs and *makespan* of a plan $(\pi_a)_{a \in A}$ are respectively defined as such; $\sum_{a \in A}(|\pi_a| - 1)$ and $\max_{a \in A}(|\pi_a| - 1)$.

Furthermore, we assume that graphs have Cartesian coordinate system, which means we can represent them as a grid.

1.2.2 ASP

Answer Set Programming [1] (ASP)is an approach for declarative logic programming. In a nutshell, programmer aims to describe problems instead of solving them.

A logic program P is defined by a finite set of rules. Given a set of atoms $\mathcal{A}$, a rule r is of the form

$$a_0 \leftarrow a_1, \ldots, a_n, not\ a_{n+1}, \ldots, not\ a_m$$

with $a_i \in \mathcal{A}, 0 \leq i \leq m$. An atom preceded by *not* refer to its *default negation*. We call literal an atom or its default negation.

In a rule r, a_0 represents the head of the rule and $\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ the body of the rule. We will refer respectively to them as such; $H(r)$ the head of the rule and $B(r)$ the body of the rule. We define $B(r) = B^+(r) \cup B^-(r)$ and $B(P) = \{B(r) | r \in P\}$. For each rule r in P, $B^+(r) = \{a_0 \leftarrow a_1, \ldots,\ a_n\}$, $B^-(r) = \{a_{n+1}, \ldots,\ a_m\}$ and $H(r) = a_0$. A **fact** is defined as a rule with an empty body. On the other hand, with an empty head, the rule becomes an integrity **constraint**. We define P_X as the reduct of a program P relative to a set of assigned atom X.

$$P_X = \{H(r) \leftarrow B^+(r) | r \in P, X \cap B^-(r) = \emptyset\}$$

A set X of atoms is a stable model of P if P_X is the smallest set under P_X.

For instance, lets encode the following problem. We want to pack as many object in our bag respecting the maximum authorized weight. First lets denote the possible objects.

```
1 object(trousers, 3).
2 object(tshirt, 2).
3 object(toothbrush, 1).
4 object(camera, 2).
5 object(tent, 8).
6 object(drinks, 2).
7 object(sandwiches, 2).
8 object(laptop, 3).
9 object(tablet, 2).
```

To decide which objects to pick, we utilize a choice rule. A choice rule provide a choice over a set of atoms.

```
1     {pick(N,W):object(N,W)}.
```

Here, the choice rule allows for various combinations of objects to be picked. We then enforce a constraint that ensures the total weight of the selected objects does not exceed 20. This constraint is expressed using an aggregate function within an integrity constraint:

```
1     :- #sum{W : pick(N,W)} > 20.
```

In addition, we add some normal rules and constraint to enforce *picking* some items if some other items are picked. The first two constraint implies that, if the *tent* object is picked, then it is not possible that *drinks* and *sandwiches* objects are not picked. The two last normal rules implies that, if *laptop* and *tablet* are picked, a *charger* item is added to the pool of object and is picked.

```
1     :- pick(tent), not pick(drinks).
2     :- pick(tent), not pick(sandwiches).
3
4     object(charger,1) :- pick(laptop,W1), pick(tablet,W2).
5     pick(charger,W) :- object(charger,W).
```

Lastly, we want to maximize the weight of the bag while still keeping it under 20. To achieve this, we use an optimization statement that seeks to find the maximum weight among the available combinations:

```
1     #maximize{W : pick(N,W)}.
```

This encoding will yield the optimal combination of objects to pack into the bag, maximizing the weight without having the weight above 20.

1.2.3 Solving MAPF with ASP

Instance Format

A MAPF instance is defined using the following predicates

1. *vertex*/1 which describe a vertex of the graph with a tuple $(X, Y), X, Y \in \mathbb{N}^+$ expressing the position.

2. *edge*/2 which describe an edge between two vertices. With two tuple $(X, Y), X, Y \in \mathbb{N}^+$ expressing the endpoints of the edge.
3. *agent*/1 which define agents of an instance with its identifier as parameter
4. *start*/2 which describe the starting position of an agent. With an agent identifier as first parameter and a vertex as second parameter expressing the starting position.
5. *goal*/2 which describe the goal position of an agent. With an agent identifier as first parameter and a vertex as second parameter expressing the goal position.

We introduce optional predicates used for the computation of individual paths. They are defined as such:

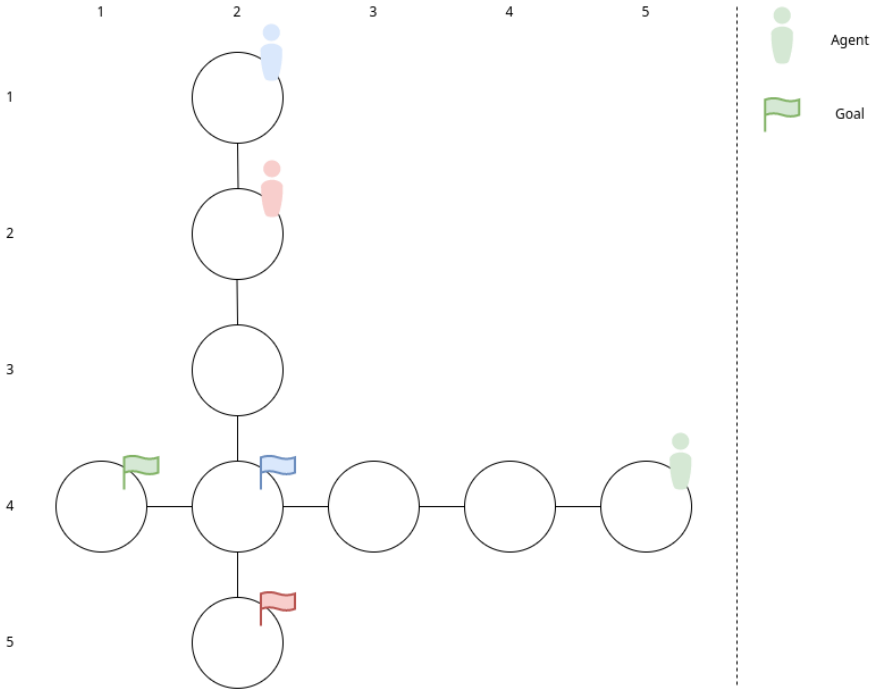
1. *shortestpath_length*/2 represents the length of the shortest path of an agent. With the identifier of an agent as first parameter and the length of a shortest path for the associated agent.
2. *makespan*/1 describes the maximum makespan of the instance. It can also be defined as a *horizon* constant

For example, the following facts describe the graph illustrated by figure 1.1.

Listing 1.1: Example of instance format

```
agent(1). start(1,(2,1)). goal(1,(2,4)).
agent(2). start(2,(2,3)). goal(2,(2,5)).
agent(3). start(3,(5,4)). goal(3,(1,4)).

vertex((2,3)). vertex((1,4)). vertex((2,4)).
vertex((3,4)). vertex((4,4)). vertex((2,1)).
vertex((5,4)). vertex((2,5)). vertex((2,2)).

edge((2,4),(1,4)). edge((3,4),(2,4)).
edge((4,4),(3,4)). edge((5,4),(4,4)).
edge((2,3),(2,2)). edge((2,4),(2,3)).
edge((2,5),(2,4)). edge((2,2),(2,1)).
edge((2,3),(2,4)). edge((2,4),(2,5)).
edge((2,1),(2,2)). edge((2,2),(2,3)).
edge((1,4),(2,4)). edge((2,4),(3,4)).
edge((3,4),(4,4)). edge((4,4),(5,4)).

% Optional
shortestpath_length(1,3).
shortestpath_length(2,2).
shortestpath_length(3,4).

makespan(4).
```

Figure 1.1: Illustrated graph of the instance format example 1.1



Encoding

As baseline for ASP MAPF solver, we adapt an encoding available in *asprilo* [7], which we call **base MAPF encoding**. In the encoding, we use two primary predicates:

1. $at(R, P, T)$ represents the position P of an agent R at time step T.
2. $move(R, U, V, T)$ represents the movement from vertex U to vertex V at time step T.

Listing 1.2: Base MAPF encoding

```
1   at(R,P,0) :- start(R,P).
2   time(1..horizon).
3
4   {move(R,U,V,T) : edge(U,V)} 1 :- agent(R), time(T).
5
6   at(R,V,T) :- move(R,_,V,T).
7   :- move(R,U,_,T), not at(R,U,T-1).
8   at(R,V,T) :-
9       at(R,V,T-1),
10      not move(R,V,_,T),
11      time(T).
12
13  :- { at(R,V,T) }!=1 , agent(R), time(T).
14
15  :- {at(R,V,T) : agent(R)} > 1, vertex(V), time(T).
16  :- move(_,U,V,T), move(_,V,U,T), U < V.
17
18  :- goal(R,V), not at(R,V,horizon).
```

In the initial section of encoding 1.2, we define, respectively on lines 1 and 2, the starting positions of each agent and their allocated time using the constant *horizon*.

The rule at line 4 describes how movements are performed. A possible interpretation could be; for each agent at each available time step and among all available edge going from U to V, we pick at most one to define one movement at each time.

The rule at line 6 states that if a move action is performed by the agent from any location (indicated by _) to a different one, the agent is positioned at this destination at this time step. Then, the rule at line 7 is a constraint that ensures consistency and coherence in the agent's locations over time. It prevents agents from moving towards a location without having been to the source location of the move at the previous time step. Furthermore, the rule at line 8 specifies that if an agent does not make a move at a specific time step, their location remains the same. Rule 13 ensures that an agent was at most one position at a time.

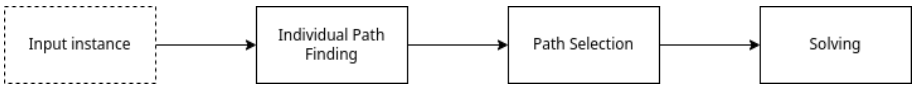
Line 15 outline vertex collision with a constraint rules; for each vertex, at most one agent can be positioned at a vertex. Secondly, the rule 16 ensure that, no movement from vertex U to V and no movement from V to U are issued at the same time. Lastly, the rules 18 forces that an agent is at their goal at the last timepoint.

The output of the encoding is the predicate $at/3$.

1.3 Overview

Figure 1.2 shows the different parts of Plan Merging: Individual Pathfinding, Path Selection and Solving.

Figure 1.2: Overview of the thesis



Plan Merging involves processing multiple paths for each agents. These paths can be computed with Individual Path Finding 2. IPF can be a constituent of Plan Merging but we can define Plan Merging without the IPF part.

The second step of Plan Merging is Path Selection 3. Path Selection has the goal of identifying paths that closely resemble valid solutions to the problem. However, it necessitates the construction of conflict-free set of path, which is computationally intensive, especially with a lot of agents and many paths. This is where we incorporate Path Elimination to reduce the number of paths by eliminating potentially "problematic" paths using a conflict-based strategy.

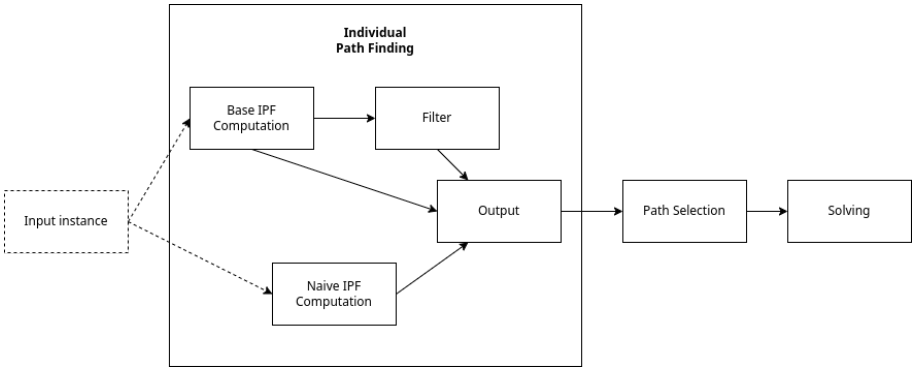
The final stage of Plan Merging involves obtaining a solution. We utilize the pre-computed paths provided by Path Selection, by using the paths in their original form, or by employing the delineation of the pre-computed paths as a subgraph in order to reduce the map size.

Chapter 2

Individual Path Finding

This chapter describes how Individual Path Finding is formalized and how computation can be achieved using ASP and Clingo.

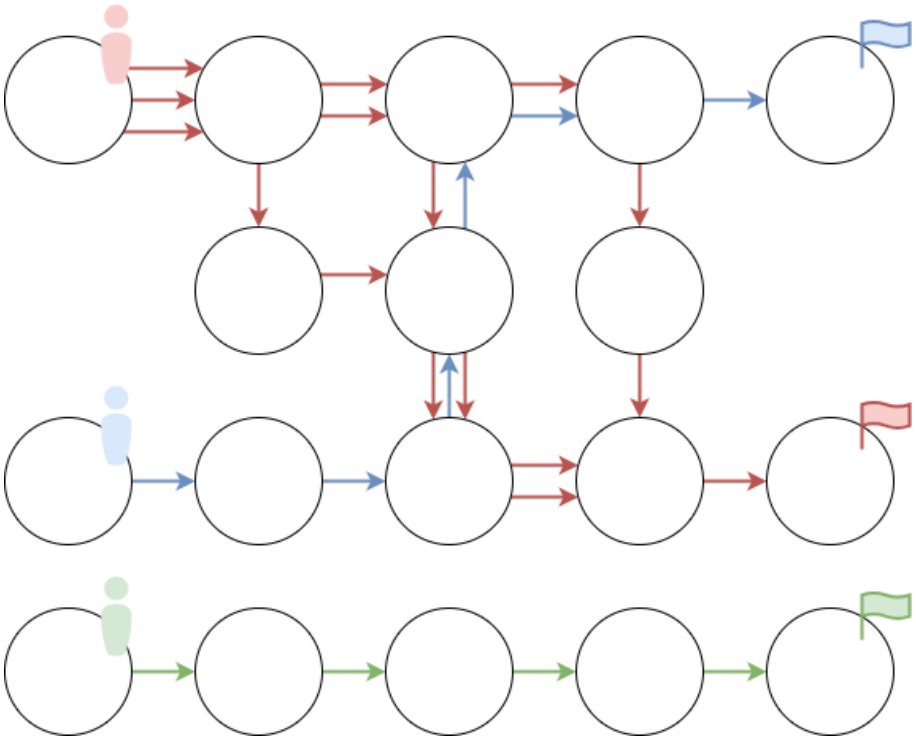
Figure 2.1: Overview of IPF



2.1 Formalization

Individual Path Finding [5, 12] (IPF) represents the computation of multiple paths for each agent without considering collision in a given MAPF problem. The idea is then to use the paths for Plan Merging. We formalize IPF as a triple (V, E, A) which is defined as in MAPF. The output is a non-empty set of set of paths τ. For each agent $a \in A$, we have $\tau[a] = \{\pi_0, \ldots, \pi_n\}$. For lighter notation, we write $\gamma_a = \tau[a]$, and also γ to refer to a set of path in general. Paths composing γ can be of different length. The following figure illustrate a $\tau = \{\gamma_r, \gamma_b, \gamma_g\}$ where $|\gamma_r| = 3$, $|\gamma_b| = 1$ and $|\gamma_g| = 1$.

Figure 2.2: Example of a $\tau = \{\gamma_r, \gamma_b, \gamma_g\}$



Paths can have different inner properties such as, length, number of bends, coverage of the graph or inter-connected properties such as number of plan conflict, diversity, distance and so on. The work achieved in the thesis has been made considering only length of paths, potential conflict, diversity and distance.

Path length is a very important property. We can splits paths into two length category; shortest paths, and non-shortest path.

2.1.1 Diversity and Distance

Diversity evaluates the variety of used vertices for a given set of paths of an agent. We introduce diversity as a sum of unique vertices visited at each time step. The function described in 2.1 computes diversity for a given γ.The formula is based on the Hamming distance [9].

$$distance(\gamma) = |\bigcup^{\pi \in \gamma} (\cup^{v,t \in \pi}(v,t))| \tag{2.1}$$

2.1: Diversity function definition

Other diversity computation can be used, such as the Jaccard coefficient [8]. The difference between Jaccard coefficient and Hamming distance is that the

first one focuses on the number of nodes that two paths have in common, while the second one focuses on the number of differences between the two paths. For our purpose, Hamming distance is more appropriate, however, it require that set are of the same length. Since paths in γ can be of different length, an adapted function was necessary. Note that the function 2.1 do not gives an index that can be compared to other γ's diversity. Furthermore, this equation is easy to implement with Clingo.

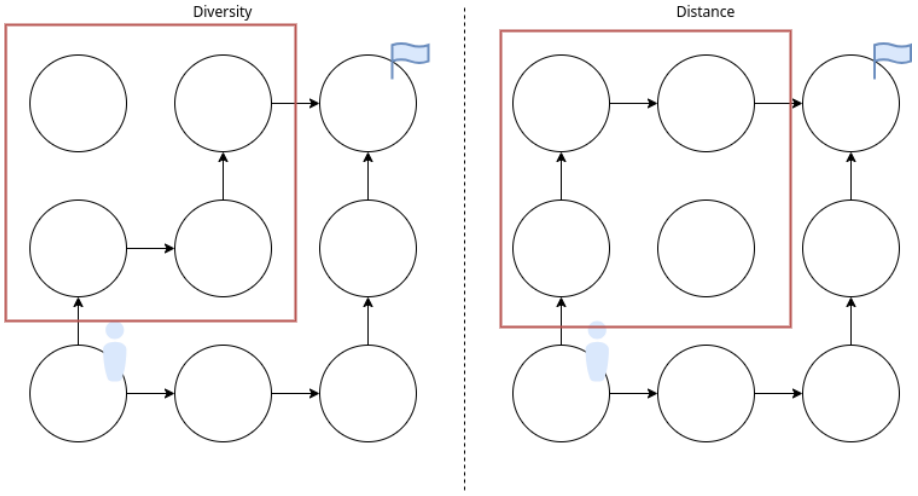
On the other hand, we can compute distance among paths in a γ. Taken pair wisely, the distance between two paths is the sum of the distance between the vertices at each time step. Note that the distance between two vertices each be an arbitrary function such as euclidean distance or the shortest path length between these two vertices. We refer to this arbitrary function as $dist(v, v')$. In our case, we consider $dist(v, v')$ as the euclidean distance between v and v'. The formalization of the distance for a given γ is defined in the following equation 2.2.

$$diversity(\gamma) = \sum_{\pi,\pi' \in \gamma, \pi \neq \pi'} \left(\sum_{t=0}^{t \to min(|\pi|,|\pi'|)} dist(\pi[t], \pi'[t]) \right) \tag{2.2}$$

2.2: Diversity function definition

Figure 2.3 shows two γ of size $|\gamma| = 2$. The first one shows an example of the most diverse γ possible for this problem (multiple solutions exist, and we consider paths in γ of same length). On the other side, we have an example of the most distant γ possible for this problem. Red squares illustrate the difference between diversity and distance.

Figure 2.3: Diversity vs Distance



2.2 Computation

In this subsection, we present two different approaches for computing multiple paths. Even though IPF constitutes a side topic for this thesis, a concern was to be able to compute x paths for each agent within a reasonable time. We implemented the following computation approaches using Clingo (ASP). Computing paths with programming languages such as C, C++, Rust, and so on might offer better speed. We opted to stick with Clingo for its straightforwardness and technology consistency throughout this work. This section may serve as a starting point for future work on computing multiple paths using ASP and Clingo.

2.2.1 Naive IPF Computation

The first approach that we describe is a direct modification of the encoding presented in the section 1.1. The copy adds a new constant to the encoding called *npaths* which denotes the number of paths to compute per agent. We modify $at/3$ and $move/4$ predicates, by adding a new argument being the identifier of the path. A unique path is now defined with an agent R and a path identifier I. With these new directives, we obtain an encoding that we call **Naive IPF** 2.1 which computes multiple paths for each agent in one call.

Listing 2.1: Naive IPF computation

```
1  agent(R,1..npaths) :- agent(R).
2
3  at(R,I,P,0) :- start(R,P), agent(R,I).
4
5  time(1..horizon).
6  {move(R,I,U,V,T) : edge(U,V)} 1 :- agent(R,I), time(T).
7
8  at(R,I,V,T) :- move(R,I,_,V,T).
9  :- move(R,I,U,_,T), not at(R,I,U,T-1).
10
11 at(R,I,V,T) :-
12     at(R,I,V,T-1),
13     not move(R,I,V,_,T),
14     time(T).
15
16 :- { at(R,I,V,T) }!=1 , agent(R,I), time(T).
17
18 :- goal(R,V), not at(R,I,V,horizon).
```

The encoding in Listing 2.1 has the exact same principles as the base MAPF encoding (see Listing 1.2). Line 1 introduces id I for paths. Note that in this case, all agents would have the same number of paths. We could use other predicates to describe how many paths are required for each agent of an instance.

Lines 6 and 8 shows predicates *move/4* and *at/3* getting changed to predicates *move/5* and *at/4* now including the ID of paths.

The main flaw that comes with the encoding described is slowness. Requesting multiple paths per agent raises the search-space considerably. Even if we remove conflict consideration in the encoding, the grounder needs to ground the rules containing predicate $move/5$ at most $|A| * npaths$ time. In addition, more variables induce a slower solving. However, this encoding gives us the opportunity to add properties to paths. For example, to implement diversity2.1, we can use an optimization statement:

```
1       #maximize{1,R,V,T : at(R,I,V,T)}.
```

More definitive additions such as constraining specific paths of specific agents to have a defined length, a defined number of bends and so on, could be used. Another flaw that comes with this encoding is that there is no guarantee that all paths for the same agent are different. However our **Naive IPF** encoding would still compute the requested number of paths for this agent a even though paths are the same. This can be fixed using collision for individual agents; the risk could be to revert to classical MAPF solving.

2.2.2 Base IPF computation

By using multi-shot solving and assumptions [11], it is possible to ground both predicates *at* and *move* once for all agents. To do so, we tweak the naive IPF encoding described in section 2.2.1 and obtain the encoding in Listing 2.2. This encoding can be interpreted as simple pathfinding that will be executed one agent at a time. Furthermore, $start/2$ and $goal/2$ predicates are not necessary; these are handled by assumptions.

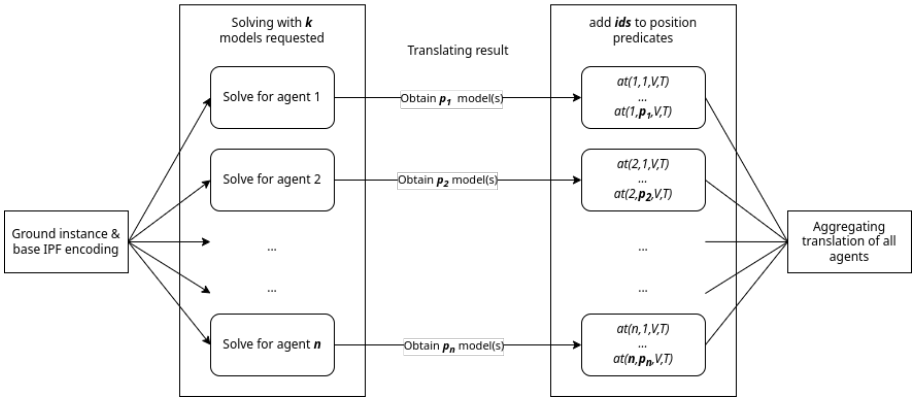
Listing 2.2: Base IPF computation

```
1       time(1..horizon).
2
3       {at(V,0) : vertex(V)} = 1.
4
5       { move(U,V,T) : edge(U,V)} 1 :- time(T).
6
7       at(V,T) :- move(_,V,T).
8       :- move(U,_,T), not at(U,T-1).
9
10      :- { at(V,T) } != 1, time(T).
11
12      at(V,T) :- at(V,T-1), not move(V,_,T), time(T).
13
14      current_agent(R) :- at(V,0), start(R,V).
15      at(GV,T) :- at(GV,T-1), time(T), goal(R,GV),
16              current_agent(R).
```

Line 14 aims to identify the current agent the solver is generating paths for. Through the assumption of the predicate $at/2$ at time step 0, we can retrieve the current agent using the corresponding $start/2$ predicate. Line 15 is then designed to guarantee that once the path of an agent reaches its goal, the agent remains at its goal location.

The flowchart described in figure 2.4 illustrates the process of base IPF computation. It first ground the instance with the encoding, then solve with the start and goal positions of an agent as assumptions. Finally, it extends *at/2* with agent ID and model number. We use the model number to identify different paths of the same agent; the model enumeration represents different possible assignations for the problem. Requesting n models means requesting n paths for an agent.

Figure 2.4: Flowchart of IPF computation using multi-shot solving and assumption



We define $1 \leq p \leq k$, if solving an instance with k requested models, the number of possible models p can be lower. We use a Python wrapping to control and manage assumptions. A pseudocode of the wrapper is denoted in Listing 2.3

Listing 2.3: Pseudo code of base IPF encoding wrapper

```
result ← list()
ground(instance + base IPF encoding)

for each agent in instance {
    assume(initial and goal position)
    solve()

    for each model {
        translate 'at/2'→'at/4'
        add translation to result
    }
}
```

A nice property that comes with this approach is that since we use model enumeration to create different paths, if Clingo enumerates fewer models than requested, we can assume that there are no other paths possible of this length.

Base IPF computing reduces computation time significantly; given the way Clingo works, decomposing the problem into one agent at a time and reusing grounding works is faster than trying to solve multiple problems at once. It does, however, compute random paths. In order to compute the shortest paths, we can use the optional predicates described in list 1.2.3. Assumptions allow us to force $goal/2$ predicate to be reached a specific time step, We can then change the goal position assumption and force it to be reached at a certain time step instead of at the horizon. We can use the same principle to compute paths of different lengths for one agent; this requires an additional solving and translation step.

Furthermore, in this process, we can compute additional paths with a modified time step associated with the goal. This addition enables agents to reach their destination xx time steps later than the originally defined duration. This strategy proves valuable when a limited number of paths are computed, as it provides the IPF solver with extra possibilities to generate additional paths.

Given the way base IPF computation works, it seems difficult to assign different properties to the paths that depends on other paths. Properties, such as, length, number of bends can easily be achieve. On the other hand, diversity and distance seems difficult. To achieve a similar result, we can request a large number of paths and create a subset out of them that satisfy chosen properties.

2.2.3 Filters

Compared to naive IPF computation, properties depending on other paths seem difficult to compute. We can however create subset of previously computed γ by filtering paths given some criteria. A filter f is defined as such; $f(\gamma) \subseteq \gamma$. We introduce two criteria functions. The first being a function selecting k most diverse paths following 2.1 and the second one a function selecting k most distant

paths following 2.2.

Listing 2.4: Encoding of diverse filter

```
1  #const npath = 5.
2
3  {dpath(R,I): at(R,I,_,_)} npath :- agent(R).
4  #maximize {1@2,R,I : dpath(R,I)}.
5
6  dpath(R,I,V,T) :- dpath(R,I), at(R,I,V,T).
7
8  #maximize {1@1R,V,T : dpath(_,_,V,T)}.
```

The diversity encoding in Listing 2.4 starts by picking, for each agent, at most *npath* paths among all path available denoted in line 3. Line 4 maximize the amount of chosen paths. Using this optimization statement, we avoid unsat results when not enough paths exist for an agent. Line 6 retrieves positions and time steps of selected paths. From these selected path, we can count the tuple (V, T) and maximize them with the optimization statement in line 8.

Listing 2.5: Encoding of distance filter

```
1  #const npath = 5.
2
3  {dpath(R,I): at(R,I,_,_)} npath :- agent(R).
4  #maximize {1@1,R,I : dpath(R,I)}.
5
6  dpath(R,I,V,T) :- dpath(R,I), at(R,I,V,T).
7
8  dist(R,I1,I2,T, (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2)) :-
9      dpath(R,I1,(X1,Y1),T),
10     dpath(R,I2,(X2,Y2),T), I1!=I2.
11
12 dsum(R,DS) :-
13     DS=#sum{D:dist(R,I1,I2,_,D), dpath(R,I1), dpath(R,I2)},
14     agent(R).
15
16 #maximize {1@2,D : dsum(R,D)}.
```

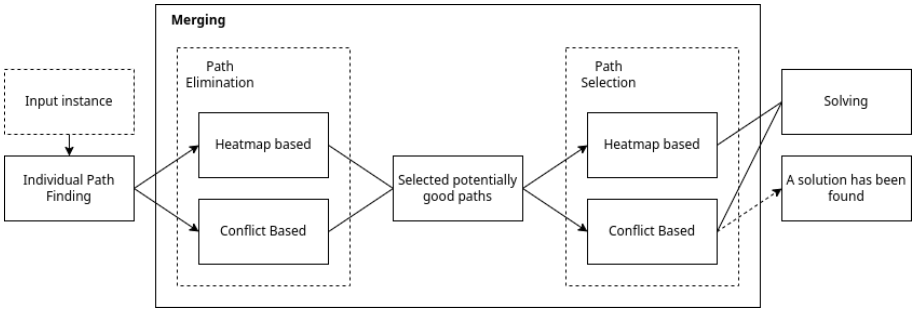
The distance encoding Listing 2.5 starts by picking for each agent, at most *npath* paths among all path available, maximizing the number of selected path and then retrieving their position and time step (from line 3 to 6). Line 8 introduces a new predicate $dist/4$ which denotes euclidean distance of two different paths of the same agent. We then have an intermediate rule line 12 which sums the distance of paths for each agents. It is then used in the optimization statement in line 16 to maximize the distance issued by D.

Chapter 3

Path Selection

In this chapter, we will describe what Path Selection (PS) is, how it works and which strategies we use to take advantage of multiple computed paths per agent. Figure 3.1 shows the different steps for Path Selection used in this work.

Figure 3.1: Overview of Path Selection



Path Selection takes a τ as its input and produces a subcomponent τ' when $\forall a \in A \mid \gamma'_a \in \tau' \subseteq \gamma_a \in \tau$. Furthermore τ' denotes a plan, where $\forall \gamma'_{a \in A' \subseteq A} \in \tau', |\gamma'_a| = 1$.

3.1 Heatmap

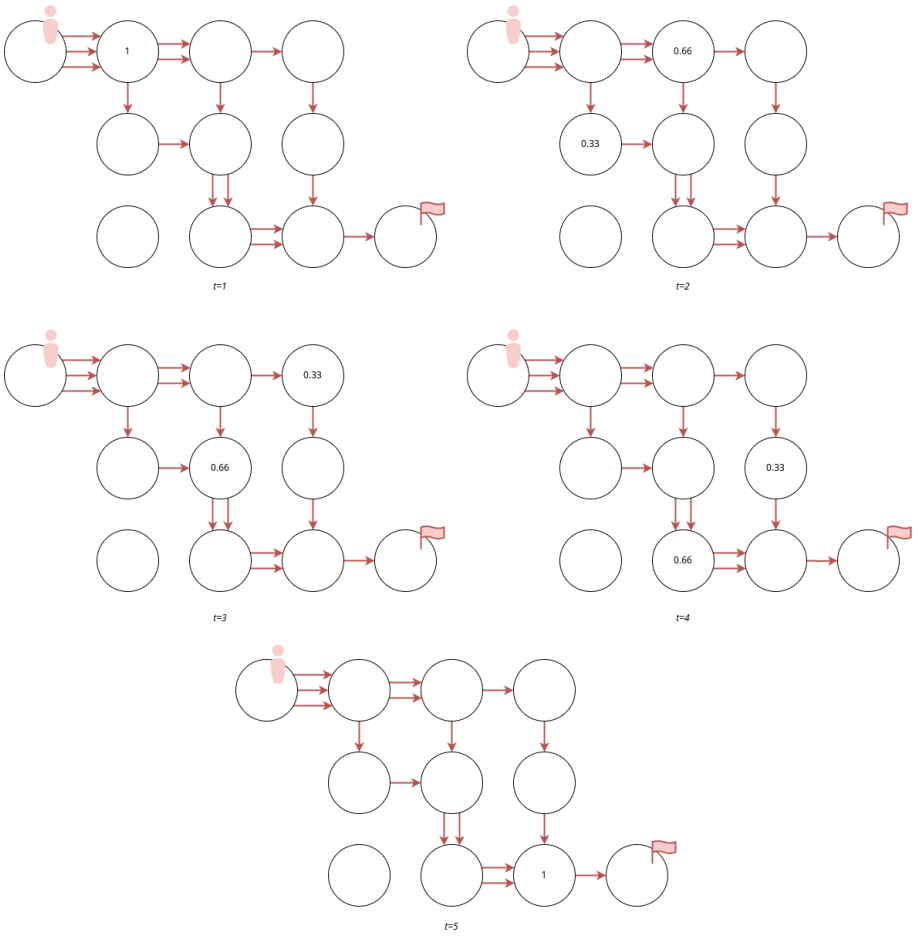
Heatmap is about projecting likelihood of presence of an agents on vertices at each step. Inspired from papers [2, 3]. Likelihood refers to, in a scope of an agent, the probability of be being at a specific vertex for each time step according to a set of path γ. This heatmap interpretation stands for the scope of one agent. We will refer to single agent heatmap as **Individual Heatmap** (IH). We formalize, in equation 3.1 an Individual Heatmap function:

$$\phi(\gamma, v, t) = \frac{|\{\pi | \pi \in \gamma, \pi(t) = v\}|}{|\gamma|} \tag{3.1}$$

3.1: Individual Heatmap

ϕ is calculated – given a γ of an agent, a vertex and a timepoint. For example, we can compute the Individual Heatmap of a given $|\gamma| = 3$ illustrated in figure 3.2.

Figure 3.2: Example of Individual Heatmap computed from a $|\gamma| = 3$



This representation produces critical vertices for an agent, where higher values correspond to a greater likelihood of vertex utilization. On the other hand, the mean heatmap value associated with γ serves as an indicator of path diversity, with elevated values signifying a low level of path diversity.

we derive a **Global Heatmap** (GH) through the aggregation of all Individ-

ual Heatmaps. It is important to note that Global Heatmaps, in this context, no longer serve as a direct indicator of the likelihood of presence but instead an indicator of "usage of vertices" .

A Global Heatmap Φ of τ is formalized in the following equation 3.2. As for Individual Heatmaps, a Global Heatmap value of Φ is computed given a vertex v and a time step t.

$$\Phi(\tau, v, t) = \frac{\sum_{\gamma \in \tau} \phi(\gamma, v, t)}{|\tau|} \tag{3.2}$$

3.2: Global Heatmap

To perform the computation of Individual Heatmaps and Global Heatmaps using Clingo, it is important to consider that Clingo inherently rounds down floating-point numbers. To obtain accurate heatmaps values in our context, we require an external program capable of performing divisions without rounding down the result. We introduce the encoding of the Individual Heatmap in listing3.1.

Listing 3.1: Individual Heatmap encoding

```
1      individual_heatmap(R,V,T,(K,N)) :-
2          K = #count{1,I : at(R,I,V,T)},
3          N = #count{1,I : at(R,I,_,_)},
4          at(R,_,V,T).
```

This short encoding introduces *heatmap*/4 predicates, with R, V and T being respectively the agent ID, a vertex and a time point. The last argument of the predicate is a tuple (K, N) where K, computed on line 3, is the number of paths of R that go through V at time step T. And N, computed on line 3, denotes the number of paths that belong to the agent K.

The calculation of the Global Heatmap is performed using **Python**[1].

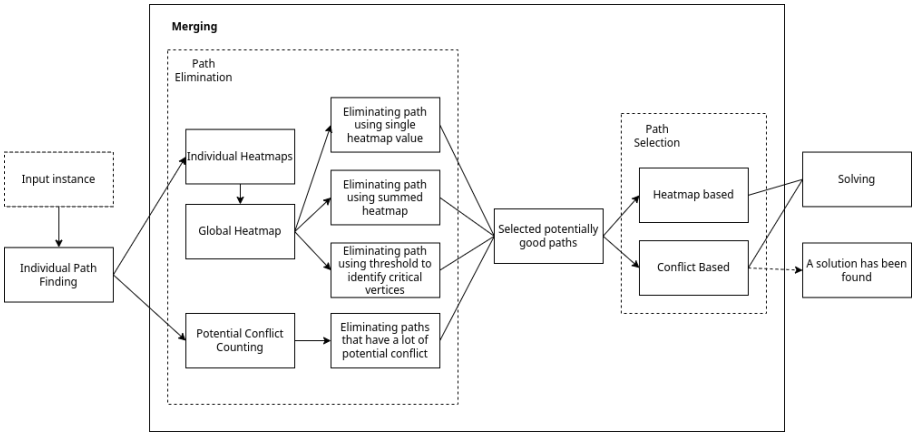
3.2 Path Elimination

Path Elimination (PE) is the process of the removing paths that exhibit potential issues. It serves to chunk through the multitude of possible paths and eliminate those that are deemed "potentially problematic". This step serves as a preprocessing stage for Path Selection (PS).

Figure 3.3 sums up the different processes in order to eliminate potentially-conflicting-paths.

[1]It is possible to provide the nominator and the denominator for the Global Heatmap at each vertex and each time step, however, compared to using a third-party programming language, it is very slow or not computationally feasible in practice.

Figure 3.3: Overview of Merging: Path Elimination



We first introduce some definition.

Definition 1. A path is classified as **killed** when it has been explicitly removed of the output of Path Elimination.

Definition 2. A path is classified as **selected** when it has been explicitly selected for the output of Path Elimination.

Definition 3. An agent is classified as **killed** when all of its paths have been explicitly killed during the Path Elimination process.

Definition 4. A vertex can be classified as **critical** at a certain time step by a Path Elimination process. It assume that there is high chance of collision.

Definition 5. An agent is classified as **critical** when at least one of the vertices composing its plan is considered as **critical** in each of its paths.

Definition 6. A **potential conflict** denotes conflict among paths of different γ. It do not represent an actual conflict.

3.2.1 Using Global Heatmap

Global Heatmap values often provide insights into vertices that are prone to conflicts. This information can be leveraged for path elimination.

A simplistic approach involves minimizing the cumulative Global Heatmap value by allowing the encoding select or not paths for the computation of GH. Unfortunately, due to the interconnections among heatmaps, practical grounding of this approach becomes feasible only on small instances.

We then present three approaches for identifying potentially conflicting paths in a reasonable computation time.

Eliminating paths using unique heatmap value

One straightforward approach involves ordering all possible assignments of (v, t) within a given τ based on their global heatmap values. We classify the top k vertices as "critical".Which can then be used as reference points for path elimination based on various criteria. We can identify which paths pass through critical vertices using the following rule:

```
1     critical_path(R,I,V,T) :-
2         critical_vertex(V,T),
3         at(R,I,V,T),
4         not path_killed(R,I).
```

We then determine n critical paths that should be eliminated. For instance, we may opt to eliminate half of them as follows:

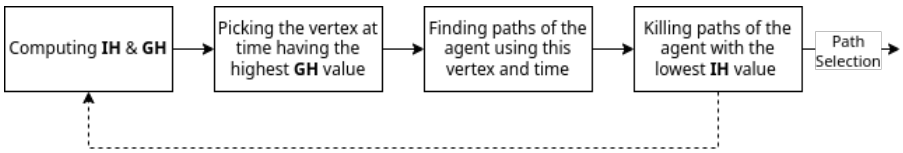
```
1     {to_kill(R,I): critical_path(R,I,V,T) } = K/2 :-
2         K = #count{1,R',I' critical_path(R',I',V,T)},
3         critical_vertex(V,T).
```

Note that we use predicate$to_kill/2$ and $path\_killed/2$ that refer to the same concept. We do differentiate them so the paths that are yet to be eliminated ($to\_kill/2$) do not interfere with already killed paths ($path\_killed/2$).

There are various parameters and variations to consider in determining the value that n can take. These may include individual heatmap values, whether an agent has already had some of its paths eliminated

The approach we settle on is the following;

Figure 3.4: Eliminating paths using unique heatmap value



As illustrated in figure 3.4, it is possible to loop the process to eliminate more paths.

As mentioned earlier, the lack of floating-point number handling of Clingo forces us to use a Python wrapper to sort Global and Individual Heatmap values.

The decision to eliminate paths of agents with the lowest Individual Heatmap values can be justified on two grounds:

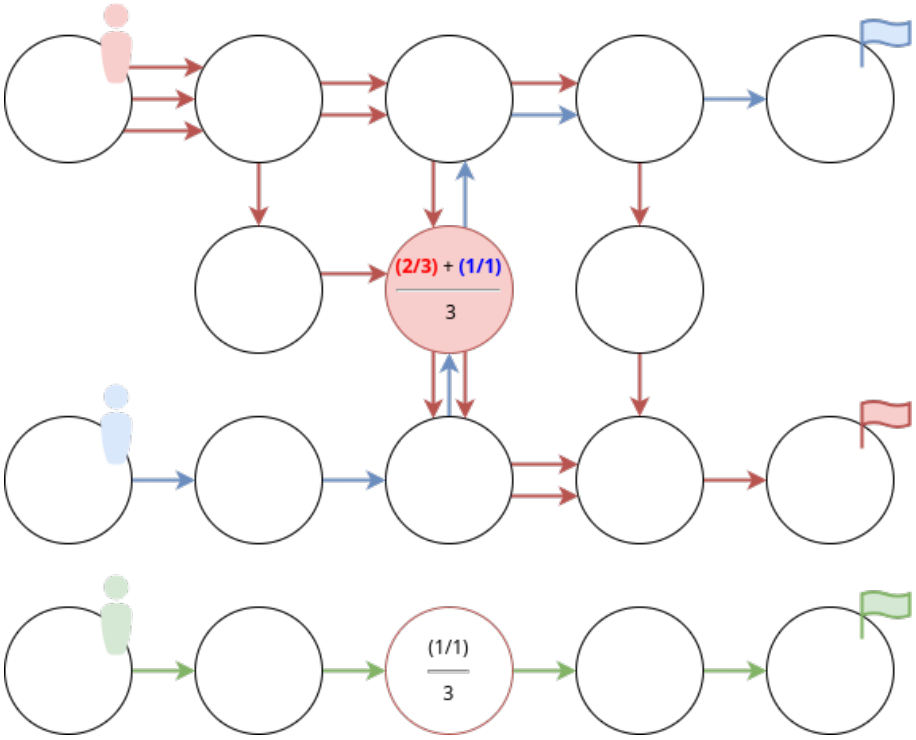
1. **Node Importance**: Higher Individual Heatmap values signify that a particular vertex holds more significance at this time step for the respective agent. Consequently, it is more suitable to eliminate paths of agents with lower interest in this vertex, as they are less likely to utilize it.

2. **Balanced Elimination**: Opting to eliminate paths of agents with low Individual Heatmap values helps maintain a balanced elimination process;

selecting agent with the lowest heatmap means killing the least possible paths involved in the **critical vertex**. It allows us to not kill agents too swiftly.

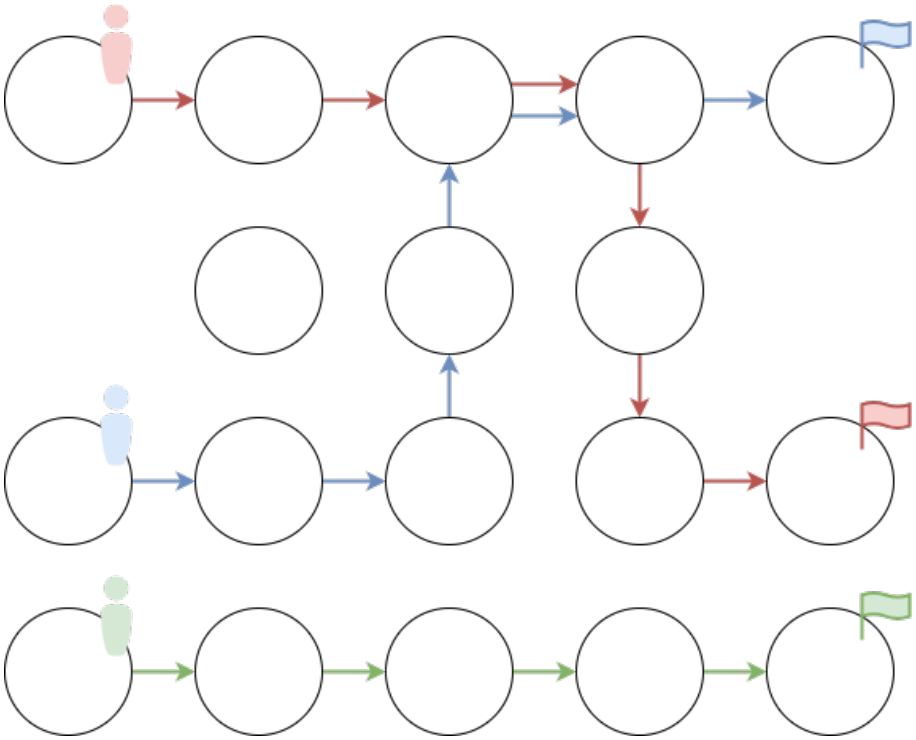
If we apply the previous approach explained above, on the example illustrated on figure 2.2. We highlighted global heatmap in red at the critical moment (time step $t = 3$), see figure 3.5.

Figure 3.5: Eliminating paths using unique Global Heatmap value process example



Between the two highlighted vertices, according to the elimination process 3.4, the vertex with the highest Global Heatmap value is the one colored in red. From this vertex, we can denote three paths going through it at time step $t = 3$. We then order the Individual Heatmap values of the two concerned agents, blue and red. from these values we can deduce that the two paths of red agents have to be killed. We obtain figure 3.5 which is in this case a valid plan:

Figure 3.6: Result of eliminating paths using unique heatmap value process



Nevertheless, despite our efforts to implement strategies aimed at balancing the elimination process among agents, adopting a vertex-oriented approach tends to kill agents swiftly.

Eliminating paths using GH threshold

One flaw that comes with the previous approach is that the process is doing one vertex after an other. It is difficult to define a value that eliminate the right amount of paths to be relevant. We propose a way to identify multiple critical vertices in one call.

The principle is simple, if the Global Heatmap value of a vertex at a specific time point is higher than a defined value $\mathcal{H}$, the vertex is denoted as critical. We call this defined value a **threshold**. Formally, we have;

$$critical\_vertices(\tau) = \{(v,t) | \Phi(\tau, v, t) > \mathcal{H}, (v,t) \in \mathcal{F}(\tau)\} \tag{3.3}$$

3.3: Identifying critical vertices using threshold

Where $\mathcal{F}$ is a function enumerating all possible tuple v, t in τ:

$$\mathcal{F}(\tau) = \bigcup_{\gamma \in \mathcal{T}} \bigcup_{\pi \in \gamma} \cup_{t=0}^{t \to |\pi|} (\pi[t], t)$$

Threshold values can be defined through different equations, considering different properties. Equation 3.4, denoted as $\mathcal{H}_{sbt}(A, \Delta)$, defines a simple biased threshold. In essence, without considering the bias, this equation implies that a vertex is labeled critical if, for all n agents with k paths each, $\frac{1}{n}$ of their paths utilize this vertex at a specific time plus one additional path. The bias adjustment allows for the flexible inclusion of more or less critical vertices.

$$\mathcal{H}_{sbt}(A, \Delta) = \frac{1 * \Delta}{|A|} \tag{3.4}$$

3.4: Simple (biased) threshold

The simplicity of the basic threshold equation makes it a general tool, but it lacks vertex-specific context. To address this limitation, we introduce a more sophisticated approach with a context-dependent threshold in equation 3.5. This threshold calculation takes into account the number of paths using a particular vertex at a specific time step (T) and the number of agents involved, providing a more precise evaluation of critical vertices. The equation incorporates a bias factor Δ that is used for finetuning.

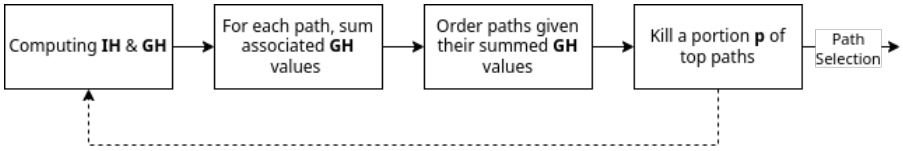
$$\mathcal{H}_{bcdt}(V, T, A, \Delta) = \frac{npath\_on(V, T) * \Delta}{nagent\_on(V, T) * |A|} \tag{3.5}$$

3.5: (Biased) Context dependent threshold

Eliminating paths using sums of heatmap values

In contrast to a vertex-oriented approach, which may label a path as "critical" even if it possesses critical attributes only on a single occurrence. An alternative strategy involves considering the scope of entire paths that may highlight conflict. The approach entails assigning a value to an entire path by summing the Global Heatmap values associated with its vertices. The approach is outlined in figure 3.7.

Figure 3.7: Eliminating paths using summed Global Heatmap values process overview



This approach tends to kill agents quite quickly when only a few paths are involved. However, it tends to be more efficient the more diverse $\gamma \in \tau$ are. Futhermore, with one call, this approach can chunk a significant portion of the paths.

3.2.2 Using conflicts

We introduce a fourth approach that centers around **potential conflicts** to guide Path Elimination. In this method, for each agent and at every vertex along their path, we calculate the count of **potential conflicts**. This approach stands apart from the heatmap-based methods in a significant way: when an agent possesses a substantial number of paths, their individual paths do not significantly influence the Global Heatmap values, even if they have the potential to cause conflicts. However, in the Potential Conflict-based approach, every path is treated with equal weight.

The encoding for the described approach is defined in Listing 3.2.

Listing 3.2: Conflict-based Path Elimination

```
1   #const n = 5.
2
3   to_determine(R,I) :- at(R,I,_,_), not path_killed(R,I).
4
5   potential_conflict(R,I,V,T) :-
6       to_determine(R,I), at(R,I,V,T),
7       to_determine(R',I'), at(R',I',V,T),
8       R'!=R.
9
10  path_composition(R,I,C) :-
11      C = #count{1,V,T : potential_conflict(R,I,V,T)},
12      to_determine(R,I).
13
14  {to_kill(R,I,C): path_composition(R,I,C)} = n.
15
16  #maximize {C:to_kill(_,_,C)}.
```

In order to decide which paths have to be killed, we first need to identify which paths can be killed. Line 3 labels paths with the predicate *to_determine*/2. **Potential conflicts** are determined through the rule at line 5. This rule identifies situations where two or more agents could potentially conflict at a given vertex and time. Rule at line 10 calculates the number of **potential conflicts** composing it. Finally, lines 14 and 16 state that exactly n paths should be marked for elimination and aims to select the set of paths to eliminate that results in the highest **potential conflicts** sums. Note that n has been set arbitrarily to 5, but it can take any value.

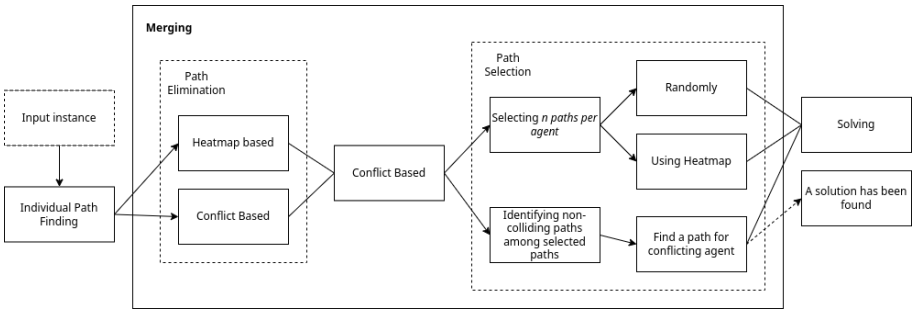
3.3 Path Selection

As mentioned above, Path Selection tends to build a $\tau' \leq \tau$. We can describe two objectives for τ'.

1). Building τ' in order to create a plan. We can identify two kinds of plans; a valid plan Π as defined earlier in the background section 1.2, and a partial plan $\hat{\Pi}$. A partial plan possesses the same attributes as a valid plan, but only for a subset of the agents. This means that for each $\gamma' \in \tau', |\gamma'| \leq 1$.

2). Building τ' in order to create a subgraph. We create a subgraph V', E' build out of the paths in τ'. We have $V' = \{v \mid v \in vertices(\tau')\}$ and $E' = \{e \mid e \in edges(\tau')\}$ where $vertices(\tau)$ and $vertices(\tau)$ respectively enumerate the vertices and edges of a given set of set of path τ. This means that each $\gamma' \in \tau', |\gamma'| \geq 1$.

Figure 3.8: Overview of Merging: Path Selection



3.3.1 Towards a (partial) plan

In this section, we explain the method for achieving the first objective outlined above. To summarize, the aim is to identify a partial plan $\hat{\Pi}$ with the maximum amount of agents. The corresponding encoding for this approach is provided in Listing 3.3.

Listing 3.3: Building a conflict free τ'

```
1   % Defining collision & possible_path
2   possible_path(R,I):- at(R,I,_,_), not path_killed(R,I).
3
4   % Constructing a (partial) plan
5   {selected_path(R,I) : possible_path(R,I) }1 :- agent(R).
6
7   collision((R1,P1),(R2,P2),T) :-
8       R1 != R2, at(R1,P1,V,T), at(R2,P2,V,T),
9       selected_path(R1,P1), selected_path(R2,P2).
10
11  collision((R1,P1),(R2,P2),T) :-
12      R1 != R2, at(R1,P1,V1,T), at(R1,P1,V2,T+1),
          at(R2,P2,V2,T),
13      at(R2,P2,V1,T+1), selected_path(R1,P1),
          selected_path(R2,P2).
14
15  :- collision((R1,P1),(R2,P2),_),
16      selected_path(R1,P1),
17      selected_path(R2,P2).
18
19  selected_agent(R) :- selected_path(R,_).
20
21  #maximize {1@1,R : selected_path(R,_)}.
```

Up to this step of Path Selection, paths could have been labeled as either **selected** or **killed**. For the remaining paths, line 2 labels them as *possible_path*/2. Then, line 5 selects paths from the candidates, ensuring that at most one path per agent is chosen for the output. Following this, lines 7 and 11 define vertex and edge collisions, respectively, among the selected paths. The associated constraint specified in line 15 ensures that the set of **selected paths** does not contain any collisions. Lastly, the optimization statement in line 21 aims to maximize the number of **selected paths**, ensuring the selection of as many paths as possible.

3.3.2 Towards a subgraph

The second objective discussed in 3.3 can defined as a pre-processing for a MAPF algorithm. Contrary to the first objective, we require at least one path for each agent. To do so, multiple approaches can be used. By definition, the output of IPF can be used directly. We will, however, describe different approaches to select some paths per agent.

The approaches that we describe are based on the output of the encoding described in Listing 3.3. The objective is to populate the set of conflict-free paths with paths of agents involved in conflicts. We define two primary approaches:

Utilizing Global Heatmaps: The first approach involves using GH and similar approaches used in Path Elimination process. This can be done either by employing the Heatmaps computed during the Path Elimination process or by recalculating them using the possible paths issued from 3.3 (or from IPF, if all paths have been eliminated)..
Path Conflict Composition: The second approach relies on path conflict composition, as outlined in Listing 3.2. Similarly to the heatmap approach, the path conflict composition can be derived either from the Path Elimination process or can be recomputed based on the possible paths obtained through path selection.

We then convert the different paths into the new vertices and edges using the following encoding 3.4.

Listing 3.4: Converting path to subgraph

```
1     nvertex(V) :- selected_path(R,I), at(R,I,V,_).
2     nedge(U,V) :-
3         edge(U,V),
4         nvertex(U),
5         nvertex(V).
```

3.3.3 Evaluating approaches

We introduced different Path Elimination approaches coupled with Path Selection, in order to evaluate them, we introduce four metrics. These metrics require a reference to compare Path Selection efficiency. We then introduce a "brute-force" Path Selection approach that computes what could be defined as the best output possible for Path Selection[2]. This approach is not used in practice because of its huge computation time.

[2]In our case, the best output possible is a partial plan $\hat{\Pi}$ as close as possible to a complete valid plan Π

Listing 3.5: "Brute-force" PS approach

```
% Defining collision & possible_path
collision((R1,P1),(R2,P2),T) :-
    R1 != R2,
    at(R1,P1,V,T),
    at(R2,P2,V,T).

collision((R1,P1),(R2,P2),T) :-
    R1 != R2,
    at(R1,P1,V1,T),
    at(R1,P1,V2,T+1),
    at(R2,P2,V2,T),
    at(R2,P2,V1,T+1).

possible_path(R,I):- at(R,I,_,_).

% Constructing set of non conflicting paths
{usable_path(R,I) : possible_path(R,I) } :- agent(R).

:- collision((R1,P1),(R2,P2),_),
    usable_path(R1,P1),
    usable_path(R2,P2).

:- selected_path(R,I), not usable_path(R,I).

% Constructing a partial plan
{selected_path(R,I) : possible_path(R,I) }1 :- agent(R).

:- collision((R1,P1),(R2,P2),_),
    selected_path(R1,P1),
    selected_path(R2,P2).

#maximize {1@1,R : selected_path(R,I)}.
#maximize {1@2,R,I : usable_path(R,I)}.
```

The major reason for the long computation time required by the "brute-force" PS approach is the identification of vertex and edge conflicts.

We introduce metrics to assess and compare the outcomes of various Path Selection approaches. We will compare the results of approaches to the brut-force output. To do so, we introduce two functions $brutforce$ and $approach$ which both take as argument a predicate name and return the associated set of facts. For instance, $approach(selected\_path/2)$ returns all the selected paths of the approach in a tuple form.

$$relevance = \frac{|approach(selected\_agent/1) \cap brutforce(selected\_agent/1)|}{|approach(selected\_agent/1) \cup brutforce(selected\_agent/1)|} \tag{3.6}$$

3.6: Relevance

Relevance computes the proportion of common selected agent designated by the approach with brutforce output.

$$absolute\_relevance = \frac{|approach(selected\_agent/1)|}{|approach(agent/1)|} \tag{3.7}$$

3.7: Absolute Relevance

Absolute Relevance evaluates the proportion of conflict-free agents found by the approach.

$$precision = \frac{|approach(selected\_path/2) \cap brutforce(usable\_path/2)|}{|approach(selected\_path/2) \cup brutforce(usable\_path/2)|} \tag{3.8}$$

3.8: Precision

Precision evaluate capacity of the approach to select path that are present in the biggest set of conflict-free paths possible.

$$chunk_proportion = \frac{|approach(path\_killed/2)|}{|brutforce(possible\_path/2)|} \tag{3.9}$$

3.9: Chunk Proportion

Chunk Proportion evaluate the proportion of path killed compared to the number of paths.

Chapter 4

(Partial) Solving

In this chapter, we will introduce the two solving approaches that we explored. Solving can be performed using already computed-paths or a subgraph defined by paths.

Solving in both cases is basically performed using the encoding defined in Listing 4.1.

Listing 4.1: Encoding of final solver

```
1   time(1..horizon).
2
3   at(R,P,0) :- start(R,P).
4
5   { move(R,U,V,T) : nedge(U,V)} 1 :- agent(R), time(T).
6
7   at(R,V,T) :- move(R,_,V,T).
8           :- move(R,U,_,T), not at(R,U,T-1).
9
10  at(R,V,T) :-
11      at(R,V,T-1),
12      not move(R,V,_,T),
13      time(T).
14
15  :- {at(R,V,T)}!=1, agent(R), time(T).
16
17  :- { at(R,V,T) : agent(R) } > 1, nvertex(V), time(T).
18  :- move(_,U,V,T), move(_,V,U,T), U < V.
19
20  goal_reached(R) :- at(R,V,horizon), goal(R,V).
21
22  #maximize{1,R : goal_reached(R)}.
```

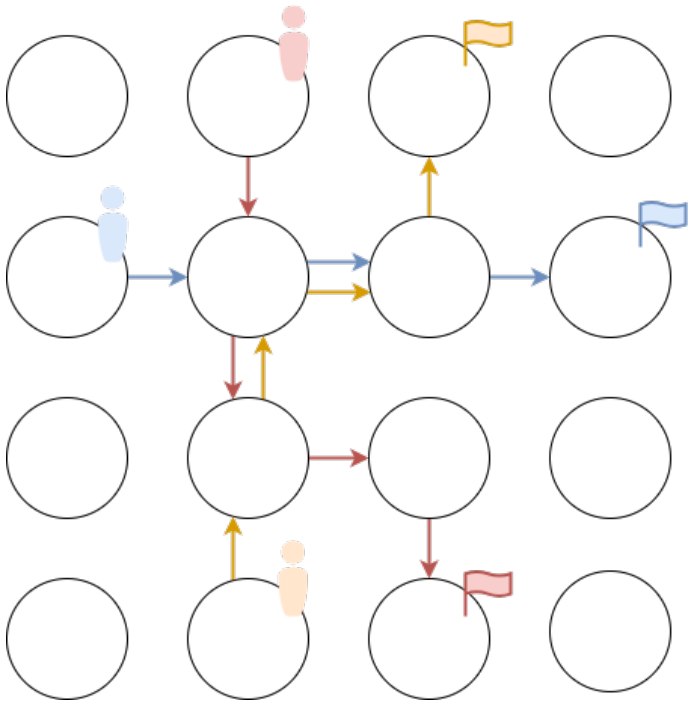
This encoding is derived from the MAPF encoding illustrated in Listing 1.2

with two significant modifications. The first modification involves the redefinition of movement predicates. Instead of relying on $vertex/1$ and $edges/2$, movement is now defined on $nvertex/1$ and $nedge/2$, which are generated from the encoding detailed in Listing 3.4.

The second distinction is highlighted in lines 20 and 22. Unlike classical MAPF, which either provides a complete solution or returns unsatisfiable if no solution exists, the solver in this context can produce a partial solution. This means that due to the steps taken to reduce the complexity MAPF problem, there is a possibility that the solver might not find a complete solution the way we defined it. However, it is possible to extend the algorithm to guarantee the solution by relaxing selected paths – in the worse case, we relax all selected paths. Which correspond to classical MAPF solving.

In order to highlight the differences between the two different kinds of solving approaches, we introduce an example in figure 4.1; a τ result of IPF of a MAPF problem $\mathcal{P}$.

Figure 4.1: Example for solving approaches. Having a $|\tau| = 3$ as result of IPF



4.1 Pre-computed paths

As mentioned in the Path Selection section 3, the output can vary based on the specified objective. In the context of the primary objective, which aims to construct a (partial) plan, we utilize the set of **selected paths** to generate

pre-computed paths. This translation is achieved through the following rules:

```
1       at(R,V,T) :- selected_path(R,I), at(R,I,V,T).
```

Through line 1, we ensure that the selected paths are incorporated into the solution. Conversely, for agents without a selected path, the encoding is employed to compute their paths.

Figure 4.2: Possible output for Path Selection & Pre computed paths

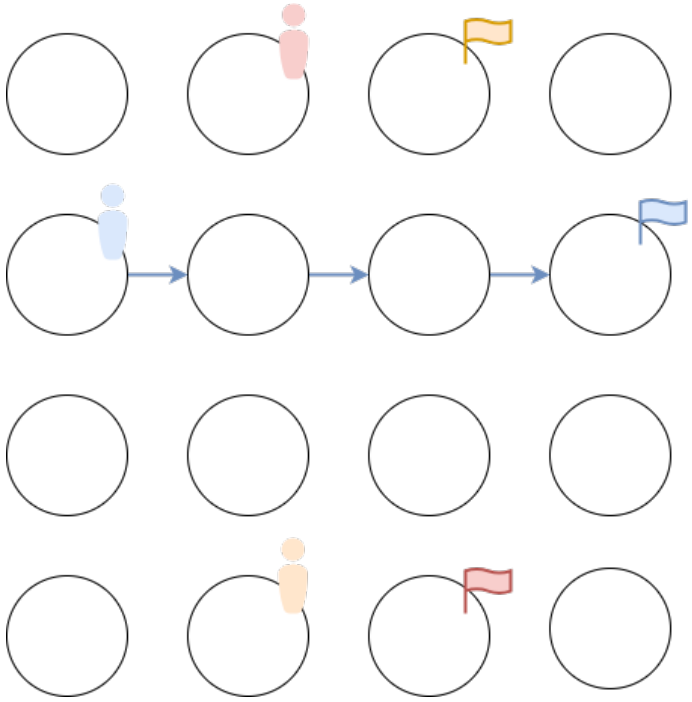
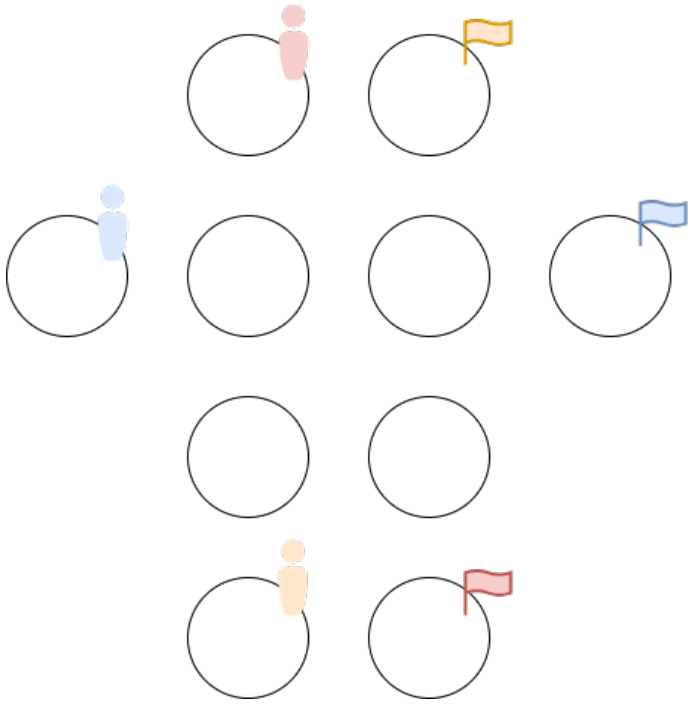


Figure 4.2 describes a partial plan $\hat{\Pi}$ where only blue agent has a path. Partial Solving now computes path for the two remaining agents, which is, hopefully easier than computing paths for the all three. In the example outlined in figure 4.2, the solving requires a makespan of five.

4.2 Subgraph

The second objective described in Section 3 aims to create a subgraph in order to reduce the size of the problem. From the paths in Figure 4.1, we obtain the subgraph shown in Figure 4.3.

Figure 4.3: Example for solving approaches. Having a $|\tau| = 3$ as result of IPF



Contrary to pre-computing path, we aim to reduce the size of the graph the agents can move on. Applying MAPF on the problems described in figure 4.3 can find a solution with a makespan of 4.

The two approaches presented can be used as one.

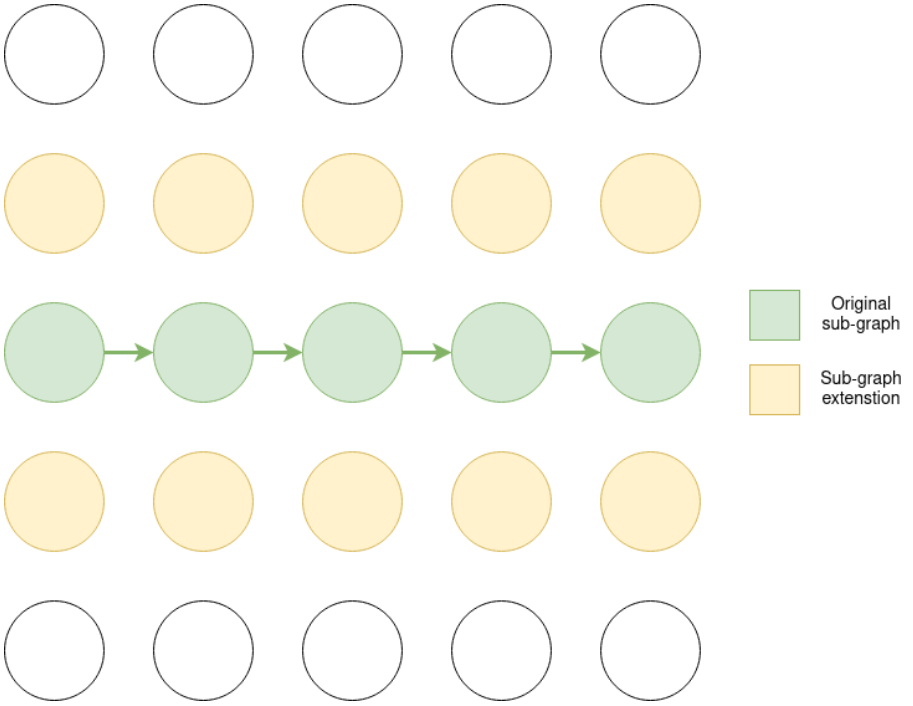
4.2.1 Subgraphs Extension Strategies

In practice, the subgraph solving approach seems to not be enough to fully solve instances; the agents seem to require too many additional time steps in order to "solve conflict". Thus, we introduce two strategies in order to extend subgraphs.

Corridor

The corridor strategy [19] is engineered to augment the subgraph-solving process by incorporating neighboring vertices and their associated edges directly into the sub-graph. Corridors can vary in size, allowing for different levels of expansion. Figure 4.4 illustrates a corridor of size one.

Figure 4.4: Example of corridor



The following encoding in Listing 4.2 describe how corridors for paths can be computed.

Listing 4.2: Corridor extension encoding

```
1   #const corridor_level = 2.
2
3   corridor(V,0) :- selected_path_for_corridor(R,I), at(R,I,V,_).
4
5   corridor(V,K+1) :-
6       K < corridor_level,
7       corridor(U,K),
8       edge(U,V).
9
10  nvertex(V) :- corridor(V,_).
```

Predicate $selected\_path\_for\_corridor/2$ highlights a path that requires a corridor extension.

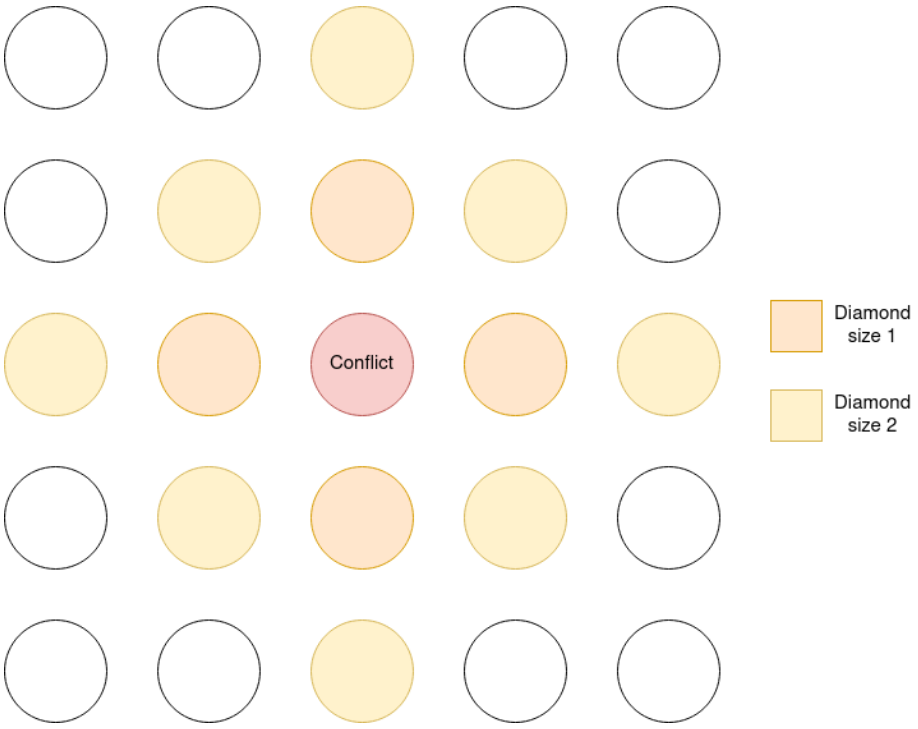
With a sufficiently large k, the entire graph can be covered; using a large corridor could essentially revert the problem back to a classical MAPF scenario.

In practice, corridors are created only for paths involved in conflicts, and a choice can be made to create corridors for only one of the two agents involved in a conflict.

Diamond

The diamond extension strategy involves expanding the sub-graph by incorporating diamond-shaped arrangements of vertices around conflicting vertices. As illustrated in figure 4.5, various levels of diamond extension can be applied, each increasing the size of the sub-graph and thereby expanding the possibilities for conflict resolution.

Figure 4.5: Example of diamond of size 1 and 2



The encoding in Listing 4.2 describes how diamonds are computed. As for corridors, the predicate $selected\_vertex\_for\_diamond/1$ highlight a vertex that requires a diamond extension. In practice, we apply diamond extension on every conflict induced by the set of paths composing the subgraph.

Listing 4.3: Diamond extension encoding

```
1  #const diamond_level = 2.
2  diamond(V,0) :- selected_vertex_for_diamond(V).
3
4  diamond(V,S+1) :-
5      diamond(U,S),
6      S<diamond_level,
7      edge(U,V).
8
9  nvertex(V) :- diamond(V,_).
```

Chapter 5

Benchmarks & Conclusion

5.1 Benchmarks

Benchmark evaluates different approaches against the base MAPF encoding introduced in background 1.2.1 section. A witness approach is also evaluated. It consists of one random path per agent converted to a subgraph. Then applying the partial solver 4.1. It aims to define a referential for the benchmark. All used approaches in the benchmark are described in the the following appendix tables 5.5.

At first, we introduce in Table 5.5 the nomenclature used to name the approaches:

Table 5.1: Nomenclature of approaches names

Step Name	Description	Identifier
IPF	No Additional path computed	**N**
	Additional Path computed	**A**
Path Elimination	Simple threshold	**St**
	Summed Heatmap	**Sh**
Solving strategy (can be both)	Pre-computed path	**Pc**
	Subgraph	**Sg**
Subgraph strategy	Corridor	**C**
	Diamond	**D**

Secondly, the benchmark has been made on a 8 cores Intel Core i7-4470 @ 3.60ghz with 16GiB of RAM. Benchmark uses variables assignation:

- 15 paths requested (in total 30 paths if additional paths are requested)
- 1 paths retrieved for killed agent on subgraph approach
- A bias of 1 for rhe simple threshold
- 70% of the highest summed heatmap paths are killed
- At most 5 minutes are allowed for each instances
- Corridors extension of size one
- Diamonds extension of size two

5.1.1 Global result

Table **??** registers the following information; the identifier of the approach. The number of instance where it found a (partial) solution (timeout is considered as unsat et vice versa). The last property outlined shows the average time required for the solver in order to compute a solution with a modified horizon (0 means that the horizon is equal to the makespan). Note that additional horizon is computed for Base MAPF for comparison purpose.

Table 5.2: Approaches: focus on average computation time

Approach	# SAT	Average time (in sec)				
		Horizon	0	1	3	5
NStPc	41		86.3	110.8	136.7	164.3
NStSgC	44		14.7	29.4	38.0	47.0
AShSg	44		16.7	29.9	38.9	48.0
AStPc	39		128.2	152.6	178.7	206.0
MAPF	44		25.4	34.1	52.6	72.3
NStSg	44		14.5	29.1	37.8	46.6
AStSgCD	37		153.7	161.2	169.4	177.8
NShSg	44		14.3	28.5	37.4	46.6
NShSgPc	37		152.0	159.3	167.1	175.4
AStSgPc	37		153.4	160.9	169.1	177.5
AStSg	43		30.9	39.5	48.5	58.0
Witness	44		9.0	17.4	26.0	34.1
NShSgPcCD	40		92.7	100.4	108.7	117.6
NStSgD	44		14.6	29.5	38.1	47.1
NShPc	41		101.9	114.0	141.3	170.1
NStSgPc	38		131.5	138.6	146.2	154.5
AShSgCD	44		17.2	30.5	39.6	48.9
NStSgCD	44		14.7	29.6	38.3	47.2

Table **??** shows the disparity of efficiency among Plan Merging approaches.

Particularly noteworthy is the **NShSg** approach, which stands out as the fastest among the tested methods. This outcome is unexpected given that, based on the results, computing additional paths appears to be slower, as showed by the comparatively longer computation time observed in approaches like **AStSg**, **AStSgPc**, and **AStPc**. It appears that precomputed paths approaches often times out or, more likely, produces unsat results. This can be attributed to way the paths are forced, described in our methodology, inevitably induces collision. Figure 5.1 illustrates a scenario where conflict(s) are inevitable using the pre-computed paths approach. In all cases, the witness approach seems to be a faster approach while providing for each instances a solution.

Figure 5.1: Inevitable conflict example for pre-computed path strategy

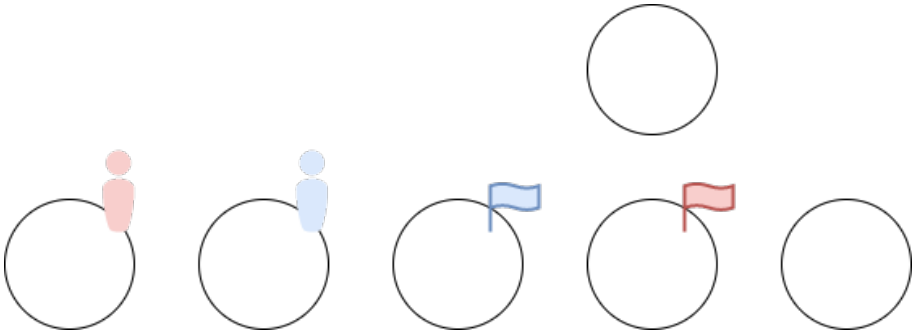


Table 5.3 we focus solving proportion. "# Fully solved instances" column refers to scenarios where all agents obtain a path. This table also presents the expected proportion of paths found given a modified horizon.

Table 5.3: Approaches: focus on solving path proportion

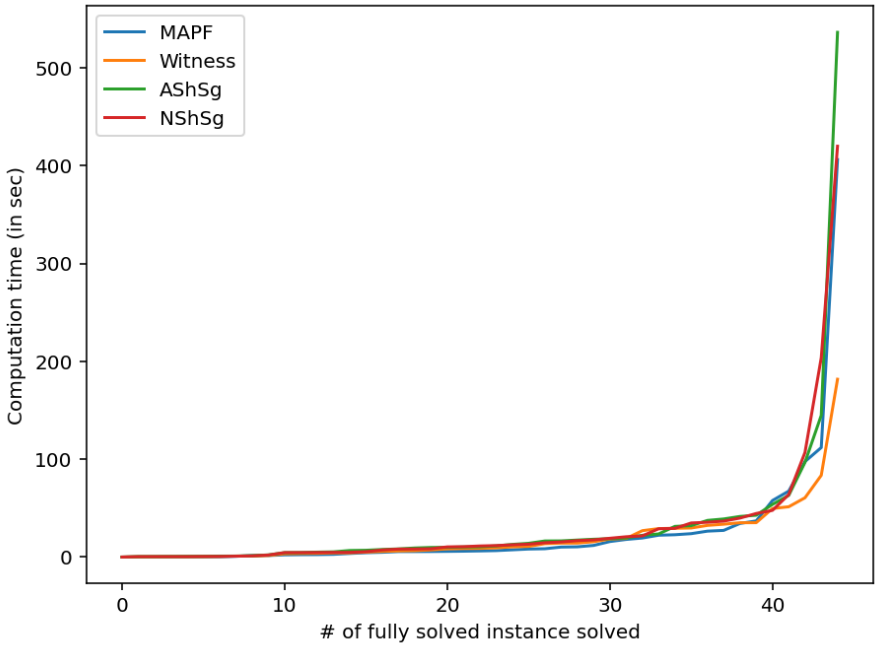
Approach	# fully solved	Expected % agent with a path			
		Horizon | 0	1	3	5
NStPc	24	86	87	88	90
NStSgC	37	97	97	98	98
AShSg	39	97	97	98	98
AStPc	22	80	80	82	83
MAPF	44	100	100	100	100
NStSg	37	97	97	98	98
AStSgCD	17	72	72	74	75
NShSg	37	96	96	97	98
NShSgPc	23	77	78	78	80
AStSgPc	17	72	72	74	75
AStSg	37	97	97	98	98
Witness	38	96	97	97	98
NShSgPcCD	21	83	83	84	85
NStSgD	37	97	97	98	98
NShPc	31	88	89	89	90
NStSgPc	15	75	76	77	78
AShSgCD	39	97	97	98	98
NStSgCD	37	97	97	98	98

This table confirms result concluded with table 5.3. Further emphasizing the effectiveness of the **NShSg** approach, which, among other, have the highest number of fully solved instances. It also confirms that pre-computed paths approach do not perform well since it register the lowest score in terms of number of fully solved instances and in terms of expected proportion of agent with a path. The data also indicates that combining Diamond and Corridor extensions does not appear to enhance the solving process. In fact, it even results in worse results compared to similar approaches without these extensions, as evidenced by **NShSgCD** compared to **NShSg** and **AStSgCD** compared to **AStSg**. Furthermore, we can notice multiple approaches that are providing really high expected proportion of fully solved instance; **NStSg**, **NStSgCD**, **NStSgD**, **NStSgC** and **AStSg**. Additionally, multiple approaches demonstrate high expected proportions of fully solved instances, including **NStSg**, **NStSgCD**, **NStSgD**, **NStSgC**, and **AStSg**.

It is noteworthy that increasing the horizon does not significantly impact the solving proportion. It is, as expected, always raise the value of Expected portion of agent with a path.

From these tables, we selected the following approaches; **MAPF**, **Witness**, **AShSg**, **AStPc**, **AStSg** and **NShSg**.

Figure 5.2: Cactus plot given selected approaches



The plot illustrated in Figure 5.2, shows the performance of the different approaches. As computed for table 3, to be considered as solved, an approach must find a path for all agents.

Plan Merging approaches can provide a partial solution faster than the base MAPF encoding, but have worse result if we try to reach a complete solution. Given the great performance of the witness solver, it seems that the Path Selection processes we defined are not efficient. It however shows that separate MAPF in two separate steps improves the computation time. With a better and smarter IPF, we possibly can provide even better results.

Detailed performance

From the best approach **NShSg** outlined by the previous result, we provide in Table 5.4, the total time and the proportion of each steps composing the approach.

The table illustrates the time distribution across different steps of the approach. While generating initial paths is relatively efficient, the process of finding partial solutions is a significant portion of the computation time. These insights tells us what required the most an optimization; **partial solving**. However, this step depends on the previous step and on the solver itself. Another possible optimization track is the IPF step which could be easily improved with faster programming language. On the other hand the **witness** approach shows

Table 5.4: Approach Decomposition

Step Name	Total time			% of the process	
	Id.	NShSg	Witness	AShSg	Witness
IPF		49	41	7%	10%
Heatmaps Computation		14	-	2%	-
Summed Heatmap per Path		0	-	0%	-
Path Elimination		7	-	1%	-
Path Selection		8	1	1%	0%
Partial Solving		551	352	87%	89%

us that one random path is enough to have really good result. We can conclude that smarter IPF could provide better results.

5.2 Conclusion

In this work, we have established different Plan Merging methods with some of them able to outperform the baseline MAPF solver when only a partial solution is requested. We provided approaches based on conflicts, in particular through heatmaps. Through the different step we defined; Individual Path Finding, Path Selection and Partial Solving, we described multiple ways or parameters to achieved theme.

We provided a baseline for Individual Path Finding with multiple parameters, such as, the number of paths, additional path computing with modified path length to provide different set of paths for each agents. While IPF could potentially be optimized further with languages like C++, the baseline we outlined remains suitable for experimental purposes.

We provided two different approach for the Path Selection step which is based on potential conflict and on heatmaps. With a particular focus on heatmap, with multiple ways of eliminating and selecting paths.

An finally, we formulated a partial solver based on the baseline MAPF solver, which computes solutions or partial solutions using subgraphs and/or precomputed paths.

The results demonstrated that the most effective approaches we produced is the witness solver. The witness being an approach using one randomly selected path for each agent converted into a subgraph. Some approaches still provide a partial solution faster than the MAPF baseline. This observation highlights the limitations of our defined approaches but also pointing towards potential improvement in future work, particularly emphasizing on the IPF and path selection processes.

References

[1] C. Anger et al. "A Glimpse of Answer Set Programming". In: *Künstliche Intelligenz* 19.1 (2005), pp. 12–17.

[2] D. Atzmon et al. "Probabilistic Robust Multi-Agent Path Finding". In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS'20)*. Ed. by J. Beck et al. AAAI Press, 2020, pp. 29–37.

[3] C. Baral, T. Nam, and L. Tuan. "Reasoning about Actions in a Probabilistic Setting". In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*. Ed. by R. Dechter, M. Kearns, and R. Sutton. AAAI Press, 2002, pp. 507–512.

[4] R. Barták and J. Svancara. "On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective". In: *Proceedings of the Twelfth International Symposium on Combinatorial Search (SOCS'19)*. Ed. by P. Surynek and W. Yeoh. AAAI Press, 2019, pp. 10–17.

[5] Daniel Delling et al. "Engineering route planning algorithms". In: *Algorithmics of large and complex networks: design, analysis, and simulation*. Springer, 2009, pp. 117–139.

[6] Daniel Foead et al. "A systematic literature review of A* pathfinding". In: *Procedia Computer Science* 179 (2021), pp. 507–514.

[7] M. Gebser et al. "Experimenting with robotic intra-logistics domains". In: *Theory and Practice of Logic Programming* 18.3-4 (2018), pp. 502–519. DOI: 10.1017/S1471068418000200.

[8] Christian Häcker et al. "Most Diverse Near-Shortest Paths". In: *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*. 2021, pp. 229–239.

[9] Tesshu Hanaka et al. "Computing diverse shortest paths efficiently: A theoretical and experimental study". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 4. 2022, pp. 3758–3766.

[10] M. Husár et al. "Reduction-based Solving of Multi-agent Pathfinding on Large Maps Using Graph Pruning". In: *Proceedings of the Twenty-first International Conference on Autonomous Agents and Multiagent Systems (AAMAS'22)*. Ed. by P. Faliszewski et al. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2022, pp. 624–632. DOI: 10.5555/3535850.3535921.

[11] R. Kaminski et al. "How to Build Your Own ASP-based System?!" In: *Theory and Practice of Logic Programming* (2021), pp. 1–63. DOI: 10.1017/S1471068421000508.

[12] Nerea Luis, Susana Fernández, and Daniel Borrajo. "Plan Merging by reuse for multi-agent planning". In: *Applied Intelligence* 50 (2020), pp. 365–396.

[13] Hang Ma et al. "Feasibility study: Moving non-homogeneous teams in congested video game environments". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 13. 1. 2017, pp. 270–272.

[14] Bernhard Nebel. "On the Computational Complexity of Multi-Agent Pathfinding on Directed Graphs". In: (2019). DOI: 10.48550/ARXIV.1911.04871. URL: https://arxiv.org/abs/1911.04871.

[15] G. Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66.

[16] R. Stern et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks". In: *Proceedings of the Twelfth International Symposium on Combinatorial Search (SOCS'19)*. Ed. by P. Surynek and W. Yeoh. AAAI Press, 2019, pp. 151–159.

[17] R. Stern et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks". In: *CoRR* abs/1906.08291 (2019). URL: http://arxiv.org/abs/1906.08291.

[18] Roni Stern. "Multi-agent path finding–an overview". In: *Artificial Intelligence* (2019), pp. 96–115.

[19] Pavel Surynek. "Candidate Path Selection Heuristics for Multi-Agent Path Finding: A Novel Compilation-Based Method." In: *ICAART (3)*. 2023, pp. 517–524.

[20] Manuela Veloso et al. "Cobots: Robust symbiotic autonomous mobile service robots". In: *Twenty-fourth international joint conference on artificial intelligence*. Citeseer. 2015.

[21] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI magazine* 29.1 (2008), pp. 9–9.

Appendix

Table 5.5: Table of the approaches reference name and their description

Name	Value
NStPc	No additional path, heatmap simple threshold elimination, pre-computed path on full graph
NStSg	No additional path, heatmap simple threshold elimination, subgraph approach
NStSgPc	no additional paths, simple threshold elimination, precomputed path on subgraph
AStPc	Additional path, heatmap simple threshold elimination, pre-computed path on full graph
AStSg	Additional path, heatmap simple threshold elimination, subgraph approach
AStSgPc	Additional paths, simple threshold elimination, precomputed path on subgraph
NStSgC	No Additional path, heatmap simple threshold elimination, subgraph approach with corridor
NStSgD	No Additional path, heatmap simple threshold elimination, subgraph approach with diamond
NStSgCD	No Additional path, heatmap simple threshold elimination, subgraph approach with diamond and corridor
AStSgCD	No Additional path, heatmap simple threshold elimination, subgraph approach with diamond and corridor
NShPc	No additional path, summed heatmap elimination, pre-computed path on full graph
NShSg	No additional path, summed heatmap elimination, subgraph approach
NShSgPc	No additional paths, summed heatmap elimination, precomputed path on subgraph
AShPc	Additional path, summed heatmap elimination, subgraph approach
NShPcCD	Additional path, summed heatmap elimination, subgraph approach with diamond and corridor
MAPF	Classical MAPF approach