Preliminary Report: Improving Suboptimal Reduction-based Approaches for Multi-agent Pathfinding

Klaus Strauch

University of Potsdam, Germany

Abstract. Multi-agent Pathfinding (MAPF) is a fundamental problem in robotics and AI, where multiple agents must navigate a shared environment without colliding. This paper presents an approach inspired by the Rolling Horizon Collision Resolution (RHCR) algorithm, in which agent paths consider collisions only for a limited number of steps ahead treating the remainder of each path as a single-agent problem. We apply these concepts in the context of Answer Set Programming (ASP). For each agent, we ground only a limited number of steps and use heuristics to guide their movement. By significantly reducing the planning horizon, our method improves on previous suboptimal reduction-based approaches on instances with large horizons.

1 Introduction

Multi-agent pathfinding (MAPF) involves moving multiple agents from their start positions to their designated goal positions within a shared environment. The objective is to find collision-free paths for all agents. MAPF has numerous real-world applications, such as warehousing [11], airport surface operations [12], and games [16], among others.

Naturally, MAPF has been extensively studied, having many different algorithms to solve it. For example, search-based algorithms like Conflict-based Search (CBS) [1] first compute individual paths for each agent and then resolve conflicts between them. The main advantage of this approach is that pathfinding for individual agents is a polynomial-time problem. The more challenging part is resolving path conflicts, which is handled by a separate conflict-resolution mechanism.

There are also rule-based algorithms where an agent's movement depends on its current state. For example, the priority inheritance with backtracking (PIBT) algorithm [14] uses a set of rules and priorities that determine each agent's movement. This results in a very fast algorithm, though it is not optimal, and is only complete for biconnected graphs, not on general graphs.

Another approach is to reduce MAPF to another formalism and leverage existing solvers. For example, the problem can be encoded as a Boolean Satisfiability (SAT) instance [20], allowing the use of any SAT solver. Hybrid approaches

are also common, where part of the problem is handled via reduction and the rest through search-based techniques (e.g. [2, 19]).

In this work, we focus on reduction-based approaches, and in particular on Answer Set Programming [10]. ASP is a declarative programming paradigm based on stable model semantics, which allows complex problems to be expressed compactly and intuitively.

Reduction-based approaches are commonly used to solve MAPF optimally [5, 20]. However, in many applications where speed is more important than optimality, sub-optimal solutions are sufficient. Here, we focus on sub-optimal MAPF, aiming to find solutions that are close to optimal while prioritizing computational speed.

2 Background

2.1 Multi-Agent Pathfinding

The *multi-agent pathfinding* problem (MAPF) [17] is a pair (G, A), where G is an undirected graph $G = (V, E)$ and A is a set of agents. Each agent $a \in A$ is associated with a start vertex $s_a \in V$ and a goal vertex $g_a \in V$.

Time is considered discrete. Between two consecutive timepoints, an agent can either move to an adjacent vertex (move action) or stay at its current vertex (wait action). The movement of an agent is captured by its path.

A *path* π_a of agent a is a list of vertices that starts at s_a and ends at g_a. Let $\pi_a(t)$ be the vertex of a at timepoint t according to π_a, then $\pi_a(0) = s_a$ and $\pi_a(|\pi_a|) = g_a$. Additionally, for all timepoints $t < |\pi_a|$, $(\pi_a(t), \pi_a(t+1)) \in E$ or $\pi_a(t) = \pi_a(t+1)$, that is, at each timepoint agent a either moves along an edge or waits at a vertex, respectively.

There is a *conflict* between paths π_a and π_b at timepoint t if $\pi_a(t) = \pi_b(t)$ (*vertex conflict*) or $\pi_a(t) = \pi_b(t+1)$ and $\pi_b(t) = \pi_a(t+1)$ (*swapping conflict*).

A *plan* Π is a set of paths, one for each agent. A *solution* is a conflict-free plan, i.e., a plan Π where no two paths of distinct agents have conflicts.

A solution is *optimal* if it has the lowest cost among all possible solutions. The cost $C(\pi_a)$ of path π_a equals the number of actions performed in π_a until the last arrival at g_a, not counting any subsequent wait actions. Formally, $C(\pi_a) = \max(\{0 < t \leq |\pi_a| \mid \pi_a(t) = g_a, \pi_a(t-1) \neq g_a\} \cup \{0\})$. Note that waiting at the goal counts towards the cost if the agent leaves the goal at any time in the future.

There are two commonly used cost functions to evaluate the quality of a plan Π:

1. *sum-of-costs* (SOC), which is the sum of costs of all paths $C_{SOC}(\Pi) = \sum_i C(\pi_i)$
2. *makespan* (MKS), which is the maximum cost among all paths $C_{MKS}(\Pi) = \max_i C(\pi_i)$.

Sum-of-costs optimal solutions are typically used in applications to minimize the usage of a resource (e.g. fuel). Makespan optimal solutions are used to minimize the time taken to complete a task.

Although the decision problem of whether a MAPF instance has any solution is polynomial [6], bounding the movement of agents turns it into an NP-complete problem [24, 18]. This makes finding an optimal solution significantly harder than simply determining the existence of any solution.

In general, reduction-based approaches solve MAPF by initially searching for a solution within a given cost bound. If no solution is found, the bound is iteratively increased until one is obtained.

The current state-of-the-art reduction-based solver for sub-optimal MAPF was introduced in [22]. It builds on the optimal algorithm from [21], where a cardinality constraint is imposed on the total number of moves across all agents. If no solution is found, this bound is increased by one. The first solution found in this process is guaranteed to be optimal with respect to sum-of-costs. The sub-optimal variant relaxes this by multiplying the bound by a factor. For instance, multiplying the bound by 1.01 allows solutions that are at most one percent worse than optimal. Alternatively, the cardinality constraint can be removed altogether. This approach, however, applies specifically to sum-of-costs optimal solutions.

A simpler approach to finding sub-optimal solutions is to increase the bound by more than one in each iteration. This works for both cost functions.

2.2 Answer Set Programming

For a more detailed view of ASP, we refer to [3]. Here, we provide a brief overview.

Let $\mathcal{A}$ be a set of atoms. A rule r over $\mathcal{A}$ in ASP has the form:

$$H \text{ :- } a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n.$$

where H is the head of the rule and has the form $\{a\}$ for some atom $a \in \mathcal{A}$, which makes it a choice rule, an atom $a \in \mathcal{A}$ and it is a normal rule, or $\bot$ and it is an integrity constraint.

Intuitively, a rule states that if all atoms in the body are true, then the atom in the head is also true. Note that for choice rules, this means that we have a choice on the truth value of the head atom. For integrity constraints, the head atom is always false. Hence, we have to avoid the situation where all atoms in the body are true.

A program $\mathcal{P}$ is a set of rules. Additionally, we can define a set of optimization directives. In these directives, we specify a weight for an atom. The solver then tries to minimize the total weight of the atoms that are true in a solution. A solution of a program $\mathcal{P}$ is a set of atoms $M \subseteq \mathcal{A}$ such that all rules in $\mathcal{P}$ are satisfied by M.

2.3 ASP Encoding for MAPF

We use the ASP encoding for MAPF from [5] which can be seen in Listing 1.1. The encoding represents the graph G using atoms of the form `vertex/1` and `edge/2`. The agents are represented using atoms of the form `agent/1`. The start

and goal positions of each agent are represented using atoms of the form `start/2` and `goal/2`, respectively. The position of an agent is given by the atom `at/3`, which indicates that an agent is at a specific vertex at a specific time step. The movement of agents is represented using the atom `move/4`, which indicates that an agent moves from one vertex to another at a specific time step.

Line 1 of the encoding defines the time steps for each agent based on its horizon. Line 2 initializes the position of each agent at time step 0 to its start position. Line 4 defines which moves are possible for an agent at a given time step, based on the edges of the graph and the reachability set. Line 6 updates the position of an agent based on its moves while Line 7 ensures that an agent can only move from its current position. Line 9 ensures that an agent remains at its position if it does not move. Lines 11 and 12 prevent vertex and swapping conflicts, respectively. Line 14 ensures that each agent has a defined position at each time step.

```
1  time(A,1..T) :- agent(A), horizon(A,T).
2  at(A,P,0) :- start(A,P), agent(A).

4  { move(A,U,V,T) : edge(U,V), reach(A,V,T), reach(A,U,T-1)} 1 :- time(A,T).

6  at(A,V,T) :- move(A,_,V,T).
7            :- move(A,U,_,T), not at(A,U,T-1).

9  at(A,V,T) :- at(A,V,T-1), not move(A,V,_,T), time(A,T), reach(A,V,T).

11 :- { at(A,V,T) } > 1, vertex(V), time(_,T).
12 :- move(_,U,V,T), move(_,V,U,T), U < V.

14 :- { at(A,V,T) } != 1, agent(A), time(A,T).
```

Listing 1.1: ASP encodings for bounded MAPF.

3 Related Work

To the best of our knowledge, the first use of a windowed approach for MAPF is in [16]. where Windowed Hierarchical Cooperative A* (WHCA*) is introduced. WHCA* uses a fixed-size window to plan paths for agents using location-time A* within the window, which is a variant of A* that avoids vertices and edges used by previously planned agents. After planning the window, the planning continues to the next window until all agents reach their goals. Notably, in their setting, agents only had to reach their goal at some point in the planning. It was not required that all agents be at their goal at the same time. Additionally, agents are planned one at a time.

Later, this idea was revisited in [9]. where the windowed approach is applied to lifelong MAPF, a variant in which agents receive new goals as soon as they reach their current ones. Similarly, the low-level search uses location-time A* to plan paths within a window, adapted to the lifelong setting in which an agent has a sequence of goals it must visit. In a similar manner, agents do not have to arrive at their goals at the same time. and agents are planned one at a time.

Windowed solvers were used by the winning teams of the League of Robot Runners[1] competition [4]. Since the competition required the solver to provide an action for all agents at one-second intervals, windowed solvers were a natural fit due to their fast planning time for a single window, which is sufficient to provide actions for the next couple of seconds.

A common disadvantage of windowed approaches is that they can be stuck in deadlocks. The usual approach to resolve them is to replan with a larger window size. However, this has the downside of increasing planning time significantly. In the worst case, it would devolve to planning the complete horizon at once, which is what windowed planners are trying to avoid. Recently, [23] introduced a heuristic for windowed solvers with completeness guarantees. Their approach relies on a heuristic value given to a configuration of agents, where a configuration is defined as the set of positions for each agent. The windowed solver plans the next window by choosing actions that lead to configurations with better heuristic values at the end of the window. To ensure completeness, the heuristic value of a configuration is increased whenever it is chosen. Hence, deadlocks are avoided simply because the heuristic value of a configuration worsens each time it is chosen, and eventually, all configurations will be explored. To avoid the obvious issue of the exponential number of configurations, the authors propose clustering agents into interacting groups and computing the heuristic value based on the groups and not the complete problem. The low-level solver for the configurations is a modified version of CBS that can plan for a window of size one.

All of those solvers are search-based, and they leverage some type of prioritized planning. To the best of our knowledge, our approach is the first reduction-based windowed solver for MAPF. The main difference between search-based and reduction-based solvers is that reduction-based solvers consider all agents simultaneously. Additionally, there is the benefit of easily expressing complex constraints and objectives due to the declarative nature of reduction-based approaches.

In the realm of reduction-based approaches for MAPF, most approaches focus on finding optimal solutions [5, 20]. These approaches typically involve grounding the program multiple times, starting with a lower bound on the cost or horizon, depending on the cost function. If no solution is found, the bound is increased iteratively until a solution is found. This process can lead to large grounding sizes, especially for larger problem instances. Additionally, a large amount of time is spent grounding the same set of rules repeatedly.

Sub-optimal reduction-based approaches for the Sum-of-cost objective have been explored in [22]. However, these approaches still require grounding the necessary horizon at once, which can lead to large grounding sizes and times. The approach uses the same algorithm as the optimal reduction-based approaches, however, the bound placed on the cost is relaxed. In practice, this relaxation is done by either multiplying the bound by a factor, or by not imposing a bound at all.

[1] https://www.leagueofrobotrunners.org/

Outside reduction-based approaches, Explicit Estimation CBS (EECBS) [8] is a well-known bounded-suboptimal MAPF solver. It extends the CBS algorithm by incorporating an estimation of the solution cost in the high-level search. This estimation allows EECBS to select nodes that are more likely to lead to solutions still within the sub-optimality bound. The current state-of-the-art MAPF solver is another search-based approach called LACAM [13]. It uses a similar search strategy to A*, but successor states are lazily generated using the PIBT [14] algorithm. This avoids the effort of the full expansion of all possible states, leading to significant speedups in practice. Additionally, LACAM* eventually finds optimal solutions, by expanding all possible states if given enough time.

4 Approach

Current suboptimal approaches for Multi-Agent Pathfinding (MAPF) share a common weakness: they must ground at least to the lower bound on the number of moves. For smaller problem instances, this is not an issue, but for larger instances, it can lead to a very large number of rules.

Our approach draws inspiration from the windowed rolling horizon method [23]. The windowed rolling horizon approach plans a partial solution where collisions are considered only within a specified time window. This method is primarily used for lifelong MAPF, where new goals are introduced as old ones are completed. Additionally, because execution is interleaved with planning, the method requires very fast planning algorithms to keep pace with the execution.

In our approach, we use a similar concept but do not interleave planning and execution. Instead, we perform a series of planning steps. We refer to such a step as a planning window and the amount of time steps considered is the window size. Each window begins with the agents at their last position of the previous window. The first window starts with the agents at their initial positions. Planning steps are performed iteratively until a complete solution is found.

To use this approach with ASP, we must address two key issues. First, we need a method to guide the agents toward their goals. Second, we must ensure that deadlocks are resolved.

4.1 Guiding Agents Towards Their Goals

Before presenting our methods, we define our notion of progress within a given window. Let w denote the size of the window, and let $sp(v, u)$ represent the length of the shortest path between vertices v and u. We say that an agent makes progress in a window if $sp(\pi_a(0), g(a)) > sp(\pi_a(w), g(a))$, where $\pi_a(0)$ is the agent's position at the beginning of the window, and $\pi_a(w)$ is the position at the end of the window.

We define the amount of progress as $sp(\pi_a(0), g(a)) - sp(\pi_a(w), g(a))$. Note that within each window the maximum possible progress is w.

To guide agents toward their goals, we utilize the reachability preprocessing method described in [5]. This preprocessing determines which vertices an agent

can occupy at a given time step such that it can still reach its goal within a specified time horizon. If the horizon is set to $sp(\pi_a(0), g(a))$, then the agent is restricted to making moves that reduce its distance to the goal. As a result, the agent will make progress by exactly w, where w is the window size.

However, this approach may be too restrictive, as agents must follow their shortest path and are not allowed to wait. To address this, we propose a more flexible approach that allows agents some wiggle room, by calculating reachable positions with a horizon that includes δ additional time steps. For $\delta = 0$, the agent must follow one of its shortest paths. For $\delta > 0$, agents are allowed to wait for up to δ time steps as well as stray from the shortest path by half of δ steps.

Next, we analyze how the value of δ interacts with the window size. If the window size is larger than δ, agents can wait for δ steps and still make progress of at least $w - \delta$ toward their goal. If the window size equals δ, agents can wait for the entire window, resulting in no progress. If the window size is smaller than δ, agents may make negative progress.

Carefully selecting the value of δ is crucial for the performance of our approach. If δ is too small, the movement of the agents becomes overly restricted, and they may be unable to make meaningful progress. If δ is too large, the agents might be allowed to wait excessively, resulting in only minimal progress. In a similar manner, we must also consider the window size. If the window is too small, the agents lack sufficient scope to make informed decisions. If it is too large, the time required to ground and solve each window increases significantly.

Next, we examine the use of optimization directives in our approach. Consider the case of agents on a map with non-intersecting paths. In such a scenario, the optimal solution for each window is for every agent to follow its shortest path. However, when $\delta > 1$, the solver may return a solution that does not maximize progress. To prevent this, we introduce an optimization directive that favors solutions maximizing the overall progress of all agents. In practice, we achieve this by minimizing the sum of the distances between the position of the agents at the end of the window and their respective goals. To implement this, we perform a breath-first search from each goal vertex. Then, for each reachable vertex v for an agent a, we add the distance to the goal $sp(v, g(a))$ as a fact with signature `dist/3`. We then use this fact in an optimization directive as follows:

```
#minimize{D@2,A : at(A,V,T), last(T), dist(A,V,D)}.
```

Here, `at(A,V,T)` indicates that agent A is at vertex V at time T, `last(T)` signifies that T is the final time step of the window, and `dist(A,V,D)` represents the distance D from vertex V to the goal of agent A. In short, this directive sums the distances of all agents to their goals at the end of the window and minimizes this sum.

Notice that the original optimization directive has a priority of 2, leaving room for additional directives with higher priority. If this directive is used alone, the solver may produce solutions where agents that have already reached their goals move away from them to make way for others. This behavior is undesirable, as it can increase the sum-of-costs of the solution. To prevent this, we introduce an

additional optimization directive with a higher priority. This directive encourages agents to remain at their goals once they reach them. We implement this using the following rule and optimization directive:

```
atgoal(A,T) :- goal(A,V), at(A,V,T), time(T).
#maximize{1@1,A,T : atgoal(A,T)}.
```

The first rule defines `atgoal/2`, indicating that agent A is at its goal vertex V at time T. The optimization directive then maximizes the total number of timesteps that agents spend at their goal vertices, with a priority of 1. This ensures that agents remain at their goals whenever possible, preventing unnecessary movements that could increase the sum-of-costs.

4.2 Resolving Deadlocks

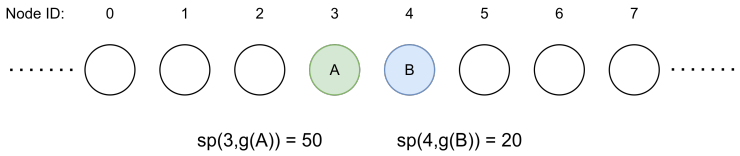
Consider the situation where two agents arrive at a corridor and must cross each other (Figure 1a). In a standard setting, the solver considers the full horizon of the problem. It would then likely allow one agent to pass through the corridor while the other waits, or find alternative paths for them. In our approach, however, the agents only consider the current window. Together with the directive to maximize total progress, the solver may attempt to have both agents cross the corridor simultaneously. This results in a deadlock, as each agent waits for the other to yield.

To address this, we introduce an additional optimization directive. In the example above, the progress optimization treats two cases as equally good: (1) both agents waiting in front of each other, and (2) one agent walking back while the other proceeds through the corridor. For example, we look at Figure 1b and 1c. At the start when the agents meet, the sum of the distances to their goals is 70. If agent A moves back while agent B proceeds, the sum remains 70. Similarly, if agent A proceeds while agent B moves back, the sum also remains 70. No matter if the agents stay in place or one agent moves back and the other forward, the total progress is zero. We can exploit this equivalence to establish a preference between these solutions. Naturally, we prefer that one agent progresses through the corridor, even if the other must temporarily backtrack. To enforce this, we add a directive that minimizes the maximum remaining distance to the goal of any agent at the end of the window. This encourages the solver to prioritize the agent with the longest remaining path, effectively breaking the deadlock. We implement this using the following rules and optimization directive:

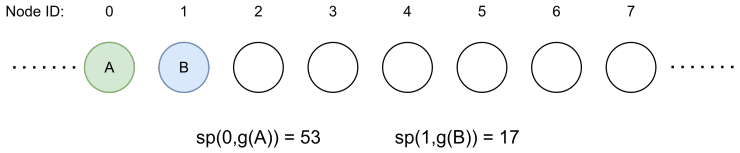
```
max(D) :- at(A,V,T), last(T), dist(A,V,D).
max(D-1) :- max(D), D > 0.
#minimize{D@3 : max(D)}.
```

Here, the first rule identifies the maximum distance D to any goal at the end of the window. The second rule generates all values from D down to zero. The optimization directive then minimizes this maximum distance. Notice that this directive has a lower priority than the previous ones, so that we choose only among solutions that maximize total progress.

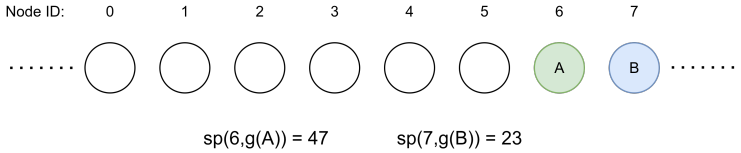
Fig. 1: Example of a deadlock. The agents start on opposite sides of a corridor and must cross each other. All possible moves lead to the same progress of zero.



(a) Both agents waiting in front of each other.



(b) Agent A moves back while Agent B proceeds.



(c) Agent A proceeds while Agent B moves back.

Unfortunately, this does not resolve all deadlocks. Consider again the situation where two agents arrive at a corridor. If both agents have similar distances to their goals, the solver is unable to break the deadlock. In this case, the identity of the agent with the longest remaining distance keeps shifting, causing the agents to move back and forth inside the corridor without making progress.

Another case occurs when both agents have their goals inside the corridor and must cross each other. Here, the solver is also unable to resolve the deadlock. We note, however, that this situation would not be problematic in a lifelong version of MAPF as an agent could simply reach its goal, receive a new one, and the deadlock would be resolved naturally.

To address both of these cases, we increase the window size until the deadlock is resolved. This approach requires detecting deadlocks effectively, making the detection strategy crucial for performance. At present, we detect deadlocks by identifying windows where no progress is made. This implies that we can only recognize a deadlock once all agents not involved have already reached their goals.

5 Benchmarks

We run the benchmarks on the instances from [5]. These consist of empty, maze, room, and random layouts, with sizes 16×16, 32×32, 64×64 and 128×128. For each instance, we start with 5 agents and increment the number of agents by 5, up to a maximum of 100. Finally, we run the experiments on an Intel Xeon E5-2650v4 under Debian GNU/Linux 10, with a memory limit of 28 GB, and a timeout of 5 minutes per instance. All benchmarks were run using *clingo* with version 5.8.

Approach	**Solved**	**SoC-dif**	**Mks-dif**	**Solved**	**SoC-dif**	**Mks-dif**
		Maze			**Random**	
`FH-ASP-1`	244	**1.02**	1.14	505	**1.01**	1.14
`FH-ASP-5`	364	1.04	**1.08**	620	1.03	**1.07**
`FH-ASP-100`	515	1.17	1.15	**779**	1.15	1.13
`RH-ASP-v1`	488	1.83	**1.08**	622	1.75	1.11
`RH-ASP-v2`	**525**	1.13	1.11	648	1.07	1.11
		Room			**Empty**	
`FH-ASP-1`	298	**1.02**	1.15	661	**1.01**	1.09
`FH-ASP-5`	437	1.04	**1.02**	711	1.03	**1.06**
`FH-ASP-100`	**662**	1.23	1.20	**756**	1.06	1.07
`RH-ASP-v1`	458	1.81	1.11	667	1.87	1.16
`RH-ASP-v2`	508	1.13	1.12	**756**	1.05	1.17
		Total				
`FH-ASP-1`	1708	**1.01**	1.11			
`FH-ASP-5`	2132	1.03	**1.07**			
`FH-ASP-100`	**2717**	1.15	1.13			
`RH-ASP-v1`	2235	1.81	1.11			
`RH-ASP-v2`	2437	1.09	1.13			

Table 1: Results by map type. First column shows the number of solved instances. Second and third column show the average difference (in percentage) to the lower bound of sum-of-costs and makespan, respectively. The best values in each column are shown in bold.

Table 1 shows the results on all instances split by map type. The configurations with `FH-ASP` (Short for Full Horizon ASP) are the existing approaches from [5] and are followed by the percentage by which the cost bound can differ from optimal. For instance, `FH-ASP-5` allows solutions that are at most five percent worse than optimal. `RH-ASP-v1` (Rolling Horizon) is our new approach. We can see that `FH-ASP-100` solves the most instances overall, followed by `RH-ASP-v2`. It is also clear that `RH-ASP-v1` produces solutions with a higher sum-of-costs, but with a comparable makespan, while the quality of the solutions of `RH-ASP-v2` remains

Approach	Solved	SoC-dif	Mks-dif
FH-ASP-1	543	**1.007**	**1.013**
FH-ASP-5	652	1.017	1.023
FH-ASP-100	682	1.044	1.036
RH-ASP-v1	787	1.985	1.016
RH-ASP-v2	**811**	1.066	1.018

Table 2: Results on instances with horizon higher than 100. First column shows the number of solved instances. Second and third column show the average difference (in percentage) to the lower bound of sum-of-costs and makespan, respectively. The best values in each column are shown in bold.

competitive. Although expected, the difference in sum-of-costs is quite significant. This is likely due to the fact that RH-ASP-v1 does not explicitly optimize for sum-of-costs during its search process. This highlights the importance of the optimization directive that tries to keep agents in their goals.

This approach is meant for instances with large horizons. Hence, we examine Table 2 which shows the results on instances with a horizon greater than 100. Here, RH-ASP-v2 solves the most instances, followed by RH-ASP-v1. While RH-ASP-v1 still shows a weakness in the quality of the solutions, RH-ASP-v2 manages to keep quality comparable.

5.1 Future Work

The main issue with the current system is that it considers all agents simultaneously. However, it is possible for some groups of agents to have no interactions with other groups. This is especially likely when considering a limited horizon. Naturally, splitting the agents into non-interacting groups can improve performance, since the solver needs to process less information at once. Moreover, handling distinct problems in the same solver can degrade performance, as solving one part may interfere with another [15].

We detect these non-interacting groups by leveraging reachability preprocessing. If the reachable positions of two agent groups do not overlap with those of any other group, then the groups are considered non-interacting. Here, reachable positions are pairs (V, T), where V is a vertex and T is a time point. This allows agents from different groups to use the same vertices at different times without conflict.

The next set of improvements concerns how we guide the agents. Currently, we rely heavily on optimization directives. While effective, these can be slow. Heuristic directives offer an alternative. Instead of optimizing for total progress, we can instruct the solver to prioritize paths that minimize progress first (i.e., prioritize moving to vertices in the shortest path). Only if these paths fail does it consider other options. Additionally, the current deadlock resolution method only works if the horizon of the involved agents is the largest. An approach where we

detect the deadlock between agents, and only apply the deadlock resolution to those agents, could be more effective.

Another way to improve speed is by identifying agents that do not interact with others in the current window. These agents can then be set to follow their shortest paths without further consideration. One way to enforce this is by assigning these agents a δ of zero, while other agents have their δ defined according to the given parameters. An alternative approach is to pre-select a path for these agents and plan paths only for the remaining agents, treating the pre-selected paths as moving obstacles.

A major issue with the current system is the sum-of-costs of the solutions. A possible solution is to introduce ways for the agents to also consider alternative paths that are not necessarily in their shortest paths. For example, if an agent wants to use a corridor that contains another agent's goal, we block the use of that corridor for that agent. This forces the agent to find an alternative path, which may lead to a solution with a lower sum-of-costs, as the other agent might not have to wait for the first agent to cross the corridor.

6 Conclusion

Current reduction-based approaches for sub-optimal MAPF rely on grounding the necessary horizon at once. This can lead to large grounding and solving times. We introduce a new solver based on the windowed rolling horizon approach to solve MAPF instances in small chunks. This leads to easier grounding and solving as each subproblem is many times smaller. We run benchmarks comparing our approach to existing methods and find that it outperforms them when the horizon is large. The downside of this approach is that resolving deadlocks can take a significant amount of time, leading to worse performance in many cases.

References

[1] E. Boyarski et al. "ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding". In: *Proceedings of the Twenty-fourth International Joint Conference on Artificial Intelligence (IJCAI'15)*. Ed. by Q. Yang and M. Wooldridge. AAAI Press, 2015, pp. 740–746. URL: http://ijcai.org/proceedings/2015.

[2] G. Gange, D. Harabor, and P. Stuckey. "Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation". In: *Proceedings of the Twenty-ninth International Conference on Automated Planning and Scheduling (ICAPS'19)*. Ed. by J. Benton et al. AAAI Press, 2019, pp. 155–162.

[3] M. Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012. DOI: 10.1007/978-3-031-01561-8.

[4] H. Jiang et al. "Scaling Lifelong Multi-Agent Path Finding to More Realistic Settings: Research Challenges and Opportunities". In: *Proceedings of the International Symposium on Combinatorial Search* 17 (2024), pp. 234–242. DOI: 10.1609/socs.v17i1.31565.

[5] R. Kaminski et al. "Improving the Sum-of-Cost Methods for Reduction-Based Multi-Agent Pathfinding Solvers". In: *Proceedings of the Sixteenth International Conference on Agents and Artificial Intelligence (ICAART'24)*. Ed. by A. Rocha, L. Steels, and H. van den Herik. SciTePress, 2024, pp. 264–271. DOI: 10.5220/0012353500003636. URL: https://www.scitepress.org/ProceedingsDetails.aspx?ID=7POrHKUPtlI=.

[6] D. Kornhauser, G. Miller, and P. Spirakis. "Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications". In: *25th Annual Symposium on Foundations of Computer Science, 1984*. 1984, pp. 241–250. DOI: 10.1109/SFCS.1984.715921.

[7] S. Kraus, ed. *Proceedings of the Twenty-eighth International Joint Conference on Artificial Intelligence (IJCAI'19)*. ijcai.org, 2019.

[8] J. Li, W. Ruml, and S. Koenig. "EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding". In: *Proceedings of the Thirty-fifth National Conference on Artificial Intelligence (AAAI'21)*. AAAI Press, 2021, pp. 12353–12362.

[9] J. Li et al. "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.13 (2021), pp. 11272–11281. DOI: 10.1609/aaai.v35i13.17344.

[10] V. Lifschitz. *Answer Set Programming*. Springer-Verlag, 2019. DOI: 10.1007/978-3-030-24658-7.

[11] H. Ma et al. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks". In: *Proceedings of the Sixteenth Conference on Autonomous Agents and MultiAgent Systems (AAMAS'17)*. ACM Press, 2017, pp. 837–845.

[12] R. Morris et al. "Planning, Scheduling and Monitoring for Airport Surface Operations". In: *AAAI Workshop: Planning for Hybrid Systems*. 2016.

[13] K. Okumura. "Engineering LaCAM*: Towards Real-Time, Large-Scale, and Near-Optimal Multi-Agent Pathfinding". In: *ArXiv* abs/2308.04292 (2023). URL: https://api.semanticscholar.org/CorpusID:261065877.

[14] K. Okumura et al. "Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding". In: *Proceedings of the Twenty-eighth International Joint Conference on Artificial Intelligence (IJCAI'19)*. Ed. by S. Kraus. ijcai.org, 2019, pp. 535–542. DOI: 10.24963/ijcai.2019/76.

[15] K. Pipatsrisawat and A. Darwiche. "A Lightweight Component Caching Scheme for Satisfiability Solvers". In: *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*. Ed. by J. Marques-Silva and K. Sakallah. Vol. 4501. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 294–299.

[16] D. Silver. "Cooperative Pathfinding". In: *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE'05)*. Ed. by R. Young and J. Laird. AAAI Press, 2005, pp. 117–122.

[17] 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.

[18] P. Surynek. "An Optimization Variant of Multi-Robot Path Planning Is Intractable". In: *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*. Ed. by M. Fox and D. Poole. AAAI Press, 2010, pp. 1261–1263. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1768.

[19] P. Surynek. "Problem Compilation for Multi-Agent Path Finding: a Survey". In: *Proceedings of the Thirty-first International Joint Conference on Artificial Intelligence (IJCAI'22)*. Ed. by L. De Raedt. ijcai.org, 2022, pp. 5615–5622. DOI: 10.24963/ijcai.2022/783.

[20] P. Surynek. "Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories". In: *Proceedings of the Twenty-eighth International Joint Conference on Artificial Intelligence (IJCAI'19)*. Ed. by S. Kraus. ijcai.org, 2019, pp. 1177–1183.

[21] P. Surynek et al. "Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective". In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence (ECAI'16)*. Ed. by G. Kaminka et al. IOS Press, 2016, pp. 810–818.

[22] P. Surynek et al. "Sub-Optimal SAT-Based Approach to Multi-Agent Path-Finding Problem". In: *Proceedings of the Eleventh International Symposium on Combinatorial Search (SOCS'18)*. Ed. by V. Bulitko and S. Storandt. AAAI Press, 2018, pp. 90–105.

[23] Rishi Veerapaneni et al. "Windowed MAPF with Completeness Guarantees". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 39.22 (2025), pp. 23323–23332. DOI: 10.1609/aaai.v39i22.34499. URL: https://ojs.aaai.org/index.php/AAAI/article/view/34499.

[24] J. Yu and S. LaValle. "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs". In: *Proceedings of the Twenty-seventh National Conference on Artificial Intelligence (AAAI'13)*. Ed. by M. desJardins and M. Littman. AAAI Press, 2013, pp. 1443–1449.