Preliminary Work on Incremental Solving for Multi-Agent Path Finding in ASP

Klaus Strauch and Jiří Švancara

University of Potsdam, Germany
Charles University, Czech Republic

Abstract. We examine the ASP encoding used for one-shot solving of Multi-Agent Path Finding (MAPF) and adapt it for incremental solving to avoid repeated grounding work.

1 Introduction

Multi-Agent Path Finding (MAPF)[9] is the problem of finding a path in a given graph for multiple agents while avoiding collisions. Agents may traverse the graph's edges or wait at their current vertex. All agents have a starting location and must arrive to their goal location. Agents may not occupy the same vertex at the same time instant and may not traverse the same edge in a different direction at the same time, called vertex and edge constraints, respectively.

A possible way to solve MAPF is via search techniques, such as conflict-based-search[2]. Another way is via reduction-based techniques, such as propositional satisfiability (SAT)[1] or answer set programming (ASP)[3], which is the focus of this work.

ASP solvers find (optimal) solutions to MAPF with a series of one-shot calls to the solver[8]. It first attempts to find a solution using the lower bound of the horizon (maximum number of moves). If it fails, it increments the horizon and tries again until a solution is found. Notice that regrounding after increasing the horizon involves introducing few extra rules. In other words, a big portion of the grounding time is spent on recomputing previously grounded rules.

Recently, MAPF encodings incorporate the notion of reachable vertices[10]. Given a horizon, a starting location, and a goal location, a vertex is reachable if there is a path from the starting to the goal location, passing through the vertex, of length at most the horizon. Current encodings make use of this information to avoid grounding unnecesary atoms.

In the following section, we assume that the reader is familiar with ASP syntanx and semantics as well as with the underlying solving mechanism. For more information, we refer to [3] and [4].

2 Single-shot Encoding

We model agent movements using the encoding shown in Listing 1.1, originally shown in [8]. Below, we provide an intuitive explanation of this encoding.

An instance for this encoding includes facts over the predicates `vertex/1` and `edge/2`, which define the graph's vertices and edges, respectively. Additionally, the facts `agent/1`, `start/2`, and `goal/2` describe the agents and their associated start and goal vertices. The lower bounds on the number of moves of an agent, and the vertices they can occupy at each time point, are specified by the facts `starting_horizon/2` and `reach/3`.

Line 1 sets up the individual time points available to each agent. The choice rule in Line 3 chooses a move for each agent for every time point. Note that it is possible to not move, which is interpreted as a wait action. Line 4 assigns the starting position of an agent to the initial time point. Line 5 moves an agent to the position given by the chosen move. If no move is chosen, Line 6 ensures that an agent stays in place. Line 8 ensures that the move starts at the correct vertex. Line 9 makes sure that an agent has exactly one position per time point. Lines 11 and 12 encode vertex and edge collisions, respectively. Finally, Line 13 ensures that all agents are at their goals at the last time point.

```
 1  time(A,1..T+D) :- starting_horizon(A,T), delta(D).

 3  {move(A,U,V,T): edge(U,V), reach(A,V,T)} 1 :- reach(A,U,T-1).
 4  at(A,V,0) :- start(A,V), agent(A).
 5  at(A,V,T) :- move(A,_,V,T).
 6  at(A,V,T) :- at(A,V,T-1), not move(A,V,_,T), time(A,T), reach(A,V,T).

 8  :- move(A,U,_,T), not at(A,U,T-1).
 9  :- {at(A,V,T)} != 1, time(A,T).

11  :- {at(A,V,T)} > 1, vertex(V), time(_,T).
12  :- move(_,U,V,T), move(_,V,U,T), U<V.
13  :- goal(A,V), not at(A,V,H+D), starting_horizon(A,H), delta(D).
```

Listing 1.1. ASP encoding for bounded MAPF.

The given encoding is sufficient to find makespan optimal solutions. However, to find sum-of-cost optimal solutions, additional rules are required. The following encoding assumes the existence of the fact `delta/1`, which specifies the maximum number of extra moves an agent is allowed to make:

```
1  penalty(A,N) :- starting_horizon(A,N+1), N>=0.
2  penalty(A,T) :- starting_horizon(A,N), at(A,U,T), not goal(A,U), T>=N.
3  penalty(A,T) :- penalty(A,T+1), T>=0.

5  bound(H+D) :- H=#sum{T,A: starting_horizon(A,T)}, delta(D).
6  :- #sum{1,A,T: penalty(A,T)} > B, bound(B).
```

Listing 1.2. ASP encoding to enforce the sum-of-cost objective for bounded MAPF.

Line 2 applies a penalty to each timepoint below the shortest path. Line 3 applies the penalty if the agent is not at their goal. On Line 4, the penalty is propagateed to all previous time points. Lines 6 and 7 make sure that the number of extra actions is respected. The cost of the solution is the sum of the penalties.

3 Challenges in Incremental Modelling

Incremental or multi-shot solving handles evolving logic programs. After the initial grounding, new rules can only be added if they are modular [6], meaning

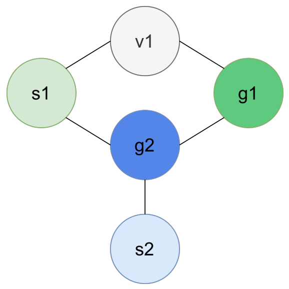

Fig. 1. Example instance with start positions *s1* and *s2*, and goal positions *g1* and *g2*, respectively.

they introduce distinct atoms. Consider the instance in figure 1. Agent 1 starts at vertex s1 and aims to reach g1, while Agent 2 starts at s2 and aims for g2. The instance is described by the following facts:

```
agent(1;2). start(1,s1). start(2,s2). goal(1,g1). goal(2,g2).
starting_horizon(1,2). starting_horizon(2,1).
vertex(s1;g1;g2;g2;v1).
edge(s1,v1). edge(v1,g1). edge(s2,g2).
edge(s1,g2). edge(g2,g1).
reach(1,s1,0). reach(1,v1,1). reach(1,g1,2).
reach(2,s2,0). reach(2,g2,1).
```

A typical incremental version of the previous encoding would consider the full graph for every agent from the beginning. For example, Agent 1 could reach its goal at timepoint 2 from two different vertices. The ground instances of the rule in Line 5 of the encoding in Listing 1.1 would look as follows:

```
at(1,g1,2) :- move(1,v1,g1,2).
at(1,g1,2) :- move(1,g2,g1,2).
```

After applying completion and converting to nogoods, we get the following:

```
{T_at(1,g1,2), F_move(1,v1,g1,2), F_move(1,g2,g1,2)}
{F_at(1,g1,2), T_move(1,v1,g1,2)}
{F_at(1,g1,2), T_move(1,g2,g1,2)}
```

These nogoods state that it is not possible for Agent 1 to be at its goal location at timepoint 2 unless the necessary moves have been performed. Additionally, if Agent 1 moves to its goal, it must be there. However, the encoding described earlier, by utilizing the `reach/3` predicate, would only generate the following rule:

```
at(1,g1,2) :- move(1,v1,g1,2).
```

with its corresponding nogoods:

```
{T_at(1,g1,2), F_move(1,v1,g1,2)}
{F_at(1,g1,2), T_move(1,v1,g1,2)}
```

It is easy to see that when vertex g2 becomes available to Agent 1, the nogoods generated by the new rule would cause a conflict. If one of the moves is selected, one set of nogoods would propagate that the agent is at its goal, while another would indicate that the agent cannot be at its goal. In fact, attempting this in Clingo results in an error.

This issue arises when new rules are added to the program with an existing atom in the head. In our context, it occurs when the grounder does not consider all vertices in the graph for a given time point simultaneously. As a result, when a new way to reach an existing position is introduced, it leads to the conflict described earlier.

This pattern is seen in state-of-the-art MAPF techniques. For example, in [7], they restrict the graph to a subset of vertices. If a solution cannot be found on this sub-graph, additional vertices are included in the next grounding attempt.

For these reasons, a one-shot solving approach is typically used. However, this results in most of the information being grounded multiple times. To address this, we propose an encoding that enables incremental solving. This encoding incorporates the techniques described in [5], along with new methods, to effectively modularize the MAPF encoding.

4 Incremental encoding

In this section, we will examine the individual components of the single-shot encoding and explain how we transform them for incremental solving.

We refer to the atoms and rules grounded in a given step as a *layer*. To represent the positions an agent can occupy at a specific time point within a layer, we introduce the predicate `reach/4` (given as a fact before grounding a layer), replacing the previous `reach/3`. The last argument in this predicate corresponds to the layer number (denoted as k in the encoding). These predicates are central to the incremental encoding, as they help determine which rules need to be grounded.

Using the `reach/4` predicate, we generate the predicate `r_edge/5`, which indicates the edges an agent can traverse at a given time point within a specific layer. Again, the last argument is the layer number. The predicate `reachk/3` accumulates all instances of `reach/4` across all layers. Finally, the predicate `newtime/3` indicates that a new time point has been introduced in the given layer.

We begin by examining the rule in Line 3 of Listing 1.1, which denotes the choice of moves. The transformation is as follows:

```
{ move(A,U,V,T) : r_edge(A,U,V,T,k)} 1 :- time(A,T).
```

This rule is almost identical to the one in the single-shot encoding, except that it replaces the `reach/3` predicates with the `r_edge/5` predicate. This ensures that we only select moves that are available in the current layer. However, a closer examination reveals that this transformation is not sufficient. The rule only enforces that at most one move is chosen for the current layer. This means that once additional moves become available in future layers, it would be possible to select one move from each layer, potentially resulting in more than one move for the agent. To address this, we add the following rules to ensure that only one move is chosen across all layers:

```
moved(A,T,k) :- move(A,U,V,T), r_edge(A,U,V,T,k).
forward_move(A,T,k) :- moved(A,T,k).
forward_move(A,T,k) :- forward_move(A,T,k-1).
:- forward_move(A,T,k-1), moved(A,T,k).
```

In the first rule, the predicate `moved/3` becomes true if a move is selected in the current layer. In the second rule, we introduce the predicate `forward_move/3` to propagate the fact that a move is chosen in the current layer to the next layer. The third rule propagates the fact that a move was selected in a previous layer to the current one. Finally, the last rule ensures that only one layer can have a move selected at any given time. This construction is based on the approach described in [5].

Now, we examine the rule in Line 9 of Listing 1.1, which ensures that an agent has exactly one position per time point. Here, we employ a similar approach as with the moves. An "exactly one" condition can be viewed as an "at-most-one" and an "at-least-one" aggregate working together. The transformation is as follows:

```
:- { at(A,V,T) : reach(A,V,T,k) } > 1, time(A,T).
has_position(A,T,k) :- at(A,V,T), reach(A,V,T,k).
forward_has_position(A,T,k) :- has_position(A,T,k).
forward_has_position(A,T,k) :- forward_has_position(A,T,k-1).
:- forward_has_position(A,T,k-1), has_position(A,T,k), time(A,T).

backward_has_position(A,T,k) :- has_position(A,T,k).
backward_has_position(A,T,k) :- backward_has_position(A,T,k+1).
#external backward_has_position(A,T,k+1): time(A,T). [false]

#external backward_has_position(A,T,k) : newtime(A,T,k). [false]
:- not backward_has_position(A,T,k), newtime(A,T,k).
```

The first rule enforces the "at-most-one" condition within the given layer. The following four rules handle the propagation of this condition across layers, similar to how we managed the moves.

The subsequent three rules begin encoding the "at-least-one" part of the constraint. In this case, the propagation works from the current layer down to the first layer. The final two rules set up the base layer, ensuring that at least one position is assigned to the agent in the initial layer. These rules are grounded once, when a new timepoint is introduced.

Next, we transform the rules in Lines 5 and 6 of Listing 1.1 that move the agent as follows:

```
1  #external move(A,U,V,T):edge(U,V), not r_edge(A,U,V,T,k), reach(A,U,T-1,k).
2  #external move(A,U,V,T):edge(U,V), not r_edge(A,U,V,T,k), reach(A,V,T,k).
3  at(A,V,T) :- move(A,U,V,T), reach(A,V,T,k).
4  at(A,V,T) :- at(A,V,T-1), not moved_from(A,V,T), time(A,T), reach(A,V,T,k).
5  moved_from(A,U,T) :- move(A,U,V,T), reach(A,U,T-1,k).
6  :- moved_from(A,U,T), not at(A,U,T-1), reach(A,U,T-1,k).
```

This transformation leverages the fact that there are at most four possible moves to and from a given location. The first two lines introduce external atoms for moves that are not yet possible. In Lines 3 and 4, we use those moves, along with the choices made in the move selection rule, to update the position of the agent. Essentially, this process pre-grounds all possible incoming and outgoing moves for a given location. Since unavailable moves are defined as external atoms, they default to false. The rules using the external atoms will not trigger unless the move is made possible in the future and a corresponding choice rule is grounded for it. This approach ensures that moves are only considered when they are valid, respecting the incremental nature of the encoding. The rule in Line 5 projects away the end point of the move. The atom is then used in Line 6 to ensure that the agent stats the move at the correct location.

Next, we address the vertex constraint in Line 11 of Listing 1.1:

```
:- at(A,V,T), at(B,V,T), A != B, reach(A,V,T,k), reachk(B,V,T).
```

Here, we utilize the `reachk` predicate to determine which vertices are available for a given agent in the current layer. This allows us to ground the constraint only when an agent-vertex pair introduced in the current layer conflicts with another agent. This ensures that the constraint is only grounded when a potential collision between agents can actually occurs. A similar construct is used for the edge constraint in Line 12 of Listing 1.1, ensuring that edge conflicts are also handled efficiently by only grounding the constraint when agents can actually collide on a specific edge.

Finally, we examine the rules that handle the penalties for sum-of-cost optimal solutions. The transformation follows the method outlined in [5], with a few modifications. Below is an intuitive explanation of how we transform these rules. Line 2 of Listing 1.2 is transformed as follows:

```
penalty(A,0..N-1) :- starting_horizon(A,N).
```

Since these penalties are facts, we ground them once in the base program.

For line 3, we use the following transformation:

```
penalty(A,T)   :- penalty(A,T,k), newtime(A,T,k), starting_horizon(A,N),
                                                               T<N+delta.
penalty(A,T,k) :- starting_horizon(A,N), T>=N, at(A,U,T), not goal(A,U),
                                                           reach(A,U,T,k).
penalty(A,T,k) :- penalty(A,T,k+1).
#external penalty(A,T,k+1) : starting_horizon(A,N), time(A,T), T>=N.

penalty(A,N+delta) :- penalty(A,N+delta,k+1), starting_horizon(A,N),
                                                      newtime(A,_,k).
```

First, we look at the second rule which is mostly a direct transformation from the single-shot verion. The main difference is in the last argument that denotes in which layer this rule is grounded. Again, by using the reach predicate, we only ground the rule for the vertices introduced in the current layer. The first rule describes that whenever this atom is true, the atom without the layer argument is also true. Notice that this rule is also only grounded once, on the layer where the time point is introduced. The third rule simply propagates the penalty from future layers to the lowest layer, where it can then be used in the first rule. In short, once an atom with a layer argument is true, it will propagate to the previous layers until it reaches the first one. Then, it will propagate the atom without the layer argument.

In the fourth rule, we introduce an external atom for the future layer atom so that is it not simplified away while grounding. Finally, the last rule in the listing exists due to a peculiarity with the reach predicate. Since we rely on the fact the agent can only be at the goal, at the last time point there will be no grounded instances of the second rule. Hence, there will also be no rule instances for rule 1. The last rule in the listing makes sure that a similar rule is grounded for the last time point.

Finally, the rule in Line 4 of Listing 1.2 is transformed as follows:

```
#external penalty(A,N+delta+1) : starting_horizon(A,N), newtime(A,_,k).
penalty(A,T) :- penalty(A,T+1), newtime(A,T,k), starting_horizon(A,N).
```

The first rule defines an external predicate for the penalty at a future time point, ensuring that the penalty in the next rule is not simplified away. The second rule propagates the penalty from one time step to the previous one, making sure to only do this for new time points.

The goal constraint in Line 13 of Listing 1.1 and the constraint enforcing a certain number of extra moves in Line 7 of Listing 1.2 are deleted and encoded anew every grounding step.

5 Experiments

The rule transformation described in the previous section is but one way to modularize the encoding. We consider the following encodings:

- *Alt-0* is the encoding as described in section 4.
- *Alt-1* encodes inertia by way of a choice rule and embeds the precondition of the move in the choice rule.
- *Alt-2* does not enforce the amount of moves an agent can have, relying on the fact that an agent has exactly 1 position to stay consistent.

To compare the performance of the incremental and one-shot solvers, we run benchmarks on the set of instances from [8]. They consists of instances with type *empty*, *room*, *maze*, and *random* of sizes ranging from 16×16 to 128×128. All maps start with 5 agents and the number of agents increases by 5 up to 100 agents.

We expect the incremental solver to perform worse on smaller instances and better on larger ones. Smaller instances have low grounding times, so the increased grounding efficiency is unlikely to offset the overhead of incremental solving. However, for larger instances, the overhead of grounding the same information multiple times is significant, and the incremental solver should outperform the one-shot solver.

Preliminary results show that the *Alt-0* and *Alt-2* variants slightly improve on the one-shot solver, solving 20 and 30 more instances, respectively. The *Alt-1* variant is the best overall, solving about 60 more instances. It also shows improvement on smaller instances, unlike the other variants. Unexpectedly, for all encodings, the incremental solver performs worse on the largest instances, where grounding is expected to be most expensive.

Size	**Instances**	**One-shot**	**Alt-0**	**Alt-1**	**Alt-2**
16	(800)	263	263	**280**	265
32	(800)	331	341	**353**	340
64	(800)	412	428	**436**	431
128	(800)	352	346	**351**	352
Total	(3200)	1358	1378	**1420**	1388

6 Conclusion

In this work, we outlined a method to transform a single-shot ASP encoding into an incremental one. We ran experiments to compare their performance and found that the incremental solver performs better than the one-shot solver on smaller instances. For larger instances, the incremental solver is slightly worse than the one-shot solver.

For future work, we will focus on improving the encoding to speed up grounding time and explore ways to reduce the overhead of incrementally grounding the problem. Additionally, we will run more experiments to confirm the results.

References

[1] 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.

[2] 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`.

[3] M. Gebser, B. Kaufmann, and T. Schaub. "Conflict-Driven Answer Set Solving: From Theory to Practice". In: *Artificial Intelligence* 187-188 (2012), pp. 52–89. DOI: `10.1016/j.artint.2012.04.001`.

[4] 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`.

[5] M. Gebser et al. "ASP Solving for Expanding Universes". In: *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. Ed. by F. Calimeri, G. Ianni, and M. Truszczyński. Vol. 9345. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2015, pp. 354–367.

[6] M. Gebser et al. "Engineering an Incremental ASP Solver". In: *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Ed. by M. Garcia de la Banda and E. Pontelli. Vol. 5366. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 190–205.

[7] 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`.

[8] 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. Jaap van den Herik. SciTePress, 2024, pp. 264–271. DOI: `10.5220/0012353500003636`. URL: `https://www.scitepress.org/ProceedingsDetails.aspx?ID=7POrHKUPtlI=`.

[9] 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.

[10] P. Surynek et al. “Migrating Techniques from Search-based Multi-Agent Path Finding Solvers to SAT-based Approach”. In: *Journal of Artificial Intelligence Research* 73 (2022), pp. 553–618. DOI: `10.1613/jair.1.13318`.