University of Potsdam

Final Thesis for a Bachelor's Degree in Computational Science

Map Abstraction for Multi-Agent Pathfinding problems with Answer Set Programming

Adrian Salewsky

supervised by

Prof Torsten Schaub

Arvid Becker

01.04.2022

Abstract. Multi-Agent Pathfinding stellt ein wichtiges Problem in der Informatik dar. Folglich wird versucht, die Geschwindigkeit von Lösungsalgorithmen ständig zu verbessern. In dieser Arbeit stelle ich meine Methode zur Kartenabstraktion für solche Probleme dar und zeige auf, wie diese genutzt werden können, um entsprechende Probleme schneller zu lösen. Es wurde die Programmiersprache Answer Set Programming genutzt, um die Abstraktion zu bilden. Weiterhin wurde Python verwendet, um verschiedene Hilfsprogramme zu erstellen. Als Vergleichswert für die Lösungszeit wurde das Framework "asprilo" genutzt, das ebenfalls die Sprache Answer Set Programming nutzt. Die Arbeit wurde in englischer Sprache verfasst.

1 Introduction

Multi-Agent Pathfinding (MAPF) is an important problem in computer science. Therefore, it is important to look for ways to improve the solving time of such problems. One way to do that, is to abstract the instance that has to be solved to reduce the size of possible plans. In this paper, I will show how I built such abstractions in the programming language Answer Set Programming (ASP). In the next two sections, I will explain what MAPF problems are and how the language ASP works. After that, I will introduce the "asprilo" framework which is used as base for the enoding and also serves as comparison for the solving time. Then, I will explain how my abstraction methods work and how the results of these abstractions look. Following that, I will show the benchmarking results of the methods. I will end the paper with a conclusion of my work.
All the used programs and instances can be found in my github repository [1]. This paper was written as final thesis for a Bachelor's degree in the course Computational Science at the university of Potsdam. This project was done in cooperation with Tarek Ramadan who worked on using the abstractions to solve a problem as a project for his final thesis in for a Bachelor's degree in the same course and university. His work can be found on his own github repository [2].

2 Multi-Agent Pathfinding

In Multi-Agent Pathfinding problems you have a set of nodes, agents (e.g. robots), start and goal positions as well as possible moves. The set of nodes declare the possible positions where robots can be and move to. The robots can move from one node to accessible neighboring nodes in discrete time steps. In this paper only orthogonal moves are considered. The goal is to let each robot find a way from its start to its goal position. It is possible to not have predetermined goal positions for each specific robot and instead let the solver assign the goal positions to the robots that can have the best way to that position. In this project, I only dealt with having a predetermined goal position for each robot. The result of the solving process is a plan with a set of moves for each robot. When executing the plan there must not be any collisions, i.e. two robots going through each other or two robots moving to the same node (INSERT PICTURE). The last time step of the plan is called horizon or makespan INSERT SOURCE FOR MAPF.

3 Answer Set Programming

Answer Set Programming is a declarative programming language. It contains a set of rules and atoms. The result is a stable model that is built upon these rules and atoms. An atom has the form:

```
a.
```

This means that the *a* is treated as a fact and always has to be true. Rules have the form:

```
b :- a.
```

which means that if *a* is true then *b* also has to be true. There are also negations:

```
b :- not a.
```

This means that *b* can only be inferred if *a* is never true in the model. There are also choice rules:

```
x {b,c,d} y :- a.
```

The atom *a* is used to infer the choice rule. The solver can then decide which of the atoms *b*, *c* and *d* he wants to infer. He can also decide to not choose no atom at all. You can set bounds for the amount of atoms chosen. In this case it means that the solver has to at least choose *x* atoms and cannot choose more than *y* atoms. You can also influence the chosen atoms by using constraint rules of the form:

```
:- a, b.
```

This means that *a* and *b* cannot be true at the same time. Combining this constraint rule with the choice rule above means that the solver will not choose to infer *b*. Atoms can also have attributes:

```
a(1). a(2).
```

This means that there is an atom *a* with the attribute 1 and an atom *a* with the attribute 2. In rules you can exchange the attributes with variables:

```
b(X) : a(X).
```

This means that the solver looks for every atom *a* with exactly one attribute and infers the atom *b* with the same attribute. The last form of rule to talk about are aggregates. I will explain two aggregates that have also been used in the abstraction encoding. The first one is *min*:

```
a(MIN) :- MIN == #min{X: b(X)}.
```

The solver looks for the attribute of the atom *b* with the lowest value and assigns MIN this lowest value. It then infers the atom *a* with the attribute MIN. SOURCE FOR ASP, CLINGO TALK ABOUT CLINGO WRITE ABOUT ADVANTAGE OF ASP

4 Asprilo

5 Abstraction Methods

I have developed several abstraction methods. All of them use some of the asprilo encoding as a base. Since the input instances look the same, I use a reduced version of the *input.lp* file to transform it into predicates, that are easier to understand. It is a reduced version because I do not care about orders, products or packing stations. The only relevant things are robots, shelves and nodes.
The goal condition is the same as in the modified asprilo encoding.
The output predicates are *new_init* and *imp_position*. The *new_init* predicates describe the new instance with a reduced node count. The form of *new_init* resembles that of the *init* predicate in the input instance. There are *new_init* predicates for robots, shelves and nodes. The ones for robots and shelves are the same as in the input instance (with of course a "new" being added) since the start and goal conditions are not changed during the abstraction. The predicates for the nodes contain only the nodes that still exist after the abstraction. The *imp_position*(R, C) predicate states that robot R is on cell C at some point while building the abstraction. This can be used as extra information when solving the instance with the use of this abstraction. The output is the same for every method except for the "Reachable Nodes" method. In that method the output is slightly different which I will explain in the respective subsection.
The first part of the abstraction building is always solving the instance without caring for conflicts. Therefore the encoding takes parts of the original asprilo encoding again. The difference is that the constraint rules which prohibit conflicts are ignored. Furthermore there are two lines extra:

```
:- move(R,_,T1), not move(R,_,T2), time(T2), isRobot(robot(R)), T2<T1.
#minimize {1,(R,T): move(R,_,T)}.
```

The first line states that a robot must always move until he is finished moving completely. This makes it so that a robot does not randomly wait instead of directly going to his goal.
The second line is used to minimize the amount of moves each robot does. This is used to ensure that the solver looks for the shortest paths of the robots.
In the following subsections, I will explain the idea behind each abstraction method and show the encoding that is only used in that method respectively.

5.1 Shortest Path

The first method is to look for the shortest path for each robot to its respective goal without considering possible conflicts as was explained in the section above. Each position the robot visits is then used to create the *imp_position* predicates.

```
imp_position(R,C) :- position(R,C,_), isRobot(robot(R)).
```

In this method, it is also possible to add the time step at which the robot visits the position as an argument to the *imp_position* predicate. However, this would mean that a solver that uses the abstraction to solve the instance would need to specifically be designed for the "Shortest Path" method. Always using the same structure for each method means that one solver can be created to work for all methods. It is also not possible to look at how many arguments the predicate has because the "Reachable Nodes" method needs a third argument for something different, which I will explain in the respective subsection. If it is not of interest to have a solver that solves by using the different abstractions the same way, the line shown above can easily be modified to contain the information about the time step. The output would then have to be modified as well.
Of the methods explained in this paper, this is the one that results in the smallest abstraction. It could also be viewed as a special case of the other methods. If there is a shortest path in an instance, this method will find it so it can be used for all solvable MAPF problems.

5.2 Node Combining

5.3 Reachable Nodes

6 Benchmarking

7 Conclusion

References

1. My Github repository

   `https://github.com/salewsky/MAPF-Project`

2. Github repository of Tarek Ramadan

   `https://github.com/tramadan-up/mapf-ba`

A Affidavit

I hereby affirm that this Bachelor’s Thesis represents my own written work and that I have used no source and aids other than those indicated. All passages quoted from publications or paraphrased from these sources are properly cited and attributed.

Date, Place	Signature