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. I will also briefly skim over some additional programs that provided support for the solving. 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 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 declares 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 that is especially useful when using it for NP-hard problems.For a complete introduction refer to INSERT ASP SOURCE. In this section, I will only shortly explain the main parts which are relevant for my encoding.

An ASP program 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. Before the solver looks for stable models, the program is *grounded*. During this process, the variables in the program are exchanged with the actual values.
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. The other aggregate, I used in my encoding is *minimize*:

```
#minimize {X : a(X)}.
```

This means that the solver looks at every atom a with one attribute and builds the sum of these attributes. It then looks for the model with the lowest sum.
There are also other aggregates which I have not explained but since I do not use them in my encoding, I will not explain them. An ASP program has the file ending *.lp*.
To ground and solve the programs in this paper, clingo-5.4.0 was used. INSERT SOURCE CLINGO

4 Asprilo

The asprilo framework is an ASP-based framework that is used to create plans for robots in a warehouse situation. It was created by Potassco. There are multiple domains in the framework. The domain that is important for this paper is the M-domain where only simple robot movement is important which turns the problem into a MAPF-problem. In the M-domain there are robots and shelves. Both have predetermined starting positions. The robots are able to move orthogonally. The goal is that each shelf is occupied by a robot when the time horizon is reached. Other domains also consider having the robots move the shelves around or having a robot battery but this is not important when dealing with simple MAPF-problems. The asprilo encoding serves as a base for my own encoding which is why I will explain the different parts of the encoding that are relevant for my programs in the following subsections. The encoding and a complete guide can be found on the website by Potassco [3].

4.1 Instances

The first thing to talk about is how the input instances look. Since the M-domain only needs nodes, robots and shelves, these only these are initiated:

```
init(object(node,N),value(at,(X,Y))).
init(object(robot,R),value(at,(X,Y))).
init(object(shelf,S),value(at,(X,Y))).
```

The first line is used to initiate a node with the ID N at the position X, Y. X and Y are the coordinates for a 2-dimensional grid. The second and third line are used to initiate a robot with the ID R and a shelf with the ID S respectively. The IDs are usually integers so the first robot, shelf or node would usually get the ID 1, the second gets the ID 2 and so on.
That is all that needs to be initiated for the M-domain. If a node between two other nodes is not initiated, it is treated as a wall.

4.2 Input

The program *input.lp* is used to turn the *init*-atoms into others that are easier to work with:

```
robot(R)  :- init(object(robot,R),_).
isRobot(robot(R))  :- robot(R).
position((X,Y))    :- init(object(node,_),value(at,(X,Y))).
position(R,(X,Y),0)  :- init(object(robot,R),value(at,(X,Y))).
position(shelf(S),(X,Y),0) :- init(object(shelf,S),value(at,(X,Y))).
```

The first line creates the *robot*-atom with simply states that the robot with ID
R exists. This is then used to create the atom *isRobot* which can be used to
check if an object is a robot. These two lines also exist for shelves.
The next three lines create the *position*-atoms. The first one takes just the
information about the coordinates from the node initiation. The second one
says that robot R is on the position it is initiated on, i.e. his starting position,
at time step 0 which is the first time step. The third one states that the shelf
with ID S is on its starting position, which then serves as a goal position, at
time step 0. The shelves are not moved in the M-domain which is why the shelf
position could have also been initiated without the time step but this way it is
easier to change it to a higher domain. It is also important to note that instead
of having just the ID as first argument, it has $shelf(S)$. This makes it easier to
distinguish between shelves and robots when using *position*.

4.3   Action

The program *action.lp* is the one where moves are created. It is the biggest
subprogram. First off, the time array is created:

```
time(1..horizon).
```

This line creates a *time*-atom with the arguments being rising integers starting
from one until the constant *horizon* is reached. This constant has to be given
externally to the program.
Then the possible directions are created:

```
direction((X,Y)) :- X=-1..1, Y=-1..1, |X+Y|=1.
nextto((X,Y),(DX,DY),(X',Y')) :- direction((DX,DY)), position((X,Y)), position((X',Y')),
                                  (X,Y)=(X'-DX,Y'-DY), (X',Y')=(X+DX,Y+DY).
```

The first rule initiates the four orthogonal movements as possible directions. The
second rule states that a node is next to another one if it can be reached by going
in one of the possible directions.
The next rule creates the actual *move*-atoms:

```
{ move(R,D,T) : direction(D) } 1 :- isRobot(robot(R)), time(T).
```

This rule states that for each robot at each time step up to one move is created
into one possible direction. It is also possible to not create a move at certain
time step. Not moving is an action equivalent to waiting. These moves have to
change the positions of robots accordingly:

```
position(R,C,T) :- move(R,D,T), position(R,C',T-1),    nextto(C',D,C).
                :- move(R,D,T), position(R,C ,T-1), not nextto(C ,D,_).
position(R,C,T) :- position(R,C,T-1), not move(R,_,T), isRobot(robot(R)), time(T).
```

The first rule changes the position of the robot according to the move taken. The second rule states that a robot cannot move to a position that does not exist. The third rule states that if a robot has not moved at a certain time step, its position does not change at that step.
There are more rules that prohibit vertex and edge conflicts. These rules are not important for my own encoding which is why I will not explain them further.

4.4 Goal

The goal condition of the normal M-domain states that at the last possible time step, i.e. horizon, each shelf must be occupied by a robot. For this project the goal condition was slightly altered:

```
goal(R,C) :- isRobot(robot(R)), position(shelf(S),C,0), R=S.
:- isRobot(robot(R)), position(R,C,horizon), goal(R,C'), C!=C'.
```

The first rule creates the *goal*-atom. It says that the goal of robot R is the cell C (cell is synonymous with node). The goal cell is the cell where the shelf with the same ID as the robot is initiated on. This means that the solver cannot decide which robot should go to which shelf but it is predetermined. The second rule states that a robot cannot be on a cell that is not its goal at the last time step.

4.5 Output and Encoding

The file *output.lp* determines how the output of the program looks. I will not show the encoding since the output of my encoding is a different one. It is enough to know that the output of the asprilo encoding is a plan, i.e. atoms that state how each robot moves at each time step.
The last file *encoding.lp* includes the other files important for solving (it does not include the input instance). This means it contains the complete solving program. My own encoding follows the same file structure which I will show in the next section.

4.6 Visualizer

The visualizer is an extra component provided by asprilo. It can be used to visualize instances and the movement of robots. All images in this paper showing instances and robots have been created with this visualizer.

5   Abstraction Methods

I have developed several abstraction methods. All of them use some of the asprilo encoding as a base. The files *input.lp* and *goal.lp* are the same as in the modified asprilo encoding.

The output atoms are *new_init* and *imp_position*. The *new_init* atoms describe the new instance with a reduced node count. The form of *new_init* resembles that of the *init* atoms in the input instance. There are *new_init* atoms 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 atoms for the nodes contain only the nodes that still exist after the abstraction. The *imp_position*(R, C) atom 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* atoms.

```
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* atom. 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 atom 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.

IMAGE OF INSTANCE BEFORE AND AFTER ABSTRACTION

5.2 Node Combining

The idea for the node combining was taken from two other papers looking for abstractions (CITE PAPERS).The approach had to be slightly altered because my goal is to only build an abstraction and not solve the instance completely. The idea is two combine several nodes together into one node. Afterwards, the solver looks for the shortest path in the combined instance. The solver then outputs all the nodes that have been visited in the combined form (SHOW IMAGES TO EXPLAIN). The combining of nodes looks like this:

```
comb_position((X,Y),(X',Y'))    :- position((X,Y)),
        X' = (X+(size_x-1))/size_x, Y' = (Y+(size_y-1))/size_y.
comb_position(C)                :- comb_position(_,C).
comb_position(R,C',0)           :- position(R,C,0), comb_position(C,C').
comb_position(shelf(S),C',0)    :- position(shelf(S),C,0), comb_position(C,C').
```

The first rule calculates the coordinates that a node has after the combining. The constants $size\_x$ and $size\_y$ say how many nodes in x- and y-direction are combined together. The second rule is used so that it is not necessary to always care about the precombining nodes. Rule 3 and 4 change the positions of robots and shelves to their new coordinates. The rest of the encoding is only changed so that the atoms *comb_position* instead of *position* is used. The output atom *imp_position* is then easily inferred by looking at the positions visited by robots:

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

It takes the positions visited by robots, looks at the nodes that were combined into that position and then infers the respective atoms.

Simply using this approach has the problem that walls are completely ignored while building the combined instance. This means that the encoding introduced so far can only be used for open instances, i.e. instances that do not have any walls except for the outer walls. This is why I call this approach "Open Node Combining". The advantage of this method is that the combining size can be freely chosen, as long as positive integers are used.

To also use the node combining approach in instances with walls, I created the approach "Complete Node Combining". This approach mostly works the same

but it can only be used when combining 2x2 squares together. The difference is that a *wall*-atom is introduced so that walls are considered after the combining:

```
wall((X1',Y1'),(X2',Y2')) :- comb_position((X1',Y1')), x_arr(X1), y_arr(Y1),
    not position((X1,Y1)), not position((X1+1,Y1)),
    X1' = (X1+(size_x-1))/size_x, Y1' = (Y1+(size_y-1))/size_y,
    X1'-1 = X1/size_x, Y1'-1 = Y1/size_y,
    comb_position((X2',Y2')), X1'=X2', Y1'-1=Y2'.
```

This is one of four rules inferring a *wall*-atom. The idea is that in a 2x2 square there are four possible walls: two nodes missing to the top, bottom, left or right. Each rule inferring a *wall*-atom is used for one of the possible walls. The rule takes a combined node and checks if there are exactly two uncombined nodes missing that would be in that combined node if they existed. The *nextto*-atom is then extended to also check if a wall is between two combined nodes before inferring the respective atom:

```
nextto((X,Y),(DX,DY),(X',Y')) :- dir((DX,DY)),
    comb_position((X,Y)), comb_position((X',Y')),
    (X,Y)=(X'-DX,Y'-DY), (X',Y')=(X+DX,Y+DY),
    not wall((X,Y),(X',Y')), not wall((X',Y'),(X,Y)).
```

As mentioned before, this way of introducing walls leads to only being able to combine 2x2 squares. I did not find a way to combine bigger sizes because looking at walls in these cases gets much more complicated.
ADD SOME IMAGES

5.3   Reachable Nodes

The last abstraction method was also inspired by a research paper (CITE PAPER). Again, a slight alteration was done because the goal is to only find an abstraction and not a complete solution. The idea is to find the shortest path for each robot as always. Then, the solver looks at how many steps each node close to the path of the robots is deviating from that path. The path itself is deviating by zero steps. Each node next to the path is deviating by one step, the nodes right next to them by 2 steps and so on (USE IMAGE FROM PAPER). The maximum amount of steps looked at is the amount of steps the robot needed in total to reach its goal.

```
makespan(R,N) :- move(R,_,N).
imp_pos(R,C,0)  :- isRobot(robot(R)), position(R,C,_).
imp_pos(R,C,N)  :- isRobot(robot(R)), imp_pos(R,C',N-1), makespan(R,N), nextto(C',_,C).
imp_position(R,C,MIN)  :- imp_pos(R,C,MIN), MIN == #min{X: imp_pos(R,C,X)}.
```

The first line looks at how many steps each robot takes. It is important to note that makespan refers to an individual robot where as horizon refers to all robots. This means that the highest makespan is equal to horizon. Another point to note is that the *makespan*-atom not only has the highest time step for an individual

robot but each step it takes infers a *makespan*-atom. The second line initializes the path of the robot itself as deviating by zero steps. The third line then infers the next set of positions and the respective deviation until the makespan of the robot is reached. This way of inferring these atoms leads to having multiple *imp_pos*-atoms for a single node. The last line makes sure that only the smallest deviation gets taken for the *imp_position*-atoms. This time the *imp_position*-atoms as well as the *new_init*-atoms have three attributes because the amount of deviating steps is important to output. The abstraction this time does not simply consist of a set of nodes but instead multiple sets of nodes with a different distance to the shortest path. A solver using this abstraction could then decide how big the distance should be and only take a subset of the nodes provided to solve an instance.

There was also a slightly different approach which did not look at all the nodes up to the complete makespan of a robot. Instead, it only looked at how many steps were still left to reach the goal and calculated the deviation up to these remaining steps. This method provides less information but the hope was that it would find the solution faster. During the functionality testing it already became clear that the solving time with this different approach was slower than the original approach. This is why this different approach was discarded during the creation process.

6 Auxiliary Programs

Apart from the ASP encoding, I have created several python programs which helped with different parts of the project. I will only briefly explain them because they are not as relevant to the project as the ASP part.

6.1 Map Generator

The map generator is used to create instances. The program can generate a map with a specified size in x- and y-direction. It also creates a specified amount of robots on random starting positions and the same amount of shelves which also have random starting positions. It can only create open maps which means that walls have to be edited in by hand. Specifying starting positions for robots or shelves is also impossible. The resulting file has a head in comment form which shows the chosen parameters, i.e. amount of nodes, robots and so on. After that, all the nodes are initiated. Following that, each robot is initiated together with its shelf:

```
% Robot N
init(object(robot,N),value(at,(X1,Y1))).
init(object(shelf,N),value(at,(X2,Y2))).
```

This way, controlling starting and goal position of a specific robot is easier when editing that is required. The result file is then called *instance.lp*.

6.2   Solvers

There are multiple solver programs. The simplest one is called $hor_inc.py$. It can be used to solve a normal instance. It takes the solving ASP program and an input instance as input arguments. It then looks for the highest Manhattan distance between a robot and its goal position. This value is then taken as the constant *horizon* for the ASP program. The program tries to solve the instance with this input. If the instance can be solved, it prints the solution and is finished. If the instance is considered unsatisfiable by clingo, it increases the *horizon* by one at tries to solves it again. This is repeated until a solution is found or the horizon gets bigger than the amount of nodes in the instance. The second case would mean that the instance is not solvable at all for an abstraction because the robot could get to any position on the map it can reach. Another possibility for the program to end is if it takes longer than a specified time threshold. If it takes too long to solve and the threshold is reached, the python program stops the solving and "Timeout" is printed. This solver was used to see if the instances used for testing the functionality of the abstraction methods. The other solvers were used for benchmarking the different methods. They are mostly the same as $hor_inc.py$ with some extras. The exact usage will be discussed in the benchmarking section.

6.3   Result Plotters

There are two more python programs which I have created. Both are used to plot the benchmarking results. The first program, $lineg_results.py$, takes the results for different methods for a specific instance, builds the average of these results and plots a line graph for each method. To compare it to asprilo , the results for the asprilo benchmarking also needs to be input as a method. The problem with this method is that the results can be very varying which makes the resulting average very strange at times. This is why the second program, $scatter_results.py$, will be used to graphically show the results of the benchmarking. This program takes the different abstraction methods and directly compares it to the asprilo encoding. It creates a scatter plot which means that it is not necessary to calculate the average. This makes the analysis of the graphic much easier. How exactly the resulting graphic looks and how it is supposed to be interpreted will be discussed in the benchmarking section as well.

7   Benchmarking

8   Conclusion

12

References

1. My Github repository

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

2. Github repository of Tarek Ramadan

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

3. Asprilo website

   `https://potassco.org/asprilo/`

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