Multi Agent Pathfinding - Individual Plan Merger with ASP

Marius Wawerek

Potsdam University

Abstract. In this paper I will present a simple overview of a "Praktikum" offered by the "Lehrstuhl" knowledge representation and reasoning of the University of Potsdam, dealing with the multi agent pathfinding problem (MAPF). As part of this module both Niklas Krämer and I created multiple MAPF solvers in clingo. We also wrote various supporting programs and scripts to create, collect and visualize all of our data. All will be explained briefly. You can find all of the things mentioned on our Github Page. [1]

1 Introduction

Logistics is a complex science on it's own. One part of logistics is the multi-agent-pathfinding problem (MAPF). Amongst other things you can find this problem in a typical warehouse setting.

We will start by formally introducing the problem setting we are dealing with. Asprilo [1] is used to represent our problem domain in Answer Set Programming (ASP). The basics will be explained, but we assume the reader has at least some experience with Asprilo and Answer Set Programming in general. We then continue with the first own contributions. To support our MAPF solvers, we created various programs. Each will be briefly introduced in the section 'Our Scripts'. Moving on, we will present most of our MAPF solvers. Due to limitations in the amount of pages, we cannot showcase all of them. Nevertheless every solver can be found our Github Page. [2] The section after this deals with the experiments we did on all of our instances. Using our written programs, highlights of the data will be presented in various plots. Finally we will have a concluding remark.

2 Problem Setting

As part of our "Praktikum", we were tasked to deal with the multi agent pathfinding problem (MAPF). However we will first take a step back and explain everything from the ground up. The general problem is known as "pathfinding". As the name implies, the main challenge is finding a path from a starting position to a goal in a given environment. Formally the problem is defined as: given

[1] https://github.com/NikKaem/mapf-project
[2] https://github.com/NikKaem/mapf-project

a Graph consisting of vertices and edges, given a starting position and given one or more goal positions. We want to find a path from the starting position to a goal. A path is a sequence of vertices that are connected by edges. This path is taken by a so called 'agent'. This is the entity actually moving throughout the environment. Each edge in the underlying graph of the environment is represented as a 'action'. This is known as a "single agent pathfinding problem" (SAPF), as a single agent is taking all of the actions. Let us illustrate all of this with an example. In figure 1 you can see a visualization of a SAPF problem.

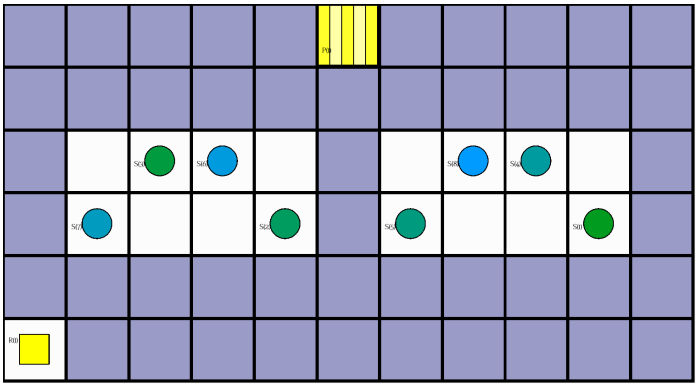

Fig. 1. Single Agent Pathfinding

Overall you can see multiple squares or so called 'nodes' in either blue or white. Each is uniquely identified by its coordinates. For example the top lefthand node is the node$(1, 1)$. The node in the bottom right corner is node$(11, 6)$, as it is in the eleventh column and the sixth row. (if you start counting from the top left corner) These represent our vertices, all possible positions our agent may move to. Our agent is in the bottom left corner, represented as a yellow square. From this position it wants to reach a goal position. Goals are represented as circles. In this case multiple goals exist. If the position of a agent and a circle are the same, the agent has reached a goal. To do this the agent can take actions, these actions are its moves. We assume a agent can move either to the right, to the left, to the top, to the bottom or stay where it is. For now our agent can only take the top, stay or right action, as it is in a corner. Our agent needs to take multiple of these actions to reach a goal. In order to sequence these action, we establish so called 'time steps'. These time steps are numbered from 0 to infinity. Changing from time step 0 to time step 1, is done by taking the first action. At each time step a agent needs to take an action. Let us assume the agent choses to go one step to the top at first. It has now gone from node $(1, 6)$ to the node $(1, 5)$. Now it takes the action right, to move one step to the right. It is now at node$(2, 5)$, which is right underneath one of the goals. Thus our agent now takes one step to the top again and is at position $(2, 4)$. One of it goals (a circle) is also at position

$(2, 4)$. Hence our agent has now reached its goal at time step 3. We have now found a path from the starting position to a goal node. $(((1, 6), (1, 5), (2, 4)))$ This sequence can be represented as a list of actions taken. ('top','right','top')

Now we can step up the notch a bit. We will introduce the "multi agent pathfinding problem". (MAPF) This was the problem domain, we were tasked to experiment on. [3] Here multiple agents have to navigate through a shared environment to goal positions. Simply put is the addition of multiple SAPF problems into one big problem. The basic idea stays the same. We have time steps, where each agent takes a action. (All actions happen synchronously at the same time.) Each agent wants to move from its starting position to its goal position. However there are multiple constraints that have to be respected. Each position (vertex) can only be occupied by at most one agent . Hence no two agents may share a position. If this is violated, we call it a vertex conflict. The second type of conflict are edge conflicts. Each edge between two positions (vertices) can only be taken by at most one agent. Hence agents may not switch positions with one another.

3 Asprilo

Asprilo is a framework that can represent multiple problem settings in Answer Set Programming (ASP). [4] Its main focus are logistical problems. More information can be found in the original paper introducing Asprilo [1] In general Asprilo is founded on the clingo ASP solver [5] In our case we will focus ourselves on the so called Asprilo "M" domain. It is the simplest of the Asprilo domains, but it can easily represent MAPF problems. In Asprilo agents are called "robots". Each uniquely identified by a number. Our Goals are "shelves", also uniquely identified by a number. Each robot wants to move under the shelf with the same number. Our Graph itself is a coordinate system of squares. Each square is also called a 'node'. Robots can only move on these nodes. The top left node is $(0, 0)$. The bottom right corner node is (X,Y), where X and Y are the dimension of the instance. As before you can imagine that the x value of a node is the column it is in, counting from the left. The y value is the row a node is in, counting from the top.

In general Asprilo comes with a multitude of tools. The one we used the most is the visualizer. You already saw a picture of how Asprilo visualizes SAPF/MAPF problems in figure 1.

[3] To be pendant, we were tasked to deal with the non-anonymized MAPF problem. In general if every goal has to be reached by a specific agent, we call it a non-anonymized MAPF problem. Otherwise if every goal has to be reached, but any robot can end on any goal we call it a anonymized MAPF problem.

[4] https://github.com/potassco/asprilo

[5] https://potassco.org/clingo/

4 Our Scripts

We created multiple programs to support or enable our project. We will explain the way they work to some detail. If you are not interested in the specifics, you can safely skip this section and continue reading from section 5 'Our Mergers'. All of our scripts can be found on our Github, as transparency is especially important. [6] We want to show that the way we create and collect data is not skewed or biased in any form. Even more details on our instances can be found in the 'Experiments' section.

4.1 Benchmark Generator

At first we wanted to use the instance generator provided by Asprilo.[7], but due to outdated libraries we could not run the program. Thus we created our own script. The benchmark generator is a script we wrote in python and can also be found on our GitHub. [8] It quickly creates new instance to test our MAPF solvers on. Given the dimensions for x and y and a number of robots, a new benchmark is created. It has the size (X,Y) where X is the number of nodes per column and Y is the number of nodes per row. In this map N robots and N shelves are placed, where N is the number of robots given as input. The output is a new directory in our benchmark directory, that contains a file with our newly generated instance in predicates accepted by the Asprilo visualizer. We start by first generating all of the available nodes. The number of nodes is $X \cdot Y$, uniquely numbered from 1 to $X \cdot Y$. Our Benchmark is generated in a for loop. Per iteration one robot and one shelf are placed on two different available nodes. If there is not enough space available the program will raise an error and stop the generation. We then generate a product on the shelf and a order saying that the newly created robot should pick up the newly created shelf. If no error occurs, we generate a new directory named "benchmark-i" where i is replaced by the number of benchmarks+1. In this directory we write a file named "xX_yY_n$X \cdot Y$_rN_sN_ps0_pr$N$_uN_oN.lp", where X,Y and N are defined as before. This convention is in accordance with the output of the Asprilo benchmark generator. (On their GitHub you can also find information what each of the letters means). These specifications are also added to a csv file called "specs-benchmark.csv". It contains the properties of every benchmark. Currently we keep track of the number of robots, number of nodes and instance size. After creating our file we start generating the predicates representing our instance. We output one predicate per node, robot, shelf, order and product.

This iterative process was chosen as we guarantee to generate correct instances of any size that have no apparent pattern. It is also quite efficient and the output can easily function with our other scripts. In the future it could be possible to add a additional input, representing the percentage of nodes that

[6] https://github.com/NikKaem/mapf-project
[7] https://github.com/potassco/asprilo
[8] https://github.com/NikKaem/mapf-project

should be 'inactive' (not available to move on) in the instance. The first step of the generative process can easily be modified to accommodate this.

4.2 Plan Creator

The next step after creating an instance is creating the individual plans per robot. We automated this process using a 'plan creator' program written in python. It iterates over every benchmark directory. It looks at whether the plans for each robot have already been generated or not. If the plans are already there, it skips over them. Otherwise it will start generating them. It takes the instance and parses it. It will then infer the number of robots. For each robot it will generate a plan by calling the Asprilo solver with a edited instance. The modified instance contains all of the nodes, one robot, one order involving the robot, one product involving the order and one shelf involving the order. We now repeately call the Asprilo solver with increasing horizon. With this procedure we are guaranteed to find a correct and optimal solution for the individual robot. We then take the solution and output it into a file called "plan_robot$i$.lp" where i is replaced by the number of the robot.

4.3 Plan Renamer

The 'plan Renamer script' does as the name implies. Put simply given a solution of our MAPF solvers, it outputs the solution with renamed predicates. A solution of our MAPF solvers uses predicates in the format "occurs'(object(robot,i),action(move,(X,Y),T)", where i is the unique number of a robot, X and Y are the change in position of robot i on the x respective y axis and T is the time step at which robot i takes this action. The plan renamer reads this input and creates a new predicate "occurs(...)" where the content of the parentheses is the same as in the "occurs"' predicate. Thus given the inputted plans, we are guaranteed to output plans that can easily be piped into the Asprilo visualizer.

4.4 Solver

To first find a solution we use what we lovingly call the 'solver' script. It is written in Python. Given a MAPF problem and one of our MAPF mergers, it computes the a solution and outputs additional information on the solution into a csv file. The solution has the minimal amount of time steps needed to find a solution with this MAPF merger. The way our 'solver' script works is by repeatedly calling clingo with the merger, the instance and a increasing constant 'horizon'. 'Horizon' is a arbitrarily chosen variable that is fixed per clingo call. It represents the maximal amount of time steps for this call. Hence it the higher the horizon, the more actions each robot can take. We start running clingo with a horizon of 1 and increase the horizon by 1, if no solution was found. There is a limit set on high the horizon can rise. This needs to be the case as running a merger on a MAPF problem is a semi decidable task. If a solution can be found

we are guaranteed to find it, but if there is no solution that can be found the program will run indefinitely. To counteract this the 'horizon' variable needs to be limited. If no solution can be found with the highest horizon, the program outputs no plans. No matter if a solution is found or not, the script writes a new line in a csv file. The output consists of the name of the merger used, the name of the instance, the time needed to find a solution, the horizon used to find the solution and the amount of atoms grounded. If a solution is found the last three values are determined by the actual data. If no solution is found, we instead write -1 for every value, thus -1 is our defined 'error' case.

4.5 Benchmarking Script

We also created a Benchmarking Script in Python. This program

4.6 Data Analysis

We created a Data Analysis Script in R. As mentioned before, we record all of our generated data in multiple csv files. With our data analysis script we collect all of the data across multiple files and can generate plots and tables from this data. All of the plots you can see in our "Experiments" section were generated using this script.

These plots are explained in that section as well.

We will cut back on the explanations for this script, as it simply showcases our data. If you are interested in the details on how they were created, you can find the program code on our Git Page. [9]

5 Our Mergers

Given initial plans for each robot, we compute conflict free plans that satisfy all goal conditions for every robot. The inputted plans are generated individually for each robot. Each robot generated the shortest possible path to its respective goal, while assuming no other robot existed. Thus if we let all robots run their initial plans, conflicts may arise. Our task is now to merge all of these plans by adapting the initial plans if required.

We have created multiple merger variants. All can be found on our GitHub Page. [10] We did this in order to find out how multiple parameters influence the resources required to find a solution and how the solution quality differs. In this section we will explain the way each merger works. We will introduce them with regards to the amount of the initial plans they keep in the solution. The first mergers will keep the least, the last mergers will keep the highest amount.

The general structure for each merger is the same. We have input, we process the input and we then output the solution. Input and output are done in the same

[9] https://github.com/NikKaem/mapf-project
[10] https://github.com/NikKaem/mapf-project

manner for every approach. We take the instance and infer: the amount of robots, the starting position per robot, the goal position per robot and all active nodes in the instance. The program now has a complete representation of the problem. We use this representation to generate our conflict-free solution, in one way or another depending on the merger. This solution is then outputted in a uniform way. We take all of the position, except the starting and goal position, and calculate the difference to the position before. This difference is the action the robot has taken. For example position(1,(2,2),0) and position(1,(2,3),1) would represent

```
At time step 0 robot 1 is at position (2,2).
At time step 1 robot 1 is at position (2,3).
```

Thus we can conclude that between time step 0 and 1 the robot moved from (2,2) to (2,3). Subtracting the new position from the old position (2-2, 3-2) leads us to the conclusion (0,1) the robot has taken one step to the right. This action is then outputted as our solution.

5.1 Random Moves

The least complex way to generate conflict free plans given the intial plans is to completly disregard them. This reduces the problem to simply generating plans. Thus we created a merger that simply tries out every possible action for every robot.

The actual processing is done in the following lines

```
adj_node(R,(X+DX,Y+DY),T) :- position(R,(X,Y),T), node(X+DX,Y+DY), DX=-1..1, DY=-1..1,
1{position(R,(X,Y),T+1) : adj_node(R,(X,Y),T)}1 :- adj_node(R,_,T).

:-  position(R,(X,Y),T), position(R',(X,Y),T), R!=R'.
:-  position(R,(X,Y),T-1), position(R,(X',Y'),T),
    position(R',(X',Y'),T-1), position(R',(X,Y),T), R!=R'.
:-  position(R,(X,Y),horizon), goal_node(R,(X',Y')), (X,Y) != (X',Y').
```

The first two line generate every possible move. The last 4 lines assure us no conflict arises and every robot reaches its goal.

5.2 Iterative conflict resolution

The veryx first idea we had was to

we want to merge these plans into conflict free plans that satisfy all goal conditions.

We started with an iterative conflict solver We got into a dead end when starting to work on the edge cases of 2 robots and multiple robots.

We chose to go down another path. random move generation.

Completely random mover then random mover that takes path till the first conflict

then we tried constraint learning

Then we tried random moves dymnamic times

6 Experiments

We evaluated our different merger approaches on 55 different benchmarks. All of these benchmarks can be found on GitHub. [11] Our examples range from simple 2 robot problems with 6 Nodes to 8 Robot Problems with 225 Nodes total. The first 27 benchmarks are handcrafted by us. The remaining tests were generated by our Instance generator.

To collect our data we used our benchmarking script. As mentioned in section "Our Scripts" it outputs the information into a csv file.

We analyzed our mergers under different perspectives. We looked at the time till a solution is found, the quality of the solution and the memory needed to find a solution. We plotted all of this data in multiple diagrams to highlight certain properties.

Let us start with the computation time. This is the time needed from starting clingo till a solution is found. It includes both the grounding and the solving of the program.

Our first plot is a bar plot showing the computation time. Here you can see the average time per approach for every benchmark. On the y axis you can see the name of each approach. The x axis is a time scale from 0 till 10 seconds. For every approach we mark the average time needed to compute solutions for every benchmark. However we only include the computation time in the average if a solution is found. If a solution is found, we compute it 10 times.

Only including the computation time if a solution is found could potentially favour approaches that work on smaller instances but do not find a solution on bigger instances. To show how much impact this has we have another graph later on.

Computing the solutions multiple times should help in reducing the influence of outside factors, like background processes using resources of the benchmarking system. To show how much outside influence or randomness plays a role in computation we have another graph later on.

We now compare the computation time of every approach against one another. We call this the "winning percentage" of every approach. The "winner" of a benchmark is the approach with the lowest average solution computation time. The x axis is the chance of a approach being a winner. The y axis is the name of every approach. The actual data are bars that show high the "winning percentage" of every approach is. To collect the data we used all of our benchmarks. For every benchmark we looked at who the winner is. We count the amount of wins per approach and divide this by the total number of benchmarks. This is the "winning percentage" of every approach marked on the plot.

We wanted to know if some approaches work better for instances with certain features. The first property that came to our mind was the number of robots. How does the number of robots influence the winning percentage of the approaches? Do some approaches work better for lower amounts of robots while others work best for high amount of robots?

[11] https://github.com/NikKaem/mapf-project

What we did now was again calculate the "winning percentage" of every approach. However we first group our benchmarks with regard to the number of robots in the instance. As mentioned before we have benchmarks ranging from 2 robots to 8 robots. What you can see in plot x is the winning percentage of each approach per group. The x axis is again the percentage of how likely a approach is to be a winner. The y axis is the name of every approach that is atleast once a winner. At the top right hand side of every subgraph is a comment 'n=Num' where Num is the amount of benchmarks per group. The actual data marked in the plots is the calculated by looking at every benchmark in a group. We then count the amount of "wins" per approach and divide this by the total amount of benchmarks in the group.

7 Conclusion

References

1. Gebser M., Nguyen V., Obermeier P., Otto T., Sabuncu O., Schaub T., Son T.C. (2018). Experimenting with robotic intra-logistics domains. TPLP. 18. 502-519.
2. Guy S., Schwitter R. (2017). The PENG ASP system: architecture, language and authoring tool. Lang. Resour. Evaluation. 51(1). 67-92.
3. Tarek R. https://github.com/tramadan-up/asprilo-project.
4. Tom S., Julian B., Hannes W. https://github.com/tzschmidt/PlanMerger
5. Marcus F., Max W. https://github.com/Zard0c/ProjektMAPF.
6. Adrian S. https://github.com/salewsky/Plan-Merging.