Multi Agent Pathfinding - Individual Plan Merger with ASP

Marius Wawerek

Potsdam University

Abstract.

1 Introduction

Logistic 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.

MAPF in General Tuple of Graph and starting positions and Goals

In General NP Hard problem One way to approach the problem -¿ K individual agents

instead of having a global omniscient being merge all the plans Every robot finds the best individual plan and then we merge all of these

Instead of exponential time only linear/polynomial time increase per robot

We did this in ASP Clingo We use the Asprilo framework [1] as the foundation of our encoding. The "M" domain is our focus as it represents the MAPF problem.

2 Asprilo / Problem Setting

Pathfinding is the main problem.

Given Graph with Vertex and Edges, Starting Positions, Goals We want to find a path (connected sequence of vertices) from starting position to goal.

We differentiate between 2 "main" groups. The first one being single agent pathfinding. It deals with the navigation of a single agent to a goal position in a given environment. The second group is Multi Agent Pathfinding (MAPF). Here multiple agents have to navigate through a shared environment to goal positions. 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.

Again multiple properties can influence the problem setting. Each Instance can be analyzed with regard to these properties. anonymized vs Non anonymized

A conflict is defined as a violation of one of two constraints. The first are vertex conflicts. Each position (vertex) can only be occupied by at most one robot . Hence no two robots may share a position. The second conflict type are

edge conflicts. Each edge between two positions (vertices) can only be taken by at most one robot. Hence robots may not switch positions with one another.

Asprilo is a framework that can represent multiple problem settings. Its main focus are logistical problems. More information can be found in the original paper introducing asprilo [1]

In this paper we will focus ourselves on the "M" domain. It is the simplest of the asprilo domains and can represent MAPF problems.

Agents are called "robots". Our Goals are shelves Our Graph is a coordinate system of grids. Top left corner is (0,0). Bottom right corner is (X,Y), where X and Y are the dimension of the instance. Asprilo comes with a multitude of tools. The one we used the most is the visualizer.

3 Our Scripts / Benchmark generator

We created multiple programs to support or enable our project.

The benchmark generator is a script we wrote in python and can also be found on our GitHub. [1] Given the dimensions for x and y and a number of robots, new benchmarks are created. They have 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.

At first we wanted to use the generator provided by asprilo.[2], but due to outdated libraries we could not run the program. Thus we created our own script.

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.

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

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 should be inactive in the instance. The first step of the generative process can easily be modified to accommodate this.

We also created a Plan Creator 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.

We also created a Benchmark Script in Python.

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 this 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. [3]

4 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. [4] 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.

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

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 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.

4.1 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.

4.2 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

5 Experiments

We evaluated our different merger approaches on 55 different benchmarks. All of these benchmarks can be found on GitHub. [5] 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.

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

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?

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.

6 Comparision with other mergers

As part of the "Praktikum" of the "Lehrstuhl" Knowledge Representation and Reasoning multiple students were tasked to create mergers for the MAPF Problem with k individual agents. In total 5 groups emerged.

In this section we will compare our mergers with one merger of each other group. These mergers were chosen by the groups themselves. We will explain the basics of each approach, but more information can be found in the respective paper.

We tested all these mergers on a subset of benchmarks. To reduce the bias every group added 4 benchmarks to the pool.

All this data was created using our plan creator, plan renamer, benchmarker and data analysis scripts.

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.