Multi Agent Pathfinding - *k* Individual Agent Mergers in Asprilo

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 [2]. 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. [4]

1 Introduction

Logistics is a complex science on its 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 our first 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 some 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. [4] 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 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.

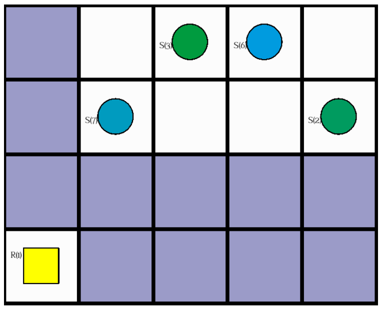

Fig. 1. Single Agent Pathfinding

In figure 1 you can see a visualization of a SAPF problem. 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$(5, 4)$, as it is in the fifth column and the fourth 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. In our example the 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. [1] Here multiple agents have to navigate through a shared environment to goal positions. Simply put it is the addition of multiple SAPF

[1] 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.

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. We assume that we have k agents, where k is a natural number greater than zero. As before our goal is generating a sequence of actions for every robot, where each robot ends on a goal position. This sequence is called the overall 'plan'. We could potentially create this plan by trying out every move for every robot, while avoiding conflicts. Calculation does take some time, but we find a solution if it exists. When looking at the scalability of this however, we have an exponential increase in possibilities per robot added to the instance. If we increase k by one, we now have 5^{k+1} moves we have to try out. Another downside is the fact that we cannot keep our previously computed solution. As the new robot may cause new conflicts to arise, we have to solve everything once again. One possible approach to mitigate these disadvantages is called 'k-individual agent merger'. To compute our overall plan, we first create multiple individual plans. Each robot assumes that in the given environment no other robot exists. We then calculate the shortest possible path to a goal position. The main task now is to 'merge' these initial plans together. If we run all of these initial plans, conflicts may arise due to the independence assumption each robot had. However instead of computing completely new plans, we only need to adjust these initial plans. We can add 'wait' moves to the initial plan or we calculate different moves only for a part of the initial plans. These different modification strategies are the main difference between different 'k-individual agent mergers'. With this approach we also increased scalability. If we increase k by one, we can generally keep the previously computed plans. The only modifications needed are for plans that have a conflict with our newly added plan. In the worst case this can still be a exponential amount of adjusted moves, however in the best case we only have a polynomial amount of moves that need to be modified.

3 Asprilo

Asprilo is a framework that can represent multiple problem settings in Answer Set Programming (ASP). [2] 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 [3] In our case we will focus ourselves on the Asprilo "M" domain. It is the simplest of the Asprilo domains, but it can easily represent MAPF problems. Asprilo comes with a multitude of tools. The one we used the most is the visualizer. You already saw a picture of

[2] https://github.com/potassco/asprilo
[3] https://potassco.org/clingo/

how Asprilo visualizes SAPF/MAPF problems in figure 1. 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.

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 [4], as transparency is important. We want to show that the way we create and collect data is not skewed or biased in any form. That is why even more details on our instances can be found in the ‘Experiments’ section. In this paper we will explain the way our instance generator works in detail. We will also briefly mention our plan creator, plan renamer, solver, benchmarking and data analysis script. More detail on all of those can be found in the final report of the “Praktikum” on our Github repository.

4.1 Benchmark Generator

At first we wanted to use the instance generator provided by Asprilo, 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. It quickly creates a 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 columns and Y is the number rows. In this map N robots and N shelves are placed, where N is the number of robots given as input. If there is not enough space available the program will raise an error and stop the generation. 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 output one predicate per node, robot and shelf[4] in our instance. We also keep track of the number of robots, number of nodes and instance size in a csv file.

4.2 Plan Creator and Plan Renamer

To run our MAPF solvers, we need to create input and adjust our output. The plan creator generates the initial plan each robot has. It outputs these into

[4] We also create a order and a product per robot and output a predicate for each.

separate files, that can be piped into our mergers. From this input our mergers generate a solution. However to prevent duplicate predicates with the initial plans, we have to output our final plan with a slightly modified predicate name. This does not conform to the Asprilo visualizer format, thus we created a plan renamer script in Python. It takes the outputted solution and modifies the file, so it once again adheres to the accepted format.

4.3 Solver, Benchmarking and Data Analysis Script

Running experiments is interesting itself. We can see which instances our mergers can handle and which it cannot. However we are also interested in the process of finding a solution itself. How long does it take to compute a solution? Which is the first horizon [5] a solution is found. To record this data we created both a 'solver' and a 'benchmarking' script. The 'solver' script takes a MAPF merger, a MAPF instance and every initial plan. It generates a solution while also recording which horizon was required to find it and how many atoms were grounded. The solution is guaranteed to be (one of) the optimal solutions for this merger. The program repeatedly runs clingo with our merger and a increasing horizon. We start with horizon 1 and iterate until we either find a solution or reach the max horizon set in the program. If we reach the maximum and find no solution, we stop the 'solver' and output that the given merger cannot find a solution in the given horizon. Thus the 'solver' script records data on the surrounding resources and variables required to compute a solution. There is one resource that is not collected however, the time required to calculate a solution. We outsourced this to a separate script called 'benchmarking'. It takes the recorded data from the 'solver' and calcutates the solution multiple times with the given merger and horizon. The program writes down how much time is required to calculate each solution. Hence we have multiple samples of calculation time per instance and merger. Amongst other things, this was done in order to counteract how much randomness factors into calculation. Thus analyzing the data is quite important. We created a data analysis script in R, that helps us with this task. It combines all of our generated data and visualizes this via various plots. You can see some examples in section 6 'Experiments'. Once again all of the source code is available on our Repository on Git.

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

[5] 'Horizon' is a arbitrarily chosen variable that is fixed per call of clingo with our merger. It represents the maximal amount of time steps for this call. Hence the higher the horizon, the more actions each robot can take.

the initial plans if required. We have created multiple merger variants that each uses a different modification strategy for the inital plans. 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. The actual encodings can be found on our Git repository as well.

The general structure for each merger is the same. We have input, we process the input and we then output the solution. To represent the position each robot has at a given time step, we use a predicate called 'position(R,(X, Y),T)'. R is the ID of the robot. (X, Y) are the coordinates of the node where robot R is at. X refers to the column the robot is currently in. Y respectively refers to the row the robot is currently in. The remaining variable T is the time step. For example if $T = 0$, the position (X, Y) is equal to the starting position of robot R. This position predicate is the backbone of our encoding. From it we can calculate every conflict, every solution and every action taken by each robot. Hence input and output can be done in the same manner for every merger variant. We take the instance and infer: the amount of robots, the starting position per robot, the goal position per robot, the initial plans 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 part of 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. With this approach it strays the furthest afar from 'k-individual agent mergers'. We call it 'random moves'. It works as a reference on how our mergers differ from exponential computation time.

5.2 Iterative conflict resolution

This merger creates various 'layers' to solve conflicts. Each 'layer' solves one conflict by predefined movements that are added to a plan. In this layer we then keep all the previously adapted moves and add our new moves. This process repeats until a solution is found. This approach however only works on instances with at most 2 robots. It could potentially be modified to handle more cases, however this would require an immense effort. You would have to revamp the layer management system from the ground up.

5.3 Random moves - specific conflict

This merger is similiar to 'random moves'. But it takes part of the initial plans. If a conflict arises it keeps the original plans until one time step before. It then tries out every possible move for the robots involved in the conflict. Thus if a robot is not involved in a conflict it keeps the original plan.

We also have a variant of this solver called 'global conflict'. It also keeps the initial plans until a conflict has risen. However as soon as one conflict is found, every robot starts adapting its plan. This is required for some of our benchmark, as you would otherwise be stuck in a dead end. The robots that are not yet involved in a conflict will continue on their initial plan (in the original variant), which makes it impossible to solve some instances.

5.4 Random moves - change time

This is similiar to 'specific conflict' in that it also keeps parts of the original plan. If you imagine the original plan as a time line, what this approach does is: If a conflict is found, we cut out a 'window' of moves in the original plans and replace them by newly generated moves. In practice we set the window size to 2. Thus if a conflict arises at time step T, we remove the 3 moves before T, $T-1$, $T-2$. We also remove the 2 moves $T+1$, $T+2$ after the conflict. We need to replace these moves with new moves. Once again we simply try out every possible move, until we find moves that 'connect' us from the original plan before the conflict to the original moves after.

5.5 Random moves - dynamic time

'Dynamic time' resembles 'change time'. It also keeps part of the original plans and only cuts out moves if a conflict arises. However in contrast to 'change time' the 'window' of cut moves is determined dynamically. In the extreme case it can replace every move from the start to the conflict. What makes this approach able to handle more instances is how it deals with the moves after the conflict. Where 'change time' replaces the moves afterwards, 'dynamic time' just delays them. If required it adds additional moves to the initial plan and then keeps on moving as if nothing happened. For example it may add a move 'up' and then 'down' to dodge another robot. It will then move towards its goal unfazed. We also have a variant called dynamic time minimize. It does not lower the horizon to find a solution, but it does raise the quality of the found solution. With 'dynamic time' it sometimes is possible that robots wait in front of a shelf until the horizon is reached. They then move onto the goal at the same time step as every other robot. In general this is still a solution, however in practice you would like that a robot goes to a goal as soon as possible and waits there. ‘Dynamic time minimize’ forces this behaviour, but is a bit slower in computation as a trade off.

6 Experiments

We evaluated our different merger approaches on 56 different benchmarks. All of these benchmarks can be found on GitHub[4]. Our examples range from simple 2 robot problems with 6 Nodes to 8 Robot Problems with 225 Nodes total. 28 of those benchmarks are handcrafted by us. They often are composed of special cases, where a merger may not find a solution even though a solution exists. The remaining tests were generated by our Instance generator. As they are usually bigger instances with no obstacles, they are more adequate to judge performance. To collect the data presented we used our programs, especially the benchmarking script. We analyzed our mergers under different perspectives. We looked especially at the time till a solution is found. We plotted this information 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. Figure 2 is a bar plot showing the computation time. Here you can see the average time each approach needs to solve a benchmark.

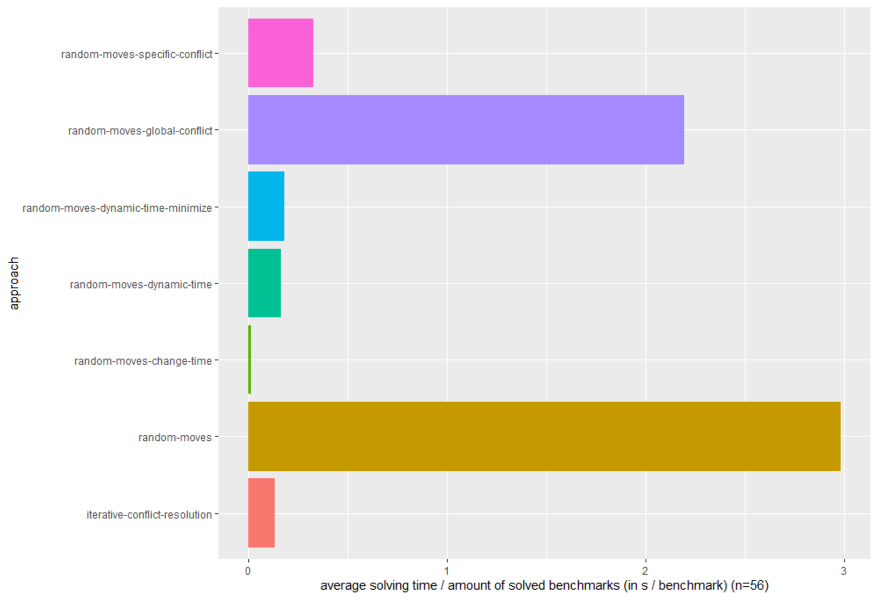

Fig. 2. Average solving time (per solved instance)

On the y axis you can see the name of each approach. The x axis is a time scale from 0 to 3 seconds per benchmark. For example random moves is at 3, which means it needs on average 3 seconds to calculate one of our instances. However we only include the computation time if a solution is found. 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. We want to showcase how ‘powerful’ each approach is in table 1. Here you can see how many of our 56 benchmarks each approach could solve. One approach we want to highlight is ‘random moves change time’. While it is the fastest in solving time, it is also the one with the second least amount of solved benchmarks. The same is true for ‘iterative conflict resolution’. It can handle every 2 robot instance, however if more robots are added it will fail. This is why its computation time seems like it works really well. But if you look into the actual data, you are disappointed. This is part of the trade off users face. What do you value more: fast solving time or a MAPF solver that can handle every instance? Maybe you want both, which merger should you choose then? Again no general conclusion can be drawn, but we have a guideline you can follow. When looking at different instances the main difference which distinguishes more computationally challenging instances is the amount of robots. The more robots are involved, the more conflicts can potentially be caused. What happens if you group benchmarks by the amount of robots each benchmark has and compare the computation time of approaches per group? For that, we introduce the concept of a ‘winner’ of a benchmark. The “winner” is the approach with the lowest average solution computation time. For every benchmark we looked at who the ‘winner’ is. We then sort each benchmark based on how many robots are in the instance. Finally we count the amount of wins per approach in each group and this is what you can see in figure 3. At the top right hand side of every subgraph is a comment ‘n=*Num*’ where *Num* is the amount of benchmarks per group. The x axis is the amount of wins of a approach. The y axis is the name of every approach, which won atleast once. You can see a clear change in the approaches with the most wins depending on the group. For lower numbers of robots ’random moves’ seems to work well. However for more complex instances, it is less likely to be a winner. Once again an exception to this is ’change time’. If it can solve an instance, it has a respectable chance of being a winner.

Approach	Solved Benchmarks
Specific Conflict	55
Global Conflict	56
Dynamic Time minimize	56
Dynamic Time	56
Change Time	41
Random Moves	56
Iterative Conflict	23

Table 1. Amount of solved benchmarks per approach

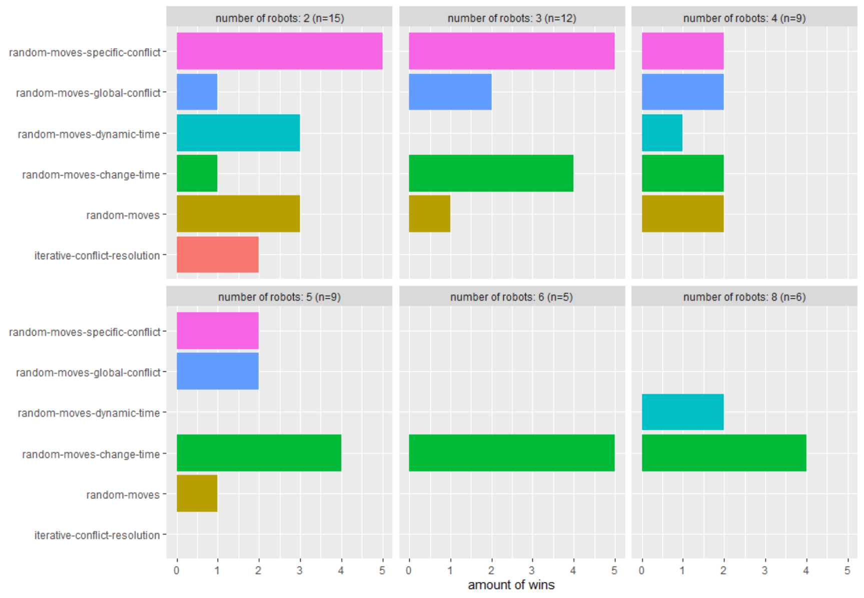

Fig. 3. Amount of 'wins' per approach grouped by the number of robots in instance

7 Conclusion

Overall in this paper you saw a excerpt of MAPF solvers. You can find everything on we did on our Github repository. [4] Supporting our mergers are multiple programs, that can potentially be used in other problem domains as well. Thanks to the "Praktikum" we had the opportunity to test out many interesting ideas for MAPF solvers. There is no absolute conclusion that can be drawn. No 'k-individual agent merger' is the best on all instances. Some are fast, but cannot solve every instance. Others are slower, but work in general. We cannot evaluate our mergers against the original Asprilo solver, as we are missing the time to create initial plans for every robot. Still what can be reported is that when comparing 'k-individual agent mergers' to general MAPF solvers that ignore initial plans like 'random moves', we are happy to report that our mergers fare better in most cases. Especially when increasing the complexity of the instance the difference becomes clear. We hope in the future our data and insights may help others in their task to solve MAPF problems.

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. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T. (2014). Clingo = ASP + Control: Preliminary Report. CoRR, abs/1405.3694.
3. Sharon G., Stern R., Felner R., Sturtevant N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence. 219. 40-66. https://doi.org/10.1016/j.artint.2014.11.006.
4. https://github.com/NikKaem/mapf-project