Multi Agent Pathfinding - k Individual Agent Mergers in Asprilo

Marius Wawerek and Niklas Kämmer

Potsdam University

Abstract. In this paper we will present a simple overview of an internship 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 we created multiple MAPF solvers in clingo 5.4.0 [2]. We also wrote various supporting programs and scripts to generate, collect and visualize all of our data. The details to all of our work will be explained briefly. You can find all of the things mentioned on our GitHub Page. [5]

1 Introduction

Logistics is a complex science on its own. One part of it 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 [4] 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 showing the details to our work that we have done over the course of the semester. We created various programs to support the MAPF solvers that we wrote. Each will be briefly introduced in the section 'Our Scripts'. Moving on, we will present the strategies and coding details of our MAPF solvers. Every merger can be found our Github Page. [5] The section after this deals with the experiments we did on all of our instances and approaches. Using our written programs, highlights of the data will be presented in various plots. The same procedure will then be used to present a comparison between our best merger and the work of the other groups that also took part in the internship. 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 to find 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. Based on that 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 an '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.

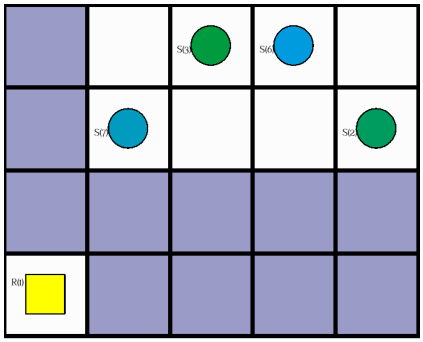

Fig. 1. Single Agent Pathfinding

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 which describe all of the 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 an agent and a circle are the same, the agent has reached a goal. To do this the agent can take actions which are represented by the moves it can execute. We assume an 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 an agent needs to take an action. Let us assume the agent chooses to go one step to the top at first. It has now gone from node $(1, 4)$ to the node $(1, 3)$. Now it takes the action right, to move one step to the right. It is now at node$(2, 3)$, which is next to one of the goals. Thus our agent now takes one step to the top again and is at position $(2, 2)$. One of the goals which is represented by a circle is also at position $(2, 2)$. Hence our agent has now reached its goal at time step 3. We have found a path from the

starting position to a goal node. $(((1,4),(1,3),(2,3),(2,2)))$ 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 problems into one big problem. The basic idea stays the same. We have time steps, where each agent takes a action although now all of the actions happen simultaneously. 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 or vertex can only be occupied by at most one agent at a time. 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 or vertices can only be taken by at most one agent at a time. 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 mentioned before our goal is to generate 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. The calculation would take some time, but we find a solution if it exists. When looking at the scalability of this, however, one can see an exponential increase in possibilities per robot added to the instance. If we increase k by one, we now have up to 5^{k+1} moves that 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 anew. 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 naively, conflicts may arise due to the independence assumption each robot had before. However instead of computing completely new plans, we now only need to adjust these initial plans. We can add 'wait' moves or 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 raise the 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.

[1] To be precise, 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 node we call it an anonymized MAPF problem.

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. [4] 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 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 below 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 represented as $(0, 0)$. 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 located, counting from the left. The y value is the row a node is in, counting from the top.

[2] https://github.com/potassco/asprilo
[3] We are using the version 5.4.0. It can be found on https://potassco.org/clingo/

4 Our Scripts

We created multiple programs to support 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 [5], 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 benchmarks 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 code 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. It quickly offers a quick instance generation in order to test MAPF solvers on. The first step consists of building all of the available nodes. The number of nodes is $X \cdot Y$, uniquely numbered from 1 to $X \cdot Y$. The remaining objects are 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 product on this shelf. If no error occurs, a new directory is created named "benchmark-i" where i is replaced by the number of instances that are currently present in the 'benchmarks' directory plus one. 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 are defined as before and where N represents the number of nodes. 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 [4].

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 instance file we write all of the generated predicates into it. This will result in one predicate per node, robot, shelf, order and product.

This iterative process was chosen since 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 not available to move on or 'inactive' in the instance. The first step of the generative process can easily be modified to accommodate this.

4.2 Plan Creator

After creating an instance the individual plans per robot have to be generated. We automated this process using a 'plan creator' program written in python. It iterates over every instance in the 'benchmark' directory. It confirms first whether the plans for each robot have already been generated or not. If the plans are already there, it skips over the instance. Otherwise it will start generating them. It takes the benchmark and parses the predicates from it. It will then infer the number of robots. For each robot it will generate a plan by calling the Asprilo solver with an edited instance via the Python API of clingo. The modified instance contains all of the nodes, only one robot, one order involving this robot, the corresponding product involving the order and the shelf where the product is placed upon. This process is repeated with increasing horizon for one robot at a time. 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, located in the respective benchmark directory, called "plan_robot$i$.lp" where i is replaced by the number of the robot.

4.3 Plan Renamer

The 'Plan Renamer Script' does what the name implies. Given a solution of our MAPF solvers, it outputs the same solution with renamed predicates. This has to be done because the visualizer for example needs 'occurs' predicates in order to show the plans. A resulting plan of our MAPF solvers, however, 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 or y axis respectively and T is the time step at which robot i takes this action. It is impossible for us to output the right 'occurs' predicates directly since they are already used as input from the individual robot plans. That is why the plan renamer reads in the output of the merger and creates new "occurs(...)" where the content of the parentheses is the same as in our output predicate. Thus given the original plans, we have an easy way to produce plans that can easily be piped into the Asprilo visualizer.

4.4 Solver

In order to find out if a specific approach can solve a benchmark we use what we call the 'Solver' script. It is written in Python. Given a MAPF problem and one of our MAPF mergers, it tries to compute a solution and outputs additional information on the solution into a csv file. The way our 'solver' script works is by repeatedly calling clingo via the Python API 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 that are allowed for this call. Hence the higher the horizon, the more actions each robot can take. We start with a horizon of one and increase the horizon by one, if no solution was found. There is a constant that limits the maximum value

of the 'horizon'. 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 would run indefinitely. This procedure will find the minimal amount of time steps needed to compute a solution with this MAPF merger. The solution will be parsed from the return value of the clingo call which in turn will be written into a file in the 'solution/*merger*' directory of the specific benchmark where *merger* is replaced by the name of the used approach. If no solution can be found with the limited horizon, the program outputs no plans. However, no matter if a solution is found or not, the script writes a new line into a csv file. This line 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 the instance was not solved, we instead write '-1' for every value, thus '-1' is our defined 'error' case.

There are two additional *.txt* files that can be found in the script directory of our repository. They include each approach and benchmark respectively. To exclude an approach or a benchmark from the solving process one can put a '#' in front of the desired line. Otherwise the solver will iterate over every included benchmark for every included approach and execute the procedure that was explained above.

4.5 Benchmarking Script

The script that we use for benchmarking purposes is quite similar to the 'Solver' script in a sense that it also parses the desired benchmarks and approaches from the two *.txt* files that we mentioned earlier. It then iterates over both of them as well and tries to find a solution to one of the instances using one of the mergers. However, there is one clear difference compared to the other script. Since we want to actually try out how fast an approach solves a given problem the 'Benchmarking' script parses the minimal horizon from the csv file generated by the 'Solver' script instead of iterating the constant again. What this means is that we only measure the time needed to solve an instance with an already known horizon. The actual solving process is done similarly to the 'Solver' script.

We use the Python API of clingo to solve a desired instance with a given approach. The difference here is that we do not yield the solution that clingo outputs since we already know that there is one for the used horizon. We did this in order to minimize the impact the API call has on the resulting time. In our early tests it seemed like the yielding process extended the time needed although we cannot conclude this for sure.

This resulting time measurement is computed ten times in a row for every desired merger and each desired benchmark in order to minimize data fluctuations cause by external factors. After this loop the results get written into another csv file where each line only contains the approach, the benchmark and the time needed to solve the latter using the former. These data points can then be used to visualize the variance in time for each approach and benchmark or to compute

the mean of solving time in order to further process them in our data analysis scripts afterwards.

4.6 Data Analysis

In order to evaluate the data from the *.csv* files we had to use some additional scripts. These are written in R and use mostly the 'tidyverse' and 'ggplot2' libraries. The general procedure is to import each file as a dataframe or 'tibble', merge them if necessary and generate plots using the 'ggplot2' library. A lot of the plots we created can be seen in our "Experiments" section. There are, however, multiple plots we decided not to use in the end. They can be seen in the 'plots' and 'comparison' directory on our repository [5]. We will not further explain the details to this script, as it is a just straight forward way of visualizing our data. If you are interested in how all of the plots are generated, you can find the program code on our Git Page [5].

5 Our Mergers

Our task was to build a sophisticated solution to the MAPF problem such that given initial plans and a MAPF instance, a program merges these given plans. The resulting plans have to be conflict free and each robot is supposed to end on its respective goal node. To compute conflict free plans our program has to accurately detect problems with the initial paths first of all which is a challenging task in itself. And although the solving part seems trivial for just two robots the difficulty level increases immensely when the problem involves a lot of robots since many different combinations of the two basic conflicts can occur. Because of this we created multiple variations of such a MAPF solver that each use different strategies to solve these problems. We did this in order to find out how different parameters and merging concepts influence the resources required to find a solution and its 'quality'. This includes required cpu time, memory, the horizon required to find a solution or the amount of unnecessary moves in a solution.

This section presents the basic strategies behind each merger as well as advantages and disadvantages these different strategies might have. The actual encodings can be found on our GitHub Page [5].

The general structure for each merger is the same. It takes the horizon constant, an instance which consists of different nodes, the starting position of the robots and goals they are supposed to reach. In addition to that initial robot plans are given as input as well which specify the move a robot does at a specific time step. The instance and the individual plans are used to build an internal representation of the problem which in turn is used to generate a solution that the program can output in the end. This representation consists mainly out of position predicates 'position(R,(X,Y),T)' which describe the absolute position of a robot at a certain time step. R refers to the ID of the robot. (X, Y) are the coordinates of the node where robot R is currently placed. X and Y specify the column and row position respectively. The remaining variable T is the time step where $T = 0$ describes the starting position of each robot. This position predicate is the backbone of our different encodings. It is used to calculate arising conflicts as well as the output predicates which only show the move a robot does at a certain time step such as up, down, left or right instead of the absolute position in the instance. This is done by taking all of the positions and calculating the difference to the position before. This difference describes the movement of the robot. 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) which means it went downwards.

The final solution consists of multiple of these move predicates which describe the path of the robot from its starting position to the desired goal node.

5.1 Iterative Conflict Resolution

The first merger we created is named 'Iterative Conflict Resolution'. The idea was to create various 'layers' to resolve conflicts step by step. Starting from layer 0 the merger works his way up the layers whenever such a conflict is solved. The conflict resolution is done by copying the moves from the previous layer while adding moves in order to prevent crashes between the robots. These resulting paths are copied into a new layer where the merger can continue to search for new conflicts. This process is repeated until all paths are free from any conflicts.

In this section we want to present the details of our implementation. The first step consists of loading the prerequisites into the merging program in order to create a representation of the problem.

```
1  time(1..horizon).
2
3  node(X,Y) :-
4      init(object(node,_), value(at, (X,Y))).
5
6  position(R,(X,Y),0,0) :-
7      init(object(robot, R), value(at, (X, Y))).
8
9  position(R,(X+D1,Y+D2),T,0) :-
10     occurs(object(robot,R),action(move,(D1,D2)),T),
11     position(R,(X,Y),T-1,0).
```

Listing 1.1. Input Format

For this 'Iterative Conflict Resolution' uses three predicates to represent our instance, 'time', 'node' and 'position' (Listing 1.1). The time predicate is true for every value from one to the specified horizon. Each value represents a time step, where a robot can make a move. Thus we use it to limit the amount of moves. The node predicate is true for every instance node that actually exists. This can be used to check whether the robot is moving out of bounds for example. To finish the input the merger loads every position starting with the initial position at time step 0 by parsing them from the instance using the init predicate. The following positions are generated by moving the robot as stated in the 'occurs' predicates. The position predicate differs from the one mentioned before. We increased its arity by one to keep track of the layer we are in. The initial plans are copied to layer zero. Each conflict resolution increases this layer number. Hence the highest layer contains the final solution.

The next step for the merger is to detect the two basic conflict types - edge conflicts and vertex conflicts.

```
1  conflict_vertex (R1,R2,T,L+1) :-
2      position(R1,(X,Y),T,L),
3      position(R2,(X,Y),T,L),
4      R2>R1, L<=horizon.
5
```

```
conflict_edge(R1,R2,T,L+1) :-
    position(R1,(X,Y),T,L),
    position(R1,(X',Y'),T+1,L),
    position(R2,(X',Y'),T,L),
    position(R2,(X,Y),T+1,L),
    R2>R1, L<=horizon.
```

Listing 1.2. Conflict Detection

This was done using two conflict predicates 'conflict_vertex' and 'conflict_edge'. It contains both robots that are part of the conflict, the time step at which the conflict happened and the layer where new positions are generated in after the conflict is solved. The 'vertex_conflict' predicate is inferred if two robots have the same position at the same time step in the same layer (Listing 1.2). There are scenarios where multiple robots can be part of the same vertex problem. In that case multiple 'vertex_conflict' predicates are generated for each possible pair of robots. The 'conflict_edge' predicate is inferred if two robots swap positions with another (Listing 1.2). This would mean that both have to be at a position where the other robot stood a time step before. An edge conflict can only involve two robots. Thus we only generate one predicate for every violation.

Following the conflict detection, the merger needs to solve them. Vertex conflicts are solved in a simple fashion. The merger creates a waiting move for one of the robots. This can either solve the conflict or lead to an edge conflict. This can be seen in Figure 2 where robot 1 and robot 2 try to access the same square. By using our merger to resolve the conflict robot 1 would add a waiting move to its plan. Since robot 2 continues with its original path an edge problem is created. In order to resolve edge conflicts the merger tries to find all of the positions a robot could potentially move to. First the merger checks all adjacent nodes for the first robot that is part of the conflict. If there is a node that does not interfere with the path of the other robot, the first robot moves onto this node. If there is no such node the merger will check all adjacent nodes for the second robot mentioned in the 'edge_conflict' predicate. In a case where this procedure does not lead to a resolution of the conflict our program will move the first robot to the node the second robot intends to move to after resolving the conflict. This does not solve the conflict properly but it changes the position of the conflict and thus creates a new chance for the robots to look for adjacent nodes. After performing such a dodge move the robot always tries to go back to the node it came from in the next time step. After resolving either a vertex conflicts or an edge conflict the merger will generate new positions into a new

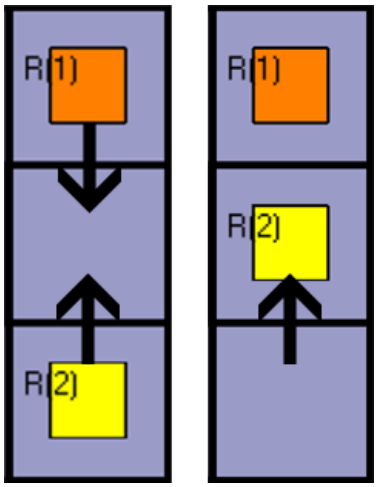

Fig. 2. Vertex Conflict that transforms into an Edge Conflict

layer. These include the waiting or the dodge move respectively. The positions of robots that are not part of a conflict will be copied into the new layer without any modification. This detection and resolution procedure is repeated until the plans are free from any conflicts.

This strategy of solving conflicts has quite a few flaws. That is why we chose not to dive too deep into the source code in this report. For instances that have more than two robots there are a lot of special cases of conflict resolution that this merger will not consider. For edge conflicts it will not look for waiting moves when trying to find a position to dodge to. It will not consider moving the second involved robot back into the way of the first robot in case the first robot cannot move anywhere. If a robot has reached its goal and another robot moves onto this node, no conflict will be detected since no positions are generated after the original plan finishes.

All of these issues can probably be fixed with some time and effort. We noticed, however, that this approach is hard to work with. Every dodge maneuver has to be predefined in some form. Keeping track of all possible edge cases to resolve conflicts makes this very tedious. In addition to that the merger does not exploit the advantages of ASP and clingo very well. It does not utilize many integrity constraints. Due to the additional layer variable in the position predicate, grounding can consume quite a lot of time. This is especially relevant for large instances. This is the reason why we chose to abandon this approach. We wanted to create a merger that can handle more robots while also scaling better for larger instances. It should be said, however, that 'Iterative Conflict Resolution' actually handles most MAPF problems with two robots pretty well even though the merger has a lot of flaws. Because of that reason we wanted to include it in this report and in our data analysis as well.

5.2 Random Moves

In order to not rely on predefined dodge maneuvers in case of conflicts we decided on a new approach. We did not want to use the layer structure any further since that was the main reason the 'Iterative Conflict Resolution' approach was so hard to maintain. Our new approach involved a much larger search space and constraints to quickly reject any models that had flaws in them. We called it 'Random Moves' to have a concise name that describes what it does in a rough way. It tries out every possible move until a solution is found, which looks like random behavior from the outside. But instead of trying out paths in a truly random manner it does so in a deterministic way. Thus this approach represents the brute force solution to the MAPF problem.

```
1 time(1..horizon).
2 node(X,Y) :- init(object(node,_), value(at, (X,Y))).
3 goal_node(R,(X,Y)) :- init(object(shelf,R),value(at,(X,Y))).
4 position(R,(X,Y),0) :- init(object(robot, R), value(at, (X, Y))).
```

Listing 1.3. Input Format

The input format is basically the same as in the earlier approach. You can see the source code in Listing 1.3. The main difference is that we did not use any of the initial plans that were generated for each robot separately. We only use the starting position given by the init predicate from the instance. In addition to that we use a new predicate 'goal_node' referring to the node where a robot wants to end his movement on. This will be relevant for the constraints later on.

```
1  adj_node(R,(X+DX,Y+DY),T) :-
2      position(R,(X,Y),T),
3      node(X+DX,Y+DY),
4      DX=-1..1, DY=-1..1, 1{|DX+DY| == 1;(DX,DY) == (0,0)}1,
5      T<horizon.
6
7  1{position(R,(X,Y),T+1) : adj_node(R,(X,Y),T)}1 :-
8      adj_node(R,_,T).
```

Listing 1.4. Move Generation

The move generation is a straight forward procedure. Our encoding can be found in Listing 1.4. The first step is to generate all adjacent positions to the current position of a robot. This is done by calculating all the possible moves a robot could take such as up, down, right, left and wait. We use the variable DX for describing horizontal movement while the variable DY describes vertical movement. Since diagonal moves are not allowed we had to ensure that only one of both variables has a value different from zero. The only exception for that is a waiting move in which case both variables are zero. After generating all the adjacent nodes a choice rule is used to pick one as the next position for the robot at a certain time step.

```
1  :- position(R,(X,Y),T),
2      position(R',(X,Y),T), R!=R'.
3
4  :- position(R,(X,Y),T-1),
5      position(R,(X',Y'),T),
6      position(R',(X',Y'),T-1),
7      position(R',(X,Y),T), R!=R'.
8
9  :- position(R,(X,Y),horizon),
10     goal_node(R,(X',Y')),
11     (X,Y) != (X',Y').
```

Listing 1.5. Constraints

In the end every model that has a conflict in it or has robots that do not end on their respective goal node should be rejected. Conflicts are detected in the same way as in the 'Iterative Conflict Resolution' approach. You can compare the encodings in Listing 1.5 with Listing 1.2 However we use the bodies of the rules as constraints in this merger. The constraint starting from line 9 ensures that every robot has reached its goal position by the last time step.

```
occurs'(object(robot,R),action(move,(X'−X,Y'−Y)),T+1) :−
    position(R,(X,Y),T),
    position(R,(X',Y'),T+1).
```

Listing 1.6. Output

If a proper solution is found the merger generates predicates that are similar to the 'occurs' from the initial plans. (Listing 1.6) The solver takes a position and its successor in order to calculate the relative moves for each robot. It does that by subtracting the new X and Y values of the successor position from the X and Y values of the original position. As mentioned before, this procedure is used in every one of our mergers. In order to make the output compatible with the visualizer, one can run the 'plan renamer' script [4] on the generated solution.

Looking back at this approach we can conclude that we gained experience with the different ASP structures and how we could use them to our advantage. It should be said, however, that this approach does not really fulfill the goal of the project. The goal was to merge the initial plans in a sophisticated way. Thus the merger is supposed to use as much of the original plans as possible. We completely disregard any of that in this approach. While it does ensure that we always find a solution, if there is one, it comes with a big disadvantage. The computation time for a solution increases exponentially the more robots and nodes there are. More information on this can be found in the section 'Experiments'.

5.3 Random Moves Specific Conflict

The goal of our next approach was to use more of the original plans compared to 'Random Moves'. However, the way conflicts are resolved was supposed to be similar. The idea behind 'Random Moves Specific Conflict' is to find the first conflict for every robot and copy the initial paths until the time step before it. In order to resolve these conflicts and guide the robots to their respective goals afterwards, the merger utilizes the same brute force system as the 'Random Moves' approach (Listing 1.4).

The processing of the input and conflict detection is quite similar to our 'Iterative Conflict Resolution' approach. The main difference is that we do not use the layer system anymore. Instead we use two predicates 'initial_position' and 'position'. 'Initial_position' predicate represents the original path of a robot while 'position' is generated after the conflict detection and represents the new conflict free path. To compensate for different lengths of the original plans, we generate waiting moves on the goal position at the end of every affected path. This allows us to detect conflicts even if a robot is already finished.

[4] https://github.com/NikKaem/mapf-project/blob/Release/scripts/plan_renamer.py

```
conflict (R,T) :-
      initial_position (R,(X,Y),T),
      initial_position (R',(X,Y),T), R!=R'.

conflict (R,T) :-
      initial_position (R,(X,Y),T-1),
      initial_position (R,(X',Y'),T),
      initial_position (R',(X',Y'),T-1),
      initial_position (R',(X,Y),T), R!=R'.
```

Listing 1.7. Conflict Detection

In this approach we could combine both edge and vertex conflict predicates. Every robot that is part of a conflict is provided with 'conflict' predicates, which tell us which robot is involved in a violation at which time step. (Listing 1.7).

```
 first_conflict (R,T_min) :-
    T_min = #min{T:conflict(R,T)},
     initial_position (R,_,0),
    T_min!=#sup.

position(R,(X,Y),T) :-
      initial_position (R,(X,Y),T),
      first_conflict (R,T'), T<T'.

position(R,(X,Y),T) :-
      initial_position (R,(X,Y),T),
    not first_conflict (R,_), T<=horizon.
```

Listing 1.8. Copying of Original Paths

Since we only want to use the initial plans until a problem arises we have to figure out the time step of the first conflict for every robot. Each move that happens before the time step can be copied. If the merger does not find a conflict for some of the robots, it keeps the entire original paths for these robots. (Listing 1.8).

The conflict resolution and goal detection procedure is the same as the one we used for 'Random Moves' (Listing 1.4). The constraints used to avoid problems with the newly generated positions (Listing 1.5) and the output (Listing 1.6) were kept the same as well.

This approach seems to work well on most of the benchmarks. It clearly uses parts of the individually generated plans for each robot which was the goal we had in mind for 'Random Moves Specific Conflict'. There is however, a group of problems we encountered where this approach fails to find a solution. One example for such a problem is our benchmark-6[5] (Figure 3).

[5] https://github.com/NikKaem/mapf-project/tree/Release/benchmarks/benchmark-6

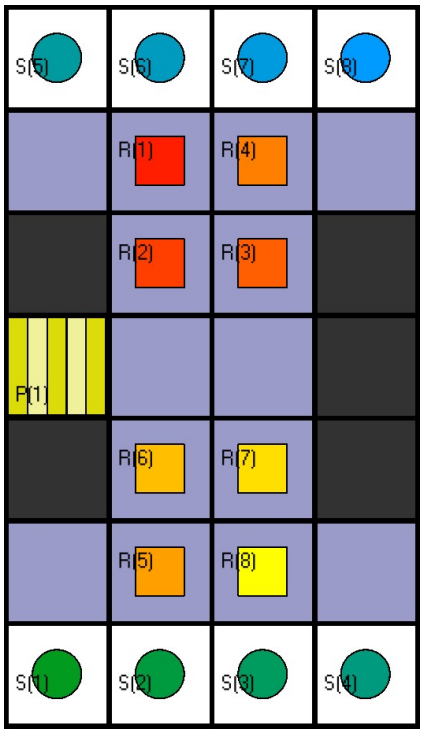

Fig. 3. Benchmark 6

The difficulty with this benchmark lies within the resulting positions after move 1. We call this phenomena "queues". As one can see in figure 3 robot-2 and robot-6 will have a conflict at the next time step T while the robots behind them will collide a time step later at T+1. The same situation arises for the robots next to them. In this scenario robot-1 and robot-5 will copy their original plans until T+1 which means that their position at time step T will be the same as for robot-2 and robot-6 respectively. To solve this problem with our approach robot-2 and robot-6 would both have to move to a different node at time step T which is impossible since there is not enough space for both of them. It is inevitable that a vertex conflict arises following this strategy which is why 'Random Moves Specific Conflict' does not find a solution for this problem.

An easy solution we found was to start adapting the plans as soon as the first conflict occurred. This would allow robot-1 and robot-5 to move backwards in order to free up space for the two robots in front of them. We call this variation of our approach 'Random Moves Global Conflict'. It worked for every benchmark we had at the creation time and it found these solutions reasonably fast. Later in the semester, however, we started working with much larger instances and discovered the performance problem this approach has, especially when a conflict is found early on. Again more on this in the section 'Experiments'.

Another issue we found was that the solution quality of the approaches based on the brute force conflict resolution is also quite low. This approach only checks whether robots end up on their goal position at the last time step. (Listing 1.5). It is possible that a subset of robots could finish earlier than the horizon time. They would still generate 'position' predicates until the horizon is reached, which can result in a lot of unnecessary movement. We refer to the amount of unnecessary movement as solution quality.

5.4 Random Moves Change Time

In order to address the solution quality problem and to use even more parts of the original plans we created our next approach 'Random Moves Change Time'. Our intention with this merging program was to not only recycle a robots movement before a conflict but also after it.

```
change_time(R,T') :-
      conflict (R,T),
      T'=T-window..T+window,
      T'>0, T'<=horizon.

position(R,(X,Y),T) :-
       initial_position (R,(X,Y),T),
      not change_time(R,T), T<=horizon.
```

Listing 1.9. Change Time

In order to do that we detect conflicts the same way, we did for the previous approach. To solve conflicts we now only change certain timesteps before and after a conflict. For every conflict we create a 'change_time' predicate that is inferred for each time step 'around' the conflict time bounded by a window constant (Listing 1.9). This window constant is an arbitrary value that can be adjusted depending on the problem setting. In our tests we had a lot of success with setting the window to a size of two. The 'change_time' predicate tells our merger which moves of the original plans can be changed. The adapted moves are generated with a brute force strategy which is similar to the one used in 'Random Moves' (Listing 1.4). Every time step that is unchanged stays the same as in the original paths (Listing 1.9).

```
:- position(R,(X,Y),T_max),
      position(R,(X',Y'),T_max+1),
      T_max = #max{T:change_time(R,T)},
      DX=X'-X, DY=Y'-Y, |DX|+|DY| >= 2.
```

Listing 1.10. Change Time Constraint

When we transition from the last move that was changed to unchanged moves, we have to ensure that the robot can reach the unchanged position of the initial plan. We specified another constraint that ensures the changed positions and the original positions are 'connected'. (Listing 1.10).

We found out that this approach works much faster on every benchmark that it can solve compared to the less sophisticated versions we presented beforehand. Instead of trying out every path from a problem point on wards this merger only has to figure out move combinations of length $2 * window + 1$ for every conflict. For small window sizes this greatly limits the possibilities for the new paths. This also improves the quality of the solution since there is not a lot of room for unnecessary movement. There is, however, a downside to this approach as well. 'Random Moves Change Time' can only replace moves from the original plans within the 'change time window'. It can never add additional movements. Thus it has to be possible to produce conflict free paths for each robot with the exact amount of steps as the original ones. Since a lot of our benchmarks violate this property, this merger failed to solve a lot of them despite being fast on all the other ones. We will refer to that later on again in section 'Experiments'.

5.5 Random Moves Dynamic Time

'Random Moves Dynamic Time' represents the last and most profound merging concept we came up with. The idea for this one was to offer dynamic window sizes that do not have to be specified by the user while still relying on the change time idea.

```
1  before(B) :-
2      B=0..T_min-1,
3      T_min = #min{T:first_conflict(_,T)},
4      T_min!=#sup.
5
6  after(R,A) :-
7      A=0..horizon-T_max,
8      r_time(R,T_max),
9       first_conflict (R,_).
10
11 1{window(R,B,A) : before(B), after(R,A)}1 :-
12      first_conflict (R,_).
13
14 change_time(R,T') :-
15      first_conflict (R,T_min),
16     T'=T_min-B..T_min+A,
17     window(R,B,A),
18     T'>0, T'<=horizon.
```

Listing 1.11. Window Generation and Change Time

Again one has to first detect the first conflict for every robot. The procedure for this is the same as for our other approaches (Listing 1.7). The merger can use the conflict time steps to calculate how many moves can be changed before and after the conflict. For that we defined a two predicates 'before' and 'after'. The predicate 'before' defines how many of the moves before a conflict can be replaced. The predicate is inferred for every time step before the first conflict overall (Listing 1.11). Although it is possible to define this for every robot individually, by looking at the first conflict of each one, we decided against it. The main reason for that was to save computation time since less positions have to be grounded for robots that have their first conflict at a late time step. If a more general solution is needed for difficult problems the 'before' predicate can easily be altered to be inferred for each robot separately. The 'after' predicate has different values for every robot. Here lies the small difference to 'Random Moves Change Time'. This approach does not replace the original moves after a conflict but rather delays them for a certain amount of steps. We start by determining the amount of moves a robot can make to resolve a conflict, while still being able to follow his original plan afterwards. We do this by calculating the difference between the horizon and the length of the original plan which is given by the predicate 'r_time'. (Listing 1.11).

After generating all the possible 'before' and 'after' predicates, we use a choice rule that selects one 'after' and one 'before'. Based on this we create a 'window' for each robot. Inside the window all moves can be changed. The 'change_time' predicate is used for this and is inferred in a similar way to 'Random Moves Change Time' (Listing 1.11). However instead of the constant window size of two that was used in the former approach the window is now defined by the two values from the 'window' predicate.

```
1  position(R,(X,Y),T) :-
2        initial_position (R,(X,Y),T),
3       window(_,B,_),
4        first_conflict (R,T_min), T<T_min-B.
5
6  position(R,(X,Y),T+A) :-
7        initial_position (R,(X,Y),T),
8       window(R,_,A),
9        first_conflict (R,T_min), T>T_min,
10      r_time(R,T_max), T<=T_max.
11
12 position(R,(X,Y),T') :-
13       initial_position (R,(X,Y),T),
14      r_time(R,T), time(T'),
15      window(R,_,A), T'>T+A, T'<=horizon.
16
17 position(R,(X,Y),T) :-
18       initial_position (R,(X,Y),T),
19      not change_time(R,_).
```

Listing 1.12. Copying of Original Paths

Based on the chosen value for the second variable 'B' of 'window' the moves until the time of the first conflict minus 'B' will be kept. The third variable 'A' in combination with the time of the conflict specifies how long the 'change_time' interval will be. The merger will append all of the original moves that were supposed to happen after the conflict to the newly generated plan. Starting from line 12 we ensure that every robot generates positions until the horizon. If a robot has already reached its goal, then we add 'wait' actions. This ensures that each path has the same length and no conflict is missed. Finally for every robot that is not involved in a conflict the original plans can be copied (Listing 1.12).

The constraints and the output are similar to the approaches mentioned before.

Despite being the most sophisticated merger we created, it still does not work perfectly. The moves that are generated for the 'change_time' predicates follow the same brute force procedure like most of our other merging programs. That means that it suffers from the same problem as our approaches in terms of solution quality. Under certain circumstances unnecessary moves can happen,

which we wanted to avoid as best as possible. Because of this we created a variation of 'Random Moves Dynamic Time' which we called 'Random Moves Dynamic Time Minimize'.

This variation utilizes a '$\#minimize\{1@1, T : change\_time(R, T)\}$' statement that tries to minimize the number of 'change_time' predicates generated for each robot. The resulting solutions are much closer to the initial plans which means that there is less unnecessary movement happening. The only downside this variation has compared to the original approach is the computation time that is slightly higher. This will be shown in the 'Experiments' section.

But the amount of unnecessary movement is not the only problem this merger has. There are benchmarks where 'Random Moves Dynamic Time' cannot find a solution with the minimal horizon. Since we force the robot to move according to its original plan, after a conflict is resolved it can happen that the merger fails to see much shorter paths to the goal node. For example if a robot moves to his goal node in order to resolve a conflict it will still have to move back to the original plan even though it could technically just stay there. However, the merger found solutions, even though some were sub-optimal, for every benchmark we had at the creation time. It does that way faster than any of our other approaches that work for the general case. The details to that will be presented in the 'Experiments' section.

6 Experiments

We evaluated our different merging approaches on 56 different benchmarks excluding the ones that are later presented in the 'Comparison to other Groups' section. While some of them were created by us a lot of them were also imported from other groups in order to test our mergers in even more scenarios. All of these benchmarks and the references to the other groups 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. Some of them are handcrafted. They are often composed out of special cases, where a merger could fail to find a solution even though a solution exists due to missing features. Other benchmarks 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 python programs, especially the benchmarking script. As mentioned before this script solves the instance not once but multiple times with the same merger. Thus we collected a sample of 10 computations per benchmark per approach. In total we have 3350 data samples. These repeated computations allow us minimize additional parameters that can influence the time our MAPF solver needs. An example for that would be the reduction of the impact other processes running on the hardware have. The amount of data we have might not be enough to actually conclude a hypothesis that one could have but it is definitely enough to give a proper intuition and overview over the performance of each merger.

The analysis of our mergers was done by evaluating different stats that we measured. We looked especially at the time that was needed to compute a solution when the right horizon is already given. That was done because it reflects much better what would happen in reality, at least according to our opinion. One would not necessarily start to iterate from horizon on wards but rather use a value that seems plausible. We plotted this information in multiple diagrams to highlight the findings.

In order to get a proper representation of the average solving time an approach needs for a benchmark it is necessary to find out how many instances a merger actually managed to solve. For that we computed this stat for each of our programs. The findings can be seen in the table below (Table 1).

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

Table 1. Amount of solved benchmarks per approach

You can clearly see that 'Iterative Conflict Resolution' and 'Random Moves Change Time' seem to perform the worst for the general case. The reason for

that were mentioned earlier already. 'Iterative Conflict Resolution' cannot solve any benchmarks with more than two robots and 'Random Moves Change Time' will not be able to find a proper result if the instance is not solvable within the time of the original plans. 'Random Moves Specific Conflict' has a problem with one benchmark that was also shown in the respective 'Our Mergers' subsection (Figure 3).

The amount of solvable instances can then be used to calculate the average computation time. Figure 4 is a bar plot showing the average of the total computation time needed for every benchmark divided by the amount of solvable benchmarks per approach. It should also be mentioned that we used the mean of the ten data points of each approach for each benchmark in order to produce this and the following plots.

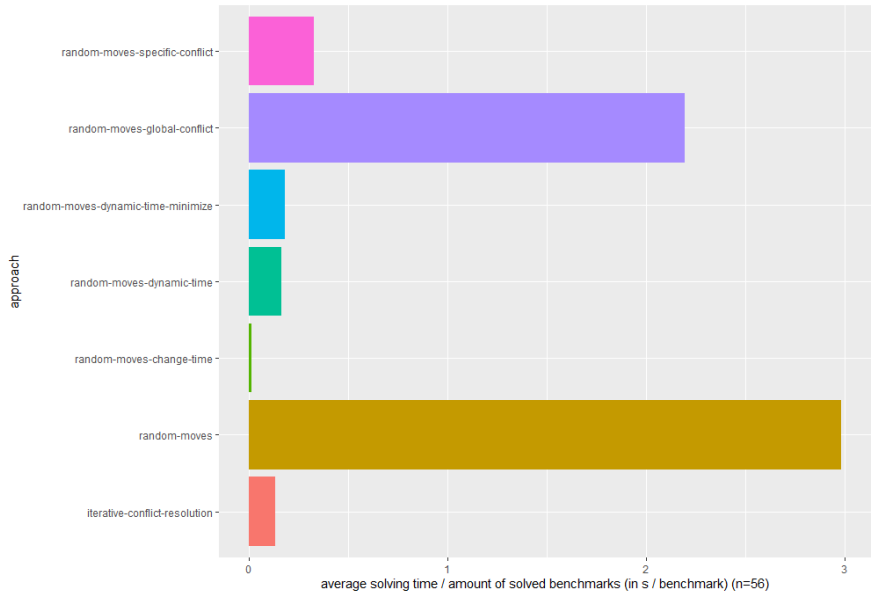

Fig. 4. Average solving time (per solved instance)

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 the bar of random moves ends at 3, which means that it needs on average 3 seconds to calculate one of our instances. However we only include the computation time if a solution is found.

It is quite obvious that 'Random Moves' and 'Random Moves Global Conflict' are the slowest of all the approaches. This was expected since both of them use either nothing or just a small part of the original plans. That results in a lot more predicates that need to be grounded. 'Random Moves Change Time' on the other hand seems to perform phenomenal. It is way faster than any of the other approaches. But sadly this approach only works for some of the instances which means that it might not be advisable to use it if you want a merger that works independent of the scenario. 'Random Moves Dynamic Time'

and the 'Random Moves Dynamic Time Minimize' variation rank second best. The latter is a bit slower however which is explained by the extra minimize statement that we used for it. They are faster than any of the other approaches by quite a margin excluding 'Random Moves Change Time' and 'Iterative Conflict Resolution'. We were happy to see that the merger that we submitted as our final product seems to make a good trade-off between solving time and the amount of benchmarks it can solve. What was surprising to see was that 'Iterative Conflict Resolution' scored that well. It turns out, however, that it only looks like that because this approach does not solve any complex benchmarks that require a lot of computation time. Only including the absolute time without normalizing it favours the approaches that work on smaller instances but do not find a solution on bigger instances. Out of that reason the plot can definitely be misleading.

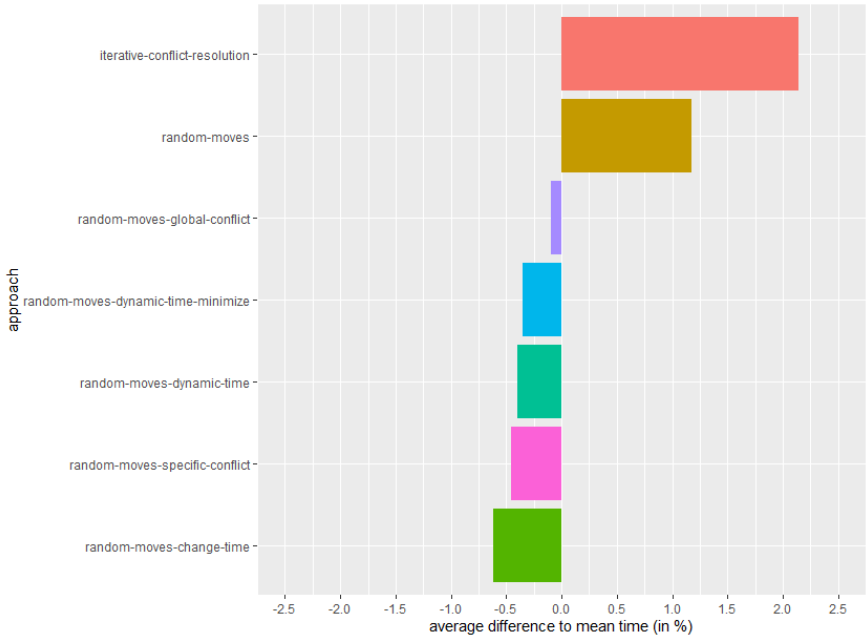

Fig. 5. Average Difference to the Mean Time needed to solve a Benchmark

Looking at Figure 5 you can see that 'Iterative Conflict Resolution' actually performs badly even on the instances that it solves. This plot was produced in multiple steps. First we calculated the mean time over all the approaches that was needed to solve each benchmark. Then this mean time was divided by the time of each specific approach. Using the mean of these values results in the plots below. It shows the average difference to the mean time needed to solve a benchmark by our approaches in percent. The findings are quite similar to the ones of the earlier shown plot. However, as said before it can be seen that 'Iterative Conflict Resolution' is not a good approach even for small instances. 'Random Moves Specific Conflict' in turn scores better and is on average a tiny bit faster compared to the 'Dynamic Time' approaches. The problem is that it

also has a problem with solving every instance presented, even if it is not as apparent compared to 'Random Moves Change Time' for example.

This is part of the trade off users face. One has to decide whether fast solving times or a MAPF solver that can handle every instance is important to oneself. In this last paragraph we want to present guidelines on which merger you should use in certain scenarios. Again no definite conclusion can be drawn based on the small sample size we used but we can at least give the user an idea of what might be the best to try.

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. To show a proper distinction of how approaches work for different robot amounts we produced another plot (Figure 6). It is also possible to distinguish between the amount of nodes but since for our benchmarks the amount of nodes and robots are mostly correlated we decided to do this only for the robot count. First we

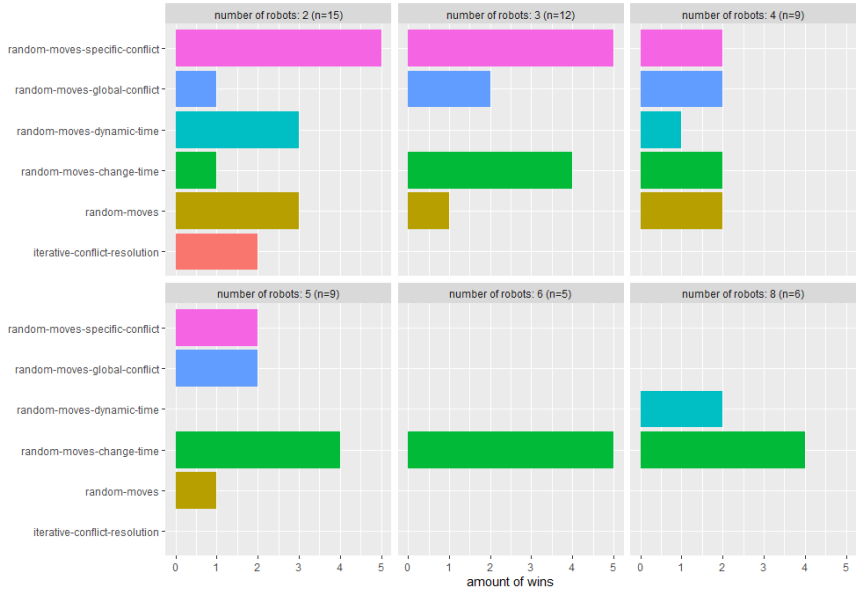

Fig. 6. Amount of 'wins' per approach grouped by the number of robots in instance

introduce the concept of a 'winner' for a benchmark. The "winner" of an instance is the approach with the lowest computation time for a solution. For every benchmark we picked out the 'winner'. We then create facets for each occurring robot count in an instance. Finally we count the amount of wins per approach in each group and this is what you can see in figure 6. At the top right hand side of every sub-plot 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

a lower number of robots 'Random Moves Specific Conflict' seems to work the best. However for more complex instances, it is less likely to be a winner. Once again an exception to this is 'Random Moves Change Time'. If it can solve an instance, it has a respectable chance of being a winner. What this plot does not show is the difference to the other approaches. It is possible that one approach never wins but is just a couple of milliseconds slower than the winner. This plot would then present a wrong impression to the reader. There are also only few benchmarks used in the groups with more robots since we did not have time too add even more. This can definitely manipulate our findings is well. These two things should be kept in mind when looking at the plot.

The last question that has to be answered is which approach a user should rely on. If the setting involves large benchmarks with a lot of robots and it is not crucial that the merger works for every presented case 'Random Moves Change Time' might be the best approach to use. Large instances offer a lot more freedom to the robots to solve their conflicts which is why they can probably manage to find a conflict free path within the time given by their original plans. This is where 'Random Moves Dynamic Time' seems to come short. For large instances a lot of grounding has to happen which makes this approach slower compared to the former. In return 'Random Moves Dynamic Time' seems to work independent of the scenario presented. That means it is especially good for cases where it is important that a solution is found even it takes a lot of time. As a concluding remark we want to say that it is hard to make an exact claim on what the best merger is. We did not have measurements to properly confirm our hypothesis and the amount and variety of benchmarks could also be higher. However, we hope that we could at least present an informative intuition on the performance of each of our approaches in case someone wants to work with one of our mergers in practice.

7 Comparison to other Groups

As part of the internship of the "Lehrstuhl" Knowledge Representation and Reasoning multiple students were tasked to create mergers for the MAPF Problem with k individual agents. Five groups emerged from all the participating students. Each of these with their own final merger that was submitted. The details to the merging strategy each group implemented can be found in their respective paper. Their repositories are linked in the 'References' section. We decided to share the 'Random Moves Dynamic Time' approach since it seems to be the most robust of our mergers while still maintaining a low computation time.

In this section we will compare our end product to the ones of every other team. Sadly there were some issues for us with Tarek Ramadans merger and we did not have enough time to fix them. Out of that reason we will omit his program from our comparison. Details to the performance of his merger can be found in his paper [6].

Each group also shared four benchmarks so that we could test all of the mergers on a common subset of problems. Some of these problems are rather small benchmarks that require the merger to have certain features in order to solve them.

Fig. 7. Corridor Benchmark by Adrian Salewsky

One example for this is a benchmark which was submitted by Adrian Salewsky [9] (Figure 7). A solution can only be found if the merger is able to detect that robot-1 cannot go straight to its goal node but has to dodge away in order to let robot-2 pass. Fortunately every approach we tested was able to detect this case. Thus a solution for this benchmark could be found for each group.

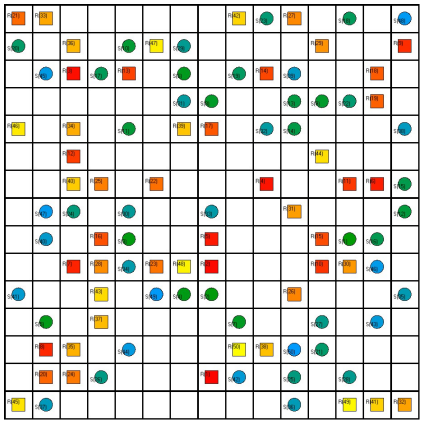

Fig. 8. Large Benchmark by Tom Schmidt, Hannes Weichelt and Julian Bruns

Other benchmarks were a lot larger which means that the solving process takes a lot of computation time. The most difficult benchmark in that regard was submitted by Tom Schmidt, Hannes Weichelt and Julian Bruns [7] (Figure 8). This instance has 225 nodes and 50 robots that each try to reach their respective goal node. The only program that was able to solve this benchmark in an acceptable amount of time was the merger from the group themselves. All of the other mergers failed to find a solution in a acceptable amount of time. We could not determine whether the reason for that were optimization issues of the other approaches, including our own, or if they are truly not able to create conflict free plans.

In order to test all of the mergers on every benchmark we used our 'solver' script. This script starts executing clingo beginning at horizon 1 and iteratively increases the horizon until a solution is found. The difference between the time where we start the iteration and the time after a benchmark is solved is used for the comparison. If an approach took more than half an hour computing on one benchmark without finding a way to create conflict free plans we aborted the process and noted this problem as not solvable for the merger. However, this does not mean that its truly unsolvable for the merging program. There is a chance, especially for the large benchmarks, that the computation time simply did not suffice.

Since we only use one data point per approach and benchmark pair there is some uncertainty regarding the actual time a merger needs to solve such a problem. Different things that run on a computer can influence the solving time and create some variance in the measurements. This should definitely be considered while reading the results.

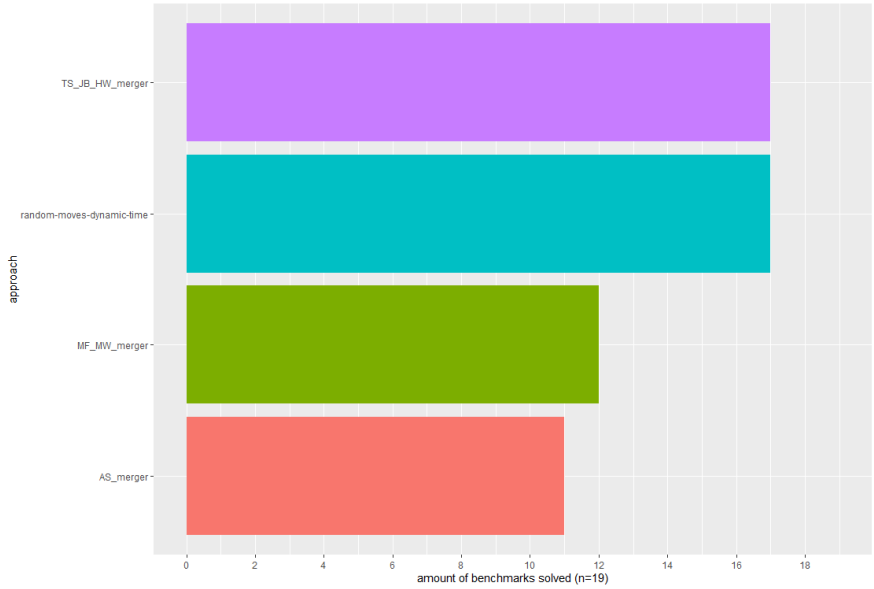

Fig. 9. Number of Benchmarks solved by each Approach

The first thing that we wanted to analyse was the number of benchmarks solved by each approach (Figure 9). This amount describes roughly how good each merger would work in practice. In a realistic environment one would want to have a merging program that works for every scenario. It is clear that the merger of Tom Schmidt, Hannes Weichelt and Julian Bruns [7] as well as our merger work best regarding this stat. Their merger only fails to solve two of all the 19 problems, which are benchmarks that require some additional conflict detection features which they did not implement. The details to that can be found in their report [7]. Our merger in turn does not find a solution for the two

largest instances that use 30 and 50 robots respectively. It is still unclear to us whether this just a time problem or an actual issue with our strategy. The merger of Adrian Salewsky [9] as well as the one of Marcus Funke and Max Wiedehöft [8] each solve more than half of the benchmarks but fail in a lot of cases. This again does not mean that their mergers will not find a solution at all but it was not possible for them to do so within our designated computation time. Still, looking at a setting where it is important to find a solution independent of the benchmark structure the merger of the group of three or ours might be a better fit.

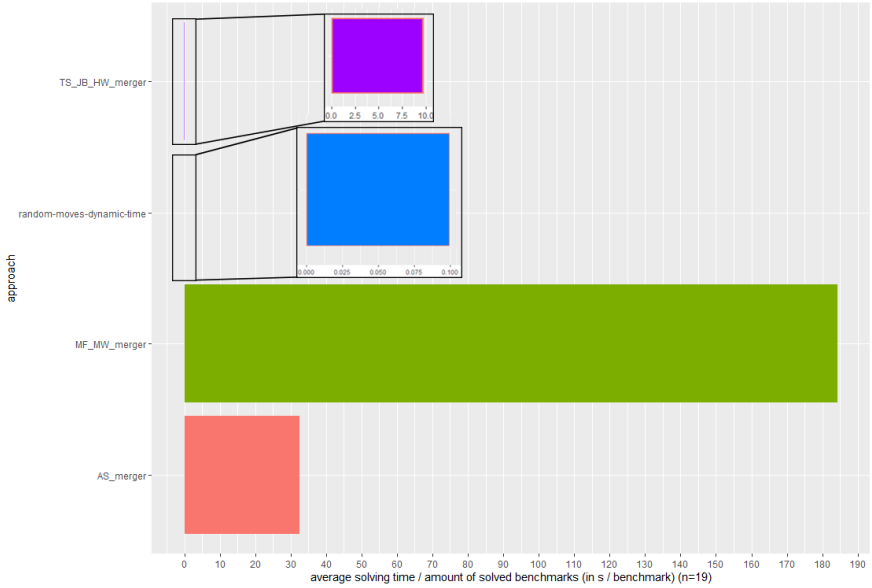

Fig. 10. Average Solving Time per Benchmark

The second point that was important to us was to figure out how fast a merger can find a solution to a benchmark it can definitely solve. For that we computed the average solving time for each approach divided by the number of benchmarks they solved (Figure 10).

Before evaluating the results one should know that each approach was able to solve different benchmarks. If one approach could only solve easy benchmarks reasonably fast while another approach solved only difficult benchmarks, this average solving time can be misleading. Because of that we omitted the two biggest benchmarks which would skew the number of the first merger since this is the only one that can solve it. The rest of the benchmarks have a similar magnitude of time needed to solve them which means that our findings should present a proper intuition.

It is clear to see that the merger of Marcus Funke and Max Wiedehöft [8] performs much worse than Adrian Salewskys [9] program despite solving some more instances. However, they both seem to be a bit unoptimized compared to

the merger of Tom Schmidt, Hannes Weichelt and Julian Bruns [7] and ours. The program of the group of three finds a solution in 10 seconds on average while ours finds one in 0.1. This quite a surprising result and it should be taken with a grain of salt since we did not have time to properly verify all of our measurements. The ones we could verify are correct though. In addition to that, one should also be aware that their merger seems to solve large benchmarks in a reasonable amount of time while ours fails at that task. If they manage to fix their conflict detection issues it might be wise to use their program if you want a product that works for every case. If the goal is to compute a solution as fast as possible and the environment is reasonably small our strategy seems to have the upper hand.

All in all one can see that every single one of the tested mergers work for a lot of the benchmarks. There are, however, clear differences in efficiency and optimization which should be considered when picking the best one is the goal.

8 Future Work

Overall we can say that we got quite far with this project during the semester. Our end product works well and solves most of the problems presented. However, there is still a lot to improve. In this section we want to present different ways for people who are interested in our project to continue with our work.

The main focus is supposed to be 'Random Moves Dynamic Time'. It is our most refined version and should be seen as a solid basis to improve upon. The first thing one might do would be to find out if there is an error with our approach or if it is just too unoptimized to solve large instances such as the benchmark in Figure 8. Based on the answer the approach either has to be fixed first or one could directly try to optimize it. The optimization can be divided into two parts.

The first one has to do with reducing the grounding time. There may be several predicates that can be shortened or even be completely removed if the functionality can be implemented in another way. One example for that would be the 'adj_node' predicate which can be integrated into the choice rule for the position generation (1.4). You can find one attempt at this in our 'test' directory, however we have not proven that all solutions are correct. Before each node with up to four adjacent nodes had to be grounded for every time step. With the adaption that would not be necessary anymore which would save a lot of computation time. Another possibility would be to improve the constraints. Currently edge constraints use four different 'position' predicates to determine a conflict (1.5). It is definitely possible to reduce this number by relying on the moves a robot executes from a certain position instead of using the successor of this position explicitly. There are surely even better solutions. We just wanted to show that there is a variety of ways to reduce the arity of predicates, remove some predicates all together or limit the range of values a variable can take. All of these things would reduce the grounding time by a significant margin and would help to decrease the overall computation time for large instances.

The second part of the optimization concerns the solution quality. We already mentioned that our strategy involves getting the robot back to its original plan after resolving a conflict. This sometimes results in sub-optimal solutions and could even be the problem why some of the large instances were not able to be solved. An idea to improve on that would be to mix 'Random Moves Dynamic Time' with the 'Random Moves Change Time' approach. The merger should first try to solve the conflict within the range of the original plans and only if that is impossible add extra moves to resolve conflicts. It would also be a possibility to not append the whole plan after a conflict occurred but only a part that fits to the position of the robot at the end of the 'change_time' interval. What one could try as well is changing the ordering of the initial moves. For example if a robot wants to go down and then left, the merger could instead try what happens if the robot first goes to the left and then down. It would be possible to add additional strategies on top of this. One of those versions should always find the minimal horizon. Implementing this can get quite complex but the general approach would be to change the way the moves are appended (Listing 1.12)

and the generation of 'before' and 'after' (Listing 1.11). The result would be a merger that behaves at worst like 'Random Moves' in case only the first and last positions are recycled and at best like 'Random Moves Change Time' with a minimal window size if only one move has to be changed for example. This should find a solution in every scenario while still maintaining the best solution quality possible. Especially in combination with the 'Random Moves Dynamic Time Minimize' variation. However, the grounding time might be negatively impacted by offering the merger more freedom in terms of solution finding. There probably has to be a trade off in the end between the two optimization approaches.

But not only 'Random Moves Dynamic Time' can be improved. One could for example allow 'Random Moves Change Time' to find the necessary window size itself by using another choice rule. One could also continue working with 'Iterative Conflict Resolution' and improve it in order to make it work for multiple robots for example. We do not recommend this, however, since this approach is definitely not a good example for a proper ASP program. We also thought about improving the 'Random Moves' merger in another way. We were inspired by the CBS Algorithm of [3]. What would happen if we try out every possible move but reduce the search tree dynamically by 'learning' constraints. Basically the idea is that we still look at every possible move. If one move would cause a conflict between two robots, we generate a new constraint. This constraint ensures that only one of the two robots can take the intended action, while the other has to adapt. It is a 'pass' for one of the robots to take the action and no other robot may step on this position at this time step. Over time we could filter out every possible no-good and only be left with valid solutions. Multiple variants of this approach can be found in our 'wip-encodings' directory. The mergers do run on some instances. However we need a lot of memory to 'remember' all of the constraints. Memory was always a limiting factor for our approaches, however this merger took it to the extreme. It did not have great computation times, while using a lot of RAM. This should be kept in mind if one decides to actually work on this idea.

There are various possibilities to continue working on our project and we would be happy to see other people bringing in new ideas and implementing them in order to create a merger that is as perfect as possible.

9 Conclusion

At the end of this paper we want to conclude our work in this short section. We presented you with an excerpt of what MAPF solvers are supposed to accomplish. Our mergers presented different strategies we tried to apply to solve the MAPF problem. Hopefully they can be of benefit to everyone who wants to work in this domain. Supporting our mergers we also created multiple programs, that can potentially be used in other problem settings as well. Thanks to the internship all of this was possible. We had the opportunity to be creative and try out many interesting ideas for MAPF solvers and even beyond that. For example, as we came up with our own scripts we could learn about the API clingo offers for Python. We also did some data analysis which taught us how to approach our mergers from a statistical point of view. It was a lot of fun for us both and there is a good chance that we will continue working on the things we presented. In case of our merging programs there is no absolute conclusion that can be drawn. No 'k-individual agent merger' seems to be the best on all instances. Some are fast, but cannot solve every instance. Others are slower, but work in general. It would have been interesting to compare our approaches to the original Asprilo solver. Sadly we were missing the time for that. 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. For that, everything we did and will do in the future can be found on our Github repository. [5]

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. Github Repository of Asprilo https://github.com/potassco/asprilo-encodings
5. Github Repository of us (Niklas Kämmer and Marius Wawerek). https://github.com/NikKaem/mapf-project
6. Github Repository of Tarek R. https://github.com/tramadan-up/asprilo-project
7. Github Repository of Tom S., Julian B. and Hannes W. https://github.com/tzschmidt/PlanMerger
8. Github Repository of Marcus F. and Max W. https://github.com/Zard0c/ProjektMAPF
9. Github Repository of Adrian S. https://github.com/salewsky/Plan-Merging