Intelligent Logistics Technology
A plan merging project in Asprilo

Nils Bröcker, Daniel Schreitter Ritter von Schwarzenfeld, Jakob Westphal

Universität Potsdam, 14469 Potsdam, DE
`[nbroecker,schreitterrittervonschwarzenfeld,westphal2]@uni-potsdam.de`

Abstract. In this paper we will present our plan merging project and our approaches to solve it. We used the ASP language and Clingo together with the Asprilo environment to work on this. For every approach we will talk about why it was not used, except for the final one, where we will continue with a set of benchmarks and an evaluation. In the end we managed to find a solution that solves all our benchmark instances, which contain up to 32 robots, in under 3 seconds.

Keywords: Artificial Intelligence · ASP · Asprilo · Logistics · MAPF.

1 Introduction

Our task was to work on a plan merging project for robots (also called 'agents') in the environment of Asprilo using Answer Set Programming. Asprilo here is an environment, were those logistic domains can be specified, containing the grid, representing the storage room and the robots, which are working on jobs they get, where they are going to a product shelf, picking contents up and then bringing it to a packing station. The Asprilo domain here is the simplified m-domain, where robots just have to reach their shelves. Anything beyond this is not relevant, thus objects like picking stations and charging stations do not have a use here. Our domains then contain of a full domain grid, all relevant robots and shelves and for each robot a predetermined plan how they drive from their starting position to their assigned shelf. To keep these plans clear to look onto, a robot with number x will, most of the time, have a plan that determines how it will get to the shelf with the number x. On this path they can have collisions, so points where they would crash with other robots, if the plans would be executed as given. Solving these collisions without generating completely new plans is referred to as Multi-Agent-Pathfinding (MAPF) or plan merging.
In the beginning we will talk about the general problems that had to get solved an the approaches we tried out. Then we will further go into detail about our final encoding, explaining what exactly it does and showing how efficient it works using benchmarks on example instances.

2 Basic problems and first approaches

In general, collisions between two robots can be subdivided into two main collision types. The first one is the node collision, which occurs when two robots want to move onto the same node at the same time stamp (see Fig. 1). These collisions are the easiest to detect as it just needs checking for different robots being at the same node at the same time.

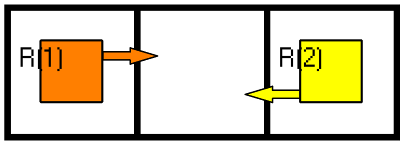

Fig. 1: node collision

The second basic collision is a bit harder to detect, thus also needing more time and space. The so called edge collision occurs when two robots on adjacent nodes want to switch their positions within one time frame, so that after one time frame each robot has the position of the other one (see Fig. 2). As they do not collide on a node, but in between two of these (on the edge of the nodes), it is called an edge collision.

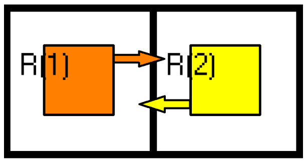

Fig. 2: edge collision

There can be other collisions, but it is possible to sort them into these two main groups. The basic concepts of solving these collisions are consisting of three main approaches:

- waiting
- evading
- plan switching

For the concepts of waiting and evading there has to be a form of a prioritization. We will use a very simple form, where the robot with the number 1 (lowest number) has the highest priority and then a descending prioritization all the way to the robot with the highest number that then has the lowest priority. For plan switching a prioritization is not needed, as we are just changing the plans of both robots. Keep in mind that after plan switching is applied it is not given anymore, even if it was before, that robot x will go to shelf x. For a visualization look onto Figure 3 for Waiting, Figure 4 for Evading and Figure 5 for plan switching.

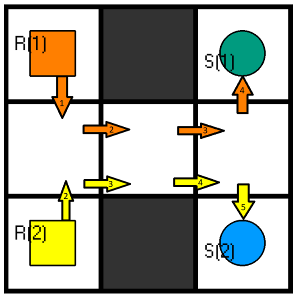

Fig. 3: waiting scenario

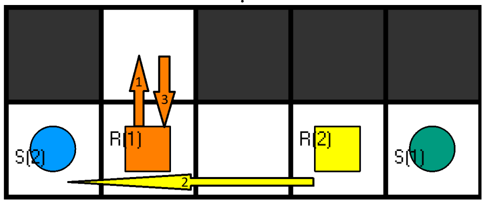

Fig. 4: evading scenario

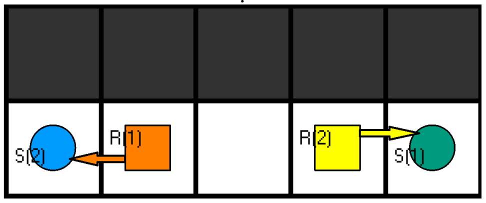

Fig. 5: plan switching scenario

3 Plan merging approaches

3.1 Layer Approach

Our first approach to solve the plan merging task was a layer approach. A robot with a plan that comes directly from the input is located in the first basic layer. If there is a collision detected, then it will get handled depending on the type. Following the collision handling, the new plan is getting pushed into a new layer with $layer_{new} = layer_{old} + 1$. If every collision in layer one was handled, then the program continues with the following layers. These cycles will be repeated until there are no collisions left. Additionally there is a maximum of layers that

can not be crossed.
Solving very small instances with our implementation of this approach worked fine, but even with a light increase of the number of robots, the program took exponentially longer to solve these problems. It was not possible to solve instances with 10+ robots in a reasonable amount of time, most of them never got solved before getting killed in response of running out of RAM. Thus we started developing new approaches.

4 Plan extending approach

The main idea of this approach is, to avoid collision by detaching one robot from his input path, generate one or two steps and glue it together, with the rest of the input path. The approach starts with collision detection and if a collision is found, one agent(the one with the higher number) uses map finding for the next one or two steps (depending on the kind of collision) to avoid the collision, then again entering the input path and continuing his route. The program should be used iterative until no collisions occur any more. We abandoned the idea because first it comes to heavy problems when dealing with corridors, in some instances it was not able to terminate because robots were not able to get back on their input path (i.e. a corridor with two opposing robots A and B on each site when colliding, robot B moves backwards till robot A reaches his goal, robot B is then so far offside, thus it can not reach his input path again).
Second because in certain instance, where a lot of collision occur after manipulating one robot the approach tends to blow up.

5 Virtual robot approach

Within the project we developed the idea of using a virtual robot (also known as meta agents). Like in the previous approaches, at first there is a collision detection. If there are at least two robots, that do not have any collision, then two of them are getting picked and merged together into one bigger virtual robot, that contains both robots and both paths at the same time (the idea is similar to virtual machines on a PC or virtual servers). Then we repeat selecting a robot that has no collision with the virtual robot (which means no collision with every robot inside) and merging it into the virtual robot, until no robot with this condition is left. It follows an iteration over all remaining robots, individually solving the conflict of the current robot and the virtual robot and then merging it into the virtual robot.
The advantage here is that we do not have to care about creating new collisions, as this is not possible here.
Unfortunately we were not able to get a working implementation in Clingo, and concepts like plan switching might be very difficult, if not impossible here.

6 Final Approach

While trying to optimize our layer approach to plan merging, we quickly realized that this approach was very deterministic, while ASP is generally more suited for somewhat non deterministic, declarative approaches. We therefore decided to try a completely new approached based on these ideas. Instead of searching for specific problems and then solving each problem in a specific, predetermined way – which is more the conventional programming way – we decided to first generate many potential solutions and then use constraints and optimization statements to find the solution that best suits us. To achieve this, we first identified that plan switching and waiting are the most efficient ways to solve collisions between robots and are also sufficient to solve any instance. We then decided to handle both of these strategies independently. In a first step plan switching between the robots would be done in a way that not only eliminates all the collisions that can not be solved by waiting but also optimizes the routes of the robots to reduce the total time needed for all robots to reach their goal and also reduce the number of possible collisions between robots. Then, in a second step, waiting would solve all the remaining collisions.

6.1 Plan Switching

The basic approach to the plan switching we developed is very simple. First we define which robots are eligible to switch with which other robots. Then we produce random switches between eligible robots and finally we use optimization criteria to find the best set of switches.

Who can switch and where to switch? Now what exactly does 'eligible to switch' mean for two robots? Following our philosophy of not creating any new movement during plan merging, any two robots who move across the same node at any point during their route can potentially switch their plans. They do not need to reach this node at the same time. The plan switching is done at this common node. When the first robot reaches this node, he takes over the plan that the second robot would follow after reaching this node and vice versa. See figure 6 for an example.

One can easily imagine that it is often the case that two robots have more than just one common node between their plans. In that case we can assume that not all nodes produce an equally optimal solution when used for the plan switching. We decided that, between a pair of robots with multiple common nodes, the first common node that each robot reaches is the best node to use for the plan switching. It is important to note that this is generally not the same node for both robots. For visualization see figure 7.

This is a very important optimization which allows us to shorten the paths of the robots and avoid many potential collisions by finding the optimal position at which to plan switch for each robot. In the figure below a simple example of this

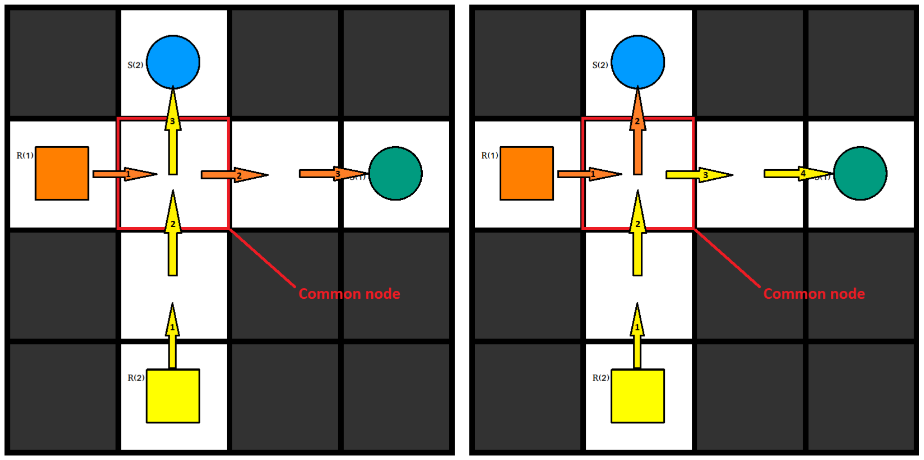

Fig. 6: Plans before switch (left) and after the switch (right) at the common node

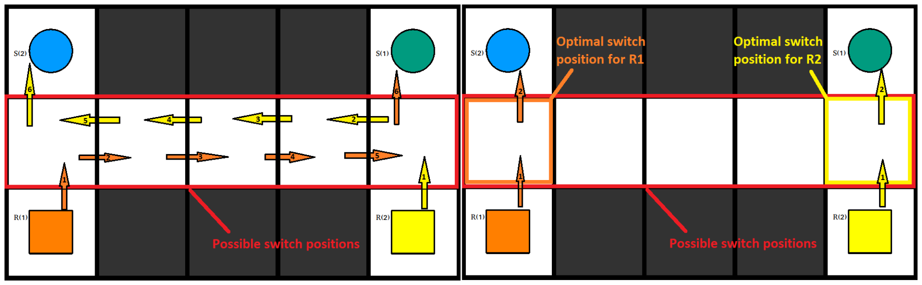
S(2)
S(1)
R(1)
R(2)
Possible switch positions
Optimal switch position for R1
Optimal switch position for R2
S(2)
S(1)
R(1)
R(2)
Possible switch positions

Fig. 7: Plan switch between two robots with multiple possible switch positions. Plans before switch (left) and after an optimal switch (right)

'optimal plan switching' is shown. With our original approach to plan switching both robots would have to enter the corridor first, then turn around and exit the corridor again after the plan switch happens right before the collision. As this new approach is not based on collisions but on common nodes, we can realize a plan switch which avoids that either robot ever enters the corridor. This reduces the length of each robots' plan significantly and also eliminates potential collisions with other robots that have to pass through this corridor.

How to switch? So now that we have found which robots are eligible to switch with which other robots and we have also found where this switch should happen for each robot we can use a simple choice rule to have our encoding choose a random switch for each robot. For this we have purposefully avoided forcing pairwise switching – meaning that if robot nr.1 takes on the plan of robot nr.2 then robot nr.2 has to also take the plan of nr.1 – instead making it possible for each robot to switch individually. This allows for more freedom in the choice of switch partners and can lead to more optimal – as in more time efficient for the robots – solutions. We have to however make sure that two robots can not switch to the same robot and that at the end of the switching each goal is being reached by one robot. This can be done with the help of some simple integrity constraints.

Choosing the best model. The final step is to make sure that out of all the possible solutions the best one is chosen. The most obvious criterion for whether or not a solution is good would be to check if it got rid of all the robot collisions in the plan or at least all the collisions that cannot be solved by waiting. However since we did not need to do any collision detection so far we decided to use a different approach that also is in line with our secondary goal of shortening paths for robots. We simply use a minimize statement, which minimizes the total timesteps for all robots. More specifically: if R is the set containing all robots in the instance and T_r is the time at which robot r reaches his goal we minimize the following value:

$$\sum_{r \in R} \sum_{n=1}^{T_r} n = \sum_{r \in R} \frac{T_r(T_r+1)}{2} \tag{1}$$

This means shorter paths are better, but it also penalizes paths more the longer they are meaning that two robots each reaching their goal in 6 timesteps (optimization 42) is better than one robot reaching it in 2 timesteps and the other in 10 (optimization 58). This is in our opinion generally speaking the preferred behavior and it also helps solve a problem with a specific type of collisions: what we called a sleeping collision.

A **sleeping collision** happens, when one robot has already reached his goal and is therefore stationary for the rest of the time and another robot wants to pass through the node on which the first robot is standing. This collision falls under the category of node collisions, but it requires special treatment one of the robots involved has already reached the end of his plan and has therefore a special status. Additionally, contrary to normal node collisions, this type of collision is best solved by plan switching. By having the two robots involved switch their plans the collision can be avoided and the plan length can even be reduced by one step while waiting will always increase the plan length. Our specific optimization criterion forces a plan switch in this situation. For an example look at figure 8

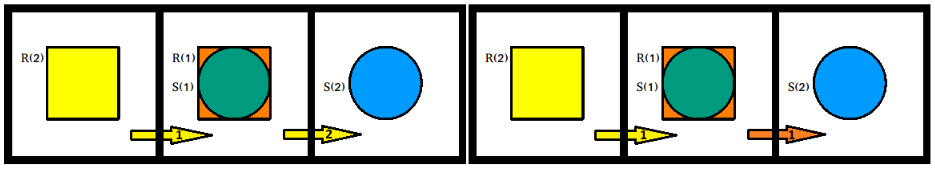

Fig. 8: Example of a sleeping collision and its solution via plan switching

We have found that using this optimization criterion during plan switching yield solutions that are now completely solvable using only waiting and that are also significantly more optimized than the original plan.

6.2 Waiting

Similarly to our new approach to plan switching we also decided to try a more declarative approach for waiting. Instead of looking for collisions and then solving those collisions by having one robot wait, we decided to simply assign a random wait time – limited by a max wait parameter – to every robot. The robots then have to wait for the assigned time and only start executing their plans once their wait time is over. This way we create a lot of models with differing wait times and then use constraints to eliminate any models that still have any kind of collisions. Finally we again use optimization statements to make sure no unnecessary waiting is introduced.
This approach relies on simply guessing the correct number of waits for each robot so that no more collisions are happening. Another big difference to our original approach is that here all robots will wait at their starting position and only start to move afterwards. Once they have started moving they will no longer stop at any point to wait. Because the robots no longer wait at some points in the middle of their route this should generally reduce the amount of additional collisions created from robots crashing into waiting robots in their path. While the starting positions of each robot can also be in the path of other robots it is reasonable to assume that they are more towards the edge of the instance / in more remote spots. In comparison, collisions generally happen in spots that are very busy and therefore waiting right in front of such a spot is more likely to generate additional collisions. Because of this we believe that this approach also supports our secondary goal of having the shortest paths possible.

6.3 Optimization

Once we implemented this new approach it quickly became obvious that while it was more performant than our layer approach, it was still not scaling very well for bigger instances. The main problem was for both the plan switching and the waiting encoding that the amount of possible models was simply too high and therefore the solving process of finding the optimal model would take very long. To fix this we had to constrain the process of generating models by introducing more restrictions.

Plan switching optimization
For the plan switching encoding we realized this by adding collision detection back in. Unrestricted plan switching between any two robots is necessary to create optimal results but for the primary goal of getting rid of all collisions that cannot be solved by waiting it should be enough to only switch plans between robots that collide with each other in one of 3 ways: a ‘normal’ edge collision, a

'extended' edge collision where two robots switch places over two timesteps and a sleeping collision (see figure 9).
Unfortunately, specific instances exist where just a single switch between robots

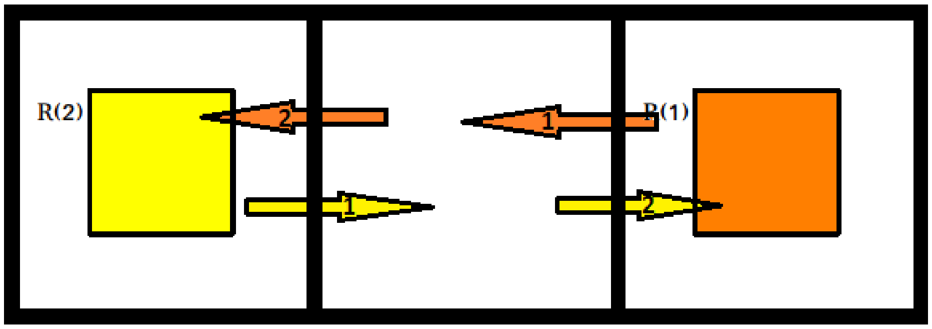

Fig. 9: Example of an extended edge collision. This could also be classified as a node collision but can not be solved via waiting!

with these collisions is not sufficient to eliminate all collisions. However running the same encoding again, therefore allowing for the robots to switch a second or third time solves this problem. So even though this meant that we had to switch to a multi-shot approach and were also sacrificing some of the optimality of the resulting plans, we decided that the increase in time performance was more important.
Upon further testing we noticed that the first application of the plan switching would take the huge majority of the total solving time compared to the subsequent ones. This makes sense since the first application would generally get rid of 90 to 100 percent of all collisions and the following applications would only need to solve very few collisions. To alleviate this, we switched from running the exact same encoding multiple time to running slightly modified versions of the encoding sequentially. The first encoding only contains collision detection for the normal edge collision and not the other two – meaning that there are a lot less possible switches and therefore the time performance increases significantly. The second encoding contains both the normal edge collision as well as the extended edge collision. The final encoding contains all 3 collisions. The full plan switch for a problem is then done by first running the first encoding once, then the second once and finally the third encoding at least once and then until the best solution is found (i.e. the optimization does not get better any more). This means we use the simpler encodings to reduce the complexity of the problem and then use the more powerful but less efficient encodings once the instance has decreased in complexity, leading to both a decent time performance and a reasonably optimized solution.

Waiting optimizaiton

In order to optimize our waiting encoding we used a very similar strategy. The main factor that influences the time performance in this case is the max wait parameter since for every robot every possible wait time has to be checked – the higher the max wait parameter the more possible different wait times ((Number of robots)^{maxwait} possibilities to check). Yet while most instances only require a low max wait parameter once plan switching has been done, there are specific instances where plan switching can not change anything and a very high max wait parameter is needed. Hence we decided to use a multi-shot approach where we use an encoding with a smaller max wait parameter multiple times.
The idea is that the a result where one robot waits for 10 timesteps can either be achieved by using a max wait of 10 or by sequentially running an encoding with a max wait of 5 twice, but the latter has way better execution time in our case ($x^{10} > 2 * x^5$ for $x > 1$) We pushed this concept to the limit by only allowing a max wait of 1 per execution of our waiting encoding.
This did cause some problems though. First we had to switch from integrity constraints that eliminated any models that still had collisions in them to minimize statements that minimized the number of remaining collisions in each application of the encoding. Secondly reducing the max wait parameter means reducing the 'foresight' of the encoding. Similarly to the optimization for the plan switching, this means the result is potentially not as optimized as the single-shot approach. This is because the optimization happens for every single iteration. A specific distribution of waits among the robots might minimize the amount of collisions in the first iteration but across all iterations a different distribution might have lead to a more optimal solution. This loss in optimization can unfortunately not be avoided and we considered it necessary in order to achieve good time performance.
One last optimization we used is using two slightly different encodings for waiting. One in which any robot can be assigned a wait and another one in which only those robots that are involved in a node collision are allowed to wait. The first encoding is more precise and finds potentially more optimal solutions but it is slower, the second is simpler and thus faster. In this case we have to apply the slower encoding first and can only use the faster one once the first encoding has been run a few times. This is due to the problem with lacking 'foresight' mentioned above. Because of this the more optimal encoding has to first lay a solid foundation before the second encoding can start working otherwise it might lead to a situation where some collisions cannot be eliminated. We have found that for all our benchmarks, running the first encoding twice is enough and afterwards we can run the simpler encoding repeatedly until no more collisions are left.

Finally we used a relatively simple python script to join together both the plan switching and the waiting. It uses the clingo python API to simply sequentially run the encodings in the order that was specified above.

7 Benchmarks

In Table 1 we present our benchmarks. They consist of the input instance name and the full time in seconds needed for computation. The instance names are given as Xr_name, where X is the amount of robots in this instance. It is noticable that with rising amounts of robots, the time needed for computation will also rise in general. But as it can be seen clearly for some examples like $15r\_node$, it is also very important how the instance is structured and the robots are placed. These examples that need especially long compared to other instances with the same amount of robots, are consisting mainly of waiting problems, that need to be solved by our waiter merger. This waiter merger is far less time efficient than our planswitch mergers, thus resulting in longer computation times. For visualization of all benchmarks see Figure 10, where you can clearly see the two main spikes, which are heavy node collision instances. To better see the differences on the other benchmarks see Figure 11, where the two extraordinary high spikes $(2-3s)$ got excluded, because all other benchmarks are below $0.35s$.

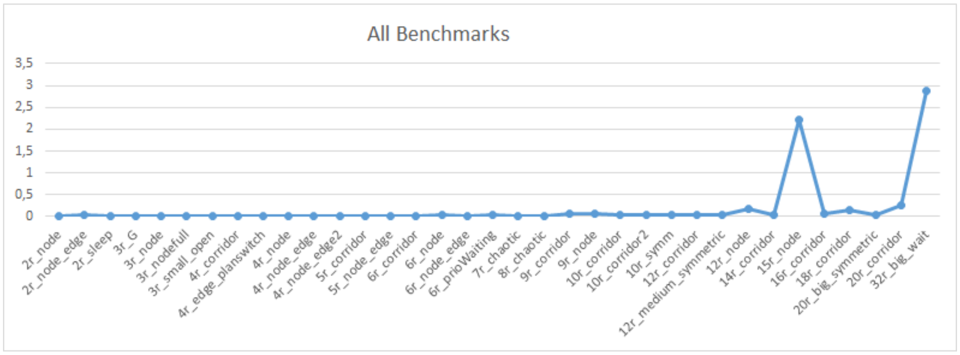

Fig. 10: All Benchmarks (x: name, y: time in seconds)

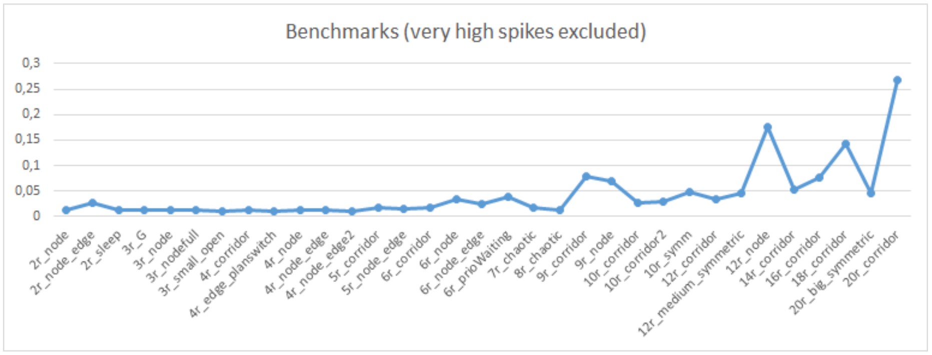

Fig. 11: Benchmarks, but the two highest spikes excluded (x,y as above)

Table 1: Benchmarks

Instance name	Time
$2r\_node$	0,014
$2r\_node\_edge$	0,026
$2r\_sleep$	0,012
$3r\_G$	0,014
$3r\_node$	0,014
$3r\_nodefull$	0,014
$3r\_small\_open$	0,010
$4r\_corridor$	0,014
$4r\_edge\_planswitch$	0,010
$4r\_node$	0,013
$4r\_node\_edge$	0,014
$4r\_node\_edge2$	0,011
$5r\_corridor$	0,017
$5r\_node\_edge$	0,016
$6r\_corridor$	0,018
$6r\_node$	0,033
$6r\_node\_edge$	0,025
$6r\_prioWaiting$	0,038
$7r\_chaotic$	0,018
$8r\_chaotic$	0,013
$9r\_corridor$	0,079
$9r\_node$	0,070
$10r\_corridor$	0,028
$10r\_corridor2$	0,029
$10r\_symm$	0,048
$12r\_corridor$	0,035
$12r\_medium\_symmetric$	0,045
$12r\_node$	0,176
$14r\_corridor$	0,053
$15r\_node$	2,228
$16r\_corridor$	0,076
$18r\_corridor$	0,143
$20r\_big\_symmetric$	0,047
$20r\_corridor$	0,267
$32r\_big\_wait$	2,876

8 Evaluation

In our project we had many different ideas how to approach the solution of this plan merging problem. Most of them were either inefficient or we were not able to implement them in ASP, but in the end we managed to find and implement an approach that works extremely efficient with plan switching problems (biggest benchmark here is with 20 robots and only took under 0.3 seconds) and definitely fine node collision problems, even if its remarkable slower than the plan switching problems (biggest benchmark here is with 32 robots and was solved in under 3 seconds). Thus all of our benchmarks were solved in under 3 seconds and the biggest benchmark even was a node collision problem.
In our opinion the plan switching solution is pretty optimized and is also scaling very good. The node collision solution works much slower and tended to scale bad that instances with 32 robots could not be solved or in over 10 seconds, but we managed to find a solution for this, thus it seems to scale fine now, but as our biggest benchmark is with 32 robots we can obviously not say anything about bigger instance, as it would be possible that it scales linear (would be good) as well as exponential (would be bad), if the breaking point is beyond 32 robots.

References

1. Ma, H.; König, S.: Optimal Target Assignment and Path Finding for Teams of Agents (2016). https://arxiv.org/abs/1612.05693
2. Asprilo Homepage, https://asprilo.github.io/. Last accessed 7 March 2022
3. Asprilo Git, https://github.com/potassco/asprilo. Last accessed 7 March 2022
4. Asprilo-Encodings Git, https://github.com/potassco/asprilo-encodings. Last accessed 7 March 2022