Multi Robot Plan Merging Project

Project report

by

Tom Schmidt
796970

Hannes Weichelt | **Julian Bruns**
798247 | **796290**

Under the supervision of

Etienne Tignon

INSTITUTE OF COMPUTER SCIENCE

UNIVERSITY OF POTSDAM

WISE 2020 - 2021

DECLARATION

Project Title	Multi Robot Plan Merging Project
Authors	*Tom Schmidt*, *Hannes Weichelt*, and *Julian Bruns*
Student IDs	796970, 798247, and 796290
Supervisor	Etienne Tignon

We declare that this project report titled *Multi Robot Plan Merging Project* is the result of our own work except as cited in the references.

Tom Schmidt
796970

Institute of Computer Science
University of Potsdam

Hannes Weichelt
798247

Institute of Computer Science
University of Potsdam

Julian Bruns
796290

Institute of Computer Science
University of Potsdam

Date: March 23, 2021

Contents

Chapter 1

Introduction

1.1 General Setting

This project deals with the topic of warehouse logistics, specifically a warehouse storage system that is managed by robots. Such a warehouse consists in essence of multiple shelves, which contain certain products and some robots. These robots are responsible for delivering shelves that contain ordered products to one of the many possibly picking stations, where the products can be dropped off and prepared for shipping etc.

1.2 Asprilo

Asprilo[1] is a benchmarking framework which is perfectly suited for modelling these intra-logistic processes and also automating them with ASP. It was created by Philipp Obermeier and Thomas Otto in 2017 and since then has been updated several times up to version 0.3.0 in July 2019, which we are using for this project. With this framework we are able to model a warehouse layout, composed of nodes, with all its components (shelves, robots, etc.) and visualize it and the processes in it graphically. But most importantly of all with the help of clingo[2], an ASP solver that is part of Potassco (Potsdam Answer Set Solving Collection)[3], we are able to automatically generate plans for the warehouse robots to fulfill their orders and solve all the connected tasks efficiently.
Here the automatic plan creation is separated into 4 domains:

1. The A-Domain which contains the full problem setting in its original form as described above.
2. The B-Domain which simplifies the setting of A so that product quantities are completely disregarded.
3. The C-Domain which simplifies the B-Domain in the way that at a picking station all of a robots deliveries can be processed in one time step in parallel.
4. And finally the M-Domain which is a further simplification that only focuses on the movements of the robots to their destination shelves. This setting thereby also enforces these simplifications[1]:
   (a) the number of robots and orders is identical
   (b) the number of shelves and products is identical
   (c) for each product, there is globally only one unit stored on a single shelf
   (d) each order has exactly one order line that requests exactly one unit of a product
   (e) there are no two orders that request the same product

In this project we will mainly focus on the M-Domain and only touch on the higher domains in an exploratory way.

1.3 Problem Setting : Plan Merging

As good as the automatic plan generation of Asprilo is there is one crucial disadvantage. The planning always takes into account all the robots in a warehouse and so has to deal with all their conflicts. This leads to constant re-planning to achieve an optimal solution, which expands the search space dramatically. As a result the computation time can reach astronomical heights, especially for big instances.

Here comes the idea of plan merging into play. Instead of going for an optimal solution that keeps all the robots and their respective plans in mind, we create a simple one-robot-plan for each robot that disregards the other robots and then afterwards merge these plans into one.

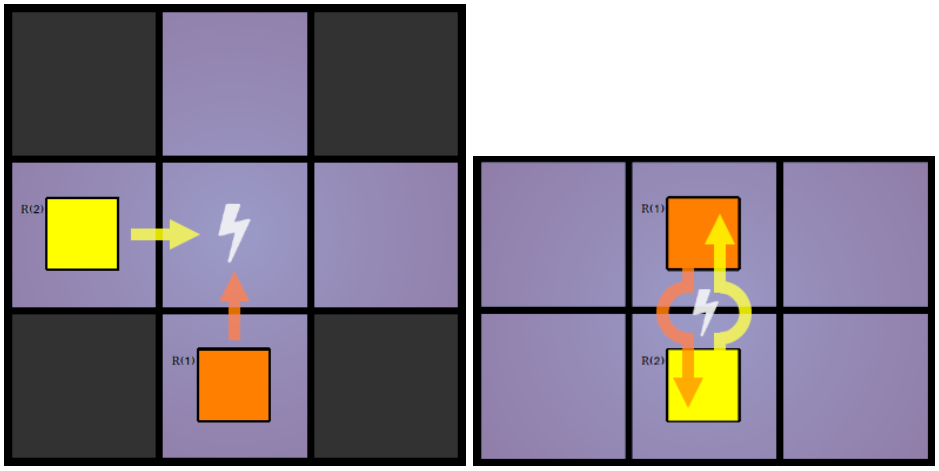
While merging we have to lookout for possible conflicts that can now arise due to the previous mutual disregard between the robots. These conflicts can mainly be summarised by two typical conflict patterns.

1.3.1 Vertex-Conflicts

A vertex conflict occurs when multiple robots try to move onto the the same node at the same time (Figure 1.1a). In a real life scenario would result in a crash of the two robots.

1.3.2 Edge-Conflicts

Edge conflicts on the other hand occur when two robots face each other on the edge between two nodes and then try to switch places in the next time step (Figure 1.1b). Like the vertex conflicts this would result in a crash.



(a) Vertex conflict (b) Edge conflict

Figure 1.1: Possible conflicts

Chapter 2

Merger

To solve the problem described in Section 1.3, we split the M-domain merging problem into 4 smaller sub-problems.

1. Vertex conflicts, described in Section 1.3.1.
2. Edge conflicts, described in Section 1.3.2.
3. Out of bounds moves. Robots are not allowed to make invalid moves, for example moving into a wall.
4. Compliance with the time limit (horizon). Each robot has to reach its goal within the given time limit.

While working on the sub-problems we found two approaches for our merger. The first one being an iterative approach[1]. Here the main idea is to detect and solve conflicts in the original plans through multiple iterations. The second approach[2] interprets the merging problem as one big Constraint Satisfaction Problem (CSP), with the above mentioned sub-problems as its constraints. Our tests showed that the CSP approach can solve more benchmarks and scales better, which is why we focused our work on that approach. In the following sections both approaches are described in a more detail.

[1] https://github.com/tzschmidt/PlanMerger/tree/main/encodings/merger/approach1
[2] https://github.com/tzschmidt/PlanMerger/tree/main/encodings/merger/approach2

2.1 Merging via iterative approach

As already stated above, in our first approach we used a conflict based method, that detects and then solves conflicts in an iterative way. Detecting conflicts only with the starting positions and the relative movements of the robots can be quite difficult, so we decided to create statements, that represent the absolute position for each robot. These ”position”-statements hold the spatial and temporal coordinate of each robot, but also the so called “conflict depth”. The reason why we need a “conflict depth” attribute is that, in an iterative approach such as ours, we solve conflicts by giving robots a new path. In other word we need to create new ”position”-statements and at the same time render older ones invalid. In ASP however, true statements can not be rendered false again, so we use a conflict depth of +1 to denote the new path of the robot.

Listing 2.1: Relative Positions (mergeV2.lp)

```
55 % Initializing absolute positions of the robots
56 position(R,(X,Y),0,0) :- init(object(robot,R),value(at,(X,Y)))
      .
57 position(R,(X+DX,Y+DY),T,0) :- occurs1(object(robot,R),action(
      move,(DX,DY)),T), position(R,(X,Y),T-1,0).
```

To make sure that our merger will terminate in a reasonable time, we use two kinds of constraints. First a time constraint that disallows movements after the time horizon. This horizon is dependent on the respective benchmark. Secondly we use a constraint for our conflict depth, which we will explain later on. All tested benchmarks work with maximum conflict depth of 4, which prevents unjustifiable benchmarks from using to much computation time.

Listing 2.2: Time and conflict depth constraints (mergeV6.lp)

```
 4 % Time limit
 5 time(0..horizon).
 6 % Conflict depth limit
 7 depth(0..4).
 8
 9
10 % No moevement is allowed after time horizon
11 :-position(R,(X1,Y1),T,C), position(R,(X2,Y2),T-1,C), (X1,Y1)
      !=(X2,Y2), not time(T).
```

2.1.1 Vertex Problem

These absolute position statements allow us to detect both collision types much more easily. For the detection of vertex collisions, we simply check if two different robots have the same spatial coordinates at the same time. The tricky question now is which one of the robot paths we need to change via our "wait"-statement. In our first version we simply let the robot with the lower ID wait.

Listing 2.3: First idea for wait statements (mergeV2.lp)

```
60 % Add a wait statement if two robots are at the same node at
      the same time
61 wait(R1,T,CONFLICT_DEPTH_1) :- position(R1,(X,Y),T,
      CONFLICT_DEPTH_1), position(R2,(X,Y),T,CONFLICT_DEPTH_2),
      R1<R2.
```

To create the new path for a robot from our wait statements, we need to create a new set of positions, that have a conflict depth of +1. This new set also needs to have a slight alteration compared to the older set of positions to accommodate for the waiting robot. Both can be done with the following code.

Listing 2.4: Generate a new path where the robot waits at one point (mergeV2.lp)

```
65 % Generate a full new position map with one of the robots
      waiting before the conflict occurs
66 position(R,(X,Y),T,CONFLICT_DEPTH+1) :- position(R,(X,Y),T,
      CONFLICT_DEPTH), wait(R,T_WAIT,CONFLICT_DEPTH), T<T_WAIT.
67 position(R,(X,Y),T+1,CONFLICT_DEPTH+1) :- wait(R,T+1,
      CONFLICT_DEPTH), position(R,(X,Y),T,CONFLICT_DEPTH).
68 position(R,(X,Y),T+1,CONFLICT_DEPTH+1) :- position(R,(X,Y),T,
      CONFLICT_DEPTH), wait(R,T_WAIT,CONFLICT_DEPTH), T>=T_WAIT.
```

While this works fine a lot of the time, we later learned that there are a number of situations in which vertex collisions still occurs. One of these situations is illustrated in Figure 2.1. (Note that this example illustrates a vertex collision problem, and as such, we ignore all edge collisions).

This problem makes it apparent, that if our merger wants to solve more complex vertex collisions we need to allow it, to decide which of the involved robots should wait. One way this could be done is by creating a set of the two possible wait statements and letting the program choose one of them via the following code.

Listing 2.5: Secound idea for wait statements

```
% Add a wait statement if two robots are at the same node at
   the same time
{wait(R1,T,CONFLICT_DEPTH_1);wait(R2,T,CONFLICT_DEPTH_2)}=1 :-
    position(R1,(X,Y),T,CONFLICT_DEPTH_1), position(R2,(X,Y),T
   ,CONFLICT_DEPTH_2), R1<R2,time(T),depth(CONFLICT_DEPTH_1),
   depth(CONFLICT_DEPTH_2).
```

If we then add a constraint that makes it impossible for multiple robots to occupy the same space at the same time, the merger should in theory be able to chose the better of the two wait statements.

Listing 2.6: Vertex constraint (mergeV6.lp)

```
% No two robots occupie the same space at the same time
:-position(R1,(X,Y),T,C1),position(R2,(X,Y),T,C2),R1<R2,
   max_conflict_depth(R1,C1),max_conflict_depth(R2,C2).
```

Unfortunately this approach creates a new problem. If we have, for example, 3 robots which collide with each other, like we had in Figure 2.1a, the merger would detect 3 vertex collisions. One between robot 1 and 2, one between robot 2 and 3 and the last one between robot 3 and 1. But because the merger can freely chose one from each pair, it is possible, that the merger chooses all 3 of them. In that case all 3 robots would wait for each other forever.

This, however can be solved by simply splitting the problem into 3 cases, depending on if 2, 3 or 4 robots collide and then giving every robot but one a wait statement.

Listing 2.7: Wait statements (mergeV6.lp)

```
23 % Our porgramm creates a variable amount of wait statements
      depending on how manny robots collide
24 {wait(R1,T,C1);wait(R2,T,C2)}<=1:-
25 {position(R,(X,Y),T,C)}=2,position(R1,(X,Y),T,C1),position(R2
      ,(X,Y),T,C2),R1<R2,time(T),depth(C),depth(C1),depth(C2).
26
27 {wait(R1,T,C1);wait(R2,T,C2);wait(R3,T,C3)}<=2:-
28 {position(R,(X,Y),T,C)}=3,position(R1,(X,Y),T,C1),position(R2
      ,(X,Y),T,C2),position(R3,(X,Y),T,C3),R1<R2,R2<R3,time(T),
      depth(C),depth(C1),depth(C2),depth(C3).
29
30 {wait(R1,T,C1);wait(R2,T,C2);wait(R3,T,C3);wait(R4,T,C4)}<=3:-
31 {position(R,(X,Y),T,C)}=4,position(R1,(X,Y),T,C1),position(R2
      ,(X,Y),T,C2),position(R3,(X,Y),T,C3),position(R4,(X,Y),T,C4
      ),R1<R2,R2<R3,R3<R4,time(T),depth(C),depth(C1),depth(C2),
      depth(C3),depth(C4).
```

With this, we have a pretty useful solver for vertex constraints and are finally able to solve our problem. A possible solution can be seen in Figure 2.2.

2.1.2 Edge Problem

Now we can take a look at our second big problem, the Edge collisions. First we need to detect the collisions themselves, which is, thanks again to our absolute position statements, relatively easy. We simply check if two different robots change their positions after one time frame and then, check if the relative movement has an absolute value of exactly one. We need to check the latter in order to confirm, that both robots are next to each other and move towards each other. Now we want to, in a similar way to the vertex problem, create a set of possible ”edge”-statements and let the merger chose exactly one of them.

Listing 2.8: Edge collisions (mergeV6.lp)

```
44 %chose one of 4 possible edge statements if two robots try to
      switch places at the same time
45 %the 4 possible edge statements differ only in which of the 2
      robots dodges and in which direction
46 {edge(R1,(X2-X,Y2-Y),T+1,CONFLICT_DEPTH_1);edge(R2,(X2-X,Y2-Y)
      ,T+1,CONFLICT_DEPTH_2);
47 edge(R1,(-(X2-X),-(Y2-Y)),T+1,CONFLICT_DEPTH_1);edge(R2,(-(X2-
      X),-(Y2-Y)),T+1,CONFLICT_DEPTH_2)} =1 :-
48 position(R1,(X2,Y2),T,CONFLICT_DEPTH_1),position(R2,(X,Y),T,
      CONFLICT_DEPTH_2),
49 position(R1,(X,Y),T+1,CONFLICT_DEPTH_1), position(R2,(X2,Y2),T
      +1,CONFLICT_DEPTH_2),
50 R1<R2,dir(X2-X,Y2-Y),time(T),depth(CONFLICT_DEPTH_1),depth(
      CONFLICT_DEPTH_2).
```

To solve the edge collisions, we create a dodge maneuver for each robot whose ”edge”-statement holds true.

Listing 2.9: Edge maneuver (mergeV6.lp)

```
100 % Generate a full new position map where one of the robots
      does a dodge maneuver
101 %path before the edge conflict
102 position(R,(X,Y),T,CONFLICT_DEPTH+1) :- position(R,(X,Y),T,
      CONFLICT_DEPTH), edge(R,(I,J),T2,CONFLICT_DEPTH), T<T2.
103 %dodge maneuver
104 position(R,(X+J,Y+I),T,CONFLICT_DEPTH+1):- position(R,(X,Y),T
      -1,CONFLICT_DEPTH), edge(R,(I,J),T2,CONFLICT_DEPTH), T=T2.
105 position(R,(X,Y),T+1,CONFLICT_DEPTH+1) :- position(R,(X,Y),T
      -1,CONFLICT_DEPTH), edge(R,(I,J),T2,CONFLICT_DEPTH), T=T2.
106 %path after the edge conflict
107 position(R,(X,Y),T+2,CONFLICT_DEPTH+1) :- position(R,(X,Y),T,
      CONFLICT_DEPTH), edge(R,(I,J),T2,CONFLICT_DEPTH), T>=T2.
```

Besides the robot ID, the time and the conflict depth of each edge statement also holds the parameter ”(I,J)”. This is used to denote in what direction the robot can dodge. If the edge collisions occurred in the x-direction ”(1,0)” or ”(-1,0)” the robot can only dodge in y-direction ”(X,Y+1)” or ”(X,Y-1)”. This is also the reason why we have 4 possible edge statements. (We have two possible robot which can each

increment or decrements their x-coordinate, or their y-coordinate depending on the edge statement).

There is still a problem though. Because this dodge maneuver can move robots outside their original path, we need a constraint that makes it impossible for a robot to not stand on a node at any given time. This can be done via the following code.

Listing 2.10: Node constraint (mergeV6.lp)

```
71 %a robot can't move to a space without a node
72 :-position(R,(X,Y),T,CONFLICKT_DEPTH), not init(object(node,_)
      ,value(at,(X,Y))).
```

Now we have, just like before, created a new constraint that makes invalid "edge"-statements impossible and helps our merger to chose a better "edge"-statement.

2.1.3 Output

Now that all of the detected collisions have been resolved, we can transform our absolute positions back into relative movements. But first, we have to find the highest conflict depth of each robot. This is needed so that we use the final positions and not older, invalid ones. We do this by adding all the different conflict depth variables for each robot together and then saving that value via the "max conflict depth" statement. Note that we need decrement the value first by one to account for the fact that the starting conflict depth is 0.

Listing 2.11: Max conflict depth (mergeV6.lp)

```
76 % OUTPUT
77 % calculate the max conflict depth for each robot
78 max_conflict_depth(R,MAX_D-1) :- init(object(robot,R),value(at
      ,(X,Y))), MAX_D == #sum{1,D:position(R,(X_TMP,Y_TMP),T,D)}.
```

Because we now have the highest conflict depth for each robot, we are able to detect the final positions of each robot. With that we can simply transform the absolute positions into relative movements which are represented via the "out"-statements. Finally we simply transform the out statements into the occurs notation and output it via a simple show statement.

Listing 2.12: Occurs statements (mergeV6.lp)

```
79 % get the absolute positions of the last confict depth for
      each robot
80 final_position(R,(X,Y),T) :- position(R,(X,Y),T,D),
      max_conflict_depth(R,D).
81
82 % CONVERT
83 % generate relative movements
84 out(R,(X_2-X_1,Y_2-Y_1),T) :- final_position(R,(X_2,Y_2),T),
      final_position(R,(X_1,Y_1),T-1), T>0.
85
86 %generate output in the "occurs" format
87 occurs(object(robot,R),action(move,(X,Y)),T) :- out(R,(X,Y),T)
      .
```

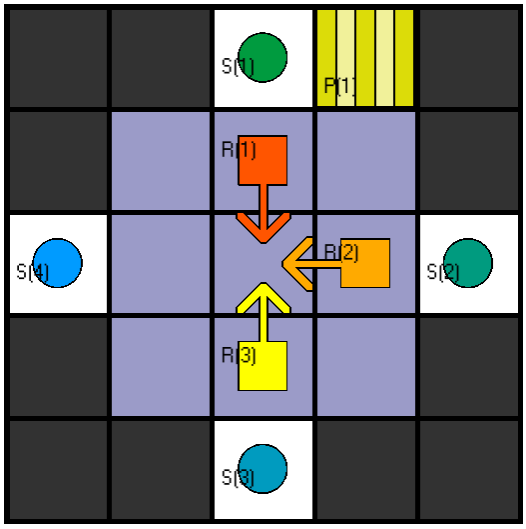
2.1.4 Evaluation

Relatively early in our development, we decided to simultaneously develop a different approach. There where two main reasons for this decision. One reason is, that, because our approach is iterative, we need the conflict depth variables. These however make our program a lot more complicated and slow because we effectively add a new dimension to the problem. Now, we have to deal with a time, a space and a conflict depth component, which creates a lot more possibilities for our positions statements which slows the program in certain situations down.

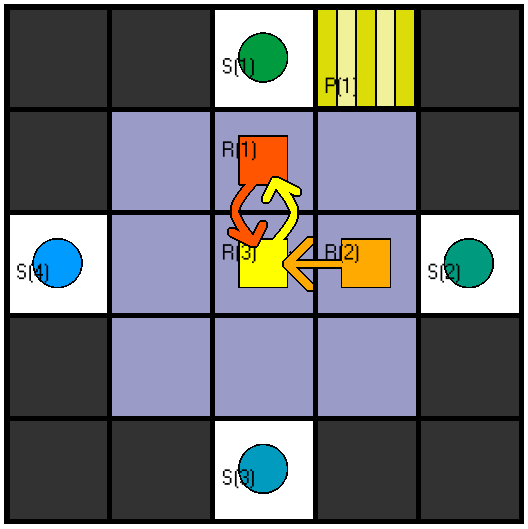
The second problem is that our merger reacts only directly before a collision occurs via edge or wait statements. This means that problems, where a robot would, for instance, need to wait a few time frames before the collision occurs, aren't solvable. A good example for this is our benchmark 6, in which two robots collide in a small tunnel and one of the robots needs to wait before entering the tunnel to let the other robot pass.

While our first approach has a pretty big handy cap because of this, it still manages to solve 12 out of 18 of our benchmarks. while being very slow on some benchmarks, like benchmark 3, it can also solve most of our benchmarks in a reasonable time.

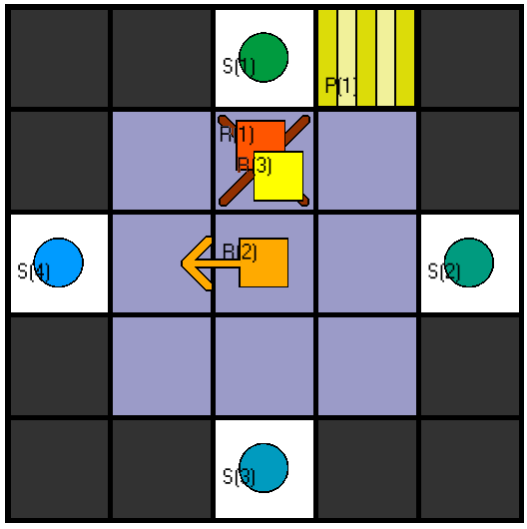
In summary, our first approach is being held back by it's iterative structure, but is still able to solve a good number of problems a reasonable time.



(a) 3 robots would collide, so the two robots with lower ID wait.

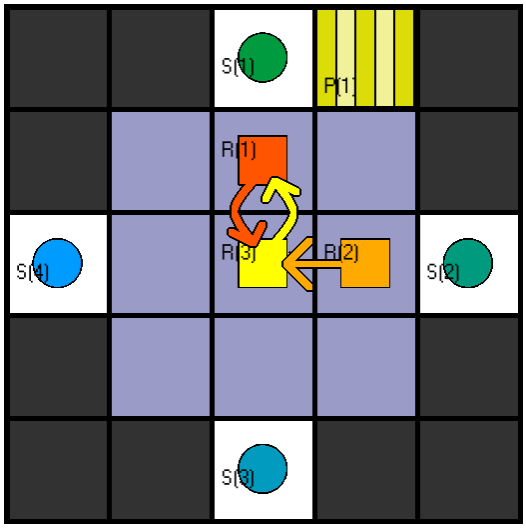


(b) Robot 1 and 2 would collide so robot 1 waits. But because of this robot 1 and 3 would collide but robot 3 can't wait because it has the higher ID

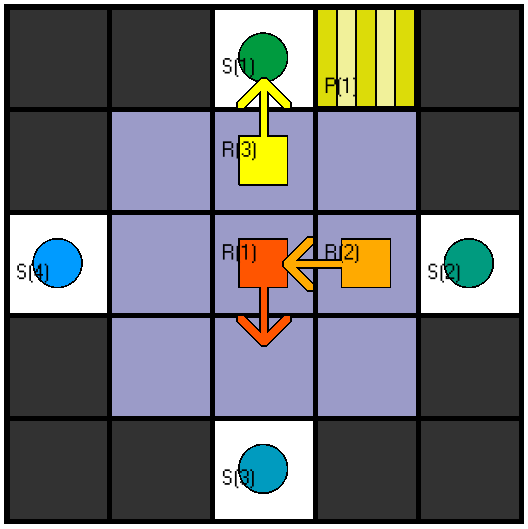


(c) Edge collision between robot 1 and 3

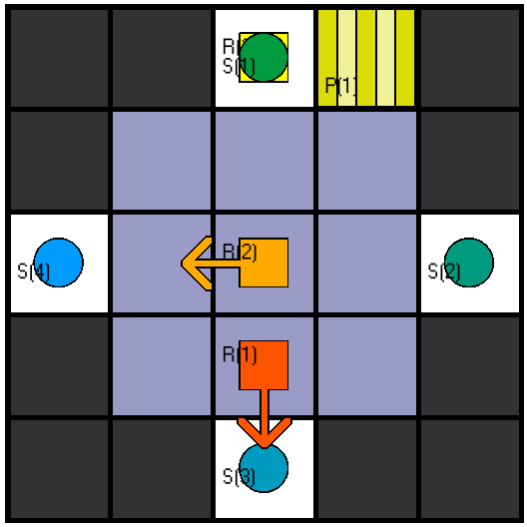
Figure 2.1: Vertex collision problem



(a) Robot 1 and 2 would collide. This time robot 2 is able to wait.



(b) Note that robot 1 and 2 can change place because we ignore edge constraints in this example.



(c) All robot can now go directly to their shelf.

Figure 2.2: Vertex conflict solved

2.2 Merging as Constraint Satisfaction Problem

Our second approach interprets the merging problem as a Constraint Satisfaction Problem (CSP). CSPs are defined by a set of variables whose state must satisfy several constraints. Each of these variables has its own domain of values. Applied to our problem the set of variables is our new plan. The variables are the positions of each robot at each time step and the constraints are our sub-problems. Each state of this set would be a variation of the original plan, which, when satisfying all constraints, returns a possible merged plan.

2.2.1 Setup

In a first step we initialize all variables we need later. This includes two time helper statements, one counting from 1 and one from 0 up to the given horizon, all possible directions, up, down, left and right, the robots and all nodes (Listing 2.13)[3].

Listing 2.13: Variables

```
6  %############## SETUP #############
7  time(1..horizon).
8  nulltime(0..horizon).
9
10 % directions for robot dodges
11 dir(1,0). dir(-1,0). dir(0,1). dir(0,-1).
12
13 robot(R) :- init(object(robot,R),value(at,(X,Y))).
14 node(X,Y) :- init(object(node,N),value(at,(X,Y))).
```

The original plan is made of `occurs1/3` statements which represent robot actions, i.e., relative movements to the initial positions. Because it is hard to alter plans and check them for errors only using relative positions, we first transform them into absolute positions. To do so, the coordinates of the initial positions of each robot will become the coordinates for the `position/3` statement at time step `T=0` (Listing 2.14, line 17). After that, positions of the robot at time step `T+1` can be calculated using the movement and the position at time `T` (Listing 2.14, line 18).

[3] https://github.com/tzschmidt/PlanMerger/blob/main/encodings/merger/approach2/final_M_merger.lp

Finally in line 19 the original plan will be extended to the horizon. To prevent having two position statements at the same time the expression "`T1>=horizon-L, tlimit(R,L)`" was added. `tlimit/2` is the available time for each robot in which it can wait and perform dodges (see below for a more detailed explanation). This rule causes the plan to extend only after the original moves finished.

Listing 2.14: Transform relative to absolute positions

```
16 % get robot positions
17 position(R,(X,Y),0) :- init(object(robot,R),value(at,(X,Y))).
18 position(R,(X+DX,Y+DY),T+1) :- occurs1(object(robot,R),action(
      move,(DX,DY)),T+1), position(R,(X,Y),T).
19 position(R,(X,Y),T2) :- position(R,(X,Y),T1), T2=T1+1, T2<=
      horizon, T1>=horizon-L, tlimit(R,L).
```

2.2.2 Plan generation

After all positions of all robots at all times are known we can start generating variations of the plan. To prevent generating unnecessary wrong plans, we calculate `tlimit`, which is the time each robot can get delayed, so it can still fulfill its original plan within the time horizon. To do so we subtract the sum of all actions each robot takes from the horizon (Listing 2.15, line 23).

In this CSP Merger robots can perform two actions to solve conflicts, "wait" and "dodge". The `wait/3` statement determines, how long each robot gets delayed at time `T` to avoid possible collisions. Same goes for dodges with `dodge_t/3`, but the delay only occurs in steps of two; one time step for moving out of the way and one time step back.

All possible combinations of these `wait/3` and `dodge_t/3` statements are generated (Listing 2.15, lines 26-33) for each time starting with time step `T=0`.

Our main idea is to add the delay to the original positions, thus creating the new plans. This requires all delays to be in ascending order. The constraints in lines 36-37 (Listing 2.15) remove any models which do not satisfy this condition.

Listing 2.15: Generate time delays

```
21 %############## MOVE GENERATOR #############
22 % determine available time
23 tlimit(R,L) :- L = horizon - N,  N == #sum{1,T:occurs1(object(
      robot,R),_,T)}, robot(R).
24
25 % generate wait times
26 count(R,0..L) :- tlimit(R,L).
27 1{wait(R,T,N) : count(R,N)}1 :- robot(R), nulltime(T).
28
29 % generate time delay for dodge moves
30 dcount(R,0) :- robot(R).
31 dcount(R,N) :- tlimit(R,L), dcount(R,N-2), N<=L.
32 dodge_t(R,0,0) :- robot(R).
33 1{dodge_t(R,T,N) : dcount(R,N)}1 :- robot(R), time(T).
34
35 % remove invalid order
36 :- wait(R,T1,N1), wait(R,T2,N2), T1<T2, N1>N2.
37 :- dodge_t(R,T1,N1), dodge_t(R,T2,N2), T1<T2, N1>N2.
```

Example: Table 2.1 below shows an example of three possible combinations, also called models, for one robot with `horizon=6` and `tlimit=3`. According to the requirement above only model 3 is valid, because only its delays are in ascending order.

Table 2.1: 3 example models for one robot with `horizon=6`, `tlimit=3` after execution of line 33 in Listing 2.15

model	delays by wait/3	delays by dodget/3
1	0-1-3-2-1-1-3	0-2-2-0-0-2-0
2	1-2-3-3-2-1-2	0-0-0-2-0-0-0
3	0-1-1-2-3-3-3	0-0-0-2-2-2-2

Now that we got the timings for the dodges, we also need the directions of these dodges. To do so we again generate all possible combinations of the four directions defined in line 11 of Listing 2.13. But now only at an increase in the delay of dodge_t from one time step to another. As this indicates a dodge happened (Listing 2.16, line 40). It is important to note, that in the current version of our merger it does not make a difference by how much the delay increases. So an increase of 0-2 is the same as 0-4 and will only result in the generation of a single dodge. For easier

handling all time steps without a dodge get an empty ”dodge” with direction (0,0) (Listing 2.16, line 41).

Listing 2.16: Generate dodge directions

```
39 % rise in dodge_t means a dodge happened -> generate dodge
      move before dodge delay
40 1{dodge_m(R,T,(DX,DY)) : dir(DX,DY)}1 :- dodge_t(R,T,N1),
      dodge_t(R,T+1,N2), N1<N2, nulltime(T), robot(R).
41 dodge_m(R,T,(0,0)) :- dodge_t(R,T,N1), dodge_t(R,T+1,N2), N1>=
      N2, nulltime(T), robot(R).
```

After all variations have been generated we have to transform them back into new positions for each robot. At first we just delay the original positions by adding our wait- and dodge-delay (Listing 2.17, lines 44-45). For the dodges we need to add two new positions as seen in the bottom of Figure 2.3 .

The dodging itself is done by lines 48-49 in Listing 2.17. By delaying steps through our wait statements we get ’holes’ or ’jumps’ in our plan. These get filled in line 52.

Listing 2.17: Get new robot positions

```
43 % get new positions
44 newpos(R,(X,Y),0) :- robot(R), position(R,(X,Y),0).
45 newpos(R,(X,Y),T+W1+W2) :- robot(R), position(R,(X,Y),T),
      nulltime(T), wait(R,T,W1), dodget(R,T,W2).
46
47 % add new dodge positions
48 newpos(R,(X+DX,Y+DY),T+W1+W2+1) :- robot(R), position(R,(X,Y),
      T), nulltime(T), wait(R,T,W1), dodge_t(R,T,W2), dodge_m(R,T
      ,(DX,DY)), 1{DX!=0; DY!=0}1.
49 newpos(R,(X,Y),T+W1+W2+2) :- robot(R), position(R,(X,Y),T),
      nulltime(T), wait(R,T,W1), dodge_t(R,T,W2), dodge_m(R,T,(DX
      ,DY)), 1{DX!=0; DY!=0}1.
50
51 % fill posistions
52 newpos(R,(X,Y),T) :- newpos(R,(X,Y),T-1), not newpos(R,(X+1,Y)
      ,T), not newpos(R,(X-1,Y),T), not newpos(R,(X,Y+1),T), not
      newpos(R,(X,Y-1),T), robot(R), nulltime(T).
```

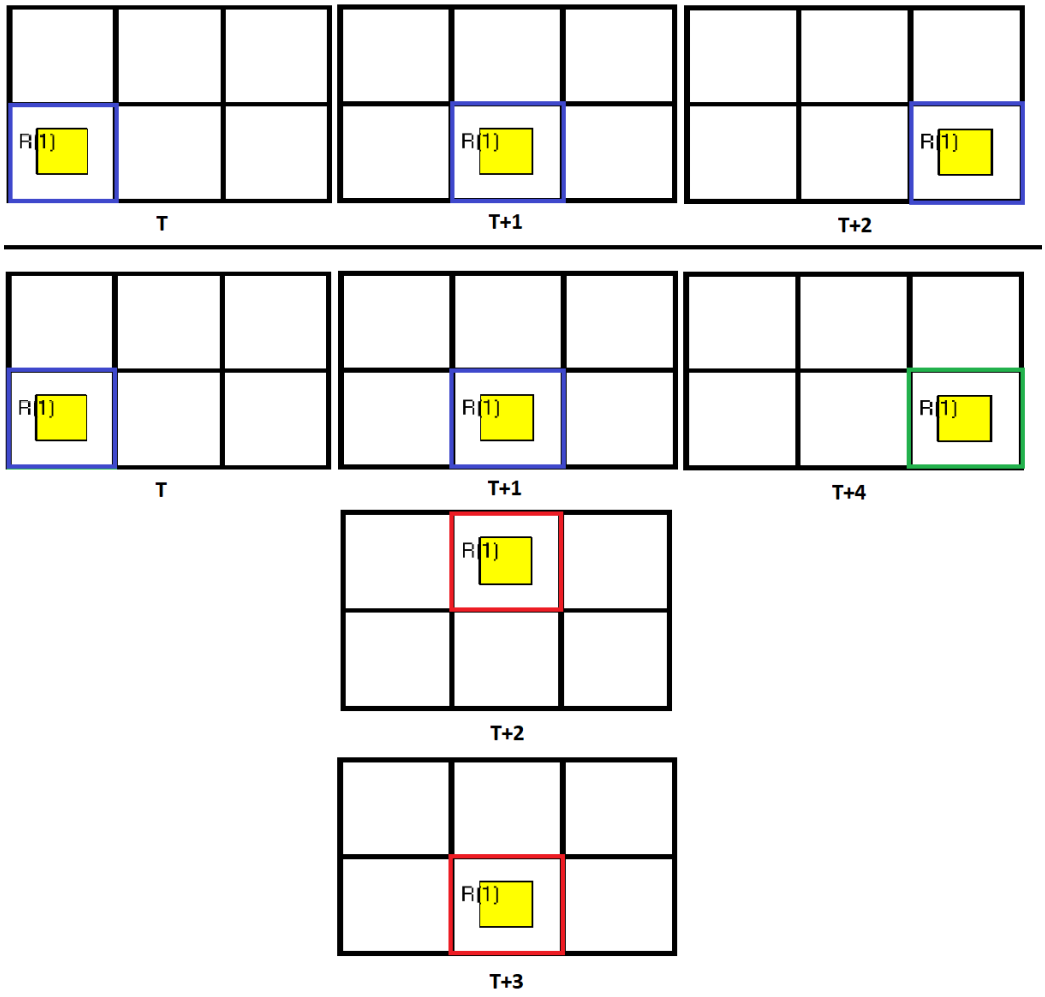


Figure 2.3: original positions, new positions due to delay, new positions by dodging

2.2.3 Constraining and output

Now that we have all our new plans generated, we just have to check, which of them fulfill all our constraints. These being, as written in the beginning of Chapter 2, Vertex constraint, Edge constraint, Out of bounds moves and the compliance with the time horizon. The Vertex constraint can simply be expressed as ”2 robots are not allowed on the same position at the same time” as seen in line 56 in Listing 2.18. The Edge constraint states ”2 robots are not allowed to swap positions” (Listing 2.18, line 58).

For our Out of bound moves constraint in line 60 we simply check if all positions exist as nodes.

In the M-domain the main goal is, that all robots end their plan under a shelf. The original plans meet this requirement. Therefore for our last constraint we check that all robots end their plan at the same position as in the original plan at the time horizon (Listing 2.18, lines 62-63).

Listing 2.18: Constraints

```
54 %############## CONSTRAINTS ################
55 % vertex constraint
56 :- newpos(R1,(X,Y),T), newpos(R2,(X,Y),T), nulltime(T), robot(
      R1), robot(R2), R1!=R2.
57 % edge constraint
58 :- newpos(R1,(X1,Y1),T), newpos(R2,(X2,Y2),T), nulltime(T),
      robot(R1), robot(R2), R1!=R2, newpos(R1,(X2,Y2),T-1),
      newpos(R2,(X1,Y1),T-1).
59 % out of bounds
60 :- newpos(R,(X,Y),T), robot(R), nulltime(T), not node(X,Y).
61 % horizon
62 :- newpos(R,(X1,Y1),horizon), robot(R), position(R,(X2,Y2),
      horizon), X1!=X2.
63 :- newpos(R,(X1,Y1),horizon), robot(R), position(R,(X2,Y2),
      horizon), Y1!=Y2.
```

After applying all our constraints only valid plans remain. All that is left to do is transform these ”position” plans with absolute positions back into ”action” plans with relative positions. This can be done by simply calculating the difference between the positions at time step T+1 and T (Listing 2.19 line 74).

Listing 2.19: Output transformation

```
71 %############ OUTPUT ######################
72 % transform new positions into output
73 occurs(object(robot,R),action(move,(DX,DY)),T) :- time(T),
      newpos(R,(X1,Y1),T), newpos(R,(X2,Y2),T-1), DX=X1-X2, DY=Y1
      -Y2, 1{X1!=X2; Y1!=Y2}1.
```

2.2.4 Evaluation

As written in the evaluation of our first merger (Section 2.1.4) we developed this CSP Merger approach to improve on the weaknesses of our first approach. Even

though our new merger can handle problems like benchmark 6, it still can not solve every possible problem. The reason for that is the limitation in our dodge moves (only one move in one direction), which can lead to problems like benchmark 20, where two consecutive dodges are needed, being still unsolvable by the CSP Merger. Solving this is not as simple as it might seem. Just increasing the possible dodge moves to 2 would lead to the same problem for a benchmark, where three dodges are needed and so on. We decided to stay with one possible dodge move mainly because of two reasons.

The first one being, that with increasing possible dodge moves, we would have to face an exponential increase in run time.

And secondly in real life warehouses there are more than often free spaces instead of labyrinth like wall structures. Cases, where multiple dodges are necessary, are therefore very rare. In all of our random generated large scale benchmarks it did not happened even once.

Ignoring the limitation described above our CSP Merger improved on our iterative merger, by being able to react to collisions before they happen. Furthermore the code of the CSP Merger is easier to understand and better maintainable than the Iterative Merger and allows for easier modification, which will be discussed in Section 2.3 "Additional Features".

The performance of the CSP Merger is mainly dependent on the given horizon, because it always uses all the available time. This can results in bad performance for poorly chosen horizons. To fix this issue we wrote a Python script which interprets the given horizon as a maximum and tries to find an earlier solution. This improves the performance on larger benchmarks with poorly chosen or big horizons by quite a lot. More on that can be found in Section 2.3.2.

While our Iterative Merger could solve a lot of simple problems, it becomes apparent that our CSP approach uses the concepts of ASP far more efficiently, giving it the ability to solve bigger benchmarks in less time. Therefore the CSP approach is a worthy upgrade of our first Iterative Merger, while keeping its ideas to solve conflicts and improves upon them.

So we conclude that, when it comes to using ASP for a plan merger, looking at the task via a Constraint Satisfaction Problem is superior, regarding performance and scalability, than trying to use an iterative approach.

2.3 Additional Features

2.3.1 Plan Locking

The structure of the CSP merger allows us to easily add additional features. One minor feature we decided to add, was the ability to lock specific robots. Locked robots are not allowed to change their plan at all. This can easily be implemented by introducing a new constraint. This constraint, here split into lines 68-69 in Listing 2.20[4], only allows empty waits and dodges as long as the `lock/1` statement is set for a specific robot. This feature could be improved by stopping locked robots from generating wait and dodges in the first place to reduce the size of the search tree.

Listing 2.20: Locking feature

```
65 %############ ADDITIONAL FEATURES #########
66 % possible improvement: move before generation
67 % locked robots can't change plan -> benchmark contains lock(
      object(robot,R)).
68 :- wait(R,horizon,N1), lock(object(robot,R)), robot(R), N1!=0.
69 :- dodge_t(R,horizon,N2), lock(object(robot,R)), robot(R), N2
      !=0.
```

2.3.2 Horizon Optimisation using Python

As written above our CSP mergers performance and solving time still mainly depend on the given horizon, because all the available time is used to generate waits and dodges. This can in the case of a poorly chosen horizon lead to poor plans with many unnecessary waits and dodges. To avoid this scenario and simultaneously find the fastest possible solution (for our merger) we created a python script [5] that iterates over all possible horizons until it finds a solution. When one is found, we know that this solution is the best possible solution for our merger.

But there is still the possibility that there is no solution and the script will get stuck in an endless loop trying to find the smallest possible horizon. To avoid this we set the original horizon of the benchmarks as an upper limit for the search. Now

[4] https://github.com/tzschmidt/PlanMerger/blob/main/encodings/merger/approach2/final_MA_merger.lp

[5] https://github.com/tzschmidt/PlanMerger/blob/main/share/Final_Merger_TS_JB_HW/main.py

it iterates from 1 to the maximal horizon and if we cant find a solution the problem is unsatisfiable in the given horizon range. The maximal horizon can here also be picked easily because if the benchmark is solvable a high maximal horizon has no impact on the computation time if the solution can be found within the range. In the other case where the problem is unsolvable we don't want to waste much time trying out unnecessary horizons. So we end up with a trade-off where we have to approximate or guess the maximal horizon not to low to still include a possible solution but also not to high to waste time calculating unsolvable benchmarks.

In practice clingo and our CSP merger encoding comes to help us with this trade-off. It shows that unsatisfiable solutions can be thrown out quite fast because of our chosen constraints and clingos[3] way of search. So we only have to choose a reasonably high maximal horizon and the python merger will be able to find an (for our merger) optimal solution.

2.3.3 A-domain

One major modification of the CSP merger is the ability to handle A-domain problems. In the A-domain robots not only have to reach one shelf, but have to pickup shelves and deliver them to a picking station. After fulfilling its order the shelf has to be put down on a valid node, so the robot can get the next shelf if needed.

There are two big issues in merging A-domain problems. The first one being the ability of robots in their original plans to block each others paths by putting shelves in their way. And the second one being the uncertainty where shelves are. For example if one robot moves a shelf, the second robot still expects the same shelf at its original position.

Our main idea to solve both of these issues at the same time is to modify the original plans in a way, that robots return the shelves back to their original position after delivering.

Because we want the CSP merger to handle both A- and M-domain problems, we introduce the `adomain/0` statement. This statements needs to be set, if there is an A-domain problem to be solved.

Analogous to our M-domain merger we first transform the move actions of all robots into position statements (Listing 2.21, lines 18-19). We add the prefix "pre" to show that they refer to the original plan. Line 21 in Listing 2.21 works similar

to line 20, which we already know from the M-domain, and extends the positions until the horizon for the A-domain.

We also have to get the positions of the other actions connected to the A-domain (pickup,putdown and deliver) (Listing 2.21, lines 23-25).

Listing 2.21: Get original positions

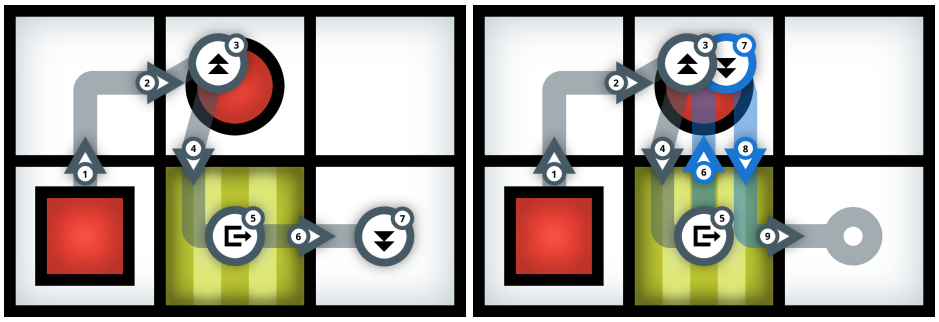
```
17 % get robot positions
18 prepos(R,(X,Y),0) :- init(object(robot,R),value(at,(X,Y))).
19 prepos(R,(X+DX,Y+DY),T+1) :- occurs1(object(robot,R),action(
       move,(DX,DY)),T+1), prepos(R,(X,Y),T).
20 prepos(R,(X,Y),T2) :- prepos(R,(X,Y),T1), T2=T1+1, T2<=horizon
       , T1>=horizon-L, tlimit(R,L), not adomain.
21 prepos(R,(X,Y),T2) :- prepos(R,(X,Y),T1), T2=T1+1, T2<=horizon
       , T1>=horizon-L-TA+TD/2, tlimit(R,L), tdifft(R,horizon,TA),
        tdiff(R,TD,_,N), not tdiff(R,_,_,N+1), adomain.
22
23 prepos(R,C,T+1) :- prepos(R,C,T), occurs1(object(robot,R),
       action(pickup,()),T+1).
24 prepos(R,C,T+1) :- prepos(R,C,T), occurs1(object(robot,R),
       action(putdown,()),T+1).
25 prepos(R,C,T+1) :- prepos(R,C,T), occurs1(object(robot,R),
       action(deliver,_),T+1).
```

For easier handling we first get all the original times for the pickup and deliver actions and match them together (Listing 2.22 lines 28-33). We again add the prefix "pre" to show that all times refer to the original plan. The putdown action is not important, because we will put down the shelves at new positions anyway.

Listing 2.22: Get action times

```
27 % get times for pickup and delivery and match them
28 prepickuphelp(R,0,0) :- robot(R).
29 prepickuphelp(R,T1,N) :- occurs1(object(robot,R),action(pickup
       ,()),T1), prepickuphelp(R,T2,N-1), T1>T2.
30 prepickup(R,T,N) :- prepickuphelp(R,T,N), not prepickuphelp(R,
       T,N+1).
31 predeliverhelp(R,0,0,0) :- robot(R).
32 predeliverhelp(R,T1,D,N+1) :- occurs1(object(robot,R),action(
       deliver,D),T1), predeliverhelp(R,T2,_,N), T1>T2.
33 predeliver(R,T,D,N) :- predeliverhelp(R,T,D,N), not
       predeliverhelp(R,T,_,N+1).
```

To adjust the original plan we first need to do some preparation work. To know where our robots have to go, to return the shelves, we "copy" all positions between matching pickup and deliver actions. (Listing 2.23, line 38). Obviously performing these extra moves consumes some time, by which following actions must be delayed. To calculate these delays we first calculate the time for each "back-and-forth" case (Listing 2.23, line 41). An example of such case is seen in Figure 2.4. After we got all these delays we can add those, which happened before time `T` to get `tdifft/3` at time `T`. These tdifft statements describe the total delay at time `T` (Listing 2.23, lines 43-44).



(a) Original plan (b) Modified plan

Figure 2.4: Example for plan modification

Listing 2.23: Preparation work for adjusting the original plan

```
35 % edit plan
36 % possible improvement: only edit if pickup!=putdown, first
      tests showed significantly worse performance
37 % copy path from pickup to delivery, remember delivery time
      for future step
38 reppos(R,C,T,T2) :- prepos(R,C,T), prepickup(R,T1,N),
      predeliver(R,T2,_,N), T1<=T, T<T2, N!=0.
39 % calculate extra time needed
40 % per copy
41 tdiff(R,2*(T2-T1),T2,N) :- prepickup(R,T1,N), predeliver(R,T2,
      _,N), N!=0.
42 % at time T
43 tdifft(R,0,0) :- robot(R).
44 tdifft(R,T,TA) :- robot(R), time(T), TA = #sum{T1,T2:tdiff(R,
      T1,T2,_), T2<=T}.
```

Now that we got all prerequisites we can start creating the adjusted plan by adding new positions; once from delivery to pickup and once back (Listing 2.24, lines 47-51). All other positions get delayed by the respective `tdifft/3` delay value (Listing 2.24, line 53). It is to note, that this transformation of the original plan is not perfect. For example there still will be some remnants of the original plan, that the robot has to perform, which are unnecessary in the adjusted plan. This adjusted plan will serve as the new original plan for further work of our merger.

Listing 2.24: Add additional positions

```
46 % add copied path
47 % delivery -> pickup
48 position(R,C,T+(T1-T)*2-1+TA) :- reppos(R,C,T,T1), tdifft(R,T,
       TA).
49 % pickup -> delivery
50 % possible improvement: last step unnecessary, remnant of
       orginal deliver
51 position(R,C,T+TD+TA) :- reppos(R,C,T,T1), tdiff(R,TD,T1,_),
       tdifft(R,T,TA).
52 % adjust old plan
53 position(R,C,T+TA) :- prepos(R,C,T), tdifft(R,T,TA).
```

All that is left to do before we can can hand over to our merger is to adjust all other actions. While all pickup and deliver actions just get delayed, we have to add new putdown actions at the same position as pickup. The time of the putdown from the pickup simply is twice time needed from pickup to delivery (Listing 2.25).

Listing 2.25: Adjust actions

```
55 % adjust actions
56 pickup(R,0,0,0) :- robot(R).
57 pickup(R,T+TA,S,N) :- tdifft(R,T,TA), prepickup(R,T,N),
       position(R,C,T+TA), shelf(S,C,0), N!=0.
58 deliver(R,0,0,0) :- robot(R).
59 deliver(R,T+TA,D,N) :- tdifft(R,T-1,TA), predeliver(R,T,D,N),
       N!=0.
60 % create new putdowns
61 putdown(R,0,0) :- robot(R).
62 putdown(R,T1+2*(T2-T1),N) :- pickup(R,T1,S,N), deliver(R,T2,_,
       N), N!=0.
```

These actions and the new original plan will serve our CSP merger as input. To adjust to the A-domain only minor modifications are necessary. Because we had to use some move time to alter the original plan, the available time, each robot has, changes compared to the M-domain. Instead of subtracting only the sum of actions, we also subtract the delay value of `tdifft` at the horizon from the horizon (Listing 2.26, line 69).

After each robots last putdown it is not requiered to do its back move. That's why we add half of the delay value of the last `tdiff` to the available time (Listing 2.26, line 68).

Listing 2.26: New `tlimit` for the A-domain

```
68 athelp(R,TD/2) :- tdiff(R,TD,_,N), not tdiff(R,_,_,N+1),
      adomain.
69 tlimit(R,L) :- L = horizon-N-TA+TH,  N = #sum{1,T:occurs1(
      object(robot,R),_,T)}, tdifft(R,horizon,TA), athelp(R,TH),
      robot(R), adomain.
```

After all waits and dodges were generated like for the M-domain, another modification has to be implemented to avoid dodges through shelves while carrying a shelf. To do so we first add the wait and dodge delays to the pickup and putdown actions (Listing 2.27, lines 101-102). Now it is possible to calculate when each robot is carrying a shelf (Listing 2.27, lines 103). The won information will be used in the new constraints below. For complexity reasons we refrain from calculating each shelves position at all times.

Listing 2.27: Determine when shelves are carried

```
100 % determine when shelves are carried
101 newpickup(R,T+W1+W2,S,N) :- time(T), pickup(R,T,S,N), wait(R,T
      ,W1), dodge_t(R,T,W2), N>0.
102 newputdown(R,T+W1+W2,N) :- time(T), putdown(R,T,N), wait(R,T,
      W1), dodge_t(R,T,W2), N>0.
103 carry(R,T1,T2,S) :- newpickup(R,T1,S,N), newputdown(R,T2,N),
      robot(R), N!=0.
104 % possible improvement: determine position of shelves at all
      times
105 %shelf(S,C,T) :- shelf(S,C,T-1), newpickup(R,T1,S,N+1),
      newputdown(R,T2,S,N), T2<=T, T<T1.
```

Naturally by moving our CSP merger from the M- to the A-domain we also have to revisit our constraints. The only constraint that changes, is the one regarding the horizon or our goal. Instead of having to stand under a shelf at the end of the horizon, it does not matter where each robot end its plan. It is only important, that each robot achieves its last putdown (Listing 2.28, line 119) and does not end its plan under a shelf to avoid blocking other robots (Listing 2.28, line 120).

Additionally two new constraints must be added regarding shelves. The first one being the inability to dodge with shelves through shelves. This is done by simply prohibiting robots to dodge into original shelves positions, as these are the only positions, where shelves can be (Listing, 2.28 line 125). This means robots can not dodge into these positions even if the respective shelf is carried at the moment. The cases in which this actually hinders the merger are rare. So we decided to keep the merger as simple and fast as possible.

The second constraint regarding shelves is the restriction, that carried shelves can not be picked up (Listing 2.28, line 129).

Listing 2.28: A-domain constraints

```
107 %############## CONSTRAINTS ################
108 % vertex constraint
109 :- newpos(R1,(X,Y),T), newpos(R2,(X,Y),T), nulltime(T), robot(
       R1), robot(R2), R1!=R2.
110 % edge constraint
111 :- newpos(R1,(X1,Y1),T), newpos(R2,(X2,Y2),T), nulltime(T),
       robot(R1), robot(R2), R1!=R2, newpos(R1,(X2,Y2),T-1),
       newpos(R2,(X1,Y1),T-1).
112 % out of bounds
113 :- newpos(R,(X,Y),T), robot(R), nulltime(T), not node(X,Y).
114 % horizon, different for M and A domain
115 % M domain stop under original shelf
116 :- newpos(R,(X1,Y1),horizon), robot(R), position(R,(X2,Y2),
       horizon), X1!=X2, not adomain.
117 :- newpos(R,(X1,Y1),horizon), robot(R), position(R,(X2,Y2),
       horizon), Y1!=Y2, not adomain.
118 % A domain stop (anywhere) after last putdown, but not under
       shelf
119 :- T1=T+W1+W2, time(T), putdown(R,T,N), wait(R,T,W1), dodge_t(
       R,T,W2), not putdown(R,_,N+1), T1>horizon, adomain.
120 :- newpos(R,C,horizon), robot(R), shelf(_,C,0), adomain.
121
122 % A domain specific shelf constraints
123 % cant move with shelf through shelf
124 % cant dodge into original shelf position, regardless of
       whether there actually is a shelf or not (rare case, where
       this actually hinders the merger)
125 :- carry(R,T1,T2,S1), T1<=T, T<=T2, shelf(S2,C,0), S1!=S2,
       robot(R), newpos(R,C,T), nulltime(T).
126 % possible improvement: dodge is possible if shelf is not
       there
127 %:- newpos(R,(X,Y),T), robot(R), nulltime(T), shelf(S,(X,Y),0)
       , carry(R,T1,T2), T1<=T, T<=T2.
128 % robots can only pickup shelves that are not carried
129 :- newpickup(R1,T,S,N), carry(R2,T1,T2,S), T1<=T, T<=T2, time(
       T), R1!=R2.
```

The last modification we added to our merger is the output transformation of the new A-domain specific actions, which can be seen in Listing 2.29. To do so we simply take our already calculated new times for pickup and putdown and add the new deliver time.

Listing 2.29: Output transformation

```
142 % adjust a-domain specific actions
143 occurs(object(robot,R),action(pickup,()),T) :- newpickup(R,T,S
       ,N).
144 occurs(object(robot,R),action(putdown,()),T) :- newputdown(R,T
       ,N).
145 occurs(object(robot,R),action(deliver,D),T+W1+W2) :- time(T),
       deliver(R,T,D,N), wait(R,T,W1), dodge_t(R,T,W2), N>0.
```

This new merger can merge both M- and A-domain problems. Because of how this merger uses our CSP merger for the M-domain as a framework it inherits both the benefits and the disadvantages as described in Section 2.2.4. While it is no perfect A-domain merger, it can handle most benchmarks and do so in acceptable time.

Chapter 3

Results and Benchmarking

While working on our plan merger, we always verified our newest progress on Benchmarks.

3.1 Benchmark Engine

3.1.1 Benchmark Evaluation

To assist us in this time consuming process we wrote a python script. It first of all uses the clingo python API to solve the different benchmarks and more importantly directly evaluates the mergers result and thereby checks for:

1. vertex conflicts
2. edge conflicts
3. invalid movements (into walls or outside the maps borders)
4. that the solution lies within the specified horizon range

This script can therefore run a specified merger on multiple predefined benchmarks and display the results of its evaluation as a plain markdown table or a more stylised HTML evaluation sheet.

Though it's important to keep in mind that the Benchmark Engine script[1] is still a work in progress and a long way from being a finished script. At this point it is still pretty much tailor-made for our own merger and benchmark style, so when

[1] https://github.com/tzschmidt/PlanMerger/tree/main/scripts/benchmark_engine

using it with a different merger which requires different presets or so on errors can easily arise.

3.1.2 Random Benchmarks

The script is also capable to produce randomly generated Benchmarks that can be defined by :

1. the number of robots and shelves
2. the spacial dimensions of the map (in X and Y direction)
3. a time horizon
4. and a random seed for reproducibility

When a random benchmark is invoked this way random non covering robot and shelf positions are generated and matching simple linear plans from each robot to its destination shelf. Which robots gets which shelf as destination is simply decided by their ids. Robot 1 has to reach Shelf 1 and so on (Examples on our Groups GitHub Repository)[4].

3.2 Results

Now that we have developed our solution we want to evaluate it and compare it to other solutions to this problem. We will first compare our CSP Merger to our deprecated version the Iterative Merger and after that will compare the CSP Merger with the results of the other groups working on this project.

3.2.1 Internal Comparison (Iterative- vs. CSP Merger)

For the internal comparison of our CSP Merger against our Iterative Merger, which we developed before, we could observe a substantial improvement. While our iterative merger works on simple benchmarks with 2 to 4 robots, if the number of robots is increased it most of the time cant find a solution. In Table 3.1 below we collected solving results on five sample benchmarks. The simple Benchmarks 01-03[5] are being solved by both mergers but on benchmark 06 the Iterative Merger

begins to struggle. This is because our Iterative Merger solves its problems based on the detection of a conflict. In the case of Benchmark 06[5] which is a tunnel where two robots collide in the middle, the iterative solver just recognizes the problem when the robots are already deep in the tunnel and now can't dodge anywhere which makes this benchmark unsolvable. Our CSP Merger on the other hand can through its CSP structure already generate many wait statements for one robot in the beginning until the other robot has passed through the tunnel and now the second robot can pass through the now empty tunnel unhindered.

But it is also important to acknowledge that Benchmarks like Benchmark 20[5], which require multiple successive dodges, to be solvable are still impossible to be solved by both Mergers. Our reasoning behind this is, that a multi dodge supporting script would first of all blow up the mergers search tree drastically, which would increase the solving time. And second of all we think that an approach like this couldn't be called a proper "merger" anymore because it would in the end just generate completely new plans. That would contradict our problem setting of "merging already computed plans because an informed generation would take too much time". Another reason why we decided not to cover these type conflicts is that they are extremely unlikely to happen in a normal warehouse layout. If we for example take the Manhattan style layout which is generated by the Asprilo Generator[1] there are no long tunnels, big chunks of blocked nodes or extremely dense clusterings of robots in which this type of conflict only arises.

Table 3.1: Comparison of correct solutions of CSP Merger and Iterative Merger on 5 sample benchmarks

merger	Bench.01	Bench.02	Bench.03	Bench.06	Bench.20
Iterative - solvable	True	True	True	False	False
Iterative - correct	True	True	True	False	False
CSP - solvable	True	True	True	True	False
CSP - correct	True	True	True	True	False

We can also look at the solving times of our two different approaches (Table 3.2), which makes it obvious that our CSP Merger is not only more capable and scalable than our Iterative Merger but also solves the same problems way faster.

Table 3.2: Comparison of Solving Times of CSP Merger and Iterative Merger on the same 5 sample benchmarks

merger	Bench.01	Bench.02	Bench.03	Bench.06	Bench.20
Iterative	0.0468s	0.0468s	3.6939s	0.6623s	0.8285s
CSP	0.0192s	0.0181s	0.0829s	0.1198s	0.0578s

3.2.2 External Comparison (CSP Merger vs. other Groups)

We now want to compare our CSP Mergers results to the results of the other groups working on this project. We have in total 5 Groups which are the ones of:

1. A. Salewsky[6]
2. M. Funke and M. Wiedenhöft[7]
3. M. Wawerek and N. Kämmer[8]
4. T. Ramadan.[9]
5. And our Group[4]

Together we developed a selection of benchmarks which we would all test our mergers on to compare their functionalities. Our Group created a big evaluation table (Figure 3.1), that contains all the useful information we could derive from testing on these selected benchmarks[10] with all the groups mergers.

Benchmark Name	Instance 1	Instance 5	Insatnce 6	Instance 7	bench_test_2	bench_test_3	bench_test_16_mod_1	benchmark-5	benchmark-6	benchmark-42	benchmark-51	B_03	B_05	B_R1	B_R2	benchmark_1	benchmark_2	benchmark_3	benchmark_4
Gruppe	Adrian				Max + Marcus			Niklas + Marius				TJH				Tarek			
Num. Robots	2	4	2	8	2	2	4	4	8	5	6	4	3	50	30	3	2	3	2
Map-Dimension X	5	4	7	8	4	4	5	5	4	10	15	5	4	15	40	3	7	6	9
Map-Dimension Y	3	3	2	8	4	5	5	8	7	10	15	5	5	15	40	3	5	6	2
Max Horizon	5	3	14	10	7	4	9	11	15	10	21	10	10	40	100	5	19	10	16
c_nr (for Adrian)	1	3	9	3	1	1	5	14	6	1	1	4	1	1	1	8	10	2	4
nn (for Tarek)	4	10	9	10	2	3	4	7	5	2	3	4	3	5	5	4	16	6	8
Merger of Group :																			
Adrian																			
Correct	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	WAHR
Satisfiable	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	KILLED	KILLED	FALSCH	FALSCH	WAHR	WAHR
Timeout	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	KILLED	KILLED	FALSCH	WAHR	FALSCH	FALSCH
Time (in s)	0,0114	0,0607	#########	########	0,0122	0,0067	3,2160	13,4690	358,7314	3,1784	218,2776	5,3833	0,0354	-	-	0,3224	1800,0000+	0,1982	0,5161
Max + Marcus																			
Correct	WAHR	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSE (M)	FALSE (H+)	FALSCH	FALSE (H+)	FALSCH	FALSE (M)	FALSE (M)	FALSCH	FALSCH	FALSCH	FALSCH	FALSE (M)	FALSE (M)
Satisfiable	WAHR	FALSCH	WAHR	KILLED	WAHR	WAHR	WAHR	WAHR	KILLED	WAHR	KILLED	WAHR	WAHR	KILLED	KILLED	FALSCH	FALSCH	WAHR	WAHR
Timeout	FALSCH	FALSCH	FALSCH	KILLED	FALSCH	FALSCH	FALSCH	FALSCH	KILLED	FALSCH	KILLED	FALSCH	FALSCH	KILLED	KILLED	FALSCH	FALSCH	FALSCH	FALSCH
Time (in s)	0,2238	0,1713	1,4714	-	0,3973	0,0519	3,6219	25,9768	-	540,0421	-	8,3662	1,4516	-	-	0,1227	3,7384	13,0752	4,0606
Niklas + Marius																			
Correct	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	WAHR	WAHR	WAHR
Satisfiable	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	KILLED	KILLED	FALSCH	WAHR	WAHR	WAHR
Timeout	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	KILLED	KILLED	FALSCH	FALSCH	FALSCH	FALSCH
Time (in s)	0,0055	0,0070	0,0328	0,1137	0,0142	0,0050	0,0244	0,1260	0,5552	0,1082	0,0938	0,0316	0,0125	-	-	0,0049	0,3070	0,0336	0,0528
TJH																			
Correct	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	FALSCH	WAHR	WAHR
Satisfiable	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	FALSCH	WAHR	WAHR
Timeout	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH
Time (in s)	0,0269	0,0171	0,0270	0,2555	0,0399	0,0190	0,1108	0,0888	1,7390	0,1399	2,3586	0,0848	0,0183	87,6203	618,3541	0,0274	1,5511	0,0862	0,3454
Tarek																			
Correct	WAHR	FALSCH	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	WAHR	WAHR	FALSCH	FALSCH	WAHR	WAHR	FALSCH	WAHR
Satisfiable	WAHR	FALSCH	WAHR	FALSCH	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	WAHR	FALSCH	KILLED	WAHR	WAHR	WAHR	WAHR
Timeout	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	KILLED	FALSCH	FALSCH	FALSCH	FALSCH
Time (in s)	0,0186	0,0090	0,0310	1,0490	0,0094	0,0099	0,0173	0,0759	0,1245	0,2054	1,9946	0,0192	0,0099	91.175s	-	0,0143	0,2817	0,0786	0,0281

Figure 3.1: Benchmarking Results[11]

Now when we compare our merger to the ones of the other groups some distinct differences become evident. First of all our merger is able to solve all of the benchmarks except these following three:

1. `bench_test_2` : by M. Wiedenhöft and M. Funke
2. `benchmark_1` and `benchmark_2`: by T. Ramadan

If we look at these benchmarks in detail we can observe that they are again variations of the in Section 3.2.1 "Internal Comparison" described problem of multiple successive dodges. This makes it obvious why they do not work on our merger and we can be confident that our merger is working on all the scenarios it was intended for.

Another interesting thing we can observe with our submitted benchmarks `B_R1` and `B_R2` is, that our merger is the only one being able to solve them without needing more time than 30 minutes (the upper computation time limit we decided on) or allocating more than 16GB of space for its clingo grounding and as a result of that being killed by our testing system. These two cases of timeout and killing can be in a realistic scenario interpreted as not solvable because it would just be unreasonable to dedicate so much time and computation power to a task like this.

But why do all the other mergers fail? First of all this has to do with the Benchmarks which represent pretty big warehouses (15×15 Nodes and 40×40 Nodes), contain many robots (50 and 30) and therefore also have a high maximal horizon (40 and 100). Second of all this has to to with the other groups approaches to the problem setting. Both the groups of A. Salewsky and the one of M. M. Wawerek and M. Funke created a more iterative way to solve the problem, which as we observed on our own Iterative Merger, is just not a really scalable solution for bigger benchmarks. The other two groups focused more on random move generation to solve scenarios like the multi dodge conflicts. This indeed makes them able to solve these scenarios and also generally solves small benchmarks even faster then our CSP Merger. But also contributes to their mergers downfall on bigger benchmarks. With a large map and a high maximal horizon the size of the search tree just increases exponentially and thereby makes it nearly impossible to effectively compute solutions which such an approach. Because our merger just generates waits and simple dodges it is better equipped for these kind of situations and also takes less time and space to solve.

Chapter 4

Conclusion

4.1 Conclusion

When we compare our two approaches, it becomes very apparent that the second design is vastly superior to the first one. It achieves better results in performance, versatility and scalability.

Despite this, we should not discard our first approach completely, because it is after all able to solve quite a number of benchmarks. But it is still a fact that our second approach is simply better adapted to ASP and, as such, can utilise it much more effectively.

Our first approach shows us that trying to force our iterative approach on ASP makes the problem unnecessarily complicated. This can be best shown by by the fact that we needed to create a entirely new dimension to this problem via the conflict depths. Something that is not needed if we interpret this problem as Constraint Satisfaction Problem.

With that we can say that our assessment, that the first approach has to many limitations, was spot on and our decision to work on a newer approach early on, was the right one.

4.2 Outlook

While our second approach has a lot of strong suits, there are still a few things that could be improved upon.

Firstly, while our merger is fast when it comes to generating paths, the paths themselves aren't always very efficient. The reason for this is, that our merger often

adds unnecessary wait or dodge statements as long as they aren't contradicting any constraints. This means that we can have paths, where the robot waits or dodges in seemingly random situations, even when it's not needed. It might be possible to alleviate this problem with some clever constraints, that remove unnecessary wait or dodge maneuvers. But one also has to be carefully not to reduce the time efficiency of the merger, with these constraints.

The second problem is that it is impossible for our merger to let a robot dodge again, while it is already in a dodge procedure. This means a robot only has the simple wait and dodge actions to work with. There are however specific situations where these simply aren't enough and our merger can't solve such problems. While we can simply allow our merger to use nested dodges, this would sadly cause a strong decrease in time efficiency. We would first need to find a way to elevate these disadvantages, if we want our merger to be even more versatile.

Bibliography

[1] P. Obermeier and T. Otto. Asprilo. https://asprilo.github.io/.

[2] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide togringo, clasp, clingo, and iclingo. https://www.cs.utexas.edu/users/vl/teaching/lbai/clingo_guide.pdf, 2010.

[3] University of Potsdam. Potassco. https://potassco.org/.

[4] T. Schmidt, H. Weichelt, and J. Bruns. Planmeger repository. https://github.com/tzschmidt/PlanMerger.

[5] T. Schmidt, H. Weichelt, and J. Bruns. Benchmark folder in repository. https://github.com/tzschmidt/PlanMerger/tree/main/instances/benchmarks.

[6] A. Salewsky. Repository of a. salewsky. https://github.com/salewsky/Plan-Merging.

[7] M. Funke and M. Wiedenhöft. Repository of m. funke and m. wiedenhöft. https://github.com/Zard0c/ProjektMAPF.

[8] M. M. Wawerek and N. Kämmer. Repository of m. m. wawerek and n. kämmer. https://github.com/NikKaem/mapf-project.

[9] T. Ramadan. Repository of t. ramadan. https://github.com/tramadan-up/asprilo-project.

[10] Benchmark collection used for comparison. https://github.com/tzschmidt/PlanMerger/tree/main/benchmark_collection.

[11] T. Schmidt, H. Weichelt, and J. Bruns. Benchmarking results. https://unipotsdamde-my.sharepoint.com/:x:/g/personal/hweichelt_uni-potsdam_de/EY8Sa-6OsTtLqimDGTLBqyEBfQxNp3pl-HRtOqKi2h_Tnw?e=sxDvkr.

Multi Robot Plan Merging Project

Project report

by

Tom Schmidt
796970

Hannes Weichelt | **Julian Bruns**
798247 | **796290**

INSTITUTE OF COMPUTER SCIENCE

UNIVERSITY OF POTSDAM

WISE 2020 - 2021