KRR Project

Philomena Moek, Denis Andreev

Potsdam University

Abstract. This project is based on the "Constraint-Based Search", a search-algorithm used to solve the problem of "Multiple Agent Path Finding". This problem consists of multiple agents. For each agent paths have to be found, so they reach an agent-specific goal within a defined field of movement. The difficulty of this problem is to find short paths without collisions so that the agents can still reach their goals as fast as possible. This problem simulates robots moving autonomously in a warehouse from shelf to shelf.
Two different complete versions of the search-algorithm are implemented and tested. These different versions each use different heuristics, the cost functions, which will also be discussed. The main difference between the two approaches is that one of them is optimal and the other only semi-optimal, but instead significantly faster and the benefits of both will be presented statistically. The goal is not to decide which one is better, but to compare and emphasise the differences and their impact.

Keywords: Multiple Agent Path Finding (MAPF) · Constraint-Based Search (CBS) · greedy Constraint-Based Search (gCBS) · Answer Set Programming (ASP)

1 Introduction

Multi-agent path finding - or MAPF for short - is a theoretical problem inspired by the problem of automatised robots moving in a warehouse. Within the problem a number of agents need to be moved through a space by finding paths for them.

The difficult part of the problem is to avoid collisions between the agents. The Brute Force approach would be to calculate every possible combination of paths of every robot, which has an exponential time-complexity with just one of the following increasing: the size of the field, the number of agents or the number of conflicts.

Our project is based on the asprilo framework and uses Answer Set Programming (ASP) and Python. In asprilo, warehouses are represented by a chess-like grid on which the robots are placed.

Starting points of each warehouse-robot and their destinations, a specified shelf (which represents products being stored), are utilized as input. The shelves are also placed on the grid. Additionally, problem files (or "instances") contain the shortest paths for each robot to their goal - not considering other robots. A robot's designated shelf is specified in the beginning and cannot change.

To solve the problem each robot has to reach the field their shelf is placed on and stay there. All of this has to happen within a specified time limit (called the "horizon"). We do not refer to actual time. Instead, a unit of time in this context is the time a robot needs to move from one coordinate to one that is vertically or horizontally adjacent.

In the following, we introduce two solutions to the problem: An implementation of the optimal Constraint Based Search (CBS) and a faster, but suboptimal greedy CBS (gCBS). Both of them are complete (they will find a solution if given enough time) if the horizon is set high enough.

The output of both programs are conflict-free paths for every robot from its start node to its goal node, if a solution exists.

We chose to implement these two solutions because this way you have the opportunity to find an optimal solution but can also choose to go for the quicker algorithm which is not optimal.

2 Background: What is Constraint Based Search (CBS)?

The CBS algorithm is a popular approach to MAPF problems. The algorithm takes an initial plan with the shortest paths for each robot, not considering conflicts.

CBS is proven to be complete and optimal. That means it will always halt and find an optimal solution, if there is any, that minimizes its cost function. The algorithm consists of two parts: a highlevel search and a lowlevel search.

CBS solves a conflict by only changing the path of a single robot each step the algorithm takes. Because of this, it scales really well with increasing the size of the field and the number of robots, as long as the number of conflicts is relatively low.

2.1 Highlevel

The highlevel-search uses a binary tree structure to solve conflicts. Each node in the tree saves:

1. A "problem" consisting of a grid, robots, their goals and their shortest paths to said goals.
2. A set of constraints (A constraint prohibits a certain robot to be at certain coordinates at a specific time)
3. And the cost of the paths. (Different cost functions are possible here. See chapter 3 for more detail)

The root node is initiated with the plan given as input, an empty set of constraints and a cost of zero. Each edge in the tree represents a solution to a single conflict - in form of an added constraint.

Of course the initial paths could be calculated as well, so they wouldn't have to be an input, but it is easier to measure and compare the implementations of the algorithm this way. The output of the CBS algorithm is a solution that is optimal relative to the cost function. The details are described below:

```
1: ROOT = the node containing the initial plan, no constraints and cost zero
2: use lowlevel on ROOT to calculate its conflicts
3: TREE = binary tree of plans
4: QUEUE = initiate priority queue
5: enqueue ROOT
6: set ROOT as the root of TREE
7: while there are still nodes in QUEUE do
8:     CURRENT = dequeue the node with the lowest cost
9:     if CURRENT has no conflicts then
10:        return CURRENT as the solution
11:        end
12:    end if
13:    FC(R1,R2,COO1,COO2,T) = one of the first conflicts of CURRENT
   with:
14:         R1 and R2 being the robots that collide
15:         COO1 and COO2 being the coordinates of the collision
16:         T being the time of the collision
17:    create two constraints C1 and C2 as follows:
18:         C1 constrains R1 from being at COO1 at time T
19:         C2 constrains R2 from being at COO2 at time T
20:    create two children for CURRENT, CH1 and CH2:
21:         CH1 inherits the constraints of its parent and additionally C1
22:         CH2 inherits the constraints of its parent and additionally C2
23:    use lowlevel on CH1, calculating CH1's plan and conflicts and do the
   same for CH2
24:    calculate the costs for CH1 and CH2
25:    enqueue CH1 and CH2
26: end while
27: since QUEUE is empty, return that the initial problem is unsolvable
```

2.2 Lowlevel

The lowlevel-search takes a plan, the constraints of the parent node and the additional, new constraint. It returns a new plan that satisfies the new constraints. Additionally it returns the new resulting conflicts (with a specified first conflict). It works as follows:

```
1: ROB = robot restricted by the new constraint
2: calculate new shortest path P for ROB that satisfies all given constraints
3: if there is no possible path P then
4:     return the information, that there is no possible solution with the given
   constraints
```

```
5: else
6:     to get the new plan, substitute the old path of ROB with P in the old
       plan
7:     calculate all conflicts for the new plan
8:     if there is at least one conflict then
9:         choose one of the conflicts with the lowest time as first-conflict
10:        return the new plan, conflicts and the first-conflict
11:    else
12:        return the new plan, which will be a solution
13:    end if
14: end if
15: end
```

The presented lowlevel would be semidecidable. That means if there is no possible solution for a problem, it might not return that, but give us the same conflicts over and over again at different time-steps, thus never terminating. This is why a “horizon” is used, which gives us a time limit after which there are no movements allowed and paths automatically fail.

Using the horizon, completeness of the algorithm is assured. Finding a solution if one exists can be guaranteed by setting the horizon accordingly: If the horizon is at least $\frac{\#fields!}{(\#fields-\#robots)!}$, then every possible combination of robots on a field-grid can be achieved. Therefore a solution can be found, if there is any.

With a bigger horizon, the positioning of the robots on the grid would start to be repetitious. Which of course would not lead to a solution, because they already did not solve the instance the first time. But calculating that many plans is not realistically possible for non-trivial instances, so the horizon is usually chosen proportionally to the size of the grid, not exponential. This is usually sufficient as will be demonstrated experimentally.

The lowlevel part of CBS can also be used to calculate the cost of the plan.

3 Cost-Functions

Although CBS is optimal, you still have to think about the cost function you want to use and by which the optimality is implicitly defined. Based on what a warehouse scenario might have as requirements an optimal solution could have all the robots finish their movements as fast as possible (without collisions). Additionally you might strive for robots to have as few movements as possible.

3.1 Makespan

Thinking about the solution as a singular isolated instance makes it easy to pick this approach. If you want your agents, as a total, to be finished as fast as possible, the only thing you have to do is minimize the time of the last movement made. This would be the makespan, which treats the time of the last movement as the cost. This approach has a major downside: if a robot A takes longer than some other robot B, than robot B is unrestricted to do some extra movement. For example:

Let there be two robots Robot 1 and Robot 2. Robot 1 moves “(0,0), (1,0), (2,0), (3,0)” to get to their destination. Robot 2 starts at “(0,1)” and is only one step away form their destination at “(1,1)”. Robot 1 has the longest direct path and therefore dictates the minimal possible cost of 3 using makespan. The shortest path for Robot 2 would be simply “(0,1), (1,1)”, keeping the cost of 3. On the other hand the path “(0,1), (0,2), (1,2), (1,1)” would also be a valid path for Robot 2 and also keeps the cost of 3. Using the makespan those two paths would be equally valid as optimal solutions, because the cost is minimal.

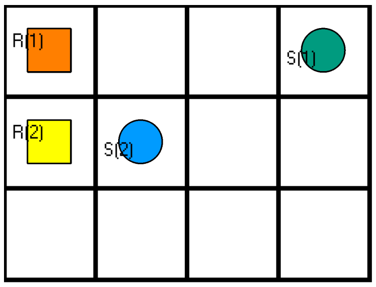


Fig. 1: Potential for unnecessary movement with makespan

This is not a problem if our goal is simply to find a solution and makespan suffices as a criterion for optimality, but it does not reflect the reality in a warehouse very well. If a robot in a warehouse is finished with a task, it would usually be assigned a new one as fast as possible. So the sooner a robot is finished, the better, let alone the cost from unnecessary movement. Also with robots moving randomly in their free time there might be more unpredictable conflicts to solve than if a robots stands still and is easily avoided.

3.2 Sum of Movement-Times (SOMT)

To solve the problem, which arises from makespan, an approach would be SOMT. SOMT adds up the time of every movement that is made. This has two general benefits:

1. SOMT punishes unnecessary movement. If an additional movement is made, the time at which it is made is also added to the cost. So in general not making a movement is better than to make it, unless the rest of the path is altered.
2. SOMT prefers sooner movement, since a later time means a higher cost. Every move will happen as soon as possible, while considering the movements of other robots.

But SOMT also has downsides. It prefers sooner movement not regarding the actual impact this has. If a robot has a path of 8 moves, but it has to wait for one time-unit at any given moment (for example because the destination is occupied at the time 8), then it is much cheaper in terms of cost to wait at the time 8 and move to the destination at time 9 (I), than it is to wait at time 1 and then move to the destination without further interruption (II). This is true, although the path and the time of completion is the same. The reason for this is the following: The increase in cost from waiting is decided by the number of moves made after the wait, in this case it would be 1 in I and 8 in II.

$$cost(I) = \sum_{i=1}^{7} i + 9 = 37 \tag{1}$$

$$cost(II) = \sum_{i=2}^{9} i = 44 \tag{2}$$

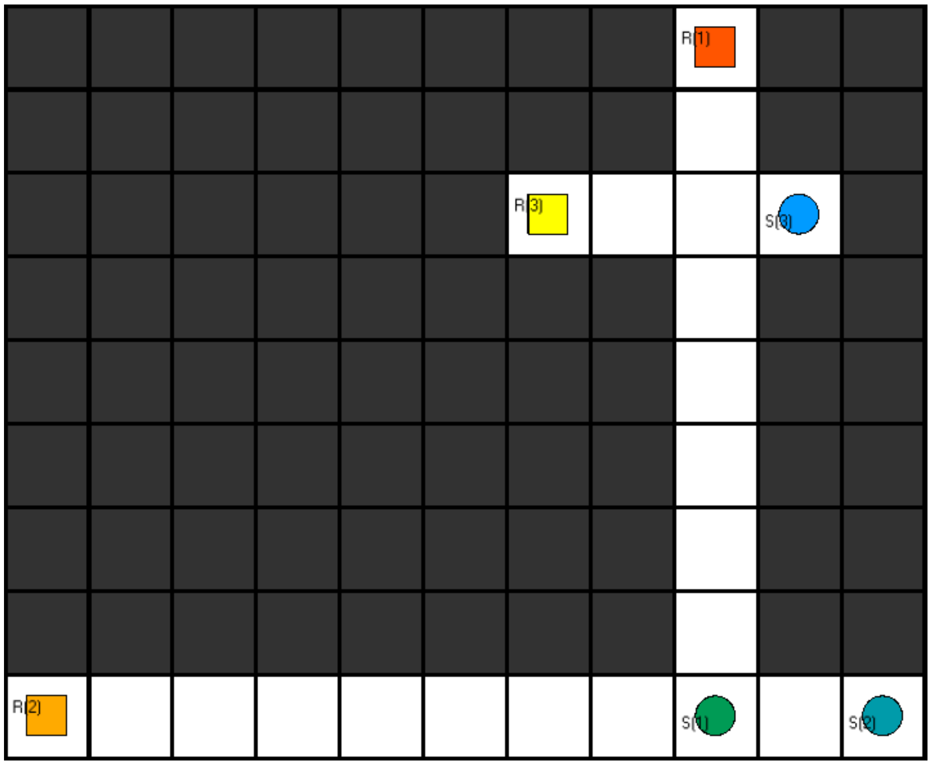


Fig. 2: Potential for unnecessary waiting with SOMT

This is a great difference in cost between two solutions, which intuitively should be equivalent. So it is possible that another robot would have to wait avoidably, because the increase in cost for our second robot is lower that it would be for our first robot (Which would mean that our robot 2 has less movements left than the first would have). Also usually more nodes have to be visited in bigger instances, because more complicated sub-solutions might get a lower cost than simpler ones, including the actual solution.

3.3 Sum of Lastmoves (SOL)

SOL combines parts from makespan and SOMT. While makespan and SOMT use the cost as the minimization-criterion in the lowlevel as well, this works a little differently in SOL. The cost used in the highlevel of CBS with SOL is the sum of the time of the last movement of every single robot. By minimizing just this in the lowlevel, it could lead to the same problem as in makespan and partially SOMT as well, namely that some robots make unnecessary/avoidable movements. For example if a robot would have to wait for 2 units of time before moving on, it could instead just as well move back and forth somewhere and its last movement would be at the same time.

This is why there is another minimization-criterion in the lowlevel: the number of movements. This directly ensures the absence of unnecessary moves. Together with the guarantee, that the paths are finished as early as possible, it makes SOL the best optimal solution for warehouse-instances from the ones introduced. The number of movements is not included in the cost, because it is not relevant for the best solution on a highlevel-scale. As long as there are no unnecessary movements, only the time at which the robots reach their goal is important.

3.4 Approach for a Greedy-Algorithm: Number of Conflicts (NOC)

While SOL is the best solution for the lowlevel-minimization-criterion we found, it is debatable if it is the best solution for the highlevel-cost. If optimality isn't as important as runtime, then a suboptimal solution might be better. For that NOC can be used. The theory behind this solution is based on a simple interpretation of the principle of CBS: How do you get a solution? You eliminate all the conflicts. Derived from this, the cost is the number of conflicts. A solution is found, if there are no more conflicts, so the cost is 0.

4 Experimental Evaluation

To benchmark our implementations of CBS and greedy CBS we developed two scripts that automatically generate different test instances.

Each generated instance has a square shaped field-grid with only robots and shelves in it. Since a warehouse-like layout is the most probable in real-world applications, we also limited our benchmarking instances to that. A robot cannot be generated on their respective shelf. Additionally our test datasets are structured as follows:

1. Different sizes and robot-densities
   – This data set contains 200 instances.
   – It is comprised of 20 different categories with 10 instances per category.
   – The categories are the cross product of the sizes {5x5, 6x6, 7x7, 8x8} and the densities {20%, 25%, 30%, 35%, 40%}. The naming-convention we used specifies the size and density of every instance (e.g. folder/size5x5/density25/ex10.lp).
2. Incrementally increasing the number of robots
   – Additionally there are six other test-datasets of a different format. Each of these instances has the size 7x7.
   – Each instance in a dataset is created iteratively by adding a robot and a shelf to the previous instance. The first instance of each subset starts with two robots and two shelves.
   – Three of the six datasets have a maximum number of robots of 19 and the other three of 35. 19 robots on a 7x7 field equals a 40% robot-density and 35 robots are roughly equivalent to a 70% robot-density.
   – The smaller datasets (going up to 19 robots) are for testing the CBS implementation. Since greedy CBS is more performant it has to be tested on a bigger dataset.

Since our resources are limited and we have many examples, there is a limit to what we measure. If a program takes more than 5 minutes (300 seconds) to solve an instance, the process is terminated with an empty output.

We used these multiple benchmarking approaches to analyse a broad spectrum of influences on the runtime and, in the case of the greedy algorithm, optimality. This spectrum can be categorised in two main aspects:

1. How do different factors affect the runtime of the respective algorithm? Namely the factors are the size of the field and the density of robots.
2. The difference between the two different cost functions, SOL (sum-of-lastmove) and NOC (number-of-conflicts).

There are of course many other aspects, but these two main aspects can and will be categorically differentiated. The following results are strictly derived from experimental testing on a random dataset of examples.

4.1 Impact of Grid-Size and Density

Non-Greedy CBS The non-greedy CBS program isn't good on huge problems, especially those with a high density. As a result, a lot of instances timed out during benchmarking which results in many NaN values. Without those values the runtime evaluation (as well as other stats) will be optimistic, because we ignore anything, that took longer than 300 seconds (and thus is very difficult to solve for non-greedy CBS). So instead of dropping the NaN-values, we include them and claim a runtime of 300 seconds for them. This is still quite optimistic, but less so then if we just ignore them. As a compromise we marked the areas, in which there are more that 3 NaN values (in other words, where more than a third of the values are NaN) with a red line.

The plots show the mean-runtime of every density and size. This mean is calculated from all ten instances we measured for every density/size. It is apparent, that the runtime is exponential with respect to the instance-size as well as density (Figures 3 and 4). But because of the many missing values this is much easier to see in the plot of the number of NaN values per size/density-category, which is also exponential (Figures 5 and 6)).

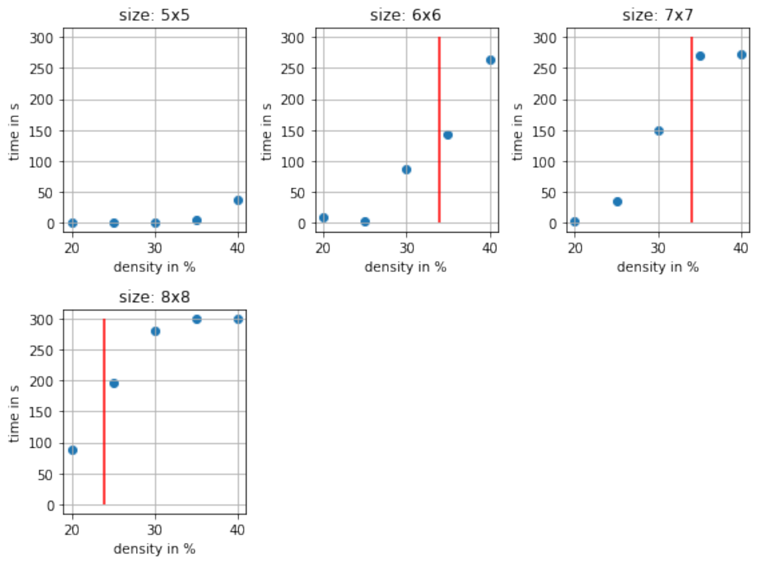


Fig. 3: The different sizes for each density in CBS

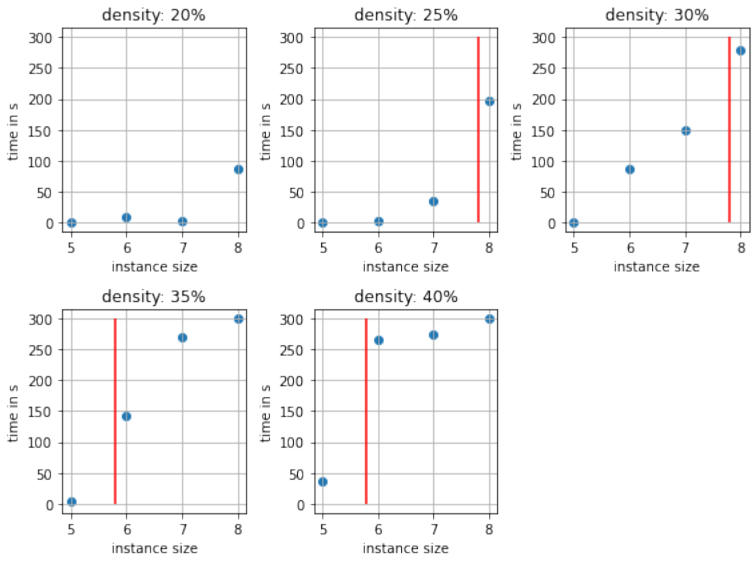


Fig. 4: The different densities for each size in CBS

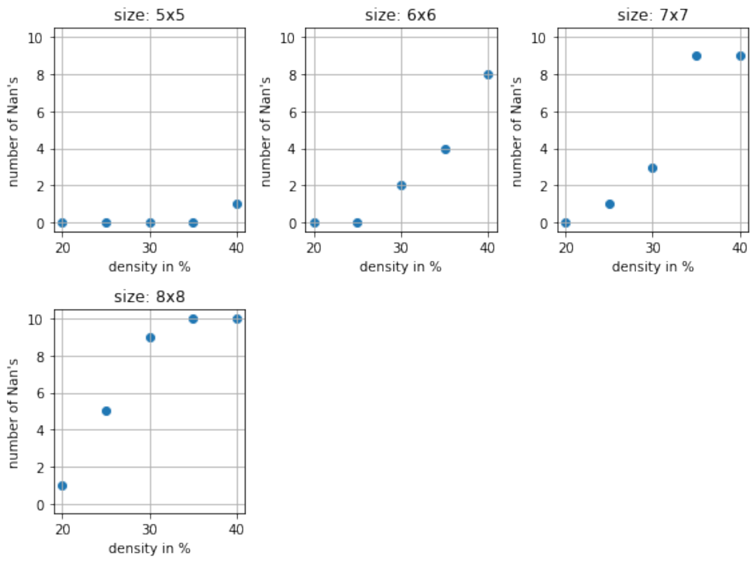


Fig. 5: The different sizes for each density against Nan

Greedy CBS Since there are less timeouts for greedy-CBS compared to non-greedy CBS, this is easier to interpret. The increase is obviously exponential for size and density respectively. Noteworthy is, that for the “simpler” instances -

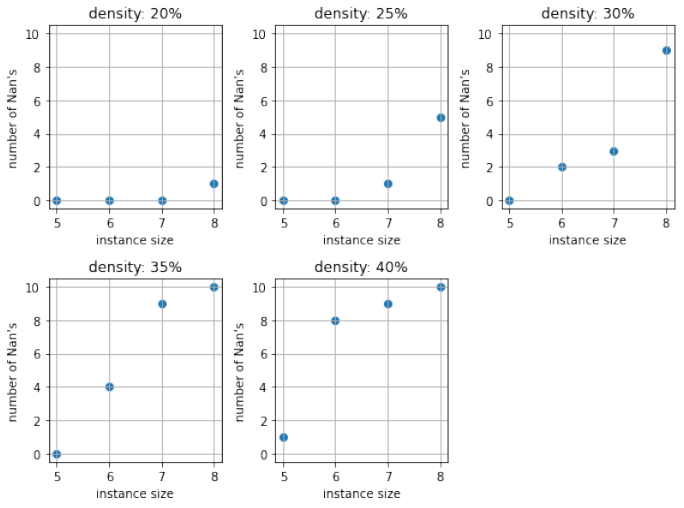


Fig. 6: The different densities for each size against Nan

those with lower robot-density and size - the values of SOL are comparable to NOC, while for “more complicated” instances this difference seems to increase greatly.

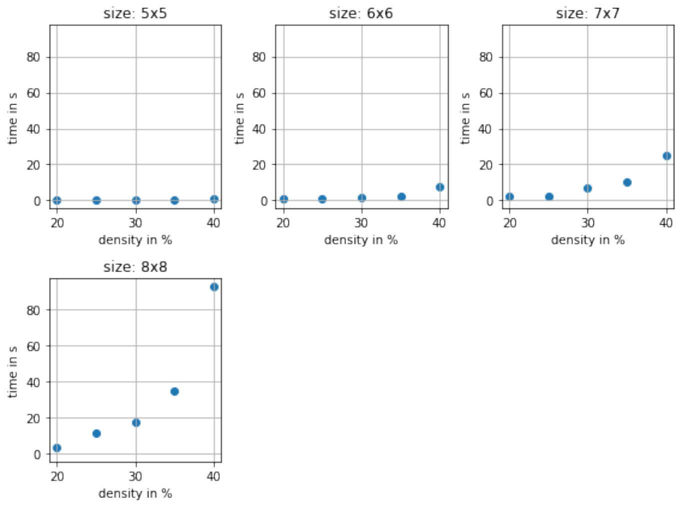


Fig. 7: The different sizes for each density in gCBS

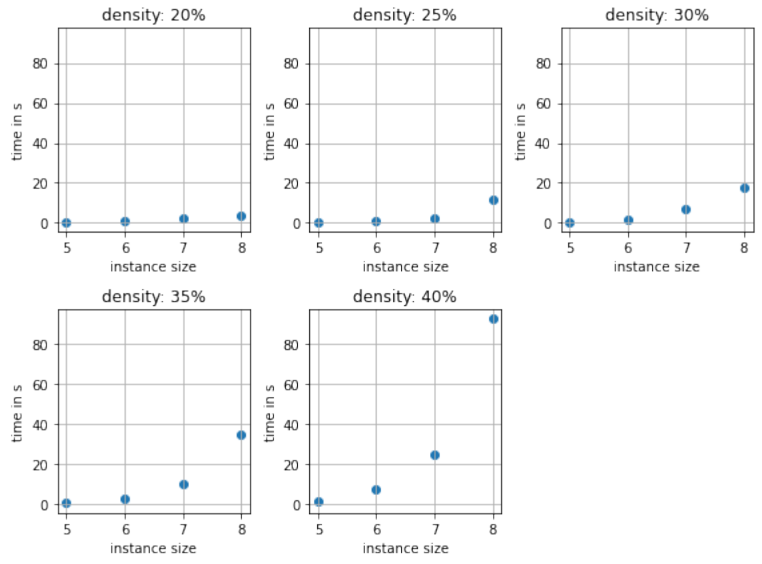


Fig. 8: The different densities for each size in gCBS

4.2 Comparison between Greedy and Non-Greedy CBS

Optimality of the CBS Algorithms

Sum of Moves The CBS algorithm can be optimal in different ways: The CBS implementation in this paper minimizes SOL and therefore produces a solution with the minimal sum of lastmoves.

Figure 9 and figure 10 show the difference in the sum of *moves* between the non-greedy CBS and greedy CBS implementation. The data for figure 9 was gathered on a five by five field with a robot-density of 40% (10 robots). We measured the amount of moves for ten random instances. Figure 10 shows the same data but measured on a seven by seven field with a 25% robot-density (12 robots). These two figures were chosen as examples, the other figures can be found on our GitHub Repository[1].

For the instances 8 from figure 9 and 5 from figure 10 the data for the non-greedy CBS is missing. This is due to a timeout while testing.

It becomes apparent in the plots that greedy CBS is slightly less optimal. On average the greedy implementation takes 2 moves longer in figure 9 and 4.4 moves in figure 10. The maximum difference between the greedy CBS move-sum and the non-greedy CBS move-sum is twelve moves for example 7 in figure 10. The numbers speak for themselves: Even though greedy CBS is not as good it is not greatly inferior to the non-greedy version.

[1] https://github.com/Philomena892/AndreevMoek

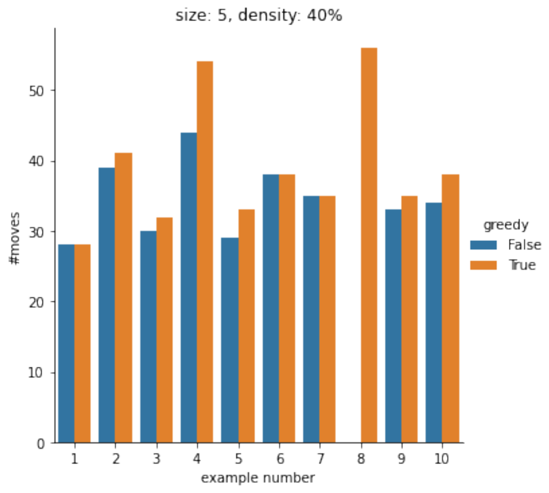


Fig. 9: Move-sum on 5x5 instance with 40% robot-density

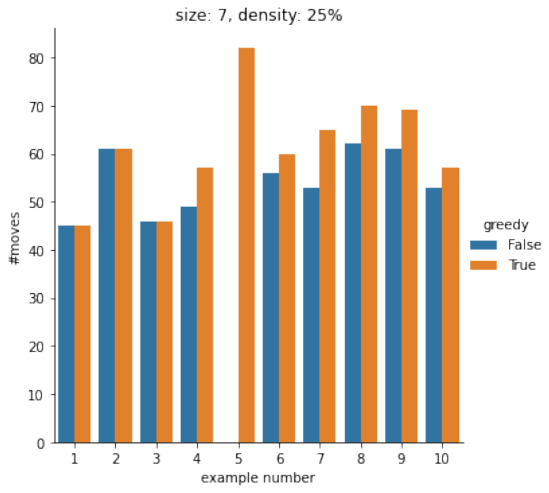


Fig. 10: Move-sum on 7x7 instance with 25% robot-density

Makespan The second optimality criterion for CBS is the makespan. The data in figure 11 and figure 12 was gathered with the same methods as the data for the sum of moves, only this time we plotted the makespan of the solution.

Both graphs illustrate that the non-greedy CBS (11) as well as the greedy CBS (12) can find the better solution concerning the makespan. Reason for this is the cost function we chose for the non-greedy CBS: SOL.

SOL calculates the same cost for the following two examples but the makespan is different:

1. Two robots reach their goal node after five steps and five time units each. SOL calculates a cost of $5 + 5 = 10$, because each robot makes their last move at time step five. Makespan results in a cost of five, because the last move in total takes place at time step five.
2. One of the two robots reaches their goal after two steps, but the other needs eight steps. If you apply the SOL cost function to this example you get a cost of $2 + 8 = 10$. The makespan however will result in a cost of eight because the second robot takes its last step at time point eight.

Even though neither implementation is optimal with respect to the makespan, they are not dissatisfactory either. Since the sum of lastmoves is minimized, unnecessary waiting times are eliminated. To cross a square-shaped instance from one corner to the one the farthest from it, a robot will need at least $(size - 1) \times 2$ steps. From the two graphs (as well as the additional ones) it becomes apparent that the makespan is generally lower than that. Figure 11 for example has a size of six. Because of that a robot would need ten steps to walk directly from one corner to the one opposing it. As seen in the graph, most of the makespans of the instances do not even surpass eight.

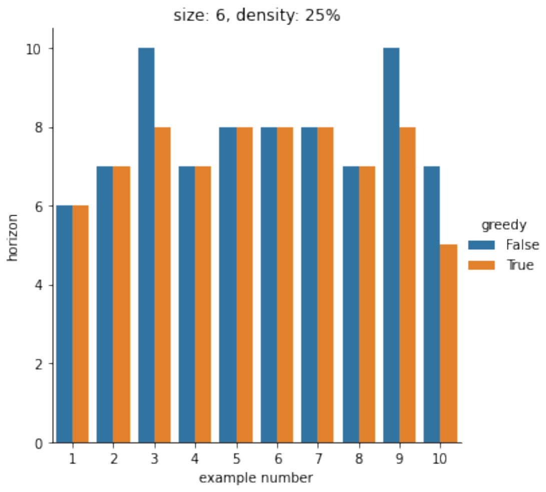


Fig. 11: Makespan on 6x6 instance with 25% robot-density

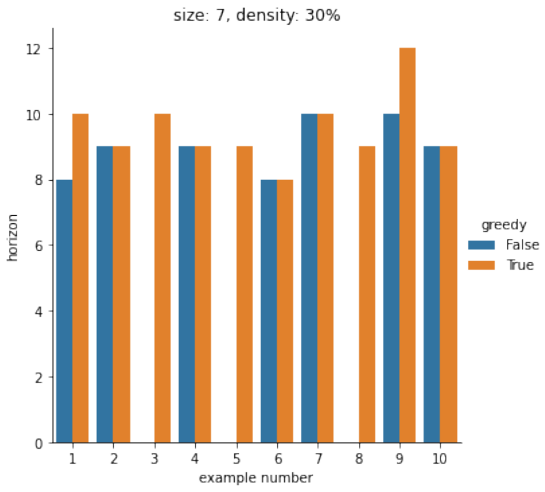


Fig. 12: Makespan on 7x7 instance with 30% robot-density

Impact of Different Features on the Runtime To analyze the runtime of both algorithms, we tried to ascertain the impact each of the values we measured has on it. For easier understanding of the data both the linear (black) and the exponential curve (orange) that would fit the data best are plotted as well. The data for each plot was gathered from ten random instances, each of them having the same size and robot-density.

Number of Nodes Figure 13 shows an example of how the runtime of the non-greedy CBS and greedy CBS is correlated with the amount of nodes that were traversed in the search tree. The solving time appears to be linearly related to the amount of nodes traversed for both CBS algorithms. This makes sense, since everything CBS does is evaluate node after node until a solution is found. A longer runtime is thus owed to a bigger amount of nodes that need to be evaluated.

Pathlength The results presented in figure 14 explain the speed difference between the two algorithms: The runtime of non-greedy CBS rises exponentially with the pathlength whereas for greedy CBS the increase is linear (On the left side the orange curve fits the data better and the black on the right). Pathlength refers here to the depth of the solution-node in the search tree. The reason that non-greedy CBS is slower than its alternative is that the cost function is less effective (because of the optimality). SOL favours solutions with few movements so it will take longer to find a solution that moves the robots enough to dodge every conflict. NOC however directly minimizes the number of conflicts and therefore finds a solution at the same depth significantly faster.

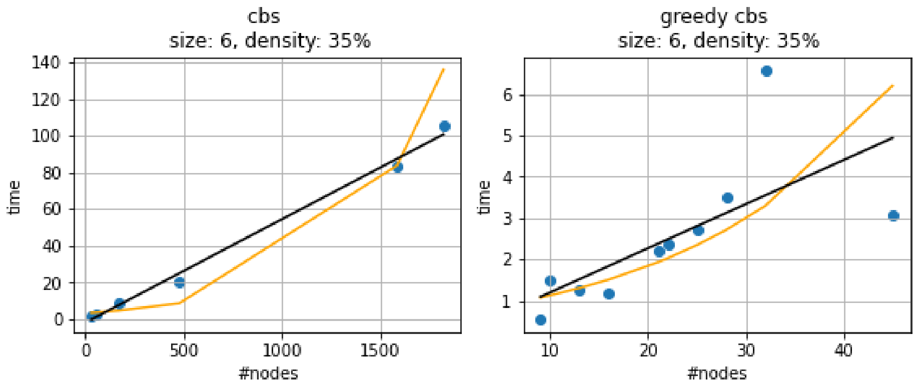


Fig. 13: Solving speed depending on the amount of nodes that were searched in the search tree

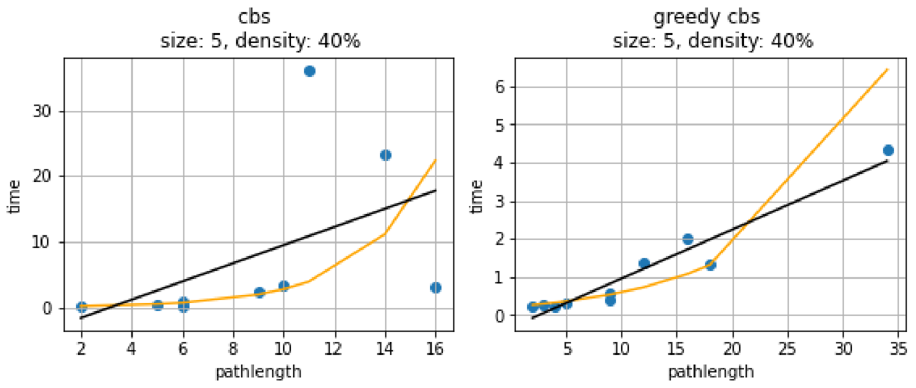


Fig. 14: Solving speed depending on the depth of the solution in the search tree

Horizon Figure 15 displays the solving speed depending on the makespan of the solution. Even though according to the black line, there seems to be a linear relation, that relationship is - if existent - negligible. The slope of the curve is very small and by directly looking at the scatter plot there also appears to be no clear relation visible.

Sum of Moves Similar to the pathlength, the sum of moves also has an exponential relationship with the runtime of non-greedy CBS and a linear one with greedy CBS (figure 16). As has been mentioned before this stems from the fact that the amount of moves in the non-greedy solution is strictly limited by the SOL cost function. Because of that more nodes are explored before the solution will be at the front of the priority queue.

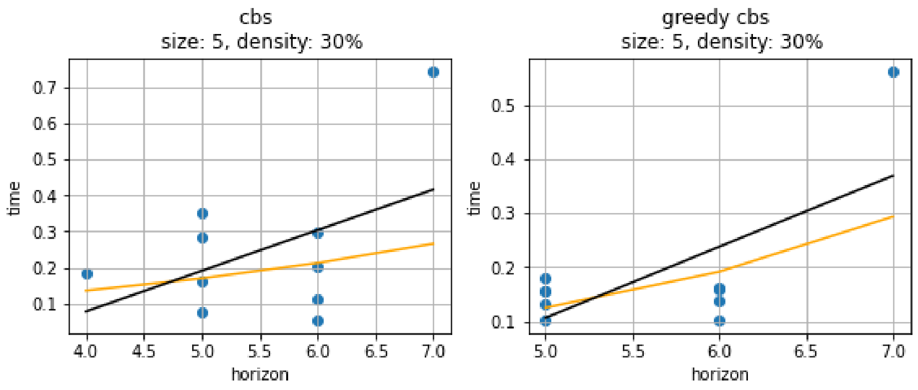


Fig. 15: Solving speed depending on the makespan of the solution

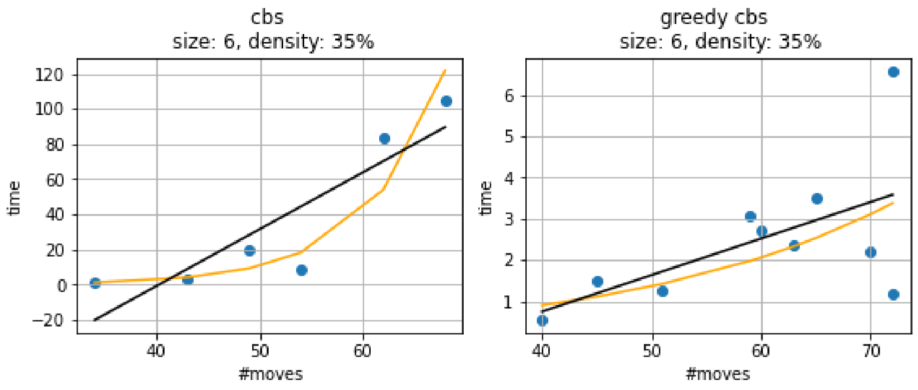


Fig. 16: Solving speed depending on the sum of moves of the solution

Initial Conflicts The initial amount of conflicts in the instance is roughly linearly correlated with the runtime, but not strongly (figure 17).

4.3 Impact of Incremental Increase in Number of Robots

In subsection 4.2 we technically already analyse the impact of density, which effectively is the increase of number of robots. The difference to this subsection is, that we focus on increasing the number of robots specifically, which leads to smaller and more consistent steps. More consistent in this context means, that we do not have to approximate percentages using whole numbers of robots.

CBS and gCBS both show strong indication, that the increase in time is exponential compared to the linear increase in the number of robots. But gCBS can calculate problems notably faster than CBS. Although a significant difference is only visible starting at a number of robots of about 11-14.

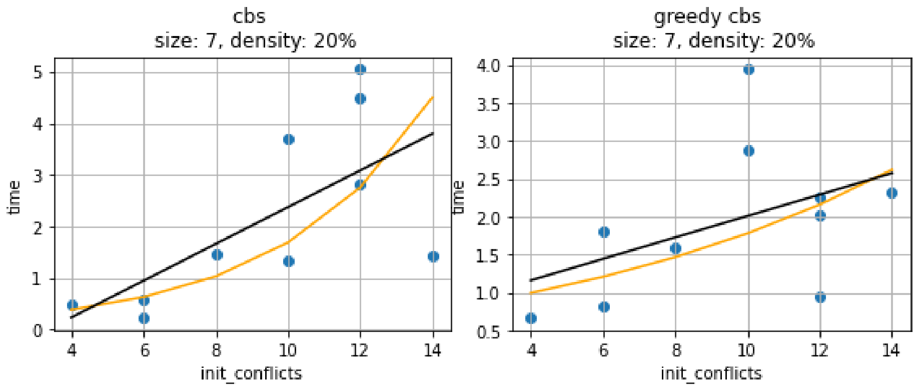


Fig. 17: Solving speed depending on initial amount of conflicts

A “significant difference” is one that is very unlikely to be caused by external factors like a random delay because of the hardware. In this case we deem a difference of about ten seconds as significant.

Given our limit in runtime of 5 minutes, gCBS is able to calculate about double the amount of robots CBS is able to.

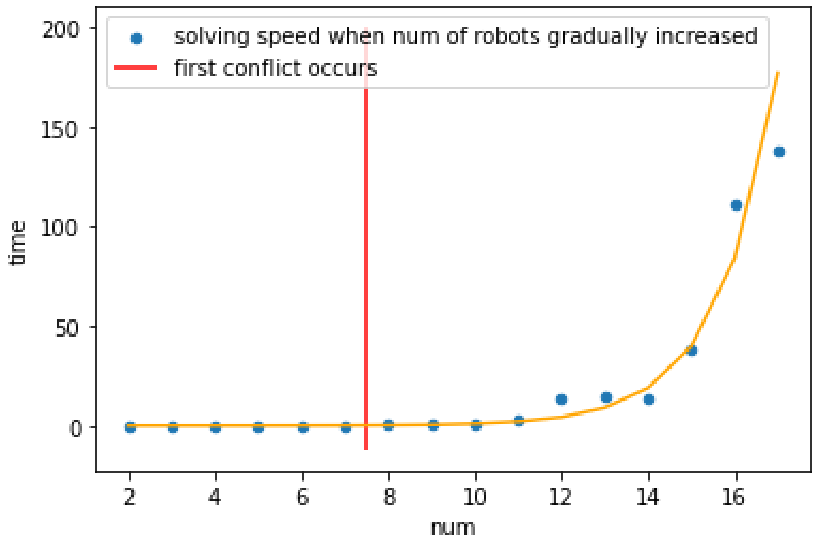


Fig. 18: Time depending on the number of robots in CBS

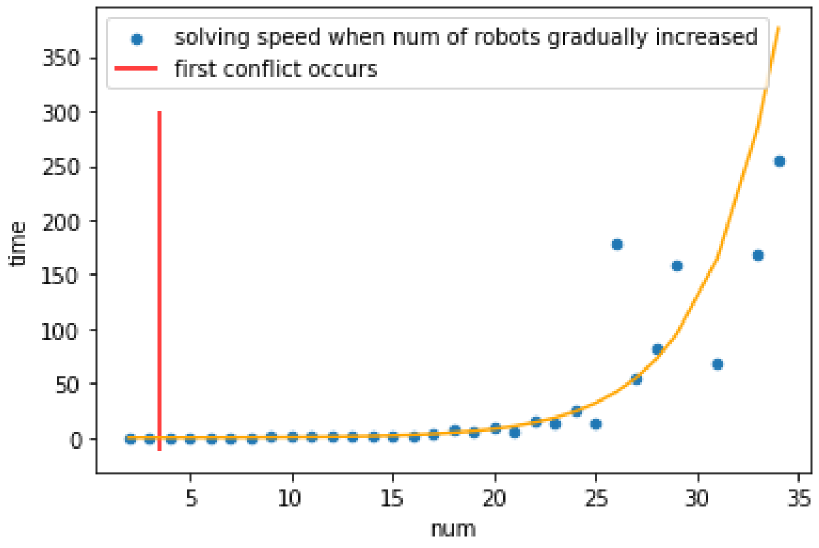


Fig. 19: Time depending on the number of robots in gCBS

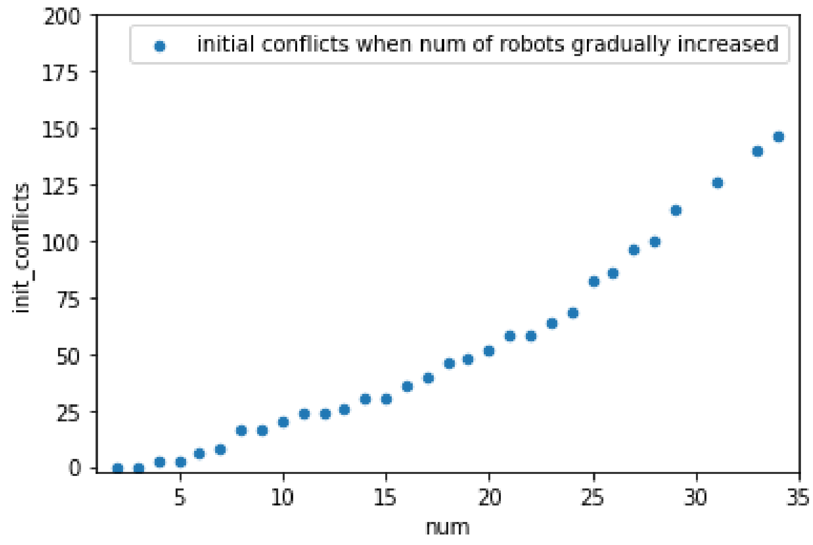


Fig. 20: Init_conflicts depending on the number of robots in gCBS

4.4 Comparison to Other Groups

Other groups This project had multiple groups (seven in total) working on solvers for the MAPF problem. In order to compare the results, we agreed that each group would create two instances.

Additionally a benchmark-program would be useful for comparing results, since doing that manually is not possible in the amount of time we were given. There would be a total of 14 instances for every group without a benchmarking program. Looking at that many solutions by hand is not feasible in the amount of time we were given and that is without considering that many groups provided multiple solvers. Because of that we exclude groups without a benchmarking program from our analysis, while nonetheless including their instances.

From the two instances the groups chose, the first instance should be one they performed well on, while the other should be difficult for them to solve. We ran every benchmark-program on every instance and the results can be seen in figure 21.

This process was accompanied with several complications. In the end we had four more or less working benchmark-programs and 12 instances from all groups. We had four days to completely test and interpret the results, however many of the benchmark-programs and instances where available only later, so effectively we had less time to benchmark.

We used the same limits for these tests as we did in our other tests: We stopped the execution of each program after 5 minutes.

SOC (Sum of Cost) is a new value that other groups used to determine optimality. It is a cost-function different to SOL, so it is not directly comparable to our implementation.

Our Group (AndreevMoek) Our benchmark-program, the same we used before, did run on all the instances. Any missing values in figure 21 did not terminate in 5 minutes.

Groups without Provided Benchmarks but with Instances The two groups of Andrés Córdova and Aleksandra Khatova (KhatovaCordova), as well as Jan Behrens (Behrens), did not provide benchmarks with documentation on their GitHub page. However they provided instances, which were of fitting format and which we included. They used the same instance-format that as our own group used.

Alexander Benjamin Glätzer and Mert Akil (GlätzerAkil) The instances of this group where not compatible with our program, because their movement started at time(0), while our time steps started at one (at time(0) were the initial positions in our interpretation).

Although we fixed this, other problems followed (possibly from fixing the first problem) and we cannot be sure that we fixed all of them. However the benchmark-programs accepted the fixed version.

The benchmark-program of this group unfortunately only worked on their own instances and gave an output that we could not interpret in relation to the other groups outputs. However we managed to isolate some values for time:

– total_processing of mini_maze.lp: 968750000

– total_real_world of mini_maze.lp: 1001423700
– total_processing of warehouse.lp: 1468750000
– total_real_world of warehouse.lp: 1481458900

total_processing and total_real_world probably referring to the total time attributed to the process and the total time including interruptions of the process. Since there was no documentation provided, we could not translate these numbers to seconds. However, we noticed during the execution, that this group took roughly 1.5 times longer to solve warehouse.lp than to solve mini_maze.lp. This can be compared to the other groups.

Nico Sauerbrei and Paul Raatschen (RaatschenSauerbrei) The instances of this group were compatible with all benchmark-programs. At the same time their benchmark-program could run on all instances. However their benchmark-program always ran 12 different solvers which they developed.

Usually when we tried to run the program, it would just abort somewhere in the process. Additionally this would be far to much data from just one group and we had a strict time-limitation, so we used only the first 3 outputs. This ended up to be more than half of the gathered data anyway. Their last ten solvers were just implementations of variants of the CBS-algorithms, so we deemed one CBS-version to be enough as a representation.

The first algorithm, using iterative solving (IS), claimed almost all of the instances to be unsolvable. All but the ones it has a time for in figure 21, "AndreevMoek/ex8.lp", "Behrens/mix.lp" and "Pan/x6_y11_r16_fo1_cmw1_cx4_cy4_rom2_instance_002.lp". The latter 3 instances took more than 5 minutes.

However, on the two instances "Behrens/mix.lp" and "Pan/x6_y11_r16_fo1_cmw1_cx4_cy4_rom2_instance_002.lp" the failure lead to the program freezing and failing entirely. When we tried to skip this algorithm, the benchmark-program did not work at all. So we were not able to run the other algorithms of this group on these examples either.

This also lead to the question of how their other solvers are dependent on this one and in what way this could affect the actual time needed to solve an instance with just a single solver.

The second solver, using prioritised planning (PP), also claimed two of the instances to be unsolvable. Namely the two instances of the group Pan.

Both of these solvers were, however, fast in saying that the instances are unsolvable by them. This behavior is practical if one wants to first use an easy algorithm and, if that fails, start a more complete one, like CBS.

The third algorithm, which used CBS as our group did, took more than 5 minutes to calculate "RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp" and "Pan/x6_y11_r32_fo2_cmw1_cx4_cy4_rom2_instance_004.lp".

Steven Pan (Pan) The instances of this group at first did not have the format we needed for our program to run. To be more precise, they lacked initial plans[2].

We unfortunately did not have a program to calculate the initial plans of the first of the two instances in a reasonable time. However this problem could be resolved easily after some communication.

Ironically none of the programs, except for their own, could solve this instance anyway. And the group's solver did not need the initial plans.

The solver of this group is a suboptimal CBS-algorithm. But it did, at first, only work on their own instances and the ones of the group RaatschenSauerbrei. We are not sure what the exact reason is, but we know that the program needed additional statements in the instance-file to work, because it did not assume initial plans as given.

Later a "fix" for the instances of GlätzerAkil was provided on their GitUP page after which the program worked. Even later other files where provided for the other groups, but the program did still not work on them. There was more comunication, but we ran out of time to find a solution. All the missing values from this group are from instances which returned an error.

Important Note To gather the data presented in this subsection, we used different benchmark-programs which have not been created by standards we set together as a group. That means, that every group used its own measurement of time, which could make a great impact on the runtime. Additionally each and every one of the groups measured different values that cannot really be compared if we do not have them from everyone.

So small differences in the runtime should be interpreted with great caution. Although we cannot say what the threshold for significance is, in general the rule is: the difference is relative to the total time a program needed to calculate an instance.

Also we had no way of checking for optimality except for a few additional categories. These categories can only be compared to RaatschenSauerbrei, since there is no overlap in the measured values between our group and Pan.

Evaluation As mentioned before, the IS and PP programs of RaatschenSauerbei are usally fast in these instances, but incomplete and suboptimal.But PP was always faster than IS and almost always found a solution, so in general it is probably the better option of the two. It is even always the fastest, whether it finds a solution or not. So it could be a great option in combination with a complete algorithm, if the time is more important than anything else.

The instances of GlätzerAkil where unfortunately finished to fast to be significant, be it due to a problem in the instance or because it were just easy examples. So we have no way to compare the results of their benchmark-program to the other groups.

[2] It was agreed upon to take them as given, so that calculating them would not influence the benchmarking.

The benchmark-program of Pan was fast. It was the only program that could solve the instances of Pan in under 5 minutes. Unforunately we lack data to compare it on a lot of instances. In RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp it rivals the time of PP and in RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-w.lp it is much faster then the other CBS-implementations too.

The only indication we have of how suboptimal it is, is the SOC value, which we can only compare to RaatschenSauerbrei. It is always lower than PP, so in that sense it is better. But we could only compare it in one instance to the CBS of RaatschenSauerbrei. In this instance RaatschenSauerbrei war only slightly better, but can only guess how it would be in more complicated instances.

Between our programs and the CBS of RaatschenSauerbrei we can compare the most. Our non-greedy CBS was significantly slower that theirs. This is probably partly due to the different cost-functions we used. SOL is more complicated than SOC, because we set high goals for it to fulfill. In that meaning it is hard to compare the optimality, since SOL does not try to simply lower the horizon or the moves.

Our greedy CBS however is closer in terms of time to the CBS of RaatschenSauerbrei. Although it is slightly slower in the easier instances, this is hard to say for certain. This is because we found, that the RaatschenSauerbrei benchmark-program measured its time more optimistically. This was hard to measure and even harder to record, because we only found this manually with a stopwatch. This is not a problem in itself, since different groups used different benchmark-programs, but it does make it difficult to compare and should be kept in mind. We did not test this for other benchmark-programs except our own.

In RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp our greedy CBS is significantly faster and it is the most complicated instance, that both programs managed to solve. And while the horizon is the same in both solutions, our greedy CBS does need more movements. This difference would habe also been interesting to explore on bigger instances.

5 Conclusion

5.1 Our Own Solvers

We do not have enough data to make definite conclusions. While we did evaluate 356 instances, or 556 solvings, we spread those on many categories and subsets to analyse many aspects and impacts. So each smaller conclusion has relatively little data supporting it. So for the conclusions to be rock solid, we would need to test it on more data.

For small instances the difference between CBS and gCBS was insignificant, but on bigger instances gCBS was much faster while being still relatively good in terms of optimality. We did not have the resources to analyse more complex instances. It would be interesting to analyse how the suboptimality of gCBS develops on a greater scale.

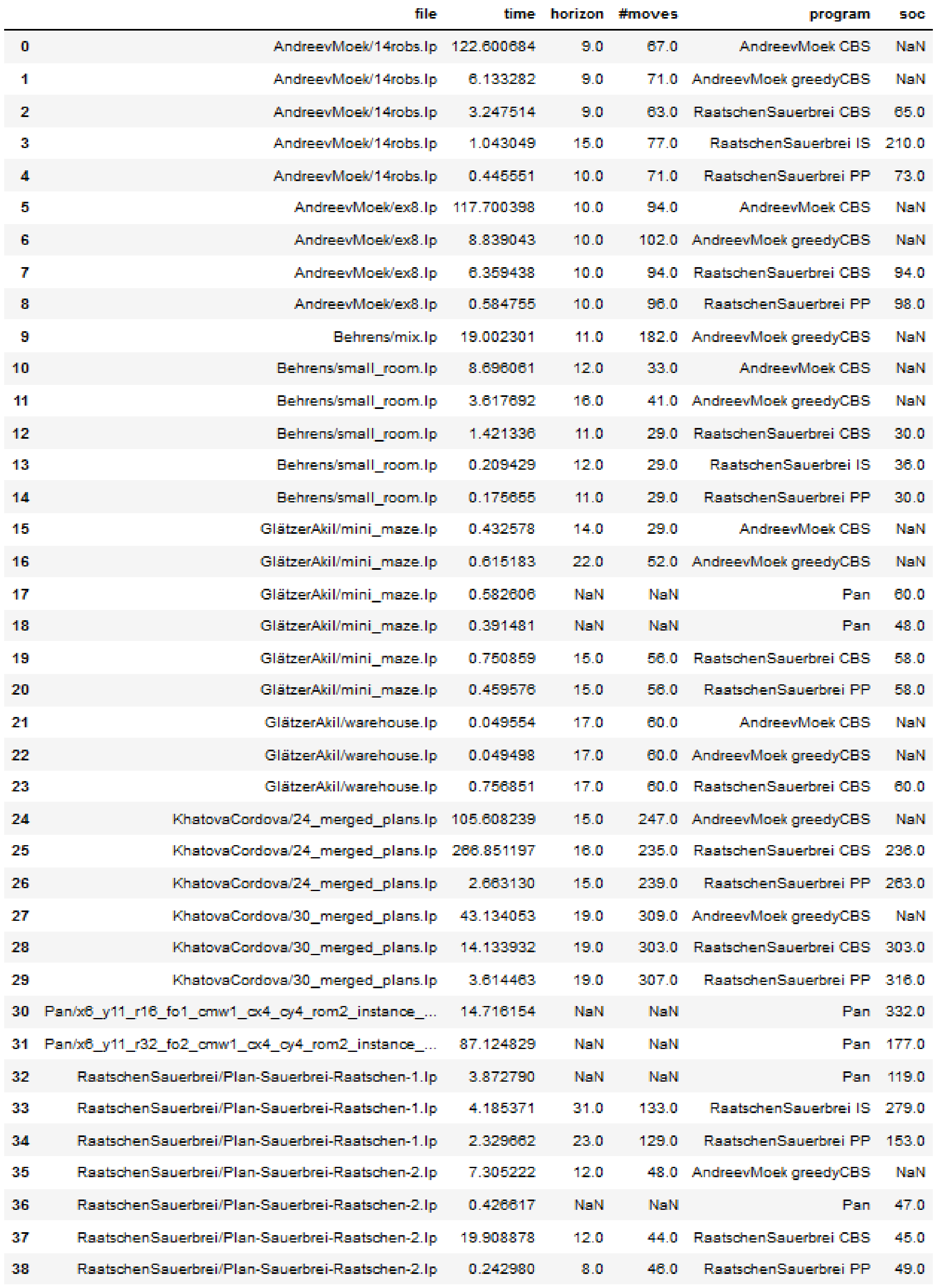
The only benefit of CBS we found is, that it is guarantied to be optimal in SOL. In that sense, it would be interesting to see how it would behave if robots had more that one goal, so they could actually benefit from the premise of SOL.

5.2 Other Groups

Unfortunately we had very little data to go on. But it became apparent, that our CBS was very slow, even compared to other CBS-implementations. Although the very complicated instances, the ones that our CBS-implementation could not solve in time, were of rather high density or unusual set-up. For example one of these instances was 9 robots trying to squeeze through the same door at the same time in two opposite directions, which is probably unlikely in a warehouse scenario.

Still, SOL is mostly interesting only on a theoretical level, because the time to solve instances with it is just to high if there are only singular goals for the robots.

Our gCBS on the other hand could compete with other CBS-implementations and was not much worse in terms of optimality. NOC is probably closer to sum of cost than SOL, which would also be interesting to analyse.



	file	time	horizon	#moves	program	soc
0	AndreevMoek/14robs.lp	122.600684	9.0	67.0	AndreevMoek CBS	NaN
1	AndreevMoek/14robs.lp	6.133282	9.0	71.0	AndreevMoek greedyCBS	NaN
2	AndreevMoek/14robs.lp	3.247514	9.0	63.0	RaatschenSauerbrei CBS	65.0
3	AndreevMoek/14robs.lp	1.043049	15.0	77.0	RaatschenSauerbrei IS	210.0
4	AndreevMoek/14robs.lp	0.445551	10.0	71.0	RaatschenSauerbrei PP	73.0
5	AndreevMoek/ex8.lp	117.700398	10.0	94.0	AndreevMoek CBS	NaN
6	AndreevMoek/ex8.lp	8.839043	10.0	102.0	AndreevMoek greedyCBS	NaN
7	AndreevMoek/ex8.lp	6.359438	10.0	94.0	RaatschenSauerbrei CBS	94.0
8	AndreevMoek/ex8.lp	0.584755	10.0	96.0	RaatschenSauerbrei PP	98.0
9	Behrens/mix.lp	19.002301	11.0	182.0	AndreevMoek greedyCBS	NaN
10	Behrens/small_room.lp	8.696061	12.0	33.0	AndreevMoek CBS	NaN
11	Behrens/small_room.lp	3.617692	16.0	41.0	AndreevMoek greedyCBS	NaN
12	Behrens/small_room.lp	1.421336	11.0	29.0	RaatschenSauerbrei CBS	30.0
13	Behrens/small_room.lp	0.209429	12.0	29.0	RaatschenSauerbrei IS	36.0
14	Behrens/small_room.lp	0.175655	11.0	29.0	RaatschenSauerbrei PP	30.0
15	GlätzerAkil/mini_maze.lp	0.432578	14.0	29.0	AndreevMoek CBS	NaN
16	GlätzerAkil/mini_maze.lp	0.615183	22.0	52.0	AndreevMoek greedyCBS	NaN
17	GlätzerAkil/mini_maze.lp	0.582606	NaN	NaN	Pan	60.0
18	GlätzerAkil/mini_maze.lp	0.391481	NaN	NaN	Pan	48.0
19	GlätzerAkil/mini_maze.lp	0.750859	15.0	56.0	RaatschenSauerbrei CBS	58.0
20	GlätzerAkil/mini_maze.lp	0.459576	15.0	56.0	RaatschenSauerbrei PP	58.0
21	GlätzerAkil/warehouse.lp	0.049554	17.0	60.0	AndreevMoek CBS	NaN
22	GlätzerAkil/warehouse.lp	0.049498	17.0	60.0	AndreevMoek greedyCBS	NaN
23	GlätzerAkil/warehouse.lp	0.756851	17.0	60.0	RaatschenSauerbrei CBS	60.0
24	KhatovaCordova/24_merged_plans.lp	105.608239	15.0	247.0	AndreevMoek greedyCBS	NaN
25	KhatovaCordova/24_merged_plans.lp	266.851197	16.0	235.0	RaatschenSauerbrei CBS	236.0
26	KhatovaCordova/24_merged_plans.lp	2.663130	15.0	239.0	RaatschenSauerbrei PP	263.0
27	KhatovaCordova/30_merged_plans.lp	43.134053	19.0	309.0	AndreevMoek greedyCBS	NaN
28	KhatovaCordova/30_merged_plans.lp	14.133932	19.0	303.0	RaatschenSauerbrei CBS	303.0
29	KhatovaCordova/30_merged_plans.lp	3.614463	19.0	307.0	RaatschenSauerbrei PP	316.0
30	Pan/x6_y11_r16_fo1_cmw1_cx4_cy4_rom2_instance_...	14.716154	NaN	NaN	Pan	332.0
31	Pan/x6_y11_r32_fo2_cmw1_cx4_cy4_rom2_instance_...	87.124829	NaN	NaN	Pan	177.0
32	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp	3.872790	NaN	NaN	Pan	119.0
33	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp	4.185371	31.0	133.0	RaatschenSauerbrei IS	279.0
34	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-1.lp	2.329662	23.0	129.0	RaatschenSauerbrei PP	153.0
35	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-2.lp	7.305222	12.0	48.0	AndreevMoek greedyCBS	NaN
36	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-2.lp	0.426617	NaN	NaN	Pan	47.0
37	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-2.lp	19.908878	12.0	44.0	RaatschenSauerbrei CBS	45.0
38	RaatschenSauerbrei/Plan-Sauerbrei-Raatschen-2.lp	0.242980	8.0	46.0	RaatschenSauerbrei PP	49.0

Fig. 21: Summary of all group's benchmarks