Plan Merging Project Report

Nico Sauerbrei and Paul Raatschen

Potsdam University

Abstract. In this report, we present three different plan merging based multi agent path finding (MAPF) solvers for the Asprilo framework and their implementation in Python and Clingo. 1. Iterative Solving, which merges plans based on iterative handling of edge and vertex conflicts.
2. Prioritized Planning, which merges the plans sequentially, where subsequent plans have to avoid collisions with existing plans.
3. Conflict based Search by Sharon et al. [7], which is a optimal MAPF solver based on a two level search.
We further benchmark our implementations on different asprilo instances, measuring solvability, runtime and solution quality according to different cost functions. We find that Iterative Solving generally performs poorly in terms of solvability and solution quality. Prioritized Planning performs well in terms of runtime and solution quality, dynamically adjusting agent priorities through backtracking further improves solvability at the cost of runtime. Our CBS implementation delivers the best solution quality for different cost functions, but scales poorly in terms of runtime for larger instances.

Keywords: Multi Agent Path Finding · Answer Set Programming · Conflict Based Search.

Table of Contents

1 Introduction

Multi agent path finding (MAPF) is the problem of planning conflict free paths for multiple agents to their targeted destination in a given environment. MAPF is a complex problem with many real world applications, such as robotics, autonomous driving or video games [8]. Plan merging is an approach to solve the MAPF-problem, where agents start off by planning their paths individually, without regard for avoiding conflicts with others agents. These plans are then merged into a MAPF-solution, where conflicts between the individual agents paths have to be mitigated. There a several approaches for plan merging, which include optimal solvers, whose solutions minimizes a chosen cost function and sub-optimal solvers, that trade solution quality for a shorter execution time. Our goal for this project is to explore both optimal and sub-optimal plan merging approaches to MAPF and compare their performance in terms of their ability to find a valid solution, execution time and solution quality. For this purpose, we implement and evaluate three plan merging approaches to the MAPF problem, namely Iterative Solving (IS), Prioritized Planning (PP) and conflict based search (CBS).

We use the Asprilo framework [5] as a basis for describing and visualizing both problem instances and their solutions. Asprilo provides different domains that model a warehouse environment, through which logistics robots navigate to deliver product orders. We specifically focus on the M-domain, where the goal is for robots to move to their assigned shelf from their starting location, excluding more complex interactions with the environment. For implementing our plan merging approaches, we use the answer set programming (ASP) language Clingo [4], in combination with Python, through Clingos Python API.

The following section introduces the problem setting, which we base our plan merging approaches on. ASP components introduces the base ASP encodings used in the implementation of the MAPF solvers. Plan Merging approach details the concepts and implementation of the MAPF solvers. Experimental evaluation presents the benchmarking result for our solver implementations. Conclusion summarizes our conclusions and delivers an outlook on possible improvements.

2 Problem Setting

The Asprilo instances we use as a basis for the MAPF problem setting, are based in a 2D-plane, consisting of agents (robots), shelf's, available nodes and unavailable nodes. An agent can only move into one of four directions, horizontally up and down, or vertically left or right. The plane is structured as a grid, meaning that one move marks the transition from one neighboring cell of the grid to another. At a given time step, an agent can either execute a move or wait in its current position. Each agent has a starting position (X,Y) on the grid and is assigned a shelf on the grid as its goal. For simplicity, we assume that each agent has a number corresponding to their shelf and that no agents share the same number. We define two types of conflicts that can occur between agents. *Vertex Conflict* refers to the case where two robots try to occupy the same node (vertex) on the grid at the same time. *Edge Conflict* refers to the case where two robots try to traverse the same edge between two nodes at the same time , i.e. traverse each others paths in opposite direction. A valid solution to a given instance is a sequence of moves (represented by occurs atoms in the Asprilo syntax), such that all agents reach their goal without either of the two conflict types occurring. An agent at their goal will also remain an obstacle to other agents that have not reached their goal, to keep the setting more realistic. Additionally, we focus on the non-anonymous MAPF setting, hence agents are not allowed to swap their goals in order to avoid conflicts. We choose this restriction, since it makes the problem setting more interesting.

The primary concern of our plan merging approaches is solvability, i.e to find a valid solution for an instance as defined above, if there exists one. To compare the quality of valid solutions, we focus on two metrics commonly used in the

MAPF literature [8]. We define *sum of cost* (SoC) as the number of all move and wait actions agents execute before they reach their goal. We further define *makespan* (MS) as the number of time steps required for all agents to reach their goal. In a single agent context, makespan and Soc are equivalent, but in a multi agent context, optimal makespan and Soc solutions are likely to differ, since a solution optimized for makespan minimizes the maximum as opposed to the aggregate cost.

3 ASP Components

Before we go into detail on our plan merging approaches, we introduce the answer set programming based components that are used in their implementation. As mentioned before, or implementation relies on a hybrid of Python and Clingo encodings, where the high level control flow of the algorithms is implemented in Python and the key solving functions are ASP based. All our ASP encodings are located in *solvers/encodings* and python files in *solvers* within our git repository [6].

3.1 Parsing the Instance

To acquire information about the initial state of the asprilo instance file, we use the encoding in *setup.lp*, which generates a set of atoms describing the available nodes, robots, shelf's and directions of movement.

```
node((X,Y))  robot(R)  goal(R,(X,Y)) direction(DX,DY)
```

Here, X and Y refer to the coordinates on the grid and R to the number of the robot. These atoms are stored in a solution object, which is implemented in *solution.py.* The solution object acts as a container for all information relevant to the solving process and also stores agent plans, as well statistics such as makespan, sum of costs, number of moves and the execution time of the solving process. Furthermore, to be able to compare solutions for different instances, we also save the normalized versions of makespan, sum of costs and number of moves. These are the absolute values divided by the respective values for the initial plans before merging. That way we can measure how well the solvers perform under different conditions, for example, two instances of vastly different size will most like result in solutions with vastly different makespan, Soc etc.

3.2 Conflict Detection

A key aspect of all our plan merging approaches is detecting conflicts between the plans of agents. The files *verify.lp* and *conflict_detection.lp* create a set of conflict atoms, based on position atoms of the form:

```
position(R,(X,Y),T)
```

These describe the position (X,Y) of an agent R for a given point in time T. The conflict atoms differentiate between vertex and edge conflicts.

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

conflict(edge,R,R',((X,Y),(X',Y')),T-1) :-
position(R,(X,Y),T-1), position(R,(X',Y'),T),
position(R',(X',Y'),T-1), position(R',(X,Y),T), R!=R'.

conflict(vertex,R,R',(X,Y),T) :-
position(R,(X,Y),T), goal(R',(X,Y)),
goalReached(R',T'), R!=R', T >= T'.
```

The last conflict rule retains the positional information of agents that are stationary at their goal. Hence, when plans of different sizes are merged, collisions of agents that are beyond the scope of one of their plans are also detected.

3.3 Plan generation

Plan generation is another central function used in all our plan merging approaches. To generate a path for a single agent from its origin to its goal, we use the *singe_agent_pf.lp* file. This encoding uses aspects of the base Asprilo encodings[1] and Clingos multishot solving incremental mode. Following Clingos incremental mode, our encoding is separated into a step and a check program. The step program generates the plan for a given time step as described above. The check program verifies if the agent has reached their goal at the current time step.

```
#program step(t).
{ move(r,D,t) : direction(D) } 1.
position(r,(X+DX,Y+DY),t) :- move(r,(DX,DY),t), position(r,(X,Y),t-1).
:- move(r,(DX,DY),t), position(r,(X,Y) ,t-1), not node((X+DX,Y+DY)).
position(r,C,t) :- position(r,C,t-1), not move(r,_,t).
```

The choice rule in the second line allows agents to execute at most one move in one direction at a given time t. The position atom of the agent is then updated according to the chosen move for conflict detection. The integrity constraint in the forth line ensures that moves only lead to valid grid positions. The last rule also generates position atoms if an agent does not execute a move, where merely the time is incremented and the location is constant.

```
#program check(t).
#external query(t).
goalReached(r,t) :- goal(r,C), position(r,C,t).
:- not goalReached(r,t), query(t).
```

[1] https://github.com/potassco/asprilo-encodings/tree/develop

The search for a path starts at the time point t = 0 and expands the horizon incrementally, through multiple calls to the ASP solver. The integrity constraint in the last line requires the agent to have reached its goal for satisfiability. The query atom is set externally, to ensure that the constraint only applies to the current time step. If the solver call produces no model for a time step t, t is incremented by one and the solver is called again. This incremental approach ensures, that the path obtained will have a minimal horizon , since the search terminates for the first horizon, where the solving result is satisfiable, i.e. the agent reaches its goal. In situations where there is no valid path, the search would not terminate, therefore we limit the number of iterations with a maximum horizon [2].

For our implementations, we decided to generate the initial shortest plans of agents to their goal as part of the solving process. This is merely a choice of convenience to make the testing of the solvers more flexible. All our plan merging approaches could be trivially adapted to use precomputed initial plans.

4 Plan Merging Approach

The following section describes the conceptual ideas and implementation of our plan merging approaches.

4.1 Iterative Solving

The Iterative Solving approach is initialized by finding a set of optimal plans i.e shortest paths for all agents to their goal, which is implemented with our single agent pathfinding encoding introduced in 3.3. Subsequently, existing conflicts between the agents plans are solved one at a time, meaning, we solve one conflict between two agents, by altering their paths, before checking again for new conflicts. Here, we again distinguish conflicts into vertex and edge conflict,with the additional difference, that vertex conflicts are solely those conflicts, that can be solved by one agent waiting right before the point of conflict. Edge conflicts are all conflicts where agents traverse onto each other paths after the point of conflict, requiring at least one agent to leave their path to avoid the conflict.
To solve the conflicts efficiently, we prioritize the resolution of edge conflicts over vertex conflicts, since it alters the agent paths instead of simply inserting wait actions. Algorithm 1 illustrates the high level control flow of the procedure, which is implemented in *iterative_solving.py*. The following sections will detail the handling of vertex and edge collisions.

4.1.1 Vertex Collision Handling As seen in Fig. 1, to solve a vertex conflict, one of the two agents has to wait for the other to pass, before they can

[2] We choose a maximum horizon of 2*Number of available nodes, which should leave enough room for even heavily constraint paths to terminate. A larger horizon would negatively impact performance for unsolvable path finding problems.

continue on their original path. To choose the agent that has to wait, we observe the number of moves an agent has left towards its goal position, starting from the point of conflict. Choosing the agent with the shorter remaining path can ensure, that the joint makespan does not have to increase as a result of the wait action, unless both agent have the same distance to their goal. // The only exception to this rule occurs, when one of the agents is heading in the direction the other agent came from. In a scenario like this, it could happen, that the agent not wanting to move into the direction of the other agent, waits. This would cause the other agent to continue onto the point of conflict, before facing the waiting agent and finding him blocking the path forward. In a scenario like this, the agent which would block the path would move first, as to not cause a further edge collision. The handling of a vertex conflicts is implemented in the *solve_vertex_cl.lp* encoding.

```
Rules for the priority system.

choice(R',prio(1)):- firstvertextConflict((X0,Y0),T0,R,R'),
position(R,P1,T0-1), position(R',P1,T0+1).

{choice(R,prio(0)) : firstvertextConflict((X,Y),T,R,R') ,
position(R',(X'',Y''),TLongest), longestTime(TLongest)} = 1.
```

The rules above determines the waiting agent based on a priority system. Priority 1 means that an agent would blockade the path by waiting, while priority 0 indicates the agent with the shortest path. The restriciton choice rule ensures that only on agent with priority 0 is chosen, for the case where both paths have the same remaining length.

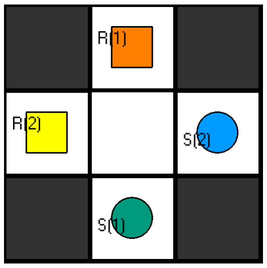

Fig. 1. A visualisation of a vertex conflict, in which both agents want to cross over the same point at the same time.

4.1.2 Edge Collision Handling The handling of edge conflicts is more complex than that of vertex conflicts, as seen in Fig. 2. An edge conflict occurs whenever two agents share the same segment of their path, in the same time interval,

while moving in opposing directions. Our approach to solve a conflict like this, is that one agent has to leave its path and move into a non blocking position, while the non evading agent passes by. To achieve this, we let both agents search for a non blocking free space, along the section of their path that they have already passed, before the collision takes place. We require for simplicity's sake, that the free space is never visited by the non evading agent. In the right circumstances, the evaiding agent can find a space which will not interfere with the remaining path of the other agent. After all valid evading positions are found, we choose the one closest to the point of collision, since the further away the evasion step is from the conflict, the longer the evaiding agent has to wait for the other agent to pass by.
The time an agent has to wait is estimated pessimistically, by choosing the last time point, at which the non evading agent occupies the location the evading agent left to dodge out of the way. This means, that if an agent wanders over the same square multiple times, with opportunities for the other agent to continue its path, the opportunity will not be taken. This rule is mainly in place for complex instances, in which agents modify their path multiple times, where leaving the evading spot prematurely could lead to further conflicts.

An edge case of edge collisions can occur, when an instance has more than 3 agents, with one agent waiting in a spot of another agent's path. In cases like this, the waiting agent often moves out of the way, due to this mostly happening to agents either at their goal or agents waiting for other agents to pass. The handling of edge conflicts is implemented in *solve_edge_cl.lp*.

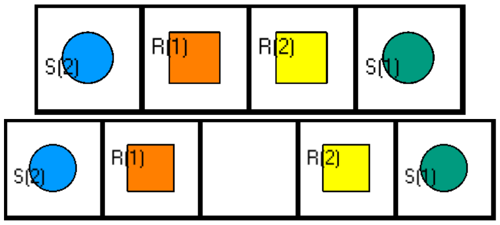

Fig. 2. A visualisation of the two kinds of edge conflicts. The upper showing the agents wanting to swap position, the ladder showing the agents wanting to move onto the same path.

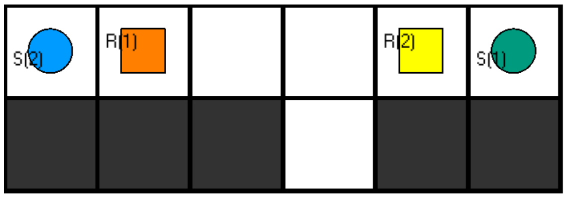

Fig. 3. An example of a simple edge collision, where agent number 2 will let agent number 1 pass.

4.1.3 Applying Both Handling Procedures To solve all occurring conflicts between a given set of initial plans, we first solve all edge conflicts iteratively, with the procedure described above. After all edge conflicts are resolved, we iterativly solve all vertex conflicts, to the point where a new edge conflict appears, or no more conflicts exist. In the first case, we revert back to the iterative solving of edge conflicts, in the second case the solver terminates successfully.
Iterative solving delivers no formal guarantees for a fast execution time, an optimal solution, or to find a solution to the instance at all. In the end, every agent will traverse down their original path, altered with actions which either force the agent to wait in place, or to move to the a non blocking location and to let another agent pass. Furthermore, the more iterations the solver conducts, the more complex the path of each conflicting agent becomes. Due to this, areas with a high conflict density can cause the algorithm to get stuck in loops, where each agent blocks one another, despite some agents being theoretically able to go to their goal without causing a conflict. Since previous iteration dictate that an agent should evade first, the agent may simply wait for a cluster of agents to separate, that all wait for the same event without it ever occurring.
To ensure the algorithm eventually terminates, we added a maximum number to both edge and vertex iterations, to prevent the algorithm from entering infinite loops. All in all, Iterative Solving is prone to accumulate unnecessary moves as soon as multiple agent conflict with each other, since conflict resolutions from previous iteration will always remain, even when subsequently solved conflicts negate the need for the earlier solutions. The extend of this conceptual flaw will be evaluated in the benchmarking section.

Algorithm 1: Iterative Solving

```
Input: Agents: $a_1, .., a_n$, edgeIter, vertexIter
Output: Plans: $p_1, ..., p_n$
for $agent_i \in order$ do
    $p_i \leftarrow$ Shortest path to goal ;
end
conflicts $\leftarrow$ Conflicts between paths $p_1, ..., p_n$ ;
solveEdge() ;
solveVertex() ;
if conflicts is empty then
    plans $\leftarrow \{p_1, ..., p_n\}$ ;
end
return plans ;
Function solveEdge():
    while $edgeIter > 0$ do
        if edge conflict in conflicts then
            Solve edge conflict and update plans $p_1, ..., p_n$ ;
            conflicts $\leftarrow$ Conflicts between paths $p_1, ..., p_n$ ;
            edgeIter $\leftarrow$ edgeIter - 1 ;
        else
            return ;
        end
    end
return ;
Function solveVertex():
    while $vertexIter > 0$ do
        if vertex conflict in conflicts then
            Solve vertex conflict and update plans $p_1, ..., p_n$ ;
            conflicts $\leftarrow$ Conflicts between paths $p_1, ..., p_n$ ;
            vertexIter $\leftarrow$ vertexIter - 1 ;
            if edge conflict in conflicts then
                solveEdge() ;
                if edge conflict in conflicts then
                    break ;
                end
            end
        else
            return ;
        end
    end
return ;
```

4.2 Prioritized Planning

Prioritized planning is a well known approach for solving MAPF problems in the context of robotics [3][9]. The idea behind it is simple. All agents $a_1, ..., a_n$ are ordered according to a given priority, where each agent has a distinct place in the ordering. The agents then plan their paths sequentially in this order. The first agent in the ordering can freely plan the shortest path to their goal. All consecutive agents will have to account for the plans of the agents that planned before them i.e. they have to avoid collisions with already existing plans. Algorithm 2 illustrates the Prioritized Planning procedure, with the optimal extensions of schedule optimization and backtracking.

The high level control flow of the algorithm is implemented in *prioritized_planning.py* in the *solvers* directory of our repository. It assumes the numeric order of agents as the default ordering and plans them sequentially. The single agent pathfinding (Line 13 of algorithm 2) is implemented with our ASP encoding introduces in 3.3, with additional rules.

```
:- position(r,(X,Y),t), position(R,(X,Y),t), r!=R.
:- position(r,(X,Y),t-1), move(r,(DX,DY),t),
position(R,(X+DX,Y+DY),t-1), position(R,(X,Y),t), r!=R.
:- position(r,(X,Y),t), goal(R,(X,Y)), goalReached(R,T),
r!=R, t > T.
block(1..T-1) :- position(R,C,T), goal(r,C), r!=R.
```

Conflicts with known plans of other agents are prevented through integrity constraints. The first constraint above prevents vertex conflicts, while the second one prevents edge conflicts. The last two rules prevent agents from colliding with other agents stationary at their goal. The position atoms of all prior paths are added in grounding to the searches of subsequent agents, in order receive a collision free path. If, for any agent, no path is found within the given horizon, the solver will consider the instance unsolvable and terminate.

Prioritized planning is not complete. Figure 4 represents a minimal example of an instance that can never be solved by Prioritized Planning, even if the order of priority is changed. In general, whenever a situation occurs where a solution requires every agent to deviate from their optimal path, Prioritized Planning will not be able to solve it. In addition to that, Prioritized Planning does not guarantee optimality of the solution with respect to makespan or sum of costs.The procedure takes no consideration of joint cost and the strict priority of earlier plans may lead to arbitrarily bad cost increases in the following plans, in order to avoid conflicts. The main advantage of Prioritized Planning is speed. The run time complexity of the base algorithm is in $O(N * A)$, where N is the number of agents and A the average time it takes to compute a path for a single agent. To address the solvability limitations of Prioritized Planning, we introduce to extensions covered in the following sections.

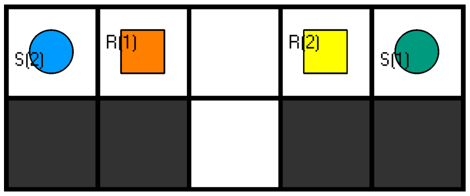

Fig. 4. Minimal example for an instance that is not solvable with Prioritized Planning.

Algorithm 2: Prioritized Planning

```
Input: Agents: $a_1, .., a_n$, optimize, maxIter
Output: Plans: $p_1, ..., p_n$
order ← $a_1, .., a_n$
orderings ← {*order*}
plans ← $\emptyset$
if *optimize* then
    forall *agent*$_i \in$ *order* do
        $p_i$ ← Shortest path to goal
    end
    Find conflicts between paths $p_1, ..., p_n$
    order ← Agents sorted by number of conflicts
end
while *maxIter* $> 0$ do
    forall *agent*$_i \in$ *order* do
        $p_i$ ← Shortest path to goal without collisions with $p_1, ..., p_{i-1}$
        if $p_i.cost = \infty$ then
            orderings ← orderings ∪{All orderings beginning with $a_1, ..., a_i$}
            Change agents $a_i$ and $a_{i-1}$
            order ← $\{a_{i-1}, ..., a_n\}$
            if *order* $\in$ *orderings* then
                order ← Closest ordering that is not in orderings
            end
            orderings ← orderings ∪{*order*}
            maxIter ← maxIter - 1
            break
        else
            plans ← plans ∪{$p_i$}
        end
    end
end
return *plans*
```

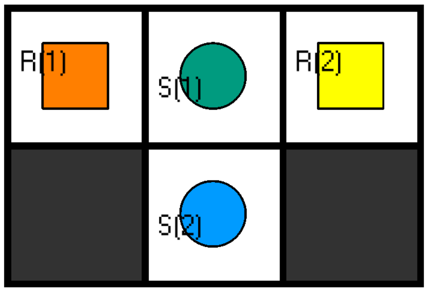

Fig. 5. Minimal example for an unsolvable initial ordering (1,2)

4.2.1 Optimizing Agent Priority The initial ordering of agents can affect the quality and even the existence of a solution obtained by Prioritized Planning. Hence, absent the need for a specific order (due to schedules etc.), the schedule should be chosen in a way that allows for a solution (if there is one) and ideally optimizes either makespan or sum of cost (under the constraint of a priority order). Finding an optimal ordering without actually computing the solutions for all possible orderings is not a trivial task. Hence, we opted for a simple heuristic solution, that delivers no guarantees on improving the solution. Our optimization heuristic is illustrated in lines 4-9 of Algorithm 2. For each agent, we compute the shortest path to their goal without avoiding collisions with other plans. We then use *verify.lp*, to determine the number of conflicts between these initial plans for each agent. The new ordering is then arranged by sorting the agents by the number of conflicts they're involved in. The idea is to prioritize agents with no or little conflicts to minimize the impact of their priority on subsequent plans. If two agents have the same conflict count, the agent with the shorter path is prioritized, since shorter paths should generally be less likely to block other agents. Of course, there are scenarios where this heuristic may be detrimental to the solution quality or event prevent a valid solution. We therefore made the optimization mode optional to the solver and test its impact empirically.

4.2.2 Backtracking In addition to initial schedule optimization, we added backtracking as another expansion to Prioritized Planning, with the goal of increasing the chance of finding a valid solution. Backtracking is illustrated in lines 11-28 of Algorithm 2.The basic idea of backtracking is to adapt the ordering, if the current one does not lead to a solution and then try again. If the initial ordering leads to an invalid solution for agent a_i and a_i is not the first agent, the agent is swapped with its predecessor a_{i-1} in the ordering. Additionally, all permutations of agents starting with the current order up to agent a_{i-1} are added to a list of used orderings, since they cannot lead to a valid solution[3]. The new

[3] If an agent finds no solution at position i in an ordering it will also not find a solution at positions $> i$ if the order of agents at positions $< i$ is not changed.

ordering is also added to the used list, in order to avoid repetition on subsequent re-orderings. Line 18 of the procedure checks if the new ordering has already occurred or is invalid. If that is the case, the ordering with maximum distance to the already tried ones will be chosen, if there is one. The idea behind this is to maximize the amount of information that can be gained from the new ordering and thereby the number of additional orderings that can be excluded, if the new one is invalid as well. Once the new ordering has been chosen, the planning procedure backtracks to the first agent in the new ordering, who's priority has changed and proceeds planning sequentially. Backtracking significantly expands the space of valid solutions that can be obtained with Prioritized Planning. It also affects the run-time complexity of Prioritized Planning, as, in the worst case, $N!$ permutations of agents will have to be tested, where N is the number of agents. On average, the number of backtracks is likely a lot lower, since any unsolvable ordering that is identified before the last agent will mark a multitude of permutations as used. *permutation_tools.py* implements helper functions for keeping track of used orderings. Instead of saving the actual permutations, we map them to an index $\in 0...N! - 1$ and store invalid orderings as a set of non overlapping index ranges. This limits memory usage, as well as lookup time, when checking for already used orderings.

4.3 Conflict Based Search

Conflict based search (CBS) by Boyarski et al. [1] is a MAPF algorithm based on a two level search. At the high level, CBS searches a constraint tree consisting of nodes that contain constraints on agent movements and plans that satisfy these constraints. Nodes are chosen in a best first manner, based on a cost function computed for the plans they contain. The low level search finds a plan for an agent in accordance with a set of constraints. Algorithm 3 details the procedure, including the improved CBS and meta agent CBS extensions, which are discussed in a separate section. At the start, the constraint tree is initialized with a root node that has an empty set of constraints. The low level search is invoked for the root node, planning an optimal path for each agent. Our base CBS implementation utilized the single agent pathfinding encoding, introduced in section 3.3, for the low level search of CBS. The root node is then inserted into the open queue. While the open queue is not empty, the lowest cost node in the queue is removed and its plans are evaluated for conflicts. In our implementation, this is done with the *verify.lp* encoding introduced in section 3.2. If no conflicts are found, the node is a solution node and the search terminates. If conflicts are found, the first conflict is chosen for branching. The branch procedure is detailed in lines 41-50 of algorithm 3. When a conflict tree node is branched at the vertex conflict $(a_1, a_2, (X, Y), T)$, two new nodes are created. The first node contains the additional constraint $(a_1, (X, Y), T)$, prohibiting agent a_1 from accessing the position (X, Y) at time T. The second node contains the same constraint for agent a_2. Correspondingly, for an edge conflict $(a_1, a_2, (X, Y), (X', Y'), T)$, the constraint $(a_i, (X, Y)(X', Y'), T)$ would prohibit agent a_i from moving from (X, Y) to (X', Y') at time T. For each of the new nodes, the low level search is invoked for the agent with the additional constraint. If it returns a valid solution, the cost of all plans in the node (according to a specified cost function such as SoC or MS) is updated with the cost of the new plan and the node is inserted into the open queue, in ascending order of cost.

Constraint symbols are generated in Python and added in grounding when the low level search encoding is invoked. They are saved in a plan object, which exist for every agent in our implementation of a conflict tree node.

```
:- constraint(r,C,t), position(r,C,t).
:- constraint(r,C,M,t-1), position(r,C,t-1), move(r,M,t).
block(1..T-1) :- constraint(r,C,T), goal(r,C).
```

The integrity constraints above are an extension to 3.3 for our CBS implementation. The first rule prevents an agent from accessing a position C at time t, while the second one prevents the execution of move M from position C at time t. The last rule serves to block an agent from finishing its plan before the time T, in order to avoid conflicts with other agents at their goal[4].

Conflict Bypassing

Boyarsky et al.[2] suggest conflict bypassing as an improvement to CBS, which we implement in all of our CBS versions. The bypassing procedure is detailed in lines 28-33 of algorithm 3. When a conflict is chosen for branching, conflict bypassing checks whether a valid bypass for a conflict exists in one of the nodes before branching. A valid bypass for a conflict exists, if one of the two agents can find an alternative path to its goal, that has the same cost and does not create a new conflict. Hence, after creating a node with a new constraint and invoking the low level search for the newly constraint agent, we compare cost and conflict count to its parent. If the cost has not increased and the conflict count is lower, the node is inserted back into the open queue without branching. This limits the number of branches, as well as the number of low level search invocations and should improve overall performance, without affecting the solution quality.

Completeness and optimality

CBS is both complete and optimal. The intuitive argument for completeness is ,that, starting from the constraint free root node, each branching action creates a branch for each possibility of avoiding the conflict. Hence, the whole tree preserves all possible scenarios that allow for a valid solution. The optimally argument follows from the order in which nodes are explored. Since CBS always chooses the lowest cost node from the open queue and all possible solutions are part of the constraint tree (due to completeness), the first solution node has to be the lowest cost solution [5]. Therefore, baring implementation errors, we expect our CBS solver to obtain optimal solutions for the specified cost function.

[4] The integrity constraint in the “check” part of our pathfinding encoding in 3.3 is adapted to require the absence of a block atom.

[5] A more formal proof for the completeness and optimality of CBS can be found in [7]

Cost function variation and greedy search Since the high level search is guided by the solution cost of the nodes in the open queue, CBS allows for the optimization of different cost function. We include a cost function variation in our implementation, that allow to either chose sum of costs (default) or makespan as a cost function to be used at the high level of CBS. In addition to that, we implement a greedy version of the high level search. In this variant, nodes are evaluated based on the chosen cost function + the number of conflicts that exist between the current plans of the node. This obviously breaks the optimality of CBS, since the cost function is no longer the sole determinant of node expansion. However, choosing a node with fewer conflicts should, on average, lead to a solution more quickly, improving the runtime of CBS. We test the affect of this greedy heuristic in the benchmarking section.

4.3.1 Meta Agent CBS Meta agent CBS (MACBS) is a generalisation of the base CBS procedure [7]. The idea is, to allow for the merging of heavily conflicting agents into a meta agent. The meta agent is then treated as a single agent by the low level search, hence the low level search has to be a MAPF solver.This reduces the size of the conflict tree, since conflicts between the agents in a meta-agent are handled via the low level search, instead of branching at the high level. The MACBS procedure is detailed in lines 10-20 of algorithm 3. When a conflict between agents a_1 and a_2 has been identified, their bilateral conflict count is incremented. In our implementation, each node in the constraint tree stores a conflict matrix object, that keeps track of the conflict count for all pairs of agents. If the conflict count exceeds a specified threshold, the agents are chosen for merging. MACBS allows for a trade-off between high level and low level search, which can be controlled with the size of the conflict threshold. A low threshold will prefer merging over branching, reducing the size of the conflict tree, but increasing low level search effort. If the agents are chosen for merging, they are marked as a new meta agent in the conflict matrix object of the current node and all its children. After merging the MAPF low level search is invoked for the node and the node is reinserted into the open queue with updated costs, if a solution was found. We use an adapted version of our single agent pathfinding encoding from 3.3 as a MAPF solver.

```
cost(N,t) :-
N = #sum{1, R, T : position(R,C,T), not goal (R,C)}.
#minimize{ N : cost(N,t)}.
```

The minimization statement above represents the most significant change from the single agent version of the encoding. While in the single agent context, the search finds both a makespan and sum of cost optimal solution, this is not the case in the multi agent context. The search only terminates, once the agent with the longest path has found its goal, optimizing makespan but not necessarily sum of costs for agents with shorter plans. The minimization statement aims to additionally reduce sum of cost, by minimizing the number of positions agents assume before reaching their goal. Unfortunately, we were not able to create a

Algorithm 3: Conflict Based Search

```
Input: Agents: $a_1, .., a_n$, meta, icbs
Output: Plans: $p_1, ..., p_n$
root ← Node initialized with the shortest paths for all agents;
open ← {root};
while open not empty do
    node ← Lowest cost node from open;
    conflicts ← Conflicts between plans in node;
    if conflicts is empty then
        return node.plans
    end
    if meta and threshold exceeded for agents $a_1, a_2$ in first conflict then
        current.meta-agents ← current.meta-agents$\cup\{(a_1, a_2)\}$;
        if icbs then
            restart search with $(a_1, a_2)$ as meta agent.
        end
        Update plans in node with low-level search for meta agent $(a_1, a_2)$;
        if node.cost $< \infty$ then
            open ← open $\cup$ {node};
            continue;
        end
    end
    if icbs then
        if cardinal conflict c in conflicts then
            branch(node,c);
            continue;
        end
        if semi cardinal conflict sc in conflicts then
            branch(node,sc);
        else
            if valid bypass exists for conflict then
                Update plan in node with bypass;
                open ← open $\cup$ {node};
            else
                branch(node,non cardinal conflict nc in conflicts);
            end
        end
    else
        branch(node,first conflict c in conflicts);
    end
end
return plans;
Function branch(node n, conflict c):
    $node_1$ ← Node with added constraint for agent $a_1$ in conflict;
    $node_2$ ← Node with added constraint for agent $a_2$ in conflict;
    forall $node_i \in \{node_1, node_2\}$ do
        Update plan in $node_i$ for agent under additional constraint;
        if $node_i.cost < \infty$ then
            open ← open $\cup\{node_i\}$;
        end
    end
return ;
```

MAPF encoding that produces makespan or sum of costs optimal solutions in an efficient manner, since the minimization clause introduces a significant increase in runtime for most instances. Violating optimality at the low level, leads to sub-optimal solutions for the entire solver, however our encoding appears to find close to optimal in most scenarios. We therefore opted for the current version, which still produces high quality solutions at an affordable runtime, at least for smaller problem settings.

4.3.2 Improved CBS Improved CBS is another improvement to CBS suggested by Boyarski et al. [1] It consist of two extensions, namely *merge and restart* and *prioritizing conflicts*, detailed in lines 11-13 and 20-34 of algorithm 3. Merge and restart affects only MACBS. It causes, that after every merging of (meta) agents, the entire search is restarted from the root, retaining merely the meta agent information of the current node. In regular MACBS, meta agents are local to the node that merged them and its children, which may lead to a lot of redundant merging of the same (meta) agents across different branches. Merge an restart, makes the status of the meta agents global to the entire constraint tree, which prevents this problem from happening at the cost of losing all other search progress prior to the merging.
The second change ICBS introduces is prioritizing conflicts when branching. For this purpose, three different conflict types are distinguished. Cardinal conflicts lead to a increase in solution cost in both nodes after branching. Semi cardinal conflicts will increase the cost in one of the nodes. Non cardinal conflicts do not increase costs in both branched nodes. The base version of CBS branches conflicts arbitrarily, which can lead to an unnecessary inflation of the conflict tree, if the choice of conflict is poor. Cardinal conflict occur at points that both conflicting agents have to pass for their optimal path. Boyarksi et al. argues that cardinal conflicts should be branched with priority over the other conflict types, since they will have to be handled to obtain a solution. Delaying them over other conflict types will result in a lot of additional branching of nodes with the same cost, which still contain the cardinal conflict.

Our implementation of ICBS branches all conflicts detected by *verify.lp* sequentially. If a cardinal conflict is found by comparing the cost of both new nodes to their parent, the search terminates and the nodes are inserted into the open queue. Otherwise, all conflicts are branched, but only the nodes of one conflict are inserted into the open queue, with semi cardinal conflicts taking priority over non cardinal conflicts. This done in order to not inflate the open queue with the children of a single node. Our chosen conflict evaluation procedure is quite costly, especially when cardinal conflicts are evaluated last or do not occur at all. This may reduce the performance gain of ICBS or even be detrimental to overall performance in certain instances. Due to a lack of time, we did not investigate further improvements to conflict evaluation and instead leave this open for future work.

5 Experimental Evaluation

In this section, we describe the procedure and results of evaluating the implementations of our plan merging approaches.

5.1 Procedure

We evaluate the performance of each algorithm based on the following criteria:

- Makespan.
- Normalized makespan, refering to the makespan of the solution, divided by the makespan of the initial plans before plan merging.
- Sum of cost.
- Normalized sum of cost, refering to the sum of costs of the solution, divided by the sum of costs of the initial plans before merging.
- Total moves, referring the number of all of moves executed by agents in the solution.
- Solvability, being a boolean indicator if the algorithm managed to produce a valid solution for an instance.
- Execution time, measured from beginning to the end of the solving procedure (excluding file I/O for the solution).

To accurately analyze the scaleability of each algorithm, we wrote an instance generator, which is able to generate randomized yet structured layouts for the instances. We benchmark on generated instances with increasing complexity in regards to the layout itself, as well as to the number of agents. For the generation of instances we provide 3 basic layouts, which features can be adjusted as needed. The first type of layout is simply called "Random" and randomly places walls within the filed. The second type of instances is called "Grid" 8, generating a grid like pattern, after specifying a number of vertical as well as horizontal slices, for the given instance size. Each cell is then connected via a door to its neighbours. The last layout an instance can be generated as, is called "Rooms", and as the name suggest, it generates a small rectangle of walls in the instance, again connecting the inner space with the other space. Yet to avoid a simplification of this layout, for example, when two rooms generate right next to each other and block each other doors, causing the instance to be divide into 3 independent spaces, we ensure an empty space between two walls. While being the layout with most variation we decided not to use it, due to the layout already requiring a decent instance size, causing CBS solvers to be unable to scale in terms of execution time.
Furthermore, to gather a greater accuracy on the results we iterate multiple times over separate instances of the same complexity level, allowing us to take the average from multiple runs to get a more stable estimate and minimize the effect of outlier instances.

We further benchmark our solvers on the instances provided by other groups, including the instances provided by us.

5.2 CBS Benchmarks

Due to the generally slow execution times of our CBS implementation, finding instances for testing the strength and weaknesses of the different CBS versions compared to each other is rather challenging. Hence, the instances we use remain simple in scope, in order to obtain results within a reasonable time.

The naming scheme of the variation of CBS is as followes:

1. CBS type: Base (" "), Greedy ("G") Meta agent ("M")
2. Improved CBS variant? Yes ("I") No (" ").
3. Cost function used for high level search: Makespan ("CBS-MS")Sum of cost ("CBS-SOC")

We chose a number of different CBS setups for evaluation. CBS represents the base CBS procedure without the Meta agent or ICBS extensions. MCBS is the meta agent version of CBS. We chose the meta agent merging threshold as 2 for all of our benchmarking, since we observe that at larger thresholds, very little merging occurs for the given benchmarks. ICBS is the base CBS version extended with prioritized conflict branching. We all CBS version with both makespan an sum of costs optimization. We further test ICBS with the greedy high level variant, since it would be (a priori) expected to yield the best runtime of all CBS variants (excluding meta agent CBS due to the problems with the low level search encoding).

We examine the procedurally generated instance X5Y5R5RB, with the width and height of 5 and 5 agents. With every iteration, we increasing number of randomly blocked nodes to reduce the number of available nodes and make pathfinding more challenging. We note that despite the blocked nodes being randomly determined, the agent goals are placed so that the initial single agent path finding will always be able to determine a valid path. With every iteration one blocked node gets added, starting by 1 and ending by 8, with each iteration having 4 sub-iterations with the same constraints, used to determine an average performance of the solvers. 6. We chose this setup to determine how our CBS versions respond to environments of different density. The Table 6 presents the execution time for the CBS evaluation on X5Y5R5RB. It most notably shows an enormous spike for some solvers, which are the result of solvers being unable to solve certain sub-instances of X5Y5R5RB within the specified time frame of 5 minutes. The solvers which did not solved some of the sub-instances in time were "CBS-SOC", "ICBS-SOC", "CBS-MS" and "ICBS-MS", meaning only greedy and meta variants managed to solve the instance in a reasonable time. If we only plot those solvers we receive a comparison between the six remaining solvers. 7

One surprising result is the relatively large amount of time greedy improved CBS for sum of cost takes in the later iterations. A possible explanation for this behavior could be that optimizing for sum of costs is inherently more challenging, since it requires optimally in the plans for all agents, whereas makespan only requires optimality for the agents with the longest paths. The meta agent sum of cost variants may be less affected, due to the suboptimality in the in low level search procedure. We do not observe a clear trend between the iterations for other

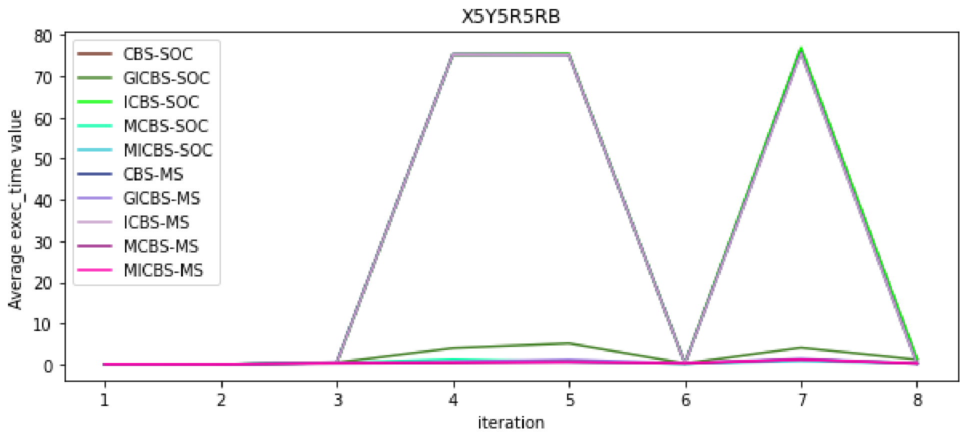

Fig. 6. Average execution time of the CBS solvers on X5Y5R5RB.

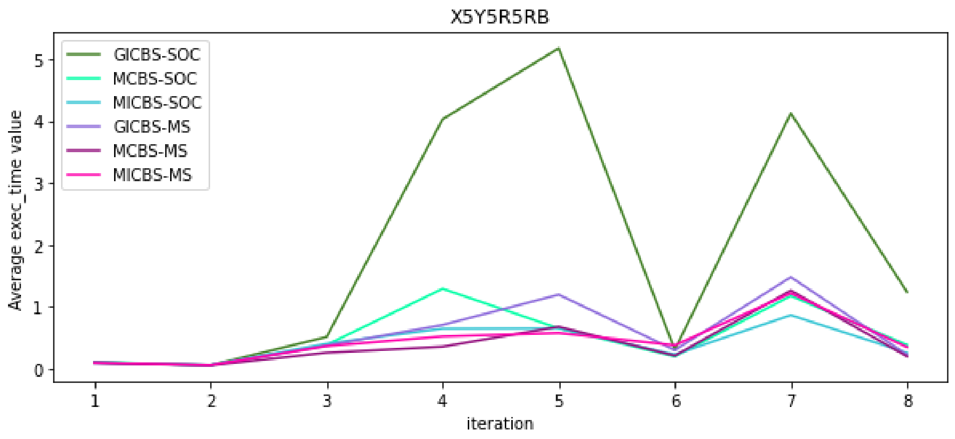

Fig. 7. Average execution time of the CBS solvers, which managed to solve all instances in reasonable time, on X5Y5R5RB.

solvers, though the high density iterations appear to exhibit a slightly longer runtime for all solvers on average. Furthermore, due to the CBS variants in figure 7 performing way better than their non greedy and non meta agent counterparts, we only will focus on them in the following comparison to prioritized planning and iterative solving. This allows us to increase the difficulty in tests, while receiving results in a resonable time.

5.3 Compared Benchmarks

To compare Iterative Solving as well as prioritizes planning with the fast versions of CBS, we let them solve multiple grid like instances, where the size is fixed by 10 x 10 and the layout was the same as in 8 , with an ever increasing number of agents per iteration. Again for each number of agents, we average the results over 4 variations of the instance, to receive a more stable estimate of the performance. Our intention with this benchmark is to access the scalability in terms of number of agent between the different solvers.

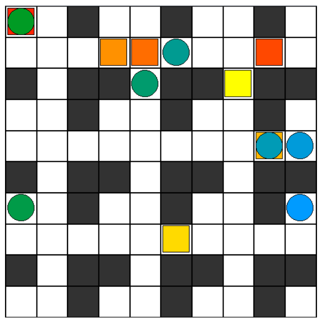

Fig. 8. An example instance for X10Y10Grid3R, with 7 agents in total.

Fig. 9 presents the execution time for the experiment. We observe, that for all iterations up until 6, all solvers find a solution fairly quickly. In iteration 8, the values for the meta agent CBS variant skyrocketed dramatically, which can likely be attributed the bad scaling behavior of our low level MAPF encoding. Both the makespan and sum of cost optimized variants of greedy improved CBS performed fairly well on the last iteration, even outperforming prioritized planning, probably due to the need for backtracking on the last iteration.

Figure 10 presents the average normalized makespan and sum of costs for all iterations of the experiment. We observe, that all solvers exhibit a very similar performance, with the notable exception of Iterative Solving severely underperforming in both metrics. Both prioritized planning and greedy CBS are surprisingly close to the other CBS variants, though it should be mentioned again, that the meta agent CBS variant will likely not find optimal solutions, due to their sub optimal low level search.

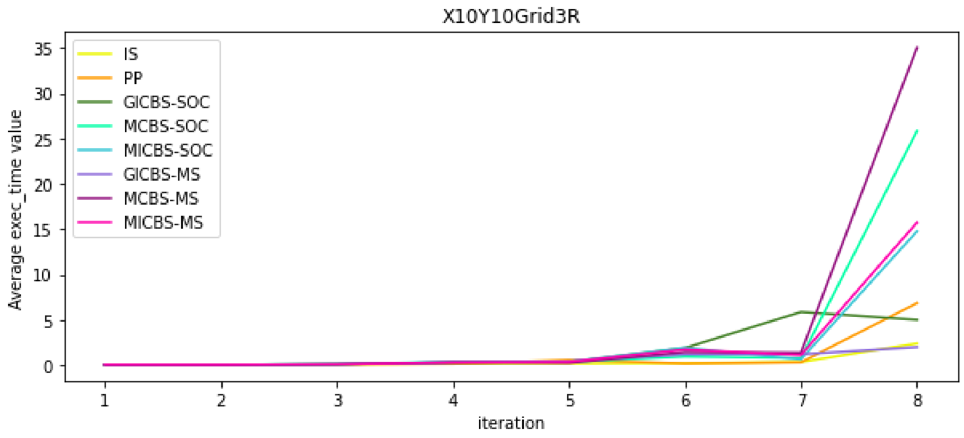

Fig. 9. Average execution time of Iterative Solving, Prioritized Planning, as well as the fast CBS methods, for X10Y10Grid3R. Iteration = Number of agents. Note, we filtered only by solved instances which only affected Iterative Solving, since it had an instance it could not solve.

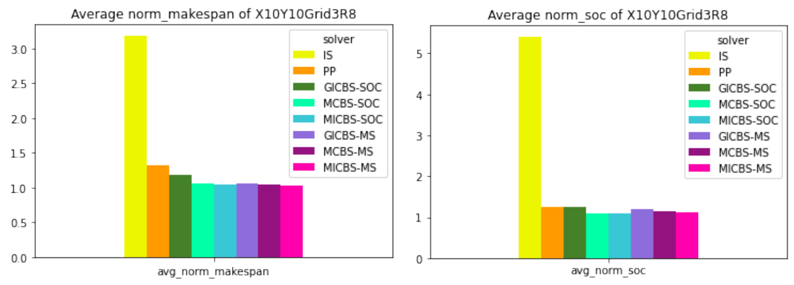

Fig. 10. Average normalized makespan and sum of cost for X10Y10Grid3 with 8 agents.

5.4 Benchmarks from other groups

In the following section, we present our benchmark results on the joint group benchmarks. For all instances from others group, we let each solver run with a timeout of 20 minutes. We test the same CBS configurations as in the first CBS benchmark. For Prioritized Planning we enable backtracking by default and test the schedule optimization separately. The result tables show the makespan, sum of costs, number of moves and execution time in seconds. Solvers that did not find a solution within the timeout are not listed.

For images of the instances, please check our repository [6] or preferably our presentation slides.

Steven Pan

The instances from Steven Pan were by far the hardest instances for all of our solvers, which is likely due to their high density and number of agents. The results for the first instance can be seen in Table 1. Only prioritized planning and the makespan optimized version of greedy improved CBS were able to find a solution within the timout. The latter finds a significantly better solution, both in terms of makespan and sum of costs, in a little more than 3 times the execution time of prioritized planning. The runtime of prioritized planning is also significantly larger compared to most other instances, which is due to a large amount of backtracking required, in order to find a solution. The second instance presented in Table 2 was only solved by Prioritized Planning. Here, the schedule optimization heuristic leads to a significant decrease in runtime, which was not the case for the first instance.

Table 1. Steven Pan Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	20	201	160	38.000
PP-OPT	23	229	175	44.000
GICBS-MS	14	175	140	141.726

Table 2. Steven Pan Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	22	353	285	12.500
PP-OPT	24	339	291	1.480

Moek Andreev

Tables 3 and 4 present the results for the instances of the group Moek Andreev. For both instances, most CBS variants and Prioritized Planning where able to find a solution fairly quickly, with the notable exception of MICBS-SOC in instance 1. Additionally, all CBS versions that exceeded the timeout used meta agent CBS, which is further evidence of the performance issues with our low level MAPF encoding. Generally, Prioritized Planning performed the best in terms of execution time and also surprisingly well in terms of makespan and sum of costs, compared to the optimal CBS solutions. The greedy heuristic on CBS does also not significantly reduce solution quality for these instances, while proving slighly better runtime on average.

Table 3. Moek Andreev Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	10	73	71	0.471
PP-OPT	10	69	69	0.728
CBS-SOC	9	65	63	2.848
GICBS-SOC	9	69	65	4.190
ICBS-SOC	9	65	63	2.222
MCBS-SOC	9	65	63	2.658
MICBS-SOC	9	65	63	113.399
CBS-MS	9	71	67	2.578
GICBS-MS	9	72	63	2.561
ICBS-MS	9	72	63	3.743

]

Table 4. Moek Andreev Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	10	98	96	0.483
PP-OPT	10	108	100	0.903
CBS-SOC	10	94	94	5.806
GICBS-SOC	10	99	96	4.684
ICBS-SOC	10	94	92	3.762
MCBS-SOC	10	95	94	7.440
CBS-MS	10	101	100	4.535
GICBS-MS	10	100	96	3.805
ICBS-MS	10	100	96	5.467

Jan Behrens
For the first instance of Jan Behrens' 5 only prioritized planning and MICBS found a solution within the timeout, which is likely a result of the large number of agents in the instance. Again prioritized planning is close to CBS in terms of solution quality. We also continue to observe that patterns, that makespan optimization appears to be more efficient than sum of costs optimization for CBS. For the second instance 5 all solvers find a solution in under one second. MICBS notably does not find an optimal both in terms of sum of cost and makespan.

Table 5. Jan Behrens Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	11	195	162	0.597
PP-OPT	11	195	162	0.886
MICBS-SOC	11	185	164	53.831
MICBS-MS	11	185	164	27.951

Table 6. Jan Behrens Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
IS	14	40	29	0.155
PP	11	30	29	0.100
PP-OPT	11	30	29	0.188
CBS-SOC	11	30	29	0.762
GICBS-SOC	11	30	29	0.376
ICBS-SOC	11	30	29	0.764
MCBS-SOC	11	30	29	0.492
MICBS-SOC	12	33	31	0.639
CBS-MS	11	30	29	0.784
GICBS-MS	11	30	29	0.357
ICBS-MS	11	30	29	0.768
MCBS-MS	11	30	29	0.410
MICBS-MS	12	33	31	0.634

Glätzer Akil

The instances 78 from Glätzer and Akil's group were both solved efficiently by all solvers, with the exception of iterative solving not being able to solve the second one. It also exhibits a significantly worse solution quality for the first instance.

Table 7. Glaetzer Akil Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
IS	23	90	64	0.443
PP	15	58	56	0.234
PP-OPT	15	58	56	0.391
CBS-SOC	15	58	56	0.376
GICBS-SOC	15	58	56	0.361
ICBS-SOC	15	58	56	0.398
MCBS-SOC	15	58	56	0.364
MICBS-SOC	15	58	56	0.372
CBS-MS	15	58	56	0.408
GICBS-MS	15	58	56	0.401
ICBS-MS	15	58	56	0.358
MCBS-MS	15	58	56	0.453
MICBS-MS	15	58	56	0.494

Table 8. Glaetzer Akil Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	17	60	60	0.255
PP-OPT	17	60	60	0.443
CBS-SOC	17	60	60	0.366
GICBS-SOC	17	60	60	0.377
ICBS-SOC	17	60	60	0.383
MCBS-SOC	17	60	60	0.370
MICBS-SOC	17	60	60	0.353
CBS-MS	17	60	60	0.384
GICBS-MS	17	60	60	0.380
ICBS-MS	17	60	60	0.364
MCBS-MS	17	60	60	0.370
MICBS-MS	17	60	60	0.363

Cordova Khatova
Tables 9 and 10 present the results for the instances of group Cordova Khatova. For the first instance, all Meta agent CBS version exeeded the timeout, likely do to performance issues with the minimization statement in the MAPF encoding. We continue to observe the trend that makespan optimization is more efficient for CBS. Also the greedy CBS variants yielded a significant reduction in runtime, while only suffering a small increase in sum of costs and none in makespan. For the second instance all solvers managed to find a solution, where interestingly base CBS performs the best out of all CBS variants. Prioritized planning again finds a close to optimal solution in a fraction of the runtime of CBS.

Table 9. Cordova Khatova Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	15	263	239	1.374
PP-OPT	15	261	237	2.384
CBS-SOC	16	236	235	216.957
GICBS-SOC	15	249	233	37.274
ICBS-SOC	16	236	233	265.565
CBS-MS	15	260	237	30.610
GICBS-MS	15	258	233	22.595
ICBS-MS	15	259	239	57.423

Table 10. Cordova Khatova Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	19	316	307	2.382
PP-OPT	19	321	307	4.635
CBS-SOC	19	303	303	9.287
GICBS-SOC	19	304	303	12.997
ICBS-SOC	19	303	303	10.856
MCBS-SOC	19	303	303	11.494
MICBS-SOC	19	303	303	61.630
CBS-MS	19	303	303	15.251
GICBS-MS	19	337	307	17.904
ICBS-MS	19	337	307	22.749
MCBS-MS	19	303	303	14.711

Nemes Murphy

Tables 11 and 12 present the results for the instances of the group Nemes Murphy. For the first instance, only Prioritized Planning and the meta agent CBS variants reach a solution within the timeout. We observe a notable difference in the sum of cost of the MICBS and MCBS variants, illustrating again the sub optimality of our meta agent CBS implementation. The second instance is solved rather quickly by all solvers (except iterative solving), with the notable exception of makespan optimized CBS and meta agent CBS taking significantly longer. Comparing both instances, we again observe that CBS seems to perform better in less dense environments.

Table 11. Nemes Murphy 1

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	9	52	41	4.048
MCBS-SOC	9	52	47	13.441
MICBS-SOC	9	46	43	32.535
MCBS-MS	9	52	47	10.904
MICBS-MS	9	46	43	28.284

Table 12. Nemes Murphy 2

Solver	Makespan	SoC	Num of Moves	Execution Time
PP	10	58	56	0.149
PP-OPT	10	53	52	0.282
CBS-SOC	10	50	48	2.363
GICBS-SOC	10	52	50	0.821
ICBS-SOC	10	50	48	2.372
MCBS-SOC	10	51	50	0.732
MICBS-SOC	10	51	48	2.387
CBS-MS	10	65	56	20.891
GICBS-MS	10	52	50	1.199
ICBS-MS	10	52	50	1.879
MCBS-MS	10	54	50	13.019

Sauerbrei Raatschen

We selected our first instance as an example for an instance that CBS struggles with, due to a single point of conflict that all agents have to pass. The second instance is an example for a small high density instance that can be solved relatively quickly by all of our solvers. Tables 13 and 14 present the results for our own two instances. For the first instance, Iterative Solving and Prioritized Planning find a solution rather quickly, though Iterative Solving performs significantly worse in both makespan and sum of costs, mainly as a result of additional waiting actions. MICBS is the only CBS variant that succeeds within the timeout, though it still needs more than 13 minutes to terminate. It also only reaches a marginal makespan improvement and no sum of costs improvement over Prioritized Planning. For the second instance all solvers terminate in reasonable time. For CBS, the meta agent variants and GICBS for sum of costs perform well, while regular ICBS performs quite badly for makespan optimization, taking almost 3 time the runtime of base CBS.

Table 13. Sauerbrei Raatschen Instance 1

Solver	Makespan	SoC	Num of Moves	Execution Time
IS	31	279	133	2.088
PP	23	153	129	1.347
PP-OPT	28	171	135	2.355
MICBS-SOC	21	153	147	788.276

Table 14. Sauerbrei Raatschen Instance 2

Solver	Makespan	SoC	Num of Moves	Execution Time
IS	26	176	50	0.470
PP	8	49	46	0.122
PP-OPT	12	45	44	0.178
CBS-SOC	12	45	44	11.591
GICBS-SOC	12	45	44	3.603
ICBS-SOC	12	45	44	11.843
MCBS-SOC	8	47	42	2.683
MICBS-SOC	8	53	44	2.070
CBS-MS	8	50	46	9.090
GICBS-MS	11	51	44	13.544
ICBS-MS	8	50	44	27.841
MCBS-MS	8	48	44	2.017
MICBS-MS	8	53	44	2.016

Overall the benchmark results are mixed. Iterative solving exhibited the worst performance of all solvers, finding no solution for the majority of instances and highly sub optimal solutions otherwise. Prioritized Planning performed well in terms of execution time and solvability, Solving all instances, often in less than a few seconds and still under a minute when backtracking was necessary. The schedule optimization heuristic yielded inconclusive result in terms of a performance increase, especially since the sample size of instances requiring backtracking was very low. We also did not observe any systematic influence on the solution quality in terms of makespan and sum of costs. For CBS, the results are also somewhat inconclusive. Neither ICBS nor meta agent CBS where consistently able to outperform base CBS, which in the case of meta agent CBS is likely due to performance issues with the low level MAPF solver. ICBS also likely suffers from the enormous overhead of conflict evaluation, that only pays of when a sufficient number of cardinal conflicts occurs. Greedy CBS, on average, performed better than non greedy CBS, with a small reduction in solution quality. Meta agent CBS also produced sub optimal solutions, though like greedy CBS, the divergence from the optimal solution was usually quite small.

6 Conclusion

In this report we presented and evaluated our implantation of three plan merging based MAPF solvers. For Iterative Solving we can conclude that our initial "naive" conflict solving procedure of waiting and evading do not scale very well to problems that require the solving of a multitude of conflicts. Iterative solving was not able to solve a significant number of benchmarks correctly, since the iterative conflict solving has no formal guarantee of monotonically decreasing the number of conflicts. For the instances it did solve, accumulated waiting actions from the conflict solving procedures lead to very suboptimal sum of cost and makespan values. We therefore consider the approach conceptually flawed and probably not worth pursuing further.

Our Prioritized planning implementation delivered the best solver performance overall, but especially with respect to execution time. Sum of cost and makespan values were more often than not close to the CBS solutions, while the runtime never exceeded 1 minute on all of our benchmarks. Introducing backtracking also seems to significantly improve the range of solvable instances, as we did not encounter an unsolvable instance during the benchmarking with backtracking enabled. The backtracking procedure offers room for further improvement, since our current implementation does not always chose the ordering that reveals the maximum amount of information on untested orderings. The schedule optimization heuristic had mixed empirical success and can also be improved further, with a more complex approach.

Our CBS implementation succeeded in providing the best solution quality but performed poorly in terms of execution time, making it mostly impractical for use on instances beyond a certain size. Both MACBS and ICBS did not provide a significant improvement in execution time, which is likely due to our poorly optimized implementation of them. The MACBS variant can be improved by finding a better low level MAPF encoding, which we lack due to us not having the time to optimize further. ICBS can similarly be improved by finding a better conflict evaluation method, possibly using an estimate instead of actually branching each conflict and evaluating the resulting nodes. Our sub optimal greedy CBS variant could also be improved, for instance by varying the high level cost computation for the best first search of the conflict tree. Another possibility would be the relaxation of the optimality in the low level search, in favour of a faster approach.

Overall we can conclude that focusing on three solvers and their variations was probably too much for the scope of this project, which is why we could not further pursue a lot of the above mentioned optimizations. Nevertheless, we gained some insight in the comparison of optimal and sub optimal MAPF solvers, most notably, the generally small loss in solution quality between Prioritized Planning and CBS.

References

1. Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., Shimony, E.: Icbs: Improved conflict-based search algorithm for multi-agent pathfinding. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
2. Boyrasky, E., Felner, A., Sharon, G., Stern, R.: Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 25, pp. 47–51 (2015)
3. Čáp, M., Novák, P., Kleiner, A., Seleckỳ, M.: Prioritized planning algorithms for trajectory coordination of multiple mobile robots. IEEE transactions on automation science and engineering **12**(3), 835–849 (2015)
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot asp solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019)
5. Gebser, M., Obermeier, P., Otto, T., Schaub, T., Sabuncu, O., Nguyen, V., Son, T.C.: Experimenting with robotic intra-logistics domains. Theory and Practice of Logic Programming **18**(3-4), 502–519 (2018)
6. Sauerbrei, N., Raatschen, P.: Plan merging project repository, `https://github.com/PaulRaatschen/Plan-Merging-Project-Sauerbrei-Raatschen`
7. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40–66 (2015)
8. Stern, R., Sturtevant, N.R., Felner, A., Koenig, S., Ma, H., Walker, T.T., Li, J., Atzmon, D., Cohen, L., Kumar, T.S., et al.: Multi-agent pathfinding: Definitions, variants, and benchmarks. In: Twelfth Annual Symposium on Combinatorial Search (2019)
9. Van Den Berg, J.P., Overmars, M.H.: Prioritized motion planning for multiple robots. In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 430–435. IEEE (2005)