Solving the Multi-Agent Path Finding with Waypoints Problem Using Subdimensional Expansion

Jeroen van Dijk
TU Delft

Abstract

Little to no research has been done on the multi-agent path finding with waypoints problem (MAPFW) even though it has many important real world applications. In this paper we extend an existing algorithm for the multi-agent path finding problem (MAPF) called M* [1]. We do so by ordering the waypoints using a Travelling salesman problem solver before creating a policy for each waypoint and the target. Lastly we extend the underlying A* planner to keep track of the visited waypoints. This results in the new WM* algorithm which will be shown to be performing better than other recently created algorithms with respect to run-time while being optimal in the case of the ordered waypoint variant and near optimal in the unordered variant.

1 Introduction

The multi-agent path finding problem (MAPF) problem is the problem of finding a path of minimal cost for each agent from their original location to their target location without having any collisions with other agents [2]. To solve this problem efficiently is crucial with real life applications being robot navigation in warehouses or the scheduling of public transport[1] [3]. At the moment there are already many algorithms available that solve the MAPF problem such as conflict-based search [4], branch-and-cut-and-price [5] and M* [1] [6].

Many of these algorithms are optimal but run in exponential time since MAPF is an NP-hard problem [7]. There are some more efficient algorithms available at the cost of optimality. Selecting an algorithm to solve a specific MAPF problem is often a trade off between the time to find a solution and the quality of the solution. Even though we are able to solve the general MAPF problem, not much research has been done on the MAPF problem with extra constraints such as waypoints. An exception is the multi-label A* algorithm [8] which solves the MAPF problem with one waypoint, which is known as the Multi-Agent Pickup-and-Delivery (MAPD) problem.

In this paper we extend the M* algorithm to solve the multi-agent path finding with waypoints problem (MAPFW). This extended algorithm will be called WM*. To extend M* we generate a policy for each waypoint and the target of every agent and extend the underlying A* planner to keep track of the visited waypoints. We then use sub-dimensional expansion [1] to find the solution with the lowest cost and without collisions. To evaluate how well our proposed algorithm solves the MAPFW problem we compare the results with the results of recently developed algorithms [9] [10] [11] [12] on a set of benchmarks.

WM* is shown to outperform these recently developed algorithms on all benchmarks with respect to run time with the cost being negligible larger than the optimal cost. WM* can be further improved by making it recursively call itself or by implementing an Operator Decomposition.

First we give a formal description of the problem, followed by an explanation of the M* algorithm and my contribution. Then we describe the experimental setup and show the results of the new algorithm in section 4 where we also compare it to recently developed algorithms. Section 5 reflects on the ethical aspects of our research and discuss the reproducibility of our methods. Lastly we summarize the research question and answer it, while also giving some further questions and ideas for improvements.

2 Background information

2.1 The MAPFW problem

In this section we define the MAPF problem and then describe the extension to the MAPFW problem. The MAPF problem is a part of Graph Theory and in this paper it is defined as an undirected graph $G = (V, E)$, where V is a set of vertices and E a set of undirected edges. Every vertex also has a an edge to itself. Furthermore, the set A contains agents and each agent $a \in A$ has a start vertex and target vertex $s_a, t_a \in V$. A path p_a with cost $c \in \mathbb{Z}$ is a list of locations $(v_0, v_1, ..., v_{c-1})$ where $v_0 = s_a$ and $v_{c-1} = t_a$ and $(v_j, v_{j+1}) \in E$ for all $j \in 0, 1, ..., i-2$. The cost of a path p is the sum of the traversed edges $e \neq (t_a, t_a)$. So a solution for the MAPF problem consists of a path for each agent where at any time step there are no collisions. A collision can be either 2 agents who are at the same vertex at the same time or when during the same time-step one agent traverses from $a \in V$ to $b \in V$ while another agent traverses from a to b. An optimal solution minimizes the total cost of all the paths combined.

For the MAPFW problem every agent is also assigned a list of waypoints $\forall w \in W \subset V$ which it has to visit. The waypoints can be either ordered or unordered. If they are ordered a path p is valid if it visits all the waypoints in the order, but if they are unordered a path p is valid if it visits all the waypoints.

2.2 The M* algorithm

Before we describe how we extended the M* algorithm [1] [6] we explain how the M* algorithm works and the aspects that we build on. The most important things to know and where we build upon are the policies and the configurations. The first thing the M* algorithm does is generating an optimal policy for every agent. That policy describes how to reach the target from any node. The configurations are a datastructure which stores the current location of every agent. Moreover, a configuration has a cost, a backpropagation set, a back pointer and a collision set. The cost describes the cost to reach this configuration from the starting configurations. The backpropagation set of configuration x contains configurations for which the planner has considered x a possible successor. So when a configuration gets expanded it is added to the backpropagation set of its neighbours. The back pointer points to the predecessor configuration. The collision set of configuration x is the sets of robots involved in a collision at x or along a path passing through x already explored by the planner.

M* relies on subdimensional expansion to find the best solution. In the best case every agent is able to follow their policy without colliding with other agents and the optimal path is easily found. Otherwise it encounters a collision at some configuration x. Then that collision is added to the collision set of each configuration in the backpropagation set of x. This is done recursively until that collision is already in the collision set of that configuration. Then for configurations with a non-empty collision set, the agents not in that set follow their optimal policy and those not in the set consider all neighbouring nodes. Since an agent can wait on a vertex that is also considered as a neighbour. In the pseudocode this is done by the $get\_neighbours$ method. From this it is clear to see that if all the agents are in the collision set the amount of configurations expanded is 5 to the power of the number of agents.

3 Your contribution

3.1 One waypoint

We first extend the M* algorithm to solve the MAPF problem with one waypoint per agent before solving the ordered and unordered MAPFW problem. To do so we compute an individual optimal policy for every agent's waypoint and target. This means that from any vertex the shortest path and distance to the way point and target are known. We will then extend the configuration to also include a list called target indices containing booleans that indicate whether agent with index i has already visited the waypoint. This allows us to differentiate between configurations with the same coordinates but where one configuration has already visited one or more waypoints. The target indices also allow us to calculate a minimal heuristic since for every agent you can sum the distance of the agent's vertex to the waypoint and the distance from the waypoint to the target vertex if the waypoint has not been visited. In case the waypoint has been visited we just use the distance from the agent's vertex to the target vertex. Furthermore we add an extra check before returning the final path, namely whether every agent has visited its waypoint. If that is not the case the algorithm keeps running until we encounter a configuration which is at the target and has visited all the waypoints or there are no more configurations to expand.

The main body of the algorithm is still very similar to M* since in the best case every agent just follows its optimal policy, first to the waypoint and then to the target vertex. In cases where collisions occur we rely on sub dimensional expansion described in Section 2.2. This also means that the given solution is optimal since the policies are optimal and M* is optimal with optimal policies [1] [6].

3.2 Multiple ordered waypoints

Secondly we extend the algorithm to solve the ordered variant of the MAPFW problem. In this variant every agents needs to visit the waypoints in the order that is given to them. To do this we start by computing an individual optimal policy for every agent's waypoints and target. Next we convert target indices to a list of integers and we create a list of vertices called targets. This is basically the list of waypoints with the target appended to it. The current target is read from the target list using the target indices. This allows us once again to easily differentiate between configurations and calculate a minimal heuristic. The main body of the algorithm is once again the same and this is also still an optimal algorithm since the policies are optimal and one can not change the order of the waypoints.

3.3 Multiple unordered waypoints

Lastly we look at the unordered variant of the MAPFW problem. In this variant we are allowed to change the order of the waypoints. The general idea here is the same as for the ordered waypoint variant except that the order of the way points can be changed. So we turn it into an ordered variant after changing the order of the waypoints. To change the order of the waypoints an optimal Travelling Salesman Problem (TSP) solver will be used. The TSP problem finds a tour given a graph where a tour is the shortest route which visits every node and returns at the starting node.

This is extremely useful since we can create a graph per agent where every waypoint, the starting vertex and target vertex are the nodes. We can then calculate the distance between every node and turn those into weighted edges. To force the tour to visit the start and end node in order an extra node is added between the start and end node with the edges having a weight of 0. Since the tour must visit every node it must thus visit this extra node as well. Which then forces the start and end node to be following each other although maybe in the wrong order. In which case you must reverse the tour.

The generated tours are optimal and thus it is an optimal order for the waypoints. Nonetheless this will not result in an optimal algorithm for the unordered variant. This is because we do not generate an optimal policy. Only an optimal order

Algorithm 1 WM*

```
Generate an optimal policy for every waypoint
Sort the waypoints as described in Section 3.2
$start.target\_indices \leftarrow [0] * n\_agents$
$start.cost \leftarrow 0$
$open \leftarrow v_s$
while $open.empty = False$ do
    $current \leftarrow open.pop()$        ▷ Get the cheapest configuration
    if $current = target$ & all the waypoints have been visited then
        [We found the solution]
        return $current.back\_ptr + current$
    for $nbr \in get\_neighbours(current)$ do
        $nbr.target\_indices \leftarrow current.target\_indices$
        Increment the target index for any agent that is on their target waypoint in nbr
        [$\phi$ returns the index of the agents in collision]
        $nbr.collisions.update(\phi(nbr, current))$
        $nbr.back\_set.append(current)$
        $backpropagate(current, nbr.collisions, open)$
        $f \leftarrow$ the cost of the move from current to nbr
        if $\phi(nbr, current) = \emptyset$ & $current.cost + f < nbr.cost$ then
            $nbr.cost \leftarrow current.cost + f$
            $nbr.back_ptr \leftarrow current.back\_ptr + current$
            $open.push(nbr)$
end
```

for the starting configuration has been generated. To generate an optimal policy would be unfeasible since in the worst case it would require running an optimal TSP-solver from every node for every subset of waypoints. We need to do this because of collisions an agent can wander off their optimal path in which case a different order of waypoints might result in a solution with less cost. The reason why this is unfeasible is because TSP is an NP-hard problem. But using a TSP solver at the start will still give a very close approximation to the optimal solution.

We will also introduce the inflated WM* algorithm which is extremely similar to the inflated M* algorithm [1] [6] since we multiply the heuristic with some $\epsilon > 1$. This reduces the time needed to find a solution and the cost is $\epsilon *$ cost of WM*. The inflated heuristic has two advantages, the first being that it gives more priority to configurations closer to the target solution. Because the heuristic is smaller there it gets less influenced by the inflation. Secondly, these configurations will generally have less collisions and thus inflated WM* will generally run in a lower dimensionality.

4 Experimental Setup and Results

To gather results for the WM* algorithm [13] it was run on benchmarks created on mapfw.nl. This website was made by Stef Siekman and Noah Jadoenathmisier and contains both single situation benchmarks (Figure 1) and benchmarks that are randomly generated and increase in difficulty. First we show the cost comparison between the algorithms to show

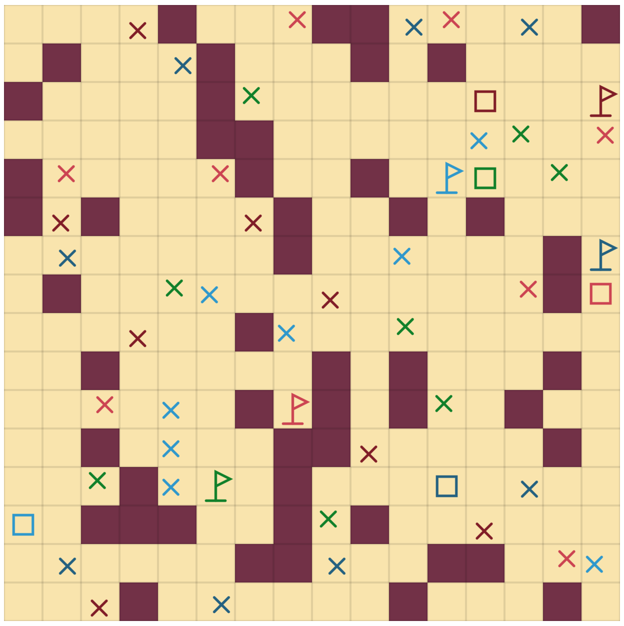
Figure 1: An example of a benchmark on mapfw.nl.

that although WM* is not optimal it is extremely close to optimal and even in the case of the inflated WM* it is still within that 10% extra cost. It is an extra 10% since we use an ϵ of 1.1. Then we also show the results on the progressive benchmarks of a random grid of 16x16, 32x32 or 64x64 where 20% of the locations is a wall with 5,10 or 13 waypoints with an increasing amount of agents. The algorithm has 20 seconds to find a solution which is always available. Such a small timeout still gave useful results but also allowed us to test multiple versions. Lastly there are benchmark created by Timon Bestebreur to compare the recently developed algorithms on certain specific scenario's. The results of those will be discussed in [14].

The results of the individual benchmarks seen in Figure 4 and Figure 5 have been generated using an AMD Ryzen 5 3600 processor with 16GB RAM available and a WM* implementation in C++ [13]. The results of the progressive benchmarks seen in Figure 2 and Figure 3 have been generated on a 14 core Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz processor with 8GB RAM available provided by the TU Delft using the python implementation of WM*[13]. This makes the progressive benchmarks more easily comparable since everyone has run those on the TU Delft server using python except BCP which used C++.

As you can see in Figure 4 and Table 1 WM* is not always the optimal solution as we already showed but on average it is always really close and even inflated WM* is always within 10% from the optimal cost. From Table 2 it is also clear to see that in the worst case scenario the extra cost is under 2,5% for WM* and under 5% for inflated WM*. As expected WM* performs worse on maps with many choke points such as corridors. This is because agents are unable to change their order of waypoints. Thus if they all need to go through the choke-point at the same time most agents will

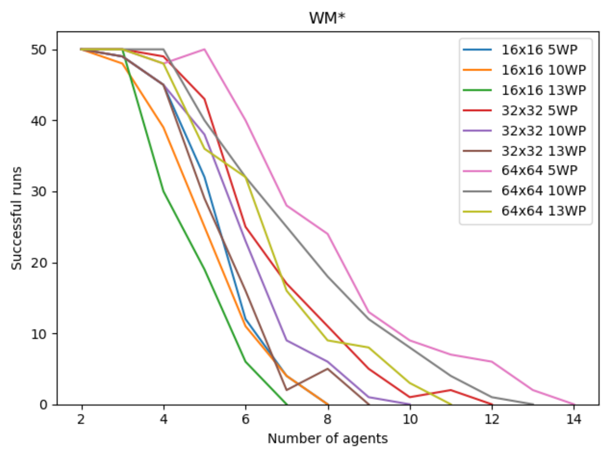

Figure 2: The results of the WM* algorithm on the progressive benchmarks.

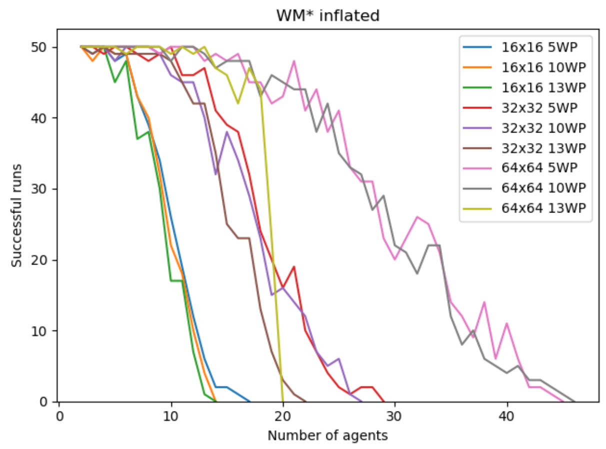

Figure 3: The results of the inflated WM* algorithm on the progressive benchmarks.

Progressive benchmark	WM*	inflated WM*
32x32 5WP (#70)	0.02	0.36
32x32 10WP (#71)	0.02	0.28
32x32 13WP (#72)	0.04	0.21
Average grade (#79)	0.02	0.27
Chokepoints (#80)	0.30	0.50
Corridors (#81)	0.10	0.42
Waypoint overlap (#82)	0.05	0.29

Table 1: The average percentage increase of the cost of WM* and inflated WM* in comparison to the optimal cost for different kind of progressive benchmarks. The number behind them refer to the benchmark number on mapfw.nl.

Progressive benchmark	WM*	inflated WM*
32x32 5WP (#70)	0.90	2.47
32x32 10WP (#71)	0.80	2.04
32x32 13WP (#72)	0.75	1.20
Average grade (#79)	0.67	1.92
Chokepoints (#80)	2.26	3.76
Corridors (#81)	2.16	4.56
Waypoint overlap (#82)	0.97	2.14

Table 2: The maximum percentage increase of the cost of WM* and inflated WM* in comparison to the optimal cost for different kind of progressive benchmarks. The number behind them refer to the benchmark number on mapfw.nl.

wait near the choke-point while it is probably more efficient to visit another waypoint first.

Furthermore, inflated WM* solves more individual benchmarks as seen in Figure 4 because it is more target oriented. This is also clear from Figure 2 and Figure 3 which shows the results of the WM* algorithm in comparison to inflated WM* on the progressive benchmarks. It is obvious that inflated WM* is able to solve instances on the same board with more agents, with the number of maximum agents is generally twice as high for inflated WM*. One interesting thing to note is that at the 64x64 map with 13 waypoints it drops down from over 40 successful runs to 0 while only adding 2 extra agents. We believe that this is caused by the TSP-solver taking up most of the 20 seconds. Then WM* has little time to find the paths in a relatively large map which therefore is less of a hinder in the 16x16 and 32x32 progressive benchmarks.

From Figure 5 you can also see that the run-times of WM* and inflated WM* are generally rather low compared to the other algorithms. The exception is the extension of the heuristic algorithm Multi-Label A* [9], but this does show that inflated WM* is generally much faster at the expense of having a solution that in the worst case has 10% extra cost compared to the optimal solution. Yet in practive this is generally much lower. There are some benchmarks where the run-time is much worse, these are often problems with many agents that will have collisions. This is the worst case since the algorithm then becomes A* and explores n^5 configurations where n is the number of agents. This could be improved on as will be discussed in section 6.

The generated results of WM* seem to be better than those of other algorithms that solve the MAPFW problem. As you can see in Figure 4 and Figure 5 and it is always close to the minimal cost and it is also able to solve more benchmarks then the optimal solvers. This is especially true for the inflated heuristic which is still within 10% of the optimal cost and on average below 1% as seen in Table 1. Additionally it is able to find solutions much faster because its a heuristic approach. This is further reinforced by the progressive benchmarks where inflated WM* is able to solve problems with double the agents of WM* and with more agents than the other recently developed algorithms.

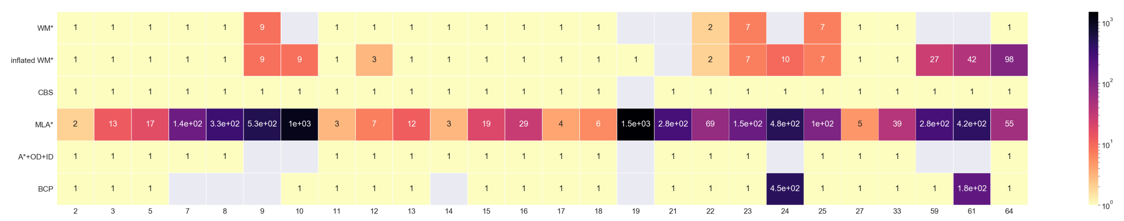

Figure 4: Heatmap comparing the costs for every algorithm's cost for the benchmarks on mapfw.nl. The algorithms are on the y-axis and the number of the benchmark on the x-axis. The colors are on a logarithmic scale so 1 is the minimum value since it can not show 0. Benchmarks where the cost was the same for every algorithm have been left out.

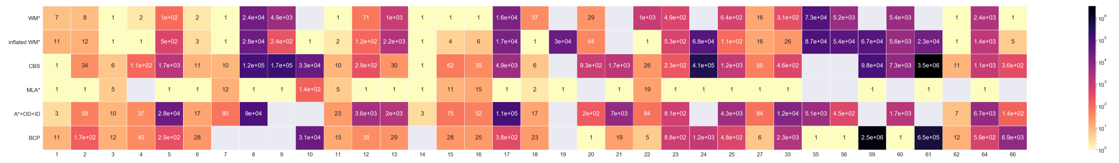

Figure 5: Heatmap comparing the times for every algorithm's time for the benchmarks on mapfw.nl. The algorithms are on the y-axis and the number of the benchmark on the x-axis. The colors are on a logarithmic scale so 1 is the minimum value since it can not show 0.

5 Responsible Research

This research has little to do with ethics but it is really important to be able to reproduce the results of the experiments. Therefore it is great that mapfw.nl hosts all the benchmarks and tests that are run by all the versions of WM*. Therefore it would be easy to confirm the results we have presented in this paper by running the benchmarks with the source code [13]. Once again we want to remark that the individual benchmarks results have been generated by the C++ implementation on my own PC while those of the progressive benchmarks are generated by the python implementation on a TU Delft server. The impact of the server is not really severe since when I ran the progressive benchmarks with the python implementation on my own PC I got similar results to the ones presented in this paper. The impact of the programming language is definitely notable though since C++ is on average twice as fast as python for the individual benchmarks. Furthermore it should be clear on how to write your own implementation of WM* given the pseudocode and explanation presented. This in turn should then give similar results on the benchmarks to the one we have given in this paper.

6 Conclusions and Future Work

In this paper we presented a way to extend the M* algorithm to solve the multi-agent path finding with waypoints problem (MAPFW). We consider both the ordered and unordered version of the problem. For the unordered problem we ordered the waypoints based on an optimal TSP solver. Then we used the order of waypoints to generate a policy per waypoint and for the target. Lastly we extended the underlying A* planner to be able to keep track of the waypoints that have been visited and still need to be visited. This resulted in an algorithm called WM* that is able to solve the ordered MAPFW optimally and the unordered variant close to optimality. The performance of the algorithm depends heavily on how many agents will collide when following their optimal policy since in the worst case every agent collides and it turns into a regular A* algorithm which is exponential in the number of agents. Therefore we also introduced the inflated heuristic algorithm which solves problems way more efficiently with only a maximum of 10% more cost compared to the base algorithm. Given a square grid where 20% of the locations is a wall and a fixed number of waypoints per agent inflated WM* is able to solve problems with more agents than other algorithms that solve MAPFW and twice as many agents in as the base algorithm.

In the future we want to implement more variants of the M* algorithm into the extended version. The major drawback of the current algorithm is the amount of neighbours added in case of many collisions which is exponential in the number of agents that are in collision. This also leads to an extremely large open set with many configurations having the same cost + heuristic. To solve the prior the original paper uses an Operator Decomposition (OD) to create ODM*, and to solve the latter Enhanced Partial Expansion A* is used to create EPEM*. For the Operator Decomposition it is also wise to look at Stef Siekman implementation for A* extended to the MAPFW [11] since he implemented an Operator Decomposition. Both ODM* and EPEM* can also be combined with inflated heuristics to get that same speed up at the cost of a maximum 10% of the original cost. They can also be combined with recursive M* (rM*) where instead of a collision set every configuration has a list of disjoint sets and uses recursion to find the optimal path for the agents in the disjoint sets.

7 Acknowledgements

I would first of all like to thank my supervisors Jesse Mulderij and Mathijs de Weerdt for their guidance and advice but also for giving me the freedom to do what I wanted to do. I would also like to thank my peers who developed their algorithm at the same time and who helped me along the way. Additionally, I want to thank my parents and friends for supporting me while having little to no clue what I am actually doing. Lastly, I want to thank my friends who do know their way with a computer for making fun of my code and helping me when I was stuck.

References

[1] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1 – 24, 2015.

[2] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Barták. Multi-agent pathfinding: Definitions, variants, and benchmarks. *CoRR*, abs/1906.08291, 2019.

[3] Jesse Mulderij, Bob Huisman, Denise Tönissen, Koos van der Linden, and Mathijs de Weerdt. Train unit shunting and servicing: a real-life application of multi-agent path finding, 2020.

[4] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40 – 66, 2015.

[5] Edward Lam, Pierre Le Bodic, Daniel D. Harabor, and Peter J. Stuckey. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1289–1296. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[6] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3260–3267, 2011.

[7] Bernhard Nebel. On the computational complexity of multi-agent pathfinding on directed graphs, 2019.

[8] Florian Grenouilleau, Willem-Jan van Hoeve, and John N. Hooker. A multi-label a* algorithm for multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling*, volume 29, pages 181–185, 2019.

[9] Arjen Ferwerda. Extending the Multi-Label A* Algorithm for Multi-Agent Pathfinding with Multiple Waypoints. *TU Delft Repository*, June 2020.

[10] Andor Michels. Multi-agent pathfinding with waypoints using Branch-Price-and-Cut. *TU Delft Repository*, June 2020.

[11] Stef Siekman. Extending A* to solve multi-agent pathfinding problems with waypoints. *TU Delft Repository*, June 2020.

[12] Noah Jadoenathmisier. Extending CBS to efficiently solve MAPFW. *TU Delft Repository*, June 2020.

[13] Jeroen van Dijk. WM* source code. https://github.com/jeroen11dijk/WMstar, 2020.

[14] Timon Bestebreur. Analysis of the influence of graph characteristics on MAPFW algorithm performance. *TU Delft Repository*, June 2020.