Extending A* to solve multi-agent pathfinding problems with waypoints

S.A.P. Siekman[1]
Supervisors: M.M. de Weerdt[1] , **J. Mulderij**[1]
[1]Delft University of Technology
sapsiekman@student.tudelft.nl,
{M.M.deWeerdt, J.Mulderij}@tudelft.nl

Abstract

In the field of cooperative multi-agent pathfinding (MAPF) the optimal set of non-conflicting paths must be found for a set of agents in a graph. The addition of waypoints to this problem (MAPFW) gives rise to the possibility of more complex applications, such as in vehicle routing, aviation, computer games or robotics. Yet no algorithms have been proposed that can provide optimal solutions efficiently in practice. Starting with an established algorithm for multi-agent pathfinding, A* with operator decomposition and independence detection, this paper provides an extension to find optimal results when waypoints are present that the agents need to visit. This is achieved by altering the heuristic to make use of a solver of an adapted version of the traveling salesperson problem, to calculate the shortest path to visit all the waypoints for each agent individually. It is then empirically shown that this algorithm can solve problem instances of decent sizes, and it is compared to another algorithms that were developed at the same time based by others based on different pre-existing MAPF solvers. While the extension of A* is less performant than alternatives, the difference in performance can be small depending on the type of problem. Together with the relative ease of implementation, this means that A* can be an effective algorithm to optimally solve multi-agent pathfinding problems with waypoints. Additionally, the extension techniques used to adapt A* to MAPFW can also be useful to other algorithms that utilise A* in the context of waypoints.

Introduction

The problem of pathfinding, in the sense of finding a path that avoids obstacles to reach a goal, is a typical AI problem, which already had efficient algorithms proposed over 50 years ago, like A* (Hart, Nilsson, & Raphael, 1968). However, there are many variants of pathfinding problems that have been proposed since. One such problem is multi-agent pathfinding (MAPF). In this variation, multiple agents share the same graph and must reach their destinations without conflicts, i.e. 'bumping into each other'.

Even though MAPF is an NP-hard problem (Nebel, 2019), there exists plenty of research proposing optimal solvers that run in reasonable time (Grenouilleau, van Hoeve, & Hooker, 2019; Lam, Le Bodic, Harabor, & Stuckey, 2019; Sharon, Stern, Felner, & Sturtevant, 2012; Wagner & Choset, 2011; Sharon, Stern, Goldenberg, & Felner, 2013). One of these algorithms is the extension of the aforementioned A* with operator decomposition and independence detection (Standley, 2010). However, for this problem extended with waypoints (MAPFW) there do not yet exist any fast optimal solvers. With this variation of the problem, agents each have to visited a set of waypoints before reaching their goal. Since this is an extension of MAPF, this problem is also NP-hard and finding optimal solutions is even more complex.

Solving MAPFW problems optimally would be useful in many different applications. Many of these applications are similar to those of regular MAPF, but the addition of waypoints always for more complex problems to be solved. These include the fields of computer games, robots, aviation, vehicle routing (Standley & Korf, 2011; Felner et al., 2017) and train maintenance and scheduling (Mulderij, Huisman, Tönissen, van der Linden, & de Weerdt, 2020). The addition of waypoints to these problems allows determining optimal routings in cases where the order of points is not important. In the case of vehicle routing and scheduling this might mean visiting different stops for maintenance in no particular order on their way to a goal. With robotics this could mean planning the movement of multiple robotic arms in a manufacturing process to perform tasks at multiple different locations.

We will propose and extension for waypoints to A* with operator decomposition and independence detection and determine whether this is an effective algorithm for solving MAPFW problems. To determine this, the extension will be empirically compared to different both optimal and heuristic algorithms for MAPFW based on different pre-existing MAPF solvers that were developed by others simultaneously.

This paper first gives a formal problem definition of multi-agent pathfinding problems with waypoints. Then the extension will be described in detail, after which its performance is evaluated first on its own and then compared to alternative algorithms. This data is then used to draw conclusions on the effectiveness of the extension of A* for MAPFW and whether

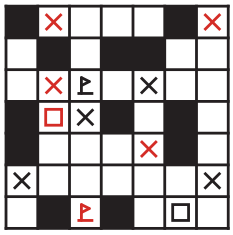

Figure 1: Example problem instance instance.

the findings are useful for other MAPFW implementations.

Problem formulation

The formulation of the multi-agent pathfinding with waypoints problem, closely resembles that of the variant without waypoints. (Standley, 2010) The MAPF problem is defined by an undirected graph $G = (V, E)$ and a set of n agents labeled $a_1, a_2, \ldots a_n$, where each agent a_i has a start and goal vertex $s_i, g_i \in V$. Time is represented in discrete steps, for each of which each agent moves from a vertex $v_{from} \in V$ to a vertex $v_{to} \in V$, given that $(v_{from}, v_{to}) \in E$, where it is also possible that $v_{from} = v_{to}$, corresponding with an agent waiting at their vertex. The path P_i of an agent a_i is represented as a list of vertices $v_{i,0}, v_{i,1}, \ldots v_{i,T}$ for each time step until some time T, where $v_{i,0} = s_i$ and $v_{i,T} = e_i$. The solution is the set of paths for each agents, which is valid if no conflicts occur. A conflict exists between some agent a_i and a_j in two cases: a *vertex conflict* or an *edge conflict*. A vertex conflict occurs when at some time t, $v_{i,t} = v_{j,t}$. If the movement during this time step of the agents is represented as $(v_{i,t,from}, v_{i,t,to})$ and $(v_{j,t,from}, v_{j,t,to})$ an edge conflict occurs when $v_{i,t,to} = v_{j,t,from}$ and $v_{i,t,from} = v_{j,t,to}$. The goal of MAPF is to find the optimal solution, i.e. one with the minimal *sum-of-costs*, which is the sum of the length of each agent's path.

For MAPFW, waypoints are added to this problem as a set $W_i \subseteq V$ for each agent a_i. The solution is then constrained by requiring $\forall w \in W_i$ that $w \in P_i$ for each agent a_i. This means that each agent must have visited all their respective waypoints at least once.

In order to compare the solver proposed in this paper with others, a 4-connected grid for G is used. Using a grid for the graph allows easy comparison with existing literature, where grids are used extensively for pathfinding problems. While grids closely relate to real-world pathfinding problems, as one can be imposed on a map to plan movement (Yap, 2002), it is still an abstract representation that allows the focus to be on the algorithm itself rather than a specific application. In this grid, some vertices and their respective edges may be removed, allowing the restriction of the movements of agents. to backupn example of a problem instance can be seen in Figure 1, where start vertices are denoted by squares, goal vertices by flags and waypoints by crosses.

Base algorithm

To solve the MAPFW problem, a pre-existing MAPF algorithm will be extended, namely A* with operator decomposition and independence detection (Standley, 2010). A* works by selectively choosing which *nodes* to extend while searching in such a way that it is guaranteed that a node is reached with the lowest cost. In the case of MAPF these nodes represent the positions of all the agents. Each node can then be extended to all possible nodes that are new valid positions of all the agents. With n agents and k legal moves per agent, this results in the creation of k^n new nodes upon expansion.

Each node has a certain cost ($g()$), which in the case of MAPF is the sum of the length of the paths of all agents up to that point. A* guarantees that the cost a node is optimal once it is visited by this algorithm. To ensure this, the expansion of nodes is prioritised based on the lowest $f()$-value, which is defined as $f() = g() + h()$ where $h()$ is a heuristic for the cost still required to reach the goal state from that node. Optimality is achieved when this heuristic is admissible, meaning that it never over-estimates the cost.

Operator decomposition

The aim of operator decomposition is to decrease the size of the search space. In the case of the 4-connected grid that is used in this paper, there are up to 5 possible moves for each agent at a given time step. This means that given n agents for every node that is expended over the run time of the algorithm 5^n new nodes are created and enqueued. With larger numbers of agents this exponential increase results in a great number of nodes being expanded that might not even work towards the goal state, i.e. have a lower heuristic.

Operator decomposition negates this problem by treating each agent individually. When a node is expanded, all the moves of only one agent are considered, meaning the search tree is extended by only 5 nodes at a time. Then the regular A* approach is applied to find the next node to expand, i.e. the one with the lowest $f()$ value. This means that for a given node, some agents might be in a state where they have not yet been assigned a move for that time step. To ensure that all moves are considered, it is also possible for agents to move into the vertex currently occupied by an agent which can still be moved during that time step, as long as that other agent is then in fact moved when its node is expanded.

This can best be illustrated with an example as shown in Figure 2. Here the expansion from a node A can be seen. Since agent 2 has not yet moved, all 5 options for expansion are still possible by agent 1, resulting in 5 new nodes. However, in the case of node D, where agent 1 has moved to the vertex of agent 2, this means that agent 2 now only has 3 new possible nodes, since a wait is not possible as agent 1 can no longer move this time step and a move *up* would result in an edge crossing conflict.

Independence detection

The power of independence detection lies in the fact that often the multi-agent problem can be seen as a single-agent, or fewer-agents, problem without conflicts. This means that the pathfinding problem is first solved for each agent individually. Only when conflicts arise between them are agents grouped together, and is their solution computed cooperatively as described before with A* and operator decomposition. When conflicts arise between groups of agents, the same principle

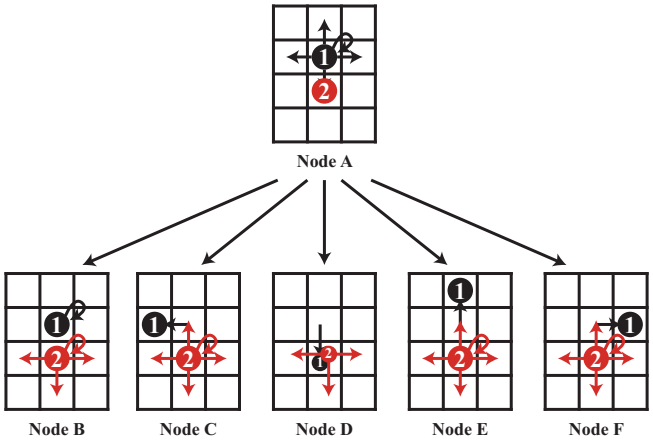

Figure 2: Example of node expansion using operator decomposition.

is applied, and groups are merged together and then solved cooperatively.

Before (groups of) agents are merged, first an attempt is made to find an alternative solution that does not conflict. Additionally, to avoid conflicts so-called *conflict avoidance tables* are used, which keep track of how many conflicts a group of agents has with other groups. This table is a collection of the current path of all the agents, meaning also the agents that are in different groups. When expanding nodes in the A* search, conflicts within the same group are not allowed, however conflicts with the paths of agents in other groups might occur. The number of conflicts that occur with these agents from other groups are counted for each node. The nodes with lower conflict counts are then expanded first to avoid having to merge different groups later on in the independence detection algorithm.

This results in Algorithm 1 for independence detection. Note that this is not a solver on its own, but it simply calls the aforementioned A* solver with operator decomposition to solve sub problems.

Extension with waypoints

To adapt the base algorithm to solve problems with waypoints it has to be extended in two major ways. Firstly, nodes in the A* search need additional information to keep track of which waypoints have been visited. Second, the heuristic used to direct the search needs to be changed. This heuristic is used to estimate the cost required for a single agent to reach its goal vertex (v_g) from the location it is at for the node for which the heuristic is calculated (v_s). The sum of these heuristic values is the heuristic for a node. There are many ways in which this heuristic can be implemented. As long as the cost is never overestimated, the heuristic is *admissible* and will thereby ensure an optimal solution to the problem. This section will explore two approaches to calculate such a heuristic.

For both of these approaches the distances between vertices are required. At the start of the algorithm distance maps are created from all waypoints and the goal vertex to all other points on the grid using breath-first search. While the Manhattan distance could be used as well, this was not done to ensure the algorithm would be easier to adapt to graphs other than grids.

Algorithm 1 Independence detection (Standley, 2010)

```
assign each agent to a group with just that agent
cooperatively plan a path for each group
fill conflict avoidance table with every path
while conflicts occur do
    simulate path until a conflict between two groups G1 and G2 is found
    if groups have not conflicted before then
        fill illegal moves table with path of G2
        find another path with same cost for G1
        if failed to find alternative then
            fill illegal moves table with path of G1
            find another path with same cost for G2
        end if
    end if
    if failed to find alternative path then
        merge G1 and G2 into a single group
        cooperatively plan new group
    end if
    update conflict avoidance table with new paths
end while
return paths of all groups combined
```

Minimum spanning tree

The heuristic needs to estimate the cost of a single agent traveling from v_s to v_g via the vertices of all unvisited waypoints, which we will call v_{w_i} for all $w_i \in W$ if w_i has not yet been visited at the current node. The vertices $\{v_s, v_g, v_{w_0}, v_{w_1}, \ldots, v_{w_n}\}$ can be used as the vertices in a new complete graph G_H where the cost of these edges is equal to the distance between the vertices in G (the graph of the main MAPFW problem). Since the heuristic only needs to guarantee that the cost it finds is not an overestimate it is possible to take the cost of the minimum spanning tree (MST) of G_H. By definition of a minimum spanning tree, there is no way to connect all the vertices in the graph with a lower cost, so this heuristic can never be an overestimate. This estimation does not take any other agents into account, as these conflicts can never decrease the actual cost.

Travelling salesperson problem

Another possible heuristic is to ensure that the heuristic is correct with regards to a single agent. This means that for any vertex its heuristic is the shortest distance via all unvisited waypoints to the goal vertex. The result of this heuristic is that the A* expansion is only used to resolve conflicts with other agents, as the algorithm can simply follow the heuristic when no conflicts occur.

With an increasing number of waypoints, calculating this heuristic becomes increasingly difficult. The shortest path is obtained by running an adapted version of a dynamic programming solver for the traveling salesperson problem (TSP), which is known to be an NP-hard problem (Laporte, 1992). Since this algorithm practically works up to around 12 to 15

nodes, no better can be expected from the extension from MAPFW, since the heuristic is calculated frequently.

Since computing the optimal path via all waypoints is so computationally expensive, it is worthwhile to store the result as efficiently as possible. When an agent is moving to a waypoint, the heuristic needs to be calculated repeatedly, while the set of visited waypoints does not change. To take advantage of this, the heuristic is not directly calculated from v_s, but rather from all v_{w_i} to v_g visiting all other v_{w_j} as long as $i \neq j$. The result of this calculation does not change as long at the agent does not visit any waypoints, allowing the values to be re-used. To attain the heuristic from v_s, we can utilise the results from this calculation along with the remaining distances to all waypoints. By simply adding the values calculated for each waypoints with the distance to this waypoint and taking the minimum of this combined value for all unvisited waypoints, we get the heuristic for v_s. Algorithm 2 describes how the heuristic is implemented.

Algorithm 2 TSP based heuristic calculation

1: $d(v_a, v_b) \leftarrow$ distance map from v_a to v_b
2: $W \leftarrow$ set of waypoints that have not yet been visited
3: **if** $W = \emptyset$ **then**
4: **return** $d(v_s, v_g)$
5: **end if**
6: $path(w_i) \leftarrow$ calculate the path lengths of each $w_i \in W$ to v_g via all waypoints in $W - \{w\}$
7: **return** $min(dist(v_s, v_{w,i}) + path(w_i))$ for all $w_i \in W$

The dynamic programming approach has an additional benefit with Algorithm 2. As it calculates the shortest path to all waypoints in one go, it is possible to find the distance to the goal from each starting waypoint's vertex at once. This means that only one call is required to the $path(w_i)$ function for every time Algorithm 2 is performed.

The most computationally expensive part of this calculation, finding the shortest way through all the remaining waypoints, can be cached effectively. The calculation of the $path()$ function, as shown on line 6 of Algorithm 2, only depends on the waypoints that still need to be visited by an agent, we call this set of waypoints W. By keeping a cache hash-map, of which the key is W, the heuristic only needs to be recalculated once a new waypoint is visited by an agent. Because the heuristic only considers a single agent, it is possible to retain the caching between different runs of the independence detection algorithm.

Since dynamic programming is used to solve this subproblem, it would also be possible to store all possible values of W when the calculation is made for the case where W is all the waypoints. However, since the order in which the waypoints will be visited is not known ahead of time, this would mean that $2^{|W|}$ entries would need to be stored in this precomputed table, which would need to be searched for every new heuristic calculation.

In applications where agents share a large subset of waypoints, it could be worthwhile to implement caching across different agents. However, if these agents do not share the same goal vertices, care would have to be taken to account for this. One possible approach would be to calculate the shortest distance from all unvisited waypoints, to all unvisited waypoints via all other unvisited waypoints. This would result in a large matrix for each set of unvisited waypoints. Experiments could determine whether this leads to increased performance. Since this research does not focus on cases with shared waypoints, the additional caching is not implemented.

Node expansion prioritisation

In the paper describing the extension of A* with operator decomposition and independence detection (Standley, 2010) it is mentioned that it is preferable to prioritize the expansion of nodes with a lower heuristic value before prioritizing based on the number of conflicts based on the conflict avoidance table. (This is of course *after* prioritizing based on the $f()$ value used by A*, to ensure optimality of the solution.) Testing revealed that this is not the case for the extension to MAPFW problems, so this is further empirically evaluated in the Experiments section. If nodes are first prioritized based on the conflicts, and then based on the heuristic, a larger fraction of more complex problem could be solved.

Experiments

Experiments were performed on an Intel Xeon Gold 6248 CPU @ 2.50GHz using a Python 3 implementation of the extended algorithm as described before. The problems were generated randomly similarly to the generation previously used in MAPF literature (Silver, 2005). 20% of the vertices on 32x32 grid were randomly marked as walls, independent regions were also filled with walls. The agents start, stop, and waypoint vertices were randomly chosen from the remaining non-wall vertices. This choice was made to more easily compare results to those in previous literature.

These experiments where performed to give an indication of the performance of the implementation as well as to test two implementation choices. First, the performance difference of using a TSP-based heuristic over the MST alternative is shown. Second, the impact of prioritising avoiding conflicts over a lower heuristic during the A* expansion is evaluated. Third, an empirical evaluation is performed using an implementation with the best combination of these two choices to show the ability of the algorithm to solve MAPFW problems.

Heuristic

Implementations of the extension using MST and TSP were run on random grids as described before. Problem instances were generated with an increasing number of agents per instance for 5, 10 and 13 waypoints per instance. Per difficulty, i.e. the number of agents, 50 such random instances were generated. The algorithms were then run to see how many of the problems could be solved within 20 seconds and how fast this was achieved. These numbers were chosen because experimentation revealed that these numbers gave meaningful results for most algorithms while still having a reasonable total run time.

The results of running these two version for 5 waypoints can be seen in Figure 3. For more waypoints per problem

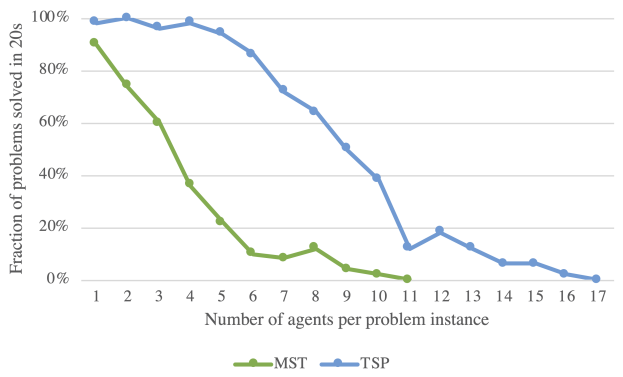

Figure 3: Fraction of random problem instances with 5 waypoints that could be solved using a heuristic based on TSP or MST.

instance the ability to solve problems decreased even further, with the MST-based alternative being able to solve 20% of the instances at just 2 agents, while this point was at 10 agents for the TSP-based version. Additionally, of the problems that were solved within the timeout by both alternatives, the TSP-based alternative was able to solve them on average 20x faster.

Node prioritisation

The same benchmarks that were performed for comparing the two alternative means to calculate the heuristic, were performed for the different orders of prioritising node expansion. The prioritising a low conflict could could consistently solve problem instances with any number of agents and waypoints, however always by a relatively small margin. On average for these tests, prioritising lower conflict counts first resulted in about 20% faster run times. No correlation was found between the number of agents and waypoints in a benchmark and the performance difference between these approaches.

Performance evaluation

To analyse the performance of the extension based on the aforementioned TSP heuristic and prioritising conflict avoidance, the implementation was run on increasingly difficult problems on a 32x32 grid, where difficulty is interpreted as the number of agents and the number of waypoints each of these agents has. For each of the problem difficulties, the implementation was run on 100 different instances, and it was determined whether it was able to solve this problem within 100 seconds. The results of this can be seen in Figure 4.

There are two interesting observations that can be made from these results. Firstly, there is a point at around 12 waypoints per agent where the runtime of solving the TSP optimally becomes the predominant bottleneck. Since the subroutine for TSP runs in exponential time, solving it becomes impossible in the given 100 seconds. Given more time it becomes clear that solving TSP becomes the main bottleneck. With the results in Figure 4, the same benchmarks were performed with a greater limit for the run time, but only with 5 agents, as this is a point were there seems to be a clear limit caused by the number of waypoints. As can be seen in

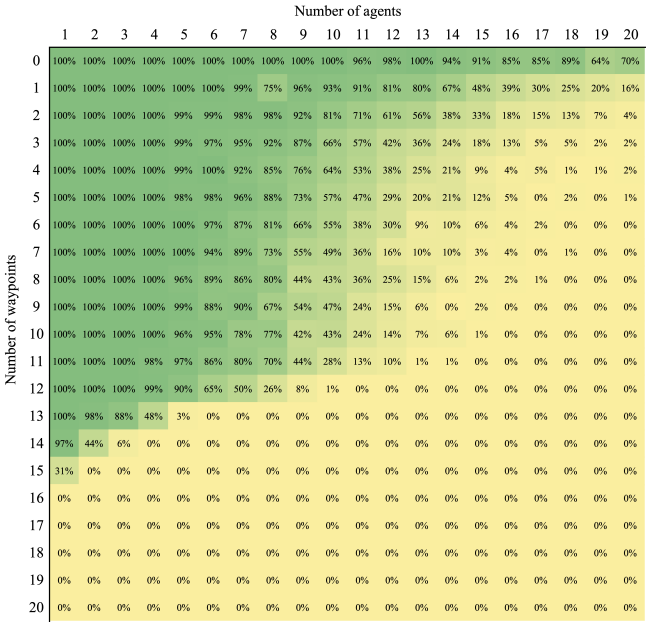

Figure 4: Fraction of 100 random 32x32 problems per difficulty that could be solved using a TSP heuristic within 100 seconds.

Figure 5 adding more run time allows problems to be solved with more waypoints, however there is a sharp increase in the required time when the number of waypoints increases, indicative the exponential nature of solving TSP, which is exponential only in terms of the number of waypoints.

Secondly, adding any waypoints at all makes the problem much harder, even for small numbers of waypoints. This can be seen clearly in Figure 6, which looks just at the solve times of 4 waypoints and less. This can be partially explained by the fact that the addition of waypoints requires more calculation for the waypoint, since with one waypoint just the distance to the goal can be used. An additional explanation is that adding waypoints randomly will cause the agents movement to be spread out much more than just the path to their goal, thus increasing the number of possible conflicts that need to

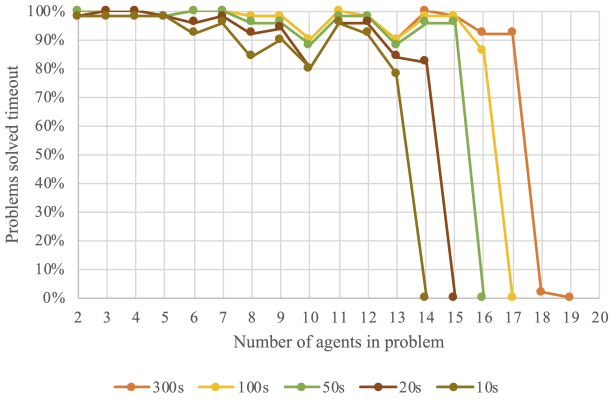

Figure 5: Fraction of 50 random 32x32 problems with 5 agents per instance that could be solved within different timeouts.

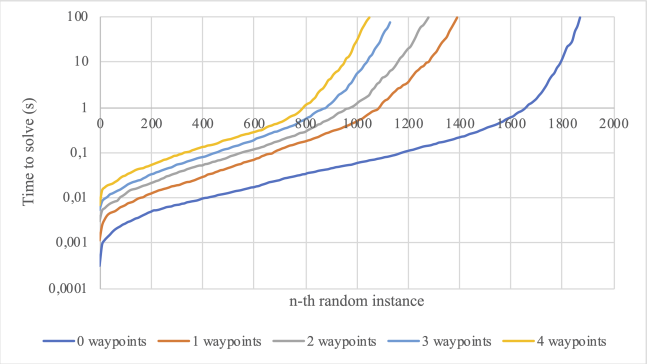

Figure 6: Solve time for random 32x32 problems with uniformly distributed number of agents between 1 and 20 using a TSP heuristic.

be resolved.

Other solvers

The implementation of the extension was compared to four other extension for MAPFW that were based on pre-existing MAPF solvers: CBSW (Jadoenathmisier, 2020), MLA* (Ferwerda, 2020), BCP (Michels, 2020), and M* (Dijk, 2020). It should be noted that the implementations for these extensions vary in the languages that were used, which should be kept in mind when empirically comparing the runtimes of these different implementations. While the extensions of CBSW, M* and MLA* were implemented in Python 3, the extension of BCP was written entirely in C++. However, to reduce further differences, all benchmarks were performed on the same system as mentioned in the previous section.

All the alternative extensions, as well as the one described in this paper, were run on randomly generated problem instances of 32x32 grids and with a different number of waypoints per agent. For a given number of waypoints per agent the algorithms were run on the same random instances, with an increasing number of agents in the problem. For each of these numbers of agents, 50 random problems were generated, and it was recorded how fast and how many of these problems could be solved within 20 seconds as well as the cost of the solution found, since not all extensions guarantee optimal solutions. These numbers were experimentally chosen to ensure that the difference between algorithms would be clearly visible, while not being biased. The fraction of these random problems that could be solved can be seen in Figure 7. To compare the run times of the algorithms, Figure 8 shows the time taken to solve each randomly generated problem instance up to those with 11 agents.

Since CBSW is the only other optimal algorithms that is also implemented in Python, it is the only algorithm that this extension can directly be compared with. While the extension of A* described in this paper performs worse than CBSW, this performance difference decreases with more difficult problems (i.e. the ones with more waypoints per agent). The same observation holds for the run time of the algorithm: the extension based on A* is slower than the alternative, but this difference decreases as the number of waypoints per agent increases. It should be noted that the version of CBSW that was used for the comparison has not been proven to be optimal, although this is assumed.

Since MLA* and M* do not guarantee optimal solutions, it is important to evaluate the extra cost of the solutions resulting from these algorithms compared to the optimal solution when this could be determined by the extension of A*. This extra cost for the performed benchmarks of MLA* is shown in Figure 9, where it can be seen that the approximation error of the MLA* extension is between 10% and 25% and increases as the number of waypoints per agent increases, and slightly increases as the number of agents in a problem increases. For M* this approximation error was less than 0.5%, and did not vary depending on the number of agents or waypoints given the sample size. While the waypoint extension of M* and one described in this paper are quite similar, they differ in their approach with regards to the ordering of waypoints: M* pre-computes the optimal routing via the waypoints only once at the start of the algorithm, while the extension of A* does this continuously.

Responsible research

Numerical evaluations of the performance of an algorithm is inherently difficult to compare and precisely reproduce. The performance results are dependent on the implementation that was used, so potentially better results can be achieved with a better implementation. To aid reproduction, the exact implementations used in the paper are provided, as well as the algorithms and seeds used to generate the random problem instances. Where possible the exact same computer system and environment was used to benchmark the different algorithms, meaning that the comparison between them depended as little as possible on such external factors.

To compare these different algorithms is was important to make sure that all the implementation could interface with the same problems instances and that results could be easily compared. To aid this, a website and corresponding API was developed together with N.J.M. Jadoenathmisier. Using this, identical problem instances could be generated for all implementations and equality of the benchmarks could be ensured to the greatest extent possible. Additionally, during development it was beneficial to be able to have visualisations and identify where implementations might work incorrectly.

It should be noted that the speed of an algorithm in numerical evaluations depends greatly on its implementation. While some of the different algorithms were written in the same language, none were developed side-by-side, meaning that one might be more efficiently implemented than the other. Additionally, the use of a different programming language might increase or decrease the performance difference between the implementations.

Conclusion

This paper describes the problem of multi-agent pathfinding with waypoints. An algorithm was proposed to extend an existing solver for standard multi-agent pathfinding to the variant with waypoint: A* with operator decomposition and independence detection. By using the algorithm for an adapted version of the traveling salesperson problem, the heuristic

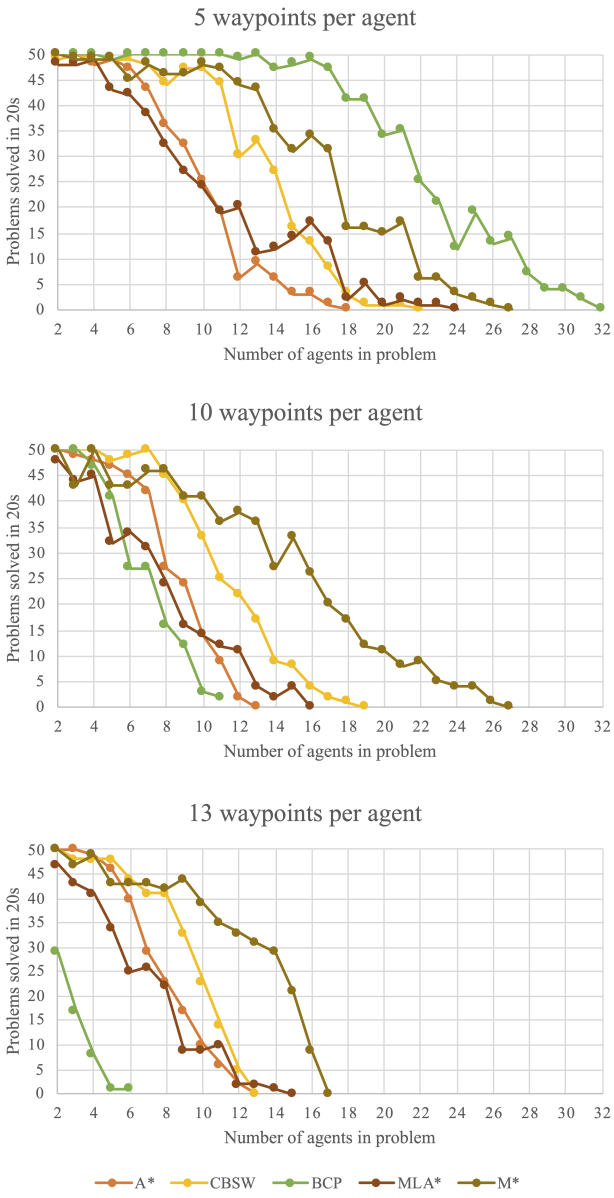

Figure 7: Problems out of 50 randomly generated per number of agents that could be solved by alternative MAPFW algorithms.

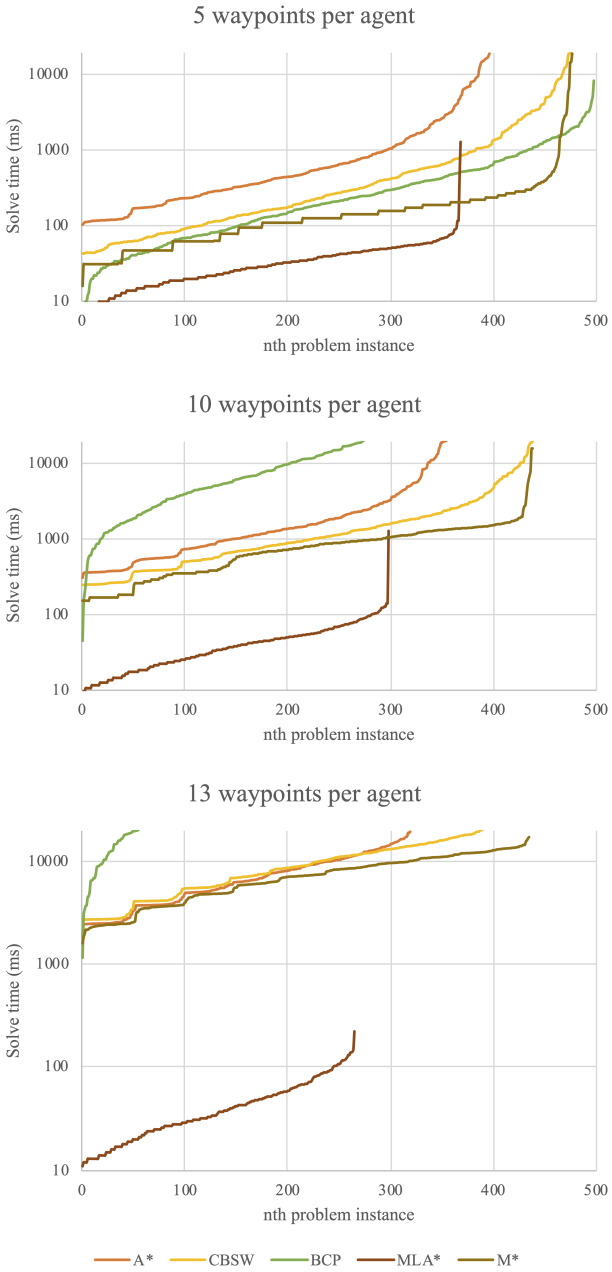

Figure 8: Time taken to solve random 32x32 problems of between 2 and 11 agents per problem.

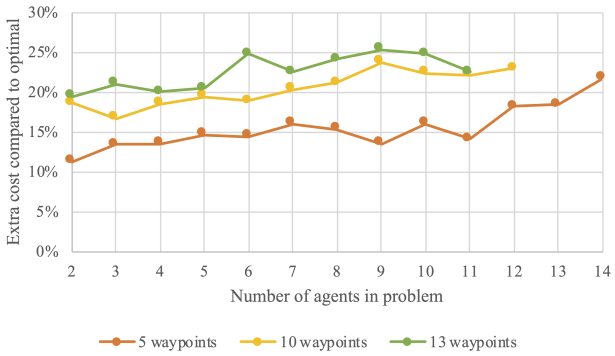

Figure 9: Approximation error of the MLA* extension on 32x32 grids.

of the standard algorithm was extended to efficiently solve the problem with waypoints. It was shown that using the TSP-based heuristic performs better than the MST alternative, even though it is more computationally expensive per heuristic calculation. Prioritising avoiding conflicts over a lower heuristic during the A* expansion was beneficial to the run time of the algorithm. Using these insights, the algorithm can effectively solve problem instances with roughly up to a dozen agents and waypoint in grid-like instances sized 32x32.

Compared to an algorithm based on Conflict Based Search, named CBSW, it was shown that the extension of A* was less performant, yet not by a great margin, since the performance is to a large extent limited by the complexity of the TSP. The extension of BCP seems to be the best performing algorithm for problem instances with fewer waypoints, but its ability to solve problems drastically decreases as more waypoints are added per agent. MLA* has the advantage of much faster run times compared to the alternatives, however the solutions are far from optimal. While the run time does not increase by much for more difficult problems, the ability to solve these problems does, meaning that the algorithm cannot always be used. If a solution needs to be close to optimal, and a small approximation error is not an issue, the extension of M* seems to be the best alternative. Since other extensions already perform better than the extension based on A* proposed in this paper, it would be better to pursue improvements of those algorithms rather than the one proposed in this paper, although techniques described can still be useful in these algorithms where they use A*.

Further work can be done to combine the methods described in this paper with the alternative algorithms for MAPFW, where ever this has not yet been done. To create a true comparison, it would be preferable to create implementation as similar as possible of all the extensions, for instance with regard to the programming language used. With all algorithms implemented in C++, for instance, it could be expected that the performance of the Python implementations would increase. While this paper focuses on grid-like graphs, the comparison might be different on graphs that are not grids, which could be evaluated as well. Even though this problem representation is easy to work with, it might not be a suitable reduction for some constraints present in real-world applications where this algorithm might be useful, for instance train maintenance scheduling. Evaluating the ability to adapt this algorithm for such problem variations, has not yet been attempted.

References

Dijk, J. v. (2020). Solving the multi-agent path finding with waypoints problem using subdimensional expansion. *TU Delft Repository*.

Felner, A., Stern, R., Shimony, S. E., Boyarski, E., Goldenberg, M., Sharon, G., … Surynek, P. (2017). Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Tenth annual symposium on combinatorial search.*

Ferwerda, A. (2020). Extending the Multi-Label A* Algorithm for Multi-Agent Pathfinding with Multiple Waypoints. *TU Delft Repository*.

Grenouilleau, F., van Hoeve, W.-J., & Hooker, J. N. (2019). A Multi-Label A* Algorithm for Multi-Agent Pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling* (Vol. 29, pp. 181–185). (Issue: 1)

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107. (Publisher: IEEE)

Jadoenathmisier, N. J. M. (2020). Extending CBS to efficiently solve MAPFW. *TU Delft Repository*.

Lam, E., Le Bodic, P., Harabor, D., & Stuckey, P. J. (2019). Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), International Joint Conferences on Artificial Intelligence Organization* (pp. 1289–1296).

Laporte, G. (1992). The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, *59*(2), 231–247.

Michels, A. C. (2020). Multi-agent pathfinding with waypoints using Branch-Price-and-Cut. *TU Delft Repository*.

Mulderij, J., Huisman, B., Tönissen, D., van der Linden, K., & de Weerdt, M. (2020). *Train unit shunting and servicing: a real-life application of multi-agent path finding.*

Nebel, B. (2019). On the Computational Complexity of Multi-Agent Pathfinding on Directed Graphs. *arXiv preprint arXiv:1911.04871*.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2012). Meta-Agent Conflict-Based Search For Optimal Multi-Agent Path Finding. *SoCS*, *1*, 39–40.

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, *195*, 470–495. (Publisher: Elsevier)

Silver, D. (2005). Cooperative pathfinding. *AIIDE*, *1*, 117–122.

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *Twenty-Fourth AAAI Conference on Artificial Intelligence.*

Standley, T. S., & Korf, R. (2011). Complete algorithms for cooperative pathfinding problems. In *Twenty-second international joint conference on artificial intelligence.*

Wagner, G., & Choset, H. (2011). M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ international conference on intelligent robots and systems* (pp. 3260–3267). IEEE.

Yap, P. (2002). Grid-based path-finding. In *Conference of the canadian society for computational studies of intelligence* (pp. 44–55).