Matching in Multi-Agent Pathfinding using M*

Jonathan Dönszelmann*

TU Delft

Abstract

Multi-agent pathfinding (*MAPF*) is the process of finding collisison-free paths for multiple agents. *MAPF* can be extended by grouping agents into teams. In a team, agents need to be assigned (or matched) to one of the team's goals such that the *sum of individual costs* is minimised. This extension is called *MAPF* with matching (*MAPFM*). *M** is a complete and optimal algorithm to solve *MAPF* problems. In this paper, two strategies are proposed which allow *M** to solve *MAPFM* problems. These strategies are called *inmatching* and *prematching*. It is shown that *prematching* is generally preferable to *inmatching*, the benefits of different optimisations for *M** are compared, and it is shown that the performance of *M** performs very comparably to other *A**-derived algorithms.

1. Introduction

A large number of real-world situations require the planning of collision-free routes for multiple agents.[1] For example, the routing of trains over a rail network [1], directing robots in warehouses [2] or making sure autonomous cars do not collide on the road [3]. Problems of this nature are called *Multi-agent pathfinding* problems, which will hereafter be abbreviated to *MAPF*.

In *MAPF*, each agent has a starting position and a goal position. For every agent, a route needs to be found from their start to their goal, without two agents colliding. Finding these collision-free routes has been formally proven to be *NP-hard* [4].

One algorithm to solve *MAPF* is called *M** [5]. *M** is derived from the *A** algorithm [6] as described by Standley [7]. *A** generally plans the paths of agents together. This means that in each timestep, the number of possible next states grows exponentially with the number of agents. Contrasting that, in *M**, agents follow an individually optimal path whenever possible. In each timestep, only the subset of colliding agents is jointly planned and if necessary, backtracking is performed.

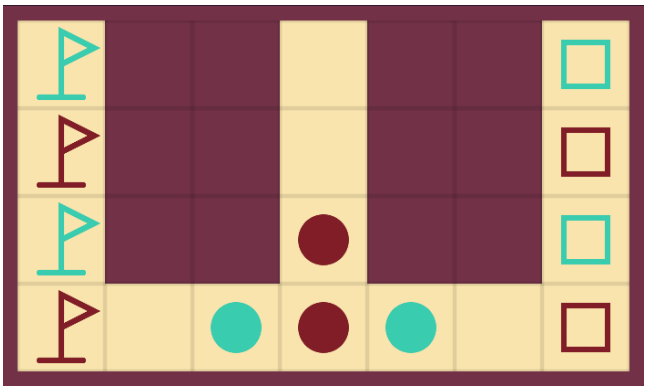


Figure 1: A snapshot of agents solving a *MAPFM* problem. Two teams of agents are moving from their starts to their goals. Squares, flags and circles are starts, goals and agents respectively. One red agent (at 4 squares from the left, 3 squares from the top) waits to let the three other agents pass.

When directing robots through warehouses, there may be different models of robots with different capabilities. Some robots may be able to restock shelves while others collect orders. The *MAPF* problem can be generalised to represent such a problem with teams of agents. In contrast to *MAPF*, an agent does not have a single goal in this generalisation. Instead, agents can use any goal associated with the team it is in, but no two agents can end at the same goal. An algorithm solving this problem must assign each agent to a goal. Such an assignment is called a matching. This problem is therefore named multi-agent pathfinding with matching (hereafter abbreviated to *MAPFM*).

In this paper, *MAPFM* is first defined, after which two

*Supervised by Jesse Mulderij (j.mulderij@tudelft.nl) and Mathijs de Weerdt (m.m.deweerdt@tudelft.nl)

[1] An agent is the collective name for a single unit for which a path is calculated. The word 'agent' could refer to a single robot, a train or a car, but can also refer to an abstract or simulated entity. In MAPF, paths are planned for multiple agents.

techniques are discussed which extend M^* to also solve *MAPFM* problems. These two techniques are then compared, to each other, and to a number of other algorithms which also solve *MAPFM* problems. Apart from this comparison, a number of extensions to M^* are discussed which improve M^*'s performance. Some of these extensions can also improve the performance of M^* on plain *MAPF* problems.

1.1. Prior work

Before this research, a separate study has been performed [8] on a problem called the target-assignment and pathfinding (*TAPF*) problem. The difference between *TAPF* and *MAPFM* is that in *TAPF*, the *makespan* (the cost of the longest path) instead of the *sum of individual costs* (the cost of all paths combined) is minimised. To solve *TAPF*, [8] uses conflict based search (*CBS*) [9], using a min-cost flow based algorithm for the lower level. The min-cost flow is used to solve *TAPF* for a single team in polynomial time. The higher level of *CBS* combines single team solutions to find a solution for all teams. According to [10], such a Min-cost flow based algorithm can also be used to solve *MAPFM* problems.

To the best of the author's knowledge, solving *TAPF* or *MAPFM* with other algorithms than *CBS* (such as M^*) has not yet been explored.

In contrast, a lot of research has been done finding and improving algorithms that solve *MAPF* [5, 7, 9, 11–14]. The current state-of-the-art *MAPF* algorithms are *Conflict Based Search* (*CBS*) and *Branch-and-Cut-and-Price* (*BCP*). However, which of the existing algorithms for *MAPF* can be adapted such that they also perform well at solving *MAPFM* problems is yet unclear.

2. Definitions of *MAPF* and *MAPFM*

The definition of multi-agent pathfinding used in this paper is based on Stern's definition given in [15]. The definition is given below.

A *MAPF* problem P consists of the following elements:

$$P = \langle G, s, g \rangle$$

- G is an undirected graph $\langle V, E \rangle$
  - V is a set of vertices
  - E is a set of unweighted edges between vertices
- s is a list of k vertices where every s_i is a starting position for an agent a_i
- g is a list of k vertices where every g_i is a goal position for an agent a_i

The goal of *MAPF* is to find a path for all agents a_i from s_i to g_i, without vertex and edge conflicts. This means that in a timestep t, two agents may not be on the same vertex, and between two timesteps, two agents may not travel over the same edge.

The cost c_i of a path π_i is the number of timesteps until the last time that agent a_i arrives at its goal g_i. [15] defines two primary approaches to define the cost c of a solution to a *MAPF* instance. The *sum of individual costs* is the sum of the cost of all paths ($c = \sum_{n=0}^{k} c_n$). Alternatively, the *makespan* is equal to the maximum cost of all paths ($c = \max_{n=0}^{k} c_n$). An optimal solution to a *MAPF* instance is a set of paths π with the smallest possible cost c.

Although the algorithms presented in this paper would work on any graph G, in examples and experiments, G is simplified to a 4-connected grid. In this paper, this grid is sometimes called a 'map', on which agents move from their start positions to their goal positions.

MAPFM

Now, the concept of matching is added to the definition of *MAPF*, to create multi-agent pathfinding with matching (*MAPFM*). The new *MAPFM* problem adds two elements to P, as shown below:

$$P' = \langle G, s, g, sc, gc \rangle$$

- sc is a list of colours. Each starting vertex s_i is assigned a colour sc_i.
- gc is a list of colours. Each starting vertex s_i is assigned a colour sc_i.

This definition divides all agents into teams. An agent a_i's team colour is the colour of its starting location sc_i. In total there are K teams $k_1 \dots k_n$. K is equal to the number of different colours in sc and gc.

In *MAPFM*, agents do not have a single goal. Instead, teams have several goals which all have the same colour gc_i. A team has as many goals as there are agents. A solution to *MAPFM* is a set of collision-free paths for each agent, in which every agent travels to a goal with a colour equal to the colour of that agent.

MAPFM uses the *sum of individual costs* as an optimisation criterion.

3. A description of *M**

M^* [5] is a complete and optimal pathfinding algorithm, similar to *A**. However, unlike *A**, M^* is specifically designed to solve *MAPF* problems. To use *A** for *MAPF*, the planning of paths for all agents needs to be coupled to make sure no states are expanded in which collisions occur. Coupled planning means that the states used in *A** contain the positions of all agents at the same time, and every expansion expands new states for all agents at the same time. The search space is said to be k-dimensional where k is the number of agents for which paths are planned.

Unlike in *A**, in M^* the planning of agents is initially not coupled. Instead, M^* plans agents according to an individually optimal path. An individually optimal path is the minimal cost path for an agent to their goal, not taking into account the location and movement of other agents. This may therefore lead to the collision of agents. However, as long as collisions do not occur,

this strategy separates the planning of agents, making the search space one-dimensional.

However, with many map layouts, collisions between agents following individually optimal paths are bound to occur. When M^* detects the creation of a state s in which two or more agents do collide, this state is discarded. Instead, parent states of s are reassessed to see if the colliding agents can plan differently, in a way which may not be individually optimal but avoids the collision in s.

During this reassessment, the planning of only the agents which collided in s is coupled like in A^*. The search space of M^* can therefore be though of as being one-dimensional, but with a dynamically growing dimensionality around collisions. Around collisions, the search space is n-dimensional where n is the number of colliding agents.

Just like A^*, M^* uses a priority queue and a heuristic to prioritise the exploration of states. This finds optimal solutions as long as an admissible heuristic is used such as the Manhattan- or Euclidean distance to goal location of each agent.

4. *M** AND MATCHING

To add matching to M^*, this paper proposes two options which are called "inmatching" and "prematching". In this section, both are explained and their advantages and disadvantages are discussed.

4.1. INMATCHING

Inmatching is the process of performing matching as a part of the pathfinding algorithm that is used. To understand it, it is useful to first look at inmatching in A^*. With A^*, the expansion of a state consists of all possible moves for all agents. A^* searches through the search space, until the goal state has been removed from the frontier. With an admissible heuristic, A^* guarantees that backtracking from this first goal state gives a shortest path between the start and goal location.

With *inmatching*, there is not one goal state. Instead, any state in which all agents stand on a goal of their own colour is considered a goal state. This means there is more than one goal state. An admissible heuristic for *inmatching* M^* is the Manhattan- or Euclidean distance to the closest goal location for each agent.

Inmatching applied to *M**

Inmatching can similarly be used with M^*. However, in M^* *inmatching* is likely to be quite inefficient (an experimental evaluation of this inefficiency can be found in Section 7.1). The reason for this inefficiency is that in M^*, agents try to follow an individually optimal path. In *MAPF* problems, an agent only has one individually optimal path because an agent only has one goal position. However, in *MAPFM*, agents commonly have multiple goal positions. With *inmatching*, the paths to all goal positions of an agent need to be considered to guarantee optimality. In a sense, an agent has multiple individually optimal paths. One towards each goal position of the agent's team.

In regular M^* the branching factor [2] is 1 when agents do not collide. However, because with *inmatching* there are multiple individually optimal paths, the branching factor of M^* may be larger than one even when there are no collisions.

The upper bound of this branching factor can be expressed as follows:

$$\prod_{i=1}^{k} \text{goals}(a_i)$$

Here, the *goals* function gives the number of goals in team of agent a_i. All agents need to consider moving towards all goals of the team they are in. A new state needs to be created for every combination of movements of agents, because a single state contains the location of all agents combined. Therefore the upper bound for the number of expanded states is a product of the movements for a single agent.

inmatching M^* will hereafter be abbreviated to *im*M^*

4.2. PREMATCHING

Algorithm 1: prematch M^*

Result: Find the matching m_{best} with smallest cost

$m_{best} \leftarrow \varnothing$;

$matchings \leftarrow$ find all matchings$(starts, goals)$;
foreach $m \in matchings$ **do**
  $s \leftarrow$ mstar$(starts, m)$ {Evaluate with M^*};
  $m_{best} \leftarrow min(m_{best}, s)$;
end

As an alternative to *inmatching*, there is *prematching*. With *prematching* the *MAPFM* problem is transformed into a number of *MAPF* problems. Each possible matching is calculated in advance, and normal M^* as described by [5] is performed on each matching exhaustively. Algorithm 1 shows this exhaustive search.

In Section 5.3 an improvement to *prematching* M^* is proposed which uses a heuristic to prune some of the matchings which otherwise needed to be evaluated.

prematching M^* will hereafter be abbreviated to *pm*M^*

5. EXTENSIONS TO *M**

M^* can be improved upon in various ways. Some of these extensions are only applicable to M^* with match-

[2]The branching factor is the number of child states which are created from one parent state. A branching factor of 1 means that each parent only creates one child state. In this text, a state refers to an M^* state as described in Section 3.

ing, while others improve *M** itself. In this section these extensions are described in detail.

5.1. Precomputed paths and heuristics

A part of solving *M** instances is finding the individually optimal path for each agent. This can be done with conventional pathfinding algorithms such as *A**. But to avoid repeated calculation, it is important to first create a lookup table of the minimal distance to each goal, for every open square on the map. This table can be created using a breadth-first search from every goal. With this table, finding in which way an agent should move to reach the goal in an individually optimal manner becomes a constant time operation.

When using *prematching*, the same goal locations and start locations are (although in a different permutation) reused multiple times. The lookup table can remain the same for all evaluations of matchings.

Using this lookup table, a better heuristic for *M** can be created. Instead of using the *euclidean-* or *Manhattan distance* to the goal, the lookup table can be used to find the exact distance, taking the obstacles on the map into account.

5.2. Operator decomposition

In [7], Standley describes a technique called operator decomposition (*OD*). Operator decomposition is designed to be an extension to *A** that improves the runtime of solving of *MAPF* problems.

In *MAPF*, when an agent moves, it can perform one of five actions (assuming a grid map is used): wait on the same square, or move in one of the four cardinal directions. With *A**, the movement of all agents is coupled. When a state is expanded, every child state contains the new position of every agent. This means that if there are 5 agents, which may all perform one of 5 possible actions, $5^5 = 3125$ child states are created and added to the frontier. This is called a full expansion.

Operator decomposition can lower this large branching factor by introducing the concept of partial states. As the name implies, when a state is removed from the frontier, it is not fully expanded. Instead, only the 5 possible actions of a single agent are evaluated. These 5 partial states are added to the frontier. When a partial state is removed from the frontier, the actions of the next agent are evaluated too until the actions of all agents are evaluated and a full expansion is reached again.

So in the worst case, *OD* still performs a full expansion. However, the advantage of *OD* is that partial states can also be prioritised in the frontier. When the expansion of a subset of agents is found to result in a low heuristic, the partial state is given a higher priority to be further expanded. This may lead to finding solutions to *MAPF* instances more quickly.

Even though *M** tries to avoid the coupled planning of agents as much as possible, when collisions occur, branching factors can still become large. *OD* can help *M** to reduce this branching factor when solving *MAPF* problems. [16] In Section 7.2 the benefits *OD* has on *M** when solving *MAPFM* problems is evaluated.

5.3. Pruning of matchings

Algorithm 2: prematch *M** with pruning

```
Result: Find the matching m_best with smallest
        cost, pruning when possible.
m_best ← ∅
---------------------------------------------
M ← find all matchings(starts, goals);
sort(M) {on heuristic; ascending};
foreach m ← M do
    if heuristic(m) < cost(m_best) then
        s ← mstar(starts, m);
        m_best ← min(m_best, s);
    end
end
```

*pmM**, as previously described, has to evaluate every matching. However, it is sometimes possible to discard some matchings without evaluating them with *M** at all, by using heuristics.

To do this, an admissible heuristic (e.g. the sum of distances between start and goal locations) of the initial state of every matching is calculated. This represents a lower bound for the cost of this matching.

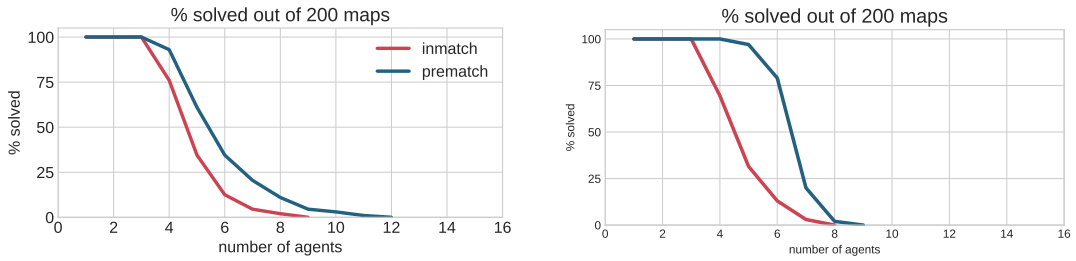
Then, when *prematching* evaluates every matching m, it keeps track of the best matching m_{best} which is the matching with best cost so far. However, when heuristic$(m) \geq cost(m_{best})$, m can immediately be pruned, because this matching m can never yield a solution which has a lower cost than m_{best}.

Sorting
To take maximum advantage of pruning, matchings can be sorted based on their heuristic. Making sure the matching with the lowest heuristic is evaluated first can increase chances that later matchings are pruned. Algorithm 2 shows how pruning with sorting works.

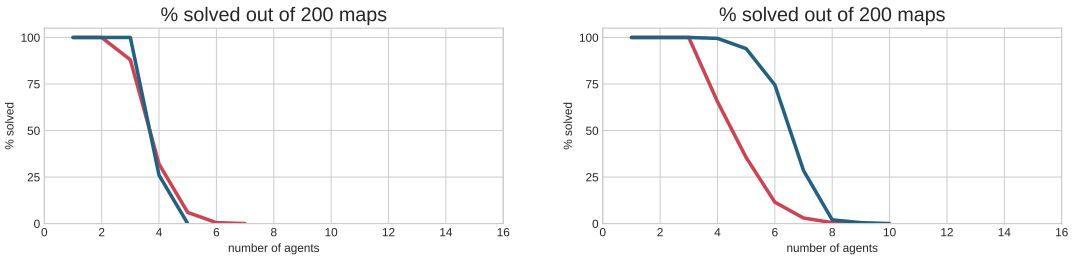
6. Experimental setup

To evaluate the performance of matching *M**, a number of experiments were performed. For each of these experiments, the algorithm used is written in Python 3.9 and benchmarks were run on a virtualized system with a 12 core Intel Xeon E5-2683 running at 2GHz, which has 8Gib of RAM.

There are various factors influencing how a *MAPFM* solving algorithm performs in benchmarks. Three important factors are the total number of agents, the number of teams over which the agents are distributed, and the layout of the maps on which benchmarks are run.



(a) Agents split over 3 teams. Left: 75% wall, right: 25%wall



(b) All agents in 1 team. Left: 75% wall, right: 25%wall

Figure 2: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps.

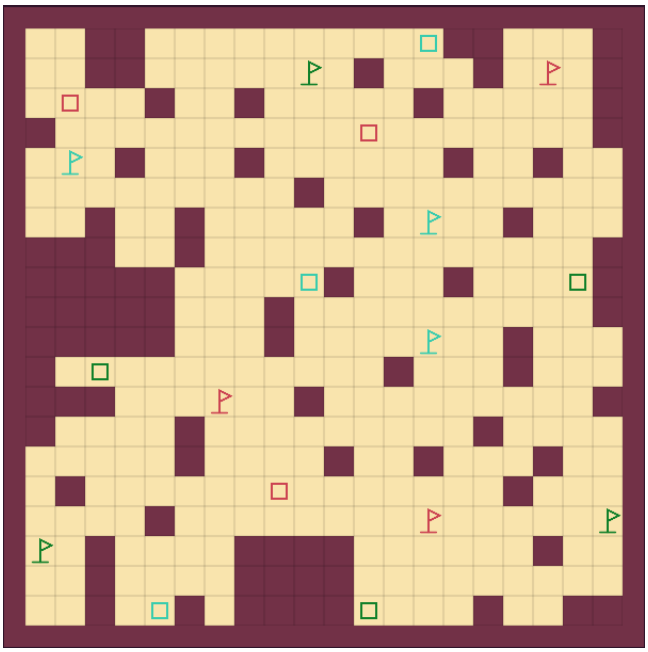


Figure 3: A map which is 25% obstacle, with agents distributed in three teams. This map was one of many used in the performed experiments.

The experiments are grouped into sets of four *scenarios*, to show the differences in performance as these parameters vary. Performance is assessed on maps where either 25% or 75% of the grid is an obstacle, and in situations where agents are grouped into either 1 or 3 teams.

For these benchmarks, 20×20 grid maps[3] are randomly generated such that they are guaranteed to be solvable (a proof of which can be found in A). Four sets of maps were generated for each experiment with the previously discussed parameters. Agents and goals are uniformly distributed on this map and teams are assigned randomly. When the team size does not divide the number of agents, there will be one smaller team. An example of such a randomly generated map can be found in Figure 3.

Because *MAPFM* is an *NP-Hard* problem, it is possible that finding the solution to an instance of the problem takes an unreasonable amount of time. Therefore, in each experiment, a timeout of 2 minutes is used. Experimental results show the percentage of maps (out of a set of 200) which the algorithm manages to solve within this timeout.

7. Results and Discussion

In this section, the results of a number of experiments is shown. The purpose of this is firstly to show which of the two matching strategies is superior, and why this is the case. Secondly, the experiments quantify the benefits of different extensions to *pmM**. Lastly, the results of the experiments are put into context by comparing *M** to other *MAPFM* algorithms performing the same experiments.

7.1. Matching strategies

In Section 4, two strategies were proposed for adding matching to *M**. Both strategies were tested following the experimental setup described in Section 6. In Figure 2 the results of this experiment are shown.

From these graphs, it can be seen that *prematching* is generally superior to *inmatching* in all but one of the

[3]Maps of different sizes will likely also show differences in performance. However, experiments with other map sizes are not included in this paper.

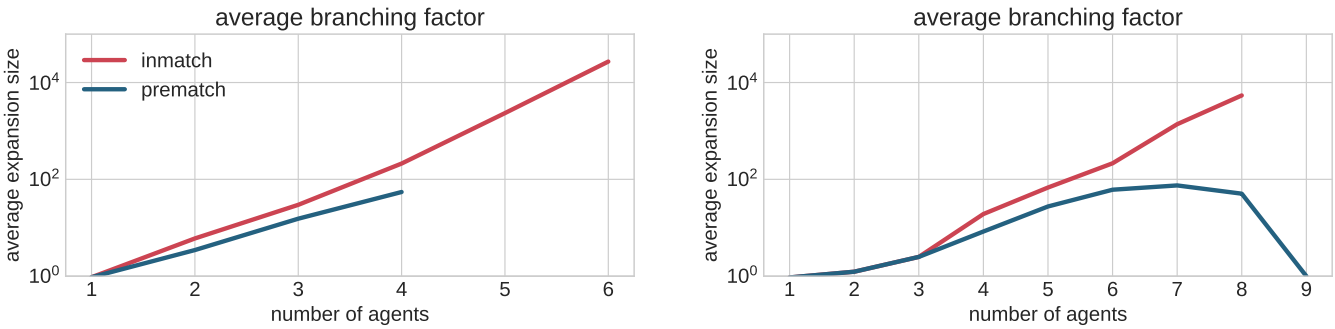


Figure 4: Average branching factor on the same maps as used in figure 2b (i.e. all agents in one team). Left: 75% wall, right: 25% wall. The blue line in the left figure stops earlier because *prematching* was not able to solve maps with more agents.

scenarios. Only in the 75% benchmark of figure 2b *inmatching* outperforms *prematching* slightly.

Branching factor
In Section 4.1, it was hypothesised that this difference in performance could occur, because of the larger branching factor.

To demonstrate this is indeed the case, another, separate, experiment was performed. Again on 200 random maps as described in Section 6, but this time showing the average number of states expanded each expansion. The exact same maps were used as in Figure 2b.

In Figure 4, the outcomes of this experiment are shown on a logarithmic scale. Indeed, on average, inmatching expands many more states.

When *inmatching* is better
However, in the scenario where 75% of the map is an obstacle in Figure 2b *inmatching* does seem to perform slightly better than *prematching*. When agents are all in one team, there are more matchings compared to when agents are separated into multiple teams. With *prematching*, each matching needs to be exhaustively searched, which is slow.

In contrast, the problem of *inmatching* expanding so many states is reduced in this scenario. There are fewer directions for agents to go because of the large number of obstacles.

So in general, when there are many matchings, but few options for agents to move in, *inmatching* can perform better.

7.2. Extensions to *M**

Various extensions have been proposed in Section 5. To show the benefits of each extension, they were compared following the experimental setup described in Section 6.

In Figure 5, the performance of these extensions is shown. Each line in the graph shows a new extension added to all of the previous extensions. For example the line displaying operator decomposition also uses precomputed heuristics and all other previous extensions.

Because in Section 7.1, *prematching* was observed to generally perform better, only the *prematching* matching strategy is used in this experiment.

Pruning and sorting
In the experiments where 25% of the maps is filled with obstacles, pruning and sorting have a large impact on performance. However, in graphs where 75% is a wall there is barely a difference at all. The heuristic used to prune, is the Manhattan distance between the agent start locations and the closest goal. This does not take into account the obstacles in the map.

Pruning is only possible when the heuristic is larger than the best matching found so far (as described in Section 5.3). This does not happen often with this inaccurate heuristic. Therefore, using the precomputed heuristic (also shown in Figure 5) makes a large difference.

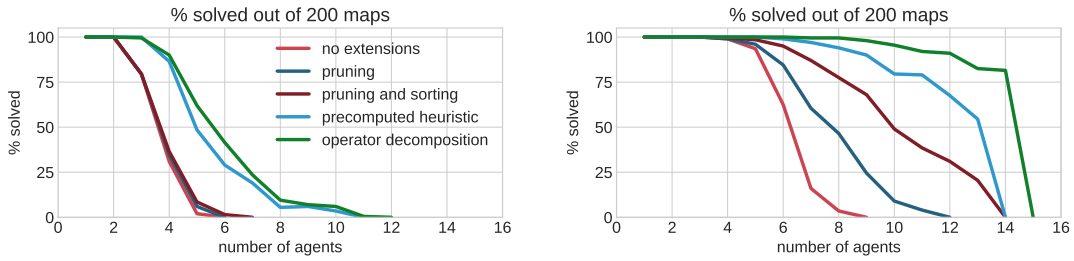
When there are fewer obstacles in the map, the probability of pruning is much higher, explaining the difference in performance of pruning and sorting between these two scenarios.

Memory usage
The best variant of *M** can be seen solving maps with 14 agents in Figure 5a. After this, the percentage abruptly drops to 0. The reason for this is not that finding the solution takes too long. Instead, attempts were terminated by the operating system because too much memory was used. Attempts on maps with 15 agents saw memory usages of over 8GiB to solve a single instance.

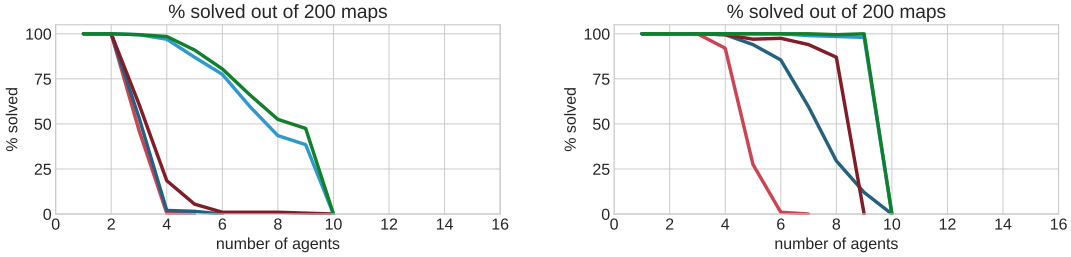
7.3. Comparison with other algorithms

This research is part of a set of parallel studies on how to extend a collection of *MAPF* algorithms with matching. All these studies used exactly the same problem definition but using one of the following base algorithms:

- Extended partial expansion *A** (*EPEA**) [12] (implementation and research by [17])
- *A** with operator decomposition and independence detection (*A*+OD+ID*) [7] (implementation and research by [18])



(a) 3 teams. Left: 75% wall, right: 25%wall



(b) 1 team. Left: 75% wall, right: 25%wall

Figure 5: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps. Each line and extension to M^* is graphed in combination with all previous extensions. For example, the line displaying *operator decomposition* also uses precomputed heuristics and all other previous extensions.

- Increasing cost tree search (ICTS) [14] (implementation and research by [19])
- Conflict-Based Min-Cost-Flow (CBM) [8] (implementation and research by [10])

Experiments were performed following the experimental setup described in Section 6. All algorithms were benchmarked on the same computer to ensure a fair comparison. Still, it is hard to compare algorithms well because, for example, *CBM* made use of external libraries written in *C++*. But, compared to the exponential nature of *MAPF*, such a strategy providing a linear speed-up is relatively insignificant. Figure 6 shows the results of this comparison.

In Figure 6b, it may look like an error was made. *CBM* solves 100% of the maps up to maps with 25 agents. In fact, *CBM* is able to solve maps with many more agents. *CBM* makes use of min-cost flow to solve *MAPFM* for single teams in polynomial time. Solutions for single teams are checked, and modified when conflicts are found between agents in different teams.

This means that in cases where there *is* only one team, *CBM* is able to solve *MAPFM* in polynomal time, scaling linearly with the number of agents for which a route needs to be found.

In the experiments where multiple teams were used (as shown in Figure 6a), *CBM* performs more comparably with other algorithms. Still, when there are few obstacles, *CBM* excels.

Apart from *CBM*, all other algorithms (including M^*) show very similar performance characteristics. All these algorithms use a form of prematching, where all matchings are exhaustively searched. This seems to be a factor which limits the capabilities of all these algorithms.

Only *A*+OD+ID* performs exceptionally well on maps 25% filled with obstacles, and with 3 teams. The reason for this is likely due to the way independence detection works in the implementation described in [18].

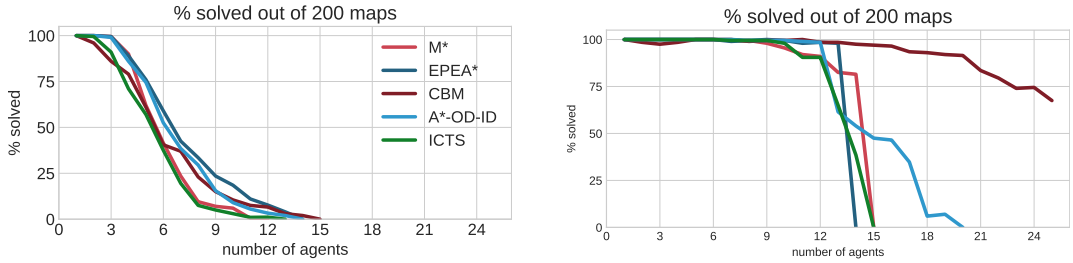
It must however be said, that this comparison is **not** complete. For example, testing with agents split over more teams, on larger maps, or with specifically crafted obstacles such as long corridors may show very different results.

8. Conclusion

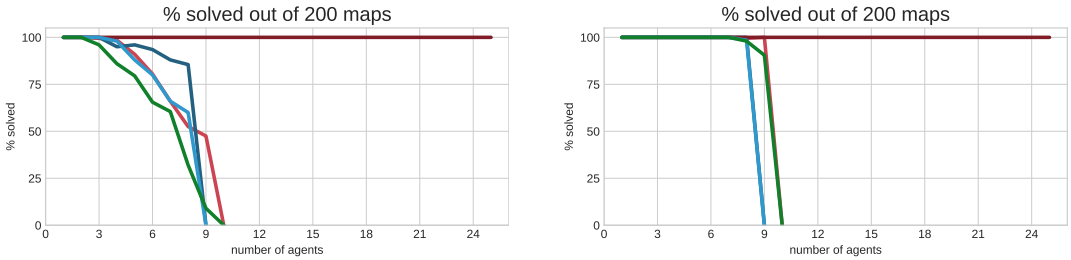
In this paper, a generalisation of multi agent pathfinding (*MAPF*) was introduced, called matching. This new problem is called *MAPFM*. M^* an algorithm for *MAPF*, was modified to solve these *MAPFM* problems. To do this, two strategies were explored called *inmatching* and *prematching*. Experimental results showed that in many cases, *prematching* is superior to *inmatching*.

Subsequently, several improvements to *prematching* M^* were considered and their benefits were experimentally evaluated. Pruning and sorting of matchings, using precomputed heuristics, and independence detection were shown to have a large effect on performance.

Finally, it was shown that *pmM** has very similar performance characteristics as *ICTS*, *EPEA** and *A*+OD+ID*. *CBM* performed notably better than M^* in experiments performed in this paper.



(a) 3 teams. Left: 75% wall, right: 25%wall



(b) 1 team. Left: 75% wall, right: 25%wall. In the right graph, the lines for *EPEA** and *A*+OD+ID* overlap, and so do the lines for *ICTS* and *M**.

Figure 6: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps.

9. Future work

Partial Expansion

One problem with the *inmatching* strategy presented in Section 4.1 is that the branching factor is very large. However, a lot of the expanded states are never needed or accessed again. Even with *prematching* this was a problem. For example, while creating the graphs in Figure 5, the data points where 14 or 15 agents were involved could use as much as 8GiB of memory.

Partial- or *enhanced partial expansion* as presented in [12] and [13] only expands states when they are needed to reduce memory usage. It is likely that partial expansion can also be applied to *M** to improve its memory performance.

Conflict based search

In this paper, matching *M** was compared to several other algorithms. One of those was CBS [9]. However, this implementation used a min-cost flow based approach similar to the one described in [8], but altered to find the *sum of individual costs* instead of the *makespan*.

CBS, without matching and these min-cost flow extensions, is one of the most studied algorithms for *MAPF* ([9, 20–23]). Comparing *M** with *inmatching* and *prematching* to *CBS* using *inmatching* and *prematching* as well, could provide useful data.

Waypoints

In previous research [24–27], *MAPF* was extended with waypoints instead of matching. It may be possible to combine these two extensions. For example, each team may have a set of waypoints. It does not matter which agent visits which waypoint as long as all waypoints are visited. Alternatively, each agent has their own waypoints, but the goals are still shared with a team of other agents.

Agents and goals

In the definition of matching presented in this paper, in each team the number of goals was always equal to the number of agents. However, in real world scenarios this may not always be the case.

For example, there may be a number of robots who have a number of tasks to do. Some robots cannot perform certain tasks so robots are teamed. But there may be more tasks than robots in each team, and robots will need to prioritise. Some research has already been done in this area, but this uses *TAPF* as a basis [28].

Recursive M*

In [5], an extension to *M** is described called recursive *M**. It could increase the performance of *M** on regular *MAPF* problems. Whether recursive *M** makes a difference combined with matching, has not yet been verified.

10. Reproducibility

Results in this paper have been generated using an implementation of *M** made in Python specifically for this research. The code for this, together with the maps and raw experimental results, are publicly available on Github at https://github.com/jonay2000/research-project, doubly licensed under the Apache 2.0 and MIT licenses. Reports of bugs and new additions to this repository are always welcome.

Acknowledgements

Thanks to Anna Kaal, Sebastiaan Dönszelmann, Noah Jadoenathmisier, Laura Pîrcălăboiu, Tim Anema and Jonathan Brouwer for their feedback on this paper. Thanks to Ivar de Bruin, Jaap de Jong, Robbin Baauw and Thom van der Woude for their feedback, and their experimental results which form an integral part of this paper's experiments. And finally thanks to Jesse Mulderij and Mathijs de Weerdt for their feedback and for supervising this project.

References

[1] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt, "Train unit shunting and servicing: A real-life application of multi-agent path finding," *arXiv preprint arXiv:2006.10422*, 2020.

[2] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," *arXiv preprint arXiv:2005.07371*, 2020.

[3] A. Mahdavi and M. Carvalho, "Distributed coordination of autonomous guided vehicles in multi-agent systems with shared resources," in *2019 SoutheastCon*, IEEE, 2019, pp. 1–7.

[4] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem"," *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[5] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *2011 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2011, pp. 3260–3267.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[7] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, 2010.

[8] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," *arXiv preprint arXiv:1612.05693*, 2016.

[9] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[10] R. Baauw, "Adapting cbm to optimize the sum of costs," 2021.

[11] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding.," in *IJCAI*, 2019, pp. 1289–1296.

[12] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, "Enhanced partial expansion a*," *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014.

[13] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. C. Holte, "Partial-expansion a* with selective node generation," *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pp. 180–181, 2012.

[14] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2013.

[15] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *arXiv preprint arXiv:1906.08291*, 2019.

[16] C. Ferner, G. Wagner, and H. Choset, "Odrm* optimal multirobot path planning in low dimensional search spaces," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 3854–3859.

[17] J. de Jong, "Multi-agent pathfinding with matching using enhanced partial expansion A*," 2021.

[18] I. de Bruin, "Solving multi-agent pathfinding with matching using A*+ID+OD," 2021.

[19] T. van der Woude, "Multi-agent pathfinding with matching using increasing cost tree search," 2021.

[20] E. Boyarski, A. Felner, R. Stern, G. Sharon, D. Tolpin, O. Betzalel, and E. Shimony, "Icbs: Improved conflict-based search algorithm for multi-agent pathfinding," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[21] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018.

[22] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Meta-agent conflict-based search for optimal multi-agent path finding.," *SoCS*, vol. 1, pp. 39–40, 2012.

[23] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search.," in *IJCAI*, 2019, pp. 442–449.

[24] N. Jadoenathmisier, "Extending cbs to efficiently solve mapfw," 2020.

[25] A. Michels, "Multi-agent pathfinding with waypoints using branch-price-and-cut," 2020.

[26] S. Siekman, "Extending a* to solve multi-agent pathfinding problems with waypoints," 2020.

[27] J. van Dijk, "Solving the multi-agent path finding with waypoints problem using subdimensional expansion," 2020.

[28] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[29] S. M. Johnson, "Generation of permutations by adjacent transposition," *Mathematics of computation*, vol. 17, no. 83, pp. 282–285, 1963.

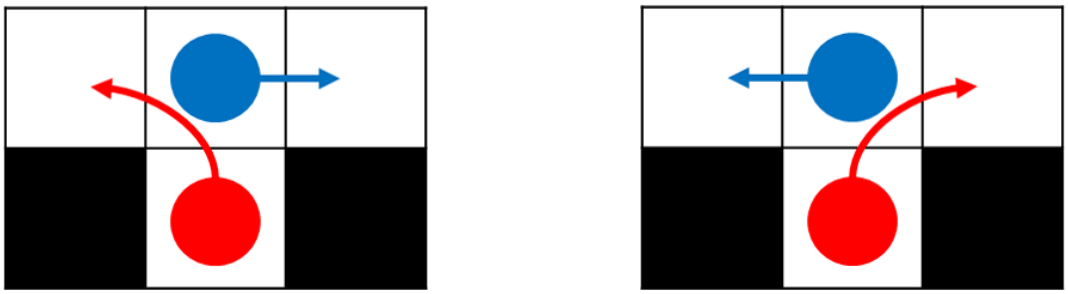


Figure 7: Agents passing each other on branch vertices

A. Generating feasible maps

By Jonathan Dönszelmann and Jaap de Jong

For experiments in this paper, *MAPFM* instances (sometimes called maps) are randomly generated. To generate these random maps, we created a program called the Multi-Agent Pathfinding instance generator (*MPIG*) which creates these maps. We designed *MPIG* to always create maps which are feasible. In this appendix, we show how *MPIG* works, and how *MPIG* can guarantee that each map is feasible by providing a generic procedure which can be used to solve any map generated by *MPIG*.

To simplify the explanation, we first demonstrate the process of generating feasible *MAPF* instances.

A.1. Properties of a feasible *MAPF* instance

MAPF instances are feasible whenever it is possible for every agent to reach their goal. *MPIG* ensures that this is always possible, by guaranteeing that every generated map has the following two properties:

1. Every map is connected. There are no disconnected subgraphs.
2. Maps with k agents and k goal vertices contain at least k vertices with three or more neighbours (i.e. vertices where at least three adjacent vertices are traversable). From now on, these locations will be called *branch vertices*. Branch vertices are important because at these locations, agents can pass each other as can be seen in Figure 7.

To guarantee the first property is satisfied, *MPIG* starts generation of maps at a single location, and neighbours of that vertex are recursively expanded (by either adding obstacles or traversable locations) to generate the rest of the map. Obstacles are not created when this would cause a disconnected subgraph to be created. The second property is guaranteed by simply discarding a map and generating a new map whenever there are fewer than k branch vertices. Discarding is used because chances are high that random maps contain more than k branch vertices.

Some maps which do not have these two properties may still be feasible, but this can not be guaranteed by the proof given in Section A.2.

A.2. Proof of feasibility

In this section, it is proven that when maps are connected, and there are at least k branch vertices, they are feasible. This proof consists of the following three parts which will be considered separately:

1. Every agent can always travel to a branch vertex from their starting location
2. When every agent is on a branch vertex, they can move to reorder themselves such that every agent can be on any of the branch vertices.
3. There exists a configuration of agents on branch vertices that allows all agents to go to their goal.

Part 1

Theorem A.1. *There is always at least one agent which can reach a branch vertex without collisions.*

Proof of Theorem A.1. Let v be a branch vertex and a_1 be the agent with the shortest path to v. Then, a_1 can reach v without collisions. □

Theorem A.2. *All agents can reach a branch vertex without collisions.*

Proof of Theorem A.2. There are at least k branch vertices. Thus, there are enough branch vertices to accommodate all agents. The process for every agent to reach this branch vertex is as follows:

Step 1: a single agent moves to a branch vertex, which is possible according to Theorem A.1.

Step 2: an attempt is made to move another agent a_i to a branch vertex. Two situations may occur:

1. Agent a_i can move to a branch vertex u without obstruction.
2. Another agent a_j which previously moved to a branch vertex v obstructs a_i from reaching a branch vertex u.

In the second situation, agent a_j can instead move to vertex u, freeing vertex v for agent a_i.

Step 2 can be repeated until all agents reach a branch vertex thus proving Theorem A.2. □

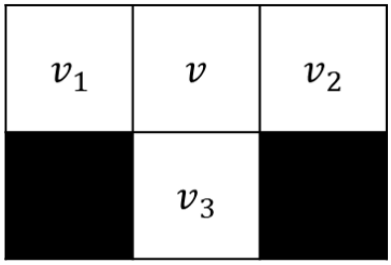


Figure 8: The trivial map with a single branch vertex

Part 2

Theorem A.3. *In every map, from every neighbour of a branch vertex u, there exists a path to a non-branch vertex that does not traverse u.*

Proof of Theorem A.3. A proof by construction follows:

In the trivial map with a single branch vertex (see figure 8), Theorem A.3 holds, since each neighbour is a non-branch vertex.

Every possible map with at least one branch vertex can be derived from the trivial map by adding more open vertices around it.

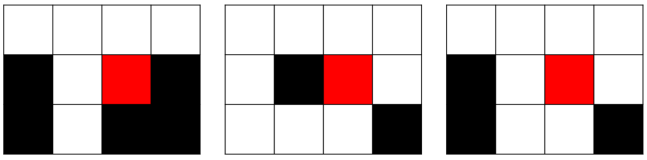


Figure 9: Three different ways of connecting vertices. Red vertices are added to maps.

Adding a new vertex u can have one of three effects on each neighbour v:

1. v has a single neighbour. Connecting to v makes v a two-neighbour vertex. Theorem A.3 trivially holds for v, because v is not a branch vertex.
2. v has two neighbours v_1 and v_2. Connecting to v makes v a three-neighbour vertex, i.e. a branch vertex. If v_1 or v_2 have fewer than three neighbours, then Theorem A.3 trivially holds. If v_1 or v_2 is a branch vertex, then they must already be part of the map and Theorem A.3 holds for v_1 and v_2. Since u is a branch vertex, it is possible to pathfind to one of the remaining neighbours of u, which are directly or indirectly connected to a non-branch vertex. Since Theorem A.3 holds for all neighbours v_1, v_2 and u of v, it must now also hold for v.
3. v has three neighbours, v_1, v_2 and v_3. Connecting to v makes v a four-neighbour vertex. The same reasoning used in effect 2 can be used to show that Theorem A.3 still holds for v.

Adding a new vertex u can also have one of the following effects on u itself:

1. It can create a new two-neighbour vertex u by connecting two vertices (shown figure 9). Since u has fewer than three neighbours, Theorem A.3 still holds for u.
2. It can create a new three-neighbour vertex u by connecting three vertices (shown figure 9). A vertex with which a connection is made (called v), can be in one of three possible configurations for which Theorem A.3 holds, as explained in the previous part of the proof. Since Theorem A.3 holds for all neighbours of u, it also holds for u itself since every neighbour is always directly or indirectly connected to a non-branch vertex.
3. It can create a new four-neighbour vertex u by connecting four vertices (shown in figure 9). A vertex with which a connection is made (called v), can be in one of three possible configurations for which Theorem A.3 holds. The reasoning from the previous effect can be used to show that Theorem A.3 also holds in this effect.

For the trivial map from Figure 8, Theorem A.3 trivially holds. Every map with one or more branch vertices can be constructed from the trivial map by adding vertices to it. Adding vertices to a map for which Theorem A.3 holds, was shown to exclusively create new maps for which the Theorem still holds. If a map cannot be derived from the trivial map, then it does not contain branch vertices. Theorem A.3 trivially holds for maps without any branch vertices.

Therefore, Theorem A.3 holds for all generated maps. □

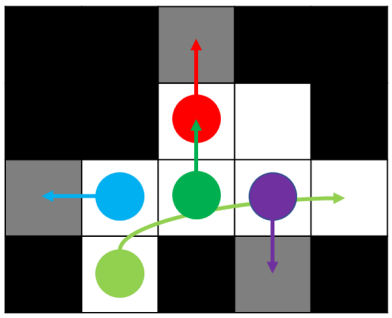


Figure 10: An example of how an agent can pass other agents even if there is no space between branch vertices

Consider the scenario where each agent is positioned on a branch vertex. As a result of theorem A.3, each neighbour of a branch vertex u has a so-called *diversion vertex*, which is the non-branch vertex that is reachable from the neighbour without visiting u.

Theorem A.4. *An agent a_i on a branch vertex v can always be passed by another agent a_j*

Proof of Theorem A.4. The branch vertex v has three neighbours v_1, v_2 and v_3 (as shown in Figure 8). a_j passing v means that it is coming from one of the neighbours of v (say v_1) and needs to travel to another one of the neighbours (say v_2). For a_j to travel from v_1 to v_2, a_i must move out of the way to v_3. v_3 can either be:

- A non-branch vertex. It is therefore empty because all agents are on branch vertices. a_i can simply move to v_3 and let a_j pass.
- A branch vertex. In this case, there may be an agent a_k on v_3. If there is an agent on v_3, it must also move out of the way. Theorem A.3 shows that it is always possible to pathfind to a non-branch vertex from neighbours of branch vertices. Since non-branch vertices are empty, this provides a place for agents to move in to make room for passing agents. Therefore, a_k must move either onto an empty vertex, or move onto a vertex with another agent which after possible repetitions will always find an empty *diversion vertex* to move onto. Figure 10 shows how all agents move out of the way to *diversion vertices* to allow the lime agent to pass.

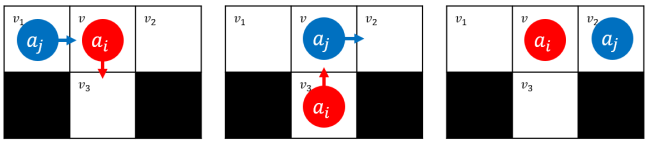


Figure 11: An example of how an agent can pass another agent.

After having encountered one of these two scenarios, agent a_j has moved to v, and a_i has moved out of the way to v_3. For agent a_j to now completely pass a_i, a_j must continue to v_3 (these steps are show in Figure 11). However, if v_3 is another branch vertex, another agent a_k may be on it. Two situations can now occur:

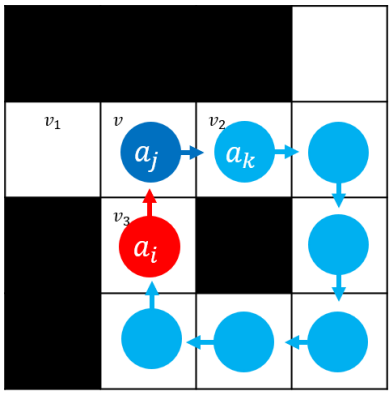


Figure 12: An example of how agents can pass with a single diversion vertex.

- a_k can move out of the way just like a_j did. Theorem A.3 shows that this is always possible to find a diversion vertex. a_i can now also move back to v.
- a_k can not move out of the way. Even though Theorem A.3 shows that there is always a diversion vertex to move out of the way, a_j moving out of the way may have taken up this diversion vertex. However, if both v_2 and v_3 have the same diversion vertex, a connection must exist between v_2 and v_3. Because the definition of *MAPF* allows following, it is now possible for a_k to move out of the way, following agents in front of a_k in a chain. The head of the chain is a_i. a_i moves back to v, in a way making v the diversion vertex. This motion can be seen in Figure 12

After this process, a_i is back on v and a_j has passed to v_3 □

Theorem A.5. *Any two agents on adjacent branch vertices (i.e. directly connected or connected with a corridor) can swap places, both moving to the branch vertex where the other agent was standing.*

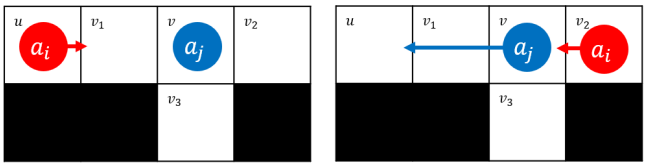


Figure 13: An illustration of two agents swapping by passing each other.

Proof of Theorem A.5.

Lemma A.6. *The swapping of two agents a_i and a_j, positioned on branch vertices u and v respectively, is equivalent to a_i passing a_j (or vice versa). After the passing, both agents can move to the vertex where the other agent used to be without conflict.*

Theorem A.4 shows that an agent coming from one neighbour of a branch vertex can always pass the branch vertex to move to another neighbour of the branch vertex. Agents a_i and a_j can swap by one of the agents passing the other agent on its branch vertex and both agents moving back to the swapped branch vertices without collision. This process can be seen in Figure 13. There is always enough space for an agent to pass another agent because of the diversion squares.

Therefore, two agents on adjacent branch vertices can swap places. □

Theorem A.7. *If all agents are assigned to and located on a branch vertex, they can move to create every other assignment of agents to branch vertices.*

Proof of Theorem A.7. Any permutation of a set of elements can be created using only pairwise swaps by using the Steinhaus–Johnson–Trotter algorithm [29].

Not all pairwise swaps are swaps between adjacent elements. However, any pairwise swap of two non-adjacent elements a and b can be performed by swapping all the elements between a and b. The procedure

The proof of theorem A.5 shows that pairwise swaps of agents on adjacent branch vertices are possible on any map. □

Part 3

Theorem A.8. *Every connected map with n agents on n branch vertices is directly solvable from at least one assignment p of agents to branch vertices.*

Proof of Theorem A.8. Consider the scenario where every agent is positioned on its corresponding goal. By theorem A.2, the agents can all travel to branch squares without collision. This results in a assignment p of agents to branch squares. □

Theorem A.9. *Every connected map with n agents and at least n branch vertices is feasible.*

Proof of Theorem A.9. By Theorem A.2, it is possible for every agent to reach a branch vertex. By Theorem A.7, every assignment of agents to branch vertices can be created. Theorem A.8 shows that there is always an assignment for which the map is solvable. Therefore, every map with at least n branch vertices is feasible. □

A.3. Solving *MAPFM* instances

Theorem A.10. *Every connected MAPFM map with n agents and at least n branch vertices is feasible.*

Proof of Theorem A.10. A *MAPFM* instance can be decomposed into many *MAPF* instances by considering all possible assignments of agents to goals, which can exhaustively be searched. Theorem A.9 shows that every *MAPF* instance is feasible. As a result, every possible assignment of agents to goals of a *MAPFM* instance is also feasible. Therefore, all *MAPFM* instances with n agents and n branch vertices are feasible as well. □