Multi-Agent Pathfinding with Matching using Enhanced Partial Expansion A*

Jaap de Jong[*]
TU Delft

Abstract

For the Multi-Agent Pathfinding (MAPF) problem, a set of non-colliding paths must be found for multiple agents. In Multi-Agent Pathfinding with Matching (MAPFM), this problem is extended: agents and goals are added to a team and each agent has to navigate to a goal that belongs to the same team. In this paper, two extensions of the EPEA* MAPF solver will be discussed that enable it to solve MAPFM problems. The first extension modifies the EPEA* algorithm to directly allow it to solve MAPFM problem instances. The second extension generates all possible goal assignments for each agent and runs EPEA* on these assignments. This last extension is shown to have a superior performance in most cases. The second extension is also compared to extensions of other MAPF algorithms.

1 Introduction

The Multi-Agent Pathfinding (MAPF) problem is the problem of planning paths for multiple entities without the entities colliding with each other. There are multiple real-world applications for this problem, such as robot routing in warehouses or docks [1] and train routing on shunting yards [2].

In MAPF, the moving entities such as trains or robots are called *agents*. Each agent has a starting location and a goal location. A solution to a MAPF problem is a combination of non-conflicting paths from start to goal locations. Solutions are optimal if the sum of path lengths for all agents, also called the Sum of Costs (SoC), is minimized [3]. Finding a solution to the MAPF problem with a minimal SoC is NP-hard [4].

Various algorithms have been proposed for solving the MAPF problem. Current state-of-the-art algorithms are Conflict-Based Search (CBS) [5] and Branch-and-Cut-and-Price (BCP) [6]. Other algorithms include ICTS [7], M* [8] A*+OD+ID [9] and EPEA* [10, 11].

An extension of the MAPF problem is the Multi-Agent Pathfinding with Matching (MAPFM) problem, where each agent and each goal is assigned to a team. Each agent can travel to each goal belonging to the same team. In this paper, EPEA* will be used as a base algorithm for solving MAPFM problem instances, because it is a relatively fast algorithm that uses a very small amount of memory and can easily be extended upon.

There is a substantial number of real-world applications for MAPFM. A classic example of this is a warehouse where multiple robots are able to perform the same tasks. Similarly, when a train has to depart from a shunting yard, the precise train instance does not matter but the train type generally does.

To date, there are no existing EPEA*-based algorithms that support matching. Ma et. al [12, 13] have researched extending the Conflict-Based Search (CBS) algorithm with matching by developing an algorithm called Conflict-Based Min-cost-flow (CBM). In this research, the problem is reduced to a multi-commodity flow problem. However, there is no evident way to combine a reduction to a multi-commodity flow problem with an A*-based algorithm such as EPEA*. In addition, CBM minimizes the makespan instead of the SoC, so the CBM algorithm is not directly applicable to the MAPFM problem.

The objective of this research is to find a way to extend EPEA* to acquire an algorithm that finds optimal solutions to the MAPFM problem. This is accomplished by introducing two new extensions, called heuristic matching and exhaustive matching, and comparing them with each other.

This paper will first give a formal definition of the MAPFM problem, followed by a description of EPEA*. Then, an overview of the EPEA* algorithm will be given, followed by a description of the heuristic matching and exhaustive matching extensions. After that, the extensions are compared with each other and with MAPFM solvers based on other MAPF algorithms in terms of runtime performance and memory usage.

2 Multi-Agent Pathfinding with Matching

Because Multi-Agent Pathfinding with Matching (MAPFM) is an extension of the standard MAPF problem, the definition and terminology of MAPF will first be introduced. This definition will then be extended to MAPFM.

[*]Supervised by Jesse Mulderij (J.Mulderij@tudelft.nl) and Mathijs de Weerdt (M.M.deWeerdt@tudelft.nl)

2.1 The MAPF Problem

Input

To model the input of a MAPF problem with k agents, the following terminology is used [3]:

- G is an undirected graph. The EPEA* implementation uses a 4-connected grid which eases the generation of problem instances and the comparison and testing of the algorithms.
- s is a list of length k where s_i is the starting location for agent a_i.
- g is a list of length k where g_i is the goal location for agent a_i.

Solution

A solution to the MAPF problem is a set of paths π_i, where each path corresponds to an agent a_i. A path is a list of vertices that defines the location of each agent at every time step. It has the following properties:

- The first vertex $\pi_i[0] = s_i$
- The last vertex $\pi_i[|\pi_i|] = g_i$

The agents stay at their goal after their path is completed, so other agents can still collide with them. This is called stay-at-target [3].

Constraints

Several constraints could be considered when finding paths for all agents [3]. In this research, only two constraints will be considered:

- **Vertex conflict** - A vertex conflict occurs when two agents are at the same vertex at the same time t, that is $\pi_i[t] = \pi_j[t] \land i \neq j$.
- **Edge conflict** - An edge conflict occurs when two agents travel along the same edge at the same time step t. In a 4-connected grid this corresponds to swapping the agents, that is $\pi_i[t] = \pi_j[t+1] \land \pi_i[t+1] = \pi_j[t] \land i \neq j$.

Objectives

A MAPF problem can have different objective functions [3]:

- **Makespan** - A MAPF algorithm with minimal makespan as an objective minimizes the length of the longest path, $\max_i |\pi_i|$.
- **Sum of Costs (SoC)** - A MAPF algorithm with minimizing the sum of costs as objective tries to minimize the total path length, $\sum_i |\pi_i|$.

In this research, the SoC is minimized because it is the most common objective function in search-based MAPF algorithms [3].

2.2 The MAPFM Problem

The Multi-Agent Pathfinding with Matching (MAPFM) extends the MAPF problem by introducing teams of agents. Starting locations in s and goal locations in g no longer correspond to a specific agent, but rather to a team. There are K teams. Each team j has:

- k_j starting locations s_i^j for $i = 1..k_j$.
- k_j goal locations g_i^j for $i = 1..k_j$

The number of starting locations k_j is equal to the number of goal locations for each team.

In the solution, an agent with starting position s_i^x can travel to any goal g_j^y as long as $x = y$, meaning that they are in the same team.

3 EPEA*

This section gives an overview of the existing A*, PEA* and EPEA* algorithms that serve as a foundation of the MAPFM extensions described in Section 4.

3.1 A*

A* is a heuristic-based search algorithm that is commonly used for the single-pair shortest path problem[1]. An effective heuristic for MAPF A* is the Sum of Costs (SoC) heuristic. It is defined as the sum of all individual agent heuristics:

$$h(n) = \sum_{i=1}^{k} h'(a_i) \tag{1}$$

where $h(n)$ is the heuristic of an A* node n and $h'(a_i)$ is the individual agent heuristic for agent a_i. The individual agent heuristic is the single-pair shortest path distance from the agent to the goal. This distance is needed for every agent every time a node is expanded. Since agents can have the same position in different nodes, the same distance is often calculated multiple times. Therefore, the single-pair shortest path distance is precomputed for every agent. An effective way of precomputing this is to start a breadth-first search at the goal location and running the search until it has found a distance for all traversable locations in the grid.
An A* node contains:

- The state of the grid, that is the positions of the agents
- The heuristic $h(n)$
- The cost of reaching the node $c(n)$
- A reference to the parent of the node. This is necessary for retrieving the set of paths once the optimal solution has been found.

The A* nodes are ordered in increasing order by $f(n) = c(n) + h(n)$, that is the sum of cost and heuristic. The node with the lowest $f(n)$ is expanded first.

3.2 Partial Expansion A*

Partial Expansion A* (PEA*) reduces the space complexity of A* by collapsing the nodes in the frontier [14]. In addition to the properties mentioned in Section 3.1, a PEA* node also stores the value $f(n)$. Initially, $f(n) = c(n) + h(n)$, as described before.

When a child node n' is generated, only the nodes with $f(n') = f(n)$ are added to the frontier and the other nodes are discarded. The algorithm keeps track of the lowest $f(n')$ of the discarded children and increases the parent node value

[1]The single-pair shortest path problem is the problem of finding the shortest path from a single source to a single goal location

Moves	$\Delta f(n)$
NORTH, EAST	0
WAIT	1
SOUTH, WEST	2

Figure 1: An example of a PDB-table

$f(n)$ to $f(n')$. The parent is reinserted into the frontier if at least one child was discarded, which causes it to act as a placeholder for the discarded child nodes. This prevents the child nodes from taking up space in the frontier.

3.3 Enhanced Partial Expansion A*

A big flaw in PEA* is that it takes a significant amount of time to generate the child nodes and to check for conflicting moves. Because numerous child nodes are immediately discarded, PEA* often has to do this multiple times for the same node. Enhanced Partial Expansion A* (EPEA*) solves this problem by using an Operator Selection Function (OSF) to determine which operators (agent moves) will produce a node with the desired $f(n)$ before the node is generated [10, 11, 15].

Before the main search starts, a Pattern Database (PDB) is constructed. This pattern database contains a table for each agent for each possible location in the grid. For that location, the heuristic values for all possible agent moves are retrieved from the stored heuristic data. The change in node value $\Delta f(n) = \Delta h(n) + 1$ is stored for each direction. The results are then sorted on increasing $\Delta f(n)$ and the directions are grouped by $\Delta f(n)$-values. An example of such a table can be found in Figure 1. During the A* search, the OSF is used to select a direction from the PDB for each agent such that

$$\sum \Delta f(n') = f(n) - c(n) - h(n) \tag{2}$$

In other words, it finds the combination of moves that will result in the desired $\Delta f(n')$ value [11].

3.4 Independence Detection

Independence Detection (ID) [9] can be used as an extension of EPEA*. ID starts by planning a path for each agent individually, without considering collisions with other agents. When two individual paths are in conflict with each other, the paths are discarded, the corresponding agents are grouped and ID runs EPEA* on the agents in the group. When two groups have conflicting paths, all paths that belong to the groups are discarded, the groups are merged and ID runs EPEA* on the new group to find a set of non-conflicting paths.

The effectiveness of ID is dependent on the problem instance on which it is run. ID will improve the runtime only if it is able to prevent having to plan all agents together.

4 EPEA* with Matching

In this section, the MAPFM extensions to EPEA* are discussed. First, an approach called heuristic matching is introduced, followed by an exhaustive matching approach together with several optimizations for the exhaustive matching approach.

4.1 Heuristic matching

One way of solving the MAPFM problem using EPEA* is to adapt EPEA* to also take MAPFM problem instances as input. This can be achieved by modifying the heuristic function.

Instead of using the distance to a single goal as an agent heuristic, the distance from the agent to the closest goal of the same team is used. The resulting agent heuristics are then added together to produce the SoC heuristic value. For the results to be optimal, the modified SoC heuristic has to be admissible, which means that it should never overestimate the distance to the goal. The agent heuristic never overestimates, since it always gives the distance to the closest goal. The SoC heuristic gives the cost of the optimal solution if collisions would be allowed. Since collisions cannot decrease the path length, this cost is a lower bound and it can never overestimate, thus making it admissible. When an admissible heuristic is used, A* generates optimal solutions [16].

In heuristic matching, the heuristic function is the same for agents of the same team. Therefore, only one heuristic function has to be computed for every team. An effective way of doing this is by using a breadth-first search of the entire grid, as can be seen in Algorithm 1. The breadth-first search is modified to start with every goal of the team as a starting location (Algorithm 1 line 5-7).

Algorithm 1 Team heuristic calculation

```
1: function PRECOMPUTE HEURISTIC(goals)
2:     $grid$ ← 2d grid with all values set to $\infty$
3:     let $Q$ be a queue
4:     let $explored$ be an empty set of locations
5:     for all $goal \in goals$ do
6:         enqueue $(goal, 0)$ into $Q$
7:     end for
8:     repeat
9:         $(loc, distance)$ ← first element in $Q$
10:        update location $loc$ in $grid$ with $distance$
11:        add $loc$ to $explored$
12:        for all $neighbour \in$ neighbours of loc do
13:            if $neighbour \notin explored$ then
14:                enqueue $(neighbour, distance + 1)$
15:            end if
16:        end for
17:    until $Q = \emptyset$
18: end function
```

4.2 Exhaustive matching

A different way of solving the MAPFM problem using EPEA* is to split the MAPFM problem into multiple MAPF problems. The key idea is to find every possible assignment of goals to agents and to solve each assignment using EPEA*. As a result, EPEA* needs to run M times:

$$M = \prod_{j=1}^{K} k_j! \tag{3}$$

This can be made more efficient by pruning the A* search. Since A* and EPEA* always expand the most promising

Algorithm 2 Exhaustive matching

```
1: function EXHAUSTIVE MATCHING(grid, starts, goals)
2:     Calculate heuristic function $h$ for every goal using $grid, goals$
3:     Calculate OSF for all goals using $h$
4:     $goal\_assignments \leftarrow$ all possible goal assignments with their initial heuristic
5:     Sort $goal\_assignments$ on ascending initial heuristic
6:     $best\_cost \leftarrow \infty$
7:     Initialize $best\_solution$
8:     for all $goal\_assignment \in goal\_assignments$ do
9:         $h \leftarrow$ initial heuristic of $goal\_assignment$
10:        if $h \geq best\_cost$ then
11:            return $best\_solution$
12:        end if
13:        Run EPEA* on $goal\_assignment$
14:        $s \leftarrow$ solution
15:        $c \leftarrow$ cost
16:        if $c < best\_cost$ then
17:            $best\_cost \leftarrow c$
18:            $best\_solution \leftarrow s$
19:        end if
20:    end for
21:    return $best\_solution$
22: end function
```

nodes first and the heuristic never overestimates, it will never find a solution with a lower cost than the heuristic of the node it is expanding. The exhaustive matching solver keeps track of the cost of the best solution found so far. When the $f(n)$ value of the node being evaluated in the EPEA* algorithm exceeds the cost of the best known solution, it is guaranteed that the best solution of the goal assignment has a higher cost than the best known solution of already evaluated goal assignments. Therefore, the EPEA* search is terminated and the exhaustive matching continues with the next goal assignment.

4.3 Exhaustive matching with sorting

When the optimal solution corresponds to the first goal assignment that is evaluated, the EPEA* search can be pruned much more effectively than when it is the last goal assignment that is evaluated. It is not possible to know beforehand which goal assignment corresponds to the optimal solution. However, it is possible to start with the most promising goal assignments. How promising a goal assignment is, is determined by the initial SoC heuristic, which is defined as the sum of distances for each agent towards their goal without considering collisions.

Each goal assignment is stored together with the corresponding initial heuristic (Algorithm 2 line 4). These assignments are then sorted on the initial heuristic value and evaluated one by one (line 5). Once the initial heuristic value exceeds the cost of the best known solution, the solution of the remaining goal assignments will not improve the best known solution, so this solution is guaranteed to be optimal (line 11).

Optimization I

A problem with exhaustive matching with sorting is the calculation of the initial heuristic. To calculate the initial heuristic for a goal assignment, a single-pair shortest path needs to be found for every agent. Calculating this can take a long time which is unnecessary since the heuristic function is precomputed once the matching will be solved.

This problem is solved by immediately precomputing the heuristic function when the matching is generated instead of when the matching is evaluated. However, this solution causes a new problem. Because of the factorial growth of goal assignments with team size, there is an insufficient amount of memory to store all precomputed heuristic functions.

In an initial attempt to repair this problem, a priority queue was introduced for the goal assignments. This allows the algorithm to only sort a predetermined number of goal assignments, thus limiting the number of precomputed heuristic functions that have to be stored at once. However, it also decreases the runtime performance of the algorithm.

Optimization II

In a second attempt to repair the problem of the factorial memory requirements, the need to store different precomputed heuristic functions was removed altogether. This was accomplished by introducing a single precomputed heuristic function that can be used for all goal assignments.

First, a new set of goals has to be obtained with each goal assigned to a unique team, similar to a standard MAPF problem. The heuristic function can then be precomputed using a breadth-first search from each goal, similar to the algorithm described in Algorithm 1. The initial heuristic value for a goal assignment can be calculated by performing a lookup in the heuristic function for each goal, with the position of the agent that is assigned to the goal.

When evaluating a goal assignment, each agent is inserted into a team with only the assigned goal of the agent, which gives the EPEA* solver the information of which goal to use when retrieving the heuristic from the heuristic function.

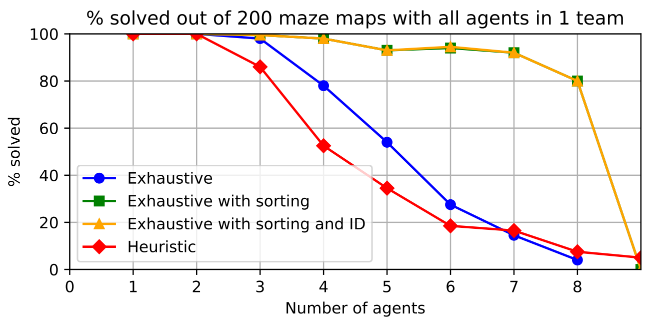
Next to solving the problem of the large number of stored precomputed heuristic functions, the single heuristic function also improves the standard exhaustive matching algorithm. Instead of precomputing a heuristic function for every goal assignment, the heuristic function only has to be precomputed once, thus removing the computational overhead caused by calculating the heuristic multiple times.

Since the PDB for the OSF is also precomputed multiple times, the algorithm was modified to also share the PDB between goal assignments. This does not save memory, because only one PDB is stored at any time. However, sharing the PDB does save the computation of constructing a PDB for each goal assignment.

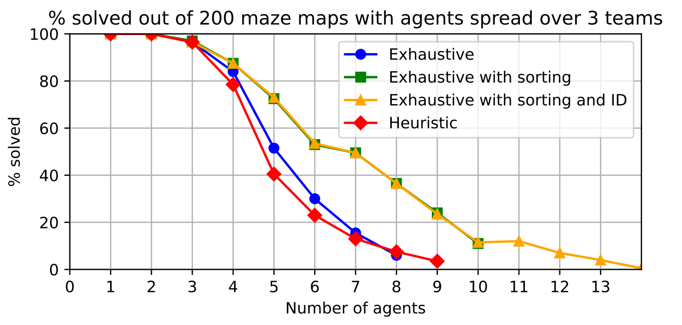
4.4 Exhaustive matching with sorting and ID

The addition of Independence Detection (ID) to exhaustive matching is an optimization that aims to reduce the number of matchings that have to be evaluated. This addition is described in detail by [17], but this section will give a short overview.

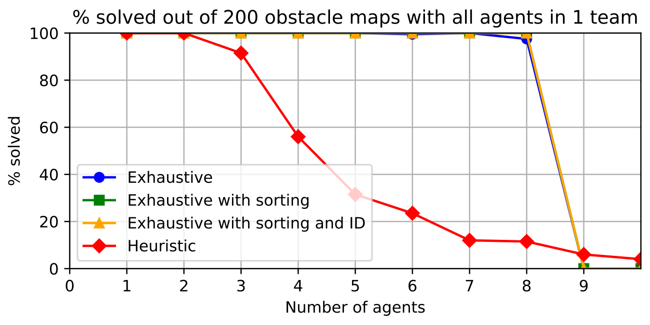
The idea is similar to the idea described in Section 3.4, but



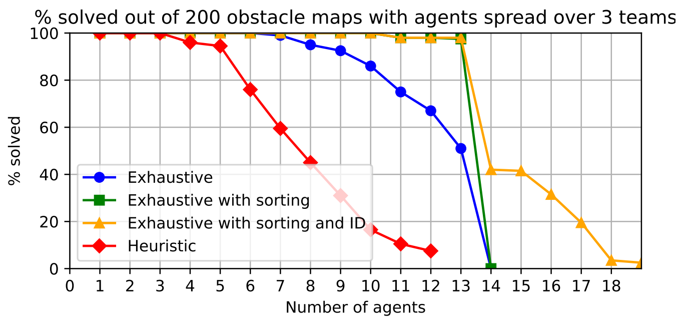
(a) Maze maps with all agents in a single team



(b) Maze maps with agents divided over three teams



(c) Obstacle maps with all agents in a single team



(d) Obstacle maps with agents divided over three teams

Figure 2: Comparison of different EPEA*-based MAPFM algorithms on maze maps and obstacle maps

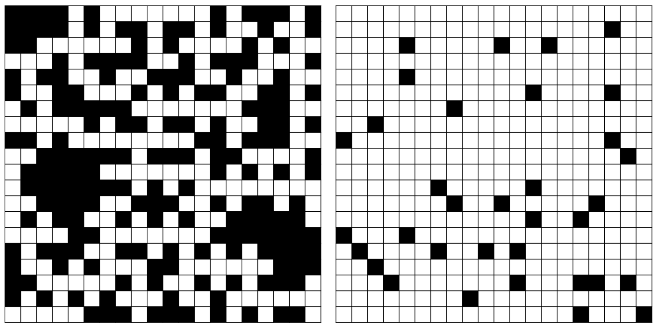


Figure 3: A maze map (left) and an obstacle map (right)

it is now applied on teams of agents. First, the top-level ID runs the exhaustive matching algorithm on every team. If the resulting paths are conflicting, the teams are inserted into a group and the exhaustive matching algorithm is executed on the group to find a set of non-conflicting paths. This process is repeated until a set of non-conflicting paths has been found. When teams can be planned independently, the ID will reduce the number of goal assignments that have to be evaluated.

5 Experimental Setup, Results and Discussion

The aim of this section is to assess the runtime and memory performance of the developed extensions and their optimizations and to compare them with extensions based on other MAPF algorithms. The setup of the experiments is discussed first, after which the results of the experiments are discussed.

5.1 Random map generation

To evaluate the algorithms described above, a random map generator was developed. The map generator always generates solvable connected maps (Appendix A contains a proof of solvability). There are multiple different kinds of maps that can be generated using the map generator, but during this research, two kinds of maps were used:

- **Maze maps** - Maze maps have a high wall density and are acyclic. The narrowness of the passages cause numerous collisions, which makes this map a good candidate for testing how algorithms handle collisions.
- **Obstacle maps** - Obstacle maps have a low wall density and are cyclic. Collisions can easily be avoided, so this map is able to test the limits of the amount of goal assignments that the exhaustive matching algorithms can handle.

An example of both map types can be seen in Figure 3. The performance of the different matching algorithms was quantified using 20x20 maze maps and obstacle maps generated by the map generator.

5.2 Benchmarks

All runtime performance benchmarks were run in a virtualized Linux environment with 12 cores running at 2.0 GHz on an Intel Xeon E5 2683 v3 CPU. To save time, the algorithm was run on 10 benchmarks simultaneously, because of the number of threads that were available and the low RAM usage. There was 8 GiB of RAM available, and the memory usage stayed well within that during all benchmarks.

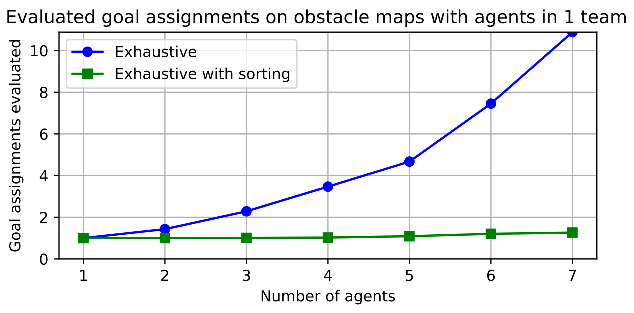


Figure 4: Comparison on the average number of evaluated goal assignments in 200 samples between exhaustive matching and exhaustive matching with sorting

The benchmarks were run using random maps generated using the generator described in Section 5.1. All the implementations that are compared are implemented and executed with Python 3, but the CBM-based implementation makes use of a C++ library.

5.3 Results

Heuristic versus Exhaustive

The performance of heuristic matching and standard exhaustive matching is very similar on maze maps. Figure 2a and 2b show that exhaustive matching is able to solve a bit more problems with fewer agents, but above a certain number of agents, the performance of exhaustive matching drops slightly below the performance of heuristic matching.

A similar observation can be made on obstacle maps with all agents in one team (Figure 2c), except that there is a bigger dissimilarity in performance. Exhaustive matching is able to solve almost all problems with up to eight agents, while heuristic matching only solves close to 50% of problems with four agents. However, while exhaustive matching is unable to solve problems with nine agents within the timeout, heuristic matching still solves a very small number.

This phenomenon is due to the inner workings of heuristic matching. Heuristic matching uses the distance to the closest goal as a heuristic for each agent. When each agent has the distance to a unique goal as a heuristic, there will seldom be any collisions. However, when two agents are guided to the same goal by the heuristic function, there will eventually be a collision at the goal, causing ID to plan the agents together.

An example of a map that can be solved by heuristic matching but cannot be solved by exhaustive matching can be seen in Figure 5. There are multiple goal locations close to starting locations and two-thirds of the agents can travel to their closest goal for an optimal solution.[2]

Sorting

This section discusses the effect of sorting on the performance of exhaustive matching. Figure 2a and 2b show a clear difference between exhaustive matching with and without sorting on maze maps.

[2]An interactive version of the solution can be found at https://mapf.nl/solutions/2609

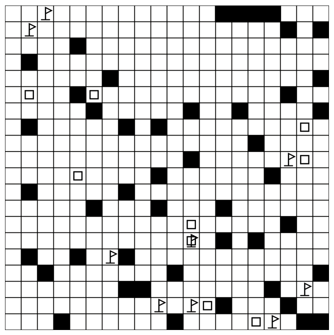


Figure 5: An example of an obstacle map with nine agents in a single team that can be solved using heuristic matching but cannot be solved by exhaustive matching within two minutes. Squares are starting locations and flags are goal locations.

What stands out in Figure 2a, where all agents are in the same team, is the steady decline in the percentage of maps solved with exhaustive matching, whereas exhaustive matching with sorting maintains a high solving percentage up to problems with eight agents. The difference is smaller when agents are divided over three teams, as can be seen in Figure 2b.

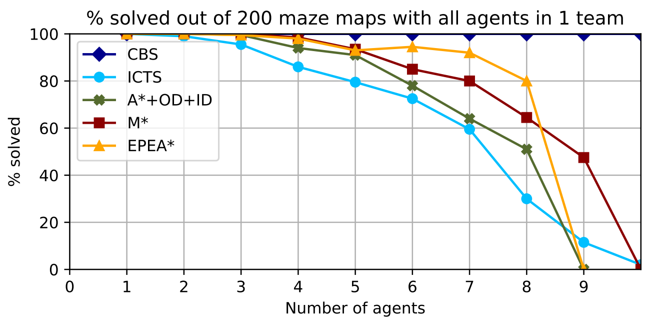
An interesting observation is that the performance of exhaustive matching with sorting deteriorates when agents are split into three teams on maze maps. Maze maps generally cause a high number of collisions because of the narrow passages and lack of alternative routes due to the acyclicity of the maps. This causes the initial heuristic to be a less precise indicator of the cost of the solution. As a result, the efficiency of exhaustive matching is reduced.

When all agents belong to the same team, the decrease in efficiency is covered up by the sorting. With sorting exhaustive matching, agents of the same team generally do not collide because if this were the case, a goal assignment where the goals of the colliding agents are swapped would have a lower heuristic value causing it to be evaluated earlier.[3]

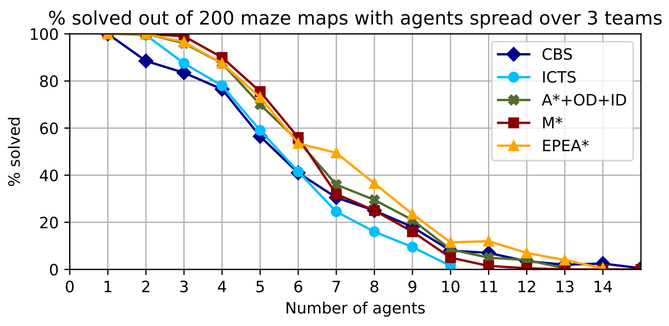
Figure 2d shows that sorting also improves performance on obstacle maps with agents divided over three teams. However, on obstacle maps with all agents in one team, there is very little difference between exhaustive matching with and without sorting, as can be seen in Figure 2c. Because of the openness of obstacle maps, the number of collisions is limited and the lower level ID is able to evaluate a goal assignment very quickly. However, for nine agents there is such a large number of goal assignments that it is not possible to evaluate enough of them to guarantee an optimal solution within the timeout.

The effect of sorting is still visible when looking at the number of evaluated goal assignments. Exhaustive matching with sorting is able to keep the average number of evaluated goal assignments close to one, whereas without sorting it grows exponentially, as indicated by Figure 4. Note that the evaluated goal assignments for normal exhaustive matching are still far below the total number of goal assignments due

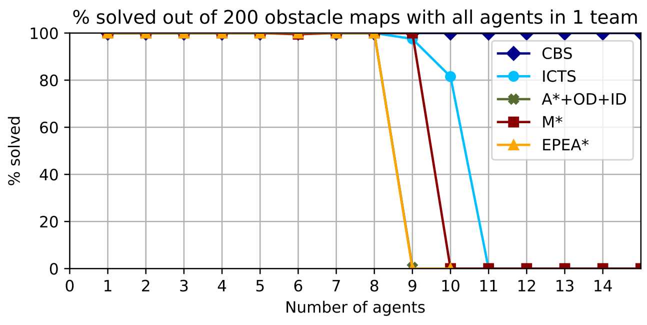
[3]Technically it is still possible for agents to collide, but only if agents are routed through the same passage at the same time or if an agent has to avoid a different collision.



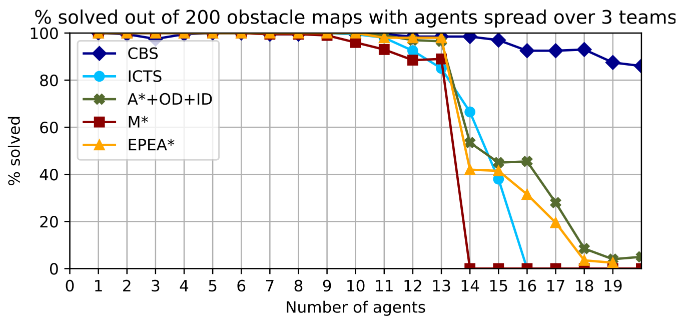
(a) Maze maps with all agents in a single team



(b) Maze maps with agents divided over three teams



(c) Obstacle maps with all agents in a single team



(d) Obstacle maps with agents divided over three teams

Figure 6: Comparison of different MAPFM algorithms on maze and obstacle maps. Lines are labelled with the MAPF algorithm on which the algorithm is based.

to pruning. With 7 agents, exhaustive matching evaluates approximately 11 goal assignments on average, while the total number of goal assignments is $7! = 5040$.

Independence Detection

Figure 2a and 2c show that the addition of ID has no visible effect when all agents are in the same team. This is expected, since matching ID always plans teams together and does not influence the operation of the matching algorithm. On maze maps with agents divided over three teams (Figure 2a), there is also no difference between matching with and without ID.

This is due to the fact that the collisions are the limiting factor on maze maps with three teams rather than the number of goal assignments, as discussed in the previous section. ID reduces the number of goal assignments that have to be evaluated, but since the algorithm is not limited by the number of goal assignments, there is no difference in the number of problems that can be solved.

The only scenario where the effect of ID is visible is on obstacle maps with agents divided over three teams, as can be seen in Figure 2d. Without ID, the algorithm is not able to solve any instances with 14 agents or more. However, ID is still able to solve over 20% of problems with 17 agents.

Comparison with other MAPF algorithms

CBMxM CBMxM is a matching algorithm based on CBM [18]. CBMxM is able to solve a lot more problems within the timeout than the exhaustive matching solvers in most cases. Figure 6a and 6c show that CBMxM can solve all instances when all agents are in the same team. However, with agents spread over three teams on maze maps, the performance is a bit worse than the performance of EPEA*, as can be seen in Figure 6b. Figure 6d shows CBMxM has a much higher performance on obstacle maps with three teams, where it outperforms exhaustive matching solvers.

The high performance of CBMxM is due to the fact that it makes use of min-cost flow, allowing it to solve single-team problems in polynomial time. With a higher number of teams, the benefit of this decreases, which is why it performs worse when agents are divided into three teams.

Exhaustive matching solvers The solvers that are based on exhaustive matching all have a very similar performance. Next to EPEA*, the exhaustive matching solvers are

- The A*+OD+ID MAPFM solver [17].
- The M* MAPFM solver [19].
- The ICTS MAPFM solver [20].

While these solvers take a similar approach, a few differences in performance are still visible. Figure 6a and 6b show that on maze maps, EPEA* is able to maintain a higher solving percentage with a higher number of agents. However, M* and ICTS eventually surpass both EPEA* and A*+OD+ID on instances with all agents in the same team because the solved instances only go to zero for instances with ten or more agents, while A*+OD+ID and EPEA* are unable to solve any instance with nine agents within the timeout. Figure 6c shows the same effect.

The plot in Figure 6d demonstrates the effect of the matching ID of A*+OD+ID and EPEA*. M*, which does not have

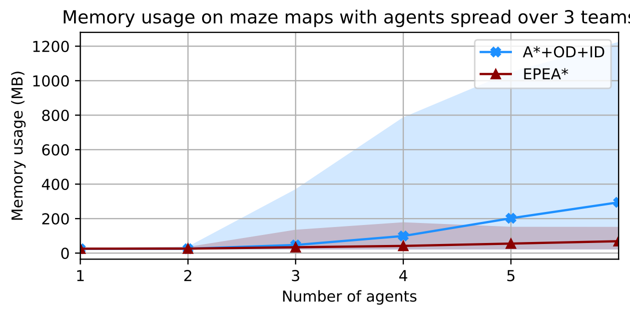


Figure 7: Comparison of the memory that was used for solving maze problems with agents divided over three teams. The lines represent the mean value and the coloured surfaces around them represent the range from the minimum amount to the maximum amount of memory used.

matching ID, is unable to solve problems with 14 agents or more, while EPEA* and A*+OD+ID still can.

Memory

An important benefit of EPEA* is the low amount of memory that is required. An implementation of exhaustive matching can also be used with a limited memory availability because only one instance of EPEA* is run at a time. Figure 7 shows a comparison of the average value of the maximum memory usage of the solver on 200 maze maps for each number of agents. When the solver is not able to solve the problem within a two minute timeout, the maximum memory usage up to the timeout is included in the data. The data shows that EPEA* outperforms A*+OD+ID in terms of memory on maze maps with agents divided over three teams.

6 Conclusion

This paper discusses extending EPEA*, allowing it to solve a problem in which pathfinding is combined with matching, called MAPFM. Heuristic matching and exhaustive matching are introduced as extensions of EPEA*. The results show that exhaustive matching is the faster extension in almost all scenarios.

Next to that, a number of improvements in the form of sorting and independence detection of the exhaustive matching approach were discussed and compared. The results show that sorting is able to improve the performance of exhaustive matching in every case. Independence detection is also able to improve performance on open maps with the agents divided into multiple teams.

When compared to other MAPFM algorithms, the EPEA* based solver described in this paper has a similar performance to the other exhaustive matching solvers, with it performing better in most cases on maze maps. However, CBMxM performed a lot better than the exhaustive matching solvers in most scenarios.

It should be noted that these results were generated on a specific type of map, with only a few parameters being varied. For example, the effect of the map size on the algorithm has not been tested. Theoretical analysis and anecdotal evidence suggests that CBMxM scales badly with an increase in map size, while this does not have a big impact on the performance of EPEA* and other algorithms. This is yet to be confirmed.

7 Future work

The main focus of this research was to extend EPEA* with matching to make it more useful in real-world applications. While the proposed extensions provide a good foundation, there are a number of optimizations that can still be explored.

A possible optimization of the heuristic matching approach is to calculate the min-cost matching when calculating the heuristic instead of directly using the distance to the closest goal for each agent. This prevents the problem of multiple agents being guided towards the same goal. However, this does make the heuristic calculation more complex, which especially affects algorithms with partial expansion such as EPEA*, since node expansions are more common compared to a regular A* solver. Whether this is a useful addition has yet to be determined.

Additionally, improvements could be made to EPEA* with exhaustive matching. The option of caching sub-solutions that are found in the ID of EPEA* has yet to be explored. This increases the memory requirements for the algorithm, but can also prevent unnecessary executions of EPEA*.

It might also be worth looking at improving the pruning of exhaustive matching. When a goal assignment is evaluated where the low-level ID finds an edge conflict in paths of agents that belong to the same team, the evaluation of the matching can be ceased since the goal assignment where the goals assigned to the individual agents are swapped will always lead to a superior solution. Whether this influences performance has yet to be determined. It could be argued that it is not very common for edge conflicts to occur in exhaustive matching with sorting, since the goal assignment with the goals swapped will have a lower or equal heuristic value, causing it to be evaluated earlier.

Next to improving the efficiency of matching EPEA*, its usefulness could also be improved. There are real-world applications that require more flexibility in the problem definition. An example is the routing of trains on shunting yards, where trains have to visit certain waypoints while travelling to the goal where they are cleaned or serviced. An optimal algorithm already exists for A*+ID+OD [21] so it is worth exploring if this can also be adapted to EPEA* because of the similarity of the algorithms.

8 Reproducibility

To facilitate the reproducibility of this research, the Python codebase for the MAPFM algorithms described in this paper has been made publicly available on GitHub.[4] The code is available under the MIT license and readers are encouraged to use or extend the code. The randomly generated benchmark maps and unprocessed results of the benchmarks are also available in this repository.

[4]https://github.com/jaapdejong15/matching-epea-star

Acknowledgements

I wish to express my gratitude to Robbin Baauw, Ivar de Bruin, Jonathan Dönszelmann and Thom van der Woude for the feedback and data that they provided. I also want to thank Jesse Mulderij and Mathijs de Weerdt for supervising this project and giving feedback on this research paper.

References

[1] Wolfgang Honig et al. "Persistent and Robust Execution of MAPF Schedules in Warehouses". In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1125–1131. ISSN: 23773766. DOI: 10.1109/LRA.2019.2894217.

[2] Jesse Mulderij et al. "Train Unit Shunting and Servicing: a Real-Life Application of Multi-Agent Path Finding". In: *arXiv* (2020), pp. 1–14. ISSN: 23318422. arXiv: 2006.10422.

[3] Roni Stern et al. "Multi-agent pathfinding: Definitions, variants, and benchmarks". In: *Proceedings of the 12th International Symposium on Combinatorial Search, SoCS 2019* (2019), pp. 151–158. arXiv: 1906.08291.

[4] Jingjin Yu and Steven M. LaValle. "Structure and intractability of optimal multi-robot path planning on graphs". In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence, AAAI 2013* (2013), pp. 1443–1449.

[5] Guni Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66. ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.006. URL: http://dx.doi.org/10.1016/j.artint.2014.11.006.

[6] Edward Lam et al. "Branch-and-Cut-and-Price for Multi-Agent Pathfinding." In: *IJCAI*. 2019, pp. 1289–1296.

[7] Guni Sharon et al. "The increasing cost tree search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 195 (2013), pp. 470–495. ISSN: 00043702. DOI: 10.1016/j.artint.2012.11.006. URL: http://dx.doi.org/10.1016/j.artint.2012.11.006.

[8] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with optimality bounds". In: *Lecture Notes in Electrical Engineering* 57 LNEE (2013), pp. 167–181. ISSN: 18761100. DOI: 10.1007/978-3-642-33971-4_10.

[9] Trevor Standley. "Finding optimal solutions to cooperative pathfinding problems". In: *Proceedings of the National Conference on Artificial Intelligence* 1 (2010), pp. 173–178.

[10] Ariel Felner et al. "Partial-expansion A* with selective node generation". In: *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012* (2012), pp. 180–181.

[11] Meir Goldenberg et al. "Enhanced partial expansion A*". In: *Journal of Artificial Intelligence Research* 50 (2014), pp. 141–187. ISSN: 10769757. DOI: 10.1613/jair.4171.

[12] Hang Ma and Sven Koenig. "Optimal target assignment and path finding for teams of agents". In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS* (2016), pp. 1144–1152. ISSN: 15582914. arXiv: 1612.05693.

[13] Hang Ma. "Target Assignment and Path Planning for Navigation Tasks with Teams of Agents". PhD thesis. University of Southern California, 2020.

[14] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. "A* with Partial Expansion for large branching factor problems". In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence* Figure 2 (2000), pp. 923–929.

[15] Meir Goldenberg et al. "A* variants for optimal multi-agent pathfinding". In: *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012* (2012), pp. 157–158.

[16] Peter E Hart and J Nils. "Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: 2 (1968), pp. 100–107.

[17] Ivar De Bruin, Jesse Mulderij, and Mathijs De Weerdt. "Solving Multi-Agent Pathfinding with Matching using A * + ID + OD". 2021.

[18] Robbin Baauw, Matthijs De Weerdt, and Jesse Mulderij. "Adapting CBM to optimize the Sum of Costs". 2021.

[19] Jonathan Dönszelmann, Jesse Mulderij, and Mathijs De Weerdt. "Matching in Multi-Agent Pathfinding using M *". 2021.

[20] Thom Van der Woude, Jesse Mulderij, and Mathijs De Weerdt. "Multi-Agent Pathfinding with Matching using Increasing Cost Tree Search". 2021.

[21] Stef Siekman. "Extending A * to solve multi-agent pathfinding problems with waypoints". In: *TUDelft Repository* (2020).

[22] Selmer M. Johnson. "Generation of Permutations by Transposition". In: *Mathematics of Computation* 15.74 (1963), pp. 282–285. ISSN: 00255718. DOI: 10.2307/2004229.

Acronyms

BCP	Branch-and-Cut-and-Price.
CBM	Conflict-Based Min-cost-flow.
CBS	Conflict-Based Search.
EPEA*	Enhanced Partial Expansion A*.
ICTS	Increasing Cost Tree Search.
ID	Independence Detection.
MAPF	Multi-Agent Pathfinding.
MAPFM	Multi-Agent Pathfinding with Matching.
OSF	Operator Selection Function.
PDB	Pattern Database.
PEA*	Partial Expansion A*.
SoC	Sum of Costs.

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 that 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 that 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 8.

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 This part shows that all agents can travel to a branch vertex without collision. First, it will be proven that there is at least one agent that can travel to a branch vertex.

Theorem A.1. *There is always at least one agent that 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. □

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 9), 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.

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.

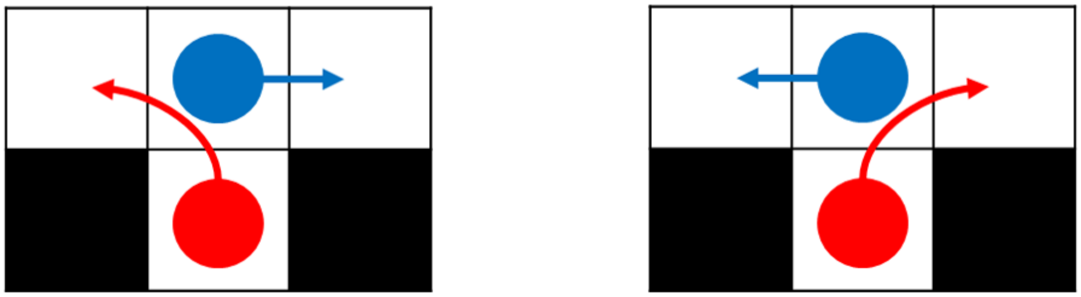


Figure 8: Agents passing each other on three-neighbour nodes

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 in Figure 10). 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 in Figure 10). 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 10). 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 9, 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. □

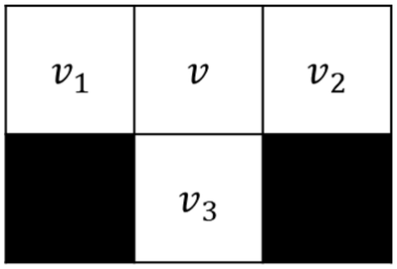


Figure 9: The trivial map with a single branch vertex

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 9). 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 11 shows how all agents move out of the way to diversion vertices to allow the lime agent to pass.

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 shown in Figure 12). However, if v_3 is

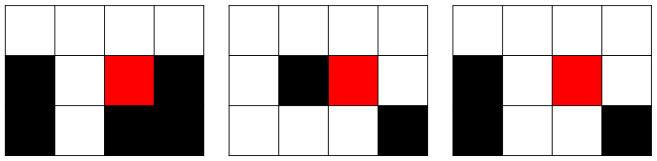


Figure 10: Three different ways of connecting vertices. Red vertices are added to maps.

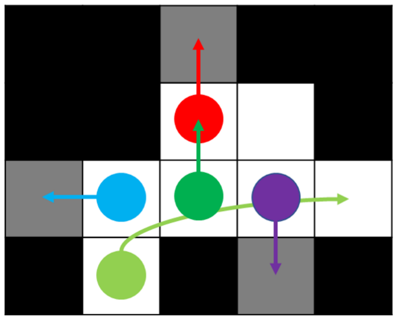


Figure 11: An example of how an agent can pass other agents even if there is no space between branch vertices

another branch vertex, another agent a_k may be on it. Two situations can now occur:

- 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 13

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.*

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

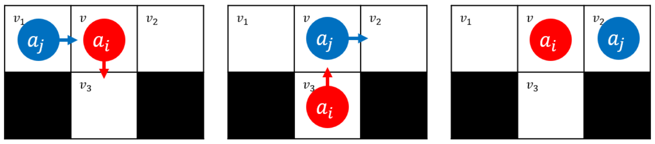


Figure 12: An example of how an agent can pass another agent.

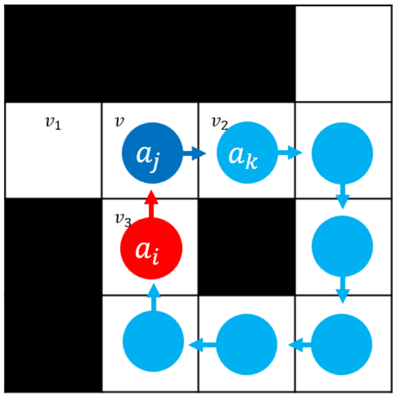


Figure 13: An example of how agents can pass with a single diversion vertex.

agent on its branch vertex and both agents moving back to the swapped branch vertices without collision. This process can be seen in Figure 14. 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 [22].

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 an assignment p of agents to branch squares. □

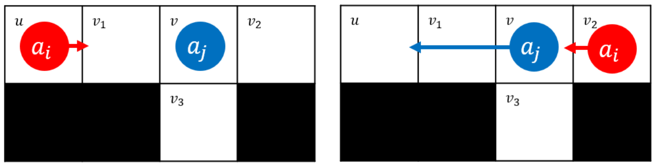


Figure 14: An illustration of two agents swapping by passing each other.

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. □