Adapting CBM to optimize the Sum of Costs

Robbin Baauw*
TU Delft

Abstract

In the Multi-Agent Pathfinding with Matching (*MAPFM*) problem, agents from a team are matched with and routed towards one of their team's goals without colliding with other agents. The sum of path costs of all agents is minimized. In prior works, Conflict Based Min-Cost-Flow (CBM) has been proposed. This algorithm solves a similar problem that instead minimizes the maximum path length. In this paper, an extension upon CBM is presented, called CBMxSOC. It consists of several changes to CBM that allow it to minimize the sum of path costs. CBMxSOC is experimentally compared to other *MAPFM* solvers and is shown to be able to scale to many agents when there are few conflicts between different teams.

1 Introduction

Multi-agent Pathfinding (*MAPF*) deals with finding paths for agents through a graph. Each agent has a corresponding start and end location and two agents can never be on the same vertex at any time step or occupy the same edge in between two time steps[1]. The *Sum of Costs* (SoC), the sum of the path length of each agent, is minimized. The *MAPF* with matching (*MAPFM*) problem is a generalization of *MAPF*, in which both agents and goals are assigned a *team*. Any agent in a team can end on any goal belonging to that team.

An efficient solution to the *MAPFM* problem can be beneficial for solving the Train Unit Shunting and Servicing Problem (TUSS)[2], in which trains are routed through shunting yards to allow maintenance to be performed. Currently, this problem is solved using local heuristics. Therefore, solving instances of the *MAPFM* problem could help train operators to optimize how they make use of their shunting infrastructure by scheduling different types of trains to different types of shunting yards. Other use-cases include autonomous aircraft towing [3] and scheduling warehouse robots [4].

The base *MAPF* problem can be solved in a multitude of ways, for instance using A*-OD+ID[5], M*[6] and CBS[7]. This paper will focus on Conflict Based Search (CBS) and an existing extension upon this, called Conflict Based Min-Cost-Flow (CBM)[4]. CBM solves the target-assignment and pathfinding problem (TAPF) however, in which the *makespan* - the minimum time step at which all agents have reached their goals - is minimized, instead of the SoC which is required by our formulation of MAPFM.

This paper presents two separate extensions of and several improvements on top of CBM which can solve the *MAPFM* problem. The first extension uses Integer Linear Programming (ILP) to solve the MAFPM problem. The second extension uses Goldberg-Tarjan's successive shortest path (SSP) algorithm[8] and is called CBMxSOC.

These contributions will be presented by first formally introducing the exact problem statement and prior works. Then, the aforementioned extensions and improvements upon CBM are proposed. A theoretical discussion on optimality and completeness is held and the algorithm is compared to other algorithms that solve the same problem using a set of experiments. The extensions and several other properties are also experimentally evaluated, after which the reproducibility of this study will be discussed and conclusions will be drawn.

2 Problem Description

This section introduces the base MAPF problem and then formally introduces the problem statement.

2.1 MAPF

Multi-agent Pathfinding[1] (*MAPF*) deals with finding paths through a graph $G = (V, E)$ for k agents $a_1 \dots a_k \in A$. Each agent a_i has a start position s_i and a goal position g_i. In a solution, agent a_i has a corresponding path π_i consisting of t discrete time steps where $\pi_i(1) = s_i$ and $\pi_i(t) = g_i$. At each time step in a path, an agent either moves to a neighbouring position or waits at its current position. Both operations have a cost of 1. Collisions with other agents are forbidden. More specifically, *vertex conflicts*, in which $\pi_i(t) = \pi_j(t)$ and *edge conflicts*, in which both $\pi_i(t) = \pi_j(t+1)$ and $\pi_i(t+1) = \pi_j(t)$, are not allowed for any t and $i \neq j$. The total cost of an agent's path is the time step at which the agent has reached its goal and will remain stationary. The objective is to optimize the *Sum of Costs* (SoC), which is the sum of all agent's path costs: $\sum_{\pi_i \in \pi} |\pi_i|$. This is also commonly called the *Sum of Individual Costs* (SIC).

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

2.2 MAPFM

Multi-agent Pathfinding with Matching (*MAPFM*) is a generalization of the base *MAPF* problem. Instead of a one-to-one mapping between an agent a_i and goal g_i, there are K teams of agents which consist of k_j agents $a^j = \{a_1^j, \ldots, a_i^j\}$ and a set of k_j goals $g^j = \{g_1^j, \ldots, g_i^j\}$ for team j and $i = 1 \ldots k_j$. Each agent a_i^j must move towards one of the goals in g^j without colliding with any other agent. This introduces the concept of finding *matchings* between agents and goals. Similar to the *MAPF* formulation, the Sum of Costs is minimized, and an agent does not disappear once it has reached its goal. This means that an agent can be part of a vertex conflict while standing on its goal.

3 Existing *MAPF* solvers

Multiple algorithms are able to solve *MAPF* problems. These algorithms can generally be assigned to one of two categories[9]:

- **Search-based solvers** are often made to optimize the Sum of Costs objective. Some A* variants use heuristics to search a global search space in which each state represents a placement of agents on vertices [5, 6]. Other search-based solvers use different types of search trees to reduce the search space[7, 10].
- **Reduction-based solvers** are often made to optimize the makespan objective. They reduce the *MAPF* problem into problems that have been readily solved[4, 9]. This reduction is generally done in polynomial time.

3.1 Conflict Based Search

Conflict Based Search (CBS) [7] solves *MAPF* instances in two layers: a high-level solver which searches through a search space of solutions and a low-level solver in which each agent solves an underlying pathfinding problem in a decoupled fashion.

The high-level solver consists of a *Constraint Tree* (CT), which is a binary tree in which each node N contains the following information:

- A set of constraints. These are either *vertex constraints* in the form $\langle v, a_i, t\rangle$, or edge constraints in the form $\langle v_a, v_b, a_i, t\rangle$, where t is the time step at which agent a_i is not allowed to be on vertex v or cross the edge (v_a, v_b) respectively. The root has an empty set of constraints, and each child inherits all its parent's constraints and adds one new constraint.
- A solution. This solution contains paths for each agent. These paths conform to all constraints of N. They may however have conflicts that will have to be resolved in further iterations.
- The total cost. In the *MAPFM* formulation, this cost is the sum of individual path costs.

For each high-level node, low-level paths are found that are consistent with (i.e. do not violate) the given constraints. When either a vertex conflict $\langle a_i, a_j, v, t\rangle$ or edge conflict $\langle a_i, a_j, v_a, v_b, t\rangle$ is detected, two new CT child nodes are created representing newly added constraints for agent a_i and agent a_j. This newly added constraint prevents the conflict by preventing one of the agents from visiting the vertex or edge such that the specific conflict cannot occur anymore. In the next iteration, the currently found solution with the lowest cost is picked to iterate on. This process continues until no further conflicts are detected.

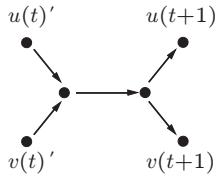


Figure 1: The gadget that denotes the edges (u_t, v_{t+1}) and (v_t, u_{t+1})[13]. This gadgets prevents edge-conflicts between the u and v vertices by only allowing a single unit flow in both directions combined.

In the low-level solver, any pathfinding algorithm can be used. In the original description of CBS[7], A* was used for this purpose. This low-level solver must abide by the constraints of the high-level solver.

3.2 Conflict Based Min-Cost Flow

The Conflict Based Min-Cost Flow (CBM) [4] algorithm solves a variant of *MAPFM* in which the makespan is minimized. In the high-level solver, it modifies CBS by grouping the agents from one team into a single meta-agent [11]. This means that the low-level search is executed per team instead of per agent. High-level conflicts subsequently only happen *between* teams, given that conflict resolution for agents *within* teams happens in the low-level solver.

In the low-level solver, the problem is now modeled as a T step time-expanded network-flow problem $\mathcal{N} = (\mathcal{V}, \mathcal{E})$[12], also called a *time-expanded graph* (TEG):

- Each vertex $v \in V$ is translated to the vertices $v_t^{out} \in \mathcal{V}$ for $t = 0 \ldots T$ and $v_t^{in} \in \mathcal{V}$ for $t = 1 \ldots T$, both having zero demand/supply. The $(s_i)_0^{out}$ and $(g_i)_T^{out}$ nodes for $i = 1 \ldots k$ have unit supply and demand respectively, representing the start and end locations of the agents and thus forcing a matching to be found.
- Each vertex $v \in V$ is translated to the edges $(v_t^{out}, v_{t+1}^{in})$ for $t = 0 \ldots T-1$ (representing a "wait" action, cost 1) and (v_t^{in}, v_t^{out}) for $t = 0 \ldots T$ (representing "being" on vertex v at time t, and subsequently preventing vertex conflicts, cost 0). Both have a unit capacity.
- Each edge $(u, v) \in E$ is translated to a gadget of vertices and edges. First off, the two vertices $w_t, w_t' \in \mathcal{V}$ are created. Then, the five edges $(u_t^{out}, w_t), (v_t^{out}, w_t), (w_t, w_t'), (w_t', u_{t+1}^{in}), (w_t', v_{t+1}^{in}) \in \mathcal{E}$ are created. All edges have a unit capacity, and only (w_t, w_t') has a unit cost. This gadget prevents edge conflicts between nodes u and v at time t, since (w_t, w_t') is traversed for both directions of the (u, v) edge, while having only a unit capacity. This is illustrated by Figure 1.

The resulting network flow problem can be solved using Integer Linear Programming (ILP)[12, 13]. However, CBM proposes a more elaborate way to solve the problem. To start, CBM tries to avoid conflicts on the low level by increasing the edge costs for edges that are visited by other teams. It also uses Goldberg-Tarjan's successive shortest path algorithm[8]. The combination of adjusting edge weights and using a dedicated network flow solver yields significant speed improvements over the basic ILP method.

4 Extensions upon CBM

In this section, changes to CBM are presented that allow it to minimize the *Sum of Costs*. First, two high-level techniques are discussed which determine the maximum time step T for which a SoC can be found. Then, two ways in which the time-expanded graph (TEG) can be adapted to optimize the SoC are described. Lastly, a change that improves the runtime performance of CBMxSOC is introduced.

A term that will be used throughout this and the following sections is *stay-on-goal-edge*. It denotes an edge $(v_t^{out}, v_{t+1}^{in})$ where $v \in g$ (i.e. an edge which represents an agent waiting on goal v between time step t and $t+1$). This term is used since in the *MAPFM* formulation, an agent remains on its goal until all other agents have also reached their goals.

4.1 Time step with optimal SoC

To build the TEG, it is important to know the maximum time step T for which to construct the TEG. This needs to be as low as possible since larger TEGs require more CPU time and memory to build. First off, it is important to note that an optimal makespan solution in time T does not mean an optimal SoC solution exists for a T-step time-expanded graph[14, 15], as is illustrated in Figure 2. There are two approaches that can be used in order to find a T for which there is guaranteed to be an optimal SoC solution.

The approach as described by [14] uses the fact that given a SoC solution of $LB(SoC) + \delta$ cost, one is guaranteed to find this solution in a time-expanded graph of $LB(Mks) + \delta$ layers. Here, $LB(SoC)$ is a lower bound on the sum of costs, which can be computed by computing a sum of shortest paths to any goal belonging to the agent's team without taking conflicts and constraints into account. $LB(Mks)$ is a lower bound on the makespan, which can be computed by computing the longest shortest path to any goal belonging to the agent's team.

This can aid in finding an optimal SoC solution: when a makespan-optimal solution has been found with a SoC of SoC, we define $\delta = SoC - LB(SoC)$ to determine the layer at which an optimal SoC can definitely be found. An adapted version of this algorithm for this instance of the problem is described in Algorithm 1. In this algorithm, SOC_low_level denotes the low-level solver which optimizes the SoC metric for a given team.

A similar approach is used by [15], but it instead tries to find an optimal SoC solution directly, by iteratively asking its underlying SAT solver, which also uses a TEG, to find a SoC solution of $LB(SoC) + \Delta$, and incrementing Δ each iteration until such a solution is found. An adapted version of this algorithm is described in Algorithm 2.

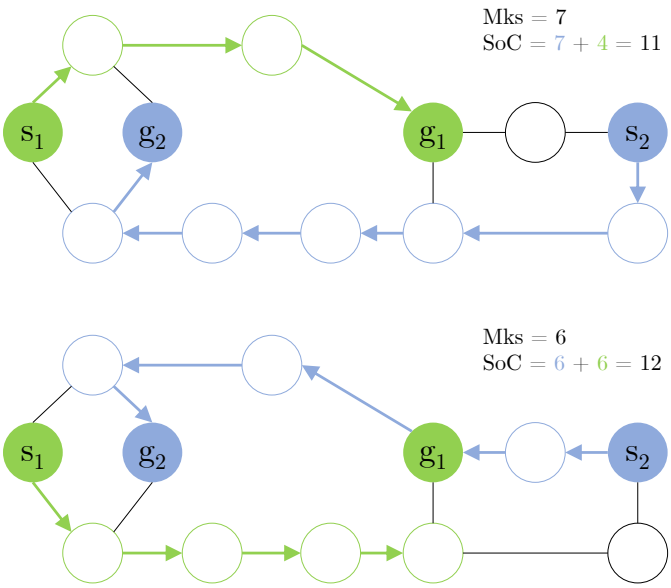


Figure 2: Illustration of a *MAPF* instance with a blue agent and a green agent which shows the difference between optimizing for the SoC (top) and the makespan (bottom). Even though a makespan-optimal solution is found after maximum time step $T = 5$, a SoC optimal solution can only be found in a TEG with $T = 6$. Image adapted from [14].

Algorithm 1: Approach 1 for finding the optimal SoC

$\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$;
$LB(Mks) = max_{i \in A} SP_i$;
$LB(SoC) = \sum_{i \in A} SP_i$;
$\gamma \leftarrow 0$;
while *No solution found* **do**
  $SoC \leftarrow CBM\_low\_level(T = LB(Mks) + \gamma)$;
  $\gamma \leftarrow \gamma + 1$;
end
$\delta \leftarrow SoC - LB(SoC)$;
$SoC\_low\_level(T = LB(Mks) + \delta)$;

Both algorithms provide an optimal SoC solution, however, Algorithm 1 only requires a single SoC pass, whereas Algorithm 2 requires multiple. Therefore, determining which to use depends on the runtime performance of CBM compared to the SoC low-level algorithm. In the case of CBMxSOC, the first approach is used.

4.2 Adapting ILP for SoC

The Integer Linear Programming (ILP) model as described by [4] is solved using standard ILP network flow techniques [13]. The standard cost function that is optimized for the ILP approach is the sum of edge costs for edges with flow. Given a T-step time-expanded network, this would mean the makespan is optimized. However, to optimize the SoC, the path cost should stop increasing when an agent has reached its goal and remains stationary.

This can be done by changing the objective of the model by subtracting the number of time steps for which an agent

Algorithm 2: Approach 2 for finding the optimal SoC

$\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$;
$LB(Mks) = max_{i \in A} SP_i$;
$LB(SoC) = \sum_{i \in A} SP_i$;
$\Delta \leftarrow 0$;
while *No solution found* **do**
  $\mu \leftarrow LB(Mks) + \Delta$;
  $SoC \leftarrow SoC\_low\_level(T = \mu)$;
  if *No solution found or* $SoC > LB(SoC) + \Delta$ **then**
    $\Delta \leftarrow \Delta + 1$;
  end
end

remains on its goal after reaching it. Upon transforming our program into a Mixed-Integer Linear Program this is achieved by adding binary indicator variables i_t^v for $v \in g$ and $t = 0 \dots T-1$ which is 1 when the following properties are true:

- The agent stays on its goal at current time t
- The agent stays on its goal for the time steps $t+1 \dots T$. In other words, the indicator variables for $t+1 \dots T$ are also 1

Formally speaking, we assign

$$i_t^v = i_{t+1}^v \wedge (f((v_t^{out}, v_{t+1}^{in})) = 1)$$

for $t = 0 \dots T-1$ and $i_T^v = 1$ where $f(e)$ is the flow of edge e.

The first part of this statement is to check whether the agent does not leave its goal in the future, thus not subtracting any cost for this t. The second part of this statement is to check whether the agent has flow on the stay-on-goal-edge, thus remaining on its goal. This is illustrated in Figure 3.

The objective of the low-level ILP model will now change from $\sum_{e \in E} c_2(e) \cdot f(e)$ to $\sum_{e \in E} c_2(e) \cdot f(e) - \sum_{v \in g} \sum_{t \in 0 \dots T-1} i_t^v$ where $c_2(e)$ is the cost and $f(e)$ is the flow of edge e. Since the network flow cost of this new objective is equal to the SoC that we want to minimize, this technique finds an optimal solution for the *MAPFM* problem.

4.3 Adapting SSP for SoC

The actual low-level algorithm used by CBM is the successive shortest path algorithm[8, 16], with adjusted edge weights to try to avoid high-level conflicts in the low-level solver. This has been shown to be faster than the ILP algorithm[4].

In order to make the cost of the network flow equal to the SoC to solve the *MAPFM* problem, however, edges should have a zero cost when an agent has reached its goal and remains stationary. This decision problem cannot be modeled with the standard network flow definition. Therefore this needs to be done differently: 'push' agents towards a goal as soon as possible, without increasing the path costs for other agents. For this to solve the problem, the following conditions need to hold:

- The cost of stay-on-goal-edges should be lower than other edges. This causes agents to prefer to go to goal nodes at the lowest t possible.

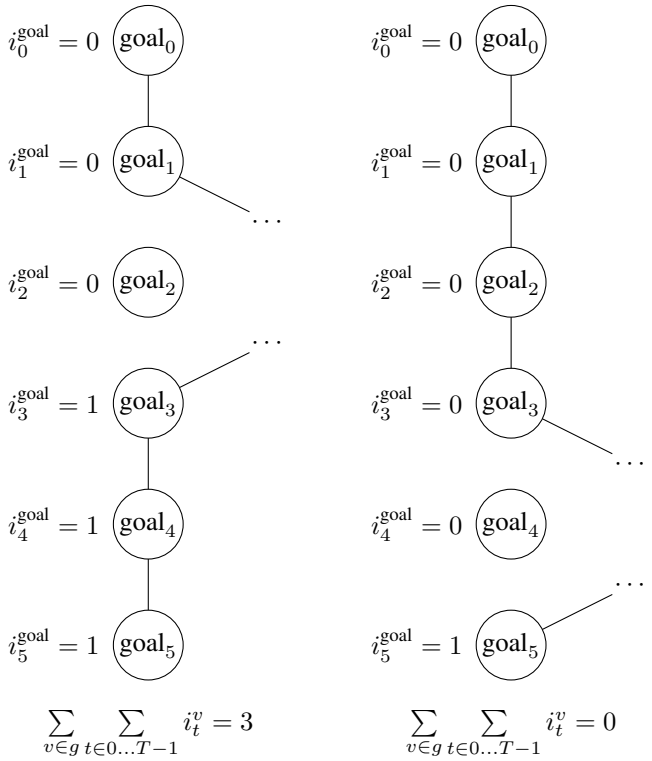


Figure 3: An illustration of the values of binary indicator variables for different time steps at which the goal is reached. In the left illustration, an agent starts on a goal but moves away from this goal at $t = 1$, thus the binary indicators for $t = [0, 1, 2]$ are 0. After after $t = 3$, however, the agent remains on its goal and thus these binary indicators will have a value of 1, yielding a binary indicator sum of 2 (note that the sum of binary indicators is computed for $t = 0 \dots T-1$). In the right illustration, an agent remains on the goal until $t = 3$, and only returns to the goal at $t = 5$ which thus yields a binary indicator sum of 0.

- The cost of stay-on-goal-edges should decrease over time, up until the maximum time T of the TEG. This causes agents to prefer to go to their final goal node sooner instead of waiting on another goal node and moving at a later moment (and subsequently increasing the SoC).
- The cost of a non stay-on-goal-edge should be higher than $(k_j - 1) \cdot \sum_{t \in 0 \dots T-1} C((\text{goal}_t, \text{goal}_{t+1}))$ for team j and edge cost $C(e)$. In other words, the cost of a non stay-on-goal-edge should be higher than the combined stay-on-goal-edge costs of the other agents within team j. This high cost for non stay-on-goal-edges is required so that agents do not collectively leave their goals to help one other agent obtain a smaller cost, while collectively adding more path length.

Given that k_j denotes the number of agents in team j, a distribution of edge-costs for which this holds is as follows:

- Stay-on-goal-edge at time t: $T - t - 1$
- Non stay-on-goal-edges: $k_j \cdot \sum_{t=0}^{T-1} t$

This is illustrated in Figure 4.

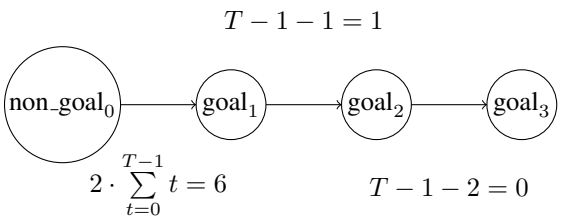


Figure 4: Example path costs for the path of a single agent, with $T = 3$ and two agents in the team of this agent. The first edge at $t = 0$ is a non stay-on-goal-edge, and thus incurs the high cost. The subsequent two edges at $t = 1$ and $t = 2$ are stay-on-goal-edges and thus their cost is reduced.

Since agents are pushed to their goals as fast as possible and incurring extra path costs is penalized, the NF cost of larger sums of paths is likely always higher than the NF cost of smaller sums of paths. Thus, the SSP technique is likely optimal. It is complete given that only the costs in the NF graph have been changed with respect to CBM [4], which is proven complete.

4.4 Reusing TEG

In both Algorithm 1 and Algorithm 2, new T-step time-expanded graphs are used for each T and for each conflict solved. Given the fact that both the number of network flow vertices $|\mathcal{V}|$ and edges $|\mathcal{E}|$ grow with respect to both T and the size of the grid, building this network from scratch each time it is used is very computationally expensive.

Instead, this TEG can be reused by temporarily updating edge capacities and costs and resetting these the next time the graph is used. The following cases need to be taken into account:

- **Smaller** T: it could be the case that a TEG graph has been used for a higher t than for which is currently being solved. This means the edge capacity of v_{out} nodes for $t = T$ should be assigned a 0.
- **Constraints**: in order to conform to the constraints from the high-level, edge capacities of edges that denote illegal movements should be equal to 0.
- **CBM edge weights**: when searching for a makespan-optimal solution, some edge costs are also zeroed in order to reduce the number of conflicts.

Keeping track of and reverting these updated edge capacities and costs is not very expensive and can thus reduce the runtime significantly, as is described by Section 5.2.

5 Experimental Setup and Results

In this section, multiple experimental results are presented. The first experiment is a comparison between ILP and SSP. The second experiment consists of a comparison between reusing the TEG and creating one from scratch each time. Then, some limitations and strengths of CBMxSOC are shown and to conclude, a comparison to other *MAPFM* solvers is made. All generated maps can be found in the repository as described by Section 6.

Experiments one through three were run on a 3.7 GHz AMD Ryzen 9 5900X PC with 32 GB of RAM. The last experiment was run on a 2 GHz Intel Xeon E5-2683 with 8 GB of RAM. All experiments were run using a single-threaded Python 3.9 application. In order to solve the ILP problem, the ILP solver Gurobi (https://gurobi.com) was used. The SSP problem was solved using a fork of Google's OR-tools (https://developers.google.com/optimization/flow/mincostflow). This fork simply allows for adjusting edge weights and capacities, which is required for reusing the TEG.

A general note for each experiment is that the difficulty of the generated maps, which were generated using the generator as described by [17], varies from map to map, even with similar numbers of agents, teams, vertices, and walls. The distribution of walls and agents has a very large influence on the runtime performance of CBMxSOC due to the fact that the number of conflicts is significantly different for each generated map. This means that many results presented will have large standard deviations which need to be taken into account when viewing the graph.

5.1 Experiment one: Comparison between ILP and SSP

In this test, the approaches described in Section 4.2 and 4.3 are compared to each other in terms of created network flow nodes and resulting runtime. It is to be expected that the SSP approach has a better runtime performance due to the usage of a purpose-built algorithm to solve the min-cost max-flow problem.

For the experiment, 4000 dense random maps consisting of two teams with three agents have been generated. In each of the maps, 75% of the vertices are walls. Because each maze has a different distribution of walls and agents and differs in map size, the resulting number of conflicts also differs widely per map. Each algorithm has been run on each of these maps with a time limit of 1200 seconds, and the number of vertices of the map is compared to the runtime performance.

Results from Figure 5 show that SSP outperforms ILP by a large margin. These results also show that SSP scales very slowly with the map size. However, the ILP model increases in size very fast, which results in slower runtimes for larger maps.

5.2 Experiment two: reusing TEG

This experiment aims to show the effectiveness of reusing and expanding the TEG instead of rebuilding it each time it is used. This was tested using the SSP implementation, with a new random set of 1000 generated mazes with a size of 20x20 and a different number of agents spread over three teams. As in the previous experiment, 75% of the vertices are walls. The intent of this test is to show the effect of reusing the TEG on the number of created network flow nodes and the number of conflicts and to relate that to the runtime performance.

The results in Figure 6 show a large difference in runtime performance. This is also evident in the number of maps that the algorithms were able to solve. However, even though

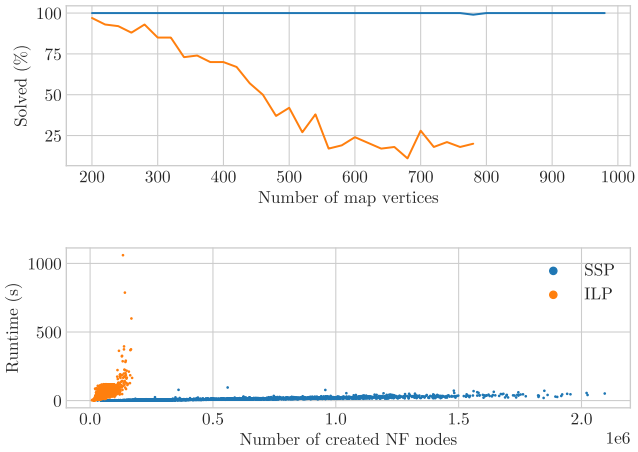


Figure 5: Results comparing the performance of the SSP implementation with the performance of the ILP implementation. The top figure plots the percentage of solved maps against the number of vertices in the given map. The bottom figure plots the runtime of maps that were solved by the algorithms against the number of created network flow nodes.

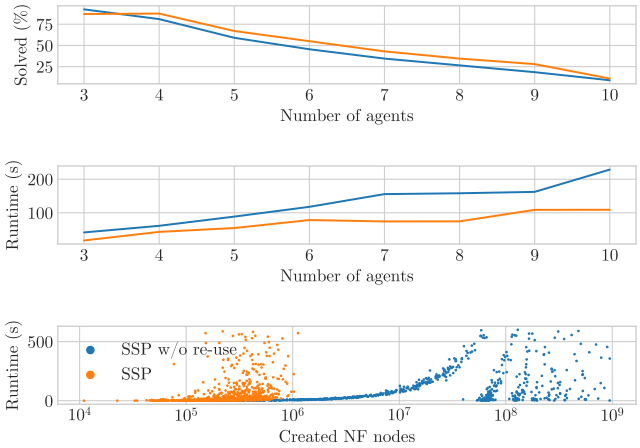


Figure 6: Results comparing the performance of the reusing the Time Expanded Graph versus not reusing it. The top two figures show the percentage of maps that could be solved and their average runtime relative to the number of agents. The bottom figure (note the logarithmic x-axis) shows the number of created network flow nodes with respect to the runtime of the program.

reusing the TEG does make a significant difference, the difference in maps that could be solved within the given time limit is not extremely large, due to the fact that the number of times the low-level search has to be executed grows significantly.

The bottom figure also shows an interesting statistic, namely that indeed many more nodes are created when not reusing the TEG. However, the runtime can still be equal when there a difference in the number of nodes created. This is due to the fact that the amount of conflicts that were encountered affects the runtime performance more significantly. Besides, the only data points that are shown in the bottom figure are the ones in which the map was solved within the 600-second timeout. Therefore, a survivorship bias needs to be taken into account when interpreting this graph.

5.3 Experiment three: Agent and map size limit

As seen in previous experiments, the map size does not exponentially affect the runtime performance of CBMxSOC. The same goes for the number of agents when there is only one team since there will be no conflicts in the high-level solver. This experiment will test the limits of these values, to emulate conflict-free maps.

First, the map size limit was tested by creating maps consisting of eight agents, one team, and increasing the map size from 10x10 to continuously larger maps to check at which point CBMxSOC cannot create the network flow graph within the given time limit of 120 seconds. The results in Figure 7 show that the runtime performance does indeed decrease as the map size becomes larger, and this linear scaling suggests that solving maps with tens of thousands of nodes will take a significant amount of time to solve, especially when many conflicts are involved.

Next, the agent limit was tested by increasing the number of agents on an open 20x20 map, again with one team. This tests whether the runtime of the algorithm increases if the number of agents grows when there are no high-level conflicts. Similarly, a time limit of 120 seconds was in place. The results in Figure 8 show that an increasing number of agents in one team does increase the runtime of the algorithm but in very low amounts. This is due to the fact that the SSP algorithm runs in polynomial time.

5.4 Experiment four: Comparison with other MAPFM solvers

In this experiment, CBMxSOC is compared to other algorithms that can solve the *MAPFM* problem. These algorithms [17–20] were simultaneously developed by peers and thus follow the same problem definition. These algorithms are compared by the runtime performance and the number of solved maps as a function of the number of agents.

It is important to note that even though all algorithms are implemented in single-threaded Python 3.9 code, some implementations may contain more optimizations in the form of caching for example. Given the fact that CBMxSOC uses Google's OR-tools and thus C++, it will likely have a performance advantage over a native Python implementation of an SSP solver.

The results as presented in Figure 9 are based on experiments on two types of maps: a dense maze of which 75% of the vertices consist of walls and a map in 25% of the vertices are walls. On the maps, an increasing number of uniformly distributed agents are placed, in either one or three teams. The graphs are generated using the data gathered by [19] and show interesting characteristics of CBMxSOC with respect to the other algorithms.

However, it is important to note that some of these results have a large standard deviation, as shown by the confidence interval on the graphs. There are two main reasons for this. First off, if few maps can be solved within the given time

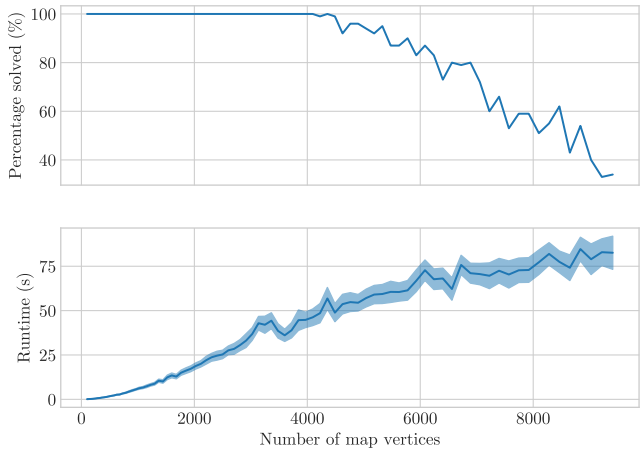


Figure 7: Results showing the runtime performance of CBMxSOC as the map size increases. A 95% confidence interval is also drawn.

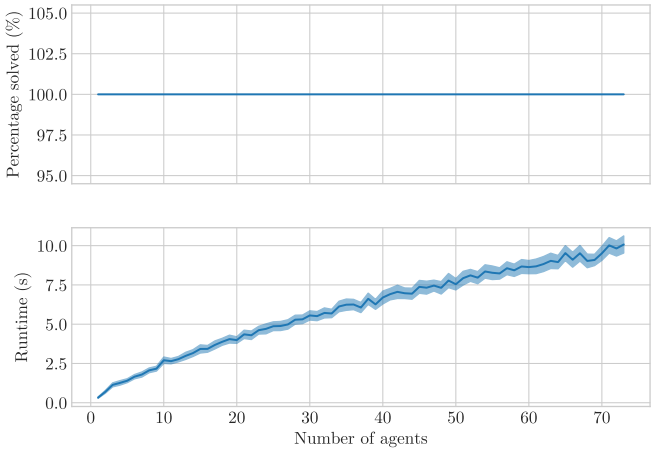


Figure 8: Results showing the runtime performance of CBMxSOC as the number of agents increases. A 95% confidence interval is also drawn.

limit, there are fewer data points that are used to determine the runtime. Fewer data points result in larger deviations and survivorship bias, i.e. only the data points of agents that finished within 120 seconds are shown. Another reason for this large deviation is the fact that the number of agents is not a perfect indicator for the runtime performance of the algorithm. The position of agents and the structure of the map determine this. However, the number of agents is a metric that can easily be influenced when generating maps. Besides, it does show a general increase in runtime for all algorithms when the number of agents increases.

The first finding is that CBMxSOC has a start-up time that is irrespective of the number of conflicts. This time is needed to build the time-expanded graph. The result of this is that CBMxSOC is slower on simple-to-solve maps with fewer conflicts than the other algorithms.

Another finding is that there is a clear difference in runtime between maps with one and three teams for CBMxSOC. CBMxSOC performs well when few conflicts occur. With only one team and thus no conflicts to be solved by the high-level solver, CBMxSOC clearly performs best since it can solve this specific case in polynomial time due to the use of the SSP algorithm. The other algorithms have a steep drop at a certain number of agents, simply due to the fact that the number of possible matchings between agents and goals becomes too large.

The performance of CBMxSOC is similar compared to the other algorithms when there are many conflicts. However, CBMxSOC does not actively try to avoid paths that have already been taken by agents in other teams. In other words, CBMxSOC has no conflict avoidance in its low-level solver. Adding this could reduce the number of high-level conflicts that occur and subsequently reduce the runtime of CBMxSOC.

A caveat here not shown by the graphs is the fact that CBMxSOC is not able to run on very large maps in a reasonable amount of time, as shown in Section 5.3. Other algorithms are able to do solve large maps, since their runtime performance does not scale with map size in the same way that CBMxSOC does.

6 Reproducibility

Special attention has been paid to the reproducibility of this work. In order to achieve this, all the source code of CBMxSOC and the benchmarks used to experimentally evaluate it are publicly available on Github at https://github.com/RobbinBaauw/CBMxSOC. The source code to experimentally compare CBMxSOC to other MAPFM solvers is publicly available on Github at https://github.com/jonay2000/research-project.

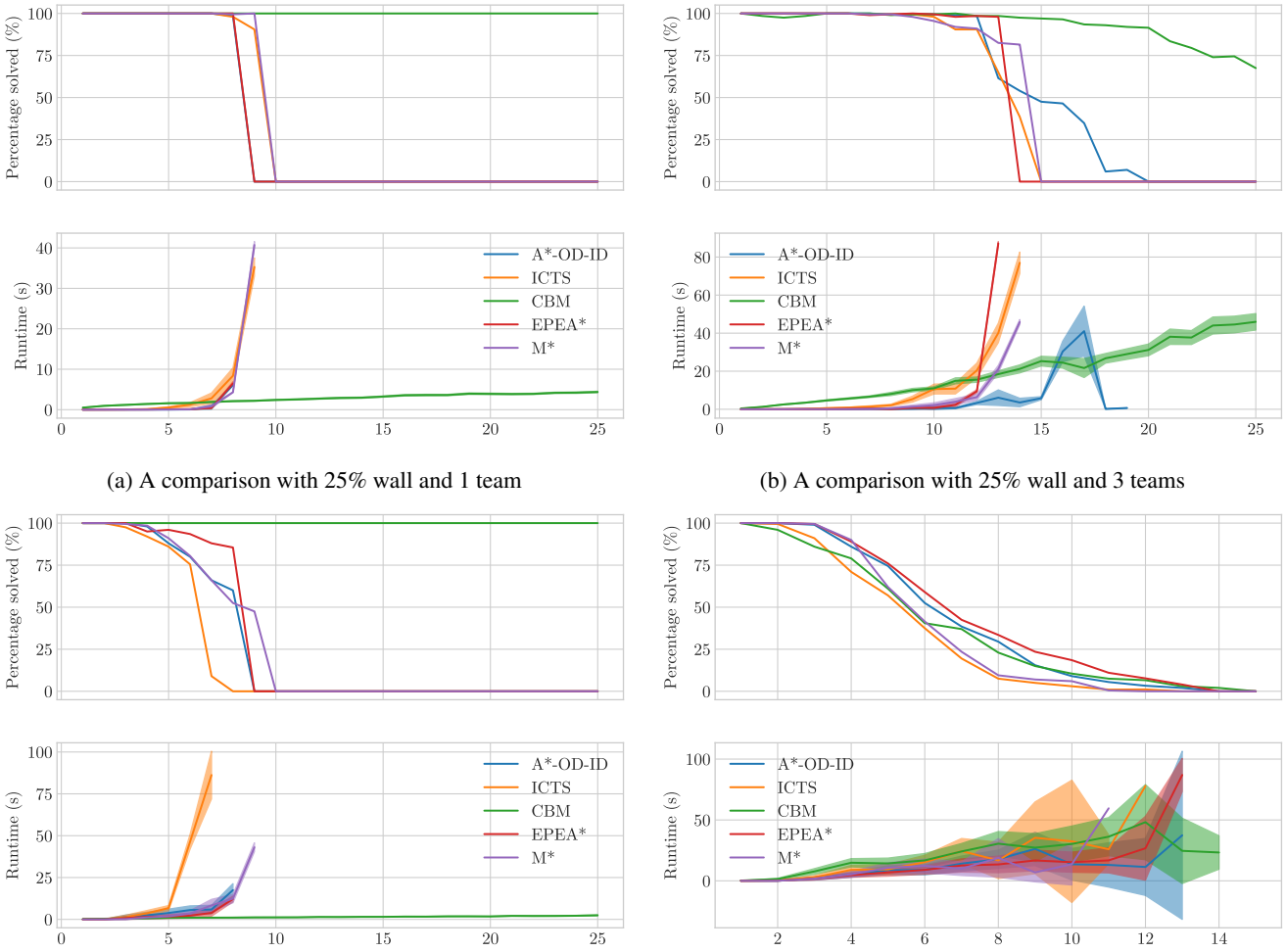
7 Conclusions

In this paper, a generalization of the Multi-Agent Pathfinding (*MAPF*) was introduced, named Multi-Agent Pathfinding with Matching (*MAPFM*). Two extensions of CBM are presented that can solve this problem, using Integer Linear Programming (ILP) and the Successive Shortest Path (SSP) algorithm. These two extensions are experimentally compared to each other. A performance improvement regarding reusing the time-expanded graph is also presented and compared with not reusing this graph.

Experimental results show that the SSP extension, called CBMxSOC, has a better runtime performance than the ILP extension. The SSP extension adjusts edge weights such that the Sum of Costs is optimized instead of the makespan. Besides this, reusing the time-expanded graph has also been experimentally shown to make it possible to solve more maps and solve them faster than not reusing this graph. CBMxSOC is also compared to other *MAPFM* solvers and is shown to outperform them when there are few conflicts between teams and perform similarly when there are many conflicts.

8 Future work

This work answers several questions regarding using CBS and CBM to solve the *MAPFM* problem. It shows that using a TEG to solve the problem is very promising and opens



(a) A comparison with 25% wall and 1 team

(b) A comparison with 25% wall and 3 teams

(c) A comparison with 75% wall and 1 team

(d) A comparison with 75% wall and 3 teams

Figure 9: Results comparing the performance of the different algorithms that can solve the *MAPFM* problem. For each subfigure, the numbers of agents are compared to the percentage of solved maps and the runtime in seconds which is plotted with a 95% confidence interval.

up possibilities for future work. This section aims to describe some future questions and propose possible answers to these questions.

8.1 Reducing conflicts

As described in Section 5.4, CBMxSOC is slow when it comes to handling many conflicts between teams, since the low-level SoC search of CBMxSOC does not actively avoid paths that have already been taken by agents in other teams. In future works, a suitable extension to provide low-level conflict avoidance can be found, for example by increasing edge weights for edges that have already been visited by other teams.

8.2 Adapting MDD-SAT

As shown by the changes to the ILP technique compared to the changes in the SSP technique, an ILP solver is significantly more flexible. Therefore, a direct SAT-based approach as used by [9] may make more sense than combining a search-based solver and a reduction-based solver. Matching can likely be added relatively easily to this approach by making a TEG for each team instead of each agent, and including all goals in V_j for team j instead of just one goal in V_i for agent i. This looks like a promising approach in terms of runtime performance.

8.3 Disappearing agents

If the problem statement were slightly adapted to make agents disappear once they have reached their goal, a simpler adaptation of CBM can be used, similar to the solution described by [12]. Solving this adapted problem can simply be done by adding extra goal sinks and connecting goal nodes at each t to this node. Proving optimality of this algorithm is likely trivial since the network flow cost is equal to the SoC metric. A proof-of-concept version of this algorithm is already implemented in the Github repository mentioned by Section 6. Initial testing shows that some maps can be solved much faster, as fewer conflicts have to be taken into account.

References

[1] 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.

[2] 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.

[3] R. Morris, C. Pasareanu, K. Luckow, W. Malik, H. Ma, T. K. S. Kumar, and S. Koenig, "Planning, scheduling and monitoring for airport surface operations," in *AAAI Workshop: Planning for Hybrid Systems*, 2016.

[4] H. Ma and S. Koenig, *Optimal target assignment and path finding for teams of agents*, 2016. arXiv: 1612.05693 [cs.AI].

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

[6] 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.

[7] 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.

[8] A. Goldberg and R. Tarjan, "Solving minimum-cost flow problems by successive approximation," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87, New York, New York, USA: Association for Computing Machinery, 1987, pp. 7–18, ISBN: 0897912217. DOI: 10.1145/28395.28397.

[9] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "Efficient sat approach to multi-agent path finding under the sum of costs objective," in *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ser. ECAI'16, The Hague, The Netherlands: IOS Press, 2016, pp. 810–818, ISBN: 9781614996712. DOI: 10.3233/978-1-61499-672-9-810.

[10] 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, ISSN: 0004-3702. DOI: 10.1016/j.artint.2012.11.006.

[11] 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.

[12] J. Yu and S. M. LaValle, "Multi-agent path planning and network flow," in *Algorithmic Foundations of Robotics X*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 157–173, ISBN: 978-3-642-36279-8.

[13] J. Yu and S. M. LaValle, "Planning optimal paths for multiple robots on graphs," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 3612–3617. DOI: 10.1109/ICRA.2013.6631084.

[14] R. Barták and J. Svancara, "On sat-based approaches for multi-agent path finding with the sum-of-costs objective," in *SOCS*, 2019.

[15] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "Efficient sat approach to multi-agent path finding under the sum of costs objective," Sep. 2016. DOI: 10.3233/978-1-61499-672-9-810.

[16] U. Bünnagel, B. Korte, and J. Vygen, "Efficient implementation of the goldberg–tarjan minimum-cost flow algorithm," *Optimization Methods and Software*, vol. 10, no. 2, pp. 157–174, 1998. DOI: 10.1080/10556789808805709.

[17] I. de Bruin, *Solving Multi-Agent Pathfinding with Matching using A*+ID+OD*, Jun. 2021.

[18] J. de Jong, *Multi-Agent Path Finding with Matching using Enhanced Partial Expansion A**, Jun. 2021.

[19] J. Donszelmann, *Matching in Multi-Agent Pathfinding using M**, Jun. 2021.

[20] T. van der Woude, *Multi-Agent Path Finding with Matching using Increasing Cost Tree Search*, Jun. 2021.