Improved Anonymous Multi Agent Path Finding Algorithm

Zain Alabedeen Ali[1], **Konstantin Yakovlev**[2, 3]
[1] Moscow Institute of Physics and Technology, Moscow, Russia
[2] Federal Research Center for Computer Science and Control of Russian Academy of Sciences, Moscow, Russia
[3] AIRI, Moscow, Russia
ali.za@phystech.edu, yakovlev@isa.ru

arXiv:2312.10572v1 [cs.AI] 17 Dec 2023

Abstract

We consider an Anonymous Multi-Agent Path-Finding (AMAPF) problem where the set of agents is confined to a graph, a set of goal vertices is given and each of these vertices has to be reached by some agent. The problem is to find an assignment of the goals to the agents as well as the collision-free paths, and we are interested in finding the solution with the optimal makespan. A well-established approach to solve this problem is to reduce it to a special type of a graph search problem, i.e. to the problem of finding a maximum flow on an auxiliary graph induced by the input one. The size of the former graph may be very large and the search on it may become a bottleneck. To this end, we suggest a specific search algorithm that leverages the idea of exploring the search space not through considering separate search states but rather bulks of them simultaneously. That is, we implicitly compress, store and expand bulks of the search states as single states, which results in high reduction in runtime and memory. Empirically, the resultant AMAPF solver demonstrates superior performance compared to the state-of-the-art competitor and is able to solve all publicly available MAPF instances from the well-known MovingAI benchmark in less than 30 seconds.

Introduction

Multi-Agent Path Finding (MAPF) problem is the problem which generally asks to find a set of collision-free paths for a set of agents that operate in a shared environment and have to reach predefined goal locations from the current (start) ones. MAPF has many applications including automated warehouses, autonomous vehicles, video games and is being widely studied in the literature. Depending on the application, many variants of MAPF have been proposed (Stern et al. 2019b) and numerous solutions have been already presented. One variant is the Anonymous MAPF (AMAPF). In AMAPF the agents are interchangeable and each agent may be assigned to any goal, assuming that in the end all goal locations will be reached (in case the number of goal locations is smaller or equal to the number of agents) or each agent will arrive to one goal location (otherwise). This problem naturally arises in such environments where the tasks can be performed by any agent, e.g., think of the identical robots that carry packages/inventory-pods in automated warehouses.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this work we are interested in solving AMAPF problem optimally w.r.t *makespan* cost function, which is the arrival time of the last agent. That is, our task is to find a solution where the last agent arrives at its goal location as early as possible. State-of-the-art optimal AMAPF solvers are reduction-based, i.e. the initial problem is reduced to another one and the latter is solved with the off-shelf solvers. In case of AMAPF the reduction is the following. Based on the input graph another one is constructed and then a maximum flow problem on this auxiliary graph is formulated and solved. The latter can be interpreted as finding several paths (subject to certain constraints) on the reduced network. The major bottleneck here is that the size of the network is much larger, both in the number of vertices and edges, compared to the initial AMAPF graph and finding paths on it is burdensome. Moreover, the AMAPF reduction scheme in general assumes that numerous networks may be consecutively constructed (each one being larger than the previous one) and the search should be repeated.

To this end, we present an improved optimal AMAPF solver that follows the reduction-to-the-flow-problem approach but utilizes a novel search method to find paths on the (flow) networks. The crux of this search method is the concept of bulk states and implicit expansions. In brief, instead of generating and expanding numerous search states we compress them into the bulks that form a sequence, exploiting the special structure of the underlying network, and explicitly store in the search tree only the certain representatives of those bulks (while implicitly reasoning about all other states in the bulk). On the theoretical side we show that our search method, which we dub Bulk Search, is complete. On the practical side we compare our improved AMAPF solver, that utilizes Bulk Search, with the state-of-the-art optimal AMAPF solver and show that our algorithm notably scales better to large maps (due to significantly lower number of expansions when finding the paths on the flow networks) and outperforms the competitor on all maps of the well-known MAPF benchmark from (Stern et al. 2019a).

Related Works

In conventional MAPF formulation (Stern et al. 2019a) a set of agents and their start and goal locations are given

as well the specification where each agent starts and what goal it should reach. Even when both the time in discretized into the time steps and workspace is discretized to a graph (which are the two default assumptions in MAPF), obtaining an optimal solution w.r.t. one the most widely-used objectives, e.g. makespan or sum-of-costs, is known to be NP-Hard problem (Yu and LaValle 2013b). Surprisingly, the AMAPF problem, which is a combined problem of both MAPF and goals assignment, can be optimally solved w.r.t. makespan (but not sum-of-costs) in polynomial time (Yu and LaValle 2013a). The seminal method, introduced in (Yu and LaValle 2013a), is based on the reduction of AMAPF to a series of specific graph-search problems, i.e. the problems of finding a maximum flow (Ford Jr and Fulkerson 2015) on a graph of special structure (network) induced by the inpunt MAPF graph. For sum-of-costs objective an adaptation of the seminal MAPF solver, Conflict-Based Search (CBS) (Sharon et al. 2015), was suggested in (Hönig et al. 2018). Indeed, this algorithm is not polynomial. Suboptimal AMAPF was studied in (Okumura and Défago 2022) and several computationally efficient algorithms were proposed in this work which were empirically shown to provide high-quality solutions (however no bound on sub-optimality is theoretically guaranteed). A variant of the AMAPF problem with some additional practically-inspired assumptions, i.e. that the number of goals exceeds the number of agents and thus agents have to move to the new goals upon completing the current ones, was studied in (Nguyen et al. 2017) and solved using the Answer Set Programming (ASP).

More involved variants of AMAPF were studied in (Ma and Koenig 2016; Barták, Ivanová, and Švancara 2021). There it was assumed that the agents are partitioned into the teams (colors) and each team is assigned a set of interchangeable targets (of the same color). In (Ma and Koenig 2016) a combination of CBS and min-cost max-flow algorithm (Ford Jr and Fulkerson 2015) was suggested to solve a problem. In (Barták, Ivanová, and Švancara 2021) several solvers that utilize reduction to SAT were proposed. Indeed, AMAPF can be viewed as a special instantiation of the colored MAPF problem (i.e. when there exists only one team of agents of the same color as all the goals).

Among the other problems that are closely related to AMAPF one can name Lifelong MAPF (LMAPF) (Li et al. 2021) and Multi-agent Pickup and Delivery (MAPD) (Ma et al. 2017). These MAPF variants assume that the agents continuously operate in the environment reaching the specified goals (associated with certain pickup-and-delivery tasks in case of MAPD). However the assignments of goals (tasks) to agents is commonly assumed to be realized by an external procedure and, thus, the assignment sub-problem is not typically considered as a part of a LMAPF/MAPD problem. However the works that consider a combined problem, indeed, exist (Chen et al. 2021; Xu et al. 2022).

Finally, there exist works that study AMAPF in continuous domains, i.e. not assuming that the agents are confined to a given graph but rather allowing them to freely move in the (geometric) workspace (Adler et al. 2015; Solovey and Halperin 2016).

Problem Statement

We follow a classical approach (Stern et al. 2019b) to define the problem under investigation – AMAPF. We consider a graph $G = (V, E)$, whose vertices correspond to the locations in the environment and edges – to the transitions between them. k agents are confined to this graph, i.e., initially each agent occupies a (distinct) vertex – s_i, the start vertex, and at each time step of the discretized timeline it can either wait in its current vertex or move to an adjacent one. The duration of both types of actions (move or wait) is 1 time step. k goal vertices, $g_1, ..., g_k$, are also distinguished and it is assumed that any agent can reach any goal, i.e. there is no pre-defined assignment of agents to the goals.

A plan for an agent, $\pi(s, g)$, is a sequence of (move/wait) actions, s.t. it begins at vertex s, ends at vertex g and each action in a plan starts where the the previous action ends. The cost of the plan is the time step by which g is reached. Two plans are said to contain a vertex (similarly – edge) conflict if the agents following them occupy the the same vertex (use the same edge) at the same time step.

The problem now is to find a set of plans $\Pi = \{\pi_1, ..., \pi_k\}$, s.t. (1) each pair of plans is conflict-free and (2) all goals are reached. This problem is, in essence, a combination of the assignment problem, where one needs to decide which agent goes to which goal, and the (multi-agent) pathfinding problem, where one need to construct a set of the conflict-free plans.

We consider the following cost objective: $makespan(\Pi) = \max_{i \in 1,...,k}(cost(\pi_i))$, where $cost(\pi)$ is the cost of the individual plan (i.e. the earliest time step when an agent reaches a goal vertex). In this work, we are interested in obtaining makespan-optimal solutions of the problem at hand (AMAPF).

Background

Network Flow

Generally, network flow problem might come in different flavours, see (Ahuja, Magnanti, and Orlin 1995) for an overview. Here we focus on a specific variant of the problem needed for solving AMAPF problems.

A network is a tuple $N = (G, cap, s, g)$, where $G = (V, E)$ is a directed graph, $cap : E \rightarrow Z^+$ is the mapping defining the capacities of the edges, $s \in V$ is the source vertex and $g \in V$ is the sink vertex. For a vertex $v \in V$, let $\sigma^+(v)$ (resp. $\sigma^-(v)$) denote the set of edges of G going to (resp. leaving) v. A feasible s, g-flow on the network is a mapping $f : E \rightarrow Z^+$ that satisfies three types of constraints: edge capacity constraints,

$$\forall e \in E, f(e) \leq cap(e), \tag{1}$$

the flow conservation constraints at non terminal vertices,

$$\forall v \in V \setminus \{s, g\}, \sum_{e \in \sigma^+(v)} f(e) - \sum_{e \in \sigma^-(v)} f(e) = 0, \tag{2}$$

and the flow conservation constraints at terminal vertices,

$$F(f) = \sum_{e \in \sigma^-(s)} f(e) = \sum_{e \in \sigma^+(g)} f(e) \tag{3}$$

The quantity $F(f)$ is called the *value* of the flow f. Another interpretation of the flow as defined above is that the flow is a set of s-g paths in G (possibly overlapping or even duplicating), where each path caries a unit of flow from s to g, such that the sum of units passing through any edge does not exceed its capacity.

The standard single-commodity maximum flow problem asks the question: given a network N, what is the maximum $F(f)$ that can be pushed through the network? Alternatively, find a set of s-g paths that carry the maximum units of flow through the network.

From AMAPF to Network Flow

In (Yu and LaValle 2013b), the authors reduced the T-steps AMAPF problem, i.e. the one which allows any agent to do at most T actions, to a maximum flow (MF) problem. In specific, it was proved that a T-steps AMAPF problem has a solution *iff* the reduced MF problem has a flow equal to the number of agents. The makespan for an AMAPF instance can therefore be found by finding the smallest T such that T-steps AMAPF instance has a solution. We now explain the suggested reduction with a slight modification suggested by (Liu et al. 2019) that simplifies it.

Consider a T-steps AMAPF instance with the graph $G = (V, E)$. We first create $2T + 1$ copies of V and mark them as follows: $0, 1, 1', 2, 2', ..., T'$ – see Fig. 1. Hereinafter, we will use the term vertices to denote the elements of the original AMAPF graph and the term nodes to denote the elements of the constructed network. We will also call copies t' (with apostrophe) and copies t for $t = 0, 1, ..T$ as outer and inner copies, respectively. The copied vertices of the original graph form the nodes of the network. Each node is identified by (v, h), where h stands for the copy, alternatively referred to as height. Higher node means greater copy, and h' is higher than h. Indeed, the node (v, h) (as well as (v, h')) corresponds to the state of an agent located in vertex v at time step h.

Then, for each edge $e(u, v)$ in the original graph, we connect the nodes (u, h') and $(v, h+1)$ for $h > 0$, and also connect $(u, 0)$ and $(v, 1)$. These edges correspond to the move actions and we call them the *move* edges. Then we add the *wait* edges that connect the nodes (u, h') and $(u, h + 1)$ for $h > 0$ and the nodes $(u, 0)$ and $(u, 1)$. Additionally, we add the edges between the nodes (u, h) and the (u, h'). These edges do not denote to any action but are added to forbid any two s-g paths in the network to share any node and therefore to avoid node-collisions. We call them the *restriction* edges. Finally, we add the source s and sink g nodes and connect s to all nodes $(u, 0)$ where u is a start vertex, and connect g to nodes (v, T') where v is a goal vertex.

Hereinafter whenever a path in a network is mentioned we mean s-g path. The matching between the AMAPF and MF is now straightforward. Each plan for an agent can be matched to a path in the network by matching the agent actions with the *move* or *wait* edges, and using the *restriction* edges to connect between them. In the same way, a path in the network is matched to a plan for an agent where the *move*/*wait* edges are matched to move/wait actions. See Fig. 1 for an self-contained example. It was proved that there

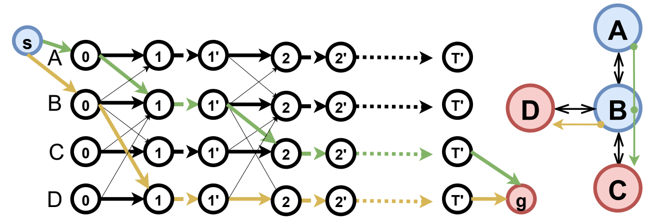

Figure 1: Example of the flow network (left) for a T-steps AMAPF instance (right). Each line in the flow network presents the copies of a single vertex in the original AMAPF graph. The diagonal, solid horizontal and dashed horizontal edges in the network denote *move*, *wait* and *restriction* edges. In this example, the AMAPF instance has two start vertices A, B and two goal vertices C, D, so the source node s in the network is connected to the nodes $(A, 0), (B, 0)$ and the sink g is connected to $(C, T'), (D, T')$. The example also shows the matching between the plans for the agents and the s-g paths in the network (green and yellow edges).

are no shared nodes in any two paths in the network (that form the solution to the MF problem) which infers that all matched plans have no node-collisions. Edge-collision may happen if two paths pass the edges $((u, h') \rightarrow (v, h+1))$ and $((v, h') \rightarrow (u, h+1))$ as these two different edges in the network refer to the same edge in the original graph. However, using the approach suggested in (Liu et al. 2019) these collision can be eliminated in the following fashion. Instead of two conflicting agents move to their next vertices they swap plans and continue moving by each others plan. As a result, the agents' plans can be obtained by finding the maximum number of paths (maximum flow) in the described network.

Solving Maximum Flow

Ford-Fulkerson algorithm (Ford and Fulkerson 1956) was suggested in (Yu and LaValle 2013b) to solve the maximum flow problem. The algorithm is simple, easy to implement and its complexity on the reduced networks is $O(kEV)$ (formulated in (Yu and LaValle 2013b)) , where E, V stand for the number of nodes and edges in the original graph. Since in our MF problem, all edges have a capacity of one and the value of the flow is bounded by the number of agents (as each path is to be matched to an agent's plan), Ford-Fulkerson can be considered as one of the most efficient solvers for our case (see (Cruz-Mejía and Letchford 2023) for more wider details about Maximum Flow algorithms).

We refer the reader to original paper for the detailed description of Ford-Fulkerson and here just briefly describe how the algorithm works specifically for the reduced network. In particular, the following two steps are sequentially repeated. First, we find a path p from s to g in the network. Second, we reverse all edges of p. We keep repeating these two steps until no path can be found. The found paths form the maximum flow in the network (in our case, they form the plans of the agents in the original AMAPF problem).

Solving MF Efficiently On The Introduced Networks

The maximum value of T needed to solve AMAPF problem can be up to $k+V-2$ (as shown by (Yu and LaValle 2013b)). This implies that the network size may be quadratic in the number of vertices of the original graph. Therefore, finding an s-g path may become a bottleneck when solving AMAPF instances involving large input graphs. To this end, we propose an algorithm to enhance the search process taking advantage from the specific structure of the reduced network and as a result solving AMAPF problem much faster.

First, we present some definitions to use in the algorithm description. In our algorithm, the search state corresponds to the network node. We will use $n(v, h)$ to denote the search states. Recall that the network node is defined by the vertex in original graph, v, and the height (copy) h (higher node means higher copy, and h' is higher than h). Let us define a *connected-sequence* as a sequence of nodes with the same vertex in which we can achieve the last node from the first node using only *wait* and *restriction* not-reversed edges, and it cannot be extended in neither side (i.e. it has maximum length). An example is depicted in Fig. 2. Initially, for each vertex v of the original graph we have only one connected-sequence $(v, [0, T'])$ i.e. the one that starts with the node with height 0 and ends with the node of height T' (shown by yellow crossbars). After a path (green path) is found, its edges are reversed (red edges in lower graph). As a result, some connected-sequences disconnect which results in several connected-sequences at the same vertex (see the lower network).

Idea The suggested algorithm is a graph traversal algorithm where the order of the search states in the search set (OPEN) is determined by their heights, i.e. the states with the lower heights are expanded first. The crucial idea of the algorithm is to expand states in *bulks* while searching. That is, instead of expanding one search state, i.e. generating its successors and marking it as visited (adding it to CLOSED), in each search iteration we (implicitly) expand a bulk of states at once. Such a bulk expansion can be effectively implemented using the introduced notion of the connected sequence, resulting in the reduction of the expansions number, time and memory compared to expanding states individually. Next, we describe how we can form the bulks of states and how the successors can be found compactly and fast.

We note that a similar but a more specific concept (in time-spaces), was used in (Phillips and Likhachev 2011), (Gonzalez, Dornbush, and Likhachev 2012) to implicitly compress and expand the states generated by the wait actions of agents [1]. Let us assume that while traversing the graph naively by single nodes, we are to expand a state (v, h) which is located in a connected-sequence $(v, [h_{min}, h_{max}])$. We can note that the nodes with the higher heights inside the connected-sequence (i.e. the nodes $(v, x) : x \in [h+1, h_{max}]$) are all achievable from (v, h) via the *wait* and *restriction* edges. Hence, the idea is to directly (and implicitly) generate all of these states ($(v, x) : x \in [h+1, h_{max}]$), form an *implicit* bulk consisting of these states (including the originally picked-up node (v, h)), and expand them all at once. We will refer to the described mechanism of generating the sequential successors of states that uses only *wait* and *restriction* edges as to the *straightforward* generation. Note that it is enough to store the vertex and the height bounds of the bulk to define it. So technically when forming a bulk (for future expansion) we only need to find the last *straightforwardly* achievable state (i.e. the highest state in the connected-sequence).

When the bulk is ready, it is expanded in the following fashion. First, let us assume the naive expansion of a bulk when for every node that resides in it we generate all of its immediate successors. Now observe that as a result of such procedure we are likely to have numerous successors that are characterized by the same graph vertex and different heights. Moreover many of these successors may belong to the same connected sequence. Thus, instead of explicit generation of them we generate only the ones with lower heights in their sequences. As the other search states from these sequences will be *straightforwardly* generated later on (i.e. when the search state with the lowest copy will be picked for expansion and its bulk will be formed as described above).

In other words, to expand a bulk of states $(v, [h_l, h_u])$, we do the following. We iterate over all connected-sequences in neighbor vertices of v, in which we can achieve at least one node (from a node $(v, x) : x \in [h_l, h_u]$). Then, we only generate the accessible node with the minimum height in each of these connected-sequences. The periodic structure of the reduced networks allows us to find the node with minimal height fast (as we will show later). As the number of connected-sequences is much less than the number of individual nodes, this leads to high reduction in the number of generated states. We call the algorithm that utilizes the described concepts Bulk-Search (BS). Next, we describe the details of the implementation.

Details Algorithm 1 shows the pseudo-code of a graph traversal algorithm with our modifications i.e. bulk search. First we order the states inside the search set (OPEN) by their height i.e. we always choose the state with the minimum height for the expansion. This will help us in reducing the number of expansions as we will see later. Second change is that whenever we need to check whether a chosen state was expanded before, we additionally check if the state can be *straightforwardly* generated from the previously opened states. In this case, we do not need to expand it, as we assume that it was implicitly expanded before, and its successors were already generated and inserted into the search set. This can be done by checking if any state in the same connected-sequence and lower height was expanded before (i.e. stored in CLOSED set)(lines 11-14). This check should also be done when inserting new states into OPEN set (lines 20-23).

[1] The mentioned algorithms were originally tailored to graph-based pathfinding in the presence of dynamic obstacles where each vertex of the graph had to be annotated with the (safe) time intervals and the search utilized implicit move-then-wait actions. Our suggested algorithm on the other hand handles explicit graphs, and the notion of connected sequence in general is not obliged to relate to time dimension, as will be shown later in the paper.

Algorithm 1: Bulk Search

Input: Network $N(G, cap, s, g)$
Output: Path from s to g if exists
1: OPEN $\leftarrow \phi$, CLOSED $\leftarrow \phi$
2: insert $s \rightarrow$ OPEN
3: **while** OPEN $\neq \phi$ **do**
4: remove $n(v, h)$ with the minimum h from OPEN
5: **if** $n = g$ **then**
6: **return** path from s to n
7: **end if**
8: **if** $n \in$ CLOSED **then**
9: continue
10: **end if**
11: $x(v, h') \leftarrow$ the state from CLOSED in the same connected-sequence of n and minimum height
12: **if** x exists **and** $h' <= h$ **then**
13: continue
14: **end if**
15: $succ \leftarrow getSuccessors(n)$
16: **for** $n'(u, h')$ in $succ$ **do**
17: **if** $n' \in$ OPEN $\cup$ CLOSED **then**
18: continue
19: **end if**
20: $x(u, h'') \leftarrow$ the state from OPEN $\cup$ CLOSED in the same connected-sequence of n' and minimum height
21: **if** x exists **and** $h'' <= h'$ **then**
22: continue
23: **end if**
24: insert $n' \rightarrow$ OPEN
25: **end for**
26: **end while**
27: **return** no answer

Algorithm 2: Generating successors

Input: Network $N(G, cap, s, g)$, Node $n(v, h)$
Output: The successors of $n(v, h)$
1: **if** $n = s$ **then**
2: **return** all neighbor nodes of n in G
3: **end if**
4: $succ \leftarrow \phi$
5: $[h_{min}, h_{max}] \leftarrow$ height bounds of the connected-sequence where n is located
6: **if** $h = h_{min}$ **then**
7: insert all neighbor nodes of n in $G \rightarrow succ$
8: **end if**
9: insert all neighbor nodes of node (v, h_{max}) in $G \rightarrow succ$
10: **for** each connected-sequence $cs(u, [h_l, h_u])$ with vertex u is a neighbor of v **do**
11: $c'_{from} \leftarrow$ the height of the minimum outer copy in $[max(h_{min} + 1, h), h_{max} - 1]$
12: $c_{to} \leftarrow$ the height of the minimum inner copy in $[h_l, h_u]$
13: **if** $c'_{from} + 1 <= h_u$ **and** $c_{to} - 1 <= h_{max}$ **then**
14: $c_{min} \leftarrow max(c_{to}, c'_{from} + 1)$
15: insert $(u, c_{min}) \rightarrow succ$
16: **end if**
17: **end for**
18: **return** $succ$

The third change is how we generate the successors of all states (the taken one from OPEN along with its *straightforwardly* generated states) fast. The pseudo-code is presented in Algorithm 2. Firstly (lines 1-3), if the node is the source node then we have only one node as input (i.e. no implicit states) and thus we need to only generate neighboring successors (i.e. connected nodes in the network) and return them. Otherwise, if the input node $n(v, h)$ is an inner node in the connected-sequence $(v, [h_{min}, h_{max}])$, we should generate the successors of all states $n(v, x) : x \in [h, h_{max}]$. This can be done as follows. If the state (v, h) is located in the beginning of the connected-sequence (i.e. $h = h_{min}$), this state may have reversed edges, so we always generate all neighboring nodes of this state in the network. The same thing is applied on (v, h_{max}) where we also generate all its neighboring nodes in the network. Other nodes (i.e. nodes $(v, x) : x \in [max(h_{min} + 1, h), h_{max} - 1]$) have only *move* edges to connect to nodes in other connected-sequences, so we can do the following (lines 10-17) to generate their successors. We iterate over all connected-sequences cs in neighbor vertices of v. We then check whether we can achieve at least one node in cs (from nodes $(v, x) : x \in [max(h_{min} + 1, h), h_{max} - 1]$). As shown in lines 11-13, this can be done by checking whether there is at least one *move* edge which starts from an outer copy in $[max(h_{min} + 1, h), h_{max} - 1]$ and ends at inner copy in cs. If so, we generate the node with the minimum accessible height in cs and added to successors set (lines 14-15).

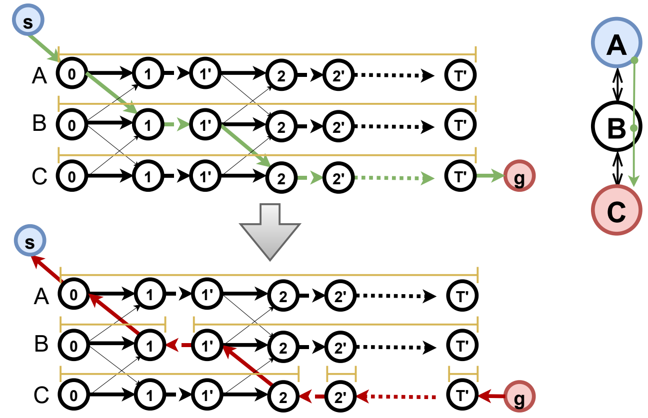

Figure 2: Example showing the connected-sequences on the network. Yellow crossbars denote to the connected-sequences on each vertex. Initially, we have the connected-sequences as shown in the upper figure. After a path (green one) is found and its edges are reversed, the connected-sequences are divided as shown in the lower figure.

As a result, any achievable node from the source node is either explicitly or implicitly (from a node with lower height in the same connected-sequence) expanded. Therefore, we can immediately state the following theorem.

Theorem 1. *BS is a complete algorithm.*

Theoretical analysis The search states inside the search set are sorted according to the height. This helps in reducing the number of expansions as lower height states implicitly expands the higher height states in the same connected-sequence but the opposite is not applicable. Theoretically

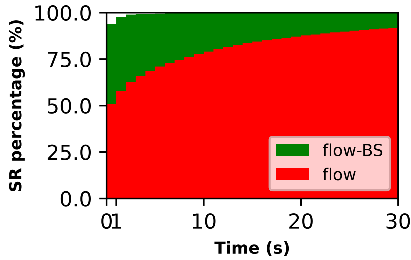

Figure 3: The (normalized) number of instances solved by a certain time cap.

multiple single states in one connected-sequence may be expanded individually e.g. if they opened in descending-height way. This is possible even if we order the states according to heights as we have reversed edges which generates states with lower height. However, in practice only few states are expanded in each connected-sequence and thus the algorithm mainly depends on the number of connected-sequences. Fortunately, the number of connected-sequences is much smaller than the size of the network. Initially, we have a number of connected-sequences equals to the number of original graph vertices $|V|$. After each path is found, T (the length of the path) new connected-sequences appear. As a result, in the whole search for all k agents, the number of appeared connected-sequences equal $\sum_{i=1}^{i=k} |V| + T(i-1) = k|V| + Tk(k-1)/2$. Therefore, we have a theoretical reduction in the number of nodes (comparing with $k|V|T$, the number of nodes without compressing in connected-sequences) equals $min(|V|/k, T/2)$. This reduction is significantly high that allows us to obtain the fastest full success (to our knowledge) in optimally solving all public MAPF benchmarks for a MAPF-family problem, as will be shown in next section.

Experimental Evaluation

We implemented the improved AMAPF solver in C++[2] and compare it with the state-of-the-art AMAPF solver that does not use the introduced bulk search to solve MF but rather utilizes the standard Ford-Fulkerson as suggested in (Yu and LaValle 2013b). The code of the competitor was taken from public repo [3] that accompanied the paper (Okumura and Défago 2022). We kept all the optimization techniques designed by the code authors. We will denote these two solvers as *flow-BS* (ours) and *flow* (state of the art). The experiments were conducted on a PC with Intel Core i7-10700F CPU @ 2.90GHz × 16 and 32Gb of RAM.

The MAPF maps and instances were taken from the publicly available MAPF benchmark (Stern et al. 2019b). We use all 33 maps available in this benchmark and all 25 random scenarios. Each scenario on each map (except some small ones), contains 1000 pairs of start-goal positions. To test a solver on a scenario, we run it with the first 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1000 pairs, sequentially. Whenever a solver fails to solve a problem under a time limit of 30 seconds, we terminate testing on this scenario and move to the next one.

Map	Width	Height	Algorithm	
			flow	flow-BS
empty-8-8	8	8	**100%**	**100%**
empty-16-16	16	16	**100%**	**100%**
maze-32-32-2	32	32	**100%**	**100%**
room-32-32-4	32	32	**100%**	**100%**
maze-32-32-4	32	32	**100%**	**100%**
random-32-32-20	32	32	**100%**	**100%**
random-32-32-10	32	32	**100%**	**100%**
empty-32-32	32	32	**100%**	**100%**
empty-48-48	48	48	**100%**	**100%**
den312d	65	81	**100%**	**100%**
room-64-64-8	64	64	**100%**	**100%**
random-64-64-20	64	64	**100%**	**100%**
room-64-64-16	64	64	**100%**	**100%**
random-64-64-10	64	64	**100%**	**100%**
warehouse-10-20-10-2-1	161	63	**100%**	**100%**
ht_chantry	162	141	**100%**	**100%**
maze-128-128-1	128	128	3%	**100%**
ht_mansion_n	133	270	96%	**100%**
warehouse-10-20-10-2-2	170	84	**100%**	**100%**
lt_gallowstemplar_n	251	180	82%	**100%**
maze-128-128-2	128	128	4%	**100%**
ost003d	194	194	57%	**100%**
lak303d	194	194	44%	**100%**
maze-128-128-10	128	128	17%	**100%**
warehouse-20-40-10-2-1	321	123	91%	**100%**
den520d	256	257	16%	**100%**
w_woundedcoast	642	578	1%	**100%**
warehouse-20-40-10-2-2	340	164	8%	**100%**
brc202d	530	481	1%	**100%**
Paris_1_256	256	256	5%	**100%**
Berlin_1_256	256	256	6%	**100%**
Boston_0_256	256	256	4%	**100%**
orz900d	1491	656	0%	**100%**

Table 1: Succes rates of *flow-BS* and *flow* solvers with a timeout of 30 seconds.

In the first experiment we used a precise estimator of T suggested by (Okumura and Défago 2022) (which solves bottleneck assignment problem (Gross 1959)) to estimate the lower bound of the makespan. As this estimator takes non-negligible time (up to 10 seconds) and both algorithms use it, we did not account its time from the 30s time budget.

For each map, we collected the total number of success instances over all scenarios. The table 1 summarizes the results. As can be shown, our solver was able to solve all instances in all maps under 30s. However, this was not the case for *flow* solver. The latter searches the network node-by-node and, therefore, was able to solve accomplish test only on the the small maps. When maps are large (like city maps that are 256×256) or the makespan is high (like in the maze maps), it often was not able to produce solution under the imposed limit. Fig. 3, shows the total success rate for all instances in all maps every one second. In fact, *flow-BS* was

[2]https://github.com/PathPlanning/AMAPF-with-MF-and-Bulk-Search.git

[3]https://github.com/Kei18/tswap

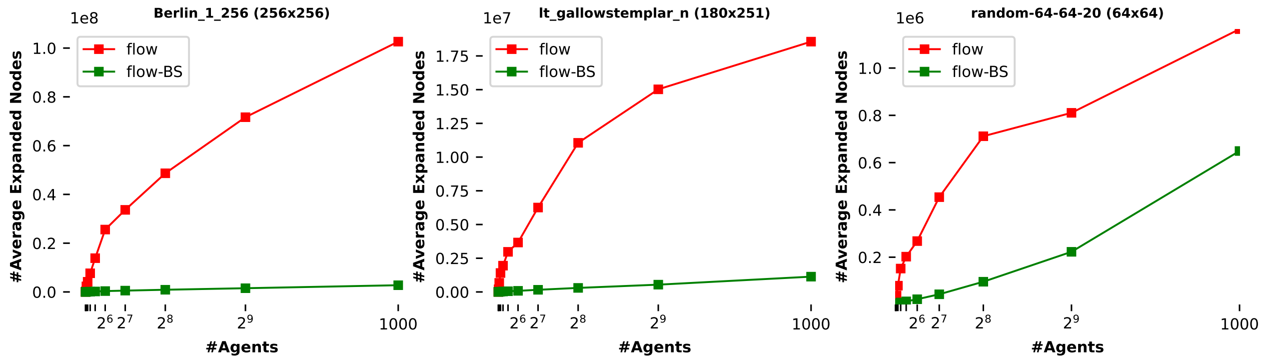

Figure 4: The number of expanded nodes with different number of agents on different maps.

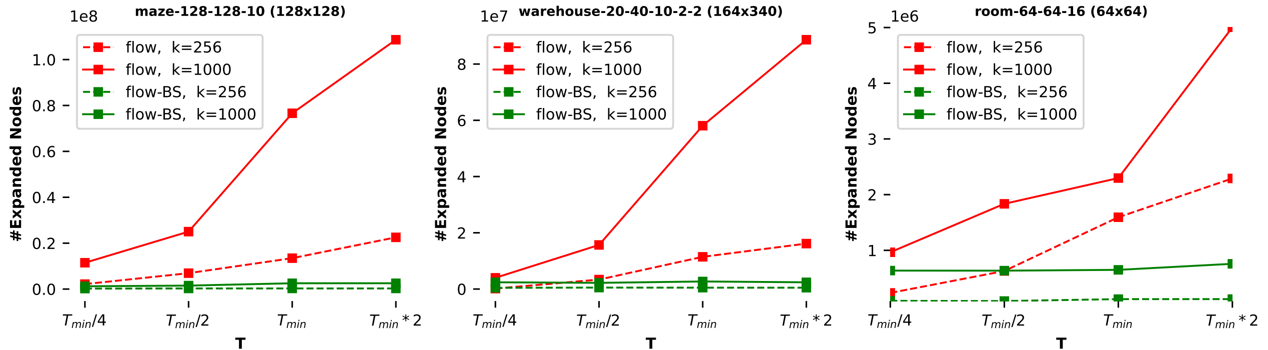

Figure 5: The figure shows for specific map and specific number of agents, how the number of expansions changes for increasing values of T (the maximum height (copy)). The values of T are chosen relative to the value of the real makespan T_{min}.

able to solve the hardest instance (`orz900d` map, scenario 20 with 256 agents) in 17s. On the other hand, *flow* showed very small increase of SR after solving the easy instances (75% of instances) around the 8th second.

In the second experiment, we investigated how the number of agents affects the performance. For this purpose, we selected three maps of different topology and size and plotted the average number (over all scenarios) of expanded nodes (machine-independent metric) while searching, against the number of agents. The results (plotted in Fig. 4) show that our algorithm expands much less nodes than standard algorithm (as expected). This again proves the dependence of our algorithm on the number of connected-sequences in the network but not the network size. Moreover, the number of connected-sequences increases linearly with the number of agents (it approaches to quadratically when the number of agents approaches the size of the network) as was deduced theoretically earlier in this paper. For standard Ford-Fulkerson algorithm, the search also increases linearly with the number of agents, but with the size of the whole network. It worth to note here, that the relation between the runtime and number of expansions is not fixed but also increases. This is because the single memory-access/read/write operation takes more time when the amount of the stored data increases.

So far, we assumed that we have a good estimator which can estimate an optimal or near-optimal lower bound (T) of the makespan. However this may be not the case, therefore the search may be repeated several times until finding the optimal solution. Therefore, we also conducted experiments to show the practical performance of both solvers for different T when we fix the map and the number of agents. The tests were designed as follows. For each one of three fixed maps (the maps `warehouse-20-40-10-2-2`, `maze-128-128-10` and `room-64-64-16` were chosen), and for a number of agents $\in \{256, 1000\}$, we run the solvers on networks with maximum height (copy) equal one of the values $\{T_{min}/4, T_{min}/2, T_{min}, T_{min} * 2\}$, where T_{min} is the optimal makespan. Again, we recorded the average number of expanded nodes over all scenarios (see Fig. 5). Obviously, the number of nodes expanded by *flow* significantly increases with the value of T, while *flow-BS* does not demonstrate such growth. That means that our solver is especially useful when one can not estimate the optimal makespan accurately before actually solving an AMAPF problem instance.

Conclusion

In this paper we have revisited the reduction-based approach to optimally solving Anonymous MAPF problem, when the latter is reduced to a search problem on an auxiliary graph of a special structure. We have suggested an improved AMAPF solver that is based on a specific search algorithm, tailored to find paths on the auxiliary graphs exploiting their spe-

cific topology. We have showed that our improved AMAPF solver significantly outperforms state of the art on a large variety of setups, due to its better scalability to the size of the input graph.

Acknowledgments

This work was partially supported by the Analytical Center for the Government of the Russian Federation in accordance with the subsidy agreement (agreement identifier 000000D730321P5Q0002; grant No. 70-2021-00138).

References

Adler, A.; De Berg, M.; Halperin, D.; and Solovey, K. 2015. Efficient multi-robot motion planning for unlabeled discs in simple polygons. In *Algorithmic Foundations of Robotics XI: Selected Contributions of the Eleventh International Workshop on the Algorithmic Foundations of Robotics*, 1–17.

Ahuja, R. K.; Magnanti, T. L.; and Orlin, J. B. 1995. *Network flows: theory, algorithms and applications*. Prentice Hall.

Barták, R.; Ivanová, M.; and Švancara, J. 2021. From classical to colored multi-agent path finding. In *Proceedings of the 14th International Symposium on Combinatorial Search (SoCS 2021)*, 150–152.

Chen, Z.; Alonso-Mora, J.; Bai, X.; Harabor, D. D.; and Stuckey, P. J. 2021. Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. *IEEE Robotics and Automation Letters*, 6(3): 5816–5823.

Cruz-Mejía, O.; and Letchford, A. N. 2023. A survey on exact algorithms for the maximum flow and minimum-cost flow problems. *Networks*.

Ford, L. R.; and Fulkerson, D. R. 1956. Maximal flow through a network. *Canadian journal of Mathematics*, 8: 399–404.

Ford Jr, L. R.; and Fulkerson, D. R. 2015. *Flows in networks*, volume 54. Princeton university press.

Gonzalez, J. P.; Dornbush, A.; and Likhachev, M. 2012. Using state dominance for path planning in dynamic environments with moving obstacles. In *2012 IEEE International Conference on Robotics and Automation*, 4009–4015. IEEE.

Gross, O. 1959. *The bottleneck assignment problem*. Rand.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J.; and Ayanian, N. 2018. Conflict-based search with optimal task assignment. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, 11272–11281.

Liu, M.; Ma, H.; Li, J.; and Koenig, S. 2019. Task and path planning for multi-agent pickup and delivery. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Ma, H.; and Koenig, S. 2016. Optimal Target Assignment and Path Finding for Teams of Agents. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, 1144–1152.

Ma, H.; Li, J.; Kumar, T.; and Koenig, S. 2017. Lifelong multi-agent path finding for online pickup and delivery tasks. *arXiv preprint arXiv:1705.10868*.

Nguyen, V.; Obermeier, P.; Son, T. C.; Schaub, T.; and Yeoh, W. 2017. Generalized target assignment and path finding using answer set programming. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 1216–1223.

Okumura, K.; and Défago, X. 2022. Solving Simultaneous Target Assignment and Path Planning Efficiently with Time-Independent Execution. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 270–278.

Phillips, M.; and Likhachev, M. 2011. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE international conference on robotics and automation*, 5628–5635. IEEE.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Solovey, K.; and Halperin, D. 2016. On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research*, 35(14): 1750–1759.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019a. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search (SoCS)*, 151–158.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019b. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Twelfth Annual Symposium on Combinatorial Search*.

Xu, Q.; Li, J.; Koenig, S.; and Ma, H. 2022. Multi-Goal Multi-Agent Pickup and Delivery. In *Proceedings of the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2022)*.

Yu, J.; and LaValle, S. M. 2013a. Multi-agent path planning and network flow. In *Algorithmic foundations of robotics X*, 157–173. Springer.

Yu, J.; and LaValle, S. M. 2013b. Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation*, 3612–3617. IEEE.