Bulk Search For Optimally Solving Two Variants Of Anonymous Multi-agent Pathfinding

ZAIN ALABEDEEN ALI*, MIR AI, Russia
KONSTANTIN YAKOVLEV, Saint-Petersburg University, Russia and CogAI Lab, Russia

We consider the Anonymous Multi-Agent Path-Finding (AMAPF) problem where the agents are 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 seek solutions that minimize either the sum of costs of all paths (SOC) or the maximum path cost (makespan). Two assumptions are usually used for the agents' behavior after reaching the goals. The first assumption is that agents stay at the goals after reaching them and wait for the other agents to reach their goals, while the second assumption is that agents disappear upon arrival at goal vertices. We refer to the AMAPF problem under the first assumption and to the problem under the second assumption as **AMAPF** and **AMAPFD**, respectively. AMAPF can be solved in polynomial time with respect to the makespan objective (although existing approaches have a critical bottleneck in one part of the solving process). For the SOC objective, the known optimal approaches have worst-case exponential complexity. For AMAPFD with the SOC objective, to the best of our knowledge, no dedicated optimal solver has been reported. To this end, in this paper, we enhance the existing solution of the AMAPF-makespan problem and introduce a novel solution to AMAPFD-SOC. Specifically, for the AMAPF-makespan problem, the polynomial approach to solving 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 for the standard Maximum Flow solvers. We suggest a specific search algorithm (called Bulk Search) 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, reducing the runtime and memory consumption. Empirically, the resulting AMAPF solver outperforms the competing approach based on the standard MF solver, and solves all publicly available MAPF instances from the MovingAI benchmark in less than 30 seconds. For the AMAPFD-SOC problem, we suggest a method to reduce the problem to a Minimum-Cost Maximum-Flow (MCMF) problem on a similar auxiliary graph used in the AMAPF problem but now with non-zero edge costs. After that, we propose a modified version of Bulk Search, called Generalized Bulk Search to help solve the MCMF problem efficiently. As a result, the AMAPFD-SOC problem is solved by an efficient solver that could solve 98.5% of all MovingAI benchmark instances in less than 30 seconds.

JAIR Track: Multi-Agent Path Finding

JAIR Associate Editor: Daniel Harabor

JAIR Reference Format:
Zain Alabedeen Ali and Konstantin Yakovlev. 2026. Bulk Search For Optimally Solving Two Variants Of Anonymous Multi-agent Pathfinding. *Journal of Artificial Intelligence Research* 86, Article 2 (May 2026), 40 pages. DOI: 10.1613/jair.1.19379

*Corresponding Author.

Authors' Contact Information: Zain Alabedeen Ali, ORCID: 0000-0001-5833-3172, zainalabedeenali@gmail.com, MIR AI, Moscow, Russia; Konstantin Yakovlev, ORCID: 0000-0002-4377-321X, k.yakovlev@spbu.ru, Saint-Petersburg University, Saint-Petersburg, Russia and CogAI Lab, Moscow, Russia.

DOI: 10.1613/jair.1.19379

1 Introduction

The Multi-Agent Path Finding (MAPF) problem is a problem that 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 (Stern et al. 2019). MAPF has many applications, including automated warehouses, autonomous vehicles, and video games and is being widely studied in the literature. Depending on the application, many variants of MAPF have been proposed and numerous solutions have been already presented, see (Alkazzi and Okumura 2024; Felner, Stern, et al. 2017; Ma 2022) for overviews on MAPF.

In a conventional MAPF formulation the environment is represented as a graph to which the agents are confined. I.e. each agent can wait only at the graph vertices and move only along the graph edges. Time is discretized and each action takes exactly one time step. MAPF under such assumptions, sometimes called Classical MAPF, resembles much the seminal Pebble Motion on Graphs problem which is known to be solvable (i.e., a feasible solution can be found) in polynomial time (Kornhauser et al. 1984). In MAPF community such polynomial-time algorithms as Push and Swap (Luna and Bekris 2011), Push and Rotate (Wilde et al. 2013), Bibox (Surynek 2009), PIBT (Okumura, Machida, et al. 2022) etc. are known.

The cost of the solution in MAPF is most commonly measured either as the maximum over the durations of the agents' plans – *makespan*, or as the sum of these durations – *flowtime* or *Sum of Costs (SOC)*. The fast MAPF solvers mentioned in the previous paragraph are neither optimal nor provide any bounds on the suboptimality of the solution. Unsurprisingly, optimally solving MAPF with respect to makespan or SOC is NP-hard (Surynek 2010; Yu and LaValle 2015). Still, as obtaining high-quality solutions is vital from both theoretical and practical perspectives, a plethora of optimal (yet limited in scalability and computationally inefficient in practice for a large number of robots) MAPF solvers exist (Sharon et al. 2015; Standley 2010; Yu and LaValle 2016).

Next, in practice many other different variants of MAPF problems arise. One of such problems is the so-called Anonymous MAPF (AMAPF), also referred in some works as Unlabeled MAPF (Dergachev and Yakovlev 2024; Fine et al. 2023; Ma and Koenig 2016). In AMAPF the goal vertices are not assigned to the agents beforehand and each agent may occupy any goal. In this paper we are interested in this problem. Figure 1 illustrates the difference between MAPF and AMAPF on a small grid.

Intuitively to solve AMAPF one needs to tackle two (difficult) sub-tasks: distributing the goals between the agents and planning collision-free paths. Thus it is tempting to infer that optimally solving AMAPF is at least as hard as optimally solving MAPF, i.e., NP-hard. Surprisingly, optimal solutions of AMAPF with respect to makespan can be obtained in polynomial time as shown in (Yu and LaValle 2013a). In this paper a reduction-based approach to AMAPF is introduced when the original problem is reduced to a series of specific graph-search problems, each of which can be solved in polynomial time. These problems are the problems of finding a maximum flow (Goldberg and Tarjan 2014) on a graph of special structure – network – induced by the input MAPF graph. The main computational bottleneck of the resultant solver (and the other ones that are based on the similar reductions) is that the size of the arising networks is very large (especially when the input MAPF graph is large), and searching over such large network becomes burdensome. This is a major problem and the one we are focusing in this paper.

Another focus is considering AMAPF with disappearing agents – AMAPFD – problem. In this case it is assumed that upon reaching a goal the agent is removed from the graph and this vertex can be used by the other agents if needed. The motivation for studying this AMAPFD variant is twofold. First, disappearing agents arise naturally in applications where goal locations represent exits or service points outside the traversable workspace. Upon reaching such a goal, an agent leaves the shared operating area and no longer interacts with the remaining traffic (e.g., trains entering terminal bays, or warehouse robots delivering inventory pods to fill/replenishment/charging stations located beyond the aisle network). Figure 2 illustrates this setting.

Fig. 1. Two instances of MAPF and AMAPF problems in simple 4×4 grid environments. On the left, we have a MAPF instance where two robots 1 and 2 need to visit two goal cells 1 and 2 (denoted by light red circles), respectively. The optimal solution (to visit the goal cells as fast as possible) is to move the robots by the green arrows. In the right image, we have an AMAPF instance where each robot can go to any goal cell. The optimal solution is to move each robot to the goal near it.

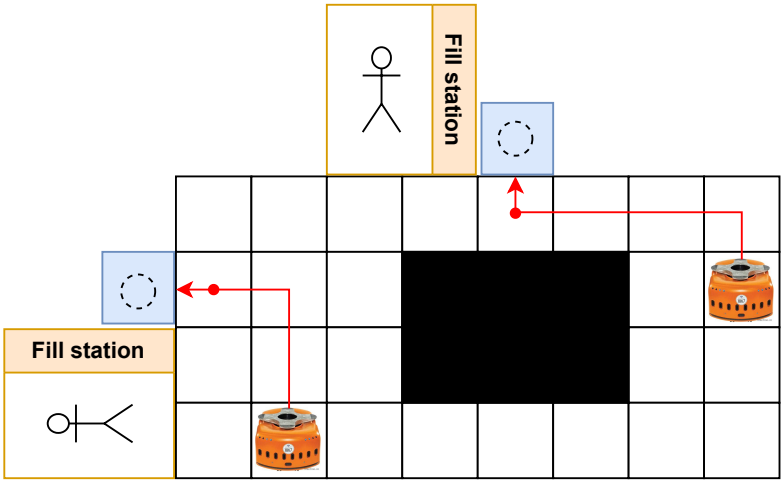

Fig. 2. An example of the AMAPFD problem. Two identical robots must reach the fill stations. Any robot may be assigned to any station, and upon entering its assigned station cell, it leaves the traversable area (i.e., disappears) and no longer participates in collision avoidance.

Second, AMAPFD solutions are useful as a subroutine in realistic Lifelong AMAPF (LAMAPF) settings, where a pool of homogeneous, interchangeable agents operates continuously (e.g., transporting goods in a warehouse). In such problems, once an agent completes its currently assigned task, a new goal is revealed, and the system must repeatedly decide both (i) which agent should be assigned to which newly available goal and (ii) how the agents should be routed. In this context, directly invoking an AMAPF solver can be ill-suited because it enforces an additional synchronization requirement: after reaching a goal, an agent must remain there (via waiting-at-goal

actions) for the remainder of the joint plan. In operational systems, however, agents are typically not required to occupy the goal locations for extended periods but rather are free to leave them, which is better captured by the AMAPFD model. Consequently, in practical LAMAPF pipelines that integrate task assignment with multi-agent pathfinding, it can be preferable to employ an AMAPFD solver rather than a standard AMAPF solver. Such an approach was used by the authors in a system that ranked third in the Scheduler track of the League of Robot Runners 2024 contest[1], an open competition sponsored by Amazon and focused on MAPF problems motivated by real-world robotic logistics.

Our scope and contribution. In this work we study two variants of AMAPF problem which we will denote further on as AMAPF-makespan and AMAPFD-SOC. The first problem is to find a makespan optimal solution for conventional AMAPF, as described above. The second problem assumes that the agents disappear when they reach their target locations (goal vertices) and the task is to find a SOC optimal solution. We are interested in solving both problems not only optimally but also efficiently (computationally-wise).

To this end we, first, introduce a novel search technique – Bulk Search (BS) – that is tailored to efficiently find paths (not necessarily shortest) in the large networks that arise in AMAPF reductions. The crux of this search technique 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, BS, is complete. On the practical side, we compare our improved AMAPF-makespan solver that utilizes BS with the state-of-the-art optimal AMAPF-makespan solver and show that our algorithm scales much 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 instances of the well-known MAPF benchmark from (Stern et al. 2019) – the so-called MovingAI benchmark.

Then we extend Bulk Search to tackle finding shortest paths on the networks that contain edges with costs. Such networks arise in the reduction of AMAPFD-SOC problem as we show in the paper. We add several improvements to Bulk Search and prove that the resultant search method, Generalized Bulk Search (GBS), is complete and optimal. Building upon GBS we present the first, to the best of our knowledge, optimal solver for the AMAPFD-SOC problem. We empirically evaluate it and compare with the straightforward adaptation of the AMAPF-SOC solver – CBS-TA (Hönig et al. 2018). The results of our comprehensive evaluation involving all problem instances from the MovingAI benchmark show that our solver notably outperforms the competitor.

Relation to the conference paper. This paper is an extension of the previously published conference paper on the topic (Ali and Yakovlev 2024). In the latter, however, only the AMAPF-makespan problem was studied. Thus, the current paper significantly extends the conference one as it:

- additionally studies another problem, i.e., AMAPFD-SOC;
- presents a novel (i.e., not previously described) reduction of this problem to a variant of the maximum flow problem which assumes edges with costs in the flow network – Minimum-Cost Maximum-Flow (MCMF);
- extends Bulk Search to efficiently find shortest paths in the networks with costs by generalizing it and introducing several heuristics, resulting in GBS algorithm;
- establishes the theoretical properties of GBS;
- empirically evaluates AMAPFD-SOC solver based on GBS.

[1] https://www.leagueofrobotrunners.org

2 Related Works

The problem we are investigating in this work, i.e., AMAPF, can be viewed as a combination of target assignment (TA) problem and MAPF problem. As we wish to obtain optimal solutions we, first, review optimal TA and MAPF solvers, and then discuss existing works that specifically consider AMAPF or similar problems.

Optimally solving multi-agent pathfinding. Obtaining optimal MAPF solutions with respect to a large variety of objective functions, including makespan and SOC is NP-hard (Surynek 2010; Yu and LaValle 2015). Still, as high-quality solutions are vital from both theoretical and practical perspectives, a plethora of optimal MAPF solvers exists. They can be roughly divided into two classes: search-based and reduction based. Reduction-based methods transform the MAPF problem into some other well-established problem in AI or computer science and rely on the off-the-shelf algorithms to solve the reduced problem. The most common reductions of MAPF are reductions to SAT (Barták and Švancara 2019; Surynek et al. 2016), ILP (Yu and LaValle 2013b), ASP (Erdem et al. 2013) and others.

Search-based methods rely on heuristic search as a fundamental approach to construct a solution, although the way in which search techniques are used may differ significantly from method to method. For example, A*+OD+ID (Standley 2010) searches in the composite action space of the agents (which grows exponentially with the number of agents) and utilizes several effective techniques to reduce the branching factor. M* (Wagner and Choset 2015) introduces the so-called sub-dimensional expansion scheme where the dimensionality of the search space is grown gradually from 1 (independent planning for each agent) up to the dimensionality of the fully-coupled system upon encountering the conflict between the agents. Another well-known approach for optimally solving MAPF is Conflict Based Search (CBS), introduced in (Sharon et al. 2015). CBS is a hierarchical planner that, at the higher level, imposes constraints in response to arising conflicts between the agents' plans and, at the lower level, conducts individual planning subject to the imposed constraints. CBS is a very popular optimal MAPF solver and a wide variety of enhancing techniques for it is known, such as bypassing and prioritizing conflicts (Boyarski et al. 2015), disjoint splitting (Li, Harabor, et al. 2019), high-level heuristics (Felner, Li, et al. 2018) to name a few. CBS is also known for its adaptations to non-classical MAPF setups such as large agents (Li, Surynek, et al. 2019), kinodynamic constraints (Kottinger et al. 2022; Moldagalieva et al. 2024), non-uniform (or even continuous) time (Andreychuk et al. 2019; Cohen et al. 2019).

Optimally solving task assignment. Assignment problem in its basic form is a problem of optimally matching the elements of two set colloquially termed 'agents' and 'tasks'. The classical method to solve this problem is the so-called Hungarian method presented in (Kuhn 1955)[2]. The method is polynomial and runs in $O(n^4)$ time, where n is the number of agents/tasks. It was later improved in several works, including (Tomizawa 1971), to run in $O(n^3)$ time. Original basic assignment problem seeks to optimize the cumulative cost of the assignment, which is analogous to SOC in MAPF. The variant that seeks to minimize the maximum cost among the individual assignments (analogous to makespan in MAPF) is known as bottleneck assignment problem (Ravindran and Ramaswami 1977). Many other variants of the assignment problems have been introduced and studied through the decades, such as quadratic assignment problem (Loiola et al. 2007), where the cost function is expressed in terms of the quadratic inequalities, or generalized assignment problem (Cattrysse and Van Wassenhove 1992), where an agent has a capacity and can be assigned to several tasks. A comprehensive review on assignment problems is presented in (Pentico 2007). Notably, while many involved variants of the assignment problem are NP-hard to solve, the variants that are the most relevant to this work, i.e., the basic assignment problem and the bottleneck assignment problem, can be solved (optimally) in polynomial time.

[2]The relation of this method to Hungary, and more specifically to Hungarian mathematics, is explained in (Frank 2005).

Anonymous multi-agent pathfinding and related problems. In (Yu and LaValle 2013a) the seminal optimal (with respect to makespan) AMAPF solver was introduced, that is based on the reduction of AMAPF to a series of specific graph-search problems, i.e., the problems of finding a maximum flow on a graph of special structure (network) induced by the input MAPF graph. For the sum-of-costs objective an adaptation of the previously mentioned search-based MAPF solver, CBS, was suggested in (Hönig et al. 2018) – CBS-TA. Notably, reduction-based polynomial-time AMAPF-SOC solvers do not exist, to the best of our knowledge, which may suggest that AMAPF-SOC is a harder problem to solve optimally (compared to AMAPF-makespan).

Suboptimal AMAPF was studied in (Okumura and Défago 2022). Several computationally efficient algorithms were proposed in this work which were empirically shown to provide high-quality solutions. However, no bound on sub-optimality was 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 explored in (Nguyen et al. 2017) and solved using the Answer Set Programming (ASP). A decentralized variant of AMAPF was studied in (Dergachev and Yakovlev 2024) and the corresponding solver was introduced.

AMAPF with disappearing agents (AMAPFD) was studied in (Fine et al. 2023) in combination with the assumption that there exists a deadline for each target location. However, the latter work presents a solver that minimizes travel distances rather than travel times (thus ignoring the cost of the waiting actions).

More involved variants of AMAPF were studied in (Barták, Ivanová, et al. 2021; Ma and Koenig 2016). 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 was suggested to solve this Colored MAPF problem. Barták, Ivanová, et al. (Barták, Ivanová, et al. 2021) proposed several solvers that utilize reduction to SAT. Indeed, AMAPF can be viewed as a special instantiation of the Colored MAPF problem (i.e., the one when there exists only a single 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, Tinka, et al. 2021) and Multi-agent Pickup and Delivery (MAPD) (Ma, Li, 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 the 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 part of the LMAPF/MAPD problem. Still, there are papers that consider a combined problem (Chen et al. 2021; Xu et al. 2022).

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

Overall, the most closely related works to this study are:

(i) (Yu and LaValle 2013a), where the reduction-based optimal AMAPF-makespan solver was proposed;
(ii) (Hönig et al. 2018), where the search-based optimal AMAPF-SOC solver was introduced;
(iii) (Fine et al. 2023), where AMAPFD was studied in combination with the assumption that individual deadlines for reaching goal locations exist.

In our work we use the same reduction as in *(i)* to optimally solve AMAPF-makespan, but we introduce a novel effective method to solve the reduced problem, and empirically confirm that our solver scales much better. In contrast to *(iii)* we focus on cost objective that takes time into account, not only travel distance and we do not consider deadlines. To this end, we introduce an original (i.e., not described in literature before) reduction to minimum-cost maximum flow problem and an efficient solver for it. We are not aware of any published work that presents an optimal AMAPFD-SOC solver. Thus, to conduct an empirical evaluation we use a modified (by

us) CBS-TA algorithm proposed in *(ii)* to solve AMAPF-SOC, which is a harder problem due to non-disappearing agents.

3 Problem Statement

We follow a classical approach (Stern et al. 2019) to define the problems under investigation – AMAPF/AMAPFD. 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. A set $\mathcal{G}$ of k goal vertices, $\mathcal{G} = \{g_1, \ldots, g_k\}$, are also distinguished. 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 $\pi^i = \{v^i_0, v^i_1, \ldots, v^i_{l^i}\}$ for an agent i, is a sequence of $l^i + 1$ vertices from V where $v^i_0 = s^i$ and for any $j \in \{0, \ldots, l^i - 1\}$ either $v^i_j = v^i_{j+1}$ (a wait action) or $(v^i_j, v^i_{j+1}) \in E$ (a move action i.e., traversing an edge in the graph). The cost of the plan is the time step by which the final vertex is reached, i.e., $cost(\pi^i) = l^i$. Two plans are said to contain a vertex (similarly, an edge) conflict if the agents following them occupy the same vertex (use the same edge) at the same time step. In AMAPF, the conflicts between agents can still occur after arriving at their final vertices, where this is not possible in AMAPFD. Formally, the vertex and edge conflicts between two paths $\pi^i = \{v^i_0, v^i_1, \ldots, v^i_{l^i}\}$ and $\pi^j = \{v^j_0, v^j_1, \ldots, v^j_{l^j}\}$ for both AMAPF and AMAPFD are defined as follows:

$$\begin{aligned}
&\text{Vertex conflict AMAPF:} && \exists x \in \{0, \ldots, \max(l^i, l^j)\} : v^i_{\min(x, l^i)} = v^j_{\min(x, l^j)}. \\
&\text{Vertex conflict AMAPFD:} && \exists x \in \{0, \ldots, \min(l^i, l^j)\} : v^i_x = v^j_x. \\
&\text{Edge conflict in AMAPF/AMAPFD:} && \exists x \in \{0, \ldots, \min(l^i, l^j) - 1\} : v^i_x = v^j_{x+1} \wedge v^i_{x+1} = v^j_x.
\end{aligned} \tag{1}$$

The problem now is to find a set of plans $\Pi = \{\pi^1, \ldots, \pi^k\}$, s.t. (1) each pair of plans is conflict-free, and (2) all goals are reached i.e.,

$$\{v^i_{l^i} | i \in \{1, \ldots, k\}\} = \mathcal{G}. \tag{2}$$

Essentially, this problem is 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 needs to construct a set of conflict-free plans.

We consider the following cost objectives: $makespan(\Pi) = \max_i cost(\pi^i)$, and $SOC(\Pi) = \sum_i cost(\pi^i)$. In this work, we are interested in obtaining makespan-optimal solutions for AMAPF problem and SOC-optimal solutions for AMAPFD problem.

4 Background

The methods to solve both AMAPF and AMAPFD problems include reduction to Network Flow problems. Therefore, we start by giving a brief background about network flow problems and then proceed to explaining the methods.

4.1 Network Flow

Generally, network flow problem might come in different flavors; see (Ahuja et al. 1995) for an overview. Here we focus on a specific variant of the problem needed for solving AMAPF/AMAPFD problems.

A network is a tuple $N = (\mathrm{G}, cap, cost, src, sink)$, where $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ is a directed graph (V is the set of nodes, and E is the set of arcs), $cap : \mathrm{E} \to \mathbb{Z}_{\geq 0}$ is the mapping defining the capacities of the arcs, $cost : \mathrm{E} \to \mathbb{Z}$ is the cost function which defines the costs of the arcs, $src \in \mathrm{V}$ is the source vertex, and $sink \in \mathrm{V}$ is the sink vertex. For

vertex $v \in \mathrm{V}$, let $\sigma^+(v)$ (resp. $\sigma^-(v)$) denote the set of arcs of G going to (resp. leaving) v. A feasible *src–sink* flow on the network is a mapping $f : \mathrm{E} \rightarrow \mathbb{Z}_{\geq 0}$ that satisfies three types of constraints:
Arc capacity constraints:

$$\forall e \in \mathrm{E}, f(e) \leq cap(e), \tag{3}$$

the flow conservation constraints at non-terminal vertices:

$$\forall v \in \mathrm{V} \setminus \{src, sink\}, \sum_{e \in \sigma^+(v)} f(e) - \sum_{e \in \sigma^-(v)} f(e) = 0, \tag{4}$$

and the flow conservation constraints at terminal vertices:

$$F(f) = \sum_{e \in \sigma^-(src)} f(e) = \sum_{e \in \sigma^+(sink)} f(e). \tag{5}$$

The quantity $F(f)$ is called the *value* of the flow f. Another interpretation of the flow is that the flow is a set of *src–sink* paths (possibly overlapping or even duplicating), where each path carries a unit of flow from *src* to *sink*, such that the sum of units passing through any arc does not exceed its capacity.

The cost of a flow f is defined by the value

$$Cost(f) = \sum_{e \in \mathrm{E}} cost(e) * f(e). \tag{6}$$

The standard single-commodity Maximum Flow (MF) problem asks the following question: Given a network N, what is the maximum $F(f)$ that can be pushed through the network? Alternatively, find a set of *src–sink* paths that carry the maximum units of flow through the network. The Minimum-Cost Maximum-Flow (MCMF) problem asks the question: Among all maximum flows f find the one which has the minimum total cost $Cost(f)$.

5 Solving AMAPF-makespan

In (Yu and LaValle 2013a), the authors reduced the T-steps AMAPF problem, i.e., the one that allows any agent to do at most T actions, to a maximum flow (MF) problem. Specifically, it was proven that a T-steps AMAPF problem has a solution *iff* the reduced MF problem has a flow equals 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.

5.1 Network Construction

Consider a T-steps AMAPF instance with the graph $G = (V, E)$. The reduced network is constructed by first creating $2T + 1$ copies of V and mark them as follows: 0, 1_{in}, 1_{out}, 2_{in}, 2_{out}, ..., T_{out} (see Figure 3). 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 h_{in} and copies h_{out} for $h = 0, 1, ..T$ as *in-* and *out-* copies, respectively. The copied vertices of the original graph, along with a source node *src* and sink node *sink*, form the nodes of the network. We call the set of *src* and *sink* nodes, as *non-layer* nodes, and the remaining nodes as *layer* nodes. Each *layer* node is identified by (v, h), where v is the original graph vertex and h stands for the copy number, alternatively referred to as height. Indeed, the node (v, h_{in}) (as well as (v, h_{out})) corresponds to the state of an agent located at a vertex v at time step h. Analogously, whenever we say the nodes *at* or *with* vertex v, we mean the nodes that have v as their vertex i.e., $\{(v, 0)\} \cup \{(v, h_{\{in,out\}}) | h \in \{1, \ldots, T\}\}$.

Then we add the following arcs to the network. For each arc $e(u, v)$ in the original graph, we connect the nodes (u, h_{out}) and $(v, (h + 1)_{in})$ for $h > 0$, and $(u, 0)$ and $(v, 1_{in})$ in the network. These arcs correspond to the move actions, and we call them the *move* arcs. Then, we add the *wait* arcs that connect the nodes (u, h_{out}) and $(u, (h + 1)_{in})$ for $h > 0$ and the nodes $(u, 0)$ and $(u, 1_{in})$. Additionally, we add the arcs between the nodes (u, h_{in})

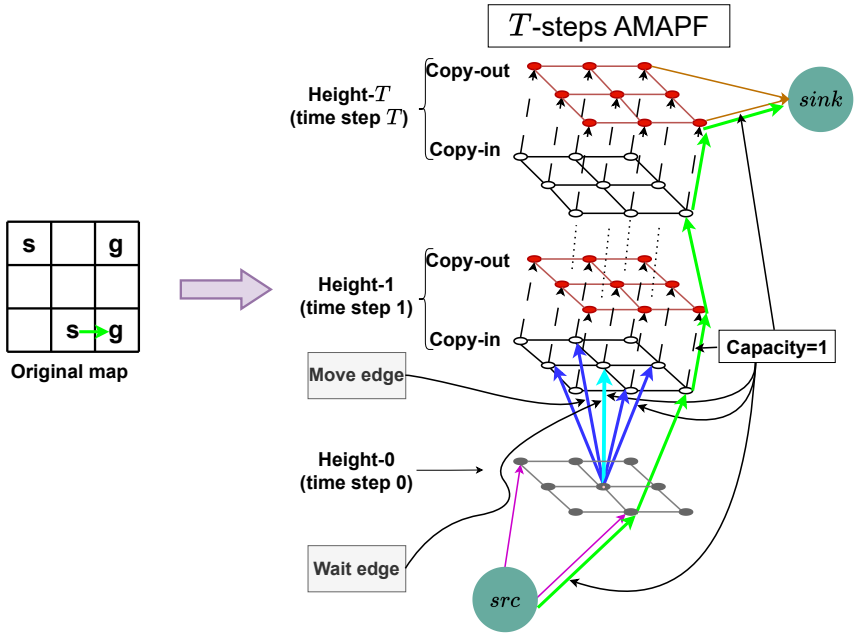

Fig. 3. Example shows the reduction to network flow problem for AMAPF problem. Left: the original problem (a grid map with 2 start cells and 2 goal cells) and right: the reduction of T-steps AMAPF problem to MF problem. Blue and cyan directed arcs should be added analogously to all nodes in *out* copies (they are deleted for clarity). Green arcs show how a path be reflected on the network.

and the (u, h_{out}). These arcs do not denote any action but are added to forbid any two *src*–*sink* paths in the network from sharing any node and, therefore, to avoid node-collisions. We call them the *restricting* arcs. Finally, we connect *src* to all nodes $(u, 0)$ where u is a start vertex, and connect *sink* to nodes (v, T_{out}) where $v \in \mathcal{G}$ is a goal vertex. The capacity of all arcs is fixed to be 1. Note that every *src*–*sink* path necessarily starts from a start-vertex copy at layer 0 and reaches a goal-vertex copy at layer T, because *src* connects only to layer-0 start copies and *sink* is reachable only from layer-T goal copies.

Hereinafter, whenever a path in a network is mentioned we mean the *src*–*sink* 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 to the *move* or *wait* arcs and using the *restricting* arcs to connect between them. Similarly, a path in the network is matched to a plan for an agent where the *move*/*wait* arcs are matched to move/wait actions. See Figure 3 for a self-contained example. It was proven that there are no shared nodes in any two paths in the network (that form the solution to the MF problem), which implies that all matched plans have no node-collisions. An edge-collision may happen if two paths pass the arcs $((u, h_{out}) \rightarrow (v, (h+1)_{in}))$ and $((v, h_{out}) \rightarrow (u, (h+1)_{in}))$ as these two different arcs in the network refer to the same edge in the original graph. However, using the approach suggested in (Liu et al. 2019), these collisions can be eliminated in the following fashion. Instead of two conflicting agents moving to their next vertices, they swap plans and continue moving by the other's plan. As a result, the AMAPF solution can be obtained by finding the maximum number of paths (maximum flow) in the described network.

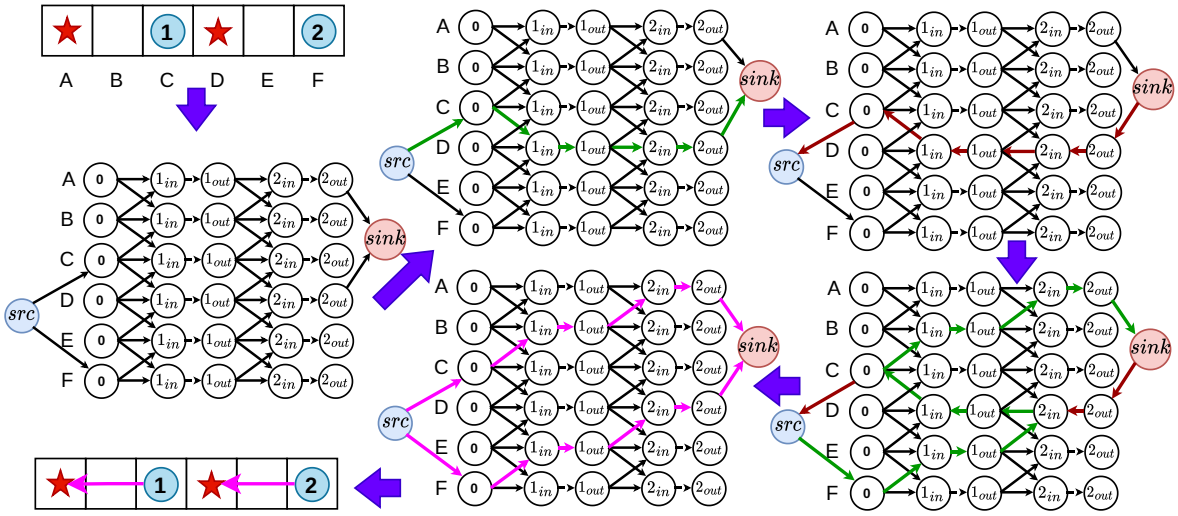

Fig. 4. Example shows how the Maximum Flow algorithm works on the reduced-from-AMAPF networks step by step, and how the solution can be converted back to AMAPF solution. In the first image (in the upper left corner), the original AMAPF problem with two agents is shown. Initially agents occupy cells C and F and they need to go to cells A and D. In the next image we show the matching reduced network with maximum time step $T = 2$, where we need to solve MF problem. Next, assume that the green path is found from the source node *src* to sink node *sink* (note that there are many paths leading from source to sink, and any one can be chosen). This path currently matches with the AMAPF plan in which the agent (1) should go to the goal at cell D. Next, we add a one unit flow to the arcs of the path (i.e., to arcs $((src), (C, 0)), ((C, 0), (D, 1_{in}))$, $((D, 1_{in}), (D, 1_{out})), ((D, 1_{out}), (D, 2_{in})), ((D, 2_{in}), (D, 2_{out})), ((D, 2_{out}), (sink))$), and then reverse these arcs to be used in next steps. Next, another green path is found from *src* to *sink* which passes from some previously reversed arcs as illustrated in the graph. We again add one unit flow to the arcs of the path which were not-reversed and remove the flow from the arcs of the path which were reversed $(((D, 1_{in}), (D, 1_{out})), ((D, 1_{out}), (D, 2_{in})))$. Next, no new path can be found from *src* to *sink*, so we mark all arcs with positive flows and form the AMAPF plan from them. The resulting plan is: For agent (1), move from C to B, then move from B to A. For agent (2), move from F to E, then move from E to D.

5.2 Solving Maximum Flow Problem

A huge number of algorithms with different complexities have already been published in the literature to solve the general MF problem (Cruz-Mejía and Letchford 2023). For the special case when all arcs of the network have a capacity of 1, the best-known algorithm has complexity $O(\min(|V|^{2/3}, |E|^{1/2})|E|)$ (Goldberg, Hed, et al. 2017) (where |V| and |E| are the number of nodes and arcs in the network, respectively). However, as in our special problem the flow (the number of *src*–*sink* paths) is bounded by the value of the number of agents k, running simple Ford–Fulkerson algorithm (Ford and Fulkerson 1956) (as suggested in (Yu and LaValle 2013a)) yields an even faster algorithm with a complexity of $O(k|E|)$. Therefore, taking into consideration its simplicity (ability to be improved as we will see later) and best efficiency, Ford–Fulkerson was chosen to solve the MF problem on the reduced networks from AMAPF instances.

We refer the reader to the original paper for the detailed description of Ford–Fulkerson while briefly describing 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 *src* to *sink* in the network. Second, we reverse all arcs 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). See Figure 4 an illustrative example.

In the worst case, the value of T needed to solve an AMAPF problem can be up to $k + |V| - 2$, as shown by (Yu and LaValle 2013a). This implies that the network size may be quadratic in the number of vertices of the original graph i.e., $|V|_{maximum} = |V|(2T_{maximum} + 1) = |V|(2(k + |V| - 2) + 1) = 2|V|k + 2|V|^2 - 3|V|$. For instance, for solving an AMAPF instance with 100 agents on a 400×400 grid, the network size may be larger than 5.12×10^{10} nodes, where a path should be found 100 times. Therefore, finding a *src–sink* path may become a bottleneck when solving AMAPF instances involving large input graphs even when using simple linear algorithms like Breadth First Search (BFS). To this end, we propose an algorithm, dubbed Bulk Search, that takes advantage of the specific structure of the reduced networks to find the *src–sink* paths faster and, as a result, is able to solve AMAPF problems faster.

5.3 Bulk Search Algorithm

First, we present some definitions to use in the algorithm description. Recall that a *layer* network node (v, h) is defined by the vertex in original graph, v, and the height (copy number) h (a higher node means a higher copy number, and h_{out} is higher than h_{in}). Hereinafter, for simpler explanation we will distinguish between h_{in} and h_{out} only when it is required, otherwise we will use the symbol h. In particular, a node (v, h) can be viewed as a node with the ingoing arcs of (v, h_{in}), and the outgoing arcs of (v, h_{out}). We define a *connected-sequence* as a sequence of nodes with the same vertex in which we can reach the last node from the first node using only *wait* and *restricting* not-reversed arcs, and it cannot be extended in either side (i.e., it has the maximal length). An example is shown in Figure 5. Initially, for each vertex $v \in V$ of the original graph, we have only one connected-sequence $(v, [0, T_{out}])$ i.e., the one that starts with the node with height 0 and ends with height T_{out} (shown by blue crossbars). After a path (green path) is found, its arcs are reversed (red arcs in the lower graph). As a result, some connected-sequences disconnect which leads to several connected-sequences at the same vertex (see the lower network). We use these connected-sequences (along with *src* and *sink* nodes) to carry the search states in our algorithm as will be seen next.

According to Ford–Fulkerson algorithm, the *src–sink* path does not need to be the shortest but can be any path. Therefore, the task is to only determine whether the *sink* is reachable from *src* using the arcs of the network, and by which arcs it is reached (so these arcs can be reversed in the second step of the algorithm). This task is known as graph traversal problem. The crucial observation in these AMAPF networks, that whenever a node is reached in a connected-sequence, all the next nodes in the same connected-sequence can be reached, too. We use this observation to develop the Bulk Search (BS) algorithm enhancing the general graph traversal algorithm.

We provide a brief overview of the general traversal algorithm. The algorithm begins by inserting the start nodes (in our case, solely the source node *src*) into a set called the **OPEN** set. Then, we iteratively pop a node n from OPEN and *expand* it by inserting all nodes directly reachable from n via an edge in the graph into the OPEN set. To avoid redundant processing, each node is inserted into OPEN only once—a **CLOSED** set tracks already expanded nodes for this purpose. The algorithm continues this process until either the OPEN set is exhausted or the goal node (*sink*) is popped from OPEN. Once the goal node is reached, the path from *src* to *sink* can be reconstructed by *backtracing* through the nodes that led to its insertion into OPEN.

Bulk Search Idea. The enhancing idea is described as follows. Instead of inserting only the directly-connected nodes from the popped node n, we will insert a bulk of nodes at once. In particular, this bulk consists of all nodes which can be reached from the node n in the same connected-sequence. More precisely, whenever a node $n(v, h)$ is reached and to be inserted into OPEN, we directly insert the bulk of nodes $\{(v, x)|x = \{h, h + 1, \ldots, h_u\}\}$ where h_u is the upper bound copy of the connected-sequence where n is located. In practice, we do not enumerate all nodes in the bulk; instead, we store the bulk compactly (as an interval within each connected-sequence) and keep its minimum-height node as a representative for ordering in OPEN.

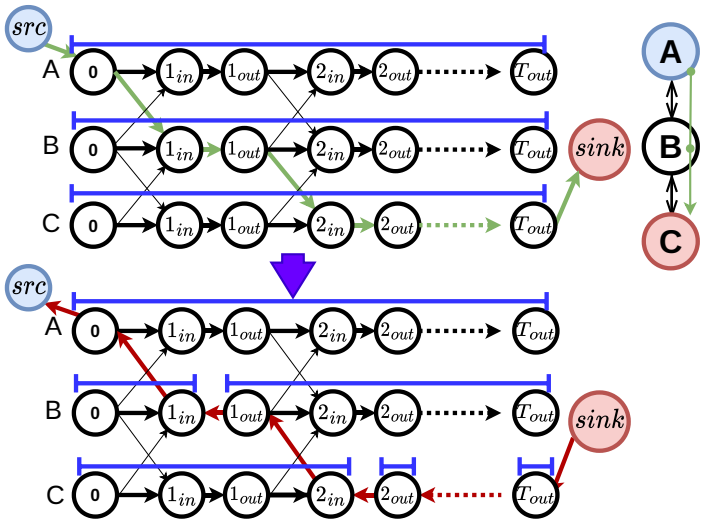

Fig. 5. Example showing connected-sequences on the network. Blue crossbars denote the connected-sequences on each vertex. Initially, we have the connected-sequences as shown in the upper figure (one connected-sequence at each vertex). After a path (green one) is found and its arcs are reversed, the connected-sequences are divided as shown in the lower figure (for each reversed *wait* or *restricting* arc, the connected-sequence is divided into two connected-sequences).

However, in order for this idea to be useful and can improve the efficiency of the general algorithm, three questions should be answered. The first question: How can the nodes of the bulk be defined fast? That is, is it faster than passing through all nodes in linear time? The second question: How can inserting the nodes of the bulk into the OPEN set be faster than inserting them individually one by one? The third question: How can expanding the nodes of the bulk be done faster than expanding the nodes individually. These three questions are important to address so it is guaranteed that the algorithm is not just equivalent to the standard algorithm where all of these nodes will be generated, inserted and expanded individually during the standard search. In other words, any linear pass on the nodes of the bulk will nullify the advantage of the suggested idea.

We answer the first question (How can the nodes of the bulk be defined fast), as follows. As stated previously, the bulk of nodes generated from a node n consists of the sequence of nodes starting from n itself until the last node in the connected-sequence where n is located. Therefore, to define the bulk, we only need to define the last node of the connected-sequence (the nodes between them are then easily generable when needed). During the Ford–Fulkerson algorithm, each reversed arc splits an existing connected-sequence into two connected-sequences. Consequently, maintaining the connected-sequences can be performed in time linear in the number of arcs along the found paths. A connected-sequence can be represented by three identifiers: its vertex, the height of its lowest copy, and the height of its highest copy. By storing the connected-sequences in suitable data structures, bulk nodes can be identified efficiently, e.g., in $O(1)$ or $O(\log cs)$ time, where cs denotes the number of connected-sequences, depending on the chosen data structures.

The answer to the second question (How can inserting the nodes of the bulk into the OPEN set be faster than inserting them individually one by one) is that instead of explicitly inserting all nodes of the bulk into OPEN, we insert one customized state which precisely, compactly and implicitly defines the nodes of the bulk. Similarly to connected-sequences, the bulks can be also defined by three identifiers; the vertex of the bulk's nodes, the height of the lowest copy in the bulk and the height of the highest copy in the bulk. In Figure 6, the bulk of

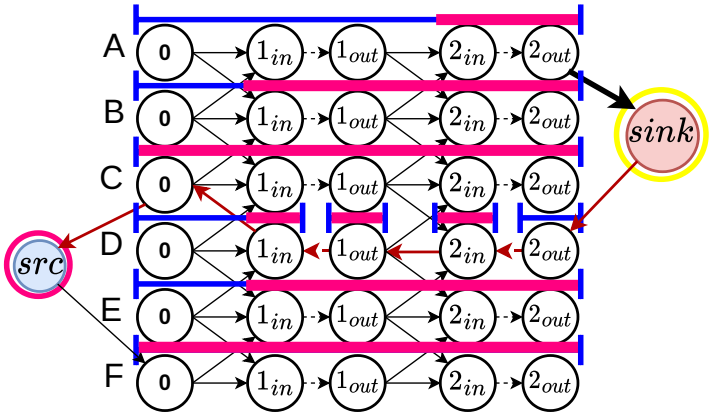

Fig. 6. Example shows the expanded states while searching a path from *src* to *sink*. Blue crossbars denote the connected-sequences. All 23 single nodes covered by pink lines will be expanded using naive graph-traversal search. On the other hand, each pink line is considered as one bulk state and therefore only 8 bulk states will be expanded in the proposed Bulk Search algorithm.

nodes generated from $(F, 0)$ equals the connected-sequence $(F, [0, 2_{out}])$, where the one generated from node $(E, 1_{in})$ is the bulk $(E, [1_{in}, 2_{out}])$ which is a subsegment of the connected-sequence $(E, [0, 2_{out}])$. Inserting these customized states only into OPEN equals inserting all the individual nodes of the inserted bulk but in a much more efficient way.

To answer the third question (how can expanding the nodes of the bulk be done faster than expanding each node individually), we note that it is not necessary to pass through all bulk individual nodes and insert their direct neighbors in the network into OPEN, but the following can be done. We define all connected-sequences where at least one node can be reached from this bulk. For each one of these connected-sequences, we search for the node with the lowest height, and store it (let it be in a temporary set). From each of these stored nodes, we generate a bulk-state (until the upper bound of the enclosing connected-sequence) and insert it into OPEN. This is enough because in this way, it is guaranteed that all nodes which can be reached from the nodes of the input bulk state, will be included in the inserted bulks. More formally, 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 reach at least one node (from a node (v, x) where $x \in \{h_l, \ldots, h_u\}$). Then, we find the accessible node with the minimum height in each of these connected-sequences, generate a bulk from it, and insert it into OPEN. As the number of connected-sequences is much fewer than the number of individual nodes, this leads to a profound reduction in the number of generated states.

The last important detail is how to avoid inserting redundant nodes into OPEN. As we now have bulks (intervals) of single nodes instead of nodes, detecting the redundant nodes needs more effort than exact matching i.e., two intervals of two different states may not be identical but may share a sequence of nodes. We can do this procedure as follows. Let us assume that we want to insert the bulk state s with the interval $[h_l, h_u]$ which is located in a connected-sequence cs. We take the union $it_{inserted}$ of the intervals of all states in CLOSED and OPEN which are located in cs, let it equal $it_{inserted} = [h_l^{inserted}, h_u]$. Note that all generated bulk states in the same connected-sequences have the same upper-bound, so both s and the $it_{inserted}$ have the same upper-bound h_u. Now, if h_l is included into $it_{inserted}$, then s can be discarded. That is, all individual nodes of s are either already expanded (the part in CLOSED) or already inserted for future expansion (the part in OPEN). Otherwise when

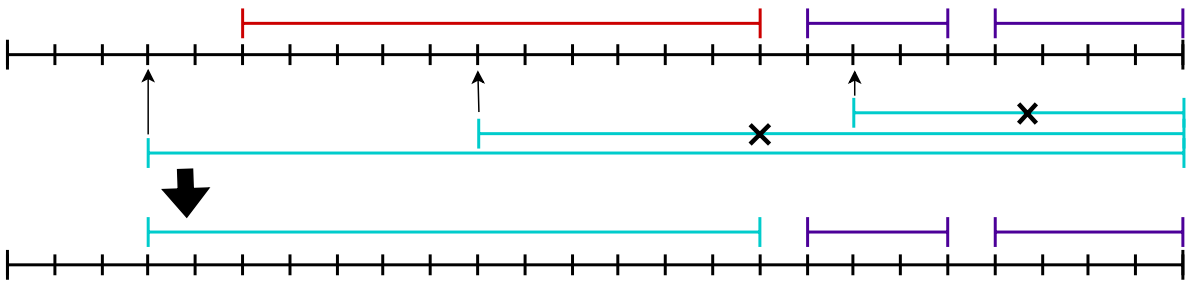

Fig. 7. A scheme shows the states inside OPEN and CLOSED sets for a given connected-sequence, and the cases when a new interval (a bulk state) in the same connected-sequence needs to be inserted. The black interval denotes a connected-sequence, where each tick refers to a single node in the network. The violet and red intervals denote the states which have been already inserted into CLOSED and OPEN, respectively. The cyan intervals denotes the interval which need to be inserted into OPEN. The two upper cyan intervals should not be inserted into OPEN as all there single nodes are already inserted (either in CLOSED or OPEN). The lowest cyan interval can be inserted, removing the one currently inserted into OPEN (so its single nodes are not expanded more than once), and being trimmed to not cover the CLOSED intervals (as they have been already expanded).

$h_l < h_l^{inserted}$, the states which are located in *cs* in OPEN are removed, and s is inserted instead. However, s should be inserted with a shrunk interval so it does not intersect with the union of intervals of the states in CLOSED (see Figure 7 for an example with a visual illustration). The technical operations which are needed for this procedure are: Finding the state in OPEN in *cs*, finding the state with the lowest lower-bound in *cs* from CLOSED, removing a state from OPEN operation and inserting a state into OPEN. All of these four operations can be implemented efficiently as will be shown later.

Example. Figure 6 depicts an example of how BS uses bulks. Assume that we have the shown network, and we need to find a path from *src* to *sink*. Initially, we have the only node *src* in OPEN set. From *src*, we can only reach the node $(F, 0)$ by a direct arc (note that when the node is not inside the network, i.e., nodes *src* and *sink*, no bulks can be generated). The node $(F, 0)$ is located inside the connected-sequence $(F, [0, 2_{out}])$. Therefore, from node $(F, 0)$, we can generate the bulk which extends from the node $(F, 0)$ itself until the last node of the enclosing connected-sequence $(F, 2_{out})$, i.e., bulk=$(F, [0, 2_{out}])$. As a result, *src* node is popped from OPEN and the generated bulk $(F, [0, 2_{out}])$ is inserted into OPEN. In the next step, we expand the customized node $(F, [0, 2_{out}])$. F vertex has one neighbor E in the original graph. At vertex E, there is one connected-sequence $(E, [0, 2_{out}]$ only. The minimum node (the least-height) which can be reached in this connected-sequence from $(F, [0, 2_{out}])$ is the node $(E, 1_{in})$. $(E, 1_{in})$ in its turn generates the bulk $(E, [1_{in}, 2_{out}])$ which is then inserted into OPEN. In the next step, we expand $(E, [1_{in}, 2_{out}])$. E has two neighbors F and D in the original graph. In the connected-sequences of vertices F and D, the minimum nodes which can be reached are $(F, 2_{in})$ and $(D, 2_{in})$, which generate the bulks $(F, [2_{in}, 2_{out}])$ and $(D, [2_{in}, 2_{in}])$, respectively. The bulk $(F, [2_{in}, 2_{out}])$ is fully included in the previously-expanded bulk $(F, [0, 2_{out}])$, and hence should not be inserted again into OPEN. The bulk $(D, [2_{in}, 2_{in}])$ is inserted and the same steps are repeated until the goal node *sink* is reached.

Pseudocodes. The pseudocode of the main body of the Bulk Search is presented in Algorithm 1. The input of the algorithm is the network reduced from the MAPF instance. Here, we note that the network does not need to be given in explicit form. It is enough to input the original MAPF graph, the connected-sequences and the reversed arcs in the network, as all other arcs and nodes can be known implicitly and quickly when needed. The algorithm starts by adding the *src* node to the OPEN set (Line 3). Analogously, to the general graph traversal

Algorithm 1: Bulk Search (BS)

Input : Network $N = (\mathrm{G}, cap, src, sink)$ induced by G, T, and the current set of reversed arcs
Output: A src–$sink$ path if one exists (otherwise `failure`)

```
1  OPEN ← ∅, CLOSED ← ∅
2  goal_state ← ⊥
3  insertToOpen(src)
4  while OPEN ≠ ∅ do
5      pop a state s from OPEN with minimum height
6      insert s → CLOSED
7      if s = sink then
8          goal_state ← s
9          break
10     succ ← getSuccessors(s, N)
11     foreach s' ∈ succ do
12         insertToOpen(s')
13 if goal_state ≠ ⊥ then
14     return reconstructPath(goal_state)
15 return failure
```

algorithms, while OPEN is not empty we do the following. We pop a state s from OPEN, and expand it. Here s, heuristically, is chosen to be the search state with the minimum height, where *sink* is assumed to have the height -1 (that is, it is given the highest priority to be popped) and the height of a bulk state is the lower bound of its interval (Line 5). This is not necessary for the completeness or validity of the algorithm, and may add more requirements on the data structure used for OPEN. However, it was empirically noticed that it helps to reduce the number of expansions. That is, the bulk states with smaller lower-bounds may generate new states with smaller lower-bounds than the ones which are currently in OPEN (and therefore replace the current ones with bigger states (i.e., with longer intervals) avoiding expanding multiple states). In Line 10, we generate the successors of the popped state s using a special function `getSuccesors` (Algorithm 2) which will be explained later. Then we insert them to OPEN using the function `insertToOpen` (Algorithm 3) where several conditions are checked in order to avoid inserting redundant states in OPEN or insert states which are already inserted into CLOSED, as explained previously. The main loop breaks when the sink is popped from OPEN (Line 7).

The function `getSuccessors` (Algorithm 2) generates the successors of the popped search states. It takes as input the state along with the network (again, the network can be given in an implicit form). In this function, we first (Lines 1–16) generate the successor nodes in the network, so that at each reachable connected-sequence we generate only the minimum-height node. At the end of the function (Lines 17–23), we extend these generated network nodes (the layer ones) to the bulk states and return them. In the first case (Line 2), when the input state is simply the source node *src*, the successor nodes are the direct neighbors of the node *src* in the network N (Lines 3,4). In the other case, i.e., when the input state is a bulk state $s(v, [h_l, h_u])$, we generate the successors as follows: If there are reversed arcs connected to any of the nodes at the interval bounds h_l, h_u of the input state, i.e., (v, h_l), (v, h_u), then we generate all nodes connected to these nodes by reversed arcs in N (Lines 5–10). Then, if s is connected to *sink* node, we generate it (Lines 9,10). Lastly, we generate the minimum accessible node in any connected-sequence at the neighbor vertices of v using the operations coded in Lines 11–16.

Algorithm 2: Generating successors in BS

Function `getSuccessors`$(s, N = (\mathrm{G}, cap, src, sink))$**:**

1 $succ\_nodes \leftarrow \emptyset$
2 **if** *s is a non-layer node state* **then**
// s is a node n
3 **foreach** *outgoing arc* (n, n') *in* N **do**
4 insert $n' \rightarrow succ\_nodes$
else
// the input state is a bulk state, i.e., $s = (v, [h_l, h_u])$
5 **foreach** *out-neighbor node* m *of node* (v, h_l) *by a reversed arc* **do**
6 insert $m \rightarrow succ\_nodes$
7 **foreach** *out-neighbor node* m *of node* (v, h_u) *by a reversed arc* **do**
8 insert $m \rightarrow succ\_nodes$
9 **if** *s is connected to sink* **then**
10 insert $sink \rightarrow succ\_nodes$
11 **foreach** *connected-sequence* $cs(u, [h_l^u, h_u^u])$ *with vertex u is a neighbor of v* **do**
12 $h_{from} \leftarrow$ the height of the minimum *out*-copy in $[h_l, h_u]$
13 $h_{to} \leftarrow$ the height of the minimum *in*-copy in $[h_l^u, h_u^u]$
14 **if** $h_{from} + 1 \leq h_u^u \wedge h_{to} - 1 \leq h_u$ **then**
15 $h_{min} \leftarrow \max(h_{to}, h_{from} + 1)$
16 insert $(u, h_{min_{in}}) \rightarrow succ\_nodes$
17 $succ\_states \leftarrow \emptyset$
18 **foreach** $(n, c) \in succ\_nodes$ **do**
19 **if** *n is a non-layer node* **then**
20 insert $n \rightarrow succ\_states$
else
// n is a layer node (v, h)
21 $h_{max} \leftarrow$ the upper bound of the connected-sequence where node n is located
22 insert $(v, [h, h_{max}]) \rightarrow succ\_states$
23 **return** $succ\_states$

Function `insertToOpen` (Algorithm 3) shows the cases where an input state s is eligible to be inserted into OPEN set. In the first case when s is a non-layer node (Lines 1–4), s is discarded when it is exactly found in any of OPEN or CLOSED sets (Lines 2–3), or inserted otherwise (Line 4). In the other case (Lines 5–15): 1) s is discarded when it is covered totally by a state in OPEN (case at Line 8), 2) another state from OPEN is removed as it is totally covered by s (case at Line 9), 3) s is discarded if it is totally covered by a state in CLOSED set (case at Line 13), 4) a trimmed part of s is inserted into OPEN, as it is partially covered by a state in CLOSED set (case at Line 14), 6) if s is not discarded or trimmed before, it is finally inserted into OPEN (case at Line 15).

5.3.1 Theoretical analysis. We next prove the completeness of the algorithm, and show the overall gain in performance of BS algorithm.

Algorithm 3: Inserting a state into OPEN in BS

```
Function insertToOpen(s):
    if s is non-layer state then
        if s ∈ {OPEN ∪ CLOSED} then
            return
        else
            insert s → OPEN
    else
        // state s is a bulk state s = (v, [h_l, h_u])
        x^open(v, [h_l^open, h_u^open]) is the state in the same connected-sequence of s from OPEN
        if x^open exists then
            if h_l^open ≤ h_l then
                return  // no insertion
            else
                remove x^open from OPEN
        x^closed(v, [h_l^closed, h_u^closed]) is the state in the same connected-sequence of s from CLOSED with the
          minimum lower-bound
        if x^closed exists then
            if h_l^closed ≤ h_l then
                return  // no insertion
            else
                insert (v, [h_l, h_l^closed − 1]) → OPEN
        else
            insert (v, [h_l, h_u]) → OPEN
```

Theorem 1. *Bulk Search is a complete algorithm i.e., it always finds a path from src to sink if it exists.*

Proof. As the algorithm explores all generated nodes (unless the goal node *sink* is found), i.e., without any use of any pruning techniques, we only need to prove the completeness of the successors-generator function. In the used successors-generator function `getSuccessors`, there are two phases. In the first phase, we generate the least-height node in any reachable connected-sequence from the input search state. In the second, phase, we extend the generated nodes to the upper bound of the connected-sequences (and compress them in bulk states). As a result, at least all reached nodes from all nodes of the input search state are generated. □

In the BS algorithm, the search states depend mainly on the number of connected-sequences. That is, any reached node is directly extended to the upper bound of the enclosing connected-sequence; therefore, no further search states within the same connected-sequence at a higher level are generated. On the other hand, in standard algorithms, any node in the network may be generated as a separate search state. Next, we show how much the total number of connected-sequences created across all max-flow augmentations can be substantially fewer than the number of nodes in the network

Search Space Reduction. Initially, the number of connected-sequences equals $|V|$, i.e., the number of vertices in the original graph. After each augmenting path is found and its arcs are reversed, at most $2T$ arcs become

reversed; consequently, the number of connected-sequences can increase by at most $2T$ (each reversed *wait* or *restricting* arc splits one connected-sequence into two). Therefore, over all *src*–*sink* searches for the k agents, the total number of connected-sequences that can appear is bounded by

$$\sum_{i=1}^{k} \big(|V| + 2T(i-1)\big) = k|V| + Tk(k-1).$$

Without compression into connected-sequences, the number of explored nodes can be as large as $2k|V|T$ (since, for each agent, the entire network of $2|V|T$ nodes may be explored). Hence, the compression yields a theoretical reduction in the number of search states by a factor of $\min(|V|/k, T)$, i.e.,

$$\big(k|V| + Tk(k-1)\big) \cdot \min(|V|/k, T) \;\leq\; 2k|V|T.$$

Fortunately, the factors $|V|/k$ and T are positively correlated: the denser the instance (larger k relative to $|V|$), the smaller the average makespan. Thus, in difficult cases where $|V|/k$ is large, T is also large, and the reduction factor becomes large as well (if T dominates, then the problem becomes close to searching in the original MAPF graph). Conversely, when $|V|/k$ is small (agents are densely distributed), T is typically small as well (recall that the problem is anonymous and any agent can go to a near goal vertex). In such cases, the original search space is already small and the problem is comparatively *not hard*.

Although a full precise complexity bound for the BS algorithm is not provided (since there is no limit on revisits within each bulk node), empirically this reduction is substantial, enabling 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 the last section.

6 Solving AMAPFD-SOC

To obtain SOC-optimal solutions to AMAPFD problems we, as before, introduce a reduction of AMAPFD-SOC to a variant of the network flow problem and then design an efficient solver of this variant. As our reduction is novel we, first, show that it is provably correct.

6.1 Reduction to Minimum-Cost Maximum-Flow problem

We now introduce a specific flow network that can be used to optimally solve AMAPFD-SOC. The construction of the network is based on the previous approach of converting AMAPF to MF problem. However, there are several modifications needed for the new reduction: First, the arcs should have accurately-defined costs (for us to be able to minimize SOC and not makespan as before) and second, additional nodes and arcs are needed to capture the fact that the agents disappear at goals.

6.1.1 Network Construction. We use the same network created for reducing T-steps AMAPF to MF problem (with the same capacities for the arcs and costs 0), with the following differences (as shown in Figure 8). First, the network height is now unlimited. That is, there is now an infinite number of copies of the original graph. Second, the sink node is now connected to the goal vertices from all copies through new nodes called *hub*s. More specifically, for each AMAPFD goal vertex $g \in \mathcal{G}$ we add a *hub* node hub_g to the network (shown in ocean blue color in Figure 8). We add directed ingoing arcs to each hub node hub_g (orange arrows in Figure 8) from the nodes at vertices g in the copies *out* of the layers (i.e., nodes $(g, h_{out}) : h \in \{0,\ 1,\ \ldots,\ \infty\}$), with cost equals the height of the node (i.e., h) and capacity equals 1. We call these arcs as *goal arcs*. We also call the nodes at goal vertices as *goal nodes*. Finally, we add an arc from each *hub* node to the network sink node *sink* with zero cost and one capacity. We call these arcs the *sink arcs*. We also call the set of *src*, *sink* and *hub* nodes, as *non-layer* nodes, and the remaining nodes as *layer* nodes. The construction of the network is finished.

The same matching between the agents' plans in AMAPF instances and the *src–sink* paths in the reduced networks can be applied here between the agents' plans in AMAPFD instance and the *src–sink* paths in the newly reduced networks. The only addition is the matching between passing a *goal* arc in the network and the *disappearing* action in AMAPFD problem. The matching in the objective between the MCMF problem on the reduced networks and the AMAPFD-SOC instance is also direct: Problem MCMF searches for 1) the maximum flow, 2) the minimum cost of it i.e., the minimal sum of costs of *src–sink* paths. This can be matched to 1) the maximum number of agents' plans, 2) the minimum total sum of costs of the agent's plans. This leads to the following theorem.

Theorem 2. *AMAPFD has a solution iff the MCMF problem on the corresponding network has a solution. Moreover, optimal solution of MCMF problem can be matched to an optimal (with respect to SOC) solution of AMAPFD problem.*

There is another equivalent MCMF reduction for AMAPFD, shown in Figure 16. In our work, we adopt our first formulation, which accumulates costs on the *goal* arcs (rather than distributing them over move/wait arcs), because it is more convenient for developing and proving the algorithm and pruning techniques introduced below when the inter-layer arcs have zero cost. On the other hand, the second reduction is easier for standard solvers, as mentioned in the Experimental Evaluation section.

6.2 Solving Minimum-Cost Maximum-Flow On The Introduced Networks

MCMF is a well-known problem with dozens of already published solvers with different complexities (Cruz-Mejía and Letchford 2023). However, using any of them to directly solve MCMF on our networks is likely to yield high runtime as the size of the networks is large. Similarly to solving MF problem on the reduced network, the *Successive Shortest Paths Algorithm* (Edmonds and Karp 1972) (SSPA) algorithm (which consists of repeatedly solving single path finding problems) is chosen to solve the introduced MCMF problem as it is one of the simplest and most efficient algorithms to work on such networks (e.g., the network given in Figure 16 when the capacities are ones, the costs are ones and zeros, and the flow is limited by the number of agents).

For our networks (i.e., when the capacities are all ones), the algorithm consists of two repeated steps. The first step is to find the shortest (minimum cost) path p from *src* to *sink*. In the Second step, the network is modified according to the newly found path as follows. For each arc $e(u, v)$ with cost c in p, we remove the arc e from the network and add the arc (v, u) to the network with cost $-c$. We keep repeating these two steps until no path can be found from *src* to *sink*. The flow of the arcs can be calculated as follows. For each reserved arc (v, u) in the final network, this means that the arc (u, v) in the original network has the flow 1. The plans for the agents can then be converted from the arcs which have positive flow. See Figure 9 for a full example.

6.2.1 Single Path Finding problem (Generalized Bulk Search algorithm). Originally, our flow networks have infinite number of layers. However, the upper bound on SOC for AMAPF is established in (Yu and LaValle 2013a) and can be used in our case as well to limit the number of the layers. Still, this bound is relatively large and in practice the size of the networks is large. Thus, in both cases, searching for the shortest *src–sink* paths becomes a bottleneck.

The typical shortest path algorithm Dijkstra is not applicable here because the network may include negative arcs. Instead, a more general *label-correcting* graph-search method (Ahuja et al. 1993) is typically employed, in which nodes may be re-expanded (i.e., *corrected*) whenever a lower cost (label) is found.

A well-known example of a label-correcting approach is the Bellman–Ford algorithm. Although Bellman–Ford solves the problem, it performs exactly $O(\text{VE})$ expansions (where V and E correspond to the number of nodes and arcs of the network), which is impractical for large networks. For example, taking the upper bound for time horizon T, $T_{max} \leq \frac{(k-1)(k-2)}{2} + V$ for AMAPF-SOC from (Yu and LaValle 2013a), for a graph with 400×400 nodes and $k = 100$ agents, V may have more than 5.27×10^{10} nodes. Therefore, the number of expansions required by Bellman–Ford, which is $O(\text{VE})$, becomes infeasible at this scale.

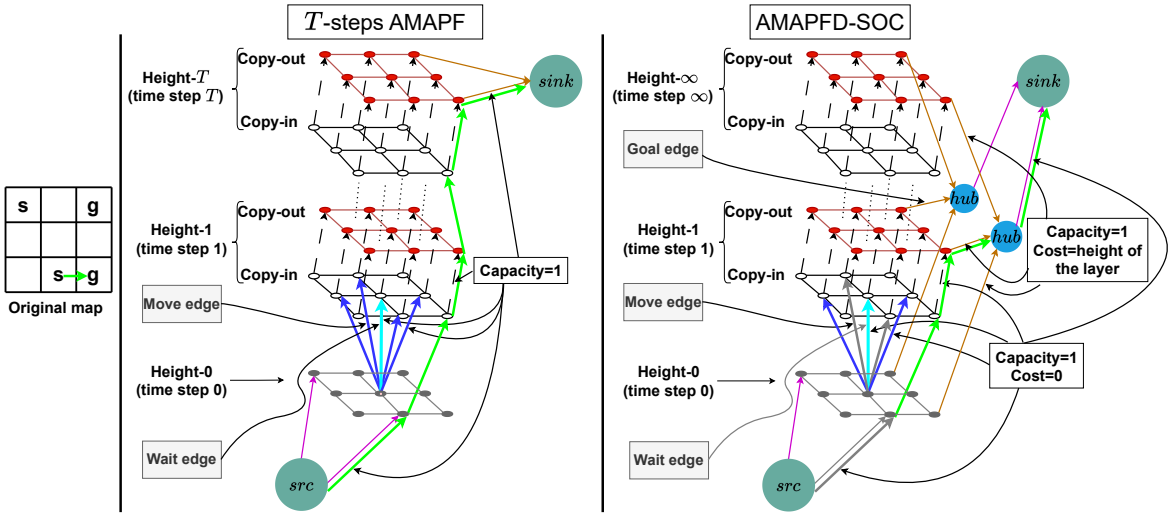

Fig. 8. Example shows the reduction to network flow problem for both AMAPF and AMAPFD problems side-by-side. Left: the original problem (a grid map with 2 start cells and 2 goal cells), middle: the reduction of T-steps AMAPF problem to MF problem, right: the reduction of AMAPFD-SOC problem to MCMF problem. Blue and cyan directed arcs should be added analogously to all nodes in *out* copies (they are deleted for clarity). Green arcs show how a path be reflected on the network.

A common improvement is the Shortest Path Faster Algorithm (SPFA), widely used in minimum-cost maximum-flow (MCMF) settings. SPFA's main advantage over standard Bellman–Ford is its use of a queue to sort and expand nodes (i.e., it works like BFS with re-expansions). In practice, SPFA runs in $O(E)$ time on average; however, its worst-case complexity remains $O(VE)$. As a result, both algorithms are inapplicable in our case.

To this end, in order to effectively solve the path finding problems in the reduced networks (and thus solve MCMF overall), we propose three techniques as follows.

(1) We introduce Generalized Bulk Search (GBS) algorithm by extending the Bulk Search algorithm so that the computational complexity of the single-path finding problem depends primarily on the number of vertices/edges in the original map, rather than on the number of network nodes/arcs. This extension requires a single condition on the arc costs, under which the algorithm becomes applicable to networks of similar structures.
(2) Exploiting the specific combination of the structure, capacities and costs of the reduced networks, we introduce:
   (a) an **admissible heuristic** that accelerates the search and enables termination upon reaching the goal, even in the presence of negative-cost arcs; and
   (b) an **enhanced pruning rule** that identifies and discards unproductive search states prior to expansion.

In GBS, BS is extended to find the shortest path in weighted graphs. The same workflow is used, but several additions and modifications are applied as follows (see Algorithm 4 where the blue text denotes the changes from BS and the red text denotes the further optimizations developed later specifically for our reduced networks).

GBS search states. Similar to BS algorithm, in GBS there are two types of states, node states which consist of one network node, and bulk states which consist of interval of network nodes. Additionally to the search node, the search states of GBS should have costs. The cost of a state s, denoted by $g\_cost(s)$ (inherited from A*

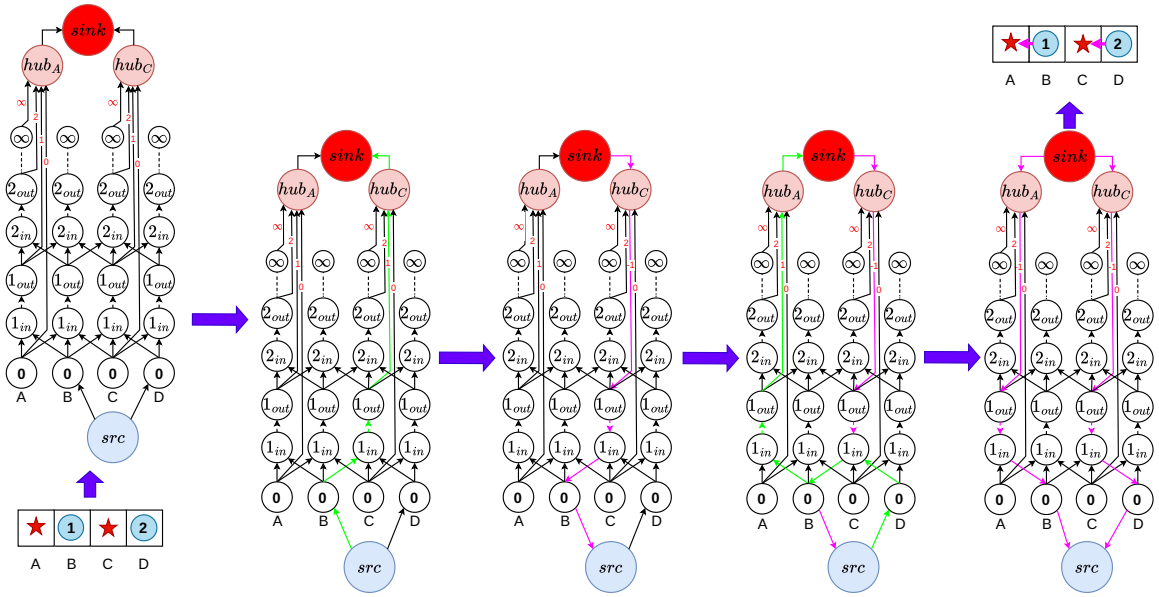

Fig. 9. An example describes how to reduce an AMAPFD instance to MCMF instance and solve the reduced problem by SSPA algorithm. We have two agents 1, 2 on cells B, D need to move to cells A, C in the minimal total arriving time. The first network shows the reduced network. All arcs have 1 capacity. The numbers on the arcs denote the costs of the arcs. The arcs which have zero cost are shown without label. To solve the MCMF problem, the two steps of SSPA algorithm are repeated two times. First, the green shortest path is found with cost 1 as shown in the second network. Next, the found path's arcs are reversed, and their costs are negated. The same two steps are repeated in the next networks (until no *src*–*sink* path can be found). Finally, the reversed arcs denote the arcs of the MCMF final flow, and therefore they are converted to optimal plans for the AMAPFD agents (where the first agent should go to cell A and the second agent should go to cell C).

algorithm), is the cost to reach this state from the initial states, i.e., from *src*. The cost of states consisting of single network nodes, is defined to be the cost to reach the node, and the cost of bulk states is the cost to reach the node with the lowest height in the bulk.

Constraints on arcs costs. To use GBS, the following property should be satisfied in the network.

Given any two interval of nodes $cs_1 = (v, [h_l^1, h_u^1]), cs_2 = (u, [h_l^2, h_u^2])$ *where* v, u *are neighbor vertices in the graph, and there is at least one node* (u, h) *in the second connected-sequence reachable from a node* $(v, h-1)$ *in the first connected-sequence* cs_1*, i.e.,* $h_l^2 \le h \le h_u^2$*,* $h_l^1 \le h-1 \le h_u^1$ *(we denote such node with the lowest* h *by* (u, h_{mn})*). Among all possible paths to move from node* (v, h_l^1) *in* cs_1 *to nodes* $\{(u, h) | h_{mn} \le h \le h_u^2\}$ *in* cs_2 *using only not-reversed move, not-reversed restricting and not-reversed wait arcs, one of the cheapest paths is the one which uses one and the earliest move arc i.e., the path* $(v, h_l^1) \rightsquigarrow (v, h_{mn}-1) \rightarrow (u, h_{mn}) \rightsquigarrow (u, h)$.

This property is naturally satisfied in time-expanding networks which usually have:

- The *wait* arcs (which usually denote to the waiting action) between the same layers (at the same time-step) have equal costs (i.e., *wait* arcs $((v, h_{out}), (v, (h+1)_{in}))$ for a vertex $v \in V$ and height $h \in \{0, \ldots, T-1\}$ have the same cost). The same for *restricting* (which usually helper arcs and have zero cost) and *move* arcs (which usually denote to move actions at specific time-step).
- The *restricting* and *wait* arcs have non-negative costs (usually all actions have non-negative costs).

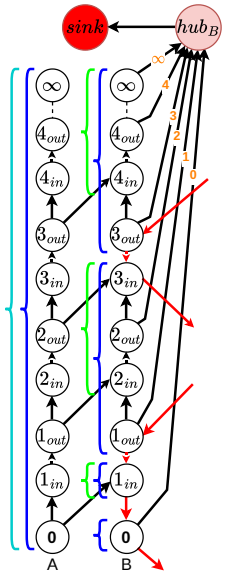

Fig. 10. A cropped part from a network at some iteration of SSPA algorithm, where the blue intervals denote the current connected-sequences in the network, the cyan interval denotes the input state from which we want to generate the successors, the green intervals denote the generated states.

- Between any two different consecutive layers, the costs of *move* arcs are greater than or equal the *wait* arcs. Moreover, the costs of the *move* arcs should be positively correlated with the heights of their endpoint layers. That is, for any two *move* arcs $e_1 = ((u, h_{1_{out}}), (v, (h_1 + 1)_{in}))$, $e_2 = ((w, h_{2_{out}}), (q, (h_2 + 1)_{in}))$ with costs $cost(e_1), cost(e_2)$ for vertices $u, v, w, q \in V$ and heights $h_1, h_2 \in \{0, \ldots, T-1\}$ where $u \neq v, w \neq q$, we have the following: if $h_1 > h_2$ then $cost(e_1) \geq cost(e_2)$, if $h_1 < h_2$ then $cost(e_1) \leq cost(e_2)$. In other words, move action is always greater than or equal to the cost of waiting, and the costs of move actions are non-decreasing with time.

Apparently, all of these properties are satisfied by both reductions for the AMAPFD problem described above (Figures 1 and 16).

Generating successors. In this procedure, we follow the same BS steps, but additionally compute the cost of the successor states. Specifically, as in BS, for each reachable connected-sequence from the input state, we identify the reachable node with the minimal height. Starting from this node, we construct the complete successor bulk state by extending it up to the upper bound of the enclosing connected-sequence. If there are non-layer nodes adjacent to the input state, they are generated individually, without being expanded into bulks.

The cost of a successor state is obtained by adding the cost of the lowest-height node in the input state (which is already stored as the input state's *g_cost*) to the cost of the cheapest transition from the input state to that successor. The transition cost is defined as the cost of moving from the lowest-height node in the input state (or the input node itself, if it is a non-layer node) to the lowest-height node in the successor state (or to the successor node itself, if it is a non-layer node). This transition proceeds along the cheapest path described in the previous paragraph. More details are presented next in pseudocode seetion.

In Figure 10, there is an example of this process. If the parent node is the bulk $(A, [0, \infty])$, then the successors are the bulks $(B, [1_{in}, 1_{in}])$, $(B, [2_{in}, 3_{in}])$ and $(B, [4_{in}, \infty])$ as done in BS algorithm. The path from the parent

state to reach these states are $(A, 0) \rightarrow (B, 1_{in})$, $(A, 0) \rightsquigarrow (A, 1_{out}) \rightarrow (B, 2_{in})$, $(A, 0) \rightsquigarrow (A, 3_{out}) \rightarrow (B, 4_{in})$, and therefore when calculating the cost of these successors, the sum of costs of the arcs of these paths should be added.

Pseudocodes. **GBS**. The main body of the GBS algorithm is given in Algorithm 4 in black and blue color. Its overall structure follows BS (Algorithm 1); the modifications are highlighted in blue. The red text in Algorithm 4 denoting the additional improvements for our reduced networks is explained separately (it is not considered as a part of the general GBS algorithm).

In Line 8, GBS pops from OPEN a state s. This state can be chosen with minimum (g_cost), and this is practically may be faster to finish, however such search (i.e., best-first search with re-expansions in the presence of negative arcs) may expand exponential number of times in worst cases (Martelli 1977). The rest of the procedure follows the standard label-correcting search pattern: expanded states are inserted into CLOSED, successors are generated, and eligible successors are inserted into OPEN. Note that GBS does not terminate when the goal state is first popped. Because negative-cost arcs may exist, a cheaper path to the goal may be discovered later; therefore, the algorithm continues until OPEN becomes empty.

For our reduced networks, we later introduce an additional term, an admissible heuristic function (h_cost), and we pop the nodes from OPEN with minimum ($g\_cost + h\_cost$) value. Theoretically, this search is still exponential in some artificially designed cases (as h_cost is not consistent) (Martelli 1977), however, practically the search terminates fast because our heuristic function does not highly underestimate the cost of the shortest path to the goal. [3] Most importantly, this enables safe early termination when the goal state is first popped. We also introduce, in Line 15, a helper procedure `stateCanBePruned` that detects states that cannot improve the best goal cost and can therefore be discarded before expansion.

The successor-generation routine `getSuccessors` (Algorithm 5) additionally returns, for each successor, the cost of reaching it from the parent state; these costs are passed to `insertToOpen`. Finally, if the goal node is popped at least once, GBS returns the path to the best (lowest-cost) goal state found; otherwise, it reports that the goal state *sink* is unreachable from *src*.

`getSuccessors` function (Algorithm 5). The function generates successor states in the same manner as the BS algorithm, with the additional step of inserting *hub* nodes when applicable. Its only extension is to assign a cost to each successor state. The cost of a successor state is defined as the sum of (i) the cost of the input (predecessor) state and (ii) the cost of the transition from the input state to the successor state.

The general rule for transitions from an input state to a successor state is as follows. Each transition starts at the lowest-height node in the input state (or the node itself if the state contains a single node) and ends at the lowest-height node in the successor state (or the node itself if the successor contains a single node). Among all feasible paths, we select the minimum-cost one.

In Lines 4 and 6, the transition consists of a single arc; therefore, the transition cost equals the cost of that arc. In Line 8, successors are generated from the node with the maximum height in the input bulk-state. Accordingly, when computing the transition cost, we first compute the cost of moving from the minimum-height node in the input state to the maximum-height node of the same state (using only arcs internal to the input state) and then add the cost of the transition to the successor nodes. In Line 10, we move to a non-layer *hub* node via a minimum-cost path. In our networks, where *wait* and *restricting* arcs have zero cost and *hub* arcs have costs that increase with height, the minimum-cost transition uses the lowest-height *hub* arc directly (or includes one *restricting* arc beforehand if the lowest-height node in the input state lies in the *in* copy of the layer).

[3] Moreover using the node-selection techniques introduced in (Martelli 1977), (Mérõ 1984), (A. Mahanti and Bagchi 1985), (A. S. Mahanti and Ray 1987), (Zhang et al. 2009) or (Felner, Zahavi, et al. 2011), one can turn GBS into a theoretically lower-complexity (i.e. lower than exponential) algorithm without compromising any of its properties.

Algorithm 4: Generalized Bulk Search (GBS)

```
Function GBS(N(G, cap, cost, src, sink)):
1   OPEN ← ∅, CLOSED ← ∅
2   g_cost(·) ← +∞
3   goal_state ← ⊥
4   best_goal_cost ← ∞
5   g_cost(src) ← 0
6   insert src into OPEN
7   while OPEN is not empty do
8       pop s from OPEN with minimum g_cost(s) + h_cost(s)
9       insert s into CLOSED
10      if s = sink then
11          if g(s) < best_goal_cost then
12              goal_state ← s
13              best_goal_cost ← g(s)
14          break
15      if stateCanBePruned(s) then
16          continue
17      Succ ← getSuccessors(s, N)
18      foreach (s', cost_s') ∈ Succ do
19          insertToOpen(s', cost_s')
20  if goal_state ≠ ⊥ then
21      return reconstructPath(goal_state)
22  return failure
```

Finally, in Line 17, we generate successor bulk-states. In this case, the transition proceeds from the minimum-height node in the input state to the minimum-height node in the successor state, using the least-cost path defined under *Constraints on arcs costs*.

The last Lines 20–24 are similar to the ones in BS, where we extend the bulks to the upper bound of the enclosing connected-sequences.

`insertToOpen` function (Algorithm 6). Preventing duplicate entries in OPEN and CLOSED is more involved in GBS than in BS. In particular, the same node may appear multiple times in OPEN or CLOSED, provided that each new insertion carries a lower cost than the one stored previously. Hence, before inserting a state into OPEN, we must verify that it improves coverage of at least one network node, i.e., it assigns a strictly smaller cost to at least one node it represents.

For *non-layer* nodes, this check is straightforward: we simply compare the node's current cost with the candidate cost (Lines 1–11). For bulk states, insertion is allowed only if either (1) the bulk contains at least one node that is not covered by any bulk state already in OPEN or CLOSED, or (2) it contains at least one node whose cost is smaller than the cost currently recorded for that node in OPEN or CLOSED.

At the code level, consider inserting a new state s with cost c. We first identify the *previous* state x^{prev}, i.e., the state that covers the first node of the bulk interval of s (Line 13). If x^{prev} reaches this node with cost at most c,

Algorithm 5: Generating successors in GBS

Function `getSuccessors`$(s, N(\mathrm{G}, cap, cost, src, sink))$**:**

1 $succ\_node\_cost\_pairs \leftarrow \emptyset$
2 **if** *s is a non-layer node state* **then**
// s is a node n
3 **foreach** *outgoing arc* (n, n') *in* N **do**
4 insert $(n',\ g\_cost(s) + cost(n, n')) \rightarrow succ\_node\_cost\_pairs$
else
// the input state is a bulk state, i.e., $s = (v, [h_l, h_u])$
5 **foreach** *out-neighbor node* m *of node* (v, h_l) *by a reversed arc* **do**
6 insert $(m, g\_cost(s) + cost((v, h_l), m)) \rightarrow succ\_node\_cost\_pairs$
7 **foreach** *out-neighbor node* m *of node* (v, h_u) *by a reversed arc* **do**
8 insert $(m, g\_cost(s) + cost((v, h_l) \rightsquigarrow (v, h_u)) + cost((v, h_u), m)) \rightarrow succ\_node\_cost\_pairs$
9 **if** *v is a goal vertex* **then**
10 $transition\_cost \leftarrow$ the minimum-cost path from (v, h_l) to hub_v
11 insert $(hub_v,\ g\_cost(s) + transition\_cost) \rightarrow succ\_node\_cost\_pairs$
12 **foreach** *connected-sequence* $cs(u, [h_l^u, h_u^u])$ *with vertex* u *is a neighbor of* v **do**
13 $h_{from} \leftarrow$ the height of the minimum *out*-copy in $[h_l, h_u]$
14 $h_{to} \leftarrow$ the height of the minimum *in*-copy in $[h_l^u, h_u^u]$
15 **if** $h_{from} + 1 \leq h_u^u \wedge h_{to} - 1 \leq h_u$ **then**
16 $h_{min} \leftarrow \max(h_{to}, h_{from} + 1)$
17 $transition\_cost = cost((v, h_l) \rightsquigarrow (v, (h_{min} - 1)_{out})) + cost((v, (h_{min} - 1)_{out}), (u, (h_{min})_{in}))$
18 insert $((u, h_{min_{in}}),\ g\_cost(s) + transition\_cost) \rightarrow succ\_node\_cost\_pairs$
19 $succ\_states \leftarrow \emptyset$
20 **foreach** $(n,\ c) \in succ\_node\_cost\_pairs$ **do**
21 **if** *n is a non-layer node* **then**
22 insert $(n,\ c) \rightarrow succ\_states$
else
// n is a layer node (v, h)
23 $h_{max} \leftarrow$ the upper bound of the connected-sequence where node n is located
24 insert $((v, [h, h_{max}]),\ c) \rightarrow succ\_states$
25 **return** *succ_states*

then s is discarded (Lines 14–15). Since bulk states extend up to the upper bound of their enclosing connected sequence, dominance at the first node implies dominance over the entire bulk interval.

Otherwise, s is inserted because it improves the current best cost for at least the first node. In this case, x^{prev} is trimmed (or removed) to eliminate any overlap with s (Lines 16–19).

As the upper bound of s coincides with that of the enclosing connected sequence, we scan the subsequent states (of higher height) to test whether s is dominated or dominates existing states. It suffices to compare costs at the first node of each subsequent state. Let x^{nxt} denotes the immediate successor. If the cost induced by s at

Algorithm 6: Inserting a state into OPEN in GBS (part 1 of 2)

```
   Function insertToOpen(s, cost_s):
1      if s is non-layer state then
2          if s ∈ {OPEN ∪ CLOSED} then
3              if g_cost(s) ≤ cost_s then
4                  return
               else
5                  if s ∈ CLOSED then
6                      remove s from CLOSED
7                      insert s into OPEN
9                  g_cost(s) ← cost_s
           else
10             insert s →OPEN
11             g_cost(s) ← cost_s
       ...
```

the start node of x^{nxt} is not better than the cost of x^{nxt} (i.e., bigger than the cost of x^{nxt} when $x^{nxt} \in$ OPEN, and bigger than or equal to the cost of x^{nxt} when $x^{nxt} \in$ CLOSED), then s is truncated to end immediately before x^{nxt} begins. Otherwise, s dominates x^{nxt}; we remove x^{nxt} and repeat the same comparison with the next state. Lines 20–31 implement this procedure.

Figure 11 illustrates three different cases for inserting a new state into OPEN in our reduced networks (i.e., under the assumption that the *wait* and *restricting* arcs have zero cost).

Next we establish the core theoretical properties of GBS.

Theorem 3. *Generalized Bulk Search (GBS) algorithm is complete i.e., it finds a solution if it exists, and reports failing otherwise.*

Proof. The completeness of GBS can be directly deduced from the proof of completeness of BS algorithm. That is, regardless of the value of the cost of the generated states, GBS will generate and insert into OPEN the states which cover exactly the same number of network nodes which would be covered by BS algorithm. In GBS, there are additional nodes like *hub* nodes, however, these nodes are considered single nodes and being processed as in general graph search algorithms. □

Theorem 4. *Generalized Bulk Search (GBS) always returns a minimum-cost solution.*

Proof. GBS is a label-correcting search algorithm and continues expanding states until OPEN becomes empty. For a label-correcting algorithm to return a shortest path, it suffices that the *successor-generation* procedure is correct. Concretely, the function `getSuccessors` must satisfy: (1) it generates all successors of the input state, and (2) it assigns to every generated successor its minimum possible cost from the input state. Condition (1) is established in Theorem 3; thus, it remains to prove (2).

Define the *representative* of a state s, denoted $rep(s)$, as follows: if s is a non-layer state, then $rep(s)$ is its (single) node; if s is a bulk-state, then $rep(s)$ is the minimum-height node in that bulk. During cost propagation, `getSuccessors` (i) applies transitions only from $rep(s)$ (i.e., even when the input is a bulk-state, it does not consider starting from other nodes in the bulk), and (ii) computes costs only to $rep(s')$ for each successor state s' (so that, when s' is a bulk-state, the costs of its other nodes are induced from $rep(s')$). We show that this produces

Algorithm 6: Inserting a state into OPEN in GBS (part 2 of 2)

Function `insertToOpen`(s, $cost\_s$):

```
    ...
12  if s is layer (bulk) state (v, [h_l, h_u]) then
        // find the already-inserted state that covers the node (v, h_l)
13      x^prev(v, [h_ℓ^prev, h_u^prev]) is the state in the same connected-sequence of s from {OPEN ∪ CLOSED}
          that covers (v, h_l)
14      if x^prev exists ∧ g_cost(x^prev) + cost((v, h_ℓ^prev) ⇝ (v, h_l)) ≤ cost_s then
            // x^prev dominates s
15          return
        else
16          if x^prev exists then
                // s dominates a part of x^prev, so we remove it
17              modify x^prev: h_u^prev ← h_l − 1
18              if h_u^prev < h_ℓ^prev then
19                  remove x^prev from its set (OPEN or CLOSED)
            // now we check which state dominates the overlapping interval between s and next states
20          x^nxt(v, [h_ℓ^nxt, h_u^nxt]) is the state in the same connected-sequence of s from {OPEN ∪ CLOSED}
              with the least lower bound greater than h_l
21          while x^nxt exists do
22              if x^nxt ∈ OPEN then
23                  if cost_s + cost((v, h_l) ⇝ (v, h_ℓ^nxt)) ≤ g_cost(x^nxt) then
24                      remove x^nxt from OPEN // the whole interval of x^nxt is dominated by s
                    else
25                      h_u ← h_ℓ^nxt − 1 // a part of the interval of s is dominated by x^nxt
26                      break
                else
27                  if cost_s + cost((v, h_l) ⇝ (v, h_ℓ^nxt)) < g_cost(x^nxt) then
28                      remove x^nxt from CLOSED // the whole interval of x^nxt is dominated by s
                    else
29                      h_u ← h_ℓ^nxt − 1 // a part of the interval of s is dominated by x^nxt
30                      break
31              x^nxt(v, [h_ℓ^nxt, h_u^nxt]) is the state in the same connected-sequence of s from {OPEN ∪
                  CLOSED} with the least lower bound greater than h_l
        // finally insert s (possibly after shrinking its interval)
32      g_cost((v, [h_l, h_u])) ← cost_s
33      insert (v, [h_l, h_u]) → OPEN
```

the correct (minimum) successor costs; the only nontrivial case is when states are bulk-states (single-node states are straightforward).

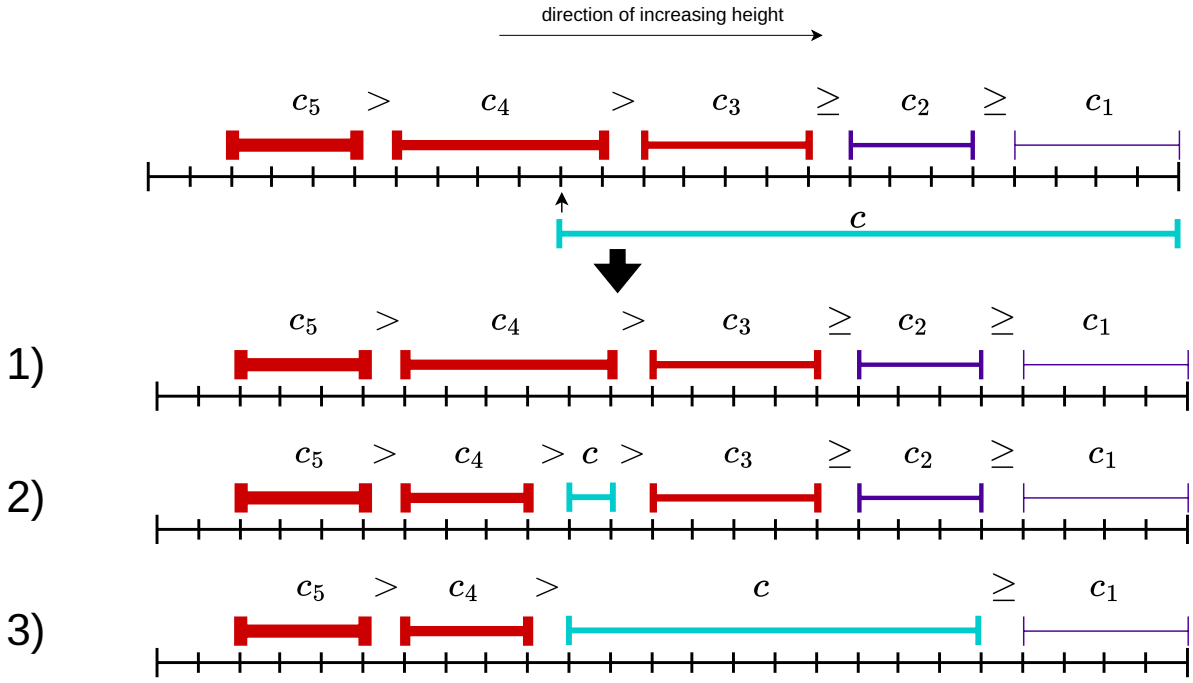

Fig. 11. The figure illustrates how bulk states are distributed across the OPEN (red) and CLOSED (violet) sets during GBS (run on our reduced networks) for a given connected-sequence (black interval). It highlights three representative situations in which a new state (cyan) must be inserted into OPEN. Unlike BS, GBS associates a cost with each state; as a result, a new bulk state may be admissible even when its individual nodes have been generated previously, provided that it offers a lower cost than the corresponding entries currently stored in OPEN. Note that, for a general network, cost comparisons must additionally incorporate the contributions of the *wait* and *restricting* arcs as implemented in Algorithm 6.

We first justify part (ii). By definition, the cost of a state s is the cost of its representative $rep(s)$. For a bulk-state s, the cost of any node in s is implicitly defined as the cost of $rep(s)$ plus the internal cost from $rep(s)$ to that node within the bulk. Thus, we need to show that this implicit definition yields correct costs for bulk-states when they are generated. There are two cases. In the first case, $rep(s)$ is the single entry point through which the bulk-state is reached when it is generated; then the induced costs are clearly correct. This occurs, for example, when transitioning from the single *src* or *hub* nodes, or when generating bulk-states via reversed arcs, where the contact with the successor bulk-state is single (through $rep(s)$). In the second case, the parent state is itself a bulk-state, so different nodes in the successor bulk could, in principle, be reached through different transitions. However, under the assumption stated in the *Constraints on arcs costs* section, one of the minimum-cost ways to reach the successor bulk necessarily passes through its representative $rep(s)$. Therefore, computing the successor cost only for $rep(s)$ and inducing the remaining node costs from it remains correct. This establishes part (ii).

Part (i) then follows directly. Since the minimum costs of nodes in a successor bulk are assumed to be achieved via the representative of that bulk, initiating transitions from $rep(s)$ (on behalf of all nodes in the parent bulk-state) yields an overall cost that is no greater than initiating transitions from any other node: the cost of the chosen transition-starting node is added to the total cost of the successor, so starting from $rep(s)$ is never worse.

Therefore, `getSuccessors` correctly propagates the minimum possible cost from an input state to all its successor states. Hence, GBS returns a minimum-cost solution.

□

Since GBS may (re)insert a state multiple times to guarantee optimality, an admissible heuristic function can substantially reduce the number of expanded states and, in some cases, enable early termination. Specifically, as shown in Algorithm 4, if OPEN is ordered by $g_{cost} + h_{cost}$ and the state with minimum $g_{cost} + h_{cost}$ is expanded, then the search may terminate as soon as the goal node is popped from OPEN. Intuitively, at that point no remaining state in OPEN has a lower-bound estimate to the goal that is smaller than the cost of the popped goal, and therefore no cheaper solution can exist. Next, we provide further improvements developed specifically for our reduced networks.

Heuristic function. We define a heuristic function $h\_cost : \mathrm{V} \rightarrow \mathbf{N}$ to estimate the cost from an input node $n \in \mathrm{V}$ to *sink* node. A heuristic function is characterized as *admissible* if it never overestimates the real cost. This function helps to focus the search and therefore expanding fewer states.

We define the shortest distance from a vertex u to vertex v in the original AMAPFD graph (ignoring all agents) through the function $d(u, v)$. Additionally, we also define the minimum distance from a vertex u to a goal vertex by $mnD(u)$, i.e., $mnD(u) = \min_{v \in \mathcal{G}} d(u, v)$. We now define the function h_cost as follows:

$$h\_cost(n) = \begin{cases} h + mnD(v), & \text{if } n \text{ is a layer node } (v, h) \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Theorem 5. *The function h_cost defined in Equation 7 is an admissible heuristic function in the AMAPFD reduced networks in every iteration of SSPA iterations.*

Proof. The proof of the first case when the node is a layer node consists of two parts. We first prove the admissibility of the function in the initial state when all arcs have positive costs and no reversed arc exists. Second we prove the admissibility of the function in the presence of the negative and reversed arcs.

For both cases, reaching the sink node must be done through the hub nodes, i.e., each path consists of the *src* node, a sub-path in the network layers which ends by a goal node, a hub node and finally *sink* node, in this order.

Initially, the only arcs which have positive costs are the goal arcs (i.e., the arcs between goal nodes and hub nodes) which have costs equal the height of the goal nodes. Therefore, a lower-limit of the cost of going from a layer node to the sink node is the minimum height that can be reached at a goal node from this layer node (the goal node is connected to a hub node which is connected to the sink node directly). In this case, there are no reversed arcs, and therefore moving directly from a layer node $n = (v,\ h)$ to the closest goal node will take $mnD(v)$ move actions and this increases the height by the same value. This is because each *move* arc is directed from a node with a lower height to a neighboring node with a higher-by-one height. As a result, the minimum height to get at a goal node equals $h + mnD(v)$. The first part is proved.

We proved that going directly from a layer node to *sink* node through the closest goal node will increase the cost of the layer node by $h + mnD(v)$, in the first case when all arcs have positive costs and no reversed arc exists. Now we will prove that even when passing reversed and arcs with negative costs, it is not possible to reach *sink* node in lower than this cost $h + mnD(v)$. From the theoretical properties of SSPA, the reversed arcs are always created in the form of independent (i.e., sharing no nodes or arcs) reversed paths from *sink* to *src* (see for example the last network in Figure 9). Moreover, in iteration i, it is guaranteed that the set of i found paths have the minimum possible sum-of-cost which can be reached so far. Let us suppose we are at a layer node $n(v,\ h)$ with a suggested heuristic cost $h\_cost(n) = h + mnD(v)$. As already proved, using not-reversed arcs we cannot get a smaller cost. Now, let us suppose that the minimum-cost path from n to *sink*: $p = n \rightsquigarrow n_1 \rightsquigarrow n_2 \rightsquigarrow sink$ passes from the reversed sub-path $n_1 \rightsquigarrow n_2$ (i.e., $n_1 \rightsquigarrow n_2$ consists of reversed arcs). Suppose that the sub-path

$n_1 \rightsquigarrow n_2$ is the first reversed sub-path along p beginning from n. We denote the cost of parts $n \rightsquigarrow n_1$, $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow sink$ by c_{pos_1}, c_{neg} and c_{tail}, respectively. Let p_{old} be a src–$sink$ path found in a previous SSPA iteration j, where $n_1 \rightsquigarrow n_2$ is a reversed part of it (p_{old} is reversed in the network of current iteration). We also denote the cost of the sub-path of p_{old} from n_1 to $sink$ in the network of iteration j (i.e., before reversing p_{old} arcs) by c_{pos_2}. Now, we need to prove that

$$h\_cost(n) \leq c_{pos_1} - c_{neg} + c_{tail} \tag{8}$$

The minus sign due to the reversed arcs with negated costs. First, we have that

$$h\_cost(n) \leq c_{pos_1} + c_{pos_2} \tag{9}$$

because all arcs in these two sub-paths are original (not-reversed) arcs. We also know that

$$c_{tail} \geq c_{neg} + c_{pos_2} \tag{10}$$

because otherwise the path p_{old} would have the part $n_2 \rightsquigarrow sink$ (with cost c_{tail}) instead of the part $n_2 \rightsquigarrow n_1 \rightsquigarrow sink$ (with cost $c_{neg} + c_{pos_2}$) because it yields a smaller cost, and we know that SSPA always searches for the minimum-cost paths. Substituting c_{pos_2} from Equation 10 in Equation 9, we get

$$h\_cost(n) \leq c_{pos_1} + c_{pos_2} \leq c_{pos_1} + c_{tail} - c_{neg} \tag{11}$$

As a result, we get Equation 8, and the first case in Equation 7 is proved.

For the second case, we note the following. The first found path in SSPA cannot be negative as all arcs have zero or positive costs. From SSPA properties (Edmonds and Karp 1972), it is guaranteed that the path found in any iteration has a cost greater than or equal to the cost of the previous ones. Therefore, no found path in SSPA can have negative cost and hence the case when n is src is proved. Moreover, in SSPA if we do not have a negative cycle in the beginning, there will be no negative cycle in any iteration. Therefore, the case when the n is $sink$ node is proved. The case for hub nodes can be proved as in the first case. The second case is proved, and the proof is complete now. □

Additional pruning. We introduce an observation in AMAPFD reduced networks which helps to further reduce the number of expanded search states. Let the resultant network N at iteration i of SSPA, have some goal nodes at vertices $\mathcal{G}_{vis} \subset \mathcal{G}$ connected to hub nodes by reversed arcs. Let us denote the height of the connected goal node at vertex $u \in \mathcal{G}_{vis}$ by h_{vis_u}.

For each vertex v in the original map, we define the function *maximum height* $h_{max} : V \longrightarrow \mathbf{N}$ by the following equation:

$$h_{max}(v) = \begin{cases} 0, \text{ if } \max\limits_{u \in \mathcal{G}_{vis}} h_{vis_u} + 1 - d(v,\ u) \leq 0 \\ \max\limits_{u \in \mathcal{G}_{vis}} h_{vis_u} + 1 - d(v,\ u), \text{ otherwise} \end{cases} \tag{12}$$

Lemma 1. *Starting from any node* $(v,\ h) : v \in V,\ h \in \{h_{max}(v),\ h_{max}(v) + 1,\ \ldots\}$*, it is not possible to reach a layer node connected to a reversed arc before passing from a hub node.*

Example In the example in Figure 12, we have two reversed paths from $sink$ to src. Yellow circles, refer to the nodes with maximum heights (at each vertex) which are connected to a reversed arc. In order for any node to not be able to pass from a reversed arc, it should be far enough from the yellow circles. More precisely, the difference between heights should be smaller than the distance between vertices. For example, the distance between vertex A and C is 2 and between vertex H and vertex C is 5. The yellow circles in A and H have heights 3 and 2, respectively. Therefore, the *maximum height* h_{max} at vertex C should be strictly greater than $3 - 2 = 1$ and strictly greater than $2 - 5 = -3$, i.e., it should be greater than or equal 2. That is, beginning from node $(C,\ 2)$ we can never reach a node with a reversed arc (we reach vertex A at a height at least 4, and vertex H at a height

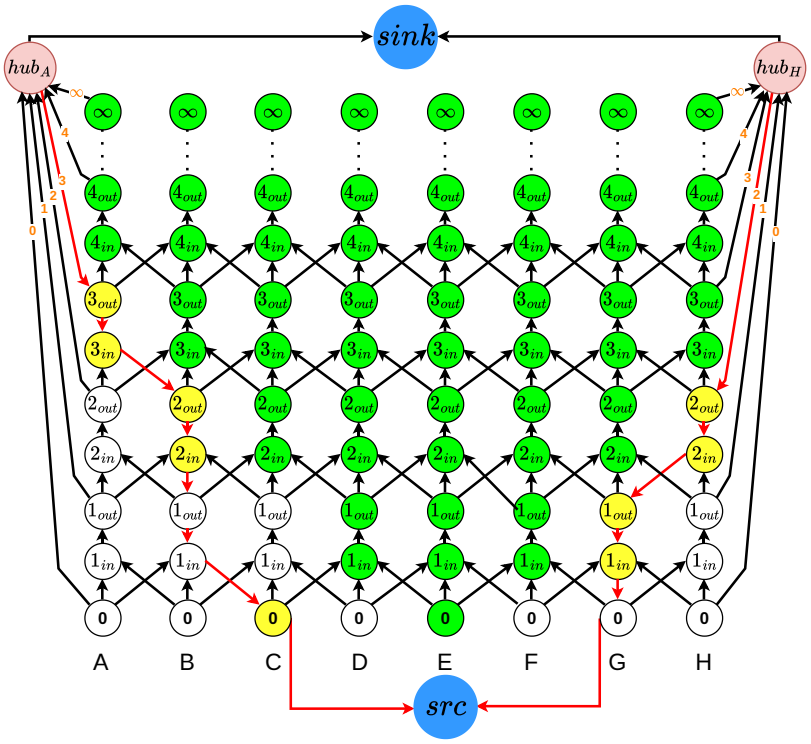

Fig. 12. An example shows the *maximum heights* at each vertex. The nodes with heights equal to or higher than the maximum heights (the green circles) cannot reach any node with a reversed arc.

at least 7). It is enough to compare with vertices A and H because they are the start layer nodes of the reversed paths. The green nodes in the Figure 12 refer to the nodes, beginning from which, it is not possible to pass a reversed arc before arriving at a hub node.

Proof. Now we formally prove Lemma 1. In SSPA, in the resulting networks after each iteration, the reversed arcs are directed down from $(u,\ h_{vis_u}) : u \in \mathcal{G}_{vis}$ to one of the start nodes at height 0, decreasing one height in each move to a different vertex. Therefore, the maximum possible height of a node at a vertex w which may be connected to a reversed arc (in the path coming down) from a goal node $(u,\ h_{vis_u})$ is $h_{vis_u} - d(u,\ w)$ (as $d(u,\ w)$ is the minimum number of moves to get at vertex w from vertex u). Hence, the maximum possible height for a node at a vertex w which may be connected to a reversed arc (in a path coming down) from any goal node is $\max_{u \in G_{vis}} h_{vis_u} - d(u,\ w)$ (the yellow nodes in Figure 12).

Now, starting from a vertex v, to reach a node with vertex w, we need at least $d(v,\ w)$ number of moves. Therefore, starting from node $(v, h_{\max}(v))$, any reachable node at vertex w must have height at least $h_{min} = h_{max}(v) + d(v,\ w)$. Substituting the $h_{max}(v)$ with its maximum value from Equation 12, we get:

$$\begin{aligned} h_{min} &= \max_{u \in G_{vis}} h_{vis_u} + 1 - d(v,\ u) + d(v,\ w) \\ h_{min} &= \max_{u \in G_{vis}} h_{vis_u} + 1 - (d(v,\ u) - d(v,\ w)) \end{aligned} \tag{13}$$

We have $d(v,\ u) \leq d(v,\ w) + d(w, u)$ for any three vertices v, u, w in the original map. Substituting in Equation 13, we get:

$$h_{min} \geq \max_{u \in G_{vis}} h_{vis_u} - (d(w,\ u)) + 1 \tag{14}$$

This means that starting from a vertex v at the height $h_{max}(v)$ we never reach a node connected to a reversed arc. □

Lemma 2. *If the sink node can be reached from a node n_1 with a height $h_1 > h_{max}(v)$ at a map vertex v with a cost c, then the sink node can be also reached from node $n_2 = (v,\ h_2) : h_1 > h_2 \geq h_{max}(v)$ by a cost exactly equals $c - (h_1 - h_2)$.*

Proof. To reach the sink node, the path should pass through at least one hub node necessarily. In lemma 1, we proved that starting the search from a node n at a vertex v at height $h_{start} \geq h_{max}(v)$, we never meet a node with a reversed arc before passing a hub node. Therefore, starting from n, we can reach a node at any vertex u by exactly $d(v,\ u)$ number of moves and zero cost. Each one move increases the height by one layer, and therefore the height becomes $h_{start} + d(v,\ u)$. To get at a hub node, we first get at a goal layer node connected to it, then we move to the hub node by a cost equals the height of the goal node. Therefore, starting from nodes n_1, n_2, any hub node can be reached and the difference between costs always equals the difference in starting heights $h_1 - h_2$. The path to sink node, if possible, will continue identically from hub nodes for both n_1 and n_2, and the theorem is proved. □

Using Lemma 2 we can identify states that can be effectively pruned in the following way. Assume we need to check whether a state $s(v, h)$ can be pruned, by checking that it is dominated by some state in $X = OPEN \cup CLOSED$. To this end, we iterate over states $s' = (v, h') \in X$ and for each such state check whether the following conditions are met:

$$\begin{aligned} c' &\leq c + (h - h') \ \text{ where } h_{max}(v) \leq h' \leq h \\ c' &\leq c + (h - h_{max}(v)) \ \text{ where } h' < h_{max}(v) \end{aligned} \tag{15}$$

Where c' is the cost of the state s'.

If 15 is met for some s', then s' can for sure find cheaper path than s. Thus s is dominated and can be discarded.

7 Experimental Evaluation

7.1 AMAPF-makespan

We have implemented the improved AMAPF solver in C++[4] and compared 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 2013a). The code of the competitor was taken from public repository [5] that accompanied the paper (Okumura and Défago 2022) [6]. 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. 2019). We use all 33 maps available in this benchmark and all of the 25 random scenarios. Each scenario on each map

[4] https://github.com/PathPlanning/AMAPF-MF-BS

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

[6] In (Okumura and Défago 2022) an optimal AMAPF solver was evaluated against the suboptimal one, TSWAP, which actually was the main contribution of the latter paper. Since our paper focuses on optimal algorithms for AMAPF/AMAPFD, we do not evaluate TSWAP itself and refer an interested reader to (Okumura and Défago 2022) for a discussion on trade-offs between optimality and practical efficiency in solving AMAPF.

Table 1. Success rates of *flow-BS* and *flow* solvers for AMAPF-makespan problem and CBS-TA, GENERAL (reduction-to-MCMF using general solver) and MCMF-GBS (our method) for AMAPFD-SOC problem, with a timeout of 30 seconds.

Map	Width	Height	Number of Free Cells	AMAPF		AMAPFD		
				flow	flow-BS	CBS-TA	GENERAL	MCMF-GBS
empty-8-8	8	8	64	**100**%	**100**%	78%	**100**%	**100**%
empty-16-16	16	16	256	**100**%	**100**%	22%	**100**%	**100**%
maze-32-32-2	32	32	666	**100**%	**100**%	7%	**100**%	**100**%
room-32-32-4	32	32	682	**100**%	**100**%	6%	**100**%	**100**%
maze-32-32-4	32	32	790	**100**%	**100**%	5%	**100**%	**100**%
random-32-32-20	32	32	819	**100**%	**100**%	7%	**100**%	**100**%
random-32-32-10	32	32	922	**100**%	**100**%	6%	**100**%	**100**%
empty-32-32	32	32	1024	**100**%	**100**%	7%	**100**%	**100**%
empty-48-48	48	48	2304	**100**%	**100**%	4%	**100**%	**100**%
den312d	65	81	2445	**100**%	**100**%	3%	**100**%	**100**%
room-64-64-8	64	64	3232	**100**%	**100**%	3%	**100**%	**100**%
random-64-64-20	64	64	3270	**100**%	**100**%	5%	**100**%	**100**%
room-64-64-16	64	64	3646	**100**%	**100**%	4%	**100**%	**100**%
random-64-64-10	64	64	3687	**100**%	**100**%	5%	**100**%	**100**%
warehouse-10-20-10-2-1	161	63	5699	**100**%	**100**%	4%	**100**%	**100**%
ht_chantry	162	141	7461	**100**%	**100**%	5%	35%	**100**%
maze-128-128-1	128	128	8191	3%	**100**%	4%	0%	**100**%
ht_mansion_n	133	270	8959	96%	**100**%	4%	20%	**100**%
warehouse-10-20-10-2-2	170	84	9776	**100**%	**100**%	4%	0%	**100**%
lt_gallowstemplar_n	251	180	10021	82%	**100**%	4%	0%	**100**%
maze-128-128-2	128	128	10858	4%	**100**%	5%	0%	**100**%
ost003d	194	194	13214	57%	**100**%	4%	0%	**100**%
lak303d	194	194	14784	44%	**100**%	4%	0%	**100**%
maze-128-128-10	128	128	14818	17%	**100**%	4%	0%	**100**%
warehouse-20-40-10-2-1	321	123	22599	91%	**100**%	5%	0%	**100**%
den520d	256	257	28178	16%	**100**%	5%	0%	**100**%
w_woundedcoast	642	578	34020	1%	**100**%	5%	0%	**99**%
warehouse-20-40-10-2-2	340	164	38756	8%	**100**%	5%	0%	**100**%
brc202d	530	481	43151	1%	**100**%	5%	0%	**93**%
Paris_1_256	256	256	47240	5%	**100**%	4%	0%	**100**%
Berlin_1_256	256	256	47540	6%	**100**%	5%	0%	**100**%
Boston_0_256	256	256	47768	4%	**100**%	5%	0%	**100**%
orz900d	1491	656	96603	0%	**100**%	4%	0%	**59**%

(except some small ones) contains 1,000 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, and 1,000 pairs of start-goals sequentially. Whenever the 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.

In the first experiment, we have 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 have not accounted for its runtime.

For each map, we have collected the total number of success instances over all scenarios. Table 1 summarizes the results. As can be seen, our solver is able to solve all instances in all maps in less than 30 seconds. However, this is not the case for *flow* solver. The latter searches the network node-by-node and, therefore, is able to solve the test instances only on the small maps. When the maps are large (like city maps that are 256×256) or the makespan is high (like in the maze maps), it is often unable to produce a solution under the imposed time limit. Figure 13 shows the success rate for all instances in all maps second by second. In fact, *flow-BS* is able to solve the hardest instance (`orz900d` map, scenario 20 with 256 agents) in 17s. On the other hand, *flow* has shown a very slight increase of SR after solving the easy instances (75% of instances) around the 8th second.

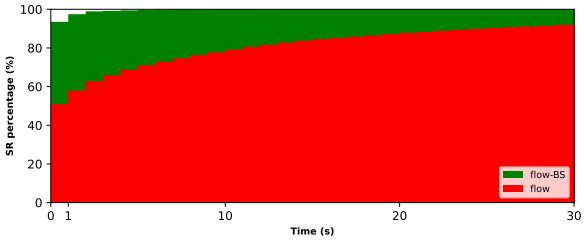

Fig. 13. The (normalized) number of instances solved by a certain time cap.

In the second experiment, we have 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 of the expanded nodes against the number of agents. The reason we show the number of expanded nodes instead of the runtime is that, the time required to process a single expansion differs substantially between *flow* and *flow-BS*. Although *flow-BS* has a more involved successor-generation procedure, *flow* maintains an explicit time-expanded network, which results in considerably higher memory traffic and read/write latency. Because this overhead depends on hardware characteristics (e.g., cache behavior and available RAM), we report the number of expanded nodes to make the comparison less hardware-dependent. The results (plotted in Figure 14) show that our algorithm expands much fewer nodes than the standard algorithm (as expected). We additionally add Table 2, which reports the per-expansion runtime on the Berlin map.

So far, we have assumed that we have an estimator which can identify an exact bound (T) of the makespan. However, this may be not the case, so the search may be repeated several times until finding the optimal solution

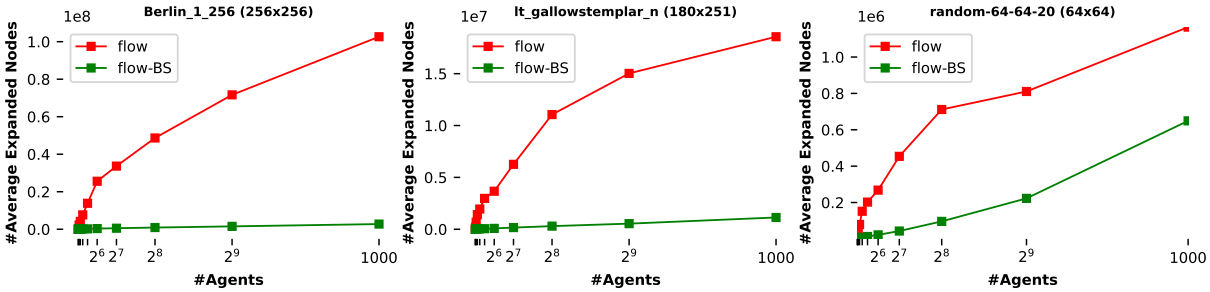

Fig. 14. The number of expanded nodes with a different number of agents on different maps.

Table 2. Runtime of one expansion of both *flow* and *flow-BS* algorithms in microseconds for Berlin_1_256 map. Note that when the makespan is high (the case when we have a few number of agents), the network stored by *flow* is large, and hence the per-expansion time increases.

number of agents	1	2	4	8	16	32	64	128	256	512	1000
flow	16686	4318	309	4	2	1	1	1	1	1	1
flow-BS	1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1

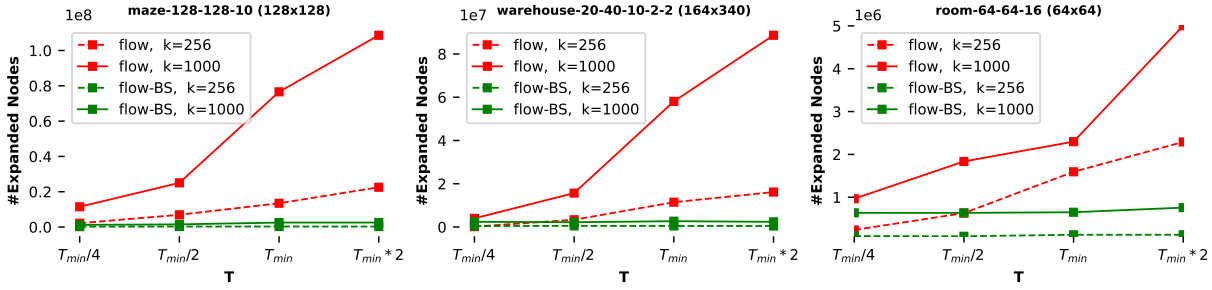

Fig. 15. The figure shows (for a specific map and a specific number of agents) how the number of expansions changes with the increasing values of T. Here T_{min} denotes the optimal makespan.

(e.g., using binary search). Therefore, we have 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 are designed as follows. For each one of the three fixed maps, `warehouse-20-40-10-2-2`, `maze-128-128-10`, `room-64-64-16`, and for a number of agents $\in \{256, 1000\}$, we run the solvers on networks with a maximum height equals one of the values $\{T_{min}/4, T_{min}/2, T_{min}, T_{min} * 2\}$, where T_{min} is the optimal makespan. Again, we have recorded the average number of expanded nodes over all scenarios (see Figure 15). Obviously, the number of nodes expanded by *flow* significantly increases with the value of T, while *flow-BS* does not demonstrate such growth. It means that our solver is especially useful when one cannot estimate the optimal makespan accurately before actually solving an AMAPF problem instance.

7.2 AMAPFD-SOC

We used the same tests from previous section to evaluate the performance of the algorithms used to optimally solve AMAPFD-SOC problem.

Three methods were used in the tests. The first method is the Conflict-Based Search with Task Assignment CBS-TA (Hönig et al. 2018) method. In CBS-TA, a search tree filled by the theoretically best assignments of agents (according to the lowest-bounds of the agents' paths costs) is created. Then each node also opens a CBS search sub-tree. Then, a best-first search on this whole forest is run until a valid node (node with real paths and without conflicts) better than all other nodes' costs is found. Theoretically, CBS-TA has exponential complexity in both parts, assignment part and MAPF part, as the number of assignments is exponential and each conflict between two paths generates two new nodes, so the number of nodes grows exponentially with the number of conflicts. However, the method usually works faster in practice. The code of CBS-TA was taken from the algorithm authors' library [7], but it was adapted to the AMAPFD i.e., once an agent reaches its goal, it is removed from subsequent conflict checks.

[7] https://github.com/whoenig/libMultiRobotPlanning/tree/main

In the second method (denoted GENERAL), we apply the reduction-to-MCMF approach and solve the resulting instance using a black-box, general-purpose MCMF solver. We use the Min-Cost Flow solver from the Google OR-Tools optimization library.[8] Since the solver requires a finite input graph, we truncate the time-expanded network at a fixed horizon. To give this method the strongest possible advantage, we set the horizon to the smallest makespan sufficient to obtain an SOC-optimal solution for each instance (as determined by first solving the instance with our method). For this solver, we use the second reduction shown in Figure 16, because, without heuristic guidance, distributing costs earlier can help the solver account for costs sooner and avoid spending effort on large zero-cost portions of the network.

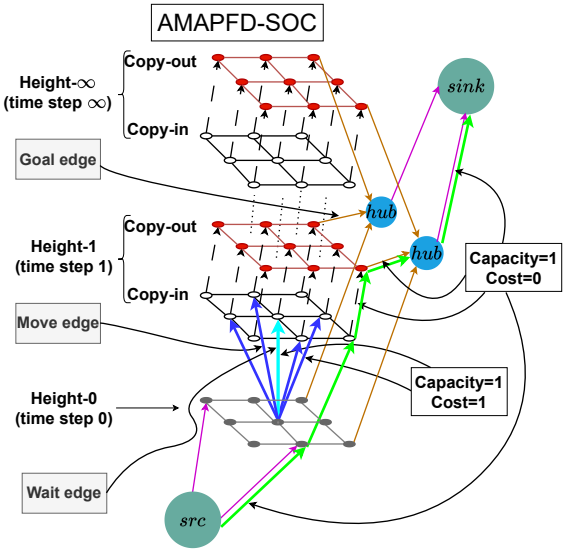

Fig. 16. Another reduction of the AMAPFD problem where the costs of *move* and *wait* arcs are ones and all other arcs have costs zeros. This reduction works faster for general MCMF solvers. In empirical evaluation, this reduction was used when testing GENERAL solver.

In the third method, we use our proposed AMAPFD solver, which solves the reduced MCMF instances using the GBS algorithm described in this paper. In this setting, we use the networks introduced in Section 5.1 (where the inter-layer edges have zero cost, Figure 8), which allows us to implement the additional pruning and heuristic techniques presented in the paper.[9]

We collect for each method the average number of success instances in less than 30 seconds in each map. The success rates of all methods are presented in three rightmost columns of the Table 1. The results show that our method solves substantially more instances within the imposed time limit than the baselines. That is, while MCMF-GBS could solve almost all instances in all maps (except some instance with high number of agents in maps `orz`, `brc202d` and `w_woundedcoast`, which however, they were solved under 60 seconds), CBS-TA and GENERAL could solve only the sparse and small instances, respectively.

Additionally, we analyze the runtime of our solver, MCMF-GBS, with respect to different number of agents and different maps – see Figure 17. Indeed, as can be directly observed, the algorithm's runtime grows linearly

[8] https://github.com/google/or-tools
[9] The code is available at https://github.com/PathPlanning/AMAPFD-SOC

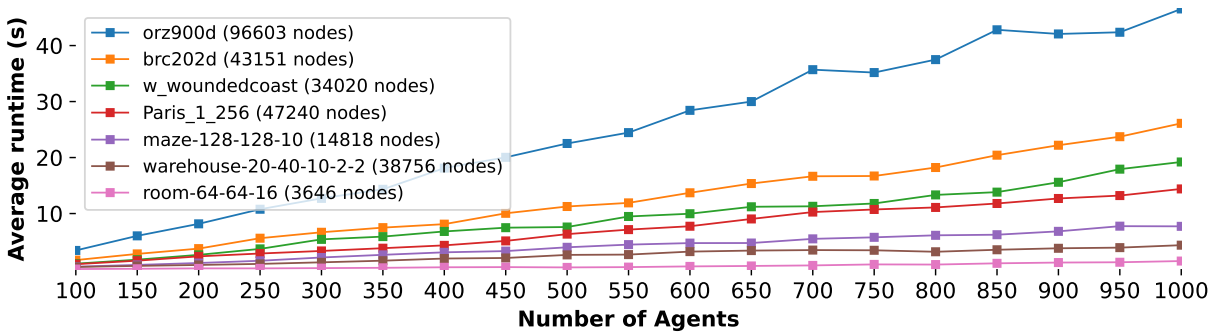

Fig. 17. The average runtime of MCMF-GBS.

with the number of agents. Moreover, we can also note the linear relation between the runtime and the size of the map (number of traversable grid cells). For example, consider two maps of the similar topology: `orz900d` and `w_woundedcoast`. The former contains 2.8 more free grid cells and the average runtime of MCMF-GBS on it is 2.9 higher than on the latter. Lastly, it is worth noting that there is a hidden factor for the algorithm's runtime which is the length of the individual path for each agent. That is, the maps which have long average path for the agents require more runtime than the maps which have shorter average paths. This explains why map `maze-128-128-10` requires more runtime than the larger map `warehouse-20-40-10-2-2`.

8 Conclusion

In this paper, we have revisited the reduction-based approach to optimally solving the Anonymous MAPF problem (with the makespan as the cost objective), 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, Bulk Search, tailored to find paths on the auxiliary graphs while exploiting their specific topology. Next, we have considered the Anonymous MAPF with Disappearing Agents (AMAPFD) problem with sum-of-costs as the objective. The latter can also be reduced to finding a path on a specific auxiliary graph; in this case, the graph is weighted. To efficiently search for least-cost paths on such graphs, we have extended Bulk Search to its generalized version, Generalized Bulk Search, that can correctly handle costs. Moreover, we have introduced heuristic functions and additional pruning techniques to accelerate the search. We have established the theoretical properties of the suggested algorithms and proved that both of them are complete, while Generalized Bulk Search is guaranteed to find optimal solutions as well. Consequently, when these algorithms are used as the backbones for the corresponding AMAPF/AMAPFD solvers, the latter are guaranteed to find solutions of minimal cost. On the empirical side, we have conducted a massive experimental evaluation to compare the resultant improved AMAPF/AMAPFD solvers to the state of the art. Clearly, the former significantly outperform the competitors on a large variety of setups, leveraging the better scalability of the underlying search techniques to the sizes of the input graphs. Thus, we have notably raised the bar in optimally solving AMAPF/AMAPFD problems.

Promising directions for future work include the following. First, it is natural to extend the proposed approaches toward AMAPF solvers that optimize (i.e., minimize) alternative objective functions. We believe that deriving a makespan-optimal AMAPFD solver is relatively straightforward by leveraging our findings. In contrast, obtaining a sum-of-costs-optimal AMAPF solver may be non-trivial: in our view, the main bottleneck would likely be the reduction itself rather than the efficiency of search on the reduced graph. A related direction is to solve makespan-optimal AMAPF in a single run (rather than iteratively solving the T-step version). This variant can be reduced to a minimum-cost maximum-flow (MCMF) problem and then solved efficiently using GBS.

Second, several practically motivated extensions of classical AMAPF/AMAPFD naturally arise and merit investigation. For example, in real-world settings, "disappearing" (i.e., leaving the transportation area) may take time rather than occurring instantaneously. Unfortunately, even allowing an agent a single additional time step is non-trivial and difficult to support within the reduced networks. Similar difficulties arise when relaxing standard modeling assumptions, such as extending edge-collision constraints to settings with intersecting edges (e.g., 8-connected or higher-connectivity grids, AMAPF for large-agent geometries, etc.).

Finally, the suggested solvers can be utilized in tackling the lifelong variants of AMAPF. In this setting it may be beneficial to utilize the suggested AMAPFD solver as in lifelong setting there is no need for any agent to stay on its goal upon arrival waiting for the other agents to complete their current tasks. However, in some applications, the other agents may be unable to change their goal locations (e.g., a warehouse robot that has already picked up an item and must deliver it to a specific station). In such cases, a deadlock may occur if the agent that finishes its task (and is assumed to disappear) is located on the planned path of an agent whose goal cannot be changed. It would be interesting to consider such mixed cases and develop appropriate techniques.

Acknowledgments

The study was supported by the Ministry of Economic Development of the Russian Federation (agreement No. 139-15-2025-013, dated June 20, 2025, IGK 000000C313925P4B0002).

References

A. Adler, M. De Berg, D. Halperin, and K. Solovey. 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.

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

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall. ISBN: 978-0136175490.

Z. A. Ali and K. Yakovlev. 2024. "Improved Anonymous Multi-Agent Path Finding Algorithm." In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI 2024)*, 17291–17298.

J.-M. Alkazzi and K. Okumura. 2024. "A Comprehensive Review on Leveraging Machine Learning for Multi-Agent Path Finding." *IEEE Access*.

A. Andreychuk, K. Yakovlev, D. Atzmon, and R. Stern. 2019. "Multi-Agent Pathfinding with Continuous Time." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 39–45.

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

R. Barták and J. Švancara. 2019. "On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective." In: *Proceedings of the 12th Annual Symposium on Combinatorial Search (SOCS 2019)*, 10–17.

E. Boyarski, A. Felner, R. Stern, G. Sharon, O. Betzalel, D. Tolpin, and E. Shimony. 2015. "ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding." In: *Proceedings of The 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 740–746.

D. G. Cattrysse and L. N. Van Wassenhove. 1992. "A survey of algorithms for the generalized assignment problem." *European journal of operational research*, 60, 3, 260–272.

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

L. Cohen, T. Uras, T. S. Kumar, and S. Koenig. 2019. "Optimal and Bounded-Suboptimal Multi-Agent Motion Planning." In: *Proceedings of the 12th Annual Symposium on Combinatorial Search (SOCS 2019)*, 44–51.

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

S. Dergachev and K. Yakovlev. 2024. "Decentralized Unlabeled Multi-Agent Pathfinding Via Target And Priority Swapping." In: *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI 2024)*. IOS Press, 4344–4351.

J. Edmonds and R. M. Karp. 1972. "Theoretical improvements in algorithmic efficiency for network flow problems." *Journal of the ACM (JACM)*, 19, 2, 248–264.

E. Erdem, D. G. Kisa, U. Oztok, and P. Schüller. 2013. "A general formal framework for pathfinding problems with multiple agents." In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013)*, 290–296.

A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, and S. Koenig. 2018. "Adding heuristics to conflict-based search for multi-agent path finding." In: *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 83–87.

A. Felner, R. Stern, S. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N. Sturtevant, G. Wagner, and P. Surynek. 2017. "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges." In: *Proceedings of the 10th International Symposium on Combinatorial Search (SoCS 2017)*, 29–37.

A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang. 2011. "Inconsistent heuristics in theory and practice." *Artificial Intelligence*, 175, 9, 1570–1603. doi:https://doi.org/10.1016/j.artint.2011.02.001.

G. Fine, D. Atzmon, and N. Agmon. 2023. "Anonymous Multi-Agent Path Finding with Individual Deadlines." In: *Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023)*, 869–877.

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

A. Frank. 2005. "On Kuhn's Hungarian method—a tribute from Hungary." *Naval Research Logistics (NRL)*, 52, 1, 2–5.

A. V. Goldberg, S. Hed, H. Kaplan, and R. E. Tarjan. 2017. "Minimum-cost flows in unit-capacity networks." *Theory of Computing Systems*, 61, 987–1010.

A. V. Goldberg and R. E. Tarjan. 2014. "Efficient maximum flow algorithms." *Communications of the ACM*, 57, 8, 82–89.

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

W. Hönig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian. 2018. "Conflict-based search with optimal task assignment." In: *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018)*, 757–765.

D. Kornhauser, G. Miller, and P. Spirakis. 1984. "Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications." In: *Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS) 1984*, 241–250.

J. Kottinger, S. Almagor, and M. Lahijanian. 2022. "Conflict-based search for multi-robot motion planning with kinodynamic constraints." In: *Proceedings of the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2022)*. IEEE, 13494–13499.

H. W. Kuhn. 1955. "The Hungarian method for the assignment problem." *Naval research logistics quarterly*, 2, 1-2, 83–97.

J. Li, D. Harabor, P. J. Stuckey, A. Felner, H. Ma, and S. Koenig. 2019. "Disjoint splitting for multi-agent path finding with conflict-based search." In: *Proceedings of The 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 279–283.

J. Li, P. Surynek, A. Felner, H. Ma, and S. Koenig. 2019. "Multi-agent path finding for large agents." In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7627–7634.

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

M. Liu, H. Ma, J. Li, and S. Koenig. 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)*.

E. M. Loiola, N. M. M. De Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido. 2007. "A survey for the quadratic assignment problem." *European journal of operational research*, 176, 2, 657–690.

R. Luna and K. E. Bekris. 2011. "Push and Swap: Fast cooperative path-finding with completeness guarantees." In: *Proceedings of The 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 294–300.

H. Ma. 2022. "Graph-based multi-robot path finding and planning." *Current Robotics Reports*, 3, 3, 77–84.

H. Ma and S. Koenig. 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.

H. Ma, J. Li, T. Kumar, and S. Koenig. 2017. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks." In: *Proceedings of The 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2017)*. International Foundation for Autonomous Agents and Multiagent Systems, 837–845.

A. Mahanti and A. Bagchi. Jan. 1985. "AND/OR graph heuristic search methods." *J. ACM*, 32, 1, (Jan. 1985), 28–51. doi:10.1145/2455.2459.

A. S. Mahanti and K. Ray. 1987. "Algorithms for Heuristic Search Analysis and Applications." *Journal of the ACM*, 34, 2, 298–312. doi:10.1145/24717.24719.

A. Martelli. 1977. "On the complexity of admissible search algorithms." *Artificial Intelligence*, 8, 1, 1–13. doi:https://doi.org/10.1016/0004-3702(77)90002-9.

L. Mérõ. 1984. "A heuristic search algorithm with modifiable estimate." *Artificial Intelligence*, 23, 1, 13–27. doi:https://doi.org/10.1016/0004-3702(84)90003-1.

A. Moldagalieva, J. Ortiz-Haro, M. Toussaint, and W. Hönig. 2024. "db-CBS: Discontinuity-Bounded Conflict-Based Search for Multi-Robot Kinodynamic Motion Planning." *Proceedings of the 2024 IEEE International Conference on Robotics and Automation (ICRA 2024)*, accepted.

V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh. 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.

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

K. Okumura, M. Machida, X. Défago, and Y. Tamura. 2022. "Priority inheritance with backtracking for iterative multi-agent path finding." *Artificial Intelligence*, 310, 103752.

D. W. Pentico. 2007. "Assignment problems: A golden anniversary survey." *European Journal of Operational Research*, 176, 2, 774–793.

A. Ravindran and V. Ramaswami. 1977. "On the bottleneck assignment problem." *Journal of Optimization Theory and Applications*, 21, 451–458.

G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. 2015. "Conflict-based search for optimal multi-agent pathfinding." *Artificial Intelligence*, 219, 40–66.
K. Solovey and D. Halperin. 2016. "On the hardness of unlabeled multi-robot motion planning." *The International Journal of Robotics Research*, 35, 14, 1750–1759.
T. S. Standley. 2010. "Finding optimal solutions to cooperative pathfinding problems." In: *Proceedings of The 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, 173–178.
R. Stern et al.. 2019. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks." *Symposium on Combinatorial Search (SoCS)*, 151–158.
P. Surynek. 2009. "A novel approach to path planning for multiple robots in bi-connected graphs." In: *Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009)*, 3613–3619.
P. Surynek. 2010. "An optimization variant of multi-robot path planning is intractable." In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, 1261–1263.
P. Surynek, A. Felner, R. Stern, and E. Boyarski. 2016. "Efficient SAT approach to multi-agent path finding under the sum of costs objective." In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*. IOS Press, 810–818.
N. Tomizawa. 1971. "On some techniques useful for solution of transportation network problems." *Networks*, 1, 2, 173–194.
G. Wagner and H. Choset. 2015. "Subdimensional expansion for multirobot path planning." *Artificial intelligence*, 219, 1–24.
B. de Wilde, A. W. ter Mors, and C. Witteveen. 2013. "Push and rotate: cooperative multi-agent path planning." In: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, 87–94.
Q. Xu, J. Li, S. Koenig, and H. Ma. 2022. "Multi-Goal Multi-Agent Pickup and Delivery." In: *Proceedings of the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2022)*.
J. Yu and S. M. LaValle. 2013a. "Multi-agent path planning and network flow." In: *Algorithmic Foundations of Robotics X: Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics*. Springer, 157–173.
J. Yu and S. M. LaValle. 2015. "Optimal multi-robot path planning on graphs: Structure and computational complexity." *arXiv preprint arXiv:1507.03289*.
J. Yu and S. M. LaValle. 2016. "Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics." *IEEE Transactions on Robotics*, 32, 5, 1163–1177.
J. Yu and S. M. LaValle. 2013b. "Planning optimal paths for multiple robots on graphs." In: *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*, 3612–3617.
Z. Zhang, N. R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner. 2009. "A* search with inconsistent heuristics." In: *Twenty-First International Joint Conference on Artificial Intelligence*.

Received 02 June 2025; accepted 22 March 2026