Fast and Scalable Rule-Based Search for Deadline-Constrained Anonymous Multi-Agent Path Finding

Sahar Badri[1], **Serafino Cicerone**[1], **Alessia Di Fonso**[1]

[1]University of L'Aquila, Via Vetoio I, 67100 L'Aquila, Italy
sahar.badri@graduate.univaq.it, serafino.cicerone@univaq.it, alessia.difonso@univaq.it

Abstract

Anonymous Multi-Agent Path Finding (AMAPF) requires coordinating a set of interchangeable agents to reach a set of target locations. In the variant with Individual Deadlines (AMAPFwID), each target must be reached before a specified time limit. Although AMAPFwID is solvable in polynomial time, state-of-the-art approaches rely on max-flow computations over time-expanded networks, whose size grows quadratically with the workspace. This makes them computationally impractical for large-scale instances involving thousands of agents.

We introduce DART (Deadline-Aware Rapid Target-swapping), a highly scalable rule-based framework for AMAPFwID. The method decomposes the problem into two integrated phases: (1) a *Task Assignment Phase*, using either Bottleneck Assignment with Cost Refinement (BACR) or Deadline-Aware Assignment with Conflict Refinement (DACR) to produce initial pairings; and (2) a *Reactive Search Phase*, which employs seven deterministic motion rules to resolve local spatio-temporal conflicts. A key component of the approach is an *Excess Time* heuristic that prioritizes agents based on their temporal slack, guiding the search toward deadline-feasible configurations.

We provide a theoretical analysis of correctness and test DART on standard MAPF benchmarks. The results show that DART scales to maps with thousands of agents, reducing runtime by several orders of magnitude compared to the optimal solver while maintaining near-optimal solution quality (around 1.01 times the optimal sum-of-moves).

Introduction

Multi-agent path finding (MAPF) addresses the problem of computing collision-free paths for multiple agents moving on a shared graph from given start vertices to designated targets (Stern et al. 2019) . In many practical settings, however, agents are not preassigned to specific goals. Instead, both the assignment of agents to targets and the computation of feasible paths must be determined jointly. This leads to the anonymous or unlabeled variant (AMAPF), where agents and targets are interchangeable (Kloder and Hutchinson 2006; Yu and LaValle 2012). A further level of complexity arises when each target is associated with a deadline that specifies the latest time by which it must be reached and continuously occupied (Fine, Atzmon, and Agmon 2023). This variant, known as AMAPF with Individual Deadlines (AMAPFwID), captures important constraints in time-critical applications such as warehouse logistics, robotic fleets, and large-scale autonomous systems (e.g. (Wang and Chen 2022; Ma et al. 2018)) Algorithmically, AMAPFwID combines assignment, routing, and temporal feasibility constraints into a single combinatorial optimization problem. The classical (non-anonymous) MAPF is NP-hard for different known objective functions (Yu and LaValle 2013; Yu 2016; Surynek 2010; Banfi, Basilico, and Amigoni 2017). In contrast, optimal solutions to AMAPF and AMAPFwID can be obtained in polynomial time via reductions to network flow (Yu and LaValle 2012; Fine, Atzmon, and Agmon 2023). These approaches rely on time-expanded graphs whose size grows quadratically with the environment, limiting their practical applicability to relatively small instances. This motivates the development of fast, scalable, and suboptimal algorithms that can handle hundreds or thousands of agents. Recently, search-based and rule-based approaches have shown strong empirical performance on large-scale MAPF and AMAPF instances (Li et al. 2019; Okumura and Défago 2023). However, none of the existing methods support deadline constraints, and therefore cannot be applied directly to AMAPFwID.

In this paper, we address this limitation and propose DART (Deadline-Aware Rapid Target-swapping), a highly scalable rule-based search framework for solving the AMAPFwID problem. DART decouples the problem into two integrated phases:

1. The **Task Assignment Phase**, which pairs agents with targets using two distinct strategies. The Bottleneck Assignment with Cost Refinement (BACR) minimizes the maximum travel distance to optimize makespan, then refines the solution to reduce total cost while preserving that bottleneck value. Deadline-aware Assignment with Conflict Refinement (DACR) applies the Hungarian algorithm to minimize sum-of-moves and subsequently applies local search to reduce potential path conflicts.
2. The **Reactive Search Phase** that resolves spatial–temporal conflicts through seven deterministic motion rules. A key component is the *Excess Time* heuristic that dynamically prioritizes agents based on their temporal slack, guiding the search toward

deadline-feasible configurations.

From a combinatorial search perspective, the second phase of DART can be viewed as an iterative process that dynamically resolves conflicts and assignment inconsistencies through local search and structured swapping operations.

Contributions We formally prove the correctness of DART and evaluate its computational complexity. To assess its practical applicability, we conduct an extensive experimental evaluation on four-connected grids from classic MAPF benchmarks (Stern et al. 2019), comparing DART against the only available algorithm for AMAPFWID, that is the polynomial-time optimal algorithm of (Fine, Atzmon, and Agmon 2023) under the sum-of-moves objective. These are the key results:

- On highly intricate grids with many agents, DART may fail (it is not complete), especially under the BACR assignment strategy, which may increase spacial and temporal conflicts compared to the more robust DACR variant. Nevertheless, DACR succeeds in nearly all cases, achieving a 100% success rate in most of the cases and above 93% in the remaining ones.
- Remarkably, solution quality remains near-optimal throughout: DACR usually stays below a 1.06 suboptimality ratio, while BACR generally falls within 1.02–1.05 of the optimal sum-of-moves.
- DART is several orders of magnitude faster than the optimal flow-based solver across all maps and agent densities. While the optimal method's runtime grows rapidly with the number of agents, DART consistently remains below a few seconds and scales approximately linearly up to thousands of agents.

All together, these results demonstrate that DART offers an effective and practical framework for deadline-constrained AMAPF, and represents a breakthrough in large-scale practical applications.

Problem Definition

We are given an undirected unweighted graph $G = (V, E)$ where V is a set of vertices, corresponding to the locations in the shared environment, and $E \subseteq V \times V$ is a set of edges, corresponding to spatial connections between locations. We call $P = (s = v_1, v_2, \ldots, v_k = t)$ a **path** connecting a pair of vertices $s, t \in V$ in G when $\{v_i, v_{i+1}\} \in E$ for all $i \in [1, k-1]$. The *length* of a path P is the number of edges of P. A **shortest path**, for a pair $s, t \in V$, is a path having minimum length among all paths in G connecting s and t. The **distance** $d(s, t)$ between s and t is the length of a shortest path P_{st}. We use $diam(G)$ to denote the diameter of G, that is, the maximum distance between any two vertices in the graph G. Given a vertex $v \in V$, $N_G(v) = \{u \in V | \{v, u\} \in E\}$ identifies the neighborhood of v in G, whereas $N_G[v] = N_G(v) \cup \{v\}$ is its closed neighborhood. $\Delta(G)$ represents the maximum degree, that is, the size of the largest set $N_G(v)$ for each vertex v.

Formalization of the Problem For clarity, we adopt the notation and terminology used in (Fine, Atzmon, and Agmon 2023; Stern et al. 2019). Accordingly, an instance of the AMAPFWID problem can be modeled as a 5-tuple (G, A, A_l, T, T_d), where:

- $G = (V, E)$ is an undirected graph;
- $A = \{a_1, a_2, \ldots, a_n\}$ is a set of n *agents*;
- $A_l : A \to V$ is an injective function identifying the initial *agent locations* in G;
- $T = \{g_1, g_2, \ldots, g_n\} \subseteq V$ is a set of n *target* or *goal locations* in G, disjoint from the agents' locations;
- $T_d : T \to \mathbb{N}$ is a function defining the *target deadlines*;

Agents are initially located on the vertices of G as specified by the location function A_l. Then, agents can move on G by performing a single **action** at each timestep (time is assumed to be discretized). As in classical MAPF, an action is an operation performed by an agent between two consecutive timesteps t and $t + 1$. There are two types of actions: **wait** and **move**. A wait action means the agent stays in its current vertex for another timestep. A move action implies that the agent moves from its current vertex v to an adjacent vertex $v' \in N(v)$ in the graph.

A **path for an agent** $a_i \in A$ is defined as a sequence of locations (vertices of G) π_i such that the agent either performs wait or move actions between two consecutive timesteps. Formally, if $\pi_i = (v_0, v_1, \ldots, v_{k-1})$ is a path for the agent $a_i \in A$, then for each t, $0 \leq t < k-1$, either $(v_t, v_{t+1}) \in E$ or $v_t = v_{t+1}$. We use notation $\pi_i[t]$ to identify the location of agent a_i at timestep t, with $0 \leq t \leq k-1$. A **plan** for the agents in A is a set $\Pi_A = \{\pi_1, \pi_2, \ldots, \pi_n\}$ containing one path π_i for each agent $a_i \in A$ and such that π_i starts exactly at the initial location of a_i, that is $\pi_i[0] = A_l(a_i)$.

We say there is a **conflict** in a plan Π_A if one of the following two situations occurs:

1. there is a **vertex conflict** when at least two agents occupy a vertex at the same timestep; formally, there exist a vertex $v \in V$, two distinct agents $a_i, a_j \in A$, and a timestep $t > 0$ such that $\pi_i[t] = \pi_j[t] = v$;
2. there is an **edge conflict** where two agents traverse an edge at the same timestep; formally, there exist $e = (u, v) \in E$, $a_i, a_j \in A$, and a timestep $t > 0$ such that $\pi_i[t] = u$, $\pi_j[t] = v$, $\pi_i[t+1] = v$, and $\pi_j[t+1] = u$.

If Π_A does not have a conflict, then we say the plan is **conflict-free**.

Informally, solving AMAPFWID requires computing a conflict-free plan Π_A that allows each target to be reached (or acquired), by an agent within its deadline. We also assume that agents are allowed to leave a target location at any time, even after the deadline provided that another agent simultaneously takes their place so that the target remains acquired. In particular, if an agent a_i that first acquires a target g (i.e., a_i is on g at a given timestep $t \geq T_d(g)$), it may move away at timestep $t + 1$ only if another agent a_j moves onto g at the same timestep, ensuring continuous acquisition. We refer to this transfer of target acquisition between agents as **swapping behavior**.

Finally, a **solution** for AMAPFWID that allows the swapping behavior is a conflict-free plan $\Pi_A = \{\pi_1, \pi_2, \ldots, \pi_n\}$

such that for each $g \in T$ and for each $t \geq T_d(g)$, there exists a path $\pi_i \in \Pi_A$ such that $\pi_i[t] = g$.

The quality of a solution $\Pi_A = \{\pi_1, \pi_2, \ldots, \pi_n\}$ is usually measured according to one of the following metrics: **makespan** (the first timestep T at which all targets are acquired) and **sum-of-moves** (the total number of moves summed over all agents, i.e., total travel distance).

The Algorithm

DART is presented in the pseudo-code shown in Algorithm 1. It takes as input an instance $I = (G, A, A_l, T, T_d)$ of AMAPFWID, and returns a plan Π_A solving I. If I is unfeasible or the algorithm is not able to compute a solution for I, then DART returns $\Pi_A = \emptyset$.

The algorithm assumes the availability of a primitive getAssignment(), treated as an external module that computes an *initial* assignment $M : A \to T$ together with a set of paths $\mathcal{P}$ containing a path P_i from $A_l(a_i)$ to $M(a_i)$ for each agent $a_i \in A$. Different assignment strategies may significantly affect solution quality, and therefore their discussion is deferred to the next section. The pair $(M, \mathcal{P})$ is not intended to be a valid solution to AMAPFWID: when agents follow the paths in $\mathcal{P}$ concurrently, path intersections and congestion may arise. For this reason, DART includes a reactive search module that can revise both assignments and paths at runtime, allowing agents to swap targets when necessary to meet deadlines and avoid conflicts.

The reactive search module implemented by DART proceeds in discrete timesteps. Each agent a_i maintains two variables: (1) $a_i.v$ its current location (initialized to $A_l(a_i)$), and (2) $a_i.g$, its current goal (initialized to $M(a_i)$). Each target $g \in T$ stores (1) $g.acquiredTime$, the timestep at which some agent first reached g, and and (4) $g.lost$, the timestep at which the agent that acquired g moved away. Both are initialized to -1 (see lines 2-8 of DART for the variables' initialization).

The Excess Time Heuristics At every timestep, agents are processed one by one to determine their next vertex in the final plan. Because deadlines impose temporal constraints, the order in which agents are considered is crucial. This ordering is based on the notion of *excess time* associated to each agent as follow:

Definition 1. *The excess time of a_i right after timestep t is defined as follows:* $a_i.excessTime = (T_d(a_i.g) - t) - d(a_i.v, a_i.g)$.

Notice that, for an agent a_i, $T_d(a_i.g) - t$ is the remaining time before the deadline of $a_i.g$ expires. Hence, when $d(a_i.v, a_i.g) > 0$, $a_i.excessTime$ represents the maximum number of waiting steps the agent can afford. According to $excessTime$, the set A can be partitioned into two groups: A^+ with agents a_i such that $d(a_i.v, a_i.g) > 0$ (i.e., not yet at their target), and A^0 with agents a_i such that $d(a_i.v, a_i.g) = 0$. Then, A^+ and A^0 are used to construct an ordered list A^{ord}:

- Agents in A^+ are inserted first, sorted by increasing $a_i.excessTime$, giving priority to those with less temporal slack.

Algorithm 1 DART

Input: Instance $I = (G, A, A_l, T, T_d)$ of AMAPFWID
Output: Plan Π_A solving I ($\Pi_A = \emptyset$ for I unfeasible or unsolvable)

```
1: M, P ← getAssignment(A, T)
2: for each a_i ∈ A do
3:     a_i.v ← A_l(a_i)
4:     a_i.g ← M(a_i)
5:     π_i[0] ← a_i.v
6: for each g ∈ T do
7:     g.acquiredTime ← −1
8:     g.lost ← −1
9: t ← 1
10: while ∃ a ∈ A such that a.v ≠ a.g do
11:     for each a_i ∈ A do
12:         a_i.excessTime ← (T_d(a_i.g) − t + 1) − d(a_i.v, a_i.g)
13:     A^ord = getOrderedAgents(A)
14:     for each a_i ∈ A^ord do
15:         makeAMove(a_i)
16:     for a_i ∈ A do
17:         π_i[t] ← a_i.v
18:     t ← t + 1
19: if ∃g ∈ T : g.acquiredTime > T_d(g) :
20:     return Π_A ← ∅
21: else
22:     return Π_A ← {π_1, π_2, ..., π_n}
```

- Agents in A^0 follow, sorted by decreasing $a_i.excessTime$. Agents that acquired their target long ago typically have very small (possibly negative) excess time and therefore receive low priority. They must still be considered because rules may trigger target swaps.

For convenience, we set $a_i.priority = |A| - pos(a_i, A^{ord})$, where $pos(a_i, A^{ord})$ denotes the position of a_i into A^{ord}, so agents appearing earlier in A^{ord} receive higher priority. At every timestep, DART computes A^{ord} and processes agents in this priority order starting with the A^+ agent with the smallest excess time.

Concerning the pseudo-code, the main cycle at Lines 10 iterates until all agents reach their targets. This implements the reactive search module that evaluates the elements of $\mathcal{P}$ as candidate paths and adjusts them when path intersections or congestion occur. When the reactive search module starts, $a_i.excessTime$ is set for each agent a_i (see Line 11), and then these values are used to construct the ordered list of the agents A^{ord} (see Line 13). Then, the cycle at Line 14 analyzes each agent a_i in A^{ord}, following the order defined by $a_i.priority$, by calling procedure makeAMove(a_i) (see the pseudo-code in Algorithm 2) which selects the appropriate rule for agent a_i at the timestep t. Note that the ordering in A^{ord} may change dynamically within the same timestep through conflict-resolution rules.

Rules Procedure makeAMove determines, through an ordered sequence of condition checks, which of the seven rules applies to a_i at timestep t. The applied rule determines the next position of a_i, using either the original path P_i or its updated version.

For an agent a_i, the function $nextVertex(a_i.v, P_i)$ returns the next vertex u along P_i from the current position $a_i.v$ toward the target $a_i.g$. Each rule has an applicability condition based on the status of u (e.g., whether it is occupied), and is evaluated according to a precedence order. When a rule is applied, the agent's position $a_i.v$ is updated; its target $a_i.g$ and the path P_i may also be updated.

Note that five of the seven rules are adapted from TSWAP (Okumura and Défago 2023) solving AMAPF. Except for StayOnTarget, the original rules are modified to account for deadlines, while the two new rules (Detour and Enqueue) are specifically designed to handle temporal constraints. Collectively, the rules aim to maximize concurrent agent movements while ensuring deadline feasibility. We now give a high-level description of each rule, formalized in Algorithm 2. Note that, the comment "R. #k" indicates the start of rule k testing.

1. **StayOnTarget**: it applies when the agent a_i has already reached its target (i.e., $a_i.v = a_i.g$). In this case, a_i remains in place.
2. **SwapTargets**: it applies when the next vertex u of agent a_i along P_i is occupied by an agent a_j that has already reached its target (i.e., $a_j.v = a_j.g$). Here $d(a_j.v, a_j.g) = 0$, so $a_j \in A^0$ and $a_i \in A^+$, implying $a_i.priority > a_j.priority$. Since a_j does not move during this timestep (due to StayOnTarget), letting a_i enter u would cause a vertex conflict. To prevent this, the rule swaps the targets of a_i and a_j, updates P_i and P_j, and sets $agentWaiting[u] = a_i$. When a_j is processed later in the same timestep, if it leaves u, the waiting agent a_i is immediately reconsidered and may move into u.
3. **Enqueue**: it is applied when the next vertex u along P_i is occupied by another agent a_j and $a_i.priority > a_j.priority$. As in the swap target rule, the algorithm sets $agentWaiting[u] = a_i$. When a_j is processed during the same time step, if it moves from u, then a_i, the agent waiting at u, is re-evaluated and, upon finding u free, moves into it.
4. **Detour**: This rule applies when the next vertex u along P_i is occupied by an agent a_j and $a_i.priority < a_j.priority$. Since a_j has already been processed in the current timestep, u will remain occupied. The rule therefore searches the neighborhood of a_i for an alternative unoccupied vertex u' that still allows a_i to meet its deadline. If such a vertex exists, a_i moves to u' (by applying MoveToTarget rule), and P_i is updated accordingly.
5. **Deadlock**: a set of agents $A' = \{a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}\}$, $k \geq 2$, is in a deadlock when $nextVertex(a_{i_j}.v, a_{i_j}.g) = a_{i_{j+1}}.v$ for each $0 \leq j \leq k-1$ (in this definition of deadlock of k agents, the addition is intended modulo k). The detection of a deadlock is done by incrementally checking whether the next location of each agent is occupied by another agent and concurrently checking the existence of a loop. During the analysis of a possible move for an agent a_i, the existence of a deadlock must be checked. If the algorithm detects a deadlock in a set A' containing a_i, the algorithm first "rotates" targets by letting $a_{i_{j+1}}.g = a_{i_j}.g$ for each $0 \leq j \leq k-1$, and then updates path P_{i_j} for each $0 \leq j < k$. Then, MoveToTarget rule is applied to agent a_i.
6. **MoveToTarget**: Assume that a_i is located at w (i.e., $a_i.v = w$). If the next vertex u along P_i is unoccupied, the agent moves to u (i.e., $a_i.v = u$). If u is a target, the rule updates $u.acquiredTime$ (if this is the first acquisition) and resets $u.lost = -1$. Finally, if some agent is waiting at w (i.e., $agentWaiting[w]$ is defined), that agent is immediately processed.
7. **Wait**: If none of the rules above apply, then the agent remains at its current vertex.

Once all agents have reached their targets, a feasibility check is performed at line 19 of Algorithm 1: if there exists a target $g \in T$ such that $g.acquiredTime > T_d(g)$, then instance I is infeasible or DART is unable to compute a valid solution. In both cases, the algorithm returns an empty plan Π_A.

Algorithm 2 makeAMove

Input: Agent a_i
Output: Determine which rule can be applied to a_i

if $a_i.v = a_i.g$ **:** ▷ *R. #1*
  return
$u \leftarrow \text{nextVertex}(a_i.v, P_i)$
if $\exists a_j : a_j.v = u$ **:**
  $u.\text{agent} \leftarrow a_j$
else
  $u.\text{agent} \leftarrow \text{nil}$
if $u.\text{agent} \neq \text{nil} \wedge u.\text{agent}.g = u$ **:** ▷ *R. #2*
  $a_i.g, u.\text{agent}.g \leftarrow u.\text{agent}.g, a_i.g$
  Update paths in $\mathcal{P}$ for a_i and $u.\text{agent}$
  $\text{agentWaiting}[u] \leftarrow a_i$
  return
if $u.\text{agent} \neq \text{nil} \wedge a_i.\text{priority} > u.\text{agent.priority}$ **:** ▷ *R. #3*
  $\text{agentWaiting}[u] \leftarrow a_i$
  return
if $u.\text{agent} \neq \text{nil} \wedge a_i.\text{priority} < u.\text{agent.priority}$ **:** ▷ *R. #4*
  if $\exists u' \in N_G(a_i.v) : u'.\text{agent} = \text{nil} \wedge T_d(a_i.g) - t + 1 \geq d(u', a_i.g) + 1$ **:**
    $u \leftarrow u'$
    Update path in $\mathcal{P}$ for a_i ▷ *then, R. #6 applies*
if $\exists$ deadlock for $A' \subseteq A \wedge a_i \in A'$ **:** ▷ *R. #5*
  Rotate targets for agents in A'
  Update paths in $\mathcal{P}$ for each agent in A'
  $u \leftarrow \text{nextVertex}(a_i.v, P_i)$ ▷ *then, R. #6 applies*
if $u.\text{agent} = \text{nil}$ **:** ▷ *R. #6*
  $\text{prev}, a_i.v \leftarrow a_i.v, u$
  if $u \in T$ **:**
    if $u.\text{acquiredTime} = -1 \vee u.\text{lost} < t$ **:**
      $u.\text{acquiredTime} \leftarrow t$
    $u.\text{lost} = -1$
  if $\text{prev} \in T$ **:**
    $\text{prev.lost} \leftarrow t$
  if $\exists a_j \in A : a_j = \text{agentWaiting}[\text{prev}]$ **:**
    $\text{makeAMove}(a_j)$
  return
return ▷ *R. #7*

Correctness and Complexity

In this section, we formally analyse the correctness and the computational complexity of DART.

Lemma 2. *If DART is executed over an instance $I = (G, A, A_l, T, T_d)$ of the* AMAPFwID *problem, then the cycle at Line 10 terminates, and each agent reaches a target.*

Proof. Consider the potential function $\phi = \sum_{a_i \in A} d(a_i.v, a_i.g)$. Clearly, when $\phi = 0$, each agent has reached the assigned target.

Assume first that the Detour rule is never applied. We now prove that ϕ decreases for each timestep t elaborated by DART in the cycle at Line 10. For any given timestep $t > 0$, if there exists any agent a_i that perform the MoveToTarget rule, then ϕ decreases by at least one unit. By contradiction, let us assume that there is a timestep t' in which no agent performs MoveToTarget. This implies that there exists a subset A' of agents forming a deadlock, while the others perform Wait.

According to the Deadlock rule, DART: (1) resolves the deadlock by rotating all the targets for agents in A', (2) updates paths in $\mathcal{P}$ for each agent in A', and (3) forces one (or more) agent to perform MoveToTarget, a contradiction. Hence, ϕ decreases at each timestep $t > 0$. This in turn implies that in a finite number of timesteps, each agent reaches the assigned target, and the loop at Line 10 terminates. Hence, a plan Π_A is returned by DART.

Assume now that Detour is applied by an agent a_i. The new path computed for a_i may increase ϕ. Since the new path must always guarantee that the target is reached within the deadline, Detour can be applied to a_i for a limited number of times. Therefore, even though ϕ may fluctuate, there is certainly a finite time at which ϕ becomes zero. □

Theorem 3. *Let Π_A be the plan obtained by running DART over an instance $I = (G, A, A_l, T, T_d)$ of the* AMAPFwID *problem. If $\Pi_A \neq \emptyset$, then Π_A is a solution for I.*

Proof. By Lemma 2, DART returns a plan Π_A. According to the statement, assume $\Pi_A = \{\pi_1, \pi_2, \ldots, \pi_n\}$. Algorithm DART uses a valid initial assignment $M : A \to T$ to initialize $\pi[0]$ for each agent a_i (see Line 5). If the algorithm terminates after t^* timesteps, the condition at Line 10 assures that $\pi[t^*]$ contains a target in T for each agent $a_i \in A$.

Π_A is conflict-free. In fact, if t is any timestep such that $0 < t < t^*$, a vertex u occupied at timestep t by any agent a_i cannot also be occupied by another agent a_j, $i \neq j$, because the MoveToTarget rule checks that only unoccupied vertices can be destinations of a move. Moreover, the Deadlock rule avoids that any edge in G is traversed in opposite directions by two agents at the same timestep t.

If an agent $a_i \in A$ acquires its target $g \in T$ at a timestep $t' < t^*$, then either a_i remains on target till the final timestep t^* (cf. StayOnTarget rule), or a_i is replaced by other agents (cf. SwapTargets rule). In any case, g remains acquired at each timestep from t' to t^*. If a_i acquires g, but g is not its target, then g remains acquired as long as a_i remains on g: if a_i leaves g at a timestep t, g remains acquired only if another agent a_j move on g at the same timestep t (cf. MoveToTarget and properties $g.acquiredTime$ and $g.lost$).

Finally, all the targets are acquired in time before their respective deadlines, as ensured by the feasibility check at Line 19 within the makeAMove procedure. □

Concerning the time complexity of DART, let $\Gamma = \max_{g \in T}(T_d(g))$ be the maximum deadline.

Theorem 4. *Disregarding the time required to compute the initial assignment, the time complexity of DART for computing a solution $\Pi_A \neq \emptyset$ is $O(\Gamma|A|(|V| + |E|))$.*

Proof. We first analyze the complexity of makeAMove (see Algorithm 2), invoked at Line 15 of DART. This procedure evaluates all the rules, and its complexity is dominated by the Detour and Deadlock rules. The Detour rule applies when an agent a_i discovers that its next hop toward the target is occupied by an agent a_j having a higher priority. In this case, a_i attempts to find an alternative path to reach the target within the deadline. In the worst-case, the shortest path distances from all its neighboring nodes to the target $a_i.g$ must be computed. Since G is unweighted, these distances can be obtained via a single backward BFS in $\mathcal{O}(|V| + |E|)$ time. For the Deadlock rule, consider the worst-case scenario in which all agents are involved and, for each of them, all shortest paths to their targets must be recomputed. By the same argument used for Detour, this requires $O(|A|(|V| + |E|))$ time.

Turning to DART, the initialization phase (Lines 2 and 6) requires $O(|A|)$ and $O(|T|)$ time. The main loop (Lines 10-18) repeats until all agents reach their targets. The number of iterations is bounded by the makespan guaranteed by the algorithm, which is trivially upper-bounded by the maximum deadline $\Gamma = \max_{g \in T} T_d(g)$. During each iteration of the main loop, the algorithm first computes the ordered list A^{ord} in $O(|A| \log |A|)$ time, then calls makeAMove for each agent, costing $O(|A|(|V| + |E|))$ time, as discussed above, and finally updates Π in $O(|A|)$ time. The feasibility check at Line 22 requires $O(|T| + |A|)$ time. Summing up all contributions, the total running time required by DART is

$$O(|A| + |T| + \Gamma(|A| \log |A| + |A|(|V| + |E|))),$$

which results in a $O(\Gamma|A|(|V| + |E|))$ total time. □

Lemma 2 proves that DART is complete without deadlines. With deadlines, DART may fail (the computed plan does not meet some deadlines). However, as we will see in the experimental section, DART achieves a very high success rate across the majority of benchmark tests.

Target Assignment

Solving AMAPFwID begins with assigning agents to targets and taking target deadlines into account. We adopt two assignment strategies that emphasize different aspects of solution quality: the first is designed to favor a smaller makespan (although it is refined to also reduce the other metric), while the second aims to reduce the overall sum-of-moves.

Bottleneck Assignment with Cost Refinement (BACR) The assignment is computed through a two-stage procedure. The first stage solves the classical bottleneck assignment problem (Gross 1959), finding a perfect matching that minimizes the largest agent–target distance and therefore the makespan. The second stage refines the assignment by computing a minimum-cost maximum matching that minimizes the sum-of-moves while preserving the bottleneck value. Optimizing firstly the bottleneck and then the total cost is a standard technique in the assignment literature (see, e.g., (Burkard, Dell'Amico, and Martello 2009)) and has been applied to MAPF in (Turpin et al. 2014; Okumura and Défago 2023). For implementing BACR, we extended the strategy of (Okumura and Défago 2023) to handle individual deadlines. The final complexity of BACR is $O(\max(A(V+E), A^4))$.

Deadline-Aware Assignment with Conflict Refinement (DACR) It is a two-stage assignment strategy designed to pursue a dual objective: first, to compute paths that minimize the overall travel cost, and second, to identify and reduce potential conflicts among these paths. This approach provides a global, a-priori view of agent interactions, allowing the algorithm to anticipate and mitigate conflicts that would otherwise be handled only locally during the reactive search phase. To this end, DACR minimizes the following objective function: $F(\mathcal{P}) = \sum_{i=1}^{k} |P_i| + \sum_{i \neq j} conf(P_i, P_j)$.

In the first stage, DACR optimizes the first component of the objective. It computes an initial assignment according the minimum-cost matching problem using the Hungarian algorithm (Kuhn 2010), which runs in $O(|A|^3)$. The algorithm is applied to a deadline-aware cost matrix where, for each agent a_i and target g_j, the entry is defined as $C_{ij} = d(a_i, g_j)$ if $d(a_i, g_j) \leq T_d(g_j)$, ∞ otherwise. This yields to an optimal assignment for the sum-of-moves metric, but does not account for conflicts between the agents, these are addressed in the second stage.

In the second stage, DACR attempts to reduce possible conflicts among paths induced by the initial assignment. Conflicts are defined as follows. Two paths $P' = (u_1, u_2, \ldots, u_{k'})$ and $P'' = (v_1, v_2, \ldots, v_{k''})$ are in *conflict* if there exists j, with $t < \min\{k', k''\}$, such that $u_j = v_j$. If P' and P'' are not in conflict, then $conf(P', P'') = 0$; otherwise, $conf(P', P'') = c$ if there are $j_1 < j_2 < \ldots < j_c$ such that $u_{j_i} = v_{j_i}$ for each $1 \leq i \leq c$. DACR refines the initial solution through a conflict-aware local search. It identifies the paths that generate the highest number of conflicts and computes up to k alternative routes for each of them. Each candidate is evalutated by updating the global objective function and the original path is replaced with the best improving alternative, if one exists. The complexity of this refinement phase is $O(I\ k\ |A||V|(|V|+|E|))$ where I is the number of iterations of the local search and k is the number of the computed alternative paths. Summarizing, DACR requires $O(A^3 + I\ k\ |A||V|(|V|+|E|))$ time. In the experimental setting, both k and I are fixed constants, making the second stage practically efficient.

Experimental Evaluation

We evaluate the performance and scalability of DART on the AMAPFWID problem. First, we study the impact of the two assignment strategies, BACR and DACR, on solution quality and computational efficiency. Then, we assess the scalability of DART by measuring its execution time in increasingly dense environments and on large-scale maps. Before presenting the results, we describe the experimental setup used throughout the evaluation.

The Optimal Algorithm We compare DART against the polynomial-time algorithm of (Fine, Atzmon, and Agmon 2023) (HOT variant), denoted as OPT, which solves AMAPFWID optimally under the sum-of-moves metric by reducing it to a minimum-cost maximum-flow instance, via a time-expanded network construction (Yu and LaValle 2012). Given the underlying graph G, OPT unfolds it over the discrete timesteps 0 to Γ (the maximum deadline), producing a time-expanded graph G' whose size grows quadratically with $|G|$. A super-source connects to each agent's start vertex at time 0, and a super-sink connects to each target at time Γ. By assigning appropriate arc costs, a minimum-cost flow on G' yields collision-free paths that minimize the sum-of-moves metric. We use OPT as our baseline because it is the only algorithm that directly solves AMAPFWID. Suboptimal MAPF solvers do not support individual deadlines and adapting them to handle such constraints lies outside of the scope of this work.

Implementation We implemented both DART and OPT in Python. For OPT, we constructed the time-expanded network via the reduction of (Fine, Atzmon, and Agmon 2023), and then we solved the Minimum Cost Flow problem using Orlin's enhanced capacity-scaling algorithm, which was shown to be highly efficient in (Kovács 2015). All the experiments ran on a MacBook Pro with an Apple M4 Max CPU and 36 GB RAM (Sequoia 15.7 OS).

Experiments We test our algorithm on the MovingAI benchmark maps (Stern et al. 2019), a widely used testbed in MAPF research (e.g., see (Aakash and Saha 2024; Fine, Atzmon, and Agmon 2023; Okumura and Défago 2023)). Maps presents unique obstacle patterns and bottlenecks, which challenge the algorithm in different ways. They cover a wide range of environments, including complex mazes and well-structured indoor layouts (e.g. warehouses, rooms with narrow passages, game dungeons, city-like layouts). Operatively, each map is converted into a graph on which agents move.

Given a benchmark graph $G = (V, E)$, we generate an AMAPFWID instance with n agents as follows:

- We sample n vertices uniformly at random to form the set T of target locations.
- We then sample another n distinct vertices from $V \setminus T$, uniformly at random, to define the agent's initial locations via the mapping $A_l : A \to V$.
- For each target $g \in T$, we assign a deadline $T_d(g)$ drawn uniformly at random from the integer interval

$\left\{\frac{diam(G)}{2}, \frac{diam(G)}{2} + \frac{|A|}{2}\right\}$. This strategy avoids both infeasible configurations and trivial instances with overly relaxed deadlines.

For a graph $G = (V, E)$, we consider 4 different agent densities, setting $n \in \{\sqrt{|V|}/4, \sqrt{|V|}/2, \sqrt{|V|}, 2\sqrt{|V|}\}$. For every fixed n, we generate 30 feasible instances of AMAPFWID by randomly sampling initial agents' locations, targets, and deadlines. (Random instances found unfeasible by OPT were discarded). We then run both DART (with each assignment strategy) and OPT on all the 30 instances. We evaluate algorithmic performances under a progressive doubling of agents and targets, a standard methodology for exposing trends and parameter sensitivity (McGeoch 2012). Solution quality is measured through makespan and sum-of-moves, while performance is assessed via total running time and success rate, defined as the percentage of instances solved by either algorithm over the full set. In the scalability experiment DART is further tested on large maps with an extended range of agent counts, obtained by doubling the number of agents from $\sqrt{|V|}/4$ up to $8\sqrt{|V|}$.

Results

The experimental results are reported in Table 1. Across all maps and agent densities, DART runs several orders of magnitude faster than OPT, while preserving near-optimal solution quality under both assignment strategies.

The **running times** of DART remains extremely low, typically below a few tenths of a second. In contrast, OPT running times increase rapidly with the number of agents (e.g., from 29 seconds to over 3500 seconds on a warehouse map with 300 agents).

The **success rate** of DART is consistently high, though it decreases as agents density increases and the environment becomes more congested. With the DACM assignment, DART typically solves over 93% of the instances even at high densities, dropping slightly only in the most challenging cases (e.g., 83–93% on the largest maps). BACR assignment shows a similar trend with a slightly sharper decline (e.g., 73–80% in the densest cases) suggesting that enforcing tighter makespan constraints, can increase spatial and temporal congestion during the reactive search phase, making conflict resolution harder. Empirically, failures occur only in very dense environments, where the limited slack prevents resolving conflicts within the required time windows. This effect is amplified by the instance generation process: deadlines are assigned uniformly at random, and in high-density settings even a small subset of agents may receive tight deadlines. Since all deadlines must be satisfied simultaneously, even the failure of a single agent makes the entire instance unsolved. Importantly, this limitation arises exclusively from the presence of deadlines. Without deadlines, DART is complete: it always terminates and returns a valid collision-free plan. In the deadline constrained setting, however, the strictly local nature of the decision rules, based on a 1-step look-ahead in the path-planning phase may prevent the algorithm from finding a feasible schedule even when one exists.

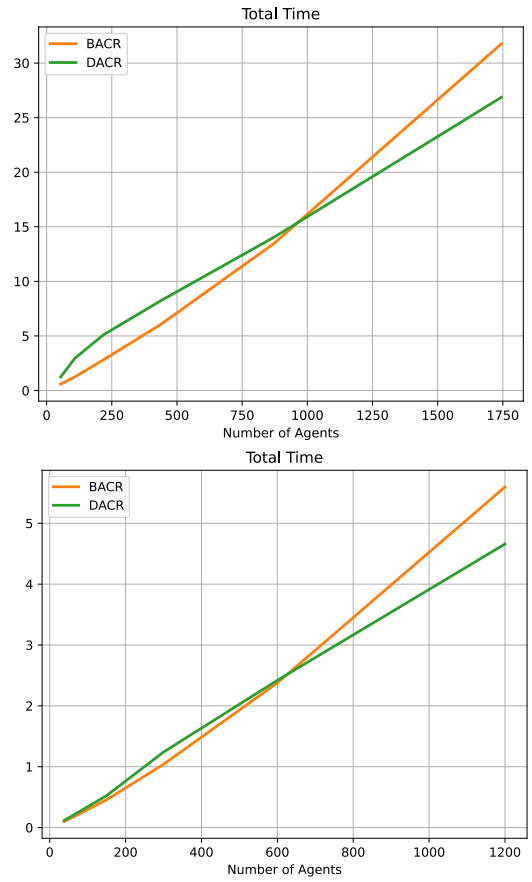

Figure 1: Results of the scalability experiment for the Berlin_1_256 and warehouse-20-40-10-2-1 maps, shown at the top and bottom respectively.

The **suboptimality** of DART remains remarkably low throughout all experiments. Both assignment strategies produce solutions very close to the optimal sum-of-moves. DACR assignment achieves the best performance, with suboptimality steadily around 1.01 and only marginal increases in the most congested settings (one case reaching 1.06). BACR shows slightly higher values (generally between 1.02 and 1.05), yet still maintains high overall quality.

Regarding **scalability**, experiments with up to thousands of agents, show that the running time of DART grows approximately linearly with the number of agents across all the tested maps (see Figure 1). This trend indicates that the algorithm preserves its efficiency as problem size increases, making it a promising approach for very large instances.

Finally, a notable outcome concerns the **makespan**. In all experiments, DART produce makespan significantly smaller than those obtained by OPT. This behavior is consistent with the different optimization goals of the two approaches: while both OPT and DART minimizes the sum-of-moves, DART 's conflict-aware strategy—applied both during the assignment phase (in the case of DACR) and during the reactive search phase, naturally favors shorter execution times. As a result, DART offers a qualitative advantage in scenarios where also

Map, Size, $\lvert V\rvert$	$\lvert A\rvert$	Time (s)			Sum-of-Moves					Makespan	
					Success rate (%)			Sub-optimality			
		OPT	BACR	DACR	OPT	BACR	DACR	BACR	DACR	BACR	DACR
den312d, 65×81, $\lvert V\rvert = 2445$	12	28.931	**0.007**	0.009	100.0	100.00	**100.00**	1.028	**1.000**	**48.833**	54.400
	24	41.116	**0.013**	0.049	100.0	96.67	**100.00**	1.033	**1.001**	**42.276**	56.345
	49	66.781	**0.031**	0.122	100.0	**96.67**	93.33	1.027	**1.002**	**38.074**	66.444
	98	118.625	**0.072**	0.265	100.0	73.33	**100.00**	1.027	**1.006**	**28.409**	67.136
room-64-64-8, 64×64, $\lvert V\rvert = 3232$	14	49.769	**0.009**	0.020	100.0	100.00	**100.00**	1.034	**1.005**	**47.767**	59.533
	28	71.001	**0.020**	0.059	100.0	100.00	**100.00**	1.050	**1.010**	**47.933**	65.700
	56	107.981	**0.044**	0.102	100.0	96.67	**100.00**	1.053	**1.019**	**42.724**	64.138
	112	181.748	**0.104**	0.348	100.0	76.67	**96.67**	1.110	**1.060**	**55.947**	71.421
random-64-64-20, 64×64, $\lvert V\rvert = 3270$	14	25.857	**0.009**	**0.009**	100.0	100.00	**100.00**	1.043	**1.001**	**34.600**	42.233
	28	40.913	**0.020**	0.029	100.0	100.00	**100.00**	1.045	**1.001**	**29.767**	42.533
	57	67.309	**0.044**	0.061	100.0	100.00	**100.00**	1.042	**1.002**	**23.333**	39.467
	114	133.447	**0.099**	0.141	100.0	80.00	**93.33**	1.047	**1.008**	**18.609**	35.000
room-64-64-16, 64×64, $\lvert V\rvert = 3646$	15	105.411	**0.012**	0.049	100.0	100.00	**100.00**	1.024	**1.000**	**59.233**	70.933
	30	158.462	**0.025**	0.064	100.0	**100.00**	96.67	1.024	**1.003**	**51.310**	76.724
	60	231.121	**0.056**	0.274	100.0	86.67	**100.00**	1.032	**1.008**	**45.308**	84.192
	120	383.907	**0.125**	0.478	100.0	73.33	**93.33**	1.026	**1.013**	**32.476**	74.000
random-64-64-10, 64×64, $\lvert V\rvert = 3687$	15	34.039	**0.011**	**0.011**	100.0	100.00	**100.00**	1.033	**1.000**	**31.633**	38.767
	30	51.450	**0.024**	0.033	100.0	100.00	**100.00**	1.044	**1.001**	**25.233**	39.533
	60	90.234	**0.054**	0.075	100.0	96.67	**100.00**	1.034	**1.002**	**21.207**	37.552
	120	177.040	**0.118**	0.150	100.0	96.67	**100.00**	1.038	**1.002**	**16.138**	32.793
ht-chantry, 162×141, $\lvert V\rvert = 7461$	22	311.722	**0.027**	0.061	100.0	100.00	**100.00**	1.050	**1.000**	**75.200**	90.367
	43	453.366	**0.058**	0.188	100.0	100.00	**100.00**	1.033	**1.001**	**68.567**	98.267
	86	676.964	**0.150**	0.603	100.0	96.67	**100.00**	1.039	**1.001**	**54.897**	99.276
	172	1420.259	**0.406**	1.438	100.0	83.33	**96.67**	1.038	**1.004**	**47.625**	112.167
warehouse-10-20-10-2-2, 170×84, $\lvert V\rvert = 6372$	24	95.764	**0.031**	0.036	100.0	100.00	**100.00**	1.047	**1.001**	**30.800**	40.967
	49	137.187	**0.068**	0.082	100.0	100.00	**100.00**	1.043	**1.001**	**24.833**	37.567
	98	277.127	**0.168**	0.201	100.0	100.00	**100.00**	1.049	**1.002**	**20.433**	40.833
	196	604.658	**0.356**	0.423	100.0	86.67	**93.33**	1.040	**1.003**	**15.792**	33.292
warehouse-20-40-10-2-1, 321×123, $\lvert V\rvert = 13399$	38	642.955	**0.124**	0.147	100.0	100.00	**100.00**	1.058	**1.000**	**39.700**	59.400
	75	827.931	**0.234**	0.274	100.0	100.00	**100.00**	1.047	**1.002**	**30.770**	57.500
	150	1519.466	**0.496**	0.572	100.0	100.00	**100.00**	1.048	**1.002**	**25.200**	50.200
	300	3563.529	**1.162**	1.399	100.0	96.67	**100.00**	1.045	**1.006**	**22.100**	41.600

Table 1: OPT against DART (with BACR and DACR). Bold values highlight the best results between BACR and DACR.

the completion time must be taken into account.

Conclusions

In this paper we proposed DART, a highly scalable rule-based framework for the AMAPFWID problem. A theoretical limitation of the current approach is its incompleteness (even if, in practice, it succeeded in nearly all the experimental cases). Failures occur only in very dense environments, where congestion reduces the effective slack available to each agent. This behavior suggests an interesting direction for future work. We plan to investigate more deeply the dynamics of delayed solutions, with the goal of understanding when and why local decisions prevent agents from recovering feasible timing. Enhancing the algorithm with a limited look-ahead, slack-aware heuristics, or improved conflict resolution mechanisms — both in the assignment phase and in the reactive search phase—may significantly reduce deadline-related failures. While such extensions will likely increase running times, the current efficiency of DART leaves ample room for these improvements.

Another promising direction is to adapt the DART framework to solve again the AMAPFWID problem, but optimizing the makespan. The Reactive Phase could remain unchanged with appropriate new assignment modules. Based on the last two columns of Table 1, BACR remains a strong candidate, potentially in a revised version where its second phase attempts to minimize conflicts similarly to DACR.

Acknowledgments

This article is published with funding from Università degli Studi dell'Aquila under the Call for Proposals for Fundamental research and Early-career research grants - Year 2026, the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National

Innovation Ecosystem grant ECS00000041 - VITALITY, and the Italian National Group for Scientific Computation (GNCS-INdAM).

References

Aakash; and Saha, I. 2024. Optimal Makespan in a Minute Timespan! A Scalable Multi-Robot Goal Assignment Algorithm for Minimizing Mission Time. In Wooldridge, M. J.; Dy, J. G.; and Natarajan, S., eds., *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024*, 10280–10287. AAAI Press.

Banfi, J.; Basilico, N.; and Amigoni, F. 2017. Intractability of Time-Optimal Multirobot Path Planning on 2D Grid Graphs with Holes. *IEEE Robotics Autom. Lett.*, 2(4): 1941–1947.

Burkard, R. E.; Dell'Amico, M.; and Martello, S. 2009. *Assignment Problems*. SIAM. ISBN 978-0-89871-663-4.

Fine, G.; Atzmon, D.; and Agmon, N. 2023. Anonymous Multi-Agent Path Finding with Individual Deadlines. In Agmon, N.; An, B.; Ricci, A.; and Yeoh, W., eds., *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023*, 869–877. ACM.

Gross, O. 1959. The bottleneck assignment problem. Technical report, RAND Corporation, Santa Monica, California.

Kloder, S.; and Hutchinson, S. 2006. Path planning for permutation-invariant multirobot formations. *IEEE Trans. Robotics*, 22(4): 650–665.

Kovács, P. 2015. Minimum-cost flow algorithms: an experimental evaluation. *Optim. Methods Softw.*, 30(1): 94–127.

Kuhn, H. W. 2010. The Hungarian Method for the Assignment Problem. In Jünger, M.; Liebling, T. M.; Naddef, D.; Nemhauser, G. L.; Pulleyblank, W. R.; Reinelt, G.; Rinaldi, G.; and Wolsey, L. A., eds., *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, 29–47. Springer.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 442–449. ijcai.org.

Ma, H.; Wagner, G.; Felner, A.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2018. Multi-Agent Path Finding with Deadlines. In Lang, J., ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 417–423. ijcai.org.

McGeoch, C. C. 2012. *A Guide to Experimental Algorithmics*. Cambridge University Press. ISBN 978-0-521-17301-8.

Okumura, K.; and Défago, X. 2023. Solving simultaneous target assignment and path planning efficiently with time-independent execution. *Artif. Intell.*, 321: 103946.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In Surynek, P.; and Yeoh, W., eds., *Proceedings of the Twelfth International Symposium on Combinatorial Search, 2019*, 151–158. AAAI Press.

Surynek, P. 2010. An Optimization Variant of Multi-Robot Path Planning Is Intractable. In Fox, M.; and Poole, D., eds., *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 1261–1263. AAAI Press.

Turpin, M.; Mohta, K.; Michael, N.; and Kumar, V. 2014. Goal assignment and trajectory planning for large teams of interchangeable robots. *Auton. Robots*, 37(4): 401–415.

Wang, H.; and Chen, W. 2022. Multi-Robot Path Planning With Due Times. *IEEE Robotics Autom. Lett.*, 7(2): 4829–4836.

Yu, J. 2016. Intractability of Optimal Multirobot Path Planning on Planar Graphs. *IEEE Robotics Autom. Lett.*, 1(1): 33–40.

Yu, J.; and LaValle, S. M. 2012. Multi-agent Path Planning and Network Flow. In *Algorithmic Foundations of Robotics X - Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics, WAFR 2012*, volume 86 of *Springer Tracts in Advanced Robotics*, 157–173. Springer.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013*, 1443–1449. AAAI Press.