Journal of Artificial Intelligence Research 78 (2023) 357-384 Submitted 10/2022; published 09/2023

Non-Crossing Anonymous MAPF for Tethered Robots

Xiao Peng XIAO.PENG@INSA-LYON.FR
Olivier Simonin OLIVIER.SIMONIN@INSA-LYON.FR
Christine Solnon CHRISTINE.SOLNON@INSA-LYON.FR
CITI, INSA Lyon, Inria, F-69621 Villeurbanne, France

Abstract

This paper deals with the anonymous multi-agent path finding (MAPF) problem for a team of tethered robots. The goal is to find a set of non-crossing paths such that the makespan is minimal. A difficulty comes from the fact that a safety distance must be maintained between two robots when they pass through the same subpath, to avoid collisions and cable entanglements. Hence, robots must be synchronized and waiting times must be added when computing the makespan. We show that bounds can be efficiently computed by solving linear assignment problems. We introduce a variable neighborhood search method to improve upper bounds, and a Constraint Programming model to compute optimal solutions. We experimentally evaluate our approach on three different kinds of instances.

1. Introduction

Multi-Agent Path Finding (MAPF) is a very active research topic which has important applications for robotics in industrial contexts such as transport in fulfillment centers or autonomous tug robots, for example. The goal of MAPF is to find a set of collision-free paths from starting points to target points. Usually, there is an objective function to optimize such as the duration of the longest path (called *makespan*), the sum of all path durations, or the number of agents that cannot reach their targets within a given makespan (Ma, Wagner, Felner, Li, Kumar, & Koenig, 2018). The problem is $\mathcal{NP}$-hard in the general case (Yu & LaValle, 2013a). In the case of *anonymous* MAPF (AMAPF), the target of each agent is not known, *i.e.*, there is a set of targets and each agent must be first assigned to a target before searching for paths (Stern, Sturtevant, Felner, Koenig, Ma, Walker, Li, Atzmon, Cohen, Kumar, et al., 2019). Regardless of whether the goal is to minimize the makespan or the sum of all paths' costs, AMAPF can be generally solved in polynomial time (Yu & LaValle, 2013b).

In some cases, robots are attached to anchor points with flexible cables which allow them to have continuous access to energy, water, or network, for example. These cables may be kept taut by a system that pulls on cables when robots move back. This is the case for our industrial partners in a European project[1] where a fleet of mobile robots is used for inspecting and cleaning large structures, as illustrated in Fig. 1. The main difficulty with such tethered robots comes from the fact that robots are not able to cross cables. It may be possible that robots share a same subpath, provided that their paths do not cross, but in this case we must synchronize robots in order to ensure a safety distance that prevents

1. H2020 project BugWright2: Autonomous Robotic Inspection and Maintenance on Ship Hulls and Storage Tanks, 2020-24 (see `https://www.bugwright2.eu/`)

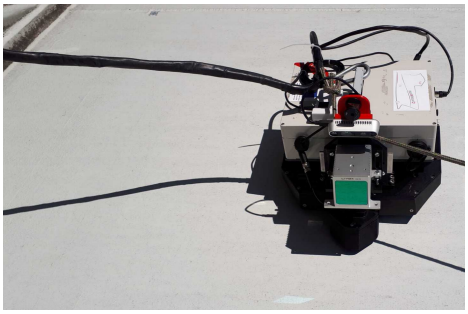

Figure 1: Example of tethered robot used to clean industrial surfaces in H2020 project BugWright2.

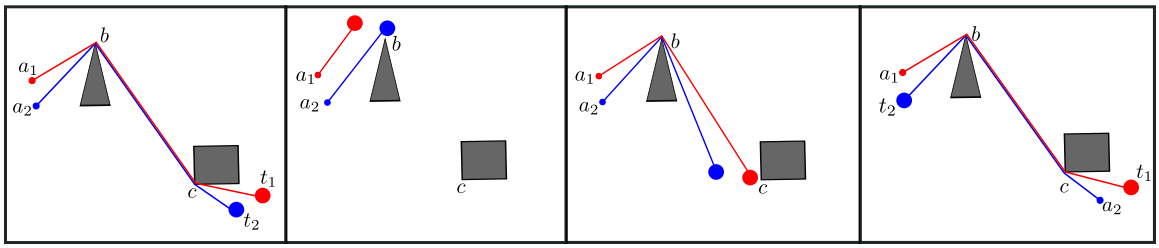

Figure 2: (a): a_1 and a_2 are the anchor points of 2 robots that both have to go to points b and c before reaching their final targets t_1 and t_2 (gray polygons are obstacles). (b): The red robot could arrive first on point b (because it is closer), but it has to wait for the blue robot to pass point b (because the blue robot cannot travel between the red cable and point b if the red robot has already passed point b). (c): Then, the blue robot could arrive first on point c (because it left point b first), but it has to wait for the red robot to pass point c. (d): When the anchor point and the target of the blue robot are exchanged, a deadlock occurs because the red robot must wait for the blue one to pass b first, while the blue one cannot pass c before the red robot.

collisions and entanglements of cables (usually, this safety distance is larger than the size of the robots). As a consequence, robots may have to wait at some points, as illustrated in Fig. 2(a-c), and these waiting times must be taken into account when computing the makespan. Also, deadlocks may occur, as illustrated in Fig. 2(d).

1.1 Related Work

Classical MAPF problems are often solved by using a two-level approach called *Conflict Based Search (CBS)* (Sharon, Stern, Felner, & Sturtevant, 2015): path searching at the low level and resolution of path conflicts at the high level. As a special case, the AMAPF

problem is usually reduced to a Network Flow problem, and the maximum flow algorithms (Ford-Fulkerson, 1956) may be applied to solve it in polynomial time (Yu & LaValle, 2013b). However, both CBS and the Network Flow-based method are limited in handling agents' collisions occurring at an instant or over a short period of time and cannot effectively address the topological constraints associated with cables.

For the motion planning of a single tethered robot, studies mainly focus on finding the shortest path given the initial and final configurations (Salzman & Halperin, 2015), and planning paths for coverage and exploration tasks (Shnaps & Rimon, 2014).

When considering the case of multiple tethered robots, the challenge is that cables can easily get tangled, and avoiding cable crossings makes the planning harder. (Sinden, 1990) introduces the tethered robot problem where robots may have to visit several points in a workspace that does not contain any obstacle: in this case, the problem basically involves searching for matchings in bipartite graphs such that selected edges do not cross. In the work of (Hert & Lumelsky, 1994, 1997, 1999), the workspace does not contain obstacles and robots cannot cross cables but they may push and bent the cables of other robots (which is not the case for our robots). Also, the goal is to minimize the sum of all path lengths whereas our objective is to minimize the makespan. Hert & Lumelsky describe efficient algorithms for detecting whether two paths are crossing or not, from a purely geometric standpoint. The procedure that we use for detecting path crossings (described in Section 2.2) is adapted from this work.

(Zhang & Pham, 2019) consider the same problem as (Hert & Lumelsky, 1994), but the workspace may contain obstacles and there is no objective function to optimize: they simply search for a valid schedule such that robots do not cross cables. To avoid crossings, precedence constraints and waiting times are introduced, and a precedence graph is used to detect deadlocks. Zhang & Pham propose algorithms for iteratively removing deadlocks. However, there is no guarantee that all deadlocks can be suppressed, and robots are constrained to diverge from straight line motions whenever there are non resolved deadlocks.

In (Peng, Solnon, & Simonin, 2021), we introduced the *Non-Crossing AMAPF (NC-AMAPF)*: given a workspace that contains obstacles and a set of anchor points and targets, the goal is to find non-crossing paths from anchor points to targets so that the makespan is minimized. In this preliminary study, we considered bodiless robots so that two robots could share a same subpath without having to include waiting times. With this simplifying assumption, we showed that the NC-AMAPF is related to an Euclidean bipartite matching problem: a lower bound and an upper bound may be computed in polynomial time by solving a *Linear Bottleneck Assignment Problem (LBAP)* and a *Linear Sum Assignment Problem (LSAP)*, respectively. We also introduced an approach based on the sequential combination of LSAP, *Variable Neighborhood Search* (VNS) and *Constraint Programming* (CP) to solve the problem to optimality.

1.2 Contributions and Overview of the Paper

In Section 2, we formally define the NC-AMAPF for tethered robots. The main difference with the NC-AMAPF introduced in (Peng et al., 2021) is that we add precedence constraints between robots that share a same subpath in order to take into account the fact that a safety distance must be maintained between them to avoid collisions and cable entanglements. This

new setting definitely changes the definition of the makespan as we may have to introduce waiting times to satisfy precedence constraints.

In Section 3, we introduce an algorithm that exploits a precedence graph to suppress deadlocks by reassigning targets. We also prove that the optimal solution of LSAP cannot contain deadlocks. This allows us to compute two different upper bounds of the optimal solution: the first one is computed by solving the LSAP; the second one is computed by first solving the LBAP and then using our new algorithm to remove deadlocks, if any.

In Section 4, we introduce a VNS approach to improve the upper bounds computed in Section 3. This algorithm is similar to the VNS approach introduced in (Peng et al., 2021), but it enlarges the neighborhood by taking into account non-shortest paths, as the optimal solution may contain non-shortest paths in order to avoid crossings.

In Section 5, we extend the CP model of (Peng et al., 2021) to include waiting times due to interactions between pairs of robots when computing the makespan. This CP model relaxes constraints due to interactions of more than two robots, and we show how to lazily generate constraints to compute the optimal solution. We also introduce a dichotomous approach to avoid computing useless paths.

In Section 6, we discuss further work.

2. Problem Statement

In Section 2.1, we define notations and definitions (most of them are coming from (Peng et al., 2021)). Then, we show how to decide whether two paths are crossing or not (Section 2.2), how to remove crossings (Section 2.3), how to detect deadlocks (Section 2.4) and how to compute the makespan of a set of paths (Section 2.5). Finally, we define the NC-AMAPF problem for tethered robots (Section 2.6), and we describe the benchmarks used for evaluating our approach (Section 2.7).

2.1 Notations and Definitions

Robots move on a 2 dimensional workspace $\mathcal{W} \subset \mathbb{R}^2$. This workspace is defined by a bounding polygon $\mathcal{B}$ and a set $\mathcal{O}$ of convex obstacles: every obstacle in $\mathcal{O}$ is a polygon within $\mathcal{B}$, and $\mathcal{W}$ is composed of every point in $\mathcal{B}$ that does not belong to an obstacle in $\mathcal{O}$. We denote $\mathcal{V}_\mathcal{O}$ the set of all obstacle vertices.

Given two points $u, v \in \mathcal{W}$, we denote $\overline{uv}$ the straight line segment that joins u to v, and $|uv|$ the Euclidean distance between u and v (*i.e.*, $|uv|$ is the length of $\overline{uv}$). We assume that all robots have the same speed which is equal to 1, so that $|uv|$ is also equal to the time needed by a robot to traverse $\overline{uv}$. We say that a segment $\overline{uv}$ crosses an obstacle if $u, v \in \mathcal{W}$ and $\overline{uv} \not\subset \mathcal{W}$.

Given three points $u, v, w \in \mathcal{W}$, we denote $\angle uvw$ the angle between $\overline{vu}$ and $\overline{vw}$, and we denote $\measuredangle uvw$ the size of this angle measured in degrees when considering a counterclockwise order from $\overline{vu}$ to $\overline{vw}$.

A path in $\mathcal{W}$ is composed of a chain of incident segments $\overline{u_0u_1}, \overline{u_1u_2}, \ldots, \overline{u_{i-1}u_i}$, which is represented by the vertex sequence $\pi = \langle u_0, u_1, u_2, \ldots, u_i \rangle$. The length of a path π is denoted $|\pi|$ and is the sum of the lengths of its segments, *i.e.*, $|\pi| = \sum_{j=1}^{i} |u_{j-1}u_j|$. Given two paths π_i and π_j, we denote $\pi_i.\pi_j$ the path obtained by concatening π_j at the end of π_i. We use set operators to denote vertex membership and path inclusion: $u \in \pi$ denotes the

fact that path π contains vertex u (*i.e.*, $\exists \pi_i, \pi_j, \pi = \pi_i.\langle u \rangle.\pi_j$) and $\pi_i \subseteq \pi$ denotes the fact that path π contains the subpath π_i (*i.e.*, $\exists \pi_j, \pi_k, \pi = \pi_j.\pi_i.\pi_k$). Given a set Π of paths and a vertex u, we denote Π_u the set of all paths of Π that contain u, *i.e.*, $\Pi_u = \{\pi \in \Pi \mid u \in \pi\}$.

As the workspace $\mathcal{W}$ is continuous, there exists an infinite number of paths from an anchor point a to a target t. However, as each cable is kept taut, the number of different cable positions that start from a and end on t is finite (provided that we forbid infinite loops). More precisely, the cable position associated with a robot path from a to t is a chain of incident segments $\langle u_0, u_1, \ldots, u_i \rangle$ such that (i) $u_0 = a$ and $u_i = t$, (ii) no segment crosses an obstacle, and (iii) every internal point is an obstacle vertex, *i.e.*, $\forall j \in [1, i-1], u_j \in \mathcal{V}_\mathcal{O}$.

As the length of a robot path cannot be smaller than the length of its cable position, we can simplify our problem by assuming that the path of a robot is its cable position. Hence, we search for paths in a visibility graph (Lozano-Pérez & Wesley, 1979) defined below.

Definition 1 (Visibility graph (Lozano-Pérez & Wesley, 1979))**.** The visibility graph associated with a workspace $\mathcal{W}$, a set of anchor points $\mathcal{A}$, and a set of targets $\mathcal{T}$ is the directed graph $(\mathcal{V}, \mathcal{E})$ such that vertices are either points of $\mathcal{A}$ and $\mathcal{T}$ or obstacle vertices, *i.e.*, $\mathcal{V} = \mathcal{A} \cup \mathcal{T} \cup \mathcal{V}_\mathcal{O}$, and edges correspond to segments that do not cross obstacles and that do not contain any other vertex, *i.e.*,

$$\mathcal{E} = \{(u, v) \in (\mathcal{A} \cup \mathcal{V}_\mathcal{O}) \times (\mathcal{T} \cup \mathcal{V}_\mathcal{O}) \mid \overline{uv} \subset \mathcal{W} \ \wedge \ \forall w \in \mathcal{V} \setminus \{u, v\}, w \notin \overline{uv}\}$$

The graph is directed because edges starting from targets or ending on anchor points are forbidden.

Given two edges (u, v) and (u', v'), we say that they are incident if they have one common endpoint (*i.e.*, $|\{u, v\} \cap \{u', v'\}| = 1$), and we say that they cross if they share one point (called the crossing point) which is not an endpoint (*i.e.*, $\{u, v\} \cap \{u', v'\} = \emptyset$ and $\overline{uv} \cap \overline{u'v'} \neq \emptyset$).

In (Lozano-Pérez & Wesley, 1979), it is shown that visibility graphs can still be used when robots have a non-negligible size as obstacles may be expanded to compensate for robot sizes. In our problem, the robot size has no significant impact on the geometric properties of the cable. Even though the location of the cable does not perfectly overlap the trajectory along which the robots move, the cable can remain taut and the topological relationship between the cables (crossing or not) is not affected.

A path in the visibility graph $(\mathcal{V}, \mathcal{E})$ is a sequence of vertices $\langle u_0, \ldots, u_i \rangle$ such that $(u_{j-1}, u_j) \in \mathcal{E}, \forall j \in [1, i]$. This path also corresponds to a chain of segments and its length is the sum of the lengths of its segments. We only consider elementary paths, *i.e.*, a vertex cannot occur more than once in a path. Indeed, if a path is not elementary, then it can be replaced by a shorter elementary path obtained by removing its cycles.

Given an anchor point $a \in \mathcal{A}$ and a target $t \in \mathcal{T}$, we denote $sp(a, t)$ the shortest path from a to t in the visibility graph.

2.2 Detecting Crossing Paths

Whenever two paths π_1 and π_2 have a non-empty intersection, we need to decide whether they are crossing or not. The different cases of non-empty intersection paths are illustrated

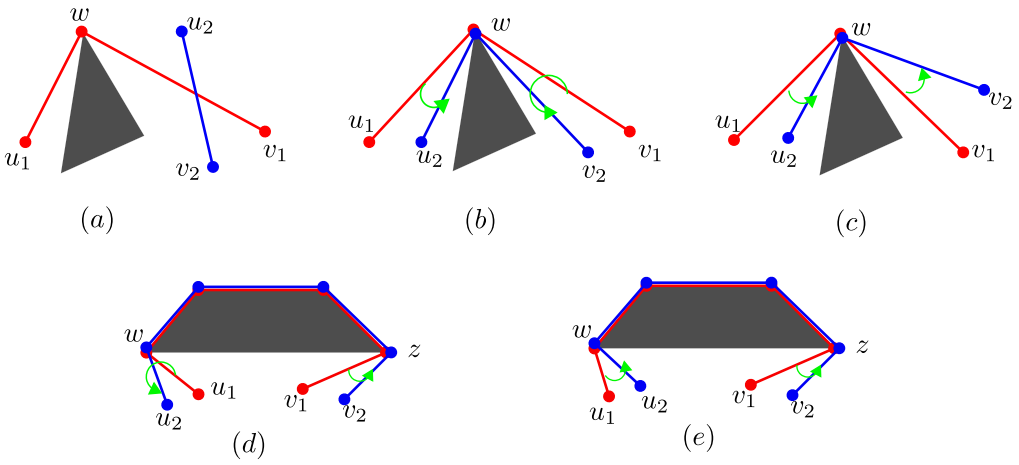

Figure 3: Different cases of two paths with a non empty intersection (angle sizes are represented by green arrows). (a): The paths are trivially crossing because the intersection is a point that does not belong to an obstacle. (b) and (c): the intersection is a single obstacle vertex w. (b) corresponds to non-crossing paths ($\measuredangle u_1wu_2 < 180°$ and $\measuredangle v_1wv_2 > 180°$), whereas (c) corresponds to crossing paths (both angles are lower than 180°). (d) and (e): the intersection is a sequence of segments from w to z. (d) corresponds to non-crossing paths ($\measuredangle u_1wu_2 > 180°$ and $\measuredangle v_1zv_2 < 180°$), whereas (e) corresponds to crossing paths (both angles are lower than 180°).

in Fig. 3. The case in Fig. 3(a) is trivial: when two paths share a same point that does not belong to any obstacle, then they are crossing.

When the intersection corresponds to an obstacle vertex, the two paths may be crossing or not, as illustrated in Fig. 3(b) and 3(c). To decide whether they are crossing or not, we consider the following definition which is an adaptation of (Hert & Lumelsky, 1997) to our context.

Definition 2 (Intersecting paths)**.** Let us consider two paths $\pi_1 = \langle u_1, w, v_1\rangle$ and $\pi_2 = \langle u_2, w, v_2\rangle$ such that $\{u_1, v_1\} \cap \{u_2, v_2\} = \emptyset$ (*i.e.*, the intersection of π_1 and π_2 contains the single vertex w). π_1 and π_2 are crossing if and only if $\measuredangle u_1wu_2$ and $\measuredangle v_1wv_2$ are either both lower than 180° or both greater than 180°.

This may be extended in a straightforward way to the case where the intersection is a sequence of segments π_3 instead of a single vertex, *i.e.*, $\pi_1 = \langle u_1\rangle.\pi_3.\langle v_1\rangle$ and $\pi_2 = \langle u_2\rangle.\pi_3.\langle v_2\rangle$ and $\{u_1, v_1\} \cap \{u_2, v_2\} = \emptyset$. Let w and z be the first and last vertices of π_3, respectively. In this case, the two paths are crossing if and only if $\measuredangle u_1wu_2$ and $\measuredangle v_1zv_2$ are either both lower than 180° or both greater than 180°, as illustrated in Fig. 3(d) and 3(e).

Hence, we can decide whether two paths are crossing or not in $\mathcal{O}(k^2)$ where k is the maximum number of segments in a path. Indeed, we first check in $\mathcal{O}(k^2)$ that there is no crossing segments (there are $\mathcal{O}(k^2)$ pairs of segments and we check if two segments are crossing in constant time). Then, we search for all common subpaths in $\mathcal{O}(k)$, and for each

common subpath we check that there is no crossing by measuring two angles in constant time.

2.3 Removing Crossings

In (Peng et al., 2021), we have shown that whenever two paths π_i and π_j are crossing, we can always replace them by two non crossing paths π'_i and π'_j such that $|\pi_i| + |\pi_j| \geq |\pi'_i| + |\pi'_j|$. The basic idea is to exchange the end of π_i with the end of π_j, starting from the crossing point, and to replace the resulting paths with taut paths whenever they are not taut (see the proof of Theorem 3 in (Peng et al., 2021) for more details).

Given a set of paths with crossings, these pairwise exchanges of path ends may be iterated until all crossings have been removed. The resulting set of non-crossing paths has a total length smaller than or equal to the initial set of crossing paths.

2.4 Detecting Deadlocks

In (Peng et al., 2021), we considered bodiless robots, *i.e.*, we assumed that a robot can always travel between the cable of another robot and an obstacle. With this simplifying assumption, a set of non-crossing paths is always a consistent solution. In this paper, we take into account the fact that a safety distance must be maintained between two robots when they pass through the same sub-path, to avoid collisions and cable entanglements. This is done by constraining robots to pass shared obstacles in a given order, as illustrated in Fig. 2. This order depends on the relative path positions with respect to the obstacle: if π_i is closer to the obstacle than π_j, then the robot associated with π_i must reach the vertex before the robot associated with π_j. For example, in Fig. 2(a), the blue path is closer to the triangle than the red path whereas the red path is closer to the rectangle than the blue path. Given two paths that share a same obstacle vertex, we can decide in constant time which one is the closer to the obstacle, as described in (Zhang & Pham, 2019). This may be extended to the case of two paths that share some sub-path, by ensuring that paths never cross. For example, in Fig. 4, once we have decided that $\pi_1 \prec_d \pi_2$, we can propagate the relative positions of π_1 and π_2 at vertex b, *i.e.*, π_2 is above π_1 and, therefore, $\pi_2 \prec_b \pi_1$. We use this to define total orders among paths that traverse a same obstacle vertex.

Definition 3 (Total order $\prec_u$)**.** Let Π be a set of non-crossing paths, and $u \in \mathcal{V}_\mathcal{O}$ be a vertex of an obstacle $o \in \mathcal{O}$. We denote $\prec_u$ the strict total order on Π_u such that $\forall\{\pi_i, \pi_j\} \subseteq \Pi_u, \pi_i \prec_u \pi_j$ if and only if π_i is closer to o than π_j (as defined in (Zhang & Pham, 2019)). In this case, the robot associated with π_i must visit u before the robot associated with π_j, and we say that π_i has a higher priority than π_j for vertex u.

These total orders are used to define a precedence graph which is similar to the Pair Interaction Graph of (Zhang & Pham, 2019). This precedence graph models precedence constraints between path steps, where a path step is a couple (u, π_i) that represents the visit of a vertex u by the robot associated with a path π_i. Besides the precedence constraints induced by total orders, this graph also models precedence constraints due to the fact that a robot must visit vertices in the order defined by its path.

Definition 4 (Precedence graph)**.** Let Π be a set of non-crossing paths. The precedence graph associated with Π is the directed graph $G_\Pi = (\mathcal{V}_\Pi, \mathcal{E}_\Pi)$ such that vertices correspond

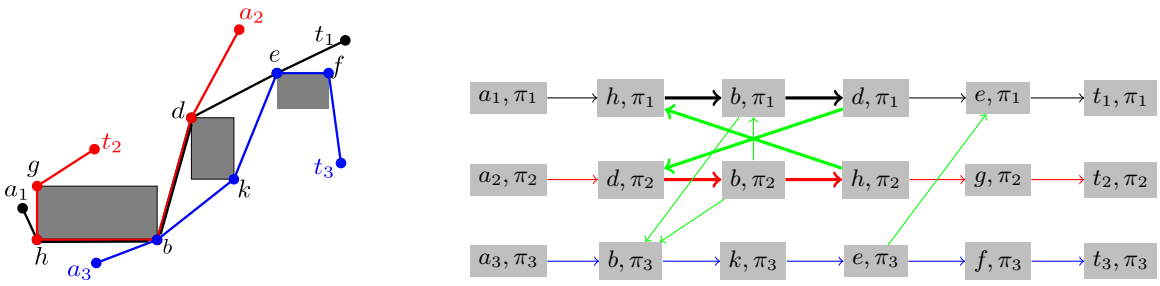

Figure 4: Deadlock example. Left: Three paths $\pi_1 = \langle a_1, h, b, d, e, t_1 \rangle$ in black, $\pi_2 = \langle a_2, d, b, h, g, t_2 \rangle$ in red, and $\pi_3 = \langle a_3, b, k, e, f, t_3 \rangle$ in blue. The robot associated with π_1 should pass h after the robot associated with π_2 because $\pi_2 \prec_h \pi_1$, meanwhile, the robot associated with π_2 cannot pass d until the robot associated with π_1 has passed it because $\pi_1 \prec_d \pi_2$. Right: The corresponding precedence graph (vertical edges are in green whereas horizontal edges have the same color as their corresponding path). This graph contains the cycle $c = \langle (h, \pi_1), (b, \pi_1), (d, \pi_1), (d, \pi_2), (b, \pi_2), (h, \pi_2), (h, \pi_1) \rangle$ (displayed in bold).

to path steps, *i.e.*, $\mathcal{V}_\Pi = \{(u, \pi_i) \mid \pi_i \in \Pi \wedge u \in \pi_i\}$, and edges correspond to precedence constraints between path steps, *i.e.*, $\mathcal{E}_\Pi = \{((u, \pi_i), (v, \pi_i)) \mid \pi_i \in \Pi \wedge \langle u, v \rangle \subseteq \pi_i\} \cup \{((u, \pi_i), (u, \pi_j)) \mid u \in \mathcal{V}_\mathcal{O} \wedge \{\pi_i, \pi_j\} \subseteq \Pi_u \wedge \pi_i \prec_u \pi_j\}$. Edges $((u, \pi_i), (v, \pi_i))$ between two path steps in a same path are called *horizontal edges* whereas edges $((u, \pi_i), (u, \pi_j))$ between two path steps in two different paths are called *vertical edges*.

Two paths that both visit two vertices may have different precedence constraints on these two vertices. For example, let us consider the paths π_1 and π_2 displayed in Fig. 4. Both paths visit vertices h and d. However, $\pi_2 \prec_h \pi_1$ (because π_2 is closer to the left most obstacle than π_1) whereas $\pi_1 \prec_d \pi_2$ (because π_1 is closer to the middle obstacle than π_2). As π_1 must visit h before d whereas π_2 must visit d before h, a deadlock occurs.

More generally, deadlocks occur if and only if the precedence graph contains cycles, as cyclic precedence constraints cannot be satisfied. These deadlocks may be detected in linear time with respect to the size of G_Π by performing a depth first search (Cormen, Leiserson, Rivest, & Stein, 2009).

2.5 Computing the Makespan

We aim at minimizing the makespan, *i.e.*, the arrival time of the latest robot including waiting times due to the fact that robots must pass shared obstacles in a given order. This may be computed by exploiting the precedence graph. To this aim, we define a cost function $c_\Pi : \mathcal{E}_\Pi \rightarrow \mathbb{R}^+$ such that:

- the cost of an horizontal edge associated with a path segment is the time needed to travel through this segment, *i.e.*, $\forall((u, \pi_i), (v, \pi_i)) \in \mathcal{E}_\Pi, c((u, \pi_i), (v, \pi_i)) = |uv|$;
- the cost of a vertical edge associated with a priority at a vertex is the time that a robot should wait to let another robot pass before it, *i.e.*, $\forall((u, \pi_i), (u, \pi_j)) \in$

$\mathcal{E}_\Pi, c((u, \pi_i), (u, \pi_j)) = dt$ where dt is a parameter corresponding to the duration for traveling the safety distance (which depends on the size of robots, among other things).

The makespan is equal to the size of the longest path in G_Π when considering cost function c_Π. It may be computed in linear time with respect to the size of G_Π by topologically sorting vertices of $\mathcal{V}_\Pi$ and then relaxing edges of $\mathcal{E}_\Pi$ according to this topological order (Cormen et al., 2009). We do not define the exact locations of the waiting times as this does not affect the computation of the overall makespan. Obviously, robots must wait before starting to follow shared sub-paths and the choice of waiting locations should be considered when planning robot actions, which is not in the scope of this work.

2.6 Definition of the NC-AMAPF Problem for Tethered Robots

An instance of the NC-AMAPF problem for tethered robots is defined by:

- a workspace $\mathcal{W}$ (as defined in Section 2.1);
- a set $\mathcal{A} \subseteq \mathcal{W}$ of n different anchor points (also corresponding to starting points);
- a set $\mathcal{T} \subseteq \mathcal{W}$ of n different targets such that $\mathcal{A} \cap \mathcal{T} = \emptyset$;
- a positive value $dt \in \mathbb{R}^+$ corresponding to the time a robot must wait to let another robot pass before it at some shared point.

A solution is a couple (m, Π) such that:

- $m : \mathcal{A} \to \mathcal{T}$ is a bijection that assigns a different target to each anchor point;
- Π is a set of n paths such that (i) for each anchor point $a \in \mathcal{A}$, Π contains a path from a to its assigned target $m(a)$; (ii) paths in Π do not cross (as defined in Section 2.2); and (iii) Π contains no deadlock (as defined in Section 2.4).

Given a solution $s = (m, \Pi)$, we denote $\mathit{makespan}(s)$ the makespan (as defined in Section 2.5). An optimal solution is a solution $s = (m, \Pi)$ such that $\mathit{makespan}(s)$ is minimal.

When the workspace $\mathcal{W}$ has no obstacle, our problem is equivalent to the bottleneck matching problem with edge-crossing constraints which has been shown to be $\mathcal{NP}$-hard by (Carlsson, Armbruster, Rahul, & Bellam, 2015). Hence, our problem is also $\mathcal{NP}$-hard in the more general case where $\mathcal{W}$ contains obstacles.

2.7 Description of Benchmarks

To study the sensibility of our algorithms to different configurations, we generate instances according to a random model that has three parameters o, n, and d which are described below. For all instances, the bounding polygon is the square $\mathcal{B} = [0, 200]^2$.

The first parameter o is used to set the number of obstacles. For each value of $o \in \{5, 10, 15, 20\}$, we have randomly generated one set $\mathcal{O}_o$ of o obstacles such that each obstacle is a rectangle[2] whose height and width belong to $[1, 40]$, and such that the distance between

2. We have made experiments with other kinds of obstacles and obtained similar conclusions as with rectangular obstacles.

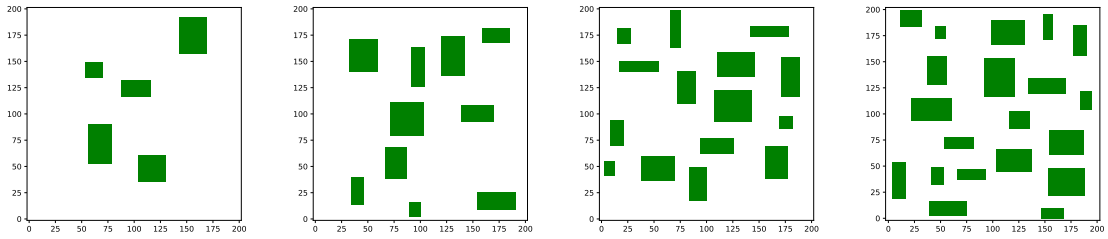

Figure 5: Workspaces $\mathcal{W}_o$ with $o \in \{5, 10, 15, 20\}$ (obstacles are displayed in green).

two obstacle vertices is larger than 10. For each value of o, the workspace is defined by $\mathcal{W}_o = \mathcal{B} \setminus \mathcal{O}_o$. These workspaces are displayed in Fig. 5.

The second parameter n is used to set the number of robots, and we consider values of $n \in \{10, 20, 30\}$ to study scale-up properties.

The third parameter d is used to generate anchor points and targets in $\mathcal{W}_o$, and we consider three different kinds of distributions $d \in \{U, B, A\}$, in order to study the impact of this distribution on bound quality and instance hardness:

- when $d = U$ (Uniform), anchor points and targets are randomly generated in $\mathcal{W}_o$ according to a uniform distribution;
- when $d = B$ (Bipartite), anchor points (resp. targets) are randomly generated on the left (resp. right) part of $\mathcal{W}_o$, by constraining their abscissa to be smaller than 60 (resp. greater than 140).
- when $d = A$ (Alternate) anchor points are randomly generated with their abscissa being equally constrained in $[0, 20] \cup [40, 60] \cup [120, 140]$, and targets are equally distributed in $[80, 100] \cup [140, 160] \cup [180, 200]$.

For the three distributions, we ensure that the distance between two points is always larger than 4 by rejecting any point that does not satisfy this constraint. We believe that setting this minimum distance is suitable considering the sizes of the workspace and the obstacles. Examples of instances are displayed in Fig. 6.

For each value of n, o, and d, we have randomly generated 30 instances (all instances with a same value of o share the same workspace). For all instances, the value of dt is set to 4 (see the conclusion for a discussion on the impact of this parameter on instance hardness).

All experiments reported in this paper are run on Grid5000 (Balouek, Carpen Amarie, Charrier, Desprez, Jeannot, Jeanvoine, Lèbre, Margery, Niclausse, Nussbaum, Richard, Pérez, Quesnel, Rohr, & Sarzyniec, 2013) with an AMD EPYC 7642 with 512GB of RAM.

3. Computation of Bounds for the Makespan

In (Peng et al., 2021), we considered the NC-AMAPF problem with bodiless robots and we showed how to efficiently compute lower and upper bounds for the makespan by solving assignment problems in the complete bipartite graph $G_{\mathcal{A},\mathcal{T}} = (\mathcal{A}, \mathcal{T}, \mathcal{A} \times \mathcal{T})$ such that the cost of an edge $(a, t) \in \mathcal{A} \times \mathcal{T}$ is $|sp(a, t)|$ (*i.e.*, the length of the shortest path from a to t in

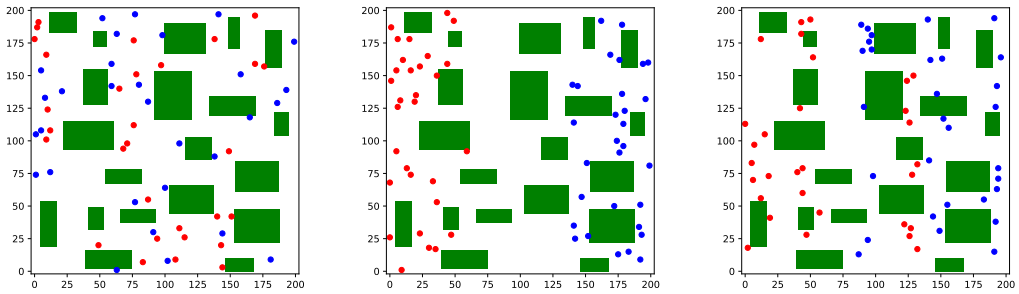

Figure 6: Instance examples with $n=30$, $o=20$, and $d=U$ (left), B (middle), and A (right).

the visibility graph). More precisely, we showed that a lower bound may be computed by solving an LBAP in $G_{\mathcal{A},\mathcal{T}}$, *i.e.*, searching for a matching that minimizes the longest selected edge. Also, we showed that an upper bound may be computed by solving an LSAP in $G_{\mathcal{A},\mathcal{T}}$, *i.e.*, searching for a matching that minimizes the sum of the selected edges. This comes from the fact that, for every pair of selected edges (a_i, t_i) and (a_j, t_j) in an LSAP solution, either $sp(a_i, t_i)$ and $sp(a_j, t_j)$ are not crossing, or they can be replaced by two non-crossing paths π_i and π_j such that $|sp(a_i, t_i)| + |sp(a_j, t_j)| = |\pi_i| + |\pi_j|$.

When taking into account the physical shape of robots, a set of non-crossing paths is not necessarily solution because it may imply deadlocks, as illustrated in both Fig. 2(d) and Fig. 4. In this section, we first introduce an algorithm that removes all deadlocks from a set of paths by iteratively exchanging targets, and we show that this algorithm decreases the path length sum (Section 3.1). Then, we show how to compute a lower bound and two different upper bounds by exploiting LSAP and LBAP solutions (Section 3.2). Finally, we experimentally compare these bounds (Section 3.3).

3.1 Algorithm for Removing Deadlocks

To remove deadlocks, we remove cycles in the precedence graph G_Π by exchanging targets. Given a cycle c in G_Π, let Π_c denote the set of paths involved in c, where a path $\pi_i \in \Pi$ is involved in c if c contains a vertex (u, π_i). Given a path $\pi_i \in \Pi_c$, let $in_c(\pi_i)$ (resp. $out_c(\pi_i)$) denote the set of vertices of π_i that have an incoming (resp. outgoing) vertical edge in c, *i.e.*,

$$\begin{aligned} in_c(\pi_i) &= \{u \in \pi_i : \exists \pi_j \in \Pi_c \setminus \{\pi_i\}, \langle (u, \pi_j), (u, \pi_i) \rangle \subseteq c\} \\ out_c(\pi_i) &= \{u \in \pi_i : \exists \pi_j \in \Pi_c \setminus \{\pi_i\}, \langle (u, \pi_i), (u, \pi_j) \rangle \subseteq c\} \end{aligned}$$

For example, let us consider the precedence graph G_Π and the cycle c displayed in Fig. 4: $in_c(\pi_1) = \{h\}$ and $out_c(\pi_1) = \{d\}$.

Property 1. Let Π be a set of paths and c be an elementary cycle in G_Π. For each path $\pi_i \in \Pi_c$, the number of vertical edges of c that end on a vertex of π_i is equal to the number of vertical edges of c that start from a vertex of π_i, *i.e.*, $\#in_c(\pi_i) = \#out_c(\pi_i)$.

Algorithm 1: RemoveCycle(Π, c)

Input: A set Π of non-crossing paths and an elementary cycle c in G_Π such that, for each path π_i involved in c, $\#in_c(\pi_i) = \#out_c(\pi_i) = 1$

Output: A set of non-crossing paths Π' with the same sets of anchor and target points as Π and such that $G_{\Pi'}$ no longer contains cycle c

1 $\Pi' \leftarrow \Pi \setminus \Pi_c$
2 **for** *each path* $\pi_i \in \Pi_c$ **do**
3 Let $((u, \pi_j), (u, \pi_i)) \in c$ be the incoming vertical edge of π_i
4 Let π_{i1} and π_{i2} be the prefix and suffix of π_i such that $\pi_i = \pi_{i1}.\langle u \rangle.\pi_{i2}$
5 Let π_{j1} and π_{j2} be the prefix and suffix of π_j such that $\pi_j = \pi_{j1}.\langle u \rangle.\pi_{j2}$
6 Let $\pi'_i = \pi_{i1}.\langle u \rangle.\pi_{j2}$
7 **if** π'_i *is not a taut path* **then**
8 Replace π'_i with the shortest path from the first vertex of π'_i to the last vertex of π'_i and in the same homotopy class as π'_i
9 Add π'_i to Π'
10 Remove crossings from Π' as explained in Section 2.3

Proof. This is a straightforward consequence of the fact that (i) c is a cycle and (ii) every horizontal edge connects two vertices in a same path. Indeed, each time the cycle reaches a path $\pi_i \in \Pi_c$, using a vertical edge that ends on a vertex of $in_c(\pi_i)$, it must use a vertical edge that starts from a vertex of $out_c(\pi_i)$ to leave π_i. □

Property 1 ensures that, whenever $\#in_c(\pi_i) = \#out_c(\pi_i) = 1$ for each path $\pi_i \in \Pi_c$, there are exactly $\#\Pi_c$ vertical edges, and these vertical edges define a permutation on the paths of Π_c. In this case, we can remove cycle c by replacing the target of each path π_i with the target of the previous path π_j in the permutation, as described in Algorithm 1. More precisely, the new path π'_i is composed of the prefix of π_i that ends on the vertex u such that $in_c(\pi_i) = \{u\}$, and the suffix of π_j that starts from u (lines 3-6). This new path is valid as it is composed of two valid sub-paths that share a same vertex u. However, it may not be taut and, in this case, we have to replace it by its corresponding taut path (lines 7-8), which is the shortest path in the same homotopy class as defined in (Bhattacharya, Likhachev, & Kumar, 2012). It may be possible that the new taut paths contain crossings and, in this case, we use the procedure described in Section 2.3 to remove all crossings (line 10).

For example, let us consider the precedence graph G_Π displayed in Fig. 4. This graph contains two different elementary cycles.

- Let us suppose that RemoveCycle is called with $c = \langle (h, \pi_1), (b, \pi_1), (d, \pi_1), (d, \pi_2), (b, \pi_2), (h, \pi_2), (h, \pi_1) \rangle$. The vertical edge of c that reaches π_1 is $((h, \pi_2), (h, \pi_1))$. Hence, the new path π'_1 is composed of the prefix of π_1 that ends on h and the suffix of π_2 that starts from h, *i.e.*, $\pi'_1 = \langle a_1, h, g, t_2 \rangle$. The vertical edge of c that reaches π_2 is $((d, \pi_1), (d, \pi_2))$. Hence, the new path π'_2 is composed of the prefix of π_2 that ends on d and the suffix of π_1 that starts from d, *i.e.*, $\pi'_2 = \langle a_2, d, e, t_1 \rangle$. These two paths are not taut, and their associated taut paths are $\langle a_1, g, t_2 \rangle$ and $\langle a_2, t_1 \rangle$.

Algorithm 2: BuildSolution(m, Π)

Input: A bijection $m : \mathcal{A} \to \mathcal{T}$ and a set Π of n paths such that, $\forall a \in \mathcal{A}$, Π contains a path from a to $m(a)$

Output: A solution to the NC-AMAPF problem

1 Remove crossings from Π as explained Section 2.3
2 **while** G_Π *contains cycles* **do**
3 Search for an elementary cycle c in G_Π
4 **while** *there exists a path* π_i *involved in* c *such that* $\#in_c(\pi_i) > 1$ **do**
5 Let $\pi_{i,1}$ be the longest prefix of π_i such that $\forall w \in \pi_{i,1}, w \notin in_c(\pi_i)$
6 Let $\pi_{i,2}$ be the longest suffix of π_i such that $\forall w \in \pi_{i,2}, w \notin out_c(\pi_i)$
7 Let $\pi_{i,3} = \langle u_1, \ldots, u_k \rangle$ be the sub-path of π_i such that $\pi_i = \pi_{i,1}.\pi_{i,3}.\pi_{i,2}$
8 Let c' be the sub-path of c that starts on (u_1, π_i) and ends on (u_k, π_i)
9 Replace c' with $\langle (u_1, \pi_i), (u_2, \pi_i), \ldots, (u_k, \pi_i) \rangle$
  /* For each path π_i involved in c, we have $\#in_c(\pi_i) = \#out_c(\pi_i) = 1$ */
10 $\Pi \leftarrow$ RemoveCycle(Π, c)
11 Update m with respect to Π and return (m, Π)

- Let us suppose that RemoveCycle is called with $c = \langle (b, \pi_1), (d, \pi_1), (d, \pi_2), (b, \pi_2), (b, \pi_1) \rangle$. The vertical edge of c that reaches π_1 is $((b, \pi_2), (b, \pi_1))$. Hence, the new path π'_1 is $\langle a_1, h, b, h, g, t_2 \rangle$. The vertical edge of c that reaches π_2 is $((d, \pi_1), (d, \pi_2))$. Hence, the new path π'_2 is $\langle a_2, d, e, t_1 \rangle$. These two paths are not taut, and their associated taut paths are $\langle a_1, g, t_2 \rangle$ and $\langle a_2, t_1 \rangle$.

It may be possible that new deadlocks are introduced by Algorithm 1, either when paths are replaced with taut paths, or when crossings are removed. In this case, we have to call again Algorithm 1 in order to remove them. The following property ensures that such an iterative process eventually stops.

Property 2. Let $\Pi' =$ RemoveCycle(Π, c). We have: $\sum_{\pi_i \in \Pi} |\pi_i| > \sum_{\pi'_i \in \Pi'} |\pi'_i|$

Proof. Let us first note that the cycle c necessarily contains at least two horizontal edges because vertical edges correspond to total order relations that cannot contain cycles.

Now, let us consider Algorithm 1 without lines 7-8 (*i.e.*, non taut paths are not replaced with taut paths). In this case, every horizontal edge of c no longer appears in the new paths and we have $\sum_{\pi_i \in \Pi} |\pi_i| = \sum_{\pi'_i \in \Pi'} |\pi'_i| + \sum_{\langle (u,\pi_i),(v,\pi_i) \rangle \subset c} |uv|$. As c contains at least two horizontal edges, we have $\sum_{\pi_i \in \Pi} |\pi_i| > \sum_{\pi'_i \in \Pi'} |\pi'_i|$.

Finally, let us consider Algorithm 1 with lines 7-8. Replacing a non taut path with a taut path can only reduce the length of the path because a taut path is the shortest path within the same homotopy class. This may introduce some crossings, but we have shown in (Peng et al., 2021) that path length sums can only be reduced when removing crossings. Hence, we still have $\sum_{\pi_i \in \Pi} |\pi_i| > \sum_{\pi'_i \in \Pi'} |\pi'_i|$. □

Algorithm 2 exploits Algorithm 1 to build a valid solution given a set of paths that may contain crossing paths or deadlocks. It first removes crossing paths using the approach described in Section 2.3 (line 1). Then, while the precedence graph G_Π contains cycles,

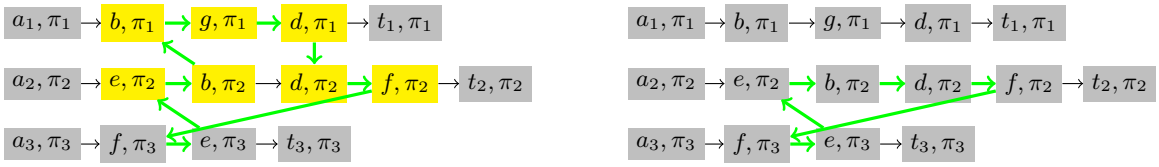

Figure 7: Illustration of Algorithm 2. Left: A precedence graph with an elementary cycle c displayed in green. $in_c(\pi_2) = \{e, d\}$, $out_c(\pi_2) = \{b, f\}$, $\pi_{2,3} = \langle e, b, d, f\rangle$. Right: The precedence graph obtained after replacing the sub-path c' from (e, π_2) to (f, π_2) (displayed in yellow on the left) with $\langle(e, \pi_2), (b, \pi_2), (d, \pi_2), (f, \pi_2)\rangle$.

it searches for an elementary cycle c (with a simple depth-first-search, for example) and iteratively simplifies c until $\#in_c(\pi_i) = \#out_c(\pi_i) = 1$ for every path π_i involved in c (lines 4-9). To simplify c, we search for a path π_i such that $\#in_c(\pi_i) = \#out_c(\pi_i) > 1$, and we shortcut c by replacing its sub-path that goes from the leftmost vertex $u_1 \in in_c(\pi_i)$ to the rightmost vertex $u_k \in out_c(\pi_i)$ with the sub-path of π_i that joins u_1 to u_k. This removes at least one vertex from both $in_c(\pi_i)$ and $out_c(\pi_i)$. An example is depicted in Fig. 7. In some cases, the number of paths involved in c is also decreased, but this number cannot become smaller than two as c still contains at least two vertical edges (one that ends on u_1 and one that starts from u_k) and a vertical edge necessarily involves two different paths. Hence, the loop lines 4-9 is ensured to reach a cycle c such that $\#in_c(\pi_i) = \#out_c(\pi_i) = 1$ for every path $\pi_i \in \Pi_c$ in at most $v/2 - 2$ iterations, where $v = \sum_{\pi_i \in \Pi_c} \#in_c(\pi_i)$ is the initial number of vertical edges in c. Finally, the cycle is removed by using Algorithm 1 (line 10). Property 2 ensures that the loop lines 2-9 is performed a finite number of times: As each cycle removal decreases the total path length, the process necessarily stops with a deadlock free situation.

3.2 Computation of Makespan Bounds from LSAP and LBAP Solutions

Let us now show how to compute lower and upper bounds for our NC-AMAPF problem by solving assignment problems in $G_{\mathcal{A},\mathcal{T}}$. We have shown in (Peng et al., 2021) that we can remove all crossings from the optimal solution of LSAP without changing the sum of all selected path lengths. Let $m_{LSAP} : \mathcal{A} \to \mathcal{T}$ be the optimal solution of LSAP without crossings, and Π_{LSAP} be the corresponding set of paths, *i.e.*, $\Pi_{LSAP} = \{sp(a, m_{LSAP}(a)) : a \in \mathcal{A}\}$. Similarly, let m_{LBAP} denote the optimal solution of LBAP and $\Pi_{LBAP} = \{sp(a, m_{LBAP}(a)) : a \in \mathcal{A}\}$.

The following property shows us that $s_{LSAP} = (m_{LSAP}, \Pi_{LSAP})$ may be used to build a solution of the NC-AMAPF problem.

Property 3. The precedence graph $G_{\Pi_{LSAP}}$ contains no cycle.

Proof. Let us suppose that $G_{\Pi_{LSAP}}$ contains cycles. In this case, we could use Algorithm 2 to remove these cycles. However, this would decrease the sum of all path lengths, which is in contradiction with the fact that m_{LSAP} minimizes the sum of all path lengths. □

Hence, s_{LSAP} is a solution of our NC-AMAPF problem and, therefore, $makespan(s_{LSAP})$ is an upper bound.

$s_{LBAP} = (m_{LBAP}, \Pi_{LBAP})$ provides a lower bound to our NC-AMAPF problem and it is the optimal solution whenever paths in Π_{LBAP} do not cross nor do they share vertices. However, s_{LBAP} is not a solution if Π_{LBAP} contains crossings or deadlocks. In this case, we may use Algorithm 2 to remove all crossings and deadlocks (in other words, buildSolution(s_{LBAP}) is a solution) so that $makespan$(buildSolution(s_{LBAP})) is an upper bound.

To summarize relations between bounds, if *opt* denotes the optimal makespan of our NC-AMAPF problem, we have:

$$\begin{array}{rcl} opt & \leq & makespan(s_{LSAP}) \\ \max_{\pi_i \in \Pi_{LBAP}} |\pi_i| \leq opt & \leq & makespan(\text{buildSolution}(s_{LBAP})) \end{array}$$

However, $makespan(s_{LSAP})$ and $makespan$(buildSolution(s_{LBAP})) are not comparable.

3.3 Experimental Evaluation

Algorithms have been implemented in Python.

In Fig. 8, we display the gap between the optimal makespan *opt* and the lower bound $lb_{LBAP} = \max_{\pi_i \in \Pi_{LBAP}} |\pi_i|$, the upper bound $ub_{LSAP} = makespan(s_{LSAP})$, and the upper bound $ub_{LBAP} = makespan$(buildSolution(s_{LBAP})). This gap is computed as a percentage (*i.e.*, we display $100 * \frac{b-opt}{opt}$ for each bound b, where *opt* is computed with the exact methods introduced in Section 5), and on average for 30 instances per combination (n, o, d). lb_{LSAP} is usually rather close to the optimal solution, especially for U instances, and the bound tends to move away from the optimal solution when increasing n, especially for B and A instances. The number of obstacles o does not seem to have a strong influence on the quality of the lower bound.

There is no clear winner between the two upper bounds. When $d = U$, ub_{LBAP} tends to be closer to *opt* than ub_{LSAP}, whereas when $d = B$ the two bounds are very close and, when $d = A$, ub_{LSAP} is better than ub_{LBAP} for 8 (o, n) combinations (among 12). In Fig. 9, we display scatter plots to compare the two upper bounds on a per instance basis when $n = 30$. For U instances, the gap of ub_{LBAP} is equal to 0% for 8 instances whereas it is greater than 125% for two instances, and the gap of ub_{LSAP} is equal to 0% for 9 instances whereas it is equal to 112% for one instance. For B instances, gaps are always smaller than 25%. For both B and A instances, the difference between the two bounds is less important than for U instances.

In Table 1, we display the time needed to compute all shortest paths from anchor points to targets, and the time needed to compute the two upper bounds (when excluding the time needed to compute paths). For U instances, we always spend more time to compute all shortest paths than to compute a bound. Times increase when increasing the number of obstacles o or the number of robots n, but bounds are always computed within a few tenths of a second. For A instances, the time needed to compute ub_{LSAP} is larger than for U instances (*e.g.*, 0.4s instead of 0.02s when $n = 30$ and $o = 20$), and this time is even larger for B instances (*e.g.*, 1.4s when $n = 30$ and $o = 20$). This comes from the fact that the optimal solution of LSAP for U instances nearly never contains crossing paths,

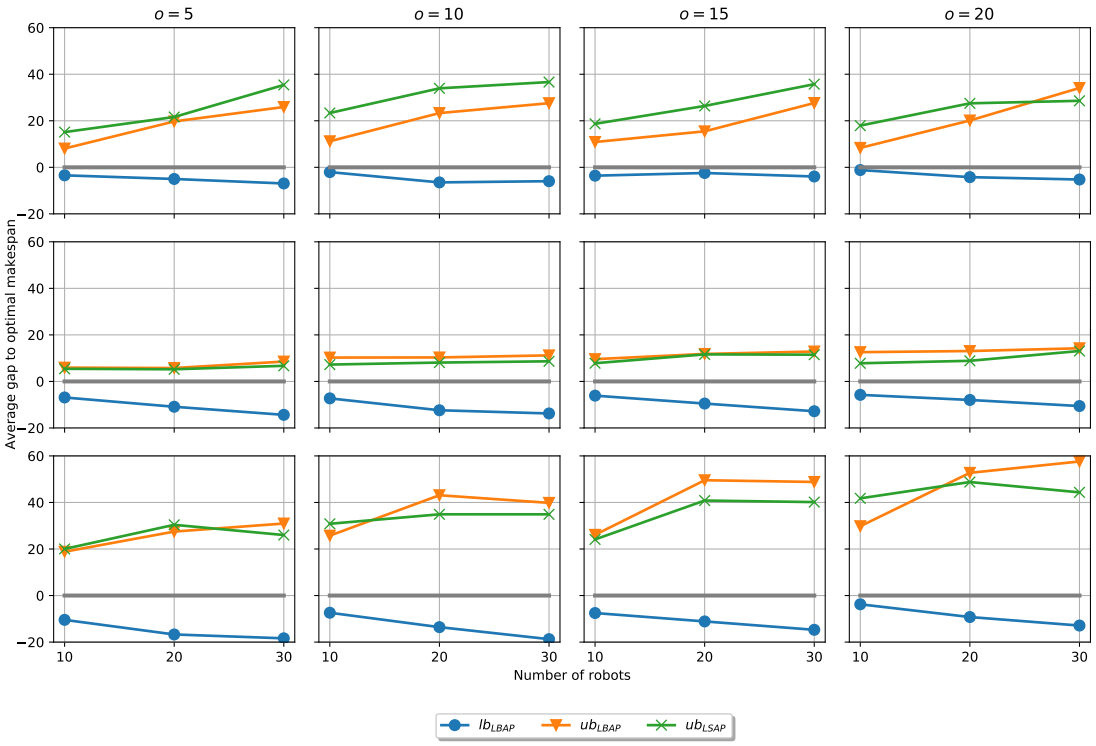

Figure 8: Gap in percentage (y-axis) between the optimal makespan and lb_{LBAP}, ub_{LSAP}, and ub_{LBAP} when $n \in \{10, 20, 30\}$ (x-axis), $o \in \{5, 10, 15, 20\}$ (from left to right), and $d = U$ (Top), B (Middle), or A (Bottom). Every point corresponds to an average value over 30 instances.

whereas it more often contains crossing paths for A and B instances. For example, when $n = 30$ and $o = 20$, the average number of crossing paths is equal to 0.6 (resp. 3.2 and 10) for U (resp. A and B) instances (and replacing these crossing paths by non-crossing paths is time-consuming). This conclusion also holds for ub_{LBAP}, as we can observe that the computation time also increases with m and o for all types of instances. However, compared to ub_{LSAP}, ub_{LBAP} takes more time, notably, when $n = 30$, this difference can go up to several seconds for A and B instances. We know that the optimal solution of LBAP is computed without considering the non-crossing constraints, so there are more crossings to remove. When $n = 30$ and $o = 20$, this average number is equal to 11.3 (resp. 44.2 and 58.1) for U (resp. A and B) instances. We do not report times needed to solve LBAP (and compute lb_{LBAP}): this time is always smaller than 0.03s and is negligible with respect to other times.

In conclusion, ub_{LSAP} is more efficiently computed than ub_{LBAP}, while the two bounds are rather comparable in quality. Therefore, we use the result of ub_{LSAP} as an initial upper bound.

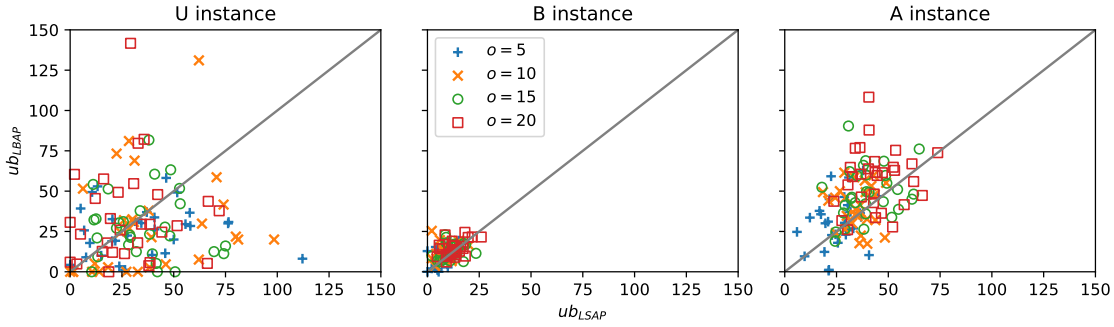

Figure 9: Comparison of ub_{LSAP}, and ub_{LBAP}: Each point (x, y) corresponds to an instance with $n = 30$ such that $x = 100 * \frac{ub_{LSAP} - opt}{opt}$ and $y = 100 * \frac{ub_{LBAP} - opt}{opt}$. The color of the point depends on o. U (resp. B and A) instances are displayed on the left (resp. middle, right).

Table 1: Time (in seconds) needed to compute all shortest paths (t_{path}), and the two bounds ub_{LSAP} and ub_{LBAP} (when excluding the time needed to compute shortest paths), on average over the 30 instances per combination (n, o, d).

		n=10				n=20				n=30			
		o=5	o=10	o=15	o=20	o=5	o=10	o=15	o=20	o=5	o=10	o=15	o=20
d=U	t_{path}	0.03	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.3	0.3	0.4
	ub_{LSAP}	0.0	0.0	0.01	0.01	0.0	0.01	0.01	0.01	0.01	0.01	0.01	0.02
	ub_{LBAP}	0.02	0.03	0.1	0.1	0.05	0.1	0.1	0.2	0.1	0.2	0.3	0.3
d=B	t_{path}	0.03	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.3	0.3	0.4
	ub_{LSAP}	0.01	0.04	0.1	0.2	0.04	0.2	0.5	0.7	0.1	0.4	1.0	1.4
	ub_{LBAP}	0.1	0.3	0.7	1.3	0.3	0.8	2.4	4.3	0.7	1.8	4.9	8.3
d=A	t_{path}	0.03	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.3	0.3	0.4
	ub_{LSAP}	0.0	0.0	0.03	0.02	0.01	0.05	0.1	0.1	0.04	0.1	0.3	0.4
	ub_{LBAP}	0.04	0.1	0.6	0.3	0.2	0.4	1.0	1.4	0.4	1.0	2.4	3.8

4. Improving the Upper Bound with VNS

We have introduced in (Peng et al., 2021) a VNS approach for improving the solution s_{LSAP} that minimizes the sum of costs: the neighborhood of a solution (m, Π) contains every solution (m', Π') such that m' is a matching obtained by permuting the targets of k anchor points in m, and $\Pi' = \{sp(a, m'(a)) : a \in \mathcal{A}\}$ does not contain crossing paths; k is initialized to 2 and it is incremented each time the current solution is locally optimal; k is reset to 2 each time an improving move has been found; the search is stopped when k exceeds a given upper bound k_{max} or when a time limit is reached.

Algorithm 3: newVNS(m, Π, k_{max})

Input: an initial solution (m, Π), and a parameter $k_{max} \in \mathbb{N}$
Output: an improved solution (m, Π)

1 $k \leftarrow 2$
2 **while** $k \leq k_{max}$ **do**
3 let π_{max} be the path of Π with the latest arrival time and a_{max} its anchor point
4 $\mathcal{C} \leftarrow \{\pi \in \Pi : \pi$ is in the same connected component as π_{max} in $G_\Pi\}$
5 **if** $\#\mathcal{C} < k$ **then**
6 add to C the $k - \#\mathcal{C}$ paths whose anchors are the closest to a_{max}
7 **for** *each* $\mathcal{S} \subseteq \mathcal{C}$ *such that* $\#\mathcal{S} = k$ *and* $\pi_{max} \in \mathcal{S}$ **do**
8 let $\mathcal{A}_\mathcal{S}$ be the set of anchor points of paths in $\mathcal{S}$
9 **for** *each permutation* $\sigma : \mathcal{A}_\mathcal{S} \rightarrow \mathcal{A}_\mathcal{S}$ *such that* $\forall a \in \mathcal{A}_\mathcal{S}, |sp(a, m(\sigma(a)))| < makespan(m, \Pi)$ **do**
10 **for** *each* $a \in \mathcal{A}_\mathcal{S}$ **do**
11 compute the set Π_a of every taut path π from a to $m(\sigma(a))$ such that $|\pi| < makespan(m, \Pi)$, and π does not cross any path in $\Pi \setminus \mathcal{S}$
12 **for** *each set* $\mathcal{S}'$ *which contains exactly one path of* Π_a, $\forall a \in \mathcal{A}_\mathcal{S}$ **do**
13 **if** $\Pi' = (\Pi \setminus \mathcal{S}) \cup \mathcal{S}'$ *is valid and improving* **then**
14 replace Π with Π' and update m consequently
15 set k to 2, and go to line 2
16 increment k
17 return (m, Π)

To adapt this VNS to the case where robots have physical shapes, we simply have to forbid deadlocks and to modify the computation of the makespan by integrating waiting times in case of shared vertices, as defined in Sections 2.4 and 2.5. This VNS approach is denoted oldVNS.

In (Peng et al., 2021), we showed that optimal solutions may contain non-shortest paths. For some instances (in particular those generated with $d \neq U$), optimal solutions widely use non shortest paths and have much shorter makespans than solutions computed with shortest paths only (as done by oldVNS). For this reason, we introduce a new VNS approach, denoted newVNS, which takes into account non-shortest paths as described in Algorithm 3.

newVNS starts the search from an initial solution (m, Π) (which may be s_{LSAP} or BuildSolution(s_{LBAP}), for example). At each iteration of the loop lines 2-16, it explores the neighborhood of the current solution (m, Π). This neighborhood contains couples (m', Π') such that Π' is obtained from Π by replacing a set $\mathcal{S}$ of k paths with a set $\mathcal{S}'$ of k new paths (which are not necessarily shortest paths). Obviously, $\mathcal{S}$ must contain the path π_{max} with the latest arrival time (corresponding to the makespan), as this is a mandatory condition to reduce the makespan. The search for $\mathcal{S}'$ is done in four steps:

Step 1 (lines 4-6): As it is time-consuming to compute new paths, the neighborhood is deliberately reduced by constraining paths of $\mathcal{S}$ to belong to a limited set of candidate paths $\mathcal{C}$ which contains all paths in the same connected component as π_{max} in G_Π. If there are less than k paths in $\mathcal{C}$, it is completed by selecting the paths whose anchor points are the closest to the anchor point of π_{max}.

Step 2 (lines 7-9): For each subset $\mathcal{S}$ of k paths in $\mathcal{C}$ (including π_{max}), we enumerate every permutation σ of the k anchor points of $\mathcal{S}$ such that the length of the shortest path from any of these k anchor points a to $m(\sigma(a))$ is smaller than the current makespan (otherwise the permutation cannot lead to a shorter makespan).

Step 3 (lines 10-11): For every anchor point a involved in $\mathcal{S}$, we compute the set Π_a of all paths from a to the new target $m(\sigma(a))$ associated with a, while limiting the search to taut paths that do not cross paths of $\Pi \setminus \mathcal{S}$ and that have a length smaller than the current makespan.

Step 4 (lines 12-15): We search for a new set $\mathcal{S}'$ of k paths in Π_a such that the k new paths are non-crossing and the makespan of $(\Pi \setminus \mathcal{S}) \cup \mathcal{S}'$ is smaller than the makespan of Π. If such an improving set of paths is found, the current solution is updated, and a new improving move is searched with $k = 2$.

If no improving neighbor is found in the loop lines 7-15, then k is increased to enlarge the neighborhood.

4.1 Experimental Evaluation

Algorithms have been implemented in Python[3].

In Fig. 10, we display the evolution of the gap to optimality (in percentage) of bounds computed by oldVNS and newVNS when $k_{max} \in \{1, 3, 5, 7\}$ and when the initial solution is s_{LSAP}. oldVNS and newVNS both return s_{LSAP} when $k_{max} = 1$ as k is initialized to 2 and the search is stopped whenever $k > k_{max}$. Increasing k_{max} decreases the makespan, but we observe larger improvements from 1 to 3 than for 3 to 5 and then to 7. We have made experiments with values of k_{max} larger than 7 and observed that this does not allow us to significantly improve the makespan.

For U instances, we obtain a better upper bound with oldVNS which only considers the shortest paths, while newVNS works better for B and A instances. This difference is due to two reasons. First, compared to oldVNS, newVNS takes into account non-shortest paths. For A and B instances, we more often need to use non shortest paths to improve the solution than for U instances. Second, for the neighborhood construction, in newVNS, we restrained the number of anchor points, or the location ranges where they can be, as well as the location ranges of targets are fixed. For B instances, the anchor points and targets are located in a bipartite way, so the optimal solution could also follow a symmetric match along the Y axis. This means that our new neighborhood is smaller but more effective than the previous one. A instances also follow a bit of bipartite symmetry, so it works too. For U instances, as anchors and targets are uniformly distributed, exchanging with nearby points

3. Our implementation is publicly available at `https://gitlab.inria.fr/xipeng/tethered-amapf-jair2023.git`.

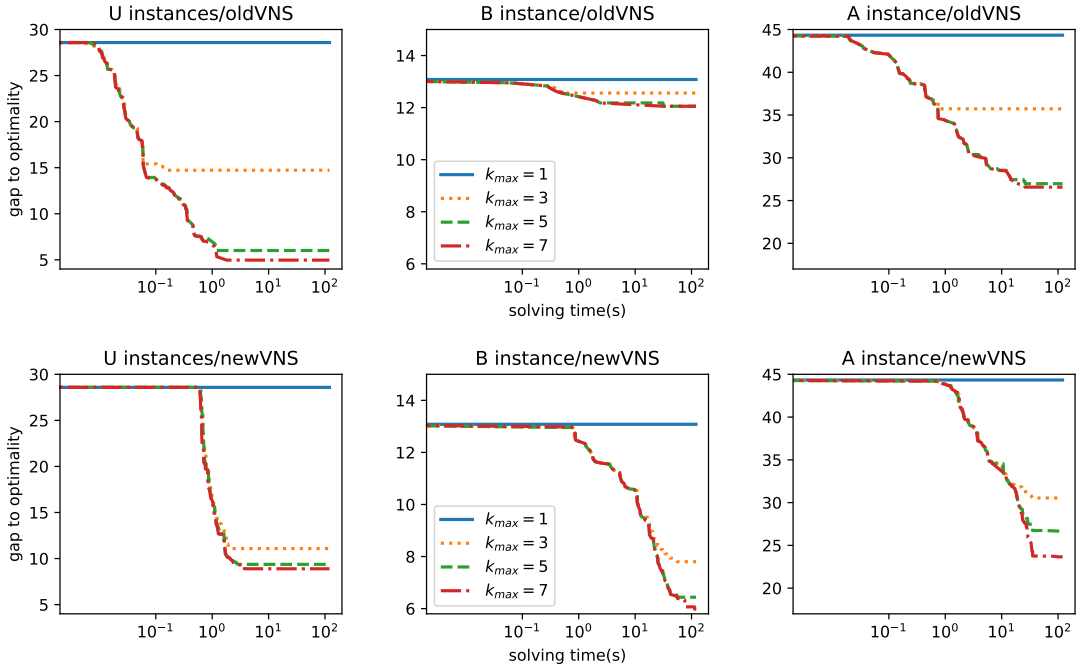

Figure 10: Evolution of the gap to optimality in percentage with respect to time (in seconds) for oldVNS (top row) and newVNS (bottom row) when $k_{max} \in \{1, 3, 5, 7\}$: average value over 30 instances when $n = 30, o = 20$, and $d = U$ (left), B (middle), and A (right). Scales on y-axis are different for U, B, and A instances (from the left to the right), whereas they are identical for oldVNS and newVNS.

can easily violate the non-crossing constraint and it becomes harder to get a better solution: in this case, performing an exhaustive search among all anchor points (as oldVNS does) is more effective.

Since oldVNS and newVNS show complementary performance, we combine them by running oldVNS first then followed by newVNS (each step is limited to 60s) in order to improve the robustness when tackling different instances, and we call this combined method combinedVNS. The switching time of 60 seconds is chosen to find a compromise between the quality of the solution and the time required for resolution. From Fig. 10, we can observe that when oldVNS and newVNS are run individually, the optimality gap curves for each tend to converge around 60 seconds. We have tested other combination methods, and this sequential combination performs better in terms of robustness and efficiency.

In Fig. 11, we show the evolution of the gap to optimality (in percentage) of bounds computed by oldVNS, newVNS, and combinedVNS when $k_{max} = 7$. For U instances, we see that running newVNS after oldVNS slightly improves the bound. For B instances, the curve drops slowly in the first phase (which corresponds to oldVNS), until the 60s time limit is reached, then it continues to improve the bound, and even reaches a level lower than

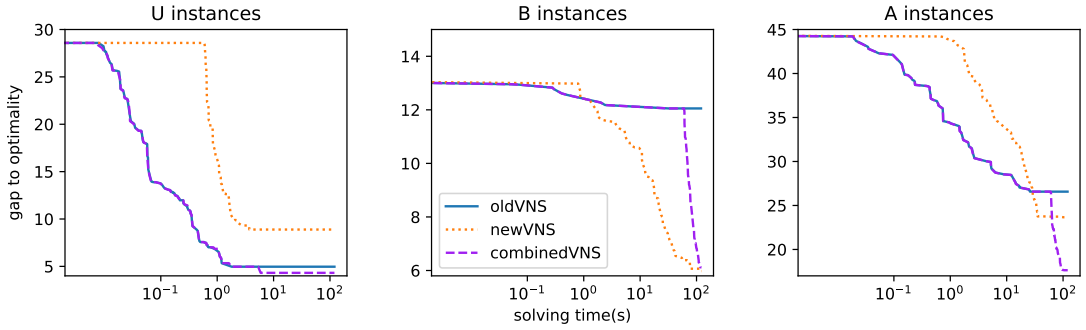

Figure 11: Evolution of the gap to optimality in percentage with respect to time (in seconds) for combinedVNS (which runs oldVNS for 60s and then newVNS for 60s) when $k_{max} = 7$: average value over 30 instances when $n = 30, o = 20, d = U$ (left), B (middle), and A (right).

the simple newVNS does. This conclusion also holds for A instances, and we can see that combinedVNS has a clear advantage over oldVNS and newVNS.

5. Computation of the Optimal Solution with CP

In this section, we first introduce a relaxed CP model, where interactions between more than two paths are ignored. Then, we show how to lazily generate constraints due to interactions between more than two paths in order to compute the optimal solution, and we introduce a dichotomous approach to reduce the number of candidate paths that must be pre-computed. Finally, we report experimental results.

5.1 Relaxed CP Model

As pointed out in (Peng et al., 2021), the optimal solution may use non-shortest paths. Hence, to find the optimal solution we must compute all relevant paths that may belong to the optimal solution. These paths must be taut, elementary, and non self-crossing. As there is an exponential number of paths with respect to the number of obstacle vertices, we add a limit l to the length of the paths. Obviously, we can set l to the best known upper bound, as the makespan cannot be smaller than the length of the longest selected path. Let ub denote this best known upper bound, computed with approaches described in the previous sections. By default, we assume that $l = ub$ (we shall describe in Section 5.3 a dichotomous approach where l is set to values smaller than ub). Given a length bound l, Π_l denotes the set of all taut, elementary, and non self-crossing paths of length smaller than l (see (Peng et al., 2021) for more details on how to compute this set). Paths in Π_l are numbered from 1 to $\#\Pi_l$. For each path $\pi \in \Pi_l$, $a(\pi)$ and $t(\pi)$ denote the anchor point and the target of π, respectively.

Our CP model uses the same variables as in (Peng et al., 2021): for each anchor point $a_i \in \mathcal{A}$, the integer variable T_i represents the target matched with a_i and the integer variable P_i represents the path used to reach T_i from a_i. The initial domains of these variables are

Algorithm 4: lazyApproach(ub, Π_l)

Input: an initial upper bound ub, and a set Π_l of candidate paths
Output: the optimal makespan

```
1 let M be the CP model described in Section 5.1
2 while M has a solution s do
3     if G_{s(P_i):a_i∈A} contains a cycle c then
4         add to M a nogood constraint that forbids c
5     else
6         let π be the longest path in G_{s(P_i):a_i∈A}
7         if |π| < ub then ub ← |π|; add to M the constraint Obj < ub;
8         else add to M a nogood constraint that forbids π;
9 return ub
```

$D(T_i) = \{t(\pi) : \pi \in \Pi_l \wedge a(\pi) = a_i\}$ and $D(P_i) = \{\pi \in \Pi_l : a(\pi) = a_i\}$. Finally, an integer variable Obj represents the makespan and its domain is $D(Obj) = [0, ub[$.

Like in (Peng et al., 2021), we channel T_i and P_i variables by posting $T_i = t(P_i)$, for each anchor point $a_i \in \mathcal{A}$, and we post an *allDifferent*($\{T_i : a_i \in \mathcal{A}\}$) constraint.

We introduce new variables for taking into account waiting times due to vertices shared by two paths: for each pair of anchor points $\{a_i, a_j\} \subseteq \mathcal{A}$, the integer variable $M_{i,j}$ represents the makespan of the two paths P_i and P_j, including the waiting times due to vertices shared by P_i and P_j, *i.e.*, the length of the longest path in the precedence graph $G_{\{P_i,P_j\}}$. The domain of this variable is $D(M_{i,j}) = [0, ub[$.

P_i and $M_{i,j}$ variables are related thanks to table constraints. For each pair of anchor points $\{a_i, a_j\} \subseteq \mathcal{A}$, we pre-compute the table $\mathcal{T}_{i,j}$ that contains every triple $(\pi_i, \pi_j, x) \in D(P_i) \times D(P_j) \times \mathbb{N}$ such that (i) $t(\pi_i) \neq t(\pi_j)$, (ii) path π_i does not cross path π_j, (iii) $G_{\{\pi_i,\pi_j\}}$ has no cycle, (iv) x is the length of the longest path in $G_{\{\pi_i,\pi_j\}}$, and (v) $x < ub$. For each pair of anchor points $\{a_i, a_j\} \subseteq \mathcal{A}$, we post the table constraint $(P_i, P_j, M_{i,j}) \in \mathcal{T}_{ij}$, and we post the constraint $Obj \geq M_{i,j}$. As P_i and P_j are channeled with T_i and T_j, condition (i) ensures that $T_i \neq T_j$ and, therefore, the *allDifferent*($\{T_i : a_i \in \mathcal{A}\}$) constraint is redundant. However, we keep it because it speeds up the solution process.

The solution that minimizes Obj is a lower bound of the optimal makespan because some constraints have been relaxed. Indeed, table constraints ensure that there is no deadlock between two paths and Obj takes into account waiting times due to pairs of paths that have common vertices. However, deadlocks may be due to a circular dependency between more than two paths. Also, in the case of dependency chains of more than two paths, the makespan may be larger than all $M_{i,j}$ variables (*e.g.*, when π_1 and π_2 share a vertex and π_2 and π_3 share another vertex, the actual makespan may be larger than $M_{1,2}$ and $M_{2,3}$). In the next section, we introduce an approach that lazily generate constraints to compute the optimal solution.

5.2 Lazy Constraint Generation

Lazy constraint generation is often used when there are too many constraints. This is the case for our problem, as we should add a constraint for each possible subset of anchor points in order to ensure that the paths associated with these anchor points do not create deadlocks[4]. In Algorithm 4, we describe an approach based on lazy constraint generation. We enumerate all solutions of the model described in Section 5.1. Each time a solution s is found, we compute the precedence graph $G_{\{s(P_i):a_i \in \mathcal{A}\}}$ (where $s(X)$ denotes the value assigned to a variable X in solution s). If s implies a deadlock, *i.e.*, the precedence graph contains a cycle, we search for a cycle c in the precedence graph, we compute the set of anchor points involved in c, and we add a nogood constraint to prevent the search from enumerating again the paths associated by s to these anchor points (lines 3-4). If there is no deadlock, we search for the longest path π in the precedence graph (line 6): the makespan corresponds to the length of this path. If this makespan is smaller than ub, we have found a new improving solution and we constrain Obj to be smaller than this makespan (line 7). Otherwise, we search for all anchor points involved in π and we add a nogood constraint to prevent the search from enumerating again the paths associated by s to these anchor points (line 8).

5.3 Dichotomous Approach

A main issue is related to the initial bound l. This bound limits the length of the paths computed in the set Π_l used to generate the relaxed CP model. The larger l, the more time needed to compute paths in Π_l, the larger the domains of P_i variables and the more triples in $\mathcal{T}_{i,j}$ tables. The upper bound ub computed with combinedVNS is often quite close to the optimal solution (the average gap is smaller than 10% for U and B instances, and smaller than 20% for A instances when $k_{max} = 7$). However, in most cases, ub is much larger than the longest selected path in the optimal solution because waiting times are added. As a consequence, when setting l to ub, a considerable number of paths of Π_l do not contribute to the optimal solution. To avoid this unnecessary computation, we use a dichotomous approach for setting l, as described in Algorithm 5.

lb and ub are the lower and upper bounds of Obj, respectively (lb is initialized to $makespan(s_{LBAP})$ and ub to an upper bound computed by LSAP or combinedVNS). α is a parameter used to control when to switch from dichotomous search to a classical search. While the gap between ub and $(lb + ub)/2$ is larger than $\alpha * ub$, we set l to $(lb + ub)/2$, we compute the set Π_l and launch lazyApproach (lines 2-4). If the bound returned by lazyApproach is smaller than l, then we have found the optimal solution (line 5); otherwise, we increase the lower bound of Obj to l as we know that the optimal solution is greater than or equal to l (line 6). When the gap between ub and $(lb + ub)/2$ becomes smaller than $\alpha * ub$, we stop the dichotomous approach and run lazyApproach with the set Π_{ub} of all paths of length smaller than ub.

4. Another possibility is to design a global constraint for ensuring that the selected set of paths does not contain deadlocks. We have implemented a propagator for this global constraint, that incrementally detects cycles in the precedence graph. However, experimental results showed us that this approach is less efficient than a lazy approach, on most instances.

Algorithm 5: dichotomousApproach(lb, ub, α)

Input: a lower bound lb, an upper bound ub, and a parameter $\alpha \in [0, 1]$
Output: the optimal makespan

```
while (ub − (lb + ub)/2) > α ∗ ub do
    l ← (lb + ub)/2
    Compute the set Π_l of all paths of length smaller than l
    ub ← lazyApproach(ub, l)
    if ub ≤ l then return ub;
    lb ← l
Compute the set Π_ub of all paths of length smaller than ub
return lazyApproach(ub, Π_ub)
```

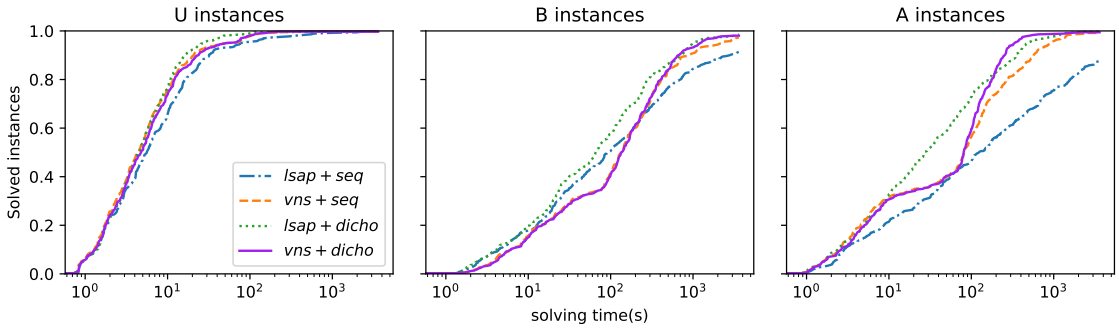

Figure 12: Percentage of solved instances with respect to time when $d = U$ (left), B (middle), and A (right).

5.4 Experimental Results

Algorithm 4 has been implemented in Java, using the Choco CP library (Prud'homme, Fages, & Lorca, 2016). Algorithm 5 has been implemented in Python.

The sequential lazy approach described in Section 5.2 is denoted SEQ and the dichotomous approach described in Section 5.3 is denoted DICHO. For DICHO, the rate α is set to 0.05 (*i.e.*, we switch to SEQ when the gap between ub and $(lb + ub)/2$ is smaller than or equal to 5%). For each approach X $\in$ {SEQ,DICHO}, the bound l is either initialized with $makespan(s_{LSAP})$ (denoted LSAP+X) or with the bound computed by COMBINEDVNS with $k_{max} = 7$ (denoted VNS+X).

In Fig. 12, we display the ratio of solved instances with respect to time for the four approaches X+Y with X $\in$ {SEQ,DICHO} and Y $\in$ {LSAP,VNS} (CPU times include the time for computing the initial bound l with Y). For each instance type $d \in \{U, B, A\}$, there are 360 instances (30 instances per value of $o \in [5, 10, 15, 20]$ and $n \in [10, 20, 30]$). DICHO is more successful than SEQ. This is more particularly sensible when the upper bound is computed with LSAP as, in this case, the initial upper bound is much greater. When the upper bound is computed with VNS, VNS+SEQ and VNS+DICHO obtain very close results for U and B

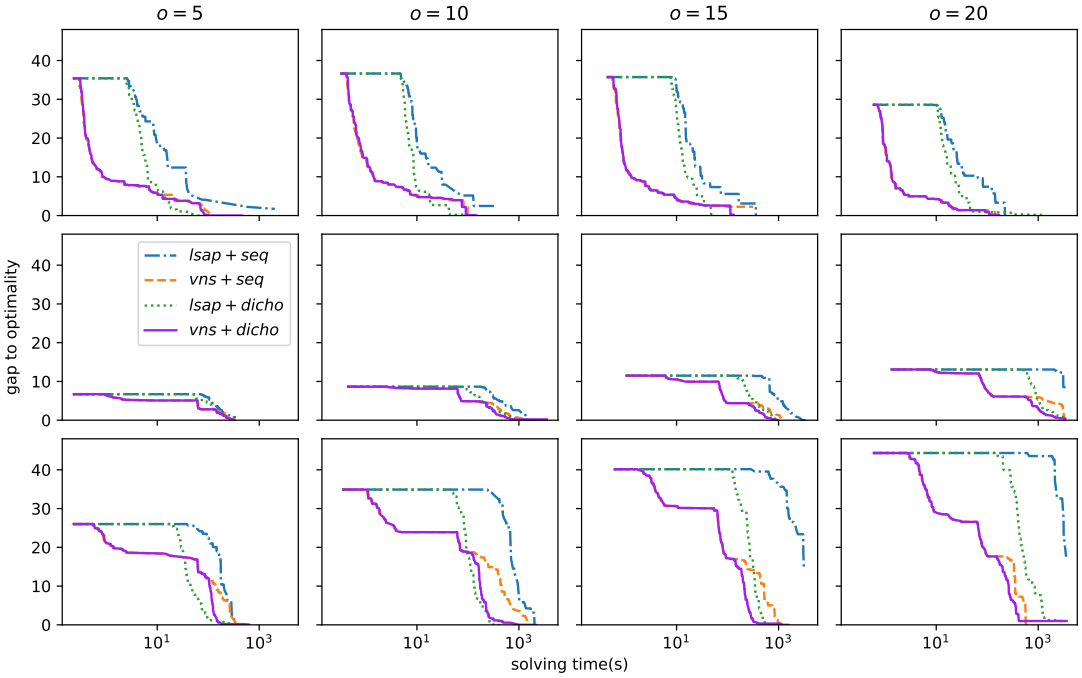

Figure 13: Evolution of the gap to optimality (in percentage) with respect to time for $n = 30$, $o \in \{5, 10, 15, 20\}$ and $d = U$ (Top), B (Middle), or A (Bottom).

instances. However, VNS+DICHO is much more successful than VNS+SEQ for A instances for time limits greater than 100s (corresponding to the time spent by COMBINEDVNS to compute the upper bound when $k_{max} = 7$). This comes from the fact that the upper bound returned by COMBINEDVNS is quite close to the optimal solution for U and B instances (less than 5% for U instances, and less than 6% for B instances), whereas the gap is equal to 18% for A instances (see Fig. 11).

In Fig. 13, we display the evolution of the gap to optimality (in percentage) with respect to time. For LSAP+SEQ and LSAP+DICHO, the curve is horizontal at the beginning of the solution process. This corresponds to the time needed to compute paths of Π_l. This horizontal part is shorter for LSAP+DICHO (especially for B and A instances), as paths are computed up to $(lb + ub)/2$ instead of ub for LSAP+SEQ. VNS+SEQ and VNS+DICHO have identical gaps at the beginning of the search process. This corresponds to the 120 seconds spent by COMBINEDVNS to improve the upper bound. After 120s, VNS+DICHO has smaller gaps than VNS+SEQ, and it is the best performing approach for most instances.

An NC-AMAPF instance has a parameter dt which corresponds to the time needed by a robot to let another robot pass before it at some shared vertex. In all experiments reported in this paper, this parameter has been set to four. We made experiments with other values of dt and observed that it has an impact on instance hardness: the higher dt, the greater the upper bound (computed with VNS) and, therefore, the more expensive the computation of

all paths of length smaller than the upper bound. However, the relative performance of the different considered approaches are not drastically changed when dt is changed.

6. Conclusion

In this work, we extend previous work on non-crossing AMAPF by considering the impact of robots's physical size. We show that motion constraints can be translated into precedence constraints, that imply waiting times when computing the makespan. We prove that the solution of LSAP cannot contain deadlocks and always provides a valid upper bound for the new problem. We propose as well an alternative way to calculate an upper bound from the LBAP solution. Experimental results show us that the two upper bounds have rather similar qualities but the LSAP upper bound is more quickly computed.

We introduce a novel VNS approach that also considers non-shortest paths as neighbors, and we show that it has complementary performance with the VNS approach of (Peng et al., 2021). We propose to combine them sequentially to improve robustness in practice. To solve the problem optimally, we firstly introduce a relaxed CP model that disregards interactions between more than two robots, thus it provides theoretically a lower bound. Then a lazy constraint generation approach is applied to compute the optimal solution.

In this problem, one of the main issues affecting efficiency is related to the initial upper bound used to generate the CP model. The larger the upper bound, the greater the number of candidate paths and the heavier the CP model, which requires more time to solve optimally. To avoid unnecessary path computations, we adopt a dichotomous approach to choose an appropriate upper bound. Experimental results on randomly generated instances have shown us that a sequential combination of VNS and CP enables efficient computation of solutions for this novel variant of AMAPF problem. Additionally, we have introduced parameters in each method to effectively control efficiency and solving time, thereby enhancing its generalization capability for solving diverse instances.

In this work, we adopted a strategy of enumerating all candidate paths that could contribute to the solution. The results show that this step can be very expensive for some difficult instances, especially when the gap between the upper bound solved by VNS and the lower bound by LBAP is large. Instead of precomputing all paths and then selecting compatible ones, we could directly search for a solution in the visibility graph. Geometric constraints associated with paths, such as non-intersecting, as well as their precedence relationships could be partially relaxed in the model (a path is a sequence of edges, and the constraints can be locally satisfied at each vertex by imposing constraints on edge selection). We plan to evaluate the interest of this approach for computing tighter bounds.

Also, to cope with real industrial constraints, we plan to consider heterogeneous velocity in robots' motion model when planning trajectory. In addition, the models and methods introduced in this paper are based on the hypothesis of a planar workspace, which are not easily adaptable to curved surfaces. It will be interesting to extend this work on 3D meshes.

Acknowledgements

This work was supported by the European Commission under the H2020 project Bug-Wright2 (871260): Autonomous Robotic Inspection and Maintenance on Ship Hulls and Storage Tanks.

References

Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., & Sarzyniec, L. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In Ivanov, I. I., van Sinderen, M., Leymann, F., & Shan, T. (Eds.), *Cloud Computing and Services Science*, Vol. 367 of *Communications in Computer and Information Science*, pp. 3–20. Springer International Publishing.

Bhattacharya, S., Likhachev, M., & Kumar, V. (2012). Topological constraints in search-based robot path planning. *Autonomous Robots*, *33*(3), 273–290.

Carlsson, J., Armbruster, B., Rahul, S., & Bellam, H. (2015). A bottleneck matching problem with edge-crossing constraints. *Int. J. Comput. Geom. Appl.*, *25*, 245–262.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. The MIT Press.

Ford-Fulkerson, L. R. (1956). Network flow theory. Tech. rep., Rand Corp Santa Monica Ca.

Hert, S., & Lumelsky, V. (1994). The ties that bind: motion planning for multiple tethered robots. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pp. 2734–2741 vol.4.

Hert, S., & Lumelsky, V. (1999). Motion planning in r/sup 3/ for multiple tethered robots. *IEEE Transactions on Robotics and Automation*, *15*(4), 623–639.

Hert, S., & Lumelsky, V. (1997). Planar curve routing for tethered-robot motion planning. *International Journal of Computational Geometry & Applications*, *7*(03), 225–252.

Lozano-Pérez, T., & Wesley, M. A. (1979). An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, *22*(10), 560–570.

Ma, H., Wagner, G., Felner, A., Li, J., Kumar, T. K. S., & Koenig, S. (2018). Multi-agent path finding with deadlines..

Peng, X., Solnon, C., & Simonin, O. (2021). Solving the Non-Crossing MAPF with CP. In *CP 2021 - 27th International Conference on Principles and Practice of Constraint Programming*, LIPIcs: Leibniz International Proceedings in Informatics, pp. 1–17, Montpellier (on line), France.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2016). Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, *6241*.

Salzman, O., & Halperin, D. (2015). Optimal motion planning for a tethered robot: Efficient preprocessing for fast shortest paths queries. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4161–4166. IEEE.

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

Shnaps, I., & Rimon, E. (2014). Online coverage by a tethered autonomous mobile robot in planar unknown environments. *IEEE Transactions on Robotics*, *30*(4), 966–974.

Sinden, F. (1990). The tethered robot problem. *The International Journal of Robotics Research*, *9*(1), 122–133.

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

Yu, J., & LaValle, S. (2013a). Structure and intractability of optimal multi-robot path planning on graphs. In *In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1444–1449.

Yu, J., & LaValle, S. M. (2013b). Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X: Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics*, pp. 157–173. Springer.

Zhang, X., & Pham, Q.-C. (2019). Planning coordinated motions for tethered planar mobile robots. *Robotics and Autonomous Systems*, *118*, 189–203.