An Efficient Modular Algorithm for Connected Multi-Agent Path Finding

Victorien Desbois, Ocan Sankur, François Schwarzentruber

▶ **To cite this version:**

Victorien Desbois, Ocan Sankur, François Schwarzentruber. An Efficient Modular Algorithm for Connected Multi-Agent Path Finding. ECAI 2024 - 27th European Conference on Artificial Intelligence, Oct 2024, Santiago de Compostela, Spain. pp.1-11. hal-04650916v2

HAL Id: hal-04650916
https://hal.science/hal-04650916v2

Submitted on 31 Jul 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

An Efficient Modular Algorithm for Connected Multi-Agent Path Finding

Victorien Desbois[a], **Ocan Sankur**[b] **and François Schwarzentruber**[c]

[a]Univ Rennes, CNRS, NewLogUp, France, victorien.desbois@inria.fr
[b]Univ Rennes, Inria, CNRS, France, ocan.sankur@inria.fr
[c]Univ Rennes, CNRS, France, francois.schwarzentruber@inria.fr

Abstract. We present a new algorithm for solving the connected multi-agent path finding problem (connected MAPF) which consists in finding paths for a set of agents that avoid collisions but also ensure connectivity between agents during the mission. Our algorithm is based on heuristic search and combines ODrM*, a well known algorithm without connectivity constraints, and an efficient but incomplete solver for the connected MAPF from the literature. We present a formal analysis of the termination and completeness of our algorithm, and present an experimental evaluation, showing a significant improvement over the state of the art.

1 Introduction

Multi-agent pathfinding (MAPF) involves computing paths for a set of agents so that each one reaches a given destination while avoiding collisions and satisfying other constraints [20]. Solving this problem entails planning for agents with different objectives that must cooperate to achieve their respective goals [17]. This has applications in planning for warehouses, search and rescue missions, and nuclear decommissioning. The problem has been extensively studied for discrete maps including grids but many algorithms also apply to maps in the Euclidean space [20].

Some applications of MAPF require *connectivity* constraints among agents. In some cases, a periodic connection between agents is sufficient [8], but specific applications require agents to remain connected during the mission; for example, in order to transmit an uninterrupted video stream or other critical data to human operators [1]. Connectivity constraints also appear as structural constraints in some motion planning problems with the objective of minimizing the stretch of group of agents [6].

The variant of MAPF with connectivity constraints, called the *connected MAPF (CMAPF)* have thus been considered and various aspects have been studied in the literature [21, 8, 3, 14, 15]. Following [20], we consider the classical setting where the environment is modeled by a discrete graph whose nodes are locations that can be occupied by a single agent at any time. The graph contains movement edges along which each agent can move at every step. Moreover, in CMAPF we consider an additional type of edges called *communication edges* which specify whether agents at two given vertices are in direct communication. This is a general setting which can model, for instance, communication by radius (two agents communicate if they are within a given distance). Agents start in a given source configuration and have to reach target locations, while forming a connected graph through communication edges at each step.

The theoretical complexity of the problem is PSPACE-complete [21], even in three-dimensional grids with communication by radius [2]. Several suboptimal algorithms have been given. Most significantly, greedy heuristic algorithms with backtracking appear in [21], and a randomized algorithm based on cooperative A* [18] appears in [2] improving the former in terms of performance; the latter being currently the best performing algorithm for CMAPF.

Our goal in this paper is to improve the performance of solving CMAPF. Let us first present the idea of M*[22] and ODrM*[7], algorithms solving basic MAPF (without connectivity); we will then explain how our solution for CMAPF builds upon these.

The algorithm M* consists in improving over a basic A* search, and the main idea can be summarized as follows: at each node, all agents are moved towards their targets for one step in a decoupled way; if this does not cause any collisions, then we keep these successors; otherwise, we backtrack and enumerate the successors for agents involved in collisions. Moreover, groups of agents entering collisions are put together to form sets of agents, called *meta-agents*. The algorithm uses recursive calls to find successors for meta-agents.

Contributions In this work, we design an algorithm that improves the state of the art for solving CMAPF by providing a suboptimal but complete algorithm, based on the ideas of M*.

M* computes optimal executions for agents by using their individual optimal paths. If a group of agents collides, the algorithm uses A* to plan for those agents. ODM* is a variant of M* that uses Operator Decomposition (OD) instead of A*. Our work is inspired by a recursive modification of ODM* called ODrM*.

While ODrM*uses recursive calls to solve instances for the meta-agents, our target was to define a modular algorithm where the resolution of the smaller instances for meta-agents is addressed by any given algorithm, including efficient but possibly incomplete algorithms. One reason behind this design choice is the following: recursive calls in ODrM*are used as "quick guesses" and the computed successors are only kept if no conflicts are detected; thus it does not make sense to spend too much computational power to compute these guesses. However, when meta-agents become large, these quick guesses become problem instances that are almost as hard as the original problem. As we will see, because connectivity constraints concern *all* agents, they cannot be dealt with considering pairs of agents locally, and they create large meta-agents. Connected MAPF is also harder computationally (the decision problem is PSPACE-complete [21] while MAPF optimization problems are NP-complete [24, 25]),

so a recursive algorithm would not have been a good choice.

In our setting, the solver used for the smaller instances is referred to as the *subsolver*. Our main algorithm for CMAPF we present here, called CODM*, can be seen as a variant of ODrM*in which the subsolver is Connected-CA* [2], which is basically cooperative A* [18] applied to CMAPF. Our algorithm can be seen as a direct improvement over the results of [2]. In fact, while Connected-CA* is efficient, the simple variant is incomplete (and suboptimal); it is only rendered probabilistically complete by a randomization scheme, which yields an exponential worst-case expected termination time. Our algorithm can be seen as a search layer above Connected-CA* which also alleviates some of its limitations: We use the simple variant of Connected-CA* without the randomization scheme as a subsolver (which is fast but incomplete), while the additional layer brought by ODrM*guides the search, and renders it complete.

Our algorithm brings several modifications to ODrM* by making it compatible with *incomplete* subsolvers, and by handling connectivity conflicts (*i.e.* disconnections). Note that when connectivity constraints are ignored, and when the subsolver is replaced by recursive calls, our algorithm becomes equivalent to ODrM*. On the other hand, if one considers a trivial subsolver which always fails, then our algorithm is equivalent to ODM*. While our main motivation and contribution are the design of a faster algorithm for the connected MAPF problem, the termination and completeness proofs we present, and the invariants we identify are also useful to understand the ODrM*algorithm whose formal correctness and termination proofs do not appear in [7]. In particular, we present a formal treatment of the merging of meta-agents and their relations inside the search graph. We prove that the search graph has the invariant that along each edge, a partial order on the meta-agents sets is maintained. This turns out to be a critical invariant (illustrated in Fig. 3) on which the completeness proof relies. This implies that if two agents belong to a meta-agent at some node, they are also in the same meta-agent in the parent node. Intuitively, this is because agents are merged into meta-agents if they are in conflict or if the subsolver fails, so propagating meta-agents means that we are aware of this difficulty already in the parent nodes. We evaluate our algorithm on several benchmarks, and show significant improvement on the success rate over previous algorithms [2].

2 Preliminaries

Topological graph We consider multiple agents moving in discrete environments. A typical example of such environments is grids, as shown in Figure 1. Here, each agent occupies a cell, which is its position. At each discrete step, an agent can either stay or move to an adjacent cell. A common setting is that two agents can communicate if they are within a given range. In this example, let us assume that two agents can communicate if they are within a distance of at most 3 (we assume that cells are of size 1×1).

For more generality and also for simplicity, environments are represented by a so-called *topological graph* $G = (V, E_M, E_C)$ where V is a non-empty finite set of positions, E_M is the set of *movement edges*, and E_C is the set of *communication edges*.

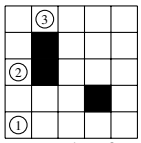


Figure 1: Example of an environment modelled by a topological graph $G = (V, E_M, E_C)$. There are also three agents: 1, 2 and 3.

Example 1. *Figure 1 is modeled by* $G = (V, E_M, E_C)$ *with* $V = \{xy \mid x, y = 1..5\} \setminus \{42, 23, 24\}$ *(we write pairs* (x, y) *as* xy *for simplicity),* $E_M = \{(xy, x'y') \in V^2 \mid |x - x'| + |y - y'| \leq 1\}$, $E_C = \{(xy, x'y') \in V^2 \mid \sqrt{(x - x')^2 + (y - y')^2} \leq 3\}$.

Configurations We consider a set $\mathsf{Ag} = \{1, ..., n\}$ of n agents that must move in (V, E_M) from their initial vertices to their target ones. A *configuration* c is a mapping from Ag into V. It can be thought as a tuple of n vertices of V, denoted $(c_1, ...c_n)$ where for $a \in \mathsf{Ag}$, c_a is the position of the agent a. Given a subset ma of agents, we write c_{ma} for the restriction of c to agents in ma, i.e. $(c_a)_{a \in \mathsf{ma}}$. A configuration is *connected* if it forms a connected sub-graph of (V, E_C).

Example 2. *Figure 1 depicts configuration* $(11, 13, 25)$*, which is connected since* $(11, 13) \in E_C$ *and* $(13, 25) \in E_C$*; that is, the sub-graph of* G *restricted to positions* $11, 13, 25$ *is connected. Intuitively, agent 1 at* 11 *can communicate with the agent 3 at* 25 *via the agent 2 at* 13 *(we consider multi-hop communication).*

Two configurations c and c' of length n are *consequent* if for each $a \in \{1, ..., n\}$ we have $(c_a, c'_a) \in E_M$; thus, each agent makes one move along an edge in E_M. We allow agents to idle, that is, E_M contains all self-loops. An *execution* of length ℓ is a sequence of configurations, denoted $(c^1, ..., c^\ell)$ such that for each $i \in \{1, ..., \ell - 1\}$, c^i and c^{i+1} are consequent.

As in MAPF (and unlike some previous work on CMAPF [8, 21]), we are interested in computing executions without collisions. We consider two types of collision constraints: a weaker form requiring agents to occupy distinct vertices at all times (e.g. following [3, 2]); and a stronger one forbidding agents to take the same edge in opposite directions as well (e.g. as in [26]). We present our experiments for both types of constraints to compare with previous work.

Formally, a configuration c is *collision-free* if all the c_a are distinct. A pair of consequent configurations (c, c') is collision-free if for all agents $a \neq b$, $c'_a = c_b$ implies that $c'_b \neq c_a$ (the agents do not swap positions in one step). An execution $(c^1, ..., c^\ell)$ is collision-free if all its configurations are collision-free, and all pairs (c^i, c^{i+1}) are collision-free. It is *connected* if all its configurations are connected.

The Connected MAPF Problem Given an initial configuration $s = (s_1, ..., s_n)$ and a final one $t = (t_1, ..., t_n)$, the goal in the *connected MAPF (CMAPF)* problem is to compute a connected and collision-free execution from s to t.

The CMAPF problem was proven to be PSPACE-complete on general graphs [21] and also for 3D grids with range-based communication [2]. All hardness results hold regardless of the types of collision constraints, including in the absence of such constraints.

Operator Decomposition When expanding a node n whose configuration is n.c, a naïve approach would consist in creating all nodes n$'$ with n$'$.c being a consequent configuration of n.c. This leads to an exponential branching factor. Operator Decomposition (OD) is a technique to reduce this branching degree [19]. The idea is to assign positions to agents one by one. We thus need to work with incomplete *configurations*, which are partial mappings from Ag to V (in which not all agents have been assigned positions yet). Given an *incomplete* configuration c, let $\mathsf{dom}(c)$ the *domain* of c, that is, the set of agents for which c is defined.

More precisely, each node contains a pair (n.c, n.c') where n.c is a *complete* configuration, and n.c' an *incomplete* configuration such that for all agents a, if c'_a is defined, then $(c_a, c'_a) \in E_M$. The configuration n.c contains the current vertices of the agents, while n.c' provides the successor vertices. A node n is *complete* when n.$c' = \emptyset$ (the empty partial map), and *incomplete* otherwise[1].

[1] In [19], these are called standard and intermediate, respectively.

Connected-CA* The Cooperative A* (*CA**) is an incomplete algorithm that consists in choosing an order of agents, and computing paths for the agents in this order while avoiding conflicts with previously computed paths [18]. Here, each agent is assigned its shortest path to target (among paths avoiding conflicts with previously assigned paths); in particular, the first considered agent follows its shortest path. This algorithm is incomplete because it only considers executions where at least one agent follows its shortest path, while some instances require none of the agents to follow their shortest path. There are several optimizations that improve the performance, namely, the Windowed Hierarchical CA* (*WHCA**) consists in rearranging the agents after a given window size, and dynamically updating the heuristic values. The algorithm Connected-CA* of [2] is a direct application of the former with an additional randomization scheme which generates temporary random target configurations which help configurations get out of deadlock situations. This improves the success rate in practice and renders it complete probabilistically (it finds a solution with probability 1, if there is one).

3 Our algorithm CODM*

We design an algorithm based on heuristic search with OD which build a search graph over configurations, following the lines of ODrM*. The originality of our algorithm is that we combine this with incomplete solvers. We first explain the nodes of the search graph, comment on its fields, and present some other notions used in the algorithm. The full algorithm is then given in Section 3.3. The implementation is available on https://github.com/VictorienDesbois/codm_star.

3.1 Node Data Structure

Following [7], a node is an object containing the following fields:

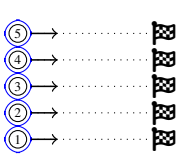
$\mathsf{n}.c$	Current configuration in n
$\mathsf{n}.c'$	Partial next configuration in n
$\mathsf{n}.\mathsf{ODAgts}$	the set of agents on which we apply OD
$\mathsf{n}.\mathsf{meta}$	the set of meta-agents, a partition of Ag
$\mathsf{n}.\mathsf{cost}$	the current cost from the source node to n
$\mathsf{n}.\mathsf{h}$	the heuristic of n to the target configuration
$\mathsf{n}.\mathsf{predecessors}$	the set of predecessors of n
$\mathsf{n}.\mathsf{bestpred}$	the best predecessor so far of n

Configurations. The configurations $\mathsf{n}.c$ and $\mathsf{n}.c'$ represent partially expanded nodes, and allow us to apply OD (see Subsection 2).

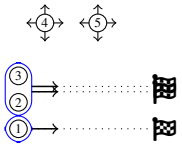
Meta-Agents. A *meta-agent* is a subset of Ag. We denote by $\mathsf{n}.\mathsf{meta}$ the set of meta-agents at node n, which is a partition of Ag. Each node stores a subset $\mathsf{n}.\mathsf{ODAgts}$ of agents on which OD is to be applied when computing successors. By a slight abuse of language, we say that an *agent is OD* at node n if it belongs to $\mathsf{n}.\mathsf{ODAgts}$.

While ODrM*processes meta-agents recursively, in our algorithm, successors are computed in two ways: either $\mathsf{n}.\mathsf{ODAgts}$ is non-empty, and OD is applied on these agents, or $\mathsf{n}.\mathsf{ODAgts}$ is empty, and all meta-agents are assigned successors using a *subsolver*, which is an external algorithm; in our case, this is Connected-CA*. A subsolver is expected to return a plausible next configuration for the meta-agent (for instance, a configuration with good heuristic value to the agents' target). While OD returns all possible successors, the use of the subsolver always yields a single next configuration for a given meta-agent, thus a single successor node for n.

Initially, in the root node (Fig. 2a), there are no OD agents, and the meta-agents are $\{1\}, \dots, \{n\}$. Accordingly, a successor node is computed by calling the subsolver independently for each meta-agent, as illustrated in Fig. 2a. The algorithm visits nodes several times, merge meta-agents at each visit, and add some agents to $\mathsf{n}.\mathsf{ODAgts}$. Thus a typical node looks like in Fig. 2b, where successors are computed differently for OD and other meta-agents.



(a) In the root node, all meta-agents are singletons and no agent is OD. Their next positions is a priori given by the subsolver.



(b) In a typical node, some agents are OD (e.g. agents 4 and 5). The others are partitioned into meta-agents (e.g. $\{1\}$ and $\{2,3\}$) and the subsolver gives the next positions for each meta-agents.

Figure 2: Treatment of regular meta-agents and OD agents in a node.

Predecessors. Each node n also stores the set of node predecessors in $\mathsf{n}.\mathsf{predecessors}$: this is the set of nodes from which the current node was reached in one step during the search. A special field $\mathsf{n}.\mathsf{bestpred}$ contains the predecessor with the least cost; this is used to compute the optimal execution to a given node. A node n' is a *successor* of n if n belongs to $\mathsf{n}'.\mathsf{predecessors}$; this is written $\mathsf{n} \to \mathsf{n}'$. We write $\mathsf{n} \xrightarrow{*} \mathsf{n}'$ if $\mathsf{n} = \mathsf{n}'$ or n' can be reached from n by successively applying $\to$.

Cost, Heuristic, and Priority. We now describe the cost, heuristic and priority of a node that are standard notions in heuristic search algorithms. The field $\mathsf{n}.\mathsf{cost}$ is the minimal *cost* of reaching n from the source node so far in the search graph.

The *heuristic* $\mathsf{n}.\mathsf{h}$ of node n is defined as

$$\mathsf{n}.\mathsf{h} = \epsilon \sum_{a=0}^{|\mathsf{n}.c|} \mathsf{shPathCost}(\mathsf{n}.c_a, t_a) \tag{1}$$

where $\mathsf{shPathCost}(\mathsf{n}.c_a, t_a)$ is the cost of a shortest path from position $\mathsf{n}.c_a$ to the target t_a. The factor ϵ is called *inflation factor* and increases the weight of the heuristic which allows to find sub-optimal executions faster but at the cost of suboptimality. Previous works have considered inflation factors varying between 1 and around 10 in the setting of MAPF [22, 4].

The *priority* of a node n in the priority queue in the search algorithm is $\mathsf{priority}(\mathsf{n}) = \mathsf{n}.\mathsf{cost} + \mathsf{n}.\mathsf{h}$.

Algorithm 1 Node constructor

```
1:  Input: a complete configuration c, a (possibly complete)
    configuration c', subset ODAgts of agents
2:  Output: a node n with n.c = c and n.c' = c'
3:  function Node(c, c', ODAgts):
4:      if N_G contains n with n.c=c and n.c'=c' then
5:          if n.ODAgts ⊈ ODAgts then
6:              n.ODAgts ← n.ODAgts ∪ ODAgts
7:          return n
8:      create an object n of type Node:
9:      if c' is complete then (n.c, n.c') ← (c', ∅)
10:     else (n.c, n.c') ← (c, c')
11:     n.cost ← +∞; set n.h by (1)
12:     n.ODAgts ← ODAgts;
13:     n.meta ← {{a} | a ∈ Ag}
14:     return n
```

Node Constructor. Algorithm 1 explains how a node is constructed. line 4 ensures that there is at most one node for a given pair (c, c'). If there is already a node n with $\mathsf{n}.c = c$ and $\mathsf{n}.c' = c'$, we return it. Otherwise, we create a fresh node. In case c' is complete, we construct a complete node (see line 8). Agents ODAgts are OD agents and thus form n.ODAgts, while all other agents are singleton meta-agents. Initially, when a node n is created, n.cost is set to $+\infty$.

3.2 Conflicts

A node that is created temporarily by the algorithm can contain collisions or can be disconnected. For a node n, we define the following set of agent pairs that are in *vertex conflict*:

$$\{(i,j) \in \mathsf{Ag}^2 \mid i \neq j, \mathsf{n}.c'_i = \mathsf{n}.c'_j \text{ or } \mathsf{n}.c_i = \mathsf{n}.c_j\}. \tag{2}$$

We moreover consider pairs of agents in *swapping conflict*:

$$\{(i,j) \in \mathsf{Ag}^2 \mid i \neq j, \mathsf{n}.c_i = \mathsf{n}.c'_j \text{ and } \mathsf{n}.c_j = \mathsf{n}.c'_i\}. \tag{3}$$

We define collisions(n) as the union of (2) and (3). In our experiments, we will distinguish the setting with only vertex conflicts (following some previous work e.g. [2]), and the one with both vertex and swap conflicts (as in e.g. [4]).

We see collisions(n) as a set of undirected edges between agents, and we write Comp(collisions(n)) for the set of connected components of that graph: it is a partition of the set of agents. For example, if $\mathsf{Ag} = \{1,2,3,4,5\}$ and collisions(n) $= \{(1,2),(2,5)\}$ then we have Comp(collisions(n)) $= \{\{1,2,5\},\{3\},\{4\}\}$. Define

$$\text{conflicts}(\mathsf{n}) = \begin{cases} \text{Comp}(\text{collisions}(\mathsf{n})) \text{ if } \mathsf{n} \text{ is } \textit{incomplete} \text{ or connected,} \\ \{\mathsf{Ag}\} \text{ if } \mathsf{n} \text{ is } \textit{complete} \text{ and not connected,} \\ \emptyset \text{ otherwise.} \end{cases}$$

A node has *conflicts* if conflicts(n) $\neq \emptyset$, and is *conflict-free* otherwise. Our definition of conflicts follows [7], except we consider the connected component of agents in conflict, and add disconnectivity. Unlike collisions, connectivity is a constraint concerning *all* agents, so it cannot be dealt with by handling pairs of agents locally. We thus form a big meta-agent by gathering all agents in case of a connectivity conflict. We will show that we can still guarantee good performance in this case thanks to the use of the efficient subsolver.

In the original ODrM*, forming such a large meta-agent would mean falling back to OD (since recursive calls require smaller instances), which is less efficient. Our insight is to rely on an efficient but possibly incomplete subsolver here (which often does find solutions), and switch to OD only as a last resort. This will be formally explained in the following description algorithm.

3.3 Main Procedure

The main procedure, given in Algorithm 2, has similar skeleton as Dijkstra's algorithm. A priority queue OPEN initially contains the root node associated with the source configuration s (line 4). As long as OPEN is non-empty, we pop a most prioritized node from OPEN. If the current n contains the target configuration t (line 9), we extract the execution by following the pointers to the best predecessors (line 10). Line 12 computes the set of successors of n. If there are no OD agents, we get exactly one successor using the subsolver as an oracle for each meta-agent. Otherwise, we compute the successors via OD.

Algorithm 2 Algorithm CODM*

```
1:  Input: graph G, source and target configurations s, t.
2:  Output a conflict-free execution from s to t.
3:  function SEARCH(G, s, t):
4:      OPEN ← priority queue containing Node(s, ∅)
5:      CLOSED ← ∅
6:      while OPEN ≠ ∅ do:
7:          pop from OPEN a node n with priority(n) minimal
8:          add n into CLOSED
9:          if n.c = t then:
10:             return the execution [s, ..., n.bestpred.c, n.c]
11:         aToMerge ← ∅
12:         S ← { {SUBSOLVERSUCC(n, t)} if n.ODAgts = ∅
                { ODSUCCESSORS(n) otherwise
13:         for n' in S do
14:             if conflicts(n') ≠ ∅ then:
15:                 aToMerge ← merge(aToMerge, conflicts(n'))
16:             else
17:                 add n to n'.predecessors
18:                 cost' ← n.cost + GETCOST(n, n')
19:                 if n' ∉ CLOSED and cost' < n'.cost then
20:                     n'.cost ← cost'
21:                     n'.bestpred ← n
22:                     add n' in OPEN
23:                 if n'∈CLOSED and n'.ODAgts was modified then
24:                     remove n' from CLOSED, add n' into OPEN
25:                 aToMerge ← merge(aToMerge, n'.meta)
26:         BACKPROPMERGE(n, aToMerge, OPEN, CLOSED)
27:         if no node was added to OPEN in this iteration then
28:             BACKPROPODAGTS(n, OPEN, CLOSED)
29:     return No Solution
```

We then process all successors n'. If n' has conflicts, we discard it but keep information about conflicting agents in aToMerge. Otherwise, we update n' in the Dijkstra/A* style (line 18-21). On line 18, GETCOST(n, n') refers to $\sum_{a \in \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')} \mathsf{dist}(\mathsf{n}.c_a, \mathsf{n}'.c_a) + \sum_{a \in \mathsf{dom}(\mathsf{n}'.c') \setminus \mathsf{dom}(\mathsf{n}.c')} \mathsf{dist}(\mathsf{n}.c_a, \mathsf{n}'.c'_a)$, where $\mathsf{dist}(p, p') \in \{0, 1\}$ is the distance in the topological graph between two positions p and p'. If n' is a *complete* node, then $\mathsf{n}'.c' = \emptyset$ and the second part of the sum will be zero. But if n' is an *incomplete* node, then the first part of the sum will be equal to zero because $\forall a \in \mathsf{Ag}, \mathsf{n}.c_a = \mathsf{n}'.c_a$ implies $\mathsf{dist}(\mathsf{n}.c_a, \mathsf{n}'.c_a) = 0$, the second part however will give us the number of agents that have changed positions between n and n'.

On line 24, we reopen a node n' if it is in CLOSED, and if its OD agents were modified (this modification refers to line 6 of the node constructor). This line is crucial for completeness, and is specific to our algorithm due to the use of incomplete subsolvers.

Agents to merge. When n' has conflicts, variable aToMerge gathers information about which meta-agents to merge to avoid recomputing successors with conflicts. If two agents are in conflict in the successors computed by the subsolver, these are merged into a single meta-agent (see Fig. 4a). However, to ensure completeness, such merge operations must be backpropagated to the predecessors of the node. We impose that the partition of meta-agents in a node is always finer than the ones in its predecessors, as shown in Fig. 3. The merge and its backpropagation is performed in BACKPROPMERGE.

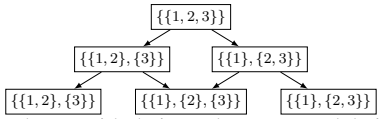
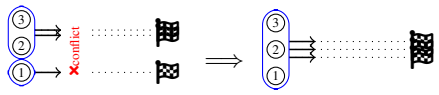
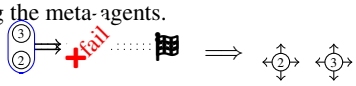


Figure 3: Nodes shown with their predecessors and their meta-agents.



(a) A conflict in the next configuration produced by the subsolver results in merging the meta-agents.



(b) If the subsolver fails to find a solution for a given meta-agent, then all the agents in that meta-agent become OD.

Figure 4: Treatment of the meta-agents: merging and OD.

OD as a last hope. If the algorithm has made no progress in a given iteration, that is, if no nodes were added to OPEN, then on line 27, BACKPROPODAGTS is called on this node, which has the effect of falling back to OD for this node and some of its predecessors. This part of the algorithm is new compared to ODrM*, and is necessary for ensuring completeness.

3.4 Successor Computation

Algorithm 3 Successors given by an optimal solver.

1: **Input:** A node n, the target configuration t.
2: **Output:** a set S of (possibly with conflict) successors for the node n.
3: **function** ODSUCCESSORS(n, t):
4: pick the smallest agent a in $\mathsf{n}.\mathsf{ODAgts} \setminus \mathsf{dom}(\mathsf{n}.c')$
5: $\mathcal{C}' \leftarrow \{\mathsf{n}.c'[a \leftarrow v] \mid (\mathsf{n}.c_a, v) \in E_M\}$
6: **return** $\{\text{NODE}(\mathsf{n}.c, c', \mathsf{n}.\mathsf{ODAgts} \setminus \{a\}) \mid c' \in \mathcal{C}'\}$

Nodes with OD agents. The set of successors is computed by Alg. 3. We pick an agent a, and create a node successor for each possible next position for a. Here, we write $\mathsf{n}.c'[a \leftarrow v]$ for denoting a new configuration which is a copy of $\mathsf{n}.c'$, but in which agent a has been assigned to position v.

Algorithm 4 Single successor given by the subsolver.

1: **Input:** A node n, the target configuration t.
2: **Output:** the successor (possibly with conflict) for the node n.
3: **function** SUBSOLVERSUCC(n, t):
4: $c' \leftarrow \emptyset$; $\mathsf{ODAgts} \leftarrow \emptyset$
5: **for** ma **in** $\mathsf{n}.\mathsf{meta}$ **do**
6: **match** SUBSOLVER($\mathsf{n}.c_{|\mathsf{ma}}, t_{|\mathsf{ma}}$):
7: **case Success**($c^1, \ldots, c^\ell$): $c'_{\mathsf{ma}} \leftarrow c^2$
8: **case Fail**: $\mathsf{ODAgts} \leftarrow \mathsf{ODAgts} \cup \mathsf{ma}$
9: **return** NODE($\mathsf{n}.c, c', \mathsf{ODAgts}$)

Nodes without OD agents. In this case, the set of successors is a singleton containing the node returned by Alg. 4. For each meta agent ma, we call the subsolver from $\mathsf{n}.c_{|\mathsf{ma}}$ with target $t_{|\mathsf{ma}}$ (line 6). We assume that the subsolver either fails, or returns a successor configuration which is conflict-free. Moreover, if the instance has a single agent, then the returned successor must be part of a shortest path to target. If this call is successful, it returns a conflict-free execution $c^1, \ldots, c^\ell$ with $c^1 = \mathsf{n}.c_{|\mathsf{ma}}$ and $c^\ell = t_{|\mathsf{ma}}$ (this execution is only for agents in ma). We then move agents of ma to their next positions given by the second configuration in that execution (line 7).

If the subsolver fails, we declare the faulty agents in ma to become OD in the next node (line 8), see Figure 4b.

Although there are no conflicts in each c'_{ma}, the obtained configuration c', thus the returned successor node, might contain conflicts since the meta-agents were processed separately. These are handled on line 14 in Alg. 2.

3.5 Merging Meta-agents

In this section, we formalize the relations between the meta-agents of the nodes of the search graph, and the property of the merge function which merges meta-agents. This formalization turns out to be crucial for expressing the invariants and the properties used in the termination and completeness proofs (see Section 4).

Definition 1. *Let A and B be two disjoint sets of meta-agents. We say that A is* included/finer *in B, denoted by $A \sqsubseteq B$ if all the meta-agents of A are a subset of at least one meta-agent of B. Formally: $A \sqsubseteq B$ if for all $\mathsf{ma} \in A$, there exists $\mathsf{ma}' \in B$ such that $\mathsf{ma} \subseteq \mathsf{ma}'$.*

Example 3. $\{\{1\}, \{2,3\}, \{4,5\}\} \sqsubseteq \{\{1,2,3\}, \{4,5\}\}$. *However,* $\{\{1,2\}, \{3,4,5\}\} \not\sqsubseteq \{\{1,2,3\}, \{4,5\}\}$.

Let us define the *merge* of two disjoints sets X and Y of meta-agents. It is the finest disjoint set that is coarser than X and Y.

Definition 2. *Let X, Y be two disjoint sets of meta-agents. We define $merge(X, Y)$ to be the disjoint set Z of meta-agents which contains all agents in X, Y (i.e. $\cup Z = \cup_{\mathsf{ma} \in X \cup Y} \mathsf{ma}$), such that $X \sqsubseteq Z$, $Y \sqsubseteq Z$ and Z is $\sqsubseteq$-minimal*[2].

Example 4. *The merge of $X = \{\{1,2,3\}\}$ (singleton set of a meta-agent made of three agents) and $Y = \{\{3,4,5\}, \{6,7\}\}$ (set of two meta-agents) is the following set of two disjoint meta-agents:* $\{\{1,2,3,4,5\}, \{6,7\}\}$.

Given two sets X, Y of meta-agents, $\mathsf{merge}(X, Y)$ can be obtained from $X \cup Y$ by repeatedly merging pairs of meta-agents with non-empty intersection. More precisely, this is a fixed point that can be computed by Algorithm 5.

Algorithm 5 Merge of two disjoint sets X and Y of meta-agents.

1: **Input:** two disjoint sets X and Y of meta-agents.
2: **Output:** the merge of X and Y.
3: **function** MERGE(X, Y):
4: $Z \leftarrow X \cup Y$
5: **while** $\exists ma, mb \in Z$ **and** $ma \cap mb \neq \emptyset$ **do**
6: $Z \leftarrow (Z \setminus \{ma, mb\}) \cup (ma \cup mb)$
7: **return** Z

Example 5. *The computation of* $\mathsf{merge}(X, Y)$ *in Example 4 proceeds as follows. Initially, we have* $Z = \{\{1,2,3\}, \{3,4,5\}, \{6,7\}\}$. *At the first iteration, we assign* $Z \leftarrow \{\{6,7\}\} \cup (\{1,2,3\} \cup \{3,4,5\})$, *and return* $\{\{1,2,3,4,5\}, \{6,7\}\}$.

3.6 Backpropagation merging

The goal of BACKPROPMERGE, given in Algorithm 6, is to merge meta-agents along the predecessors of the given node, to ensure that if $\mathsf{n} \to \mathsf{n}'$, then $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{n}.\mathsf{meta}$ (see Invariant (4) in Section 4).

There is a second back propagation algorithm BACKPROPODAGTS which puts all agents not in $\mathsf{dom}(n.c')$ to $\mathsf{n}.\mathsf{ODAgts}$; this implements the "OD as a last hope" strategy explained above, and is needed for completeness.

[2] meaning that there is no Z' with $Z \neq Z'$, $X \sqsubseteq Z$, $Y \sqsubseteq Z$ and $Z' \sqsubseteq Z$

(a) Office 2D (80×60). (b) Rooms 2D (36×43). (c) Pyramid 3D (21×15×5). (d) Obstacles 2D (70×60×5).

Figure 5: Environments from [2] used for experiments. The communication radius are respectively 1 pixel, 4 pixels, 3 pixels and 3 pixels.

Algorithm 6 Back propagation

Inputs: Node n, and a set of meta agents aToMerge
Output: true iff at least one node was added to OPEN
Effects: ⚠ Local variables of Algorithm 2 are updated.
function BACKPROPMERGE(&n, aToMerge, &OPEN, &CLOSED):
  if aToMerge $\not\sqsubseteq$ n.meta **then**
    n.meta $\leftarrow$ merge(n.meta, aToMerge)
    if n $\in$ CLOSED **then**
      remove n from CLOSED , add n into OPEN
    for n′ **in** n.predecessors **do**
      BACKPROPMERGE(n′, n.meta, OPEN, CLOSED)

Algorithm 7 Back propagation of ODAgts

Inputs: Node n, and a set of meta agents aToMerge
Effects: ⚠ Local variables of Algorithm 2 are updated.
function BACKPROPODAGTS(&n, &OPEN, &CLOSED):
  if n$\in$CLOSED **and** n.ODAgts $\neq$ Ag $\setminus$ dom(n.c') **then**
    n $\leftarrow$ NODE(n.c, n.c', Ag $\setminus$ dom(n.c'))
    remove n from CLOSED , add n into OPEN
  else if n $\notin$ OPEN **then**
    for n′ **in** n.predecessors **do**
      BACKPROPODAGTS(n′, OPEN, CLOSED)

3.7 Optimizations

We noticed that CODM* performs better when target configurations agents are not packed, since the very last steps require careful planning to allow all agents enough space to reach their targets. We consider an optimization where we alternate forward and backward search, switching between the two every k iterations (we empirically chose $k = 500$ in our experiments). Moreover, we use a simple scoring system for start and target configurations to decide at which mode to start the search. A higher score intuitively means that the agent at v is free to move around (the agents are less packed). If the target configuration has a higher score, then we start with forward search, otherwise we start with the backward search. The optimized version of the algorithm is referred to as CODM*-opt.

4 Theoretical Analysis of the Algorithm

The *search graph* denoted by $\mathcal{G}$ is, at a given iteration, a directed over the node set $\mathcal{N}_\mathcal{G}$ which contains all nodes in OPEN or CLOSED; with an edge n from n′, denoted n $\rightarrow$ n′ iff n $\in$ n′.predecessors. We use the notation $\xrightarrow{*}$ as defined previously.

Proving termination is not trivial as visited nodes can be reopened. We prove termination by arguing that at each iteration, either the number of total meta-agents decreases (*i.e.* some meta-agents are merged), or the number of OD agents increases.

Theorem 1 (Termination)**.** *Algorithm 2 terminates.*

We now address completeness which is nontrivial to prove. First notice that the algorithm ensures that each n.meta is a partition of Ag, and that all nodes in $\mathcal{G}$ are conflict-free. We prove that all nodes n, n′ $\in \mathcal{N}_\mathcal{G}$ satisfy the following invariant (see Fig. 3):

$$\text{if } n \xrightarrow{*} n' \text{ then } n'.\text{meta} \sqsubseteq n.\text{meta}. \tag{4}$$

The proof of completeness crucially relies on (4) which is used to prove more detailed invariants that prove completeness. Intuitively, we show that every node n $\in$ CLOSED is either OD and all its possible successors have been generated, or n has a descendant n′ in OPEN, which means that when n′ is processed, back propagation will possibly reach n and merge its meta-agents. But merging meta-agents means that these will eventually form a single meta-agent, and then become OD; then completeness follows since OD does generate all possible successors.

Theorem 2 (Completeness)**.** *If an instance $\langle G, s, t\rangle$ of the CMAPF problem admits a solution, then Algorithm 2 finds one.*

Proofs of both theorems are given in the long version [5].

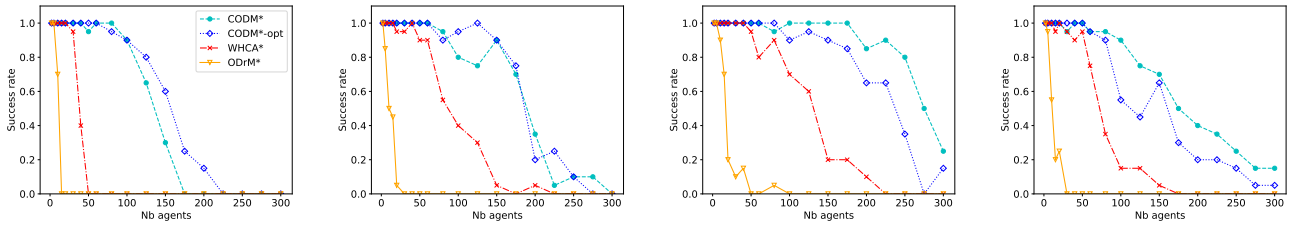
5 Experiments

We evaluate CODM* and CODM*-opt by comparing them with the implementation of Connected-CA* of [2] and a straightforward extension of ODrM*[7] with connectivity constraints. We do not compare here with the algorithms of [21] which were shown to be significantly slower than Connected-CA* [2]; and the algorithm of [13] which fails to improve over [21]. Other algorithms such as [8] consider a different model of connectivity constraints so comparison is not easy. Last, [3] describes an algorithm for a class of topological graphs but it is not clear whether the considered environments here fall in this class; and they do not provide an implementation.

While we compare with *WHCA** that is probabilistically complete, the subsolver used in CODM* is obtained by disabling the randomization scheme which means that it becomes incomplete, but always returns a result fast, either by providing a successful execution or by failing. The experiments were conducted on an Intel core i7 CPU on an Ubuntu 23.10 system. Each run was allowed 4 GB of RAM, and 120 seconds of computation time.

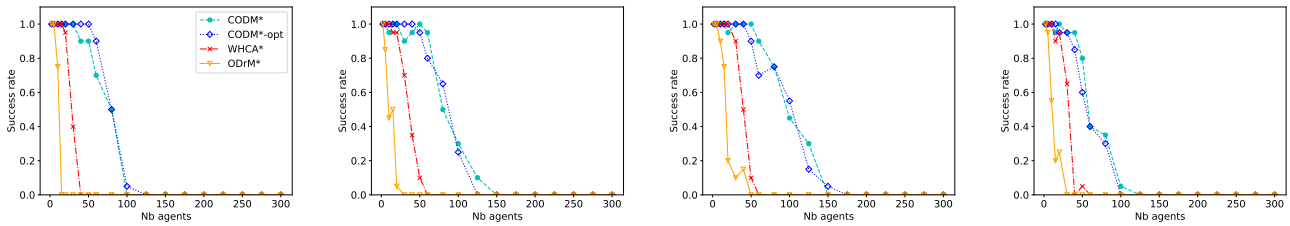
Benchmarks We consider the environments used in [2] with minor modifications (e.g. one map modified to remove an isolated pixel causing deadlocks); we randomly generated 20 fresh instances with n agents, for $1 \leq n \leq 300$. The environments are shown in Fig. 5; the 3D maps were obtained by copying the shown bitmap along the z axis, and adding random obstacles in each dimension.

Results We evaluate our algorithms by distinguishing the case with connectivity and only vertex conflicts, and the case with connectivity, vertex, and swap conflicts. Because [2] and several earlier works only considered the former setting, this allows us to compare with the previous work under the same conditions, while we believe the latter case is more realistic. CODM* performed systematically better than Connected-CA* on all benchmarks. In the case without swapping conflicts (Fig. 6), CODM* has near 100% success rate on instances



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 6: Success rates with connectivity, and vertex conflicts only. Each point (x, y) in the graph of an algorithm means that the percentage of success was y among the instances with x agents in the given environment.



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 7: Success rates considering connectivity, vertex and swapping conflicts: the y axis represent the percentage of success and the x axis the number of agents.

over 100 agents (and up to 170 agents in *Pyramid 3D*). The optimized version improved the success rate on the two first benchmarks; but not on the two others. This might be due to our scoring system giving less relevant information in higher dimensions; this will be further investigated in future work. In the case with swapping conflicts (Fig. 7), we compare our algorithm with our modification of the Connected-CA* implementation of [2] to account for these conflicts. Here, the success rate decreased faster; with CODM* still performing better than Connected-CA*: CODM* ensured a success rate of above 90% on instances with at least 60 agents, while Connected-CA* could only achieve this success rate up to 30 agents. We believe the additional difficulty in the case of swapping conflicts is due to the *density* of the start and target configurations: connectivity constraints force agents to remain close to each other, and when the number of agents increase, the algorithm fails to find successors to many agents due to congestion, which creates an excessive number of collision conflicts, and we end up processing many agents by OD.

Our algorithm computed executions whose makespan (i.e. the size of the longest path) was often comparable with that of Connected-CA*, except that on many instances that are hard for Connected-CA*, CODM* computed shorter executions. Some data comparing the makespans is provided in the long version [5].

Alternative Variants of CODM* We considered a direct extension of ODrM* for connectivity constraints, that is CODM* where recursive calls replace the calls to the subsolver. This approach hardly scaled to instances with 10 agents, so we do not include the details here. We also tried using CODM* to solve MAPF instances (without connectivity constraints). This had comparable performance to ODrM*, in some cases yielding better success rates due to lower memory usage (ODrM* can run out of memory on difficult instances rather quickly, where the subsolver CA* of CODM* needs much less memory). Despite some improvement in performance, this did not achieve the performance of state-of-the-art suboptimal solvers [9].

6 Related Works and Conclusion

The CMAPF problem is closely related to the MAPF problem (without connectivity) where a large body of work exists. Some other algorithms used for MAPF are conflict-based search (CBS) and its variants [16, 10]. These algorithms belong to the *decoupled* approach where paths are computed separately for each agent, and possible conflicts are handled at a higher search level. An alternative approach for CMAPF could be considering MA-CBS which CBS with meta-agents and using Connected-CA* for computing paths for meta-agents. Suboptimal algorithms have also been investigated for MAPF in order to improve the scalability, e.g. EECBS [11], and large neighborhood search [12]. Our algorithm is suboptimal because the subsolver is suboptimal, but we have not investigated further any techniques to exploit suboptimality in favor of performance.

Our algorithm is modular. Indeed, by replacing the functions ODSuccessors and subSolverSucc, we obtain different versions of M*. For example, if the subsolver always fails, we obtain ODM*; and if, in addition, ODSuccessors is replaced with A*, then this yields M*. Replacing ODSuccessors with A*, and using recursive calls instead of the subsolver, we obtain rM*; and using ODSuccessors with recursive calls yields ODrM*. CODM* can thus be seen as an extension of all these variants allowing the use of incomplete (and suboptimal) MAPF and CMAPF subsolvers.

Connectivity constraints imply, as a side effect, that agents remain close to each other during the mission; this is thus related to other types of constraints such as crowd planning in video games where the goal is to move agents from a configuration to another while keeping them grouped [23]. This application has however different constraints such as real-time performance, and connectivity, in our sense, is not a hard constraint; we leave the investigation of similarities for future work.

Acknowledgement. This work was supported by the ANR EpiRL project ANR-22-CE23- 0029.

References

[1] F. Amigoni, J. Banfi, and N. Basilico. Multirobot exploration of communication-restricted environments: A survey. *IEEE Intelli. Sys.*, 32(6):48–57, 2017. doi: 10.1109/MIS.2017.4531226.

[2] I. Calviac, O. Sankur, and F. Schwarzentruber. Improved complexity results and an efficient solution for connected multi-agent path finding. In *22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023)*, 2023.

[3] T. Charrier, A. Queffelec, O. Sankur, and F. Schwarzentruber. Complexity of planning for connected agents. *Auton. Agents Multi Agent Syst.*, 34(2):44, 2020. doi: 10.1007/s10458-020-09468-5. URL https://doi.org/10.1007/s10458-020-09468-5.

[4] M. Damani, Z. Luo, E. Wenzel, and G. Sartoretti. Primal _2: Pathfinding via reinforcement and imitation multi-agent learning-lifelong. *IEEE Robotics and Automation Letters*, 6(2):2666–2673, 2021.

[5] V. Desbois, O. Sankur, and F. Schwarzentruber. An Efficient Modular Algorithm for Connected Multi-Agent Path Finding. Technical report, HAL, Oct. 2024. URL https://hal.science/hal-04650916.

[6] S. P. Fekete, P. Keldenich, R. Kosfeld, C. Rieck, and C. Scheffer. Connected coordinated motion planning with bounded stretch. *Autonomous Agents and Multi-Agent Systems*, 37(2):43, 2023.

[7] C. Ferner, G. Wagner, and H. Choset. Odrm* optimal multirobot path planning in low dimensional search spaces. In *2013 IEEE international conference on robotics and automation*, pages 3854–3859. IEEE, 2013.

[8] G. A. Hollinger and S. Singh. Multirobot coordination with periodic connectivity: Theory and experiments. *IEEE Transactions on Robotics*, 28(4):967–973, 2012. doi: 10.1109/TRO.2012.2190178.

[9] J. Li. *Efficient and Effective Techniques for Large-Scale Multi-Agent Path Finding*. PhD thesis, University of Southern California, 2022.

[10] J. Li, D. Harabor, P. J. Stuckey, H. Ma, G. Gange, and S. Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301:103574, 2021.

[11] J. Li, W. Ruml, and S. Koenig. EECBS: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12353–12362, 2021.

[12] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig. MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 10256–10265, 2022.

[13] A. Queffelec, O. Sankur, and F. Schwarzentruber. Conflict-based search for connected multi-agent path finding. *arXiv preprint arXiv:2006.03280*, 2020.

[14] A. Queffelec, O. Sankur, and F. Schwarzentruber. Planning for Connected Agents in a Partially Known Environment. In *AI 2021 - 34th Canadian Conference on Artificial Intelligence*, pages 1–23, Vancouver / Virtual, Canada, May 2021. URL https://hal.archives-ouvertes.fr/hal-03205744.

[15] A. Queffelec, O. Sankur, and F. Schwarzentruber. Complexity of planning for connected agents in a partially known environment. *Theoretical Computer Science*, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.11.015. URL https://www.sciencedirect.com/science/article/pii/S030439752200679X.

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

[17] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[18] D. Silver. Cooperative pathfinding. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'05, page 117–122. AAAI Press, 2005.

[19] T. S. Standley and R. Korf. Complete algorithms for cooperative pathfinding problems. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[20] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pages 151–158, 2019.

[21] D. Tateo, J. Banfi, A. Riva, F. Amigoni, and A. Bonarini. Multiagent connected path planning: Pspace-completeness and how to deal with it. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. doi: 10.1609/aaai.v32i1.11587. URL https://ojs.aaai.org/index.php/AAAI/article/view/11587.

[22] G. Wagner and H. Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ international conference on intelligent robots and systems*, pages 3260–3267. IEEE, 2011.

[23] S. Yang, T. Li, X. Gong, B. Peng, and J. Hu. A review on crowd simulation and modeling. *Graphical Models*, 111:101081, 2020.

[24] J. Yu. Intractability of optimal multirobot path planning on planar graphs. *IEEE Robotics and Automation Letters*, 1(1):33–40, 2015.

[25] J. Yu and S. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, pages 1443–1449, 2013.

[26] Z. Zhang, Q. Guo, J. Chen, and P. Yuan. Collision-free route planning for multiple agvs in an automated warehouse based on collision classification. *IEEE Access*, 6:26022–26035, 2018. doi: 10.1109/ACCESS.2018.2819199.

A Appendix

A.1 Additional Experiment Data

Figures 8 and 9 show cactus plots of running times for the algorithms we consider, namely, ODrM* [7], Connected-CA* [2], CODM*, and CODM*-opt with and without swapping conflicts.

Figures 10 and 11 show the cactus plots of makespans of the computed executions for the algorithms we consider, with and without swapping conflicts. The makespan of an execution is the size of the longest path of an individual agent.

CODM* often computed executions with shorter makespan, especially on instances that are hard for Connected-CA*. One exception was on Office 2D with swapping conflicts (Fig. 11), where Connected-CA* found slightly shorter executions (but it failed to solve many instances).

A.2 Proofs

Before diving into the proofs, let us present the notations used to talk about the search graph. We then give the proofs.

A.2.1 Notations

At any iteration, the nodes created by the algorithm define the *search graph* denoted by $\mathcal{G}$: For a search graph $\mathcal{G}$, let $\mathcal{N}_\mathcal{G}$ denote the set of nodes (which is the union of OPEN and CLOSED). Recall that there is an edge from n to n', denoted $\mathsf{n} \to \mathsf{n}'$ iff $\mathsf{n} \in \mathsf{n}'.\mathsf{predecessors}$. We write $\mathsf{n} \xrightarrow{*} \mathsf{n}'$ if there is a sequence of nodes $\mathsf{n}_1, \dots, \mathsf{n}_n$ such that $\mathsf{n}_i \to \mathsf{n}_{i+1}$ for all $1 \leq i \leq n-1$, $\mathsf{n}_1 = \mathsf{n}$, and $\mathsf{n}_n = \mathsf{n}'$. Note that we have $\mathsf{n} \xrightarrow{*} \mathsf{n}$ by definition. We distinguish the *initial node*, denoted init, which is the node that contains the initial configuration. Note that by construction, after each iteration, for all nodes n of $\mathcal{G}$, we have $\mathsf{init} \xrightarrow{*} \mathsf{n}$.

A.2.2 Termination

We call a pair (c, c') where c is a full configuration and c' an incomplete configuration a *configuration pair*. Let $\mathcal{CC}$ denote the finite set of configuration pairs.

Proof of Theorem 1 (Termination). For each search graph $\mathcal{G}$, we define the following function $f_\mathcal{G}$ mapping configuration pairs (c, c') to $\mathbb{N}$, as follows. Given $(c, c') \in \mathcal{CC}$,

$$f_\mathcal{G}(c, c') = \begin{cases} |\mathsf{n}.\mathsf{meta}| + |\mathsf{Ag} \setminus \mathsf{n}.\mathsf{ODAgts}| & \text{if } \mathsf{n} \in \mathcal{N}_\mathcal{G} \\ 2|\mathsf{Ag}| + 1 & \text{otherwise.} \end{cases}$$

Intuitively, $f_\mathcal{G}(c, c')$ assigns, to any node of the graph, the number of meta-agents plus the number of non-OD agents at that node, and $2|\mathsf{Ag}| + 1$ to any configuration pair that is not a node in the graph. Note that the number of meta-agents in any node of $\mathcal{G}$ is at most $|\mathsf{Ag}|$, and we also have $|\mathsf{n}.\mathsf{ODAgts}| \leq |\mathsf{Ag}|$. So assigning $2|\mathsf{Ag}| + 1$ to configuration pairs not yet in the graph ensures that when a node with the said pair is added to the graph, $f_\mathcal{G}(c, c')$ decreases.

Consider

$$V = \left(\sum_{(c,c') \in \mathcal{CC}} f_\mathcal{G}(c, c'), |\mathcal{CC}| - |\mathsf{CLOSED}| \right)$$

The first component V_1 is the sum of the function $f_\mathcal{G}$ over all configuration pairs (c, c'). The second component V_2 is the difference between the size of the finite set $\mathcal{CC}$ and the size of the CLOSED set. We always have $|\mathcal{CC}| \geq |\mathsf{CLOSED}|$ since there is at most one node per configuration pair. Both components of V are nonnegative. We are going to prove that V decreases at each iteration, in the lexicographic order.

First, notice that $f_\mathcal{G}(c, c')$ can only decrease or stay unchanged at every iteration. In fact, when a fresh node is created with (c, c'), we have $f_\mathcal{G}(c, c') < 2|\mathsf{Ag}| + 1$ so the value decreases. If one modifies an existing node for a pair (c, c'), this is only done either by merging meta-agents (in which case $\mathsf{n}.\mathsf{meta}$ decreases), or by adding agents to $\mathsf{n}.\mathsf{ODAgts}$ in which case $|\mathsf{Ag} \setminus \mathsf{n}.\mathsf{ODAgts}|$ decreases (while none of the terms can increase).

Consider an iteration of Algorithm 2. On line 8, a node n is added to CLOSED: this is the only place a node is added to CLOSED, and this happens at every iteration. If no node is removed from CLOSED in iteration, then V_2 decrease and so does V. We are going to distinguish cases where a node is removed from CLOSED, and argue that V decreases at each case.

There are three places where a node is removed from CLOSED.

Case 1. Main Algorithm Assume that some node n' is removed from CLOSED in Algorithm 2, on line 24. This is only possible if $\mathsf{n}'.\mathsf{ODAgts}$ was modified in the node constructor on line 6, which means that $|\mathsf{Ag} \setminus \mathsf{n}.\mathsf{ODAgts}|$ decreases (and recall that $\mathsf{n}.\mathsf{meta}$ never increases). Thus V_1 decreases.

Case 2. BACKPROPMERGE Assume a node is removed from CLOSED inside BACKPROPMERGE. By the conditional of line 5, we have $\mathsf{aToMerge} \not\sqsubseteq \mathsf{n}.\mathsf{meta}$, and the merge operation of line 6 merges some meta-agents in n. Thus, $|\mathsf{n}.\mathsf{meta}|$ decreases, that is, V_1 decreases.

Case 3. BACKPROPODAGTS Assume a node is removed from CLOSED in BACKPROPODAGTS. By the conditional, this only happens for a node n with $\mathsf{n}.\mathsf{ODAgts} \neq \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$, while line 5 calls the node constructor with $\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$, which means that $|\mathsf{Ag} \setminus \mathsf{n}.\mathsf{ODAgts}|$ decreases, and so does V_1.

It follows that Algorithm 2 terminates. □

A.2.3 Completeness

Lemma 1. *Consider the search graph $\mathcal{G}$ at the beginning of any iteration of the algorithm. For all nodes $\mathsf{n}, \mathsf{n}' \in \mathcal{N}_\mathcal{G}$, if $\mathsf{n} \xrightarrow{*} \mathsf{n}'$, then $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{n}.\mathsf{meta}$.*

Proof. Initially the graph only contains one node, so the property holds trivially.

Consider any iteration where the property holds.

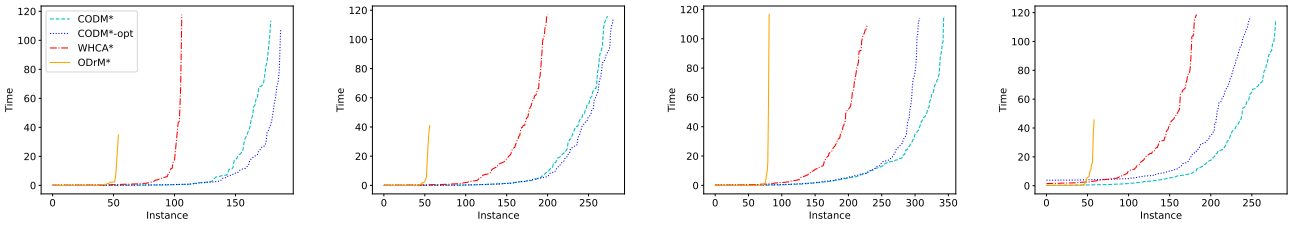
We are going to prove this property for all n, n' such that $\mathsf{n} \to \mathsf{n}'$. In fact, if $\mathsf{n} \to \mathsf{n}'' \to \mathsf{n}'$, and $\mathsf{n}''.\mathsf{meta} \sqsubseteq \mathsf{n}'.\mathsf{meta}$, $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{n}''.\mathsf{meta}$, then by transitivity of the relation $\sqsubseteq$, we have $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{n}.\mathsf{meta}$.

Let the node popped from the open list in a given iteration be called the *pivot* node. Let p denote the pivot node of the considered iteration.

There are two types of nodes whose meta-agents are modified in an iteration: newly created successor nodes of p; and all predecessor nodes n with $\mathsf{n} \xrightarrow{*} \mathsf{p}$ on which backPropagate is called (recall that backPropagate is called recursively along predecessors of p).

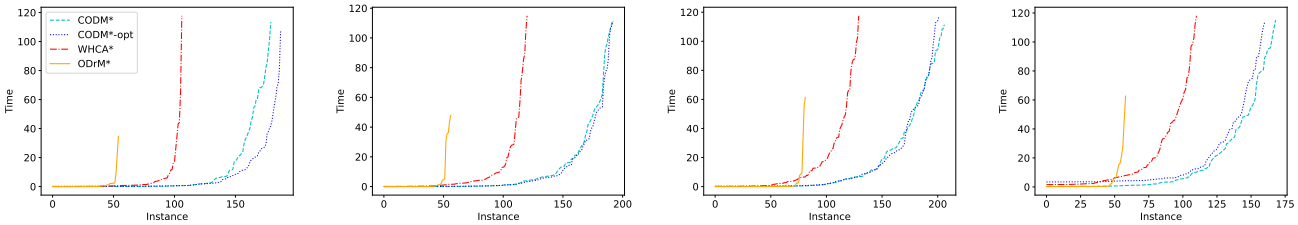
We prove that the property holds for all edges $\mathsf{n} \to \mathsf{n}'$ where backPropagate was called on n. We proceed by induction on the recursion depth of backPropagate.

Base case. The base case (with recursion depth 0) is when $\mathsf{n} = \mathsf{p}$. Let us first show that the property holds for edges of type $\mathsf{p} \to \mathsf{n}'$



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 8: Cactus plots of executions times for the algorithms in the case with connectivity, and vertex conflicts only. Each point (x, y) in the graph of an algorithm means that x benchmarks were solved each within y seconds. Thus a graph that is lower than the other means the algorithm is faster.



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 9: Cactus plots of executions times for the algorithms in the case with connectivity, and vertex with swapping conflicts. Each point (x, y) in the graph of an algorithm means that x benchmarks were solved each within y seconds. Thus a graph that is lower than the other means the algorithm is faster.

where n' was added as a successor to p. Here if n' is a new node, then, by definition, $\mathsf{n}'.\mathsf{meta}$ contains $|\mathsf{Ag}|$ singletons, and we do have $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{p}.\mathsf{meta}$ trivially. Assume that n' belongs to CLOSED or OPEN. By line 25, and Definition 2, we have $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{aToMerge}$; moreover, backPropagate is called on p in Algorithm 2 (line 26). It follows that, by line 6 of Algorithm 6, $\mathsf{p}.\mathsf{meta}$ is updated so that $\mathsf{aToMerge} \sqsubseteq \mathsf{p}.\mathsf{meta}$. By transitivity, we have $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{p}.\mathsf{meta}$.

Inductive Case. Consider now nodes $\mathsf{n} \to \mathsf{n}'$ with $\mathsf{n} \to^* \mathsf{p}$ and $\mathsf{n}' \to^* \mathsf{p}$ such that n' was visited by backPropagate before n is visited. But then backPropagate was called on n with second argument $\mathsf{n}'.\mathsf{meta}$ (on line 10), which implies that at the end of the said call, $\mathsf{n}'.\mathsf{meta} \sqsubseteq \mathsf{n}.\mathsf{meta}$. □

Proof of the Theorem 2 (Completeness). The algorithm has the following invariants:

- (I1) $\forall \mathsf{n} \in \mathcal{N}_\mathcal{G}, \mathsf{n}.\mathsf{ODAgts} \subseteq \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$.
- (I2) $\forall \mathsf{n} \in \mathsf{CLOSED}, \mathsf{n}.\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$ or $\exists \mathsf{n}' \in \mathsf{OPEN}, \mathsf{n} \xrightarrow{*} \mathsf{n}'$.
- (I3) For all nodes $\mathsf{n} \in \mathcal{N}_\mathcal{G}$, if $\mathsf{n}.\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$ then either $\mathsf{n} \in \mathsf{OPEN}$ or for some agent $a \in \mathsf{n}.\mathsf{ODAgts}$, all conflict-free nodes

$$\{\mathsf{Node}\left(\mathsf{n}.c, \mathsf{n}.c'[a \leftarrow v], \mathsf{n}.\mathsf{ODAgts} \setminus \{a\}\right) (\mathsf{n}.c_a, v) \in E_M\} \tag{5}$$

are in $\mathcal{N}_\mathcal{G}$.

We first prove (I1). Assume that the invariant holds at the beginning of an iteration. The only place where $\mathsf{n}.\mathsf{ODAgts}$ is modified for a node n is the node constructor, Algorithm 1, line 6, where $\mathsf{n}.\mathsf{ODAgts}$ is updated as $\mathsf{n}.\mathsf{ODAgts} \cup \mathsf{ODAgts}$. The node constructor is called in three places: in Algorithm 4 with $\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$, in Algorithm 3 with $\mathsf{ODAgts} = \mathsf{n}.\mathsf{ODAgts} \setminus \{a\}$, and in Algorithm 7 with $\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$. In all cases, (I1) is preserved.

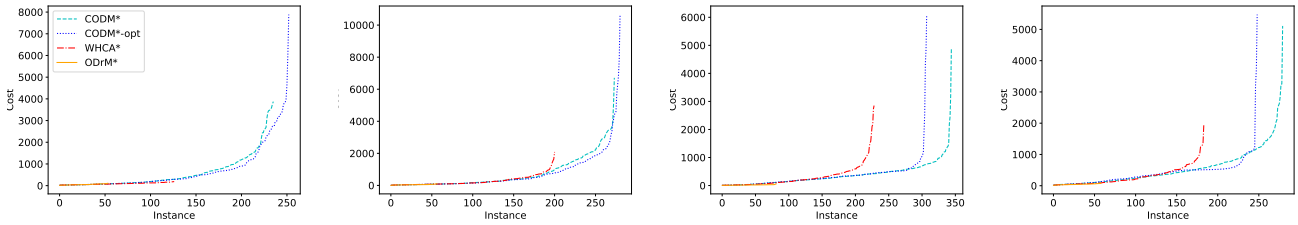
We now prove (I2). Initially, CLOSED is empty, so the invariant holds. Assume that (I2) holds at the beginning of an iteration. At each iteration, a single node n is removed from OPEN, and is added to CLOSED. Moreover, other operations in the main loop, or inside backPropagate possibly adds nodes in OPEN but do not add any other node to CLOSED. To prove that (I2) holds after the iteration, we thus only need to check whether the condition still holds for n itself and all nodes from which n is reachable, that is n' such that $\mathsf{n}' \xrightarrow{*} \mathsf{n}$.

During the iteration where n is popped from OPEN, if there exists $\mathsf{n}'' \in S$ that is without conflicts, and not in CLOSED, then the edge $\mathsf{n} \to \mathsf{n}''$ is added to $\mathcal{G}$ and n'' is in OPEN, and the invariant is satisfied for n and all n' with $\mathsf{n}' \xrightarrow{*} \mathsf{n}$. If no successor of n was added to OPEN, but some $\mathsf{n}'' \in S$ is in CLOSED, and its $\mathsf{n}''.\mathsf{ODAgts}$ was modified (either Algorithm 3, line 6, or Algorithm 4, line 9, then n'' is also added to OPEN. The invariant also holds for all nodes.

Assume no node in S was added to OPEN during the iteration. If n is added back to OPEN during the call to backPropagate (Algorithm 6, line 8), the invariant holds again (recall that $\mathsf{n} \xrightarrow{*} \mathsf{n}$). Otherwise, no node was added to OPEN in the current iteration (Algorithm 2, line 27). In this case, Algorithm 7 is called, which visits the current node and all its predecessors until, some node in CLOSED is encountered with $\mathsf{n}.\mathsf{ODAgts} \neq \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$, in which case $\mathsf{n}.\mathsf{ODAgts}$ is modified and n is added to OPEN. Thus, along each branch, all visited nodes n either $\mathsf{n}.\mathsf{ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$, or they are added to OPEN; therefore the invariant holds for all nodes that can reach n.

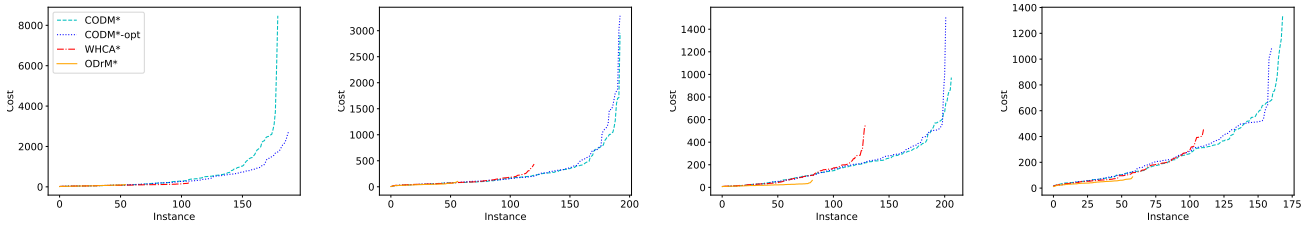
We consider now (I3). Initially, the graph only contains the initial node with $\mathsf{n}.\mathsf{ODAgts} = \emptyset$, so the invariant holds.

Consider an iteration, and assume that (I3) holds at the begin-



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 10: Cactus plots of costs for the algorithms in the case with connectivity, and vertex conflicts only. Each point (x, y) in the graph of an algorithm means that x benchmarks were solved providing an execution with makespan y (which is the size of the longest path). Thus a graph that is lower than the other means that the algorithm finds shorter executions.



(a) Office 2D (b) Rooms 2D (c) Pyramid 3D (d) Obstacle 3D

Figure 11: Cactus plots of costs for the algorithms in the case with connectivity, vertex, and swapping conflicts.

ning.

Consider an iteration in which some $\mathsf{n} \in \mathcal{N}_\mathcal{G}$ was modified or freshly created and added to $\mathcal{N}_\mathcal{G}$ and satisfies $\mathsf{n.ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$. Notice that any modification to $\mathsf{n.ODAgts}$ occurs either in the node constructor, or in BACKPROPODAGTS. In the former case, n belongs to S.

We distinguish two cases. First, assume that n belongs to S at the considered iteration, and that it is conflict-free (if it has a conflict, n is never added to $\mathcal{N}_\mathcal{G}$).

- If n is not in CLOSED and its cost was improved, then it is added to OPEN (Algorithm 2, line 22) and the property holds for n.
- If n is in CLOSED and $\mathsf{n.ODAgts}$ was just modified in the node constructor, n is re-added to OPEN, and the property holds for n.
- Otherwise, both of the conditionals evaluate to false, which means that one of the conditions hold:
  - $\mathsf{n} \in$ CLOSED and $\mathsf{n.ODAgts}$ was not modified. Consider the latest iteration where n was popped from OPEN (such an iteration exists since n is in CLOSED). Furthermore, at that iteration, we must already have $\mathsf{n.ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$ since any modification to $\mathsf{n.ODAgts}$ means n is put back to OPEN (see BACKPROPODAGTS). During this iteration, ODSuccessors was called, thus, all of successors of n were added to S for some agent $a \in \mathsf{n.ODAgts}$ (Algorithm 3, line 6). Thus, all conflict-free successors $\mathsf{n} \to \mathsf{n}'$ were added to $\mathcal{N}_\mathcal{G}$, which means `(I3)` holds.
  - $\mathsf{cost} \geq \mathsf{n.cost}$ and (either $\mathsf{n} \notin$ CLOSED or $\mathsf{n.ODAgts}$ was not modified). If $\mathsf{n} \notin$ CLOSED, then n is in OPEN (because n is in $\mathcal{N}_\mathcal{G}$), and the invariant holds for n. Otherwise, if $\mathsf{n} \in$ CLOSED, then $\mathsf{n.ODAgts}$ was not modified, and this is the previous case.

Assume now that $\mathsf{n.ODAgts}$ was modified in BACKPROPODAGTS. But this function sets $\mathsf{n.ODAgts} = \mathsf{Ag} \setminus \mathsf{dom}(\mathsf{n}.c')$ and adds n to OPEN, so `(I3)` holds.

We now prove the main statement. Consider an execution $c^1, \ldots, c^n$ that is a solution of the problem. Assume that the algorithm has terminated without a solution, and has generated the graph $\mathcal{G}$. Let $i \leq n$ be the maximal index such that there is a complete node n in $\mathcal{G}$ with $\mathsf{n}.c = c^i$. By assumption, $i < n$ since otherwise the algorithm would have returned a solution; and $i \geq 1$ since c^1 is the initial configuration. By the maximality of i, no node with configuration c^{i+1} is reachable from n.

Because the algorithm terminates without a solution only when OPEN is empty, by `(I2)`, we must have $\mathsf{n.ODAgts} = \mathsf{Ag}$. But (c^i, c^{i+1}) is a consequent pair of configurations, so by a successive application of `(I3)`, we get that, because $\mathsf{OPEN} = \emptyset$, there exists $\mathsf{n}_1, \ldots, \mathsf{n}_{|\mathsf{Ag}|}$ with $\mathsf{n} \to \mathsf{n}_1 \to \mathsf{n}_2 \to \ldots \to \mathsf{n}_{|\mathsf{Ag}|}$ where $\mathsf{n}_{|\mathsf{Ag}|}$ a complete node with $\mathsf{n}_{|\mathsf{Ag}|}.c = c^{i+1}$; such that for all $1 \leq k \leq |\mathsf{Ag}| - 1$, $|\mathsf{dom}(\mathsf{n}_k.c')| = k$, and $\mathsf{n}_k.c' = c^{i+1}_j|_{\mathsf{dom}(\mathsf{n}_k.c')}$. In other terms, c^{i+1}_j is reached step by step by applying OD $|\mathsf{Ag}|$ times. Thus, because the algorithm terminates by Theorem 1, it must terminate by returning a solution. □