Journal of Artificial Intelligence Research 62 (2018) 273–314 Submitted 10/16; published 06/18

Solving Multi-agent Path Finding on Strongly Biconnected Digraphs

Adi Botea ADIBOTEA@IE.IBM.COM
IBM Research
Dublin, Ireland

Davide Bonusi DAVIDEBONUSI@GMAIL.COM
Raffmetal S.p.a
Brescia, Italy

Pavel Surynek PAVEL.SURYNEK@FIT.CVUT.CZ
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic

Abstract

Much of the literature on suboptimal, polynomial-time algorithms for multi-agent path finding focuses on undirected graphs, where motion is permitted in both directions along a graph edge. Despite this, traveling on directed graphs is relevant in navigation domains, such as path finding in games, and asymmetric communication networks.

We consider multi-agent path finding on strongly biconnected directed graphs. We show that all instances with at least two unoccupied positions have a solution, except for a particular, degenerate subclass where the graph has a cyclic shape. We present diBOX, an algorithm for multi-agent path finding on strongly biconnected directed graphs. diBOX runs in polynomial time, computes suboptimal solutions and is complete for instances on strongly biconnected digraphs with at least two unoccupied positions. We theoretically analyze properties of the algorithm and properties of strongly biconnected directed graphs that are relevant to our approach. We perform a detailed empirical analysis of diBOX, showing a good scalability. To our knowledge, our work is the first study of multi-agent path finding focused on directed graphs.

1. Introduction

Multi-agent path finding (MAPF) is an important computational problem, with applications in areas such as robotics, traffic optimization, vessel navigation and computer games. Not surprisingly, the problem has received a considerable attention in computer science areas such as artificial intelligence, robotics and graph theory. In a common problem formulation, coined as *cooperative path finding* (Silver, 2005), the purpose is to have every agent navigate from its original location to its target location, while avoiding collisions and deadlocks. The navigation environment is typically represented as a gridmap (Silver, 2005) or as a more generic graph (Ryan, 2008).

One important category of scalable approaches to MAPF is represented by suboptimal rule-based algorithms. These approaches work under the key assumption that the underlying graph is *undirected*. For instance, rule-based algorithms Push and Swap (Luna & Bekris, 2011) and Push and Rotate (de Wilde, ter Mors, & Witteveen, 2013) implement a core primitive, called swap, where agents must move in both directions along some graph edges. Among other moves, the swap primitive involves moving an agent to an adjacent node, to allow another agent to pass through, after which the first agent comes back to its former location, traversing the graph edge in the opposite

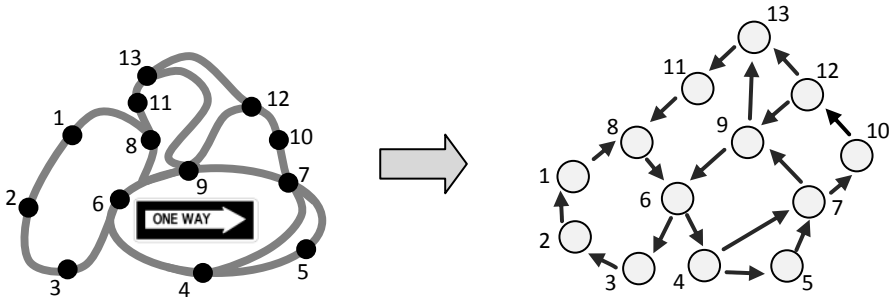

Figure 1: An example of unidirectional road network and its abstraction as a digraph.

direction. The MAPP algorithm (Wang & Botea, 2011) relies on reverting part of the recently performed moves, after an agent reaches its target. Using edges in both directions is also employed in algorithms such as TASS (Khorshid, Holte, & Sturtevant, 2011) and BIBOX (Surynek, 2009). In the latter, agents in a cycle are rotated in order to relocate a selected agent, and eventually rotated back after the agent gets out of the cycle.

In navigation domains, unidirectional edges can arise from the properties of the environment, such as the existence of one-way entrances, exits, escalators, bridges and roads. Some agents could be able to travel down the hill or float down the river, but not the other way around. Furthermore, the literature shows approaches where unidirectional traffic is imposed on purpose on a game map, with the goal of avoiding head-to-head collisions between mobile agents (Wang & Botea, 2008). Motion in directed graphs is also important in asymmetric communication networks (Marina & Das, 2002; Jetcheva & Johnson, 2006; Wu & Grumbach, 2010). Unidirectional networks also appear in traffic domains such as road maps. See Figure 1 for an illustration.

We contribute the first tractability analysis of multi-agent path finding on directed graphs (digraphs). We focus on strongly biconnected digraphs, i.e., strongly connected digraphs where the undirected graphs obtained by ignoring the edge orientations have no cut vertices.[1] We demonstrate that all instances with at least two unocuppied vertices (blanks) can be solved in polynomial time, except for the particular case of graphs with a cyclic shape, where instances may or may not have a solution. We introduce diBOX, a suboptimal algorithm, and formally discuss its completeness, correctness, and complexity. We present formal properties of strongly biconnected digraphs that are relevant to our approach. We provide a detailed empirical analysis of diBOX, showing that it scales convincingly beyond the capabilities of an optimal solver. On small instances where an optimal solver can succeed, the quality of solutions computed with diBOX ranges from nearly optimal to a deviation from optimal values by a factor of at most 4.5. To our knowledge, this is the first theoretical study of the multi-agent path finding problem on any kind of directed graphs.

Part of our theoretical results have previously been reported in a short conference paper (Botea & Surynek, 2015). Compared to that, we extend the contents substantially. We have implemented our algorithm and performed an empirical evaluation. We present techniques for improving the performance of diBOX in terms of running time and solution quality. The presentation of the theoretical results is extended with additional proofs, examples and details. The steps of the algorithms are also presented in more detail, with additional pseudocode and discussions. Part of the additional material has been included in a Master thesis (Bonusi, 2015). We fix an error present in the original

1. A cut vertex is a vertex whose removal from the graph would separate the graph into two or more disjoint subgraphs.

paper (Botea & Surynek, 2015). As presented in that paper, the algorithm has a worst-case complexity of $\mathcal{O}(n^4)$, instead of the $\mathcal{O}(n^3)$ claimed (where n is the number of graph nodes). A fairly minor modification of the algorithm ensures that the complexity is indeed $\mathcal{O}(n^3)$.

Section 2 overviews related work. This is followed by a background section. We point out a partitioning of the class of strongly biconnected digraphs into two disjoint categories in Section 4. This separation is leveraged in our approach to solving multi-agent path finding on strongly bi-connected digraphs, presented in Sections 5 and 6. Section 7 focuses on a path finding problem on strongly biconnected digraphs where one agent has to reach a given target from a given starting position. Other agents may exist on the map, but their final positions are irrelevant. This is both an important building block to our multi-agent path finding approach, and an interesting problem on its own (Wu & Grumbach, 2010). Our algorithm for multi-agent path finding on strongly biconnected digraphs, diBOX, is presented in Section 8. It is followed by a graph decomposition strategy that allows improving the performance of diBOX. The empirical evaluation comes next, followed by concluding remarks and future work ideas. For a better presentation flow, some proofs and algorithmic details are available in an appendix with three sections.

2. Related Work

Previous formal studies of multi-agent path finding, sometimes also called coordinated pebble motion in graphs, appear to be focused on undirected graphs (Wilson, 1974; Kornhauser, Miller, & Spirakis, 1984). For instance, Wilson (1974) explicitly requires that the adjacency relation between agent configurations (called "labellings") is symmetric, which is equivalent to stating that the graph is undirected. All graphs considered in these two works appear to have undirected edges, with no discussion about if or how parts of the study, such as the considered permutation groups, would be applicable to directed graphs. We see our work as complementary to previous work on undirected graphs, being a step towards achieving a similar level of understanding for directed graphs.

Algorithms like BIBOX (Surynek, 2009), Push and Swap (Luna & Bekris, 2011), TASS (Khorshid et al., 2011), and Push and Rotate (de Wilde, ter Mors, & Witteveen, 2014) build on abovementioned formal studies and typically use small sets of movement rules to achieve the final configuration of agents. The key assumption in these rules is that the underlying graph is undirected.

Among existing rule-based, suboptimal, polynomial-time algorithms for undirected graphs, we view Surynek's (2009, 2014b) algorithm BIBOX as the most related to our work. The main difference is that BIBOX works on undirected graphs, whereas our diBOX algorithm works on directed graphs. As mentioned in the introduction, BIBOX explicitly relies on the fact that edges allow travel in both directions. The concept of an ear decomposition exists for both directed and undirected graphs. The similarity between BIBOX and diBOX is in the fact that both algorithms rely on an ear decomposition of the input graph. They both adopt a high-level solving strategy where ears are solved one by one, leaving the basic cycle at the end. However, the technical details related to solving individual ears are quite different in the cases of undirected graphs and directed graphs respectively.

Although rule-based algorithms are scalable and provide completeness guarantees, their solutions can suffer in terms of quality. Therefore multiple search-based algorithms have been developed, including optimal and bounded suboptimal techniques.

Search-based techniques can be applied to both directed and undirected graphs, even though they are typically evaluated on undirected graphs, such as grid maps. Examples include suboptimal,

incomplete methods like WHCA* (Silver, 2006), that implement cooperative variants of A* (Hart, Nilsson, & Raphael, 1968) in which state expansion can easily be adapted to take the direction of edges into account.

Optimal search-based algorithms include A* with OD+ID (operator decomposition + independence detection) (Standley, 2010) and M* (Wagner & Choset, 2015). In some optimal algorithms, such as CBS (Sharon, Stern, Felner, & Sturtevant, 2015) and ICBS (Boyarski, Felner, Stern, Sharon, Tolpin, Betzalel, & Shimony, 2015), the search space is a tree of conflicts. ICTS (Sharon, Stern, Goldenberg, & Felner, 2013) transforms the search space into a sequence of multi-value decision diagrams (MDDs) that correspond to gradually increasing costs. Such techniques can also be applied to directed MAPF. Compared to the undirected case, directed edges only change the enumeration of successor states in these algorithms. Other search-based methods can provide bounded-suboptimal solutions (Barer, Sharon, Stern, & Felner, 2014; Cohen & Koenig, 2016).

A very generic category of methods with respect to the orientation of edges in the underlying graph is represented by compilation-based methods. They include both optimal and suboptimal techniques with respect to various objectives. Existing compilation-based methods reduce MAPF to *Propositional Satisfiability* (SAT), as performed by Surynek (2012, 2014a, 2016), or *Answer Set Programming* (ASP) (Erdem, Kisa, Öztok, & Schüller, 2013). Off-the-shelf SAT and ASP engines respectively can be used as a solver. Directed edges can be introduced into compilation-based methods with minor changes in existing encodings, while the top-level search strategy remains unchanged. An advantage of compilation-based methods is that new heuristics and learning techniques developed for SAT or ASP can be leveraged when solving MAPF instances. On the other hand, MAPF domain knowledge is not necessarily reflected in compiled instances to the same extent as in the case of dedicated search based algorithms.

Optimal methods are less scalable, which is not surprising, given that solving MAPF optimally has been shown to be NP-hard (Ratner & Warmuth, 1986; Surynek, 2010). A major focus in contributions to optimal methods is an improved speed performance in practice.

A specific variant of motion planning assumes the existence of one mobile robot and several mobile obstacles (Papadimitriou, Raghavan, Sudan, & Tamaki, 1994). This could be seen as a multi-agent path finding problem where only one agent has a specific target to achieve. Wu and Grumbach (2010) studied motion planning with one agent and several mobile obstacles on directed graphs, showing cases where the feasibility can be decided in linear time. Solving such instances is a building block in our algorithm for multi-agent path finding, as we show later in the paper.

3. Background

In this section we overview a few concepts and results from the literature that form a starting point to our work.

Definition 1. *A* directed graph *or* digraph *is a structure $D = (V, E)$, where V is a set of nodes (vertices) and E is a set of directed edges. A directed edge is an ordered pair (u, v) with $u, v \in V$.*

Definition 2. *Given a digraph $D = (V, E)$ and a set of agents A, a configuration of agents over D is a placement of agents in vertices of the graph, with at most one agent in each vertex. Formally, the configuration is a uniquely invertible assignment of agents to vertices $\alpha : A \longrightarrow V$.*

If we need to know what agent is placed in a given vertex we may use inverse configuration $\alpha^{-1} : V \longrightarrow A \cup \{\text{blank}\}$ which is well defined due to the unique invertibility of α (the blank is used for unoccupied vertices).

A configuration can be transformed into another one using *moves*. A move involves changing the position of one agent to a neighboring vertex, provided that the target vertex is blank. Moves are possible only along the positive orientation of edges. The time is discretized, and moves are performed at discrete time steps. For simplicity, in this work we assume that agents move one at a time.

Definition 3. *An instance of multi-agent path finding over directed graphs consists of a digraph $D = (V, E)$, a set of agents A, an initial configuration $\alpha_0 : A \longrightarrow V$, and a goal configuration $\alpha_+ : A \longrightarrow V$. The task is to find a sequence of moves over D that transform α_0 to α_+.*

An undirected graph G is biconnected if G is connected and there are no cut vertices in G. In other words, removing any single vertex would keep the graph connected. A digraph D is strongly connected if, for any two distinct vertices v and w, there exist both a path from v to w and a path from w to v in D. Given a digraph D, $\mathcal{G}(\mathcal{D})$ is the underlying graph of D, i.e., the undirected graph obtained by ignoring the orientation of the edges (Wu & Grumbach, 2010).

Definition 4 (Wu and Grumbach (2010))**.** *Let D be a digraph. D is said to be strongly biconnected if D is strongly connected and $\mathcal{G}(\mathcal{D})$ is biconnected.*

To keep our notations simple, when the context is very clear, we sometimes use a graph's name to also refer to its set of nodes or to its set of edges. For instance, if L is a graph, we can say "a node $n \in L$ or an edge $e \in L$". Likewise, a union of two or more graphs is a graph comprising the union of the nodes together with the union of the edges.

The following definition is slightly adapted from Wu and Grumbach (2010).

Definition 5. *An* ear decomposition[2] *of a digraph $D = (V, E)$ is an ordered sequence of sub-digraphs of D, say $[L_0, L_1, \ldots, L_r]$, such that:*

- *L_0 is a cycle; and*
- *$\forall i \in \{1, \ldots, r\}$, L_i is a path (chain) whose two endpoints belong to the subgraph*

$$D_{i-1} = \bigcup_{j=0}^{i-1} L_j,$$

but no other vertices or edges of L_i belong to D_{i-1}.

Figure 2 shows an example. Each sub-digraph L_i is called an ear.[3] We say that L_0 is the *basic cycle* and all other ears are *derived ears*. In the example, L_0 contains nodes $4, 5, 7, 9, 6$ and the corresponding edges $(4, 5)$, $(5, 7)$, $(7, 9)$, $(9, 6)$, $(6, 4)$.

Given an ear L, $|L|$ denotes its number of vertices. The *interior* of a derived ear L, denoted as $\text{int}(L)$, refers to the contained vertices different from its endpoints. Its number of vertices is

2. Wu and Grumbach (2010) also use the term *closed ear decomposition* to refer to an ear decomposition.
3. Besides *ears*, the literature also shows additional names, such as *handles*.

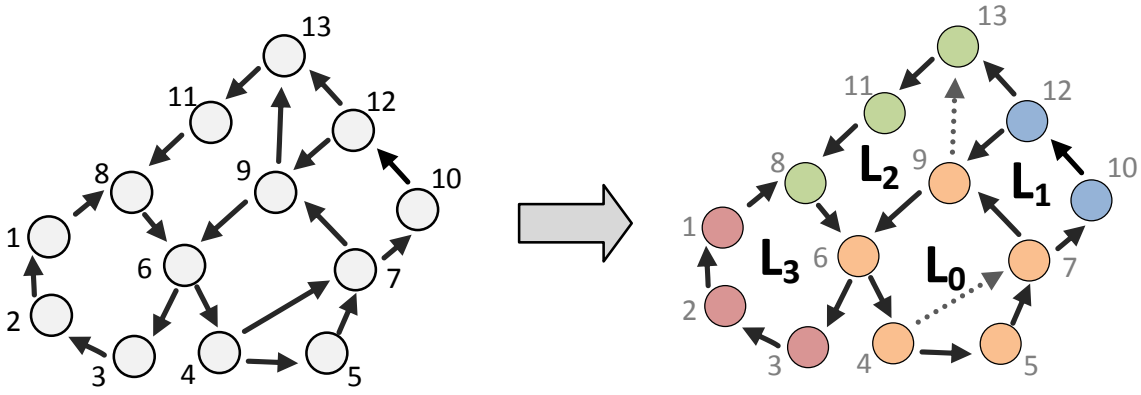

Figure 2: An example of a strongly biconnected digraph and its ear decomposition. Dashed edges are trivial derived ears (not explicitly labeled to avoid clutter).

denoted as $|\text{int}(L)|$. The endpoints are called the entrance and the exit, respectively. For instance, in Figure 2, $\text{int}(L_1) = \{10, 12\}$. Vertex 7 is the entrance of L_1, and vertex 9 is its exit. Likewise, $\text{int}(L_2) = \{13, 11, 8\}$. Ear L_2 has vertex 12 as an entrance, and vertex 6 as an exit. Finally, the entrance of L_3 is vertex 6, and the exit is vertex 8. Its interior contains vertices 3, 2 and 1.

An ear is *trivial* if it has exactly one edge (Bang-Jensen & Gutin, 2008). Edges $(4, 7)$ and $(9, 13)$ are trivial ears. An ear is *cyclic* if its endpoints are represented by a single vertex. Our example has no derived cyclic ears.

Given an ear decomposition $O = [L_0, L_1, \ldots, L_r]$, we call the *$i$-prefix decomposition* the decomposition O_i restricted to the first i ears: $O_i = [L_0, L_1, \ldots, L_i]$. Notice that this is a proper ear decomposition on its own, corresponding to a subgraph D_i of D, whose nodes and edges are precisely the nodes and edges contained in the first i ears. We call D_i, which we first introduced in Definition 5, the i-prefix subgraph. As the i-prefix decomposition uniquely determines the i-prefix subgraph, we may use these two names interchangeably when the context is sufficiently clear.

In the example, O_1 contains the nodes $4, 5, 7, 9, 6, 10, 12$ and all the edges involved in L_0 and L_1: $(4, 5)$, $(5, 7)$, $(7, 9)$, $(9, 6)$, $(6, 4)$, $(7, 10)$, $(10, 12)$, and $(12, 9)$. Notice that the trivial ear $(4, 7)$ is not part of O_1.

A digraph is said to be trivial if and only if it has only one vertex. In this paper, we work with non-trivial digraphs.

Definition 6. *An* open ear decomposition *of a digraph D is an ear decomposition with no cyclic derived ears.*

The decomposition shown in Figure 2 and discussed earlier in this section is open, as each derived ear has its entrance node distinct from its exit node (i.e., no derived ear is cyclic).

We list a powerful result that points out a certain structure which is present in strongly biconnected digraphs. In our work, we leverage this structure to solve multi-agent path finding problems on this class of graphs.

Theorem 1 (Wu and Grumbach (2010))**.** *Let D be a non-trivial digraph. D is strongly biconnected if and only if D has an open ear decomposition. Moreover, any cycle can be the starting point of an open ear decomposition.*

Corollary 1. *Let D be a strongly biconnected digraph, and let $O = [L_0, L_1, \ldots, L_r]$ be an open ear decomposition. For every i with $0 \leq i \leq r$, the i-prefix subgraph D_i is strongly biconnected.*

Proof. Obviously, $O_i = [L_0, L_1, \ldots, L_i]$ is an open ear decomposition, which further implies, according to the previous theorem, that its corresponding digraph D_i is strongly biconnected. □

Remark 1. *Let D be a strongly biconnected digraph, and let $O = [L_0, L_1, \ldots, L_r]$ be an open ear decomposition. By removing all trivial derived ears, we obtain a valid open ear decomposition, and thus a strongly biconnected digraph (see Figure 2 for an illustration).*

4. Characterizing Strongly Biconnected Digraphs

It turns out that the class of strongly biconnected digraphs can be partitioned into two complementary, disjoint categories: *partially-bidirectional cycles* and graphs for which a *regular* open ear decomposition exists. The next two definitions introduce these categories. Then we claim that they are in fact complementary, in Proposition 1. Finally, we claim that, given a strongly biconnected digraph, testing whether it belongs to one category or the other can be done in a time that is linear in the number of digraph nodes. The importance of these two categories stems from the fact that we address each category with a different multi-agent path finding approach, which we present in Sections 5 and 6.

Definition 7. *A digraph is a* partially-bidirectional cycle *if it consists of a simple cycle C, plus zero or more edges of the type (u, v), where $(v, u) \in C$ (i.e., edges obtained by swapping the direction of an edge from C).*

Figure 3 shows an example. In a way, this is a simple, degenerate case of a strongly biconnected digraph. Addressing multi-agent path finding on such graphs is simple, as we show in Section 5.

Definition 8. *We say that an open ear decomposition of a strongly biconnected digraph is* regular *if the basic cycle L_0 has three or more vertices, and there exists a non-trivial derived ear with both ends attached to the basic cycle.*

The example shown in Figure 2 is an example of a graph with a regular open ear decomposition. In particular, the open ear decomposition shown in the figure (out of many decompositions possible) is regular. Indeed, L_0 has 5 nodes (3 would be sufficient), and L_1 is a non-trivial derived ear with both endpoints attached to L_0.

Proposition 1. *For every strongly biconnected digraph $D = (V, E)$, exactly one of the following two cases holds:*

1. *D has a regular open ear decomposition; or*
2. *D is a partially-bidirectional cycle.*

Proposition 2. *Let $D = (V, E)$ be a digraph. Checking if D is a partially-bidirectional cycle can be done in $\mathcal{O}(|V|)$ time steps.*

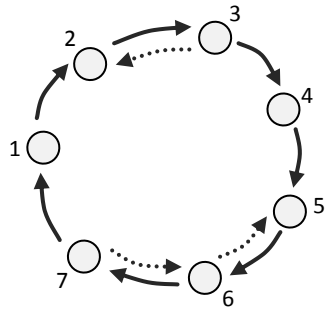

Figure 3: An example of a partially-bidirectional cycle. The edges of the simple cycle are shown as solid edges. We call their direction the forward direction. Dotted edges connect, in the opposite direction, adjacent nodes in the cycle. We call this the backwards direction.

The proofs to these two propositions are available in Appendix A.

Corollary 2. *Let $D = (V, E)$ be a strongly biconnected digraph. Checking if D belongs to one category or another can be done in $\mathcal{O}(|V|)$ time steps.*

In the coming sections we focus on solving multi-agent path finding instances on strongly biconnected digraphs. We will present a suboptimal algorithm that runs in polynomial time, and is complete for instances with at least two blanks on strongly biconnected digraphs.

5. Solving Partially-Bidirectional Cycles

In discussing multi-agent path finding on strongly biconnected digraphs, we start with the easy case of partially-bidirectional cycles. Then, in the next section, we discuss the complementary, more involved case of strongly biconnected digraphs with regular open ear decompositions.

Proposition 3. *An instance on a partially-bidirectional cycle, with at least one blank, has a solution if and only if the ordering of the agents in the initial state is identical with the ordering in the goal state.*

As no swapping between agents is possible, an instance has a solution if and only if the agents come in the right order in the first place.

Given a cycle C with at least one blank, we call *ShiftAgentsInCycle* the routine that performs forward moves in the cycle until a new desired configuration is obtained. *ShiftAgentsInCycle* can be used to solve a partially-bidirectional cycle. It will also be useful in other parts of our diBOX algorithm, as discussed in the next section.

Proposition 4. *Given a cycle C, the worst-case complexity of the method* ShiftAgentsInCycle *is* $\mathcal{O}(|C|^2)$.

Indeed, moving all agents by one step each requires $\mathcal{O}(|C|)$ moves. We may need up to $\mathcal{O}(|C|)$ such shifts in total.

Corollary 3. *Given an instance that has a solution on a partially-bidirectional cycle $D = (V, E)$, computing the solution can be performed with a time and move complexity of $\mathcal{O}(|V|^2)$.*

6. Solving Instances on Digraphs with Regular Open Ear Decompositions

In this section we show that, on digraphs with a regular open ear decomposition, every instance with two or more blanks has a solution. We present an algorithm capable of computing suboptimal solutions in polynomial time.

6.1 Overview of the Approach

The main steps are the following:

- First, transform the problem instance so that, in the goal state, at least two blanks belong to the basic cycle L_0. We call the former the *original instance*, and the latter the *transformed instance*. Having a goal state with (at least) two blanks in L_0 is useful because subsequent steps rely on such an assumption, simplifying those steps. The mapping from the original instance into the transformed instance is described in Section 6.2.

- *Solve* the derived ears, one by one, in reverse order (i.e., starting with L_r and going all the way to L_1), similarly to Surynek's (2009, 2014b) strategy for undirected graphs. By definition, we say that a derived ear is *solved* if all interior positions labeled as goals are occupied by the corresponding agents, and all other interior positions (if any) are blank. When solving a derived ear, we never touch the interior of previously solved ears.

  For example, in Figure 2, we solve the derived ears in the order L_3, L_2, L_1. When solving ear L_3, assume that an agent Mickey has node 1 as a goal, an agent Minnie has node 2 as a goal, and the goal of an agent Pluto is node 3. As nodes 1, 2 and 3 are the interior nodes of the ear L_3, this ear is considered solved when each of these three agents reach their goal node.

  Notice that solving a derived ear does not require solving the endpoint nodes (entrance and exit). For instance, the entrance of L_3 is node 6. As node 6 is also an interior point of another ear, namely L_0, node 6 will be solved later, when solving that particular ear (L_0). Likewise, node 8, the exit point of L_3, will be solved when solving ear L_2, which contains node 8 as an interior point.

  Solving derived ears is discussed in detail in Section 6.3.

- Solve the basic cycle L_0. I.e., make sure that all goals in the basic cycle are occupied by their corresponding agents. Section 6.4 presents details on this step.

- Finally, the solution to the transformed instance is mapped into a solution to the original instance. This is discussed in Section 6.2.

6.2 Mapping the Original Instance into the Transformed Instance and Back

When the original instance has less than two blanks in the basic cycle L_0, the original instance is converted into a transformed instance as follows: Identify a node g that must be blank in the goal of the original instance, and a node $n \in L_0$ that is not blank in the goal of the original instance. Let $\pi(g, n)$ be a directed path from g to n. In the transformed instance, shift all goal positions

backwards along this path, making sure that a new blank is present in L_0 in the goal state. Repeat this step if needed, to bring yet another blank to L_0. We call this procedure *BorrowBlanks*.

At the end, the solution to the transformed instance is mapped into a solution to the original instance as follows. Agents are pushed forward along the $\pi(g, n)$ path(s) mentioned earlier, making sure they reach their original goals (method *ReturnBlanks*).

6.3 Solving Derived Ears

Derived ears are solved one by one, in reverse order, starting with L_r and finishing with L_1. Solving an individual ear will rely on the following property.

Proposition 5. *When solving a derived ear L_i, (at least) two blanks are available in the digraph $D_i = \bigcup_{j \leq i} int(L_j)$.*

Proof. This is shown with an induction on the number of derived ears solved so far. In the basic case, $i = r$. Thus, D_i contains all nodes of the original digraph, which in turn has at least two blanks. When $0 < i < r$, all derived ears L_p with $p > i$ are in a goal state, which means that their actual number of blanks is equal to the number of blanks in their goal state. Since the goal state of the overall instance has (at least) two blanks in the basic cycle L_0, it follows that at least two blanks are located outside of the interiors of $L_{i+1}, \ldots, L_r$, i.e., they are located in $D_i = \bigcup_{j \leq i} \text{int}(L_j)$. □

Consider a derived ear $L_i, i > 0$. We say that its goal configuration is $q_1, q_2, \ldots, q_z$ if $q_1, q_2, \ldots, q_z$ are agents whose goal positions belong to the interior of L_i, in this order (i.e., q_1 is the closest to the entrance, and q_z is the farthest from the entrance). Solving L_i pushes agents inside the ear in order, starting with q_z. For example, when solving ear L_3 in Figure 2, we push into that ear, in order, agent Mickey (whose goal is node 1), followed by agent Minnie (goal node 2), followed by Pluto (goal node 3). The result is that, in the end, all these three agents will sit on their goal positions.

Solving a derived ear L_i is an iterative process. Each iteration pushes one more agent inside the ear. After $z - l$ iterations, the agents $q_{l+1}, \ldots, q_z$ have been pushed inside, on the first $z - l$ interior positions of L_i, and we need to insert the next agent q_l. For simplicity, and without any loss of generality, assume that all interior positions of L_i are occupied, and two blanks are located elsewhere. We distinguish between two cases:

Case 1. In this case, the next agent q_l to be inserted is currently outside the ear L_i. The strategy works as follows:

- First, bring one blank[4] to the last interior position of L_i, with movements only inside the subgraph $D_i = \bigcup_{j \leq i} \text{int}(L_j)$. As blanks travel backwards in a directed graph, the positions of $q_{l+1}, \ldots, q_z$ are not touched.
- Then, agent q_l is brought to the entrance of the ear, with movements only inside the subgraph $D_{i-1} = \bigcup_{j \leq i-1} \text{int}(L_j)$ (i.e., without touching $\bigcup_{j \geq i} \text{int}(L_j)$). This is possible with one remaining blank in use, as initially proven by Wu and Grumbach (2010) in the proof to their Theorem 14. We generically call a method that can move an agent within a k-prefix subgraph *MoveAgentInSubgraph*. We discuss concrete ways to implement such a method in Section 7 and Appendix B.

4. Bringing one blank to a vertex v of a subgraph D' works as follows. We identify a blank vertex u in D' and a path within D' from v to u. Push forward by one step all agents placed along that path. As agents travel forward, the blank travels backward, reaching node v.

- Then agent q_l is pushed inside L_i, reaching a configuration where the first $z - l + 1$ interior positions of L_i are occupied with agents $q_l, q_{l+1}, \ldots, q_z$.

Case 2. In the second case, the agent q_l is already inside L_i, but in a wrong position. The strategy, described in detail below, can be summarized as follows:

- Take agent q_l out of L_i. A side effect is that previously inserted agents, such as $q_{l+1}, \ldots, q_z$, will be disturbed from their positions.
- Insert agents $q_{l+1}, \ldots, q_z$ back into L_i.
- These steps will reduce Case 2 to Case 1.

Definition 9. *Let π be a simple path (chain)*[5] *and let n be node in the path. An edge (n, m) is an* escape door *for π if either m does not belong to π, or m is in π, being at least two positions earlier than n in π.*

Intuitively, an escape door allows an agent to avoid moving forward on a pre-established route involving path π. Agents can use the escape door and never go beyond node n along the pre-established route. We will discuss the usefulness of escape doors and show an example later in this section. Before that, we prove that escape doors exist every time when we need them.

Lemma 1. *Let $D = (V, E)$ be a strongly biconnected digraph with a regular open ear decomposition that has no trivial derived ears. Then, for every edge $(u, v) \in E$, we have that $(v, u) \notin E$.*

Proof. A proof by induction on the number of ears is straightforward. No edge in the basic cycle can be mirrored by another edge in the opposite direction, since: i) L_0 has at least three nodes (according to the definition of regular decompositions), and ii) there are no trivial derived ears. Every new non-trivial derived ear L_i maintains the desired property, since every edge in L_i has at least one endpoint that is new, i.e., it did not belong to the $(i-1)$-prefix $[L_0, \ldots, L_{i-1}]$. □

Proposition 6. *Let $O = [L_0, L_1, \ldots, L_r]$ be a regular open ear decomposition of a strongly biconnected digraph. For any derived ear L_i, there exists a path π in the $(i-1)$-prefix subgraph $O_{i-1} = [L_0, L_1, \ldots, L_{i-1}]$, from the exit to the entrance of L_i, such that π has an escape door (n, m), with m also in O_{i-1}.*

Proof. By removing all trivial derived ears, if any, we obtain a valid strongly biconnected digraph with a regular open ear decomposition. Thus, without loss of generality, we assume that the input graph has no trivial derived ears. Let X and E be the exit and the entrance of L_i. As O_{i-1} is strongly biconnected, it has a path from X to E and a path from E to X. Let π and π' be two arbitrarily chosen simple paths from X to E, and from E to X respectively. Let (a, E) be the last edge on the path π, and let (E, b) be the first edge on the path π'. According to the previous lemma, $a \neq b$. It follows that edge (E, b) is an escaping door for π. □

Recall that we are discussing Case 2, where the agent q_l is inside the ear, but in a wrong position (i.e., not right behind agent q_{l+1}). The agent has to be brought out first, after which the agents $q_{l+1}, \ldots, q_z$, if any, will be re-inserted. This method is called *TakeAgentOutsideEar*.

5. Recall that a simple path is a path without node repetitions.

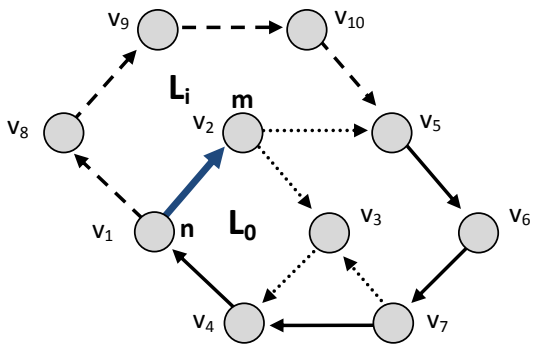

Figure 4: Taking an agent out of a partially solved ear with the help of an escape door.

It works as follows. We build a simple cycle C' by putting together L_i and π, and bring two blanks to nodes m and n mentioned in Proposition 6. Agents are pushed forward repeatedly in the cycle C'. In addition, we interleave zero or more moves through the escape door. Specifically, every time that agent q_l reaches node n, q_l is pushed aside through the escape door (i.e., pushed to node m). This ensures that q_l stays outside of the interior of L_i. The process stops when all agents $q_{l+1}, \ldots, q_z$ (if any) reach their positions inside L_i again.

Even if $m \in C'$, recall that m is not the node right before n in the cycle. This ensures that pushing q_l from n to m does not cause a deadlock (i.e., there is no repetition of the previous global agent configuration).

Algorithm 1: SOLVEDERIVEDEAR

Input: a digraph D, an ear L_i, a goal configuration α_+ of agents over D
Output: sequence of moves solving L_i

```
let L_i = [x_0, x_1, x_2, ..., x_{z+1}];
let π be a simple path from the exit of L_i to the entrance of L_i, with an escape door;
let C' be a cycle obtained from L_i and π;
for l = z down to 1 do
    q_l ← α_+^{-1}(x_l);
    if α^{-1}(x_0) ≠ q_l then
        if q_l ∈ {α^{-1}(x_z), ..., α^{-1}(x_{z-l+1})} then
            TakeAgentOutsideEar (q_l, L_i, C');
        bring blank to last interior position of L_i;
        MoveAgentInSubgraph (q_l, x_0, ⋃_{j=1}^{i-1} L_j);
    push q_l inside L_i;
```

After applying method *TakeAgentOutsideEar*, q_l is outside $\text{int}(L_i)$ and agents $q_{l+1}, \ldots, q_z$, if any, are inside L_i. This fits Case 1 and the processing continues as in that case. Method *SolveDerivedEar* is shown in pseudocode in Algorithm 1.

Figure 4 shows an example. Node v_1 is the entrance of ear L_i, and node v_5 is the exit. The path $\pi = v_5, v_6, v_7, v_4, v_1$ is a path from the exit to the entrance of L_i. The edge (v_1, v_2), also labeled as (n, m) in the figure, is an escape door for the path π. Path π together with L_i form a simple cycle C'. Assume that agents q_8, q_9, q_{10} have their goals at nodes v_8, v_9, v_{10} respectively. Assume further that q_{10} and q_9 have already been pushed inside L_i (being currently located at v_9 and v_8 respectively), and that at this iteration we handle agent q_8. Agent q_8 is also inside L_i, but in a wrong position, namely v_{10}. We push forward all agents within C' until q_8 reaches vertex v_1, the source vertex of the escape door (v_1, v_2). Then, q_8 is pushed along the escape door to v_2. Then we keep rotating agents within the cycle until q_{10} and q_9 reach again their initial positions (v_9 and v_8). We are now in Case 1, with agent q_8 being outside the ear L_i.

Proposition 7. *Solving a derived ear L_i never touches the interiors of previously solved ears $L_{i+1}, \ldots, L_r$.*

Indeed, all moves considered earlier in this section are limited to the sub-graph $D_i = \bigcup_{j \leq i} \text{int}(L_j)$.

Proposition 8. *The complexity of solving all derived ears is within $\mathcal{O}(|V|^3)$, and so is the number of moves generated.*

Proof. When solving a derived ear L_i, lines 5–11 in Algorithm 1 have a complexity of $\mathcal{O}(|V|^2)$. Indeed, line 8 has a quadratic complexity, whereas moving the blank around (line 9) has a linear complexity in the number of nodes. Line 10 has a quadratic complexity, as we will show in Appendix B.2. It follows that the loop on lines 4–11 has a complexity of $\mathcal{O}(|L_i| \times |V|^2)$. Adding this up for all derived ears sums up to $\mathcal{O}(|V|^3)$. □

6.4 Solving the Basic Cycle

After all derived ears have been solved, it remains to solve the basic cycle L_0.[6] When ordering agents inside L_0, we make use of a non-trivial derived ear L with both ends connected to L_0. Such an ear L always exists in a regular decomposition. For simplicity, and with no loss of generality, we can assume that $L = L_1$.[7] All agents belonging to interior positions of L_1 are already solved, as described earlier. Let $q_1, \ldots, q_z$ be these agents in the order they are arranged on their goal positions in the interior of the ear L_1, starting from the agent closer to the entrance. Recall that (at least) two blanks are available in L_0.

Given a cycle, we introduce two macro-moves called a *pseudo-reverse global step* and a *pseudo-reverse individual step*.

Definition 10. *Let C be a cycle with at least one blank.*

- *A* pseudo-reverse global step *is a series of moves ahead in the cycle until all agents end up one position behind their starting position.*

6. Solving the basic cycle is the part which is different from the diBOX variant presented in the original paper (Botea & Surynek, 2015). As explained in the introduction, the revised version ensures an overall complexity of the diBOX algorithm within $\mathcal{O}(n^3)$, as proved later in Proposition 12.
7. Indeed, by construction, L_1 must have both its endpoints on L_0. With no loss of generality, we can assume that there are no trivial derived ears, according to Remark 1.

- *Let q be an agent right in front of a blank. A* pseudo-reverse individual step *is a series of moves ahead in the cycle until agent q ends up one position behind (i.e., on the former position of the blank), and all other agents end up in their starting positions.*

That is, the resulting position of a pseudo-reverse global step is the same as if agents were capable of moving one step backwards, in the opposite direction of the edges (hence the name of the macro). The resulting position of a pseudo-reverse individual step is the same as if agent q were able to travel one step in the opposite direction of the edge at hand.

Observe that these macros are particular cases of the routine *ShiftAgentsInCycle*, introduced in Section 5. The worst-case number of moves is within $\mathcal{O}(|C|^2)$ for both variations of pseudo-reverse steps.

In solving the basic cycle, assume that two agents u (Mickey) and v (Minnie) need to be brought next to each other in L_0, in the order u, v in the direction of L_0. This process, described in detail below, is referred to as method *BringAgentsTogether*. This is split into several stages: Mickey's departure, Mickey's admission into the ride, Minnie's departure, the joint ride, and the cleanup. These are described in detail below, and illustrated in Figure 5.

In Figure 5 we show a sequence of seven distinct configurations, with arrows, numbered from 1 to 6, between any two consecutive configurations. Each of the arrows numbered from 1 to 6 represents part of the processing performed within the method *BringAgentsTogether*. For example, arrow 1 is Mickey's departure, which is discussed in more detail below. The meanings of all arrows will be explained below. Given an arrow, the configuration before it in the sequence is the configuration before performing the processing represented by the arrow. The configuration after the arrow is the resulting configuration.

Mickey's departure. The objective of this step is to have agent u at the entrance of the ear L_1, and a blank at the exit of L_1. This is achieved by rotating agents in the basic cycle L_0 until this condition is achieved. In Figure 5, arrow 1 illustrates Mickey's departure, as mentioned earlier.

Mickey's admission into the ride. The objective of this step is to temporarily place agent u inside L_1. To obtain this, push all agents in L_1 one step further, so that agent u is on the first interior position of L_1. If L_1 has had an agent q_z on its last interior position, a side effect is that q_z is now in L_0. Arrow 2 in Figure 5 represents Mickey's admission into the ride. In this example, q_z is agent 4.

Minnie's departure. Rotate agents in the basic cycle L_0 until v (Minnie) is at the entrance of the ear L_1, and a blank is available at the exit. Notice that, at the end of this step, Minnie is right behind Mickey in the cycle C' containing L_1 plus the corresponding part of L_0 that completes the cycle. Arrow 3 in Figure 5 illustrates Minnie's departure.

The joint ride. The objective of this step is to have both Minnie and Mickey inside L_0, next to each other, in the desired order. To complete this step, perform a pseudo-reverse global step in cycle C'. Now Mickey is right at the entrance of L_1 and, obviously, Minnie is still right behind Mickey. Notice that now the last interior position of L_1 is empty, as we had a blank at the exit of L_1 at the beginning of this step. See arrow 4 in Figure 5.

Cleanup. Here we eliminate the side effect mentioned earlier. In other words, we put q_z back inside L_1 on its goal position. First, push forward agents within L_0 until q_z is at the exit of L_1. This is illustrated by arrow 5 in Figure 5. Then, perform a pseudo-reverse individual step along the cycle

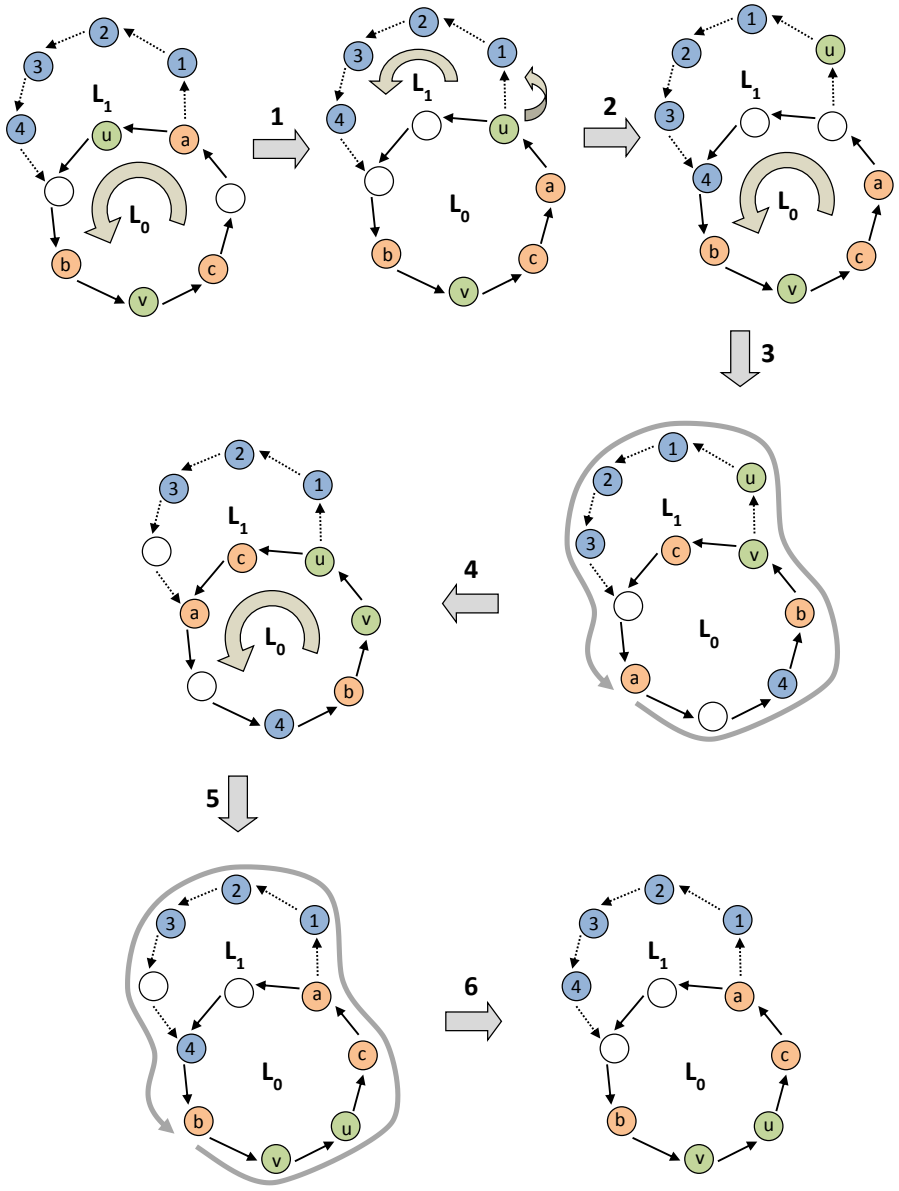

Figure 5: Method *BringAgentsTogether*: bringing u (Mickey) next to v (Minnie) as part of solving the basic cycle. In the graph, solid edges belong to the basic cycle L_0 and dotted edges belong to the derived ear L_1.

C' to place q_z onto the last interior position of L_1. This restores L_1's goal configuration. See arrow 6 in Figure 5.

After applying method *BringAgentsTogether*, L_1 has preserved its goal configuration. Agents u and v are next to each other in the desired order. Apart from repositioning u, no other ordering relationships were affected inside L_0. To see this in Figure 5, compare the first and the last configurations in the sequence.

The process is repeated, each time with a different pair (u, v), until all agents in the basic cycle L_0 are ordered as required in the goal configuration.

Algorithm 2 sketches the pseudocode of the method for solving the basic cycle. In the pseudocode, the function *NextAgent* tells in constant time what agent should come next to a given agent u in the goal configuration. At the end (line 8), after all agents are properly ordered in L_0, they are pushed forward within L_0 until they all reach their goal positions. This is performed with the method *ShiftAgentsInCycle*.

Algorithm 2: SOLVEBASICCYCLE

Input: a basic cycle L_0, an ear L_1, a goal configuration α_+ of agents over D
Output: sequence of moves bringing agents in L_0 to their goal positions

```
1 let L_0 = [x_1, x_2, ..., x_|L_0|];
2 let L_1 be connected to L_0 in x_a and x_b;
3 for l = 1 to |L_0| − 1 do
4 |   v ← α^{-1}(x_l);
5 |   if v ≠ blank then
6 |   |   u ← NextAgent (x_l, L_0, α_+);
7 |   |   BringAgentsTogether (L_0, L_1, u, v);
8 ShiftAgentsInCycle (L_0, α_+);
```

Proposition 9. *The complexity of solving the basic cycle is within $\mathcal{O}(|V|^3)$, and so is the number of generated moves.*

Proof. The method *BringAgentsTogether* involves the following parts: Mickey's departure, with a quadratic complexity on $|L_0|$; admitting Mickey into the ride, which is linear in $|L_1|$; Minnie's departure, with a quadratic complexity in $|L_0|$; the joint ride, with a quadratic complexity in $|L_0| + |L_1|$; and the cleanup, with a quadratic complexity in $|L_0| + |L_1|$. It follows that the method *BringAgentsTogether* has a complexity of $\mathcal{O}(|V|^2)$. The method is invoked no more than $|L_0|$ times, resulting in an overall complexity of $\mathcal{O}(|V|^3)$. □

The solving strategy outlined in this section allows us to formulate the following result.

Proposition 10. *On a strongly biconnected digraph with a regular open ear decomposition, all multi-agent path finding instances with two or more blanks have a solution.*

The condition of having at least two blanks is tight in the following sense:

Proposition 11. *Instances exist such that: the graph is a strongly biconnected digraph with a regular open ear decomposition; there is only one blank available; and the instance has no solution.*

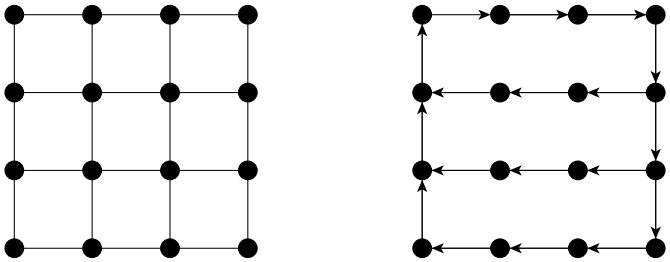

Figure 6: Obtaining a strongly biconnected digraph from the undirected graph of a 15 puzzle.

Proof. We can construct an example starting from the well-known 4×4 sliding-tile puzzle, also known as the 15 puzzle. It is known that the puzzle has instances with no solution (Johnson & Story, 1879). Given such an instance, we construct a multi-agent path finding instance on a strongly biconnected digraph. All we have to do is to modify the original undirected graph by removing some edges, and assigning directions to the remaining edges. In Figure 6 we show the original undirected graph (left) and the resulting directed graph (right). Observe that this is a strongly biconnected digraph, as it accepts a regular open ear decomposition. For example, the top two rows form the basic cycle, the third row is the interior of a derived ear L_1, and the fourth row is the interior of another derived ear, L_2.

As the only modifications performed boil down to imposing additional movement restrictions, we cannot possibly introduce new solutions. It follows that our multi-agent path finding instance has no solution. □

In summary, a strongly biconnected digraph either is a partially-bidirectional cycle, or it admits a regular open ear decomposition. On partially-bidirectional cycles, all instances with at least one blank can be solved or proven unsolvable. On digraphs with a regular open ear decomposition, all instances with at least two blanks have a solution, and they are solved with our presented approach.

7. Moving One Agent in a Strongly Biconnected Digraph

Our diBOX algorithm includes the method *MoveAgentInSubgraph*, introduced in Section 6. This method takes an agent from a current node to a given destination, being free to shuffle the other agents in the process. In this method, the other agents play the role of mobile obstacles, which can be moved in any desired way.

Thus, the problem addressed with *MoveAgentInSubgraph* can be stated formally as follows. In a strongly biconnected digraph, there is one agent, zero or more mobile obstacles, and at least one blank. Moves refer to relocating a mobile unit (agent or mobile obstacle) to an adjacent node, if that adjacent node is currently empty. The task is to get the agent into a given target position. The final locations of the mobile obstacles are irrelevant.

Wu and Grumbach (2010) studied the feasibility of this problem. With their Theorem 14 they proved that every instance with at least one blank in use has a solution. In the proof to their theorem, these authors outline a recursive strategy to (suboptimally) solve such instances.

We implemented a version of *MoveAgentInSubgraph*, based on Wu and Grumbach's (2010) proof. This involves multiple (e.g., 2) recursive calls of a method within its body, suggesting that

the number of recursive calls could possibly grow exponentially in the worst case. For this reason, we introduce a second version, with a few easy-to-perform modifications that allow to keep the complexity within $\mathcal{O}(|V|^2)$. See details in Appendix B.

Observe that, in the particular case where exactly one blank is available, the number of states in the search space is quadratic in the number of digraph nodes, since a state is determined by the position of the agent and the position of the blank. Assuming further that the digraph is sparse (and thus search space is a sparse graph), an optimal solution (i.e., minimizing the number of agent moves + obstacle moves) could be computed in a time linear in the size of the search space (i.e., quadratic in $|V|$), using for example breadth-first search. However, as the number of blanks increases, the state space quickly blows up. Optimally solving single-agent path finding with mobile obstacles on strongly biconnected digraphs is beyond the focus of this paper.

8. Putting the Pieces Together: the diBOX Algorithm

We put together all pieces of our approach, described in the previous sections, obtaining an algorithm that we call diBOX. We show its pseudo-code and analyze its complexity.

Algorithm 3: Main diBOX algorithm in pseudocode.

Input: a digraph $D = (V, E)$; a set of agents A; an initial configuration α_0 of agents over D; a goal configuration α_+ of agents over D
Output: sequence of moves transforming α_0 into α_+

```
1  diBOX (D, A, α_0, α_+)
2  if D is a partially-bidirectional cycle then
3      if α_0 and α_+ represent the same ordering in D then
4          ShiftAgentsInCycle (D, α_+);
5      else
6          return UNSOLVABLE;
7  else
8      let [L_0, L_1, ..., L_r] be a regular open ear decomp. with non-trivial ear L_1 attached to L_0;
9      if |{x ∈ L_0 | α_+^{-1}(x) = blank}| < 2 then
10         α'_+ ← BorrowBlanks (D, L_0, α_+);
11         SolveEars (D, [L_0, L_1, ..., L_r], α_0, α'_+);
12         ReturnBlanks (D, α_+, α'_+);
13     else
14         SolveEars (D, [L_0, L_1, ..., L_r], α_0, α_+);
```

Before invoking diBOX on a given MAPF instance, we assume that the following information is available due to a preprocessing step: i) whether the digraph is strongly biconnected; ii) whether the digraph is a partially-bidirectional cycle; iii) when the digraph is strongly biconnected, but not a partially-bidirectional cycle, have a regular open ear decomposition available. These are properties that depend on the input digraph, but not on details specific to a given instance, such as the initial and the goal positions of the agents. Thus, preprocessing can be performed once and re-used in solving many MAPF instances on a given digraph.

Algorithm 3 shows diBOX in pseudocode. For simplicity, we assume that the input graph is a strongly biconnected digraph. The test on line 2 tells whether the graph is a partially-bidirectional cycle, or a graph with a regular open ear decomposition. Lines 3–6 and 8–14 handle each of these cases respectively. Recall that the method *ShiftAgentsInCycle* on line 4 will rotate agents within a simple cycle until reaching the desired configuration. Method *SolveEars* is shown in Algorithm 4. As mentioned previously, ears are solved in reverse order, leaving the basic cycle to the end. The pseudocode of the methods *SolveDerivedEar* and *SolveBasicCycle* has been presented in Algorithms 1 and 2 respectively.

Algorithm 4: SOLVEEARS

Input: a digraph D, a regular open ear decomposition $[L_0, L_1, \ldots, L_r]$, an initial configuration α_0 of agents over D, a goal configuration α_+ of agents over D
Output: sequence of moves bringing agents in $[L_0, \ldots, L_r]$ to their goal positions

```
1 α ← α_0;
2 for i = r down to 1 do
3 |   SolveDerivedEar (D, L_i, α_+);
4 SolveBasicCycle (L_0, L_1, α_+);
```

Proposition 12. *The worst-case time complexity of diBOX is within $\mathcal{O}(|V|^3)$ and so is the number of moves. The property holds even if the preprocessing is done once per instance and the preprocessing costs are included in the diBOX costs.*

Proof. Considering first the steps without preprocessing, we show that, in Algorithm 3, both the time complexity and the number of moves are dominated by the method *SolveEars*. Specifically, *SolveEars* has both the time and the number of moves within $\mathcal{O}(|V|^3)$, according to Propositions 8 and 9. At the same time, none of the other parts of Algorithm 3 exceeds $\mathcal{O}(|V|^2)$. Rotating agents within a cycle C to reach a desired configuration (*ShiftAgentsInCycle*) consumes a time of $\mathcal{O}(|C|^2)$, according to Proposition 4. *BorrowBlanks* and *ReturnBlanks* do not exceed a time of $\mathcal{O}(|E|)$, producing $\mathcal{O}(|V|)$ moves.

In the preprocessing, the test used on line 2 of Algorithm 3 is performed within $\mathcal{O}(|V|)$ time steps, according to Proposition 2. Testing whether a digraph is strongly biconnected, and computing a (regular) open ear decomposition can be performed within $\mathcal{O}(|V|(|V| + |E|))$, according to Corollary 4.

Overall, both the worst-case time complexity and the number of moves for diBOX are within $\mathcal{O}(|V|^3)$. □

9. Computing Open Ear Decompositions

Algorithm 5 outlines a strategy for computing open ear decompositions. For simplicity, we assume that the basic cycle L_0 is given as input. This does not restrict the generality, as the decomposition can start from any basic cycle, according to Theorem 1.

Each iteration of the main loop adds a new *valid* ear, until all nodes are covered, or returns with a failure. In this section, we say that an ear is valid if it is a non-trivial derived ear with its two endpoints different from each other. In other words, an ear is valid if it can be included in an open ear decomposition as a non-trivial derived ear.

Algorithm 5: EARDECOMPOSITION

Input: a digraph D, a basic cycle L_0 in D
Output: open ear decomposition O, if one exists

```
1 O ← [L_0]; /* initialize O to a partial decomp. with just L_0 */
2 do
3 |   e ← CreateEar(D, O);
4 |   if a non-trivial derived ear e was found then
5 |   |   Append e to the decomposition O;
6 |   else
7 |   |   return UNSOLVABLE;
8 while D has nodes not included in O;
9 return O;
```

Theorem 2. *Let $D = (V, E)$ be a digraph and O an arbitrary partial open ear decomposition (i.e., a basic cycle plus zero or more valid ears, covering a subset of D's nodes). Computing a new* shortest *valid ear, or showing that no new valid ear exists, can be done within a time complexity of $\mathcal{O}(|V| + |E|)$.*

The proof is available in Appendix C.

Remark 2. *Let $D = (V, E)$ be a digraph and O an arbitrary partial, incomplete*[8] *open ear decomposition. If no new non-trivial derived ear with different endpoints exists, then the graph is not strongly biconnected.*

Proof sketch: Consider a node n that does not belong to O, but is a successor of a node p in O. If n belongs to no derived ear with different endpoints, it follows that either i) there is no path from n to any node in O; or ii) p is the only parent of n in O and all paths from n to a node in O pass through p. The first case implies that D is not strongly connected, and case ii) implies that p is a cut vertex.

Corollary 4. *Given a digraph D, finding an open ear decomposition, or showing that no decomposition exists, can be done within a time complexity of $\mathcal{O}(|V|(|V| + |E|))$. The result holds even when seeking decompositions with a shortest ear added at each iteration.*

The results shown so far in this section were presented for the case of finding an open ear decomposition. Recall that an open ear decomposition is *regular* if the basic cycle has at least three nodes, and there is at least one non-trivial derived ear. To search for a regular decomposition, start from a basic cycle that has at least three nodes, but it does not contain all the nodes of the digraph.

As it has been shown in the case of the BIBOX algorithm, on undirected graphs, different ear decompositions have a great impact on the performance of the algorithm (Surynek, Surynková, & Chromý, 2014). It has been shown that using many short ears instead of few long ears leads to a significantly better performance of the BIBOX algorithm. Our hypothesis is that diBOX could be impacted in a similar manner by the length of the ears in the decomposition.

For this reason, we implemented two techniques (i.e., variants of *CreateEar*). One variant adds a shortest possible ear at each iteration, whereas the other adds a longest possible ear. We use these for

8. I.e., covering some but not all of D's nodes.

evaluation purposes, to test which type of decomposition leads to a better path finding performance in diBOX (not in preprocessing).

To compute a decomposition with short edges, the method *CreateEar* performs breadth-first search. The proof to Theorem 2 shows such an approach.

To compute a decomposition with long ears, depth-first search is employed. For example, to compute a new longest valid ear, run a depth-first search from each node in O. Cycles along a branch are not allowed, but previously visited nodes may be visited again on a different branch. Nodes from O are not expanded at a depth greater than 0. Searching for a longest ear has an increased worst-case complexity, compared to searching for a shortest ear (i.e., higher complexity than shown in Theorem 2). However, the purpose of decompositions with long ears is not to be used in a "production" system, but to be used as a benchmark in our empirical evaluation. Moreover, we can also regard the computation of an ear decomposition as an offline process that produces the decomposition to be reused in many runs of the diBOX algorithm with various initial and goal configuration of agents.

10. Empirical Evaluation

In this section we evaluate the performance of diBOX on a range of problems. We start with a comparison to an optimal solver, followed by a more detailed analysis of the impact of various components of diBOX on its performance. The algorithms are implemented in C++. The experiments are performed on an Intel 1.60 GHz machine, running Ubuntu 12.04 64 bit.

10.1 Comparison to Optimal Solutions

We tested the diBOX algorithm against MDD-SAT (Surynek et al., 2016), an existing optimal solver based on propositional satisfiability (SAT). Originally, MDD-SAT produces sum-of-costs optimal solutions, counting each move and each waiting action at a position different from the agent's goal. In our evaluation, MDD-SAT has been modified to generate optimal solutions in terms of the number of moves.

Two sets of instances were used in our comparison to optimal solutions. Instances are relatively small, so that an optimal solver can succeed at least in part of the instances. The first problem set uses a digraph with 20 vertices. The basic cycle consists of 5 vertices. Derived ears vary in size from 2 to 5 internal vertices. The second set of instances uses a digraph consisting of 40 vertices. The basic cycle has 5 vertices, and derived ears have 2 to 8 internal vertices each. The number of agents in an instance is gradually increased. For each number of agents we generated 10 instances with the initial and the goal configurations set at random. The average number of moves and the average runtimes are shown in Figures 7 and 8.

In terms of scalability, in both problem sets, on instances with 11 agents or more, MDD-SAT reaches a timeout of 300 seconds. On the other hand, the diBOX algorithm scales much better than the optimal solver. diBOX solves all instances in this experiment quite easily (e.g., less than 1 second for the 20-node instances). In terms of quality, solutions computed with diBOX range from nearly optimal to a deviation from optimal values by at most a factor of 4.5.

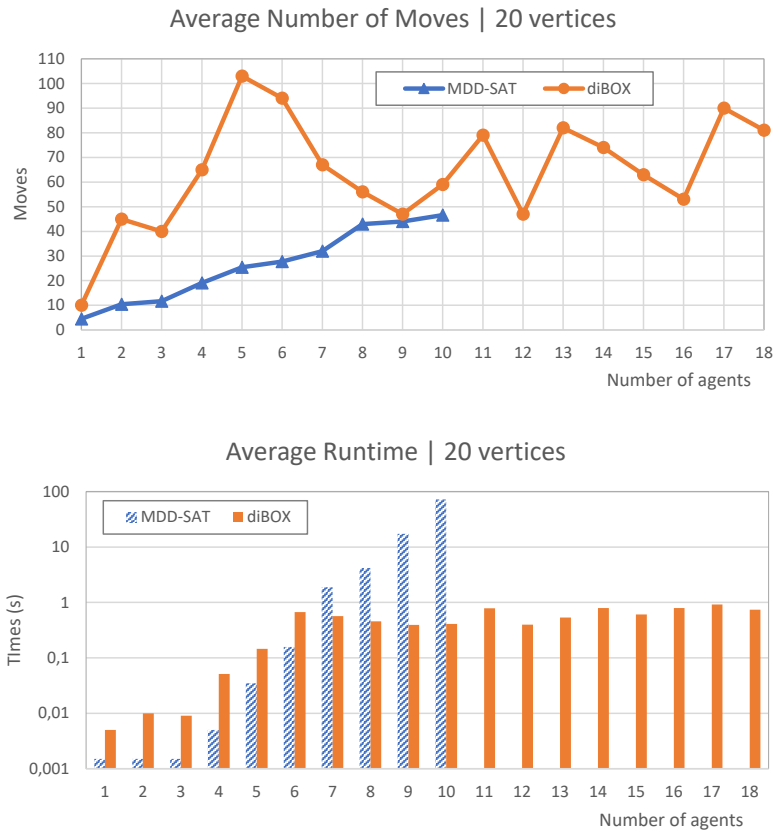

Figure 7: diBOX vs MDD-SAT on instances with 20 vertices. Average values across 10 random instances per number of agents are plotted.

10.2 Peformance on Densely-Populated Instances

In this section we evaluate the diBOX algorithm on difficult instances with only two blanks available in an instance. Our graphs range in size from 80 to 200 nodes, with an increment of 10 nodes from one graph size to the next. For each graph size, we generate three graphs, and for each graph we generate 10 instances with the initial and the goal configurations set at random. In all cases, there are only two empty graph nodes (blanks), and all other nodes are occupied by one agent each. Such instances are too hard for optimal solvers.

Our evaluation presented in this section focuses on two major aspects: how the algorithm scales as the graph size increases, and what is the impact of open ear decomposition on the performance of diBOX. To perform this, we run diBOX with each of the two ear decomposition strategies discussed in Section 9.

We now discuss the performance of the two versions of our solver. Version S aims at graph decompositions with short ears, whereas version L favors decompositions with fewer but longer

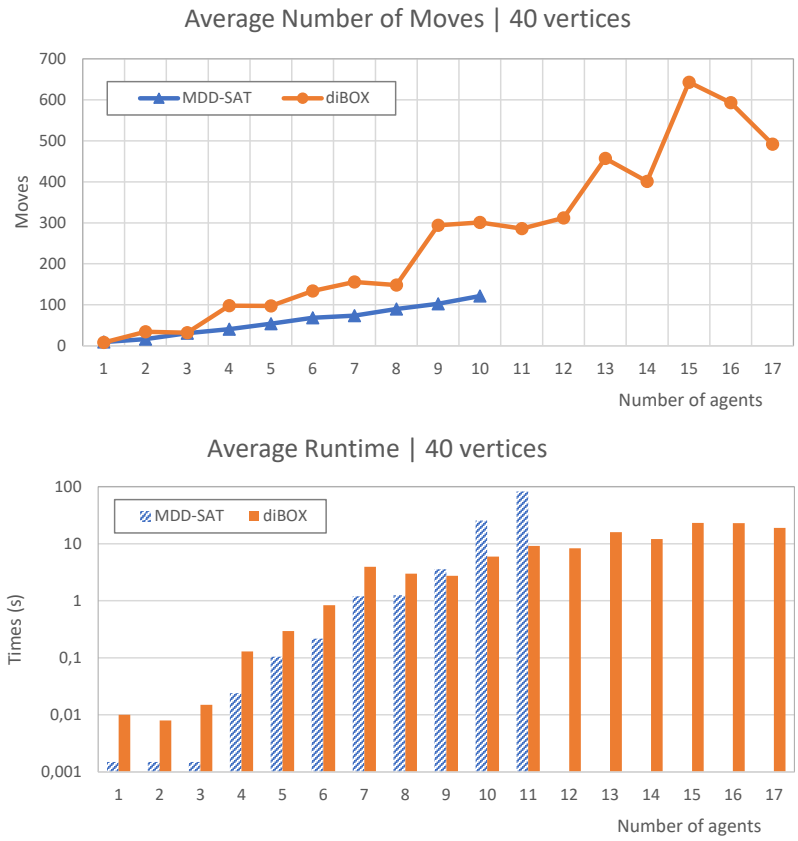

Figure 8: diBOX vs MDD-SAT on instances with 40 vertices. Average values across 10 random instances per number of agents are plotted.

ears. Table 1 shows average key statistics about the graph decomposition, such as the number of ears, the number of nodes of the biggest ear and the number of nodes of the smallest ear.

As expected, version L is significantly more costly as compared to version S. In fact, version L does not scale beyond graphs with 120 nodes, given a timeout of 30 minutes per instance.

Figure 9 and Figure 10 show average solution length data and average CPU time data, respectively. Version S performs better, both in terms of solution quality and CPU time. It generates significantly fewer moves than version L. The reason is that a graph with shorter ears shows a higher connectivity, which implies the availability of shorter paths along which agents have to move. Shorter ears require fewer moves in the algorithm's sub-routines, such as *MoveAgentInSubgraph* and *TakeAgentOutsideEar*. See an evaluation of method *MoveAgentInSubgraph* in Section 10.4.

Recall that diBOX starts with solving the derived ears, after which it solves the basic cycle. Figures 9 and 10 have shown the overall performance, when solving the entire instance (derived ears and basic cycle). For a more detailed presentation, we also show the performance corresponding to solving derived ears (Figures 11 and 12) and solving the basic cycle (Figures 13 and 14). In all

Graph size (# nodes)	80		90		100		110		120		130		140	
Type of ear decomp.	S	L	S	L	S	L	S	L	S	L	S	L	S	L
Number of ears	19	12	19	12	20	13	20	14	22	15	22		26	
Max ear size	11	32	12	30	17	38	22	49	22	49	22		22	
Min ear size	3	3	3	3	3	3	3	3	3	3	3		3	

Graph size (# nodes)	150		160		170		180		190		200			
Type of ear decomp.	S	L	S	L	S	L	S	L	S	L	S	L		
Number of ears	26		26		30		31		31		32			
Max ear size	24		26		26		29		29		31			
Min ear size	3		3		3		3		3		3			

Table 1: Average key statistics on the graph decomposition strategy. S = short-ear version; L = long-ear version.

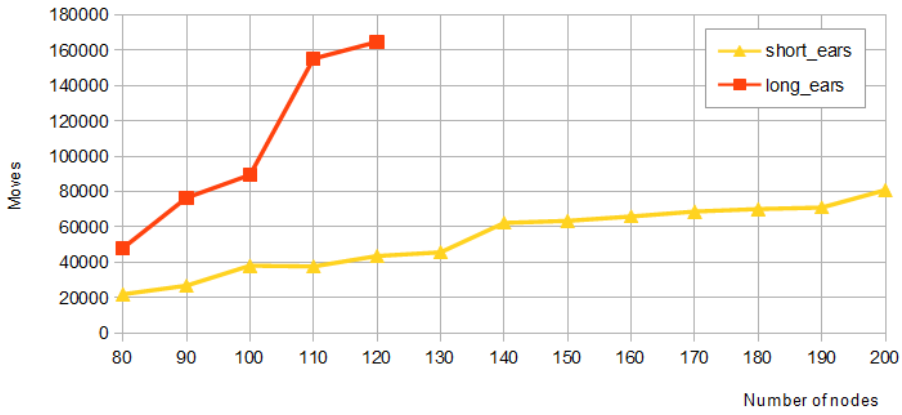

Figure 9: Average number of moves per instance.

cases, version S scales significantly better, and it produces shorter solutions as well. We conclude that decompositions with short ears should be preferred in practice.

For the version S, all figures show a good scalability of the algorithm in practice. Specifically, as the graph size increases, the increase in the solving time and the solution length is much slower than the cubic upper bounds shown in our worst-case formal analysis.

10.3 Varying the Agents-to-Vertices Ratio

In this section we evaluate diBOX on instances where the ratio between the number of agents and the number of graph nodes varies. We focus on graphs of two sizes: 80 nodes and 120 nodes, respectively. For the 80-node graph, we generate instances with $10, 20, \ldots, 70$ and 78 agents. For the 120-node graph, the number of agents is set to $10, 20, \ldots, 110$ and 118, respectively. For each combination of a graph and a number of agents, 10 random instances are generated.

Average runtime and solution-length results are shown in Figure 15 and 16 respectively. As in the case of densely populated instances, these results show a clear advantage of using short-ear

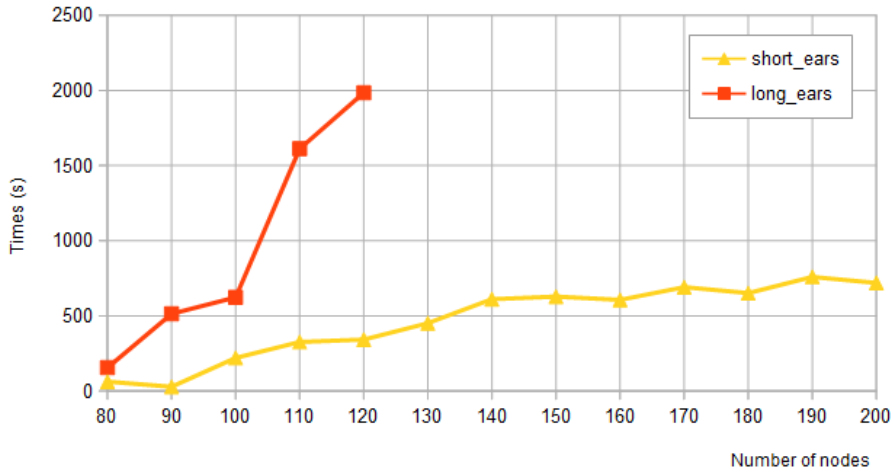

Figure 10: Average CPU time per instance.

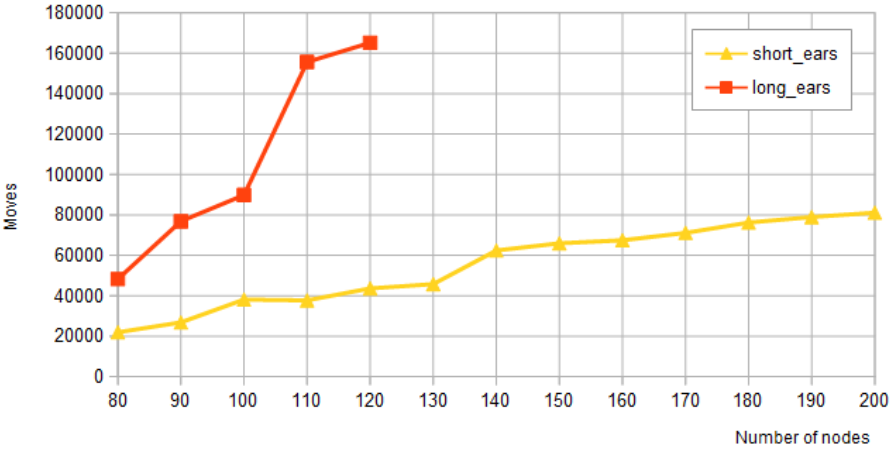

Figure 11: Average number of moves per instance to solve all derived ears, but not the basic cycle.

decompositions. Decompositions with short ears result in a better scalability, whereas long ears can exhibit a sharp increase in the solving time and solution length.

10.4 Moving One Agent in a Subgraph

Moving one agent within a subgraph, while treating all other agents in the subgraph as mobile obstacles, is an important component of our solving approach (method *MoveAgentInSubgraph*). Section 7 and Appendix B focus on how to solve this problem. In this section, we present an empirical evaluation, considering instances with only one blank available. We used the same set of input graphs as in Section 10.2: The graph size ranges from 80 to 200 nodes, with 10-node increments. There are 3 graphs for each size. For each graph, we generated 50 instances, with the

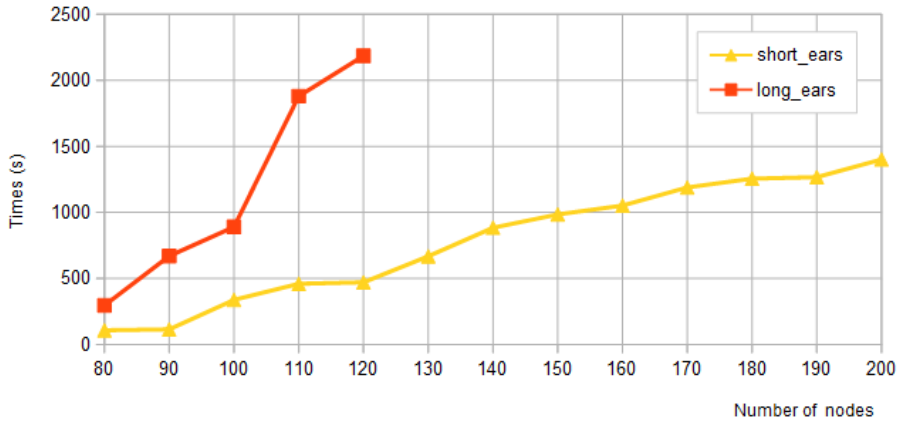

Figure 12: Average CPU time per instance to solve all derived ears, but not the basic cycle.

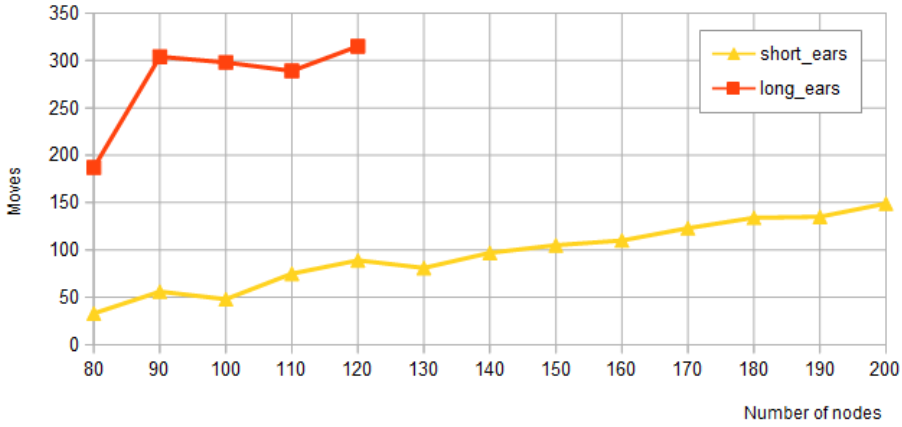

Figure 13: Average number of moves per instance to solve the basic cycle.

initial and the goal configurations picked at random, for a total of 1,050 instances. In an instance with n graph nodes, there are $n - 2$ mobile obstacles, one agent and one blank. Our evaluation focuses on three major aspects: scalability with graph size, the impact of choosing the basic cycle to contain the target or not, and the impact of sizes of ears. Four variants of the *MoveAgentInSubgraph* are run on every instance, resulting from two strategies on ear size (long or short ears), combined with the two ways of selecting the basic cycle (chosen to contain the target or not). Having the target outside the basic cycle breaks the conditions that ensure the quadratic complexity discussed in Section 7 and Appendix B.

Figure 17 summarizes the number of moves averaged per graph size. The decomposition that aims at building shorter ears leads to better results. This confirms our expectations.

Interestingly, comparing the top and bottom charts in Figure 17 shows that selecting the basic cycle in such a way that it does not contain the target works well. This implies that the 2 recursive calls in a row did not cause a performance bottleneck in our experiments. In fact, the case with the

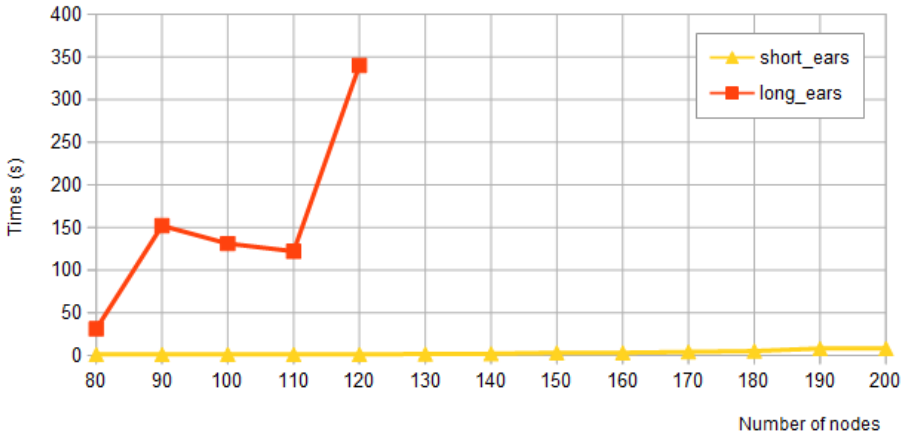

Figure 14: Average CPU time per instance to solve the basic cycle.

basic cycle containing the target has a somewhat weaker performance (e.g., an increase from close to 2,000 to close to 3,000 moves for the short-ear decomposition on 150-node graphs). We attribute this to differences in the average *ear distance* between the start s and the target t. We define the ear distance as as $|k - k'|$, where $s \in \text{int}(L_k)$ and $t \in \text{int}(L_{k'})$. When k and k' are drawn uniformly and independently from the range $0, \ldots, K$, with K being the largest ear index, the expected value of the ear distance is $K/3$. On the other hand, when k' is restricted to be 0, the expected ear distance increases to $K/2$, which in turn can increase the number of moves in a solution.

11. Conclusion

We have performed a formal and empirical study of multi-agent path finding on strongly biconnected digraphs. To our knowledge, this is the first theoretical study focused on multi-agent path planning on any type of directed graphs. We found that instances with at least two blanks have a solution, except for the trivial case of badly ordered instances on partially-bidirectional cycles. We have presented a suboptimal algorithm, called diBOX, whose worst-case time complexity and solution length are both within $\mathcal{O}(|V|^3)$, where V is the set of vertices. Similarly to BIBOX (Surynek, 2009), a method for undirected biconnected graphs, our algorithm solves ears in reverse order. However, the restriction to unidirectional movements makes our algorithm more involved than the case when bi-directional movement is possible.

We implemented the diBOX algorithm and evaluated its performance. As particular ear decompositions have a great impact on the performance of the algorithm, we implemented two strategies to decompose a strongly biconnected digraph into an open ear decomposition. One strategy aims at computing short ears and the other prefers long ones. Results clearly indicate that the preference of short ears is the better option, as it leads to better solutions and CPU times. Overall, the results show a good scalability of the method, in combination with short ear decompositions, successfully solving instances of an increasing size and difficulty. In practice, as the graph size increases, the increase of the CPU time and the solution time is much slower than the upper bounds shown in the worst-case formal analysis. diBOX scales convincingly beyond the limits of an optimal solver. On

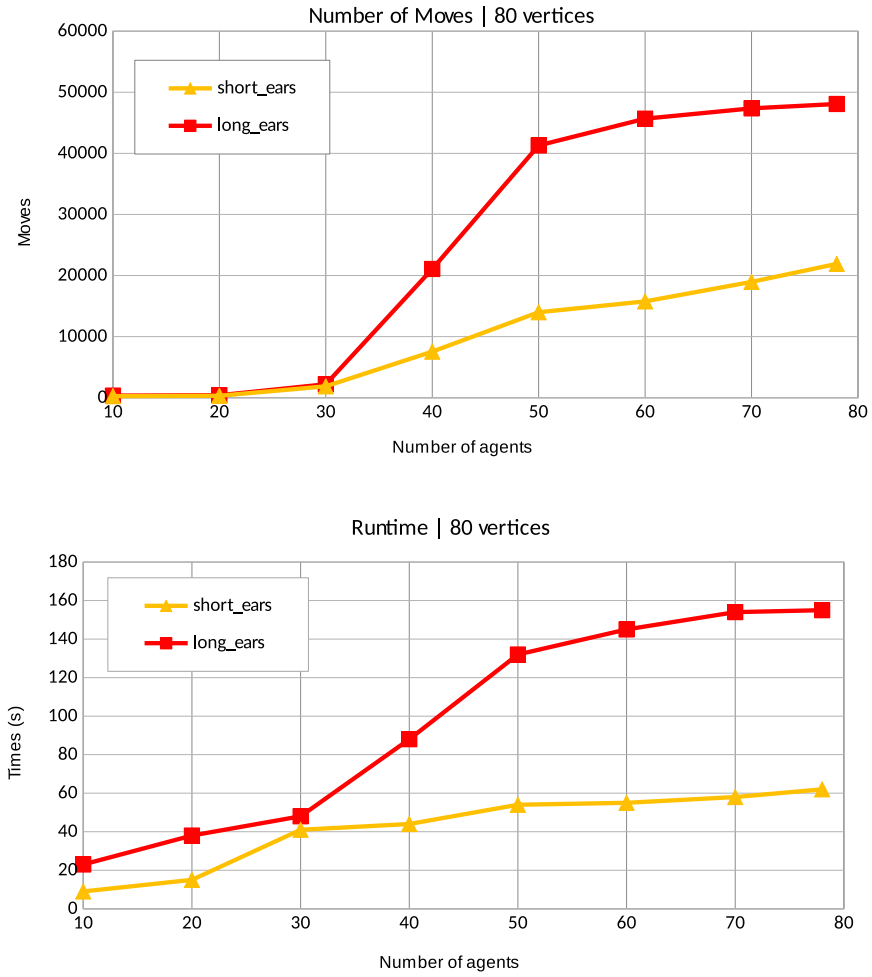

Figure 15: Varying the number of agents on a graph with 80 nodes.

small instances where an optimal solver can succeed, the quality of solutions computed with diBOX stay within a factor of at most 4.5 from optimal values.

In future work, we plan to extend our formal and empirical analysis to other classes of digraphs. In addition, we plan to investigate problems where rotations in a cycle do not require using a blank. Pushing the scalability of the system further is another interesting topic for the future.

Acknowledgment

We thank the editor and the anonymous reviewers for their valuable feedback. We thank Daniel Harabor and Pierre Le Bodic for a discussion that has led to Proposition 11. We are grateful to Akihiro Kishimoto and Inge Vejsbjerg for proof-reading our manuscript.

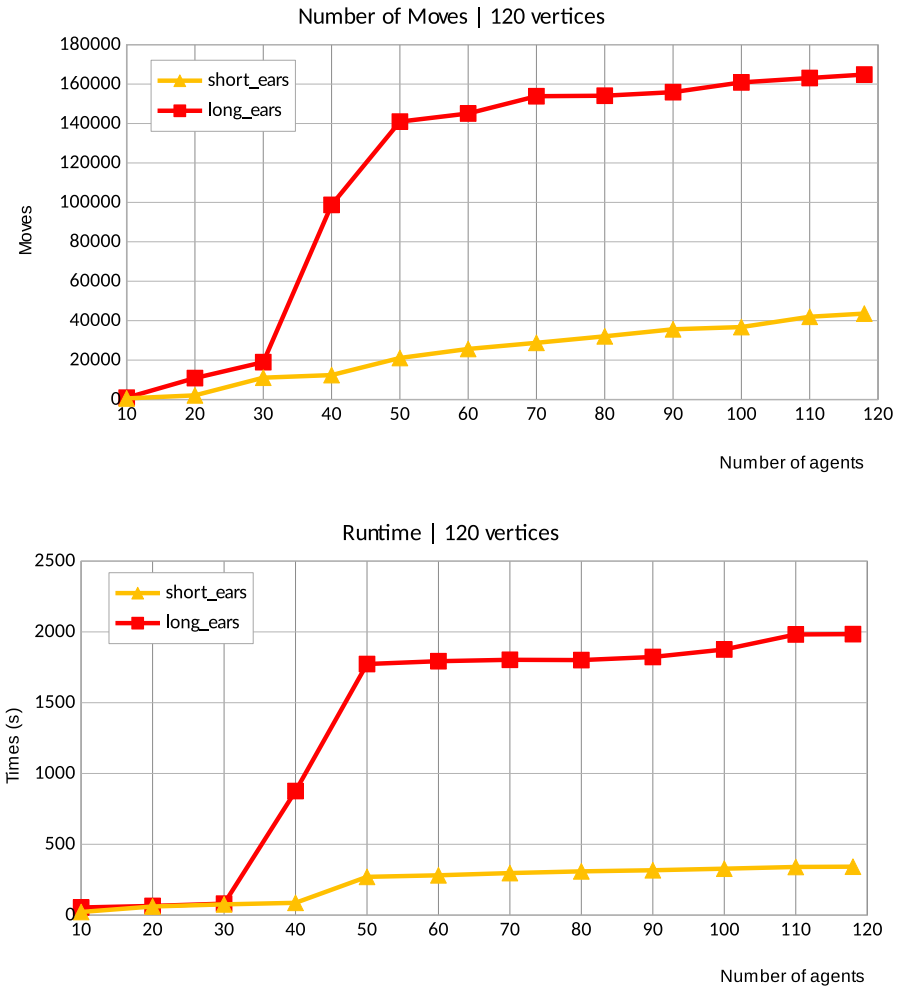

Figure 16: Varying the number of agents on a graph with 120 nodes.

Appendix A. Proofs to Propositions 1 and 2

Lemma 2. *A strongly biconnected digraph D with three or more nodes admits an open ear decomposition with the basic cycle having at least three nodes.*

Proof. As D is strongly biconnected, an open ear decomposition $[L_0, \ldots, L_r]$ exists. Unless L_0 already has three or more vertices, we modify the open ear decomposition slightly to obtain a basic cycle with that property. Indeed, when L_0 has two vertices, there must be more ears than just L_0 in the decomposition, since the graph has 3 or more vertices. L_1 is a derived non-trivial[9] ear in $[L_0, \ldots, L_r]$.[10] Ear L_1, together with one edge from L_0, become the new basic cycle, L'_0,

9. Assuming by contradiction that L_1 is trivial, it follows that L_1 is a duplicate of an edge included in the 2-node basic cycle L_0.
10. L_1 is also acyclic, as it belongs to an open ear decomposition.

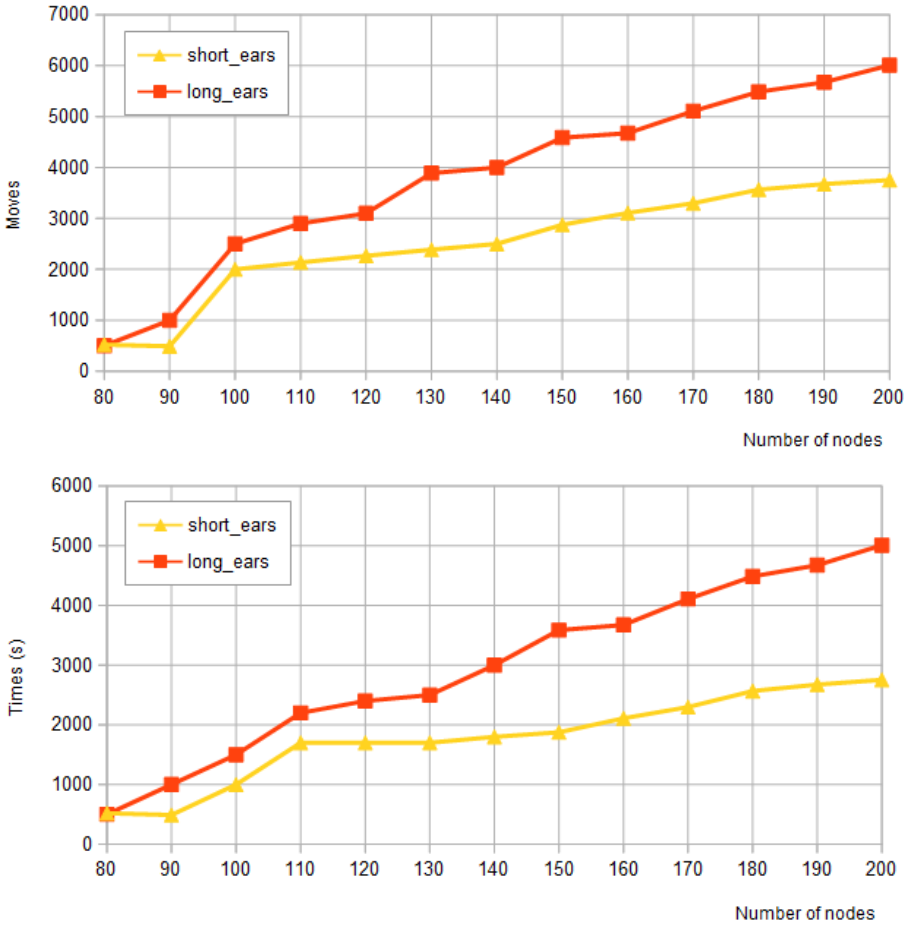

Figure 17: Average solution length for moving one agent within a subgraph. Top: target in basic cycle. Bottom: target elsewhere.

in the decomposition. The other edge from L_0 becomes a trivial derived ear, L'_1. Observe that $[L'_0, L'_1, L_2, \ldots, L_r]$ remains an open ear decomposition. □

Proposition 1. *For every strongly biconnected digraph $D = (V, E)$, exactly one of the following two cases holds:*

1. *D has a regular open ear decomposition; or*
2. *D is a partially-bidirectional cycle.*

Proof. Part A: Showing that at least one case holds. We start by noting that any strongly biconnected digraph with three or fewer vertices satisfies case 2. Thus, in the rest of Part A, we assume that $|V| \geq 4$.

As D is strongly biconnected, there is an open ear decomposition $[L_0, \ldots, L_r]$ where L_0 has at least three nodes, according to the previous lemma.

We distinguish between two scenarios. First, we consider the scenario when L_0 contains every vertex in V. If there exists an edge $e \in E$ that connects two vertices p and q that are not next to each other in the basic cycle L_0, then the edge $e = (p, q)$ and part of L_0's edges create a new basic cycle L'_0. The remaining edges of L_0 create a non-trivial derived ear. We are now in case 1 of the proposition. If no such an edge e exists, we are in case 2.

Consider now the scenario when L_0 contains only a subset of the vertices in V. As L_0 does not cover all vertices, and trivial derived ears have no interior nodes, it follows that $[L_0, \ldots, L_r]$ has at least one non-trivial derived ear. In other words, we have a regular open ear decomposition, being in case 1 of the proposition.

Part B: Showing that at most one case holds. This can be proven by contradiction. Assume a given strongly biconnected digraph D is a partially-bidirectional cycle, and it has a regular open ear decomposition. As D is a partially-bidirectional cycle, the only options for the basic cycle L_0 that could occur in an open ear decomposition are: i) L_0 has two vertices; or ii) L_0 contains all vertices. See Figure 3 for an example. Option i) contradicts the requirement that L_0 has at least three vertices. Option ii) contradicts the requirement that there exists at least one non-trivial derived ear. □

Proposition 2. *Let $D = (V, E)$ be a digraph. Checking if D is a partially-bidirectional cycle can be done in $\mathcal{O}(|V|)$ time steps.*

To prove this result, we outline a procedure that performs the test and has the claimed complexity. The intuition with the testing procedure is as follows. Assume that a graph is indeed a partially-bidirectional cycle (see again Figure 3 for an example). If we manage to start a walk in the "forward" direction, and never consider edges in the "backwards" direction, there is one unique way to progress with the walk. Eventually we complete a cycle that contains all nodes of the graph. After the cycle is built, we can verify that all edges not included in the cycle are the reversed version of some edges included in the cycle. On the other hand, if the input graph is not a partially-bidirectional cycle, the walk can sooner or later detect that, as shown in the description of the procedure, presented next.

The procedure works as follows. If D contains a node with 0 successors, or a node with at least three successors, return "no" (not a partially-bidirectional cycle). Otherwise, pick a node n_0 with the smallest number of successors (i.e., either 1 or 2 successors). We distinguish between the following two cases.

Case 1 is when n_0 has exactly one successor n_1. If D is a partially-bidirectional cycle, edge (n_0, n_1) must belong to the cycle, i.e., it must be a forward edge, since there is no other edge leaving from n_0. We perform a walk starting with that edge. In extending a walk with one more node, always ignore edges that would take us back to the previous node. If, after applying this rule, there is exactly one successor left, continue the walk with that successor. Otherwise (0 or two successors) the walk stops and the test fails. If a cycle is eventually built, all is left to check is whether i) the cycle contains all nodes; and ii) all edges not included in the cycle satisfy the definition of a partially-bidirectional cycle, i.e., each of them is the reversed version of an edge in the cycle. If both conditions hold, the answer is positive. Otherwise, the test fails.

Case 2 is when n_0 has two successors. As n_0 is picked to have a minimal out-degree, it follows that all nodes in D have two successors each. The only case when a partially-bidirectional cycle satisfies this condition is when every forward edge is mirrored with a backwards edge. In other words, we have in fact a cycle with all adjacent nodes connected in both directions. Checking whether a digraph has this structure is straightforward, and we skip the details.

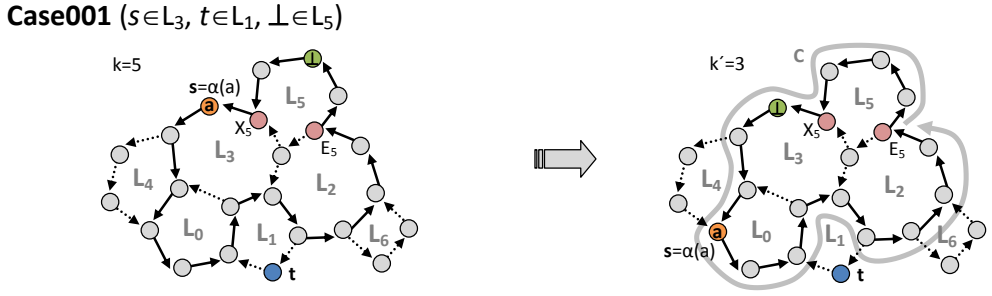

Figure 18: Illustrating Case001. Symbol $\perp$ denotes the blank and "a" is the agent.

There are several slight variations of this procedure, and the one presented is not necessarily the most effective. Our goal was to keep it as simple as possible.

Appendix B. Method *MoveAgentInSubgraph*

We present two algorithmic versions of this method. In Section B.1 we convert Wu and Grumbach's (2010) proof to their Theorem 14 into a detailed algorithm. In Section B.2 we show that, with a few easy-to-implement modifications, the complexity is within $\mathcal{O}(|V|^2)$. Without any loss of generality, in this appendix we ignore all trivial derived ears.

B.1 A Detailed Recursive Procedure

The method *MoveAgentInSubgraph-v1*, where "v1" stands for version 1, shown in Algorithm 6, takes the following input:

- s, the initial position of the agent;
- t, the target position of the agent;
- b, the blank position;
- k, the largest index so that $s \in \text{int}(L_k)$, or $t \in \text{int}(L_k)$, or $b \in \text{int}(L_k)$.

The output is a series of moves taking the agent from s to t within the subgraph $[L_0, \ldots, L_k]$.

This is a recursive method with several cases, depending on the positions of the starting location, the target, and the blank. We identify cases with the name "Case" followed by three symbols (e.g., *Case001*). The first symbol is a boolean flag telling whether $s \in \text{int}(L_k)$. The second symbol is a flag telling whether $t \in \text{int}(L_k)$. For the third symbol, we use several values, which are not necessarily mutually exclusive: 0 indicates that $b \notin L_k$; 1 stands for $b \in L_k$; * is a "don't care symbol"; and "g" denotes a "good" position of the blank, namely $b \in \text{int}(L_k) \cup \{E_k\} \cup \{X_k\}$, where E_k is the entrance of L_k and X_k is its exit.

In the basic-cycle case (line 2 in Algorithm 6), all three nodes s, t and b are in the basic cycle L_0. It is possible to bring the agent from s to t by just rotating the agents within L_0. In *Case001*

Algorithm 6: MOVEAGENTINSUBGRAPH-V1

Input: node s; node t; node b; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$

```
1  if k = 0 then
2  |  BasicCycleCase(s, t, b, k);
3  else if s ∉ int(L_k) ∧ t ∉ int(L_k) ∧ b ∈ int(L_k) then
4  |  Case001(s, t, b, k);
5  else if s ∉ int(L_k) ∧ t ∈ int(L_k) ∧ b ∉ int(L_k) then
6  |  Case010(s, t, b, k);
7  else if s ∉ int(L_k) ∧ t ∈ int(L_k) ∧ b ∈ int(L_k) then
8  |  Case011(s, t, b, k);
9  else if s ∈ int(L_k) ∧ t ∉ int(L_k) then
10 |  if b ∉ int(L_k) then
11 |  |  bring blank to X_k (or E_k);
12 |  Case10g(s, t, k);
13 else if s ∈ int(L_k) ∧ t ∈ int(L_k) then
14 |  if b ∉ int(L_k) then
15 |  |  bring blank to X_k (or E_k);
16 |  Case11g(s, t, k);
```

Algorithm 7: CASE001

Input: node s; node t; node b; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$

```
1 let P be a simple path from X_k to E_k in [L_0, ..., L_{k-1}];
2 let C be the cycle P ∪ L_k;
3 rotate in C until both agent and blank are outside int(L_k);
4 let b' be the new position of blank;
5 let s' be the new position of agent (may or may not be different from s);
6 let k' be the largest index so that s' ∈ int(L_{k'}) ∨ t ∈ int(L_{k'}) ∨ b' ∈ int(L_{k'});
7 MoveAgentInSubgraph-v1(s', t, b', k');
```

(Algorithm 7), the initial and the target positions are not in L_k, and the blank is in L_k. The method first makes sure that the blank is taken out of int(L_k) after which a recursive call with a smaller k' is performed. Figure 18 illustrates the way this method works.

It is important to notice that, when taking the blank out of int(L_k), a side effect could be that the agent is pushed within int(L_k). When this happens, we keep rotating until both the agent and the blank are out of L_k's interior. Optionally, we can require that we stop the rotations in a position where the blank is right in front of the agent. We call this the *blank-agent adjacency*, and its relevance will be clear in Section B.2.

In Algorithm 8, the initial position s is in int(L_k) but the target t is not. Also, at the beginning, the blank is guaranteed to be in a good position (i.e., somewhere along L_k), according to lines 10–11 in Algorithm 6. Method *Case10g* performs rotations in a cycle C containing L_k until both the agent and the blank are out of int(L_k). Once again, we can optionally impose the blank-agent adjacency

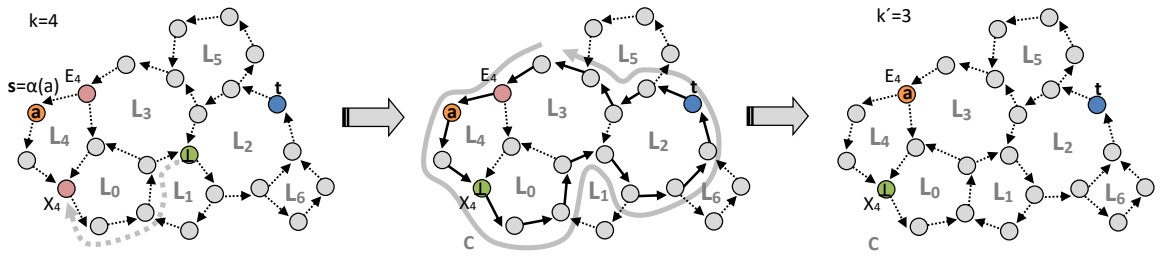

Figure 19: Illustration of Case100.

Algorithm 8: CASE10G

Input: node s; node t; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$

1 **let** P be simple path from X_k to E_k in $[L_0, \ldots, L_{k-1}]$;
2 **let** C be the cycle $P \cup L_k$;
3 rotate in C until agent reaches some node $s' \notin \text{int}(L_k)$ and blank reaches some node $b' \notin \text{int}(L_k)$;
4 **let** $k' < k$ be the largest index so that $s' \in \text{int}(L_{k'}) \vee t \in \text{int}(L_{k'}) \vee b' \in \text{int}(L_{k'})$;
5 MoveAgentInSubgraph-v1(s', t, b', k');

condition. Then, a recursive call to *MoveAgentInSubgraph-v1* takes the agent from X_k to the target t. Figures 19 and 20 illustrate cases 100 and 101.

Algorithm 9: CASE010

Input: node s; node t; node b; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$

1 **let** $k' < k$ be the largest index so that $s \in \text{int}(L_{k'}) \vee X_k \in \text{int}(L_{k'}) \vee b \in \text{int}(L_{k'})$;
2 MoveAgentInSubgraph-v1(s, X_k, b, k');
3 **let** b'' be the new position of blank;
4 **let** $k'' < k$ be the largest index so that $X_k \in \text{int}(L_{k''}) \vee E_k \in \text{int}(L_{k''}) \vee b'' \in \text{int}(L_{k''})$;
5 MoveAgentInSubgraph-v1(X_k, E_k, b'', k'');
6 move blank to X_k;
7 **let** P be simple path from X_k to E_k in $[L_0, \ldots, L_{k-1}]$;
8 **let** C be the cycle $P \cup L_k$;
9 rotate in C until agent reaches t;

In Algorithm 9 and Algorithm 10, the initial position s is not in $\text{int}(L_k)$ but the target t is. The difference between the two methods is that, in *Case011* (Algorithm 10), the blank is in L_k. In *Case010*, a recursive call to *MoveAgentInSubgraph-v1* brings the agent from s to X_k, the exit

Case101 ($s \in L_6$, $t \in L_2$, $\perp \in L_6$)

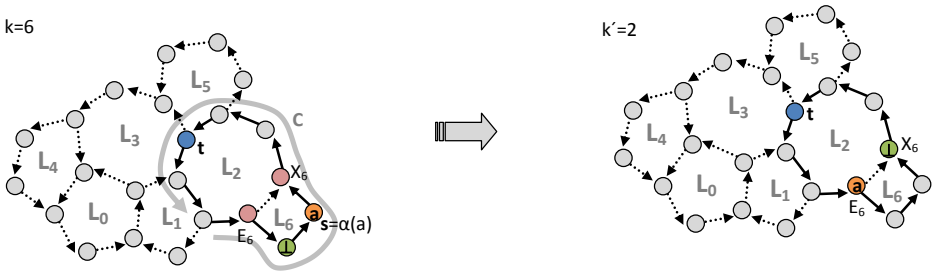

Figure 20: Illustrating Case 101.

Case010 ($s \in L_0$, $t \in L_6$, $\perp \in L_2$)

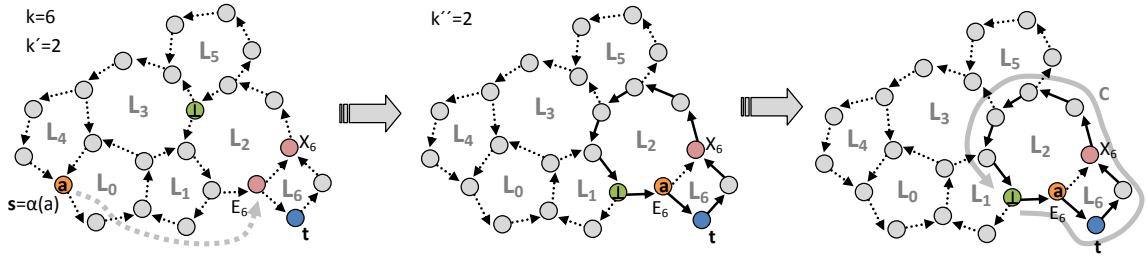

Figure 21: Illustrating Case010.

of the ear L_k. Then, another recursive call brings the agent from X_k to E_k, the entrance of the ear L_k. Then, the blank is taken to X_k. The fact that we have brought the agent from X_k to E_k ensures that the blank can be taken to X_k without having to touch the agent at its current position E_k. The blank can simply travel backwards on the trajectory followed by the agent from X_k to E_k.

Algorithm 10: CASE011

Input: node s; node t; node b; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$

let P be simple path from X_k to E_k in $[L_0, \ldots, L_{k-1}]$;
let C be the cycle $P \cup L_k$;
if $s \in C$ **then**
    rotate in C until agent reaches target;
else
    rotate in C until blank reaches X_k;
    Case010(s, t, X_k, k);

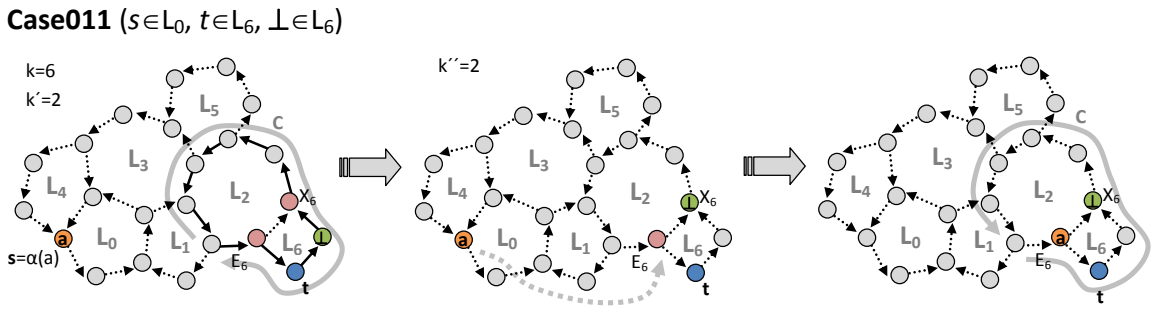

Figure 22: Illustrating Case011.

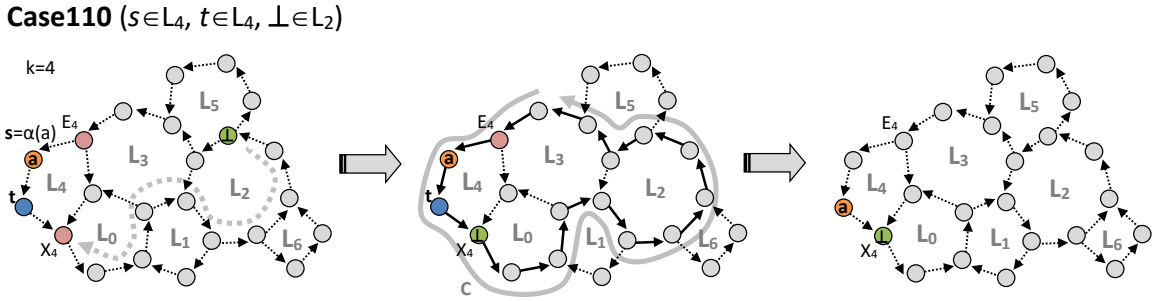

Figure 23: Illustrating Case110.

At the end, perform rotations within a cycle C containing L_k until the agent reaches t. As shown in Algorithm 10, *Case011* first repositions the blank, after which it works similarly to *Case010*. Figures 21 and 22 illustrate these two methods.

Algorithm 11: CASE11G

Input: node s; node t; integer k
Output: moves taking agent from s to t within subgraph $[L_0, \ldots, L_k]$
1 **let** P be simple path from X_k to E_k in $[L_0, \ldots, L_{k-1}]$;
2 **let** C be the cycle $P \cup L_k$;
3 rotate in C until agent reaches t;

In *Case11g* (Algorithm 11), both s and t are in $\text{int}(L_k)$. The blank is in a good position (somewhere along L_k), according to lines 14–15 in Algorithm 6. The method *Case11g* performs rotations within a cycle C containing L_k until the agent reaches t. The two relevant cases 110 and 111 are illustrated in Figures 23 and 24.

Case111 ($s \in L_3$, $t \in L_3$, $\perp \in L_3$)

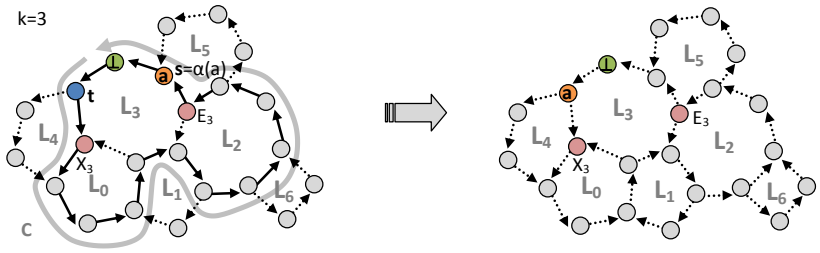

Figure 24: Illustrating Case111.

B.2 Obtaining an $\mathcal{O}(|V|^2)$ Complexity for the Recursive Approach

We introduce a few easy-to-satisfy assumptions that will lead to a quadratic-complexity procedure.

Assumption 1. *In the initial state, the blank is located right in front of the agent. That is, there is a directed edge (u, v) so that the agent is at u and the blank is at v in the initial state.*

This is easy to satisfy by finding a directed path from the initial position of the agent to the initial position of the blank, and moving the blank backwards (i.e., shifting obstacles forward) until it gets right in front of the agent.

Assumption 2. *Every time when we perform rotations in a cycle involving the agent, at the end of the rotations, the blank is right in front of the agent in the cycle. This is the* blank-agent adjacency *condition introduced in the previous section.*

This is easily achieved by having the blank travel backwards in the cycle from behind the agent to the front of the agent (i.e., push all mobile obstacles in the cycle one step ahead, as mentioned previously).

Assumption 3. *The target t belongs to the basic cycle L_0.*

This too is easy to ensure according to to Theorem 1, which says that any cycle can be used as a basic cycle in the open ear decomposition. Thus, all we have to do is to pick a basic cycle that contains the target. Computing a new ear decomposition would have to stay within $\mathcal{O}(|V|^2)$. Once an original ear decomposition is available, this is easy to ensure for new decompositions. If we ignore trivial derived ears in the original decomposition, we obtain a sparse graph. I.e., the number of edges is within the order of the number of nodes, since each ear has p nodes and either p or $p+1$ edges. Combining this with Corollary 4, we obtain the desired complexity bound.

Here is a *simplified procedure*, *MoveAgentInSubgraph-v2*, that works with these assumptions in place. Identify the edge $e = (s, b)$ such that the agent is at s and the blank is at b. Let L_k be the ear that contains the edge e.[11] Consider a cycle C_k obtained from L_k together with a path from its

11. That is, we are in one of the following three cases: 1) both s and b are interior nodes of L_k; or 2) s is the entrance E_k and b is the first interior node; or 3) s is the last interior node and b is the exit X_k.

exit of X_k to the entrance E_k in the subgraph $D_{k-1} = L_0 \cup \cdots \cup L_{k-1}$. Rotate elements (agent and obstacles) within C_k until both the agent and the blank are outside the interior of L_k, on some edge (s', b'), with the blank in front of the agent. We call this rotation within L_k a *descending step*. Now pick the ear $L_{k'}$ with the property that it contains the edge (s', b'). Clearly, $k' < k$, since the path from X_k to E_k belongs to D_{k-1}. Continue recursively with descending steps until $k = 0$, which means that both the agent and the blank are within the basic cycle L_0. Finally, rotate within L_0 until reaching the target, which is actually the *BasicCycleCase* subroutine.

To see the relation to the algorithm presented in the previous section, note that a descending step corresponds to either *Case001* or *Case10a*, with blank-agent adjacency turned on, as follows. When, in a descending step, s is the entrance of L_k and b is its first interior node, we are in *Case001*. Otherwise (i.e., $s, b \in \text{int}(L_k)$; or s is the L_k's last interior node and b is the exit) we are in *Case10a*.

Thus, the simplified procedure can include *Case001*, *Case10a*, and *BasicCycleCase*, but it never includes any of the remaining cases, such as *Case010*, *Case011* and *Case11a*. In particular, this eliminates the multiple calls-to-self in the same block of code (e.g., as done in *Case010*; see Algorithm 9, lines 2 and 5). This allows us to claim the following complexity result.

Theorem 3. *The simplified algorithm presented in this section has both the time and the solution length bounded by* $\mathcal{O}(|V|^2)$.

Proof. Each recursive iteration (i.e., a descending step or the call to *BasicCycleCase* at the end) has at most $\mathcal{O}(|L_k| \times |C_k|)$ moves, where L_k is the ear involved at that iteration and C_k is its corresponding cycle (for $k = 0$, define $C_0 = L_0$). Indeed, shifting all entities (agent and obstacles) in the cycle C_k by one step requires in the order of $|C_k|$ moves, and we perform at most $|L_k|$ such shifting operations. Summing up all these we obtain a number of moves within an order of $\sum_k |L_k| \times |C_k| \leq \sum_k |L_k| \times |V| \leq |V|^2$. □

Furthermore, the quadratic upper bound mentioned is tight. That is, instances exist whose solution requires a quadratic number of steps. Consider an instance where the entire graph is a directed cycle (i.e., a basic cycle with no derived ears) and the target of the agent is right behind its starting position. The solution plan has $(|V| - 1)^2$ steps.

Appendix C. Proof to Theorem 2

Theorem 2. *Let $D = (V, E)$ be a digraph and O an arbitrary partial open ear decomposition (i.e., a basic cycle plus zero or more valid ears, covering a subset of D's nodes). Computing a new* shortest *valid ear, or showing that no new valid ear exists, can be done within a time complexity of* $\mathcal{O}(|V| + |E|)$.

Proof. We show that a new shortest valid ear (if one exists) can be found with a breadth-first search in a digraph D_2. The nodes of D_2 contain at most two copies of each node in D, plus one new node r. For every node n in D, we use a counter to keep track of how many copies of n have been added to D_2. When a first copy of a node n is added to D_2, we denote the corresponding node in D_2 as $(n : 0)$. A second copy is denoted as $(n : 1)$.

The digraph D_2 is built on the fly as the breadth-first search advances. The root node is r. For each node p in O, define a node $(p : 0)$ in D_2, and an edge from r to $(p : 0)$. The root node r has no other successors besides these. We call the nodes $(p : 0)$, with p in O, depth-one nodes, since they have a depth of 1 in the breadth-first search. Nodes from O never have a second copy added to D_2.

For every node in D_2 different than the root r we keep track of its depth-one ancestor (by definition, a node at depth 1 is its own depth-one ancestor).

To expand a node $(n : i)$ in D_2, the successors are generated by enumerating the edges $(n, m) \in E$, and applying the following rules for each edge. If m is in O, do not create a new successor. Otherwise, if no copy of node m has been added yet to D_2, add a new copy $(m : 0)$ as a successor of $(n : i)$. Otherwise, if exactly one copy of m already exists somewhere in D_2 (i.e., a node $(m : 0)$ exists, but a node $(m : 1)$ does not), and the depth-one ancestor of $(m : 0)$ is different from the depth-one ancestor of $(n : i)$, add a new node $(m : 1)$ and make it a successor of $(n : i)$. Otherwise, no successor of $(n : i)$ is created based on the edge (n, m).

During the expansion of a node $(n : i)$ with a depth greater than 1, if an edge (n, m) with m in O is encountered, check if a valid ear has been discovered: Let $(s : 0)$ be the depth-one ancestor of $(n : i)$. If $m \neq s$, a valid ear has been discovered, as: i) the ear starts from one node in O, namely s, and ends at another node in O, namely m; ii) the ear has at least one interior node, namely n (recall that the depth of $(n : i)$ is greater than 1); and iii) no interior node belongs to O, as nodes from O generate in D_2 only depth-one nodes. The search stops after the first valid ear is discovered, or after expanding all nodes, if no valid ear is found.

The number of expanded nodes does not exceed $2|V| + 1$ (i.e., it stays within a constant factor from $|V|$). The number of successors of one search node, namely the root r, is within $\mathcal{O}(|V|)$. Every other node $(n : i)$ in D_2 has a number of successors at most within the order of the number of successors of the node n in D. In other words, the number of edges of D_2 is within $\mathcal{O}(|V| + |E|)$, and the number of nodes of D_2 is within $\mathcal{O}(|V|)$. These ensure that the search complexity stays within the desired bounds.

It remains to show that this search will discover a shortest valid ear if and only if a valid ear exists.

The "only if" part is obvious. For the "if" part, consider that at least one valid ear exists, and assume by contradiction that the algorithm misses out all shortest valid ears.

Given a path $\pi_2 = r, (c_1 : i_1), (c_2 : i_2), \ldots, (c_l : i_l)$ in D_2, we say that the path $\pi = c_1, c_2, \ldots, c_l$ is a projection of π_2 in D. Note that π is a valid path in D, starting from a node in O.

Among all shortest valid ears, consider an ear with the property that it has a longest prefix that is the projection of some path in D_2, generated in the breadth-first search. Let $\Omega = a_1, a_2, \ldots, a_p, x$, with a_1 and x in O, be such an ear, let $a_1, a_2, \ldots, a_{l-1}$ be the corresponding prefix, and let $P_2 = r, (a_1 : i_1), (a_2 : i_2), \ldots, (a_{l-1} : i_{l-1})$ be the path in D_2 projected onto the prefix. As the ear is missed out in search, it follows that $l - 1 < p$.

It follows that $(a_{l-1} : i_{l-1})$ is expanded in the breadth-first search, but no $(a_l : *)$ successor is created. According to the expansion rules described earlier, it further follows that either:

- A node $\nu = (a_l : 0)$ already exists in D_2 at the time of expanding $(a_{l-1} : i_{l-1})$, and it has the same depth-one ancestor as $(a_{l-1} : i_{l-1})$, namely $(a_1 : 0)$; or

- Two nodes $(a_l : 0)$ and $(a_l : 1)$ already exist in D_2 at the time of expanding $(a_{l-1} : i_{l-1})$. By construction, $(a_l : 0)$ and $(a_l : 1)$ cannot have the same depth-one ancestor. It follows that at least one of them has a depth-one successor different than $(x : 0)$. Let $\nu \in \{(a_l : 0), (a_l : 1)\}$ be such a node.

In either case, let $r, (b_1 : j_1), (b_2 : j_2), \ldots, (b_s : j_s), \nu$ be the path to ν in the breadth-first search. Define $\Omega' = b_1, b_2, \ldots, b_s, a_l, a_{l+1}, \ldots, x$. Observe that Ω and Ω' have the same length.[12] It follows that Ω' is a shortest valid ear. Finally, compared to Ω, Ω' has a strictly longer prefix that is the projection of a path generated in the search. This contradicts the way we have chosen Ω.

□

References

Bang-Jensen, J., & Gutin, G. Z. (2008). *Digraphs: Theory, Algorithms and Applications* (2nd edition). Springer.

Barer, M., Sharon, G., Stern, R., & Felner, A. (2014). Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *ECAI-2014 - 21st European Conference on Artificial Intelligence*, pp. 961–962. IOS Press.

Bonusi, D. (2015). Heuristic Enhancements to Multi-Agent Path Finding on Strongly Biconnected Digraphs. Master's thesis, University of Brescia, Brescia, Italy.

Botea, A., & Surynek, P. (2015). Multi-agent path finding on strongly biconnected digraphs. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pp. 2024–2030. AAAI Press.

Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., & Shimony, S. E. (2015). ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, pp. 740–746. AAAI Press.

Cohen, L., & Koenig, S. (2016). Bounded suboptimal multi-agent path finding using highways. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI 2016*, pp. 3978–3979. IJCAI/AAAI Press.

de Wilde, B., ter Mors, A., & Witteveen, C. (2014). Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, *51*, 443–492.

de Wilde, B., ter Mors, A. W., & Witteveen, C. (2013). Push and rotate: Cooperative multi-agent path planning. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS-13)*, pp. 87–94. IFAAMAS.

Erdem, E., Kisa, D. G., Öztok, U., & Schüller, P. (2013). A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*. AAAI Press.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics*, *SSC-4*(2), 100–107.

Jetcheva, J. G., & Johnson, D. B. (2006). Routing characteristics of ad hoc networks with unidirectional links. *Ad Hoc Networks*, *4*(3), 303–325.

12. Ω' cannot be shorter, as this would imply that Ω is not a shortest derived ear. As ν was generated in the breadth-first search before expanding $(a_{l-1} : i_{l-1})$, it follows that $s \leq l - 1$, which further implies that Ω' cannot be longer than Ω.

Johnson, W. W., & Story, W. E. (1879). Notes on the "15" puzzle. *American Journal of Mathematics*, *2*(4), 397–404.

Khorshid, M. M., Holte, R. C., & Sturtevant, N. R. (2011). A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Proceedings of the 4th Annual Symposium on Combinatorial Search, SOCS 2011*. AAAI Press.

Kornhauser, D., Miller, G., & Spirakis, P. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 241–250. IEEE Computer Society.

Luna, R., & Bekris, K. E. (2011). Push and swap: Fast cooperative path-finding with completeness guarantees. In Walsh, T. (Ed.), *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pp. 294–300. IJCAI/AAAI.

Marina, M. K., & Das, S. R. (2002). Routing performance in the presence of unidirectional links in multihop wireless networks. In *Proceedings of ACM MobiHoc*, pp. 12–23. ACM Press.

Papadimitriou, C. H., Raghavan, P., Sudan, M., & Tamaki, H. (1994). Motion planning on a graph. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (SFCS-94)*, pp. 511–520, Washington, DC, USA. IEEE Computer Society.

Ratner, D., & Warmuth, M. (1986). Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable. In *Proceedings of AAAI National Conference on Artificial Intelligence (AAAI-86)*, pp. 168–172. AAAI Press.

Ryan, M. R. K. (2008). Exploiting Subgraph Structure in Multi-Robot Path Planning. *Journal of Artificial Intelligence Research*, *31*, 497–542.

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

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, *195*, 470–495.

Silver, D. (2006). Cooperative pathfinding. *AI Game Programming Wisdom 3*, 99–111.

Silver, D. (2005). Cooperative pathfinding. In Young, R. M., & Laird, J. E. (Eds.), *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2005*, pp. 117–122. AAAI Press.

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI 2010*. AAAI Press.

Surynek, P. (2009). A novel approach to path planning for multiple robots in bi-connected graphs. In *IEEE International Conference on Robotics and Automation (ICRA 2009)*, pp. 3613–3619.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI 2010*. AAAI Press.

Surynek, P. (2012). Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI 2012: Trends in Artificial Intelligence - 12th Pacific Rim International Conference on Artificial Intelligence*, pp. 564–576. Springer.

Surynek, P. (2014a). Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, pp. 875–882. IEEE Computer Society.

Surynek, P. (2014b). Solving abstract cooperative path-finding in densely populated environments. *Computational Intelligence*, *30*(2), 402–450.

Surynek, P., Felner, A., Stern, R., & Boyarski, E. (2016). Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence*, pp. 810–818. IOS Press.

Surynek, P., Surynková, P., & Chromý, M. (2014). The impact of a bi-connected graph decomposition on solving cooperative path-finding problems. *Fundamenta Informaticae*, *135*(3), 295–308.

Wagner, G., & Choset, H. (2015). Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, *219*, 1–24.

Wang, K.-H. C., & Botea, A. (2008). Fast and Memory-Efficient Multi-Agent Pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-08)*, pp. 380–387. AAAI Press.

Wang, K.-H. C., & Botea, A. (2011). MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research*, *42*, 55–90.

Wilson, R. M. (1974). Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, *16*(1), 86–96.

Wu, Z., & Grumbach, S. (2010). Feasibility of motion planning on acyclic and strongly connected directed graphs. *Discrete Applied Mathematics*, *158*(9), 1017 – 1028.