Solving Abstract Cooperative Path-Finding in Densely Populated Environments

Pavel Surynek

Charles University in Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. The problem of cooperative path-finding is addressed in this work. A set of agents moving in a certain environment is given. Each agent needs to reach a given goal location. The task is to find spatial temporal paths for agents such that they eventually reach their goals by following these paths without colliding with each other. An abstraction where the environment is modeled as an undirected graph is adopted – vertices represent locations and edges represent passable regions. Agents are modeled as elements placed in the vertices while at most one agent can be located in a vertex at a time. At least one vertex remains unoccupied to allow agents to move. An agent can move into unoccupied neighboring vertex or into a vertex being currently vacated if a certain additional condition is satisfied. Two novel scalable algorithms for solving cooperative path-finding in bi-connected graphs are presented. Both algorithms target environments that are densely populated by agents. A theoretical and experimental evaluation shows that suggested algorithms represent a viable alternative to search based techniques as well as to techniques exploiting permutation groups on the studied class of the problem.

Keywords: cooperative path-finding, multi-robot path-planning, motion coordination, (N^2-1)-puzzle, N×N-puzzle,15-puzzle, sliding puzzle, domain dependent planning, makespan optimization, *BIBOX*, *BIBOX-θ*.

1. Introduction

A problem of *cooperative path-finding*– CPF (also known from literature as *multi-agent* or *multi-robot path-planning*) [15, 16, 18, 31] is addressed in this work. The task is to find spatial-temporal paths for movable agents, which can be either mobile robots or some other movable objects, so that they eventually reach given goals without colliding with each other by following these paths. The agents are moving in a certain physical or a virtual environment, which is abstracted as an undirected graph with agents placed in its vertices. Edges of the graph represent passable regions in the environment. The main source of the complexity of the problem arises from the possibility of interactions of agents with the environment and in major part from interactions among agents themselves. The agents need to avoid obstacles in the environment, which is embodied directly in the graph by absence of edges (or vertices), and they must not collide with each other, which is modeled by the constraint that at most one agent is located in a vertex at a time.

CPF is motivated by many real-life tasks ranging from *navigation of a group of mobile robots*, *rearranging of containers in storage* (see Figure 1), or *ship avoidance* to *computer generated imagery* where motion of multiple characters needs to be planned. All these tasks can be modeled as a CPF at a certain level of abstraction. Actually, the top-level abstraction generally adopted in the CPF ap-

Initial version submitted to *Computation Intelligence* on November 17, 2010. Reviews received on December 31, 2011. Revision submitted on February 18, 2012. Reviews for revision received on August 13, 2012. Minor revision submitted on September 11, 2012.

proach to these tasks uncovers challenges that must be inevitably faced if someone tries to solve these tasks – such as the question if some arrangement of agents can be reached from another one under the given physical constraints.

The *centralized* approach is adopted throughout this work. That is, all the agents and the whole environment are fully observable to the centralized planning mechanism. The individual agents make no decisions by themselves; they merely execute plans found by the centralized planner. This is an approach adopted also in all the relevant related works.

This work is specifically targeted on the case of CPF with environments densely populated by agents. Such a case is challenging from several points of view. As there is limited unoccupied space in the environment, agents cannot move freely towards their goals and are forced to cooperate intensively with each other.

At the same time, it is interesting to study the possibility of parallel movements of multiple agents at once, which may reduce the total execution time of the plan significantly. To study parallelism in CPF a variant of CPF called *parallel CPF* (*pCPF*) is defined. The pCPF variant additionally enables an agent to enter a vertex that which is simultaneously vacated by another agent if certain additional conditions are satisfied. The intended effect of the relaxed requirement on movements is to allow a chain of agents to move at once where only the leading agent needs to enter a currently unoccupied vertex and other agents follow it. Allowing such higher movement parallelism is a more realistic model in certain scenarios – especially in the case where unoccupied space is shrinking towards zero.

1.1. **Related Works**

One of the recent successful approaches to CPF was to search for a spatial-temporal path for each agent separately from other agents. If agents are considered separately, an approach is usually called *decoupled* [18, 19]. These techniques are build around the A* algorithm [14] in most cases which is used to search for a shortest path from the current location of an agent to its goal while spatial temporal paths of other already scheduled agents are considered. The positive aspect of the decoupled approach is that it often finds plans that are near to the optimum with respect to the *makespan* (the total time or the number of steps necessary to execute the plan) as short paths for agents are preferred during the search. On the other hand, these methods are extremely sensitive to prioritizing agents as it can easily happen that the already scheduled agents block paths for not yet scheduled ones. This is one of the major drawback of the WHCA* algorithm [18] which is intrinsically incomplete due to this phenomenon (up to 100 agents in the environment are reported; approximately 10% of the environment is occupied). The incompleteness is getting more prominent on cases with the increasing density of agents (see Section 4).

In [19], authors present a complete and optimal algorithm for CPF, which uses sophisticated heuristics to reduce the search space by detecting that sometimes no cooperation is necessary among agents. The trouble with incompleteness has been thus overcome in this approach. However, this method seems to be targeted on relatively sparsely populated environments where actually agents can travel most of the trajectory towards their goals without interacting with other agents (results for the occupancy of environment less than 10% are reported; up to 60 agents are reported to move in environments containing approximately 800 vertices).

Several techniques for CPF are trying to exploit structural properties of the problem to increase the performance. For instance, graph structures are heavily exploited in [15, 16]. The undirected graph modeling the environment is first decomposed into sub-graphs of some interesting structure such as cliques and others over which various known patterns of rearranging agents can be used. The

search for the final plan is then performed over an abstract map whose nodes are represented by the sub-graphs of the original graph (experiments with up to 20 agents in the environments consisting of hundreds of vertices are reported).

Another way to exploit structural properties of the problem is to observe local juxtapositions in the current arrangement of agents. This approach was adopted by authors in [29, 30, 31, 32]. If some important juxtaposition of agents is detected then known rearrangement process is applied to advance the situation towards an arrangement where agents are closer to their goals. These techniques turned out to be successful on environments containing many agents but also providing lot of unoccupied space (hundreds of agents moving in environment consisting of thousands of vertices are reported).

Cooperative path-finding has been also addressed from different perspective than as a task of finding a route from the initial location to the goal. A concept of so-called *direction maps* is introduced in [6, 7] to enable coherent movements of multiple agents in various complex patterns that often arise in computer entertainment (such as agents *patrolling* around some location in an RTS game and so on).

A rich source of related works for CPF is represented by works on *motion planning over graphs* [8, 10, 12, 13, 35]. The term of *pebble motion on graph* (PMG) used in these work denotes the same concept as CPF in fact. Particularly, important results were achieved for a special case of PMG known as (N^2-1)-puzzle or $N \times N$-puzzle [11, 12, 13], which consists of a 4-connected grid of size $N \times N$ with just one vertex unoccupied. Many algebraic and complexity results are known for (N^2-1)-puzzle and for PMG generally (some of them will be discussed and used later). It is for instance known that finding the makespan optimal solution to the (N^2-1)-puzzle is an NP-hard problem [12, 13].

Regarding general PMG, algorithms proving its membership into the P class are given in [8, 35] with asymptotic time complexities and lengths of generated solutions of $\mathcal{O}(|V|^3)$ and $\mathcal{O}(|V|^5)$ respectively ($G=(V,E)$ is a graph modeling the environment). The former one – which will be denoted as *MIT*[1] algorithm in this work – represents an algebraic approach to CPF exploiting permutation groups. This algorithm is complete and is capable of solving CPF instances irrespectively of the density of population of agents (just one unoccupied vertex is sufficient in the case with *bi-connected graph* [34] to solve all the solvable instances). The algorithm regards the arrangement of agents as a permutation and the desired goal permutation is composed of elementary permutations over triples of agents. The drawback of the MIT algorithm is that it was not designed for practical use and hence generated solutions have typically long makespan from the pragmatic point of view despite the very good theoretical upper bound of $\mathcal{O}(|V|^3)$.

1.2. **Contribution, Motivation, and Organization**

The main contribution of this work is a presentation of two scalable makespan sub-optimal algorithms *BIBOX* and *BIBOX-θ* that are designed for solving CPF on bi-connected graphs. From the pragmatic point of view, presented algorithms are primarily targeted on cases with environment densely populated by agents (that is, with limited unoccupied space).

Although the targeted case of CPF is special, it has a great practical importance since many real-life environments can be abstracted as *2D/3D grids* which are typically bi-connected. Techniques for tackling CPF in highly occupied space are worthwhile in cases when the space is a scarce resource. Consider for example storage where piles of stored items can be automatically reconfigured – Figure

[1] This working name for the algorithm was chosen by us and it was inspired by the fact that the principal author was affiliated with Massachusetts Institute of Technology (MIT) at the time publishing the article [8]. Authors themselves did not use any name for their algorithm.

1. Such kind of automation can save lot of space since without such automation the storage must be larger to make all the piles accessible. Occupying large space with buildings have considerable negative environmental and economic impacts (occupied land was in many cases arable and its occupation is difficult to revert if it is possible at all).

Both suggested algorithms have polynomial time complexity. They were considered as an alternative to the MIT algorithm. A consideration as an alternative to search based algorithms when the makespan optimal solution is not needed and speed of solving is preferred is also viable.

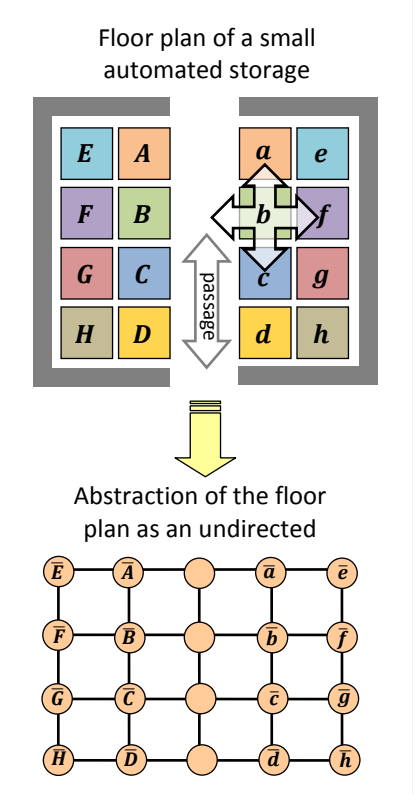


Figure 1. *Illustration of* ***modeling*** *the environment in a real scenario by undirected graph.* The scenario consists of a small automated storage with movable piles of stored items (labeled A to H and a to h). Each pile can be moved left/right/forward/backward. Items in piles are accessible from the passage – to access piles E-H or e-h the storage needs to be rearranged. The environment is modeled as grid of size 4×5 which is a bi-connected graph.

Notice that there is a growing interest in developing algorithms of such category – the very recent contribution represented by the *PUSH-AND-SWAP* algorithm [9] shares lots of aspects with our work (complexity issues and the way of rearranging agents).

Some of the results presented in this work can be also found in some form in conference proceedings [20, 21, 22, 23]. This work is accompanied with a technical report [27] where some additional details such as formal proofs of all the propositions can be found.

The organization of the work is as follows: formal definitions of PMG and pCPF are given first in Section 2. Some basic properties of these problems are discussed subsequently. New algorithms *BIBOX* and *BIBOX-θ* are presented in the main section - Section 3. The final section – Section 4 – is devoted to an extensive experimental evaluation of both new algorithms. A competitive comparison against WHCA* and MIT is presented. Finally, some concluding remarks are given and future prospects are discussed.

2. Pebble Motion on a Graph (PMG) and Parallel Cooperative Path-Finding (pCPF)

Consider an environment in which a group of mobile agents is moving. The agents are all identical (that is, they are all of the same size and have the same moving abilities). Each agent starts at a given initial position and it needs to reach a given goal position. The problem being addressed here consists of finding a spatial-temporal path for each agent so that it eventually reaches its goal by following this path. The agents must not collide with each other and they must avoid obstacles in the environment along the whole process of relocation according to constructed paths.

The environment with obstacles within that the agents are moving is modeled as an undirected graph. The vertices of this graph represent positions in the environment and the edges model passable regions from one position to another. At each time step, all the agents are located in some vertices while at most one agent is allowed per vertex. Some vertices may be vacant – precisely, at least one vertex should be vacant to allow agents to move.

If the agent is placed in a vertex at a given time step then the result of a motion is the situation where the agent is placed in the neighboring vertex at the following time step. The agent is allowed to enter the neighboring vertex supposing it is unoccupied or being vacated by another agent in a certain case while no other agent is trying to enter the same target vertex (precise definition of conditions that the movement must satisfy will follow).

We distinguish two variants of motion problems here, which differ in conditions on movements. Agents in the first one are called *pebbles* and the related problem is called *pebble motion on a graph*. Briefly said, it is required that the target vertex of the movement must be vacant. The second variant is called *parallel cooperative path-finding*. Movable agents in this variant are called *agents* and the condition on movements is relaxed so that it additionally allows movements into vertices that are currently vacated by another agent in a case when agents are moving in a chain style (like a train).

2.1. **Formal Definitions of Cooperative Path Planning Problems**

The first definition below is for the problem of *pebble motion on a graph*– PMG [8] which is also known as *cooperative path-planning/finding* – CPF [18,19, 29] or *multi-robot/agent path-planning/finding* – MRPP [15, 16, 20, 23]. All these terms from the literature denote the same concept in fact. The special variant of pebble motion on a graph is represented by (N^2-1)-puzzle (which is also known as the $N \times N$-puzzle) [12, 13].

Definition 1 ***(pebble motion on a graph*** **– PMG).** Let $G = (V, E)$ be an undirected graph and let $P = \{\bar{p}_1, \bar{p}_2, \dots, \bar{p}_\mu\}$ where $\mu < |V|$ be a set of *pebbles*. The *initial arrangement* and the *goal arrangement* of pebbles in G are defined by two uniquely invertible functions $S_P^0: P \longrightarrow V$ (that is $S_P^0(p) \neq S_P^0(q)$ for every $p, q \in P$ with $p \neq q$) and $S_P^+: P \longrightarrow V$ respectively. A problem of *pebble motion on a graph* (*PMG*) is the task to find a number ξ and a sequence of pebble arrangements $\mathcal{S}_P = [S_P^0, S_P^1, \dots, S_P^\xi]$ such that the following conditions hold (the sequence represents arrangements of pebbles at each time step – the time step is indicated by the upper index):

(i) $S_P^k: P \longrightarrow V$ is a uniquely invertible function for every $k = 1,2, \dots, \xi$;

(ii) $S_P^\xi = S_P^+$ (that is, all the pebbles eventually reach their destination vertices);

(iii) either $S_P^k(p) = S_P^{k+1}(p)$ or $\{S_P^k(p), S_P^{k+1}(p)\} \in E$ for every $p \in P$ and $k = 1,2, \dots, \xi - 1$ (that is, a pebble either stays in a vertex or moves along an edge);

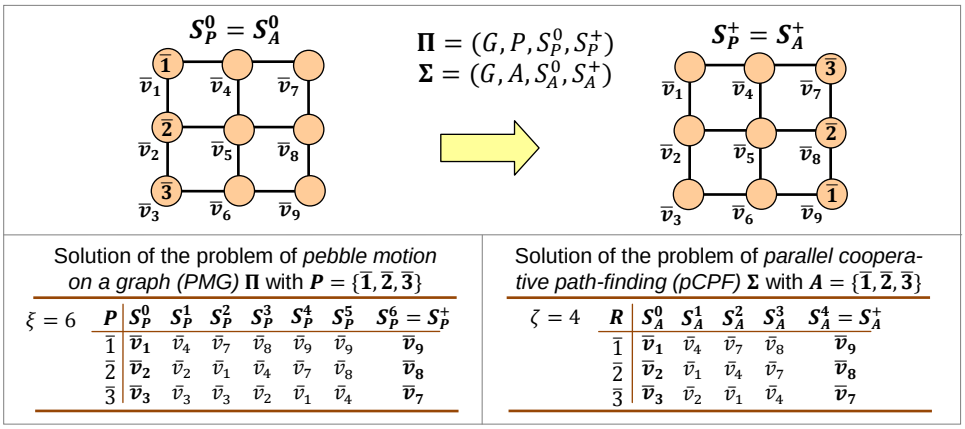
(iv) if $S_P^k(p) \neq S_P^{k+1}(p)$ (that is, the pebble p moves between time steps k and $k+1$) then $S_P^k(q) \neq S_P^{k+1}(p)$ $\forall q \in P$ with $q \neq p$ must hold for every $p \in P$ and $k = 1,2, \dots, \xi - 1$ (that is, a pebble can move into a currently unoccupied vertex only).

The instance of PMG is formally a quadruple $\Pi = (G, P, S_P^0, S_P^+)$. A solution to the instance Π will be denoted as $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \dots, S_P^\xi]$. □

When speaking about a move at a time step k, it is referred to the time step of commencing the move (the move is performed instantaneously between time steps k and $k+1$).

The second variant of motion problem on a graph adopted in this work relaxes the condition that the target vertex of a pebble/agent must be vacated in the previous time step. Thus, the motion of an agent entering the target vertex, that is simultaneously vacated by another agent and no other agent is trying to enter the same target vertex, is allowed in a certain case. However, there must be some leading agent initiating such a chain of moves by moving into a currently unoccupied vertex which no other agent is entering at the same time step (that is, agents can move "like a chain" with the leading agent moving into an unoccupied vertex in the front). The problem is formalized in the following

definition – it is called *parallel cooperative path-finding* – *pCPF* since the different style of moving basically enables higher parallelism. The same concept is sometimes also referred as *multi-robot path-planning* in the literature [22, 24, 26, 27].



Solution of the problem of *pebble motion on a graph (PMG)* Π with $\boldsymbol{P} = \{\bar{1}, \bar{2}, \bar{3}\}$

$\xi = 6$

$\boldsymbol{P}$	$\boldsymbol{S_P^0}$	$\boldsymbol{S_P^1}$	$\boldsymbol{S_P^2}$	$\boldsymbol{S_P^3}$	$\boldsymbol{S_P^4}$	$\boldsymbol{S_P^5}$	$\boldsymbol{S_P^6 = S_P^+}$
$\bar{1}$	$\boldsymbol{\bar{v}_1}$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$	$\bar{v}_9$	$\boldsymbol{\bar{v}_9}$
$\bar{2}$	$\boldsymbol{\bar{v}_2}$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\boldsymbol{\bar{v}_8}$
$\bar{3}$	$\boldsymbol{\bar{v}_3}$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\boldsymbol{\bar{v}_7}$

Solution of the problem of *parallel cooperative path-finding (pCPF)* Σ with $\boldsymbol{A} = \{\bar{1}, \bar{2}, \bar{3}\}$

$\zeta = 4$

$\boldsymbol{R}$	$\boldsymbol{S_A^0}$	$\boldsymbol{S_A^1}$	$\boldsymbol{S_A^2}$	$\boldsymbol{S_A^3}$	$\boldsymbol{S_A^4 = S_A^+}$
$\bar{1}$	$\boldsymbol{\bar{v}_1}$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\boldsymbol{\bar{v}_9}$
$\bar{2}$	$\boldsymbol{\bar{v}_2}$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\boldsymbol{\bar{v}_8}$
$\bar{3}$	$\boldsymbol{\bar{v}_3}$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\boldsymbol{\bar{v}_7}$

Figure 2. *Example of instance of* ***PMG*** *and* ***pCPF***. Both instances are illustrated on the same graph with the same initial and goal arrangements. The task is to move pebbles/agents from their initial positions specified by S_P^0/S_A^0 to the goal positions specified by S_P^+/S_A^+. A solution of the makespan 6 ($\xi = 6$) is shown for the PMG instance and a solution of the makespan 4 ($\zeta = 4$) is shown for the pCPF instance. Notice the differences in parallelism between both solutions.

Definition 2 ***(parallel cooperative path-finding – pCPF).*** Again, let $G = (V, E)$ be an undirected graph. A set of *agents* $A = \{\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_\nu\}$ where $\nu < |V|$ is given instead of the set of pebbles. Similarly, the graph models the environment where the agents are moving. The *initial arrangement* and *the goal arrangement* of agents are defined by two uniquely invertible functions $S_A^0 : A \longrightarrow V$ (that is $S_A^0(a) \neq S_A^0(b)$ for every $a, b \in A$ with $a \neq b$) and $S_A^+ : A \longrightarrow V$ respectively. A problem of *parallel cooperative path-finding* (*pCPF*) is then the task to find a number ζ and a sequence of agent arrangements $\mathcal{S}_A = [S_A^0, S_A^1, \ldots, S_A^\zeta]$ for that the following conditions hold:

(i) $S_A^k : A \longrightarrow V$ is a valid arrangement for every $k = 1,2, \ldots, \zeta$ (that is, uniquely invertible);

(ii) $S_A^\zeta = S_A^+$ (that is, all the agents eventually reach their destinations);

(iii) either $S_A^k(a) = S_A^{k+1}(a)$ or $\{S_A^k(a), S_A^{k+1}(a)\} \in E$ for every $a \in A$ and $k = 1,2, \ldots, \zeta - 1$ (that is, an agent either stays in a vertex or moves into the neighboring vertex);

(iv) if $S_A^k(a) \neq S_A^{k+1}(a)$ (that is, the agent a moves between time steps k and $k + 1$) then there must exist a sequence of distinct agents $[a = b_0, b_1, \ldots, b_\lambda]$ with $\lambda \in \mathbb{N}_0$ such that $S_A^k(c) \neq S_A^{k+1}(b_\lambda)$ $\forall c \in A$ with $c \neq b_\lambda$ (b_λ moves to a vertex that is unoccupied at time step k; b_λ is a *leading* agent of the chain of agents which the sequence is part of) and $S_A^{k+1}(b_i) = S_A^k(b_{i+1})$ for $i = 0,1, \ldots, \lambda - 1$ (agents $a = b_0, b_1, \ldots, b_{\lambda-1}$ follows the leader like a chain; they move all at once between time steps k and $k + 1$).

The instance of pCPF is formally a quadruple $\Sigma = (G, A, S_A^0, S_A^+)$. A solution to the instance Σ will be denoted as $\mathcal{S}_A(\Sigma) = [S_A^0, S_A^1, \ldots, S_A^\zeta]$. □

Notice in point (iv) that if the agent a moves into an unoccupied vertex then the required sequence of distinct agents consists of a itself ($\lambda = 0$) and the latter condition in point (iv) is empty. Notice also that the condition on unique invertibility implies that no two agents can simultaneously enter the same target vertex.

The numbers ξ and ζ represent the *makespan* of solutions. The makespan needs to be distinguished from the *size* of solution, which is the total number of moves performed by pebbles/agents. Example instances of both problems and their solutions are illustrated in Figure 2.

2.2. **Known Properties of Motion Problems and Related Questions**

Notice that a solution of PMG as well as a solution of pCPF allows a pebble/agent to stay in a vertex for more than a single time step. It is also possible that a pebble/agent visits the same vertex several times within the solution. Hence, the sequence of moves for a single pebble/agent does not necessarily form a simple path in the given graph.

Notice further that both problems intrinsically allow parallel movements of pebbles/agents. That is, more than one pebble/agent can perform a move in a single time step. However, pCPF allows higher motion parallelism due to its weaker requirements on agent movements (see Figure 2). More than one unoccupied vertex is necessary to obtain parallelism in PMG while only one unoccupied vertex is sufficient to obtain parallelism within a solution of pCPF (consider for example agents moving around a cycle). The following straightforward proposition puts into relation solutions of instances of PMG and pCPF with the same set of agents and their arrangements over the same graph.

Proposition 1 ***(problem correspondence).*** Let $\Pi = (G, P, S_P^0, S_P^+)$ be an instance of PMG and let $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \dots, S_P^\xi]$ be its solution. Then $\mathcal{S}_A(\Sigma) = \mathcal{S}_P(\Pi)$ is a solution to an instance of pCPF $\Sigma = (G, P, S_P^0, S_P^+)$. ■

To prove the proposition it is sufficient to observe that the condition (iv) in the definition of pCPF is a relaxation of the corresponding condition in the definition of PMG.

There is a variety of modifications of the defined problems. A natural additional requirement is to produce solutions with the makespan as short as possible (that is, the numbers ξ or ζ are required to be as small as possible). Unfortunately, this requirement makes both PMG and pCPF intractable. It was shown in [12, 13] that the optimization variant of a special case of PMG is NP-hard [3] – this special case is generally known as $N \times N$*-puzzle* or $(N-1)$*-puzzle*. It consists of a graph that can be embedded in the plane as a square 4-connected grid with a single unoccupied vertex. Thus, the optimization variant of general PMG is NP-hard as well.

Here we work with restrictions of both types of problems on *bi-connected graphs* [34]. Hence, it is a reasonable question what is the complexity of these classes. Since the grid graph forming the mentioned $N \times N$-puzzle is bi-connected**,** the immediate answer is that the optimization variant of PMG with a bi-connected graph is NP-hard as well.

Nevertheless, it is not possible to make any similar simple statement about the complexity of the optimization variant of pCPF. The situation here is complicated by the inherent parallelism, which can affect the makespan in some unforeseen way. Constructions used for the $N \times N$ puzzle in [12, 13] thus no longer work. Using different technique it has been recently shown by the author that the optimization variant of pCPF is *NP*-hard too [24, 26].

Observe further that reported NP-hard case of PMG have a single unoccupied vertex. This fact may raise the question how the situation is changed when there are more than one unoccupied vertices as they may simplify the situation. Unfortunately, it is not the case. PMG with the fixed number of unoccupied vertices is still NP-hard since multiple copies of the $N \times N$ puzzle from [12, 13] can be used to add as many unoccupied vertices as needed. Without providing further details, the instance of pCPF used in the reduction to prove the *NP*-hardness of the problem in [24] had many unoccupied

vertices and its graph was connected (even bi-connected). Altogether, a mere allowance of many unoccupied vertices with no additional structural conditions does not simplify the problem.

Without the requirement on the optimality of the makespan, the situation is much easier; PMG is in the *P* class as it was shown in [8, 35]. Due to Proposition 1, pCPF is in the *P* class as well. Thus, it seems that PMG and pCPF have been already resolved. However, constructions proving the membership of PMG into the *P* class used in [8, 35] generate solutions that are too long for practical use [21, 22, 23]. As the makespan of the solution is of great importance in practice, this fact makes these methods unsuitable when some real life motion problem is abstracted as an instance of PMG. Thus, alternative solving methods has been developed [20, 21, 22, 23] and they are revised in this work.

3. Sub-optimal Solving Algorithms

The basic idea of presented sub-optimal algorithms is to exploit structural properties offered by the concept of *bi-connectivity*. It is known that bi-connected graphs can be inductively constructed as a union of a sequence of *rings* or *handles* while at every stage of this construction the intermediate graph is bi-connected [33, 34].

After arranging agents into the last handle we do not need to care about it anymore and consequently the task reduces to a task of the same type but on a smaller bi-connected graph. Fortunately, bi-connected graphs have another interesting property; every two vertices are connected by at least two vertex disjoint paths, which allow quite complex rearranging of agents. For example, an individual agent can move relatively freely. One path is traversed by the agent and alternative paths are used to keep unoccupied vertex always in front of the agent. In addition, handles of the decomposition evokes the possibility that agents within them can be rotated, which is actually used in proposed algorithms. Notice that all the mentioned styles of movements are friendly to the parallelism as defined in pCPF – for example, agents in a handle can be rotated within a single time step. However, there are many technical difficulties that need to be addressed to make the above ideas workable.

3.1. **BIBOX: A Novel Algorithm for Pebble Motion on a Bi-connected Graph**

The first algorithm presented here called *BIBOX* was originally proposed in [20]. The input instance should consist of a *non-trivial bi-connected graph* (that is, bi-connected graph not isomorphic to a cycle) with exactly two unoccupied vertices. As the algorithm produces solution consisting of single move per time step it does not matter if PMG or pCPF is given on the input – in the following text pCPF will be always considered. A method how to increase parallelism in the resulting solution to take the advantage of the definition of pCPF will be discussed in Section 3.1.4.

The algorithm proceeds inductively according to the known property of bi-connected graphs that they can be built from a cycle by addition of a sequence of *handles*. Adding a handle means either to insert a new edge into the graph or to connect endpoints of a path consisting of new vertices somewhere into the graph. The important property is that currently built graph is bi-connected at every stage of the construction.

The process of building a graph by adding handles can be reverted as well. That is, the graph can be deconstructed until a cycle remains by removing handles from it. If it is somehow possible to arrange agents whose goal positions are in the handle to be removed before it is actually removed, we have a good starting point for a new solving algorithm because after removal of a handle the problem just reduced to the smaller graph. To obtain a new algorithm it remains to show how agents can be arranged into the handle and how to deal with the cycle that remains at the end of the process.

The process of removing of handles is presented here just for intuition. They actually do not need to be removed during the solving process. It is sufficient not to consider and use a handle after all the agents are properly arranged to their goal positions within that.

The intuition for arranging agents in the cycle that eventually remains is to regard their ordering as a permutation. The goal arrangement of agents in the cycle can be also regarded as a permutation. Thus, we need to change ordering of agents to form another permutation. If it is possible to exchange a pair of agents with respect to their current ordering, then every permutation of agents can be obtained. It will be shown how to utilize two unoccupied vertices to enable exchanges of agents in the remaining cycle.

It is possible to build a bi-connected graph in multiple different ways by adding handles. Hence, the algorithm as well as the produced solution is sensitive to the selection and ordering of handles used in the solving process.

3.1.1. *Graph-theoretical Preliminaries*

The BIBOX algorithm is built around the notion of *bi-connectivity* and around graph theoretical properties of bi-connected graphs [33]. Let us recall the notion of bi-connectivity and related properties briefly.

Definition 3 ***(connected graph).*** An undirected graph $G = (V, E)$ is *connected* if $|V| \geq 2$ and for any two vertices $u, v \in V$ such that $u \neq v$ there is an undirected path connecting u and v. □

Definition 4 (*bi-connected graph, non-trivial).* An undirected graph $G = (V, E)$ is *bi-connected* if $|V| \geq 3$ and the graph $G' = (V', E')$, where $V' = V \setminus \{v\}$ and $E' = \{\{u, w\} | u, w \in V \wedge u \neq v \wedge w \neq v\}$, is connected for every $v \in V$. A bi-connected graph not isomorphic to a cycle will be called *non-trivial* bi-connected graph. □

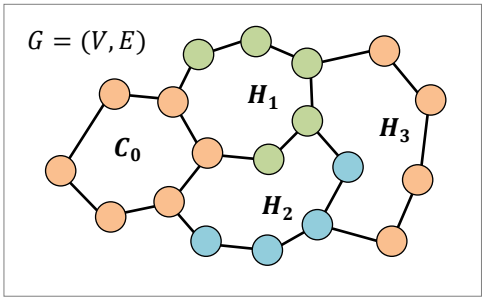


Figure 3. *Example of* ***bi-connected graph***. A handle decomposition is illustrated.

Observe that, if a graph is bi-connected, then every two distinct vertices are connected by at least two *vertex disjoint paths* (equivalently, there is a cycle containing both vertices; only internal vertices of paths are considered when speaking about vertex disjoint paths - vertex disjoint paths can intersect in their *start points* and *endpoints*). An example of bi-connected graph is shown in Figure 3.

Bi-connected graphs have an important property, which is exploited within the algorithm. Each bi-connected graph can be constructed starting from a cycle by an operation of *adding a handle* [28, 33, 34]. Consider a graph $G = (V, E)$; the new handle with respect to G is a sequence $H = [u, w_1, w_2, \ldots, w_h, v]$ where $h \in \mathbb{N}_0$, $u, v \in V$ (called *connection vertices*) and $w_i \notin V$ for $i = 1,2, \ldots, h$ (w_i are fresh vertices). The result of the addition of the handle H to the graph G is a new graph $G' = (V', E')$ where $V' = V \cup \{w_1, w_2, \ldots, w_h\}$ and either $E' = E \cup \{\{u, v\}\}$ in the case of $h = 0$ or $E' = E \cup \{\{u, w_1\}, \{w_1, w_2\}, \ldots, \{w_{h-1}, w_h\}, \{w_h, v\}\}$ in the case of $h > 0$. Let the sequence of handles together with the initial cycle be called a *handle decomposition* of the given bi-connected graph. Again, see Figure 3 for illustrative example.

Lemma 1 ***(handle decomposition)*** [28, 33, 34]**.** Any bi-connected $G = (V, E)$ graph can be obtained from a cycle by a sequence of operations of adding a handle. Moreover, the corresponding handle decomposition of the graph G can be found in the worst-case time of $\mathcal{O}(|V| + |E|)$ and the worst-case space of $\mathcal{O}(|V| + |E|)$. ■

The important property of the construction of a bi-connected graph according to its handle decomposition is that the currently constructed graph is bi-connected at every stage of the construction. This property is substantially exploited in the design of the *BIBOX* algorithm.

The algorithm is presented below using a pseudo-code as Algorithm 1 and Algorithm 2 (algorithms are illustrated with pictures for easier understanding). The algorithm starts with the last handle of the handle decomposition and proceeds to the initial cycle. Agents, that goal positions are within the last handle, are moved to their goal positions within this handle. After that, the instance reduces to a smaller bi-connected graph. That is, the last handle is not considered any more since its agents do not need to move any more. This process is repeated until the initial cycle of the decomposition remains where a different technique is used.

Let $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ be an instance of pCPF. The handle decomposition of the graph G is formally a sequence $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ with $d \in \mathbb{N}$, where C_0 is the initial cycle and H_c is a handle for $c = 1,2, \ldots, d$. The order of handle additions in construction of G corresponds to their positions in the sequence (that is, H_1 is added to C_0 first; and H_d is added as the last). A handle $H_c = [u^c, w_1^c, w_2^c, \ldots, w_{h_c}^c, v^c]$ for $c \in \{1,2, \ldots, d\}$ is assigned a cycle $C(H_c)$. The cycle $C(H_c)$ consists of the sequence vertices on a path connecting v^c and u^c in a graph before the addition of H_c followed by vertices $w_1^c, w_2^c, \ldots, w_{h_c}^c$. Specially, it is defined that $C(C_0) = C_0$.

The following lemma justifies two properties exploited by the algorithm. It justifies that it is possible to keep handy two unoccupied vertices in the not yet solved part of the graph since one unoccupied vertex is needed to solve handles and two unoccupied vertices are needed to solve the initial cycle. The lemma ensures that the original goal arrangement can be transformed to an arrangement where unoccupied vertices are located in the initial cycle. Thanks to this property it never happens that an unoccupied vertex become locked in some already solved handle. Details of the transformation are discussed later.

Lemma 2 ***(existence of two vertex disjoint paths)*.** Let $G = (V, E)$ be a bi-connected graph and let $u_1, u_2 \in V$ and $v_1, v_2 \in V$, where u_1, u_2, v_1, v_2 are pair-wise distinct, be two pairs of vertices. Then either the first or the second of the following claims holds:

(a) There exist two vertex disjoint paths φ and χ such that they connect u_1 with v_1 and u_2 with v_2 in G respectively.
(b) There exist two vertex disjoint paths φ and χ such that they connect u_1 with v_2 and u_2 with v_1 in G respectively. ■

Notice that the lemma states that individual vertices in the input pair of vertices are indifferent with respect to connecting by vertex disjoint paths. As the proof of the lemma is rather technical, we refer the reader to [27] where the detailed proof can be found. The idea of proof is that the given 4-tuple of vertices u_1, u_2, v_1, v_2 is assigned a 4-tuple of non-negative integers such that each number refers to a handle of the decomposition or the initial cycle where the corresponding vertex is located. Then the proof proceeds inductively according to the lexicographic ordering of these 4-tuples of numbers. For a selected pair of vertices partial connection paths are constructed towards handles with

lower numbers (a certain case analysis in the worst-case time of $\mathcal{O}(1)$ has to be done). Then it holds from the induction hypothesis that remaining parts of connection paths should exist since they connect 4-tuple of vertices with lower 4-tuple of assigned numbers.

3.1.2. *Pseudo-code of the BIBOX Algorithm*

Several basic operations are introduced to express the *BIBOX* algorithm in an easier way. These operations are formally described using pseudo-code as Algorithm 1. In addition to functions S_A^0 and S_A^+ there will be a function $S_A: A \longrightarrow V$ to represent the current arrangement of agents in G and functions $\Phi_A^0: V \longrightarrow A \cup \{\perp\}$, $\Phi_A^+: V \longrightarrow A \cup \{\perp\}$, and $\Phi_A: V \longrightarrow A \cup \{\perp\}$ which are generalized inverses of S_A^0, S_A^+, and S_A respectively; the symbol $\perp$ is used to represent an unoccupied vertex (that is, $(\forall a \in P)$ $\Phi_A(S_A(a)) = a$ and $\Phi_A(v) = \perp$ if $(\forall a \in A)\ S_A(a) \neq v$). Each undirected cycle appearing in the handle decomposition of the input graph is assigned a fixed orientation. Let C be an undirected cycle (a set of vertices of the cycle), then the orientation of C is expressed by functions $next_{\circlearrowleft}$ and $prev_{\circlearrowleft}$ where $next_{\circlearrowleft}(C, v)$ for $v \in C$ is a vertex following v (with respect to the positive orientation) and $prev_{\circlearrowleft}(C, v)$ is a vertex preceding v (with respect to the positive orientation). The orientation of a cycle given by $next_{\circlearrowleft}$ and $prev_{\circlearrowleft}$ is observed also whenever vertices of the cycle are explicitly enumerated in the code.

Each vertex of the input graph is either *locked* or *unlocked*. Auxiliary operations *Lock* (X) and *Unlock*(X) locks or unlocks a set of vertices $X \subseteq V$. The state of a vertex is used to determine whether an agent can move into it. Typically, an agent is not allowed to enter a locked vertex (see the pseudo-code for details).

It is assumed that it holds that $|A| = |V| - 2$ (that is, there are exactly two unoccupied vertices in the graph G). It is required by the main phase of the algorithm that the two unoccupied vertices are located in the first two vertices of the initial cycle within the goal arrangement. This requirement is treated by a function *Transform-Goal* and a procedure *Finish-Solution*. The function *Transform-Goal* determines two vertex disjoint paths from unoccupied vertices in the goal arrangement to the first two vertices in the initial cycle of the handle decomposition. Existence of these paths is ensured by Lemma 2. The goal arrangement is transformed so that finally unoccupied vertices are located in the initial cycle. This is done by shifting agents within the goal arrangement along the two found paths. After the modified instance is solved, the function *Finish-Solution* moves unoccupied vertices back to their goal locations in the original unmodified goal arrangement. The final placement of unoccupied vertices is done by shifting agents along the same two paths but in the opposite direction.

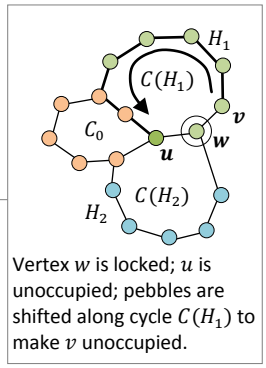
It is assumed that the input bi-connected graph G is non-trivial for further simplifying the pseudo-code; that is, it is not isomorphic to a cycle. The case when the graph is isomorphic to a cycle can be treated easily in a separate branch of the execution.

Several upper level primitives are exploited by the *BIBOX* algorithm. It is possible to make any vertex unoccupied in a connected graph (especially in a bi-connected one) – implemented by procedure *Make-Unoccupied*. Let v be a vertex to be made unoccupied. A path ϕ connecting v and some of the unoccupied vertices avoiding the locked vertices is found. Then agents along the path ϕ are shifted towards the currently unoccupied vertex.

An operation of moving an agent into an unoccupied vertex is implemented by a procedure *Move-Agent-Unoccupied* – the meaning is that the unoccupied space and the agent are swapped. The procedure also updates functions S_A and Φ_A to reflect the new arrangement of agents and constructs the next arrangement S_A^ζ for the output solution sequence.

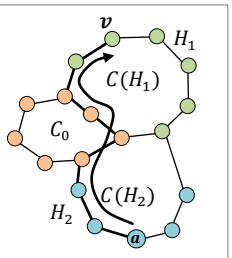
Algorithm 1. *Basic **agent movement** operations.* These operations are used as building blocks for the *BIBOX* algorithm.

procedure *Make-Unoccupied*(v)
/* Makes a vertex v unoccupied while locked vertices remain untouched.
Parameters: v - a vertex to be made unoccupied. */
1: **let** $u \in V$ such that $\Phi_A(u) = \perp$ and u is not locked
2: **let** $\phi = [u = w_1, w_2, \ldots, w_j = v]$ be a (shortest) path
3: ⌊ connecting u and v in G not containing locked vertices
4: **for** $i = 1,2,\ldots,j-1$ **do**
5: ⌊ Move-Agent-Unoccupied(w_{i+1}, w_i)



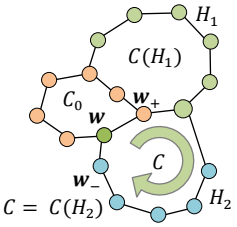
Vertex w is locked; u is unoccupied; pebbles are shifted along cycle $C(H_1)$ to make v unoccupied.

procedure *Move-Agent*(a, v)
/* Moves an agent$a$ into a vertex v avoiding locked vertices.
Parameters: a – an agent to move,
v - a target vertex.*/
/* complexity issues impose special selection of φ */
1: **let** $\varphi = [S_A(a) = w_1^\varphi, w_2^\varphi, \ldots, w_{j_\varphi}^\varphi = v]$ be a path
2: ⎪ connecting $S_A(a)$ and v in G not containing
3: ⌊ locked vertices
4: **for** $i = 1,2,\ldots,j_\varphi - 1$ **do**
5: ⎪ Lock $(\{w_i^\varphi\})$
6: ⎪ Make-Unoccupied(w_{i+1}^φ)
7: ⎪ Unlock$(\{w_i^\varphi\})$
8: ⌊ Move-Agent-Unoccupied$(w_i^\varphi, w_{i+1}^\varphi)$



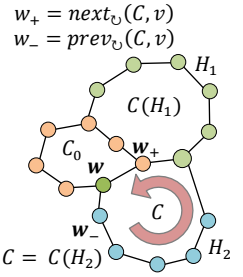
Agent a is moved to v through cycles $C(H_2)$, C_0, and $C(H_1)$.

procedure *Rotate-Cycle*$^+(C, w)$
/* Rotates agents in a cycle C in the positive direction.
Parameters: C - a cycle to rotate
w - unoccupied vertex, $w \in C$. */
1: **for** $i = 1,2,\ldots,|C|$ **do**
2: ⎪ Move-Agent-Unoccupied$(prev_\circlearrowleft(C, w), w)$
3: ⌊ $w \leftarrow prev_\circlearrowleft(C, w)$



Vertex w is unoccupied; $C(H_2)$ is rotated in the positive direction.

procedure *Rotate-Cycle*$^-(C, w)$
/* Rotates agents in the cycle C in the negative direction.
Parameters: C - a cycle to rotate, $w \in C$. */
1: **let** $w \in C$ such that $\Phi_A(w) = \perp$ and w is not locked
2: **for** $i = 1,2,\ldots,|C|$ **do**
3: ⎪ Move-Agent-Unoccupied$(next_\circlearrowleft(C, w), w)$
4: ⌊ $w \leftarrow next_\circlearrowleft(C, w)$



Vertex w is unoccupied; $C(H_2)$ is rotated in the negative direction.

procedure *Move-Agent-Unoccupied*(u, v)
/* Swaps agent and the unoccupied space; vertex v is supposed to be unoccupied; u contains an agent.
Parameters: u, v – vertices between which agent is moved. */
1: $S_P(\Phi_P(u)) \leftarrow v$
2: $\Phi_P(v) \leftarrow \Phi_P(u)$
3: $\Phi_P(u) \leftarrow \perp$
4: $S_P^\xi \leftarrow S_P$
5: $\xi \leftarrow \xi + 1$

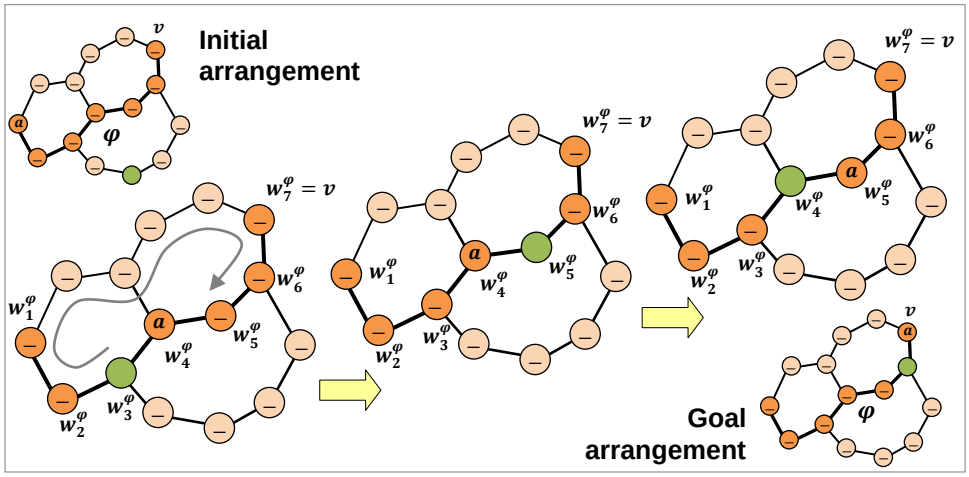


Figure 4. *An illustration of **moving** an agent in bi-connected graph.* The task is to move an agent a from the initial position to a vertex v. A paths φ connecting the initial position of the agent a with v is found (the path is distinguished by color). It is then traversed by the agent a while the unoccupied vertex is restored in front of a after every edge traversal. This is possible thanks to bi-connectivity of the graph – a path connecting unoccupied vertex and the target vertex avoiding the vertex containing a must always exist. The symbol _ stands for an anonymous agent.

The next important process is moving an agent into a given target vertex. It is implemented by a procedure *Move-Agent*. Let an agent a be moved to a vertex v. A path φ is found such that it connects vertices $S_A(a)$ (which is a vertex currently occupied by a) and v.

Edges of φ are then traversed by an agent a. A vertex on φ just in front of a with respect to the direction of the movement is made unoccupied every time a needs to traverse an edge of φ. The agent a should not move during relocation of the unoccupied vertex therefore it is locked before the relocation of the unoccupied vertex starts. Thus, a path along that the unoccupied vertex is moved must avoid the vertex containing a. Such a path always exists due to the bi-connectivity of the graph in which the relocation of the agent a takes place (see Figure 4 for illustration).

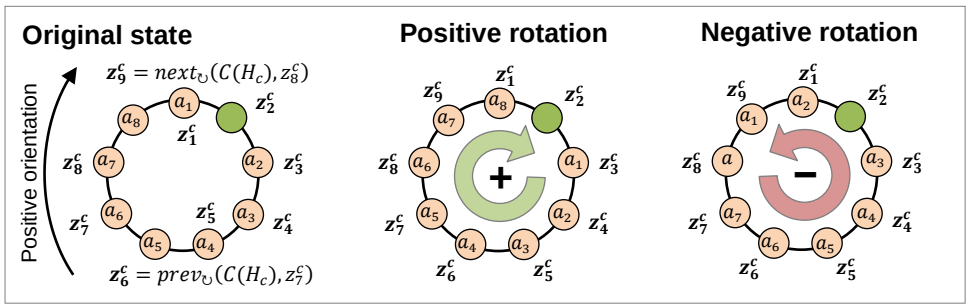
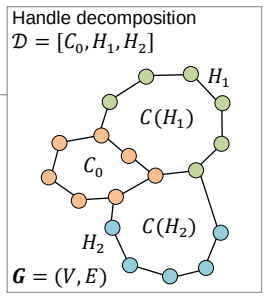


Figure 5. *An illustration of **rotation** of agents along a cycle.* An orientation of the cycle is determined by functions $next_{\circlearrowleft}$ and $prev_{\circlearrowleft}$. There is a single unoccupied vertex in the cycle to enable the rotation.

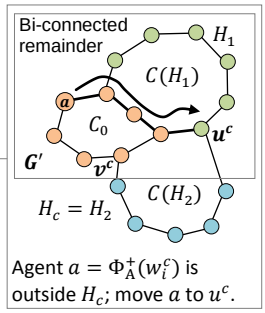
The last basic operation is a rotation of agents along a cycle (see Figure 5). This operation is implemented by procedures *Rotate-Cycle*$^+$ and *Rotate-Cycle*$^-$. The former rotates agents in the positive direction and the latter rotates agents in the negative direction. It supposed that at least one vertex in the given input cycle is unoccupied and it is given as the parameter. The input unoccupied vertex enables the rotation; it remains on its place after the rotation is finished.

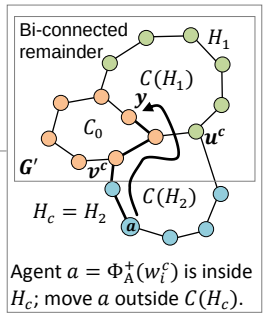
Algorithm 2. *The **BIBOX** algorithm.* The pseudo-code is built around operations from Algorithm 1. It solves a given agent motion problem on a non-trivial bi-connected graph with exactly two unoccupied vertices. The algorithm proceeds inductively according to the handle decomposition of the graph of the input instance. The two unoccupied vertices are necessary for arranging agents within the initial cycle of the handle decomposition.

function *BIBOX-Solve*$(G = (V, E), P, S_A^0, S_A^+)$: **pair**
  /* Top level function of the BIBOX algorithm; solves a given problem of agent motion on a graph.
  Parameters: G - a graph modeling the environment,
  A- a set of agents,
  S_A^0 - a initial arrangement of agents,
  S_A^+- a goal arrangement of agents. */
1: **let** $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of G
2: $(S_A^+, \varphi, \chi) \leftarrow$ Transform-Goal(G, P, S_A^+)
3: $S_A \leftarrow S_A^0$
4: $\xi \leftarrow 1$
5: **for** $c = d, d-1, \ldots, 1$ **do**
6: | **if** $|H_c| > 2$ **then**
7: | | Solve-Regular-Handle(c)
8: Solve-Original-Cycle
9: Finish-Solution(φ, χ)
10: **return**$(\xi, [S_A^0, S_A^1, \ldots, S_A^\zeta])$

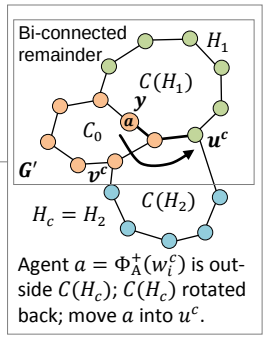


procedure *Solve-Regular-Handle*(c)
  /* Places agents which destinations are within a handle H_c; agents placed in the handle H_c are finally locked so they cannot move any more.
  Parameters: c – the index of a handle */
1: **let** $[u^j, w_1^j, w_2^j, \ldots, w_{h_c}^j, v^j] = H_j \; \forall j \in \{1,2,\ldots,d\}$
  /* Both unoccupied vertices must be located outside the currently solved handle. */
2: **let** $w, z \in V \setminus \bigcup_{j=c}^{d}(H_j \setminus \{u^j, v^j\})$ such that $w \neq z$
3: Make-Unoccupied(w)
4: Lock$(\{w\})$
5: Make-Unoccupied(z)
6: Unlock$(\{w\})$
7: **for** $i = h_c, h_c - 1, \ldots, 1$ **do**
8: | Lock$(H_c \setminus \{u^c, v^c\})$
  | /* An agent to be placed is outside the handle H_c. */
9: | **if** $S_P(\Phi_A^+(w_i^c)) \notin (H_c \setminus \{u^c, v^c\})$ **then**
10: | | Move-Agent$(\Phi_A^+(w_i^c), u^c)$
11: | | Lock$(\{u^c\})$
12: | | Make-Unoccupied(v^c)
13: | | Unlock(H_c)
14: | | Rotate-Cycle$^+(C(H_c), v^c)$
  | /* An agent to be placed is inside the handle H_c. */
15: | **else**
16: | | Make-Unoccupied(u^c)
17: | | Unlock(H_c)
18: | | $\rho \leftarrow 0$
19: | | **while** $S_P(\Phi_A^+(w_i^c)) \neq v^c$ **do**
20: | | | Rotate-Cycle$^+(C(H_c), u^c)$
21: | | | $\rho \leftarrow \rho + 1$
22: | | Lock$(H_c \setminus \{u^c, v^c\})$
23: | | **let** $y \in V \setminus (\bigcup_{j=c+1}^{d}(H_j \setminus \{u^j, v^j\}) \cup C(H_c))$
24: | | Move-Agent$(\Phi_A^+(w_i^c), y)$
25: | | Lock $(\{y\})$





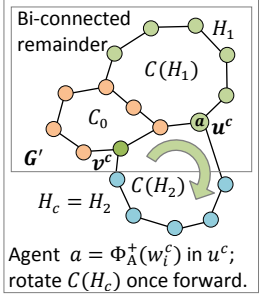
```
26:     Make-Unoccupied(u^c)
27:     Unlock(H_c)
28:     while ρ > 0 do
29:         Rotate-Cycle⁻(C(H_c), u^c)
30:         ρ ← ρ − 1
31:     Unlock({y})
32:     Lock(H_c \ {u^c, v^c})
33:     Move-Agent(Φ_A^+(w_i^c), u^c)
34:     Lock ({u^c})
35:     Make-Unoccupied(v^c)
36:     Unlock(H_c)
37:     Rotate-Cycle⁺(C(H_c), v^c)
38: Lock(H_c \ {u^c, v^c})
```



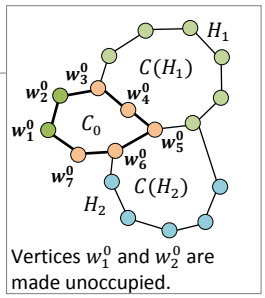
Agent $a = \Phi_A^+(w_i^c)$ is outside $C(H_c)$; $C(H_c)$ rotated back; move a into u^c.

procedure *Solve-Original-Cycle*

```
    /* Places agents which destinations are within the
    initial cycle; it is assumed that unoccupied vertices
    of the goal arrangement of agents are located within
    the initial cycle. */
1:  let u ∈ C_0 and v ∈ V \ C_0 such that {u, v} ∈ E
2:  let [w_1^0, w_2^0, ..., w_l^0] = C_0
    /* According to the assumption on the goal arrangement
    it holds that Φ_A^+(w_1^0) =⊥ and Φ_A^+(w_2^0) =⊥. */
3:  for i = 3,4, ..., l do
4:      Make-Unoccupied(w_1^0)
5:      Lock({w_1^0})
6:      Make-Unoccupied(w_2^0)
7:      Unlock({w_1^0})
8:      if Φ_A^+(w_i^0) ≠ Φ_A(w_i^0) then
9:          Exchange-Agents (Φ_A^+(w_i^0), Φ_A(w_i^0), u, v)
10: Make-Unoccupied(w_1^0)
11: Lock({w_1^0})
12: Make-Unoccupied(w_2^0)
13: Unlock({w_1^0})
```



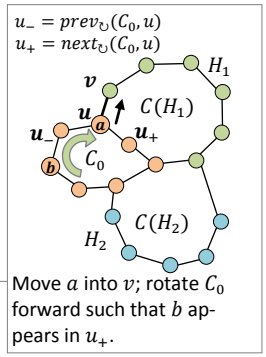
Agent $a = \Phi_A^+(w_i^c)$ in u^c; rotate $C(H_c)$ once forward.



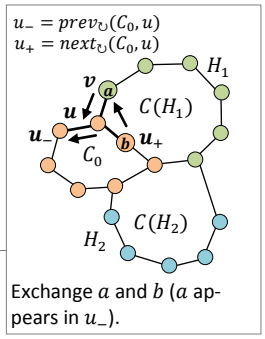
Vertices w_1^0 and w_2^0 are made unoccupied.

procedure *Exchange-Agents*(a, b, u, v)

```
    /* Exchanges a pair of agents within the initial
    cycle of the handle decomposition.
    Parameters:  a, b - a pair of agents to be exchanged,
                 u, v - a pair of neighboring vertices where
                 v is used as a storage space. */
1:  c ← Φ_A(v)
2:  Make-Unoccupied(u)
3:  Move-Agent-Unoccupied(v, u)
4:  while S_A(a) ≠ u do
5:      Rotate-Cycle⁺(C_0, v)
6:  Move-Agent-Unoccupied(u, v)
7:  Lock({u})
8:  Make-Unoccupied(next_↻(C_0, u))
9:  ρ ← 0
10: while S_P(b) ≠ next_↻(C_0, u) do
11:     Rotate-Cycle⁺(C_0, u)
12:     ρ ← ρ + 1
13: Make-Unoccupied(prev_↻(C_0, u))
14: Move-Agent-Unoccupied(v, u)
15: Move-Agent-Unoccupied(u, prev_↻(C_0, u))
16: Move-Agent-Unoccupied(next_↻(C_0, u), u)
17: Move-Agent-Unoccupied(u, v)
```

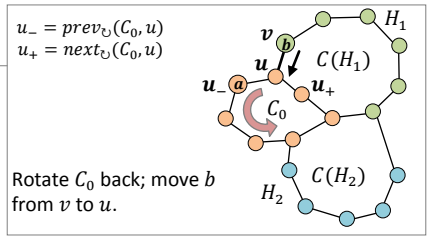


Move a into v; rotate C_0 forward such that b appears in u_+.



Exchange a and b (a appears in u_-).

18: **while** $\rho > 0$ **do**
19: ⎸ Rotate-Cycle$^{-}(C_0, u)$
20: ⎸ $\rho \leftarrow \rho - 1$
21: Move-Agent-Unoccupied(v, u)
22: **while** $S_A(c) \neq u$ **do**
23: ⎸ Rotate-Cycle$^{+}(C_0, v)$
24: Move-Agent-Unoccupied(u, v)
25: Unlock($\{u\}$)



The process of placing agents according to the given goal arrangement is formally described as Algorithm 2. Agents, which goal positions are within the currently solved handle, are placed in a stack like manner. This process is carried out by a procedure *Solve-Regular-Handle* (iteration through the handle is at lines 7-37). Let $H_c = [u^c, w_1^c, w_2^c, \ldots, w_{h_c}^c, v^c]$ for $c \in \{1,2,\ldots,d\}$ be a current handle. Suppose that an agent which goal position is in w_i^c for $i \in \{1,2,\ldots,h_c\}$, that is an agent $\Phi_A^+(w_i^c)$, is processed in the current iteration. Inductively suppose that agents $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$, …,$\Phi_A^+(w_{i+1}^c)$ are located in vertices $w_{h_c-i-1}^c$, $w_{h_c-i-2}^c$, …, w_1^c respectively. An analogical situation for the next agent $\Phi_A^+(w_{i+1}^c)$ must be produced at the end of the iteration.

The agent $\Phi_A^+(w_i^c)$ is moved to the vertex u^c and then the cycle $C(H_c)$ is positively rotated once which causes the agent $\Phi_A^+(w_i^c)$ to move to w_1^c and agents $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…,$\Phi_A^+(w_{i+1}^c)$ stacks in the cycle so that they are located in $w_{h_c-i}^c$, $w_{h_c-i-1}^c$, …, w_2^c. We have just described one iteration of stacking agents into the handle H_c. However, the process has some difficulties. At least, two major cases must be distinguished. In both cases, the first step is that internal vertices of the handle H_c are locked (line 8 of *Solve-Regular-Handle*).

If the agent $\Phi_A^+(w_i^c)$ is not located in the internal vertices of the handle H_c (line 9-14 of *Solve-Regular-Handle*) it is just moved to u^c. This is possible since an invariant holds that both unoccupied vertices are located outside the internal vertices of the handle and the graph without the internal vertices of the handle is connected. This holds at the beginning, since both unoccupied vertices are explicitly moved outside the handle H_c (lines 2-6 of *Solve-Regular-Handle*) and it is preserved through all the iterations. Observe that these movements do not affect agents already stacked in the handle. The agent $\Phi_A^+(w_{h_c}^c)$ is fixed in u^c by locking u^c and then an unoccupied vertex is relocated to v^c which makes the rotation of the cycle $C(H_c)$ possible. The positive rotation of $C(H_c)$ then finishes the iteration.

If the agent $\Phi_A^+(w_i^c)$ is already located in some of the internal vertices of the handle H_c (lines 15-37 of *Solve-Regular-Handle*), the above process is reused but it must be preceded by relocating $\Phi_A^+(w_{h_c}^c)$ outside the handle. The vertex u^c is made unoccupied and the cycle $C(H_c)$ is positively rotated until the agent $\Phi_A^+(w_i^c)$ gets outside the internal vertices of H_c; that is, $\Phi_A^+(w_i^c)$ appears in v^c. Notice, that this series of rotations preserves the order of the already stacked agents. To restore the situation however, the cycle must be rotated back the same number of times. A vertex y outside the already finished part of the graph (that is outside $C(H_c)$ and outside H_j for $j > c$) is selected; the agent $\Phi_A^+(w_i^c)$ is moved into y and it is fixed there by locking.

The vertex u^c is made unoccupied again since the preceding process may move some agent into it (this is possible since w alone cannot rule out the existence of a path from an unoccupied vertex to u^c in the bi-connected graph; there is always an alternative path). The cycle is rotated back so that inductively supposed placement of $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…,$\Phi_A^+(w_{i+1}^c)$ is restored. The situation is now the same as in the previous case with $\Phi_A^+(w_i^c)$ outside the handle.

After the last iteration within the handle H_c it holds that the agents $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…, $\Phi_A^+(w_1^c)$ are located in vertices $w_{h_c}^c$,$w_{h_c-1}^c$,…,w_1^c respectively. Moreover it holds that unoccupied vertices are both outside the internal vertices of H_c. Thus, the solving process can continue with the

next handle in the same way while the already solved handles remain unaffected by the subsequent steps. Notice, that only one unoccupied vertex is sufficient for stacking agents into handles. See Figure 6 for detailed illustration.

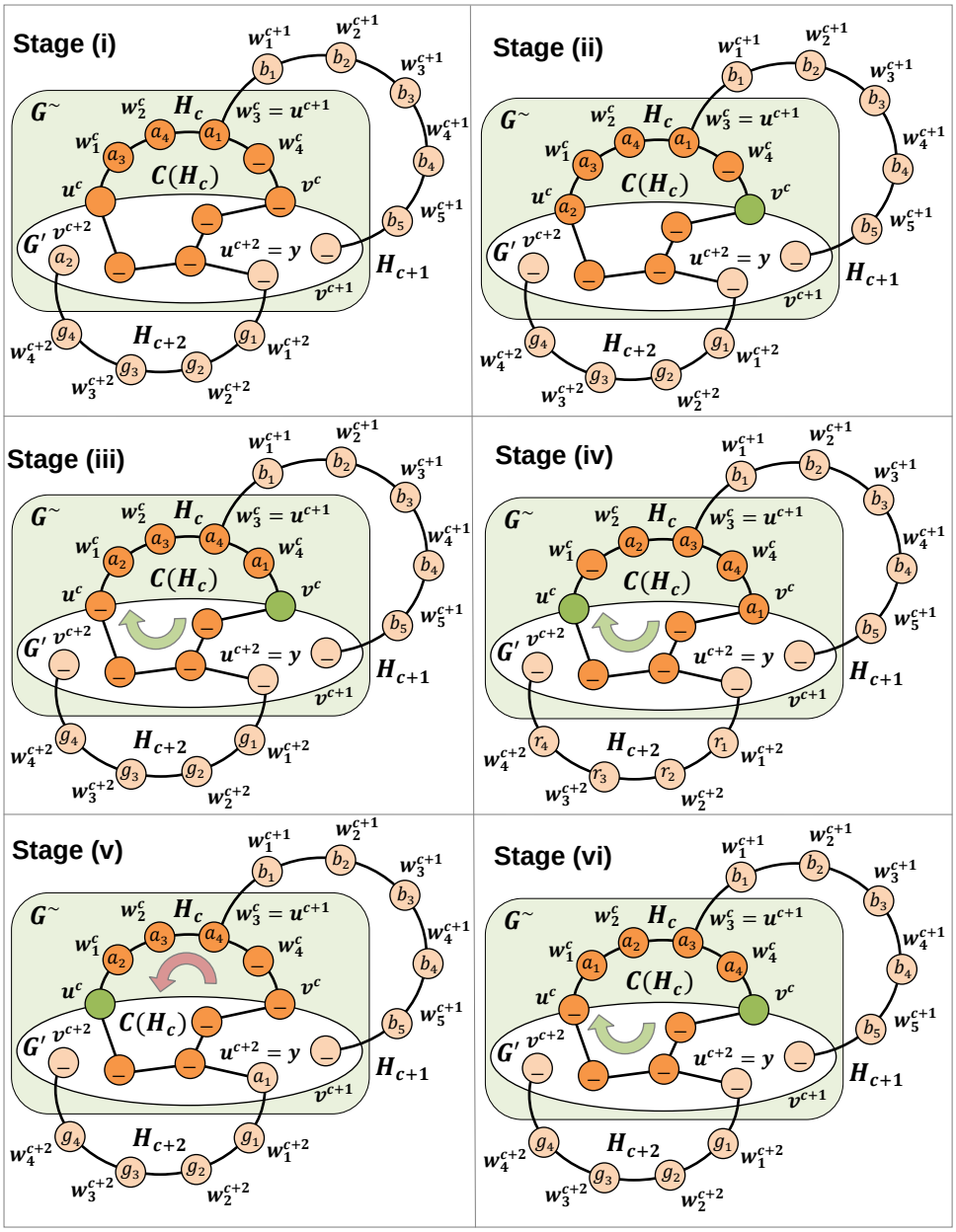


Figure 6. *A process of **stacking an agent** into a handle.* Agents a_1, a_2, a_3, and a_4 are to be stacked into H_c (that is, $S_A^+(a_1) = w_1^c$, $S_A^+(a_2) = w_2^c$, $S_A^+(a_3) = w_3^c$, and $S_A^+(a_4) = w_4^c$); handles H_{c+1} and H_{c+2} are already solved (that is, $S_A^+(b_1) = w_1^{c+1}$,…, $S_A^+(b_5) = w_5^{c+1}$, and $S_A^+(g_1) = w_1^{c+2}$,…, $S_A^+(g_4) = w_4^{c+1}$). Observe that the agent a_2 is originally outside the handle while the agent a_1 is inside. Stage (i) shows situation after agents a_3 and a_4 were stacked into the handle H_c. Then vacant vertex is relocated to the connection vertex v^c; using empty v^c H_c is rotated such that a_1 appears in the second connection vertex u^c (stages (ii), (iii), and (iv)). The handle H_c then needs to be rotated back but before a_1 must be moved outside the cycle associated with H_c (stage (v)). Finally, the agent a_1 is moved to the first connection vertex u^c and H_c is rotated once so that a_1 appears in the first internal vertex of H_c. The symbol _ stands for an anonymous agent.

The initial cycle C_0 of the handle decomposition must be treated in a different way. Here, the second unoccupied vertex is utilized. An arrangement of agents within C_0 can be regarded as a permutation. The task is to obtain the right permutation corresponding to the goal arrangement. This can be achieved by exchanging several pairs of agents. More precisely, if an agent residing in a vertex of C_0 differs from an agent that should reside in this vertex in the goal arrangement, this pair of agents is exchanged. The process is implemented by a procedure *Solve-Original-Cycle* and by an auxiliary procedure *Exchange-Agents* for exchanging a pair of agents.

The procedure *Exchange-Agents* expects that first two vertices of the initial cycle are unoccupied in the current arrangement. However, the function generally does not preserve this property. Hence, the vacancy of the first two vertices of the initial cycle must be repeatedly restored (lines 4-7 and 10-13 of *Solve-Original-Cycle*). The process of exchanging a pair of agents a and b itself exploits a pair of vertices u and v where these two vertices are connected by an edge and it holds that $u \in C_0 \wedge v \notin C_0$. The vertex v is used as an auxiliary storage place.

The need of two unoccupied vertices is imposed by the fact that an agent from C_0 to be stored in v must be rotated into u first. During this process, some vertex of the cycle must be unoccupied to make the rotation possible and the vertex v must be unoccupied as well to make storing possible.

When exchanging the pair of agents a and b it is necessary to preserve ordering of the other vertices. First, an agent occupying the vertex v is moved into the cycle C_0 in order to make v vacant (lines 1-3 of *Exchange-Agents*). Then the cycle is rotated until the agent a appears in u (since there was an agent in u at the beginning of the rotation, there is always some agent in u after all the rotations) and the agent a is stored in v (lines 4-6 of *Exchange-Agents*). Next, the cycle C_0 is rotated positively so that b appears in $next_{\circlearrowleft}(C_0, u)$ (the next vertex to u with respect to the positive orientation) while the number of rotations is recorded (lines 10-12 of *Exchange-Agents*).

Next, agents a and b are exchanged so that ordering of a in the cycle C_0 is the same as of b before the exchange (lines 13-17 of *Exchange-Agents*). Then, the cycle is rotated in the negative direction recorded number of times so that the place within the cycle where a was originally ordered appears in u; thus b is ordered here (lines 18-20 of *Exchange-Agents*). Finally, the agent that has been located in v before the exchange of agents a and b, is put back into v (lines 22-25 of *Exchange-Agents*).

3.1.3. *Summary of Theoretical Properties and Real-life Extensions*

As the proof of soundness and completeness of the *BIBOX* algorithm are mainly technical, we refer the reader to the appendix where detailed proofs can be found. Regarding the proof of soundness it is necessary to verify that the following step of the algorithm is always defined particularly at non-deterministic steps where existence of some object – vertex or path – is required (this concerns for example existence of paths at lines 1-3 of *Move-Agent*). Some special care needs to be devoted to verifying that its existence is ensured in the unlocked part of the graph.

It can be shown that the worst-case time complexity of the *BIBOX* algorithm is $\mathcal{O}(|V|^3)$ with respect to the input graph $G = (V, E)$. Again, the detailed proof can be found in the appendix. It needs to be observed that at most $|A|$ agents need to be placed in regular handles. Each agent placement in the handle requires $\mathcal{O}(|V|)$ rotations of the handle and the constant number of relocations of agents (*Move-Agent*). It is not difficult to observe that single rotation by one position requires $\mathcal{O}(|V|)$ steps hence we have $\mathcal{O}(|V|^2)$ steps per handle rotations. For each relocation of an agent, two vertex disjoint paths need to be found which can be done in worst-case time of $\mathcal{O}(|V|)$. Then agent needs to traverse the path. In the worst-case, $\mathcal{O}(|V|)$ edges need to be traversed. An unoccupied vertex needs to be moved in front of the agent per each edge traversal. This has to be done carefully – for example, we cannot afford to search for a path to the front of the agent in the original graph, as it is too much

time consuming. Fortunately, the relocation of the unoccupied vertex can be carried out in $\mathcal{O}(|V|)$ steps using the knowledge of the handle decomposition. In total, we have time of $\mathcal{O}(|A||V|^2)$ for placing agents into regular handles.

Regarding the initial cycle, it is needed to observe that at most $|A|^2$ exchanges of pairs of agents are needed while the single exchange consumes $\mathcal{O}(|V|)$ steps. Altogether, the worst-case time complexity is $\mathcal{O}(|V|^3)$. Exactly the same calculation can be done for determining the total number of moves which is also $\mathcal{O}(|V|^3)$. As the total number of moves is the upper bound for the makespan, the makespan of generated solution is $\mathcal{O}(|V|^3)$ as well.

The natural question is how to apply the *BIBOX* algorithm if there are more than two unoccupied vertices in the input instance (that is, $|A| < |V| - 2$). It is easy to adapt the algorithm to utilize additional unoccupied vertices when it is suitable and to ignore them if they are to cause unnecessary movement. The utilizing additional unoccupied vertices is done through replacement of the non-deterministic selection of an unlocked unoccupied vertex (such as that at line 1 of *Make-Unoccupied*) by the selection of the nearest one (this is also done in the real implementation). On the other hand, if for example rotation of a handle is to be done due to unoccupied position in the handle, which is redundant in fact, then such a movement is automatically ignored. More details about this adaptation of the algorithm for sparse environments are given in [27].

Some further optimizations should be used in the real-life implementation to reduce the makespan of the produced solution. Here, various assumption are explicitly enforced in order to make the pseudo-code simpler (for example, the precondition of having first two vertices of the initial cycle of the handle decomposition unoccupied before a pair of vertices is exchanged within the cycle - lines 4-6 of *Solve-Original-Cycle*). This approach should be avoided and lazier approach should be adopted in the real-life implementation (in the case of exchanging agents, locations of unoccupied vertices should be detected implicitly in subsequent steps by more sophisticated branching of the code). These kind of more complex branching of the algorithm is used in the experimental implementation.

The real-life implementation of procedures *Solve-Regular-Handle* and *Solve-Original-Cycle* should also use more opportunistic selection of vertices to store agents (vertex y - line 23 of *Solve-Regular-Handle* and vertices u, v - line 1 of *Solve-Original-Cycle*). The nearest vertex to the target agent should be always used. Moreover, selection of these vertices within the procedure *Solve-Original-Cycle* should be done not only at the beginning, but also in every iteration of the main loop.

3.1.4. *Making Solution Parallel*

A simple post-processing step needs to be done to obtain parallel solution of pCPF. Suppose to have a solution of Σ – denoted as $S_A^{\prec}(\Sigma)$ – as a sequence of moves; that is, $S_A^{\prec}(\Sigma) = [a_1: u_1 \to v_1; a_2: u_2 \to v_2; \dots; a_\zeta: u_\zeta \to v_\zeta]$ with $a_i \in A$ and $u_i, v_i \in V$ meaning that an agent a_i moves from u_i to v_i at time step i. Actually, such a sequential solution is produced by the *BIBOX* algorithm. Now, we need to distinguish which pairs of moves can be executed in parallel and which must be executed one by one sequentially. Following two definitions captures this intuition.

Definition 5 ***(concurrent moves).*** A move $a_k: u_k \to v_k;\ k \in \{1,2,\dots,\zeta\}$ is *concurrent* with a move $a_h: u_h \to v_h;\ h \in \{1,2,\dots,\zeta\}$ with $h < k$ if $a_h \neq a_k,\ u_h = v_k \wedge v_h \neq u_k$, and there is no other move $r_\hbar: u_\hbar \to v_\hbar$ in $S_A^{\prec}(\Sigma)$ with $h < \hbar < k$ such that $\{u_\hbar, v_\hbar\} \cap \{u_h, v_h, u_k, v_k\} \neq \emptyset$. Concurrent move are denoted as $r_h: u_h \to v_h \preccurlyeq r_k: u_k \to v_k$. □

The definition captures the fact that although the moves are interfering they can be executed at the same time. The relation of concurrence is anti-reflexive due to the requirement on different agents involved and anti-symmetric due to the ordering of moves within the sequential solution.

Definition 6 ***(dependent moves).*** A move $a_k: u_k \rightarrow v_k;\ k \in \{1,2,\ldots,\zeta\}$ is *dependent* on a move $a_h: u_h \rightarrow v_h;\ h \in \{1,2,\ldots,\zeta\}$ with $h < k$ if $\{u_h, v_h\} \cap \{u_k, v_k\} \neq \emptyset$, either $a_h = a_k$ or $u_h \neq v_k \vee v_h = u_k$, and there is no other move $a_\hbar: u_\hbar \rightarrow v_\hbar$ in $\mathcal{S}_A^{\prec}(\Sigma)$ such that $h < \hbar < k$ such that $\{u_\hbar, v_\hbar\} \cap \{u_h, v_h, u_k, v_k\} \neq \emptyset$. The notation of dependence is $a_h: u_h \rightarrow v_h \prec r_k: u_k \rightarrow v_k$. □

The relation of dependence of moves is reflexive and anti-symmetric due to the ordering of moves within the sequential solution. It puts into relation moves that must be executed sequentially as they either concern the same agent or they interfere spatially through shared vertices. Notice that the definition of dependence is complementary to the definition of concurrence.

It is not difficult to show that every function $t: \bigcup \mathcal{S}_R^{\prec}(\Sigma) \longrightarrow \{1,2,\ldots,\zeta\}$ that satisfies conditions that $t(a_h: u_h \rightarrow v_h) < t(a_k: u_k \rightarrow v_k)$ whenever $a_h: u_h \rightarrow v_h \prec a_k: u_k \rightarrow v_k$ and $t(a_h: u_h \rightarrow v_h) \leq t(a_k: u_k \rightarrow v_k)$ whenever $a_h: u_h \rightarrow v_h \preccurlyeq a_k: u_k \rightarrow v_k$ correctly assigns execution time steps to moves with respect to the definition of pCPF. Particular time-step assignment function t can be found by the *critical path* method [14] for instance. Schedule obtained from the critical path method is optimal in certain sense – details are discussed in [27].

3.2. **BIBOX-θ: An Algorithm for a Bi-connected Graphs Exploiting Optimal Macros**

The drawback of the *BIBOX* algorithm is that it requires at least two unoccupied vertices. Observe that the second unoccupied vertex is necessary only in the last stage where agents are placed into the initial cycle. Thus, if there is only one unoccupied vertex, the *BIBOX* algorithm would be able to place almost all the agents except those whose goal positions are within the initial cycle.

It is possible to apply the *MIT* algorithm [8] to finish placement of agents in the initial cycle. The *MIT* algorithm is capable of solving instances on all the non-trivial bi-connected graphs with just one unoccupied vertex (the instance with just one unoccupied vertex may be unsolvable; indeed, the *MIT* algorithm can detect such a case). Thus if we combine both algorithms, the combined algorithm can proceed as *BIBOX* for placing agents into all the internal vertices of handles and it can proceed as *MIT* over the remaining initial cycle and the first handle. Unfortunately, the process how *MIT* places agents generates excessively long sequences of moves (see experiments in Section 4).

Despite above facts the idea of using alternative solving process for the initial cycle is still promising. Since the initial cycle and the first handle constitute a structurally simple graph (these graphs are called *θ-like graphs* in the following text), it is feasible to try to solve selected instances of pCPF over these graphs makespan optimally. The good candidate instances for optimal solving are those from which an overall solution of any instance over the graph can be composed. Moreover, the optimal solutions to selected instances can be pre-computed and stored in the database for future use. Since solutions from that the overall solution is composed are optimal, it is reasonable to expect that the makespan of the resulting solution will be short as well. Nevertheless, this is a conjecture that should be proven.

3.2.1. *Algebraic Foundation of the Algorithm*

The bi-connected graph, whose handle decomposition consists of an initial cycle and a single handle, represents structurally the simplest bi-connected graphs over that the non-trivial rearrangement of

agents is possible supposed there is a single unoccupied vertex (the structurally simpler bi-connected graph is a cycle where only rotations of agents are possible). These graphs will be referred to as *θ-like graphs*.

Definition 7 ***(θ-like graph).*** Let $X = [\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_\alpha]$, $B = [\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_\beta]$, and $Z = [\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_\gamma]$ be three sequences of vertices satisfying that $\alpha \geq 1 \wedge \beta \geq 2 \wedge \gamma \geq 1$. An undirected graph $\theta(X, Y, Z) = (V_\theta, E_\theta)$ for such three sets is constructed as follows: $V_\theta = X \cup Y \cup Z$ and $E_\theta = \{\{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_2, \bar{x}_3\}, \ldots, \{\bar{x}_{\alpha-1}, \bar{x}_\alpha\}; \{\bar{y}_1, \bar{y}_2\}, \{\bar{y}_2, \bar{y}_3\}, \ldots, \{\bar{y}_{\beta-1}, \bar{y}_\beta\}; \{\bar{z}_1, \bar{z}_2\}, \{\bar{z}_2, \bar{z}_3\}, \ldots, \{\bar{z}_{\gamma-1}, \bar{z}_\gamma\}; \{\bar{x}_1, \bar{y}_1\}, \{\bar{y}_\beta, \bar{z}_\gamma\}, \{\bar{y}_1, \bar{z}_1\}, \{\bar{x}_\alpha, \bar{y}_\beta\}\}$. An undirected graph $G = (V, E)$ is called a *θ-like graph* if there exist three sets of vertices X, Y, and Z as above such that G is isomorphic to $\theta(X, Y, Z)$. □

The notation of the set union is used over sequences in the definition of the set of vertices V_θ. This is an abbreviation for the union of ranges of individual sequences. Notice that $\theta(X, Y, Z)$ itself is a θ-like graph and $\theta(X, Y, Z)$ may be identical to G if sets X, Y, and Z consist of vertices of G. Hence, no distinction is made between G and $\theta(X, Y, Z)$ in the following text and the notation $\theta(X, Y, Z)$ is used exclusively. An example of θ-like graph is shown in Figure 7.

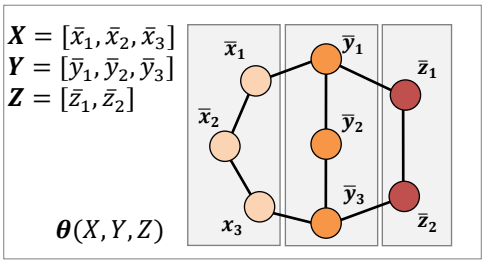


Figure 7. *An example of* ***θ-like graph***. θ-like graphs are bi-connected graphs consisting of a cycle and one handle.

There are $\mathcal{O}(|V|^2)$ non-isomorphic θ-like graphs over a set of vertices V (consider the set V linearly ordered and partitioned over sub-sets X, Y, and Z, where these sub-sets form continuous sub-sequences within the ordered V; there is $\mathcal{O}(|V|^2)$ possibilities to place separation points among X, Y, and Z). However, the number of all the possible instances of CPF with a single unoccupied vertex on a fixed θ-like graph $\theta = (V_\theta, E_\theta)$ is $|V_\theta|!$ since the difference between the initial and the goal arrangement can be regarded as a permutation of $|V_\theta|$ elements. Hence, it is not feasible to pre-compute and to store optimal solutions to all the instances of the problem on a fixed θ-like graph. The number of selected instances should be bounded polynomially to make their pre-computation and storing feasible. At the same time, it should be possible to compose solution to any instance over the θ-like graph from the solutions to selected instances.

Without loss of generality, assume that the unoccupied vertex within the initial and the goal arrangement of an instance over $\theta = (V_\theta, E_\theta)$ is always $\bar{y}_1$ (the unoccupied vertex can be simply relocated to any vertex). The algebraic structure of such instances over θ is isomorphic to the group of all the permutations of $|V_\theta| - 1$ elements which is called a *symmetric group* on $|V_\theta| - 1$ elements and it is denoted $Sym(|V_\theta| - 1)$ [2, 17].

A ***transposition*** is a permutation, which exchanges a pair of elements and keeps other elements fixed. It is well known that $Sym(|V_\theta| - 1)$ can be generated by the set of transpositions. A permutation is called *odd* if it can be composed of an odd number of transpositions. A permutation is called *even* if it can be composed of an even number of transpositions. A permutation is either odd or even but not both. In fact, if a permutation is assigned a *sign* by a sgn function which is $+1$ if the permutation is even and -1 if the permutation is odd, then sgn represents a *group homomorphism* between $Sym(|V_\theta| - 1)$ and the group $(\{-1, +1\}, *, +1, -)$ where multiplication $*$ corresponds to the product of two permutations, neutral element $+1$ corresponds to the identical permutation and unary minus $-$ corresponds to the inverse permutation.

Another simple fact, that can be derived from above statements, is that the set of all an even permutations on the same set of elements forms a proper sub-group of $Sym(|V_\theta| - 1)$; it is called an *alternating group* on $|V_\theta| - 1$ elements and it is denoted as $Alt(|V_\theta| - 1)$.

A *rotation along a 3-cycle* is a permutation that rotates given three elements and keeps other fixed. It is easy to compose any even permutation from rotations along 3-cycles on the same set of elements [8]. As rotation along a 3-cycle itself it is an even permutation it can never generate an odd permutation.

The number of distinct transpositions over n elements is $\mathcal{O}(n^2)$ and the number of distinct rotations along 3-cycles over n elements is $\mathcal{O}(n^3)$. This is polynomial hence, optimal solutions of corresponding instances seem to be good candidates for storing into the database. Moreover, if the corresponding instances are solvable, then they satisfy the property that a solution to any (in the case of transpositions) or almost any (in the case of 3-cycle rotations) instance on the same graph can be composed of them.

Suppose to have a θ-like graph $\theta(X,Y,Z) = (V_\theta, E_\theta)$ with $X = [\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_\alpha]$, $Y = [\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_\beta]$, and $Z = [\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_\gamma]$ and a set of agents $A = \{a_1, a_2, \ldots, a_{|V_\theta|-1}\}$ for the following three definitions.

Definition 8 ***(even and odd case).*** Let S_A^0 be an initial arrangement of agents such that $S_A^0(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is initially unoccupied) and let S_A^+ be a goal arrangement of agents such that $S_A^+(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is finally unoccupied). If S_A^+ forms an even permutation with respect to S_A^0, then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called an *even case*. If S_A^+ forms an odd permutation with respect to S_A^0, then the instance Σ is called an *odd case*. □

Definition 9 ***(transposition case).*** Let S_A^0 be an initial arrangement such that $S_A^0(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is initially unoccupied) and let S_A^+ be a goal arrangement such that there exist $b_1, b_2 \in A$ with $b_1 \neq b_2$ for which it holds that $S_A^+(b_1) = S_A^0(b_2) \wedge S_A^+(b_2) = S_A^0(b_1) \wedge (\forall a \in P)(a \neq b_1 \wedge a \neq b_2) \Rightarrow S_A^+(a) = S_A^0(a)$ (agents b_1 and b_2 are to be exchanged while locations of other agents are kept; consequently $\bar{y}_1$ is finally unoccupied). Then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called a *transposition case* with respect to b_1 and b_2. □

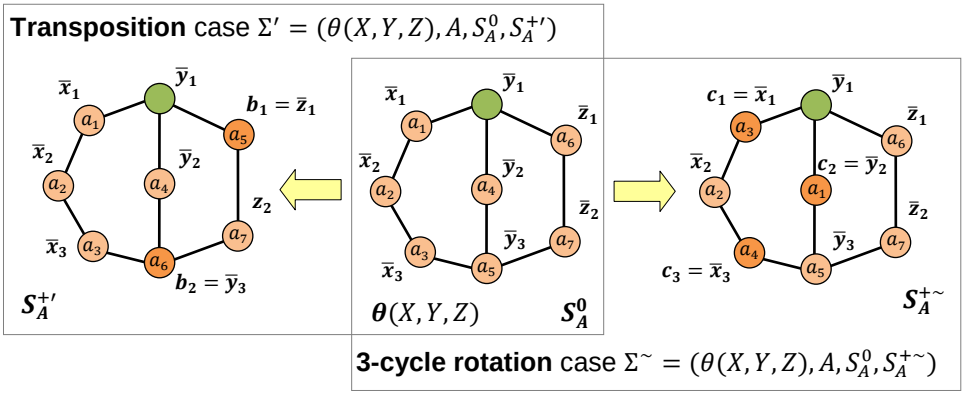


Figure 8. *An example of* ***transposition*** *and* ***3-cycle*** *rotation cases a θ-like graph.* The transposition case is shown for vertices $b_1 = \bar{z}_1$ and $b_2 = \bar{y}_3$. The 3-cycle rotation case is shown for vertices $c_1 = \bar{x}_1$, $c_2 = \bar{y}_2$, and $c_3 = \bar{x}_3$. A solution to any instance over θ-like graph with one vertex unoccupied can be composed of solutions to transposition and 3-cycle rotation cases.

Definition 10 ***(3-cycle rotation case).*** Let S_A^0 be an initial arrangement such that $S_A^0(a) \neq \bar{y}_1 \ \forall a \in A$ ($\bar{y}_1$ is initially unoccupied). Let S_A^+ be a goal arrangement such that there exist pair wise distinct $c_1, c_2, c_3 \in A$ and it holds that $S_A^+(c_1) = S_A^0(c_2) \wedge S_A^+(c_2) = S_A^0(c_3) \wedge S_A^+(c_3) = S_A^0(c_1) \wedge (\forall a \in A)(a \neq c_1 \wedge a \neq c_2 \wedge a \neq c_3) \Rightarrow S_A^+(a) = S_A^0(a)$ (agents c_1, c_2, and c_3 are to be rotated while positions of other agents are kept; $\bar{y}_1$ is finally unoccupied). Then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called a *3-cycle rotation case* with respect to c_1, c_2, and c_3. □

See Figure 8 for illustrations of transposition case and 3-cycle rotation case. Notice, that both cases would be worthless if they are not solvable. Fortunately, several positive results regarding solvability of these cases are shown in [8]. Following propositions and corollaries recall some of them (without proofs).

Proposition 2 ***(solvability of an odd case)*** [8]**.** An odd case of pCPF $\Sigma = (\theta(X, Y, Z), A, S_A^0, S_A^+)$ with $|X| \neq 2 \vee |Y| \neq 3 \vee |Z| \neq 2$ is solvable if and only if θ contains a cycle of an odd length. ■

Let the θ-like graph $\theta(X, Y, Z)$ with $|X| = 2 \wedge |Y| = 3 \wedge |Z| = 2$ be denoted as $\theta(2,3,2)$. It represents a special case where some instances over it are solvable and some are unsolvable. The case of $\theta(2,3,2)$ will be treated separately.

Since the transposition is an odd permutation, the following corollary is a direct consequence of the above proposition.

Corollary 1 ***(solvability of transposition case)*** [8]**.** A transposition case $\Sigma = (\theta(X, Y, Z), A, S_A^0, S_A^+)$ with $\theta(X, Y, Z)$ non-isomorphic to $\theta(2,3,2)$ is solvable if and only if $\theta(X, Y, Z)$ contains a cycle of an odd length. ■

Proposition 3 ***(solvability of an even case)*** [8]**.** An even case $\Sigma = (\theta(X, Y, Z), A, S_A^0, S_A^+)$ with $\theta(X, Y, Z)$ non-isomorphic to $\theta(2,3,2)$ is always solvable. ■

Analogically, since rotation along 3-cycle is an even permutation, the following corollary is a direct consequence of the above proposition.

Corollary 2 ***(solvability of 3-cycle rotation case)*** [8]**.** A 3-cycle rotation case $\Sigma = (\theta(X, Y, Z), A, S_A^0, S_A^+)$ with $\theta(X, Y, Z)$ non-isomorphic to $\theta(2,3,2)$ is always solvable. ■

Similar results hold not only for θ-like graphs, but also for the more general class of non-trivial **bi-connected** graphs non-isomorphic to $\theta(2,3,2)$ [8]. The important properties directly exploited by the algorithm are that if the input graph does not contain a cycle of an odd length and the initial and the goal arrangement of agents form an odd permutation then the instance is unsolvable. Similarly, if the input and the goal arrangements form an even permutation (and the input graph is non-isomorphic to $\theta(2,3,2)$) then the instance is always solvable (observe that, this is the corollary of the *BIBOX* algorithm and Proposition 3).

The following propositions are important with respect to the length of the overall solution composed of the optimal solutions to the transposition cases and 3-cycle rotation cases. Propositions appeared in [2, 8, 17] but most likely they are just a general knowledge.

Proposition 4 ***(solving an odd case).*** A solution to any odd case on a θ-like graph $\theta = (V_\theta, E_\theta)$ can be composed of at most $|V_\theta| - 2$ solutions to transposition cases on the same graph. ■

Similarly, a solution of an even case can be composed of at most $|V_\theta| - 2$ solutions to transposition cases as well.

Proposition 5 ***(solving an even case).*** A solution to any even case on a θ-like graph $\theta = (V_\theta, E_\theta)$ can be composed of at most $|V_\theta| - 2$ solutions to 3-cycle rotation cases on the same graph. ■

Proofs are shown within the pseudo-code of the *BIBOX-θ* algorithm. The above facts justify that transposition and 3-cycle rotation cases are suitable for optimal solving. The corresponding optimal solutions are hence good building blocks for solutions to general instances over θ-like graphs. It is out of scope of this work to give any detailed description of how to compute optimal solutions of instances over θ-like graphs. Applications of several variants of iterative deepening search for this task were studied in [21].

The case of θ-like graph $\theta(2,3,2)$ represents a situation where there is no simple characterization of solvable instances. Since it is a small graph, it is feasible to pre-compute and to store optimal solutions to all the solvable instances over it.

The solving process of the new algorithm over the initial cycle and the first handle is based on the knowledge of how to solve instances over θ-like graphs. In this context, it is necessary to guarantee that insolvability of an sub-instance over $\theta(2,3,2)$ does not contradict solvability of the instance as the whole if the initial cycle and the first handle unluckily become isomorphic to $\theta(2,3,2)$. The following lemma states that this contradictory case can be always avoided. This crucial treatment ensures the upcoming algorithm to proceed correctly. The proof the lemma enumerates all the possible cases and for its length is omitted here (in can be found [27]).

Lemma 3 ***(avoiding*** $\theta(2,3,2)$***).*** If a non-trivial bi-connected graph G is non-isomorphic to $\theta(2,3,2)$ then it subsumes a θ-like sub-graph $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$. Moreover, if G contains an odd cycle then it subsumes $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$ that additionally satisfies that $2 \nmid |X| + |Y|$ (that is, sets X and Y together form an odd cycle). Having a θ-like sub-graph satisfying above conditions, there exists a handle decomposition of $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ of G such that $\theta(X,Y,Z) = C_0 \circ H_1$. ($C_0 \circ H_1$ denotes the sub-graph of G constructed by addition of the handle H_1 to the initial cycle C_0). ■

3.2.2. *Pseudo-code of the BIBOX-θ Algorithm*

The new algorithm is called *BIBOX-θ* according to the concept of θ-like graph. Let $\Sigma = (G = (V,E), A, S_A^0, S_A^+)$ be an input pCPF instance on a bi-connected graph with a single unoccupied vertex. If G is non-isomorphic to $\theta(2,3,2)$ and it subsumes a cycle of an odd length then a handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ of G such that C_0 is of an odd length and $C_0 \circ H_1$ is non-isomorphic to $\theta(2,3,2)$ is computed. Lemma 3 guarantees that this is possible. If G is isomorphic to $\theta(2,3,2)$ then $C_0 \circ H_1$ corresponds to G. If G does not contain an odd cycle then some arbitrary handle decomposition $\mathcal{D}$ is computed.

As in the case of *BIBOX* algorithm, the main phase of the algorithm requires that the finally unoccupied vertex is located in the initial cycle C_0. Thus, a function *Transform-Goal* is applied to modify the goal arrangement S_A^+ by shifting goal locations of agents along a path φ to relocate the unoccu-

pied vertex into C_0. The modified instance is then solved by the process implemented by the *BIBOX-θ* algorithm. The solution is finished by calling a function *Finish-Solution* which shifts agents back along the path φ.

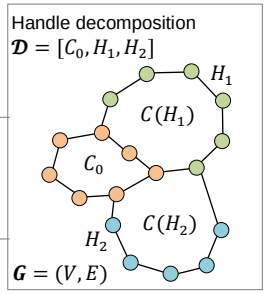
The *BIBOX-θ* algorithm proceeds according to the handle decomposition $\mathcal{D}$ from the last handle H_d to the second handle H_2. The process of placement of agents within the individual handles of the handle decomposition is the same as in the case of the *BIBOX* algorithm. The problem of reaching the goal arrangement of agents within the first handle H_1 and the initial cycle C_0 is solved as an instance over θ-like graph formed by C_0 and H_1. It is supposed that the optimal solutions to all the solvable transposition and 3-cycle rotation cases over θ-like graphs of the size up to the certain limit are pre-computed and stored in the database. Next, it is supposed that the optimal solutions to all the instances over the θ-like graph $\theta(2,3,2)$ are pre-computed into the database as well. A solution to an instance over the θ-like graph is composed of the corresponding optimal solutions stored in the database. If the required record is not stored in the database (which may happen when the size of the θ-like graph is greater than the limit) an alternative solving process must be used. For example, the solving process implemented by the *MIT* algorithm can be employed in such a case.

The pseudo-code of the *BIBOX-θ* algorithm is listed as Algorithm 3. It reuses primitives, functions, and procedures introduced within the context of *BIBOX*. For simplicity, it is supposed that all the required optimal solutions are stored in the database (so there is no treatment when the size of the θ-like graph exceeds the limit).

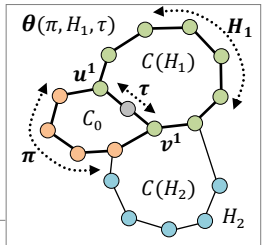
The database with optimal solutions to selected instances over θ-like graphs is represented by three tables: $table_T^\theta$, $table_3^\theta$, and $table_{232}^\theta$. Optimal solutions to transposition cases over a particular θ-like graph θ are stored in the table $table_T^\theta$ – records are addressed by a pair of vertices in which agents are transposed. Similarly, the optimal solutions to 3-cycle rotation cases are stored in the table $table_3^\theta$ – records are addressed by a triple of vertices in which agents are rotated. Finally, the table $table_{232}^\theta$ contains optimal solutions to all the solvable instances over the θ-like graph $\theta(2,3,2)$ - records are addressed by permutations determined by the difference between the initial and the goal arrangement of agents (a function *difference* is used for calculating this differencing permutation).

Algorithm 3. *The **BIBOX-θ** algorithm.* The algorithm solves a given pCPF on a non-trivial bi-connected graph with exactly one unoccupied vertex. It employs a pattern database containing optimal solutions to sub-problems over the initial cycle and the first handle. Functions and procedures from Algorithm 1 and Algorithm 2 are reused here.

function *BIBOX-θ-Solve*$(G = (V, E), A, S_A^0, S_A^+)$: **pair**
  /* Top level function of the BIBOX-θ algorithm; solves
  a given instance of pCPF with a single unoccupied vertex.
  Parameters: G - a graph modeling the environment,
  A - a set of agents,
  S_A^0 - an initial arrangement of agents,
  S_A^+ - a goal arrangement of agents. */
1: **if** G contains a cycle of an odd length **then**
2: **let** $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of G
3: such that C_0 is of an odd length and $C_0 \circ H_1$ is
4: a θ-like sub-graph non-isomorphic to $\theta(2,3,2)$ if possible
  /* if this is not possible then G is isomorphic to $\theta(2,3,2)$ */
5: **else**
6: **let** $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of G
  /* $C_0 \circ H_1$ is always non-isomorphic to $\theta(2,3,2)$ */
7: $(S_P^+, \varphi) \leftarrow$ Transform-Goal(G, A, S_A^+, C_0)
8: $\xi \leftarrow 1$
9: $S_A \leftarrow S_A^0$

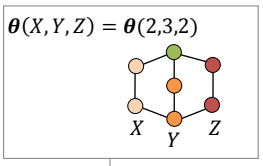


10: **for** $c = d, d-1, \ldots, 2$ **do**
11: ⎸ **if** $|H_c| > 2$ **then**
12: ⎸ ⎸ Solve-Regular-Handle(c)
13: **let** $[u^1, w_1^1, w_2^1, \ldots, w_{h_1}^1, v^1] = H_1$
14: Lock (V)
15: Unlock$(C_0 \cup H_1)$
16: Make-Unoccupied(u^1)
17: **let** π, τ be two vertex disjoint paths connecting
18: ⎸ u^1 and v^1 in C_0
19: $\pi \leftarrow \pi \setminus \{u^1, v^1\}$
20: $\tau \leftarrow \tau \setminus \{u^1, v^1\}$
21: θ-BOX-Solve$(\theta(\pi, H_1, \tau), C_0 \cup H_1, S_P, S_P^+)$
22: Finish-Solution(φ)
23: **return** $(\xi, [S_A^0, S_A^1, \ldots, S_A^\xi])$

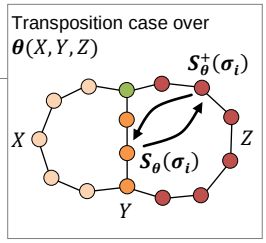


A θ-like graph $\theta(\pi, H_1, \tau)$ matched over C_0 and H_1.

procedure *θ-BOX-Solve*$(\theta(X, Y, Z), V^+, S_\theta^0, S_\theta^+)$
/* Solves a sub-problem over a given θ-like subgraph; a set of goal vertices into which agents must be placed is specified.
Parameters: $\theta(X, Y, Z)$ - a θ-like subgraph modeling the sub-problem
V^+ - a set of goal vertices
S_θ^0 - an initial arrangement of agents
S_θ^+ - a goal arrangement of agents
(only $S_\theta^+|_{V^+}$ is considered) */

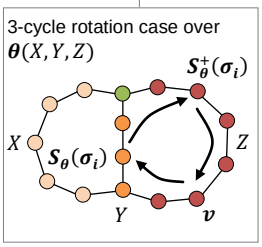


1: **let** $(V_\theta, E_\theta) = \theta(X, Y, Z)$
2: **let** $\{\sigma_1, \sigma_2, \ldots, \sigma_{|V^+|-1}\} = \{\sigma | S_\theta^0(\sigma) \in V^+\}$
3: **if** $|X| = 2 \wedge |Y| = 3 \wedge |Z| = 2$ **then**
4: ⎸ $\omega \leftarrow table_{232}^\theta[difference(S_\theta^0, S_\theta^+)]$
5: ⎸ **if** $\omega = NIL$ **then fail** /* the instance is unsolvable */
6: ⎸ Apply-Macro(ω, S_θ)
7: **else**
8: ⎸ $S_\theta \leftarrow S_\theta^0$
9: ⎸ **if** G_θ contains a cycle of an odd length **then**
10: ⎸ ⎸ **for** $i = 1,2, \ldots, |V^+| - 2$ **do**
11: ⎸ ⎸ ⎸ **if** $S_\theta(\sigma_i) \neq S_\theta^+(\sigma_i)$ **then**
12: ⎸ ⎸ ⎸ ⎸ $S_\theta \leftarrow$ Apply-Macro$(table_T^\theta[S_\theta(\sigma_i), S_\theta^+(\sigma_i)], S_\theta)$
/*G_θ does not contain any odd cycle */



13: ⎸ **else**
14: ⎸ ⎸ **if** S_θ^+ constitutes an odd permutation w.r.t. S_θ **then**
15: ⎸ ⎸ ⎸ **fail** /* the instance is unsolvable */
/* S_θ^+ constitutes an even permutation w.r.t. S_θ */
16: ⎸ ⎸ **else**
17: ⎸ ⎸ ⎸ **for** $i = 1,2, \ldots, |V^+| - 3$ **do**
18: ⎸ ⎸ ⎸ ⎸ **if** $S_\theta(\sigma_i) \neq S_\theta^+(\sigma_i)$ **then**
19: ⎸ ⎸ ⎸ ⎸ ⎸ **let** $v \notin \{S_\theta(\sigma_i), S_\theta^+(\sigma_1), S_\theta^+(\sigma_2), \ldots, S_\theta^+(\sigma_i)\}$
20: ⎸ ⎸ ⎸ ⎸ ⎸ $S_\theta \leftarrow$ Apply-Macro$(table_3^\theta[S_\theta(\sigma_i), S_\theta^+(\sigma_i), v], S_\theta)$

function *Apply-Macro*(ω, S_θ): **assignment**
/* Applies a given sub-solution on a global arrangement S_A and on an arrangement over θ-like subgraph.
Parameters: ω - a solution of a sub-problem
S_θ - arrangement over θ-like subgraph */



1: **let** $[(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)] = \omega$
2: **for** $i = 1,2, \ldots, k$ **do**
3: ⎸ Move-Agent-Unoccupied(u_i, v_i)
4: ⎸ $S_\theta(\Phi_P(u_i)) \leftarrow v_i$
5: ⎸ **return** S_θ

The main framework of the algorithm as it was described above is represented by the function *BIBOX-θ-Solve* which gets a pCPF instance on a non-trivial bi-connected graph $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ with just a single unoccupied vertex as a parameter and returns the length of the solution and the solution itself. The difference from the original *BIBOX* algorithm is that the handle decomposition is computed with a special care (lines 1-6) and the final solving process (lines 13-21) over the θ-like graph formed by C_0 and H_1 exploits the solution database. The middle section of the whole solving process (lines 10-12), when agents are placed into handles, is the same as in the case of the *BIBOX* algorithm. To mitigate the need of care about the location of an unoccupied vertex, the first connection vertex of the handle H_1 is vacated (lines 14-16) – this vertex correspond to the vertex $\bar{y}_1$ from the definition of the θ-like graph. Recall, that the transposition, the 3-cycle rotation, and the case of $\theta(2,3,2)$ suppose the unoccupied vertex to be right there.

An auxiliary function *Apply-Macro* is used to apply a record ω from the solution database (the optimal solution for a sub-instance is called a *macro* in this context) on the current arrangement of agents S_θ in a given θ-like graph as well as on the global current arrangement represented by S_A and Φ_A. The optimal solution has the form of a sequence of moves where the move is an ordered pair of vertices of G - the first vertex contains an agent to be moved; the second vertex is unoccupied at the time step of execution of the move and represents the target vertex. The execution of the macro over the current arrangement is carried out by *Move-Agent-Unoccupied*; the function also constructs the next step in construction of the output solution.

The very novel part in comparison with the *BIBOX* algorithm is the process of reaching the goal arrangement over a θ-like graph. This is represented by a function *θ-BOX-Solve*. The function gets as parameters the θ-like graph itself as $\theta(X, Y, Z)$, an initial and a goal arrangement of agents as S_θ^0 and S_θ^+ respectively, and a set of goal vertices as V^+ which is a sub-set of vertices of θ.

If θ is isomorphic to $\theta(2,3,2)$ (lines 3-6) then the goal arrangement is reached at once using a record from the database. It may happen that the required record is not found in the database (line 5). In such a case, the algorithm terminates with the answer that the given instance is unsolvable. A special function *difference* is used in this execution branch. The function calculates a permutation from two arrangements of agents. The interpretation of a permutation calculated by the *difference* function is that it transforms an arrangement given as the first argument to an arrangement given as the second argument.

If θ is non-isomorphic to $\theta(2,3,2)$ and it contains an odd cycle (lines 7-12) then all the goal arrangements are reachable. The goal arrangement is reached by composing several transposition cases. This is done by traversing the set of agents that should be placed. If the current location of an agent given by S_θ differs from its goal location given by S_θ^+, then agents at these two locations are exchanged using a solution for the transposition case from the database of solutions.

If θ is non-isomorphic to $\theta(2,3,2)$ and all the subsumed cycles are of an even length (lines 14-20) then the treatment of unsolvable cases must be done. If the goal arrangement S_θ^+ forms an odd permutation with respect to the initial arrangement S_θ then the given instance is unsolvable (lines 14-15). The algorithm terminates with the negative answer in such a case. If this is not the case (that is, S_θ^+ forms an even permutation with respect to S_θ) then the goal arrangement is reached using 3-cycle rotations (lines 17-20).

This is done almost in the same way as in the case of transposition cases in fact. Again, agents that should be relocated are traversed. The relocation of an agent σ_i to its goal location $S_\theta^+(\sigma_i)$ from $S_\theta(\sigma_i)$ is done by a rotation along a 3-cycle formed by $S_\theta(\sigma_i)$, $S_\theta^+(\sigma_i)$, and v, where v is a vertex

different from $S_\theta(\sigma_i)$, $S_\theta^+(\sigma_i)$, and also different from all the goal vertices of all the already placed agents. Notice, that it is sufficient to traverse all the agents except last two. They must be inevitably placed to their goal vertices after the last 3-cycle rotation since otherwise the goal arrangement S_θ^+ forms an odd permutation with respect to S_θ which has been ruled out at the beginning of this branch.

3.2.3. *Summary of Theoretical Properties and Extensions of the BIBOX-θ Algorithm*

The detailed theoretical analysis of soundness and completeness of the *BIBOX-θ* algorithm can be found in [27]. The crucial ingredient for the correctness of the algorithm is represented by Lemma 3.

The worst-case time complexity of the algorithm is $\mathcal{O}(|V|^4)$ [27] with respect to the input instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. The makespan is also $\mathcal{O}(|V|^4)$ [27]. This result can be obtained from the fact that the length of optimal solutions of special cases is bounded by $(|V|^3)$ [8]. As $\mathcal{O}(|V|)$ optimal solutions of special cases are necessary, the upper bound of $\mathcal{O}(|V|^4)$ is obtained.

If the size of the database containing optimal solutions is not accounted, the space required by the algorithm is of $\mathcal{O}(|V| + |E|)$ in the worst-case. The space required by the part of the database where optimal solutions to $\theta(2,3,2)$ are stored is $\mathcal{O}(1)$ (the size of $table_{232}^\theta$) and the space required by the part of the database where solutions to transposition and 3-cycle rotation cases over a θ-like graph $\theta(X, Y, Z) = (V_\theta, E_\theta)$ are stored is $\mathcal{O}(|V_\theta|^5)$ (the size of $table_T^\theta$) and $\mathcal{O}(|V_\theta|^6)$ (the size of $table_3^\theta$) respectively.

Practically, it is better to use slightly adapted special cases. Observe that special cases as described above preserve all the agents except the affected pair or triple at their original positions. This is not necessary in fact, since only agents that already reached their goal positions need to be preserved. Preserving other agents just imposes additional constraints on the solution and may prolong it unnecessarily. The no less important fact is that it is easier to find a less constrained optimal solution. The modified special cases, where relocation of agents that have not yet reached their goal positions are neglected, are called a *weak transposition case* and a *weak 3-cycle rotation case* respectively. The detailed description of weak special cases is given in [27].

4. Experimental Evaluation

As algorithms *BIBOX* and *BIBOX-θ* were primarily developed as an alternative to the *MIT* algorithm [8], the experimental evaluation will be primarily aimed on the competitive comparison of *BIBOX* and *BIBOX-θ* with *MIT*. Nonetheless, we also provide comparison with the *WHCA** algorithm [18] to obtain more complete image.

All the tested algorithms were implemented in C++. The implementation of algorithms *BIBOX* and *BIBOX-θ* follows the presented pseudo-code. Several optimizations mentioned in Section 3.1.3 were adopted in the implementation of *BIBOX* and *BIBOX-θ* algorithms as well.

The database of optimal solutions used by the *BIBOX-θ* algorithm has been generated on-line (on demand) by a variant of IDA* algorithm enhanced with learning [21]. Details of this algorithm are out of scope of this study. Pseudo-code and experimental analysis can be found in [21]. Notice, that it is a time consuming task to find an optimal solution to a pCPF instance even on a small θ-like graph. Therefore, the timeout of 8.0 seconds was used after that the solving process switched to the *MIT* style. The database with optimal solutions should be pre-computed off-line in the real-life applications.

The *MIT* has been re-implemented according to [8]. The algorithm is designed for general graphs, however the major technique concerns bi-connected partitions. Briefly said, the algorithm finds a configuration of vertices in the input graph on that a 3-cycle rotation is possible. At the same time, it

is ensured that every triple of agents can be relocated to this configuration and back to their original locations. By composing these three basic operations – relocation to the 3-cycle rotation configuration, 3-cycle rotation there, and relocation back to original locations – we are actually able to make 3-rotation of every triple of agents. This consequently means that agents can be relocated according to every even permutation by the outlined process (see also Proposition 5). If additionally there is an odd cycle in the input graph, all the permutations are possible.

Similar optimization techniques as in the case of the *BIBOX* algorithm have been used. When an unoccupied vertex was necessary, the nearest unoccupied vertex was found and relocated to the location where needed. More details about the re-implementation of the *MIT* algorithm can be found in [23].

The WHCA* algorithm was also re-implemented by ourselves. It searches for a path for each agent individually while spatial-temporal positions occupied by the already scheduled agents are avoided. This algorithm is inherently incomplete since some agents may block another agent and prevent it from moving; thus, only few of tested setups were solvable by this algorithm.

In order to allow reproducibility of all the presented results the source code and supporting data is provided at the web site: http://ktiml.mff.cuni.cz/~surynek/research/j-multirobot-2010. Additional experimental results and raw experimental data are provided as well.

Experimental evaluation has been performed on two computers. The first computer has been used to generate experimental results regarding runtime - *runtime configuration* [1]; the second computer has been used to generate all the remaining results - *default configuration*[2].

4.1. **Makespan Comparison**

The first series of experiments is devoted to comparison of the makespan of solutions generated by tested algorithms. All the tested algorithms were used to generate a sequential solution of a given instance, which has been parallelized subsequently by the critical path method. The result was a parallel solution complying with the definition of the solution of pCPF. A set of testing instances of pCPF consists of instances on *randomly generated bi-connected* graphs and of instances on *grids*.

A randomly generated bi-connected graph has been generated according to its handle decomposition. First, a cycle of random length from *uniform distribution* where certain minimum and maximum lengths were given has been generated. Then a sequence of handles of random lengths from uniform distribution (again the minimum and the maximum length of handles was given) has been added. Each handle has been connected to randomly selected connection vertices in the currently constructed graph. The addition of handles has terminated when the required size of the graph has been reached. An instance on a randomly generated graph itself further consists of random initial arrangement and goal arrangement of agents over the graph where at least the given number of vertices remains unoccu-

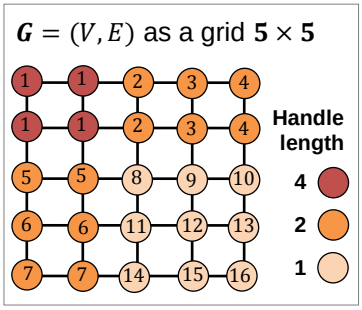


Figure 9. *An illustration of* ***handle decomposition of a grid*** *graph*. The ordering of the addition of individual handles is depicted by numbers in vertices. Three types of handles/cycles are used.

[1] ***Runtime*** *configuration*: 2x AMD Opteron 1600 MHz, 1GB RAM, Mandriva Linux 10.1, 32-bit edition, gcc version 3.4.3, compilation with –O3 optimization level.

[2] ***Default*** *configuration*: 4x AMD Opteron 1800 MHz, 5GB RAM, Mandriva Linux 2009.1, 64-bit edition, gcc version 4.3.2, compilation with –O3 optimization level.

pied. The handle decomposition used by solving algorithms was exactly that one used for generating the graph.

The situation with instances over the grid is similar. The square 4-connected grid graph of a given size has been generated together with a random initial and a goal arrangement of agents. Again, a given number of vertices remain unoccupied. First, an initial cycle with 4 vertices was constructed (placed on the left upper corner of the grid); then handles were added to fill in the grid successively according to its rows and columns. The first row and the first column were added at the beginning (handles with 2 internal vertices). Then rows of the grid were constructed by adding handles from the left to the right and from the top to the bottom (handles with 1 internal vertex). See Figure 9 for the ordering of addition of vertices in the construction of the grid.

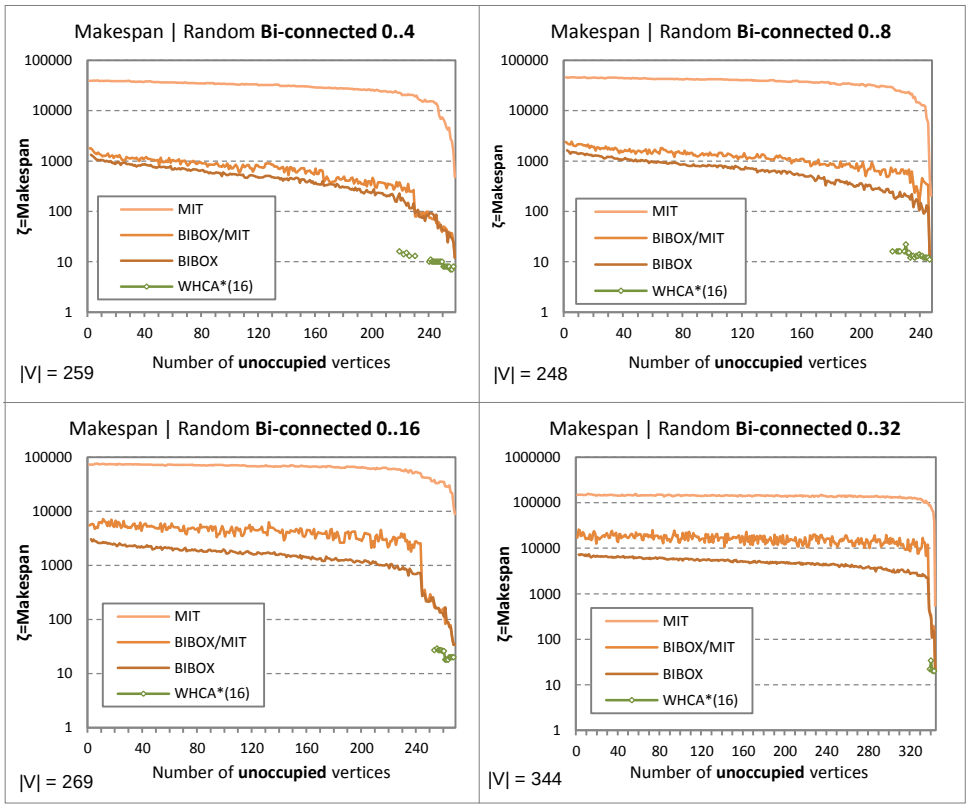


Figure 10. ***Makespan*** *comparison of solutions to instances over* ***random bi-connected*** *graphs.* Four algorithms are compared: the standard *BIBOX*, a variant of *BIBOX* where the last phase when agents are placed into the θ-like graph is solved by *MIT* – *BIBOX/MIT*, the *MIT* algorithm, and WHCA* with the window size of 16. Solutions were parallelized using the presented parallelism-increasing scheme [27] (critical-path method). Four setups of random bi-connected graphs are shown – random lengths handles have uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32 respectively. The makespan tends to decrease for the increasing number of unoccupied vertices. WHCA* was able to solve only several sparsely populated instances.

Results shown in Figure 10 and Figure 11 are targeted on the comparison of the makespan. Results in Figure 10 show makespans of solutions of instances over randomly generated bi-connected graphs. Graphs of size up to 344 vertices were used (the graph had been grown by addition of handles until the size of 256 vertices had been reached). Four graphs, which differ in the average length of the initial cycle and handles of the handle decomposition, were used. Lengths of the initial cycle and handles have the uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32. The length of the

handle is equal to the number of its internal vertices. Figure 11 is devoted to structurally regular graphs – grid graphs of the size 8×8, 16×16, and 32×32 were used.

Four algorithms were compared: the standard *BIBOX*, a variant of *BIBOX* where the last phase when agents are placed into the θ-like graph was solved by *MIT* – *BIBOX/MIT*, the *MIT* algorithm, and *WHCA** with the window size of 16.

Random initial and goal arrangements are obtained as a random permutation of agents in the vertices of the graph. The random permutation is generated from identical one by applying quadratic number of transpositions. This process generates random arrangements of the appropriate quality (of randomness) for the use in the test.

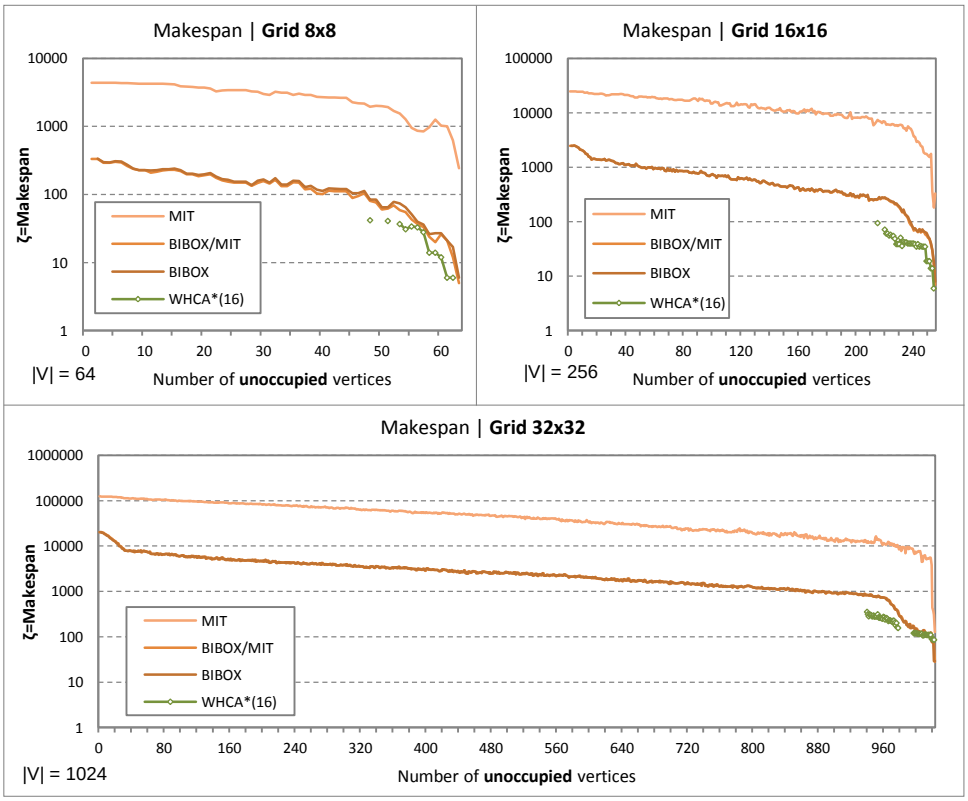


Figure 11. ***Makespan*** *comparison of solutions of instances over* ***square grids***. Four algorithms are compared: the standard *BIBOX*, *BIBOX/MIT*, *MIT*, and *WHCA**(16) on three grids: 8×8, 16×16, and 32×32.

It can be observed that the *BIBOX* algorithm generates solutions of the makespan approximately 10 times to 100 times smaller than that of solutions generated by the *MIT* algorithm. In the setup with random bi-connected graphs, the difference between *BIBOX* and *MIT* is becoming smaller as the size of handles increases. In the setup with the grid graph, the *BIBOX* algorithm generates solutions that have approximately 10 times smaller makespan than that of the *MIT* algorithm. A steep decline of the makespan can be observed when the portion of unoccupied vertices reaches approximately 95%. This is some kind of a phase transition when agents are becoming arranged sparsely enough over the graph so that there are almost no interactions between them (that is, they do not need to avoid each other). This phase transition seems to depend on the average size of handles – for the smaller size of handles the ratio of the number of agents to the number of vertices characterizing this phase transition tends to be higher. The *WHCA** algorithm generates better solutions than *BIBOX* in

most cases (the ratio between the makespan of *BIBOX* and *WHCA** is from 0.5 to 3.0). However, *WHCA** manages to do so only on sparsely occupied environments (number of unoccupied vertices more than 90%). As *WHCA** generates near optimal solutions with respect to the makespan we also have certain indication how far from the optimum solutions generated by *BIBOX* algorithms are. Let us note, that the most difficult instance from our test suite took *WHCA** approximately 2.0 seconds on the runtime configuration (80 agents in the 32×32 grid).

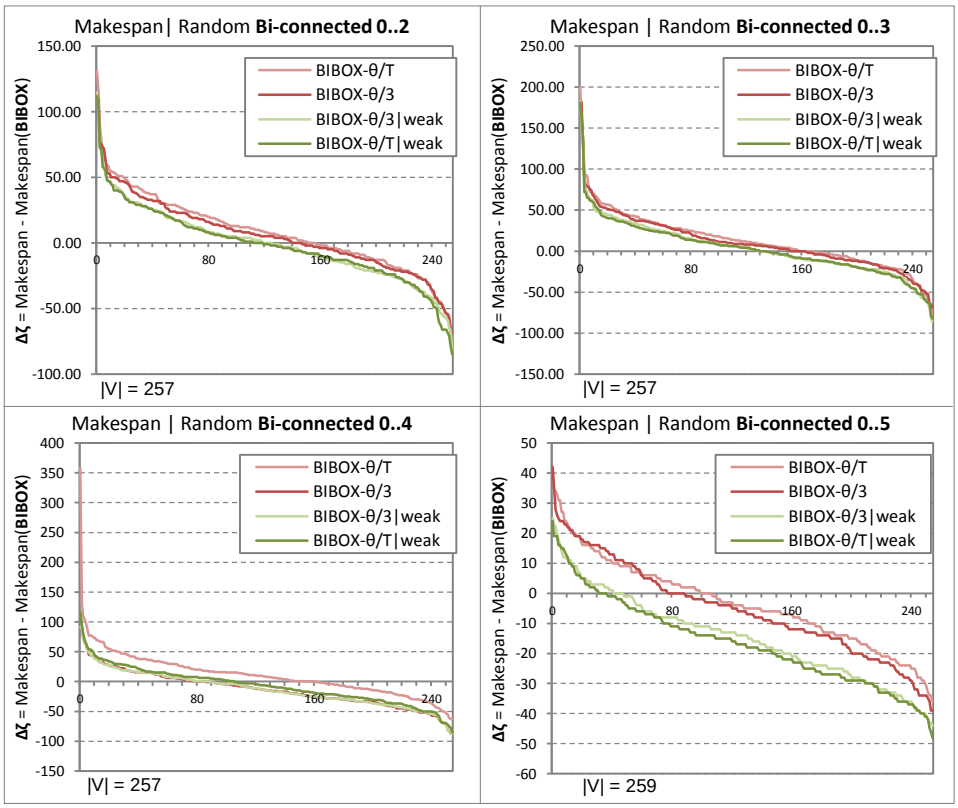


Figure 12. *An evaluation of the benefit of the use of* ***weak special cases*** *instead of the standard ones.* Four variants of the *BIBOX-θ* algorithm are compared: *BIBOX-θ*/T (the standard transposition case is used preferably), *BIBOX-θ*/3(the standard 3-cycle rotation case is used preferably), *BIBOX-θ*/T|weak (the weak transposition case is used preferably), and *BIBOX-θ*/3|weak (the weak 3-cycle rotation case is used preferably). The **difference** of the makespan of solution produced by these algorithms from those produced by the *BIBOX* algorithm is shown (values below zero indicate that the tested algorithm was better than *BIBOX*). Four random bi-connected graphs with the increasing number of unoccupied vertices are used; they have handles of lengths with uniform distribution of ranges: 0..2, 0..3, 0..4, and 0..5 respecitvely. To make the difference visible, results for individual algorithms are sorted in descending order.

The *BIBOX/MIT* algorithm exhibits performance influenced by the size of the initial θ-like graph. The larger is the graph the worse is the performance of the *BIBOX/MIT* algorithm. This behavior can be observed from the results shown in Figure 10 and Figure 11 using the fact that the longer handles induce larger initial θ-like graph. Grid graphs represent the extreme case – almost all the handles are of the size 1. Both algorithms – *BIBOX* as well as *BIBOX/MIT* – generate solutions of the very similar makespan (the only difference is observable in the case of grid 8×8 with low occupation where *BIBOX/MIT* is marginally better).

Regarding the makespan, the *BIBOX* style solving process represents the better alternative than *MIT* when at least two unoccupied vertices are provided.

An interesting question is whether the use of optimal solutions to weak cases instead of standard ones does really help. Results reported in Figure 12 are devoted to this question. A comparison of the *BIBOX* algorithm with the variants of the *BIBOX-θ* algorithm is shown.

Four variants of the *BIBOX-θ* algorithm are compared: *BIBOX-θ*/T (the standard transposition case is used preferably), *BIBOX-θ*/3 (the standard 3-cycle rotation case is used preferably), *BIBOX-θ*/T|weak (the weak transposition case is used preferably), and *BIBOX-θ*/3|weak (the weak 3-cycle rotation case is used preferably). Notice, that the variant presented in the pseudo-code as Algorithm 3 prefers standard transposition cases. If the transposition case is not possible to apply, the corresponding 3-cycle rotation case is used instead (which is always possible). Other variants implement the preference in the analogical way.

The comparison in Figure 12 shows difference of the makespan of solution generated by mentioned three variants of *BIBOX-θ* from the makespan of the corresponding solution generated by the standard *BIBOX* (negative values of the difference indicate that *BIBOX* generated solution with the greater makespan). Four random bi-connected graphs were used for the experiment; the number of vertices was up to 259 (again, the graph had been grown by addition of handles until the size of 256 vertices had been reached). The length of the initial cycle and handles has been selected randomly with the uniform distribution of ranges: $0..2$, $0..3$, $0..4$, and $0..5$, respectively. The relatively small ranges are used in order to be able to calculate all the optimal solutions of the special cases in the timeout of 8.0. The size of the θ-like graph, on that special cases appear, directly corresponds to the length of the initial cycle and handles of the handle decomposition. Makespans have been collected for instances with 2 to $|V| - 1$ unoccupied vertices for each graph $G = (V, E)$. To make differences among performances of tested algorithms clearly visible, the difference in makespans has been sorted in the descending order. The difference in makespan tends to be greater for instances with few unoccupied vertices (hence, it is expected that these makespans are sorted to the left or to the right margin in each plot).

Results shown in Figure 12 can be interpreted as that solutions with the smallest makespan are produced by *BIBOX-θ*/T|weak closely followed by *BIBOX-θ*/3|weak. Hence, it is possible to conclude that the use of optimal solutions to weak special cases is beneficial. Moreover, a solution to a weak special is easier to generate since it is less *constrained* than the solution of the corresponding standard case.

Since values of the makespan differences deviate from the uniform distribution around 0 marginally, it is also possible to conclude that variants of *BIBOX-θ* does not improve the makespan significantly in comparison with *BIBOX* on instances with at least two unoccupied vertices. Thus, the use of *BIBOX-θ* is substantiated only for instances with just a single unoccupied vertex (where the *BIBOX* algorithm is not applicable).

4.2. **Parallelism Evaluation**

The exact meaning of the term *parallelism* is the value obtained as the ratio of the total number of moves divided by the makespan. The result is the average number of moves performed at each time step. High parallelism is typically desirable since it implies the smaller makespan.

In the experiments, we observed how the average parallelism changes while the number of unoccupied vertices is increasing. The same set of setups as in the case of makespan evaluation was used. Results regarding bi-connected graphs are shown in Figure 13 results regarding grids are shown in Figure 14. The parallelism-increasing algorithm [27] was used to post-process the solutions. In case of WHCA* the initial solution was already parallel but in the sense of PMG; we parallelized it further according to pCPF (which however made almost no change as in instances solvable by WHCA* agents were rather isolated).

On random bi-connected graphs, the parallelism of solutions slightly increases as the number of unoccupied vertices reaches approximately 50% occupancy. This behavior is yet more expressed on the grid graphs. The increase of the parallelism is steeper in this case. When the number of unoccupied vertices is higher than some threshold a different behavior can be observed. The fewer agents are in the graph the lower is the parallelism. It can be also observed that parallelism correlates with the average length of handles of the handle decomposition – this is caused by the fact that all the agents in the handle are moving at once. Another characteristic, which the parallelism correlates with, is the *diameter* [33] of the graph. This correlation can be observed on tests with grid graphs in Figure 14. The reason for this correlation is the fact that all the agents along a path connecting two vertices in the graph moves at once when the unoccupied vertex is relocated. The average length of such paths correlates with the diameter of the graph.

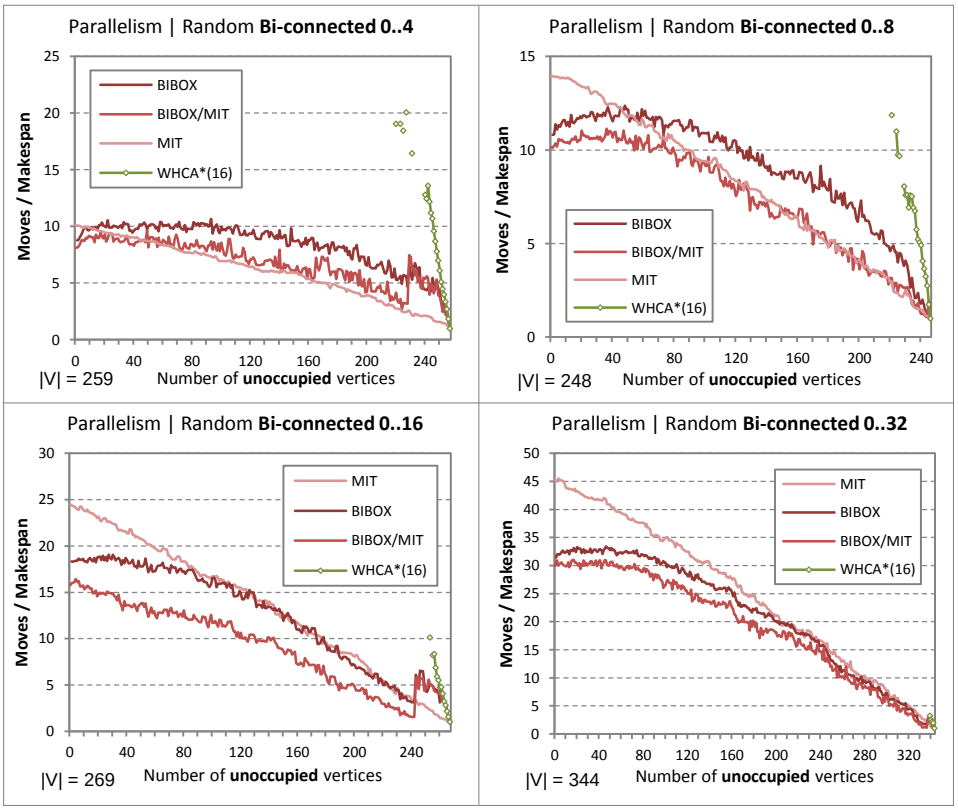


Figure 13. ***Average parallelism*** *of solutions generated by tested algorithms for instances over* ***random bi-connected*** *graphs. BIBOX, BIBOX/MIT, MIT,* and *WHCA** are compared. Four random bi-connected graphs were used – random lengths of initial cycle and handles of the handle decomposition have uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32. The average parallelism is the total number of moves divided by the makespan.

Regarding the *MIT* algorithm, it can be observed that the parallelism of its solutions decreases almost linearly with the increasing number of unoccupied vertices. Without providing further details, the explanation of this behavior is that all the phases of the algorithm are rather homogenous. Thus, as occurrence of agents is getting linearly sparser the parallelism decreases almost linearly. Recall, that the *BIBOX* algorithm behaves differently. All the movements take place in the unfinished part of the graph only, which is relatively getting smaller as the *BIBOX* algorithm proceeds.

Generally, it can be concluded from Figure 13 and Figure 14 that solutions generated by the *BIBOX* and *BIBOX/MIT* algorithms allow higher parallelism than that of *MIT*. Consequently, it can be observed together from Figure 10, Figure 11, Figure 13, and Figure 14 that the total number of moves, which solutions generated by *BIBOX* and *BIBOX/MIT* consist of, are still order of magnitude smaller than that of *MIT*. Thus, the performance of the *BIBOX* algorithms is not caused by the higher parallelism but also by the smaller size of the generated sequential solutions.

Results regarding WHCA* indicate that typically all the agents move. The explanation is that the no-op (that is, an agent does not move) is chosen only if it is necessary to avoid another agent, which is relatively rare situation. Otherwise a move through that an agent can approach its goal is chosen. On random bi-connected graphs WHCA* tends to reach higher parallelism than the other tested algorithms. On grid it seems that no simple statement can be done.

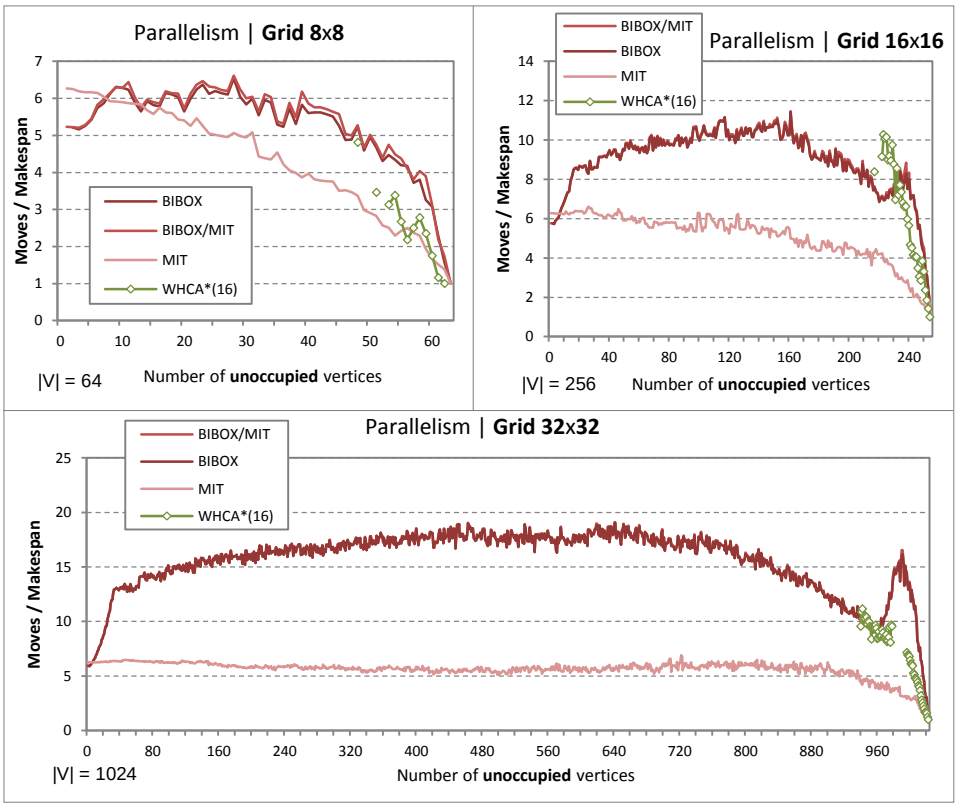


Figure 14. ***Average parallelism*** *comparison of solutions of instances over* ***square grids****. BIBOX, BIBOX/MIT, MIT,* and *WHCA** are compared on three grids: 8 × 8, 16 × 16, and 32 × 32.

The development of the number of movements per time step called *step parallelism* is shown in Figure 15. This experiment was done with the *BIBOX* algorithm only on a random bi-connected graph where lengths of the initial cycle and handles were randomly selected with the uniform distribution with of the range 0..4. There were exactly two unoccupied vertices in the input graph.

Peaks in Figure 15 correspond to parallel movements along long paths. The density and height of peaks is getting slightly smaller as the algorithm proceeds. This is caused by the fact that the part of the graph affected by movements is getting smaller. Other values correspond to various rotations along cycles are done intensively by the algorithm. The absolute number of parallel movements cor-

responding to these rotations does not change as the algorithm proceeds (the average size of a cycle in the unfinished part of the graph is still the same since the graph was generated uniformly).

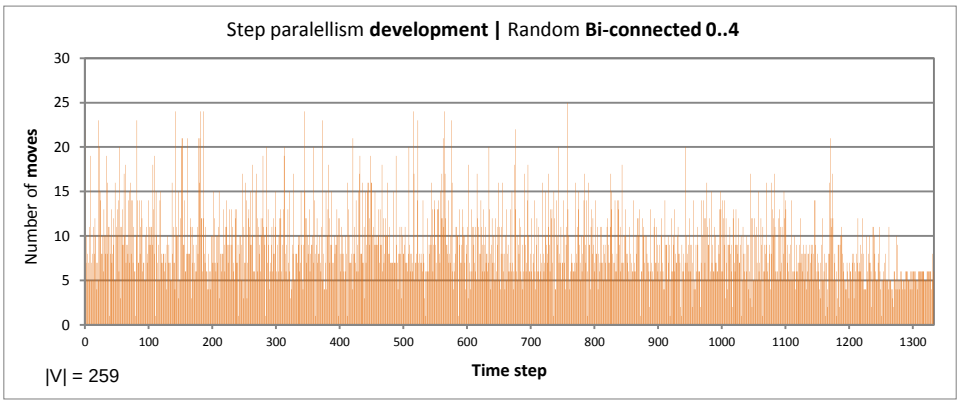


Figure 15. ***Step parallelism development*** *of in a solution generated by BIBOX*. The random bi-connected graph was generated with the length of the initial cycle and handles having uniform distribution of the range 0..4. There were exactly two unoccupied vertices. The development of the *step parallelism* (number of moves per time step) over time is shown.

4.3. **Scalability Evaluation**

Scalability tests were aimed on the makespan of generated solution and the overall runtime necessary to produce the parallel solution while the number of unoccupied vertices was fixed to 2 and the size of the graph was varying. The overall time is the time necessary to produce a sequential solution plus the time needed to increase its parallelism.

The *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T|weak, *BIBOX-θ*/3|weak, and *MIT* were compared. Algorithms *BIBOX-θ*/T and *BIBOX-θ*/3 were ruled out since they are outperformed by *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak respectively as it has been shown in Section 4.1. *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak are slightly faster supposed that all the records in the database of optimal solutions are precomputed off-line (the shorter resulting solution needs to be produced than in the case of *BIBOX-θ*/T and *BIBOX-θ*/3). However, they are significantly faster if the optimal solutions need to be computed on-line (on demand) [21, 22] as the optimal solution to weak special case is easier to find than the optimal solution to the standard special case.

Tests targeted on scalability used the different setup of instances of pCPF than previous tests. Now, approximately 250 instances on bi-connected graphs with the size varying from 16 to 256 vertices were generated. Random lengths of the initial cycle and handles of the handle decomposition were selected randomly from uniform distribution with ranges: 0..2,…, 0..16. Such selection guarantees that graphs with short handles as well as graphs with long handles are included. There were exactly two unoccupied vertices in all the tested instances.

Scalability evaluation for the makespan is shown in Figure 16. The makespan for the increasing number of vertices is shown. Experiments in Figure 17 used the same setup (the same set of instances); the difference from Figure 16 is just that the runtime is shown. In both figures, algorithms are compared pair-wise from the worst performing to the best performing pair (the pair of algorithms that are closest to each other according to the given characteristic is compared).

Results regarding makespan show that the *MIT* algorithm performs as worst while the standard *BIBOX* algorithm produces the best solutions. *BIBOX/MIT*, *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak are somewhere in the middle. The makespan of solutions generated by *BIBOX-θ*/T|weak and *BIBOX-*

θ/3|weak sometimes jumps up to the makespan of the corresponding solution generated by *BIBOX/MIT*. This happens if *BIBOX-θ*/T|weak or *BIBOX-θ*/3|weak do not manage to compute optimal solution to the special case in the given timeout of 8.0 seconds. In such a case *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak produces exactly the same solution as *BIBOX/MIT* since they have to switch to the *MIT* mode of generating (sub-optimal) solutions to special cases.

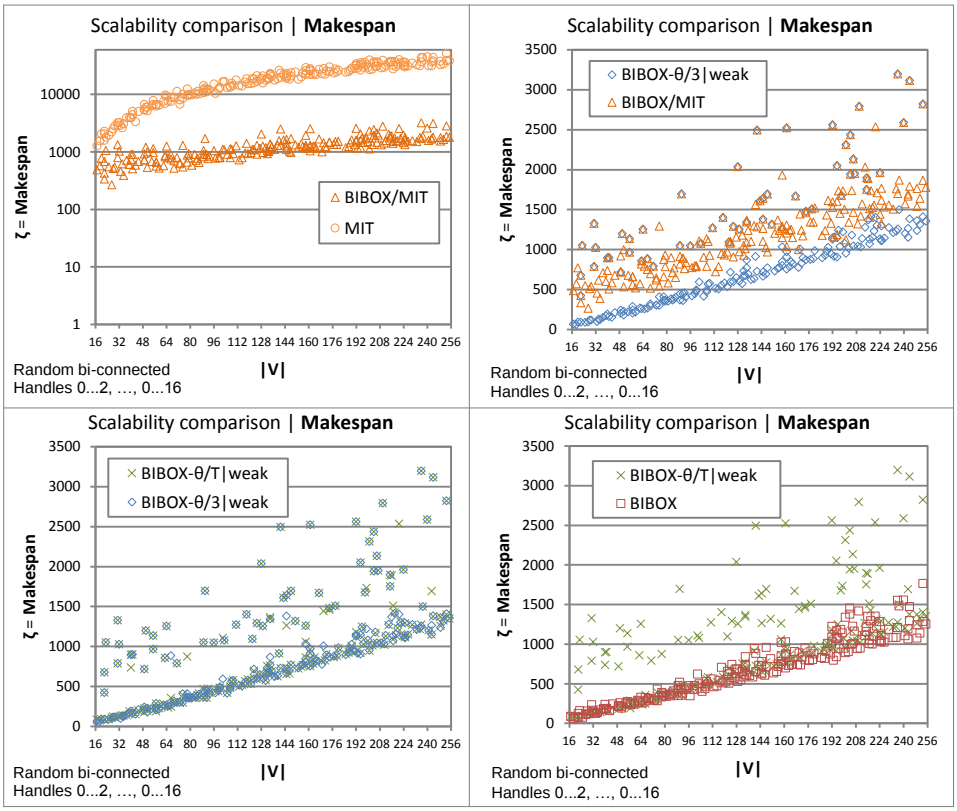


Figure 16. *A **comparison of the scalability** of tested algorithms with respect to the **makespan***. Five algorithms were compared: *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T|weak, *BIBOX-θ*/3|weak, and *MIT*. Approximately 250 pCPF instances over various random bi-connected graphs containing 16 to 256 vertices were used. The range of the uniform distribution of lengths of handles in the random generation was: 0..2, …, 0..16. Algorithms are sorted from left/top to right/bottom according to the increasing performance (*MIT* – worst; *BIBOX* - best). Each sub-plot shows the relative comparison of two algorithms.

A quite surprising result is that even though *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak compose the resulting solution over the θ-like graph consisting of the initial cycle and the first handle from the optimal solutions to special cases, it still has the worse makespan than the corresponding solution generated using agents exchanges by the *BIBOX* algorithm. Hence, the second unoccupied vertex has the significant impact on simplifying the solving process.

Results regarding the overall runtime of tested algorithms generally show that *BIBOX-θ*/T|weak and *BIBOX-θ*/3|weak are as slow as the given timeout for computing optimal solutions to the special cases. The more interesting situation is with *MIT*, *BIBOX/MIT*, and *BIBOX* since they have very similar runtimes. The *BIBOX/MIT* tends to be faster than *MIT* while there is only marginal difference between *BIBOX* and *MIT* on larger graphs in favor of *BIBOX*. Observe that the runtime does not exactly correspond to the length of the generated solutions. In other words, certain computations used

by *BIBOX* are more time consuming than that of *MIT* (for example *BIBOX* extensively searched for a path when agent is moved).

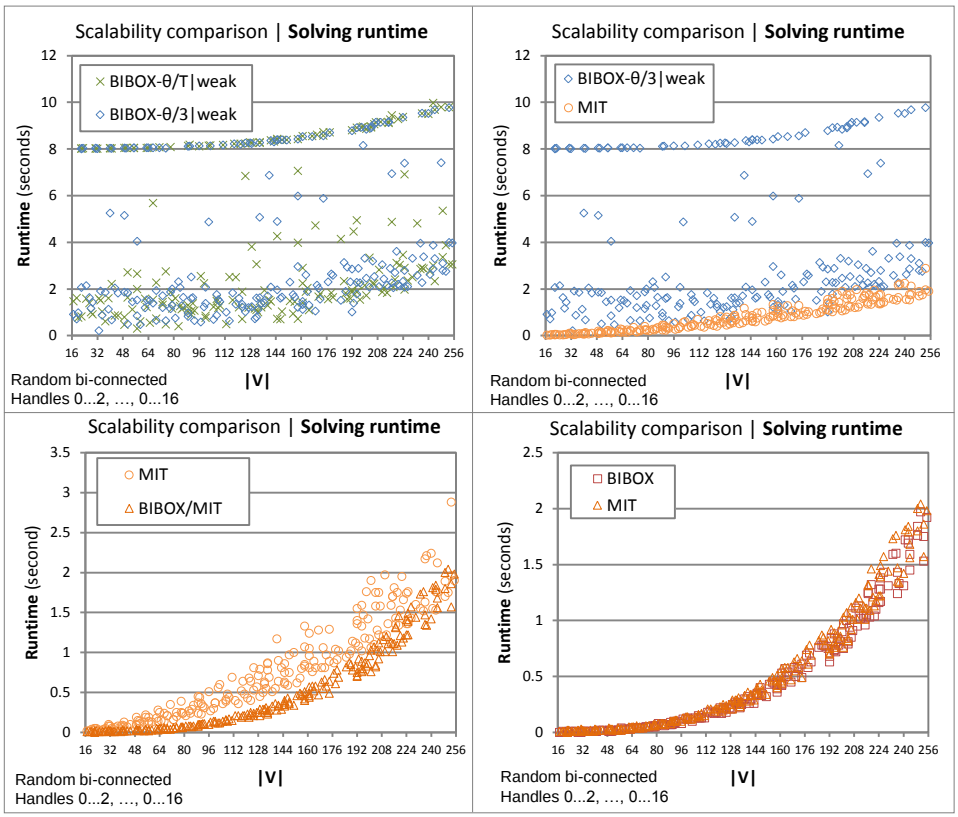


Figure 17. *A **comparison of the scalability** of tested algorithms with respect to the **runtime**. BIBOX, BIBOX/MIT, BIBOX-θ/T|weak, BIBOX-θ/3|weak, and MIT* were compared. The setup of instances is the same as for the experiment from Figure 16. Algorithms are sorted from left/top to right/bottom according to the increasing performance. The runtime (the total time necessary to produce sequential solution plus the time for making it parallel) is shown. The runtime increases for the increasing size of the instance (number of vertices).

5. Conclusion and Future Work

Two new algorithms – called *BIBOX* and *BIBOX-θ* – for solving the abstract multi-agent cooperative path-finding with special regard on parallelism (so called pCPFs) were described in this work. Both algorithms are designed for the case when environment is modeled as a bi-connected graph and is densely occupied by agents. Several modified variants of the *BIBOX-θ* algorithm were described as well.

The precise theoretical foundation and experimental analysis of these algorithms is provided. The theoretical foundation is targeted on correctness of the design of algorithms. The experimental analysis is primarily targeted on comparison with the *MIT* algorithm that employs permutation group theory and is capable of solving pCPF instances characterized by the small unoccupied space. To provide the complete image with respect to the related works in cooperative path-finding the comparison with the WHCA* algorithm, which is one of the most commonly used benchmark algorithm for CPF, is given as well.

Although the *MIT* algorithm has promising theoretical properties it has been outperformed by *BIBOX* in terms of the makespan by the order of one to two magnitudes. Although the asymptotic estimation for the makespan is the same for both *BIBOX* and *MIT*, the multiplication factor in the estimation in the case of *BIBOX* is smaller. Regarding the runtime, *BIBOX* algorithm is slightly faster than *MIT*, which itself is relatively fast (instances with graphs of hundreds of vertices occupied by hundreds of agents are solved within seconds on today's commodity hardware).

The minor drawback of the *BIBOX* algorithm is that it is not able to solve instances of pCPF with just a single unoccupied vertex. This issue has been addressed in this work by proposing modified algorithm called *BIBOX-θ* and its variants called *BIBOX/MIT*, *BIBOX-θ*/T, *BIBOX-θ*/3, *BIBOX-θ*/T|weak, and *BIBOX-θ*/3|weak. They use a different approach to solve the situation on the simple bi-connected graphs consisting of one cycle and one handle connected to it – called θ-like graphs. Except the first mentioned algorithm, all the other algorithms use the database with optimal solutions to special instances over these θ-like graphs – called special cases – of which solutions to all the instances over θ-like graphs can be composed.

Regarding the makespan, all the alternative algorithms outperform *MIT*. If the database of optimal solutions is available in advance, then *BIBOX-θ* algorithms almost match the performance of *MIT* in terms of runtime. If the required optimal solutions to special cases are not available, they need to be computed on-line which is difficult. It can cause a significant slowdown of the algorithm.

Notice, that the performance of both presented algorithms depends on the handle decomposition of the input graph. An interesting question is how to optimize handle decomposition in order to improve makespan or runtime. Is it better to use a small number of large handles or a large number of small handles? This question is out of the scope of this work and it is left for future work.

A considerable drawback of presented algorithms is their limitation on bi-connected graph. Notice, that search-based techniques like WHCA* are not limited to any special class of graphs. Hence, extension of presented algorithms to the general case is of interest. One of the possible approaches is to decompose a given general graph into the tree of bi-connected components [33, 34]. Any of the presented algorithms for bi-connected case can be used over the individual bi-connected components. However, agents need first to be relocated to the target bi-connected components. It may happen that an agent needs to go to the neighboring bi-connected component different from that where it is currently located. If the *bridge* connecting these components is longer than the number of unoccupied vertices then the relocation of the agent will not be possible. Hence, there will be relatively many unsolvable instances in the general case.

Regarding future work, it is also interesting to resolve the question whether optimal solutions of pCPF can be approximated by a (pseudo-) polynomial time algorithm. If an approximation algorithm with (pseudo-) polynomial time complexity is available, it is possible to estimate how far the current solution is from the optimal one even for large and densely occupied instances (currently we have only intuition for sparsely populated instances thanks to experiments with WHCA*). Some study of this kind of approximation algorithms for the special case of $(N^2 - 1)$-puzzle has been done in [11, 12, 13].

Another interesting topic for future work is to study how solutions generated by presented algorithm can be improved. A first view work has been already done in [25]. It is based on identifying and eliminating redundancies from solutions. The performed experiments showed that it is a promising technique.

Acknowledgments

This work is supported by The Czech Science Foundation (Grantová agentura České republiky - GAČR) under contracts number 201/09/P318 and GAP103/10/1287, by The Ministry of Education, Youth and Sports, Czech Republic (Ministerstvo školství, mládeže a tělovýchovy ČR – MŠMT ČR) under the contract number MSM 0021620838, and by Japan Society for the Promotion of Science (JSPS) within the post-doctoral fellowship of the author (reference number P11743).

I would like to express grateful thanks to anonymous reviewers for their thorough comments. Their recommendations helped me to improve the paper dramatically from its original version.

References

1. T. H. **Cormen**, C. E. **Leiserson**, R. L. **Rivest**, and C. **Stein**. *Introduction to Algorithms (Second edition)*, MIT Press and McGraw-Hill, 2001, ISBN 0-262-03293-7.
2. J. D. **Dixon** and B. **Mortimer**. *Permutation Groups*. Graduate Texts in Mathematics, Volume 163, Springer, 1996, ISBN 978-0-387-94599-6.
3. M. R. **Garey** and D. S. **Johnson**. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979, ISBN: 978-0716710455.
4. J. E. **Hopcroft**, R. **Motwani**, J. D. **Ullman**. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2000, ISBN: 978-0201441246.
5. E. **Hordern**. *Sliding Piece Puzzles.* Oxford University Press, 1986, ISBN: 978-0198532040.
6. M. R. **Jansen** and N. R. **Sturtevant**. *Direction maps for cooperative pathfinding.* Proceedings of (AIIDE 2008), pp..AAAI Press, 2008.
7. M. R. **Jansen** and N. R. **Sturtevant**. *A new approach to cooperative pathfinding.* Proceedings of AAMAS 2008, pp. 1401 - 1404, 2008.
8. D. **Kornhauser**, G. L. **Miller**, and P. G. **Spirakis**. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications*. Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.
9. R. **Luna**, K. E. **Berkis**. *Push-and-Swap: Fast Co-operative Path-Finding with Completeness Guarantees*. Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 294-300, IJCAI/AAAI Press, 2011.
10. C. H. **Papadimitriou**, P. **Raghavan**, M. **Sudan**, and H. **Tamaki**. *Motion Planning on a Graph.* Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994), pp. 511-520, IEEE Press, 1994.
11. I. **Parberry**. *A Real-Time Algorithm for the (n^2-1)-Puzzle.* Information Processing Letters, Volume 56(1),pp. 23-28, Elsevier,1995.
12. D. **Ratner** and M. K. **Warmuth**. *Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable*. Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.
13. D. **Ratner** and M. K. **Warmuth**. *N×N Puzzle and Related Relocation Problems.* Journal of Symbolic Computation, Volume 10 (2), pp. 111-138, Elsevier, 1990.
14. S. **Russell** and P. **Norvig**. *Artificial Intelligence: A Modern Approach (second edition).* Prentice Hall, 2003, ISBN: 978-0137903955.
15. M. R. K. **Ryan**. *Graph Decomposition for Efficient Multi-Robot Path-Planning*. Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI Conference, 2007.
16. M. R. K. **Ryan**. *Exploiting Subgraph Structure in Multi-Robot Path-Planning*. Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.
17. P. E. **Schupp** and R. C. **Lyndon**. *Combinatorial group theory*. Springer, 2001, ISBN 978-3-540-41158-1.
18. D. **Silver**. *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press.

19. T. **Standley**. *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 173-178, AAAI Press, 2010.
20. P. **Surynek**. *A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs*. Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.
21. P. **Surynek**. *Towards Shorter Solutions for Problems of Path Planning for Multiple Robots in θ-like Environments.* Proceedings of the 22nd International FLAIRS Conference (FLAIRS 2009), pp. 207-212, AAAI Press, 2009.
22. P. **Surynek**. *Making Solutions of Multi-Robot Path-Planning Problems Shorter Using Weak Transpositions and Critical Path Parallelism*. Proceedings of the 2009 International Symposium on Combinatorial Search (SoCS 2009), University of Southern California, 2009, http://www.search-conference.org/index.php/Main/SOCS09 [July 2009].
23. P. **Surynek**. *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path-planning.* Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.
24. P. **Surynek**. *An Optimization Variant of Multi-Robot Path-Planning is Intractable.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 1261-1263, AAAI Press, 2010.
25. P. **Surynek** and P. **Koupý**.*Improving Solutions of Problems of Motion on Graphs by Redundancy Elimination.* Proceedings of the ECAI 2010 Workshop on Spatio-Temporal Dynamics (ECAI STeDy 2010), pp. 37-42, University of Bremen, 2010.
26. P. **Surynek**. *Abstract Path Planning for Multiple Robots: A Theoretical Study.* Technical Report, ITI Series, 2010-503, http://iti.mff.cuni.cz/series/index.html,Institute for Theoretical Computer Science, Charles University in Prague, Czech Republic, 2010.
27. P. **Surynek**. *Abstract Path Planning for Multiple Robots: An Empirical Study.* Technical Report, ITI Series, 2010-504, http://iti.mff.cuni.cz/series/index.html,Institute for Theoretical Computer Science, Charles University in Prague, Czech Republic, 2010.
28. R. E. **Tarjan**. *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.
29. K. C. **Wang** and A.**Botea**. *Tractable Cooperative path-finding on Grid Maps*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 1870-1875, IJCAI Conference, 2009.
30. K. C. **Wang**. *Bridging the Gap between Centralised and Decentralised Multi-Agent Pathfinding.* Proceedings of the 14th Annual AAAI/SIGART Doctoral Consortium (AAAI-DC 2009), pp. 23-24, AAAI Press, 2009.
31. K. C. **Wang** and A.**Botea**. *Fast and Memory-Efficient Multi-Agent Pathfinding.* Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008), Australia, pp. 380-387, AAAI Press, 2008, ISBN 978-1-57735-386-7.
32. K. C. **Wang** and A. **Botea**. *Scalable Multi-Agent Pathfinding on Grid Maps with Tractability and Completeness Guarantees*. Proceedings of the European Conference on Artificial Intelligence (ECAI 2010), IOS Press, 2010.
33. D. B. **West**. *Introduction to Graph Theory.* Prentice Hall, 2000, ISBN: 978-0130144003.
34. J. **Westbrook**, R. E. **Tarjan**. *Maintaining bridge-connected and biconnected components on-line*. Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
35. R. M. **Wilson**. *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.

Appendix

Lemma 4 ***(soundness of Move-Agent).*** If an original location of an agent a, a goal location v, and an unoccupied vertex are all located in the same unlocked bi-connected component of the graph G, then the procedure *Move-Agent* correctly moves the agent a from its original location to v. ■

Proof. Recall how the procedure *Move-Agent* works. First, a path $\varphi = [w_1^\varphi, w_2^\varphi, \ldots, w_{j_\varphi}^\varphi]$ connecting $S_A(a)$ and v that is contained in the same bi-connected component is found. The path φ is then traversed while the agent a is moved along its edges.

The proof of soundness will proceed as mathematical induction according to the number of edges of φ already traversed. In all the steps, the agent a and the unoccupied vertex should be located in the bi-connected component containing φ. Initially, this condition holds. Consider that an agent a is located in w_i^φ for $i \in \{1,2,\ldots,j_\varphi\}$ and need to be moved to w_{i+1}^φ. The vertex w_i^φ is locked and w_{i+1}^φ is made unoccupied. To make w_{i+1}^φ unoccupied an unlocked path connecting the original location of the unoccupied vertex and w_{i+1}^φ must exist in the bi-connected component. Since it is supposed that w_i^φ, w_{i+1}^φ, and the unoccupied vertex are all in the same bi-connected component the alternative path connecting w_{i+1}^φ and the unoccupied vertex in this bi-connected component avoiding w_i^φ must exist (since otherwise removal of w_i^φ would make the bi-connected component disconnected which is a contradiction). This path is used to transfer the unoccupied vertex to w_{i+1}^φ. Having w_{i+1}^φ unoccupied the vertex w_i^φ is unlocked and a is moved to w_{i+1}^φ along the edge $\{w_i^\varphi, w_{i+1}\}$. After this step, the required condition holds again (a supporting illustration is shown in Figure 4). ■

Lemma 5 ***(soundness of Exchange-Agents).*** If the arrangement of agents within the cycle C_0 is regarded as a permutation, then the output arrangement produced by the procedure *Exchange-Agents* corresponds to a permutation where the input agents a and b are transposed with respect to the permutation corresponding to the input arrangement. ■

Proof. It is needed to check whether the orderings of agents between a and b and between b and a (with respect to the positive orientation of the cycle) remain unchanged while a and b are transposed. This is done using detailed case analysis of what happens. Let $C_0 = [w_1^0, w_2^0, \ldots, w_l^0]$, then there are $l-2$ agents located in C_0 at the moment before the cycle is rotated positively (situation at line 9 of *Exchange-Agents* - see stage (i) in Figure 18). The agent a is already stored in v and the two unoccupied vertices are u and $next_\circlearrowleft(C_0, u)$. Let agents occupying vertices of the cycle in the interval between $\Phi_A(b)$ and u with respect to the positive orientation (excluding boundaries) are denoted $b_1, b_2, \ldots, b_k$ respectively; let agents occupying vertices of the cycle in the interval between $next_\circlearrowleft(C_0, u)$ and $\Phi_A(b)$ with respect to the positive orientation (again excluding boundaries) are denoted as $a_1, a_2, \ldots, a_{l-k-3}$. The series of ρ positive rotation of C_0 follows to move the agent b into $next_\circlearrowleft(C_0, u)$ (see stage (ii) in Figure 18). Now, all the agents $b_1, b_2, \ldots, b_k$, $a_1, a_2, \ldots, a_{l-k-3}$, and b are ρ steps forward with respect to their location before the series of rotations. Then the second unoccupied vertex (other than u) is moved in the positive direction towards $prev_\circlearrowleft(C_0, u)$ (recall, that the movement in the negative direction is not possible, since u is locked at the moment - see stage (iii) in Figure 18). Next, agents are exchanged: that is, b is moved to v and a is moved to $prev_\circlearrowleft(C_0, u)$ (see stage (iv) in Figure 18 and lines 14-17 of *Exchange-Agents*). At this step, agents $b_1, b_2, \ldots, b_k$ are ρ steps forward with respect to their location before the series of rotations; agent $a_1, a_2, \ldots, a_{l-k-3}$ are $\rho - 1$ forwards with respect to their location before the series of rotations (the difference is caused by the fact that unoccupied vertex went through agents $a_1, a_2, \ldots, a_{l-k-3}$ but not through agents

$b_1, b_2, \ldots, b_k$). Finally, the agent a is $\rho - 1$ steps forward with respect to the location of b before the series of rotations.

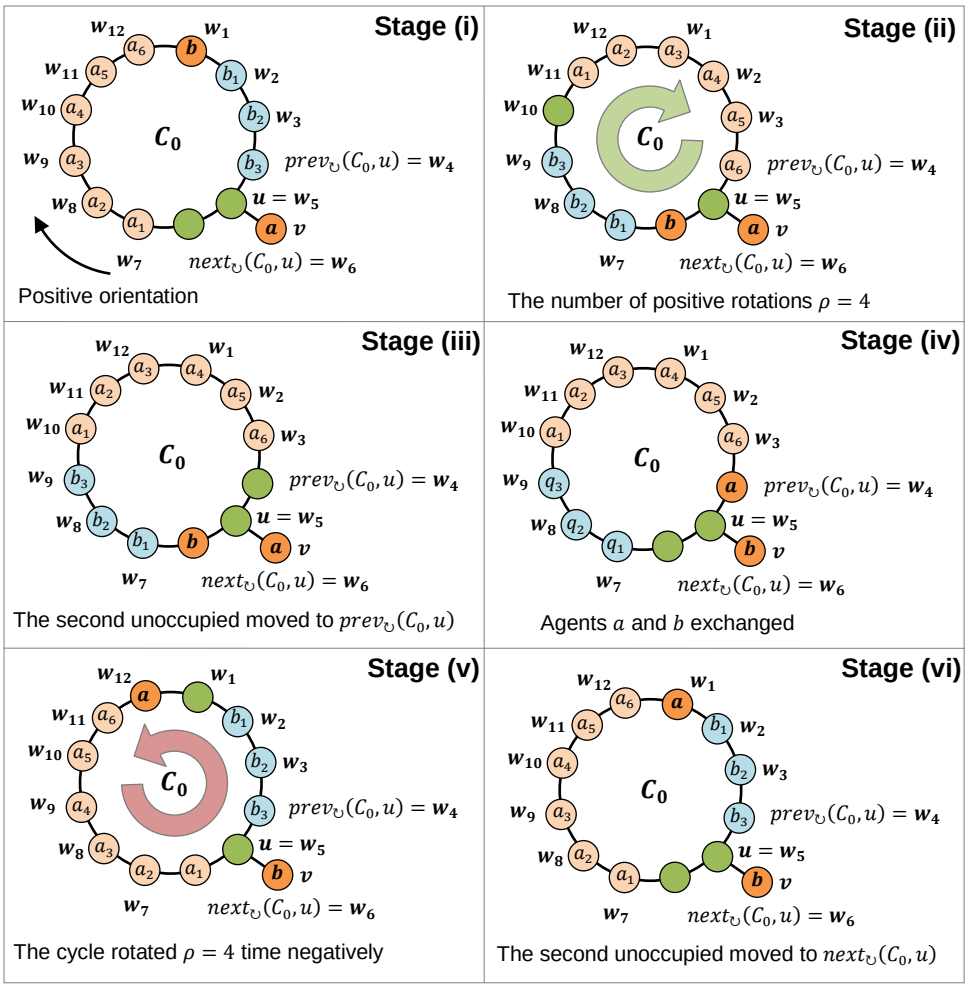


Figure 18. *The **progression of the exchange** of a pair of agents within an initial cycle of the handle decomposition.* Agents a and b in a cycle consisting of 12 vertices are exchanged while the ordering of other agents within the cycle is preserved. The figure illustrates the progression of the procedure *Exchange-Agents* from line 7 to 20.

The series of ρ rotation in the negative direction places agents $b_1, b_2, \ldots, b_k$ to their original positions; agents $a_1, a_2, \ldots, a_{l-k-3}$ are placed 1 step backward with respect to their original position before rotations, and a is 1 step backward with respect to the original position of b before the series of rotations (see stage (v) in Figure 18). This inconsistency however, is caused by a different location of the second unoccupied vertex which now between a and b_1 with respect to the positive orientation of the cycle (this was not the case in the original arrangement before rotations).

To see that the transposition of a and b has been really obtained, the movement of the second unoccupied vertex into $next_{\circlearrowleft}(C_0, u)$ in the negative direction can be done. This moves agents $a_1, a_2, \ldots, a_{l-k-3}$ to their original positions before rotations and the agent a to the original position of b (see stage (vi) in Figure 18). As this is a step used only for purposes of the proof, the algorithm actually does not perform it. ■

Proposition 6 ***(BIBOX - soundness and completeness).*** The *BIBOX* algorithm always terminates and produces a solution of a given input instance of pCPF $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. ■

Proof. To verify soundness and completeness of the *BIBOX* algorithm it is necessary to check preconditions of each operation performed in the course of its execution. This is a trivial task in almost all the cases except the case of searching for a path satisfying certain conditions. This issue concerns the search for vertex disjoint paths φ and χ within the main function *BIBOX-Solve* at line 2 and the search for a path connecting a given pair of vertices avoiding the locked ones.

The existence of vertex disjoint paths φ and χ has been already treated by Lemma 2. Thus, it remains to verify that a required unlocked path always exists.

A path containing unlocked vertices is constructed within the procedure *Make-Unoccupied* (lines 2-3) which is called by *Solve-Regular-Handle* (lines 3, 5, 12, 16, 26, and 35), *Solve-Original-Cycle* (lines 4, 6, 10, and 12), and *Exchange-Agents* (lines 2, 8, and 13). A pair of vertex disjoint paths containing unlocked vertices constructed within the procedure *Move-Agent* (lines 1-5) which is called by *Solve-Regular-Handle* (lines 10, 24, and 33). All these cases must be examined.

There will be the following invariant within *Solve-Regular-Handle* – at the beginning of every iteration of the loop at line 7, an unoccupied vertex must be located in the not yet solved part of the graph. More precisely, let the bi-connected subgraph without the internal vertices of the already solved handles be denoted as $G^{\sim}$ and let $G^{\sim}$ without the internal vertices of H_c, where H_c is the currently solved handle by *Solve-Regular-Handle*, be denoted as G' (see Figure 6). Then an unoccupied vertex is needed to be located in G' every time the loop at line 7 starts. The assumption holds at the beginning and it is needed to check if it holds after every iteration of the loop. Furthermore, an invariant that both unoccupied vertices are located in $G^{\sim}$ at the start of *Solve-Regular-Handle* will be also checked. Again, this invariant holds at the beginning.

Vertices w and z which are used as parameters of the call of *Make-Unoccupied* at lines 3 and 5 respectively of *Solve-Regular-Handle* are both in G'. Since $G^{\sim}$ is completely unlocked at lines 3 of *Solve-Regular-Handle* and it is assumed that an unoccupied vertex is located in $G^{\sim}$, an unlocked path connecting w and an unoccupied vertex must exist. The construction of a path within the call at line 5 of *Solve-Regular-Handle* must additionally take into account that w is locked. As the subgraph $G^{\sim}$ is bi-connected, it remains connected even if w is removed and hence the path exists.

At line 12 of *Solve-Regular-Handle*, a connection vertex v^c of the currently solved handle H_c is being made unoccupied while internal vertices of H_c and the second connection vertex u^c are locked. According to above invariants and the fact that the call of *Move-Agent* at line 10 does not invalidate them, as it is prevented from using internal vertices of H_c by locking them at line 8, an unoccupied vertex is now located in G' (except u^c). The graph G' is bi-connected and without u^c, which is locked just before, it is all unlocked and still connected. Hence the required path exists.

The call of *Make-Unoccupied* at line 16 of *Solve-Regular-Handle* has the connection vertex u^c of the currently solved handle H_c as a parameter. The subgraph G' is now unlocked and according to invariants an unoccupied vertex is located in G'. Since G' is connected, there exists an unlocked path connecting u^c and the unoccupied vertex.

At line 26 of *Solve-Regular-Handle*, the connection vertex u^c of the handle H_c is made unoccupied. The situation is that a vertex y, which is in G' and outside the cycle associated with the current handle H_c, is locked while the rest of G' is unlocked. Again, the unlocked part of the graph corresponds to a bi-connected subgraph G' from which one vertex was removed. Thus, the unlocked part of the graph constitutes a connected component. An unoccupied is also located in the unlocked part.

This holds from the invariants and from the fact that movements at lines 20 and 24 cannot relocate it outside G' as *Rotate-Cycle*$^+$ does not relocate the input unoccupied vertex and *Move-Agent* cannot go outside the unlocked part which is exactly G' at the moment due to locking of internal vertices of H_c at line 22. Hence, there exists an unlocked path connecting the unoccupied vertex and u^c.

At line 35 of *Solve-Regular-Handle* the task is to make unoccupied a connection vertex v^c of the handle H_c. The situation is again very similar; the unlocked part of the graph is constituted by G' without u^c. Thus, unlocked vertices constitute a connected subgraph. The unoccupied vertex must be located in the unlocked part as it was located in u^c after the execution of line 26 and subsequent movements cannot relocate it outside G' (*Rotate-Cycle*$^-$ at line 29 does not relocate the input unoccupied vertex and *Move-Agent* at line 33 remains in the unlocked part). Thus, there exists an unlocked path connecting the unoccupied vertex and v^c.

An unoccupied vertex is located in G' at the end of the iteration of the loop starting at line 7 since it is v^c in both major execution branches (notice that calls of *Rotate-Cycle*$^+$ at line 14 and 37 respectively preserve positions the unoccupied vertex v^c). Thus, the first invariant holds. Since it is assumed that goal positions of unoccupied vertices are within the initial cycle, no unoccupied vertex can be stored in H_c. Hence, both unoccupied vertices are in G' at the end of the execution of the loop (that is, they are within $G^{\sim}$ with respect to the processing of next handle).

The soundness of the procedure *Solve-Original-Cycle* is partially implied by the soundness of the procedure *Exchange-Agents* which is treated by Lemma 5. The basic assumption of *Solve-Original-Cycle* is that both unoccupied vertices are located in the original cycle C_0 of the handle decomposition; all the vertices of the graph except C_0 are locked. The assumption directly corresponds to the second invariant preserved along the calls of *Solve-Regular-Handle* within the loop at lines 5-7 of *BIBOX-Solve*.

At line 4 of *Solve-Original-Cycle* a vertex w_1^0 (the first vertex of the cycle with respect to the positive orientation) is being made unoccupied. An unlocked path in the cycle from any of its vertices to w_1^0 exists, hence making w_1^0 unoccupied is possible. The situation at line 6 of *Solve-Original-Cycle* is little bit different; now the vertex w_1^0 is locked and a vertex w_2^0 (the second vertex of C_0 with respect to the positive orientation) is being made unoccupied. Thus, an unlocked path connecting the second unoccupied vertex with w_2^0 is searched. Such path exists since removing w_1^0 from the cycle does not disconnect it. The situation at lines 10 and 12 of *Solve-Original-Cycle* is analogical.

The soundness of the procedure *Move-Agent* itself is treated separately by Lemma 4. However, preconditions of the Lemma 4 need to be checked – that is, whether all the calls of *Move-Agent* moves an agent within the single unlocked bi-connected component and whether the unoccupied vertex is located in the same unlocked bi-connected component as well.

The situation before the call of *Move-Agent* at line 10 of *Solve-Regular-Handle* is that G' is unlocked while the rest of the graph is locked. An unoccupied vertex is located in G' which is ensured by the invariant. The task is to move an agent $\Phi_A^+(w_i^c)$, which is known to be located in G' (this is, treated by the execution branch at line 9), to the connection vertex u^c of the handle H_c. As the unoccupied vertex and both the agent $\Phi_A^+(w_i^c)$ and u^c are located in G' constituting a bi-connected component, preconditions of Lemma 4 are satisfied.

The call of *Move-Agent* at line 24 of *Solve-Regular-Handle* moves the agent $\Phi_A^+(w_i^c)$ to a vertex y which is located in G' and outside the cycle associated with the handle H_c at the same time. Again, G' is unlocked while the rest of the graph is locked. The agent $\Phi_A^+(w_i^c)$ is known to be located in the connection vertex v^c of H_c and one of the unoccupied vertices is the second connection vertex u^c.

Thus, the unlocked vertices constitutes a bi-connected component where the agent $\Phi_A^+(w_i^c)$, the vertex y, and the unoccupied vertex are located. Hence, preconditions of Lemma 4 are satisfied.

Finally, the task of the call of *Move-Agent* at line 33 of *Solve-Regular-Handle* is to move an agent $\Phi_A^+(w_i^c)$ to a connection vertex u^c of the current handle H_c which is assumed to be unoccupied at the moment. It is known that the agent $\Phi_A^+(w_i^c)$ is located in y from the previous case. G' is again unlocked while the rest of the graph is locked. Thus, the agent $\Phi_A^+(w_i^c)$ and the unoccupied vertex u^c are both located in G' which is a bi-connected component. Thus, preconditions of Lemma 4 are satisfied again.

At this point, it is possible to conclude that all the steps of the algorithm are correctly defined. Since the number of successfully placed agents strictly increases as the algorithm proceeds, the algorithm always terminates and produces a solution to the input instance. ■

Proposition 7 ***(BIBOX – worst-case time complexity).*** The **worst-case time** complexity of the *BIBOX* algorithm is $\mathcal{O}(|V|^3)$ with respect to an input pCPF instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. ■

Proof. The construction of a handle decomposition (line 1 of *BIBOX-Solve*) takes $\mathcal{O}(|V| + |E|)$ steps (Lemma 1). The same estimation holds for transforming the goal arrangement of agents (line 2 of *BIBOX-Solve*) and augmenting the final solution (line 9 of *BIBOX-Solve*) according to a pair of vertex disjoint paths φ and χ.

There are at most $|V|$ agents (since $|A| < |V|$) to be placed within handles of a handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$. Placing an agent a within H_c with $c \in \{1,2,\ldots,d\}$ requires at most $|H_c|$ rotations of the cycle $C(H_c)$ in the positive direction (procedure *Rotate-Cycle*$^+$) in the case when a is needed to be moved outside H_c. Then, at most $|H_c|$ rotations of $C(H_c)$ in the negative direction (procedure *Rotate-Cycle*$^-$) are necessary to put agents in H_c back to their original positions; and finally, one rotation of $C(H_c)$ in the positive direction is necessary to get the agent a to its position within H_c. Altogether at most $2|H_c| + 1$ rotations of $C(H_c)$ are necessary. One rotation of the cycle $C(H_c)$ requires at most $|C(H_c)|$ steps. If the agent a does not need to be moved outside H_c only one positive rotation of $C(H_c)$ is needed. Thus, all the rotations needed to place the agent a consume at most $|C(H_c)| \cdot (2|H_c| + 1)$ steps.

It is also necessary to move the agent a (procedure *Move-Agent*) during the placement operation. There are up to 2 calls of *Move-Agent* per agent placement within the handle H_c. A more careful analysis must be done here since the agent a must be moved along a path of the length up to $|V|$ while non-trivial amount of work needs to be done per each edge traversal.

A vertex in front of the current location of a needs to be made unoccupied every time an edge is traversed by a. Thus a path connecting the unoccupied vertex and the location in front of a must be found while the vertex containing a should be avoided by the path. Agents are then shifted along the found path. The path should be searched in the graph constituted by the initial cycle and handles of the handle decomposition that contains at least one internal vertex. Such a graph contains only linear number of edges with respect to the number of vertices and thus the search for the path can be completed in $\mathcal{O}(|V|)$ steps. The subsequent shifting of agents consumes at most $|V|$ steps. Hence, the single traversal of an edge by the agent a requires $\mathcal{O}(|V|)$ steps. Altogether, $\mathcal{O}(|V|^2) + \mathcal{O}(|V| + |E|)$ steps are required by operations for moving of agents.

There are also up to 5 calls of the operation for making some vertex unoccupied (procedure *Make-Unoccupied*) per agent placement. The operation for making some vertex unoccupied requires $\mathcal{O}(|V| + |E|)$ steps; this is accounted to the search for a shortest path connecting the original and the

goal location. Shifting agents itself along the found path is less consuming; it requires at most $|V|$ steps. Thus, at most $5|V|+ \mathcal{O}(|V| + |E|)$ steps are consumed by making vertices unoccupied in course of placing a.

In total, at most $(2|H_c| + 1) \cdot |C(H_c)| + 2|V|^2 + 5|V| + \mathcal{O}(|V| + |E|)$ steps are necessary to place a into H_c. Since $|H_c| \leq |C(H_c)| \leq |V|$, the total number of steps is at most $(2|V| + 1) \cdot |V| + 2|V|^2 + 5|V| + \mathcal{O}(|V| + |E|)$ which is $\mathcal{O}(|V|^2)$.

The remaining operations consume the constant time. Since there are at most $|V|$ agents, the whole process of placing agents into handles takes $\mathcal{O}(|V|^3)$ steps.

It remains to analyze the time required by placing agents within the original cycle C_0. Each agent a requires 2 operations of making a vertex unoccupied (the first and the second vertex C_0 are made unoccupied – lines 4 and 6 of *Solve-Original-Cycle*) and at most one operation of exchanging agents. Since the initial and the goal position of both mentioned relocations of the unoccupied vertex are located in C_0, the operation requires only $|C_0|$ steps in the worst-case. The operation of exchanging agents requires at most $2|C_0|$ rotations in the positive direction (lines 5 and 11 of *Exchange-Agents*) and at most $|C_0|$ rotations in the negative direction (line 19 of *Exchange-Agents*). Next, there are 3 calls of the operation for making some vertex unoccupied (call of the procedure *Make-Unoccupied* at lines 2, 8, and 13). Observe that the unoccupied vertex and the target vertex of the relocation are located in C_0 in all the cases. Thus, each of these operations requires at most $|C_0|$ steps. Altogether, $3|C_0|$ steps are required for making vertices unoccupied during exchanging a pair of agents. The time consumption of the remaining operations performed during a single exchange of agents is constant.

A single exchange of a pair of agents requires at most $3|C_0|^2 + 5|C_0|$ steps in total. Placing all the agents into the original cycle hence consumes at most $|C_0| \cdot (3|C_0|^2 + 5|C_0|)$ steps. Since $|C_0| < |V|$, the total number of steps required for solving the initial cycle is at most $|V| \cdot (3|V|^2 + 5|V|)$ which is $\mathcal{O}(|V|^3)$.

It was shown that the worst-case time of $\mathcal{O}(|V|^3)$ is necessary to solve regular handles as well as the initial cycle thus the worst-case time complexity of the BIBOX algorithm is $\mathcal{O}(|V|^3)$. ■

Using almost the same arguments as in the above proof it is possible to calculate the worst-case makespan of solutions generated by the BIBOX algorithm. Notice that the algorithm generates movement of the agent in almost every step referred in the time complexity analysis.

Proposition 8 ***(BIBOX – makespan of the solution)*****.** The **worst-case makespan** of the solution produced by the BIBOX algorithm (that is, the number ζ) for an input instance of pCPF $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ is $\mathcal{O}(|V|^3)$. ■