Abstract Path Planning for Multiple Robots: An Empirical Study

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 multi-robot path planning is addressed in this work. The task is to construct a sequence of moves for each robot of the group of robots that are moving in certain environment. Initially each robot is placed in some location in the environment and it needs to go to the given goal position. Robots must avoid obstacles and must not collide with each other along the process of relocation according to the constructed sequences of moves. An abstraction where the environment is modeled as an undirected graph is adopted – vertices represent locations in the environment and edges represent unblocked way between two neighboring locations. Robots are represented as elements placed in vertices of the graph while at least one vertex is unoccupied to allow robots to move. The move is allowed into the unoccupied vertex or into the vertex being vacated by an allowed move supposed that no other robot is entering the same target vertex.

Two polynomial time algorithms for solving the problem of multi-robot path planning sub-optimally with respect to the makespan and their variants are presented in this work. Both algorithms are targeted on the case with bi-connected graphs with relatively small number of unoccupied vertices. The precise theoretical and experimental analysis of presented algorithms is provided. It has been shown theoretically and experimentally that presented algorithms outperform the only existent algorithm capable of solving the given class of the problem in terms of quality of generated solutions. In terms of speed, presented algorithms proved to be as fast as the existent one at least.

Keywords: multi-robot, path planning, multi-agent, coordination, sliding puzzle, (n^2-1)-puzzle, 15-puzzle, domain dependent planning, makespan optimization, *BIBOX*, *BIBOX-θ*.

1. Introduction and Motivation

This manuscript is devoted to a problem of *path planning for multiple robots* [12, 13, 16]. Consider a group of mobile robots that are moving in some environment (for example in the 2-dimensional plane with obstacles). Each robot of the group is given an initial and a goal position in the environment. The question of interest is how to determine a sequence of motions for each robot of the group, which relocates the robot from the given initial position to the given goal one. Physical limitations must be respected by robots along the whole process of relocation: robots must **not collide** with each other and they must **avoid obstacles** in the environment during their movements.

The problem of multi-robot path planning is **motivated** by many practical tasks. Various tasks of navigating a group of mobile robots can be formulated as multi-robot path planning. However, the primary motivations for the problem are tasks of moving certain entities within an environment with a limited free space. Thus, the formulation of the problem is not restricted to the case where robots are actually represented by mobile robots. Such real-life examples include rearranging of shipping containers in warehouses (a robot is represented by a shipping container - see Figure 1) or coordination of vehicles in

dense traffic (robot = vehicle). Moreover, the reasoning about these rearrangement/coordination tasks should not be limited to physical entities only. A robot may be represented by a virtual entity or by a piece of commodity as well. Thus, many tasks such as planning of data transfer between communication nodes with limited storage capacity (robot = data packet), commodity transportation in the commodity transportation network (robot = certain amount of commodity), or even the motion planning of large groups of virtual agents in the computer-generated imagery can be expressed as the problem of multi-robot path planning.

Figure 1. *An illustration of* ***shipping container rearranging***. This problem can be formulated as path planning for multiple robots where robots are represented by containers.

The **primary aim of this manuscript** is to develop scalable algorithms for solving the problem of multi-robot path planning for the case where robots are moving in the environment with relatively **small free space**. This case of the problem represents the most difficult situation since the probability of collisions between robots is very high. It can be intuitively perceived, that the problem is much easier with lot of free space in the environment. In such a case, probability of collisions between robots is low. Hence, it is possible to plan movements of individual robots almost independently using algorithms for finding shortest paths [1] connecting their initial and goal positions.

An **abstraction** of the problem has to be adopted to be able to make some reasoning about the problem. The abstraction used in this manuscript consists in modeling the environment where robots are moving as an undirected graph. Vertices of the graph represent locations within the environment and edges represent possibility of going from one location to the neighboring one through the edge. Robots are placed in vertices of the graph and they are allowed to move into the neighboring vertex if it is **unoccupied or currently being vacated** while no other robot is entering the same target vertex. Time is discrete in this abstraction – individual time steps are isomorphic to the structure of *natural numbers*. Movements of robots are **instantaneous**; that is, a robot can move from a vertex to the neighboring one between two succeeding time steps while no middle positions are considered.

There is variety of ways how to create an abstract instance of a given specific real-life multi-robot path planning instance. It is necessary to make decisions how to sample locations in the original environment in order to make the abstract instance accurate enough to model the real-life situation as precisely as needed. Nevertheless, these issues are out of scope of this work.

The main **contribution** of this manuscript is presentation of scalable algorithms for solving the problem and their precise theoretical and experimental analysis. Algorithms

presented in this work have been already published by the author in several conference proceedings [16, 17, 18, 19]. However, space limitations of proceedings did not allow providing complete theoretical and experimental analysis. Thus, the analysis provided in this work represents the new material.

In the context of multi-robot path planning, works on problems of *motion planning over graphs* must be mentioned [8, 9, 10, 31] since they are closely related. Namely, works on so called problems of *pebble motion on graphs* (which the most widely known representative is the *15-puzzle* or *(n^2-1)-puzzle*) [8, 10, 31] represents almost the same problem as multi-robot path planning. The difference lays in the condition on the dynamicity in the problem - moves are allowed into **currently unoccupied** vertices only while no other pebble is entering the same target vertex in the problem of pebble motion on a graph. Many theoretical results are known for pebble motion on a graph. It is known that the problem can be solved in polynomial time (particularly in $\mathcal{O}(|V|^3)$ for $G = (V, E)$ modeling the environment) with solution consisting of polynomial number of moves (again it is $\mathcal{O}(|V|^3)$ moves) [8, 31]. Moreover, it is known that the decision version of the optimization variant of pebble motion on a graph is *NP*-complete [10] (this has been actually shown for generalized variant of the 15-puzzle). Recently, it has been shown that the decision version of the optimization variant of the problem of multi-robot path planning is *NP*-complete as well [20, 22].

Many results from works on pebble motion on graphs are utilized in the development of solving algorithms for multi-robot path planning within this work. As it is not tractable to produce optimal solutions of the problem in the perspective of the above negative results, all the algorithms developed in this work thus produce sub-optimal solutions. However, quality of solutions is still an objective especially with respect to real-life instances.

One of the minor aims of this manuscript is to clarify terminology, since many works use the term multi-robot path planning for pebble motion on a graph in fact. This evokes an impression that these problems are different, but deeper analysis shows that authors are dealing with pebble motion on a graph in all the cases. The different and more reasonable definition of multi-robot path planning is introduced in this work. A more detailed discussion on this aspect is given along with definitions of problems.

The organization of the manuscript is as follows: a formal definition of the problem of pebble motion on a graph is recalled and a definition of the abstraction of multi-robot path planning is given in **Section 2**. Some basic properties of problems and their correspondence is discussed in this section too. **Section 3** represents the core of the manuscript – two new algorithms for solving the problem of multi-robot path planning are presented here. The detailed analysis regarding the correctness and the complexity of presented algorithms is also provided in this section. The next section – **Section 4 -** is devoted to parallelism increasing techniques, which consequently increase quality of solutions. An extensive experimental evaluation of presented algorithms is provided in Section 5. The experimental evaluation is targeted on the competitive comparison with the existent algo-

rithm for pebble motion on a graph which is applicable on multi-robot path planning as well. Concluding remarks and related works are discussed in **Section 6**.

2. Pebble Motion on a Graph and Multi-robot Path Planning

Problems of *pebble motion on a graph* and *multi-robot path planning* are formally defined in this section. A relation of both problems is discussed and their theoretical properties are described.

The primary problem studied in this manuscript is the problem of multi-robot path planning. It is very similar to the problem of pebble motion on a graph. The problem of pebble motion on a graph has been already studied in the literature and lots of theoretical results are known for it. Here, the problem of pebble motion on graph and the related results are used as a theoretical foundation for studying the problem of path planning for multiple robots.

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

A relatively strong abstraction is adopted in this work. The environment with obstacles within that the robots are moving is modeled as an undirected graph. The vertices of this graph represent positions in the environment and the edges model an unblocked way from one position to another. The time is discrete in this abstraction; it is an infinite linearly ordered set isomorphic to the set of natural numbers where each element is called a *time step* (time steps are numbered starting with 0). At each time step, each robot is located in a vertex. A motion of a robot is an instantaneous event. That is, if the robot is placed in a vertex at a given time step then the result of the motion is the situation where the robot is placed in the neighboring vertex at the following time step. The is allowed to enter a neighboring vertex supposed it is unoccupied or being vacated by another robot using an allowed move while no other robot is trying to enter the same target vertex.

The problem of pebble motion on a graph works with pebbles instead of robots (which however does not introduce any formal difference itself). The most important difference in both problems consists in a condition on the allowed moves. Allowed motions of pebbles are more restrictive than in the case of robots in multi-robot path planning.

2.1. *Formal Definitions of Motion Problems*

The following two definitions formalize a problem of *pebble motion on a graph* (also called a *pebble motion puzzle*, *sliding box puzzle*; special variants are known as *15-puzzle* and (n^2-1)-*puzzle*) [5, 19] and the related problem of *multi-robot path planning* [12,13 16]. Both problems and their solutions are illustrated in Figure 2.

Definition 1 ***(problem of pebble motion on a graph).*** Let $G = (V, E)$ be an undirected graph. Next, let $P = \{\bar{p}_1, \bar{p}_2, \dots, \bar{p}_\mu\}$ where $\mu < |V|$ be a set of pebbles. The graph models an environment in which pebbles are moving. An ***initial arrangement*** of pebbles is defined by a uniquely invertible function $S_P^0 : P \longrightarrow V$ (that is $S_P^0(p) \neq S_P^0(q)$ for every $p, q \in P \in$ with $p \neq q$). A ***goal arrangement*** of pebbles is defined by another uniquely invertible function $S_P^+ : P \longrightarrow V$ (that is $S_P^+(p) \neq S_P^+(q)$ for every $p, q \in P$ with $p \neq q$). A problem of ***pebble motion on a graph*** is the task to find a number ξ and a sequence $\mathcal{S}_P = [S_P^0, S_P^1, \dots, S_P^\xi]$ where $S_P^k : P \longrightarrow V$ is a uniquely invertible function for every $k = 1,2, \dots, \xi$. Additionally, the following conditions must hold for the sequence $\mathcal{S}_P$:

(i) $S_P^\xi = S_P^+$; that is, all the pebble reaches their destination vertices.

(ii) 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 can either stay in a vertex or move into the neighboring vertex between each two successive time steps.

(iii) 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 $S_P^k(q) \neq S_P^{k+1}(p)\ \forall q \in P$ such that $q \neq p$; must hold for every $p \in P$ and $k = 1,2, \dots, \xi - 1$; that is, a pebble can move into an unoccupied neighboring vertex only. This condition together with unique invertibility of functions forming $\mathcal{S}_P$ implies that no two pebbles can enter the same target vertex at the same time step.

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

The **notation** with a stripe above the symbol is used to distinguish a constant from a variable (for example, $p \in P$ is a variable while $\bar{p}_2$ is a constant; sometimes a constant parameterized by a variable or by an expression will be used – for example $\bar{p}_i$ denotes a constant parameterized by an index $i \in \mathbb{N}$; the parameterization by an expression will be clear from the context).

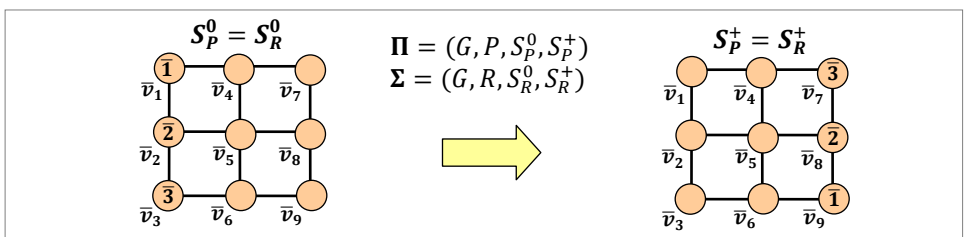
When speaking about a move at time step k, it is referred to the time step of commencing the move (exactly, the move is performed between time steps k and $k + 1$).

The term multi-robot path planning has been already used in literature. However, it did not introduce any new concept since it has been used as synonym for pebble motion on a graph in fact. In the work titled "*Exploiting Subgraph Structure in Multi-Robot Path Planning*" [13] the dynamicity of the problem is described as follows:

> *"Further, we shall assume that the map is constructed so that collisions only occur when one robot is entering a vertex v at the same time as another robot is occupying, entering or leaving this vertex."*

In other words, a robot can enter a vertex if and only if it is unoccupied at the time of commencing the move and no other robot is entering the same target vertex, which is exactly the definition of the dynamicity in the problem of pebble motion on a graph.

An alternative supposedly more reasonable definition of multi-robot path planning is adopted in this work. A problem of multi-robot path planning is a **relaxation** of the problem of pebble motion on a graph. The condition that the target vertex of a pebble/robot must be vacated in the previous time step is relaxed. Thus, the motion of a robot entering the target vertex, that is simultaneously vacated by another robot and no other robot is trying to enter the same target vertex, is allowed in multi-robot path planning. However, there must be some leading robot initiating such chain of allowed moves by moving into a currently unoccupied vertex which no other robot is entering at the same time step (that is, robots can move "like a train" with the leading robot in front). The problem is formalized in the following definition.



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

$\xi = 6$

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

Solution of the problem of *path planning for multiple robots* Σ with $R = \{\bar{1}, \bar{2}, \bar{3}\}$

$\zeta = 4$

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

Figure 2. *An illustration of problems of* ***pebble motion on a graph*** *and* ***multi-robot path planning***. Both problems are illustrated on the same graph with the same initial and goal positions. The task is to move pebbles/robots from their initial positions specified by S_P^0/S_R^0 to the goal positions specified by S_P^+/S_R^+. A solution of the makespan 6 ($\xi = 6$) is shown for the problem of pebble motion on a graph and a solution of the makespan 4 ($\zeta = 4$) is shown for the problem of multi-robot path planning. Notice the differences in parallelism between both solutions – multi-robot path planning allows a higher number of moves to be performed in parallel thanks to weaker requirements on solutions.

Definition 2 ***(problem of multi-robot path planning).*** Again, let $G = (V, E)$ be an undirected graph. Now a set of robots $R = \{\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_\nu\}$ where $\nu < |V|$ is given instead of the set of pebbles. Similarly, the graph models an environment in which robots are moving. The ***initial arrangement*** of robots is defined by a uniquely invertible function $S_R^0 : R \longrightarrow V$ (that is $S_R^0(r) \neq S_R^0(s)$ for every $r, s \in R$ with $r \neq s$). The ***goal arrangement*** of robots is defined by another uniquely invertible function $S_R^+ : R \longrightarrow V$ (that is $S_R^+(r) \neq S_R^+(s)$ for every $r, s \in R$ with $r \neq s$). A problem of ***multi-robot path planning*** is the task to find a

number ζ and a sequence $\mathcal{S}_R = [S_R^0, S_R^1, \dots, S_R^\zeta]$ where $S_R^k: R \longrightarrow V$ is a uniquely invertible function for every $k = 1,2, \dots, \zeta$. The following conditions must hold for the sequence $\mathcal{S}_R$:

(i) $S_R^\zeta = S_R^+$; that is, all the robots reaches their destination vertices.

(ii) Either $S_R^k(r) = S_P^{k+1}(r)$ or $\{S_R^k(r), S_R^{k+1}(r)\} \in E$ for every $r \in R$ and $k = 1,2, \dots, \zeta - 1$; that is, a robot can either stay in a vertex or move to the neighboring vertex at each time step.

(iii) If $S_R^k(r) \neq S_R^{k+1}(r)$ (that is, the robot r moves between time steps k and $k+1$) and $S_R^k(s) \neq S_R^{k+1}(r)\ \forall s \in R$ such that $s \neq r$ (that is, no other robot s occupies the target vertex at time step k), then the move of r at the time step k is called to be ***allowed*** (that is, the robot r moves into a currently unoccupied neighboring vertex – a ***leading*** robot). If $S_R^k(r) \neq S_R^{k+1}(r)$ and there is $s \in R$ such that $s \neq r \wedge S_R^k(s) = S_R^{k+1}(r) \wedge S_R^k(s) \neq S_R^{k+1}(s)$ (that is, the robot r moves into a vertex that is being left by the robot s) and the move of s at the time step k is allowed, then the move of r at the time step k is also ***allowed***. All the moves of robots at all the time steps **must be allowed**. Analogically, this condition together with the requirement on unique invertibility of functions forming $\mathcal{S}_R$ implies that no two robots can enter the same target vertex at the same time step.

The instance of the problem of multi-robot path planning is formally a quadruple $\Sigma = (G, R, S_R^0, S_R^+)$. The solution of the problem Σ will be sometimes denoted as $\mathcal{S}_R(\Sigma) = [S_R^0, S_R^1, \dots, S_R^\zeta]$. □

The numbers ξ and ζ are called ***makespan*** of the solution of pebble motion on a graph and multi-robot path planning respectively. The makespan need to be distinguished from the ***size*** of the solution, which is the total number of moves performed by pebbles/robots.

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

Several basic properties of solutions of problems of pebble motion on graphs and multi-robot path planning are summarized in this section.

Notice that a solution of the problem of pebble motion on a graph as well as a solution of the problem of multi-robot path planning allows a pebble/robot to **stay** in a vertex for more than a single time step. It is also possible that a pebble/robot **visits** the same vertex **several times** within the solution. Hence, the sequence of moves for a single pebble/robot does not necessarily form a simple path in the given graph.

Notice further that both problems intrinsically allow parallel movements of pebbles/robots. That is, more than one pebble/robot can perform a move in a single time step. However, multi-robot path planning allows higher motion parallelism due to its weaker requirements on robot movements (the target vertex is required to be unoccupied only for the leading robot in the current time step – see Figure 2). More than one unoccupied vertex is necessary to obtain parallelism in the problem of pebble motion on a graph. On the other hand, it is sufficient to have a single unoccupied vertex to obtain parallelism within

the solution of multi-robot path planning problem (consider for example robots moving around a cycle).

Proposition 1 ***(problem correspondence).*** Let $\Pi = (G, P, S_P^0, S_P^+)$ be an instance of the problem of pebble motion on a graph and let $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \ldots, S_P^\xi]$ be its solution. Then $\mathcal{S}_R(\Sigma) = \mathcal{S}_P(\Pi)$ is a solution of an instance of the problem of path planning for multiple robots $\Sigma = (G, P, S_P^0, S_P^+)$. In other words, the instance of the multi-robot path planning problem consists of the same graph, the set of robots is represented by the set of pebbles, and the initial/goal positions of robots are the same as in the case of pebbles. Then the solution of the instance of the pebble motion problem can be used as a solution of the corresponding instance of the multi-robot path planning problem. ■

Proof. The proof of the statement is straightforward using Definition 1 and Definition 2. The condition on sequence of moves required by Definition 2 needs to be checked for $\mathcal{S}_P(\Pi)$. Conditions (i) and (ii) of Definition 2 are trivially satisfied. Condition (iii) is also satisfied since it holds that if $S_P^k(p) \neq S_P^{k+1}(p)$ then $S_P^k(q) \neq S_P^{k+1}(p)$ $\forall q \in P$ such that $q \neq p$ is true for every $p \in P$ and $k = 1,2, \ldots, \xi - 1$. In other words, all the moves within $\mathcal{S}_P(\Pi)$ are **allowed**. ■

There is a variety of modifications of the defined problems. A natural additional requirement is to produce solutions with the **shortest possible makespan** (that is, the numbers ξ or ζ respectively are required to be as small as possible). Unfortunately, this requirement makes the problem of pebble motion on a graph intractable. It is shown in [10] that the optimization variant of a special case of the problem of pebble motion on a graph is ***NP*-hard [**3**]**. The restriction forming the special case adopted in [10] works with a graph that can be embedded in plane as a square grid with a single unoccupied vertex - this case is known as $N \times N$ *puzzle* (also known as $(n^2 - 1)$-*puzzle*). Hence, the general optimization variant of the problem of pebble motion on a graph NP-hard as well.

A restriction of both types of problems on *bi-connected graphs* [30] (for the precise definitions see Section 3.1.1) represents an important subclass with respect to the existence of a solution. Hence, it is a reasonable question what is the complexity of these classes of problems. Since the grid graph forming the mentioned $N \times N$ puzzle is **bi-connected,** the immediate answer is that the optimization variant of the problem of pebble motion on a bi-connected graph with a single unoccupied vertex is again NP-hard.

However, it is not possible to simply make any similar statement about the complexity of the optimization variant of multi-robot path planning based on the above facts. The situation here is complicated by the inherent parallelism, which can reduce the makespan of the solution significantly. Constructions used for the $N \times N$ puzzle in [10] thus no longer work. Nevertheless, it has been recently shown by the author that the optimization variant of multi-robot path planning is *NP*-hard too [20, 22].

Observe further that difficult cases of the problem of pebble motion on a graph have a single unoccupied vertex. This fact may raise the question how the situation is changed

when there are **more** than one **unoccupied vertices**. More unoccupied vertices may simplify the problem. Unfortunately, it is not the case. The pebble motion problem on a general graph with the fixed number of unoccupied vertices is still NP-hard since multiple copies of the $N \times N$ puzzle from [10] can be used to add as many unoccupied vertices as needed - the resulting graph may be disconnected. Without providing further details, the the instance of multi-robot path planning used in a reduction to prove *NP*-hardness of the problem had many unoccupied vertices and its graph was connected (or even bi-connected). Thus, a mere allowance of many unoccupied vertices with no additional conditions does not simplify the problem.

Without the requirement on the **optimality** of the makespan of solutions, the situation is much easier; the problem of pebble motion on a graph is in the ***P* class** as it is shown in [8, 31]. Due to Proposition 1, the problem of path planning for multiple robots is in the *P* class as well. Thus, it seems that pebble motion on a graph and multi-robot path planning problems have been already resolved. However, constructions proving the membership of the problem of pebble motion on a graph into the *P* class used in [8, 31] generate solutions that are too long for practical use [17, 18, 19]. 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 a problem of pebble motion on a graph. Thus, alternative solving methods has been developed and will be described in this work [16, 17, 18, 19].

3. Sub-optimal Solving Algorithms

This section is devoted to algorithms for solving problems of motion on a graph in **polynomial time** that generate solution of the sub-optimal makespan. All the algorithms developed in the following text are designed for the problem of pebble motion on a graph. Due to Proposition 1, algorithms for pebble motion on a graph apply also for multi-robot path planning. However, the practice of solving multi-robot path planning problems using algorithms for pebble motion on a graph does not reflect the possibility of higher parallelism in multi-robot path planning. Particularly, parallelism in the form of the "train like" movement of a queue of robots is never produced in this way. This drawback can be augmented by a post-processing step that increases parallelism. Fortunately, this post-processing step can be made fast enough to settle with such an approach (see Section 3.3 which is solely devoted to this issue).

There already exist sub-optimal algorithms for solving the problem of pebble motion on a graph in polynomial time described in [8, 31]. Thus, a question why to develop a new algorithm of the same kind for this problem may arise in this context. The main reason for developing new algorithms is that existent ones from [8, 31] produce solution that are too long with respect to the makespan and hence unsuitable for practice. This claim will be shown experimentally in Section 5 where the new and the existent algorithm are compared.

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

The first algorithm that will be recalled here comes from [16]. It was originally called a novel algorithm since it represents an alternative to algorithms from [8, 31]. This algorithm solves in fact a yet more special variant of the problem of pebble motion on a graph. The input problem should consist of a *non-trivial bi-connected graph* (that is, bi-connected graph not isomorphic to a cycle) with exactly **two unoccupied** vertices. Several augmentations will be described in following sections that shift this algorithm to be able to solve a general variant of the problem of pebble motion on a bi-connected graph.

3.1.1. *Graph-theoretical Preliminaries*

Some notions from the graph theory need to be established before the algorithm is introduced. The following two definitions describe the notion of *bi-connectivity*.

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

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*). If a graph is not bi-connected then it is either disconnected or there exists a vertex which removal partitions the graph into at least two connected components – this vertex is called an *articulation point*. Several examples of bi-connected graphs are shown in Figure 3.

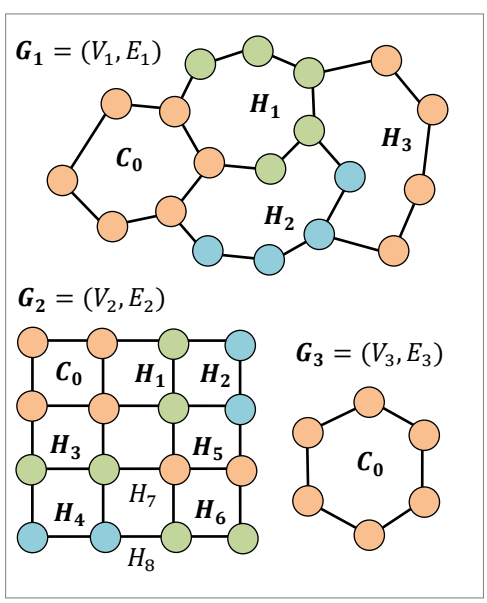


Figure 3. *Examples of* ***bi-connected graphs***. Three bi-connected graphs G_1, G_2, and G_3 and their handle decompositions are shown using colors (handles H_7 and H_8 of the decomposition of G_2 consist of a single edge).

Bi-connected graphs have an important property, which is exploited within the algorithm. Each bi-connected graph can be constructed from a **cycle** by an operation of ***adding a handle*** to the graph [24, 29, 30]. Consider a graph $G = (V, E)$; the new handle with respect to G is a sequence $L = [u, w_1, w_2, \ldots, w_l, v]$ where $l \in \mathbb{N}_0$,

$u, v \in V$ (called *connection vertices*) and $w_i \notin V$ for $i = 1,2, \ldots, l$ (w_i are new vertices). The result of the addition of the handle L to the graph G is a new graph $G' = (V', E')$ where $V' = V \cup \{w_1, w_2, \ldots, w_l\}$ and either $E' = E \cup \{\{u, v\}\}$ in the case of $l = 0$ of $E' = E \cup \{\{u, w_1\}, \{w_1, w_2\}, \ldots, \{w_{l-1}, w_l\}, \{w_l, v\}\}$ in the case of $l \geq 0$. Let the sequence of handles together with the initial cycle be called a ***handle decomposition*** of the given graph. See Figure 3 for illustrative examples.

Lemma 1 ***(handle decomposition)*** [24, 29, 30]**.** 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 effectively 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 any stage of the construction. This property is substantially exploited in the design of the ***BIBOX*** solving algorithm [16] for the pebble motion problem on a bi-connected graph.

The algorithm is presented below using a pseudo-code as Algorithm 1 (the algorithm is illustrated with pictures for easier understanding). The algorithm starts with the last handle of the handle decomposition and proceeds to the original cycle. Pebbles, which goal positions are within the last handle, are moved to their goal positions within this handle. The instance of the problem now reduces to the instance of the same type indeed on a smaller bi-connected graph. That is, the last handle is not considered any more since its pebbles do not need to move any more. This process is repeated until the original cycle of the decomposition remains.

Let $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ be an instance of the pebble motion problem. The following notation is used in the formalization of the algorithm. The handle decomposition of the graph G is formally a sequence $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$, where C_0 is the initial cycle and H_i is a handle for $i = 1,2, \ldots, d$. 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; while H_d is added as the last to the currently constructed graph). A handle $H_c = [u^c, w_1^c, w_2^c, \ldots, w_{h_i}^c, v^c]$ for $c \in \{1,2, \ldots, d\}$ can be assigned a cycle $C(H_c)$ if the input graph G is connected. 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 $w_1^c, w_2^c, \ldots, w_{h_i}^c$. Specially, it is defined that $C(C_0) = C_0$.

The following lemma is important for the design of the algorithm as well. Notice that the lemma states that individual vertices in the input pair of vertices are indifferent with respect to connecting by vertex disjoint paths.

Lemma 2 ***(two paths existence)*.** 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. ■

Proof. The idea of the proof is to proceed by inductively according to the size of the handle decomposition of the graph $G = (V, E)$. Let $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of the graph G. A function $c_{\mathcal{D}}: V \longrightarrow \mathbb{N}_0$ is defined as follows: $c_{\mathcal{D}}(v) = 0$ if $v \in C_0$ and $c_{\mathcal{D}}(v) = c$ if $v \in H_c$ for some $c \in \{1,2,\ldots,d\}$ (v is one of the internal vertices of the handle L_c). Observe, that $c_{\mathcal{D}}$ is a correctly defined function.

A given 4-tuple of vertices (u_1, u_2, v_1, v_2) is assigned a 4-tuple of integers defined using the function $c_{\mathcal{D}}$: $(c_{\mathcal{D}}(u_1), c_{\mathcal{D}}(u_2), c_{\mathcal{D}}(v_1), c_{\mathcal{D}}(v_2))$. The mathematical induction will proceed according to the **lexicographic** ordering of the 4-tupules **sorted in descending order** assigned using the function $c_{\mathcal{D}}$. Several cases must be distinguished.

Case (i): Let the 4-tuple of vertices (u_1, u_2, v_1, v_2) is assigned a 4-tuple of numbers (1,1,1,1), that is, all the vertices u_1, u_2, v_1, v_2 are located within the initial cycle C_0. Then the following juxtapositions of vertices u_1, u_2, v_1, and v_2 within C_0 with respect to the positive orientation of the cycle can occur: (u_1, v_1, v_2, u_2), (u_1, v_2, v_1, u_2), (v_1, u_1, v_2, u_2), (v_1, v_2, u_1, u_2), (v_2, u_1, v_1, u_2), and (v_2, v_1, u_1, u_2) (vertices are listed according to the positive orientation of the cycle; there is in total $4! = 24$ candidates for juxtapositions of 4 vertices; however, the remaining juxtapositions are isomorphic to the listed ones using a rotation along the cycle). In all the cases either the claim (a) or the claim (b) holds. See Figure 4 for detailed case analysis – for example in the juxtaposition (u_1, v_1, v_2, u_2), u_1 should be connected in positive orientation with v_1 and u_2 should be connected in negative orientation with v_2.

Case (ii): Let the 4-tuple of vertices (u_1, u_2, v_1, v_2) is assigned a sorted 4-tuple (C, c_2, c_3, c_4) where $C > c_2 \wedge C > c_3 \wedge C > c_4$. Using the interchangeability of vertices u_1, u_2, v_1, v_2, it is possible to suppose that $c_{\mathcal{D}}(u_1) = C$ without loss of generality. Let $H_C = [u^C, w_1^C, w_2^C, \ldots, w_{l_C}^C, v^C]$, then there exists a path π connecting u_1 and u_C consisting of the internal vertices of L_C. Since the sorted 4-tuple $(c_{\mathcal{D}}(u^C), c_{\mathcal{D}}(u_2), c_{\mathcal{D}}(v_1), c_{\mathcal{D}}(v_2))$ is lexicographically strictly less than (C, c_2, c_3, c_4), the induction hypothesis implies that the lemma holds for the 4-tuple of vertices u^C, u_2, v_1, v_2 and the graph G without the internal vertices of the handle H_C; let this smaller graph be denoted as G'. That is either (a) or (b) holds in G'. Without loss of generality, suppose that (a) holds. Then there exist vertex disjoint paths φ' and χ' connecting u^C with v_1 and u_2 with v_2 in G' respectively. The path π is vertex disjoint with χ' and it shares exactly one vertex u^C with φ. Let φ be a path formed by the concatenation of π with φ' (the vertex u^C is used only once) and let $\chi = \chi'$. Then φ and χ are vertex disjoint paths substantiating the claim (a) for 4-tuple of vertices u_1, u_2, v_1, v_2 in G. See Figure 4 for detailed illustration of the case.

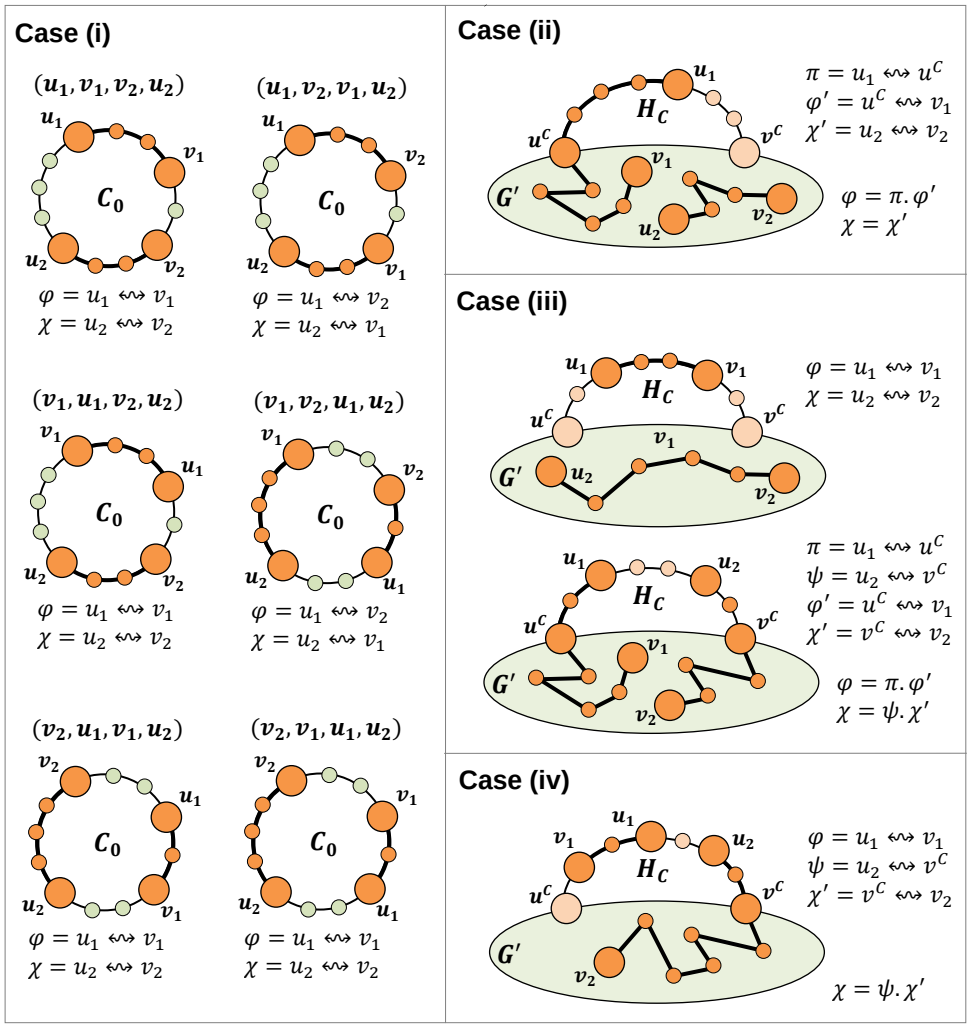


Figure 4. *An illustration of the existence of* ***two vertex disjoint paths*** *connecting two pairs of vertices in a bi-connected graph.* The figure illustrates the case analysis from the proof of Lemma 2 which states that there exist two vertex disjoint paths $\boldsymbol{\varphi}$ and $\boldsymbol{\chi}$ connecting a pair of vertices u_1 and u_2 with a pair of vertices v_1 and v_2 in a bi-connected graph G. The proof proceeds as mathematical induction according to the size of the handle decomposition of the graph G.

Case (iii): The next case is that the 4-tuple of vertices (u_1, u_2, v_1, v_2) is assigned a sorted 4-tuple (C, C, c_3, c_4) where $C > c_3 \wedge C > c_4$. Again using the interchangeability of vertices only some of all these cases are actually interesting. The first case is that $c_{\mathcal{D}}(u_1) = C$ and $c_{\mathcal{D}}(v_1) = C$ (that is, a pair of vertices to connect is within the handle H_C) and the second case is that $c_{\mathcal{D}}(u_1) = C$ and $c_{\mathcal{D}}(u_2) = C$ (that is, one vertex of a pair to connect is within the handle and the other is outside the internal vertices of the handle). In the first case, it is sufficient to construct a path φ connecting u_1 and v_1 consisting of the

internal vertices of H_C and a path χ connecting u_2 and v_2 in G' (G' is a connected graph). The constructed paths φ and χ are vertex disjoint and hence they substantiate the claim (a) of the lemma. In the second case, it is necessary to distinguish between two juxtapositions of u_1 and u_2 within H_C with respect to the positive orientation of the handle: (u_1, u_2) and (u_2, u_1). In the case of juxtaposition (u_1, u_2), a path π connecting u_1 and u^C and a path ψ connecting u_2 and v^C are constructed (with the exception of u^C and v^C only the internal vertices of L_C are used). The second juxtaposition just interchanges u_1 and u_2. The sorted 4-tuple $(c_{\mathcal{D}}(u^C), c_{\mathcal{D}}(v_C), c_{\mathcal{D}}(v_1), c_{\mathcal{D}}(v_2))$ is lexicographically strictly less than (C, C, c_3, c_4), hence the lemma holds for the 4-tuple of vertices u^C, v^C, v_1, v_2 in the graph G'. Without loss of generality suppose that the case (a) holds; that is, there exists a path φ' that connects u^C with v_1 in G' and a path χ' that connects v^C with v_2 in G' while φ' and χ' are vertex disjoint. Observe, that π and ψ are vertex disjoint as well. It is sufficient to set a path φ to be a concatenation of π and φ' and a path χ to be a concatenation of ψ and χ'. Then φ and χ are the paths substantiating the claim (a) of the lemma for the 4-tuple of vertices u_1, u_2, v_1, v_2 in G. Again, see Figure 4 for detailed illustration of the case.

Case (iv): Let the 4-tuple of vertices (u_1, u_2, v_1, v_2) is assigned a sorted 4-tuple (C, C, C, c_4) where $C > c_4$. Without loss of generality, suppose that $c_{\mathcal{D}}(v_2) = c_4$. Then the following interesting juxtapositions of vertices u_1, u_2, and v_1 within the handle L_C with respect to the positive orientation can occur: (v_1, u_1, u_2), (u_1, v_1, u_2), and (u_1, u_2, v_1) (interchangeability of u_1 and u_2 is used to rule out the second half of juxtapositions). All the cases can be treated in the same way, thus it is sufficient to show only one case – for example the case of (v_1, u_1, u_2). Let φ be a path connecting v_1 and u_1 consisting of the internal vertices of the handle H_C. Next, let ψ be a path connecting u_2 with v^C that uses internal vertices of the handle H_C and the vertex v^C. Let χ' be a path connecting v^C and v_2 in G' (such a path exists since G' is a connected graph). Observe, that φ is vertex disjoint with ψ as well as with χ'. Thus, if χ is set to be a concatenation of ψ and χ', then φ and χ substantiate the claim (a) of the lemma for the 4-tuple of vertices u_1, u_2, v_1, v_2 and the graph G. Again, see Figure 4 for illustration of the case.

Case (v): The last case occurs if a sorted 4-tuple (C, C, C, C) where $C > 1$ is assigned to the 4-tuple of vertices (u_1, u_2, v_1, v_2). This case reduces to the case with all the vertices of the input 4-tuple located within the original cycle of the handle decomposition. However, instead of the original cycle a cycle $C(H_C)$ should be used. ■

3.1.2. *Pseudo-code of the BIBOX Algorithm*

Several primitives are introduced to express the *BIBOX* algorithm in an easier way. Except functions S_P^0 and S_P^+ there is a function $S_P: P \longrightarrow V$ that represents the current arrangement of pebbles in the graph. Additionally, functions $\Phi_P^0: V \longrightarrow P \cup \{\perp\}$, $\Phi_P^+: V \longrightarrow P \cup \{\perp\}$, and $\Phi_P: V \longrightarrow P \cup \{\perp\}$ which are generalized inverses of S_P^0, S_P^+, and S_P respectively; the symbol $\perp$ is used to represent an unoccupied vertex (that is, $(\forall p \in P)\Phi_P(S_P(p)) = p$ and $\Phi_P(\perp) = \perp$ if $(\forall p \in P)S_P(p) \neq v$). Next, 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 the vertex following v (with respect to positive orientation) in the cycle C and $prev_{\circlearrowleft}(C, v)$ is the vertex preceding v (with respect to positive orientation). The orientation of a cycle given by $next_{\circlearrowleft}$ and $prev_{\circlearrowleft}$ is respected as well when vertices of the cycle are explicitly enumerated in the code. Auxiliary operations *Lock* (X) and *Unlock*(X) locks or unlocks a set of vertices $X \subseteq V$. Each vertex of the input graph is either locked or unlocked. The state of a vertex is used to determine whether a pebble can move into a vertex. Typically, a pebble is not allowed to enter a locked vertex (see the pseudo-code for details). Finally, there is assumed a potentially infinite sequence of functions $S_P^0, S_P^1, S_P^2, \ldots$ which finite prefix is used to form a solution. Actually, these variables are not needed to be stored in memory, the output solution can be directly printed to the output. For convenience, several variables such as those representing handle decomposition are global, that is, they are shared among all the functions and procedures in the pseudo-code.

It is assumed that for the number of pebbles it holds that $|V| = \mu - 2$, where $|P| = \mu$ (that is, there are exactly two unoccupied vertices in the graph G). Furthermore, it is required for the successful progression of the algorithm that the unoccupied vertices within the goal arrangement are located in the first two vertices of the original cycle (according to the positive orientation) of the handle decomposition. 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 first two vertices in the original cycle of the handle decomposition. Since the unoccupied vertices are indifferent, it does not matter what unoccupied vertex is associated with the first or with the second vertex of the initial cycle. Thus, preconditions of Lemma 2 are satisfied and hence the existence of mentioned two vertex disjoint paths is ensured.

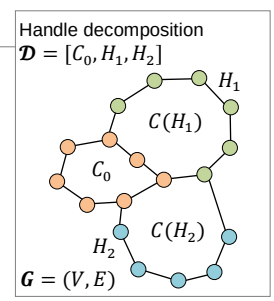
The goal arrangement is changed by the function *Transform-Goal* so that finally unoccupied vertices are located in the original cycle. This is done by shifting pebbles within the goal arrangement along the two determined paths. After the modified instance is solved, the function *Finish-Solution* moves unoccupied vertices back to their goal positions given by the original unmodified goal arrangement. This final placement of unoccupied vertices is done by shifting pebbles along the two paths determined by the function *Transform-Goal* in the opposite direction.

It is further supposed that the input 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.

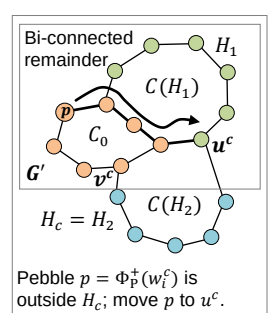
Several upper level primitives are exploited by the *BIBOX* algorithm. It is possible to make any vertex **unoccupied** in a connected graph (especially in a bi-connected graph). Making a given vertex unoccupied is implemented by a procedure *Make-Unoccupied*. Let v be a vertex to be made unoccupied.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

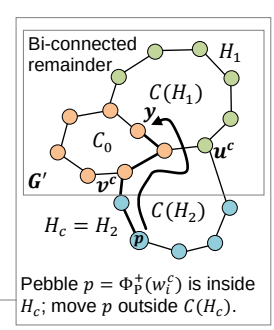
Algorithm 1. *The **BIBOX** algorithm.* The algorithm was originally proposed in [16]. It solves a given pebble motion problem on a non-trivial bi-connected graph with exactly **two unoccupied** vertices. The algorithm proceeds inductively according to the handle decomposition of the graph of the input instance. The two unoccupied vertices are necessary for arranging pebbles within the original cycle of the handle decomposition.

function *BIBOX-Solve*$(G = (V, E), P, S_P^0, S_P^+)$: **pair**
/* Top level function of the BIBOX algorithm; solves a given problem of pebble motion on a graph.
Parameters: G - a graph modeling the environment,
P - a set of pebbles,
S_P^0 - a initial arrangement of pebbles,
S_P^+ - a goal arrangement of pebbles. */
1: **let** $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of G
2: $(S_P^+, \varphi, \chi) \leftarrow$ Transform-Goal(G, P, S_P^+)
3: $S_P \leftarrow S_P^0$
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** $(\xi, [S_P^0, S_P^1, \ldots, S_P^\xi])$

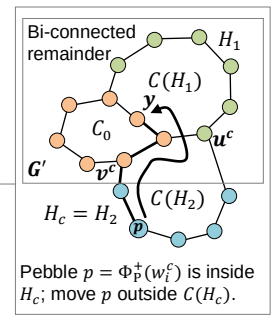


procedure *Solve-Regular-Handle*(c)
/* Places pebbles which destinations are within a handle H_c; pebbles placed in the handle H_c are finally locked so they cannot move any more.
Parameters: c – the index of a handle */
1: **let** $[u^j, w_1^j, w_2^j, \ldots, w_{h_c}^j, v^j] = H_j \; \forall j \in \{1,2,\ldots,d\}$
/* 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$(H_c \setminus \{u^c, v^c\})$
/* A pebble to be placed is outside the handle H_c. */
9: **if** $S_P(\Phi_P^+(w_i^c)) \notin (H_c \setminus \{u^c, v^c\})$ **then**
10: Move-Pebble$(\Phi_P^+(w_i^c), u^c)$
11: Lock$(\{u^c\})$
12: Make-Unoccupied(v^c)
13: Unlock(H_c)
14: Rotate-Cycle$^+(C(H_c))$
/* A pebble to be placed is inside the handle H_c. */
15: **else**
16: Make-Unoccupied(u^c)

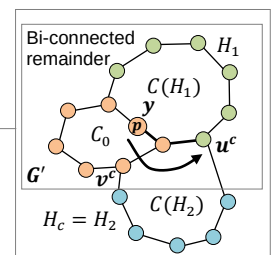




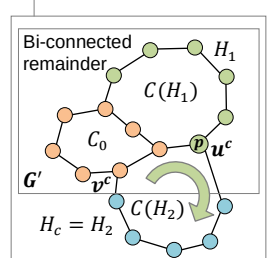
```
17:     Unlock(H_c)
18:     ρ ← 0
19:     while S_P(Φ_P^+(w_i^c)) ≠ v^c do
20:         Rotate-Cycle^+(C(H_c))
21:         ρ ← ρ + 1
22:     Lock(H_c \ {u^c, v^c})
23:     let y ∈ V \ (⋃_{j=c+1}^{d} (H_j \ {u^j, v^j}) ∪ C(H_c))
24:     Move-Pebble(Φ_P^+(w_i^c), y)
25:     Lock ({y})
26:     Make-Unoccupied(u^c)
27:     Unlock(H_c)
28:     while ρ > 0 do
29:         Rotate-Cycle^-(C(H_c))
30:         ρ ← ρ − 1
31:     Unlock({y})
32:     Lock(H_c \ {u^c, v^c})
33:     Move-Pebble(Φ_P^+(w_i^c), u^c)
34:     Lock ({u^c})
35:     Make-Unoccupied(v^c)
36:     Unlock(H_c)
37:     Rotate-Cycle^+(C(H_c))
38: Lock(H_c \ {u^c, v^c})
```



Pebble $p = \Phi_P^+(w_i^c)$ is inside H_c; move p outside $C(H_c)$.



Pebble $p = \Phi_P^+(w_i^c)$ outside $C(H_c)$; $C(H_c)$ rotated back; move p into connection u^c.

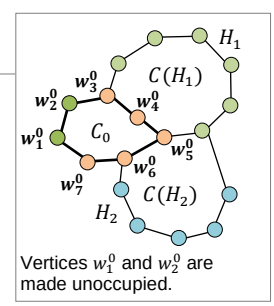


Pebble $p = \Phi_P^+(w_i^c)$ in u^c; rotate $C(H_c)$ once forward.

procedure *Solve-Original-Cycle*

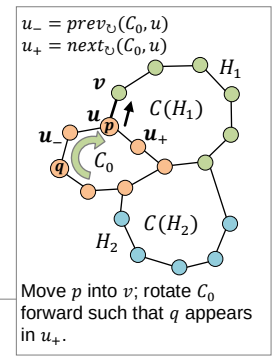
/* Places pebbles which destinations are within the original cycle; it is assumed that unoccupied vertices of the goal arrangement of pebbles are located within the original cycle. */

```
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_P^+(w_1^0) =⊥ and Φ_P^+(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_P^+(w_i^0) ≠ Φ_P(w_i^0) then
9:          Exchange-Pebbles (Φ_P^+(w_i^0), Φ_P(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})
```

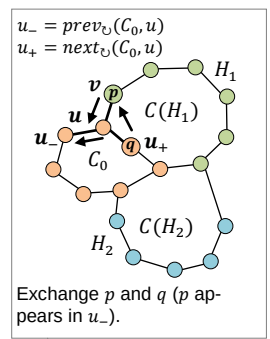


Vertices w_1^0 and w_2^0 are made unoccupied.

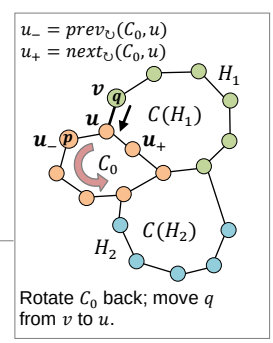
procedure *Exchange-Pebbles*(p, q, u, v)
/* Exchanges a pair of pebbles within the initial cycle of the handle decomposition.
Parameters: p, q - a pair of pebbles to be exchanged, u, v - a pair of neighboring vertices where v is used as a storage space. */
1: $r \leftarrow \Phi_P(v)$
2: Make-Unoccupied(u)
3: Swap-Pebbles-Unoccupied(v, u)
4: **while** $S_P(p) \neq u$ **do**
5: ⌊ Rotate-Cycle$^+(C_0)$
6: Swap-Pebbles-Unoccupied(u, v)
7: Lock $(\{u\})$
8: Make-Unoccupied$(next_\circlearrowleft(C_0, u))$
9: Unlock $(\{u\})$
/* Subsequent rotation must use u as the unoccupied vertex. */
10: Lock$(C_0 \setminus \{u\})$
11: $\rho \leftarrow 0$
12: **while** $S_P(q) \neq next_\circlearrowleft(C_0, u)$ **do**
13: ⌊ Rotate-Cycle$^+(C_0)$
14: ⌊ $\rho \leftarrow \rho + 1$
15: Swap-Pebbles-Unoccupied(v, u)
16: Unlock $(C_0 \setminus \{u\})$
17: Make-Unoccupied$(prev_\circlearrowleft(C_0, u))$
18: Swap-Pebbles-Unoccupied$(u, prev_\circlearrowleft(C_0, u))$
19: Swap-Pebbles-Unoccupied$(next_\circlearrowleft(C_0, u), u)$
20: Swap-Pebbles-Unoccupied(u, v)
21: Unlock$(\{u\})$
22: Lock$(C_0 \setminus \{u\})$
23: **while** $\rho > 0$ **do**
24: ⌊ Rotate-Cycle$^-(C_0)$
25: ⌊ $\rho \leftarrow \rho - 1$
26: Swap-Pebbles-Unoccupied(v, u)
27: **while** $S_P(r) \neq u$ **do**
28: ⌊ Rotate-Cycle$^+(C_0)$
29: Swap-Pebbles-Unoccupied(u, v)
30: Unlock(C_0)



Move p into v; rotate C_0 forward such that q appears in u_+.

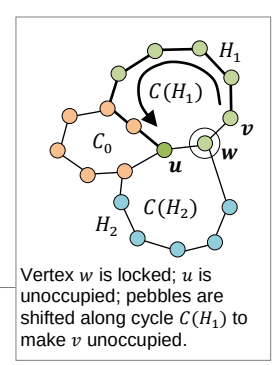


Exchange p and q (p appears in u_-).



Rotate C_0 back; move q from v to u.

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 $\Phi_P(u) = \perp$ 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: ⌊ Swap-Pebbles-Unoccupied(w_{i+1}, w_i)

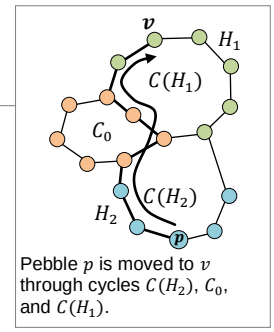


Vertex w is locked; u is unoccupied; pebbles are shifted along cycle $C(H_1)$ to make v unoccupied.

procedure *Move-Pebble*(p, v)

/* Moves a pebble p into a vertex v
avoiding locked vertices.
Parameters: p - a pebble to move,
v - a target vertex.*/
/* complexity issues impose special selection of φ */

1: **let** $\varphi = [S_P(p) = w_1^\varphi, w_2^\varphi, \dots, w_{j_\varphi}^\varphi = v]$ be a path
2: connecting $S_P(p)$ and v in G not containing
3: locked vertices such that an alternative vertex
4: disjoint path $\chi = [S_P(p) = w_1^\chi, w_2^\chi, \dots, w_{j_\chi}^\chi = v]$
5: not containing locked vertices exists
6: **for** $i = 1,2, \dots, j_\varphi - 1$ **do**
7: Lock $(\{w_i^\varphi\})$
8: Make-Unoccupied(w_{i+1}^φ)
9: Unlock$(\{w_i^\varphi\})$
10: Swap-Pebbles-Unoccupied$(w_i^\varphi, w_{i+1}^\varphi)$



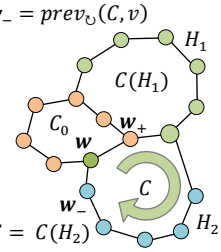
Pebble p is moved to v through cycles $C(H_2)$, C_0, and $C(H_1)$.

procedure *Rotate-Cycle*$^+(C)$

/* Rotates pebbles in a cycle C in the positive direction;
the vertex locking mechanism allows to select which one of
unoccupied vertices should be used. At least one unlocked
unoccupied vertex must be located in C.
Parameters: C - a cycle to rotate. */

1: **let** $w \in C$ such that $\Phi_P(w) = \perp$ and w is not locked
2: **for** $i = 1,2, \dots, |C|$ **do**
3: Swap-Pebbles-Unoccupied$(prev_\circlearrowleft(C, w), w)$
4: $w \leftarrow prev_\circlearrowleft(C, w)$

$w_+ = next_\circlearrowleft(C, v)$
$w_- = prev_\circlearrowleft(C, v)$



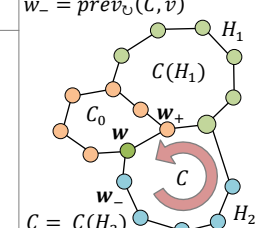
Vertex w is unoccupied; $C(H_2)$ is rotated in the positive direction.

procedure *Rotate-Cycle*$^-(C)$

/* Rotates pebbles in the cycle C in the negative direction;
again an unoccupied vertex to use can be selected by the
vertex locking mechanism. At least one unlocked
unoccupied vertex must be located in C.
Parameters: C - a cycle to rotate. */

1: **let** $w \in C$ such that $\Phi_P(w) = \perp$ and w is not locked
2: **for** $i = 1,2, \dots, |C|$ **do**
3: Swap-Pebbles-Unoccupied$(next_\circlearrowleft(C, w), w)$
4: $w \leftarrow next_\circlearrowleft(C, w)$

$w_+ = next_\circlearrowleft(C, v)$
$w_- = prev_\circlearrowleft(C, v)$



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

procedure *Swap-Pebbles-Unoccupied*(u, v)

/* Swaps pebbles in vertices u and v;
vertex v is supposed to be unoccupied.
Parameters: u, v – vertices in which pebbles
are swapped. */

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: $\xi \leftarrow \xi + 1$

An operation of **swapping** pebbles itself is implemented using a procedure *Swap-Pebbles-Unoccupied*. The procedure moves a pebble into a neighboring unoccupied vertex and the next member S_P^{ξ} of the output solution sequence is constructed together with the update of functions S_P and Φ_P according to the new arrangement of pebbles.

The next important process is **moving** a pebble into a given target vertex. This process is implemented by a procedure *Move-Pebble*. Let a pebble p is moved to a vertex v. A path φ is found such that is connects vertices $S_P(p)$ (which is a vertex currently occupied by p) and v and there exists an alternative vertex disjoint path χ connecting the same pair of vertices. The existence of the alternative is ensured by Lemma 2. Indeed, the proof of Lemma 2 provides the construction such a path. Parameters of Lemma 2 should be set as follows: u_1 is $S_P(p)$, u_2 is a neighboring vertex to u_1 in the same bi-connected component, v_1 is v, and v_2 is a neighboring vertex to v_1 in the same bi-connected component (it can be determined which of the neighboring vertices belong into the same bi-connected component from the knowledge of the handle decomposition). Vertex disjoint paths φ' and χ' resulting from Lemma 2 together with edges $\{u_1, u_2\}$ and $\{v_1, v_2\}$ form the required φ and χ. In case (a): $\varphi = \varphi'$ and $\chi = [u_1].\chi.[u_2]$; in case (b): $\varphi = [u_1].\chi'$ and $\chi = \varphi'.[u_2]$.

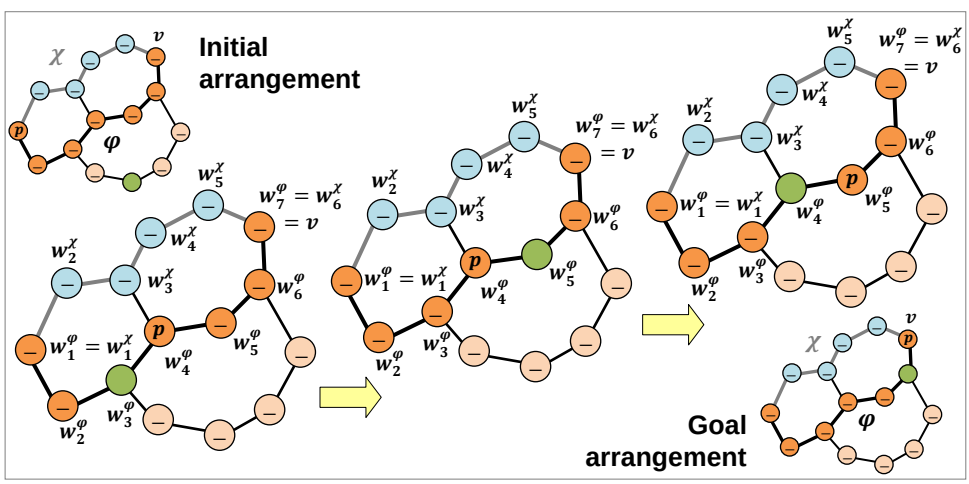


Figure 5. *An illustration of **moving** a pebble in a bi-connected graph.* The task is to move a pebble p from an initial position to a vertex v. Due to bi-connectivity of the graph there are two vertex disjoint paths φ and χ connecting the initial position with v. The path φ is traversed by the pebble p while the alternative path χ is used to make unoccupied a vertex in front of the vertex occupied by p (the vertex with p is locked). The symbol _ stands for an anonymous pebble.

Subsequently, edges of φ are traversed in the following way. The first vertex of the edge is locked so paths to be searched must avoid this vertex. An invariant holds, that p is located in the first vertex of the edge at the beginning of each traversal step and thus it cannot move. Then the second vertex of the edge is made unoccupied (the alternative path χ is used for this task); the first vertex of the edge is unlocked and the pebble p is

moved to the second vertex of the edge which is now unoccupied (see Figure 5 for detailed illustration).

The last basic operation exploited by the algorithm is a **rotation** of pebbles along a cycle. This operation is implemented by procedures *Rotate-Cycle*$^{+}$ and *Rotate-Cycle*$^{-}$. The former rotates pebbles in the positive direction and the latter rotates pebbles in the negative direction. It supposed the at least one vertex in the given input cycle is unoccupied. The rotation is done using an unlocked unoccupied vertex located in the input cycle (see Figure 6 for detailed illustration).

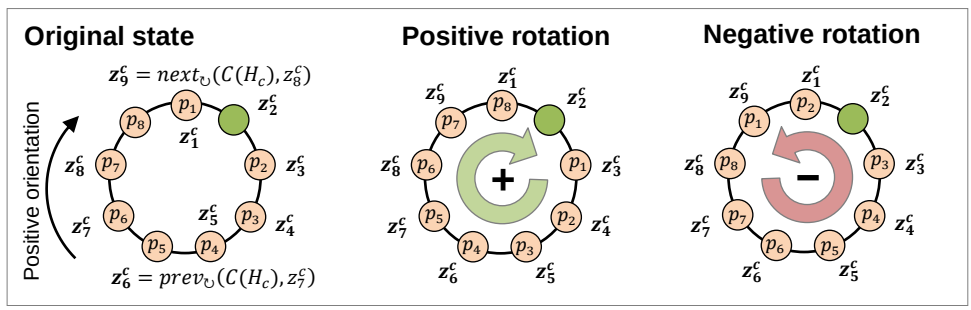


Figure 6. *An illustration of* ***rotation*** *of pebbles* ***along a cycle***. An orientation of the cycle is determined by functions $next_\circlearrowleft$ and $prev_\circlearrowleft$. There is a single unoccupied vertex in the cycle. The positive and negative rotations are shown.

During movement of an unoccupied vertex and during movement of a pebble to another vertex, arrangement of pebbles located in vertices that are not locked is generally not preserved. This behavior helps to control finished parts of the goal arrangement. On the other hand, moving pebbles must be done in a precise way so that required unlocked paths always exist.

The process of placing pebbles according to the given goal arrangement will be described now using the primitives discussed above. Pebbles, 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 $H_c = [u^c, w_1^c, w_2^c, \dots, w_{h_c}^c, v^c]$ for $c \in \{1,2,\dots,d\}$ be a current handle. Suppose that a pebble which goal position is in w_i^c for $i \in \{1,2,\dots,h_c\}$, that is a pebble $\Phi_P^+(w_i^c)$, is processed in the current iteration. Inductively suppose that pebbles $\Phi_P^+(w_{h_c}^c)$, $\Phi_P^+(w_{h_c-1}^c)$,…, $\Phi_P^+(w_{i+1}^c)$ 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 pebble $\Phi_P^+(w_{i+1}^c)$ must be produced at the end of the iteration.

The pebble $\Phi_P^+(w_i^c)$ is moved to the vertex u^c and then the cycle $C(H_c)$ is positively rotated one which causes that the pebble $\Phi_P^+(w_i^c)$ moves to w_1^c and pebbles $\Phi_P^+(w_{h_c}^c)$, $\Phi_P^+(w_{h_c-1}^c)$,…, $\Phi_P^+(w_{i+1}^c)$ plunge in the cycle so that they are located in $w_{h_c-i}^c$, $w_{h_c-i-1}^c$,…, w_2^c. The described process represents one iteration of stacking pebbles into the handle H_c. However, the process is not that easy. At least, two major cases must be

distinguished within this process. 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 pebble $\Phi_P^+(w_i^c)$ 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. It 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 pebbles already stacked in the handle. The pebble $\Phi_P^+(w_{h_c}^c)$ is fixed in u^c by locking u^c and then an unoccupied vertex is moved to v^c which makes the rotation of the cycle $C(H_c)$ possible. The positive rotation of $C(H_c)$ finishes the iteration.

If the pebble $\Phi_P^+(w_i^c)$ 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 getting the pebble $\Phi_P^+(w_{h_c}^c)$ outside the handle. Notice, that it is not possible for the pebble $\Phi_P^+(w_i^c)$ to intermix with already stacked pebbles $\Phi_P^+(w_{h_c}^c)$, $\Phi_P^+(w_{h_c-1}^c)$,…, $\Phi_P^+(w_{i+1}^c)$. The vertex u^c is made unoccupied and the cycle $C(H_c)$ is positively rotated until the pebble $\Phi_P^+(w_i^c)$ gets outside the internal nodes of H_c; that is, $\Phi_P^+(w_i^c)$ appears in v^c. This series of rotations preserves the order of the already stacked pebbles. To restore the situation however, the cycle must be rotated back the same number of times. A vertex w outside the already finished part of the graph (that is outside $C(H_c)$ and outside H_j for $j > c$) is selected; the pebble $\Phi_P^+(w_i^c)$ is move into w and it is fixed there by locking. The vertex u^c is made unoccupied again since the preceding process may move some pebble 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 $\Phi_P^+(w_{h_c}^c)$, $\Phi_P^+(w_{h_c-1}^c)$,…, $\Phi_P^+(w_{i+1}^c)$ is restored. The situation is now the same as in the previous case with $\Phi_P^+(w_i^c)$ outside the handle.

After the last iteration within the handle H_c it holds that the pebbles $\Phi_P^+(w_{h_c}^c)$, $\Phi_P^+(w_{h_c-1}^c)$,…, $\Phi_P^+(w_1^c)$ 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 pebbles into handles. See Figure 7 for detailed illustration.

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 pebbles 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 pebbles. More precisely, if a pebble residing in a vertex of C_0 differs from a pebble that should reside in this vertex in the goal arrangement, this pair of pebbles is exchanged. The process is implemented by a procedure *Solve-Original-Cycle* and by auxiliary procedure *Exchange-Pebbles* for exchanging a pair of pebbles.

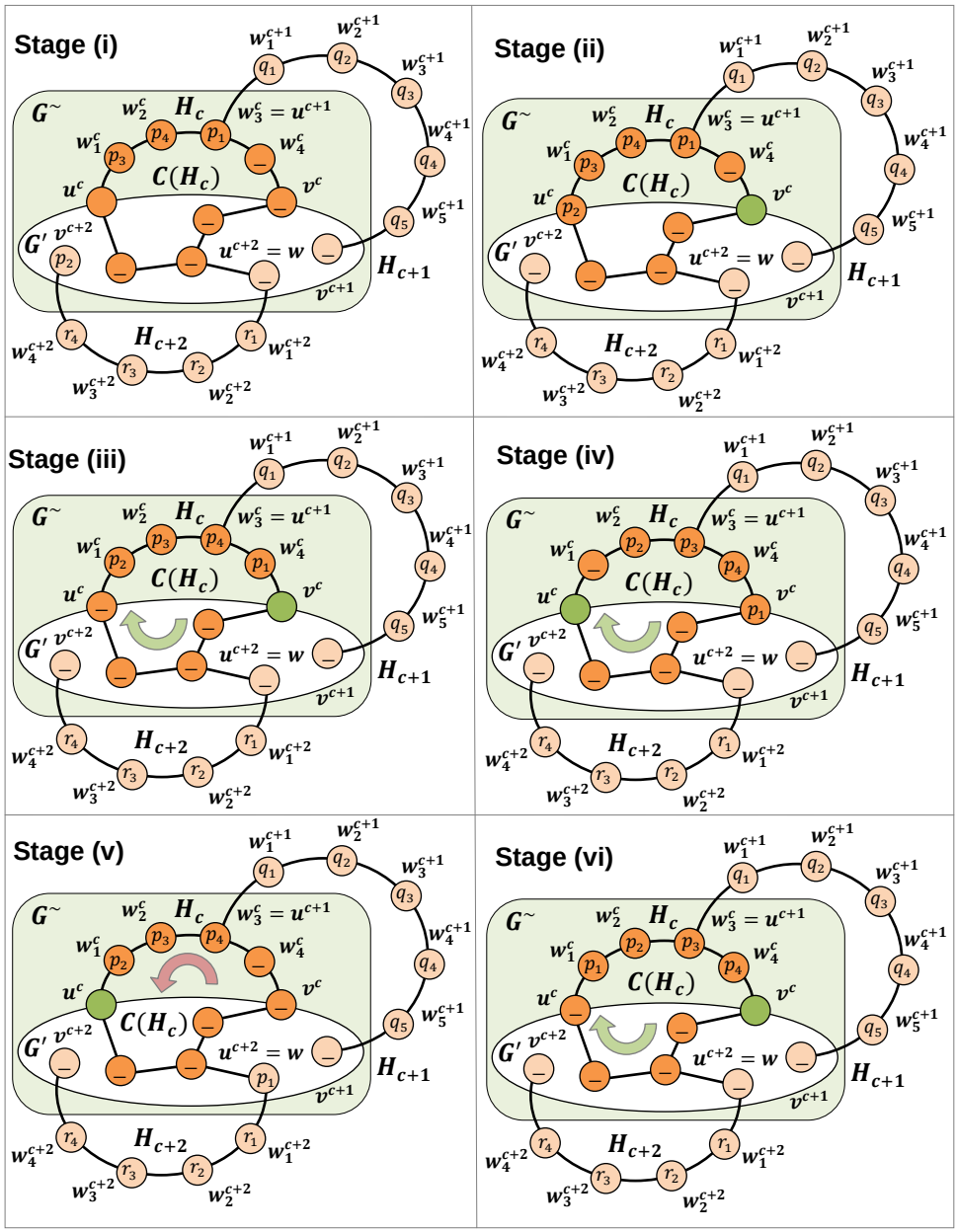


Figure 7. *An illustration of* ***stacking a pebble*** *into a handle.* The progress of stacking pebbles into the handle H_c is shown. Pebbles p_1, p_2, p_3, and p_4 are to be stacked into H_c (that is, $S_P^+(p_1) = w_1^c$, $S_P^+(p_2) = w_2^c$, $S_P^+(p_3) = w_3^c$, and $S_P^+(p_4) = w_4^c$) while handles H_{c+1} and H_{c+2} are already solved (that is, $S_P^+(q_1) = w_1^{c+1}$,…, $S_P^+(q_5) = w_5^{c+1}$, and $S_P^+(r_1) = w_1^{c+2}$,…, $S_P^+(q_4) = w_4^{c+1}$). Observe that the pebble p_2 is originally outside the handle while the pebble p_1 is inside the handle (that is, must be rotated outside the handle – stages (iv) and (v)). The symbol _ stands for an anonymous pebble.

The procedure *Exchange-Pebbles* 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 pebbles p and q itself exploits a pair of vertices u and v where these two vertices are connected by an edge and $u \in C_0 \wedge v \notin C_0$.

The vertex v is used as a storage place. The need of two unoccupied vertices is imposed by the fact that a pebble 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 o pebbles p and q it is necessary to preserve ordering of the other vertices. First, a pebble occupying the vertex v is moved into the cycle C_0 in order to make v vacant (lines **1-3** of *Exchange-Pebbles*). Then the cycle is rotated until the pebble p appears in u (since there was a pebble in u at the beginning of the rotation, there is always some pebble in u after all the rotations) and the pebble p is stored in v (lines **4-6** of *Exchange-Pebbles*). Next, the cycle C_0 is rotated positively so that q 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 **7-14** of *Exchange-Pebbles*). However, the second unoccupied vertex must not interfere with counting of rotations. Thus it is located $next_{\circlearrowleft}(C_0, u)$ at the beginning (that is, outside the sequence of pebbles between p and q which length is being counted in fact) and then moved to $prev_{\circlearrowleft}(C_0, u)$ in the positive direction (the movement of the second unoccupied in the negative direction is not possible here, since u is now locked). At this moment, pebbles p and q are exchanged using two unoccupied vertices so that ordering of p in the cycle C_0 is the same as of q before the exchange (lines **15-20** of *Exchange-Pebbles*). Then, the cycle is rotated in the negative direction recorded number of times so that place within the cycle where p was originally ordered appears in u; thus q is ordered here (lines **21-26** of *Exchange-Pebbles*). Finally, the pebble was located in v before the exchange of pebbles p and q has been commenced is put back into v (lines **27-30** of *Exchange-Pebbles*). Since the process of exchange of a pair of pebbles is quite subtle, the detailed case analysis is given within the proof of soundness of the process (see Lemma 4).

3.1.3. *Theoretical Analysis of the BIBOX Algorithm*

This section is devoted to theoretical analysis of the *BIBOX* algorithm. Particularly, the soundness of the algorithm and its complexity are analyzed.

Lemma 3 ***(soundness of Move-Pebble).*** If an original location of a pebble p, 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-Pebble* correctly moves the pebble p from its original location to v. ■

Proof. Recall how the procedure *Move-Pebble* works. First, a shortest path $\varphi = [w_1^\varphi, w_2^\varphi, \ldots, w_{j_\varphi}^\varphi]$ connecting $\Phi_P(p)$ and v is found. This path is then traversed while the pebble p is moved along its edges. The whole path φ belongs to the same bi-connected component as $\Phi_P(p)$ and v. Otherwise, there would not be the alternative vertex disjoint path χ connecting the same pair of vertices.

The proof of soundness will proceed as mathematical induction according to the number of edges of φ already traversed. In all the steps, the pebble p and the unoccupied vertex should be located in the bi-connected component containing φ. Initially, this condition holds. Consider that a pebble p 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 G. It is supposed that w_i^φ, w_{i+1}^φ, and the unoccupied vertex are all in the same bi-connected component. Thus an alternative path connecting w_{i+1}^φ and the unoccupied vertex in this bi-connected component avoiding w_i^φ must exist (such a path can be constructed by concatenating parts of φ and χ). 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 p 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 5). ■

Lemma 4 ***(soundness of Exchange-Pebbles).*** The procedure *Exchange-Pebbles* of the Algorithm 1 for exchanging a pair of pebbles p and q within a cycle C_0 is **sound**. That is, if the arrangement of pebbles within the cycle C_0 is regarded as a permutation, then the output arrangement produced by the procedure *Exchange-Pebbles* corresponds to a permutation where pebbles p and q are transposed with respect to the permutation corresponding to the input arrangement. ■

Proof. To prove the statement of the lemma some analysis of the course of the procedure must be done. Fortunately, almost all the steps of the procedure suppose preconditions that are trivial to check. However, it is not that trivial to check whether forward and backward rotations of the cycle interleaved with exchange of pebbles p and q and interfering with unoccupied vertices really produces the desired transposition. More precisely, it is necessary to check whether the orderings of pebbles between p and q and between q and p (with respect to the positive orientation of the cycle) remain unchanged while p and q are transposed. This is done using detailed case analysis of what can happen. Let $C_0 = [w_1^0, w_2^0, \ldots, w_l^0]$, then there are $l-2$ pebbles located in C_0 at the moment before the cycle is rotated positively (situation at line **11** of *Exchange-Pebbles* - see stage (i) in Figure 8). The pebble p is already stored in v and the two unoccupied vertices are u and $next_\circlearrowleft(C_0, u)$. Let pebbles occupying vertices of the cycle in the interval between $\Phi_P(q)$ and u with respect to the positive orientation (excluding boundaries) are denoted $q_1, q_2, \ldots, q_k$ respectively; let pebbles occupying vertices of the cycle in the interval between $next_\circlearrowleft(C_0, u)$ and $\Phi_P(q)$ with respect to the positive orientation (again excluding boundaries) are denoted as $p_1, p_2, \ldots, p_{l-k-3}$. The series of ρ positive rotation of C_0 fol-

lows to move the pebble q into $next_{\circlearrowleft}(C_0, u)$ (see stage (ii) in Figure 8). Now, all the pebbles $q_1, q_2, \ldots, q_k$, $p_1, p_2, \ldots, p_{l-k-3}$, and q 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 8).

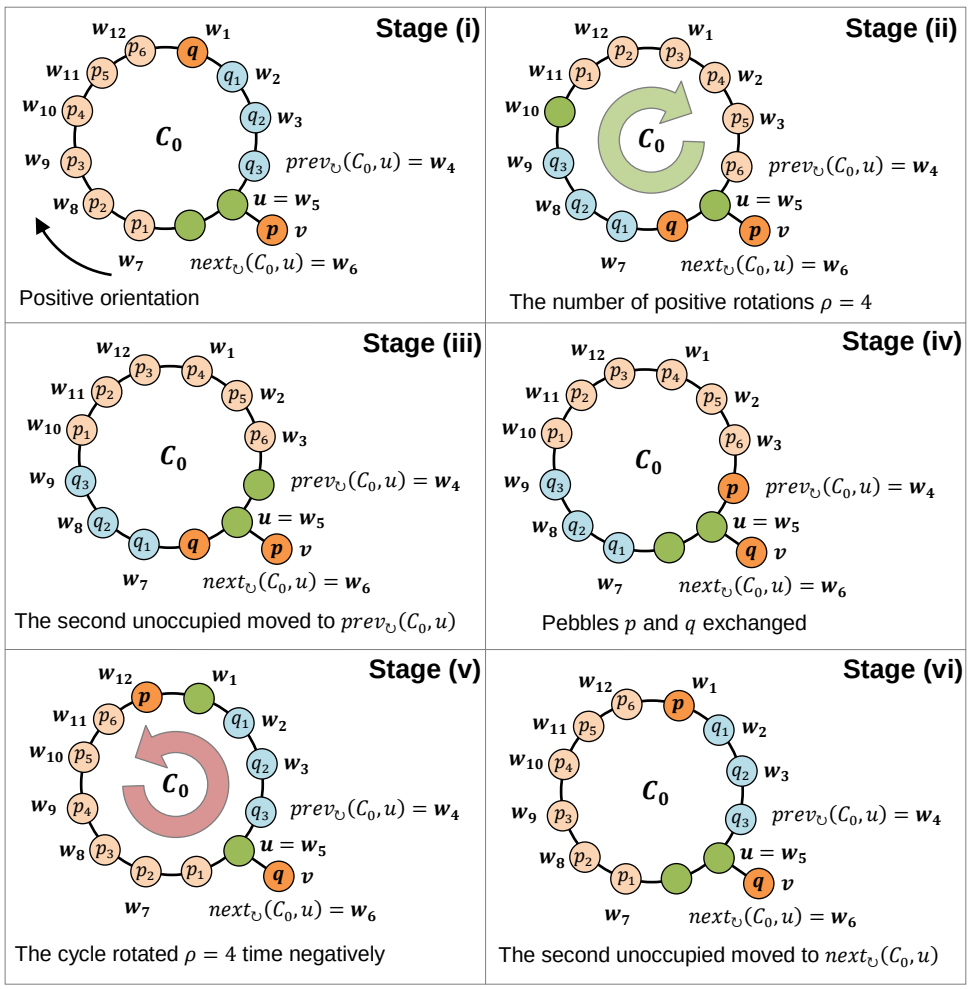


Figure 8. *The **progression of the exchange** of a pair of pebbles within an initial cycle of the handle decomposition.* Pebbles p and q in a cycle consisting of 12 vertices are exchanged while the ordering of other pebbles within the cycle is preserved. The figure illustrates the progression of the procedure *Exchange-Pebbles* from line **7** to **25**.

Next, pebbles are exchanged: that is, q is moved to v and p is moved to $prev_{\circlearrowleft}(C_0, u)$ (see stage (iv) in Figure 8). At this step, pebbles $q_1, q_2, \ldots, q_k$ are ρ steps forward with

respect to their location before the series of rotations; pebble $p_1, p_2,\ldots, p_{l-k-3}$ 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 pebbles $p_1, p_2,\ldots, p_{l-k-3}$ but not through pebbles $q_1, q_2,\ldots, q_k$). Finally, the pebble p is $\rho - 1$ steps forward with respect to the location of q before the series of rotations.

The series of ρ rotation in the negative direction places pebbles $q_1, q_2,\ldots, q_k$ to their original positions; pebbles $p_1, p_2,\ldots, p_{l-k-3}$ are placed 1 step backward with respect to their original position before rotations, and p is one step backward with respect to the original position of q before the series of rotations (see stage (v) in Figure 8). This inconsistency however, is caused by a different location of the second unoccupied vertex which now between p and q_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 p and q 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 pebbles $p_1, p_2,\ldots, p_{l-k-3}$ to their original positions before rotations and the pebble p to the original position of q (see stage (vi) in Figure 8). As this is a step used only for purposes of the proof, the algorithm actually does not perform it. ■

Proposition 2 ***(BIBOX - soundness and completeness).*** The *BIBOX* algorithm is **sound** and **complete**. That is, the algorithm always terminates and produces a solution of a given input instance of the problem of pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. ■

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**. It also concerns search for a path connecting a given pair of vertices avoiding the locked ones. The non-existence of such a path could make the following operation in the course of the execution undefined. The existence of vertex disjoint paths φ and χ is 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**), *Exchange-Pebbles* (lines **2**, **8**, and **18**). Next, a pair of vertex disjoint paths containing unlocked vertices is also constructed within the procedure *Move-Pebble* (lines **1-5**) which is called by *Solve-Regular-Handle* (lines **10**, **24**, and **33**). All these cases must be examined.

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 outside the currently solved handle H_c. 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 be denoted as G' (see Figure 7). Since $G^{\sim}$ is completely unlocked and an unoccupied vertex cannot be

located in any internal vertex of the solved handles, an unlocked path connecting w and an unoccupied vertex must exist. The construction of the second path within the call at line **5** of *Solve-Regular-Handle* must take into account that w is locked. As the subgraph $G^{\sim}$ is bi-connected, there exists a path connecting any two vertices in a subgraph $G^{\sim}$ with w removed since it must be connected.

At **line 12** of *Solve-Regular-Handle* a connection vertex v^c of the currently solved handle H_c is made unoccupied while internal vertices of H_c and the second connection vertex u^c are locked. Again, an unoccupied vertex is supposed to be located in the not yet solved part of the graph (calls at lines **3** and **5** of *Solve-Regular-Handle* ensures this), which is bi-connected, and without u^c it is still connected.

A 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 the parameter. The internal vertices of the already solved handles and the internal vertices of H_c are locked. An unoccupied vertex is located in $G^{\sim}$ at this moment. Since $G^{\sim}$ is bi-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 w, which is in $G^{\sim}$ and outside the cycle associated with the current handle H_c, is locked. Internal vertices of H_c are locked as well. Again, the unlocked part of the graph corresponds to a bi-connected subgraph $G^{\sim}$ from which one vertex was removed. Thus, the unlocked part of the graph constitutes a connected component. An unoccupied vertex is located in some of the unlocked vertices. 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 internal vertices of the already solved handles, the internal vertices of H_c, and the second connection vertex u^c are locked. Thus, unlocked vertices constitute a connected subgraph (because it is obtained by removing u^c form a bi-connected subgraph G'). Since the unoccupied vertex is unlocked, there exists an unlocked path connecting the unoccupied vertex and v^c.

The soundness of the procedure *Solve-Original-Cycle* is partially implied by the soundness of the procedure *Exchange-Pebbles* which is treated by Lemma 4. 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. 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 made unoccupied. An unlocked path in the cycle from any of its vertices to w_1^0 exists. 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 the same as that at lines **4** and **6** respectively.

The soundness of the procedure *Move-Pebble* is treated separately by Lemma 3. However, preconditions of the Lemma 3 need to be checked – that is, whether all the calls

of *Move-Pebble* moves a pebble within the same 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-Pebble* at **line 10** of *Solve-Regular-Handle* is that already solved handles are locked and the internal vertices of the currently solved handle H_c are locked too. Both unoccupied vertices are located in the not yet solved part of the graph and outside the internal part of H_c (this condition is enforced by moving unoccupied vertices at lines **2-6** of *Solve-Regular-Handle*). The task is to move a pebble $\Phi_P^+(w_i^c)$, which is known to be outside H_c as well as outside the internal vertices of the already solved handles, to the connection vertex u^c of the handle H_c. The whole not yet solved part of the graph (that is, the bi-connected subgraph $G^\sim$) without handle H_c constitutes a bi-connected component which all the vertices are unlocked. The unoccupied vertex and both the pebble $\Phi_P^+(w_i^c)$ and u^c are located in this bi-connected component and thus preconditions of Lemma 3 are satisfied.

The call of *Move-Pebble* at **line 24** of *Solve-Regular-Handle* moves a pebble $\Phi_P^+(w_i^c)$ to a vertex w. The pebble $\Phi_P^+(w_i^c)$ is known to be located in a connection vertex v^c of the current handle H_c. The vertex w is located in the not yet solved part of the graph and outside the cycle associated with the handle H_c. Solved handles and internal vertices of H_c are locked; one of the unoccupied vertices is the second connection vertex u^c of H_c. Thus, the unlocked vertices constitutes a bi-connected component (the component is exactly the subgraph $G^\sim$) where the pebble $\Phi_P^+(w_i^c)$, vertex w, and the unoccupied vertex are located. Again, preconditions of Lemma 3 are satisfied.

Finally, the task of the call of *Move-Pebble* at **line 33** of *Solve-Regular-Handle* is to move a pebble $\Phi_P^+(w_i^c)$ to a connection vertex u^c of the current handle H_c. It is known that the pebble $\Phi_P^+(w_i^c)$ is located in w as in the previous case. Internal vertices of all the solved handles and of H_c are locked at this moment. Hence, the pebble $\Phi_P^+(w_i^c)$, the vertex u^c, and the unoccupied vertex are all located in the same bi-connected component consisting of unlocked vertices (the subgraph $G^\sim$ is that component). A connection vertex u^c is known to be unoccupied. Thus, preconditions of Lemma 3 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 pebbles strictly increases as the algorithm progresses, the algorithm always terminates and produces a solution to the input instance. ■

The following propositions characterize the *BIBOX* algorithm with respect to its computational resource requirements. All the aspects of the algorithms are polynomial.

Proposition 3 ***(BIBOX – worst case time complexity).*** The worst case **time complexity** of the *BIBOX* algorithm is $\mathcal{O}(|V|^3)$ with respect to an input instance of the problem pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. ■

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 pebbles (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 (recall that this is done in order to keep unoccupied vertices outside the already finished part of the graph).

There are at most $|V|$ pebbles (since $|P| < |V|$) to be placed within handles of a handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$. Let H_c with $c \in \{1,2, \ldots, d\}$ be a handle. Placing a pebble p within H_c requires at most $|H_c|$ rotations of the cycle $C(H_c)$ (procedures procedures *Rotate-Cycle*$^+$ and *Rotate-Cycle*$^-$) in the positive direction in case when p is needed to be moved outside H_c. At most $|H_c|$ rotations of $C(H_c)$ in the negative direction are then necessary to put pebbles in H_c to their original positions. Finally, one rotation of $C(H_c)$ in the positive direction is necessary to get the pebble p to its right 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. Thus, all the rotations needed to place the pebble p consume at most $(2|H_c| + 1)|C(H_c)|$ steps.

It is also necessary to move the pebble p (procedure *Move-Pebble*) within the placement operation. There are up to 2 calls of *Move-Pebble* per pebble placement within the handle H_c. A careful analysis must be done here since the pebble p must be moved along a path of the length up to $|V|$ and lot of work is done within each edge *traversal*. To reduce time complexity of the operation, a pair of vertex disjoint paths φ and χ connecting the original location of p and the target vertex is computed at the beginning. This is done by a direct application of Lemma 2 and it consumes $\mathcal{O}(|V| + |E|)$ steps since the handle decomposition must be gone through.

Notice that a vertex in front of the current location of p needs to be made unoccupied. Therefore, an alternative path avoiding the vertex with p must be found and pebbles must be shifted along this path. The knowledge of χ allows determining of such path (which is not needed to be shortest one) in constant time since it consists of χ and parts of φ. Shifting pebbles itself consumes exactly $|V|$ steps. Thus, a single traversal of an edge of φ by the pebble p requires at most $|V|$ steps.

Altogether, $|V|^2 + \mathcal{O}(|V| + |E|)$ steps are required by the operation of moving a pebble in the worst case. That is, $2|V|^2 + \mathcal{O}(|V| + |E|)$ steps per pebble placement.

There is also up to 5 calls of the operation for making some vertex unoccupied (procedure *Make-Unoccupied*). The remaining operations consume constant time. The operation for making some vertex unoccupied requires $\mathcal{O}(|V| + |E|)$ steps; this is caused by the search for a shortest path connecting original and goal location. Shifting pebbles 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 p into H_c. In total, at most $(2|H_c| + 1)|C(H_c)| + 2|V|^2 + 5|V| + \mathcal{O}(|V| + |E|)$ steps are necessary to place the pebble p into H_c. Since $|H_c| \leq |C(H_c)| \leq |V|$, the total number of steps is at most $(2|V| + 1)|V| + 2|V|^2 + 5|V| + \mathcal{O}(|V| + |E|)$ which is $\mathcal{O}(|V|^2)$. Since there are at most $|V|$ pebbles, the whole process of placing pebbles into handles takes $\mathcal{O}(|V|^3)$ steps.

It remains to analyze time required by placing pebbles within the original cycle C_0 of the handle decomposition There are at most $|V|$ pebbles to be placed in C_0. Each pebble p 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 pebbles. Since the initial and the goal position of the mentioned transfer of the unoccupied vertex in both cases is located in C_0, the operation requires only $|C_0|$ steps in the worst case. The operation of exchanging pebbles requires at most $2|C_0|$ rotations in the positive direction (lines **5** and **13** of *Exchange-Pebbles*) and at most $|C_0|$ rotation in the negative direction (line **24** of *Exchange-Pebbles*). Next, there are 3 calls of the operation for making some vertex unoccupied (call of the procedure *Make-Unoccupied* at lines **2**, **8**, and **17**). Observe that the unoccupied vertex and the target vertex of the transfer 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 pebbles. The time consumption of the remaining operations performing during a single exchange of pebbles is constant, thus it is not necessary to account them. To exchange a pair of pebbles at most $3|C_0|^2 + 5|C_0|$ steps are needed in total. Placing all the pebbles into the original cycle requires at most $|C_0|(3|C_0|^2 + 5|C_0|)$ steps. Since $|C_0| < |V|$, the total number of steps required for the initial cycle is at most $|V|(3|V|^2 + 5|V|)$ which is $\mathcal{O}(|V|^3)$.

The worst case time complexity of the BIBOX algorithm with respect to the input instance $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ is thus $\mathcal{O}(|V|^3)$. ■

Proposition 4 ***(BIBOX – makespan of the solution).*** The **makespan** of a solution in the worst case produced by the BIBOX algorithm (that is, the number ξ) for an input instance of the problem of pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ is $\mathcal{O}(|V|^3)$. ■

Proof. The proof will proceed exactly in the same way as the proof of the worst case time complexity since the step of the algorithm corresponds to move of a pebble from a vertex to its unoccupied neighbor in almost all the cases – called a *swap* (directly corresponds to a call of the procedure *Swap-Pebbles*).

Consider the process of placing a pebble p into a handle H_c of a handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \dots, H_d]$ where $c \in \{1,2, \dots, d\}$. It requires $2|H_c| + 1$ rotations of the cycle $C(H_c)$ (procedures *Rotate-Cycle*$^+$ and *Rotate-Cycle*$^-$) where each rotation produces $C(H_c)$ swaps. That is, $|C(H_c)|(2|H_c| + 1)$ swaps are caused by rotations. Next, there are up to $|V| + 1$ edge traversals by the pebble p caused by the operation of moving a pebble (procedure *Move-Pebble*) where each edge traversal produces $|V|$ swaps. Altogether, $|V|(|V| + 1)$ are swaps are produced by moving pebbles. Finally, there are up to 5 calls of the operation for making a vertex unoccupied (procedure *Make-Unoccupied*) where each call produces at most $|V|$ swaps. Since there are at most $|V|$ pebbles and $|H_c| \leq |C(H_c)| \leq |V|$ at most $|V|(|V|(2|V| + 1) + |V|(|V| + 1) + 5|V|)$ swaps which is $\mathcal{O}(|V|^3)$ are necessary to place pebbles in handles.

It is also necessary to consider the process of placing pebbles in the initial cycle C_0. There are at most $3|C_0|$ rotations of C_0 and 5 operation of making a vertex unoccupied per pebble placement. A single rotation of C_0 produces $|C_0|$ swaps and one making a vertex unoccupied requires at most $|C_0|$ swaps (recall, that unoccupied vertex as well as its target are located in C_0). Altogether, at most $3|C_0|^2 + 5|C_0|$ swaps are produced per pebble placement into C_0. Placing all the pebbles in C_0 produces at most $|C_0|(3|C_0|^2 + 5|C_0|)$ swaps. Since $|C_0| < |V|$, it is at most $|V|(3|V|^2 + 5|V|)$ swaps which is $\mathcal{O}(|V|^3)$ for placing pebbles into the initial cycle C_0.

The final transfer of unoccupied vertices to their original locations produces at most $2|V|$ swaps which does not change the above asymptotic estimation. In total, $\mathcal{O}(|V|^3)$ swaps are produced by the *BIBOX* algorithm when solving the instance $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. As the makespan ξ is bounded by the number of swaps (that is, the makespan of the solution with no parallelism allowed), it is possible to conclude that makespan is $\mathcal{O}(|V|^3)$. ■

Proposition 5 ***(BIBOX – worst case space complexity).*** The worst case **space complexity** of the *BIBOX* algorithm is $\mathcal{O}(|V| + |E|)$ with respect to an input instance of the problem pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. ■

Proof. The size of the solution $\mathcal{S}_P$ is up to $\mathcal{O}(|V|^3)$ times $\mathcal{O}(|V|)$; that is, the makespan in the worst case ξ multiplied by the space necessary for storing an individual element of the solution S_P^i with $i \in \{0,1, \ldots, \xi\}$. Fortunately, the solution $\mathcal{S}_P$ is produced in a **stream** like manner and thus does not need be stored in the memory.

A space of $\mathcal{O}(|V| + |E|)$ is required for storing the current arrangement of pebbles expressed by functions S_P and Φ_P; a space of $\mathcal{O}(|V| + |E|)$ is required to compute (Lemma 1, [24]) and to store the handle decomposition $\mathcal{D}$; and finally a space of $\mathcal{O}(|V| + |E|)$ is required to compute shortest paths by subsequent calls of the Dijkstra's algorithm [1].

Remaining operations and local variables used by the *BIBOX* algorithm consume the space proportional to the size of the input instance Π.Thus, the space of $\mathcal{O}(|V| + |E|)$ which is proportional to the size of the input instance Π is required by the *BIBOX* algorithm in total. ■

3.1.4. *Extensions and the Real-life Implementation*

The natural question is how to apply the *BIBOX* algorithm if there are more than two unoccupied vertices in input instance (that is, $\mu - 2 \leq |V|$). The algorithm can be used directly if the graph is filled by dummy pebbles. The instance with dummy pebbles is solved by the algorithm as it is and finally movements of dummy pebbles are filtered out from the solution in an additional post-processing step.

An adaptation of the solving algorithm for sparse instances of the pebble motion problem is out of scope of this work. Nevertheless, a straightforward adaptation is to replace the **non-deterministic** selection of an unlocked unoccupied vertex (such as that at line **1**

of *Make-Unoccupied*) by the selection of the most promising one. For example, an unlocked unoccupied vertex that is nearest to the vertex that is to be made unoccupied is selected. Indeed, this behavior is adopted in the experimental implementation of the *BIBOX* algorithm (see experiments in Section 5). Some further optimizations should be used in the real-life implementation to reduce the makespan of the produced solution. Various preconditions 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 pebbles, locations of unoccupied vertices should be detected implicitly in subsequent steps by more sophisticated branching of the code).

The experimental implementation of procedures *Solve-Regular-Handle* and *Solve-Original-Cycle* uses opportunistic selection of vertices to store pebbles (vertex y - line **23** of *Solve-Regular-Handle* and vertices u, v - line **1** of *Solve-Original-Cycle*). The nearest vertex to the target pebble is always selected. 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 its main loop.

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

The significant 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 pebbles are placed into the initial cycle of the handle decomposition. Thus, if there is only one unoccupied vertex in the input instance, the *BIBOX* algorithm would be able to place almost all the pebbles of the input instance except that which goal positions are within the initial cycle of the handle decomposition.

It would be possible to apply the existent algorithm described in [8] for solving pebble motion problems to finish placement of pebbles in the initial cycle. It is referred to as the *MIT*[1] algorithm in this article. The *MIT* algorithm is able to solve instances of the problem of pebble motion on a non-trivial bi-connected graph 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, a combined algorithm can proceed as the *BIBOX* algorithm for placing pebbles into all the internal vertices of handles of the handle decomposition and it can proceed as the *MIT* algorithm over the remaining initial cycle and the first handle (the first handle is necessary to be included to form a bi-connected graph together with the initial cycle). Unfortunately, the process how the *MIT* algorithm places pebbles generates excessively long sequences of moves (see experiments in Section 5).

Despite above facts the idea of using alternative solving process for the initial cycle of the handle decomposition is still promising. Since the initial cycle and the first handle

[1] The name for the algorithm has been chosen according to the name of the institution of the principal author of the article [8] which is MIT – the Massachusetts Institute of Technology.

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 pebble motion problem over these graphs **optimally** with respect to the makespan. Notice, that as there is no parallelism with single unoccupied vertex, the makespan equals to the number of moves in the solution. The candidate instances for optimal solving are those from which solutions an overall solution of any instance over the initial cycle and the first handle can be composed. Moreover, 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 suppose that the makespan of the resulting solution will be acceptable. Nevertheless, this is a conjecture that should be proven.

3.2.1. *Algebraic Foundation of the Algorithm*

Bi-connected graph, which handle decomposition consists of the initial cycle and single handle, represent structurally simplest bi-connected graphs over that non-trivial rearrangement of pebbles is possible supposed there is single unoccupied vertex (structurally simpler bi-connected graph is a cycle where only rotations of pebbles are possible). These graphs will be referred to as *θ-like graphs*.

Definition 5 ***(θ-like graph).*** Let $A = [\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_\alpha]$, $B = [\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_\beta]$, and $C = [\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_\gamma]$ be three sequences of vertices satisfying that $\alpha \geq 1 \wedge \beta \geq 2 \wedge \gamma \geq 1$. An undirected graph $\theta(A, B, C) = (V_\theta, E_\theta)$ for such three sets is constructed as follows: $V_\theta = A \cup B \cup C$ and $E_\theta = \{\{\bar{a}_1, \bar{a}_2\}, \{\bar{a}_2, \bar{a}_3\}, \ldots, \{\bar{a}_{\alpha-1}, \bar{a}_\alpha\};\ \{\bar{b}_1, \bar{b}_2\}, \{\bar{b}_2, \bar{b}_3\}, \ldots, \{\bar{b}_{\beta-1}, \bar{b}_\beta\};\ \{\bar{c}_1, \bar{c}_2\}, \{\bar{c}_2, \bar{c}_3\}, \ldots, \{\bar{c}_{\gamma-1}, \bar{c}_\gamma\};\ \{\bar{a}_1, \bar{b}_1\}, \{\bar{b}_\beta, \bar{c}_\gamma\}, \{\bar{b}_1, \bar{c}_1\}, \{\bar{a}_\alpha, \bar{b}_\beta\}\}$. An undirected graph $G = (V, E)$ is called a *θ-like graph* if there exist three sets of vertices A, B, and C as above such that G is isomorphic to $\theta(A, B, C)$. □

The notation of 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 $\theta(A, B, C)$ itself is a θ-like graph and $\theta(A, B, C)$ may be identical to G if sets A, B, and C consist of vertices of G. Hence, no distinction is made between G and $\theta(A, B, C)$ in the following text and the notation $\theta(A, B, C)$ is used exclusively. An example of θ-like graph is shown in Figure 9.

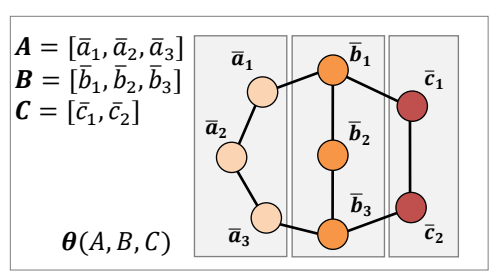


Figure 9. *An example of* ***θ-like graph***. θ-like graphs are bi-connected graphs consisting of a cycle and one handle.

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 into sub-sets A, B, and C, where these sub-sets form continuous sub-sequences within the ordered V; A is the first sub-sequence, B is the middle sub-sequence, and C is the last sub-sequence within V; there is $\mathcal{O}(|V|^2)$ possibilities

to place separation points among A, B, and C). However, the number of all the possible instances of pebble motion problem with 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 of pebbles 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 of pebble motion 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, solutions to all the possible instances of pebble motion problem with single unoccupied vertex over the θ-like graph should be possible to be composed of the solutions to the selected instances.

Without loss of generality, assume that the unoccupied vertex within the initial and the goal arrangements of an instance of the problem over $\theta = (V_\theta, E_\theta)$ is $\bar{b}_1$ (the unoccupied vertex can be simply transferred to any vertex). Thus, the space of such instances over θ is isomorphic to the group of all the permutations of $|V_\theta| - 1$ which is called a *symmetric group* on $|V_\theta| - 1$ elements and it is denoted $Sym(|V_\theta| - 1)$ [2, 14]. A ***transposition*** is a permutation, which exchanges a pair of elements and keeps other elements fixed. It is well known from the theory of groups that $Sym(|V_\theta| - 1)$ can be generated by the set of transpositions on the same set of elements. A permutation is called *odd* if it can be composed of the odd number of transpositions. A permutation is called *even* if it can be composed of the even number of transpositions. A permutation is either odd of even but not both. In fact, if a permutation is assigned a *sign* which is $+1$ if the permutation is even and -1 if the permutation is odd by a function sgn, then sgn is a *group homomorphism* between $Sym(|V_\theta| - 1)$ and the group $(\{-1, +1\}, *, +1, -)$ where multiplication $*$ corresponds to a product of two permutations, neutral element $+1$ corresponds to identical permutation and unary minus – corresponds to an inverse permutation.

Another simple fact that can be derived from above statements is that the set of all the 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 $|V_\theta| - 1$ and it is denoted as $Alt(|V_\theta| - 1)$. A ***rotation along a 3-cycle*** is a permutation which rotates given three elements and keeps other fixed. In [8] it is shown how to compose any even permutation from rotations along 3-cycles on the same set of elements.

As the number of distinct transpositions on n elements is $\mathcal{O}(n^2)$ and the number of distinct rotations along 3-cycles on n elements is $\mathcal{O}(n^3)$, optimal solutions of corresponding instances of pebble motion problem seem to be good candidates for storing. Moreover, if the corresponding instances are really solvable, then they satisfy the property that solution to any (in the case of transpositions) or almost any (in the case of 3-cycle rotations) pebble motion instance on the same graph can be composed of them. Composition of solutions can be simply implemented as their **concatenation**.

Suppose a θ-like graph $\theta(A, B, C) = (V_\theta, E_\theta)$ with $A = [\bar{a}_1, \bar{a}_2, \dots, \bar{a}_\alpha]$, $B = [\bar{b}_1, \bar{b}_2, \dots, \bar{b}_\beta]$, and $C = [\bar{c}_1, \bar{c}_2, \dots, \bar{c}_\gamma]$ and a set of pebbles $P = \{p_1, p_2, \dots, p_{|V_\theta|-1}\}$ for the following three definitions.

Definition 6 ***(even and odd case).*** Let S_P^0 an initial arrangement of the set of pebbles such that $S_P^0(p) \neq \bar{b}_1 \ \forall p \in P$ (that is, $\bar{b}_1$ is initially unoccupied) and let S_P^+ be a goal arrangement of pebbles such that $S_P^+(p) \neq \bar{b}_1 \ \forall p \in P$ (that is, $\bar{b}_1$ is finally unoccupied). If S_P^+ forms an even permutation with respect to S_P^0, then an instance of the problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, S_P^+)$ is called an *even case*. If S_P^+ forms an odd permutation with respect to S_P^0, then an instance of the problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, S_P^+)$ is called an *odd case*.. □

Definition 7 ***(transposition case).*** Let S_P^0 be an initial arrangement of the set of pebbles such that $S_P^0(p) \neq \bar{b}_1 \ \forall p \in P$ (that is, $\bar{b}_1$ is initially unoccupied) and let S_P^+ be a goal arrangement of pebbles such that there exist $q_1, q_2 \in P$ such that $q_1 \neq q_2$ for which it holds that $S_P^+(q_1) = S_P^0(q_2) \ \wedge \ S_P^+(q_2) = S_P^0(q_1) \ \wedge \ (\forall p \in P)(p \neq q_1 \wedge p \neq q_2) \Rightarrow S_P^+(p) = S_P^0(p)$ (pebbles q_1 and q_2 are to be exchanged while positions of other pebbles are preserved; consequently $\bar{b}_1$ is finally unoccupied). Then an instance of the problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, S_P^+)$ is called a *transposition case* with respect to q_1 and q_2. □

Definition 8 ***(3-cycle rotation case).*** Let S_P^0 be an initial arrangement of the set of pebbles such that $S_P^0(p) \neq \bar{b}_1 \ \forall p \in P$ ($\bar{b}_1$ is initially unoccupied). Let S_P^+ be a goal arrangement of pebbles such that there exist $q_1, q_2, q_3 \in P$ such that q_1, q_2, and q_3 are pair wise distinct for which it holds that $S_P^+(q_1) = S_P^0(q_2) \wedge S_P^+(q_2) = S_P^0(q_3) \wedge S_P^+(q_3) = S_P^0(q_1) \wedge (\forall p \in P)(p \neq q_1 \wedge p \neq q_2 \wedge p \neq q_3) \Rightarrow S_P^+(p) = S_P^0(p)$ (pebbles q_1, q_2, and q_3 are to be rotated while positions of other pebbles are preserved; consequently $\bar{b}_1$ is finally unoccupied again). Then an instance of the problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, S_P^+)$ is called a *3-cycle rotation case* with respect to q_1, q_2, and q_3. □

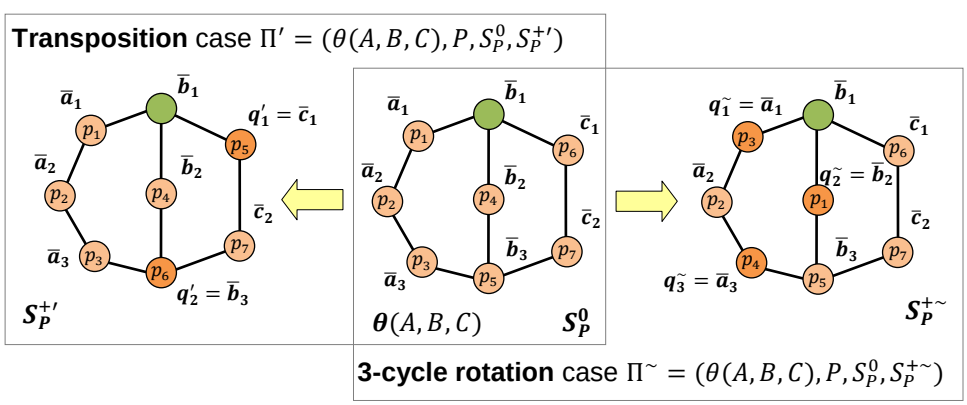


Figure 10. *An example of* ***transposition*** *and* ***3-cycle*** *rotation cases of the problem of pebble motion on a θ-like graph.* The transposition case is shown for vertices $q_1' = \bar{c}_1$ and $q_2' = \bar{b}_3$. The 3-cycle rotation case is shown for vertices $q_1^{\sim} = \bar{a}_1$, $q_2^{\sim} = \bar{b}_2$, and $q_3^{\sim} = \bar{a}_3$. Solutions of general instances of the problem of pebble motion on a given θ-like graph that are solvable can be composed of (optimal) solutions of transposition and 3-cycle rotation cases.

See Figure 10 for illustrations of transposition case and 3-cycle rotation case. Notice, that transposition and 3-cycle rotation 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 6 ***(solvability of an odd case)*****.** An **odd** case of the problem of pebble motion on a θ-like graph $\Pi = (\theta(A,B,C), P, S_P^0, S_P^+)$ with $|A| \neq 2 \vee |B| \neq 3 \vee |C| \neq 2$ is solvable if and only if θ contains a **cycle** of the **odd length**. ■

Let the θ-like graph $\theta(A,B,C)$ with $|A| = 2 \wedge |B| = 3 \wedge |C| = 2$ 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 transposition is an odd permutation, the following corollary is a direct consequence of the above proposition.

Corollary 1 ***(solvability of transposition case)*****.** A **transposition** case of the problem of pebble motion on a θ-like graph $\Pi = (\theta(A,B,C), P, S_P^0, S_P^+)$ with $\theta(A,B,C)$ non-isomorphic to $\theta(2,3,2)$ is solvable if and only if $\theta(A,B,C)$ contains a **cycle** of the **odd length**. ■

Proposition 7 ***(solvability of an even case)*****.** An **even** case of the problem of pebble motion on a θ-like graph $\Pi = (\theta(A,B,C), P, S_P^0, S_P^+)$ with $\theta(A,B,C)$ 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)*****.** A **3-cycle rotation** case of the problem of pebble motion on a θ-like graph $\Pi = (\theta(A,B,C), P, S_P^0, S_P^+)$ with $\theta(A,B,C)$ 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 the odd length and the initial and the goal arrangement of pebbles 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 a corollary of the *BIBOX* algorithm and Proposition 7).

The following propositions [2, 8, 14] 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.

Proposition 8 ***(solving the 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. ■

Of course, it is not possible to compose a solution to an odd case of rotations along 3-cycles since the composition of any even two permutations results in an even permutation and rotation along a 3-cycle is an even permutation (recall the group homomorphism). On the other hand, a solution of an even case can be composed of at most $|V_\theta| - 2$ solutions to transposition cases as well. The proof of the propositions is shown within the pseudo-code of the *BIBOX-*θ algorithm.

Proposition 9 ***(solving the 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. ■

Again, the proof is shown within the pseudo-code of the *BIBOX-*θ algorithm. The above facts justify that transposition and 3-cycle rotation case are suitable to be solved optimally and corresponding solutions can be used for composing solutions of general instances over θ-like graphs. It is out of scope of this manuscript 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 [17].

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 pebble motion instances over this θ-like graph into the database. The solving process of the new algorithm over the initial cycle and the first handle of the handle decomposition 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. The proof the lemma tediously enumerates all the possible cases.

Lemma 5 ***(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 $\theta(A,B,C)$ non-isomorphic to $\theta(2,3,2)$. Moreover, if G contains an odd cycle then it subsumes $\theta(A,B,C)$ non-isomorphic to $\theta(2,3,2)$ that additionally satisfies that $2 \nmid |A| + |B|$ (that is, sets A and B 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 $\theta(A,B,C) = C_0 \circ H_1$. ($C_0 \circ H_1$

denotes the sub-graph of G constructed by addition of the handle H_1 to the initial cycle C_0). ■

Proof. The proof will proceed as case analysis according to the number of handles of a handle decomposition of G and according to the number of internal vertices of a handle. Since the graph G is a non-trivial bi-connected graph, it holds for any handle decomposition of G that it contains at least one handle.

If G does not subsume any cycle of the odd length then a θ-like sub-graph $\theta(A,B,C)$ constructed from the initial cycle and the first handle of any handle decomposition satisfies requirement of the lemma. The graph $\theta(A,B,C)$ cannot subsume an odd cycle and hence it cannot be isomorphic to $\theta(2,3,2)$.

If G subsumes a cycle of the odd length then let $\mathcal{D}' = [C_0', H_1', H_2', \ldots, H_{d'}']$ be some fixed handle decomposition of G such that C_0' is of the **odd** length (such handle decomposition can be constructed by finding an odd cycle first and then by continuing as standard method for finding handle decomposition - see Lemma 1).

The lemma holds for $d' = 1$ since it states a trivial fact and it is possible to set $\mathcal{D} = \mathcal{D}'$ to obtain the second part of the lemma.

Assume that $d' = 2$. If $C_0' \circ H_1'$ is non-isomorphic to $\theta(2,3,2)$ then $\theta(A,B,C) = C_0' \circ H_1'$ and $\mathcal{D} = \mathcal{D}'$ fulfills the lemma again. Consider, that $C_0' \circ H_1'$ is isomorphic to $\theta(2,3,2)$. A modifying construction of $\mathcal{D}' = [C_0', H_1', H_2']$ will be shown. The result of the construction will be a new handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ that satisfies requirements of the lemma. Let $\theta'(A',B',C') = C_0' \circ H_1'$ where $A' = [a_1', a_2']$, $B' = [b_1', b_2', b_3']$, and $C' = [c_1', c_2']$; that is, the initial cycle C_0' together with the first handle are interpreted as a θ-like graph. The following cases must be distinguished (symmetric cases are not listed):

(1) Assume that H_2' interconnects a_1' and b_1':
H_2' must contain at least one internal vertex since otherwise there will be two edges connecting a_1' and b_1' which is not allowed in a standard undirected graph. Let $H_2' = [a_1', d_1, b_2']$ (d_1 is a vertex different from vertices of θ'). Then a θ-like graph $\theta(A,B,C)$ with $A = [a_2', b_3', b_2']$, $B = [a_1', b_1']$, and $C = [d_1]$ is a sub-graph of G, it is not isomorphic to $\theta(2,3,2)$, and it holds that $2 \nmid |A| + |B|$. The corresponding handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be as follows: $C_0 = [a_1', a_2', b_3', b_2', b_1']$, $H_1 = [a_1', d_1, b_1']$, and $H_2 = [b_1', c_1', c_2', b_3']$.

It is not difficult to extend the above construction for cases when H_2' contains more than one internal vertices. It causes growth of the sequence C while A and B remain the same. Thus the non-isomorphism with $\theta(2,3,2)$ and the oddness of $|A| + |B|$ are preserved.

(2) Assume that H_2' interconnects a_1' and a_2':
Again, H_2' must contain at least one internal vertex since multiple edges connecting the same pair of vertices are not allowed. Let H_2' has one internal vertex, that is $H_2' = [a_1', d_1, a_2']$. Then a θ-like graph $\theta(A,B,C)$ with $A = [b_1', b_2', b_3']$, $B = [a_1', a_2']$, and $C = [d_1]$ is a sub-graph of G which is not isomorphic to

$\theta(2,3,2)$ and satisfies that $2 \nmid |A| + |B|$. The corresponding handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be constructed as follows: $C_0 = [a'_1, a'_2, b'_3, b'_2, b'_1]$, $H_1 = [a'_1, d_1, a'_2]$, and $H_2 = [b'_1, c'_1, c'_2, b'_3]$.

The construction can be easily extended for cases when there are more than one internal vertices within H'_2. It results in growth of the sequence C which preserves non-isomorphism with $\theta(2,3,2)$ as well as oddness of $|A| + |B|$.

(3) Assume that H'_2 interconnects a'_1 and b'_2:
If $H'_2 = [a'_1, b'_2]$, that is, it has **no internal** vertices, then a θ-like graph $\theta(A, B, C)$ with $A = [c'_1, c'_2, b'_3]$, $B = [b'_1, b'_2]$, and $C = [a'_1]$ is a sub-graph of G, it is not isomorphic to $\theta(2,3,2)$, and it holds that $2 \nmid |A| + |B|$. The handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be as follows: $C_0 = [b'_1, b'_2, b'_3, c'_2, c'_1]$, $H_1 = [b'_1, a'_1, b'_2]$, and $H_2 = [a'_1, a'_2, b'_3]$.

It is easy to extend the described treatment when H'_2 interconnects a'_1 and b'_2 and has **non-zero** number of **internal** vertices. It causes growth of the sequence C while sequences A and B remain the same. Hence, the non-isomorphism with $\theta(2,3,2)$ and the oddness of $|A| + |B|$ are preserved.

(4) Assume that H'_2 interconnects a'_1 and b'_3:
If $H'_2 = [a'_1, b'_3]$ (there is **no internal** vertex) then a θ-like graph $\theta(A, B, C)$ with $A = [c'_1, c'_2]$, $B = [b'_1, b'_2, b'_3]$, and $C = [a'_1]$ and a handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ with $C_0 = [b'_1, b'_2, b'_3, c'_2, c'_1]$, $H_1 = [b'_1, a'_1, b'_3]$, and $H_2 = [a'_1, a'_2, b'_3]$ satisfy requirements of the lemma.

If $H'_2 = [a'_1, d_1, b'_3]$ (there is **one internal** vertex) then a θ-like graph $\theta(A, B, C)$ with $A = [b'_1, b'_2]$, $B = [a'_1, d_1, b'_3]$, and $C = [a'_2]$ is a sub-graph of G, it is not isomorphic to $\theta(2,3,2)$, and it holds that $2 \nmid |A| + |B|$. The corresponding handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be as follows: $C_0 = [a'_1, d_1, b'_3, b'_2, b'_1]$, $H_1 = [a'_1, a'_2, b'_3]$, and $H_2 = [b'_1, c'_1, c'_2, b'_3]$.

If $H'_2 = [a'_1, d_1, d_2, b'_3]$ (there are **two internal** vertices) then a θ-like graph $G_\theta(A, B, C)$ with $A = [c'_1, c'_2]$, $B = [b'_1, b'_2, b'_3]$, and $C = [a'_1, d_1, d_2]$ is a sub-graph of G, it is non-isomorphic to $\theta(2,3,2)$, and it holds that $2 \nmid |A| + |B|$. The corresponding handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be as follows: $C_0 = [b'_1, b'_2, b'_3, c'_2, c'_1]$, $H_1 = [b'_1, a'_1, d_1, d_2, b'_3]$, and $H_2 = [a'_1, a'_2, b'_3]$.

If H'_2 contains **more than two internal** vertices then non-isomorphism with $\theta(2,3,2)$ as well as oddness of $|A| + |B|$ remain preserved (this construction would not work for exactly one internal vertex of H'_2 because the constructed $\theta(A, B, C)$ would be isomorphic to $\theta(2,3,2)$).

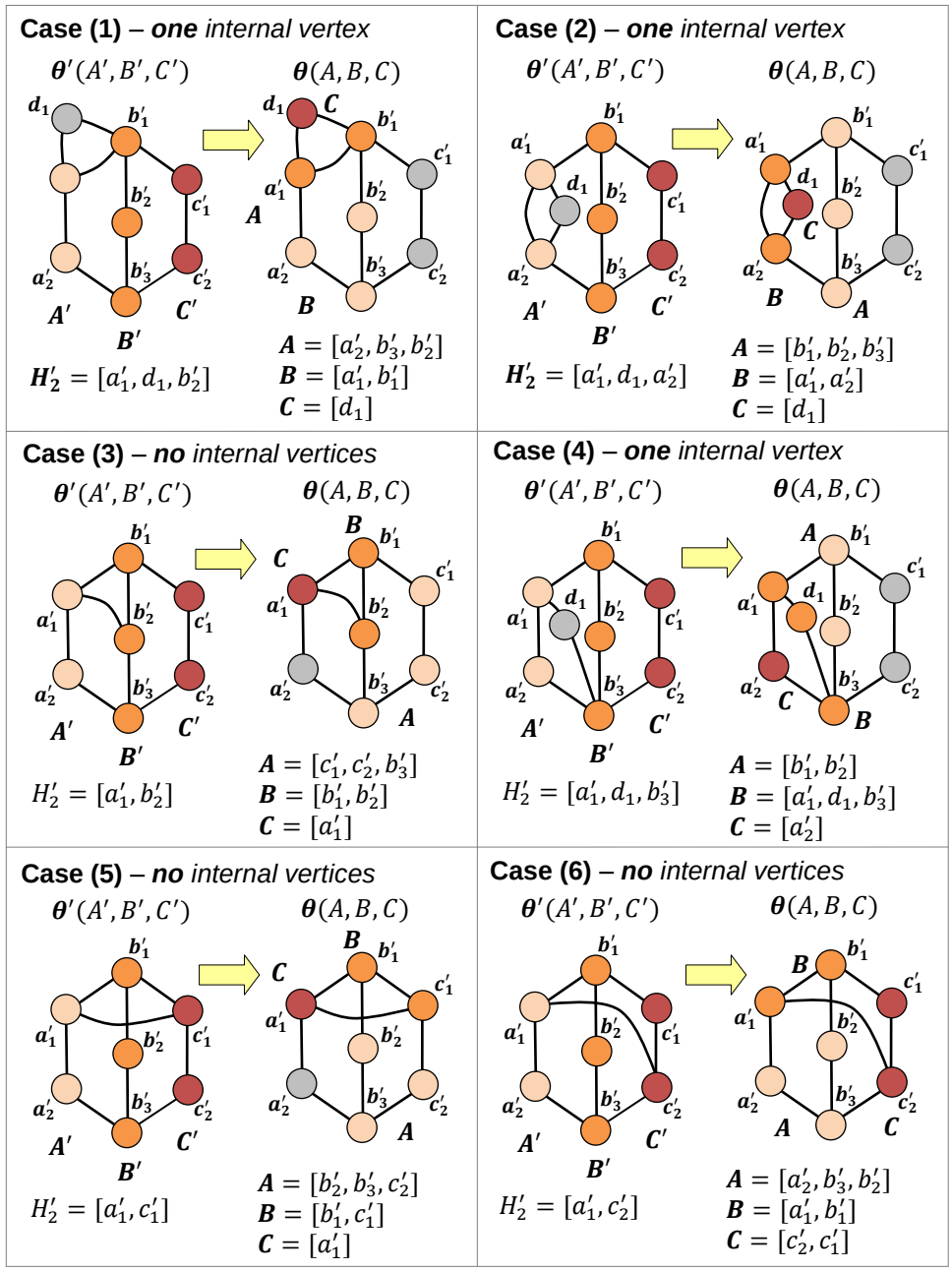


Figure 11. *An illustration of* ***avoiding*** $\theta(2,3,2)$. If a non-trivial bi-connected graph is non-isomorphic to $\theta(2,3,2)$ then it subsumes a θ-like sub-graph $\theta(A,B,C)$ non-isomorphic to $\theta(2,3,2)$. The illustration shows how to find the required $\theta(A,B,C)$ if unluckily a sub-graph isomorphic to $\theta(2,3,2)$ is encountered.

(5) Assume that H'_2 interconnects a'_1 and c'_1:
If $H'_2 = [a'_1, c'_1]$ (H'_2 has **no internal** vertex) then a θ-like graph $\theta(A, B, C)$ with $A = [b'_2, b'_3, c'_2]$, $B = [b'_1, c'_1]$, and $C = [a'_1]$ and a handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ with $C_0 = [b'_1, b'_2, b'_3, c'_2, c'_1]$, $H_1 = [b'_1, a'_1, c'_1]$, and $H_2 = [a'_1, a'_2, b'_3]$ satisfy requirements of the lemma.

Again, observe that it is easy to extend this treatment when H'_2 interconnects a'_1 and c'_1 and has **more than zero internal** vertices. Internal vertices of H'_2 cause growth of the sequence C while sequences A and B remain the same. Hence, the non-isomorphism with $\theta(2,3,2)$ and oddness of $|A| + |B|$ will be preserved again.

(6) Assume that H'_2 interconnects a'_1 and c'_2:
If $H'_2 = [a'_1, c'_2]$ (H'_2 has **no internal** vertex) then a θ-like graph $\theta(A, B, C)$ with $A = [a'_2, b'_3, b'_2]$, $B = [a'_1, b'_1]$, and $C = [c'_2, c'_1]$ is a sub-graph of G non-isomorphic to $\theta(2,3,2)$, and it holds that $2 \nmid |A| + |B|$. The corresponding handle decomposition $\mathcal{D} = [C_0, H_1, H_2]$ will be as follows: $C_0 = [a'_1, a'_2, b'_3, b'_2, b'_1]$, $H_1 = [a'_1, c'_2, c'_1, b'_1]$, and $H_2 = [b'_3, c'_2]$.

Observe again that if H'_2 contains **more than zero internal** vertices then it causes growth of the sequence C which does not affect the non-isomorphism with $\theta(2,3,2)$ and the oddness of $|A| + |B|$.

If it holds that $d' > 2$ then the above modifying construction is applied on first two handles and the initial cycle of $\mathcal{D}' = [C'_0, H'_1, H'_2, \ldots, H'_{d'}]$ which of the result are C_0, H_1, and H_2. Then it is sufficient to set $\mathcal{D} = [C_0, H_1, H_2, H'_3, H'_4, \ldots, H'_{d'}]$.

The modifying construction can be carried out in the worst case time of $\mathcal{O}(|V| + |E|)$ and the worst case space of $\mathcal{O}(|V| + |E|)$. ■

The case analysis from the proof Lemma 5 is shown in Figure 11. The lemma is crucial in showing that the upcoming algorithm is sound.

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 $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ be an input instance of the problem of pebble motion on a **biconnected** graph with **single unoccupied** vertex. If G is non-isomorphic to $\theta(2,3,2)$ and it subsumes a cycle of the 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 the odd length and $C_0 \circ H_1$ is non-isomorphic to $\theta(2,3,2)$ is computed. Lemma 5 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, it is necessary 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_P^+ by shifting goal locations of pebbles along a path φ to relocate the unoccupied vertex into C_0. The modified instance is then solved by the process im-

plemented by the *BIBOX-θ* algorithm. The solution is finished by calling a function *Finish-Solution* which shifts pebbles back along the path φ.

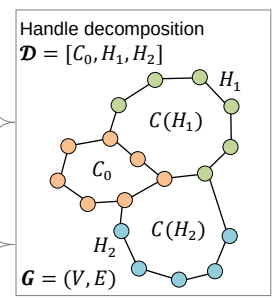
The *BIBOX-θ* algorithm proceeds according to the handle decomposition $\mathcal{D}$ from the last handle H_d to the second handle H_2 and initial cycle C_0. The process of placement of pebbles within the individual handles of the handle decomposition is the same as in the case of the *BIBOX* algorithm. The problem of reaching the goal arrangement of pebbles 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 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 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 the 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 can 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 used in such a case.

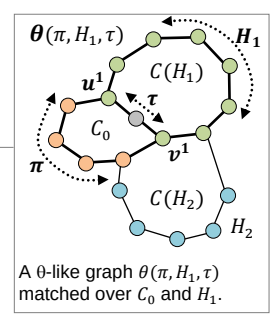
The pseudo-code of the *BIBOX-θ* algorithm is listed as Algorithm 2. It reuses primitives, functions, and procedures introduced within the context of *BIBOX* algorithm - functions and procedures from Algorithm 1 are called. For simplicity, it is supposed that all the required optimal solutions are stored in the database (so there is no treatment when the size of the θ-like graph exceeds the limit).

The database with optimal solutions of 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 pebbles are transposed. Similarly, 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 pebbles 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 pebbles (a function *difference* is used for calculating this differencing permutation).

Algorithm 2. *The **BIBOX-θ** algorithm.* This is an improved version of the *BIBOX* algorithm. The design of the algorithm originates from [17, 19]. It solves a given pebble motion problem on a non-trivial bi-connected graph with exactly **one unoccupied** vertex. The improvement with respect to the *BIBOX* algorithm consists in exploiting database containing **optimal solutions** to sub-problems over the **original cycle** and the **first handle**. This consequently led to the relaxation of the requirement to have two unoccupied vertices of the original version. Functions and procedures from Algorithm 1 are reused here.

function *BIBOX-θ-Solve*$(G = (V, E), P, S_P^0, S_P^+)$: **pair**
/* Top level function of the BIBOX algorithm; solves a given problem of pebble motion on a graph.
Parameters: G - a graph modeling the environment,
P - a set of pebbles,
S_P^0 - an initial arrangement of pebbles,
S_P^+ - a goal arrangement of pebbles. */
1: **if** G contains a cycle of the 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 the 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, P, S_P^+, C_0)
8: $\xi \leftarrow 1$
9: $S_P \leftarrow S_P^0$
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** $(\xi, [S_P^0, S_P^1, \ldots, S_P^\xi])$



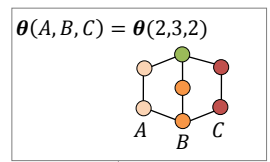


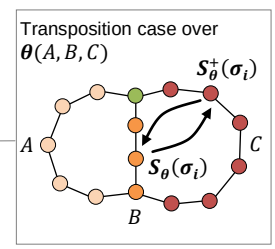
A θ-like graph $\theta(\pi, H_1, \tau)$ matched over C_0 and H_1.

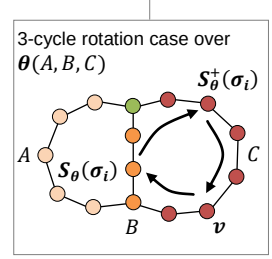
procedure *θ-BOX-Solve*$(\theta(A,B,C), V^+, S_\theta^0, S_\theta^+)$

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

1: **let** $(V_\theta, E_\theta) = \theta(A,B,C)$
2: **let** $\{\sigma_1, \sigma_2, \dots, \sigma_{|V^+|-1}\} = \{\sigma | S_\theta^0(\sigma) \in V^+\}$
3: **if** $|A| = 2 \wedge |B| = 3 \wedge |C| = 2$ **then**
4: $\quad \omega \leftarrow table_{232}^\theta[difference(S_\theta^0, S_\theta^+)]$
5: $\quad$ **if** $\omega = NIL$ **then fail** /* the instance is unsolvable */
6: $\quad$ Apply-Macro(ω, S_θ)
7: **else**
8: $\quad S_\theta \leftarrow S_\theta^0$
9: $\quad$ **if** G_θ contains a cycle of the odd length **then**
10: $\quad\quad$ **for** $i = 1,2,\dots,|V^+| - 2$ **do**
11: $\quad\quad\quad$ **if** $S_\theta(\sigma_i) \neq S_\theta^+(\sigma_i)$ **then**
12: $\quad\quad\quad\quad S_\theta \leftarrow$ Apply-Macro$(table_T^\theta[S_\theta(\sigma_i), S_\theta^+(\sigma_i)], S_\theta)$
/*G_θ does not contain any odd cycle */
13: $\quad$ **else**
14: $\quad\quad$ **if** S_θ^+ constitutes an odd permutation w.r.t. S_θ **then**
15: $\quad\quad\quad$ **fail** /* the instance is unsolvable */
/* S_θ^+ constitutes an even permutation w.r.t. S_θ */
16: $\quad\quad$ **else**
17: $\quad\quad\quad$ **for** $i = 1,2,\dots,|V^+| - 3$ **do**
18: $\quad\quad\quad\quad$ **if** $S_\theta(\sigma_i) \neq S_\theta^+(\sigma_i)$ **then**
19: $\quad\quad\quad\quad\quad$ **let** $v \notin \{S_\theta(\sigma_i), S_\theta^+(\sigma_1), S_\theta^+(\sigma_2), \dots, S_\theta^+(\sigma_i)\}$
20: $\quad\quad\quad\quad\quad S_\theta \leftarrow$ 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_P 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** $[(u_1, v_1), (u_2, v_2), \dots, (u_k, v_k)] = \omega$
2: **for** $i = 1,2,\dots,k$ **do**
3: $\quad$ Swap-Pebbles-Unoccupied(u_i, v_i)
4: $\quad S_\theta(\Phi_P(u_i)) \leftarrow v_i$
5: $\quad$ **return** S_θ

The main framework of the algorithm as it was described above is represented by the function *BIBOX-θ-Solve* which gets an instance of pebble motion on a non-trivial bi-connected graph $\Pi = (G = (V,E), P, S_P^0, S_P^+)$ with just 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 solution database. The middle section of the whole solving process (lines **10-12**) when pebble are placed into handles of the handle decomposition is the same as in the case of the *BIBOX* algorithm. In order to not to need to care about the location of an unoccupied vertex within instances over θ-like graph, the first connection vertex of the handle H_1 is made unoccupied (lines **14-16**) – this vertex correspond to the vertex $\bar{b}_1$ from the definition of the θ-like graph in fact. Recall, that transposition, 3-cycle rotation, and the case of $\theta(2,3,2)$ suppose the unoccupied vertex right there.

An auxiliary function *Apply-Macro* is used apply a record ω from the database of optimal solutions (the optimal solution for a sub-instance is called a ***macro*** in this context) on the current arrangement of pebbles S_θ in a given θ-like graph as well as on the global current arrangement represented by S_P and Φ_P. The optimal solution has the form of sequence of moves where the move is an ordered pair of vertices of G - the first vertex contains a pebble to be moved; the second vertex is unoccupied at the time step of execution of the move and represents the target. The execution of the macro over the current arrangement is carried out by *Swap-Pebbles-Unoccupied*; the function also makes 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 $\theta(A,B,C)$, initial and goal arrangements of pebbles 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 in which pebbles should be placed. The function distinguished between several cases.

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 pebbles. The interpretation of permutation calculated by the *difference* function is that it makes the second arrangement from the first one.

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 pebbles that should be placed. If the current location of a pebble given by S_θ is different from its goal location given by S_θ^+, then pebbles at these two locations are swapped using a solution for transposition case from the database of solutions. Notice, that the last application of transposition case places two pebbles thus it is not necessary to traverse the last pebble.

If θ is non-isomorphic to $\theta(2,3,2)$ and all the subsumed cycles are of the even length (lines **14-20**) then a 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, pebbles that should be relocated are traversed. The relocation of a pebble σ_i to its goal location $S_\theta^+(\sigma_i)$ from $S_\theta(\sigma_i)$ is done by the 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 different from all the goal vertices of all the already placed pebbles. Except moving pebbles from $S_\theta^+(\sigma_i)$ to v and from v to $S_\theta(\sigma_i)$ the required relocation of σ_i from $S_\theta(\sigma_i)$ to $S_\theta^+(\sigma_i)$ is done. Notice, that it is sufficient to traverse all the pebbles except last two. They must be inevitably placed to their goal vertices after the last 3-cycle rotation since otherwise (that is, in the case they are swapped with respect to their right placement) 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. *Theoretical Analysis of the BIBOX-θ Algorithm*

This section is devoted to some theoretical analysis of the *BIBOX-θ* algorithm. Soundness and completeness of the algorithm will be shown first. Then several results regarding time and space complexity will be shown.

Proposition 10 ***(BIBOX-θ - soundness and completeness).*** The *BIBOX-θ* algorithm is **sound** and **complete**. That is, the algorithm always terminates and provides a correct answer to the input instance of the problem of pebble motion on a non-trivial bi-connected graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ with just single unoccupied vertex. That is, it answers either that the instance Π is unsolvable or returns a solution in the case when Π is solvable. ■

Proof. The proof will go through the pseudo-code of the algorithm while preconditions of all the steps will be verified. Since many steps are easy to check, only important points will be discussed.

Fortunately, lot of work has been already done. The possibility of selecting required handle decomposition (lines **1-6** of *BIBOX-θ-Solve*) is ensured by Lemma 5. Moreover, large part of the proof has been already done within the proof of soundness and completeness of the original *BIBOX* algorithm. This concerns stage of the execution when *BIBOX-θ* proceeds as the original *BIBOX* to place pebbles into handles H_d, H_{d-1}, …, H_2 (lines **10-12** of *BIBOX-θ-Solve*).

Making unoccupied the first connection vertex u^1 of the first handle H_1 (lines *BIBOX-θ-Solve*) is possible (lines **13-16** of *BIBOX-θ-Solve*). At the end of the execution of placing pebbles into the handles H_d, H_{d-1}, …, H_2 by the *BIBOX* style process, an unoccupied vertex must be located in $C_0 \cup H_1$. Hence, it is possible relocate the unoccupied vertex to u^1 by moving pebbles within $C_0 \cup H_1$ only (an unlocked path from any vertex of $C_0 \cup H_1$ to u^1 exists; pebbles are to be shifted along this path).

Soundness and completeness of *θ-BOX-Solve* function is implied by the algebraic theory introduced in section 3.2.1. There is nothing to show about the function *Apply-Macro* since it merely executes sub-solution on the current arrangement of pebbles S_θ.

If the above argumentation is summarized, it is possible to conclude that the algorithm always terminates and if it produces a solution, it is a correct solution of the input instance. However, the case of the answer that there is no solution is more subtle.

There are two reasons for the non-existence of the solution. The first one is the case of an unsolvable instance over a θ-like graph θ isomorphic to $\theta(2,3,2)$. The whole instance over $\theta(2,3,2)$ is directly submitted to the function *θ-BOX-Solve*. It subsequently tries to find an appropriate record in the database (line **4** of *θ-BOX-Solve*) which is unsuccessful and the algorithm returns the answer that the instance is unsolvable.

The second reason for the non-existence of a solution is the case of an instance on a graph without an odd cycle where the initial arrangement S_P^0 and the goal arrangement S_P^+ form an odd permutation (it is not known at this point that this case is unsolvable actually; nevertheless the upcoming argumentation will provide reasons). Consider that the unoccupied vertex has some fixed location in the input graph G (for example, let it be the first connection vertex u^1 of the first handle H_1). Any movement of pebbles that preserves the unoccupied vertex must look like a shift along a cycle starting and ending in the unoccupied vertex [8]. Since there is no cycle of the odd length in the graph, this path must be of the even length. Hence, the difference between the original arrangement and the arrangement after this movement is an even permutation. Thus, if the input initial arrangement S_P^0 and the goal arrangement S_P^+ differ as an odd permutation, then the current arrangement S_P after placing pebbles into handles H_d, H_{d-1}, …, H_2 and the goal arrangement S_P^+ differ as an odd permutation as well (supposed that unoccupied vertex is fixed). This difference must be caused by pebbles in C_0 and H_1 only since other pebbles has been already placed at this moment. Hence, the initial arrangement S_θ^0 and the goal arrangement S_θ^+ over the θ-like graph θ formed by C_0 and H_1 submitted to *BIBOX-θ-Solve* make an odd permutation. Since the θ-like graph θ over C_0 and H_1 has no odd cycle, the algorithm correctly answers that there is no solution (line **14** of *θ-BOX-Solve*).

All the other cases are solvable and the algorithm provides a correct solution for them. Finally, it is necessary to investigate the termination of the algorithm from the opposite side. That is, if the algorithm terminates with the negative answer then there is no solution of the input instance.

Termination with a negative answer at line **5** of *θ-BOX-Solve* is possible only if the θ-like graph θ submitted to *θ-BOX-Solve* is isomorphic to $\theta(2,3,2)$ and the goal arrangement is unreachable from the initial one. This happens only if the input graph G is isomorphic to $\theta(2,3,2)$ and the goal arrangement is unreachable.

Similarly, termination at line **15** of *θ-BOX-Solve* is possible only if the θ-like graph θ submitted to *θ-BOX-Solve* has no cycle of the odd length and initial and the goal arrangements over θ form an odd permutation. The construction of the handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ (lines **1-6** of *BIBOX-θ-Solve*) ensures that there was no cycle of the odd length in the input graph G. The input instance cannot be solvable since otherwise there should be an odd cycle, which is a contradiction.

Altogether, the algorithm provides a solution (which is correct) if and only if the input instance of the pebble motion problem $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ is solvable. ■

Following propositions summarize estimations of the time and space complexity, and the makespan of the solution produced by the *BIBOX-θ* algorithm.

Proposition 11 ***(BIBOX-θ – worst case time complexity).*** The worst case **time complexity** of the *BIBOX-θ* algorithm is $\mathcal{O}(|V|^4)$ with respect to an input instance of the problem pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. ■

Proof. The time required to find a handle decomposition $\mathcal{D}' = [C_0', H_1', H_2', \dots, H_d']$ of G where the initial cycle C_0' has the odd length is $\mathcal{O}(|V| + |E|)$ in the worst case [24]. The handle decomposition $\mathcal{D}'$ must be subsequently augmented to another handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \dots, H_d]$ where $C_0 \circ H_1$ is not isomorphic to $\theta(2,3,2)$. This augmentation is done according to Lemma 5 and it takes time of $\mathcal{O}(|C_0'| + |H_1'| + |H_2'|)$ in the worst case which is $\mathcal{O}(|V|)$ - modification of the initial cycle and first two handles is done which can be done by traversing through its vertices according to the original handle decomposition $\mathcal{D}'$.

Transformation of the goal arrangement for relocating the unoccupied vertex to the first connection vertex of the first handle H_1 can be done in the worst case time of $\mathcal{O}(|V| + |E|)$ - a path connecting the original and the target position of the unoccupied vertex must be found.

Placing pebbles into handles $H_2, H_3, \dots, H_d$ requires time of $\mathcal{O}(|V|^3)$ as it has been already shown for the *BIBOX* algorithm. Altogether, the mentioned processing consumes time of $\mathcal{O}(|V|^3)$ in the worst case. Thus, it remains to investigate the solving process over the θ-like graph formed by C_0 and H_1.

In the worst case, it is necessary to compose $\mathcal{O}(|C_0| + |H_1|)$ optimal solutions to transposition of 3-cycle rotation cases to construct the overall solution of an instance over the θ-like graph. It is known that the makespan of any optimal solution of a pebble motion instance over a θ-like graph $\theta(A, B, C) = (V_\theta, E_\theta)$ with single unoccupied vertex is $\mathcal{O}(|V_\theta|^3)$ [8]. Hence, each optimal solution to the special case is of the size $\mathcal{O}((|C_0| + |H_1|)^3)$. As solutions to special cases are directly executed to update the current arrangement of pebbles, the overall required time is $\mathcal{O}((|C_0| + |H_1|)^4)$ which is $\mathcal{O}(|V|^4)$.

The worst case time complexity of the *BIBOX-θ* algorithm is thus $\mathcal{O}(|V|^3) + \mathcal{O}(|V|^4)$ which is $\mathcal{O}(|V|^4)$. ■

Proposition 12 ***(BIBOX-θ – makespan of the solution).*** The **makespan** of a solution in the worst case produced by the *BIBOX-θ* algorithm (that is, the number ξ) for an input instance of the problem of pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ is $\mathcal{O}(|V|^4)$. ■

Proof. Since it is assumed that there is single unoccupied vertex the makespan is exactly the same as the total number of moves within the solution. The contribution of the process of placing pebbles into handles of the handle decomposition in the *BIBOX*-style to the overall solution makespan is $\mathcal{O}(|V|^3)$.

A solution of the instance over θ-like graph formed by C_0 and H_1 is composed of $\mathcal{O}(|C_0| + |H_1|)$ optimal solutions to transposition and 3-cycle rotation cases. Each of the special cases requires makespan of $\mathcal{O}((|C_0| + |H_1|)^3)$. Thus, the makespan required to solve an instance over the θ-like graph is $\mathcal{O}((|C_0| + |H_1|)^4)$ which is $\mathcal{O}(|V|^4)$.

Finally, the transformation that relocates the unoccupied vertex to the position given by the original goal arrangement requires $\mathcal{O}(|V|)$ moves.

Altogether, the *BIBOX-θ* algorithm produces a solution to the given instance of pebble motion on graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ of the makespan of $\mathcal{O}(|V|^4)$. ■

Proposition 13 ***(BIBOX-θ – worst case space complexity).*** The worst case **space complexity** of the *BIBOX-θ* algorithm is $\mathcal{O}(|V| + |E|)$ with respect to an input instance of the problem pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, S_P^+)$ when space of the solution database is not accounted. ■

Proof. Since the *BIBOX* algorithm is used within *BIBOX-θ*, a space of $\mathcal{O}(|V| + |E|)$ is required at least (Proposition 5). As the solution database is not accounted, the algorithm does not require any additional space. ■

Proposition 14 ***(solution database size).*** 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 $table_{232}^{\theta}$). The space required by the part of the database where solutions to transposition and 3-cycle rotation cases over a θ-like graph $\theta(A, B, C) = (V_\theta, E_\theta)$ 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. ■

Proof. Since there is just single unoccupied vertex within all the instances which solutions are stored, the size of stored solutions corresponds exactly to the number of moves of which the solution consists. Hence, the size of any optimal solution to a special case over the θ-like graph $\theta(A, B, C) = (V_\theta, E_\theta)$ is $\mathcal{O}(|V_\theta|^3)$ [8].

It is necessary to store at most $((2 + 3 + 2) - 1)! = 720$ optimal solutions for the case of $\theta(2,3,2)$. Size of each optimal solution for this case is at most $7^3 = 343$. In total, the space of the size of $720 * 343 = 246960$, which is $\mathcal{O}(1)$, is necessary to store solution to all the solvable instances over $\theta(2,3,2)$.

There is $(|V_\theta| - 1)(|V_\theta| - 2)/2$ **transposition cases** over the θ-like graph θ; asymptotically this is $\mathcal{O}(|V_\theta|^2)$ of transposition cases. Thus, the space of $\mathcal{O}(|V_\theta|^5)$ is required to store optimal solutions to transposition cases into the database.

Analogically, there is $(|V_\theta| - 1)(|V_\theta| - 2)(|V_\theta| - 3)/6$ **3-cycle rotation cases** over the θ-like graph θ which is $\mathcal{O}(|V_\theta|^3)$. Hence, the space of $\mathcal{O}(|V_\theta|^6)$ is required to store optimal solutions to 3-cycle rotation cases into the database. ■

3.3. *Related Algorithms for Solving Pebble Motion Problems*

Both presented algorithms *BIBOX* and *BIBOX-θ* are designed for the relatively extreme class of problems of pebble motion on a graph and multi-robot path planning. They both suppose a small number of unoccupied vertices in the input graph.

On the other hand, if there is lot of unoccupied vertices in the graph of the input instance, then it might be disadvantageous to use these algorithms. For example with constant number of pebbles, it is more efficient to find a path to the goal vertex for each pebble independently by some path finding algorithm and to resolve eventual collisions than to use *BIBOX* algorithms. However, no threshold for the ratio of the number of pebbles to the number of unoccupied vertices above which one or the other approach is more advantageous is known. The mentioned approach where paths are searched for individual pebbles or robots independently has been actually studied in [25, 26, 27, 28]. Authors define a tractable class of the multi-robot path planning problem where non-colliding paths for individual robots exist. An instance has a chance to fall into this tractable class if there is lot of unoccupied vertices in the input graph since otherwise there is almost no possibility to have non-colliding paths for each robot.

4. Improving the Makespan of Solutions

This section is devoted to techniques for improving the makespan of solutions generated by proposed algorithms. Two techniques are described: a relaxation of special cases which optimal solutions are used to compose the overall sub-optimal solution and a technique for increasing parallelism based on *the critical path method* [11].

4.1. *Using Weak Special Cases*

It is possible to improve the course of execution of the *BIBOX-θ* algorithm when the θ-like graph formed by the initial cycle and the first handle of the handle decomposition is solved. The algorithm uses optimal solutions to transposition and 3-cycle cases at this moment. Observe that except transposing a pair of pebbles or rotating a triple of pebbles all the other pebbles within these special cases must preserve their positions. This is relatively tight constraint. In fact, it is sufficient not to relocate only those pebbles, which has been already placed to their goal vertices (these are generally not all the vertices of the θ-like graph except the last transposition or 3-cycle rotation). Other pebbles can be arranged arbitrarily, which significantly relaxes constraints on the stored optimal solutions. The stored optimal solutions thus may have smaller makespan and may be easier to compute consequently.

The above idea conjecture led to the concept of so called *weak special cases*. These special cases are represented by *weak transposition case* and *weak 3-cycle rotation case*. These concepts were introduced first in [18]. Both concepts will be briefly recalled in this section. A more detailed analysis of the benefit of using weak special cases instead of standard ones within the *BIBOX-θ* algorithm will be made in the experimental section (see Section 5).

The definition of weak special cases of the problem of pebble motion on a graph requires a slightly **generalized** notion of the goal arrangement. Instead of just one goal arrangement a set of goal arrangements will be considered. Problem of both pebble motion on a graph as well as of multi-robot path planning can be generalized naturally with

respect to the set of goal arrangements. The set of goal arrangements of pebbles will be denoted as $\mathbb{S}_P^+$ and the set of goal arrangements of robots will be denoted as $\mathbb{S}_R^+$.

A solution to an instance of the generalized problem of pebble motion on a graph $\Pi = (G = (V, E), P, S_P^0, \mathbb{S}_P^+)$ is solution of any standard instance $(G = (V, E), P, S_P^0, S_P^+)$ where $S_P^+ \in \mathbb{S}_P^+$. Similarly for the generalized problem of multi-robot path planning: a solution to an instance of the generalized problem of multi-robot path planning $\Sigma = (G = (V, E), P, S_R^0, \mathbb{S}_R^+)$ is solution of any standard instance $(G = (V, E), P, S_R^0, S_R^+)$ where $S_R^+ \in \mathbb{S}_R^+$.

The weak version of a special case which solution is to be stored with the set of pebbles P will be defined with respect to a subset of pebbles $P^\perp \subseteq P$ that must their positions (that is, pebbles from $P^\perp$ can move along the execution of the solution, however they must return to their original position eventually). Following definitions suppose a θ-like graph $\theta(A, B, C) = (V_\theta, E_\theta)$ with $A = [\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_\alpha]$, $B = [\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_\beta]$, and $C = [\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_\gamma]$ and a set of pebbles $P = \{p_1, p_2, \ldots, p_{|V_\theta|-1}\}$.

Definition 9 ***(weak transposition case).*** Let S_P^0 be an initial arrangement of the set of pebbles such that $S_P^0(p) \neq \bar{b}_1 \ \forall p \in P$ (that is, $\bar{b}_1$ is initially unoccupied). Let $P^\perp \subseteq P$ be a set of vertices and let $q_1, q_2 \in P$ be a pair of pebbles such that $q_1 \neq q_2$ and $\{q_1, q_2\} \cap P^\perp = \emptyset$. Let $\mathbb{S}_P^+$ be a set of goal arrangements of pebbles such that $S_P^+ \in \mathbb{S}_P^+$ if $S_P^+(q_1) = S_P^0(q_2) \wedge S_P^+(q_2) = S_P^0(q_1) \wedge (\forall p \in P^\perp) S_P^+(p) = S_P^0(p) \wedge (\forall p \in P)(S_P^+(p) \neq \bar{b}_1)$ (pebbles q_1 and q_2 are to be exchanged; positions of pebbles from $P^\perp$ are preserved; other pebbles can be relocated arbitrarily if $\bar{b}_1$ is avoided). Then an instance of the generalized problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, \mathbb{S}_P^+)$ is called a *weak transposition case* with respect to q_1, q_2, and $P^\perp$. □

Definition 10 ***(weak 3-cycle rotation case).*** Let S_P^0 be an initial arrangement of the set of pebbles such that $S_P^0(p) \neq \bar{b}_1 \ \forall p \in P$ ($\bar{b}_1$ is initially unoccupied). Let $P^\perp \subseteq P$ be a set of vertices and let $q_1, q_2, q_3 \in P$ be a triple of wise distinct pebbles such that $\{q_1, q_2, q_3\} \cap P^\perp = \emptyset$. Let $\mathbb{S}_P^+$ be a set of goal arrangement of pebbles such that $S_P^+ \in \mathbb{S}_P^+$ if $S_P^+(q_1) = S_P^0(q_2) \wedge S_P^+(q_2) = S_P^0(q_3) \wedge S_P^+(q_3) = S_P^0(q_1) \wedge (\forall p \in P^\perp) S_P^+(p) = S_P^0(p) \wedge (\forall p \in P)(S_P^+(p) \neq \bar{b}_1)$ (pebbles q_1, q_2, and q_3 are to be rotated; positions of pebbles from $P^\perp$ are preserved; other pebbles can be relocated arbitrarily if $\bar{b}_1$ is avoided). Then an instance of the generalized problem of pebble motion on a graph $\Pi = (\theta = (V_\theta, E_\theta), P, S_P^0, \mathbb{S}_P^+)$ is called a *weak 3-cycle rotation case* with respect to q_1, q_2, q_3, and $P^\perp$. □

Weak special cases have the additional parameter $P^\perp$ that can be assigned $2^{|P|-2}$ (transposition case) or $2^{|P|-3}$ (3-cycle rotation case) different values. Hence, it is not possible to store optimal solutions to all the possible weak special cases over θ-like graphs up to the certain size as in the case of the standard special cases.

Nevertheless, it is possible to select a small number of subsets of pebbles (linear of quadratic number) and to store optimal solutions for them. When a transposition or a

3-cycle rotation needs to be performed while pebbles from a set P' need to preserve their positions, a corresponding special case with the smallest makespan satisfying that $P' \subseteq P^{\perp}$ is used ($P^{\perp}$ is one of the parameters determining the used special case).

Observe, that the makespan of the weak special case is less than or equal to the makespan of the corresponding standard special case. Moreover, if there are multiple candidate weak special cases to be used, the most promising one is used. Hence, it is reasonable to expect that the use of weak special cases will be beneficial.

Let $\theta = (V_\theta, E_\theta)$ be a θ-like graph and let $[\sigma_1, \sigma_2, \dots, \sigma_{|V_\theta|-1}]$ be a sequence representing the ordering of vertices V_θ in which they are traversed by the *BIBOX-*θ algorithm in the last phase of the execution. Practically, it is suitable to store optimal solutions to all the weak special cases with the parameter $P^{\perp}$ ranging over all the subsets of pebbles $\bigcup[\sigma_1, \sigma_2, \dots, \sigma_i]$ for $i = 1,2, \dots, |V_\theta| - 1$ (the union notation is used to create a set from range of a sequence).

This approach is conservative with respect to memory consumption: there is **linear** increase in the space required by the database in comparison with the database containing solutions of standard special cases. The space required for storing optimal solutions of all the selected weak transposition cases over $\theta = (V_\theta, E_\theta)$ is $\mathcal{O}(|V_\theta|^6)$. Analogically, the space required to store optimal solutions of the selected weak 3-cycle rotation cases is $\mathcal{O}(|V_\theta|^7)$.

Notice, however that practically the required space may be smaller since optimal solutions of weak special cases itself are smaller than solutions of standard special case (weaker condition of the solution is required – see experimental Section 5).

4.2. *Increasing Parallelism*

This section is devoted to a method for increasing parallelism of solutions. In fact, this method represents a major technique how to utilize parallelism allowed by the definition of the problem of multi-robot path planning. Both presented algorithms - *BIBOX* as well as *BIBOX-*θ - does not utilize the possibility of parallel movements. They solve the problem of pebble motion on a graph in fact. The method presented below in intended as a post-processing technique that should be applied on a solution produced by *BIBOX* algorithms.

Definition 11 ***(sequential solution).*** A solution $\mathcal{S}_R(\Sigma) = [S_R^0, S_R^1, \dots, S_R^\zeta]$ of multi-robot path planning problem $\Sigma = (G = (V, E), R = \{\bar{r}_1, \bar{r}_2, \dots, \bar{r}_\nu\}, S_R^0, S_R^+)$ is called *sequential* (ζ is the length of the solution) if for each $k = 1,2, \dots, \zeta - 1$ there exists $j \in \{1,2, \dots, \nu\}$ such that $S_R^k(\bar{r}_j) \neq S_R^{k+1}(\bar{r}_j)$ and $S_R^k(\bar{r}_l) \neq S_R^{k+1}(\bar{r}_l)$ for each $l = 1,2, \dots, \nu \wedge l \neq j$ (at time step k a robot r_j is moved; all the other robots do not move at the time step). □

A move of a robot r from a vertex u to a vertex v will be denoted using the notation $r: u \to v$. The sequential solution of multi-robot path planning problem can be equivalently represented as a sequence of moves of the form $r: u \to v$. That is, a solution is a sequence $\mathcal{S}_R^{\prec}(\Sigma) = [r_1: u_1 \to v_1, r_2: u_2 \to v_2, \dots, r_\zeta: u_\zeta \to v_\zeta]$ (r_i are variables with the

domain $\{\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_\nu\}$; $\bar{r}_i$ are constants). Notice, that $u_k \neq v_k$ for each $k = 1,2,\ldots,\zeta$ which is ensured by the definition of sequential solution. In other words, a solution is sequential if there is just one move at each step. This, however, may prolong makespan significantly, which is not desirable.

Suppose a sequential solution $\mathcal{S}_R^{\prec}(\Sigma) = [r_1: u_1 \rightarrow v_1, r_2: u_2 \rightarrow v_2, \ldots, r_\zeta: u_\zeta \rightarrow v_\zeta]$ of an instance of multi-robot path planning $\Sigma = (G = (V, E), R = \{\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_\nu\}, S_R^0, S_R^+)$. This form of the solution of the problem is more convenient for reasoning about the possible parallelism. The following definitions refer to the sequence of moves $\mathcal{S}_R^{\prec}(\Sigma)$.

Definition 12 ***(interfering moves).*** A move $r_h: u_h \rightarrow v_h;\ h \in \{1,2,\ldots,\zeta - 1\}$ is *interfering* with a move $r_k: u_k \rightarrow v_k;\ k \in \{1,2,\ldots,\zeta - 1\}$ if $|\{u_h, v_h\} \cap \{u_k, v_k\}| \geq 1$. □

Typically, interfering moves cannot be executed in parallel. However, the situation is not so straightforward. Following definitions are trying to capture which pairs of interfering moves can be undoubtedly executed in parallel and which not.

Definition 13 ***(potentially concurrent moves).*** A move $r_k: u_k \rightarrow v_k;\ k \in \{1,2,\ldots,\zeta\}$ is *potentially concurrent* with a move $r_h: u_h \rightarrow v_h;\ h \in \{1,2,\ldots,\zeta\}$ with $h < k$ if $r_h \neq r_k$, $u_h = v_k \wedge v_h \neq u_k$, and there is no other move $r_\hbar: u_\hbar \rightarrow v_\hbar$ in $\mathcal{S}_R^{\prec}(\Sigma)$ such that $h < \hbar < k$ interfering with $r_h: u_h \rightarrow v_h$ or $r_k: u_k \rightarrow v_k$. The notation is that $r_h: u_h \rightarrow v_h \preccurlyeq r_k: u_k \rightarrow v_k$. □

The definition captures the fact that although the moves are interfering they can be executed at the same time step according to the definition of a solution of an instance of multi-robot path planning problem. The relation of potential concurrence is anti-reflexive due to the requirement on different robots involved ($r_h \neq r_k$) and anti-symmetric due to the ordering of moves within the sequential solution ($h < k$).

Definition 14 ***(trivially dependent moves).*** A move $r_k: u_k \rightarrow v_k;\ k \in \{1,2,\ldots,\zeta\}$ is *trivially dependent* on a move $r_h: u_h \rightarrow v_h;\ h \in \{1,2,\ldots,\zeta\}$ with $h < k$ if these moves are interfering, $r_h = r_k$ or $u_h \neq v_k \vee v_h = u_k$, and there is no other move $r_\hbar: u_\hbar \rightarrow v_\hbar$ in $\mathcal{S}_R^{\prec}(\Sigma)$ such that $h < \hbar < k$ interfering with $r_h: u_h \rightarrow v_h$ or $r_k: u_k \rightarrow v_k$. The notation is that $r_h: u_h \rightarrow v_h \prec r_k: u_k \rightarrow v_k$. □

The definition captures the fact that trivially dependent moves cannot be executed at the same time step. Notice, that the condition $r_h = r_k$ or $u_h \neq v_k \vee v_h = u_k$ is the negation of the condition $r_h \neq r_k$ and $u_h = v_k \wedge v_h \neq u_k$ from the definition of the potential concurrence. Observe that, when $|\{u_h, v_h\} \cap \{u_k, v_k\}| \geq 1$ (interfering moves), the condition $u_h \neq v_k \vee v_h = u_k$ can be equivalently expressed as a disjunction of several cases as follows: $(u_h \neq u_k \wedge v_h = v_k)$ or $(u_h = u_k \wedge v_h \neq v_k)$ or $(u_h = u_k \wedge v_h = v_k)$ or $(u_h = v_k \wedge v_h = u_k)$ or $(u_h \neq v_k \wedge v_h = u_k)$ (original and target vertices of each move

are different; thus, each of the conjunctions defines the situation unambiguously with respect to involved vertices). Observe, that none of the cases is actually possible if $r_h = r_k$ and with no middle move $r_\hbar: u_\hbar \to v_\hbar$ allowed. The relation of trivial dependence of moves is reflexive and anti-symmetric due to the ordering of moves within the sequential solution ($h < k$).

The notions of potential concurrence and trivial dependence are to be used as building blocks of a process that constructs parallel solution of the instance of the problem of multi-robot path planning.

Proposition 15 ***(execution order).*** Let each move of a sequential solution $\mathcal{S}_R^{\prec}(\Sigma)$ is assigned a time step of its execution by a function $t: \bigcup \mathcal{S}_R^{\prec}(\Sigma) \longrightarrow \{1,2,\dots,\zeta-1\}$. Let t satisfies that if $r_h: u_h \to v_h \prec r_k: u_k \to v_k$ then $t(r_h: u_h \to v_h) < t(r_k: u_k \to v_k)$ and if $r_h: u_h \to v_h \preccurlyeq r_k: u_k \to v_k$ then $t(r_h: u_h \to v_h) \leq t(r_k: u_k \to v_k)$. Then a standard (parallel) solution $\mathcal{S}_R(\Sigma)$ constructed from $\mathcal{S}_R^{\prec}(\Sigma)$ using the function t forms a (correct) solution of Σ (sequence of arrangements of robots in $\mathcal{S}_R(\Sigma)$ reflects changes induced by moves at time steps determined by the function t). ■

Proof. The proof will proceed by induction according to the length of the sequential solution $\mathcal{S}_R^{\prec}(\Sigma)$. If the sequence $\mathcal{S}_R^{\prec}(\Sigma)$ consists of a single element, the proposition holds. Suppose that $\mathcal{S}_R^{\prec}(\Sigma) = [r_1: u_1 \to v_1, r_2: u_2 \to v_2, \dots, r_\zeta: u_\zeta \to v_\zeta]$ is of non-trivial length. From induction hypothesis, the proposition holds for the sequence of moves $\mathcal{S}_R^{\prec}(\Sigma') = [r_1: u_1 \to v_1, r_2: u_2 \to v_2, \dots, r_{\zeta-1}: u_{\zeta-1} \to v_{\zeta-1}]$. In other words, there is a function $t': \bigcup \mathcal{S}_R^{\prec}(\Sigma') \longrightarrow \{1,2,\dots,\zeta-1\}$ such that it determines a correct parallel solution of an instance Σ' which is almost the same as Σ except the goal arrangement which differs by the last move $r_{\zeta-1}: u_{\zeta-1} \to v_{\zeta-1}$.

If there is some move $r_i: u_i \to v_i$ with $1 \leq i \leq \zeta - 1$ such that $r_\zeta: u_\zeta \to v_\zeta$ is trivially dependent on it, then t should satisfy that $t(r_i: u_i \to v_i) < t(r_\zeta: u_\zeta \to v_\zeta)$. Properties of trivial dependency ensure that execution of $r_\zeta: u_\zeta \to v_\zeta$ after $r_i: u_i \to v_i$ does not violate correctness of the solution.

If there is some move $r_j: u_j \to v_j$ with $1 \leq j \leq \zeta - 1$ such that $r_\zeta: u_\zeta \to v_\zeta$ is potentially concurrent with it, then should satisfy that $t(r_j: u_j \to v_j) \leq t(r_\zeta: u_\zeta \to v_\zeta)$. The relation of potential concurrence ensures that execution of $r_\zeta: u_\zeta \to v_\zeta$ at the same time step or after the time step with $r_i: u_i \to v_i$ does not violate correctness of the solution.

Let $t(r_h: u_h \to v_h) = t'(r_h: u_h \to v_h)$ for $h = 1,2,\dots,\zeta-2$. The function will be defined for the last element of $\mathcal{S}_R^{\prec}(\Sigma)$ specially to satisfy above inequalities with respect to all the trivially dependent and potentially concurrent moves with respect to $r_\zeta: u_\zeta \to v_\zeta$. Let $t'_{\prec} = \max\{t'(r_i: u_i \to v_i) \mid r_i: u_i \to v_i \prec r_\zeta: u_\zeta \to v_\zeta \wedge i \in \{1,2,\dots,\zeta-1\}\}$ be the time step assigned to the last trivially dependent move. Similarly, let $t'_{\preccurlyeq} = \max\{t'(r_j: u_j \to v_j) \mid r_j: u_j \to v_j \preccurlyeq r_\zeta: u_\zeta \to v_\zeta \wedge j \in \{1,2,\dots,\zeta-1\}\}$ be the time step assigned to the last potentially concurrent move. Let $t(r_\zeta: u_\zeta \to v_\zeta) \geq \max\{t'_{\prec} + 1, t'_{\preccurlyeq}\}$. The function t defined as above satisfies the proposition. ■

The parallelized solution will be constructed according to Proposition 15. To obtain small makespan and high parallelism of the solution, low execution times for execution should be assigned to the individual moves. Thus, it is recommended to assign the time step for the execution of the newly added move in the proposition as follows: $t(r_\zeta: u_\zeta \rightarrow v_\zeta) = \max\{t'_\prec + 1, t'_\preccurlyeq\}$.

The process is formalized in pseudo-code as Algorithm 3. The method described above is also known as *critical path method* in different contexts [11].

The algorithm consists of three functions: *Increase-Parallelism*, *Earliest-Execution-Time* $^\prec$, and *Earliest-Execution-Time* $^\preccurlyeq$.

The main framework of the algorithm is represented by the function *Increase-Parallelism*. The function successively includes moves into the constructed parallel solution while trivial dependency and potential concurrence with respect to already included moves is calculated. The function is build over the array *step* which is indexed by time steps. The cell *step*$[t]$ contains a set of moves that are to be executed at the time step t.

Functions *Earliest-Execution-Time* $^\prec$ and *Earliest-Execution-Time* $^\preccurlyeq$ calculates earliest execution time for the newly included move with respect to already included trivially dependent moves and potentially concurrent moves.

Proposition 16 ***(increasing parallelism).*** The algorithm for increasing parallelism has the worst case time complexity of $\mathcal{O}(|\mathcal{S}_R^\prec(\Sigma)|^2)$ for the input sequential solution $\mathcal{S}_R^\prec(\Sigma) = [r_1: u_1 \rightarrow v_1, r_2: u_2 \rightarrow v_2, \ldots, r_\zeta: u_\zeta \rightarrow v_\zeta]$. The worst case space complexity of the algorithm is $\mathcal{O}(|\mathcal{S}_R^\prec(\Sigma)|)$. ■

Proof. Each call of *Earliest-Execution-Time* $^\prec$ and *Earliest-Execution-Time* $^\preccurlyeq$ requires time of $\mathcal{O}(|\mathcal{S}_R^\prec(\Sigma)|)$. Both function are called $|\mathcal{S}_R^\prec(\Sigma)|$ times, thus the overall worst case time complexity is $\mathcal{O}(|\mathcal{S}_R^\prec(\Sigma)|^2)$.

A space of the size $\mathcal{O}(|\mathcal{S}_R^\prec(\Sigma)|)$ is required to store the input sequential solution $\mathcal{S}_R^\prec(\Sigma)$. A space of the same size is necessary for storing the array *step*. ■

Observe that the pseudo-code of the algorithm exploits an opportunistic search for the dependent move with highest execution time that has been already scheduled. Moves with already assigned time steps are traversed from the last one according to the time of the execution. When a dependent move is encountered, the search terminates. Since the probability of encountering a dependent move is supposedly much higher in the real-life instance than having a completely independent move, this strategy is a good one.

Algorithm 3. *The **parallelism-increasing** algorithm.* The algorithm produces a parallelized solution of an instance of multi-robot path planning problem from the given sequential solution. The idea of the algorithm is inspired by the **critical path method** [11].

function *Increase-Parallelism*$(\mathcal{S}_R^{\prec}(\Sigma), S_R^0)$: **pair**
/* A function for producing standard solution of
multi-robot path planning problem instance from the
sequential one.
Parameters: $\mathcal{S}_R^{\prec}(\Sigma)$ - a sequential solution of Σ,
S_R^0 - a initial arrangement of robots. */
1: **let** $\mathcal{S}_R^{\prec}(\Sigma) = [r_1 : u_1 \to v_1, r_2 : u_2 \to v_2, \dots, r_\zeta : u_\zeta \to v_\zeta]$
2: $step[1] \leftarrow \{r_1 : u_1 \to v_1\}$
3: $\omega \leftarrow 1$
4: **for** $k = 2,3, \dots, \zeta - 1$ **do**
5: $\quad t'_{\prec} \leftarrow$ Earliest-Execution-Time $^{\prec}(r_k : u_k \to v_k)$
6: $\quad t'_{\preccurlyeq} \leftarrow$ Earliest-Execution-Time $^{\preccurlyeq}(r_k : u_k \to v_k)$
7: $\quad t \leftarrow \max\{t'_{\prec} + 1, t'_{\preccurlyeq}\}$
8: $\quad step[t] \leftarrow step[t] \cup \{r_1 : u_1 \to v_1\}$
9: $\quad \omega \leftarrow \max\{\omega, t\}$
10: $S_R \leftarrow S_R^0$
11: $\zeta \leftarrow 1$
12: **for** $i = 1,2, \dots, \omega$ **do**
13: $\quad$ **for each** $(r : u \to v) \in step[i]$ **do**
14: $\quad\quad S_R(r) \leftarrow v$
15: $\quad\quad S_R^i \leftarrow S_R$
16: **return** $(\omega, [S_R^0, S_R^1, \dots, S_R^\omega])$

function *Earliest-Execution-Time* $^{\prec}(r_k : u_k \to v_k, \omega)$: **integer**
/* Calculates earliest execution time for a given move
with respect to the relation of trivial dependency.
Parameters: $r_k : u_k \to v_k$ - a move for that a time step is calculated,
ω – currently last time step. */
1: **for** $i = \omega, \omega - 1, \dots, 1$ **do**
2: $\quad$ **for each** $(r : u \to v) \in step[i]$ **do**
3: $\quad\quad$ **if** $r : u \to v \prec r_k : u_k \to v_k$ **then**
4: $\quad\quad\quad$ **return** k
5: **return** 1

function *Earliest-Execution-Time* $^{\preccurlyeq}(r_k : u_k \to v_k, \omega)$: **integer**
/* Calculates earliest execution time for a given move
with respect to the relation of potential concurrence.
Parameters: $r_k : u_k \to v_k$ - a move for that a time step is calculated,
ω – currently last time step. */
1: **for** $i = \omega, \omega - 1, \dots, 1$ **do**
2: $\quad$ **for each** $(r : u \to v) \in step[i]$ **do**
3: $\quad\quad$ **if** $r : u \to v \preccurlyeq r_k : u_k \to v_k$ **then**
4: $\quad\quad\quad$ **return** k
5: **return** 1

5. Experimental Evaluation

This section is devoted to an experimental evaluation of algorithms *BIBOX* and *BIBOX-θ* that were designed in this work. The following experimental analysis is intended as justification of the development of new algorithms as well as practical confirmation of theoretical properties shown in the experimental analysis. As algorithms *BIBOX* and *BIBOX-θ* were primarily developed as an alternative to the *MIT* [8] algorithm, the experimental evaluation will be aimed on the competitive comparison of *BIBOX* and *BIBOX-θ* with MIT.

All the tested algorithms were implemented in C++. The implementation of algorithms *BIBOX* and *BIBOX-θ* follows the pseudo-code of Algorithm 1 and Algorithm 2 respectively. Several optimizations mentioned in Section 3.1.4 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 [17]. Details of this algorithm are out of scope of this study. Pseudo-code and experimental analysis can be found in [17]. However, it is a time consuming task to find an optimal solution of a multi-robot path planning instance even on a small θ-like graph. Therefore, the timeout of 8.0 seconds is used after that the solving process switches to the *MIT* style solving of the instance over the θ-like graph (it is based on *4-transitivity* and it produces a sub-optimal solution). Instances where the solving process managed to compute a required optimal solution in the given time out are presented only when it could distort results (only scalability tests show the on-line computation of optimal solutions). Notice, that the database should with optimal solutions to special cases should be pre-computed off-line in the real-life application of the *BIBOX-θ* algorithm.

The *MIT* has been re-implemented according to [8]. A similar optimization technique as in the case of the *BIBOX* algorithm has been used. When an unoccupied vertex was necessary, the nearest unoccupied vertex was found and transferred to the place where needed. More details about the re-implementation of the *MIT* algorithm can be found in [19].

In order to allow reproducibility of the presented results all the source code is provided at the web page: http://ktiml.mff.cuni.cz/~surynek/research/jair2010. Additional experimental results and raw experimental data are provided at the same location.

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].

[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.

5.1. ***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 parallelism-increasing algorithm (see Algorithm 3). Thus, the result is a parallel solution complying with the definition of the solution of *multi-robot path planning problem*. A set of testing instances of multi-robot path planning problem 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 some 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 robots over the graph where at least the given number of vertices remains unoccupied. The handle decomposition used by solving algorithms was exactly that one used for generating the graph.

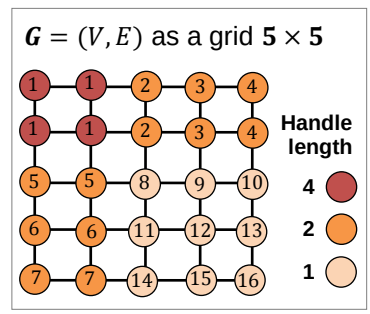


Figure 12. *An illustration of* ***handle decomposition of a grid*** *graph.* The ordering of the addition of individual handles is depicted by numbers in vertices. Three types of handles/cycles are used.

The situation with instances over the grid is similar. The **square grid graph** of a given size has been generated together with **random initial** and **goal** arrangement of robots over this graph. Again, a given number of vertices remain unoccupied. The handle decomposition for the grid graph used by solving algorithms consists of an initial cycle with 4 vertices (placed on the left upper corner of the grid); 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 12 for the ordering of addition of vertices in the construction of the grid.

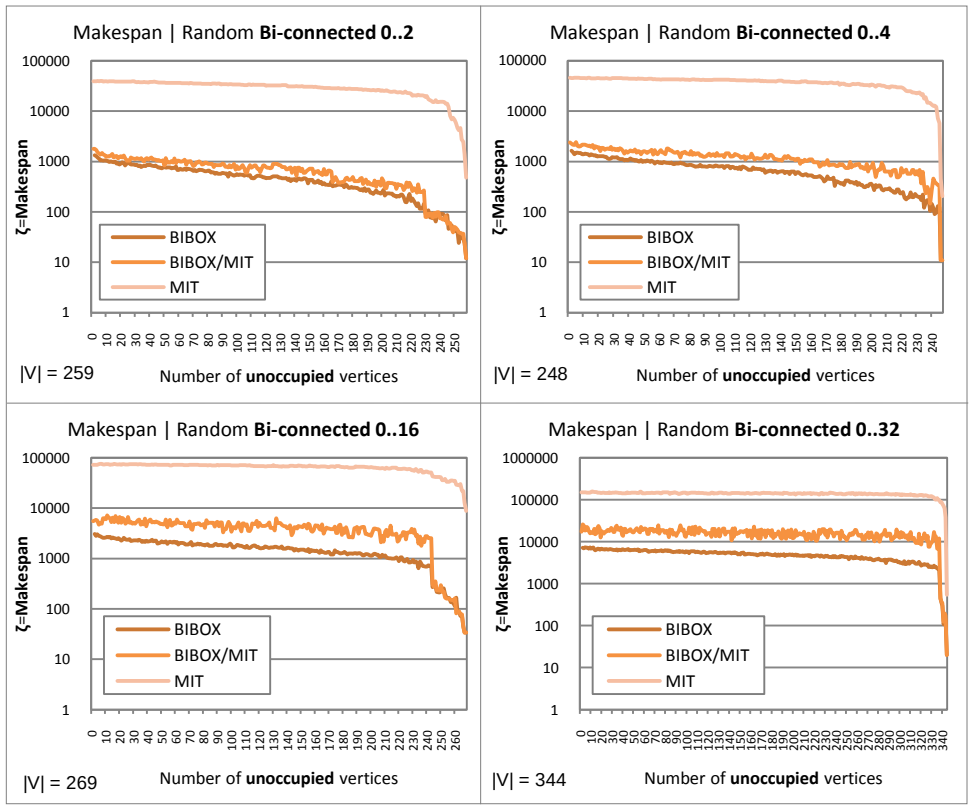


Figure 13. ***Makespan*** *comparison of solutions of instances over* ***random bi-connected*** *graphs.* Three algorithms are compared: the standard *BIBOX*, a variant of the *BIBOX* algorithm where the last phase when robots are placed into the θ-like graph formed by the initial cycle and the first handle – called *BIBOX/MIT*, and the *MIT* algorithm. Solutions were parallelized using parallelism-increasing algorithm (Algorithm 3). Four setups of generation of random bi-connected graphs has been 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 **dependence** of the makespan on the number of unoccupied vertices in the graph is shown.

In some experiments, the number of unoccupied vertices was the parameter that has been changing to observe dependency of the makespan on it while the graph of the instance remained fixed.

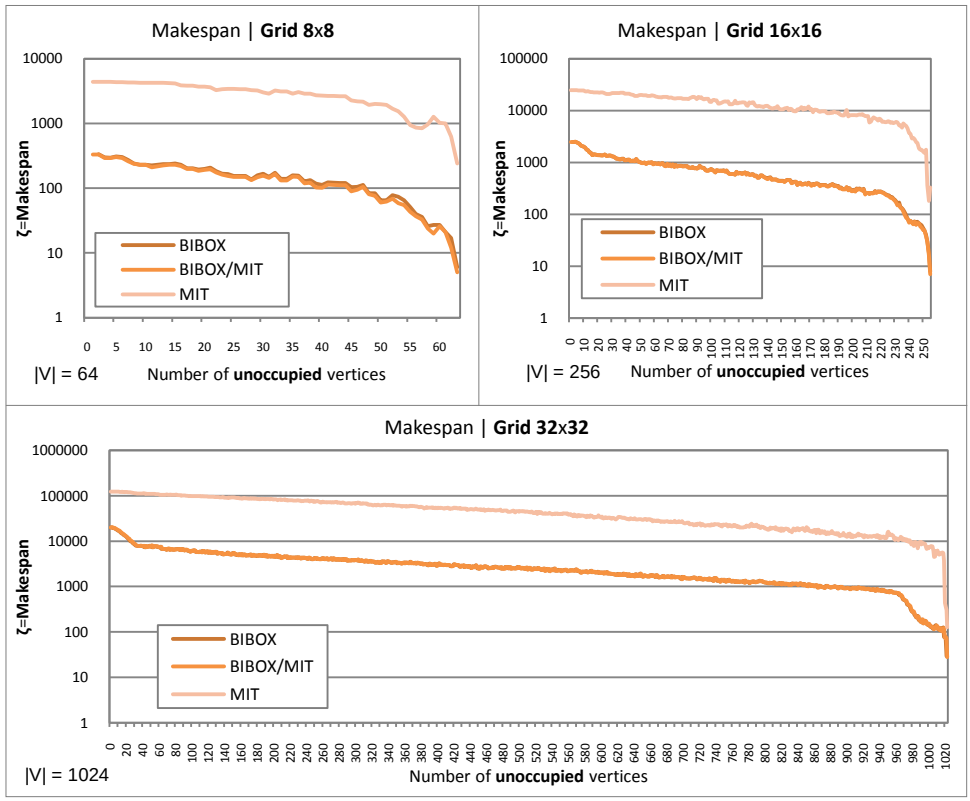


Figure 14. ***Makespan*** *comparison of solutions of instances over* ***square grids***. Three algorithms are compared: the standard *BIBOX*, *BIBOX/MIT*, and *MIT* on three grids: 8×8, 16×16, and 32×32. Solutions were parallelized using parallelism-increasing algorithm (Algorithm 3). The **dependence** of the makespan on the number of unoccupied vertices in the graph is shown.

Results shown in Figure 13 and Figure 14 are targeted on comparison of the **makespan** of solutions generated by tested algorithms. These results have been generated on the default configuration. Three algorithms were tested in this setup: the standard *BIBOX*, a variant of the *BIBOX* algorithm where the last phase when robots are placed into the θ-like graph formed by the initial cycle and the first handle – called *BIBOX/MIT*, and the *MIT* algorithm.

Results in Figure 13 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 differs in average length of initial cycle and handles of the handle decomposition were used. Lengths of the initial cycle and handles have uniform distribution of the range: $0..4$, $0..8$, $0..16$, and $0..32$ respectively (that is, four identical graphs are used). The length of the handle is equal to the number of its internal vertices. The dependence of the makespan on the number of unoccupied vertices in the graph is shown.

Figure 14 is devoted to structurally regular graphs – grid graphs of the size 8×8, 16×16, and 32×32 are used. Again, dependence of the makespan on the number of unoccupied vertices in the graph is shown.

Random initial and goal arrangements are obtained as a random permutation of robots 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 for the use in test.

It can be observed from the above tests 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. The steep decline of the makespan can be observed with the portion of occupied vertices from approximately 5% towards lower portion. This is some kind of a **phase transition** when robots 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 with 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 robots to the number of vertices characterizing the phase transition tends to be higher.

The *BIBOX/MIT* algorithm exhibits performance depending on the size of the initial θ-like graph of the handle decomposition. The larger is this θ-like graph the worse is the performance of the *BIBOX/MIT* algorithm with respect to the makespan of generated solutions. This behavior can be observed from the results shown in Figure 13 and Figure 14 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 (solutions produced by the *BIBOX/MIT* are slightly better – on very small cycles, solving process based on 4-transitivity used by *MIT* is more advantageous than exchanging robots applied by *BIBOX* when robot are placed into the initial cycle of the handle decomposition).

Regarding the makespan, the *BIBOX* style solving process represents the **better alternative** for solving the problem of multi-robot path planning with at least two unoccupied vertices than the *MIT* algorithm. For the case with one unoccupied vertex, the *BIBOX-θ* and its variants can be used. As it will be shown in the following paragraphs, *BIBOX-θ* exhibits the similar performance regarding the makespan as the standard *BIBOX* algorithm.

An interesting question is whether the use of optimal solutions of weak special cases instead of standard one does really help. The experimental evaluation which results are shown in Figure 15 is devoted to this question. Again, this test has been performed on the default configuration. A comparison of the *BIBOX* algorithm with the variants of the *BIBOX-θ* algorithm is shown consequently.

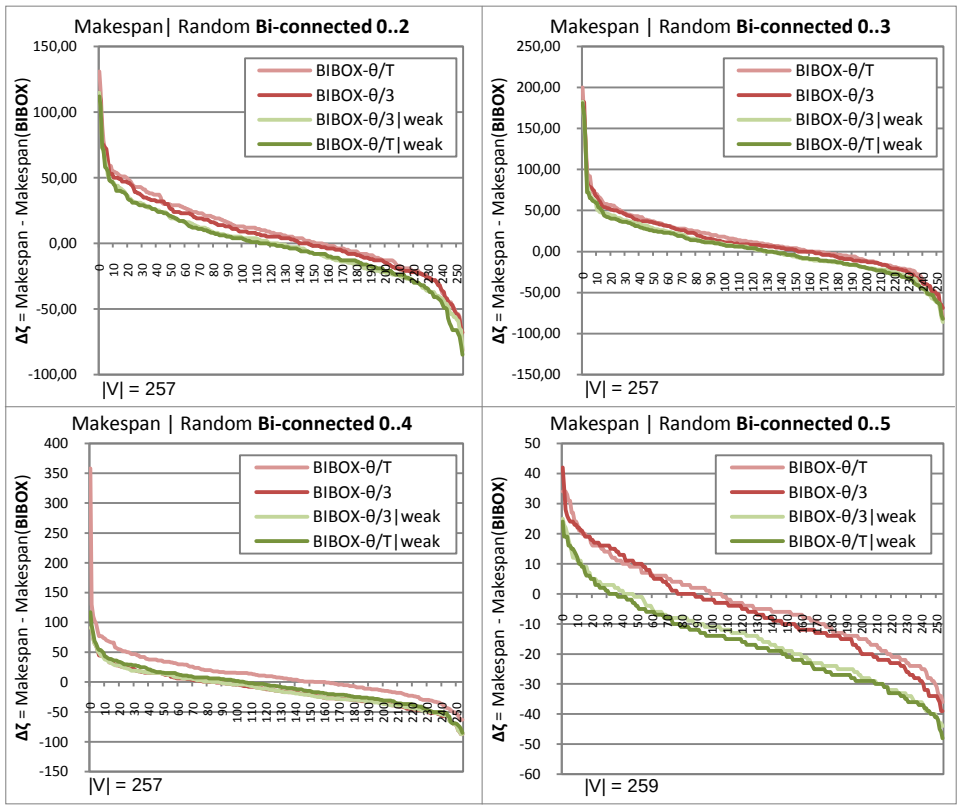


Figure 15. *An evaluation of the benefit of the use of* ***weak special cases*** *instead of standard ones.* Four variants of the *BIBOX-θ* algorithm are compared: *BIBOX-θ*/T (the standard transposition case is used preferably), *BIBOX-θ*/3(the standard 3-cycle rotation case is used preferably), *BIBOX-θ*/T|weak (the weak transposition case is used preferably), and *BIBOX-θ*/3|weak (the weak 3-cycle rotation case is used preferably). Solutions were parallelized using parallelism-increasing algorithm (Algorithm 3). 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 (thus, there is no interpretation of the horizontal axis).

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 2 prefers standard transposition cases. If the transposition case no 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 15 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 expectable that these makespans are sorted to the left or to the right margin.

Results shown in Figure 15 can be undoubtedly interpreted in the way that solutions with the smallest makespan are produced by *BIBOX-θ*/T|weak tightly followed by *BIBOX-θ*/3|weak. This is an expectable result and it is possible to conclude that the use of optimal solutions of **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. Another, interesting result that can be observed from Figure 15 concerns the makespan of solutions generated by the *BIBOX* algorithm in comparison to tested variants of *BIBOX-θ*. Since values of makespan differences are insignificantly deviate from equal distribution around 0, it is possible to conclude that variants of *BIBOX-θ* does not improve the makespan significantly in comparison with *BIBOX* on instances of the multi-robot path planning problem with at least two unoccupied vertices. Thus, the use of *BIBOX-θ* is substantiated only for instances with just single unoccupied vertex (where the *BIBOX* algorithm is not applicable).

5.2. ***Parallelism Evaluation***

The next series of experiments is devoted to an evaluation of *parallelism* of solutions generated by tested algorithm. 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 required as correlates with the small makespan. All the following experiments have been performed on the default configuration.

Again, three algorithms were tested: the standard *BIBOX*, *BIBOX/MIT*, and the *MIT* algorithm. Four random bi-connected graphs with various average length of handles has been 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 dependence of the parallelism on the number of unoccupied vertices has been evaluated. Random initial and

goal arrangements of robots in vertices have been used – for every number of unoccupied vertices a new random initial and goal arrangements have been generated. Results regarding this test are shown in Figure 16.

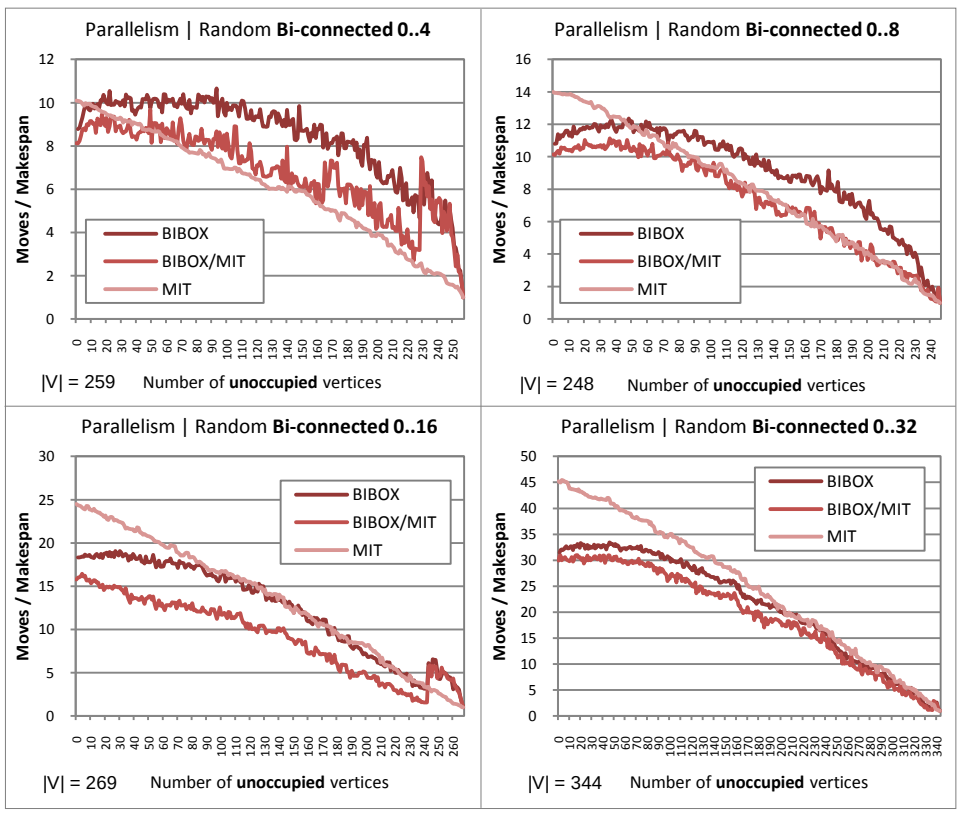


Figure 16. ***Average parallelism*** *of solutions generated by tested algorithms for instances over* ***random bi-connected*** *graphs*. Three algorithms are compared: the standard *BIBOX*, *BIBOX/MIT*, and the standard *MIT* algorithm. Solutions were parallelized using parallelism-increasing algorithm (Algorithm 3). Four random bi-connected graphs has been 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 **dependence** of the *average parallelism* on the number of unoccupied vertices in the graph is shown. The average parallelism is the number of moves, which the solution consists of, divided by the makespan.

A similar experiment has been done with the square grid graphs. Three grids were used in this setup: 8×8, 16×16, and 32×32. The dependence of parallelism on the number of unoccupied vertices has been measured. A new random initial and goal arrangement of robots has been used for every number of unoccupied vertices. Results regarding this experiment are shown in Figure 17.

In all the cases, the parallelism-increasing algorithm (Algorithm 3) has been used to post-process sequential solutions generated by tested algorithms.

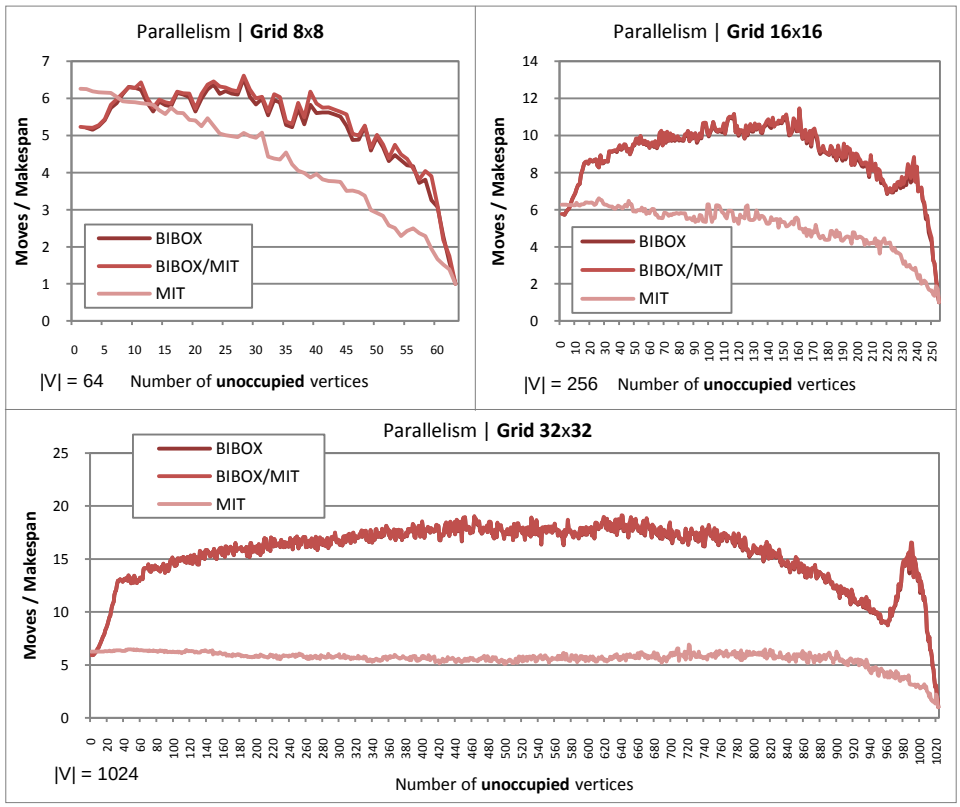


Figure 17. ***Average parallelism*** *comparison of solutions of instances over* ***square grids***. Three algorithms are compared: the standard *BIBOX*, *BIBOX/MIT*, and *MIT* on three grids: 8×8, 16×16, and 32×32. Again, solutions were parallelized using parallelism-increasing algorithm (Algorithm 3). The **dependence** of the average parallelism on the number of unoccupied vertices in the graph is shown.

It can be observed the *BIBOX* algorithm and its variant *BIBOX/MIT* exhibit relatively natural behavior regarding parallelism. On bi-connected graphs, the parallelism of solutions slightly increases as the number of unoccupied vertices increases in the almost fully occupied graph. 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 robots are in the graph the lower is the parallelism, which is quite natural. 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 robots in the handle are moving at once. Another characteristic, which the parallelism correlates with, is the *diameter* [29] of the graph. This correlation can be observed on tests with grid graphs in Figure 17. The reason for this correlation is the fact that all the robots along a path connecting two vertices in the

graph moves at once when an unoccupied vertex is relocated. The average length of such paths correlates with the diameter of the graph.

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 the size of parts of the graph affected by movements of robots is relatively small in still the same in majority of the phases of the algorithm [8] (the algorithm is more **localized** in the graph). Thus, as occurrence of robots is getting linearly sparser the parallelism decreases almost linearly. Recall, that the *BIBOX* algorithm behaves differently. The majority of movements take place in the whole unfinished part of the graph, which is relatively getting smaller as the *BIBOX* algorithm proceeds – this concerns relocation of the unoccupied vertex and individual robots across the unfinished part of the graph (the algorithm works with the graph more **globally**).

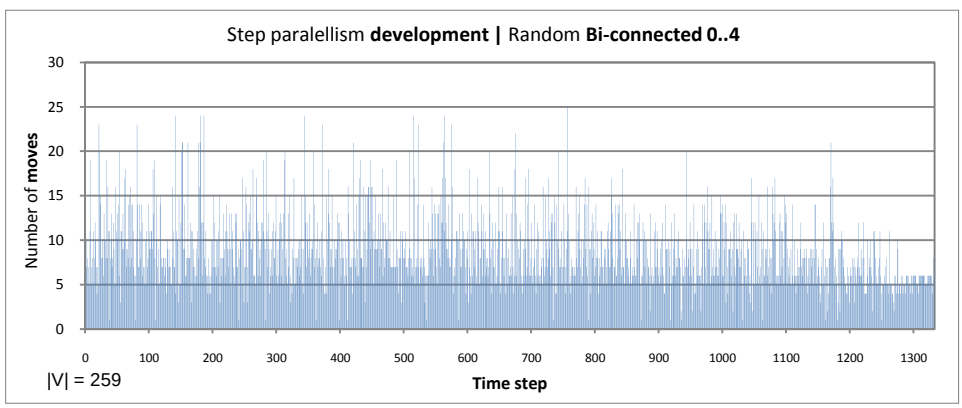


Figure 18. ***Step parallelism development*** *of a solution on a random bi-connected graph generated by the* ***BIBOX*** *algorithm.* The random bi-connected graph has been generated with the length of the initial cycle and handles having uniform distribution of the range 0..4. There were exactly two unoccupied vertices in the graph. The solution produced by the standard *BIBOX* algorithm has been parallelized using parallelism-increasing algorithm (Algorithm 3). The development of the *step parallelism* (number of moves per time step) in time is shown.

Generally, it can be concluded from Figure 16 and Figure 17 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 13, Figure 14, Figure 16 and Figure 17 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.

The development of the number of movements per time step called *step parallelism* is shown in Figure 18. This experiment has been done with the *BIBOX* algorithm, which has been used to generate solution of a random instance on a random bi-connected graph

where lengths of the initial cycle and handles of the handle decomposition have been randomly selected with the uniform distribution with the range $0..4$. There were exactly two unoccupied vertices in the input graph.

Peaks in Figure 18 correspond to parallel movements along long path. Observe, that the density and height of these peaks is slightly getting smaller as the algorithm proceeds. As it has been stated, this is caused by the fact that the part of the graph affected by this type of movements is getting smaller. Other values correspond to various rotations along cycles in the graph that are intensively done by the *BIBOX* algorithm. The absolute number parallel of movements corresponding to these rotations does not change as the algorithm proceeds (the average size of a cycle of the unfinished part of the graph is still the same since the graph was generated uniformly).

5.3. ***Scalability Evaluation***

The last series of experiments is devoted to scalability evaluation, that is how tested algorithms behave while the size of instance to solve increases. All these experiments have been performed on the runtime configuration.

Scalability tests were aimed on the makespan of generated solution and the overall runtime necessary to produce a parallel solution. The overall time is the time necessary to produce a sequential solution plus the time needed to increase its parallelism. The following algorithms were compared: *BIBOX*, *BIBOX/MIT*, *BIBOX-*θ/T|weak, *BIBOX-*θ/3|weak, and *MIT*. 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 5.1 regarding the makespan. Moreover, *BIBOX-*θ/T|weak and *BIBOX-*θ/3|weak are slightly faster if all the records in the database of optimal solutions are precomputed off-line (the shorter resulting solution is produced than in the case of *BIBOX-*θ/T and *BIBOX-*θ/3). They are significantly faster if the optimal solutions to special cases need to be computed on-line (on demand) [17, 18] (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 multi-robot path planning problem. 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: $0..2,\ldots, 0..16$. Such selection guarantees that graphs with short handles as well as graphs with long handles are represented. There were exactly two unoccupied vertices in all the tested instances (in order to make application of the *BIBOX* algorithm possible).

Scalability evaluation regarding the makespan is shown in Figure 19. The dependence of the makespan on the number of vertices of the graph of the instance is shown. Figure 20 shows the dependence of the overall solving runtime on the number of vertices. The same set of instances as in the test from Figure 19 has been used. Thus, Figure 19 and Figure 20 show the makespan and the runtime of the same individual instances. In both

figures algorithms are compared pair-wise from the worst performing to the best one (the pair of algorithms nearest according to the given characteristic is compared).

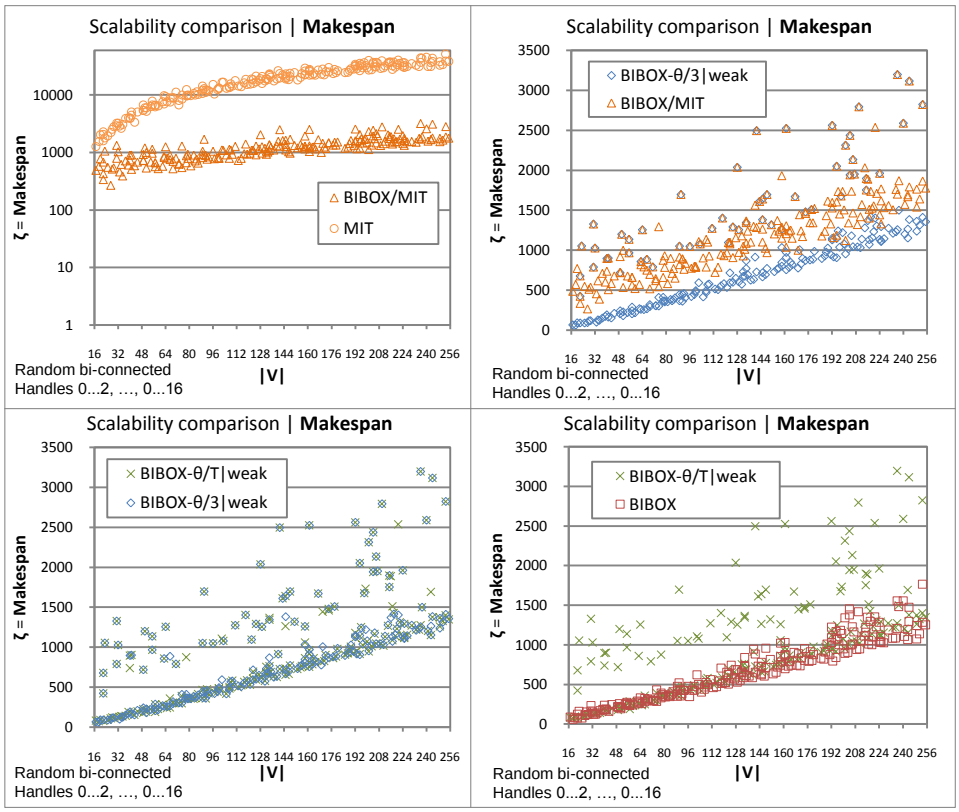


Figure 19. *A **comparison of the scalability** of tested algorithms with respect to the **makespan**.* Five algorithms are compared: *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T|weak, *BIBOX-θ*/3|weak, and *MIT*. Approximately 250 instances over various random bi-connected graphs of the size from 16 to 256 vertices were used. The range of the uniform distribution for random generation of lengths of handles was as follows: 0..2, …, 0..16. Algorithms are sorted from left/top to right/bottom according to their performance. Each sub-chart shows the relative comparison of two algorithms which performance regarding the makespan is nearest. The dependence of the makespan on the size of the graph is shown.

Results regarding **makespan** undoubtedly 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 and catches 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 *MIT* mode of generating (sub-optimal) solutions to special cases.

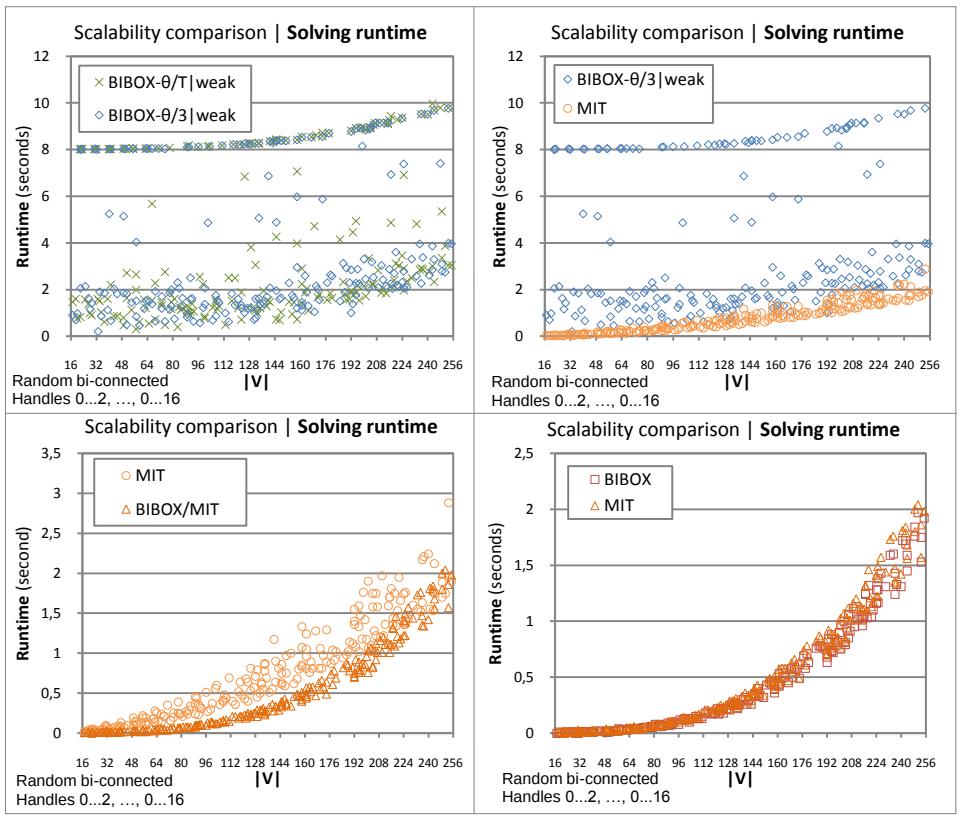


Figure 20. *A **comparison of the scalability** of tested algorithms with respect to the **runtime**.* Again, five algorithms are compared: *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T|weak, *BIBOX-θ*/3|weak, and *MIT*. The setup of instances is the same as for the experiment from Figure 19. Algorithms are sorted from left/top to right/bottom according to their performance. Each sub-chart shows the relative comparison of two algorithms which performance regarding the runtime is nearest. The runtime (the total time necessary to produce sequential solution plus the time for making it parallel) is shown is the dependence on the size of the graph.

The 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 of the handle decomposition of optimal solutions to special cases, it still has the worse makespan than the corresponding solution generated using robot exchanges by the *BIBOX* algorithm.

Results regarding 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 (however, the higher is the timeout the more special cases the algorithm manages to compute). The more interesting situation is with *MIT*, *BI-*

BOX/MIT, and *BIBOX* since they have very close runtimes. The *BIBOX* algorithm is again the best. Observe, that the runtime does not exactly correspond to the length of the generated solutions. In other words, computations used by the *BIBOX* are more time consuming than that of *MIT* (for example *BIBOX* intensively computes shortest paths).

The appearance of the charts in Figure 20 corresponds with theoretical results regarding the worst case time complexity (namely, it seems that the average time complexity meets the worst case). On the other hand, it seems that this is not the case of charts in Figure 19 which exhibit that the average makespan is lower than the worst case estimation. However, this is just a conjecture that requires further investigation (an interesting topic for future work).

6. Related Works and Conclusion

Two algorithms for solving the abstract problem of multi-robot path planning on **bi-connected** graphs were described in this manuscript – *BIBOX* and *BIBOX-θ*. Several modified variants of the *BIBOX-θ* algorithm are described as well. The precise theoretical and experimental analysis of these algorithms is provided. The theoretical analysis is targeted proofs of correctness and on asymptotic estimations of upper bound on the makespan of generated solution and on time and space consumption. The experimental analysis is targeted on comparison with the so called *MIT* algorithm [8] which has been so far the only algorithm capable of solving multi-robot path planning problems with **small unoccupied space** in the environment.

Although the MIT algorithm has promising theoretical properties – solution generated by this algorithm theoretical makespan of $\mathcal{O}(|V|^3)$ with respect to the graph $G = (V, E)$; its worst case time complexity is $\mathcal{O}(|V|^3)$ as well – it has been **significantly outperformed** by *BIBOX* in terms of the makespan by order of one or two magnitudes. Regarding the runtime *BIBOX* algorithm is **slightly faster** than *MIT* which is relatively fast (instances with graphs of hundreds of vertices full of robots are solved within seconds on today's standard hardware). Thus, although asymptotic estimations for makespan are the same for both *BIBOX* and *MIT*, the multiplication factor in the estimation in case of *BIBOX* is smaller.

The **minor drawback** of the *BIBOX* algorithm is that it is not able to solve instances of multi-robot path planning with just 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 different approach for solving the situation on trivial bi-connected graphs consisting of one cycle and one handle connected to it – called θ-like graphs. Except the first algorithm, all the other algorithms use **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 these 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. Consequently, it can cause a significant slowdown of the algorithm. The issue of computing optimal solutions to special cases is addressed in details in [17, 18]. Generally, the detailed description of pros and cons of all the algorithms is given. Thus, the user can prefer some of them according to her/his requirements.

The important **post-processing** part for all the presented algorithms, which is used to **increase parallelism** of generated solutions, is also presented. The technique is based on the method of critical path [11] and on the newly defined notion of independency between moves of robots. This is the essential step since all the algorithms including *MIT* generates sequential solutions with no parallel moves.

Notice, that 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 the small number of large handles or the large number of small handles? This question is out of the scope of this work and it is left for future work.

The direct extension of the presented algorithm can be made by extending them from bi-connected graphs to **general graphs**. However, some new issues need to be solved when doing this. First, there will be many unsolvable instances – it may happen that a robot needs to go the neighboring bi-connected component than it is currently located. If the *bridge* (*isthmus*) connecting these components is longer than the number of unoccupied vertices, the relocation of the robot is not possible.

Solving process for instances of the problem of multi-robot path planning on general graphs can be based on the already developed process for bi-connected graphs. The general graph can be decomposed into the tree of bi-connected components [29, 30]. The algorithm for bi-connected case can be used over the individual bi-connected components. However, robots must be relocated to the goal bi-connected components first. As it was mentioned this is not always possible. Without mentioning further details, the process should proceed by solving **leaf** bi-connected components first and continuing to the **root** bi-connected component. This issue is again out of scope of this work and it is left for future studies.

The important related work is represented by articles [25, 26, 27, 28]. Authors study so called *multi-agent path planning* which is similar to the notion of multi-robot path planning with some further relaxations (for example a swap of agents along an edge seems to be allowed). The number of moves is the optimized parameter. Authors define the tractable class of this optimization problem where graphs are restricted on grids and there is a relative abundance of unoccupied vertices. The major difference from the presented work is that authors are developing solving algorithms for instances with **lot of free space** in the environment. The authors showed that their approach **scale up well**. The theoretical relation of multi-agent path planning and multi-robot path planning is an interesting question for future work.

Another interesting related work is represented by [12, 13]. Despite the title, the author is solving the optimal variant of a problem of pebble motion on a graph. The solving

method is based on search (which has inherently exponential time complexity). To increase the speed the author proposed to decompose the input graph into sub-structures that are easy to solve. However, it seems that the proposed approach **does not scale up** for large number of robots/pebbles (only up to 20 robots are used; environments contain **lot of free** space again).

Recently, a new search method improved by the reduction of the branching factor has been published [15]. It generates optimal solutions to restricted version of the problem of pebble motion on a certain kind of grid graphs (in comparison with grid graphs introduced in this work, the diagonal edges are added [15]). Again, the method is targeted on special environments only and it supposes **lot of free space** in the environment (only up to 60 units are moving in the environment where there is 1000 unoccupied positions). The performance of this technique on graphs with small free space and its scalability are thus questionable. Generally, it can be concluded that it is not directly comparable with the presented work.

Another related approach is to use further abstractions of the already abstract problem formulation. This approach is adopted in [6, 7]. Authors study so called *direction maps* of the environment where the required path is searched at multiple levels of the abstraction of the direction map.

Regarding future work, it is far more interesting to resolve the question whether optimal solutions of multi-robot path planning can be **approximated** by (pseudo-) polynomial time algorithm than to make augmentations of standard search methods with exponential time complexity and to declare them "optimal solving methods". All the related works targeted on generating optimal solutions of problems related to multi-robot path planning are restricted on very specific (typically smallish) instances only (either there is extremely small number of robots or the environment is structurally simple, or both). On the other hand, if the 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 instances.

Tractable cases of the multi-robot path planning problem also worth studying in future. This topic has been already addressed in [25, 26, 27, 28] from some point of view. It seems to be worthwhile to deal with tractable cases, since this is the only approach from related works, which proved to be scalable.

Another interesting topic for future work is to study how solutions generated by presented algorithm can be improved towards optimal makespan. Some initial work has been already done in [21]. It is based identifying and eliminating redundancies from solutions. The performed experiments showed that this is a promising technique.

Acknowledgments

This work is supported by The Czech Science Foundation (Grantová agentura České republiky - GAČR) under the contract number 201/09/P318 and by The Ministry of Education, Youth and Sports, Czech Republic (Ministerstvo školství, mládeže a tělovýchovy ČR – MŠMT ČR) under the contract number MSM 0021620838.

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. 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.
10. 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.
11. S. **Russell** and P. **Norvig**. *Artificial Intelligence: A Modern Approach (second edition).* Prentice Hall, 2003, ISBN: 978-0137903955.
12. 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.
13. 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.
14. P. E. **Schupp** and R. C. **Lyndon**. *Combinatorial group theory*. Springer, 2001, ISBN 978-3-540-41158-1.
15. 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.
16. 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.
17. 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.
18. 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].
19. 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.

20. 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.
21. 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.
22. P. **Surynek**. *Abstract Path Planning for Multiple Robots: A Theoretical Study.* Technical Report, http://ktiml.mff.cuni.cz/~surynek/index.html.php?select=publications, Charles University in Prague, Czech Republic.
23. P. **Surynek**. *Abstract Path Planning for Multiple Robots: An Empirical Study.* Technical Report, http://ktiml.mff.cuni.cz/~surynek/index.html.php?select=publications, Charles University in Prague, Czech Republic.
24. 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.
25. K. C. **Wang** and A. **Botea**. *Tractable Multi-Agent Path Planning on Grid Maps*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 1870-1875, IJCAI Conference, 2009.
26. 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.
27. 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.
28. 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.
29. D. B. **West**. *Introduction to Graph Theory.* Prentice Hall, 2000, ISBN: 978-0130144003.
30. J. **Westbrook**, R. E. **Tarjan**. *Maintaining bridge-connected and biconnected components online*. Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
31. R. M. **Wilson**. *Graph Puzzles, Homotopy, and the Alternating Group*. Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.