Improving Solutions of Problems of Motion on Graphs by Redundancy Elimination

Pavel Surynek[1] and **Petr Koupý**[2]

Abstract. Problems of motion on graphs are addressed in this paper. These problems represent an abstraction for a variety of tasks which goal is to construct a spatial-temporal plan for a set of entities that move in a certain environment and need to reach given goal positions. Specifically, the quality (length) of solutions of these problems is studied. Existing state-of-the-art algorithms for generating solutions are suspected of producing solutions containing redundancies of a priori unknown nature. A visualization tool has been developed to discover such redundancies. Knowledge about solutions acquired by the tool served as a basis for the formal description of redundancies and for the development of methods for their detection and elimination. A performed experimental evaluation showed that the elimination of described redundancies improved existing solutions significantly.

Keywords: path planning, multiple robots, tractable class, graphs

1. INTRODUCTION AND CONTEXT

Problems of motion on a graph as they are introduced in [5, 8, 12] represent a basic abstraction for many real-life and theoretical tasks. The classical task that can be abstracted as a problem of motion on a graph takes place in a certain physical environment where mobile *entities* are moving (for example *mobile robots*). Each entity is given its initial and goal position in the environment. The task is to build a spatial-temporal plan for all the entities such that they reach goal positions following this plan while the plan satisfies certain natural constraints. These constraints are typically constituted by a requirement that entities must avoid obstacles in the environment and must not collide with each other.

The standard abstraction that is adopted throughout this work uses an undirected graph to model the environment. The vertices of this graph represent positions in the environment and the edges represent an unblocked way between two positions. An arrangement of entities in the environment is abstracted as a simple assignment of entities to vertices. At least **one** vertex remains **unoccupied** in order to make the movement of entities possible. The time is discrete; it is an ordered set of *time steps* isomorphic to the structure of *natural numbers*. A way how an arrangement of entities can be transformed into another can slightly differ in variants of the problem.

1.1 Motivation by Practice

The abstract problems of motion on a graph are motivated by many real-life problems. The most typical motivating example is a motion planning of a group of mobile robots that are moving in *2-dimensional space* [8]. Generally, if there is enough free space in the environment, algorithms based on search for shortest paths in a graph can be used [12]. However, if there is little free space, different methods must be used [5, 9, 10].

Many well known puzzles can be formulated as the problem of motion on a graph. The best known is so called Lloyd's 15-puzzle and its generalizations [7, 12]. In practice, various mobile or movable objects may represent the entities – for example, a rearrangement of containers in a storage area can be interpreted as a problem of motion on a graph where entities are represented by containers. Indeed, this approach has been used for planning motions of automated straddle carriers in a storage area in Patrick port facility at Port Brisbane in Queensland [8]. Although the approach suggested in [8] does not scale for larger number of entities, it clearly demonstrates the usefulness of discussed abstractions. Entities do not necessarily have to be physical objects. Virtual spaces of computer simulations and games convey many situations where motions of certain entities must be planned.

It is necessary to stress that contrary to multi-agent motion planning [4], the **centralized approach** is adopted in this work. That is, the environment is fully observable for the central planning mechanism and the individual entities merely execute the submitted centrally created plan.

1.2 Specific Open Questions

There exist several relatively efficient methods for solving problems of motion on a graph. This work is particularly targeted on solution generation methods described in [9, 10]. These methods represent state-of-the-art algorithms for the class of problems where the graph modeling the environment is *bi-connected* and where there are **many entities** placed in the graph (the graph is relatively full with **small unoccupied space**). Despite the qualities of these methods, the generated solution are suspected of containing certain *redundancies*. This is a **conjecture** whose examination is the main contribution of this paper. If it is the case that generated solutions contain redundancies, then a question how they can be removed to improve the solution arises.

The task is thus to analyze solutions of non-trivial size which is manually infeasible. Moreover, it is necessary to emphasize that searched redundancies are of a priori unknown nature. Therefore a comfortable software tool ***GraphRec*** [6] has been developed to allow visual analysis of solutions of problems of motion on a graph. The GraphRec software solves two issues that are difficult to be handled manually. First, the tool draws the graph modeling the environment of the problem on the screen. An embedding of the graph into two dimensions with few edge crossings is preferred to enable **comfortable observation**. Second, motions of entities on the graph are visualized by the tool in time.

Several types of redundancies were discovered by the GraphRec software in solutions. They are formally described in this paper. Further, methods for automated discovery and elimination of these redundancies are suggested and analyzed theoretically as well as experimentally.

The top level **organization** of the paper has two parts. The first part explains a variant of the problem of motion on a graph

[1,2] Charles University in Prague, Faculty of Mathematics and Physics, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic, pavel.surynek@mff.cuni.cz, petr.koupy@gmail.com.

This work is supported by The Czech Science Foundation under the contract number 201/09/P318 and by The Ministry of Education, Youth and Sports, Czech Republic under the contract number MSM 0021620838.

(section 2) and the basic solving algorithm (section 3); this part merely recalls existing concepts. The second part contains the main contribution of this work; the GraphRec visualization tool is introduced (section 4), redundancy elimination methods are described (section 5), and the benefit of suggested methods is justified in the experimental section (section 6).

2. PEBBLE MOTION ON A GRAPH

The basic variant of the motion problem is known as *pebble motion on a graph* [5, 12]. The role of an entity is represented by a pebble here. The task is given by an undirected graph with an *initial* and a *goal arrangement* of pebbles in the vertices of this graph. Each vertex of the graph contains at most one pebble and at least one vertex remains unoccupied. The task is to find a sequence of moves for each pebble such that all the pebbles reach their goal vertices. A pebble can move into a **neighboring unoccupied** vertex while no other pebble is entering the target vertex at the same time. The following definition formalizes the problem. An illustrative instance of the problem is shown in figure 1.

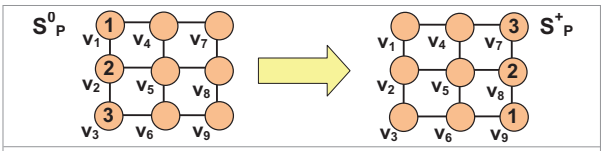
Definition 1 ***(pebble motion on a graph).*** Let $G = (V, E)$ be an undirected graph and let $P = \{p_1, p, \dots, p_\mu\}$ be a set of pebbles where $\mu < |V|$. The ***initial arrangement*** of pebbles is defined by a simple function $S_P^0 : P \longrightarrow V$ (that is $S_P^0(p_i) \neq S_P^0(p_j)$ for $i, j = 1,2, \dots, \mu$ with $i \neq j$); the ***goal arrangement*** of pebbles is defined by another simple function $S_P^+ : P \longrightarrow V$. A problem of ***pebble motion on a graph*** is the task to find a number ξ and a sequence $S_P = [S_P^0, S_P^1, \dots, S_P^\xi]$ where $S_P^k : P \longrightarrow V$ is a simple function for every $k = 1,2, \dots, \xi$. The following constraints must hold:

(i) $S_P^\xi = S_P^+$, that is, pebbles finally reach their destinations.

(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 $k = 1,2, \dots, \xi - 1$.

(iii) If $S_P^k(p) \neq S_P^{k+1}(p)$ then $S_P^k(q) \neq S_P^{k+1}(p)$ for $\forall q \in P$ 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 to a currently unoccupied vertex.

The problem described above is formally a quadruple $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. □



Solution of the problem of pebble motion on a graph with P={1,2,3}

ξ=6	P	S^0_P	S^1_P	S^2_P	S^3_P	S^4_P	S^5_P	S^6_P=S^+_P
	1	**v_1**	v_4	v_7	v_8	v_9	v_9	**v_9**
	2	**v_2**	v_2	v_1	v_4	v_7	v_8	**v_8**
	3	**v_3**	v_3	v_3	v_2	v_1	v_4	**v_7**

Figure 1. *An illustration of a problem of* ***pebble motion on a graph***. The task is to move pebbles from their initial positions specified by S_P^0 to the goal positions specified by S_P^+. A solution of length 6 is shown.

In practice, the *quality of solution* matters. The typical measures of the quality of solution are its **length** (the total number of moves) and **duration** (which corresponds to the number ξ). These numbers are required to be small. Unfortunately, requiring either the length of the solution or its duration to be as small as possible makes the problem **intractable** [7] (the decision variant of the problem is *NP*-complete). This fact is the main reason why existing methods for generating optimal solutions do not scale for larger number of entities [8] (the problem is called *multi-robot path planning* in these works). On the other hand, if there is **no requirement** on the quality, the question whether there exists a solution is in the ***P* class** [5]. However, methods giving evidence that the problem belongs to the *P* class described in [5] generate solutions that are too long and unsuitable for practice. Therefore it is necessary to find a **compromise** between the quality of solution and computational cost of its construction. Methods following this compromise are described in [9, 10]. Solutions produced by these methods will be submitted to analysis by the visualization tool in order to find out how they can be further improved.

3. SOLVING MOTION PROBLEMS

This section is devoted to a brief recall of algorithms described in [9, 10]. An insight into the structure of solutions produced by these algorithms is crucial to understand their quality.

The most important class of pebble motion problems is formed by those whose graph is *bi-connected* which intuitively means that each pair of vertices is connected by two disjoint paths. The following definition specifies bi-connectivity formally.

Definition 2 ***(connectivity, bi-connectivity).*** An undirected graph $G = (V, E)$ is ***connected*** if $|V| \geq 2$ and for every pair of distinct vertices $u, v \in V$ there exists a path connecting u and v in G. An undirected graph $G = (V, E)$ is ***bi-connected*** if $|V| \geq 3$ and for every vertex $u \in V$ the graph $G' = (V - \{u\}, E \cap \{\{v, w\} | v, w \in V \wedge v \neq u \wedge w \neq u\})$ is connected. □

The importance of this class of problems is assessed by the fact that they are **almost always solvable**. Moreover, spatial environments in real tasks are often abstracted as two dimensional **grids** which are bi-connected in most cases.

If the bi-connected graph contains at least **two unoccupied** vertices and it is not isomorphic to a cycle, then every goal arrangement of pebbles is reachable from every initial arrangement [9]. If the graph contains just **one unoccupied** vertex which can be without loss of generality fixed, then any arrangement of pebbles can be regarded as a *permutation* with respect to the initial arrangement. A permutation is *even* if it can be composed of the even number of transpositions; otherwise it is *odd*. If the goal arrangement represents an **even** permutation, then the problem is **always solvable**. In case of an odd permutation, the problem is solvable if and only if the graph contains a cycle of odd length [12]. A treatment of instances containing more than two unoccupied vertices will be discussed further.

For the sake of completeness, it is adequate to mention the case of pebble motion problems on **general graphs**. This case can be solved using methods for bi-connected case. Every undirected graph can be decomposed into a tree of bi-connected components [11]. Having such a decomposition, the pebbles need to be moved into their target bi-connected components first (this may not always be possible). Then the method for the bi-connected case is applied within individual bi-connected components.

An inductive construction of bi-connected graphs by adding *loops* is a pivotal concept in developing solving algorithms. Let $G = (V, E)$ be a graph, a *loop* with respect to G is a sequence of vertices $L = [u, x_1, x_2, \dots, x_l, v]$, where $u, v \in V$ and $x_i \notin V$ for $i = 1,2, \dots, l$ (it is allowed that $l = 0$). The result of *addition* of the loop L to the graph G is a new graph $G' = (V', E')$, where $V' = V \cup \{x_1, x_2, \dots, x_l\}$ and either $E' = E \cup \{\{u, v\}\}$ if $l = 0$ or $E' = E \cup \{\{u, x_1\}, \{x_1, x_2\}, \dots, \{x_{l-1}, x_l\}, \{x_l, v\}\}$ if $l \geq 1$. Every bi-connected graph $G = (V, E)$ can be constructed from a cycle by a sequence of loop additions. Such *loop decomposition* can be effectively determined in time $O(|V| + |E|)$ [11].

3.1 The *BIBOX-θ* Solving Algorithm

The *BIBOX-θ* algorithm [10] solves a case of the problem of pebble motion on a graph when the graph is bi-connected and there is single unoccupied vertex. The BIBOX-θ algorithm represents state-of-the-art for the described class of problems in terms of speed and quality of generated solutions. This is the main reason why solutions produced by this algorithm are studied here.

In the first phase of the algorithm, a loop decomposition is found; that is, a cycle - called *initial cycle* - and a sequence of loops is determined. Without loss of generality it is required that the unoccupied vertex within the goal arrangement of pebbles is in the initial cycle. The algorithm then proceeds inductively according to the loop decomposition from the last loop to the initial cycle with the first loop.

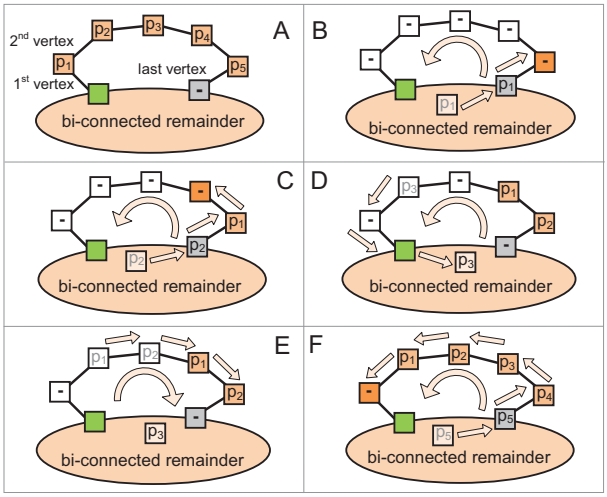


Figure 2. *The **process of placing pebbles** into a loop in the stack manner.* The goal arrangement of pebbles is shown in part **A**. Parts **B** and **C** show a process of ordering new pebbles into the loop in case when they are outside the loop. Part **D** and **E** show ordering process for a pebble when it is already inside the loop. Part **F** shows the final step in which pebbles reach their target vertices. The **green** vertex is unoccupied.

Two properties of bi-connected graphs with at least one unoccupied vertex are exploited while pebbles are placed within loops: **(i)** every vertex can be made unoccupied (this is even true for a connected graph), **(ii)** every pebble can be moved to an arbitrary vertex [9]. A loop is processed in the following way. An orientation of the loop is chosen first – this orientation determines ordering of vertices within the loop. The first and the last vertex of the loop are the connection points to the remainder graph. Then pebbles starting with the pebble whose goal position is in the second vertex of the loop are placed into the loop in the **stack manner**.

The current pebble is moved to the last vertex of the loop. Two cases must be distinguished here. If the pebble is already somewhere in the loop it must be moved outside first. If the current pebble is outside the loop, then it can be moved into the last vertex of the loop using property (ii) (only pebbles within the subgraph without the loop are moved). After placing the pebble into the last vertex of the loop, the loop is rotated once in the direction to the first vertex. The process is illustrated in figure 2.

When all the pebbles within the loop are processed the task is to solve the problem of the same type on a smaller graph – the finished loop is not considered anymore; a bi-connected graph without the last loop is bi-connected again. Nevertheless, the stack manner of placing pebbles cannot be applied for the initial cycle and the first loop of the decomposition. Therefore the algorithm uses a database containing **pre-calculated optimal solutions** for transpositions and rotation of pebbles along 3-cycles in graphs consisting of a cycle and a loop. A solution to any solvable instance on the initial cycle with the first loop is then composed of solutions from the database [10].

If it is the task to solve an instance of the problem with a bi-connected graph where there are **more than one** unoccupied vertices, then all the vertices except one are filled with **dummy pebbles**. The modified problem is then solved by the BIBOX-θ algorithm. Motions of dummy pebbles are finally filtered out of the resulting solution [9]. Such a process of producing solutions of problems with **many unoccupied vertice**s is suspected of generating *redundant moves* that may prolong the solution unnecessarily. However, this statement should be understood as a **conjecture** that has to be verified first.

4. VISUALIZATION TOOL

The examination and reviewing of the solution quality appeared to be difficult without certain automation. Therefore, a visualization tool *GraphRec* [6] has been developed (http://www.koupy.net/graphrec.php). The tool provides an animation engine for the entity movement together with features designed to support the observation of the solution time line. Any similar tool has not been available up until now. With the existing graph visualization software (e.g. *Graphviz* [1]) it is neither possible to represent entities nor move them among graph nodes.

4.1 Functional Requirements

Before the visualization can even occur, the graph on which the movement will be animated have to be embedded on the screen. Since we are dealing with bi-connected graphs, which are not necessarily *planar*, the embedding algorithm should **reduce** the amount of **crossing edges** while maintaining *Euclidean* distances between nodes proportional to the corresponding *shortest paths*.

The **animation of moving entities** is the core feature of the application. Since the solution is built over discrete time steps, these should be possible to play through or even step through in order to increase controllability of the observation. When examining certain part of the solution it is also necessary to provide adjustable speed of the animation and the possibility to jump quickly between various time steps. The clearness of the animation must be taken into attention as well. It appears that *highlighting* of moving entities greatly improves the overall perception of where the motion actually occurs. The demand for user vigilance might be further reduced by *distinguishing* between entities that are already in their final positions and that are not.

4.2 Tool Overview

GraphRec implements two *force-directed* planar embedding algorithms described in [2, 3]. Both methods are based on the simulation of a certain physical model. Whereas the model introduced in [2] considers nodes as repulsive particles and edges as contracting springs, another interpretation where chosen free node is connected by springs to the rest of anchored nodes is proposed in [3]. Owing to their physical background, force-directed algorithms often produce **expected and intuitive layouts** (figure 3).

The tool enables all graph elements to be assigned with various colors. This is especially important in scenarios such as observa-

tion of the movement of one particular entity or even group of entities, where **color differentiation** greatly improves their traceability. Colors are also utilized to distinguish entities in goal positions and to highlight moving entities as shown in figure 4.

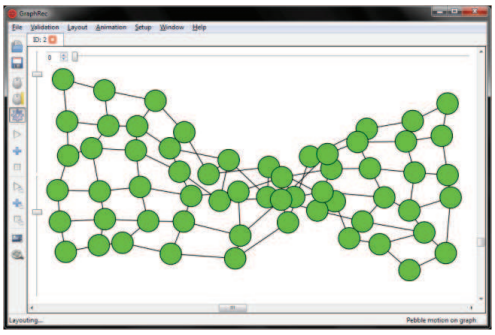


Figure 3. ***Graph layout*** gradually evolving into the regular grid.

Animation of the solution can be controlled in a similar way as playing a movie on a video recorder. Firstly, user adjusts the animation speed and specifies the starting time step. Then, it is possible to play or step through the animation time line. GraphRec supports the synchronized animation of more than one solution at once, which is for example useful when comparing differently optimized solutions for the same problem.

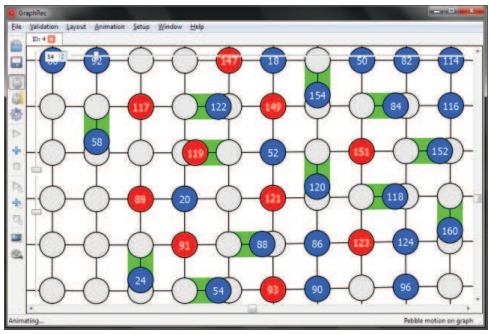


Figure 4. ***Moving entities*** emphasized by highlighted edges.

4.3 Discovering Redundancies

The presented visualization proved itself an effective way for discovering the nature of expected redundancies in solutions. Since the automatic detection of redundancies with unknown characteristics is not possible, the analysis by a human is essential. Because humans are mainly visual-oriented, the visualization of the problem seems to be suitable approach. Acquired knowledge was later used to formalize redundancies and to design methods for their removal.

4.4 Additional Features

GraphRec can find inconsistencies in solution by verifying its movements against constraints specified in the definition of the variant of motion problem. **Solution validation** is necessary to prevent the corruption of the animation. However, the validation can also be utilized for debugging of algorithms.

Moreover, GraphRec might be used as a presentation tool either in real time or to **produce media files**. The animation can be captured into raster and vector images or even into popular video formats. These files can be used within web presentations.

5. ELIMINATION OF REDUNDANCIES

Several types of redundancies were **discovered** using the GraphRec software within the generated solutions. A formal description of these redundancies and algorithms for their elimination are provided in the following sections. The process of transformation of a perception gained by the observation of the visualized solution to a formal description of a redundancy is a creative process. It is currently an open question whether some automation of this process is possible.

When reasoning about redundancies, it is convenient to assume solutions with just one move between consecutive time steps. The BIBOX-θ algorithm produces solutions in this form. A solution of this form can be viewed as a sequence of moves where the position of a move in the sequence corresponds to its time step of commencement. The notation $k_i: u_i \to v_i$ will denote a move of a pebble k_i from a vertex u_i to a vertex v_i commenced at time step i. The move is called *non-trivial* if $u_i \neq v_i$. From the formal point of view, the solution is a sequence of non-trivial moves $\Phi = [k_i: u_i \to v_i | i = 1,2,\dots,\xi-1]$ (consistency with definition 1 is also assumed).

5.1 Inverse Moves

Definition 3 ***(inverse moves).*** A pair of consecutive moves $k_i: u_i \to v_i$ and $k_{i+1}: u_{i+1} \to v_{i+1}$ with $i \in \{1,2,\dots,\xi-2\}$ are called *inverse* if $k_i = k_{i+1}$, $u_i = v_{i+1}$, and $v_i = u_{i+1}$. □

Observe that a pair of inverse moves can be left out of the solution without affecting its *validity* - it still solves the problem. However, elimination of an inverse pair may cause that another pair of inverse moves arises. Hence, it is necessary to remove inverse moves from the solution repeatedly until there are none.

The process of elimination of inverse moves is expressed below as algorithm 1. The worst case time complexity of the algorithm is $O(|\Phi|^2)$, space complexity is $O(|\Phi|)$.

Algorithm 1. Elimination of inverse moves.

function *EraseInverseMoves* (Φ): **sequence**
1: **do**
2: $\eta \leftarrow \emptyset$
3: **let** $[k_1: u_1 \to v_1, k_2: u_2 \to v_2, \dots, k_{\xi-1}: u_{\xi-1} \to v_{\xi-1}] = \Phi$
4: **for** $i = 1,2,\dots,\xi-1$ **do**
5: **if** $k_i: u_i \to v_i$ **and** $k_{i+1}: u_{i+1} \to v_{i+1}$ are inverse **then**
6: $\eta \leftarrow \eta \cup \{k_i: u_i \to v_i, k_{i+1}: u_{i+1} \to v_{i+1}\}$
7: $\Phi \leftarrow \Phi - \eta$
8: **while** $\eta \neq \emptyset$
9: **return** Φ

5.2 Redundant Moves

Definition 4 ***(redundant moves).*** A sequence of moves $[k_{i_j}: u_{i_j} \to v_{i_j} | j = 1,2,\dots,l]$, where $I = [i_j \in \{1,2,\dots,\xi-2\} | j = 1,2,\dots,l]$ is an increasing sequence of indices, is called *redundant* if $|\{k_{i_j} | j = 1,2,\dots,l\}| = 1$, $u_{i_1} = v_{i_l}$, and for each move $k_\iota: u_\iota \to v_\iota$ with $i_1 < \iota < i_l \wedge \iota \notin I$ it holds that $k_\iota \neq k_{i_1} \Rightarrow u_{i_1} \notin \{u_\iota, v_\iota\}$. □

Redundant moves represent **generalization** of inverse moves (a pair of inverse moves form a redundant sequence). It is a sequence of moves which relocates a pebble into some vertex for the second time while other pebbles do not enter this vertex at any time step between the beginning and the end of the sequence. Eliminating a redundant sequence of moves preserves validity of the solution. Again, it is necessary to remove redundant sequences repeatedly since its removal may cause that another redundant sequence arises.

Algorithm 2 formalizes the process of removing redundant moves in the pseudo-code. The worst case time complexity is $O(|\Phi|^4)$, the space complexity is $O(|\Phi|)$.

Algorithm 2. Elimination of redundant moves.

function *EraseRedundantMoves* (Φ): **sequence**
do
  $\eta \leftarrow$*FindRedundantMoves*(Φ)
  $\Phi \leftarrow \Phi - \eta$
while $\eta \neq \emptyset$
return Φ

function *FindRedundantMoves* (Φ): **sequence**
let $[k_1{:}u_1 \rightarrow v_1, \ldots, k_{\xi-1}{:}u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
for $i = 1,2, \ldots, \xi-2$ **do** {beginning of redundant sequence}
  for $j = \xi-1, \xi-2, \ldots, i+1$ **do** {end of redundant sequence}
    if $k_i = k_j \wedge u_i = v_j$ **then**
      $\eta \leftarrow \emptyset$ {redundant sequence}
      for $\tau = i, i+1, \ldots, j$ **do**
        if $k_i = k_\tau$ **then** $\eta \leftarrow \eta \cup \{k_\tau{:}u_\tau \rightarrow v_\tau\}$
      if *CheckRedundantMoves*(Φ, i, j) **then return** η
return $\emptyset$

function *CheckRedundantMoves* (Φ, i, j): **boolean**
let $[k_1{:}u_1 \rightarrow v_1, \ldots, k_{\xi-1}{:}u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
for $\iota = i+1, i+2, \ldots, j-1$ **do**
  if $k_\iota \neq k_i \wedge u_i \in \{u_\iota, v_\iota\}$ **then return** *False*
return *True*

5.3 Long Sequences

Definition 5 ***(long sequence).*** Let $O^t(P)$ be a set of vertices occupied by a set of pebbles P at a time step t. A sequence of moves $[k_{i_j}{:}u_{i_j} \rightarrow v_{i_j} | j = 1,2, \ldots, l]$, where $I = [i_j \in \{1,2, \ldots, \xi-2 | j = 1,2, \ldots, l\}]$ is an increasing sequence of indices, is called *long* if $|\{k_{i_j} | j = 1,2, \ldots, l\}| = 1$ and there exists a path $C = [c_1 = u_{i_1}, c_2, \ldots, c_n = v_{i_l}]$ in G such that $n < l$, $C \cap O^{i_1}(P - \{k_{i_1}\}) = \emptyset$, and for all the moves $k_\iota{:}u_\iota \rightarrow v_\iota$ with $i_1 < \iota < i_l \wedge \iota \notin I$ it holds that $k_\iota \neq k_{i_1} \Rightarrow \{u_\iota, v_\iota\} \cap C = \emptyset$. □

The concept of long sequence is a **generalization** of redundant sequence (the path C is empty in the case of redundant sequence). Intuitively, the long sequence can be replaced by a sequence of moves along a shorter path (cutoff path) into which other pebbles do not enter between the beginning and the end of the sequence. Replacing a long sequence of moves by a sequence of moves along the path C again preserves validity of the solution. The replacement of long sequences must be performed repeatedly since new long sequences may arise.

The process of replacement is formally expressed below as algorithm 3. The worst case time complexity is $O(|\Phi|^4 + |\Phi|^3|V|^2)$; the space complexity is $O(|\Phi| + |V| + |E|)$.

5.4 Summary of Redundancy Elimination

Redundancies described above were discovered using the GraphRec software. Notice that the gradual generalization was adopted in the description. Although long sequences subsume both less general redundancies, it is not advisable to apply their replacement directly. It is better to apply elimination of redundancies stepwise from the less general one to more general ones. The reason for this practice is the increasing time complexity of redundancy elimination algorithms. A sequence of moves submitted to the more complex algorithm is potentially shortened by eliminating less general redundancies using this practice.

It is possible to reason about the implementation of a certain level of automation in the search for other types of redundancies. The common requirement shared by all the definitions is that the resulting solution must be shorter.

Algorithm 3. Replacement of long sequences.

function *ReplaceLongMoves* (Φ, G): **sequence**
do
  $(\eta, \pi) \leftarrow$*FindLongMoves*(Φ, G)
  $\Phi \leftarrow \Phi - \eta$; $\Phi \leftarrow \Phi \cup \pi$
while $(\eta, \pi) \neq (\emptyset, [])$
return Φ

function *FindLongMoves* (Φ, G): **pair**
let $[k_1{:}u_1 \rightarrow v_1, \ldots, k_{\xi-1}{:}u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
for $i = 1,2, \ldots, \xi-2$ **do**
  for $j = \xi-1, \xi-2, \ldots, i+1$ **do**
    if $k_i = k_j$ **then**
      $\eta \leftarrow \emptyset$
      for $\tau = i, i+1, \ldots, j$ **do**
        if $k_i = k_\tau$ **then** $\eta \leftarrow \eta \cup \{k_\tau{:}u_\tau \rightarrow v_\tau\}$
      $C \leftarrow$*CheckLongMoves*$(\Phi, i, j, |\eta|, G)$
      if $C \neq []$ **then**
        let $[c_1, c_2, \ldots, c_n] = C$
        $\pi \leftarrow [k_i{:}c_1 \rightarrow c_2, \ldots, k_i{:}c_{n-1} \rightarrow c_n]$
        return (η, π)
return $(\emptyset, [])$

function *CheckLongMoves* $(\Phi, i, j, l, G = (V, E))$: **sequence**
let $[k_1{:}u_1 \rightarrow v_1, \ldots, k_{\xi-1}{:}u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
$(V', E') \leftarrow G$; $V' \leftarrow V' - O^i(P - \{k_i\})$; $E' \leftarrow E' \cap \{\{u, v\} | u, v \in V'\}$
for $\iota = i+1, i+2, \ldots, j-1$ **do**
  if $k_\iota \neq k_i$ **then**
    $V' \leftarrow V' - \{u_\iota, v_\iota\}$; $E' \leftarrow E' \cap \{\{u, v\} | u, v \in V'\}$
let C be a shortest path between u_i and v_j in $G' = (V', E')$
if C is defined **and** $|C| < l$ **then return** C
return []

6. EXPERIMENTAL EVALUATION

An experimental evaluation was made with the suggested methods for redundancy elimination. Algorithms 1, 2, and 3 were implemented in C++ and were tested on a set of benchmark instances of the problem of pebble motion.

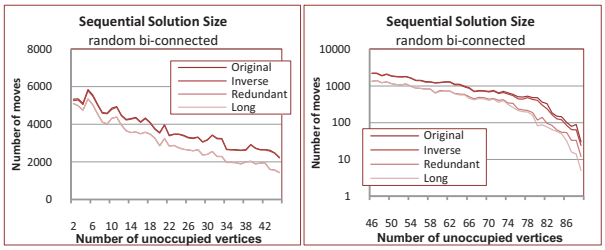


Figure 5. ***Solution length improvement*** *in random bi-connected graph.* Notice that the right part uses the logarithmic scale. The dependence on the increasing number of unoccupied vertices is shown. Up to 50% smaller solution can be obtained by eliminating redundant or long sequences.

Solutions found by the BIBOX-θ algorithm were submitted to redundancy elimination methods. The reduction of the length of the solution and runtime were measured. The implementation of redundancy elimination algorithms directly follows the pseudo-code given in section 5. It was always the case that the solution was processed by the less general redundancy elimination before it was submitted to more general one. In order to allow reproducibility of experiments the complete source code together with raw experimental data is provided at the web: http://ktiml.mff.cuni.cz/~surynek/research/ecaiw2010.

The first set of problems consists of **randomly generated bi-connected** graph with 90 vertices. The graph was constructed by

adding loops of random length (uniform distribution from 2..10) to the cycle of length 7 (actually tests were done with many random bi-connected graphs, indeed only one was selected for presentation here). The initial and the goal arrangement of pebbles were generated as random permutations.

The second set of testing instances consists of a **grid** of size 8×8 where the initial and the goal arrangement of pebbles were again random permutations. In both cases, the random permutation was generated by applying quadratic number of random transpositions of individual pebbles.

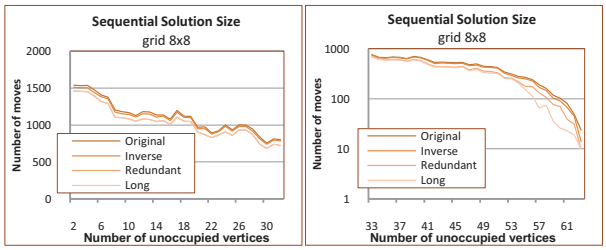


Figure 6. ***Solution length improvement*** *in the grid 8×8.* The right part uses logarithmic scale. Up to 10% smaller solution can be obtained by eliminating redundant or long sequences.

The reduction of the length of the solution depending on the increasing number of unoccupied vertices is shown in figures 5 and 6. Runtime of the individual methods is not presented due to space limitations. However, it can be summarized that the long sequence replacement is the most time consuming method. It consumed approximately 2 minutes (measured on a 2.4GHz machine) on instances with many pebbles.

It is possible to conclude that the solution can be reduced by up to 50% of the original size for problem on **random bi-connected** graph while better results are achieved when there is higher number of unoccupied vertices. For the **grid** 8×8, the reduction is not that large; the original size of the solution can be reduced by up to about 10%. Again, problems with higher number of unoccupied vertices render the possibility for better improvements.

Removal of redundant sequences represents the **best trade-off** between detection cost and solution improvement according to performed experiments. Whereas eliminating inverse moves or long sequences features utmost situations; the former brings almost no improvement; the latter is computationally too costly.

An expectable result is that the better improvement of solutions is gained when there are more unoccupied vertices in the input graph. Notice that definitions of redundancies are based on the **mutual non-interfering** motions of pebbles. The more unoccupied space is available in the graph the less interference between moves of pebbles is possible. The difference in the improvement for random bi-connected graphs and grids is partially caused by the difference of the average length of loops of the loop decomposition. The smaller these loops are the higher the interaction among pebbles is.

The most prohibitive aspect of the redundancy elimination methods with respect to their eventual practical application is quite high runtime. In additional experiments with larger graphs the runtime of removal of redundant sequences as well as the runtime of long sequence replacement was too high. However, this issue may be resolved by using better redundancy detection algorithms with lower asymptotic time complexity. This can be done by exploiting advanced data structures or by a so called opportunistic redundancy elimination which does not eliminate all the redundancies but only those that are encountered.

7. SUMMARY AND CONCLUSIONS

This work addressed the quality (length) of solutions of problems of pebble motion on a graph. Particularly, solutions generated by the existing state-of-the-art algorithm [9, 10] were analyzed with respect to presence of certain type of redundancies. If such redundancies really exist, which proved to be the case, their formal description and elimination was the next goal of this work. The new **visualization tool** *GraphRec* **has been developed** to enable comfortable analysis of solutions.

Several types of **redundancies were discovered** using the *GraphRec* software in generated solutions. **Methods for elimination** of described redundancies **were suggested** and experimentally evaluated. The performed experimental evaluation showed that solutions can be improved by up to 50% using the suggested methods. Another finding is that the better improvement can be gained for problems with higher number of unoccupied vertices.

REFERENCES

[1] Bilgin, A., Ellson, J., Gansner, E., Hu, Y., Koren, Y., North, S. *Graphviz - Graph Visualization Software*. Project web page, http://www.graphviz.org, 2009, (September 2009).

[2] Fruchterman, T. M. J. and Reingold, E. M. *Graph Drawing by Force-Directed Placement*. Software: Practice and Experience, Volume 21(November 1991), 1129-1164, John Wiley & Sons, 1991.

[3] Kamada, T. and Kawai, S. *An algorithm for drawing general undirected graphs.* Information Processing Letters, Volume 31 (January 1989), pp. 7-15, Elsevier, 1989.

[4] Kishimoto, A., Sturtevant, N. R. *Optimized algorithms for multi-agent routing.* Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Volume 3, pp. 1585-1588, IFAAMAS 2008.

[5] Kornhauser, D., Miller, G. L., Spirakis, P. G. *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.

[6] Koupý, P. *GraphRec - a visualization tool for entity movement on graph.* Student project web page, http://www.koupy.net/graphrec.php, 2010, (January 2010).

[7] Ratner, D. and Warmuth, M. K. *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.

[8] Ryan, M. R. K. *Exploiting subgraph structure in multi-robot path planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, (January 2008), pp. 497-542, AAAI Press, 2008.

[9] Surynek, P. *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.

[10] Surynek, P. *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.

[11] Tarjan, R. E. *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.

[12] Wang, K. C., Botea, A. *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.

[13] Wilson, R. M. *Graph Puzzles, Homotopy, and the Alternating Group*. Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.