Multiple Encoding of a Watermark Number into Reducible Permutation Graphs using Cotrees

Maria Chroni and Stavros D. Nikolopoulos

Abstract: *Software watermarking involves embedding a unique identifier, i.e., a watermark value, within a software to discourage software theft; to this end, several graph theoretic watermark methods encode the watermark values as graph structures and embed them in application programs using a wide range of algorithmic techniques. In this paper we propose an efficient method for encoding the same watermark value into several different graphs, we call it multiple encoding, answering thus the question we have recently left open. In particular, we propose an efficient algorithm which embed a cograph $G[\pi^*]$ into a reducible permutation graph $F[\pi^*]$ by first computing the cotree of $G[\pi^*]$, then computing a rooted binary tree having specific node-value and child-parent properties, and finally, based on these properties, producing a reducible permutation graph $F[\pi^*]$. In light of our recent encoding algorithms which encode a watermark value w as a self-inverting permutation π^* and the permutation π^* into several cographs $G_1[\pi^*], G_2[\pi^*], \ldots, G_n[\pi^*]$, we conclude that we can efficiently encode the same watermark value w into several reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$. This property causes a codec watermarking system resilient to attacks since we can embed multiple copies of the same watermark value w into an application program. We also propose decoding algorithms which efficiently extract the watermark value w from the reducible permutation graph $F[\pi^*]$. Moreover, our encoding and decoding algorithms have low time complexity and can be easily implemented.*

Key words: *software watermarking, watermark numbers, self-inverting permutations, cographs, cotrees, reducible permutation graphs, encoding, decoding, algorithms.*

INTRODUCTION

The *software watermarking problem* can be described as the problem of embedding a structure w into a program P such that w can be reliably located and extracted from P even after P has been subjected to code transformations such as translation, optimization and obfuscation [13, 15, 16, 18].

Recently, several software watermarking algorithms have appeared in the literature that encode watermarks as graph structures. From a graph-theoretic point of view, we are looking for a class of graphs $\mathcal{G}$ and a corresponding codec $(\texttt{encode}, \texttt{decode})_{\mathcal{G}}$ system with the following properties: (i) appropriate graph types, (ii) high resiliency, (iii) small size, and (iv) efficient encoding and decoding functions.

In 1996, Davidson and Myhrvold [11] proposed the first software watermarking algorithm which is static and embeds the watermark by reordering the basic blocks of a control flow-graph. Based on this idea, Venkatesan et al. [17] proposed the first graph-based software watermarking algorithm which embeds the watermark by extending a method's control flow-graph through the insertion of a directed subgraph; it is called VVS or GTW.

Collberg et al. [7] proposed an implementation of GTW, which they call $\texttt{GTW}_{\texttt{sm}}$; note that, it is the first publicly available implementation of the algorithm GTW. In $\texttt{GTW}_{\texttt{sm}}$ the watermark is encoded as a reducible permutation graph (RPG) [6], which is a reducible control flow-graph [19, 20] with maximum out-degree of two, mimicking real code. The first dynamic watermarking algorithm $(\texttt{CT})$ was proposed by Collberg and Thomborson [8]; it embeds the watermark through a graph structure which is built on a heap at runtime.

CompSysTech'12, June 22–23, 2012, Ruse, Bulgaria.

Recently, Chroni and Nikolopoulos [3] extended the class of graphs which can be efficiently used in a software watermarking system by proposing efficient codec algorithms that embed watermark values into cographs through the use of self-inverting permutations (or, for short, SiP) and extract them from the graph structures using cotrees. The main property of their codec system is its ability to encode the same integer w, through the use of a self-inverting permutation π^*, into more than one cograph $G_1[\pi^*], G_2[\pi^*], \ldots, G_n[\pi^*]$, $n \geq 2$.

In the same paper [3], Chroni and Nikolopoulos left open the problem of encoding a watermark value w, or, equivalently, a self-inverting permutation π^* into more than one reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$; note that, they have proposed efficient algorithms which encode (resp. extract) a watermark value w into (resp. from) a self-inverting permutation π^* [2].

In this paper, we propose an efficient transformation of a cograph $G[\pi^*]$, produced by the encoding algorithm of [3], into a reducible permutation graph $F[\pi^*]$. In particular, we propose an efficient encoding algorithm, we call it `Encode_Cotree.to.RPG`, which embed a cograph $G[\pi^*]$ into a reducible permutation graph $F[\pi^*]$ by first computing the cotree of $G[\pi^*]$, then computing a rooted binary tree $R[\pi^*]$ having specific node-value and child-parent properties, and finally, based on these properties, producing a reducible permutation graph $F[\pi^*]$; we also propose a decoding algorithm, we call it `Encode_RPG.to.SiP`, which extracts the SiP π^* from the reducible permutation graph $F[\pi^*]$ using the tree $R[\pi^*]$ and specific properties of the inorder sequence of the nodes of $R[\pi^*]$.

Thus, in light of our encoding algorithm which encodes a watermark integer w as a self-inverting permutation π^* [2], we conclude that we can efficiently encode the same watermark integer w into several different reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$. This property causes a codec system resilient to attacks since we can embed multiple copies of the same watermark number w into an application program. Moreover, our encoding and decoding algorithms have low time complexity and can be easily implemented.

BACKGROUND RESULTS

We consider finite graphs with no multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex (or, node) set and edge set of G, respectively. For basic definitions in graph theory refer to [12].

Self-inverting Permutations

Next, we define a type of permutations, named *self-inverting permutations*, that are key-objects in our algorithms for encoding numbers as graphs. Let π be a permutation over the set $N_n = \{1, 2, \ldots, n\}$. We think of permutation π as a sequence $(\pi_1, \pi_2, \ldots, \pi_n)$, so, for example, the permutation $\pi = (1, 4, 2, 7, 5, 3, 6)$ has $\pi_1 = 1$, $\pi_2 = 4$, ect. Notice that π_i^{-1} is the position in the sequence of the number i; in our example, $\pi_4^{-1} = 2$, $\pi_7^{-1} = 4$, $\pi_3^{-1} = 6$, etc [12].

Definition 1. The inverse of a permutation $(\pi_1, \pi_2, \ldots, \pi_n)$ is the permutation $(q_1, q_2, \ldots, q_n)$ with $q_{\pi_i} = \pi_{q_i} = i$. A *self-inverting permutation* (or, involution) is a permutation that is its own inverse: $\pi_{\pi_i} = i$.

By definition, every permutation has a unique inverse, and the inverse of the inverse is the original permutation. Clearly, a permutation is a self-inverting permutation if and only if all its cycles are of length 1 or 2; hereafter, we shall denote a 2-cycle as $c = (x, y)$ and an 1-cycle as $c = (x)$, or, equivalently, $c = (x, x)$.

Reducible Permutation Graphs

A flow-graph is a directed graph F with an initial node s from which all other nodes are reachable. A directed graph G is strongly connected when there is a path $x \to y$ for all nodes x, y in $V(G)$. A node x is an *entry* for a subgraph H of the graph G when there is a path $p = (y_1, y_2, \ldots, y_k, x)$ such that $p \cap H = \{x\}$.

Definition 2. A flow-graph is reducible when it does not have a strongly connected subgraph with two (or more) entries.

There are at least three other equivalent definitions [19, 20]; we give the following: F is reducible iff F can be transformed into a single node by repeated application of the transformations T_1 and T_2, where transformation T_1 removes a cycle-edge (x, x), while T_2 picks a non-initial node y that has only one incoming edge (x, y) and glue nodes x and y.

Cographs and Cotrees

Cographs were introduced in the early 1970s by Lerchs [14] who studied their structural and algorithmic properties. Along with other properties, Lerchs has shown that the cographs admit a unique tree representation, up to isomorphism, called a cotree. The cotree of a cograph G is a rooted tree such that:

(i) each internal node, except possibly for the root, has at least two children;

(ii) the internal nodes are labeled by either 0 (0-nodes) or 1 (1-nodes); the internal nodes that are children of a 1-node (0-node resp.) are 0-nodes (1-nodes resp.);

(iii) the leaves of the cotree are in a 1-to-1 correspondence with the vertices of G, and two vertices v_i, v_j are adjacent in G if and only if the least common ancestor of the leaves corresponding to v_i and v_j is a 1-node (see, Figure 1).

The study of cographs led naturally to constructive characterizations that implied several linear-time recognition algorithms that also enabled the construction of the corresponding tree representation (cotree) in linear time [1]. The first linear-time recognition and cotree-construction algorithm was proposed by Corneil, Perl, and Stewart in 1985 [9].

Encoding a Watermark Number into many Cographs

Recently, Chroni and Nikolopoulos presented the algorithm `Encode_W.to.SIP` for encoding an integer as self-inverting permutation [2]. The authors also presented an extraction algorithm which takes as input a self-inverting permutation π^* and returns its corresponding integer w; it is called `Decode_SIP.to.W`.

Based on the results of [2], the same authors proposed an algorithm for encoding a self-inverting permutation as a cograph; the algorithm is called `Encode_SIP.to.Cograph` [3]. Their algorithm takes as input a self-inverting permutation π^* of length $2n + 1$ produced by algorithm `Encode_W.to.SIP`, and then constructs an arbitrary cograph $G[\pi^*]$ on $2n + 1$ vertices by preserving the cycle relation of permutation π^*. We next describe the encoding algorithm by the help of an example:

Example (Encode a SiP into Cographs): Let $\pi^* = (3, 5, 1, 7, 2, 6, 4)$ be the input self-inverting permutation in the algorithm `Encode_SIP.to.Cograph` which corresponds to watermark number w. The algorithm first constructs the graph H having $V(H) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, and $E(H) = \{(v_1, v_3), (v_2, v_5), (v_4, v_7)\}$ and then computes its connected components $H_1 = H[v_1, v_3]$, $H_2 = H[v_2, v_5]$, $H_3 = H[v_4, v_7]$, and $H_4 = H[v_6]$; note that $H_1 = H[v_1, v_3]$ is the

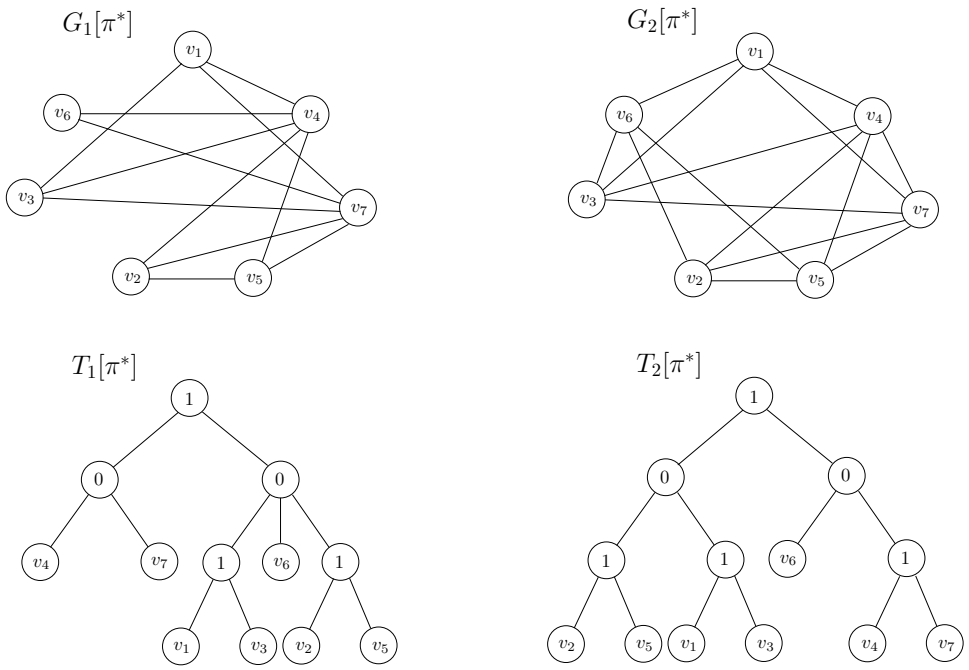

Figure 1: Two cographs $G_1[\pi^*]$ and $G_2[\pi^*]$ on 7 vertices which encode the same watermark number w, and the corresponding cotrees $T_1[\pi^*]$ and $T_2[\pi^*]$.

subgraph of H induced by the nodes v_1 and v_3 (for the construction of the cographs $G_1[\pi^*]$ and $G_2[\pi^*]$ of Figure 1, see [3]).

Chroni and Nikolopoulos also presented a decoding algorithm for extracting a self-inverting permutation from a cograph [3]. Their algorithm `Decode_Cograph.to.SIP` takes as input a cograph $G[\pi^*]$ produced by algorithm `Encode_SIP.to.Cograph` and extracts the self-inverting permutation π^* from $G[\pi^*]$ by constructing first its cotree $T[\pi^*]$ and then finding the pairs of nodes (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) such that the nodes x_i and y_i, $1 \leq i \leq n$, have the same internal node (0-node or 1-node) as parent; these pairs correspond to 2-cycles of the permutation π^*. We next give a brief description of the decoding algorithm by the help of an example:

Example (Extract a SiP from Cographs): Let $G_1[\pi^*]$ and $G_2[\pi^*]$ be two cographs produced by the encoding algorithm `Encode_SIP.to.Cograph` (see Figure 1). The decoding algorithm constructs first the corresoding cotrees $T_1[\pi^*]$ and $T_2[\pi^*]$, and then computes the pairs of nodes (v_4, v_7), (v_1, v_3) and (v_2, v_5). Then it constructs the identity permutation $\pi^* = (1, 2, 3, 4, 5, 6, 7)$, which maps every element of the set N_n to itself, and changes the positions of element 4 and 7, 1 and 3, and 2 and 5; it returns the self-inverting permutation $\pi^* = (3, 5, 1, 7, 2, 6, 4)$ which corresponds to watermark number w.

MULTIPLE ENCODING OF A WATERMARK NUMBER INTO RPGs

Having presented an encoding algorithm which embeds a watermark number into several deferent cographs [3], let us next propose an algorithm which embeds a cograph into a reducible permutation graph (or, for short, RPG) using the structure and some important properties of the class of cographs. Thus, having such an algorithm, we can encode a watermark number w into many RPGs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, where $n \geq 2$.

Encoding a Cograph as Reducible Permutation Graph

We next propose the algorithm `Encode_Cotree.to.RPG` which takes as input the cotree $T[\pi^*]$ of a cograph $G[\pi^*]$ produced by the algorithm `Encode_SiP.to.Cograph` [3], and constructs a reducible permutation graph $F[\pi^*]$ by using an efficient node elimination and subtree modification on $T[\pi^*]$. The whole encoding process takes $O(n)$ time and requires $O(n)$ space, where n is the length of the input cotree $T[\pi^*]$.

Given the cotree $T[\pi^*]$ of a cograph $G[\pi^*]$ on n vertices our encoding algorithm works on two phases:

(I) it first uses a strategy based on node elimination and subtree modification to transform the cotree $T[\pi^*]$ into a binary tree $R[\pi^*]$, which we call RPG-tree, having the property that each node has value smaller than the value of its parent;

(II) then, it constructs a directed graph $F[\pi^*]$ on $n+2$ nodes using the child-parent relation of the nodes of the tree $R[\pi^*]$.

Next, we describe in detail the encoding algorithm `Encode_Cotree.to.RPG` (see, Figure 2 and Figure 3); the two phases of the algorithm work as follows:

Phase I: Construction of the RPG-tree $R[\pi^*]$ from $T[\pi^*]$: We construct the RPG-tree (rpgtree) $R[\pi^*]$ by eliminating the internal nodes of the cotree $T[\pi^*]$ and max-merging certain subtrees in an appropriate way, as follows:

I.1. Let r be the root of the cotree $T[\pi^*]$ and t be the internal node of $T[\pi^*]$ with only one leaf, say, v;
- Replace the label of the root r with the number $n+1$ and the label of each internal node with the number -1;
- Create a new node with value 0 and make it child of the node v;

I.2. While the tree $T[\pi^*]$ contains internal nodes u with value -1, do the following:
- Find such an internal node u of maximum high and let $u_1, u_2, \ldots, u_k$ be the children of u, $k > 1$;
- Max-merge the subtrees $T(u_1), T(u_2), \ldots, T(u_k)$ and let $T(u_m)$ be the resulting subtree rooted at node u_m, $1 \leq m \leq k$;
- Make the root u_m of the resulting subtree $T(u_m)$ child of the parent of u;
- Delete the node u from the tree $T[\pi^*]$;

where the function Max-merge works as follows:

- Order the subtrees $T(u_1), T(u_2), \ldots, T(u_k)$ according to their root values, and let $u_1 < u_2 < \ldots < u_k$;
- Make the root u_i of $T(u_i)$ child of the root u_{i+1} of $T(u_{i+1})$, for all i ($1 \leq i \leq k-1$);

Phase II: Construction of the RPG $F[\pi^*]$ from $R[\pi^*]$: We construct the directed graph $F[\pi^*]$ by exploiting the child-parent relation of the nodes of the rpgtree $R[\pi^*]$, as follows:

II.1. For every node u_i of $R[\pi^*]$, $0 \leq i \leq n+1$, create a node v_i and add it to $V(F[\pi^*])$; that is, $V(F[\pi^*]) = \{s = v_{n+1}, v_n, v_{n-1}, \ldots, v_1, , v_0 = t\}$;

II.2. For every pair of nodes (v_i, v_{i-1}) of the set $V(F[\pi^*])$ add the directed edge (v_i, v_{i-1}) in $E(F[\pi^*])$, $1 \leq i \leq n+1$; we call it *list* pointer;

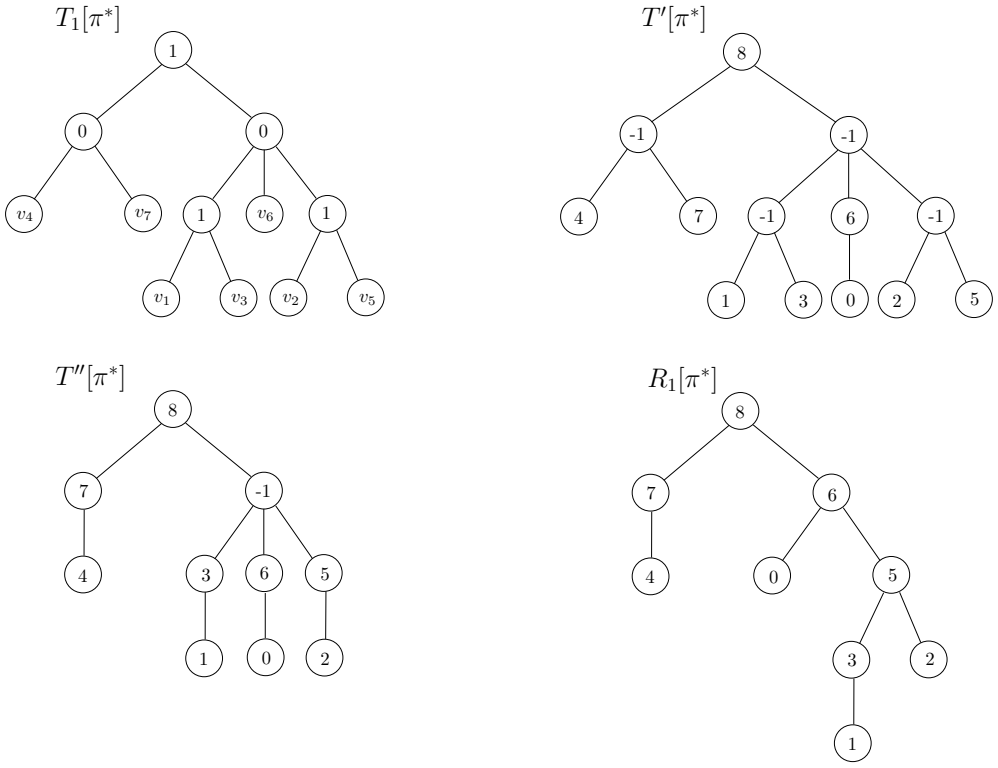

Figure 2: The cotree $T_1[\pi^*]$ of Figure 1 and the resulting RPG-tree $R_1[\pi^*]$; trees $T'[\pi^*]$ and $T''[\pi^*]$ show the contraction process.

II.3. For every node u_i of $R[\pi^*]$ compute its parent node $p(v_i)$, $0 \leq i \leq n$, and add the directed edge $(u_i, p(u_i))$ in $E(F[\pi^*])$; we call it *tree* pointer.

Theorem 1. *Let $T[\pi^*]$ be the cotree of a cograph $G[\pi^*]$ on n vertices. The encoding algorithm* `Encode_Cotree.to.RPG` *encodes the cotree $T[\pi^*]$ into a reducible permutation graph $F[\pi^*]$ in $O(n)$ time and space.*

Algorithm Decode_RPG.to.SiP

Next, we present such a decoding algorithm, we call it `Decode_RPG.to.SiP`, which is efficient: it takes time and space linear in the size of the flow-graph $F[\pi^*]$, and easily implemented: the only operations used by the algorithm are edge modifications on $F[\pi^*]$ and inorder traversal on trees.

The algorithm takes as input a reducible permutation graph $F[\pi^*]$ on $n+2$ nodes and produces a self-inverting permutation π^* of length n; it works as follows:

Algorithm `Decode_RPG.to.SiP`

1. Delete the directed edges (v_{i+1}, v_i) from the edge set $E(F[\pi^*])$, $1 \leq i \leq n$;
2. Flip all the remaining directed edges of the graph $F[\pi^*]$; let $R[\pi^*]$ be the resulting tree and let $s = v_0, v_1, v_2, \ldots, v_n, v_{n+1} = t$ be the nodes of $R[\pi^*]$;
3. Make first the binary tree $R[\pi^*]$ rooted at node $s = v_0$ and ordered, and then perform inorder traversal on $R[\pi^*]$;
4. List the nodes $s = v_0, v_1, v_2, \ldots, v_{n+1}$ of the tree $R[\pi^*]$ by the order in which they are visited; remove from the list the root node and let $v'_1, v'_2, \ldots, v'_{n+1}$ be the resulting list;

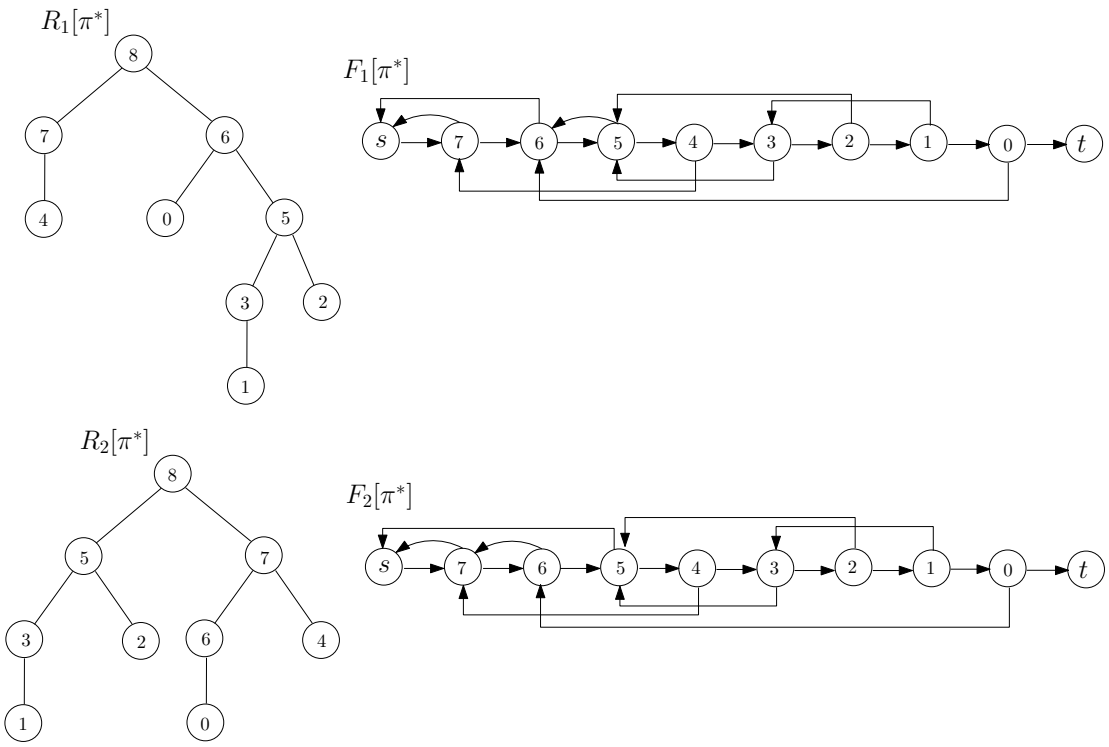

Figure 3: Two RPG-trees $R_1[\pi^*]$ and $R_2[\pi^*]$ and the corresponding reducible permutation graphs $F_1[\pi^*]$ and $F_2[\pi^*]$, respectively, produced by the algorithm `Encode_Cotree.to.RPG`.

5. Compute the pairs $(v'_1, v'_2), (v'_3, v'_4), \ldots, (v'_n, v'_{n+1})$ and remove the pair (v'_i, v'_{i+1}) for which $v'_i = 0$; let n' be the remaining pairs;

6. Construct the identity permutation on $N_{2n'}$, and then compute a self-inverting permutation π using the pairs of step 5; return $\pi^* = \pi$;

Let us now describe the decoding process using the two RPGs $F_1[\pi^*]$ and $F_2[\pi^*]$ of Figure 3. Let $F_1[\pi^*]$ be the input of the algorithm `Decode_RPG.to.SiP`. It first computes the rooted ordered tree $R_1[\pi^*]$ and then computes the inorder sequence $I_1 = (4, 7, 8, 0, 6, 1, 3, 5, 2)$; it deletes the value 8 of the root resulting the sequence $I'_1 = (4, 7, 0, 6, 1, 3, 5, 2)$; then the algorithm computes the pairs $C_1 = \{(4, 7), (0, 6), (1, 3), (5, 2)\}$ and removes the pair $(0, 6)$ from C_1. Then, it takes the identity permutation $\pi_I = (1, 2, 3, 4, 5, 6, 7)$ and using the pairs $(4, 7), (1, 3), (5, 2)$ as 2-cycles produces the SiP $\pi^* = (3, 5, 1, 7, 2, 6, 4)$.

If the algorithm takes as input the RPG $F_1[\pi^*]$ then $I_1 = (1, 3, 5, 2, 8, 0, 6, 7, 4)$ and thus $I'_1 = (1, 3, 5, 2, 0, 6, 7, 4)$. It is easy to see that the 2-cycles in C_2 are $(1, 3), (5, 2), (7, 4)$, and thus the identity permutation π_I becomes $\pi^* = (3, 5, 1, 7, 2, 6, 4)$.

Theorem 2. *Let $T[\pi^*]$ be a cotree which encodes the self-inverting permutation π^* and let $F[\pi^*]$ be a reducible permutation flow-graph of size $O(n)$ produced by the algorithm* `Encode_Cotree.to.RPG`*. The algorithm* `Decode_RPG.to.SIP` *extracts the permutation π^* from the flow-graph $F[\pi^*]$ in $O(n)$ time and space.*

CONCLUDING REMARKS

An interesting open question is whether the approach and techniques used in this paper, also in [3, 4], can help develop efficient codec algorithms and graph structures having "better" properties with respect to resilience, size, and/or time and space efficiency; we leave it as an open problem for future investigation.

REFERENCES

[1] A. Bretscher, D. Corneil, M. Habib, and C. Paul, "A simple linear time LexBFS cograph recognition algorithm," SIAM J. Discrete Math. 22 (2008) 1277–1296.

[2] M. Chroni and S.D. Nikolopoulos, "Encoding watermark integers as self-inverting permutations," 11th Int'l Conference on Computer Systems and Technologies (CompSysTech'10), ACM ICPS 471, 2010, pp. 125–130.

[3] M. Chroni and S.D. Nikolopoulos, "Encoding watermark numbers as cographs using self-inverting permutations," 12th Int'l Conference on Computer Systems and Technologies (CompSysTech'11), ACM ICPS 578, 2011, pp. 142–148.

[4] M. Chroni and S.D. Nikolopoulos, "An embedding graph-based model for software watermarking," 8th Int'l Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'12), IEEE Proceedings (to appear), 2012.

[5] C. Collberg and J. Nagra, "Surreptitious Software," Addison-Wesley (2010).

[6] C. Collberg, E. Carter, S. Kobourov, and C. Thomborson, "Error-correcting graphs for software watermarking," Proc. 29th Workshop on Graphs in Computer Science (WG'03), LNCS 2880, 2003, pp. 156–167.

[7] C. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp, "More on graph theoretic software watermarks: Implementation, analysis, and attacks," Information and Software Technology 51 (2009) 56–67.

[8] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99), 1999, pp. 311–324.

[9] D.G. Corneil, Y. Perl, and L.K. Stewart, "A linear recognition algorithm for cographs," SIAM J. Comput. 14 (1985) 926–984.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd edition), MIT Press, Inc., 2001.

[11] R.L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," US Patent 5.559.884, Microsoft Corporation, 1996.

[12] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York (1980); Second edition, Annals of Discrete Math. 57, Elsevier (2004).

[13] D. Grover, "The Protection of Computer Software - Its Technology and Applications," Cambridge University Press, New York (1997).

[14] H. Lerchs, "On cliques and kernels," Department of Computer Science, University of Toronto, March 1971.

[15] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," Electronic Commerce Research 6 (2006) 155–171.

[16] J. Nagra and C. Thomborson, "Threading software watermarks," Proc. 6th Int'l Workshop on Information Hiding (IH'04), LNCS 3200, 2004, pp. 208–223.

[17] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," Proc. 4th Int'l Information Hiding Workshop (IH'01), LNCS 2137, 2001, pp. 157–168.

[18] L. Zhang, Y. Yang, X. Niu, and S. Niu, "A survey on software watermarking," Journal of Software 14 (2003) 268–277.

[19] M.S. Hecht and J.D. Ullman, "Flow graph reducibility," SIAM J. Computing 1, pp. 188–202 (1972).

[20] M.S. Hecht and J.D. Ullman, "Characterizations of reducible flow graphs," Journal of the ACM 21, pp. 367–375 (1974).

ABOUT THE AUTHORS

Maria Chroni, MSc, PhD Candidate, Department of Computer Science, University of Ioannina, Phone: +30-265-100-8832, e-mail: mchroni@cs.uoi.gr

Stavros D. Nikolopoulos, PhD, Professor, Department of Computer Science, University of Ioannina, Phone: +30-265-100-8801, e-mail: stavros@cs.uoi.gr