Algorithm and Experiments in Testing Planar Graphs for Isomorphism

Jacek P. Kukluk

Lawrence B. Holder   Diane J. Cook

Computer Science and Engineering Department
University of Texas at Arlington
http://ailab.uta.edu/subdue/
{kukluk, holder, cook}@cse.uta.edu

Abstract

We give an algorithm for isomorphism testing of planar graphs suitable
for practical implementation. The algorithm is based on the decomposi-
tion of a graph into biconnected components and further into SPQR-trees.
We provide a proof of the algorithm's correctness and a complexity analy-
sis. We determine the conditions in which the implemented algorithm out-
performs other graph matchers, which do not impose topological restric-
tions on graphs. We report experiments with our planar graph matcher
tested against McKay's, Ullmann's, and SUBDUE's (a graph-based data
mining system) graph matchers.

Article Type	Communicated by	Submitted	Revised
regular paper	Giuseppe Liotta	September 2003	February 2005

1   Introduction

Presently there is no known polynomial time algorithm for testing if two general graphs are isomorphic [30, 31, 23, 43, 13]. The complexity of known algorithms are $O(n!n^3)$ Ullmann [47, 12] and $O(n!n)$ Schmidt and Druffel [44]. Reduction of the complexity can be achieved with randomized algorithms at a cost of a probable failure. Babai and Kŭcera [4], for instance, discuss the construction of canonical labelling of graphs in linear average time. Their method of constructing canonical labelling can assist in isomorphism testing with $\exp(-cn \log n / \log \log n)$ probability of failure. For other fast solutions researchers turned to algorithms which work on graphs with imposed restrictions. For instance, Galil et al. [21] discuss an $O(n^3 \log n)$ algorithm for graphs with at most three edges incident with every vertex. These restrictions limit the application in practical problems. We recognize planar graphs as a large class for which fast isomorphism checking could find practical use.

The motivation was to see if a planar graph matcher can be used to improve graph data mining systems. Several of those systems extensively use isomorphism testing. Kuramochi and Karypis [32] implemented the FSG system for finding all frequent subgraphs in large graph databases. SUBDUE [11, 10] is another knowledge discovery system, which uses labeled graphs to represent data. SUBDUE is also looking for frequent subgraphs. The algorithm starts by finding all vertices with the same label. SUBDUE maintains a linked list of the best subgraphs found so far in computations. Yan and Han introduced gSpan [51], which does not require candidate generation to discover frequent substructures. The authors combine depth first search and lexicographic order in their algorithm.

While the input graph to these systems may not be planar, many of the isomorphism tests involve subgraphs that are planar. Since planarity can be tested in linear time [27, 7, 8], we were interested in understanding if introducing planarity testing followed by planar isomorphism testing would improve the performance of graph data mining systems.

Planar graph isomorphism appeared especially interesting after Hopcroft and Wong published a paper pointing at the possibility of a linear time algorithm [28]. In their conclusions the authors emphasized the theoretical character of their paper. They also indicated a very large constant for their algorithm. Our work takes a practical approach. The interest is in an algorithm for testing planar graph isomorphism which could find practical implementation. We want to know if such an implementation can outperform graph matchers designed for general graphs and in what circumstances. Although planar isomorphism testing has been addressed several times theoretically [25, 28, 19], even a parallel version [22, 29, 42], to our knowledge, no planar graph matcher implementation existed. The reason might be due to complexity. The linear time implementation of embedding and decomposition of planar graphs into triconnected components was only recently made available. In this paper, we describe our implementation of a planar graph isomorphism algorithm of complexity $O(n^2)$. This might be a step toward achieving the theoretical linear time bound described by Hopcroft

and Wong. The performance of the implemented algorithm is compared with Ullmann's [47], McKay's [38], and SUBDUE's [11, 10] general graph matcher.

In our algorithm, we follow many of the ideas given by Hopcroft and Tarjan [25, 26, 46]. Our algorithm works on planar connected, undirected, and unlabeled graphs. We first test if a pair of graphs is planar. In order to compare two planar graphs for isomorphism, we construct a unique code for every graph. If those codes are the same, the graphs are isomorphic. Constructing the code starts from decomposition of a graph into biconnected components. This decomposition creates a tree of biconnected components. First, the unique codes are computed for the leaves of this tree. The algorithm progresses in iterations towards the center of the biconnected tree. The code for the center vertex is the unique code for the planar graph. Computing the code for biconnected components requires further decomposition into triconnected components. These components are kept in the structure called the SPQR-trees [17]. Code construction for the SPQR-trees starts from the center of a tree and progresses recursively towards the leaves.

In the next section, we give definitions and Weinberg's [48] concept of constructing codes for triconnected graphs. Subsequent sections present the algorithm for constructing unique codes for planar graphs by introducing code construction for biconnected components and their decomposition to SPQR-trees. Lastly, we present experiments and discuss conclusions. An appendix contains detailed pseudocode, a description of the algorithm, a proof of uniqueness of the code, and a complexity analysis.

2   Definitions and Related Concepts

2.1   Isomorphism of Graphs with Topological Restrictions

Graphs with imposed restrictions can be tested for isomorphism with much smaller computational complexity than general graphs. Trees can be tested in linear time [3]. If each of the vertices of a graph can be associated with an interval on the line, such that two vertices are adjacent when corresponding intervals intersect, we call this graph an interval graph. Testing interval graphs for isomorphism takes $O(|V| + |E|)$ [34]. Isomorphism tests of maximal outerplanar graphs takes linear time [6]. Testing graphs with at most three edges incident on every vertex takes $O(n^3 \log n)$ time [21]. The strongly regular graphs are the graphs with parameters (n, k, λ, μ), such that

1. n is the number of vertices,
2. each vertex has degree k,
3. each pair of neighbors have λ common neighbors,
4. each pair of non-neighbors have μ common neighbors.

The upper bound for the isomorphism test of strongly regular graphs is $n^{(O(n^{1/3} \log n))}$ [45]. If the degree of every vertex in the graph is bounded, theoretically, the graph can be tested for isomorphism in polynomial time [35]. If

the order of the edges around every vertex is enforced within a graph, the graph can be tested for isomorphism in $O(n^2)$ time [31]. The subgraph isomorphism problem was also studied on graphs with topological restrictions. For example, we can find in $O(n^{k+2})$ time if k-connected partial k-tree is isomorphic to a subgraph of another partial k-tree [14]. We focus in this paper on planar graphs. Theoretical research [28] indicates a linear time complexity for testing planar graphs isomorphism.

2.2   Definitions

All the concepts presented in the paper refer to an unlabeled graph $G = (V, E)$ where V is the set of unlabeled vertices and E is the set of unlabeled edges. An *articulation point* is a vertex in a connected graph that when removed from the graph makes it disconnected. A *biconnected graph* is a connected graph without articulation points. A *separation pair* contains two vertices that when removed make the graph disconnected. A *triconnected graph* is a graph without separation pairs. An *embedding* of a planar graph is an order of edges around every vertex in a graph which allows the graph to be drawn on a plane without any two edges crossed. A *code* is a sequence of integers. Throughout the paper we use a code to represent a graph. We also assign a code to an edge or a vertex of a graph. Two codes $C^A = [x_1^A, \ldots, x_i^A, \ldots, x_{na}^A]$ and $C^B = [x_1^B, \ldots, x_i^B, \ldots, x_{nb}^B]$ are equal if and only if they are the same length and for all i, $x_i^A = x_i^B$. *Sorting codes (sorted codes)* $C^A, C^B, \ldots, C^Z$ means to rearrange their order lexicographically (like words in a dictionary). For the convenience of our implementation, while sorting codes, we place short codes before long codes.

Let G be an undirected planar graph and $u_{a(1)}, \ldots, u_{a(n)}$ be the articulation points of G. Articulation points split G into biconnected subgraphs $G_1, \ldots, G_k$. Each subgraph G_i shares one articulation point $u_{a(i)}$ with some other subgraph G_j. Let *biconnected tree* T be a tree made from two kinds of nodes: (1) biconnected nodes $B_1, \ldots, B_k$ that correspond to biconnected subgraphs and (2) articulation nodes $v_{a(1)}, \ldots, v_{a(n)}$ that correspond to articulation points. An articulation node $v_{a(i)}$ is adjacent to biconnected nodes $B_l, \ldots, B_m$ if corresponding subgraphs $B_l, \ldots, B_m$ of G share common articulation point $u_{a(i)}$.

2.3   Two Unique Embeddings of Triconnected Graphs

Due to the work of Whitney [50], every triconnected graph has two unique embeddings. For example Fig. 1 presents two embeddings of a triconnected graph. The graph in Fig. 1(b) is a mirror image of the graph from Fig. 1(a). The order of edges around every vertex in Fig. 1(b) is the reverse of the order of Fig. 1(a). There are no other embeddings of the graph from Fig. 1(a).

2.4   Weinberg's Code

In 1966, Weinberg [48] presented an $O(n^2)$ algorithm for testing isomorphism of planar triconnected graphs. The algorithm associates with every edge a code.
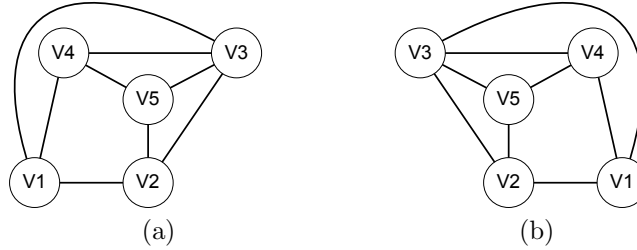
Figure 1: Two unique embeddings of the triconnected graph.

It starts by replacing every edge with two directed edges in opposite directions to each other as shown in Fig. 2(a). This ensures an even number of edges incident on every vertex and according to Euler's theorem, every edge can be traversed exactly once in a tour that starts and finishes at the same vertex. During the tour we enter a vertex on an input edge and leave on the output edge. The *first edge to the right* of the input edge is the first edge you encounter in a counterclockwise direction from the input edge. Since we commit to only one of the two embeddings, this first edge to the right is determined without ambiguity. In the data structures the embedding is represented as an adjacency list such that the counterclockwise order of edges around a vertex corresponds to the sequence they appear in the list; the first edge to the right means to take the consecutive edge from the adjacency list. During the tour, every newly-visited vertex is assigned a consecutive natural number. This number is concatenated to the list of numbers creating a code. If a visited vertex is encountered, an existing integer is added to the code. The tour is performed according to three rules:

1. When a new vertex is reached, exit this vertex on the first edge to the right.
2. When a previously visited vertex is encountered on an edge for which the reverse edge was not used, exit along the reverse edge.
3. When a previously visited vertex is reached on an edge for which the reverse edge was already used, exit the vertex on the first unused edge to the right.

The example of constructing a code for edge e_1 is shown in Fig. 2. For every directed edge in the graph two codes can be constructed which correspond to two unique embeddings of the triconnected graph. Replacing steps 2) and 3) of the tour rules from going "right" to going "left" gives the code for the second embedding given in Fig. 2(b). A *code going right (code right)* denotes for us a code created on a triconnected graph according to Weinberg's rules with every new vertex exiting on the first edge to the right and every visited vertex reached on an edge for which the reverse edge was already used on a first

unused edge to the right. Accordingly, we exit mentioned vertices to the left while constructing *code going left (code left)*. Constructing code right and code left on a triconnected graph gives two codes that are the same as the two codes constructed using only code right rules for an embedding of a triconnected graph and an embedding of a mirror image of this graph. Having constructed codes for all edges, they can be sorted lexicographically. Every planar triconnected graph with m edges and n vertices has $4m$ codes of length $2m+1$ [48]. Since the graph is planar m does not exceed $3n - 6$. Every entry of the code is an integer in the range from 1 to n. We can store codes in matrix M. Using Radix Sort with linear Counting Sort to sort codes in each row, we achieve $O(n^2)$ time for lexicographic sorting. The smallest code (the first row in M) uniquely represents the triconnected graph and can be used for isomorphism testing with another triconnected graph with a code constructed according to the same rules.
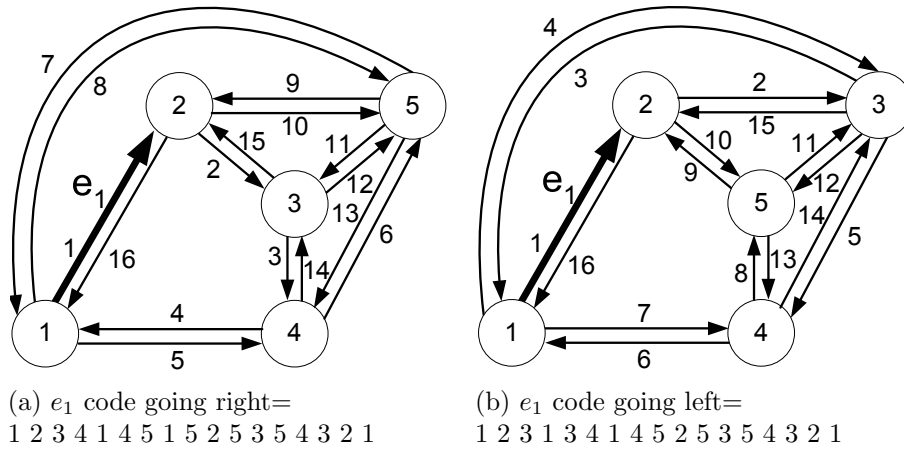

(a) e_1 code going right=
1 2 3 4 1 4 5 1 5 2 5 3 5 4 3 2 1

(b) e_1 code going left=
1 2 3 1 3 4 1 4 5 2 5 3 5 4 3 2 1

Figure 2: Weinberg's method of code construction for selected edge of the triconnected planar graph.

2.5   SPQR-trees

The data structure known as the SPQR-trees is a modification of Hopcroft and Tarjan's algorithm for decomposing a graph into triconnected components [26]. SPQR-trees have been used in graph drawing [49], planarity testing [15], and in counting embeddings of planar graphs [18]. They can also be used to construct a unique code for planar biconnected graphs. SPQR-trees decompose a biconnected graph with respect to its triconnected components. In our implementation we applied a version of this algorithm as described in [24, 1].

Introducing SPQR-trees, we follow the definition of Battista and Tamassia [17, 24, 49, 16]. Given a biconnected graph G, a *split pair* is a pair of vertices

$\{u, v\}$ of G that is either a separation pair or a pair of adjacent vertices of G. A *split component* of the split pair $\{u, v\}$ is either an edge $e = (u, v)$ or a maximal subgraph C of G such that $\{u, v\}$ is not a split pair of C (removing $\{u, v\}$ from C does not disconnect $C - \{u, v\}$). A *maximal split pair* of G with respect to split pair $\{s, t\}$ is such that, for any other split pair $\{u', v'\}$, vertices $u, v, s,$ and t are in the same split component. Edge $e = (s, t)$ of G is called a *reference edge*. The SPQR-trees $\mathcal{T}$ of G with respect to $e = (s, t)$ describes a recursive decomposition of G induced by its split pairs. $\mathcal{T}$ has nodes of four types S,P,Q, and R. Each node μ has an associated biconnected multigraph called the *skeleton* of μ. Tree $\mathcal{T}$ is recursively defined as follows

Trivial Case: If G consists of exactly two parallel edges between s and t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.

Parallel Case: If the spit pair $\{s, t\}$ has at least three split components $G_1, \ldots, G_k$ ($k \geq 3$), the root of $\mathcal{T}$ is a P-node μ, whose skeleton consists of k parallel edges $e = e_1, \ldots, e_k$ between s and t.

Series Case: Otherwise, the spit pair $\{s,t\}$ has exactly two split components, one of them is the reference edge e, and we denote the other split component by G'. If G' has cutvertices $c_1, \ldots, c_{k-1}$ ($k \geq 2$) that partition G into its blocks $G_1, \ldots, G_k$, in this order from s to t, the root of $\mathcal{T}$ is an S-node μ, whose skeleton is the cycle $e_0, e_1, \ldots, e_k$, where $e_0 = e$, $c_0 = s$, $c_k = t$, and $e_i = (c_{i-1}, c_i)$ $(i = 1, \ldots, k)$.

Rigid Case: If none of the above cases applies, let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to s, t ($k \geq 1$), and, for $i = 1, \ldots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing the reference edge $e = (s, t)$. The root of $\mathcal{T}$ is an R-node μ, whose skeleton is obtained from G by replacing each subgraph G_i with the edge $e_i = (s_i, t_i)$.

Several lemmas discussed in related papers are important to our topic. They are true for a biconnected graph G.

Lemma 2.1 [17]*Let μ be a node of $\mathcal{T}$. We have:*

- *If μ is an R-node, then skeleton(μ) is a triconnected graph.*
- *If μ is an S-node, then skeleton(μ) is a cycle.*
- *If μ is a P-node, then skeleton(μ) is a triconnected multigraph consisting of a bundle of multiple edges.*
- *If μ is a Q-node, then skeleton(μ) is a biconnected multigraph consisting of two multiple edges.*

Lemma 2.2 [17] *The skeletons of the nodes of SPQR-tree $\mathcal{T}$ are homeomorphic to subgraphs of G. Also, the union of the sets of spit pairs of the skeletons of the nodes of $\mathcal{T}$ is equal to the set of split pairs of G.*

Lemma 2.3 [36, 26] *The triconnected components of a graph G are unique.*

Lemma 2.4 [17] *Two S-nodes cannot be adjacent in $\mathcal{T}$. Two P-nodes cannot be adjacent in $\mathcal{T}$.*

Linear time implementation of SPQR-trees reported by Gutwenger and Mutzel [24] does not use Q-nodes. It distinguishes between real and virtual edges. A real edge of a skeleton is not associated with a child of a node and represents a Q-node. Skeleton edges associated with a P-, S-, or R-node are virtual edges. We use this implementation in our experiments and therefore we follow this approach in the paper.

The biconnected graph is decomposed into components of three types (Fig. 3): circles S, two vertex branches P, and triconnected graphs R. Every component can have real and virtual edges. Real edges are the ones found in the original graph. Virtual edges correspond to the part of the graph which is further decomposed. Every virtual edge, which represents further decomposition, has a corresponding virtual edge in another component. The components and the connections between them create an SPQR-trees with node type S, P, or R. The thick arrows in Fig. 3(c) are the edges of the SPQR-trees. Although the decomposition of a graph into an SPQR-trees starts from one of the graph's edges, no matter which edge is chosen, the same components will be created and the same association between virtual edges will be obtained (see discussion in the appendix). This uniqueness is an important feature that allows the extension of Weinberg's method of code construction for triconnected graphs to biconnected graphs and further to planar graphs. More details about SPQR-trees and their linear time construction can be found in [17, 24, 49, 16].

3   The Algorithm

Algorithm 1 Graph isomorphism and unique code construction for connected planar graphs

---
1: Test if $G1$ and $G2$ are planar graphs
2: Decompose $G1$ and $G2$ into biconnected components and construct the tree of biconnected components
3: Decompose each biconnected component into its triconnected components and construct the SPQR-tree.
4: Construct unique code for every SPQR-tree and in bottom-up fashion construct unique code for the biconnected tree
5: If $Code(G1) = Code(G2)$ $G1$ is isomorphic to $G2$

Algorithm 1 is a high level description of an algorithm for constructing a unique code for a planar graph and the use of this code in testing for isomorphism. For detailed algorithm, the proof of uniqueness of the code and complexity analysis refer to the appendix. Some of the steps rely on previously reported

solutions. They are: planarity testing, embedding, and decomposition into the SPQR-trees. Their fast solutions, developed over the years, are described in related research papers [39, 40, 24, 41]. This report focuses mostly on phases (4) and (5).

3.1   Unique Code for Biconnected Graphs

This section presents the unique code construction for biconnected graphs based on a decomposition into SPQR-trees. The idea of constructing a unique code for a biconnected graph represented by its SPQR-trees will be introduced using the example from Fig. 3(c). Fig. 3(a) is the original biconnected graph. This graph can be a part of a larger graph, as shown by the distinguished vertex V6. Vertex V6 is an articulation point that connects this biconnected component to the rest of the graph structure. Every edge in the graph is replaced with two directed edges in opposite directions. The decomposition of the graph from Fig. 3(a) contains six components: three of type S, two of type P and one of type R. Their associations create a tree shown in Fig. 3(b). In this example, the center of the tree can be uniquely identified. It can be done in two iterations. First, all nodes with only one incident edge are temporarily removed. They are $S3$, $S4$, and $R5$. Nodes $P1$ and $P2$ are the only ones with one edge incident. The second iteration will temporarily remove $P1$ and $P2$ from the tree. The one node left $S0$ is the center of the tree and therefore we choose it for the root and start our processing from it. In general, in the problem of finding the center of the tree, two nodes can be left after the last iteration. If the types of those two nodes differ, a rule can be established that sets the root node of the SPQR-trees to be, for instance, the one with type P before S and R. If S occurs together with R, S can always be chosen to be the root. For the nodes of type P as well as S, by Lemma 2.4, it is not possible that two nodes of the same type would be adjacent. However, for nodes of type R, it is possible to have two nodes of type R adjacent. In these circumstances, two cases need to be computed separately for each R node as a root.

The components after graph decomposition and associations of virtual edges are shown in Fig. 3(c). The thick arrows marked T_{ij} in Fig. 3(c) correspond to the SPQR branches from Fig 3(b). Their direction is determined by the root of the tree. Code construction starts from the root of the SPQR-trees. The component (skeleton) associated with node $S0$ has four real edges and four virtual edges. Four branches, T_{01}, Tr_{01}, T_{02}, and Tr_{02}, which are part of the SPQR-trees, show the association of $S0$'s virtual edges to the rest of the graph. Let the symbols T_{01}, Tr_{01}, T_{02}, and Tr_{02} also denote the codes that refer to virtual edges of $S0$. In the next step of the algorithm, those codes are requested. T_{01} points to the virtual edge of $P1$. All directed edges of $P1$ with the same direction as the virtual edge of $S0$ (i.e., from vertex $V2$ to vertex $V0$) are examined in order to give a code for T_{01}. There are two virtual edges of $P1$ directed from vertex 2 to vertex 0 that correspond to the further graph decomposition. They are identified by tails of T_{13} and T_{14}. Therefore, codes T_{13} and T_{14} must be computed before completing the code of T_{01}. T_{13} points to node $S3$. It is

(a) biconnected graph   (b) SPQR-tree

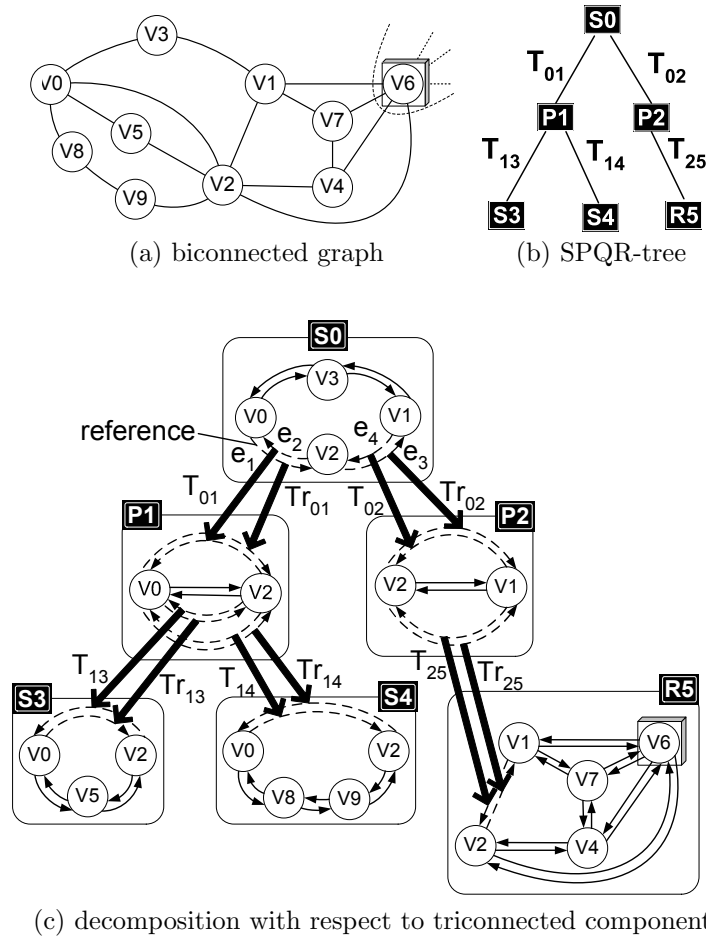(c) decomposition with respect to triconnected components

Figure 3: Decomposition of the biconnected graph with SPQR-trees.

a circle with three vertices and six edges, which is not further decomposed. If multi-edges are not allowed, $S0$ can be represented uniquely by the number of edges of $S3$'s skeleton. Since $S3$'s skeleton has 6 edges, its unique code can be given as $T_{13} =_S (\text{number of edges})_S =_S (6)_S$. Similarly $T_{14} =_S (8)_S$. Now the code for $P1$ can be determined. The $P1$ skeleton has eight edges, including six virtual edges. Therefore, $T_{01} =_P (8, 6, T_{13}, T_{14})_P$, where $T_{13} \leq T_{14}$. Applying the same recursive procedure to Tr_{01} gives $Tr_{01}=T_{01}=_P(8, 6, Tr_{13}, Tr_{14})_P$. Because of graph symmetry $T_{01} = Tr_{01}$. Codes T_{02} and Tr_{02} complete the list of four codes associated with four virtual edges of $S0$. The codes T_{02} and Tr_{02} contain the code for R node starting from symbol '$_R($' and finishing with '$)_R$'. The code of biconnected component $R5$ is computed according to Weinberg's procedure. In order to find T_{25}, codes for "going right" and "going left"

are found. Code going right of T_{25} is smaller than code going left therefore we select code going right. T_{25} and Tr_{25} are the same. The following integer numbers are assigned to vertices of $R5$ in the code going to the right of Tr_{25}: $V2 = 1$, $V1 = 2$, $V7 = 3$, $V4 = 4$, $V6 = 5$. The ' * ' after number 5 indicates that at this point we reached the articulation point (vertex $V6$) through which the biconnected graph is connected to the rest of graph structure. The codes associated with $S0$'s virtual edges after sorting:

$T_{01} = Tr_{01} = {}_P(8, \ 6, \ {}_S(6)_S, \ {}_S(8)_S)_P$
$T_{02} = Tr_{02} = {}_P(6, \ 4, \ {}_R(1\ 2\ 3\ 1\ 3\ 4\ 1\ 4\ 5^*\ 2\ 5\ 3\ 5\ 4\ 3\ 2\ 1)_R)_P$

First, we add the number of edges of $S0$ to the beginning of the code. There are eight edges. We need to select one edge from those eight. This edge will be the starting edge for building a unique code. Restricting our attention to virtual edges narrows the set of possible edges to four. Further we can restrict our attention to two virtual edges with the smallest codes (here based on the length of the code). Since T_{01} and Tr_{01} are equal and are the smallest among all codes associated with the virtual edges of $S0$, we do code construction for two cases. We traverse the $S0$ edges in the direction determined by the starting edge e_2 associated with tail of T_{01}, until we come back to the edge from which we began. The third and fourth edges of this tour are virtual edges. We introduce this information into a code adding numbers 3 and 4. Next, we add codes associated with virtual edges in the order they were visited. We have two candidate codes for the final code of the biconnected graph from our example:

$Code(e_1) = {}_S(8, 1, 4, \ Tr_{02}, \ Tr_{01})_S$
$Code(e_2) = {}_S(8, 3, 4, \ T_{02}, \ T_{01})_S$

We find that $Code(e_1) < Code(e_2)$, therefore e_1 is the reference and starting edge of the code. e_1 is also the unique edge within the biconnected graph from the example. $Code(e_1)$ is the unique code for the graph and can be used for isomorphism testing with another biconnected graph. The symbols '${}_P($', '$)_P$', '${}_S($', '$)_S$', '${}_R($', and '$)_R$' are integral part of the codes. They are organized in the order:

$$\text{`}{}_P(\text{'} < \text{`})_P\text{'} < \text{`}{}_S(\text{'} < \text{`})_S\text{'} < \text{`}{}_R(\text{'} < \text{`})_R\text{'}$$

In the implemented computer program these symbols were replaced by negative integers. Constructing a code for a general biconnected graph requires definitions for six cases. Three for S, P, and R nodes if they are root nodes and three if they are not. Those cases are described in Table 1.
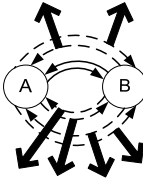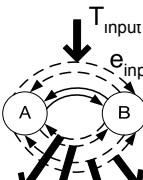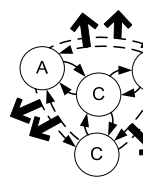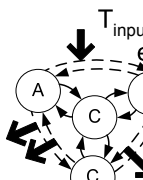
3.2   Unique Code for Planar Graphs

Fig. 4 shows a planar graph. The graph is decomposed into biconnected components in Fig. 5. Vertices inside rectangles are articulation points. Biconnected components are kept in a tree structure. Every articulation point can be split into many vertices that belong to biconnected components and one vertex that

Table 1: Code construction for root and non-root nodes of an SPQR-trees.

Type S

Root S node: Add the number of edges of S skeleton to the code. Find codes associated with all virtual edges. Choose an edge with the smallest code to be the starting reference edge. Go around the circle traversing the edges in the direction of the chosen edge, starting with the edge after it. Count the edges during the tour. If a virtual edge is encountered, record which edge it is in the tour and add this number to the code. After reaching the starting edge, the tour is complete. Concatenate the codes associated with traversed virtual edges to the code for the S root node in the order they were visited during the tour. There are cases when one starting edge cannot be selected, because there are several edges with the same smallest code. For every such edge, the above procedure will be repeated and several codes will be constructed for the S root node. The smallest among them will be the unique code. If the root node does not have virtual edges and articulation points, the code is simply $_S$(number of edges)$_S$. If at any point in a tour an articulation point is encountered, record at which edge in the tour it happened, and add this edge's number to the code marking it as an articulation point.

Non-root S node: Constructing a code for node type S, which is non-root node, differs from constructing an S root code in two aspects. (1) the way the starting reference edge is selected. In non-root nodes the starting edge is the one associated with the input (edge e_{input}). Given an input edge, there is only one code. There is no need to consider multiple cases. (2) Only virtual edges different from e_{input} are considered when concatenating the codes.

Table 1: (cont).

Type P

Root P node: Find the number of edges and number of virtual edges in the skeleton of P. Add number of edges to the code first and number of virtual edges second. If A and B are the skeleton's vertices, construct the code for all virtual edges in one direction, from A to B. Add codes of all virtual edges directed from A to B to the code of the P root node. Added codes should be in non-decreasing order. If A or B is an articulation point add a mark to the code indicating if articulation point is at the head or at the tail of the edge directed from A to B. Construct the second code following the direction from B to A. Compare the two codes. The smaller code is the code of P root node.

Non-root P node: Construct the code in the same way as for the root P node but only in one direction. The input edge determines the direction.

Type R

Root R node: For all virtual edges of an R root node, find the codes associated with them. Find the smallest code. Select all edges for which codes are equal to the smallest one. They are the starting edges. For every edge from this set construct a code according to Weinberg's procedure. Whenever a virtual edge is traversed, concatenate its code. For every edge, two cases are considered: "going right" and "going left". Finally, choose the smallest code to represent the R root node. If at any point in a tour an articulation point is encountered, mark this point in the code.

Non-root R node: Only two cases need to be considered ("going right" and "going left"), because the starting edge is found based on input edge to the node. Only virtual edges different from e_{input} are considered when concatenating the codes.

becomes a part of a biconnected tree as an articulation node (black vertices in Fig. 5). The biconnected tree from Fig. 5 has two types of vertices: biconnected components marked as $B0 - B9$ and articulation points, which connect vertices $B0 - B9$. For simplicity, our example contains only circles and branches as biconnected components. In general, they can be arbitrary planar biconnected graphs, which would be further decomposed into SPQR-trees.
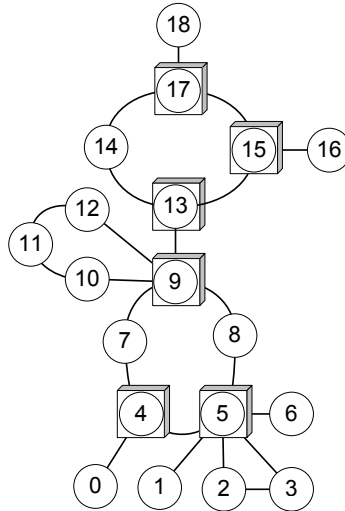

Figure 4: Planar graph with identified articulation points

Code construction for a planar graph begins from the leaves of a biconnected tree and progresses in iterations. We compute codes for all leaves, which are biconnected components in the first iteration. Leaves are easily identified because they have only one edge of a tree incident on them. Leaves can be deleted, and in the next iteration we compute the code for articulation points. Once we have codes for articulation points, the vertices can be deleted from the tree. We are left with a tree that has new leaves. They are again biconnected components. This time, codes found for articulation points are included into the codes for biconnected components. This inclusion reflects how components are connected to the rest of the graph through articulation points. In the last iteration only one vertex of the biconnected tree is left. It will be either an articulation point or a biconnected component. In general, trees can have a center containing one or two vertices, but a tree of biconnected components always has only one center vertex. Computing the code for this last vertex gives the unique code of a planar graph.

In the example given in Fig. 5(a) we identify the leaves of the tree first and find codes for them. They are: $B0$, $B1$, $B2$, $B3$, $B5$, $B8$, $B9$. Let those symbols also denote codes for those components. These codes include information about articulation points. For example, $B1=_B(_S(6^*)_S)_B$. '$_B($' and '$)_B$' mark the be-
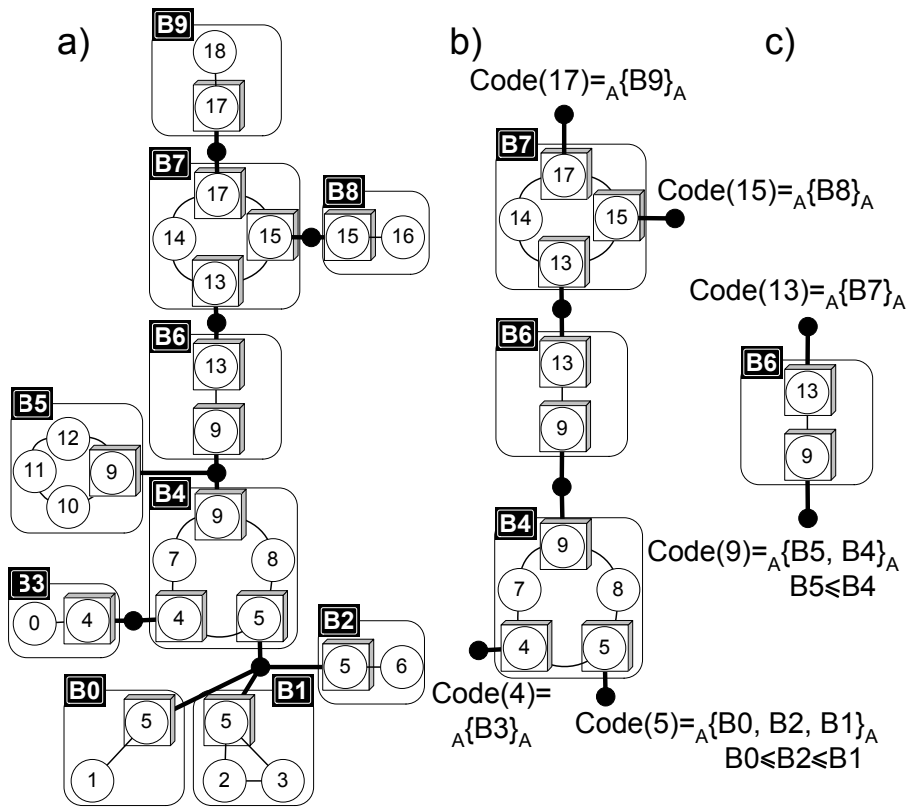
Figure 5: Constructing a unique code of a planar graph (a) the tree of biconnected components, (b) after the first iteration of the algorithm with leaves eliminated (c) before the last iteration of the algorithm

ginning and the end of a biconnected component code. $B1$ contains only a circle with one articulation point. ' * ' denotes the articulation point, and 6 represents six edges in this component after replacing every edge in the graph with two edges in opposite directions. After codes for the leaves of the biconnected tree are computed, those vertices are no longer needed and can be deleted. Fig. 5(b) presents the remaining portion of the biconnected tree. Codes for articulation points 4, 5, 15, and 17 can be computed at this point. All codes of the vertices adjacent to a given articulation point are sorted and concatenated in nonde-creasing order. Codes for articulation points 15 and 17 are just the same codes as adjacent leaves $B8$ and $B9$ with symbol '$_A($' at the beginning and '$)_A$' at the end. The symbols '$_A($' and '$)_A$' together with '$_B($' and '$)_B$' add up to total eight control symbols. Their order is:

$$\text{'}_A\text{(' } < \text{ '}\text{)}_A\text{' } < \text{ '}_B\text{(' } < \text{ '}\text{)}_B\text{' } < \text{ '}_P\text{(' } < \text{ '}\text{)}_P\text{' } < \text{ '}_S\text{(' } < \text{ '}\text{)}_S\text{' } < \text{ '}_R\text{(' } < \text{ '}\text{)}_R\text{'}$$

Constructing $Code(5)$ requires sorting codes $B0$, $B1$, and $B2$ and concatenating them in the order $B0 \leq B2 \leq B1$. $Code(5)=_A(B0,\ B2,\ B1)_A$. The codes for articulation points 9 and 13 are not know, because not all necessary codes were found at this point. In the second iteration codes for $B4$ and $B7$ can be computed. $B4$ and $B7$ are circles, therefore the rules for creating codes of S root vertices from the preceding section apply. The previously found codes of articulation points must be included in the newly created codes of $B4$ and $B7$. $B4$'s skeleton has 10 edges, therefore we place the number 10 after the symbol '$_S($'. The reference edge, selected based on the smallest code in $B4$'s skeleton, is the edge directed from vertex 8 to vertex 9. The very first vertex after the reference edge is an articulation point (vertex 9). This adds the number 1 with a ' * ' to the code since this articulation point does not have any code associated with it. After traversing three edges in the direction determined by the reference edge, we find another articulation point (vertex 4). Number 3 is placed next in the $B4$ code and is followed by the code of the encountered articulation point. The next articulation point (vertex 5) is fourth in the tour, so we concatenate the number 4 and the code for this articulation point, which completes the code for $B4$. The $B7$ code can be found in a similar way. $B4$ and $B7$ are:

$B4=_B(_S(10,\ 1^*,\ 3,\ _A(B3)_A,\ 4,\ _A(B0,\ B2,B1)_A)_S)_B$
$B7=_B(_S(8,\ 1^*,\ 2,\ _A(B8)_A,\ 3,\ _A(B9)_A)_S)_B$

In the next iteration we compute codes for articulation points, vertices 13 and 9. This step is the same as the previous one where codes for articulation points with vertices 4, 5, 16, and 17 were computed. The codes are:

$Code(9)=_A(B7)_A$
$Code(13)=_A(B5,B4)_A,\ B5 \leq B4$

After this step, the graph from our example is reduced to one biconnected component shown in Fig. 5(c). The code of $B6$ is the final code that uniquely represents the graph from our example. The undirected edge between vertices 9 and 13 is the unique edge of this graph. The $B6$ code can be used for testing the

graph for isomorphism with another planar graph. Given the order of control symbols we find that $_A(B7)_A \leq\ _A(B5,\ B4)_A$, therefore the final planar graph code is

$B6 =_P(2,\ _A(B7)_A,\ _A(B5,\ B4)_A)_P=$
$=_P(2,\ _A(B(_S(8,\ 1^*,\ 2,\ _A(B(_P(2^*)_P)_B)_A,\ 3,\ _A(B(_P(2^*)_P)_B)_A)_S)_B)_A,$
$_A(B(_S(8^*)_S)_B),\ _B(_S(10,\ 1^*,\ 3,\ _A(B(_P(2^*)_P)_B)_A,\ 4,\ _A(B(_P(2^*)_P)_B,$
$_B(_P(2^*)_P)_B,\ _B(_S(6^*)_S)_B)_A)_S)_B)_A)_P$

The presented method of code construction for planar graphs will produce the same codes for all isomorphic graphs and different codes for non-isomorphic graphs. The correctness results from the uniqueness of decomposition of a planar graph into biconnected components and biconnected components into SPQR-trees. Two isomorphic biconnected graphs will have the same SPQR-trees. If additionally all the skeletons of corresponding nodes of SPQR-trees are the same and preserve the same connections between virtual edges, than the graphs represented by those trees are isomorphic. Similarly, two isomorphic planar graphs will have the same biconnected tree. If the corresponding biconnected components of this tree are isomorphic and the connection of them to articulation points is preserved, the two planar graphs are isomorphic. For the proof see the Appendix.

4   Experiments

The purpose of the experiments is to compare the planar graph matcher described in this paper with other graph matching systems. Three of them, which do not impose topological constraints, were selected:

1. The SUBDUE Graph Matcher [11, 10] developed based on Bunke's algorithm [9]. This graph matcher is a part of the SUBDUE data mining system and has a wide range of options. It can perform exact and inexact graph matches on graphs with labeled vertices and edges. If the graphs are non-isomorphic the program can return the lowest matching cost (cost is the number of edges and vertices that must be removed from one graph in order to make the two graphs isomorphic).

2. Ullmann's algorithm, which has an established reputation and was used as a reference in many studies about isomorphism and operates on general graphs. We used the implementation developed by [20, 12].

3. McKay's Nauty graph matcher [38] was of particular interest because of its reputation as the fastest available isomorphism algorithm. McKay's Nauty graph matcher can test general graphs for isomorphism.

A desktop computer with a Pentium IV, 1700 MHz processor and 512 MB RAM was used in the experiments. The tests were conducted on isomorphic

and non-isomorphic pairs of planar, undirected, unlabeled graphs. In all experiments involving a planar graph matcher, the time spent for planarity test was included in the total time used by the planar graph matcher. In order to evaluate general properties of the graph matchers with respect to computation time, a vast number of graphs were generated. We used LEDA [41] functions that allow for generation of a planar graph with specified number of vertices and edges.

In Fig. 6 we show the average computation time versus the number of edges for planar graphs with 20, 50 and 80 vertices. McKay's, Ullmann's, SUBDUE, and the planar graph matcher are compared. The results in Fig. 6 were found based on one thousand isomorphic pairs of randomly generated, connected planar graphs. The number of edges of every generated graph was also random. Graphs were generated in the range of edges from $|V|$-1 to $3|V|$-6. This range was divided into 17 intervals. Every point marked in Fig. 6 represents average computation time within one of the 17 intervals. The two vertical arrows in Fig. 6 indicate points where Ullmann's algorithm is 20 times slower than McKay's and the planar graph matcher is 400 times slower. The planar graph matcher was outperformed by three other general graph matchers on planar graphs with 20 vertices. The average code length of the 1000 graphs with 20 vertices used in the experiment was 195 symbols.

Comparing computation time for isomorphic and non-isomorphic graphs in Fig. 6, we observe a significant drop in computation time for non-isomorphic graphs while using Ullmann's algorithm. We do not observe such differences for testing non-isomorphic and isomorphic graphs when we use McKay, SUBDUE or planar graph matcher. The runtime of McKay's graph matcher decreases with an increasing number of edges in all experiments in Fig. 6 and 7. This is due to two reasons [37]. First, major computation time of Mckay's graph matcher is spent on determining the automorphism group of a graph. There are fewer automorphisms as we approach the upper limit on the number of edges of planar graphs $3|V| - 6$, and therefore faster computation time. Second, Mckay's graph matcher is optimized for dense graphs in many of its components.

We excluded SUBDUE from experiments on graphs bigger than 20 vertices and Ullmann's graph matchers from experiments on graphs bigger than 80 vertices, because their testing time was too long. In Fig. 7 we compare testing time of McKay's and the planar graph matcher with 200, 1000, and 3000 vertices. In each of these three cases one thousand randomly generated planar, connected graphs were used in the experiment. We present results only for isomorphic graphs because we consider them to be the hardest, resulting in the longest computation time. Fig. 7(a) shows the execution time measured for every pair of graphs tested for isomorphism. Fig. 7(b) gives the average of the values from Fig. 7(a). McKay's graph matcher is faster as the number of edges in the graph increases, and it performs especially well on dense planar graphs. When examining the execution time of the planar graph matcher in Fig. 7(a), we find that there is a minimum time (about one second) required to check two graphs for isomorphism by the planar graph matcher. We do not observe any cases with smaller execution time. This minimum time represents cases of the graphs
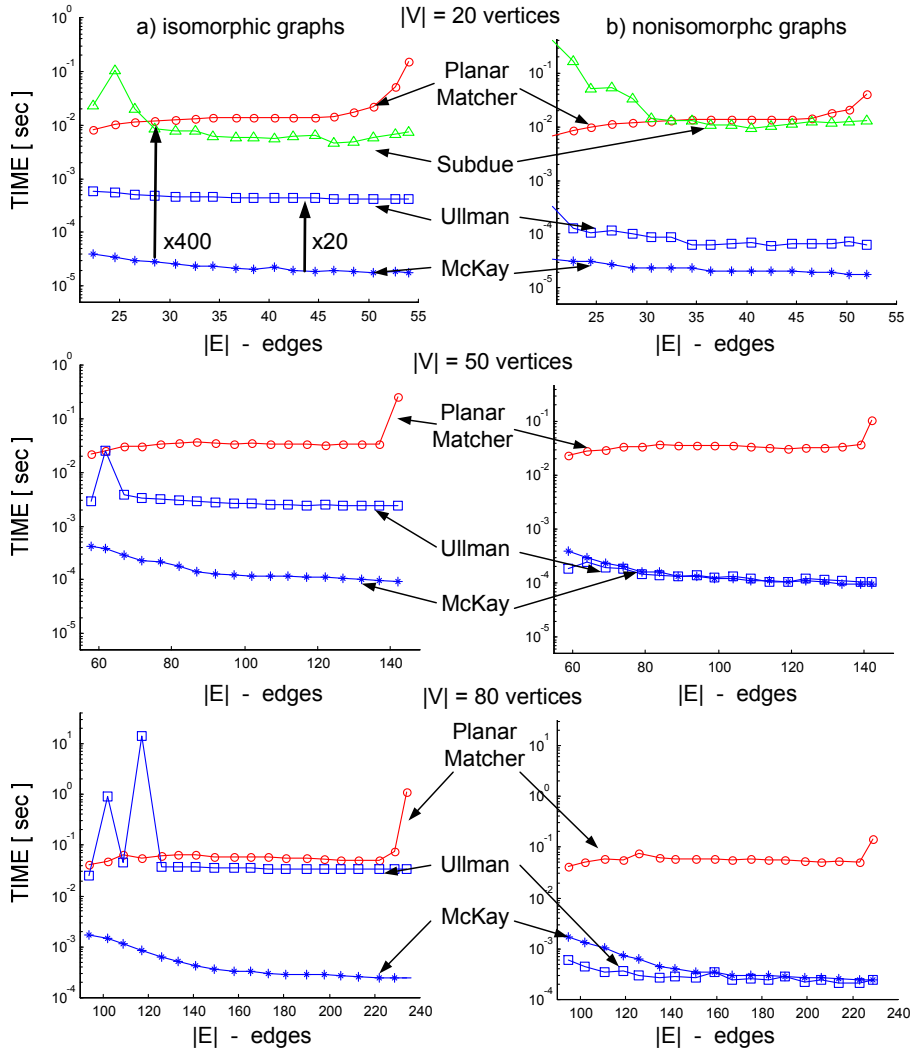
Figure 6: Average execution time of three general graph matchers and our planar graph matcher for testing isomorphism of planar graphs with 20, 50 and 80 vertices.
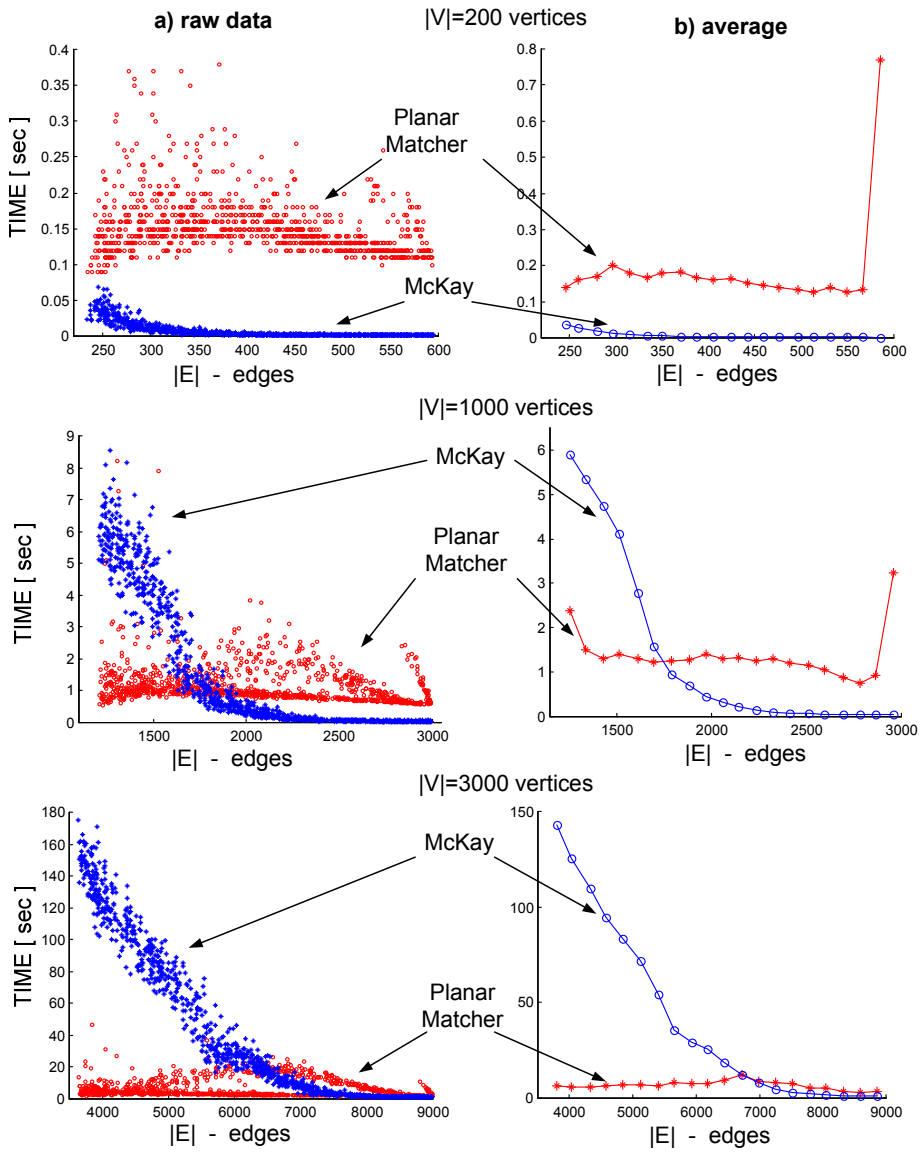
Figure 7: Execution time of McKay's and Planar Graph matcher: (a) raw data (b) average time.

tested for isomorphism in linear time. This time is spent on planarity test, decomposition into biconnected components, decomposition into SPQR-trees, and for the most part, for the construction of the code that represents the graph. Code construction is computationally very costly if computations start from a triconnected root node or if the graph is triconnected and cannot be decomposed further. These cases are more frequent as the number of edges in planar graph approaches $|E| = 3|V| - 6$. We apply Weinberg's [48] procedure of $O(n^2)$ complexity to these cases. It results in significant increase in computation time for dense planar graphs observed both in Fig. 6 and Fig. 7.

In Table 2 we collect average computation time for pairs of graphs with 10, 20, 50 and 80 vertices, both isomorphic and non-isomorphic. Table 3 gives average time for isomorphic pairs of graphs with 200, 500, 1000, 2000 and 3000 vertices. Every entry in Table 2 and Table 3 is computed based on 1000 graphs. Number of edges for every graph is found randomly from the range $|V| - 1 \leq |E| \leq 3|V| - 6$.

Table 2: Average time of testing isomorphic (left columns) and non-isomorphic (right columns) planar graphs with $|V| \leq 80$ vertices. Every entry in the table is found from 1000 pairs of graphs.

	$|V|$=10 [ms]		$|V|$=20 [ms]		$|V|$=50 [ms]		$|V|$=80 [ms]	
McKay	0.01	0.01	0.02	0.02	0.17	0.57	0.56	0.57
Ullmann	0.13	0.05	0.46	0.08	5.99	0.23	1453.12	0.51
SUBDUE	0.68	1.00	14.07	35.31	-	-	-	-
Planar	10.33	7.30	19.92	13.87	44.08	33.44	67.52	59.26

Table 3: Average time of testing isomorphic planar graphs with $|V| \geq 200$ vertices. Every entry in the table is found from 1000 pairs of graphs.

	$|V|$=200 [ms]	$|V|$=500 [ms]	$|V|$=1000 [ms]	$|V|$=2000 [ms]	$|V|$=3000 [ms]
McKay	0.01	0.01	0.02	0.02	0.17
Planar	10.33	7.30	19.92	13.87	44.08

Average time taken to test isomorphic graphs from these tables was used to plot Fig. 8. The isomorphism test time used by the planar graph matcher with graphs in the range of 10 to 3000 vertices increases almost linearly with number of vertices. On average, the planar graph matcher is faster than McKay's graph matcher on graphs with more than 800 vertices.

In Fig. 9 we present the most interesting results from our experiments. We identify the fastest graph matchers for planar graphs. We also identify planar graphs in terms of their number of vertices and number of edges for which those graph matchers outperformed all other solutions. The maximum number of edges in planar graphs is $3|V|-6$. The minimum number of edges of a connected
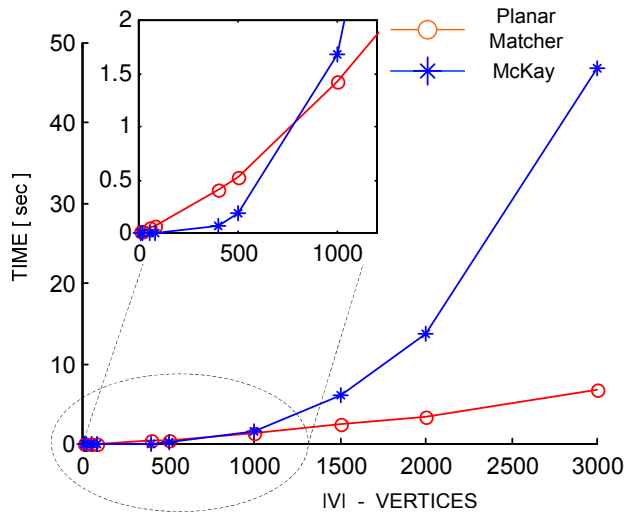
Figure 8: Average time (1000 pairs of isomorphic graphs for every point) of testing isomorphic pairs of planar graphs with McKay and the planar graph matcher.

graph is $|V|-1$. Therefore with $|E|$ edges on the horizontal axis and $|V|$ vertices on vertical axis we plot two lines $|E| = 3|V| - 6$ and $|E| = |V| - 1$. The region above the $|E| = |V|-1$ line represents disconnected graphs and the region below $|E| = 3|V| - 6$ represents non-planar graphs. The points between those two lines represent the planar graphs used in our experiments. 7000 pairs of planar graphs (1000 for every number of vertices $|V|$=200, 400, 500, 1000, 1500, 2000, 3000) were used to determine the regions in which the planar graph matcher or McKay's graph matcher is faster on average. Those regions are identified in Fig. 9. The average execution time was computed in the same way as in the experiment for which the results are displayed in Fig. 7. The circles in Fig. 9 show the points for which the average computation time was determined and the planar graph matcher outperformed McKay's graph matcher. The points marked with a star ' $*$ ' indicate the regions for which McKay's graph matcher was faster. From Fig. 9 we estimate that the planar graph matcher was faster than all other graph matchers for planar graphs with $|E| < \frac{1}{3.8}(|V| - 250)$.

5   Conclusions and Future Work

We attempted to practically verify very promising theoretical achievements in the problem of testing planar graphs for isomorphism. For this reason, we developed a computer program, which used a recently implemented linear algorithm for decomposing biconnected graphs with respect to its triconnected compo-
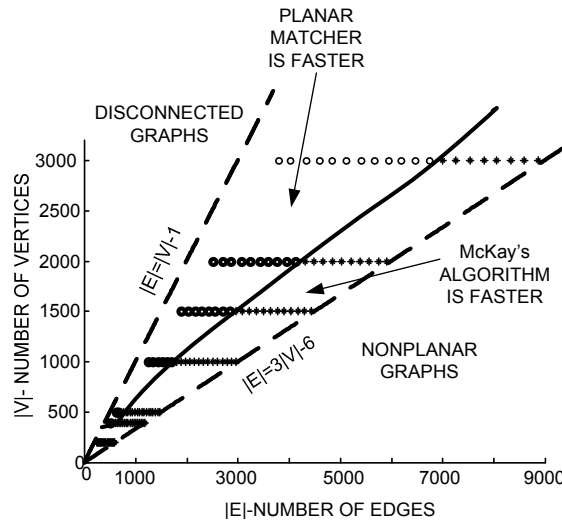
Figure 9: Identification of planar connected graphs with $|V|$ vertices and $|E|$ edges for which Planar Graph Isomorphism Algorithm outperforms (average computation time) McKay's graph matcher.

nents [24]. It is very likely that this is the first implementation that explores these planar graph properties.

Our main interest was to find out if the planar graph matcher could improve the efficiency of graph-based data mining systems. Those systems seldom perform isomorphism tests on graphs with numbers of vertices larger than 20. In this range, all three general graph matchers tested in our experiments were better than our planar graph matcher. We see some benefit of using the planar graph matcher over McKay's only for graphs with more than 1000 vertices. Even for such large planar graphs, our implementation was not better than McKay's matcher in the entire range of number of edges. We conclude that restriction to planar graphs in testing for isomorphism does not yet offer benefits that warrant the introduction of the planar graph matchers into graph-based data mining systems.

However, there is no doubt that faster solutions for testing planar graphs for isomorphism are possible and that the region, in which the planar graph matcher is the fastest, given in Fig. 9, can be made larger. If this region would reach small graphs in the range of 10 or 20, the conclusion about introducing the planar graph matcher to data mining systems would need to be revised. The planar graph matcher could also be made more applicable by extension to planar graphs with labels both on edges and on vertices. This, however, would require longer graph codes.

The result presented here might be particularly useful, if any application

would arise, which would require testing planar graphs for isomorphism with thousands of vertices. Electronic and Very Large Scale Integration (VLSI) circuits are examples [2] of such applications. The research in graph planarization [33] can extend the methods described here for isomorphism testing and unique code construction to larger classes of graphs than planar.

APPENDIX

A   The Algorithm

A.1   Pseudocode

The algorithm is divided into six parts (Algorithm 2-7) and ten procedures. The first procedure ISOMORPHISM-TEST receives two graphs, $G1$ and $G2$, computes codes for each of them, and compares the codes. Equal codes mean that $G1$ and $G2$ are isomorphic, unequal codes mean that they are not. Procedure FIND-PLANAR-CODE accepts a planar graph G and returns its code. First, G is decomposed into biconnected components (line 1). Biconnected tree T represents this decomposition. The body of the main while loop (lines 2-12) progresses iteratively finding the code associated with leaf nodes of T and articulation nodes. The loop at lines 3-5 finds codes for all biconnected components of G associated with the leaf nodes of T. The codes are stored in the code array C. C is indexed by T nodes. Every articulation point v_A adjacent to the leaf nodes of T is assigned a code at lines 6-10. Lines 7 and 9 mark in the code the beginning '$_A$(' and the end ')$_A$' of the code. Codes for articulation nodes are stored in an A array. When only one node (the center node) is left in the biconnected tree, the algorithm progresses to line 14. Lines 14-19 determine the final planar code. If the center node is an articulation node, the final planar code is retrieved from an A array. If the center node is a biconnected node, the final code of the biconnected node is computed in line 18. Line 20 returns the code of the planar graph.

Procedure FIND-BICONNECTED-CODE accepts a biconnected graph G and an array A, which contains codes associated with articulation points. All edges of G are replaced with two directed edges in opposite directions at line 1. Line 2 creates the SPQR-tree of G. Line 3 finds the center or two centers of the SPQR-tree. If there is only one center node, the code L_1 of G is computed at line 4 starting from this center node and we return L_1. If the SPQR-tree has two center nodes, the additional code L_2 is found at line 8 starting from the second center. Then, we return the smaller of L_1, L_2. Procedure FIND-BICONNECTED-CODE-FROM-ROOT recognizes the type of center nodes and calls procedures that compute the biconnected graph code using the SPQR-tree data structure with P-, S-, or R- nodes in the center.

Procedure CODE-OF-S-ROOT-NODE accepts the skeleton of a center S-node $skeleton(\mu)$, an array of codes associated with articulation points A, and an SPQR-tree $\mathcal{T}$. The loop at lines 1-3 uses the FIND-CODE procedure to find codes associated with virtual edges. These codes represent remaining portions of graph G adjacent to the S-center node. The parameter $twin\_edge\_of(e_V)$ is a virtual edge of the skeleton of a child of the center S-node. The loop at lines 4-23 creates code array CA. Each virtual edge of an S-node skeleton has its corresponding code in CA. Line 5 appends symbol '$_S$(' to the code indicating node type and the beginning of the code. Line 21 appends ')$_S$' indicating the end of the code. The internal loop at lines 9-21 traverses the skeleton circle and

---

Algorithm 2 Graph isomorphism and unique code construction for planar graphs

---

$G1$, $G2$ - graphs to be tested for isomorphism

ISOMORPHISM-TEST($G1, G2$)

1: **if** $G1$ and $G2$ are planar **then**
2:   $Code(G1)$=FIND-PLANAR-CODE($G1$)
3:   $Code(G2)$=FIND-PLANAR-CODE($G2$)
4:   **if** $Code(G1) = Code(G2)$ **then**
5:     **return** $G1$ is isomorphic to $G2$
6:   **else**
7:     **return** $G1$ and $G2$ are not isomorphic
8:   **end if**
9: **end if**

FIND-PLANAR-CODE(planar graph G)

T - a tree of biconnected components
A - articulation points code array
C - code array of biconnected components
B - array of biconnected components of G
A, B, C arrays are indexed by T nodes

1: Decompose G into biconnected components represented by tree T. Store the biconnected components in array B.
2: **while** number of nodes of $T > 1$ **do**
3:   **for** all leaf nodes $v_L \in T$, degree(v_L) = 1 **do**
4:     $C[v_L]$=FIND-BICONNECTED-CODE($A,B[v_L]$)
5:   **end for**
6:   **for** all articulation points $v_A \in T$ adjacent to leaf nodes of T **do**
7:     $A[v_A]$.append("$_A$(")
8:     from C concatenate in increasing order to $A[v_A]$ all leaf node codes adjacent to v_A
9:     $A[v_A]$.append(")$_A$")
10:   **end for**
11:   delete from T all leaves
12:   delete from T all articulation points with degree 1
13: **end while**
14: v= the remaining center node of T
15: **if** v is an articulation point **then**
16:   PlanarCode=$A[v]$
17: **else if** v represents biconnected component **then**
18:   PlanarCode=FIND-BICONNECTED-CODE($A,B[v]$)
19: **end if**
20: **return** PlanarCode

Algorithm 3 Constructing the unique code for biconnected graphs

---

FIND-BICONNECTED-CODE(articulation points code array A,
biconnected graph G)

$\mathcal{T}$ - SPQR-tree
$\{\mu_1, \mu_2\}$ - nodes in the center of an SPQR-tree
L_1, L_2 - codes of biconnected graph G starting from nodes μ_1, μ_2.

 1: make G bidirected
 2: create an SPQR-tree $\mathcal{T}$ of G
 3: $\{\mu_1, \mu_2\} = find\_center\_of\_tree(\mathcal{T})$
    {two center nodes $\{\mu_1, \mu_2\}$ appear only for symmetrical $\mathcal{T}$ tree with two
    R-nodes in the center, in all other cases we can find one center or eliminate
    the second node assigning order of preferences to S,P,R - nodes}
 4: L_1=FIND-BICONNECTED-CODES-FROM-ROOT(μ_1, A, $\mathcal{T}$)
 5: **if** $\mu_2 = NULL$ **then**
 6:    **return** L_1
 7: **else**
 8:    L_2=FIND-BICONNECTED-CODES-FROM-ROOT(μ_2, A, $\mathcal{T}$)
 9:    **return** FIND-THE-SMALLEST-CODE$\{L_1, L_2\}$
10: **end if**

FIND-BICONNECTED-CODES-FROM-ROOT(μ, A, $\mathcal{T}$)

L -code for biconnected graph starting from root node μ

 1: L.append("$_B$(")
 2: **if** $\mu = $ S node **then**
 3:    L=CODE-OF-S-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)
 4: **else if** $\mu = $ P node **then**
 5:    L=CODE-OF-P-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)
 6: **else if** $\mu = $ R node **then**
 7:    L=CODE-OF-R-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)
 8: **end if**
 9: L.append(")$_B$")
10: **return** L

Algorithm 4 Constructing the unique code for S-root node of $\mathcal{T}$

CODE-OF-S-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)

CV -array of codes associated with virtual edges
CA -code array

1: **for** all virtual edges e_V of $skeleton(\mu)$ including reverse edges **do**
2:   $\nu =$ the child of μ corresponding to virtual edge e_V
     $CV[e_V]$=FIND-CODE($twin\_edge\_of(e_V)$, skeleton(ν), A, $\mathcal{T}$)
     {When virtual edge $e_V \in skeleton(\mu_l)$ and μ_l is adjacent to μ_k in
     $\mathcal{T}$ $twin\_edge\_of(e_V)$ denotes corresponding to e virtual edge $e'_V \in skeleton(\mu_k)$}
3: **end for**
4: **for** all virtual edges e_V of $skeleton(\mu)$ **do**
5:   $CA[e_V].append($ "$_S($" $)$
6:   $CA[e_V].append($ $number\_of\_edges(skeleton(\mu))$ $)$
7:   $e =$ the edge following e_{in} in the tour around the circle in the direction
     given by e_{in}
8:   tour_counter=1
9:   **while** $e \neq e_V$ **do**
10:     **if** e is a virtual edge **then**
11:       $CA[e_V].append($ tour_counter $)$
12:       $CA[e_V].append(CV[e])$
13:     **end if**
14:     **if** $A[tail\_vertex(e)] \neq NULL$ **then**
15:       $CA[e_V].append($ tour_counter $)$
16:       $CA[e_V].append($ "$*$" $)$
17:       $CA[e_V].append(A[tail\_vertex(e)])$,   delete $A[tail\_vertex(e)]]$ code
         from A {if code $A[tail\_vertex(e)]$ does not exist nothing is appended
         }
18:     **end if**
19:     $e =$ the edge following e in the direction given by e_V
20:     tour_counter=tour_counter+1
21:   **end while**
22:   $CA[e_V].append($ "$)_S$" $)$
23: **end for**
24: **return** FIND-THE-SMALLEST-CODE(CA)

Algorithm 5 Constructing the unique code for P-root and R-root nodes of $\mathcal{T}$

CODE-OF-P-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)

$\{v_A,\ v_B\}$ the vertices of the skeleton of a P-node
CA -code array indexed by $v_A,\ v_B$
CV -table of codes associated with virtual edges

 1: **for** all $v \in \{v_A,\ v_B\}$ of $skeleton(\mu)$ **do**
 2:     **for** all virtual edges e_V directed out of v **do**
 3:         $\nu =$ the child of μ corresponding to virtual edge e_V
            $CV[e_V]$=FIND-CODE($twin\_edge\_of(e_V)$, skeleton(ν), A, $\mathcal{T}$)
 4:     **end for**
 5:     $CA[v].append($ "$_P($" $)$
 6:     $CA[v].append($ $number\_of\_edges(skeleton(\mu))$ $)$
 7:     $CA[v].append($ $number\_of\_virtual\_edges(skeleton(\mu))$ $)$
 8:     concatenate all codes from CV to CA in increasing order
 9:     **if** $A[v] \neq NULL$ **then**
10:         $CA[e_V].append($ "$*$" $)$
11:         $CA[e_V].append(A[v])$, delete $A[v]$ code from A
12:     **end if**
13:     $CA[v].append($ "$)_P$" $)$
14: **end for**
15: **return** FIND-THE-SMALLEST-CODE(CA)

CODE-OF-R-ROOT-NODE($skeleton(\mu)$, A, $\mathcal{T}$)

CV -table of codes associated with virtual edges
CA -code array

 1: **for** all virtual edges e_V of $skeleton(\mu)$ including reverse edges **do**
 2:     $\nu =$ the child of μ corresponding to virtual edge e_V
        $CV[e_V]$=FIND-CODE($twin\_edge\_of(e_V)$, skeleton(ν), A, $\mathcal{T}$)
 3: **end for**
 4: **for** all virtual edges e_V of $skeleton(\mu)$ including reverse edges **do**
 5:     $CA[e_V].append($ "$_R($" $)$
 6:     Apply Weinberg's [48] procedure to find code associated with e_V going
        right $CodeRight$ and going left $CodeLeft$. When virtual edge is encoun-
        tered during the tour, append its code to $CA[e_V]$
 7:     **if** at any vertex v during Weinberg's traversal $A[v] \neq NULL$ **then**
 8:         $CA[e_V].append(A[v])$ delete $A[v]$ code from A
 9:     **end if**
10:     $CA[e_V]$=FIND-THE-SMALLEST-CODE($[CodeRight,\ CodeLeft]$)
11:     $CA[e_V].append($ "$)_R$" $)$
12: **end for**
13: **return** FIND-THE-SMALLEST-CODE(CA)

Algorithm 6 Constructing the unique code for S,P,R non root nodes of $\mathcal{T}$

FIND-CODE(e_{in}, skeleton(μ), A, $\mathcal{T}$)

CV -table of codes associated with virtual edges
C -code

1: **if** $\mu =$ S node **then**
2:     $C.append($ "$_S($" $)$
3:     $C.append($ $number\_of\_edges(skeleton(\mu))$ $)$
4:     $e_V =$ the edge following e_{in} in the tour around the circle in the direction given by e_{in}
5:     tour_counter=1
6:     **while** $e_V \neq e_{in}$ **do**
7:         **if** e_V is a virtual edge **then**
8:             $\nu =$ the child of μ corresponding to virtual edge e_V
                C.append(FIND-CODE(e_V, skeleton(ν), A, $\mathcal{T}$))
9:         **end if**
10:         **if** $A[tail\_vertex(e_V)] \neq NULL$ **then**
11:             $C.append($ tour_counter $)$
12:             $C.append($ "$*$" $)$
13:             $C.append(A[tail\_vertex(e_V)])$ delete $A[tail\_vertex(e_V)]$ code from A
                {if code $A[tail\_vertex(e_V)]$ does not exist nothing is appended }
14:         **end if**
15:         $e_V =$ the edge following e_V in the direction given by e_{in}
16:         tour_counter=tour_counter+1
17:     **end while**
18:     $C.append($ "$)_S$" $)$
19: **else if** $\mu =$ P node **then**
20:     **for** all virtual edges $e_V \neq e_{in}$ directed the same as e_{in} **do**
21:         $\nu =$ the child of μ corresponding to virtual edge e_V
            $CV[e_V]$=FIND-CODE($twin\_edge\_of(e_V)$, skeleton(ν), A, $\mathcal{T}$)
22:     **end for**
23:     $C.append($ "$_P($" $)$
24:     $C.append($ $number\_of\_edges(skeleton(\mu))$ $)$
25:     $C.append($ $number\_of\_virtual\_edges(skeleton(\mu))$ $)$
26:     concatenate all codes from CV to C in increasing order
27:     **if** $A[tail\_vertex(e_{in})] \neq NULL$ **then**
28:         $C.append($ "$*$" $)$
29:         $C.append(A[tail\_vertex(e_{in})])$, delete $A[tail\_vertex(e_{in})]$ code from A
30:     **end if**
31:     $C.append($ "$)_P$" $)$
32: **else if** $\mu =$ R node **then**
33:     C=FIND-CODE-R-NON-ROOT(e_{in}, skeleton(μ), A, $\mathcal{T}$)
34: **end if**
35: **return** C

Algorithm 7 Constructing the unique code for R non root node of $\mathcal{T}$ and finding the smallest code

---

FIND-CODE-R-NON-ROOT(e_{in}, skeleton(μ), A, $\mathcal{T}$)

CV -table of codes associated with virtual edges
C -code

1: **for** all virtual edges $e_V \neq e_{in}$ **do**
2:     ν = the child of μ corresponding to virtual edge e_V
       $CV[e_V]$=FIND-CODE($twin\_edge\_of(e_V)$, skeleton(ν), A, $\mathcal{T}$)
3: **end for**
4: $C.append($ "$_R($" $)$
5: Apply Weinberg's [48] procedure to find code associated with e_{in} going right $CodeRight$ and going left $CodeLeft$
6: **if** at any vertex v during Weinberg's traversal $A[v] \neq NULL$ **then**
7:     $C.append(A[v])$, delete $A[v]$ from A
8: **end if**
9: **if** at any edge $e \neq e_{in}$ during Weinberg's traversal $CV[e] \neq NULL$ **then**
10:     $C.append(CV[e])$
11: **end if**
12: C=FIND-THE-SMALLEST-CODE($[CodeRight, CodeLeft]$)
13: $C.append($ "$)_R$" $)$
14: **return** C

FIND-THE-SMALLEST-CODE(CA)

1: Remove from CA all codes with length bigger than minimal code length in CA
2: $index$=0
3: **while** CA has more than one code AND $index <$length of codes in CA **do**
4:     Remove all codes from CA with smaller value of $CA[Code[index]]$ than minimum of
       $\{CA[Code1[index]], CA[Code2[index]],\dots,CA[CodeN[index]]\}$
5:     $index = index + 1$
6: **end while**
7: **return** CA[first code]

---

appends to $CA[e_V]$ codes associated with the virtual edges and the articulation points. The procedure CODE-OF-S-ROOT-NODE returns the smallest code from CA at line 24.

The procedure CODE-OF-P-ROOT-NODE in its main loop at lines 1-14 creates two codes stored in a CA array. In the first code, the virtual edges e_V are directed from vertex v_A to vertex v_B. This direction is used in the FIND-CODE procedure called at line 3. Each of the two codes starts with symbol '$_P$(' at line 5. Next, we append the number of edges and number of virtual edges at lines 6-7. We concatenate all codes associated with virtual edges in increasing order at line 8. If v_A or v_B correspond to articulation points in the original graph, we add this information at line 10. If the code associated with this articulation point exists, we append it at line 11. At line 15 we return the smaller of the two codes.

The procedure CODE-OF-R-ROOT-NODE starts from finding all codes associated with virtual edges at lines 1-3. Using Weinberg's procedure we find two codes: $CodeRight$ for triconnected skeleton of the node starting from e_V and $CodeLeft$ for mirror image of the skeleton also starting from e_V. We find the two codes starting from every virtual edge of the skeleton. We determine the smallest among these codes and return it at line 13.

Algorithm 6 describes the FIND-CODE procedure. It is a recursive procedure and it calls itself at lines 8, 21 and line 2 of FIND-CODE-R-NON-ROOT procedure. FIND-CODE uses input edge e_{in} and its direction as an initial edge to create code for non-root nodes S, P, R. Code for an S-node is found at lines 1-18, for P-node at lines 19-31 and we call FIND-CODE-R-NON-ROOT at line 33 to find code for R-node. The algorithm for non-root nodes is similar as for root nodes. In the case of non-root nodes we do not have ambiguity related to lack of a starting edge because e_{in} is the starting edge.

The procedure FIND-THE-SMALLEST-CODE accepts a code array CA. We find the length of the shortest code and eliminate from CA all codes with longer length at line 1. Then, in the remaining codes we find the minimum of values at the first coordinate of the codes. We eliminate all codes that have bigger value than minimum at the first coordinate. We do the same elimination process for the second coordinate. We continue this process until only one code is left in CA or until we reach the last coordinate. We return the first code in CA.

A.2   Complexity Analysis

Lemma A.1 [17] *The SPQR-tree of G has $O(n)$ S-, P-, and R-nodes. Also, the total number of vertices of the skeletons stored at the nodes of $\mathcal{T}$ is $O(n)$.*

Lemma A.2 *The construction of the code of a biconnected graph G by Algorithm 3-7 takes $O(n^2)$ time.*

Proof The algorithms traverse the edges of a biconnected graph G with n vertices. Graph G is planar, and therefore its number of edges does not exceed $3n-6$. By Lemma A.1 the total number of vertices of the skeletons of an SPQR-tree $\mathcal{T}$ stored at the nodes is $O(n)$. Therefore, the total number of real edges of the skeletons is $O(n)$. Since $\mathcal{T}$ has $O(n)$ nodes, also the number of virtual edges of all skeletons is $O(n)$. The algorithm works on a bidirected graph, which doubles the number of edges of G. The procedure FIND-CODE (Algorithm 6) traverses skeletons starting from initial edge e_{in}. The FIND-CODE procedure traverses every circle skeleton of S-node once in one direction. Also, the edges of a P-node skeleton are traversed once. The skeleton of an R-node is traversed two times while building a code for triconnected graph and its mirror image. All traversals starting from initial edge e_{in} on all edges that belong to all skeletons of non-root node takes $O(n)$ time. The skeletons of center nodes of $\mathcal{T}$, because of lack of an initial edge, are traversed as many times as there are virtual edges in the center node (Algorithms 4 and 5). Since the number of virtual edges in the center node cannot exceed the total number of nodes in $\mathcal{T}$, the skeleton of the center node is traversed no more than $O(n)$ times. The skeleton of a center node has $O(n)$ edges, therefore we visit $O(n)$ edges $O(n)$ times resulting in $O(n^2)$ total traversal steps. Particularly, if G is a triconnected graph, its SPQR-tree contains only one R-node. Weinberg's procedure builds codes starting from every edge and traverses all edges of G resulting in total $O(n^2)$ traversal steps. Overall, the traversal over all skeleton edges, including the center node, takes $O(n^2)$ time. The code built for G is $O(n)$ long. This is because we include two symbols at the beginning and the end of the code at each node and the code representation of the skeleton does not require more than the number of vertices and number of edges of the skeleton combined.

   Algorithms 3-7 order lexicographically the codes associated with the virtual edges of the skeletons of the P-nodes. It also finds the smallest code from the array of codes while looking for the unique code of an S-root-node and of an R-root-node. Looking for unique code requires both ordering the codes and finding the smallest code operations on code arrays of variety of sizes. However, any code created during the process has length of linear complexity with the number of vertices and edges traversed to create this code. Also, the number of created codes has the same complexity as the the number of nodes of $\mathcal{T}$, which is $O(n)$ by Lemma A.1. Therefore, all codes created during the execution of the algorithm could be stored in an array of dimension $O(n)$ by $O(n)$. The codes consist of integers. The values of integers are bound by $O(n)$, because the maximum value of an integer in the code does not exceed the number of edges of the bidirected graph G. Sorting an array of $O(n)$ by $O(n)$ dimension lexicographically with

Radix Sort, where codes can be ordered column by column using Counting Sort, takes $O(n^2)$ time. Therefore, the complexity of finding minimum code and lexicographical ordering with Radix Sort performed at every node of the SPQR-tree together is not more than $O(n^2)$. ∎

Lemma A.3 *The construction of the code of a planar graph G by Algorithm 2-7 takes $O(n^2)$ time.*

Proof Let $B_1, \ldots, B_k$ denote biconnected components of G. Let T be a biconnected tree of G. By Lemma A.2, constructing the code of the biconnected component B_i, with m_i vertices, associated with a leaf node of T takes $O(m_i^2)$ time. Also, the length of B_i code is $O(m_i)$. At every step of Algorithms 2-3, the length of a produced code has linear complexity with the number of vertices of the subgraph this code represents. Let the times spent to produce the codes of biconnected components $B_1, \ldots, B_k$ be given by $O(m_1^2), \ldots, O(m_k^2)$. Since $m_1 + \ldots + m_k = n$, and $m_1^2 + \ldots + m_k^2 \leq (m_1 + \ldots + m_k)^2$, the total time spent on construction of all codes of all biconnected components of G is $O(n^2)$. The lexicographical ordering of the codes at articulation nodes requires the sorting of biconnected codes of various lengths. The total number of sorted codes will not exceed the number of edges of G because G is planar. The length of codes are bound by $O(n)$ and maximum value (integer) in the code is bounded by $O(n)$. All partial codes of G can be stored in an array of dimension $O(n)$ by $O(n)$. Using Radix Sort with Counting Sort on every column of this table takes $O(n^2)$ time. Sorting codes at all articulation nodes combined using Radix Sort, do not exceed complexity of sorting $O(n)$ by $O(n)$ array. Therefore, the total process of lexicographical ordering at all articulation nodes of T takes $O(n^2)$ time. Since constructing codes for all biconnected components takes $O(n^2)$ and sorting subcodes of G at articulation points does not take more than $O(n^2)$, the total time for constructing code for planar graph is $O(n^2)$. ∎

A.3   The Proof of Uniqueness of the Code

We first prove that a code for a biconnected graph is unique and then use this result to prove that a code for a planar graph is unique. In saying unique code, we mean that the code produced is always the same for isomorphic graphs and different for non-isomorphic graphs, and therefore the code can be used for an isomorphism test. Battista and Tamassia [17, 5], who first introduced SPQR-trees, gave the following properties crucial to our unique code construction and the proof:

> "The SPQR-trees of G with respect to different reference edges are isomorphic and are obtained one from the other by selecting a different Q-node as the root."

> "The triconnected components of a biconnected graph G are in one-to-one correspondence with the internal nodes of the SPQR-tree: the R-nodes correspond to triconnected graphs, the S-nodes to polygons, and the P-nodes to bonds."

Since SPQR-trees are isomorphic regardless of the choice of the reference edge, we can uniquely identify the center (or two centers) of an SPQR-tree. We start the proof from the leaves of an SPQR-tree. We show that the codes associated with the leaves are unique. Next, we show that the codes of the nodes adjacent to the leaves are unique. We extrapolate this result to all nodes that are not a center of an SPQR-tree. Finally, we show that the code for a center node uniquely represents a biconnected graph.

Lemma A.4 *The smaller of the two Weinberg's codes:(1) a code of a triconnected graph G_R and (2) a code of the mirror image of G_R, found by starting from specified directed edge e_{in} as the initial edge of G_R, uniquely represents G_R.*

Proof In the proof we refer to Weinberg's paper [48]. It introduces code matrices M_1 and M_2 respective to planar triconnected graphs G_1 and G_2. Every row in the matrices is a code obtained by starting from a specified edge. Matrices M_1 and M_2 have size $4m \times (2m + 1)$ (m-number of undirected edges of the graph) because every triconnected graph has 4m codes of length 2m+1. The codes in the matrices are ordered lexicographically. G_1 and G_2 are isomorphic if and only if their code matrices are equal. It is also true that G_1 and G_2 are isomorphic if and only if any row of M_1 equals any row of M_2 [48]. Therefore, G_1 with initial edge e_{in}, and G_2 with e'_{in} that corresponds to e_{in} are isomorphic triconnected graphs if and only if the two codes of G_1 (code of G_1 and mirror image of G_1) started from directed initial edge e_{in}, and ordered in increasing order are equal to the two ordered codes of G_2 started from e'_{in}. We can select the smaller of the two codes. Since only one code is sufficient for an isomorphism test, the smallest code will uniquely represent a triconnected planar graph with the initial edge e_{in}.   ■

Lemma A.5 *The code produced by Algorithms 3-7 uniquely represents a biconnected graph.*

Proof Let G be an undirected, unlabeled, biconnected multigraph, and $\mathcal{T}$ be an SPQR-tree of G. Since SPQR-trees are isomorphic with respect to different reference edges, the unrooted SPQR-tree of G is unique [5]. Let us consider three categories of $\mathcal{T}$ nodes: (1) leaf nodes, (2) non-leaf, non-center nodes, and (3) the center node. Our algorithm does not use an SPQR-tree representation with Q-nodes. Instead, following the implementation of an SPQR-trees described in [24], it distinguishes between virtual and real edges in the skeletons and therefore we omit Q-nodes in the discussion.

Leaf nodes.
Consider leaf nodes of $\mathcal{T}$. Fig. 10 (reverse edges are omitted) shows skeletons of a P-leaf-node and an S-leaf-node. In the Parallel Case of the SPQR-tree definition we saw that the skeleton of a P-node consists of k parallel edges between the split pair $\{s,t\}$. Therefore, providing (1) the number of edges of the skeleton and (2) the node type, is sufficient to uniquely represent a P-leaf-node

skeleton. Similarly, in the Series Case of the SPQR-tree definition we saw that the S-node skeleton is a cycle $e_0, e_1, \ldots, e_k$. Providing (1) the number of edges of the skeleton and (2) the node type is sufficient to uniquely represent an S-leaf-node skeleton. By Lemma 2.1 the skeleton of an R-node is a triconnected graph. Since the input edge e_{in} to Algorithm 6 determines the initial edge, by Lemma A.4 we can build a unique code to represent the skeleton of the R-leaf-node using Weinberg's method.
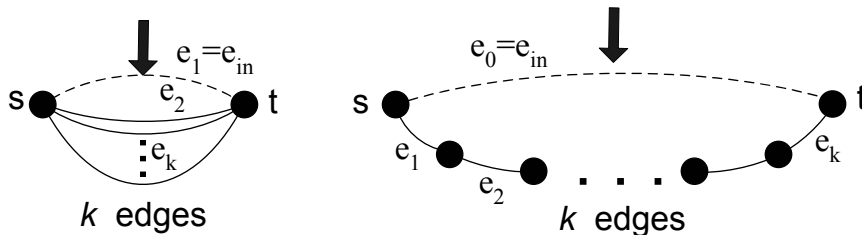

Figure 10: The skeleton of a P-leaf-node (left) and the skeleton of an S-leaf-node (right).

The skeletons of leaf nodes are in one to one correspondence to subgraphs of G; therefore the unique codes of leaf skeletons also uniquely represent the corresponding subgraphs of G. The unique codes of leaf skeletons are produced when the FIND-CODE procedure (Algorithms 6 and 7) reaches the leaves of $\mathcal{T}$.

Non-leaf, non-center nodes.
Fig. 11 shows a skeleton of a P-node adjacent to leaf nodes $\mu_{l(1)}, \ldots, \mu_{l(k)}$. The direction of the input edge e_{in} also determines the direction of virtual edges associated with leaves. From the definition of an SPQR-tree, a P-node introduces k parallel edges incident on split pair vertices {s,t}. Split components of a split pair {s,t} are also incident to s, t vertices in parallel. In reference to the discussion above, the codes of $\mu_{l(1)}, \ldots, \mu_{l(k)}$ are unique. Therefore, (1) the node type, (2) the number of virtual and real edges, and (3) the codes of $\mu_{l(1)}, \ldots, \mu_{l(k)}$ in increasing order, uniquely represent the skeleton of a P-node adjacent to leaves of $\mathcal{T}$ together with adjacent leaves.

Fig. 12 shows a skeleton of an S-node adjacent to leaves $\mu_{l(1)}, \ldots, \mu_{l(k)}$ of $\mathcal{T}$. By definition, an S-node is a circle with k edges $e_0, \ldots, e_k$. The S-node is associated with a parent node. This association allows for the unique identification of an edge $e_0 = e_{in}$ of the circle. Also, the direction of edge $e_0 = e_{in}$ is determined by the direction of the associated parent's skeleton edge. The traversal along edges $e_0, \ldots, e_k$, starting from e_0 in the direction of e_0, allows for identification of the distance from e_0 to every virtual edge of the circle. Virtual edges are associated with the unique codes of leaves $\mu_{l(1)}, \ldots, \mu_{l(k)}$. Therefore, (1) the node type, (2) the number of edges of the S-node skeleton, and (3)the codes of $\mu_{l(1)}, \ldots, \mu_{l(k)}$ together with the distance of the associated virtual edges from
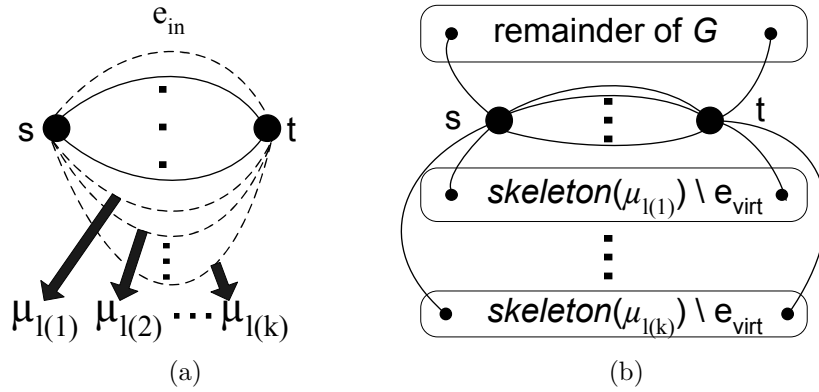
Figure 11: (a) The skeleton of a P-node adjacent to leaves $\mu_{l(1)}, \mu_{l(2)}, \ldots, \mu_{l(k)}$ of an SPQR-tree and (b) split pair $\{s, t\}$ of a P-node that splits G into subgraphs with one-to-one correspondence to the skeletons of the leaves of an SPQR-tree.

e_0, represent the S-node adjacent to the leaves of $\mathcal{T}$ together with the leaf nodes without ambiguity.

The skeleton of an R-node, by definition, is a triconnected graph. Weinberg's method traverses each edge of the skeleton once in each direction on Euler's path. The initial edge of an Euler's path is determined by the input edge. The Eulerian path started from the initial edge is deterministic in both embeddings of the skeleton, because by the property of triconnected graphs [48], the set of edges incident to a vertex has a unique order around the vertex. By lemma A.4 the code of the skeleton of an R-node is unique. If the R-node is adjacent to leaf nodes $\mu_{l(1)}, \ldots, \mu_{l(k)}$, the FIND_CODE procedure (Algorithms 6 and 7) inserts the code of a leaf node whenever the virtual edge associated to this leaf is encountered. The codes of $\mu_{l(1)}, \ldots, \mu_{l(k)}$ are unique and they are inserted into the unique code of the skeleton of the R-node deterministically when Euler's tour encounters split pairs $\{s_1, t_1\} \ldots \{s_k, t_k\}$ associated with leaf nodes (they identify how split components represented by leaves are adjacent to the subgraph of G represented by the R-node), creating a unique representation for the skeleton of the R-node and the leaves adjacent to this R-node in $\mathcal{T}$.

Since we can build unique codes for nodes adjacent to leaves of $\mathcal{T}$, by the same reasoning we can build unique codes for any non-center node. The S-, P-, and R- nodes with leaf codes associated with their virtual edges can become new leaves in the SPQR-tree. Thus, we then have a new SPQR-tree containing leaves with unique codes. Therefore we can continue to apply the above code construction until reaching a center node.

Center node.
In reference to the above discussion, a unique code exists for S-,P-,and R-nodes

$$e_0 = e_{in}$$

s k edges   t

e_1 e_k

e_2

$\mu_{l(1)}$ . . . $\mu_{l(k)}$

(a)

Remainder of *G*

s   t

e_1 e_k

. . . .

skeleton$(\mu_{l(1)})$ \ e_{virt} *skeleton*$(\mu_{l(k)})$ \ e_{virt}
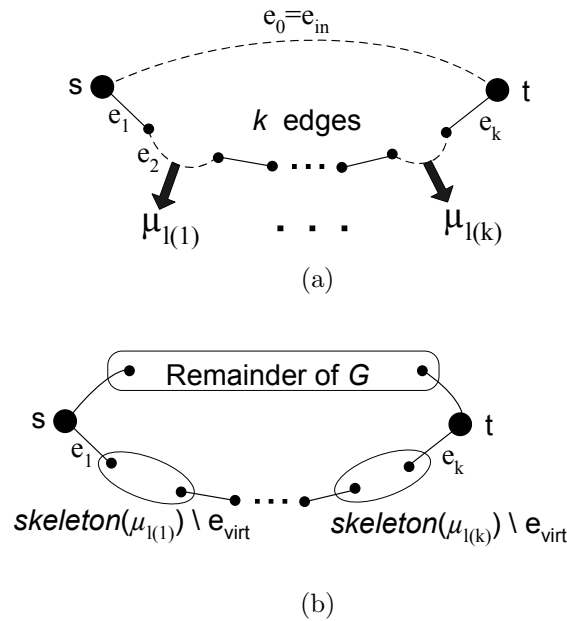
(b)

Figure 12: (a) The skeleton of an S-node adjacent to leaves $\mu_{l(1)}, \mu_{l(2)}, \ldots, \mu_{l(k)}$ of an SPQR-tree and (b) split pairs $\{s_{l(1)}, t_{l(1)}\}, \ldots, \{s_{l(k)}, t_{l(k)}\}$ that splits G into subgraphs with one-to-one correspondence to the skeletons of the leaves of an SPQR-tree.

if we can identify one initial edge of the skeleton. Let us find codes $C_1, \ldots, C_k$ starting from all virtual edges of a central node in the same way as for non-central nodes. All virtual edges are associated with unique codes of nodes adjacent to the center. Because of the deterministic traversal of R- and S-nodes, and parallel adjacency of split components to split pair vertices of a P-node, codes $C_1, \ldots, C_k$ will be the same for isomorphic graphs and different for non-isomorphic graphs. Choosing the smallest code C_{min} from $C_1, \ldots, C_k$ will uniquely identify the initial edge and from the discussion of non-root nodes, C_{min} will uniquely represent the center node and biconnected graph G. The ambiguity associated with the two center nodes of $\mathcal{T}$ is resolved by applying the procedure of finding a code from the center of $\mathcal{T}$ for every center node and choosing the smaller code.

Based on the uniqueness of an unrooted SPQR-tree, the unique order of edges around vertices of a triconnected graph that allows for the deterministic traversal of R-node skeletons, the deterministic traversal of a circle of an S-node skeleton, and the parallel adjacency of split components to split pair vertices of a P-node, we can build a unique code for a biconnected graph starting from the

leaves and progressing toward the center of an SPQR-tree.   ▉

Next, we discuss the correctness of the algorithm of the unique code construction for planar graphs. We show first that any biconnected tree has only one center node. We use the unique code of biconnected graphs to show that traversing a biconnected tree from the leaves towards the center allows for unique code construction for planar graphs.

Lemma A.6 *Any biconnected tree has only one center node.*

Proof By definition, biconnected nodes are adjacent only to articulation nodes in a biconnected tree. There are no two articulation nodes adjacent, nor two biconnected nodes adjacent. The leaves of a biconnected tree are biconnected nodes. When we remove them, new leaves are articulation nodes. We would alternately remove biconnected node leaves and articulation node leaves. This process can only result in either one biconnected node or one articulation node as the center.   ▉

Lemma A.7 *The code produced by Algorithms 2-7 uniquely represents a connected planar graph G.*

Proof The algorithm decomposes a connected planar graph G into biconnected components. The location of articulation points relative to biconnected components is given in biconnected tree. Biconnected tree has two kinds of nodes: articulation nodes and biconnected nodes. By Lemma A.5 we can produce a unique code of a biconnected graph. Let us consider leaf nodes of a biconnected tree. Leaf node B_i is adjacent to one articulation node v_i, therefore the corresponding biconnected graph G_i is connected to the remaining of G through one articulation point u_i. The code construction procedure for a biconnected graph identifies a unique edge of the biconnected graph and traverses along the edges of P-, S-, and R- skeletons deterministically. Therefore, the distance from the initial edge to the skeleton's vertex that corresponds to the articulation point is identified deterministically, and therefore will be the same regardless of how G is presented to the algorithm.

Let an articulation node v_i be adjacent to a non-leaf biconnected node B_x and to biconnected leaves $B_l, \ldots, B_m$. Procedure FIND-BICONNECTED-CODE (Algorithm 3) constructs a unique code corresponding to the biconnected components $B_l, \ldots, B_m$ of subgraphs $G_l, \ldots, G_m$. The new code associated with v_i, made from codes of $G_l, \ldots, G_m$ by concatenating them in increasing order, uniquely represents $G_l, \ldots, G_m$. Let the B_x node correspond to subgraph G_x. Since the starting edge of G can be identified uniquely and P-, S-, and R- skeletons are traversed deterministically, the distance along traversed edges from the initial edge of G_x to the articulation points is found without ambiguity. Therefore, the code of v_i and other articulation nodes of G_x are inserted into the code of G_x uniquely identifying the positions of the articulation points in G_x. The code of G_x, which includes codes of $G_l, \ldots, G_m$ uniquely identifies G_x and $G_l, \ldots, G_m$. The same reasoning we apply to all nodes of the biconnected tree

moving from the leaves to the center. By Lemma A.6, there is only one center node of a biconnected tree. The code of the center node combines codes of all biconnected subgraphs $G_l, \ldots, G_m$ and uniquely represents connected planar graph G.   ∎

References

[1] *Algorithms for graph drawing (AGD) User Manual Version 1.1.2.* Technische Universität Wien, Universität zu Köln, Universität Trier, Max-Planck-Institut Saarbrücken, 2002.

[2] A. Aggarwal, M. Klawe, and P. Shor. Multi-layer grid embeddings for VLSI. *Algorithmica*, 6:129–151, 1991.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[4] L. Babai and L. Kucera. Canonical labelling of graphs in linear average time. In *20-th Annual Symp. Foundations of Computer Science*, pages 39–46, 1979.

[5] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–168, 1998.

[6] T. Beyer, W. Jones, and S. Mitchell. Linear algorithms for isomorphism of maximal outerplanar graphs. *J. ACM*, 26(4):603–610, Oct. 1979.

[7] K. Booth and G. Lueker. Testing for the consecutive ones property interval graphs and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[8] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. In *Proc. 10th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 140–146, 1999.

[9] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–254, 1982.

[10] D. Cook and L. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[11] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[12] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *10th International Conference on Image Analysis and Processing*, pages 1172–1178, September 1999.

[13] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. Assoc. Comput. Mach.*, 17:51–64, 1970.

[14] A. Dessmark, A. Lingas, and A. Proskurowski. Faster algorithms for subgraph isomorphism of k -connected partial k -trees. *Algorithmica*, 27:337–347, 2000.

[15] G. Di Battista and R. Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441, Research Triangle Park, North Carolina, 1989. IEEE.

[16] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Automata, Languages and Programming (Proc. 17th ICALP)*, volume 443 of *LNCS*, pages 598–611. Springer-Verlag, 1990.

[17] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[18] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

[19] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.

[20] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, May 2001.

[21] Z. Galil, C. M. Hoffmann, E. M. Luks, C. P. Schnorr, and A. Weber. An $O(n^3 \log n)$ deterministic and an $O(n^3)$ Las Vegas isomorphism test for trivalent graphs. *J. ACM*, 34(3):513–531, July 1987.

[22] H. Gazit and J. H. Reif. A randomized parallel algorithm for planar graph isomorphism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219, 1990.

[23] H. Gazit and J. H. Reif. A randomized parallel algorithm for planar graph isomorphism. *Journal of Algorithms*, 28(2):290–314, 1998.

[24] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In J. Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2001.

[25] J. E. Hopcroft and R. E. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1(1):32–34, Feb. 1971.

[26] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, Aug. 1973.

[27] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, Oct. 1974.

[28] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Conference record of sixth annual ACM Symposium on Theory of Computing*, pages 172–184, Seattle, Washington, May 1974.

[29] J. Jájá and S. R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.

[30] B. Jenner, J. Köbler, P. McKenzie, and J. Torán. Completeness results for graph isomorphism. *Journal of Computer and System Sciences*, 66(3):549–566, May 2003.

[31] X. Jiang and H. Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, July 1999.

[32] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01)*, pages 313–320, 2001.

[33] A. Liebers. Planarizing graphs - a survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.

[34] G. S. Lueker and K. S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26(2):183–195, Apr. 1979.

[35] E. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[36] S. Maclaine. A structural characterization of planar combinatorial graphs. *Duke Math Journal*, (3):460–472, 1937.

[37] B. D. McKay. Australian National University. *Personal communication*.

[38] B. D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.

[39] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[40] K. Mehlhorn, P. Mutzel, and S. Naher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Research Report MPI-I-93-151, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1993.

[41] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[42] G. L. Miller and J. H. Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[43] T. Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and Computation II*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 239–256. Finkelstein and W. M. Kantor, 1996.

[44] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23(3):433–445, July 1976.

[45] D. A. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pages 576–584. ACM Press, 1996.

[46] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[47] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, Jan. 1976.

[48] L. Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circuit Theory*, 13:142–148, 1966.

[49] R. Weiskircher. *New Applications of SPQR-Trees in Graph Drawing*. PhD thesis, Universität des Saarlandes, 2002.

[50] H. Whitney. A set of topological invariants for graphs. *Amer. J. Math.*, 55:321–235., 1933.

[51] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining*, pages 721–724, Maebashi City, Japan, December 2002.