The Complexity of Planar Graph Isomorphism

Jacobo Torán and Fabian Wagner*

Abstract

The Graph Isomorphism problem restricted to *planar* graphs has been known to be solvable in polynomial time many years ago. In terms of complexity classes however, the exact complexity of the problem has been established only very recently. It was proved in [**?**] that planar graph isomorphism can be computed within logarithmic space. Since there is a matching hardness result [**?**], this shows that the problem is complete for L. Although this could be considered a result in in algorithmics its proof relies on several important new developments in the area of logarithmic space complexity classes and reflects the close connections between algorithms and complexity theory. In this column we give an overview of this result mentioning the developments that led to it.

1 Introduction

The Graph Isomorphism problem asks whether two given graphs are isomorphic or in other words whether there is a bijection between the nodes of the two graphs, preserving the adjacency relation. Graph Isomorphism is one of the few problems in NP that is not know to be in P or NP-complete. On the other hand, for many restricted graph classes like trees, graphs of bounded degree or partial k-trees, efficient algorithms for the isomorphism problem are known. We consider in this column the class of planar graphs and for simplicity restrict ourselves to undirected simple graphs. A graph is planar if it can be drawn in the plane without any crossing edges. A special class of planar graphs is that of 3-connected planar graphs. A graph is k-connected if it remains connected after deleting k arbitrary vertices. It was shown in 1933 that 3-connected planar graphs have exactly two planar embeddings [**?**]. This fact was used by Weinberg in 1966 to give an $O(n^2)$ algorithm for testing isomorphism of 3-connected planar graphs [**?**] (n is the number of vertices of the input graphs). The idea of the algorithm is simple, the two

*Universität Ulm, Institut für Theoretische Informatik, fabian.wagner@uni-ulm.de

given graphs are embedded in the plane and it is tested whether the embedding of the first graph is isomorphic to one of the two possible embeddings of the second graph. With this method it is also possible to efficiently assign codes to 3-connected planar graphs as a way to identify them. This means that there is an efficiently computable function mapping graphs to strings or codes, so that two graphs are mapped to the same code if and only if they are isomorphic. Hopcroft and Tarjan extended Weinberg's algorithm and gave the first polynomial time algorithm for the isomorphism of planar graphs [**?**]. In their algorithm the graphs are divided first in connected components, these are subdivided in biconnected components and finally these components are partitioned in 3-connected components. An initial connected component has articulation points separating its biconnected components from each other. These initial components are represented by a tree-like structure containing vertices for articulation points and vertices for biconnected components. In a similar way, the biconnected components are represented by tree structures containing vertices for its 3-connected components and vertices for pairs of nodes (called separating pairs) separating these components. Each vertex representing a 3-connected component is then labeled with the code of the component and the whole structure can then be tested for isomorphism in a similar way as it is done for tree isomorphism. The original algorithm had a running time of $O(n^2)$. This was then improved to $O(n \log n)$ in [**?**] and finally Hopcroft and Wang [**?**] obtained a linear time algorithm for the isomorphism of planar graphs. Recently Kukluk, Holder, and Cook [**?**] gave an $O(n^2)$ algorithm for planar graph isomorphism, which is suitable for practical applications.

Regarding the parallel complexity of the problem, Miller and Reif [**?**] gave the first NC algorithm for planar graph isomorphism. Their algorithm runs in time $O(\log n)$ with polynomial number of processors in the CRCW PRAM model. This corresponds to the complexity class AC^1 of problems computable by unbounded fan-in circuits of polynomial size and logarithmic depth. More recently, using a completely different approach based on descriptive complexity, Grohe and Verbitski [**?**] provided a new method for testing isomorphism of planar graphs also within AC^1. They proved that for a class G of graphs, if every graph is the class is definable in first order logic with a finite number of variables and logarithmic quantifier depth, then the isomorphism problem for G is in AC^1. Verbitski [**?**] showed that the class of planar 3-connected graphs is definable with 15 variables and logarithmic quantifier depth. Together with the AC^1 reduction from planar isomorphism to 3-connected planar isomorphism from [**?**] this provides a different way to show that planar isomorphism lies in AC^1.

We describe in this survey some of the results leading to the improvement of the upper bound from AC^1 to logarithmic space. Located between L and AC^1, the complexity class UL played an important part in the development of the results. UL (unambiguous logarithmic space) is the class of problems computable

by a logarithmic space nondeterministic machine having at most one accepting computation path for each input. The relation between the considered complexity classes is as follows:

$$\mathsf{L} \subseteq \mathsf{UL} \subseteq \mathsf{AC}^1.$$

We denote by FL the class of functions computable within logarithmic space.

All the isomorphism results described in this overview can in fact be extended to graph canonization results. For a class G of graphs we say that a function $f : \mathcal{G} \to \{0,1\}^*$ computes a *complete invariant* for the class if for every $G, H \in \mathcal{G}$ it holds that G and H are isomorphic if and only if $f(G) = f(H)$. If moreover f computes for each G a graph $f(G)$ isomorphic to G then we call f a *canonizing function* and $f(G)$ a *canon*.

We recall in Section 2 several facts that were used in the proof of the results. Similarly as in the developments leading to polynomial time test for planar graph isomorphism, the first logarithmic space isomorphism algorithms worked for trees and 3-connected graphs. These are overviewed in Sections 2 and 3. Finally the logarithmic space algorithm for planar graph isomorphism is explained in Section 4.

2 Some previous results

2.1 Tree isomorphism in L

Lindell gave in [**?**] a logarithmic space algorithm for tree isomorphism and canonization. Some of the ideas in this algorithm are used also in the new results. Lindell defined a canonical ordering between trees. For a tree T, we represent its root by t and the out-degree of t by $\#t$. The tree isomorphism ordering $<_T$ from Lindell is defined in the following way: Given two trees S and T, we say $S <_T T$ if:

i) $|S| < |T|$ or

ii) $|S| = |T|$ and $\#s < \#t$, or

iii) $|S| = |T|$, $\#s = \#t = k$ and $(S_1, \ldots, S_k) < (T_1, \ldots, T_k)$ lexicographically, were $S_1 \leqslant_T S_2, \ldots \leqslant_T S_k, and T_1 \leqslant_T T_2 \ldots \leqslant_T T_k$, are the ordered subtrees of S and T rooted at the children of s and t.

It is not hard to see that if neither $S <_T T$ nor $T <_S S$ then S and T are isomorphic. Obviously the first two tests in the definition of tree ordering can be done in logarithmic space. Lindell proved that this is also true for the third step. This is done with the following variant of depth first search:

- Find the number of children of s and t of minimal size. If these numbers do not coincide then declare the tree with the largest number of minimal size children to be smaller. Otherwise check the next child size until an equality is found or all the children have been considered.

- If s and t have the same number of children of each size, we partition the children into size classes and compare the children in each size class in increasing order of the sizes recursively as follows: Let k be the number of children in one size class. We can suppose $k > 0$.

  – If $k = 1$ then only one recursive call is made and no extra space is needed for this.

  – If $k > 1$ then for each node in the size class in S we compute its order profile. This consists of three counters $c_<, c_>$ and $c_=$ indicating the number of children in the corresponding size class of T being respectively smaller than, larger than or equal to the node under consideration. The counters are updated by making cross comparisons. We start with the children with minimal order profile, those with $>_c = 0$. They form an isomorphism class. The size of this class is compared across the trees by comparing the values of the $c_=$ counters. If they match, both trees have the same number of minimal children. To compare larger children in the same size class, the value of $c_=$ in the last step works as a threshold. This is used to search for the next minimal children of s and t . The process is then repeated and the threshold is incremented until reaching k, at which point we proceed to the next size class. If all the size classed are visited without detecting an inequality the trees are isomorphic.

2.2 Planarity testing and distance computation

A graph is planar if it can be drawn on the plane so that no edges cross. Such a drawing is a planar embedding. Allender and Mahajan [**?**] showed that the problem of determining if a given graph is planar, can be computed in the complexity class SL (symmetric logarithmic space). Some year later Reingold [**?**] proved that the classed SL and L coincide, thus bringing the planarity recognition problem to L.

A *rotation scheme* for a graph G is a set ρ of permutations, $\rho = \{\rho_v \mid v \in V\}$, where ρ_v is a permutation on E_v that has only one cycle (which is called a *rotation*). Let ρ^{-1} be the set of inverse rotations, $\rho^{-1} = \{\rho_v^{-1} \mid v \in V\}$. A rotation scheme ρ describes an embedding of graph G in the plane. If the embedding is planar, we call ρ a *planar rotation scheme*. The pair (G, ρ) is called a *combinatorial*

embedding for G. The planarity recognition result Allender and Mahajan also showed the following useful result:

Theorem 2.1. *[?] There is a logarithmic space algorithm that on input a planar graph G produces a planar rotation scheme ρ for G.*

A planar 3-connected graph has exactly two planar rotation schemes [**?**], some rotation ρ and its inverse ρ^{-1}.

An important tool in one the the isomorphism test for 3-connected planar graphs is the computation of the distances in planar graphs within the class UL:

Theorem 2.2. *[?] The distance between two given vertices in a planar graph can be computed in* $\mathsf{UL} \cap \mathsf{coUL}$.

This theorem builds on a series of results dealing with the reachability problem in directed planar graphs, [**?**],[**?**] that lead to an algorithm from Bourke, Tewari and Vinodchandran [**?**] to compute the reachability problem for planar graphs within $\mathsf{UL} \cap \mathsf{coUL}$.

2.3 Universal exploration sequences

The celebrated result from Reingold [**?**] showing that the reachability problem in undirected graphs can be computed in L, has an important consequence for the construction of universal exploration sequences in logarithmic space. This fact is used in some of the isomorphism algorithms.

For a d-regular G and a numbering of the edges, and an starting edge e_0, a sequence $(\tau_1, \tau_2, \ldots, \tau_l) \in \{0, \ldots, d-1\}^l$ defines a walk $v_{-1}, v_0, \ldots, v_k$ in G in the following way: starting at $e_0 = (v_{-1}, v_0)$ for each i, if (v_{i-1}, v_i) is the k-th edge of v_i then (v_i, v_{i+1}) is the $k + \tau_i$ edge of v_i modulo d.

A sequence $(\tau_1, \tau_2, \ldots, \tau_l) \in \{0, \ldots, d-1\}^l$ is called an (n, d) *universal exploration sequence* n if for every connected d-regular graph with at most n vertices, any numbering of its edges and any starting edge, the walk obtained from the sequence explores all the vertices in the graph.

The result that we use is that such universal exploration sequences exist and can be computed in logarithmic space.

Theorem 2.3. *[?] There exists a logarithmic space algorithm that on input* $(1^n, 1^d)$ *produces an* (n, d)*-universal exploration sequence of polynomial size.*

3 Planar 3-connected Graph Isomorphism

Weinberg's [**?**] $O(n^2)$ algorithm for testing isomorphism of planar 3-connected graphs constructs a code for every edge of G and both rotation schemes. Of all

these codes, the lexicographical smallest one is used as a canonical form for G. Weinberg's algorithm does not work within logspace, because the vertices and edges have to be stored. Thierauf and Wagner showed in [**?**] how to construct a different code in UL. Some months later, using Reingolds results on logarithmic space universal exploration sequences [**?**], Datta, Limaye and Nimbhorkar improved this to an isomorphism algorithm for planar 3-connected graphs that works in logarithmic space. We describe both results in this section.

3.1 An isomorphism algorithm in UL ∩ coUL

Theorem 3.1. *[**?**] The isomorphism problem for planar,* 3*-connected graphs is in* UL ∩ coUL.

Let (s,t) be a designated edge and ρ be a rotation scheme for G. The construction has three steps: First, we compute a canonical spanning tree T for G. Second, with help of this spanning tree and the rotation function ρ we perform a depth-first traversal on the edges of the graph and construct a canonical list L of all edges of G. Finally, we rename the vertices depending on the position of their first occurrence in the list L.

We will see that the spanning tree in step 1 can be computed in (the functional version of) UL ∩ coUL. The list and the renaming in step 2 and step 3 can be computed in FL.

The overall algorithm has to decide whether two given graphs G and H are isomorphic. To do so we fix (s,t) and ρ for G and cycle through all edges of H as designated edge and the two possible embeddings of H. Then G and H are isomorphic if and only if the canonical forms for G and H match. It is not hard to see that this outer loop works in logspace.

Step 1: Construction of a canonical spanning tree

We show that the following problem can be solved in unambiguous logspace. Given, an undirected graph $G = (V,E)$, a rotation scheme ρ for G, and a designated edge $(s,t) \in E$. Output a canonical spanning tree $T \subseteq E$ of G, which does not depend on the input representation of ρ or G, any representation will result in the same spanning tree T.

The idea to construct the spanning tree is to traverse G with a breadth-first search starting at node s. The neighbors of a node are visited in the order given by the rotation scheme ρ. Since the algorithm should work in logspace, we cannot afford to store all the nodes that we already visited, as in a standard breadth-first search. We get around this problem by working with distances between nodes.

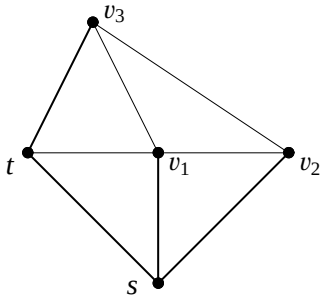
We start with the nodes at distance 1 from s. That is, write (s, v) on the output tape, for all $v \in \Gamma(s)$. Now let $d \geqslant 2$ and assume that we have already constructed T up to nodes at distance $\leqslant d-1$ to s. Then we consider the nodes at distance d from s. Let w be a node with $d(s, w) = d$. We connect w to the tree constructed so far by computing a shortest path from s to w. Ambiguities are resolved by using the first feasible edge according to ρ. We start with (s, t) as the active edge (u, v).

- If $d(u, w) > d(v, w)$, then (u, v) is the first edge encountered that is on a shortest path from u to w. Therefore we go from u to v and start searching the next edge from v. As starting edge we take $\rho_v(v, u)$, the successor of (v, u). It is the new active edge.

- If $d(u, w) \leqslant d(v, w)$, then (u, v) is not on a shortest path from u to w. Then we proceed with $\rho_u(u, v)$ as the new active edge.

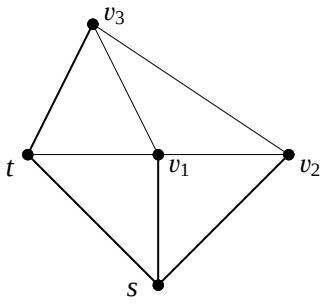
After $d-1$ steps in direction of w, the node v of the active edge (u, v) is a predecessor of w on a shortest path from s to w. Then we write (v, w) on the output tape.

The spanning tree T is canonical, because its construction depends only on ρ, edge (s, t), and edge set E. The following figure shows an example of a spanning tree T for a graph G with rotation function ρ which arranges the edges in clockwise order around each vertex.



$$
\begin{aligned}
\rho &= \{\rho_s, \rho_t, \rho_{v_1}, \rho_{v_2}, \rho_{v_3}\} \\
\rho_s &= (\,(s,t)\,(s,v_1)\,(s,v_2)\,) \\
\rho_t &= (\,(t,s)\,(t,v_3)\,(t,v_1)\,) \\
\rho_{v_1} &= (\,(v_1,s)\,(v_1,t)\,(v_1,v_3)\,(v_1,v_2)\,) \\
\rho_{v_2} &= (\,(v_2,s)\,(v_2,v_1)\,(v_2,v_3)\,) \\
\rho_{v_3} &= (\,(v_3,t)\,(v_3,v_2)\,(v_3,v_1)\,)
\end{aligned}
$$

Except for the computation of the distances, the algorithm works in logspace. We have to store the values of d, k, u and v, and the position of w, plus some extra space for doing calculations. By Theorem **??** above, the distances can be computed in $\mathsf{UL} \cap \mathsf{coUL}$. Since $\mathsf{L}^{\mathsf{UL} \cap \mathsf{coUL}} = \mathsf{UL} \cap \mathsf{coUL}$ the canonical spanning tree can be computed in $\mathsf{UL} \cap \mathsf{coUL}$.



$$\begin{aligned} L \;=\; & (s,t)(t,v_3)(v_3,v_2)(v_3,v_1)(v_3,t)(t,v_1)(t,s) \\ & (s,v_1)(v_1,t)(v_1,v_3)(v_1,v_2)(v_1,s) \\ & (s,v_2)(v_2,v_1)(v_2,v_3)(v_2,s) \end{aligned}$$

Figure 1: Computation of List L for G.

Step 2: Computation of a canonical list of all edges

With $G = (V, E)$, a rotation scheme ρ for G, a spanning tree $T \subseteq E$ of G, and a designated edge $(s, t) \in T$ we compute a canonical list L of all edges in E. The list L then still contains the original vertex names in G, it does not depend otherwise on the input representation of ρ, G or T.

The idea is to traverse the spanning tree in a depth-first manner. At each vertex u we visit all incident edges of u in a cyclic manner according to ρ_u until the next edge e of the spanning tree is reached. We go down the tree along e and recursively do the same at the node reached. At some point we will encounter e again and come back to u. Then we continue to output the edges incident to u.

More formally, we start the traversal with edge (s, t) as the active edge (u, v). We write (u, v) on the output tape and then compute the next active edge as follows:

- If $(u, v) \in T$ then we walk depth-first in T from u to v, consider the edge (v, u) and take $\rho_v(v, u)$ its successor according to ρ_v.
- If $(u, v) \notin T$ then we proceed breadth-first with $\rho_u(u, v)$.

This step is repeated until we entirely traversed E and the active edge is again (s, t). Every undirected edge is encountered exactly once in each direction.

The following figure shows an example for L.

Step 3: Renaming the vertices

The last step is to rename the vertices in the list L such that they become independent of the names they have in G. This is achieved as follows: consider the first occurrence (from left) of node v in L. Let $k - 1$ be the number of pairwise different nodes to the left of v. Then all occurrences of v are replaced by k. Recall,

that L starts with the edge (s,t). Hence, all occurrences of s get replaced by 1, all occurrences of t by 2, and so on. Call the new list $\mathsf{code}(G,\rho,s,t)$.

Given L as input, the list $\mathsf{code}(G,\rho,s,t)$ can be computed in logspace. We start with the first node v (which is s) and a counter k, that counts the number of different nodes we have seen so far. In the beginning, we set $k = 1$.

- If v occurs for the first time, then we output k and increase k by 1.
- If v occurs already to the left of the current position then we have to determine the number, that v got at its first occurrence. To do so, we determine the first appearance of v and then count the number of different nodes to the left of v at its first appearance.

Then we go to the next node in L.

Consider the example from above. The code constructed from list L for G is as follows.

$L =$	(s,t)	(t,v_3)	(v_3,v_2)	(v_3,v_1)	(v_3,t)	(t,v_1)	(t,s)
$\mathsf{code}(G,\rho,s,t) =$	$(1,2)$	$(2,3)$	$(3,4)$	$(3,5)$	$(3,2)$	$(2,5)$	$(2,1)$
sequel of L	(s,v_1)	(v_1,t)	(v_1,v_3)	(v_1,v_2)	(v_1,s)		
sequel of code	$(1,5)$	$(5,2)$	$(5,3)$	$(5,4)$	$(5,1)$		
sequel of L	(s,v_2)	(v_2,v_1)	(v_2,v_3)	(v_2,s)			
sequel of code	$(1,4)$	$(4,5)$	$(4,3)$	$(4,1)$			

It remains to argue that the new names of the nodes are independent of their names in G. Let H be a graph which is isomorphic to G, and let φ be an isomorphism between G and H. Note that $\rho \circ \varphi$ is a rotation scheme for H. Consider the computation of the code for graph H with rotation scheme $\rho \circ \varphi$ and designated edge $(\varphi(s),\varphi(t))$. The spanning tree computed in step 1 will be $\varphi(T)$ and the list computed in step 2 will be $\varphi(L)$. Now the above renaming procedure will give the same number to node v in L and to node $\varphi(v)$ in $\varphi(L)$. For example, the nodes $\varphi(s)$ and $\varphi(t)$ will get number 1 and 2, respectively. It follows that $\mathsf{code}(G,\rho,s,t) = \mathsf{code}(H,\rho\circ\varphi,\varphi(s),\varphi(t))$. We summarize:

Theorem 3.2. *[?] Let G and H be connected, undirected graphs, let ρ_G be a rotation scheme for G and (s,t) be an edge in G. Then G and H are isomorphic iff there exists a rotation scheme ρ_H for H and an edge (u,v) in H such that $\mathsf{code}(G,\rho_G,s,t) = \mathsf{code}(H,\rho_H,u,v)$.*

With a very different approach, Datta, Limaye and Nimbhorkar [**?**] improved the previous result from $\mathsf{UL} \cap \mathsf{coUL}$ to L. Their method is in some sense much

easier since it avoids the spanning tree construction eliminating the distance computations (the part in $\mathsf{UL} \cap \mathsf{coUL}$). It uses however the concept of universal exploration sequence and the non-trivial fact that such sequences can be computed in L.

Theorem 3.3. *[?] The isomorphism problem for planar, 3-connected graphs is in* L.

The idea of the algorithm is to use a universal sequence [**?**] in order to construct a canonical code for a given planar 3-connected graph G. Since Reingolds's construction requires the graph to have constant degree, there is a proprocesing step in which G is transformed into a 3-regular colored graph G' with the property that two graphs are isomorphic if and only if their transformations are also isomorphic (with a color preserving isomorphism). In a second step a canonical code is computed. The code is specific to the choice of a planar embedding ρ for G, a starting node and a starting edge. Since there are only polynomially many possible choices for these parameters, for two given graphs G and H, a logarithmic space procedure can cycle through all the possibilities and decide that the graphs are isomorphic if and only if the canonical codes match for any of the choices.

Step 1: Making the graph 3-regular

Given a 3-connected planar graph $G = (V, E)$ and a planar embedding ρ we construct a 3-regular planar graph G' with the edges colored with two colors. G' might not be 3-connected, however the planar embedding from G will be inherited to G'. Every vertex v of G is replaced in G' by a cycle $\{v_1, \ldots, v_d\}$ (d is the degree of v). The d edges $e_1, \ldots, e_d$ incident with v in G are now respectively incident to $\{v_1, \ldots, v_d\}$ in G'. The cycles edges are colored with color 1 and the edges $e_1, \ldots, e_d$ with color 2. The obtained graph G' is 3-regular and it is not hard to see that two graphs G and H are isomorphic if and only if their transformation G' and H' are isomorphic with a color preserving isomorphism.

Step 2: Obtaining the canonical code

On input an edge-colored graph G with n vertices, max. degree 3, a planar embedding ρ a starting vertex v and a starting edge $e = (u, v)$, a cannon for G is constructed. For this we compute first in logarithmic space a $(n, 3)$-universal exploration sequence $\mathcal{U}$. Then, starting at v and e we transverse G according to U and ρ giving the list L of the visited vertices as label. We can rename the vertices according to their first occurrence in L, as it is done in step 3 from Theorem 3.1. Finally we can cycle over every possible pair (i, j) checking whether it is an edge

in the renamed list and outputting its color if this is the case. This output is can be considered as a canonical colored adjacency matrix for G.

The authors prove that this method is correct by showing that for two graphs G_1 and G_2 with their respective embedding ρ_1, ρ_2 and starting vertices and edges v_1, v_2, e_1, e_2, if the canons coincide then the graph are isomorphic, and moreover, if G_i is isomorphic to G_2 there is some choice of the parameters that make thir respective canons equal.

4 Planar GI

In this section we describe the logarithmic space algorithm for planar graph isomorphism. A previous step towards this result was a logarithmic space isomorphism test for partial 2-trees [**?**]. Partial 2-trees are a subclass of the planar graphs. The class of partial 2-trees coincide with that of series-parallel graphs and contain all outer-planar graphs. For proving this result Arvind, Das and Köbler represent a partial 2-tree as a tree of cycles. Similar to Lindells algorithm [**?**] they compare two such tree representations up to isomorphism, defining a canonical ordering procedure, which finally gives a canonization algorithm.

In the isomorphism algorithm for general planar graphs a similar representation is used, namely a tree of 3-connected components.

We give a log-space algorithm for the *graph canonization problem*, to which graph isomorphism reduces. The canonization involves assigning to each graph an isomorphism invariant string of polynomial length.

The algorithms decomposed first we the planar graph into its biconnected components and construct a *biconnected component tree* in log-space [**?**]. Then, it further decomposes the biconnected planar components into their 3-connected components to obtain a 3-connected component tree in log-space. Hopcroft and Tarjan [**?**] presented a sequential algorithm for the decomposition of a biconnected planar graph into its 3-connected components. This method can be adapted to work in log-space. A biconnected planar graph is decomposed in 3-connected components, cycles or a 3-bonds, i.e. two vertices connected by three edges. The algorithm recursively removes separating pairs from the graph and puts a copy of the separating pair in each of the components so formed, i.e. the nodes in the separating pair are connected by a *virtual edge*. The decomposition stops, when the components become triconnected. Define for each component and each separating pair a node and connect a separating pair node with a component node, if the separating pair is contained in the component. The resulting graph is a tree, the *triconnected component tree*. This decomposition is unique [**?**]. Datta et. al. [**?**] prove, that such a decomposition can be computed in log-space. Figure 2 shows an example of the decomposition of a biconnected planar graph G. Its tricon-

nected components are $G_1, \ldots, G_4$ and the corresponding triconnected component tree is T. In G, the pairs (a, b) and (c, d) are the separating pairs. Since the 3-connected separating pair (c, d) is connected by an edge in G, we also get $\{c, d\}$ as triple-bond G_3. The virtual edges corresponding to the separating pairs are drawn with dashed lines.

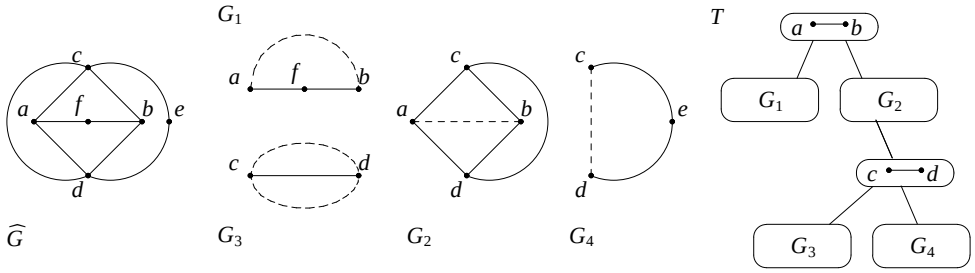


Figure 2: Decomposition of a biconnected planar graph into a triconnected component tree.

The triconnected components can be canonized in log-space [**?**]. Hence, for triconnected component trees, compute their canonical invariant in log-space, i.e. two biconnected graphs are isomorphic if their trees are found to be equal.

In section 4.0.1, we summarize, how to canonize biconnected planar graphs by applying tree canonization ideas from [**?**] to their triconnected component trees. Note that, pairwise isomorphism of two trees labelled with the canons of their components does not imply isomorphism of the corresponding graphs. Lindell 's algorithm and complexity analysis had to be modified in a non-trivial way for this step to work in log-space.

In section 4.0.2, we describe, how to canonize planar graphs using their biconnected component trees, again, using the basic structure of Lindell 's algorithm. The comparison algorithm refers to the biconnected component tree of the planar graph and when comparing biconnected components, to their triconnected component trees. This requires a detailed analysis of the interferences of both tree structures.

4.0.1 Canonization of biconnected planar graphs

Let S and T be two triconnected component trees for the biconnected planar graphs G and H, respectively. S and T are rooted at separating pair nodes, say $s = (a, b)$ and $t = (a', b')$. Therefore we also write $S_{(a,b)}$ and $T_{(a',b')}$. They have separating pair nodes at odd levels and triconnected component nodes at even levels. Figure 3 shows two trees to be compared.

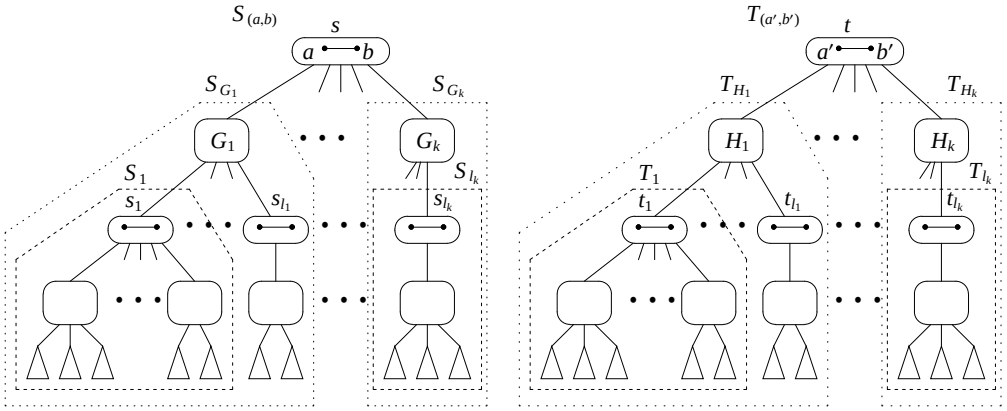


Figure 3: Triconnected component trees.

Similar as in Lindells algorithm, we define the isomorphism order of two triconnected component trees S and T rooted at separating pairs $s = (a, b)$ and $t = (a', b')$. $S_{(a,b)} <_T T_{(a',b')}$ if:

1. $|S_{(a,b)}| < |T_{(a',b')}|$ or

2. $|S_{(a,b)}| = |T_{(a',b')}|$ but $\#s < \#t$ or

3. $|S_{(a,b)}| = |T_{(a',b')}|$, $\#s = \#t = k$, but $(S_{G_1}, \ldots, S_{G_k}) <_T (T_{H_1}, \ldots, T_{H_k})$ lexicographically, where we assume that $S_{G_1} \leqslant_T \ldots \leqslant_T S_{G_k}$ and $T_{H_1} \leqslant_T \ldots \leqslant_T T_{H_k}$ are the ordered subtrees of $S_{(a,b)}$ and $T_{(a',b')}$, respectively. To compute the order between the subtrees S_{G_i} and T_{H_i} we compare lexicographically the canons of G_i and H_i and *recursively* the subtrees rooted at the children of G_i and H_i. Note that these children are again separating pair nodes.

4. $|S_{(a,b)}| = |T_{(a',b')}|$, $\#s = \#t = k$, $(S_{G_1} \leqslant_T \ldots \leqslant_T S_{G_k}) =_T (T_{H_1} \leqslant_T \ldots \leqslant_T T_{H_k})$, but $(O_1, \ldots, O_p) < (O'_1, \ldots, O'_p)$ lexicographically, where O_j and O'_j are the orientation counters of the j^{th} isomorphism classes I_j and I'_j of all the S_{G_i}'s and the T_{H_i}'s.

We say that two triconnected component trees S_e and $T_{e'}$ are *equal according to the isomorphism order*, denoted by $S_e =_T T_{e'}$, if neither $S_e <_T T_{e'}$ nor $T_{e'} <_T S_e$ holds. Two trees are $=_T$-equal, precisely when the underlying graphs are isomorphic.

We summarize now, how we can compute the isomorphism order when we compare subtrees rooted at separating pairs, e.g. $S_{(a,b)}$ and $T_{(a',b')}$, and when we compare subtrees rooted at triconnected components, e.g. S_{G_i} and T_{H_j}.

Comparing $S_{(a,b)}$ and $T_{(a',b')}$ is similar to the comparison of subtrees in Lindells algorithm. We make a cross-comparison of the children and store the counters $c_<, c_=, c_>$ for their order profile.

Assume, both subtrees are of equal size, i.e. $|S_{G_i}| = |T_{H_j}| = N$, both rooted at triconnected component nodes G_i and H_j, respectively.

First, we compare the types of G_i and H_j. We say that bonds $\leqslant_T$ cycles and cycles $\leqslant_T$ 3-connected components. 3-bonds are always equal. If both are cycles or 3-connected components then we construct the canons of G_i and H_j and compare all of them bit-by-bit.

To canonize a cycle, we traverse it starting from the virtual edge which corresponds to its parent (i.e. the parent node of G_i), and then traversing the entire cycle along the edges encountered. There are two possible traversals depending on which direction of the starting edge is chosen. Thus, a cycle has two possible canons.

To canonize a 3-connected component G_i, we use the log-space algorithm from Datta, Limaye, and Nimbhorkar [**?**]. The canon depends on the direction of the starting edge and additionally, on the embedding of the component G_i. For 3-connected components, there are two possible embeddings. Hence, we have up to four possible canons.

In the bit-by-bit comparison, we have to distinguish several cases. When we reach virtual edges in the comparison steps, we go into recursion at the subtrees rooted at the corresponding separating pairs. If we find in the recursion that one of the subtrees is smaller than the other, then we have found an inequality between the current canons we compare. We eliminate the canons which are not found to be minimal. At the end, if there remains a canon for G_i and for H_j, then both subtrees S_{G_i} and T_{H_j} are equal up to step 3.

Orientation counters. However, here it does not suffice to stop after step 3. We need a further comparison step to ensure that G and H are indeed isomorphic. To see this we give an example, consider Figure 4. Assume that s and t have two children each, G_1, G_2 and H_1, H_2 such that $G_1 \cong H_1$ and $G_2 \cong H_2$. Still we cannot conclude that G and H are isomorphic because it is possible that the isomorphism between G_1 and H_1 maps a to a' and b to b', but the isomorphism between G_2 and H_2 maps a to b' and b to a'. Then these two isomorphisms cannot be extended to an isomorphism between G and H.

To handle this problem, we introduce the notion of an *orientation of a separating pair*. A separating pair gets an orientation from subtrees rooted at its children. Also, every subtree rooted at a triconnected component node gives an orientation to the parent separating pair. If the orientation is consistent, then we define $S_{(a,b)} =_T T_{(a',b')}$ and we will show that G and H are isomorphic in this case.

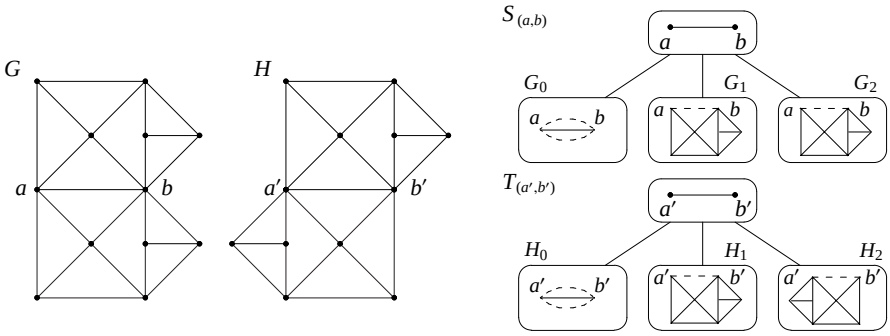


Figure 4:

We define the *orientation given to the parent separating pair of* G_i and H_j as the direction in which the minimum canon traverses this edge. If the minimum canons are obtained for both choices of directions of the edge, we say that S_{G_i} and T_{H_j} are *symmetric about their parent separating pair*, and thus do not give an orientation.

We define the *orientation given to the virtual edge in the parent triconnected component of* the corresponding separating pair node (a, b) or (a', b') by considering all the orientations given to the separating pair of their children $G_1, \ldots, G_k$, respectively. We first order the subtrees, say $S_{G_1} \leqslant_{\mathrm{T}} \cdots \leqslant_{\mathrm{T}} S_{G_k}$ and $T_{H_1} \leqslant_{\mathrm{T}} \cdots \leqslant_{\mathrm{T}} T_{H_k}$, and partition them into isomorphism classes, say $I_1, \ldots, I_p$ and $I'_1, \ldots, I'_p$. Let I_j be the smallest isomorphism class such that there are more components that give the orientation $a \to b$ to the parent than $b \to a$ (or vice versa). Then, we define $a \to b$ to be the *reference orientation* ($b \to a$ otherwise). For each isomorphism class I_j, we compute now the orientation counters $O_j = (c_j^{\rightarrow}, c_j^{\leftarrow})$ such that $c_j^{\rightarrow}$ is the number of children in I_j which give the reference orientation and $c_j^{\leftarrow}$ is the number of children in I_j which give the reverse orientation.

Recall the example of Figure 4. The graphs G and H have the same triconnected component trees but are not isomorphic. In $S_{(a,b)}$, the 3-bonds form one isomorphism class I_1 and the other two components form the second isomorphism class I_2, as they all are pairwise isomorphic. The non-isomorphism is detected by comparing the directions given to the parent separating pair. We have $p = 2$ isomorphism classes and for the orientation counters we have $O_1 = O'_1 = (0, 0)$, whereas $O_2 = (2, 0)$ and $O'_2 = (1, 1)$ and hence O'_2 is lexicographically smaller than O_2. Therefore we have $T_{(a',b')} <_{\mathrm{T}} S_{(a,b)}$.

Complexity. We argue now, that we can do the four comparison steps in logspace. The first and the second step are similar to Lindells algorithm. We define

the size of a separating pair node as 2 and the size of a triconnected component as the number of vertices in the component. For the third and fourth step, we have the following cases:

- When we compare two triconnected components G_i and H_j, then we have up to four canons. Suppose, we construct and compare two canons C_g and C_h and reach separating pairs (a, b) and (a', b'). We store the canons which are not eliminated, which of them C_g and C_h are and the direction of the virtual edges (a, b) and (a', b'). Hence, we need $O(1)$ bits.

- When we compare two separating pairs (a, b) and (a', b'), then we make a cross-comparison as in Lindells algorithm. Hence, we need counters $c_<, c_=, c_>$ to store the order profile. This way, we get the isomorphism classes. We further store the orientation counters O_j and O'_j for I_j and I'_j. We need $O(\log |I_j|)$ bits on the work-tape for all the counters.

However, we cannot guarantee yet, that the algorithm works in log-space. Let S_C be the subtree rooted at node C in a triconnected component tree. The problem is, that the subtrees (i.e. the children of C) where we go into recursion might be of size $> |S_C|/2$, we call it a *large child*.

To get around this problem, we first check whether the nodes C and C' have a large child. If so, then we compare them a priori and store the result of their comparison and the orientation given to the parent. Because C and C' have at most one large child, this needs only $O(1)$ additional bits. Whenever we would go into recursion at those large children, we just look at the work-tape for the result.

As seen above, while comparing two trees of size N, the algorithm uses no space for making a recursive call for a subtree of size larger than $N/2$, and it uses $O(\log k_j)$ space if the subtrees are of size at most N/k_j, where $k_j \geqslant 2$. Hence we get the same recurrence for the space $\mathcal{S}(N)$ as Lindell:

$$\mathcal{S}(N) \leqslant \max_j \mathcal{S}\left(\frac{N}{k_j}\right) + O(\log k_j),$$

where $k_j \geqslant 2$ for all j. Thus $\mathcal{S}(N) = O(\log N)$. Note that the number n of nodes of G is in general smaller than N, because the separating pair nodes occur in all components split off by this pair. But we certainly have $n < N \leqslant O(n^2)$ [**?**]. This proves the following theorem.

Theorem 4.1. *The isomorphism order between two triconnected component trees of biconnected planar graphs can be computed in log-space.*

The canon. Once we know the ordering among the subtrees, it is straight forward to output the canon of the triconnected component tree T. We traverse T in the tree isomorphism order as in Lindell [**?**], outputting the canon of each of the nodes along with virtual edges and delimiters. That is, we output a '[' while going down a subtree, and ']' while going up a subtree.

We need to choose a separating pair as root for the tree. Since there is no distinguished separating pair, we simply cycle through all of them and select the one, which leads to the minimum canon. Let (a, b) be this separating pair.

The canonization procedure has two steps. In the first step we compute what we call a *canonical list* for $S_{(a,b)}$. This is a list of the edges of G, also including virtual edges. In the second step we compute the final canon from the canonical list.

Canon of separating pair nodes. Consider a subtree $S_{(a,b)}$ rooted at (a, b). We start with computing the reference orientation of (a, b) with oracle calls to the canonical ordering algorithm and output the edge in this direction. Then we recursively output the canonical lists of the subtrees of node (a, b) according to the increasing isomorphism order. Among isomorphic siblings, those which give the reference orientation to the parent come first. We denote this canonical list of edges $l(S, a, b)$. If there is no reference orientation for a child, take the orientation of the parent (a, b).

Canon of triconnected component nodes. Consider the subtree S_{G_i} rooted at G_i. Let (a, b) be the parent separating pair of S_{G_i} with reference orientation (a, b). If G_i is a 3-bond then output $l(G_i, a, b) = (a, b)$. If G_i is a cycle then output $l(G_i, a, b) = (a, b)(b, v_1)(v_1, v_2) \ldots (v_n, b)$. If G_i is a 3-connected component then compute the minimum of two canons with an oracle call, with respect to the given reference orientation (a, b) and both embeddings for G_i. Output this canon as $l(G_i, a, b)$. Virtual edges are output in the direction of the reference orientation given to them, if any. Finally, we output the subtrees in the order we have virtual edges in the canon.

We give now an example. Consider the canonical list $l(S, a, b)$ of edges for the tree $S_{(a,b)}$ of Figure 3. Let s_i be the edge connecting the vertices a_i with b_i. We also write for short $l'(S_i, s_i)$ which is one of $l(S_i, a_i, b_i)$ or $l(S_i, b_i, a_i)$. The direction of s_i is as described above. Let $l_0 = 0$. Then we have:

$$\begin{aligned} l(S, a, b) &= [\ (a, b)\ l(S_{G_1}, a, b)\ \ldots\ l(S_{G_k}, a, b)\ ], \text{ where} \\ l(S_{G_i}, a, b) &= [\ l(G_i, a, b)\ [l'(S_{l_{i-1}+1}, s_{l_{i-1}+1})]\ \ldots\ [l'(S_{l_i}, s_{l_i})]\ ]\ ] \end{aligned}$$

4.0.2 Canonization of planar graphs

Consider the decomposition of a connected planar graph. For each articulation point and biconnected component we define nodes i.e. *articulation point nodes*

and *biconnected component nodes*. An articulation point node for a is connected by an edge to the nodes of biconnected components where a is contained as a vertex. The resulting graph is a tree, the *biconnected component tree*. The main difference to the triconnected component tree is, that for articulation point nodes, there is no concept of orientation as for separating pairs.

We define the isomorphism order for two biconnected component trees S_a and $T_{a'}$ rooted at nodes s and t corresponding to articulation points a and a', respectively. Also see Figure 5. Let $|S_a|$ be the sum of the sizes of the nodes in the tree. The size of an articulation point node a is defined as 1 and the size of a biconnected component node B is the size of its triconnected component tree $|\mathsf{T}(B)|$. Define $S_a <_{\mathtt{B}} T_{a'}$ if

1. $|S_a| < |T_{a'}|$ or
2. $|S_a| = |T_{a'}|$ but $\#s < \#t$ or
3. $|S_a| = |T_{a'}|$, $\#s = \#t = k$, but $(S_{B_1}, \ldots, S_{B_k}) <_{\mathtt{B}} (T_{B'_1}, \ldots, T_{B'_k})$ lexicographically, where we assume that $S_{B_1} \leqslant_{\mathtt{B}} \cdots \leqslant_{\mathtt{B}} S_{B_k}$ and $T_{B'_1} \leqslant_{\mathtt{B}} \cdots \leqslant_{\mathtt{B}} T_{B'_k}$ are the ordered subtrees of S_a and $T_{a'}$, respectively. To compare the order between the subtrees S_{B_i} and $T_{B'_j}$ we compare the triconnected component trees $\mathsf{T}(B_i)$ of B_i and $\mathsf{T}(B'_j)$ of B'_j and when we reach the first occurrences of some articulation points then we compare *recursively* the corresponding subtrees rooted at the children of B_i and B'_j. Note, that these children are again articulation point nodes.

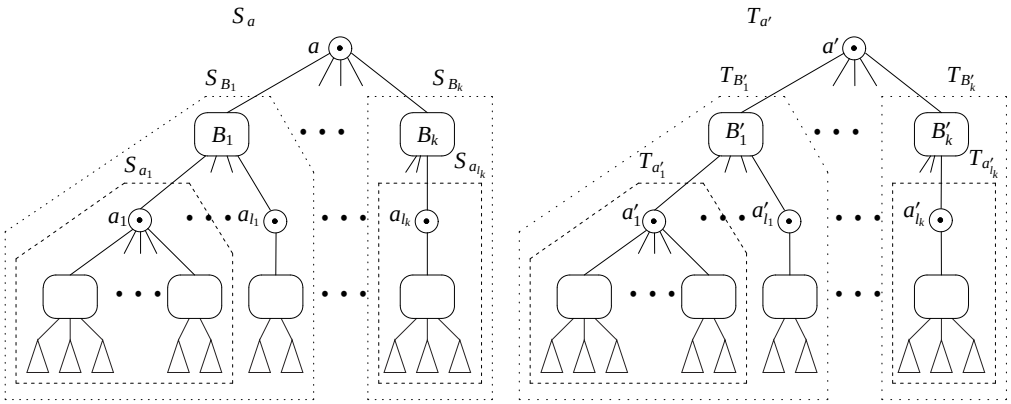


Figure 5: Biconnected component trees.

We say that two biconnected component trees are *equal*, denoted by $S_a =_{\mathtt{B}} T_{a'}$, if neither of $S_a <_{\mathtt{B}} T_{a'}$ and $T_{a'} <_{\mathtt{B}} S_a$ holds. The inductive ordering of the subtrees

of S_a and $T_{a'}$ proceeds exactly as in Lindell 's algorithm, by partitioning them into size-classes and comparing the children in the same size-class recursively.

We summarize now, how we can compute the isomorphism order when we compare subtrees rooted at articulation points, e.g. S_a and $T_{a'}$, and when we compare subtrees rooted at biconnected components, e.g. S_{B_i} and $T_{B'_j}$.

Comparing S_a and $T_{a'}$ is similar to the case when we compare subtrees rooted at separating pairs in triconnected component trees. We make a cross-comparison of the children and store the counters $c_<, c_=, c_>$ for their order profile.

When we compare biconnected components B_i and B'_j, then we cannot start comparing their biconnected canons. We even cannot compute their canons because we do not have a unique root separating pair for the trees $\mathsf{T}(B_i)$ and $\mathsf{T}(B'_j)$. The problem occurs when we have only one fixed vertex in B_i, i.e. the parent articulation point. Datta et. al. bound the number of candidates of root separating pairs of $T(B_i)$ and $T(B'_j)$. The detailed case analysis can be found in an elaborate version and is complex. Basically, except of some special cases they show that the number of edges is bounded by k, when all the isomorphism classes of the children of B_i and B'_j (i.e. children in the biconnected component tree of nodes for B_i and B'_j) are of cardinality $\geqslant k$. Hence, all the isomorphism classes contain children C such that $|S_C| \leqslant |S_{B_i}|/k$. If there is one size class of cardinality one, then we treat this child separately. If there are two or more such size classes, then we even get $O(1)$ candidates for the root. We will need this in the complexity analysis.

Complexity according to the biconnected component tree. First, when we compare articulation points a and a' in the biconnected component tree, then we have a similar complexity analysis as in Lindells algorithm. For the children of a and a', we store $O(\log k)$ bits for isomorphism classes of cardinality $k \geqslant 2$.

Second, when we compare biconnected components B and B' in the biconnected component tree then a typical query is of the form (s, r), where s is the chosen root of $\mathsf{T}(B)$ and r is the index of the edge in the canon, which is to be retrieved. If there are k choices for $\mathsf{T}(B)$ and $T(B')$, the base machine cycles through all of them one by one, keeping track of the minimum canon. This takes $O(\log k)$ space. In both cases, we also consider large children (i.e. children C of B such that $|S_C| > |S_B|/2$) a priori. We summarize. If we consider recursively how many bits we store for the roots of biconnected components then we get the recursion equation for the size function.

$$\mathcal{S}(N) = \max_j \left\{ \mathcal{S}\left(\frac{N}{k_j}\right) + O(\log k_j) \right\}$$

where $k_j \geqslant 2$. Hence, $\mathcal{S}(N) = O(\log N)$.

Complexity according to the triconnected component trees. We consider now the comparison of triconnected component trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$ of biconnected components B and B'. In the comparison of $\mathsf{T}(B)$ and $\mathsf{T}(B')$, we still go into recursion at separating pairs and when we reach virtual edges in canons for triconnected components. What is new, we go into recursion when we reach articulation points. For an example, see Figure 6.

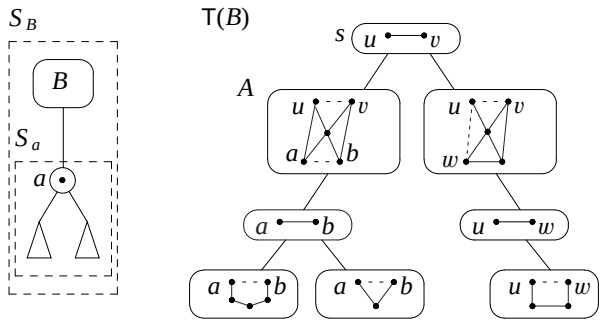


Figure 6: A biconnected component tree S_B rooted at biconnected component B which has an articulation point a as child, which occurs in the triconnected component tree $\mathsf{T}(B)$ of B. In A and the other triconnected components the dashed edges are separating pairs.

If an articulation point a belongs to many separating pairs, then it can occur in many component nodes in $\mathsf{T}(B)$. Recall, that we have a root for the tree. So, there exists a unique component A that is closest to the root, where a is contained. Observe, that the set of component nodes where a is contained is always a connected subtree in $\mathsf{T}(B)$. The authors show, that this unique component can be computed in log-space and that the first position where a occurs in the canon of A can be found in log-space. Exactly there, we go for a into recursion. For all the other occurrences of a we do not go into recursion. Call this the *reference copy* of a in $\mathsf{T}(B)$.

Assume we store the bits separately, which we need inside $\mathsf{T}(B)$ for all biconnected components B. Then we can prove for this part also a log-space bound.

Therefore, we refine the size function. Let C be a node in $\mathsf{T}(B)$. The size of the subtree S_C rooted at some node C is the sum of the size of the triconnected subtree rooted at C in $\mathsf{T}(B)$, say $|S_C|$ plus the size of all the biconnected subtrees $|S_a|$, if a is a reference copy of an articulation points in S_C. Hence, we get the same recursion equation as before. This finishes the complexity analysis. We get the following theorem.

Theorem 4.2. *The isomorphism order between two planar graphs can be computed in log-space.*

The canon. The canonization of planar graphs proceeds exactly as in the case of biconnected planar graphs. A log-space procedure traverses the biconnected component tree, makes oracle queries to the isomorphism order algorithm and outputs a canonical list of edges, along with delimiters to separate the lists for siblings. A log-space transducer then renames the vertices according to their first occurrence in this list, to get the final canon for the biconnected component tree. This canon depends upon the choice of the root of the biconnected component tree. Further log-space transducers cycle through all the articulation points as roots to find the minimum canon among them, then rename the vertices according to their first occurrence in the canon and finally, remove the virtual edges and delimiters to obtain a canon for the planar graph. This proves the main theorem.

Theorem 4.3. *A planar graph can be canonized in log-space.*

References