Planar Graph Isomorphism is in Log-Space*

Samir Datta[1] Nutan Limaye[2] Prajakta Nimbhorkar[2]

Thomas Thierauf[3†] Fabian Wagner[4‡]

[1] Chennai Mathematical Institute
sdatta@cmi.ac.in
[2] The Institute of Mathematical Sciences
{nutan,prajakta}@imsc.res.in
[3] Fak. Elektronik und Informatik, HTW Aalen
[4] Institut für Theoretische Informatik,
Universität Ulm, 89073 Ulm
{thomas.thierauf,fabian.wagner}@uni-ulm.de

June 15, 2009

Abstract

Graph Isomorphism is the prime example of a computational problem with a wide difference between the best known lower and upper bounds on its complexity. There is a significant gap between extant lower and upper bounds for planar graphs as well. We bridge the gap for this natural and important special case by presenting an upper bound that matches the known log-space hardness [JKMT03]. In fact, we show the formally stronger result that planar graph canonization is in log-space. This improves the previously known upper bound of AC^1 [MR91].

Our algorithm first constructs the biconnected component tree of a connected planar graph and then refines each biconnected component into a triconnected component tree. The next step is to log-space reduce the biconnected planar graph isomorphism and canonization problems to those for 3-connected planar graphs, which are known to be in log-space by [DLN08]. This is achieved by using the above decomposition, and by making significant modifications to Lindell's algorithm for tree canonization, along with changes in the space complexity analysis.

The reduction from the connected case to the biconnected case requires further new ideas, including a non-trivial case analysis and a group theoretic lemma to bound the number of automorphisms of a colored 3-connected planar graph. This lemma is crucial for the reduction to work in log-space.

*A preliminary version of the paper appeared at arXiv:0809.2319v1 and arXiv:0809.2319v2.
†Supported by DFG grants Scho 302/7-2 and TO 200/2-2.
‡Supported by DFG grants Scho 302/7-2 and TO 200/2-2.

Dagstuhl Seminar Proceedings 09421
Algebraic Methods in Computational Complexity
http://drops.dagstuhl.de/opus/volltexte/2010/2416

1 Introduction

The graph isomorphism problem GI consists of deciding whether there is a bijection between the nodes of two graphs, which preserves the adjacency relations. The wide gap between the known lower and upper bounds has kept alive the research interest in GI.

The problem is clearly in NP, and, by a group theoretic proof, also in SPP [AK06]. This is the current frontier of our knowledge as far as upper bounds go. The inability to give efficient algorithms for the problem would lead one to believe that the problem is provably hard. NP-hardness is precluded by a result that states if GI is NP-hard then the polynomial time hierarchy collapses to the second level [BHZ87, Sch88]. What is more surprising is that not even P-hardness is known for the problem. The best we know is that GI is hard for DET [Tor04], the class of problems NC^1-reducible to the determinant, defined by Cook [Coo85].

While this enormous gap has motivated a study of isomorphism in *general* graphs, it has also induced research in isomorphism restricted to special cases of graphs where this gap can be reduced. Tournaments are an example of directed graphs where the DET lower bound is preserved [Wag07], while there is a quasi-polynomial time upper bound [BL83].

Trees are an example of graphs where the lower and upper bounds match and are L [Lin92]. Note that for trees, the problem's complexity crucially depends on the input encoding: if the trees are presented as strings then the lower and upper bound are NC^1 [MJT98, Bus97]). Lindell's log-space result has been extended to partial 2-trees, also known as generalized series-parallel graphs [ADK08]. Trees and partial 2-trees are special cases of planar graphs.

In this paper we consider planar graph isomorphism and settle its complexity by significantly improving the known upper bound of AC^1. The result is particularly satisfying because Planar Graph Isomorphism turns out to be complete for a well-known and natural complexity class, namely log-space: L.

Planar Graph Isomorphism has been studied in its own right since the early days of computer science. Weinberg [Wei66] presented an $O(n^2)$ algorithm for testing isomorphism of 3-connected planar graphs. Hopcroft and Tarjan [HT74] extended this to general planar graphs, improving the time complexity to $O(n \log n)$. Hopcroft and Wong [HW74] further improved it to $O(n)$. Recently Kukluk, Holder, and Cook [KHC04] gave an $O(n^2)$ algorithm for planar graph isomorphism, which is suitable for practical applications. The parallel complexity of Planar Graph Isomorphism was first considered by Miller and Reif [MR91] and Ramachandran and Reif [RR94]. They showed that the upper bound is AC^1, see also [Ver07].

Recent work has dealt with a further special case, namely 3-connected planar graphs. Thierauf and Wagner [TW08] presented a new upper bound of $\mathsf{UL} \cap \mathsf{coUL}$, making use of the machinery developed for the reachability problem [RA97] and specifically for planar reachability [ADR05, BTV07]. They also show that the problem is L-hard. Further progress, in the form of a log-space algorithm is made by Datta, Limaye and Nimbhorkar [DLN08] where the 3-connected planar case is settled, by building on ideas from [TW08] and using Reingold's construction of universal exploration sequences [Rei05]. We summarise the known results for planar graphs and their restrictions as follows:

Graph class	Lower bound	Upper bound
Trees	L [MJT98]	L [Lin92]
Partial 2-trees	L	L [ADK08]
3-connected planar graphs	L [TW08]	L [DLN08]
Planar graphs	L	AC^1 [RR94]

The current work is a natural culmination of this series where we settle the complexity question for planar graph isomorphism by presenting the first log-space algorithm for the problem. In fact, 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, polynomial length string.

We consider planar undirected graphs without parallel edges and loops, also called *simple* graphs. For planar graphs that are not simple there are log-space many-one reductions to simple planar graphs (cf. [KST93]). Our algorithm consists of the following steps.

1. Decompose the planar graph into its biconnected components and construct a *biconnected component tree* in log-space ([ADK08], cf. [TW09] and Section 3).

2. Decompose biconnected planar components into their triconnected components to obtain a *triconnected component tree* in log-space. This is essentially a parallel implementation of the sequential algorithm of [HT73] (Section 3).

3. Invoke the algorithm of Datta, Limaye and Nimbhorkar [DLN08] to canonize the triconnected components of the graph.

4. Canonize biconnected planar graphs using their triconnected component trees. Lindell's algorithm [Lin92] for tree canonization and its complexity analysis had to be modified in a non-trivial way for this step to work in log-space (Section 4), i.e. from [Lin92] to their triconnected component trees.

5. Canonize planar graphs using their biconnected component trees, by substituting the biconnected components with their triconnected component trees (Section 5).

Notice, that in Step 4, pairwise isomorphism of two triconnected component trees labelled with the canons of their components does not imply isomorphism of the corresponding graphs. Figure 1 illustrates this fact. So, a naïve combination of [Lin92] and [DLN08] does not work. We need to introduce the concept of *orientations of separating pairs* (see Section 4.2 for details) to ensure the extendibility of isomorphism of individual 3-connected components to the entire biconnected planar graph.

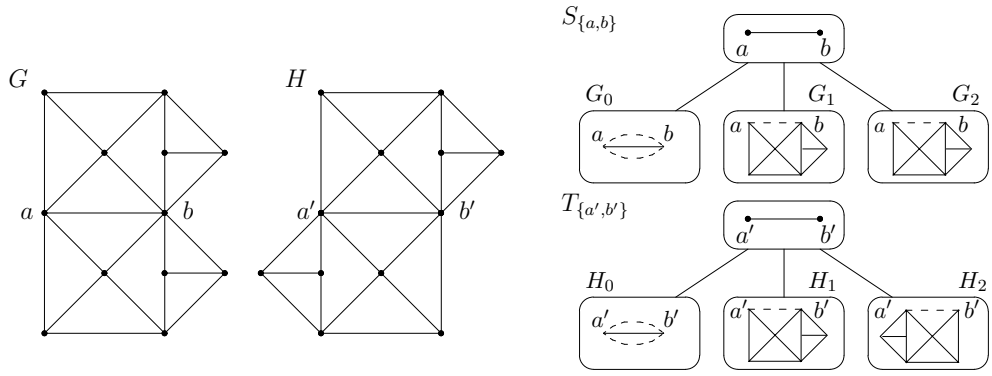

Figure 1: *The graphs G and H have the same triconnected component trees but are not isomorphic.*

Triconnected components have at most two embeddings on the sphere (cf. [Whi33]). This property helps to perform Step 4 in L. Biconnected components do not have this property. Hence, the naïve approach for Step 5 would need to keep track of exponentially many cases. We solve this problem by resorting to an intricate case analysis, and a group theoretic lemma (Lemma 5.3) to bound the number of automorphisms of a coloured 3-connected planar graph.

We emphasize that Step 5 does not use Step 4 as a black-box but as a co-routine. In fact, we maintain two (logarithmically bounded) work-tapes for the two steps.

2 Definitions and Notations

We recall some basic graph theoretic notions.

A graph $G = (V, E)$ is *connected* if there is a path between any two vertices in G. We consider (a, b) as an ordered pair and $\{a, b\}$ without any order of a and b. For $U \subseteq V$ let $G(U)$ be the *induced subgraph* of G on U. A vertex $v \in V$ is an *articulation point* if $G(V \setminus \{v\})$ is not connected. A pair of vertices $u, v \in V$ is a *separating pair* if $G(V \setminus \{u, v\})$ is not connected. A *biconnected graph* contains no articulation points. A *3-connected graph* contains no separating pairs. A *triconnected graph* is either a 3-connected graph or a cycle or a 3-bond. A *k-bond* is a graph consisting of two vertices joined by k edges. A pair of vertices $\{a, b\}$ is said to be 3-*connected* if there are three or more vertex-disjoint paths between them.

For a node v let $d(v)$ be the maximal distance that v has to any of the other nodes of G. Let C be the set of nodes v of G that have minimal value $d(v)$. The set C is called the *center of* G. In other words, vertices in the center minimize the maximal distance from other vertices in the graph. Note that if G is a tree such that every path from a leave to a leave has even length, then the center consists of only one node, namely the midpoint of a longest path in the tree.

Let E_v be the set of edges incident to v. A permuatation ρ_v on E_v that has only one cycle is called a *rotation*. A *rotation scheme* for a graph G is a set ρ of rotations,

$$\rho = \{\rho_v \mid v \in V \text{ and } \rho_v \text{ is a rotation on } E_v\}.$$

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*. Note that in this case ρ^{-1} is a planar rotation scheme as well. Allender and Mahajan [AM00] showed that a planar rotation scheme for a planar graph can be computed in log-space.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be *isomorphic* ($G_1 \cong G_2$) if there is a bijection $\phi : V_1 \to V_2$ such that $\{u, v\} \in E_1$ if and only if $\{\phi(u), \phi(v)\} \in E_2$. *Graph isomorphism* (GI) is the problem of deciding whether two given graphs are isomorphic.

A planar graph G, along with its planar embedding (given by ρ) is called a *plane graph* $\widehat{G} = (G, \rho)$. A plane graph divides the plane into regions. Each such region is called a *face*. Let Planar-GI be the special case of GI when the given graphs are planar. The biconnected (respectively, 3-connected) planar GI is a special case of Planar-GI when the graphs are biconnected (3-connected) planar graphs.

Let $\mathcal{G}$ be a class of graphs. Let $f : \mathcal{G} \to \{0, 1\}^*$ be a function such that for all $G, H \in \mathcal{G}$ we have $G \cong H \Leftrightarrow f(G) = f(H)$. Then f computes a *complete invariant* for $\mathcal{G}$. If f computes for G a graph $f(G)$ such that $G \cong f(G)$ then we call $f(G)$ the *canon* for G.

By L we denote the languages computable by a log-space bounded Turing machine.

3 Isomorphism Order of Connected Planar Graphs

In this section, we give a proof for the following theorem.

Theorem 3.1 *The decomposition of biconnected planar graphs into triconnected components is in log-space.*

Hopcroft and Tarjan [HT73] presented a sequential algorithm for the decomposition of a biconnected planar graph into its triconnected components. Their algorithm recursively removes separating pairs from the graph and puts a copy of the separating pair in each of the components so formed. The nodes in the separating pair are connected by a virtual edge. If simple cycles are split at any intermediate steps then they are combined later. This gives a decomposition which is unique [Mac37]. We describe a log-space algorithm for such a decomposition of a biconnected planar graph. We start with definitions and then prove some properties of separating pairs.

Definition 3.2 *In a plane graph $\widehat{G}$, a separating pair $\{a, b\}$ is said to* span a face f *if both its endpoints a, b lie on the boundary of f. Let $v_0, v_1, \ldots, v_k$ be a face boundary. Two separating pairs $\{v_i, v_j\}$, $\{v_{i'}, v_{j'}\}$ are called* intersecting *if $i < i' < j < j'$, and* non-intersecting *otherwise.*

Lemma 3.3 *Every separating pair spans some face.*

To see this, note that in a plane graph $\widehat{G}$, a split component of a separating pair is embedded in some face. This can be considered as the spanned face. A separating pair $\{a, b\}$ that spans a face f is called 3-*connected* if there are at least three vertex-disjoint paths between a, b i.e. there is a path between a, b in $\widehat{G}$ which is vertex-disjoint from the boundary of f. The following lemma enables us to remove all the 3-connected separating pairs simultaneously.

Lemma 3.4 *In a plane graph $\widehat{G}$, 3-connected separating pairs which span the same face are non-intersecting.*

Proof. Suppose $\{a, c\}$ and $\{b, d\}$ are two 3-connected intersecting separating pairs on face f in $\widehat{G}$ and let P be a path outside f from b to d. In particular, P does not pass through a or c.

As the pair b, d is 3-connected, it cannot be separated from the rest of the graph by any other separating pair. Let v be a vertex that gets separated from b and d when a and c are removed from the graph. Since v lies outside f, there is a path outside f from a via v to c. Since the graph is planar, this path must intersect P. Thus there is a path from v to b and d that does not pass through a or c. This contradicts the assumption that removal of a and c separated v from b and d. □

Definition 3.5 *Call a set of vertices $V' \subseteq V(\widehat{G})$* separable *if there exists a 3-connected separating pair $\{a, b\}$ in $V(\widehat{G})$ such that the removal of $\{a, b\}$ divides V' into different connected components. Otherwise V' is called* inseparable. *Given an inseparable triple $\tau = \{u, v, w\}$, define $C_\tau = \{x \mid \{u, v, w, x\}$ is inseparable$\}$.*

Note that the nodes of a simple cycle are trivially inseparable because there are no 3-connected separating pairs. The following lemma states that except for cycles, all biconnected graphs have 3-connected separating pairs and hence the sets C_τ defined above are the 3-connected components of such a graph.

Lemma 3.6 *Let G be a biconnected planar graph. If G is not 3-connected and not a cycle then G has a 3-connected separating pair.*

Proof. Let G be neither 3-connected nor a cycle and let a, b be a separating pair of G. If a and b are 3-connected then we are done. So assume that a and b are not 3-connected.

Let f be a face spanned by a and b. Then a and b are connected by two vertex-disjoint paths, say P_1 and P_2, which form the boundary of f, and the removal of $\{a, b\}$ separates these two paths. Since G is not a single cycle, it has more faces apart from f. Therefore f shares some of its edges with another face, say f'. Consider the common boundary between f and f'. The endpoints of this boundary, say $\{u, v\}$ have three vertex-disjoint paths between them, and hence are 3-connected.

Both u and v lie on P_1 or both lie on P_2, since otherwise P_1 and P_2 will not be separated on the removal of $\{a, b\}$. Without loss of generality, assume that $u, v \in P_1$. Let $P_1 = \{a = v_1, v_2, \ldots, v_k = b\}$ and consider all 3-connected pairs $\{v_i, v_j\}$ of vertices that lie on P_1. Pick a pair, say $\{v_i, v_j\}$, that is maximally apart on P_1. We claim that $\{v_i, v_j\}$ is a separating pair: if not, there exists a path outside f from $v_{i'}$ to v_j for some $i' < i$, or from $v_{j'}$ to v_i for some $j' > j$. In the first case, $\{v_{i'}, v_j\}$ is a 3-connected pair that is further apart than $\{v_i, v_j\}$, in the second case the same holds for $\{v_i, v_{j'}\}$. But this contradicts the choice of $\{v_i, v_j\}$. $\square$

Hence, with inseparable triples we can compute triconnected components. If a triple of vertices is inseparable, then it is part of the same triconnected component. For distinct τ_1, τ_2, the sets C_{τ_1} and C_{τ_2} are either disjoint or identical. This allows us to identify any such C_τ with the lexicographical smallest τ_0 (considering the labels of vertices in τ lexicographically sorted) such that $C_\tau = C_{\tau_0}$. This is the approach of Algorithm 1 below.

Algorithm 1 Algorithm to decompose a graph into triconnected components.

Input: Biconnected planar graph $G = (V, E)$.
Output: The triconnected components of G.

1: fix a planar embedding $\widehat{G}$ of G.
2: **for all** faces f of $\widehat{G}$ **do**
3: $\quad S_f \leftarrow \{\{u, v\} | \{u, v\}$ is a 3-connected separating pair that spans $f\}$
4: $\quad S \leftarrow \cup_{f \in \widehat{G}} S_f$ the set of 3-connected separating pairs
5: **for all** $\{u, v\} \in S$ **do**
6: $\quad$ **if** $\{u, v\} \in E$ **then** output a 3-bond for $\{u, v\}$
7: compute the set of all inseparable triples $\tau_1, \ldots, \tau_k$
8: **for** $i \leftarrow 1$ to k **do** {compute 3-connected components}
9: $\quad$ **if** $\forall h < i \quad \tau_i \cup \tau_h$ is a separable set **then**
10: $\quad\quad C_i \leftarrow \tau_i$ {create new 3-connected component using inseparable triple τ_i }
11: $\quad\quad$ **for** $j \leftarrow i + 1$ to k **do**
12: $\quad\quad\quad$ **if** $\tau_i \cup \tau_j$ is an inseparable set **then** $C_i \leftarrow C_i \cup \tau_j$
13: $\quad\quad$ output the induced subgraph on C_i without edges corresponding to 3-bonds, including virtual edges $\{s \in S \mid s \subseteq C_i\}$

First, the algorithm computes all 3-connected separating pairs in the set S. From these, we get all the 3-bonds. The **for**-loop from line 8 on computes the 3-connected components C_τ: In line 9, we search for the first inseparable triple $\tau \notin \{C_{\tau_h} \mid 1 \leq h < i\}$ that can be separated from all previous ones. In lines 11 and 12, we search for all $\tau_j \subseteq C_{\tau_i}$. By Lemma 3.6,

it suffices to consider the pairs in S to check whether a set is separable or not. The set C_i finally equals C_{τ_i}. In line 13 we compute the triconnected component induced by C_{τ_i}. An example of a decomposition is provided in Figure 2.

Each step in the algorithm can be implemented in log-space. For instance, a combinatorial embedding for planar graphs can be computed in log-space [AM00]. Separating pairs, inseparable triples and the triconnected components can be computed in log-space, making oracle queries to undirected reachability [Rei05].

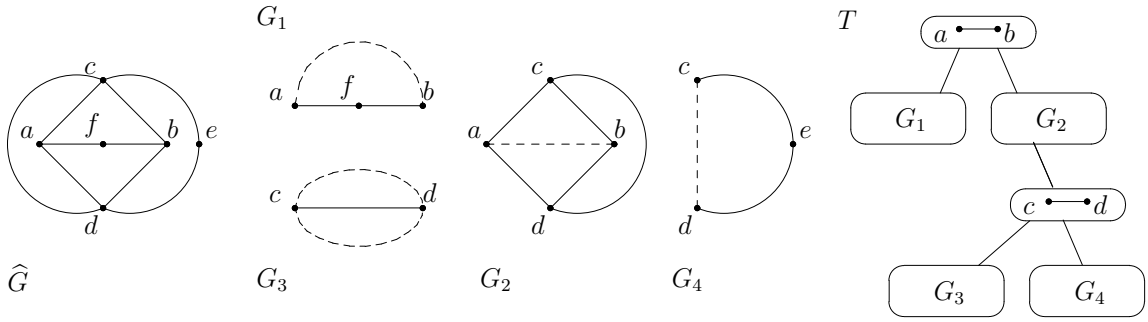

Figure 2: *The decomposition of a biconnected planar graph $\widehat{G}$. Its triconnected components are $G_1, \ldots, G_4$ and the corresponding triconnected component tree is T. In $\widehat{G}$, the pairs $\{a, b\}$ and $\{c, d\}$ are 3-connected separating pairs. The inseparable triples are $\{a, b, c\}$, $\{b, c, d\}$, $\{a, c, d\}$, $\{a, b, d\}$, $\{a, b, f\}$, and $\{c, d, e\}$. Hence the triconnected components are the induced graphs G_1 on $\{a, b, f\}$, G_2 on $\{a, b, c, d\}$, and G_4 on $\{c, d, e\}$. Since the 3-connected separating pair $\{c, d\}$ is connected by an edge in $\widehat{G}$, we also get $\{c, d\}$ as triple-bond G_3. The virtual edges corresponding to the 3-connected separating pairs are drawn with dashed lines.*

The triconnected component tree. Construct a graph T such that its nodes correspond to triconnected components and separating pairs, see Figure 2. There is an edge between a *triconnected component node* and a *separating pair node* if the vertices of the separating pair are contained in the triconnected component. Two triconnected component nodes or separating pair nodes do not share an edge.

It is easy to see that T is a tree, referred to as the *triconnected component tree of G*. Conversely, given T, we define $\mathsf{graph}(T) = G$, the graph which has the triconnected component tree T. We list some properties of T.

Lemma 3.7 *The graph T defined above has the following properties:*

1. *T is a tree and all the leaves of T are triconnected components.*

2. *Each path in T is an alternating path of separating pairs and triconnected components. Hence, a path between two leaves always contains an odd number of nodes and therefore T has a unique center node.*

3. *With an arbitrary separating pair node as root, T has odd depth.*

4. *A 3-bond is introduced as a child of a separating pair only as an indicator that the vertices of the separating pair have an edge between them in G. Hence a 3-bond is*

always a leaf node. In [HT73] it is a k-bond, where k is the number of components formed by the removal of the separating pair. Observe, k is the number of children of its parent separating pair and can be computed easily.

Proof. We only show the first claim. Suppose T has a cycle C. By definition, C is an alternating cycle of separating pairs and triconnected components, $C = (p_1, c_1, p_2, c_2 \ldots, p_r, c_r, p_1)$. Remove any separating pair p_i from C. Then the triconnected components c_{i-1} and c_i remain connected through the other elements of the cycle, contradicting the assumption that p_i separates them. □

4 Canonization of Biconnected Planar Graphs

In this section, we give a log-space algorithm to canonize biconnected planar graphs. For this, we define an isomorphism ordering on triconnected component trees which is similar to that of Lindell's tree isomorphism ordering. We first give a brief overview of Lindell's algorithm and then describe our canonization procedure.

4.1 Overview of Lindell's Algorithm for Tree Canonization

Lindell [Lin92] gave a log-space algorithm for tree canonization. The algorithm is based on an order relation $\leq$ on trees defined below. The order relation has the property that two trees S and T are isomorphic if and only if $S = T$. Because of this property it is called a *canonical order*. Clearly, an algorithm that decides the order can be used as an isomorphism test. Lindell showed how to extend such an algorithm to compute a canon for a tree in log-space. Let S and T be two trees with root s and t, respectively. The canonical order is defined $S < T$ if:

1. $|S| < |T|$, or
2. $|S| = |T|$ but $\#s < \#t$, where $\#s$ and $\#t$ are the number of children of s and t, respectively, or
3. $|S| = |T|$ and $\#s = \#t = k$, but $(S_1, \ldots, S_k) < (T_1, \ldots, T_k)$ lexicographically, where it is inductively assumed that $S_1 \leq \ldots \leq S_k$ and $T_1 \leq \ldots \leq T_k$ are the ordered subtrees of S and T rooted at the k children of s and t, respectively.

The comparisons in steps 1 and 2 can be made in log-space. Lindell proved that even the third step can be performed in log-space using *two-pronged* depth-first search, and *cross-comparing* only a child of S with a child of T. This is briefly described below:

- Find the number of minimal sized children of s and t. If these numbers are different then the tree with a larger number of minimal children is declared to be smaller. If equality is found then remember the minimal size and check for the next size. This process is continued till an inequality in the sizes is detected or all the children of s and t are exhausted.
- If s and t have the same number of children of each size then assume that the children of s and t are partitioned into *size-classes* (referred to as *blocks* in [Lin92]) in the

increasing order of the the sizes of the subtrees rooted at them. That is, the k children of s and t are partitioned into groups, such that the i-th group is of cardinality k_i and the subtrees in the i-th group all have size N_i, where $N_1 < N_2 < \cdots$. It follows that $\sum_i k_i = k$ and $\sum_i k_i N_i = n-1$. Then compare the children in each size-class recursively as follows:

Case 1, $k = 0$**.** Hence s and t have no children. They are isomorphic as all one-node trees are isomorphic. We conclude that $S = T$.

Case 2, $k = 1$**.** Recursively consider the grand-children of s and t. No space is needed for the recursive call.

Case 3, $k \geq 2$**.** For each of the subtrees S_j compute its order profile. The order profile consists of three counters, $c_<$, $c_>$ and $c_=$. These counters indicate the number of subtrees in the size-class of S_j that are respectively smaller than, greater than, or equal to S_j. The counters are computed by making cross-comparisons.

Note, that isomorphic subtrees in the same size-class have the same order profile. Therefore, it suffices to check that each such order profile occurs the same number of times in each size-class in S and T. To perform this check, compare the different order profiles of every size class in lexicographic order. The subtrees in the size-class i of S and T, which is currently being considered, with a count $c_< = 0$ form the first isomorphism class. The size of this isomorphism class is compared across the trees by comparing the values of the $c_=$ variables. If these values match then both trees have the same number of minimal children. Note that the lexicographical next larger order profile has the current value of $c_< + c_=$ as its value for the $c_<$-counter.

This way, one can loop through all the order profiles. If a difference in the order profiles of the subtrees of S and T is found then the lexicographical smaller order profile defines the smaller tree.

The last order profile considered is the one with $c_< + c_= = k$ for the current counters. If this point is passed without uncovering an inequality then the trees must be isomorphic and it follows that $S = T$.

Since $\sum_i k_i N_i \leq n$, the following recursion equation for the space complexity holds. For each new size class, the work-tape allocated for the former computations can be reused.

$$\mathcal{S}(n) = \max_i \{\mathcal{S}(N_i) + O(\log k_i)\} \leq \max_i \{\ \mathcal{S}\left(\frac{n}{k_i}\right) + O(\log k_i)\},$$

where $k_i \geq 2$ for all i. It is not hard to see that $\mathcal{S}(n) = O(\log n)$.

4.2 Isomorphism Order of Triconnected Component Trees

We describe now an isomorphism order procedure for two triconnected component trees S and T, corresponding to two biconnected planar graphs G and H, respectively. We root S and T at separating pair nodes $s = \{a, b\}$ and $t = \{a', b'\}$, respectively, which are chosen arbitrarily. Note, an isomorphism test can easily run through all posibilities of choosing these roots. The rooted trees are denoted as $S_{\{a,b\}}$ 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.

Our isomorphism ordering procedure is more complex than Lindell's algorithm, because each node of the tree is a separating pair or a triconnected component. Thus, unlike in the case of Lindell's algorithm, two leaves in a triconnected component tree are not always isomorphic.

In [DLN08], a log-space canonization algorithm for 3-connected planar graphs is described. Note that, an obvious way to canonize a triconnected component tree would be to invoke the algorithm of [DLN08] along with Lindell's algorithm. However, this approach does not work. See for example, Figure 1 on page 3. In the easiest case, the components to these leaves are not of the same size. We start by defining the size of a triconnected component tree.

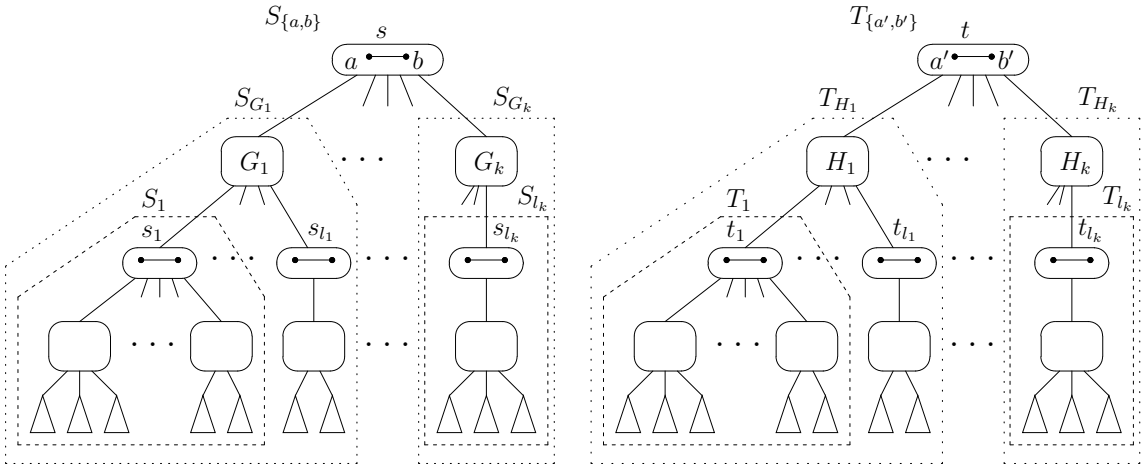

Figure 3: *Triconnected component trees.*

Definition 4.1 *For a triconnected component tree T, the* size of an individual component node C *of T is the number n_C of nodes in C. Note that the separating pair nodes are counted in in every component where they occur. The* size of the tree T, *denoted by $|T|$, is the sum of the sizes of its component nodes.*

Note that the size of T is at least as large as the number of vertices in $\mathsf{graph}(T)$, the graph corresponding to the triconnected component tree T.

We define the isomorphism order $<_{\mathsf{T}}$ for $S_{\{a,b\}}$ and $T_{\{a',b'\}}$ by first comparing their sizes, then the number of children of s and t. These two steps are exactly the same as in Lindell's algorithm. If equality is found in these two steps, then in the third step we make recursive comparisons of the subtrees of $S_{\{a,b\}}$ and $T_{\{a',b'\}}$. However, here it does not suffice to compare the order profiles of the subtrees in the different size classes as in Lindell's algorithm explained above. We need a further comparison step to ensure that G and H are indeed isomorphic.

To see this 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. For an example see Figure 1 of Page 3.

To handle this, 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 $S_{\{a,b\}} =_{\mathtt{T}} T_{\{a',b'\}}$ and we will show that G and H are isomorphic in this case.

Isomorphism order of two subtrees rooted at triconnected components. We consider the isomorphism order of two subtrees S_{G_i} and T_{H_j} rooted at triconnected component nodes G_i and H_j, respectively. We distinguish the following cases.

Case 1, *G_i and H_j are of different types.* G_i and H_j can be either 3-bonds or cycles or 3-connected components. If the types of G_i and H_j are different, we immediately detect an inequality, as it suffices to check whether each of them is a cycle or a 3-bond or neither of them. We define a canonical order among subtrees rooted at triconnected components in this ascending order: 3-bond, cycle, 3-connected component, such that e.g. $S_{G_i} <_{\mathtt{T}} T_{H_j}$ if G_i is a 3-bond and H_j is a cycle.

Case 2, *G_i and H_j are 3-bonds.* In this case, S_{G_i} and T_{H_j} are leaves, immediately define $S_{G_i} =_{\mathtt{T}} T_{H_j}$. Clearly, $G_i \cong H_j$ as all 3-bonds are isomorphic.

Case 3, *G_i and H_j are cycles or 3-connected components.* We construct the canons of G_i and H_j and compare them bit-by-bit. To canonize a cycle, we traverse it starting from the virtual edge that corresponds to its parent, 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 [DLN08]. Besides G_i, the algorithm gets as input a starting edge and a combinatorial embedding ρ of G_i. We always take the virtual edge $\{a, b\}$ corresponding to G_i's parent as the starting edge. Then there are two choices for the direction of this edge, (a, b) or (b, a). Further, a 3-connected graph has two planar combinatorial embeddings [Whi33]. Hence, there are four possible ways to canonize G_i.

We start the canonization of G_i and H_j in all the possible ways (two if they are cycles and four if they are 3-connected components), and compare these canons bit-by-bit. Let C_g and C_h be two canons to be compared. The base case is that G_i and H_j are leaf nodes and therefore contain no further virtual edges. In this case we use the lexicographic order between C_g and C_h. If G_i and H_j contain further virtual edges then these edges are specially treated in the bitwise comparison of C_g and C_h:

1. If a virtual edge is traversed in the construction of one of the canons C_g or C_h but not in the other, then we define the one without the virtual edge to be the *smaller canon.*

2. If C_g and C_h encounter virtual edges $\{u, v\}$ and $\{u', v'\}$ corresponding to a child of G_i and H_j, respectively, we need to recursively compare the subtrees rooted at $\{u, v\}$ and $\{u', v'\}$. If we find in the recursion that one of the subtrees is smaller than the other, then the canon with the smaller subtree is defined to be the *smaller canon.*

3. If we find that the subtrees rooted at $\{u, v\}$ and $\{u', v'\}$ are equal then we look at the orientations given to $\{u, v\}$ and $\{u', v'\}$ by their children. This orientation, called the *reference orientation*, is defined below. If one of the canons traverses the virtual edge

in the direction of its reference orientation but the other one not, then the one with the same direction is defined to be the *smaller canon.*

We eliminate the canons which were found to be the larger canons in at least one of the comparisons. In the end, the canons that are not eliminated are the *minimum canons*. If we have minimum canons for both G_i and H_j then we define $S_{G_i} =_{\mathtt{T}} T_{H_j}$. The construction of the canons also defines an isomorphism between the subgraphs described by S_{G_i} and T_{H_j}, i.e. $\mathsf{graph}(S_{G_i}) \cong \mathsf{graph}(T_{H_j})$. For a single triconnected component this follows from Datta, Limaye, and Nimbhorkar [DLN08]. If the trees contain several components, then our definition of $S_{G_i} =_{\mathtt{T}} T_{H_j}$ guarantees that we can combine the isomorphisms of the components to an isomorphism between $\mathsf{graph}(S_{G_i})$ and $\mathsf{graph}(T_{H_j})$.

Finally, 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. This finishes the description of the order for the case of subtrees rooted at triconnected components.

Observe, that we do not need to compare the sizes and the degree of the root nodes of S_{G_i} and T_{H_j} in an intermediate step, as it is done in Lindell's algorithm for subtrees. That is, because the degree of the root node G_i is encoded as the number of virtual edges in G_i. The size of S_{G_i} is checked by the length of the minimal canons for G_i and when we compare the sizes of the children of the root node G_i with those of H_j.

Isomorphism order of two subtrees rooted at separating pairs. The first three steps of the isomorphism ordering are performed similar to that of [Lin92] maintaining the order profiles. Now we assume that the subtrees are partitioned into isomorphism classes. The additional step involves comparison of *orientations* given by the corresponding isomorphism classes defined as follows:

Let $(G_1, \ldots, G_k)$ be the children of the root $\{a, b\}$ of $S_{\{a,b\}}$, and $(S_{G_1}, \ldots, S_{G_k})$ be the subtrees rooted at $(G_1, \ldots, G_k)$. Similarly let $(H_1, \ldots, H_k)$ be the children of the root $\{a', b'\}$ of $T_{\{a',b'\}}$ and $(T_{H_1}, \ldots, T_{H_k})$ be the subtrees rooted at $(H_1, \ldots, H_k)$. We first order the subtrees, say $S_{G_1} \leq_{\mathtt{T}} \cdots \leq_{\mathtt{T}} S_{G_k}$ and $T_{H_1} \leq_{\mathtt{T}} \cdots \leq_{\mathtt{T}} T_{H_k}$, and verify that $S_{G_i} =_{\mathtt{T}} T_{H_i}$ for all i. If we find an inequality then the one with the smallest index i defines the order between $S_{\{a,b\}}$ and $T_{\{a',b'\}}$. Now assume that $S_{G_i} =_{\mathtt{T}} T_{H_i}$ for all i. Inductively, the corresponding split components are isomorphic, i.e. $\mathsf{graph}(S_{G_i}) \cong \mathsf{graph}(T_{H_i})$ for all i.

The next comparison concerns the orientation of $\{a, b\}$ and $\{a', b'\}$. We already explained above the orientation given by each of the S_{G_i}'s to $\{a, b\}$. We define a *reference orientation* for the root nodes $\{a, b\}$ and $\{a', b'\}$ which is given by their children. This is done as follows. We partition $(S_{G_1}, \ldots, S_{G_k})$ into classes of isomorphic subtrees, say $I_1 <_{\mathtt{T}} \ldots <_{\mathtt{T}} I_p$ for some $p \leq k$, and similar $(T_{H_1}, \ldots, T_{H_k})$ into $I'_1 <_{\mathtt{T}} \ldots <_{\mathtt{T}} I'_p$. It follows that I_j and I'_j contain the same number of subtrees for every j.

- Consider the orientation given to $\{a, b\}$ by an isomorphism class I_j: For each isomorphism class I_j we compute an *orientation counter*, which is a pair $O_j = (c_j^{\rightarrow}, c_j^{\leftarrow})$, where $c_j^{\rightarrow}$ is the number of subtrees of I_j which give one orientation, say (a, b), and $c_j^{\leftarrow}$ is the number of subtrees from I_j which give the other orientation, (b, a). The larger number decides the orientation given to $\{a, b\}$. If these numbers are equal, or if each component

in this class is symmetric about $\{a,b\}$ then no orientation is given to $\{a,b\}$ by this class, and the class is said to be *symmetric about* $\{a,b\}$. Note that in an isomorphism class, either all or none of the components are symmetric about the parent.

- The *reference orientation of* $\{a,b\}$ is defined as the orientation given to $\{a,b\}$ by the smallest non-symmetric isomorphism class. If all isomorphism classes are symmetric about $\{a,b\}$, then we say that $\{a,b\}$ has *no reference orientation*.

  We order all the orientation counters $O_j = (c_j^{\rightarrow}, c_j^{\leftarrow})$ such that the first component $c_j^{\rightarrow}$ is the counter for the reference orientation of $\{a,b\}$.

Let $O'_j = (d_j^{\rightarrow}, d_j^{\leftarrow})$ be the corresponding orientation counters for the isomorphism classes I'_j. Now we compare the orientation counters O_j and O'_j for $j = 1, \ldots, p$. If they are all pairwise equal, then the graphs G and H are isomorphic and we define $S_{\{a,b\}} =_{\mathrm{T}} T_{\{a',b'\}}$. Otherwise, let j be the smallest index such that $O_j \neq O'_j$. Then we define $S_{\{a,b\}} <_{\mathrm{T}} T_{\{a',b'\}}$ if O_j is lexicographically smaller than O'_j, and $T_{\{a',b'\}} <_{\mathrm{T}} S_{\{a,b\}}$ otherwise. This finishes the definition of the order. For an example, see Figure 1. 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 $T_{\{a',b'\}} <_{\mathrm{T}} S_{\{a,b\}}$.

Summary of the steps in the isomorphism order. The isomorphism order of two triconnected component trees S and T rooted at separating pairs $s = \{a,b\}$ and $t = \{a',b'\}$ is defined $S_{\{a,b\}} <_{\mathrm{T}} T_{\{a',b'\}}$ 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 $(S_{G_1}, \ldots, S_{G_k}) <_{\mathrm{T}} (T_{H_1}, \ldots, T_{H_k})$ lexicographically, where we assume that $S_{G_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} S_{G_k}$ and $T_{H_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T_{H_k}$ 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. $|S_{\{a,b\}}| = |T_{\{a',b'\}}|$, $\#s = \#t = k$, $(S_{G_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} S_{G_k}) =_{\mathrm{T}} (T_{H_1} \leq_{\mathrm{T}} \ldots \leq_{\mathrm{T}} T_{H_k})$, but $(O_1, \ldots, O_p) < (O'_1, \ldots, O'_p)$ lexicographically, where O_j and O'_j are the orientation counters of the jth 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 $S_e =_{\mathrm{T}} T_{e'}$, if neither $S_e <_{\mathrm{T}} T_{e'}$ nor $T_{e'} <_{\mathrm{T}} S_e$ holds. The following theorem states that two trees are $=_{\mathrm{T}}$-equal, precisely when the underlying graphs are isomorphic.

Theorem 4.2 *The biconnected planar graphs G and H are isomorphic if and only if there is a choice of separating pairs e, e' in G and H such that $S_e =_{\mathrm{T}} T_{e'}$ when rooted at e and e', respectively.*

Proof. Assume that $S_e =_{\mathtt{T}} T_{e'}$. The argument is an induction on the depth of the trees that follows the inductive definition of the isomorphism order. The induction goes from depth d to $d+2$. If the grandchildren of separating pairs, say s and t, are $=_{\mathtt{T}}$-equal up to step 4, then we compare the children of s and t. If they are equal then we can extend the $=_{\mathtt{T}}$-equality to the separating pairs s and t.

When subtrees are rooted at separating pair nodes, the comparison describes an order on the subtrees which correspond to split components of the separating pairs. The order describes an isomorphism among the split components.

When subtrees are rooted at triconnected component nodes, say G_i and H_j, the comparison states equality if the components have the same canon, i.e. are isomorphic. By the induction hypothesis we know that the children rooted at virtual edges of G_i and H_j are isomorphic. The equality in the comparisons inductively describes an isomorphism between the vertices in the children of the root nodes.

Hence, the isomorphism between the children at any level can be extended to an isomorphism between the corresponding subgraphs in G and H and therefore to G and H itself.

The reverse direction holds obviously as well. Namely, if G and H are isomorphic and there is an isomorphism that maps the separating pair $\{a, b\}$ of G to the separating pair $\{a', b'\}$ of H, then the triconnected component trees $S_{\{a,b\}}$ of G and $T_{\{a',b'\}}$ of H rooted respectively at $\{a, b\}$ and $\{a', b'\}$ will clearly be equal. Hence, such an isomorphism mapps separating pairs of G onto separating pairs of H. This isomorphism describes a permutation on the split components of separating pairs, which means we have a permutation on triconnected components, the children of the separating pairs. By induction hypothesis, the children (at depth $d+2$) of two such triconnected components are isomorphic and equal according to $=_{\mathtt{T}}$. More formally, one can argue inductively on the depth of $S_{\{a,b\}}$ and $T_{\{a',b'\}}$. □

4.3 Space Complexity of the Isomorphism Order Algorithm

We analyse the space complexity of the isomorphism order algorithm. The first two steps of the isomorphism order algorithm can be computed in log-space as in Lindell's algorithm [Lin92]. We show that steps 3 and 4 can also be performed in log-space. We use the algorithm of Datta, Limaye, and Nimbhorkar [DLN08] to canonize a triconnected component G_i of size n_{G_i} in space $O(\log n_{G_i})$.

Comparing two subtrees rooted at triconnected components. For this, we consider two subtrees S_{G_i} and T_{H_j} with $|S_{G_i}| = |T_{H_j}| = N$ rooted at triconnected component nodes G_i and H_j, respectively. The cases that G_i and H_j are of different types or are both 3-bonds are easy to handle. Assume now that both are cycles or 3-connected components. Then we start constructing and comparing all the possible canons of G_i and H_j. We eliminate the larger ones and make recursive comparisons whenever the canons encounter virtual edges simultaneously. We can keep track of the canons, which are not eliminated, in constant space.

Suppose we construct and compare two canons C_g and C_h and consider the moment when we encounter virtual edges $\{a, b\}$ and $\{a', b'\}$ in C_g and C_h, respectively. Now we recursively compare the subtrees rooted at the separating pair nodes $\{a, b\}$ and $\{a', b'\}$. Note, that we cannot afford to store the entire work-tape content. It suffices to store the information of

- the canons which are not eliminated,

- which canons encountered the virtual edges corresponding to $\{a, b\}$ and $\{a', b'\}$, and
- the direction in which the virtual edges $\{a, b\}$ and $\{a', b'\}$ were encountered.

This takes altogether $O(1)$ space.

When a recursive call is completed, we look at the work-tape and compute the canons C_G and C_h. Therefore, recompute the parent separating pair of the component, where the virtual edge $\{a, b\}$ is contained. With a look on the bits stored on the work-tape, we can recompute the canons C_g and C_h. Recompute for them, where $\{a, b\}$ and $\{a', b'\}$ are encountered in the correct direction of the edges and resume the computation from that point.

Although we only need $O(1)$ space per recursion level, we cannot guarantee yet, that the implementation of the algorithm described so far works in log-space. The problem is, that the subtrees where we go into recursion might be of size $> N/2$ and in this case the recursion depth can get too large. To get around this problem, we check whether G_i and H_j have a large child, before starting the construction and comparison of their canons. A *large child* is a child which has size $> N/2$. If we find a large child of G_i and H_j then we compare them a priori and store the result of their recursive comparison. Because G_i and H_j can have at most one large child each, this needs only $O(1)$ additional bits. Now, whenever the virtual edges corresponding to the large children from S_{G_i} and T_{H_j} are encountered simultaneously in a canon of G_i and H_j, the stored result can be used, thus avoiding a recursive call.

Comparing two subtrees rooted at separating pairs. Consider two subtrees $S_{\{a,b\}}$ and $T_{\{a',b'\}}$ of size N, rooted at separating pair nodes $\{a, b\}$ and $\{a', b'\}$, respectively. We start comparing all the subtrees S_{G_i} and T_{H_j} of $S_{\{a,b\}}$ and $T_{\{a',b'\}}$, respectively. These subtrees are rooted at triconnected components and we can use the implementation described above. Therefore, we store on the work-tape the counters $c_<, c_=, c_>$. If they turn out to be pairwise equal, we compute the orientation counters O_j and O'_j of the isomorphism classes I_j and I'_j, for all j. The isomorphism classes are computed via the order profiles of the subtrees, as in Lindell's algorithm.

When we return from recursion, it is an easy task to find $\{a, b\}$ and $\{a', b'\}$ again, since a triconnected component has a unique parent, which always is a separating pair node. Since we have the counters $c_<, c_=, c_>$ and the orientation counters on the work-tape, we can proceed with the next comparison.

Let k_j be the number of subtrees in I_j. The counters $c_<, c_=, c_>$ and the orientation counters need altogether at most $O(\log k_j)$ space. From the orientation counters we also get the reference orientation of $\{a, b\}$. Let N_j be the size of the subtrees in I_j. Then we have $N_j \leq N/k_j$. This would lead to a log-space implementation as in Lindell's algorithm except for the case that N_j is large, i.e. $N_j > N/2$.

We handle the case of large children as above: we recurse on large children a priori and store the result in $O(1)$ bits. Then we process the other subtrees of $S_{\{a,b\}}$ and $T_{\{a',b'\}}$. When we reach the size-class of the large child, we know the reference orientation, if any. Now we use the stored result to compare the orientations given by the large children to their respective parent, and return the result accordingly.

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 $k_j \geq 2$. Hence we get the same recurrence for

the space $\mathcal{S}(N)$ as Lindell:

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

where $k_j \geq 2$ 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 $n < N \leq O(n^2)$ [HT73]. This proves the following theorem.

Theorem 4.3 *The isomorphism order between two triconnected component trees of biconnected planar graphs can be computed in log-space.*

4.4 The Canon of a Biconnected Planar Graph

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 [Lin92], 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. Since there are less than n^2 many separating pairs, a log-space transducer can cycle through all of them and can determine the separating pair which, when chosen as the root, leads to the lexicographically minimum canon of S. We describe the canonization procedure for a fixed root, say $\{a, b\}$.

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.

Canonical list of a subtree rooted at a separating pair. Consider a subtree $S_{\{a,b\}}$ rooted at the separating pair node $\{a, b\}$. We start with computing the reference orientation of $\{a, b\}$ and output the edge in this direction. This can be done by comparing the children of the separating pair node $\{a, b\}$ according to their isomorphism order with the help of the oracle. Then we recursively output the canonical lists of the subtrees of $\{a, b\}$ according to the increasing isomorphism order. Among isomorphic siblings, those which give the reference orientation to the parent are considered before those which give the reverse orientation. We denote this canonical list of edges $l(S, a, b)$. If the subtree rooted at $\{a, b\}$ does not give any orientation to $\{a, b\}$, then take that orientation for $\{a, b\}$, in which it is encountered during the construction of the above canon of its parent.

Assume now, the parent of $S_{\{a,b\}}$ is a triconnected component. In the symmetric case, $S_{\{a,b\}}$ does not give an orientation of $\{a, b\}$ to its parent. Then take the reference orientation which is given to the parent of all siblings.

Canonical list of a subtree rooted at a triconnected component. Consider the subtree S_{G_i} rooted at the triconnected component node 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 its canonical list $l(G_i, a, b)$ as (a, b). If G_i is a cycle then it has a unique canonical list with respect to the orientation (a, b), that is $l(G_i, a, b)$.

Now we consider the case that G_i is a 3-connected component. Then G_i has two possible canons with respect to the orientation (a, b), one for each of the two embeddings. Query the oracle for the embedding that leads to the lexicographically smaller canonical list and output it as $l(G_i, a, b)$. If we encounter a virtual edge $\{c, d\}$ during the construction, we determine its reference orientation with the help of the oracle and output it in this direction. If the children of the virtual edge do not give an orientation, we output $\{c, d\}$ in the direction in which it is encountered during the construction of the canon for G_i. Finally, the children rooted at separating pair node $\{c, d\}$ are ordered with the canonical order procedure.

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.

$$\begin{aligned}
l(S, a, b) &= [\ (a, b)\ l(S_{G_1}, a, b)\ \ldots\ l(S_{G_k}, a, b)\ ],\ \text{where} \\
l(S_{G_1}, a, b) &= [\ l(G_1, a, b)\ l'(S_1, s_1)\ \ldots\ l'(S_{l_1}, s_{l_1})\ ] \\
&\vdots \\
l(S_{G_k}, a, b) &= [\ l(G_k, a, b)\ l'(S_{l_k}, s_{l_k})\ ]
\end{aligned}$$

Canon for the biconnected planar graph. This list is now almost the canon, except that the names of the nodes are still the ones they have in G. Clearly, a canon must be independent of the original names of the nodes. The final canon for $S_{\{a,b\}}$ can be obtained by a log-space transducer which relabels the vertices in the order of their first occurrence in this canonical list and outputs the list using these new labels.

Note that the canonical list of edges contains virtual edges as well, which are not a part of G. However, this is not a problem as the virtual edges can be distinguished from real edges because of the presence of 3-bonds. To get the canon for G, remove these virtual edges and the delimiters '[' and ']' in the canon for $S_{\{a,b\}}$. This is sufficient, because we describe here a bijective function f which transforms an automorphism ϕ of $S_{\{a,b\}}$ into an automorphism $f(\phi)$ for G with $\{a, b\}$ fixed. We get the following result.

Theorem 4.4 *A biconnected planar graph can be canonized in log-space.*

5 Canonization of Planar Graphs

In this section, we give a log-space algorithm for the canonization of planar graphs. The main part is to show how to canonize *connected* planar graphs. Then, if a given graph is not connected, we compute its connected components in log-space and canonize each of these components. The canons of the connected components are output in lexicographical increasing order. Hence, from now on we assume that the given planar graph is connected.

We decompose a planar graph into its biconnected components and then construct a tree on these biconnected components and articulation points. We refer to this tree as the *biconnected component tree*. We also refer to the components as *biconnected component nodes* and *articulation point nodes*. This tree is unique and can be constructed in log-space [ADK08].

Similar to triconnected component trees, we put a copy of an articulation point a into each of the components formed by the removal of a. Thus, an articulation point a has a copy in each of the biconnected components obtained by its removal.

Note that a naive approach is to color all copies of an articulation point with a particular color and check the isomorphism of these coloured biconnected components separately. However, this approach does not work as we do not know a priori which articulation points from the two graphs will be mapped to each other. Also, using this method, we can not ensure in log-space that all the copies of an articulation are mapped to the copies of another articulation point.

In the discussion below, we refer to a copy of an articulation point in a biconnected component B as an *articulation point in* B. Although an articulation point has at most one copy in each of the biconnected components, the corresponding triconnected component trees can have many copies of the same articulation point (if it belongs to a separating pair in the biconnected component).

Given a planar graph G, we root its biconnected component tree at an articulation point. During the isomorphism ordering of two such trees S and T, we can fix the root of S arbitrarily and make an equality test for all choices of roots for T. As there are $\leq n$ articulation points, a log-space transducer can cycle through all of them for the choice of the root for T. We state some properties of articulation points.

Lemma 5.1 *Let B be a biconnected component in S and $\mathsf{T}(B)$ be its triconnected component tree. Then the following holds.*

1. *S has a unique center, similar to a triconnected component tree.*
2. *If an articulation point a of S appears in a separating pair node s in $\mathsf{T}(B)$, then it appears in all the triconnected component nodes which are adjacent to s in $\mathsf{T}(B)$.*
3. *If an articulation point a appears in two nodes C and D in $\mathsf{T}(B)$, it appears in all the nodes that lie on the path between C and D in $\mathsf{T}(B)$. Hence, there is a unique node A in $\mathsf{T}(B)$ that contains a which is nearest to the center of $\mathsf{T}(B)$. We call A the triconnected component* associated with *a. Thus, we can uniquely associate each articulation point contained in B with a triconnected component in $\mathsf{T}(B)$.*

5.1 Isomorphism Order for Biconnected Component Trees

The isomorphism order for biconnected component trees is defined in three steps that correspond to the first three steps of the isomorphism order for triconnected component trees in Section 4.2 on page 13. We mention the main differences in the isomorphism ordering for biconnected component trees from that of triconnected component trees.

1. The biconnected component nodes are connected by articulation point nodes. The resulting graph is a tree similar to the tree of triconnected component nodes and separating pair nodes. For articulation points, we do not need the notion of orientation. Instead, we color the copy of the parent articulation point in a biconnected component with a distinct color and then the pairwise isomorphism among the subtrees of S and T can be extended to the isomorphism between the corresponding planar graphs G and H in a straight forward way.
2. When we compare biconnected components B and B', then we do not have an obvious, uniquely defined edge as root for the corresponding triconnected component trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$. The naive approach would be to cycle through all separating pairs and finally

define the one as root that leads to a minimum canon. However, that way we cannot guarantee that the algorithm works in log-space. Let n_B be the size of B. Note that there can be up to $O(n_B)$ separating pairs. When we go into a recursion at some point, we need to store the edge that is currently the root. That is, we need $O(\log n_B)$ space at one level of recursion and this is too much for an overall log-space bound. Hence, our major task will be to limit the number of possible choices of roots appropriately so that the algorithm runs in log-space.

3. There are some more nontrivial tasks, to guarantee the log-space bound. It is not obvious, what to store on the work-tape when we go into recursion at some node in S or some node in $\mathsf{T}(B)$ and, what can be recomputed. We also need a new definition of the size of a subtree, for the isomorphism ordering to work correctly in log-space.

The size of a triconnected component tree is defined in Definition 4.1 on page 10. Here we extend the definition to biconnected component trees.

Definition 5.2 *Let B be a biconnected component node in a biconnected component tree S, and let $\mathsf{T}(B)$ be the triconnected component tree of B. The* size of B *is defined as $|\mathsf{T}(B)|$ as in Definition 4.1. The size of an articulation point node in S is defined as* 1*. Note that the articulation points may be counted several times, namely in every component they occur. The* size of S*, denoted by $|S|$, is the sum of the sizes of its components.*

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 (see Figure 4). 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 $S_{B_1} \leq_{\mathtt{B}} \cdots \leq_{\mathtt{B}} S_{B_k}$ and $T_{B'_1} \leq_{\mathtt{B}} \cdots \leq_{\mathtt{B}} T_{B'_k}$ 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. When we reach the first occurrences of some articulation points in $\mathsf{T}(B_i)$ and $\mathsf{T}(B_{j'})$ (i.e. the *reference copies* of these articulation points as described later) 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.

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. The book-keeping required (e.g. the order profile of a node, the number of nodes in a size-class that have been compared so far) is similar to that in Lindell's algorithm. We discuss now how to compare two such subtrees S_B and $T_{B'}$, rooted at biconnected component nodes B and B', respectively.

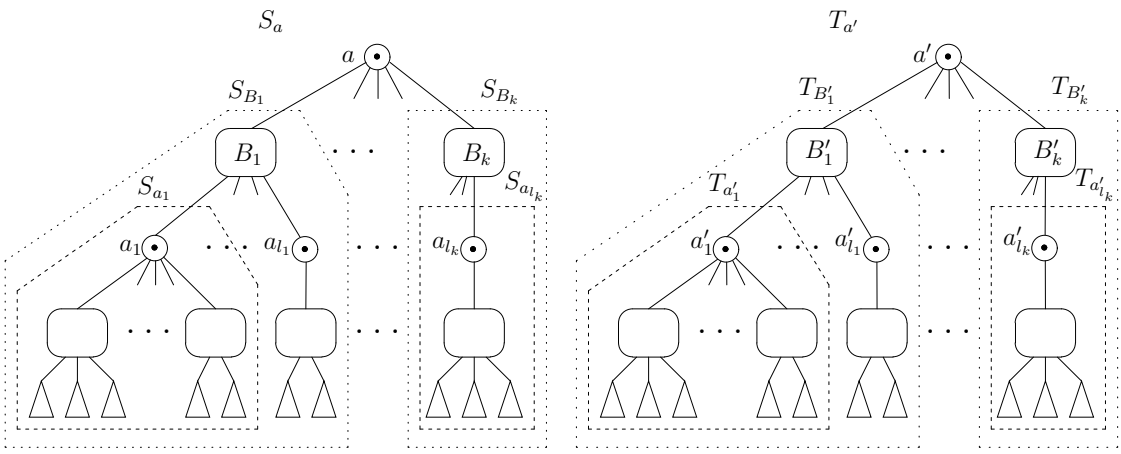

Figure 4: *Biconnected component trees.*

Isomorphism order of two subtrees rooted at biconnected components. We consider the isomorphism order of two subtrees S_{B_i} and $T_{B'_j}$ rooted at biconnected component nodes B_i and B'_j, and let a and a' be their parent articulation points, respectively. We start by constructing and comparing the canons of the triconnected component trees of B_i and B'_j. To do so, we choose a separating pair as root for each of them. As explained above, we cannot afford to simply try all possible choices as root. We will show below that we can compute in log-space a sufficiently small number of separating pairs as roots for B_i and B'_j which suffices for our purpose. That is, we make pairwise cross-comparisons of the canons obtained for these separating pairs as roots and determine the minimum canons.

The base case is that B_i and B'_j are leaf nodes and therefore contain no articulation points other than the parent articulation point. In this case, we can cycle through all the edges as roots and find the lexicographically smallest canon.

If B_i and B'_j contain articulation points, we go into recursion. Note that if an articulation point is part of a separating pair, it can occur several times in the triconnected component tree. To avoid recursion on the same pair of articulation points multiple times, we need to additionally keep track of whether a pair of articulation points is encountered for the first time in the comparison.

Also, while canonizing the triconnected component trees of B_i and B'_j, we give a separate color to the copy of a and a' in these trees, to ensure that the parent articulation points are always mapped to each other.

Limiting the number of possible choices for the root. Let S_a be a biconnected component tree rooted at articulation point a. Let B be a child of a in S_a and $\mathsf{T}(B)$ be the triconnected component tree of the biconnected component B. We show how to limit the number of potential root nodes for $\mathsf{T}(B)$. This is a crucial part to obtain a log-space bound.

Besides the parent a, let B have articulation points $a_1, \ldots, a_l$ for some integer $l \geq 0$, such that a_j is the root node of the subtree S_{a_j} of S_a (see Figure 4 on page 20). We partition the subtrees $S_{a_1}, \ldots, S_{a_l}$ into classes $E_1, \ldots, E_p$ of equal size subtrees (i.e. size according to Definition 5.2). Let k_j be the number of subtrees in E_j. Let the order of the size classes be such that $k_1 \leq k_2 \leq \cdots \leq k_p$. All articulation points with their subtrees in size class E_j are

colored with color j.

To limit the number of potential root nodes for $\mathsf{T}(B)$, we distinguish several cases below. The center of $\mathsf{T}(B)$, denoted by C, will play an important role thereby. In some of the cases we will show that the number of automorphisms of the center C is small. This already suffices for our purpose: in this case, for every edge as starting edge, we canonize the component C separately and construct a set of the edges A that lead to the minimum canon. Though the possible canons are polynomially many, the minimum ones are bounded by the number of automorphisms of C, which are small. Note that, in this process, we do not recurse on the separating pairs or articulation points.

Now we take the first separating pair encountered in each of the canons obtained from edges in A as starting edges. This set of separating pairs forms the potential root nodes for $\mathsf{T}(B)$, and hence its cardinality is bounded by the number of automorphisms of C.

If B contains no separating pairs (i.e. B=C), we take edges in A to compute the canon of B, recursively.

We start our case analysis by considering properties of the center C of $\mathsf{T}(B)$.

- **The center C of $\mathsf{T}(B)$ is a separating pair:** We choose this separating pair as the root of $\mathsf{T}(B)$. Thus, we have only one choice for the root, and the subtree rooted at B can be canonized in a unique way.

- **C is a triconnected component and a is not associated with C:** Let a be associated with a triconnected component R. We find the path from R to C in $\mathsf{T}(B)$ and find the separating pair closest to C on this path. This serves as the unique choice for the root of $\mathsf{T}(B)$.

- **a is associated with C and C is a cycle:** We canonize C for the two edges incident on a as starting edges, and a as the starting vertex. We construct these canons till a virtual edge is encountered in one or both of them. We choose the separating pairs corresponding to the first virtual edges encountered in these canons as the roots of $\mathsf{T}(B)$. Thus, we get at most two choices for the root of $\mathsf{T}(B)$.

For the following cases, we assume that the center C is a 3-connected component and a is associated with C. We proceed with the case analysis according to the number l of articulation points in B besides a.

Case I: $l = 0$. B is a leaf node in S_a, it contains no articulation points besides a. We color a with a distinct color. In this case we can cycle through all separating pairs as root for $\mathsf{T}(B)$.

Case II: $l = 1$. If B has exactly one articulation point besides a, then we process this child a priori and store the result. We color a and a_1 with distinct colors and proceed with B as in case of a leaf node.

Case III: $l \geq 2$. We distinguish several sub-cases.

1. **Some articulation point a_j in E_1 is not associated with C.** Let a_j be associated with a triconnected component $D \neq C$. Find the path from D to C in $\mathsf{T}(B)$ and select the separating pair node closest to C on this path. Thus a_j uniquely defines a separating pair. In the worst case, this may happen for every a_j in E_1. Therefore, we get up to k_1 separating pairs as candidates for the root.

2. **All articulation points in E_1 are associated with C.** We distinguish sub-cases according to the size of E_1.

   (a) If $k_1 \geq 2$, then by Lemma 5.3 below, C can have at most $4k_1$ automorphisms. Thus, we have at most $4k_1$ ways of choosing the root of $\mathsf{T}(B)$.

   (b) If $k_1 = 1$, then we consider the next larger class of subtrees, E_2. We handle the cases for E_2 exactly as for E_1. However, we do not need to proceed to E_3, because we can handle the case $k_1 = k_2 = 1$ directly.

      i. **Some articulation point a_j in E_2 is not associated with C.** We do the same as in sub-Case 1 with a_j in E_2. We get up to k_2 separating pairs as candidates for the root.

      ii. **All articulation points in E_2 are associated with C.**
      If $k_2 \geq 2$, then we process the child in E_1 a priori and store the result. By Lemma 5.3 below, C can have at most $4k_2$ automorphisms in this case. Thus, we have at most $4k_2$ ways of choosing the root of $\mathsf{T}(B)$.

      If $k_2 = 1$, then C has at least three vertices that are fixed by all its automorphisms (i.e. a and the articulation point with its subtree in E_1 and that in E_2). We will show in Corollary 5.7 below that C has at most one non-trivial automorphism in this case. Thus, we have at most two ways of choosing the root of $\mathsf{T}(B)$.

Let $N = |S_B|$. The subtrees in the size class E_m have size $\leq N/k_m$. Since the size classes are ordered according to increasing k_j's, the subtrees in E_j also have size $\leq N/k_m$ for all $j \geq m$. We may assume that all subtrees are of size $\leq N/2$ because otherwise such a subtree is considered as *large* and treated as a speacial case by the algorithm (see page 27).

In all of the cases, the number of ways for choosing the root is constant, or bounded by k_1 or k_2, but the latter only if $k_1 = 1$. Then we use constant space or $O(\log k_1)$, (respectively $O(\log k_2)$) space to keep track of which of the potential root edges is currently being used, and all subtrees are of size $\leq N/k_1$, (respectively size $\leq N/k_2$). This will suffice to bound the total space used for the subtree rooted at B by $O(\log N)$.

The following lemma gives a relation between the size of the smallest color class and the number of automorphisms for a 3-connected planar graph with one distinctly colored vertex.

Lemma 5.3 *Let G be a 3-connected planar graph with colors on its vertices such that one vertex a is colored distinctly, and let $k \geq 2$ be the size of the smallest color class apart from the one which contains a. G has $\leq 4k$ automorphisms.*

To prove Lemma 5.3, we refer to the following results.

Lemma 5.4 [Bab95](P. Mani) *Every triconnected planar graph G can be embedded on the 2-sphere as a convex polytope P such that the automorphism group of G coincides with the automorphism group of the convex polytope P formed by the embedding.*

Lemma 5.5 [AD04, Bab95, Art96] *For any convex polytope other than tetrahedron, octahedron, cube, icosahedron, dodecahedron, the automorphism group is the product of its rotation group and $(1, \tau)$, where τ is a reflection. The rotation group is either C_k or D_k, where C_k is the cyclic group of order k and D_k is the dihedral group of order $2k$.*

Proof of Lemma 5.3. Let H be the subgroup of the rotation group, which permutes the vertices of the smallest color class among themselves. Then H is cyclic since the rotation group is cyclic. Let H be generated by a permutation π.

Notice that a non-trivial rotation of the sphere fixes exactly two points of the sphere viz. the end-points of the axis of rotation. Then, the following claim holds.

Claim 5.6 *In the cycle decomposition of π each non-trivial cycle has the same length.*

Proof of Claim 5.6. Suppose π_1, π_2 are two non-trivial cycles of lengths $p_1 < p_2$ respectively in the cycle decomposition of π. Then π^{p_1} fixes all elements of π_1 but not all elements of π_2. Thus $\pi^{p_1} \in H$ cannot be a rotation of the sphere which contradicts the definition of H. □

As a consequence, the order of H is bounded by k, since the length of any cycle containing one of the k colored points is at most k. □

This leads to the following corollary, which justifies sub-Case (2b) of case III.

Corollary 5.7 *Let G be a 3-connected planar graph with at least 3 colored vertices, each having a distinct color. Then G has at most one non-trivial automorphism.*

Proof. An automorphism of G has to fix all the colored vertices. Consider the embedding of G on a 2-sphere. The only possible symmetry is a reflection about the plane containing the colored vertices, which leads to exactly one non-trivial automorphism. □

Note that if the triconnected component C is one of the exceptions stated in Lemma 5.5, it implies that C has $O(1)$ size. Thus, we do not have to limit its number of possible minimum canons. The preceding discussion implies that if two biconnected component trees are equal for the isomorphism order for some choice of the root, then the corresponding graphs are isomorphic. The reverse direction clearly holds as well.

Theorem 5.8 *Given two connected planar graphs G and H, and their biconnected component trees S and T, then $G \cong H$ if and only if there is a choice of articulation points a, a' in G and H such that $S_a =_{\mathtt{B}} T_{a'}$.*

Proof. Assume that $S_a =_{\mathtt{B}} T_{a'}$. The argument is an induction on the depth of the trees that follows the inductive definition of the isomorphism order. The induction goes from depth d to $d+2$. If the grandchildren of articulation points, say s and t, are $=_{\mathtt{B}}$-equal up to step 3, then we compare the children of s and t. If they are equal, we can extend the $=_{\mathtt{B}}$-equality to the articulation points s and t.

When subtrees are rooted at articulation point nodes, the comparison describes an order on the subgraphs which correspond to split components of the articulation points. The order describes an isomorphism among the split components.

When subtrees are rooted at biconnected component nodes, say B_i and B'_j, the comparison states equality if the components have the same canon, i.e. are isomorphic (cf. Theorem 4.2) and by induction hypothesis we know that the children rooted at articulation points of B_i and B'_j are isomorphic. The equality in the comparisons inductively describes an isomorphism between the vertices in the children of the root nodes.

Hence, the isomorphism between the children at any level can be extended to an isomorphism between the corresponding subgraphs in G and H and therefore to G and H itself.

The reverse direction holds obviously as well. Namely, if G and H are isomorphic and there is an isomorphism between G and H that maps the articulation point a of G to the articulation point a' of H, then the biconnected component trees S_a of G and $T_{a'}$ of H rooted respectively at a and a' will clearly be equal. Hence, such an isomorphism maps articulation points of G to articulation points of H. This isomorphism describes a permutation of the split components of the articulation points. By induction hypothesis, the children at depth $d+2$ of two such biconnected components are isomorphic and equal according to $=_{\mathsf{B}}$. More formally, one can argue inductively on the depth of S_a and $T_{a'}$. □

The space analysis of the isomorphism order algorithm is similar to that of Lindell's algorithm. We highlight the differences needed in the analysis first.

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 the triconnected component tree and r is the index of the edge in the canon, which is to be retrieved. If there are k choices for the root for the triconnected component trees of B and B', the base machine cycles through all of them one by one, keeping track of the minimum canon. This takes $O(\log k)$ space. From the discussion above, we know that the possible choices for the root can be restricted to $O(k)$, and that the subtrees rooted at the children of B have size $\leq |S_B|/k$, when $k \geq 2$. Hence the comparison of B and B' can be done in log-space in this case.

We compare the triconnected component trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$ according to B and B'. When we compare triconnected components in $\mathsf{T}(B)$ and $\mathsf{T}(B')$ then the algorithm asks oracle queries to the triconnected planar graph canonization algorithm. The base machine retrieves edges in these canons one by one from the oracle and compares them. Two edges (a, b) and (a', b') are compared by first comparing a and a'. If both are articulation points, we check whether we reach them for the first time. In this case, we compare the biconnected subtrees S_a and $S_{a'}$ rooted at a and a'. If these are equal then we look, whether (a, b) and (a', b') are separating pairs. If so, then we compare their triconnected subtrees. If these are equal then we proceed with the next edge, e.g. (b, c), and continue in the same way.

We now describe in detail, how to find out whether articulation points a and a' occur for the first time in our traversal, and what is stored on the work-tape when we go into recursion.

Limiting the number of recursive calls for articulation points. When we compare the triconnected component trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$, respectively (see Figure 5), then we might find several copies of articulation points a and a'. That is, a may occur in several components in $\mathsf{T}(B)$, because a can be part of a separating pair. We want to go into recursion on a to the subtree S_a only once. This will be either directly when we reach $\mathsf{T}(B)$, in the case that S_a is a large child of B, or at a uniquely defined point in $\mathsf{T}(B)$. The first case will be described in detail below on page 27. Otherwise we will define a unique component node A of $\mathsf{T}(B)$ that contains a, and we go into recursion on a only in this component. Note that a can occur several times in the canon of the triconnected component A, once for every edge connected to a. We go into recursion at the first edge where a occurs, when we examine A. We call this occurance of a the *reference copy of* a, and similar for a' in A' which is a node in $\mathsf{T}(B')$. Note that the reference copy of a depends on the chosen root for $\mathsf{T}(B)$. We will show that the position of the reference copy (i.e. the component A and the position in the canon for A) can be found again after recursion without storing any extra information on the work-tape.

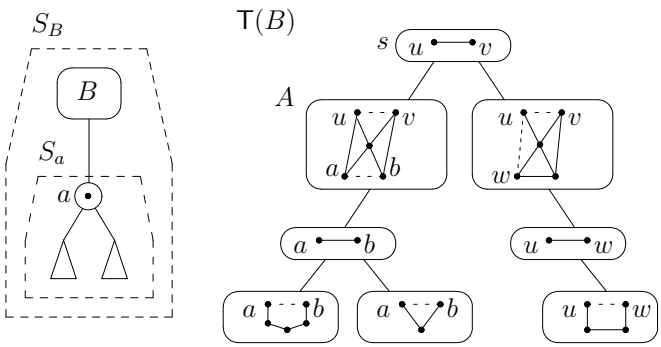

Figure 5: *A biconnected component tree S_B rooted at biconnected component B which has an articulation point a as child, which occurs in the triconnected component tree $\mathsf{T}(B)$ of B. In A and the other triconnected components the dashed edges are separating pairs.*

Lemma 5.9 *The reference copy of an articulation point a in $\mathsf{T}(B)$ and a' in $\mathsf{T}(B')$ for the comparison of triconnected component trees $\mathsf{T}(B)$ with $\mathsf{T}(B')$ can be found in log-space.*

Proof. To prove the lemma, we distinguish three cases for a in $\mathsf{T}(B)$. Assume, that we have the same situation for a' in $\mathsf{T}(B')$. If not, then we found an inequality. We define now a unique component A, where a is contained. We distinguish the following cases.

- Articulation point a occurs in the root separating pair of $\mathsf{T}(B)$. That is, a occurs already at the beginning of the comparisons for $\mathsf{T}(B)$. Then we define A as the root separating pair.
- Articulation point a occurs in separating pairs other than the root of $\mathsf{T}(B)$. Then a occurs in all the component nodes, which contain such a separating pair. By Lemma 3.7 these nodes form a connected subtree of $\mathsf{T}(B)$. Hence, one of these component nodes is the closest to the root of $\mathsf{T}(B)$. This component is always a triconnected component node. Let A be this component. Note that the comparison first compares a with a' before comparing the biconnected or triconnected subtrees, so we reach these copies first in the comparison.
- Articulation point a does not occur in a separating pair. Then, a occurs in only one triconnected component node in $\mathsf{T}(B)$. Let A be this component.

In all except the first case, we find a in a triconnected component node A first. Let a' be found first in component node A', accordingly. Assume, we start the comparison of A and A'. More precisely, we start to compare the canons C of A and C' of A' bit for bit. We go into recursion if and only if we reach the first edge in the canons which contain a and a'. Note that C can contain more than one edge with endpoint a. On all the other edges in C and C' we do not go again into recursion. It is easy to see, that we can recompute the first occurence of A and A'. □

Comparing two subtrees rooted at separating pairs or triconnected components. We go into recursion at separating pairs and triconnected components in $\mathsf{T}(B)$ and $\mathsf{T}(B')$.

When we reach a reference copy of an articulation point in both trees, then we interrupt the comparison of B with B' and go into recurison as described before, i.e. we compare the corresponding articulation point nodes, the children of B and B'. When we return from recursion, we proceed with the comparison of $\mathsf{T}(B)$ and $\mathsf{T}(B')$.

In this part we concentrate on the comparison of $\mathsf{T}(B)$ and $\mathsf{T}(B')$. We give an overview of what is stored on the work-tape when we go into recursion at separating pairs and triconnected components. Basically, the comparison is similar to that in Section 4.3. We summarize the changes.

- We use the size function according to Definition 5.2. That is, the size of a triconnected subtree rooted at a node C in $\mathsf{T}(B)$ also includes the sizes of the biconnected subtrees rooted at the reference articulation points which appear in the subtree of $\mathsf{T}(B)$ rooted at C.
- For a root separating pair node, we store at most $O(\log k)$ bits on the work-tape, when we have k candidates as root separating pairs for $\mathsf{T}(B)$. Hence, whenever we make recomputations in $\mathsf{T}(B)$, we have to find the root separating pair node first. For this, we compute $\mathsf{T}(B)$ in log-space and with the rules described above, we find the candidate edges in log-space. With the bits on the work-tape, we know which of these candidate edges is the current root separating pair. We proceed as in the case of non-root separating pair nodes described next.
- For a non-root separating pair node and triconnected component nodes, we store the same on the work-tape as described in Section 4.3, i.e. the counters $c_<, c_=, c_>$, orientation counters for separating pair nodes, and the information of the current canon for triconnected component nodes. First, recompute the root separating pair node, then we can determine the parent component node. With the information on the work-tape, we can proceed with the computations as described in Section 4.3.

For the triconnected component trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$, we get the same space-bounds as in the previous section on page 16. That is, for the cross-comparison of the children of separating pair nodes s of $\mathsf{T}(B)$ and t of $\mathsf{T}(B')$ we use $O(\log k_j)$ space when we go into recursion on subtrees of size $\leq N/k_j$, where N is the size of the subtree rooted at s and k_j is the cardinality of the j-th isomorphism class. For each such child (a triconnected component node), we use $O(1)$ bits, when we go into recursion. In the case we have large children (of size $\geq N/2$), we treat them a priori. We will discuss this below.

Comparing two subtrees rooted at articulation points. When we consider the trees S_a and $S_{a'}$ rooted at articulation points a and a' then we have for the cross comparison of their children, say $B_1, \ldots, B_k$ and $B'_1, \ldots, B'_k$ respectively, a similar space analysis as in the case of separating pair nodes. That is, we use $O(\log k_j)$ space when we go into recursion on subtrees of size $\leq N/k_j$, where $N = |S_a|$ and k_j is the cardinality of the j-th isomorphism class. Large children (of size $\geq N/2$) are treated a priori. We will discuss this below.

When we compare biconnected components B_i and B'_i, then we compute $\mathsf{T}(B_i)$ and $\mathsf{T}(B'_i)$. We have a set of separating pairs as candidates for the root of $\mathsf{T}(B_i)$. Recall, that for B_i, its children are partitioned into size classes. Let k_i be the number of elements of the smallest size class with $k_i \geq 2$, there are $O(k_i)$ separating pairs as roots for $\mathsf{T}(B_i)$. Except for the

trivial cases, the algorithm uses $O(\log k_i)$ space when it starts to compare the trees $\mathsf{T}(B_i)$ and $\mathsf{T}(B_i')$.

Assume now that we compare $\mathsf{T}(B_i)$ and $\mathsf{T}(B_i')$. In particular, assume we compare triconnected components A and A' of these trees. We follow the canons of A and A' as described above, until we reach articulation points, say a and a'. First, we recompute whether a and a' already occured in the parent node. If not, then we recompute the canons of A and A' and check, whether a and a' occur for the first time. If so, then we store nothing and go into recursion.

When we return from recursion, we recompute the components A and A' in $\mathsf{T}(B)$ and $\mathsf{T}(B')$. On the work-tape there is information about which are the current and the unerased canons. We run through the current canons and find the first occurence of a and a'.

Large children. As in the case of biconnected graphs in Section 3, we deviate from the algorithm described so far in the case that the recursion would lead to a large child. Large subtrees are again treated a priori.

However, the notion of a large child is somewhat subtle here. We already defined the size of biconnected component trees S_a and S_B with an articulation point a or a biconnected component B as root. A *large child* of such a tree of size N is a child of size $\geq N/2$.

Now consider $\mathsf{T}(B)$, the triconnected component tree of B. Let A be a triconnected component and $\{u, v\}$ be a separating pair in $\mathsf{T}(B)$. We have not yet defined the subtrees S_A and $S_{\{u,v\}}$ rooted at A and $\{u, v\}$, respectively, and this has to be done quite carefully.

We already described above that an articulation point a may occur in several components of a triconnected component tree. We said that we go into recursion to the biconnected component tree S_a only once, namely either when we reach the reference copy of a (as defined on page 24) or even before in the following case: let a be an articulation point in the biconnected component B. Let $\mathsf{T}(B)$ be the trinconnected component tree of B, and let C be the node in $\mathsf{T}(B)$ that contains the reference copy of a. Then it might be the case that S_a is a large child of S_B *and* of S_C. In this case we visit S_a when we reach B, i.e. before we start to compute the root for $\mathsf{T}(B)$. Then, when we reach the reference copy of a in C, we first check whether we already visited S_a. In this case the comparison result (with some large child $S_{a'}$ of B') is already stored on the work-tape and we do not visit S_a a second time. Note that if we would go into recursion at the reference copy a second time then we cannot guarantee the log-space bound of the transducer, because we already have written bits on the work-tape for B when we traverse the child, the biconnected subtree S_a for the second time. Otherwise, we visit S_a at the reference copy of a.

Consequently, we consider S_a as a subtree only at the place where we go into recursion to S_a. Recall, that this is not a static property, because for example the position of the reference copy depends on the chosen root of the tree, and we try several possibilities for the root. Figure 6 shows an example.

Definition 5.10 *Let B be a biconnected component and $\mathsf{T}(B)$ its triconnected component tree. Let C be a node in $\mathsf{T}(B)$, i.e. a triconnected component node or a separating pair node. The* tree S_C rooted at C *consists of the subtree of $\mathsf{T}(B)$ rooted at C (with respect to the root of $\mathsf{T}(B)$) and of the subtrees S_a for all articulation points a that have a reference copy in the subtree of $\mathsf{T}(B)$ rooted at C, with exception of those S_a that are a large child of S_B. The* size of S_C *is the sum of the sizes of its components.*

Let N be the size of S_C. A large child of S_C *is a subtree of the root of S_C of size $\geq N/2$.*

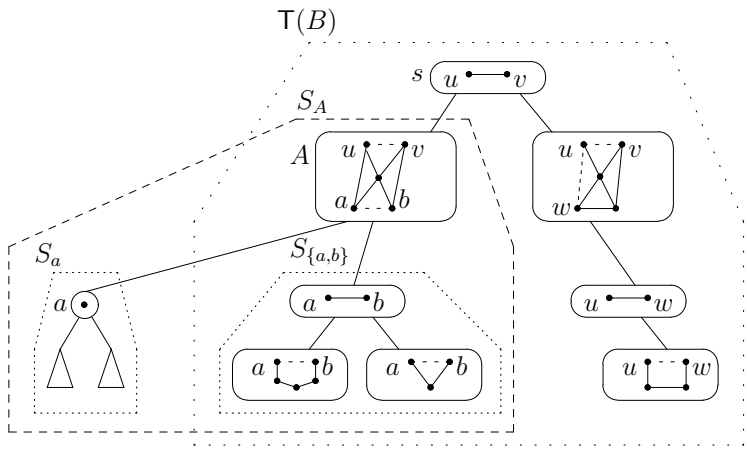

Figure 6: *The triconnected component tree* $\mathsf{T}(B)$ *of the biconnected component* B*. The triconnected component* A *contains the reference copy of articulation point* a*. If* S_a *is not a large child of* B*, then the subtree* S_A *consists of the subtree of* $\mathsf{T}(B)$ *rooted at* A **and** *the subtree* S_a*. In contrast,* S_a *is not part of the subtree* $S_{\{a,b\}}$ *because it does not contain the reference copy of* a*.*

Whenever the algorithm reaches a component a, B or C as above, it first checks whether the corresponding tree S_a, S_B, or S_C has a large child and treats it a priori. The result is stored with $O(1)$ bits. In the case of triconnected components, we also store the orientation. We distinguish large children as follows.

- Large children with respect to the biconnected component tree. These are children of node a in S_a or B in S_B. These children are biconnected component nodes or articulation point nodes. When comparing S_B with $S_{B'}$, then we go for large children into recursion before computing the trees $\mathsf{T}(B)$ and $\mathsf{T}(B')$.
- Large children with respect to the triconnected component tree. These are children of node C in S_C. These children are separating pair nodes, triconnected component nodes or reference copies of articulation point nodes in C.

Analysis of the space requirement. We analyze the comparison algorithm when it compares subtrees rooted at separating pairs and subtrees rooted at articulation points. For the analysis, the recursion goes here from depth d to $d+2$ of the trees. Observe, that large children are handled a priori at any level of the trees. We set up the following recursion equation for the space requirement of our algorithm.

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

where $k_j \geq 2$ (for all j) are the values mentioned above in the corresponding cases. Hence, $\mathcal{S}(N) = O(\log N)$.

For the explanation of the recursion equation it is helpful to imagine that we have two work-tapes. We use the first work-tape when we go into recursion at articulation point nodes, and the second work-tape when we go into recursion at separating pair nodes. The total space needed is the sum of the space of the two work-tapes.

- At an articulation point node, the value k_j is the number of elements in the j-th size class among the children $B_1, \ldots, B_k$ of the articulation point node. We store $O(\log k_j)$ bits and recursively consider subtrees of size $\leq N/k_j$.
- At a separating pair node the value k_j is the number of elements in the j-th isomorphism class among the children $G_1, \ldots, G_k$ of the separating pair node. We store $O(\log k_j)$ bits and recursively consider subtrees of size $\leq N/k_j$.

This finishes the complexity analysis. We get the following theorem.

Theorem 5.11 *The isomorphism order between two planar graphs can be computed in log-space.*

5.2 The Canon of a Planar Graph

From Theorem 5.11, we know that the isomorphism order of biconnected component trees can be computed in log-space. Using this algorithm, we show that the canon of a planar graph can be output in log-space.

The canonization of planar graphs proceeds exactly as in the case of biconnected planar graphs. A log-space procedure traverses the biconnected component tree and makes oracle queries to the isomorphism order algorithm and outputs a canonical list of edges, along with delimiters to separate the lists for siblings.

For an example, consider the canonical list $l(S, a)$ of edges for the tree S_a of Figure 4. Let $l(B_i, a)$ be the canonical list edges of the biconnected component B_i (i.e. the canonical list of $\mathsf{T}(B_i)$ with a the parent articulation point). Let $a_1, \ldots, a_{l_1}$ be the order of the reference copies of articulation points as they occur in the canon of $\mathsf{T}(B_i)$. Then we get the following canonical list for S_a.

$$\begin{aligned}
l(S, a) &= [\ (a)\ l(S_{B_1}, a)\ \ldots\ l(S_{B_k}, a)\ ],\ \text{where} \\
l(S_{B_1}, a) &= [\ l(B_1, a)\ l(S_{a_1}, a_1)\ \ldots\ l(S_{a_{l_1}}, a_{l_1})\ ] \\
&\vdots \\
l(S_{B_k}, a) &= [\ l(B_k, a)\ l(S_{a_{l_k}}, a_{l_k})\ ]
\end{aligned}$$

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. We get

Theorem 5.12 *A planar graph can be canonized in log-space.*

Conclusion. In this paper, we improve the known upper bound for isomorphism and canonization of planar graphs from AC^1 to L. This implies L-completeness for this problem, thereby settling its complexity. An interesting question is to extend it to other important classes of graphs.

Acknowledgment. We thank V. Arvind, Bireswar Das, Raghav Kulkarni, Meena Mahajan, Jacobo Torán and the anonymous referees for helpful comments and discussions.

References

[AD04] V. Arvind and Nikhil Devanur. Symmetry breaking in trees and planar graphs by vertex coloring. In *The Nordic Combinatorial Conference (NORCOM)*, 2004.

[ADK08] V. Arvind, Bireswar Das, and Johannes Köbler. A logspace algorithm for partial 2-tree canonization. In *Computer Science Symposium in Russia (CSR)*, pages 40–51, 2008.

[ADR05] Eric Allender, Samir Datta, and Sambuddha Roy. The directed planar reachability problem. In *Proceedings of the 25th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 238–249, 2005.

[AK06] V. Arvind and Piyush P. Kurur. Graph isomorphism is in spp. *Information and Computation*, 204(5):835–852, 2006.

[AM00] Eric Allender and Meena Mahajan. The complexity of planarity testing. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 87–98, 2000.

[Art96] Michael Artin. Algebra. *Prentice Hall*, 1996.

[Bab95] László Babai. Automorphism groups, isomorphism, reconstruction. *Handbook of combinatorics*, 2:1447–1540, 1995.

[BHZ87] Ravi B. Boppana, Johan Hastad, and Stathis Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[BL83] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM Symposium on Theory of Computing (STOC)*, pages 171–183, 1983.

[BTV07] Chris Bourke, Raghunath Tewari, and N.V. Vinodchandran. Directed planar reachability is in unambiguous log-space. In *IEEE Conference on Computational Complexity (CCC)*, pages 217–221, 2007.

[Bus97] Samuel R. Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *Proceedings of the 5th Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 18–33, 1997.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–22, 1985.

[DLN08] Samir Datta, Nutan Limaye, and Prajakta Nimbhorkar. 3-connected planar graph isomorphism is in log-space. In *Proceedings of the 28th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 153–162, 2008.

[HT73] John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[HT74] John E. Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[HW74] John E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th annual ACM Symposium on Theory of Computing (STOC)*, pages 172–184, 1974.

[JKMT03] Birgit Jenner, Johannes Köbler, Pierre McKenzie, and Jacobo Torán. Completeness results for graph isomorphism. *Journal of Computing and System Sciences*, 66(3):549–566, 2003.

[KHC04] Jacek P. Kukluk, Lawrence B. Holder, and Diane J. Cook. Algorithm and experiments in testing planar graphs for isomorphism. *Journal of Graph Algorithms and Applications*, 8(2):313–356, 2004.

[KST93] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem*. Birkhäuser, 1993.

[Lin92] Steven Lindell. A logspace algorithm for tree canonization (extended abstract). In *Proceedings of the 24th annual ACM Symposium on Theory of Computing (STOC)*, pages 400–404, 1992.

[Mac37] Saunders Maclane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3:460–472, 1937.

[MJT98] Pierre McKenzie, Birgit Jenner, and Jacobo Torán. A note on the hardness of tree isomorphism. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity (CCC)*. IEEE Computer Society, 1998.

[MR91] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[RA97] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. In *IEEE Symposium on Foundations of Computer Science*, pages 244–253, 1997.

[Rei05] Omer Reingold. Undirected st-connectivity in log-space. In *Proc. 37th annual ACM Symposium on Theory of Computing (STOC)*, pages 376–385, 2005.

[RR94] Vijaya Ramachandran and John H. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49:517–561, 1994.

[Sch88] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal on Computing and System Sciences*, 37(3):312–323, 1988.

[Tor04] Jacobo Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing*, 33(5):1093–1108, 2004.

[TW08] Thomas Thierauf and Fabian Wagner. The isomorphism problem for planar 3-connected graphs is in unambiguous logspace. In *25th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 633–644, 2008.

[TW09] Thomas Thierauf and Fabian Wagner. Reachability in $k_{3,3}$-free graphs and k_5-free graphs is in unambiguous log-space. Technical Report TR09-029, Electronic Colloquium on Computational Complexity (ECCC), 2009.

[Ver07] Oleg Verbitsky. Planar graphs: Logical complexity and parallel isomorphism tests. In *24th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 682–693, 2007.

[Wag07] Fabian Wagner. Hardness results for tournament isomorphism and automorphism. In *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 572–583, 2007.

[Wei66] Louis Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circuit Theory*, 13:142–148, 1966.

[Whi33] Hassler Whitney. A set of topological invariants for graphs. *American Journal of Mathematics*, 55:235–321, 1933.