Planar Graph Isomorphism Is in Log-Space

SAMIR DATTA, Chennai Mathematical Institute, India
NUTAN LIMAYE, IT University of Copenhagen, Denmark
PRAJAKTA NIMBHORKAR, Chennai Mathematical Institute, India
THOMAS THIERAUF, Aalen University, Germany
FABIAN WAGNER, Ulm University, Germany

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. The gap between the known upper and lower bounds continues to be very significant for many subclasses of graphs as well.

We bridge the gap for a natural and important class of graphs, namely, planar graphs, by presenting a log-space upper bound that matches the known log-space hardness. In fact, we show a stronger result that planar graph *canonization* is in log-space.

CCS Concepts: • **Theory of computation → Problems, reductions and completeness;**

Additional Key Words and Phrases: Computational complexity, log-space, planar graph isomorphism

ACM Reference format:
Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. 2022. Planar Graph Isomorphism Is in Log-Space. *ACM Trans. Comput. Theory* 14, 2, Article 8 (September 2022), 33 pages.
https://doi.org/10.1145/3543686

1 INTRODUCTION

The graph isomorphism problem, GI, is to decide whether there is a bijection between the vertices of two graphs that 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. It is also in the, intuitively weak, counting class SPP [5]. This is the current frontier of our knowledge with respect to upper bounds.

Not much is known with respect to lower bounds. GI is unlikely to be NP-hard, because otherwise, the polynomial-time hierarchy collapses to its second level. This result was proved in the context of interactive proofs in a series of papers [6, 11, 22, 23]. Note that it is not even known whether GI is P-hard. The best we know is that GI is hard for DET [46], the class of problems NC^1-reducible to the determinant, defined by Cook [14].

This research has been supported by DFG grants TO 200/2-2, Th 472/4, and TH 472/5-1.
Authors' addresses: T. Thierauf, Aalen University, Germany; email: thomas.thierauf@uni-ulm.de; S. Datta and P. Nimbhorkar, Chennai Mathematical Institute, India; emails: {sdatta, prajakta}@cmi.ac.in; N. Limaye, IT University of Copenhagen, Denmark; email: nuli@itu.dk; F. Wagner, Ulm University, Germany; email: wag.fab.cs@gmail.com.

1942-3454/2022/09-ART8 $15.00
https://doi.org/10.1145/3543686

Known results: 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. We mention some of the known results.

- *Tournament graphs* are an example of directed graphs where the DET lower bound is preserved [48], while there is a quasi-polynomial time upper bound [9].
- Lindell [36] showed that *tree isomorphism* can be solved in log-space. It is also hard for log-space [29]. Hence, lower and upper bounds match in this case.
- For *interval graphs*, the isomorphism problem is in log-space [31].
- For graphs of *bounded treewidth*, Bodlaender [10] showed that the isomorphism problem can be solved in polynomial time. Grohe and Verbitsky [24] improved the bound to TC^1, and Das, Tóran, and Wagner [16] to LogCFL. Finally, Elberfeld and Schweitzer [20] showed that it is in log-space, where it is complete.

In this article, we consider *planar graph isomorphism*. Weinberg [49] presented an $O(n^2)$ algorithm for testing isomorphism of 3-connected planar graphs. Hopcroft and Tarjan [26] extended this to general planar graphs, improving the time complexity to $O(n \log n)$. Hopcroft and Wong [28] further improved this to linear time. Kukluk, Holder, and Cook [34] 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 [39]. They showed that it is in AC^3. Then, Gazit and Reif [21] improved the upper bound to AC^1; see also Reference [47].

In the context of 3-connected planar graph isomorphism, Thierauf and Wagner [44] presented a new upper bound of $\mathsf{UL} \cap \mathsf{coUL}$, making use of the machinery developed for the reachability problem [43] and specifically for planar reachability [1, 12]. They also show that the problem is L-hard under AC^0-reductions.

There have been several more recent results. The most notable one is a quasi-polynomial time algorithm for isomorphism of all graphs by Babai [8]. Elberfeld and Kawarbayashi [19] extended our result from planar graphs to bounded-genus graphs. An interesting result for planar graphs is by Kiefer et al. [30], where they show that the Weisfeiler-Leman dimension of planar graphs is at most 3. The log-space isomorphism test for interval graphs has been extended to Helly circular-arc graphs by Köbler et al. [32]; another extension in this direction is due to Chandoo [13].

Our results: In the current work, we show that planar graph isomorphism is in log-space. This improves and extends the result in Reference [44]. As it is known that planar graph isomorphism is hard for log-space, our result implies that planar graph isomorphism is log-space complete. Hence, we finally settle the complexity of the problem in terms of complexity classes. In fact, we show a stronger result: We give a log-space algorithm for the *planar graph canonization problem*. That is, we present a function f computable in log-space, which maps all planar graphs from an isomorphism class to one member of the class. Thereby, we also solve the *canonical labeling problem* in log-space, where one has to compute an isomorphism between a planar graph G and its canon $f(G)$.

Proof outline: Let G be the given connected planar graph we want to canonize. As a high-level description of our algorithm, we follow Hopcroft and Tarjan [26] and decompose the graph G. The differences come with the log-space implementation of the various steps.

In more detail, we start by computing the biconnected components of G from which we get the *biconnected component tree* of G. Then, we refine each biconnected component into triconnected components and compute the *triconnected component tree*. The actual coding to get a canon for G starts with the 3-connected components. Our algorithm uses the notion of universal exploration

sequences from References [33] and [42]. Then, we work our way up to the triconnected and biconnected component trees and finally get a canonization of G. Thereby, we adapt Lindell's algorithm for tree canonization. However, we have to make significant modifications to the algorithm. In more detail, our algorithm consists of the following steps on input of a connected planar graph G. All steps can be accomplished in log-space:

(1) Decompose G into its biconnected components and construct its *biconnected component tree* ([3], cf. Reference [45]).
(2) Decompose the biconnected planar components into their triconnected components and construct the *triconnected component trees* (Section 4.1).
(3) Solve the isomorphism problem for the triconnected planar components (Section 3). In fact, we give a canonization for these graphs.
(4) Compute a canonization of biconnected planar graphs by using their triconnected component trees and the results from the previous step (Section 4).
(5) Compute a canon for G by using the biconnected component tree and the results from the previous step (Section 5).

In the last two steps, we adapt Lindell's algorithm [36] for tree canonization.

Note that, without loss of generality, we can assume that the given graph G is connected [42]. If a given graph, say H, is not connected, then we compute its connected components in log-space and canonize each of these components with the above algorithm. Then, we put the canons of the connected components of H in lexicographically increasing order. This obviously gives a canon for H.

The article is organized as follows: After some preliminaries in Section 2, we start to explain the canonization of 3-connected graphs in Section 3. In Sections 4 and 5, we push this up to biconnected and connected graphs, respectively.

Subsequent work: The log-space bound presented here has been extended afterwards to the class of of $K_{3,3}$-minor free graphs and the class of K_5-minor free graphs [18]. The previous known upper bound for these classes was polynomial time [41].

2 DEFINITIONS AND NOTATION

Space bounded Turing machines and related complexity classes. A log-space bounded Turing machine is a deterministic Turing machine with a read-only input tape and a separate work tape. On inputs of length n, the machine may use $O(\log n)$ cells of the work tape. By L, we denote the class of languages decidable by log-space bounded Turing machines. NL is the class of languages computable by nondeterministic log-space bounded Turing machines. UL is the subclass of NL where the nondeterministic Turing machines have to be unambiguous, i.e., there exists at most one accepting computation path.

We also use log-space bounded Turing machines to compute functions. Then, the machine additionally has a write-only output tape. The output tape is *not* counted for the space used by the machine. That is, the function computed by a log-space bounded Turing machine can be polynomially long.

An important property of log-space computable functions is that they are closed under composition. That is, given two functions $f, g : \Sigma^* \to \Sigma^*$, where Σ is an input alphabet, if $f, g \in \mathsf{L}$, then $f \circ g$ is also in L (see Reference [35]). Our isomorphism algorithm will compose constantly many log-space functions as a subroutine. Hence, the overall algorithm will thereby stay in log-space.

Lexicographic order and rank. Let A be a set with a total order $<$. Then, we extend $<$ to tuples of elements of A in a lexicographic manner. That is, for $a_1, \ldots, a_k, b_1, \ldots, b_k \in A$, we write

$(a_1, \ldots, a_k) < (b_1, \ldots, b_k)$ if there is an $i \in \{1, \ldots, k\}$ such that $a_j = b_j$ for $j = 1, \ldots, i-1$, and $a_i < b_i$.

For a list $L = (x_1, x_2, \ldots, x_n)$ of elements, the *rank of* x_i is i, the position of x_i in L.

Graphs. We assume some familiarity with commonly used graph theoretic notions and standard graph theoretic arguments; see for example Reference [50]. Here, we define the notions that are crucial for this article. We will assume that all the graphs are undirected unless stated otherwise. A graph is *regular* if all vertices have the same degree. For degree d, we also say that G is *d-regular*.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be *isomorphic*, $G_1 \cong G_2$ for short, if there is a bijection $\phi : V_1 \to V_2$ such that for all edges $(u, v) \in E_1$

$$(u, v) \in E_1 \iff (\phi(u), \phi(v)) \in E_2.$$

***Graph isomorphism* (GI)** is the problem of deciding whether two given graphs are isomorphic.

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, we say that f computes a *complete invariant* for $\mathcal{G}$. In case that $f(G)$ is itself a graph such that $G \cong f(G)$, then we call f a *canonization of* $\mathcal{G}$, and $f(G)$ the *canon of* G.

A graph G is called *planar* if it can be drawn in the plane in such a way that no edges cross each other, except at their endpoints. Such a drawing of G is called a *planar embedding*. A planar embedding of G divides the plane into regions. Each such region is called a *face*. For a more rigorous definition, see for example Reference [40].

For $U \subseteq V$, let $G(U)$ be the *induced subgraph* of G on U. A graph $G = (V, E)$ is *connected* if there is a path between any two vertices in G.

Let $S \subseteq V$ with $|S| = k$. We call S a *k-separating set* if $G(V - S)$ is not connected. For $u, v \in V$, we say that S *separates u from v in G* if $u \in S$, $v \in S$, or u and v are in different components of $G - S$. A k-separating set is called *articulation point* (or *cut vertex*) for $k = 1$, *separating pair* for $k = 2$. A graph G on more than two vertices is *k-connected* if it contains no $(k-1)$-separating set. Hence, a 1-connected graph is simply a connected graph. A 2-connected graph is also called *biconnected*. Note, however, that *triconnected* will *not* be used as a synonym for 3-connected. Due to the outcome of the graph decomposition algorithm, a *triconnected* graph will be either a 3-connected graph, a cycle, or a 3-bond. A 3-*bond* is a multi-graph with two vertices that are connected by three edges.

Let S be a k-separating set in a k-connected graph G. Let G' be a connected component in $G(V - S)$. A *split graph* or a *split component of* S in G is the induced subgraph of G on vertices $V(G') \cup S$, where we add *virtual edges* between all pairs of vertices in S. Note that the vertices of a separating set S can occur in several split graphs of G.

A crucial ingredient in many log-space graph algorithms is the reachability algorithm by Reingold [42].

Theorem 2.1 ([42]). *Undirected s-t-Connectivity is in* L.

Below, we give some graph theoretic problems for which a log-space upper bound is known due to Theorem 2.1.

(1) *Graph connectivity.* Given a graph G, one has to decide whether G is connected. In the enumeration version of the problem, one has to compute all the connected components of G.

To decide whether G is connected, cycle through all pairs of vertices of G and check reachability for each pair. To compute the connected component of a vertex v, cycle through all the vertices of G and output the reachable ones. Clearly, this can be implemented in log-space with the reachability test as a subroutine.

(2) *Separating set.* Given a graph $G = (V, E)$ and a set $S \subseteq V$, one has to decide whether S is a separating set in G. In the enumeration version of the problem, one has to compute all the separating sets of a fixed size k.

Recall that S is a separating set if $G - S$ is not connected. Hence, we have a reduction to the connectivity problem. To solve the enumeration version for a constant k, a log-space machine can cycle through all size k subsets of vertices and output the separating ones. In particular, we can enumerate all articulation points and separating pairs in log-space.

Let $d(u, v)$ be the distance between vertices u and v in G. The *eccentricity* $\varepsilon(v)$ *of* v is the maximum distance of v to any other vertex,

$$\varepsilon(v) = \max_{u \in V} d(v, u).$$

The minimum eccentricity over all the vertices in G is called the *radius of* G. The vertices of G that have the eccentricity equal to the radius of the graph form the *center of* G. In other words, vertices in the center minimize the maximal distance to the other vertices in the graph. For example, if G is a tree of odd diameter, then the center consists of a single node, namely, the midpoint of a longest path in the tree. Moreover, because distances in a tree can be computed in log-space, also the center node of a tree can be computed in log-space.

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

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

A rotation system ρ encodes an embedding of graph G on an orientable surface by describing a circular ordering of the edges around each vertex. If the orientable surface has genus zero, i.e., it is a sphere, then the rotation system is called a *planar rotation system*.

Conversely, a graph embedded on a plane uniquely defines a cyclic order of edges incident on any vertex. The set of all cyclic orders gives a rotation system for the planar graph, which is a planar rotation system by definition. All embeddings that give rise to the same rotation system are said to be equivalent, and their equivalence class is called a *combinatorial embedding*; see for example Reference [40, Section 4.1].

Allender and Mahajan [2] showed that a planar rotation system can be computed in log-space.

Theorem 2.2 ([2]). *Let G be a graph. In log-space, one can check whether G is planar and compute a planar rotation system in this case.*

Let ρ^{-1} be the set of inverse rotations of ρ, i.e., $\rho^{-1} = \{\, \rho_v^{-1} \mid v \in V \,\}$. Note that if ρ is a planar rotation system, then this holds for ρ^{-1} as well. Namely, ρ^{-1} corresponds to the mirror symmetric embedding of G.

It follows from work of Whitney [51] that in the case of planar 3-connected graphs, there exist only two planar rotation systems, namely, some planar rotation system ρ and its inverse ρ^{-1}. This is a crucial property in the isomorphism test of Weinberg [49] and all the other follow-up works. We also use this property in our algorithm for planar 3-connected graphs to obtain a log-space upper bound.

Universal Exploration Sequences (UXS). Let $G = (V, E)$ be a d-regular graph. The edges around any vertex v can be numbered $0, 1, \ldots, d-1$ in an arbitrary, bijective way. A sequence $\tau_1 \tau_2 \cdots \tau_k \in \{0, 1, \ldots, d-1\}^k$ together with a starting edge $e_0 = (v_0, v_1) \in E$ defines a walk $v_0, v_1, \ldots, v_k$ in G as follows: For $1 \leq i \leq k$, if $e_{i-1} = (v_{i-1}, v_i)$ is the sth edge of v_i, then let $e_i = (v_i, v_{i+1})$ be the $(s + \tau_i)th$ edge of v_i modulo d.

A sequence $\tau_1\tau_2\ldots\tau_k \in \{0,1,\ldots d-1\}^k$ is an (n,d)-***universal exploration sequence (UXS)*** *for d-regular graphs of size $\leq n$*, if for every connected d-regular graph on $\leq n$ vertices, any numbering of its edges, and any starting edge, the walk obtained visits all the vertices of the graph.

Universal exploration sequence plays a crucial role in Reingold's result that undirected reachability is in log-space. We use it in our log-space algorithm for testing isomorphism of 3-connected planar graphs.

THEOREM 2.3 ([42]). *There exists a log-space algorithm that takes as input $(1^n, 1^d)$ and produces an (n,d)-universal exploration sequence.*

3 CANONIZATION OF 3-CONNECTED PLANAR GRAPHS

In this section, we give a log-space algorithm for the canonization of 3-connected planar graph. This improves the UL ∩ coUL bound given by Thierauf and Wagner [44] for 3-connected planar graph isomorphism. Since the problem is also L-hard [44] this settles the complexity of the problem in terms of complexity classes.

THEOREM 3.1. *The canonization of 3-connected planar graphs is in log-space.*

The algorithm in Reference [44] constructs a canon for a given 3-connected planar graph. This is done by first computing a spanning tree for the graph. Then, by traversing the spanning tree, the algorithm visits all the edges in a certain order. For the computation of the spanning tree, the algorithm computes distances between vertices of the graph. This is achieved by using the planar reachability test of Bourke, Tewari, and Vinodchandran [12]. All parts of the algorithm work in log-space, except for the planar reachability test, which is in UL ∩ coUL. Therefore, this is the overall complexity bound.

In our approach, we essentially replace the spanning tree in the above algorithm by a universal exploration sequence. Since such a sequence can be computed in log-space by Theorem 2.3, this will put the problem in L.

Note that universal exploration sequences are defined for *regular* graphs. Therefore, our first step is to transform a given graph G into a 3-regular graph in such a way that

- a planar graph stays planar and
- two graphs are isomorphic if and only if they are isomorphic after this preprocessing step.

Note that every vertex has degree ≥ 3 because G is 3-connected. The following standard construction 3-REGULAR-COLOR reduces the degree of vertices to exactly three. For later use, we also 2-color the edges in the resulting graph.

Note that the resulting graph G' is 3-regular and planar, if G is planar. If G has n vertices and m edges, then G' has $2m$ vertices and $3m$ edges.

Moreover, G' is also 3-connected. An easy way to see this is via Steinitz's theorem. It states that planar 3-connected graphs are precisely the skeletons of 3-dimensional convex polyhedra. For G', we replace every vertex of the convex polyhedron for G by a (small enough) cyclic face such that the resulting polyhedral is still convex. Therefore, G' is also planar and 3-connected. It follows that also G' has only two possible embeddings, namely, the ones inherited from G.

In the following lemma, we give an elementary proof where we do not use planarity. For non-planar G, we do not have a planar rotation system according to which we put the new edges. In this case, we use an arbitrary rotation system.

LEMMA 3.2. *Let G be a 3-connected graph and G' be the 3-regular graph computed by algorithm* 3-REGULAR-COLOR(G). *Then, G' is 3-connected.*

ALGORITHM 3.1: 3-Regular-Color(G)

Input: A 3-connected graph $G = (V, E)$.
Output: A colored 3-regular graph $G' = (V', E')$.

1: Replace a vertex $v \in V$ of degree $d_v \geq 3$ by a cycle $(v'_1, \ldots, v'_{d_v})$ on d_v new vertices. This defines vertices V' and part of the edges in E'. We give color 1 to the cycle edges.
2: Fix a rotation ρ_v of the edges around v, for every $v \in V$. In case that G is planar, we use a planar rotation.
3: For every edge $(u, v) \in E$,

- let u be the ith neighbor according to ρ_v of v in G
- let v be the jth neighbor according to ρ_u of u in G

Then, we put the new edge (u'_j, v'_i), which replaces the old edge (u, v). These edges get color 2. This completes the definition of E'.
4: Output the resulting graph $G' = (V', E')$.

Proof. Let u, v be two vertices in G. Since G is 3-connected, there are three vertex-disjoint paths p_1, p_2, p_3 from u to v in G. In G', vertices u, v are replaced by cycles. The paths p_1, p_2, p_3 can be transformed to vertex-disjoint paths p'_1, p'_2, p'_3 in G'. These paths start in vertices u'_1, u'_2, u'_3 from the cycle corresponding to u and end in vertices v'_1, v'_2, v'_3 from the cycle corresponding to v, respectively.

Let u' and v' be vertices from the cycles corresponding to u and v, respectively. We show that there are three vertex-disjoint paths from u' to v' in G'. For this, we want to extend paths p'_1, p'_2, p'_3 to connect u' and v'. We consider u'. The case of v' is similar.

(1) If u' is one of u'_1, u'_2, u'_3, say u'_1, then we can extend p'_2, p'_3 on the cycle to reach u' and stay vertex-disjoint.
(2) If u' is different from u'_1, u'_2, u'_3, then we use the non-cycle edge that stems from G and go to a neighbor w' of u'. Vertex w' is on the cycle corresponding to a vertex w in G. Since G is 3-connected, there is a path p from w to v in G. Again there is a path p' in G' corresponding to p.

    We construct a new path $\widehat{p}$ that starts at u' and goes via w' to the staring point of p'. Then, we follow p' until we intersect the first time with one of p'_1, p'_2, p'_3, say p'_1. Then, $\widehat{p}$ continues on p'_1 until we reach v'_1. When we consider paths $\widehat{p}, p'_2, p'_3$ instead of p'_1, p'_2, p'_3, then we are in case 1.

This shows that vertices u', v' from different cycles are connected by three vertex-disjoint paths in G'. In case that u', v' are on the same cycle corresponding to one vertex of G, we can use two paths from the cycle and one path via some neighbor vertex of u' to v'. □

To maintain the isomorphism property, we have to avoid potential isomorphisms that map new edges from the cycles to the original edges. This is the reason why we also colored the edges. We summarize:

Lemma 3.3. *Given two* 3*-connected planar graphs* G *and* H*, let* G' *and* H' *be the colored* 3*-regular graphs computed by* 3-Regular-Color*. Then* $G \cong H$ *if and only if* $G' \cong H'$*, where the isomorphism between* G' *and* H' *has to respect the colors of the edges.*

Note that the Lemma crucially depends on the unique embedding of the graph.

Before we show how to get a canon for graph G, we compute a complete invariant as an intermediate step. The procedure $\textsc{Code}(G', \rho, u_0, v_0)$ described in Algorithm 3.2 computes a code for G' with respect to a planar rotation system ρ, a starting vertex u_0, and a starting edge (u_0, v_0).

ALGORITHM 3.2: $\textsc{Code}(G', \rho, u_0, v_0)$

Input: A 3-regular graph G' with N vertices and colored edges, a planar rotation system ρ, and vertices u_0 and v_0 such that v_0 is a neighbor of u_0.
Output: A code of G' with respect to ρ, vertex u_0 and edge (u_0, v_0).

1: Construct an $(N, 3)$-universal exploration sequence U.
2: Traverse G' according to U and ρ, starting from u_0 along edge (u_0, v_0). Thereby, we construct a list L of nodes traversed, $L = (u_0, v_0, w_0, \dots)$.
3: Relabel the vertices occurring in L according to their first occurrence in the sequence. Let L' be the resulting list. For example, u_0 and v_0 get label 1 and 2, respectively, and therefore L' starts as $L' = (1, 2, \dots)$.
4: Given L and L', compute the relabeling function π that maps the label of a node in L' to its label in L. For example, $\pi(1) = u_0$ and $\pi(2) = v_0$.
5: Output the $N \times N$ adjacency matrix $A = (a_{i,j})$ of G' with respect to the new node labels. That is, for $i, j \in \{1, \dots, N\}$, let

$$a_{i,j} = \begin{cases} c, & \text{if } (\pi(i), \pi(j)) \text{ is an edge in } G' \text{ of color } c, \\ 0, & \text{otherwise.} \end{cases}$$

The five steps of the algorithm can be seen as the composition of five functions. We argue that each of these functions is in log-space. Then, it follows that the overall algorithm works in log-space. Step 1 is in log-space by Theorem 2.3. In Step 2, we only have to store local information to walk through G'.

Step 3 requires to compute the rank of each vertex in the list L. For a vertex v occurring in L, this amounts to searching in L to the left of the current position for the first occurrence of v. Then, we have to count the number of different vertices in L to the left of the first occurrence of v. This can be done in log-space. A more detailed outline can be found in Reference [44].

In Step 4, we determine the position of node i in L' and the node v_i at the same position in L. Then, $\pi(i) = v_i$. Step 5 is again trivial when one has access to π.

Definition 3.4. The *code* $\sigma_{G'}$ of a 3-regular graph G' is the lexicographic minimum of the outputs of $\textsc{Code}(G', \rho, u_0, v_0)$ for the two choices of a planar rotation system ρ and all choices of $u_0 \in V$ and a neighbor $v_0 \in V$ of u_0.

The following lemma states that the code $\sigma_{G'}$ of G' computed so far is a complete invariant for the class of 3-connected planar graphs.

Lemma 3.5. *Let G' and H' be 3-regular planar graphs and $\sigma_{G'}$ and $\sigma_{H'}$ be the codes of G' and H', respectively. Then*

$$G' \cong H' \iff \sigma_{G'} = \sigma_{H'} .$$

Proof. If $G' \cong H'$, then there is an isomorphism φ from G' to H'. Let $\rho_{G'}$ be the planar rotations system, u_0 a vertex, and (u_0, v_0) the starting edge that lead to the minimum code $\sigma_{G'}$. Let $\rho_{H'}$ be the rotations system of H' induced by $\rho_{G'}$ and φ. Let $\sigma = \textsc{Code}(H', \rho_{H'}, \varphi(u_0), \varphi(v_0))$.

We prove that $\sigma_{G'} = \sigma$: Let w be a vertex that occurs at position ℓ in the list $L_{G'}$ computed in Step 2 in $\textsc{Code}(G', \rho_{H'}, u_0, v_0)$. Then, $\varphi(w)$ will occur at position ℓ in the list $L_{H'}$ computed in Step 2

in Code$(H', \rho_{H'}, \varphi(u_0), \varphi(v_0))$. This is because the oriented graphs are isomorphic, and the same UXS is used for their traversal. Hence, when a vertex w occurs the first time $L_{G'}$, $\varphi(w)$ will occur the first time in $L_{H'}$ at the same position. Moreover, by induction, the number of different vertices to the left of w in $L_{G'}$ will be the same as the number of different vertices to the left of $\varphi(w)$ in $L_{H'}$. Hence, in Step 3 in Code$(G', \rho_{H'}, u_0, v_0)$ vertex w will get the same name, say j, as vertex $\varphi(w)$ in Step 3 in Code$(H', \rho_{H'}, \varphi(u_0), \varphi(v_0))$. Therefore, in Step 4, the relabeling function for G' will map $\pi_{G'}(j) = w$, and the relabeling function for H' will map $\pi_{H'}(j) = \varphi(w)$. So, we will get the same output in Step 5. We conclude that $\sigma_{G'} = \sigma$.

Clearly σ is also the minimum of all the possible codes for H', because otherwise, we could switch the roles of G' and H' in the above argument and would obtain a code for G' smaller than $\sigma_{G'}$. Therefore, we have also $\sigma_{H'} = \sigma$. Hence, $\sigma_{G'} = \sigma_{H'}$.

For the reverse direction, let $\sigma_{G'} = \sigma_{H'} = \sigma$. The labels of vertices in σ are just a relabeling of the vertices of G' and H'. These relabelings are some permutations, say π_1 and π_2. Then, $\pi_2^{-1} \circ \pi_1$ is an isomorphism between G' and H'. □

To prove Theorem 3.1, we show how to construct a canon for G from the code $\sigma_{G'}$ for G'. Recall that algorithm 3-Regular-Color replaces a vertex v of degree d in G by a cycle $(v'_1, \ldots, v'_d)$ in G' and also colors the edges. In the code $\sigma_{G'}$, each node in the cycle gets a new label. We assign to v the minimum label among the new labels of $(v'_1, \ldots, v'_d)$ in G'. To do so, we start at one of the vertices, say v'_1, and traverse color 1 edges until we get back to v'_1. Thereby, we can find out the minimum label. Let $\pi(v)$ be the label assigned to v.

We are not quite done yet. Recall that G' has $2m$ vertices. Hence, the labels $\pi(v)$ we assign to the vertices of G are in the range $\pi(v) \in \{1, 2, \ldots, 2m\}$. But G has n vertices and we want the assignment to map to $\{1, 2, \ldots, n\}$. To do so, we convert π into a mapping π' such that $\pi'(v)$ is the rank of $\pi(v)$ in the ordered π-labeling sequence. Then, we have $\pi'(v) \in \{1, 2, \ldots, n\}$. The construction of π and π' can be done in log-space.

As *canon of* G, we define a coding of the adjacency matrix of G, say σ, where vertices are relabeled according to π'. Then, σ codes a graph that is isomorphic to G by construction. Moreover, for every graph H isomorphic to G, we will get the same code σ for H. This is because the relabeling functions π and π' depend only on the code $\sigma_{G'}$, which is the same for H by Lemma 3.5. Hence, our construction gives a canonization of 3-connected planar graphs. This concludes the proof of Theorem 3.1.

4 CANONIZATION OF BICONNECTED PLANAR GRAPHS

In this section, we present an algorithm that given a planar biconnected graph outputs its canon in log-space.

Theorem 4.1. *The canonization of biconnected planar graphs is in log-space.*

The proof is presented in the following five subsections. In Section 4.1, we first show how to decompose a biconnected planar graph G into its *triconnected components*. From these components, we construct the *triconnected component tree of* G.

In Section 4.2, we give a brief overview of a log-space algorithm for tree canonization, which was developed by Lindell [36]. The core of Lindell's algorithm is to come up with a total order on trees such that two trees are isomorphic if and only if they are equal with respect to this order.

In Section 4.3, we define an *isomorphism order* on the triconnected component trees similar to Lindell's order on trees. The isomorphism order we compute has the property that two biconnected graphs will be isomorphic if and only of they are equal with respect to the isomorphism order. This yields an isomorphism test. We analyze its space complexity in Section 4.4.

Finally, based on the isomorphism order, we develop our canonization procedure in Section 4.5.

4.1 Decomposition of a Biconnected Graph into Triconnected Components

Graph decomposition goes back to Hopcroft and Tarjan [27], who presented a linear-time algorithm to compute such a decomposition, and Cunningham and Edmonds [15]. These algorithms are sequential. With respect to parallel algorithms, Miller and Ramachandran [38] presented a decomposition algorithm on a CRCW-PRAM with $O(\log^2 n)$ parallel time and using a linear number of processors. In this section, we show that a biconnected graph can be decomposed into its triconnected components in log-space.

The algorithm presented below was developed in Reference [18].[1] We present the entire algorithm here for the sake of completeness.

Definition 4.2. Let $G = (V, E)$ be a biconnected graph. A separating pair $\{a, b\}$ is called 3-*connected* if there are three vertex-disjoint paths between a and b in G.

The *triconnected components of* G are the split graphs we obtain from G by splitting G successively along all 3-connected separating pairs, in any order. If a separating pair $\{a, b\}$ is connected by an edge in G, then we also define a 3-*bond* for $\{a, b\}$ as a triconnected component, i.e., a multigraph with two vertices $\{a, b\}$ and three edges between them.

We decompose a biconnected graph only along separating pairs that are connected by at least three disjoint paths. By only splitting a graph along 3-connected separating pairs, we avoid the decompositions of cycles. Therefore, we get three types of triconnected components of a biconnected graph: 3-connected components, cycle components, and 3-bonds.

Definition 4.2 leads to the same triconnected components as in Reference [27]. The decomposition is unique, i.e., independent of the order in which the separating pairs in the definition are considered [37]; see also References [15, 25].

Lemma 4.3. *The 3-connected separating pairs and the triconnected components of a biconnected graph can be computed in log-space.*

Proof. In Section 2, we argued that we can compute all separating pairs of G in log-space. To determine whether a separating pair $\{a, b\}$ is 3-connected, we cycle over all pairs of vertices u, v different from a and b and check whether the removal of u, v keeps a reachable from b. Clearly, this can be accomplished in log-space.

It remains to compute the vertices of a triconnected component. Two vertices $u, v \in V$ belong to the same 3-connected component or cycle component if no 3-connected separating pair separates u from v. This property can again be checked by solving several reachability problems. Hence, we can collect the vertices of each such component in log-space. □

The triconnected components of a biconnected graph are the nodes of the *triconnected component tree*.

Definition 4.4. Let G be a biconnected graph. The *triconnected component tree* $\mathcal{T}$ *of* G is the following graph. There is a node for each triconnected component and for each 3-connected separating pair of G. There is an edge in $\mathcal{T}$ between the node for triconnected component C and the node for a separating pair $\{a, b\}$, if a, b belong to C.

Given a triconnected component tree $\mathcal{T}$, we use graph($\mathcal{T}$) to denote the corresponding biconnected graph represented by it.

Note that graph $\mathcal{T}$ is connected, because G is biconnected and acyclic. This also implies that $\mathcal{T}$ is a tree. Each path in $\mathcal{T}$ is an alternating path of separating pairs and triconnected components. All

[1] The first log-space version of this problem appeared in the conference version of the current work [17]. This was subsequently simplified in the work of Reference [18].

the leaves of $\mathcal{T}$ are triconnected components. Hence, a path between two leaves always contains an odd number of nodes and therefore $\mathcal{T}$ has a unique center node.

By Lemma 4.3, we can compute the nodes of the component tree in log-space. We show that we can also traverse the component tree in log-space. Here, by traversal, we mean a way of systematically visiting every vertex of the tree. For example, in classical graph theory, we study many different tree traversals such as preorder, inorder, and postorder traversals. It is known that tree traversal can be performed in log-space. Unlike in a tree, the nodes of the component tress are themselves graphs. We show that, in spite of this, we can perform its traversal in log-space.

Lemma 4.5. *The triconnected component tree of a biconnected graph G can be computed and traversed in log-space.*

Proof. The traversal proceeds as a depth-first search. Assume that a separating pair is fixed as the root node of the component tree, We show how to navigate locally in the component tree, i.e., for a current node how to compute its *parent*, *first child*, and *next sibling*. We explore the tree starting at the root. Thereby, we store the following information on the tape:

- We always store the root node, i.e., the two vertices of the root separating pair.
- When the current node is separating pair $\{a_0, b_0\}$, we just store it.
- When the current node is a 3-connected component C with parent separating pair $\{a_0, b_0\}$, then we store a_0, b_0 and an arbitrary vertex $v \neq a_0, b_0$ from C.

In the last item, the vertex v that we store serves as a representative for C. As a choice for v, take the first vertex of C that is computed by the construction algorithm of Lemma 4.3. Note that v and a_0, b_0 together with the root node identify C uniquely.

The traversal continues by exploring the subtrees at the separating pairs in C, different from $\{a_0, b_0\}$. Let $\{a_1, b_1\}$ be the current separating pair in C. We compute a representative vertex for the first 3-connected split component of $\{a_1, b_1\}$ different from C. Then, we erase $\{a_0, b_0\}$ and the representative vertex for C from the tape and recursively traverse the subtrees at $\{a_1, b_1\}$.

When we return from the subtrees at $\{a_1, b_1\}$, we recompute $\{a_0, b_0\}$ and C, the parent of $\{a_1, b_1\}$. This is done by computing the path from the root node to C in the component tree. That is, we start at the root node and look for the child component that contains C via reachability queries. Then, we iterate the search until we reach C, where we always store the current parent node.

The tree traversal continues with the next sibling of C in the tree. That is, we compute the next articulation point in C after $\{a_1, b_1\}$ with respect to the order on the separating pairs. Then, we delete $\{a_1, b_1\}$ from the work tape. If C does not have a next sibling, then we return to the parent of C. □

4.2 Overview of Lindell's Algorithm for Tree Canonization

We summarize the crucial ingredients of Lindell [36] log-space algorithm for tree canonization. We will then adapt Lindell's technique to triconnected component trees.

Lindell's algorithm is based on an order relation $\leq$ for rooted trees defined below. The order relation has the property that two trees S and T are isomorphic if and only if they are equal with respect to the order, denoted by $S \equiv T$. 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.

The order $<$ on rooted trees is defined as follows:

Definition 4.6. Let S and T be two trees with root s and t, and let $\#s$ and $\#t$ be the number of children of s and t, respectively. Then, $S < T$ if

(1) $|S| < |T|$, or
(2) $|S| = |T|$ but $\#s < \#t$, 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 $S_1 \leq \cdots \leq S_k$ and $T_1 \leq \cdots \leq T_k$ 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:

- Partition the k children of s in S into *blocks* according to their sizes, i.e., the number of nodes in the subtree rooted at the child. Let $N_1 < N_2 < \cdots < N_\ell$ be the occurring sizes, for some $\ell \leq k$, and let k_i be the number of children in block i, i.e., that have size N_i. It follows that $\sum_i k_i = k$ and $\sum_i k_i N_i = n - 1$.

  Doing the same for t in T, we get corresponding numbers $N'_1 < N'_2 < \cdots N'_{\ell'}$ and $k'_1, k'_2, \ldots, k'_{\ell'}$. If $\ell \neq \ell'$, then we know that the two are not isomorphic. Otherwise, we compare the two block structures as follows:
  - If $N_1 < N'_1$, then $S < T$.
  - If $N_1 > N'_1$, then $S > T$.
  - If $N_1 = N'_1$ and $k_1 > k'_1$, then $S < T$.
  - If $N_1 = N'_1$ and $k_1 > k'_1$, then $S > T$.

  If $N_1 = N'_1$ and $k_1 = k'_1$, then we consider the next blocks similarly. This process is continued until a difference in the block structure is detected or all the children of s and t are exhausted.
- Let the children of s and t have the same block structure. Then, compare the children in each block 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 $S \equiv T$.
  Case 2: $k = 1$**.** Recursively consider the grand-children of s and t.
  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 block of S_j that are, respectively, smaller than, greater than, or equal to S_j. The counters are computed by making pairwise cross-comparisons.

  Note that isomorphic subtrees in corresponding blocks have the same order profile. Therefore, it suffices to check that each such order profile occurs the same number of times in each block in S and T. To perform this check, compare the different order profiles of every block in lexicographic order. The subtrees in the block 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 $S \equiv T$.

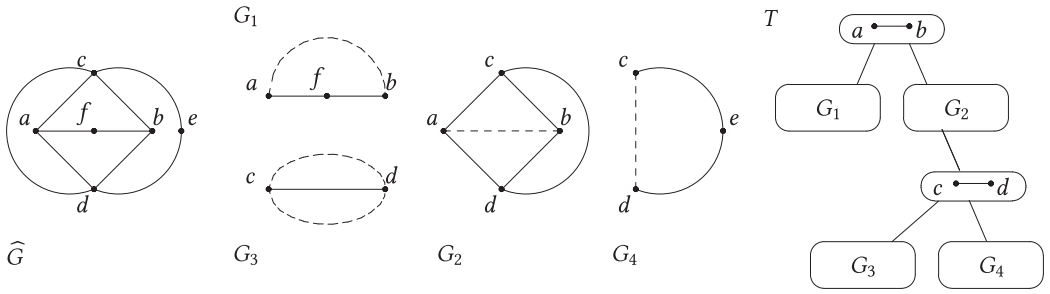

Fig. 4.1. The decomposition of a biconnected planar graph $\widehat{G}$. Its triconnected components are $G_1, \ldots, G_4$ and the corresponding triconnected component tree is T. In $\widehat{G}$, the pairs $\{a, b\}$ and $\{c, d\}$ are 3-connected separating pairs. The inseparable triples are $\{a, b, c\}$, $\{b, c, d\}$, $\{a, c, d\}$, $\{a, b, d\}$, $\{a, b, f\}$, 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.

We analyze the space complexity. Note that in case 2 with just one child, we need no space for the recursive call. In case 3, for each new block, the work-tape allocated for the former computations can be reused. Since $\sum_i k_i N_i \leq n$, the following recursion equation for the space complexity $\mathcal{S}(n)$ holds,

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

where $k_i \geq 2$ for all i. It follows that $\mathcal{S}(n) = O(\log n)$.

Lindell defines the canon of a rooted tree T as the infix coding of the tree over the three-letter alphabet $\{*, [,]\}$, which in turn can be coded over $\{0, 1\}$. The canon of a tree T with just one vertex is $c(T) = *$. The canon of a tree T with subtrees $T_1 \leq T_2 \leq \cdots \leq T_k$ is $c(T) = [c(T_1)c(T_2)\cdots c(T_k)]$.

If we have given a tree T without a specified root, then we try all the vertices of T as the root. The vertex that leads the smallest tree with respect to the order on rooted trees is used as the root to define the canon of T.

4.3 Isomorphism Order of Triconnected Component Trees

In this section, we start with two triconnected component trees and give a log-space test for isomorphism of the biconnected graphs represented by them. Recall from Definition 4.4 that a triconnected component tree T that represents a biconnected graph G consists of nodes corresponding to the triconnected components and 3-connected separating pairs of G. See Figure 4.1 for an example.

The rough idea is to come up with an *order* on the triconnected component trees, as in Lindell's algorithm for isomorphism of trees. Clearly, a major difference to Lindell's setting is that the nodes of the trees are now separating pairs or triconnected components. By using Lindell's algorithm in conjunction with the algorithm from Section 3, we canonize the 3-connected component nodes of the tree. We call this the *isomorphism order*. We ensure that the isomorphism order has the property that two triconnected component trees have the same order if and only if the biconnected graphs represented by them are isomorphic.

To define the order, we also compare the size of the tree. We first define the size of a triconnected component tree.

Definition 4.7. For a triconnected component tree T, the *size of an individual component node* C of T is the number n_C of vertices in C. The *size of the tree* T, denoted by $|T|$, is the sum of the sizes of its component nodes.

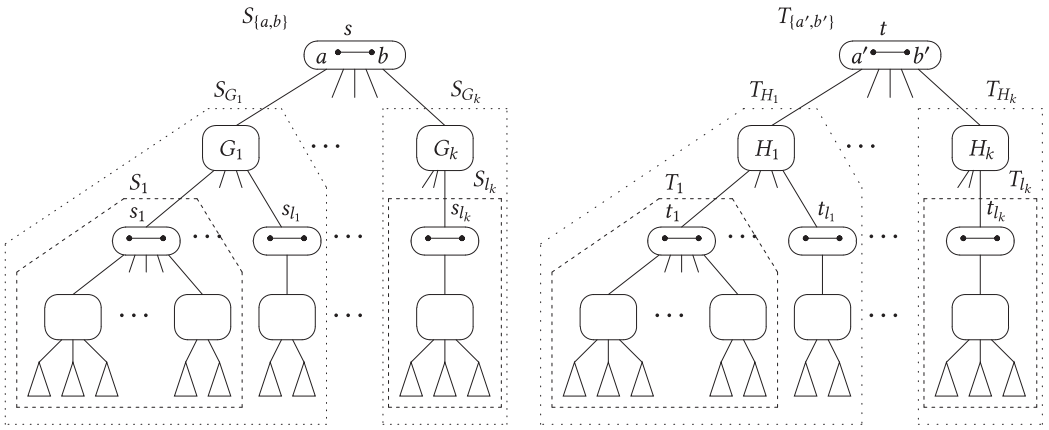

Fig. 4.2. Triconnected component trees.

Note that the vertices of a separating pair are counted in every component where they occur. Therefore, the size of T is at least as large as the number of vertices in graph(T), the graph corresponding to the triconnected component tree T.

We describe a procedure for computing an isomorphism order given two triconnected component trees S and T of 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. As Lindell, we define the final order of G and H based on the separating pairs as roots that lead to the smallest trees. 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 4.2 shows two trees to be compared.

We define the isomorphism order $<_{\mathbb{T}}$ for $S_{\{a,b\}}$ and $T_{\{a',b'\}}$ by first comparing their sizes, then the number of children of the root nodes s and t. These two steps are similar to Lindell's algorithm. If we find equality in the first 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. 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 4.3.

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

The sequential algorithm by Hopcroft and Tarjan [27] uses depth-first-search for the decomposition. They also consider the direction in which an edge is traversed by the search. Thereby, the orientation issue is handled implicitly.

In the following two subsections, we give the details of the isomorphism order between two triconnected component trees depending on the type of the root node.

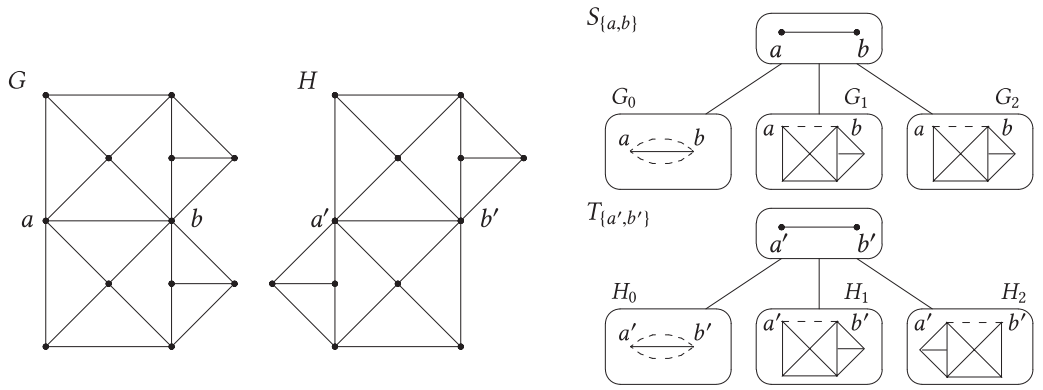

Fig. 4.3. The graphs G and H have the same triconnected component trees but are not isomorphic. In $S_{\{a,b\}}$, the 3-bonds form one isomorphism class I_1 and the other two components form the second isomorphism class I_2, as they all are pairwise-isomorphic. The non-isomorphism is detected by comparing 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 $T_{\{a',b'\}} <_{\mathbb{T}} S_{\{a,b\}}$.

4.3.1 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 first consider the easy cases.

- *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, then we immediately detect an inequality. 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., $S_{G_i} <_{\mathbb{T}} T_{H_j}$ if G_i is a 3-bond and H_j is a cycle.
- *G_i and H_j are 3-bonds.* In this case, S_{G_i} and T_{H_j} are leaves, since they cannot be decomposed further into smaller components, and we define $S_{G_i} \equiv_{\mathbb{T}} T_{H_j}$.

In case where G_i and H_j are cycles or 3-connected components, we construct the canons of G_i and H_j and compare them lexicographically.

- To canonize a cycle, we traverse it starting from the virtual edge that corresponds to its parent and then traverse 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 candidates for a canon.
- To canonize a 3-connected component G_i, we use the log-space algorithm from Section 3. 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 rotation systems [51]. Hence, there are four possible candidates for the canon of G_i.

In the latter two cases, 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 candidate 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. (For instance, if the whole graph is simply a cycle or a 3-connected

component, then the algorithm terminates here.) If G_i and H_j contain virtual edges, then these edges are specially treated in the bitwise comparison of C_g and C_h:

- 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.
- 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, then 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.
  - If we find that the canons of 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 in Section 4.3.2. 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 reference direction is defined to be the smaller canon.

We eliminate the candidate canons that were found to be the larger in at least one of the comparisons. In the end, the candidate that is not eliminated is the canon. If we have the same canons for both G_i and H_j, then we define $S_{G_i} \equiv_{\mathbb{T}} T_{H_j}$. 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 the algorithm of Section 3. If the trees contain several components, then our definition of $S_{G_i} \equiv_{\mathbb{T}} T_{H_j}$ guarantees that we can combine the isomorphisms of the components to an isomorphism between $\mathsf{graph}(S_{G_i})$ and $\mathsf{graph}(T_{H_j})$.

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. This 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.

4.3.2 Isomorphism Order of Two Subtrees Rooted at Separating Pairs. We consider the isomorphism order of two subtrees $S_{\{a,b\}}$ and $T_{\{a',b'\}}$ rooted at separating pairs $\{a, b\}$ and $\{a', b'\}$, respectively. 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 $(H_1, \ldots, H_k)$.

The first three steps of the isomorphism order are performed similar to that of Lindell [36] maintaining the order profiles. We first order the subtrees, say $S_{G_1} \leq_{\mathbb{T}} \cdots \leq_{\mathbb{T}} S_{G_k}$ and $T_{H_1} \leq_{\mathbb{T}} \cdots \leq_{\mathbb{T}} T_{H_k}$, and verify that $S_{G_i} \equiv_{\mathbb{T}} T_{H_i}$ 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 $S_{G_i} \equiv_{\mathbb{T}} T_{H_i}$ 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.

An additional step involves a comparison of the *orientations* given by the subtrees S_{G_i} and T_{H_i} to $\{a, b\}$ and $\{a', b'\}$, respectively.

Definition 4.8 (Orientation). The *orientation given to the parent separating pair* $\{a, b\}$ *of* $S(G_i)$ is the direction $\{a, b\}$ that leads to the canon of $S(G_i)$, respectively. If the canons are obtained for both choices of directions of the edge, then we say that S_{G_i} is *symmetric about their parent separating pair* and thus does not give an orientation.

The orientation given to $\{a, b\}$ by two subtrees might be different. Our next step is to extract one orientation from the orientations of all subtrees as the *reference orientation* for separating pair $\{a, b\}$.

Definition 4.9 (Reference Orientation). Let $I_1 <_{\mathbb{T}} \cdots <_{\mathbb{T}} I_p$ be a partition of $(S_{G_1}, \ldots, S_{G_k})$ into classes of $\equiv_{\mathbb{T}}$-equal subtrees, for some $p \leq k$.

- For each isomorphism class I_j, the *orientation counter* is a pair $O_j = (c_j^{\rightarrow}, c_j^{\leftarrow})$, where $c_j^{\rightarrow}$ is the number of subtrees of I_j that gives one orientation, say (a, b), and $c_j^{\leftarrow}$ is the number of subtrees from I_j that give the other orientation, (b, a). The counters are ordered such that $c_j^{\rightarrow} \geq c_j^{\leftarrow}$. Then, the *orientation given to* $\{a, b\}$ *by isomorphism class* I_j is the one from the larger counter, i.e. $c_j^{\rightarrow}$, if $c_j^{\rightarrow} \neq c_j^{\leftarrow}$.

  If $c_j^{\rightarrow} = c_j^{\leftarrow}$, that is, 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.*

For $T_{\{a',b'\}}$ we similarly partition $(T_{H_1}, \ldots, T_{H_k})$ into isomorphism classes $I'_1 <_{\mathbb{T}} \cdots <_{\mathbb{T}} I'_p$. It follows that I_j and I'_j contain the same number of subtrees for every j. 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 $S_{\{a,b\}} \equiv_{\mathbb{T}} T_{\{a',b'\}}$. Otherwise, let j be the smallest index such that $O_j \neq O'_j$. Then, we define $S_{\{a,b\}} <_{\mathbb{T}} T_{\{a',b'\}}$ if O_j is lexicographically smaller than O'_j, and $T_{\{a',b'\}} <_{\mathbb{T}} S_{\{a,b\}}$ otherwise. For an example, see Figure 4.3.

4.3.3 Summary and Correctness. We summarize the isomorphism order of two triconnected component trees S and T defined in the previous subsections. Let $s = \{a, b\}$ and $t = \{a', b'\}$ be the roots of S and T, and let $\#s$ and $\#t$ be the number of children of s and t, respectively. Then, we have $S <_{\mathbb{T}} T$ if:

(1) $|S| < |T|$, or
(2) $|S| = |T|$ but $\#s < \#t$, or
(3) $|S| = |T|$, $\#s = \#t = k$, but $(S_{G_1}, \ldots, S_{G_k}) <_{\mathbb{T}} (T_{H_1}, \ldots, T_{H_k})$ lexicographically, where we assume that $S_{G_1} \leq_{\mathbb{T}} \cdots \leq_{\mathbb{T}} S_{G_k}$ and $T_{H_1} \leq_{\mathbb{T}} \cdots \leq_{\mathbb{T}} T_{H_k}$ are the ordered subtrees of S and T, 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) $|S| = |T|$, $\#s = \#t = k$, $(S_{G_1}, \ldots, S_{G_k}) \equiv_{\mathbb{T}} (T_{H_1}, \ldots, T_{H_k})$, 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 S and T are *equal according to the isomorphism order*, denoted by $S \equiv_{\mathbb{T}} T$, if neither $S <_{\mathbb{T}} T$ nor $T <_{\mathbb{T}} S$ holds.

The following theorem shows the correctness of the isomorphism order: two trees are $\equiv_{\mathbb{T}}$-equal, precisely when the underlying graphs are isomorphic.

Theorem 4.10. *Let G and H be biconnected planar graphs with triconnected component trees S and T, respectively. Then, G and H are isomorphic if and only if there is a choice of separating pairs s, t in G and H such that $S \equiv_{\mathbb{T}} T$ when rooted at s and t, respectively.*

Proof. Assume that $S \equiv_{\mathbb{T}} T$. 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 $d + 2$ to d. If the

grandchildren of separating pairs, say s and t, are $\equiv_{\mathbb{T}}$-equal up to Step 4, then we compare the children of s and t. If they are equal, then we can extend the $\equiv_{\mathbb{T}}$-equality to the separating pairs s and t.

When subtrees are rooted at separating pair nodes, the comparison describes an order on the subtrees that 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 easily as well. Suppose 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. One needs to prove that $S_{\{a,b\}} \equiv_{\mathbb{T}} T_{\{a',b'\}}$ where these two are S and T rooted at $\{a, b\}$ and $\{a', b'\}$, respectively. One can prove this by induction on the depth of $S_{\{a,b\}}$ and $T_{\{a',b'\}}$. Note that such an isomorphism maps 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 $\equiv_{\mathbb{T}}$. One can combine this with the isomorphism of the triconnected components themselves and the definition of $\equiv_{\mathbb{T}}$ to conclude the proof of the reverse direction and of the theorem. □

4.4 Space Complexity of the Isomorphism Order Algorithm

We analyze 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 [36]. We show that Steps 3 and 4 can also be performed in log-space.

We use the algorithm from Section 3 to canonize a 3-connected component G_i of size n_{G_i} in space $O(\log n_{G_i})$. If the component is a 3-bond or a cycle, then we use the ideas presented in Section 4.3.1 to canonize them again using $O(\log n_{G_i})$ space.

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 that 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 that 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, then 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 $n \leq N \leq O(n^2)$ [27]. This proves the following theorem.

Theorem 4.11. *The isomorphism order between two triconnected component trees of biconnected planar graphs can be computed in log-space.*

4.5 The Canon of a Biconnected Planar Graph

Once we know the order among the subtrees, it is straightforward to canonize the triconnected component tree S. We traverse S in the tree isomorphism order as in Lindell's algorithm, 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 call this list of delimiters and canons of components a *canonical list* of S.

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 that, when chosen as the root, leads to the lexicographically minimum canonical list of S. We call this the *tree-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 the canonical list for $S_{\{a,b\}}$. In the second step, we compute the canon for the biconnected planar graph 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 that give the reference orientation to the parent are considered before those that 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 that 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, then 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 4.2. 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 vertices are still the ones they have in G. Clearly, a canon must be independent of the original names of the vertices. The final canon for $S_{\{a,b\}}$ can be obtained by a log-space transducer that 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 that transforms an automorphism ϕ of $S_{\{a,b\}}$ into an automorphism $f(\phi)$ for G with $\{a, b\}$ fixed. This completes the proof of Theorem 4.1.

5 CANONIZATION OF PLANAR GRAPHS

In this section, we use all the machinery built so far to obtain our main result.

Theorem 5.1. *The canonization of planar graphs is in log-space.*

The proof of this is presented in the following subsections. In Section 5.1, we first define the *biconnected component tree* of a connected planar graph and list some of its properties. In Section 5.2, we define an isomorphism order for biconnected component trees. Two trees will have the same order if and only if the planar graphs represented by them are isomorphic. The computation of such an order gives a test for isomorphism of planar graphs. In Section 5.3, we do a space analysis of our algorithm and show that isomorphism testing can be done in log-space for planar graphs. Finally, in Section 5.4, we give a log-space canonization algorithm. This proves Theorem 5.1.

5.1 Biconnected Component Tree of a Planar Graph

Biconnected component trees are defined analogously to triconnected component trees. Recall from Section 2 that when a graph is split along an articulation point a, each biconnected split component contains a copy of a.

Definition 5.2. Let G be a connected graph. The *biconnected component tree* T *of* G is the following graph. There is a node for each biconnected component and for each articulation point of G. There is an edge in T between the node for biconnected component B and the node for an articulation point a, if a occurs in B.

It is easy to see that the graph T obtained in Definition 5.2 is in fact a tree. This tree is unique, i.e., independent of the order in which the articulation points are chosen to split graph G. The biconnected component tree can be constructed in log-space: Articulation points can be computed in log-space, as explained in Section 2. Two vertices are in the same biconnected component if they are not separated by an 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 a has at most one copy in each of the biconnected components, the corresponding triconnected component trees can have many copies of a, in case 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 in Lindell's algorithm and as in Section 4.3. 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 biconnected component trees.

Lemma 5.3. *Let B be a biconnected component in the biconnected component tree S and let $\mathcal{T}(B)$ be its triconnected component tree. Then, the following holds:*

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

The proofs of the above properties follow easily through folklore graph theoretic arguments and are omitted here.

5.2 Isomorphism Order for Biconnected Component Trees

In this section, we start with two biconnected component trees of connected planar graphs and give a log-space test for isomorphism of the planar graphs represented by them. The idea is again to come up with an *order* on the biconnected component trees, similar to the case of triconnected component trees. We call the resulting order the *isomorphism order for biconnected component trees*. We ensure that two biconnected component trees are equal with respect to this order if and only if the planar graphs represented by them are isomorphic.

The size of a triconnected component tree was defined in Definition 4.7. Here, we extend the definition to biconnected component trees.

Definition 5.4. Let B be a biconnected component node in a biconnected component tree S, and let $\mathcal{T}(B)$ be the triconnected component tree of B. The *size of* B is defined as $|\mathcal{T}(B)|$. The size of an articulation point node in S is defined as 1. The *size of* S, denoted by $|S|$, is the sum of the sizes of its component nodes.

Note that the articulation points in the definition may be counted several times, namely, in every component they occur.

Let S and T be two biconnected component trees rooted at nodes s and t corresponding to articulation points a and a', and let $\#s$ and $\#t$ be the number of children of s and t, respectively. We define $S <_{\mathbb{B}} T$ if:

(1) $|S| < |T|$ or
(2) $|S| = |T|$ but $\#s < \#t$ or
(3) $|S| = |T|$, $\#s = \#t = k$, but $(S_{B_1}, \ldots, S_{B_k}) <_{\mathbb{B}} (T_{B'_1}, \ldots, T_{B'_k})$ lexicographically, where we assume that $S_{B_1} \leq_{\mathbb{B}} \cdots \leq_{\mathbb{B}} S_{B_k}$ and $T_{B'_1} \leq_{\mathbb{B}} \cdots \leq_{\mathbb{B}} T_{B'_k}$ are the ordered subtrees of S and T, respectively.

We postpone the definition of the order between the subtrees S_{B_i} and $T_{B'_j}$ in Step 3 to Section 5.2.1 below.

We say that two biconnected component trees are *equal according to the isomorphism order*, denoted by $S \equiv_{\mathbb{B}} T$, if neither of $S <_{\mathbb{B}} T$ and $T <_{\mathbb{B}} S$ holds.

Figure 5.1 illustrates the definition.

5.2.1 Outline of the Algorithm for Computing the Isomorphism Order. The Steps 1 and 2 above are easy to implement in log-space, as done before. We now give the details for Step 3.

Assume that equality is found in Step 1 and 2. The inductive ordering of the subtrees of S and T 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

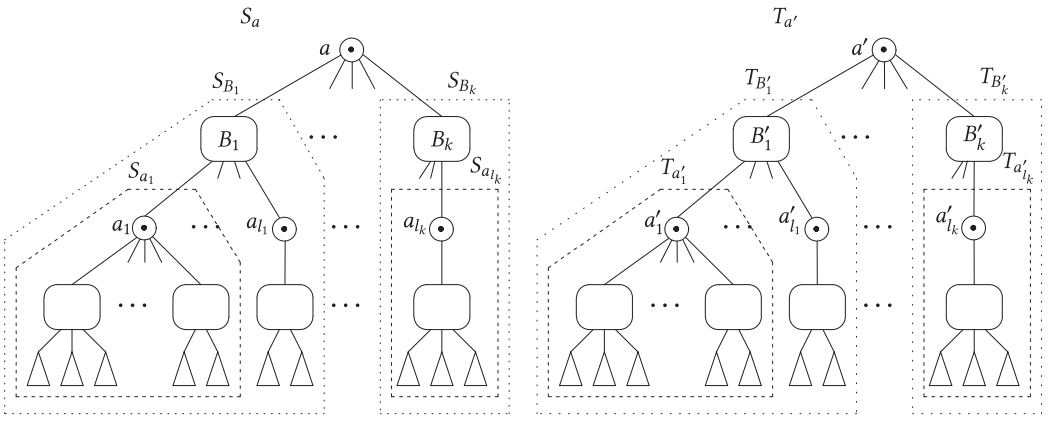

Fig. 5.1. Comparison of the biconnected component trees S_a and $T_{a'}$ rooted at nodes for articulation points a and a'. If the root nodes have the same number k of children, then we compare the nodes $B_1, \ldots, B_k$ of S_a with the nodes $B'_1, \ldots, B'_k$ of $T_{a'}$. Thereby, we recursively compare the subtrees at the articulation nodes we find in these components.

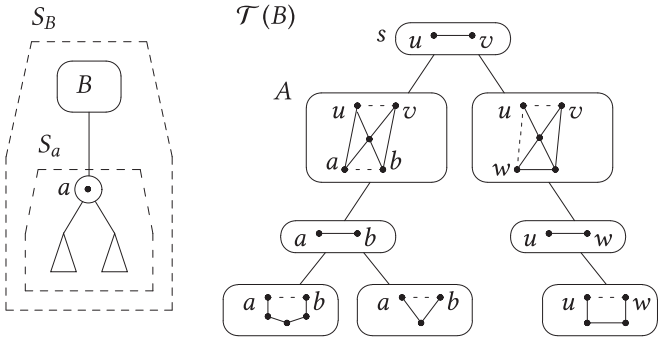

Fig. 5.2. A biconnected component tree S_B rooted at biconnected component B that has an articulation point a as child, which occurs several times in the triconnected component tree $\mathcal{T}(B)$ of B. In A and the other triconnected components, the dashed edges are separating pairs.

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.

To compare two subtrees S_B and $T_{B'}$, rooted at biconnected component nodes B and B', respectively, we start by constructing and comparing the canons of their triconnected component trees $\mathcal{T}(B)$ and $\mathcal{T}(B')$. To do so, we have to choose a separating pair as root in each of $\mathcal{T}(B)$ and $\mathcal{T}(B')$.

For notation, we call it the *outer algorithm* when we do comparisons for the biconnected component trees S_B and $T_{B'}$. The outer algorithm at this point invokes the *inner algorithm*, the recursive comparison algorithm for $\mathcal{T}(B)$ and $\mathcal{T}(B')$.

The inner algorithm may encounter several copies of articulation points a, a', inside $\mathcal{T}(B)$ and $\mathcal{T}(B')$, respectively. Figure 5.2 shows an example. We want to choose one of them where we go into recursion.

Definition 5.5. The *reference copy* of an articulation point a in the rooted triconnected component tree $\mathcal{T}(B)$ is the copy of point a that is closest to the root of $\mathcal{T}(B)$.

By Lemma 5.3, the reference copy is defined uniquely.

All but the reference copies of these articulation points are ignored by this algorithm. For the reference copies, the current order profiles computed by the inner algorithm so far are stored in the memory and the outer algorithm takes over for recursively comparing subtrees of a, a'. This switch between inner and outer algorithm thus causes some bits of storage in the memory. The main task is to limit the number of things that are stored to get an overall log-space bound.

To bound the space, it is crucial to limit the choices of separating pair nodes of $\mathcal{T}(B)$ and $\mathcal{T}(B')$, which can be used as roots for these trees. For now, we will assume that the number of choices for the root is at most κ and proceed with the description of the inner and outer algorithms. We will give appropriate bounds on κ in Section 5.2.2 below.

- For κ possibilities of roots, one is fixed for $\mathcal{T}(B)$ and the canonical ordering of it is compared with that of $\mathcal{T}(B')$ according to $<_{\mathbb{T}}$, for all choices of κ roots. This is then done for each choice of the root of $\mathcal{T}(B)$. The aim is to compare the minimum canonical codes of $\mathcal{T}(B)$ and $\mathcal{T}(B')$ and return the result.
- The comparison of $\mathcal{T}(B)$ and $\mathcal{T}(B')$ for some choices of roots is now carried out using the isomorphism order procedure for triconnected component trees. During the comparison of $\mathcal{T}(B)$ and $\mathcal{T}(B')$, if a copy of an articulation point is encountered in a canonical code of a triconnected component node C of $\mathcal{T}(B)$, but not in that of the corresponding node C' in $\mathcal{T}(B')$, then that canonical code for C is considered to be larger and is eliminated. If copies of articulation points u and u' are encountered simultaneously in nodes C and C', and if they are their reference copies, then a recursive call to the isomorphism order procedure for biconnected component trees (outer algorithm) is made to compare the subtrees of S_B and T'_B rooted at u and u'. If the copies encountered are not the reference copies, then equality is assumed and the inner algorithm proceeds. While making the recursive call, the current order profile of C or C' is stored along with the bit-vector for already eliminated canonical codes.

5.2.2 Limiting the Number of Possible Choices for the Root Separating Pair. Here, we prove that the choices for the root nodes in triconnected component trees can be bounded effectively.

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 5.1). We partition the subtrees $S_{a_1}, \ldots, S_{a_l}$ into classes $E_1, \ldots, E_p$ of equal size subtrees, where size is according to Definition 5.4. Let $k_j = |E_j|$ 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. Recall from Lemma 5.3 that articulation point a is *associated with* the unique component A in $\mathcal{T}(B)$ that contains a and is nearest to the center C_0 of $\mathcal{T}(B)$.

To limit the number of potential root nodes for $\mathcal{T}(B)$, we do a case analysis according to properties of the center C_0 of $\mathcal{T}(B)$. In some of the cases, we succeed directly to give the desired bound. In the remaining cases, we will show that the number of automorphisms of the center C_0 is small. This suffices for our purpose: In this case, for every edge as starting edge, we canonize the component C_0 separately, i.e., without going into recursion on the separating pairs and articulation points of C_0. Thereby, we construct the set of starting edges, say E_0, that lead to the minimum canon for C_0. Although there are polynomially many possible candidates for the canon, the minimum ones are bounded by the number of automorphisms of C_0, which is small.

Now, we take the first separating pair encountered in each of the candidate canons obtained when starting from edges in S. We take this set of separating pairs as the potential root nodes for $\mathcal{T}(B)$, and hence, its cardinality is bounded by the number of automorphisms of C_0.

If B contains no separating pairs, i.e., $B = C_0$, then we cycle through the edges in S to compute the canon of B.

We start our case analysis. Recall that articulation point a is the parent of B and C_0 is the center of the triconnected component tree $\mathcal{T}(B)$.

- **The center C_0 of $\mathcal{T}(B)$ is a separating pair.** We choose this separating pair as the root of $\mathcal{T}(B)$. Thus, we have only one choice for the root.
- **C_0 is a triconnected component and a is not associated with C_0.** Let a be associated with a triconnected component A in $\mathcal{T}(B)$. We find the path from A to C_0 in $\mathcal{T}(B)$ and find the separating pair closest to C_0 on this path. This serves as the unique choice for the root of $\mathcal{T}(B)$.
- **C_0 is a cycle and a is associated with C_0.** Consider the virtual edges closest to a on cycle C_0. There are at most two of them. We choose the separating pairs corresponding to these virtual edges as the root candidates of $\mathcal{T}(B)$. Thus, we get at most two choices for the root of $\mathcal{T}(B)$.
- **C_0 is a 3-connected component and a is associated with C_0.** We proceed with a 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 $\mathcal{T}(B)$.
  Case II: $l = 1$. If B has exactly one articulation point besides a, say a_1, 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 two sub-cases.

  (1) **Some articulation point a_j in class E_1 is not associated with C_0.** Let a_j be associated with a triconnected component $D \neq C_0$. Find the path from D to C_0 in $\mathcal{T}(B)$ and select the separating pair node closest to C_0 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_0.** We distinguish sub-cases according to the size of E_1.

    (a) If $k_1 \geq 2$, then by Lemma 5.8 below, C_0 can have at most $2k_1$ automorphisms. Thus, we have at most $2k_1$ ways of choosing the root of $\mathcal{T}(B)$.

    (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_0.** We do the same with a_j as in sub-case III (1). We get up to k_2 separating pairs as candidates for the root.

      (ii) **All articulation points in E_2 are associated with C_0.**
      If $k_2 \geq 2$, then we process the child in E_1 *a priori* and store the result. Similar as in sub-case III (2a), we have at most $2k_2$ ways of choosing the root of $\mathcal{T}(B)$.

      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_0 has at most one non-trivial automorphism in this case. Thus, we have at most two ways of choosing the root of $\mathcal{T}(B)$.

Let $N = |S_B|$. We assume that all subtrees are of size $\leq N/2$, because otherwise such a subtree is considered as *large* and processed *a priori* by the algorithm as opposed to going into the recursion for it (see the paragraph on *large children* below).

It remains to prove the bounds claimed above on the number of automorphism of the 3-connected components. We use the following lemma that provides an automorphism preserving embedding of a 3-connected planar graph on the 2-sphere.

Lemma 5.6 ([7] (P. Mani)). *Every 3-connected planar graph G can be embedded on the 2-sphere as a convex polytope P such that the automorphism group of G is induced by the automorphism group of the convex polytope P formed by the embedding.*

The following corollary of the lemma justifies sub-case III (2b ii).

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 from Lemma 5.6. The only possible symmetry is a reflection about the plane containing the colored vertices, which leads to exactly one non-trivial automorphism. □

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 when the size of the second largest color class is at least 2 as considered in subcase III (2a).

Lemma 5.8. *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 that contains a. Then, G has $\leq 2k$ automorphisms.*

Proof. Point a is fixed, therefore the orientation preserving part of the automorphism group is cyclic (see, e.g., Lemma 3 in Reference [4]) and extends as rotations to the sphere. By Lemma 5.6, this implies that there are at most k such rotations. Thus, if we add the rotation reversing part, then we get an upper bound of $2k$ on the order of the automorphism group. □

5.2.3 Summary and Correctness of the Isomorphism Order. We argue that two biconnected component trees are equal for the isomorphism order for some choice of the root if and only if the corresponding graphs are isomorphic.

Theorem 5.9. *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 $S_a \equiv_{\mathbb{B}} T_{a'}$.*

Proof. Assume that $S_a \equiv_{\mathbb{B}} T_{a'}$. 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 $d + 2$ to d. If the grandchildren of articulation points, say s and t, are $\equiv_{\mathbb{B}}$-equal up to Step 3, then we compare the children of s and t. If they are equal, then we can extend the $\equiv_{\mathbb{B}}$-equality to the articulation points s and t.

When subtrees are rooted at articulation point nodes, the comparison describes an order on the subgraphs that 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 4.10) 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 easily as well. Suppose 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. One needs to prove that the biconnected component trees S_a of G and $T_{a'}$ of H rooted, respectively, at a and a' will be $\equiv_{\mathbb{B}}$. Again, we proceed by induction on the depth of S_a and $T_{a'}$. An isomorphism maps articulation points of G to articulation points of H. Further, 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 $\equiv_{\mathbb{B}}$. Thus, combined with the isomorphism of corresponding biconnected components and the definition of $\equiv_{\mathbb{B}}$, this yields the reverse direction and completes the proof. □

5.3 Space Complexity of the Isomorphism Order Algorithm

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', then 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 $\mathcal{T}(B)$ and $\mathcal{T}(B')$ according to B and B'. When we compare triconnected components in $\mathcal{T}(B)$ and $\mathcal{T}(B')$, 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, then we check whether we reach them for the first time, i.e., whether we are at the reference copies of a and a'. 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.

Next, we show that the position of the reference copy of an articulation point, 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.

Lemma 5.10. *The reference copy of an articulation point a in $\mathcal{T}(B)$ and a' in $\mathcal{T}(B')$ for the comparison of triconnected component trees $\mathcal{T}(B)$ with $\mathcal{T}(B')$ can be found in log-space.*

Proof. To prove the lemma, we distinguish three cases for a in $\mathcal{T}(B)$. Assume that we have the same situation for a' in $\mathcal{T}(B')$. 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 $\mathcal{T}(B)$. That is, a occurs already at the beginning of the comparisons for $\mathcal{T}(B)$. Then, we define A as the root separating pair.
- Articulation point a occurs in separating pairs other than the root of $\mathcal{T}(B)$. Then, a occurs in all the component nodes, which contain such a separating pair. By the construction of the tree, these nodes form a connected subtree of $\mathcal{T}(B)$. Hence, one of these component nodes is the closest to the root of $\mathcal{T}(B)$. 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 $\mathcal{T}(B)$. 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 that 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 that 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 occurrence 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 $\mathcal{T}(B)$ and $\mathcal{T}(B')$. 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 recursion 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 $\mathcal{T}(B)$ and $\mathcal{T}(B')$.

In this part, we concentrate on the comparison of $\mathcal{T}(B)$ and $\mathcal{T}(B')$. 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 4.4. We summarize the changes.

- We use the size function according to Definition 5.4. That is, the size of a triconnected subtree rooted at a node C in $\mathcal{T}(B)$ also includes the sizes of the biconnected subtrees rooted at the reference articulation points that appear in the subtree of $\mathcal{T}(B)$ 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 $\mathcal{T}(B)$. Hence, whenever we make recomputations in $\mathcal{T}(B)$, we have to find the root separating pair node first. For this, we compute $\mathcal{T}(B)$ 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 4.4, 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 4.4.

For the triconnected component trees $\mathcal{T}(B)$ and $\mathcal{T}(B')$, we get the same space-bounds as in the previous section. That is, for the cross-comparison of the children of separating pair nodes s of $\mathcal{T}(B)$ and t of $\mathcal{T}(B')$, 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 jth 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 $\geq N/2$), and we treat them *a priori*. We will discuss this below.

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 jth 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 $\mathcal{T}(B_i)$ and $\mathcal{T}(B'_i)$. We have a set of separating pairs as candidates for the root of $\mathcal{T}(B_i)$. 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 $\mathcal{T}(B_i)$. Except for the trivial cases, the algorithm uses $O(\log k_i)$ space when it starts to compare the trees $\mathcal{T}(B_i)$ and $\mathcal{T}(B_i')$.

Assume now that we compare $\mathcal{T}(B_i)$ and $\mathcal{T}(B_i')$. 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 occurred 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 $\mathcal{T}(B)$ and $\mathcal{T}(B')$. 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 occurrence of a and a'.

Large children. As in the case of biconnected graphs in Section 4.1, 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 $\mathcal{T}(B)$, the triconnected component tree of B. Let A be a triconnected component and $\{u, v\}$ be a separating pair in $\mathcal{T}(B)$. 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.

Definition 5.11. Let B be a biconnected component and $\mathcal{T}(B)$ its triconnected component tree. Let C be a node in $\mathcal{T}(B)$, i.e., a triconnected component node or a separating pair node. The *tree* S_C^* *rooted at* C consists of the subtree of $\mathcal{T}(B)$ rooted at C (with respect to the root of $\mathcal{T}(B)$) and of the subtrees S_a for all articulation points a that have a reference copy in the subtree of $\mathcal{T}(B)$ rooted at C, except 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 C of size $\geq N/2$.

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 or even before in the following case: Let a be an articulation point in the biconnected component B and let C be the node in $\mathcal{T}(B)$ 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 $\mathcal{T}(B)$. 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, 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 5.3 shows an example.

We summarize, 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 $O(1)$

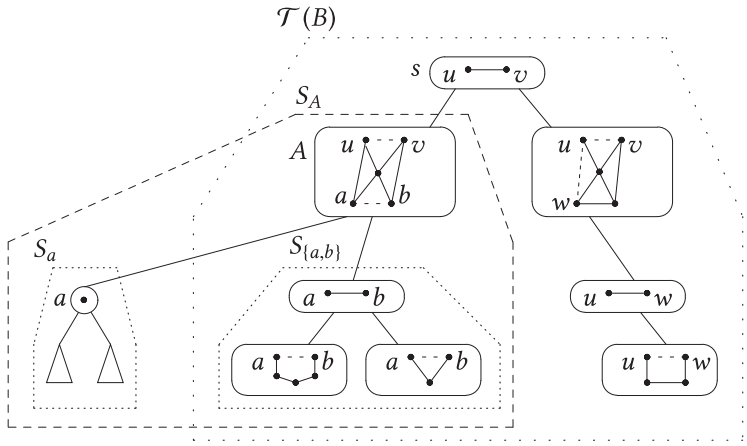

Fig. 5.3. The triconnected component tree $\mathcal{T}(B)$ of the biconnected component B. The triconnected component A contains the reference copy of articulation point a. If S_a is not a large child of B, then the subtree S_A consists of the subtree of $\mathcal{T}(B)$ rooted at A and the subtree S_a. In contrast, S_a is not part of the subtree $S_{\{a,b\}}$, because it does not contain the reference copy of a.

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 $\mathcal{T}(B)$ and $\mathcal{T}(B')$.
- 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.
- Large children with respect to S^*_C, where C is a node in $\mathcal{T}(B)$. These are children of node B in S_B, which are not large children of B. These children are articulation point nodes that have a reference copy in C.

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 jth 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 jth 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 5.12. *The isomorphism order between two planar graphs can be computed in log-space.*

5.4 The Canon of a Planar Graph

From Theorem 5.12, 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 5.1. Let $l(B_i, a)$ be the canonical list of edges of the biconnected component B_i, i.e., the canonical list of $\mathcal{T}(B_i)$ 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 $\mathcal{T}(B_i)$. 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. This proves Theorem 5.1.

6 CONCLUSION

In this article, 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. 2009. Planar and grid graph reachability problems. *Theor. Comput. Syst.* 45, 4 (2009), 675–723. DOI: https://doi.org/10.1007/s00224-009-9172-z

[2] Eric Allender and Meena Mahajan. 2004. The complexity of planarity testing. *Inf. Computat.* 189, 1 (2004), 117–134. DOI: https://doi.org/10.1016/j.ic.2003.09.002

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

[4] V. Arvind and Nikhil Devanur. 2004. Symmetry breaking in trees and planar graphs by vertex coloring. In *the Nordic Combinatorial Conference (NORCOM)*. Department of Mathematical Sciences.
[5] V. Arvind and Piyush P. Kurur. 2006. Graph isomorphism is in SPP. *Inf. Computat.* 204, 5 (2006), 835–852.
[6] L. Babai. 1985. Trading group theory for randomness. In *17th ACM Symposium on Theory of Computing (STOC)*. ACM Press, 421–429.
[7] László Babai. 1995. Automorphism groups, isomorphism, reconstruction. *Handb. Combinat.* 2 (1995), 1447–1540.
[8] László Babai. 2016. Graph isomorphism in quasipolynomial time [extended abstract]. In *48th ACM Symposium on Theory of Computing, (STOC)*. ACM Press, 684–697.
[9] László Babai and Eugene M. Luks. 1983. Canonical labeling of graphs. In *15th ACM Symposium on Theory of Computing (STOC)*. ACM Press, 171–183.
[10] H. L. Bodlaender. 1990. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algor.* 11 (1990), 631–643.
[11] Ravi B. Boppana, Johan Hastad, and Stathis Zachos. 1987. Does co-NP have short interactive proofs? *Inform. Process. Lett.* 25, 2 (1987), 127–132.
[12] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. 2009. Directed planar reachability is in unambiguous Log-Space. *ACM Trans. Computat. Theor.* 1, 1 (2009), 4:1–4:17. DOI: https://doi.org/10.1145/1490270.1490274
[13] Maurice Chandoo. 2016. Deciding circular-arc graph isomorphism in parameterized logspace. In *33rd Symposium on Theoretical Aspects of Computer Science (STACS) (LIPIcs*, Vol. 47*)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
[14] Stephen A. Cook. 1985. A taxonomy of problems with fast parallel algorithms. *Inf. Contr.* 64, 1-3 (1985), 2–22.
[15] William H. Cunningham and Jack Edmonds. 1980. A combinatorial decomposition theory. *Canad. J. Math.* 32 (1980), 734–765.
[16] Bireswar Das, Jacobo Torán, and Fabian Wagner. 2012. Restricted space algorithms for isomorphism on bounded treewidth graphs. *Inf. Computat.* 217 (2012), 71–83.
[17] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. 2009. Planar graph isomorphism is in Log-Space. In *IEEE Conference on Computational Complexity (CCC)*. IEEE Computer Society, 203–214.
[18] Samir Datta, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. 2009. Isomorphism for $K_{3,3}$-free and K_5-free graphs is in Log-Space. In *29th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 145–156.
[19] Michael Elberfeld and Ken-ichi Kawarabayashi. 2014. Embedding and canonizing graphs of bounded genus in Log-Space. In *Symposium on Theory of Computing (STOC)*. ACM Press, 383–392.
[20] Michael Elberfeld and Pascal Schweitzer. 2017. Canonizing graphs of bounded tree width in Log-Space. *ACM Trans. Computat. Theor.* 9, 3 (2017). DOI: https://doi.org/10.1145/3132720
[21] Hillel Gazit and John H. Reif. 1998. A randomized parallel algorithm for planar graph isomorphism. *J. Algor.* 28, 2 (1998), 290–314.
[22] O. Goldreich, S. Micali, and A. Wigderson. 1991. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM* 38 (1991), 691–729.
[23] S. Goldwasser and M. Sipser. 1989. Private coins versus public coins in interactive proof systems. *Adv. Comput. Res.* 5 (1989), 73–90.
[24] M. Grohe and O. Verbitsky. 2006. Testing graph isomorphism in parallel by playing a game. In *33rd International Colloquium on Automata, Languages and Programming (ICALP) (Lecture Notes in Computer Science*, Vol. 4051*)*. Springer-Verlag, 3–14.
[25] John E. Hopcroft and Robert E. Tarjan. 1972. *Finding the Triconnected Components of a Graph*. Technical Report 72-140. Cornell University.
[26] John E. Hopcroft and Robert Endre Tarjan. 1972. Isomorphism of planar graphs. In *Complexity of Computer Computations (The IBM Research Symposia Series)*. Plenum Press, New York, 131–152.
[27] John E. Hopcroft and Robert E. Tarjan. 1973. Dividing a graph into triconnected components. *SIAM J. Comput.* 2, 3 (1973), 135–158.
[28] John E. Hopcroft and J. K. Wong. 1974. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *6th ACM Symposium on Theory of Computing (STOC)*. ACM Press, 172–184.
[29] Birgit Jenner, Johannes Köbler, Pierre McKenzie, and Jacobo Torán. 2003. Completeness results for graph isomorphism. *J. Comput. Syst. Sci.* 66, 3 (2003), 549–566.
[30] Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. 2017. The Weisfeiler-Leman dimension of planar graphs is at most 3. In *32nd ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1–12.
[31] Johannes Köbler, Sebastian Kuhnert, Bastian Laubner, and Oleg Verbitsky. 2011. Interval graphs: Canonical representations in logspace. *SIAM J. Comput.* 40, 5 (2011), 1292–1315.

[32] Johannes Köbler, Sebastian Kuhnert, and Oleg Verbitsky. 2013. Helly circular-arc graph isomorphism is in logspace. In *38th Symposium on Mathematical Foundations of Computer Science (MFCS) (Lecture Notes in Computer Science,* Vol. 8087*)*. Springer, 631–642.
[33] Michal Koucký. 2002. Universal traversal sequences with backtracking. *J. Comput. Syst. Sci.* 65, 4 (2002), 717–726.
[34] Jacek P. Kukluk, Lawrence B. Holder, and Diane J. Cook. 2004. Algorithm and experiments in testing planar graphs for isomorphism. *J. Graph Algor. Applic.* 8, 2 (2004), 313–356.
[35] John Lind and Albert R. Meyer. 1973. A characterization of log-space computable functions. *SIGACT News* 5, 3 (July 1973), 26–29. DOI: https://doi.org/10.1145/1008293.1008295
[36] Steven Lindell. 1992. A logspace algorithm for tree canonization (extended abstract). In *24th ACM Symposium on Theory of Computing (STOC)*. ACM Press, 400–404.
[37] Saunders Maclane. 1937. A structural characterization of planar combinatorial graphs. *Duke Math. J.* 3 (1937), 460–472.
[38] Gary L. Miller and Vijai Ramachandran. 1992. A new graph triconnectivity algorithm and its parallelization. *Combinatorica* 12 (1992), 53–76.
[39] Gary L. Miller and John H. Reif. 1991. Parallel tree contraction part 2: Further applications. *SIAM J. Comput.* 20, 6 (1991), 1128–1147.
[40] Bojan Mohar and Carsten Thomassen. 2001. *Graphs on Surfaces*. Johns Hopkins University Press, Baltimore (MD), London. Retrieved from http://opac.inria.fr/record=b1131924.
[41] Ilia N. Ponomarenko. 1991. The isomorphism problem for classes of graphs closed under contraction. *J. Math. Sci.* 55 (1991), 1621–1643.
[42] Omer Reingold. 2008. Undirected connectivity in Log-Space. *J. ACM* 55, 4 (2008), 1–24.
[43] Klaus Reinhardt and Eric Allender. 2000. Making nondeterminism unambiguous. *SIAM J. Comput.* 29, 4 (2000), 1118–1131. DOI: https://doi.org/10.1137/S0097539798339041
[44] Thomas Thierauf and Fabian Wagner. 2010. The isomorphism problem for planar 3-connected graphs is in unambiguous logspace. *Theor. Comput. Syst.* 47, 3 (2010), 655–673.
[45] Thomas Thierauf and Fabian Wagner. 2014. Reachability in $K_{3,3}$-free graphs and K_5-free graphs is in unambiguous logspace. *Chicago J. Theoret. Comput. Sci.* 2 (2014), 1–29.
[46] Jacobo Torán. 2004. On the hardness of graph isomorphism. *SIAM J. Comput.* 33, 5 (2004), 1093–1108.
[47] Oleg Verbitsky. 2007. Planar graphs: Logical complexity and parallel isomorphism tests. In *24th International Symposium on Theoretical Aspects of Computer Science (STACS)*. Springer, 682–693.
[48] Fabian Wagner. 2007. Hardness results for tournament isomorphism and automorphism. In *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS) (Lecture Notes in Computer Science,* Vol. 4708*)*. Springer, 572–583.
[49] Louis Weinberg. 1966. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circ. Theor.* 13 (1966), 142–148.
[50] Douglas B. West. 2000. *Introduction to Graph Theory* (2nd ed.). Prentice Hall.
[51] Hassler Whitney. 1933. A set of topological invariants for graphs. *Amer. J. Math.* 55 (1933), 235–321.

Received July 2020; revised May 2022; accepted June 2022