Space-Efficient Biconnected Components and Recognition of Outerplanar Graphs

Frank Kammer[1], Dieter Kratsch[2], and Moritz Laudahn[3]

1 **Institut für Informatik, Universität Augsburg, Augsburg, Germany**
kammer@informatik.uni-augsburg.de
2 **LITA, Université de Lorraine, Metz, France**
dieter.kratsch@univ-lorraine.fr
3 **Institut für Informatik, Universität Augsburg, Augsburg, Germany**
moritz.laudahn@informatik.uni-augsburg.de

Abstract

We present space-efficient algorithms for computing cut vertices in a given graph with n vertices and m edges in linear time using $O(n + \min\{m, n \log\log n\})$ bits. With the same time and using $O(n + m)$ bits, we can compute the biconnected components of a graph. We use this result to show an algorithm for the recognition of (maximal) outerplanar graphs in $O(n \log\log n)$ time using $O(n)$ bits.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases graph algorithms, space efficiency, cut vertices, maximal outerplanar graphs

Digital Object Identifier 10.4230/LIPIcs.MFCS.2016.56

1 Introduction

Nowadays the use of small mobile devices like tablets and smartphones is ubiquitous. Typically they will not be equipped with large memory and common actions like storing (many) pictures may even decrease the available memory significantly. This triggers the interest in data structures and algorithms being space-efficient. (Time) Efficient algorithms are a classical subject in computer science. The corresponding algorithms course is often based on the textbook of Cormen et al. [8]. There is also a long tradition in the development of algorithms that use as few bits as possible; famous results are the two results of Savitch [19] and Reingold [18] on reachability in directed and undirected graphs, respectively. However, the running times of their algorithms are far away from the fastest algorithms for that problem and are therefore of small practical interest. Moreover, Edmonds et al. [10] have shown in the so-called NNJAG model that only a slightly sublinear working-space bound is possible for an algorithm that solves the reachability problem when required to run in polynomial time. This motivates the recent interest in *space-efficient algorithms*, i.e., algorithms that use as few working space as possible under the condition that their running time (almost) matches the running time of the best algorithm(s) without any space restrictions.

A useful model of computation for the development of algorithms in that context is the word RAM with a read-only input, a read-write working memory, and a write-only output. As usual, we assume that for a given instances of size n, the word size is $\Omega(\log n)$. One of the first problems considered for space-efficient algorithms is sorting [16], which was finally solved to optimality [5, 17]. Other researchers considered problems in geometry [1, 3, 4].

Recent research started to focus on space-efficient graph algorithms [2, 11]. However, there is still only a very short list of problems with space-efficient graph algorithms: depth-first search, breadth-first search, (strongly) connected components, topological sorting, shortest path.

We continue this work on space-efficient graph algorithms and consider the basic problems to compute the cut vertices and to decompose a given undirected graph into its biconnected components. Tarjan's linear time algorithm [20] solving this problem has been implemented in almost any usual programming language. However the algorithm requires $\Omega(n \log n)$ bits on n-vertex graphs. The idea of our algorithm is to classify the edges via a DFS as tree and back edges and to mark those tree edges that build a cycle with a back edge. The marking allows us subsequently to determine the cut vertices and the biconnected components. Given a graph with n vertices and m edges, the whole algorithm runs in $O(n + m)$ time using $O(n + \min\{m, n \log\log n\})$ bits, which is $O(n)$ in sparse graphs. Due to the lower bound of Edmonds et al. [10], there is not much hope for an algorithm that uses $o(n)$ bits.

Finally we study the recognition of outerplanar graphs, i.e., those graphs having a planar embedding with all vertices on the outer face. The problem has been studied in various settings and linear time algorithms have been given, e.g., by Mitchell [15] and Wiegers [21]. However, both algorithms modify the given graph by removing vertices of degree 2, which is not possible in our model. An easy solution would be to copy the given graph in the working memory, but this requires $\Omega(n \log n)$ bits for a graph with n vertices. Another problem is that if the neighbors of a removed vertex are not adjacent, then both algorithms above want to add a new edge connecting the neighbors. Storing all these new edges also can require $\Omega(n \log n)$ bits. Our algorithm runs in time $O(n \log\log n)$ and uses $O(n)$ bits, and determines if the input graph is outerplanar, as well as if it is maximal outerplanar. To obtain our algorithm, we can not simply remove vertices of degree 2. With each removed vertex v we have to remove the so-called chain of vertices of degree 2 that contains v and we have to choose the chains carefully such that we have only very few new edges at a time in our graph.

2 Preliminary

For graph-theoretic notions not defined in the paper we refer to the monograph of Diestel [9]. For basic notions in construction and analysis of algorithms and a large collection of fundamental algorithms like, e.g., depth-first search (DFS) we refer to the textbook of Cormen et al. [8]. To develop space-efficient algorithms, the details of the representation of an input graph are more important than in the classic setting because it is rarely possible to modify and store a given representation. We use the terminology of [11]. In particular, if we say that a graph is represented via *adjacency arrays*, then we assume that, given a vertex u and an index i, we can determine the ith edge $\{u, v\}$ of u in constant time. Moreover, *cross pointers* allow us to determine the index of $\{u, v\}$ in the adjacency array of v in constant time. As usual, we always assume that an n-vertex graph has vertices $V = \{1, \ldots, n\}$.

Our algorithms make use of *rank-select data structures*. A rank-select data structure is initialized on a bit sequence $B = (b_1, \ldots, b_n)$ and then supports the following two queries.
$\boldsymbol{rank_B(j)}$ $(j \in \{0, \ldots, n\})$: Return $\sum_{i=1}^{j} b_i$
$\boldsymbol{select_B(k)}$ $(k \in \{1, \ldots, \sum_{i=1}^{n} b_i\})$: Return the smallest $j \in \{1, \ldots, n\}$ with $rank_B(j) = k$.
Rank-select data structures for bit sequences of length n that support rank and select queries in constant time and occupy $O(n)$ bits can be constructed in $O(n)$ time [7].

Assume that $d_1, \ldots, d_n \in \mathbb{N}_0$ and that it is desired to allocate n bit strings $A_1, \ldots, A_n$ such that, for $k = 1, \ldots, n$, A_k consists of d_k bits and (the beginning of) A_k can be located

in constant time. Take $N = \sum_{j=1}^{n} d_j$. We say that $A_1, \ldots, A_n$ are stored with *static space allocation* if we allocate $A_1, \ldots, A_n$ within an array of size N. In $O(n + N)$ time, we can compute the sums $s_k = k + \sum_{j=1}^{k-1} d_j$ for $k = 1, \ldots, n$ and a rank-select data structure for the bit vector B of size $n + N$ whose ith bit, for $i = 1, \ldots, n + N$, is 1 exactly if $i = s_k$ for some $k \in \{1, \ldots, n\}$. This allows us, given a $k \in \{1, \ldots, n\}$ to compute the number of bits used by the arrays $A_1, \ldots, A_{k-1}$ and thus the location of A_k in constant time by evaluating $select_B(k) - k$. One application of static space allocation—as already shown by [13]—is to store data for each vertex v consisting of $O(\deg(v))$ bits where $\deg(v)$ is the degree of v. Given a vertex, we then can locate its data in constant time and the whole data can be stored with $O(n + m)$ bits.

To maintain subsets of vertices or of edge indices we use a data structure by Hagerup and Kammer [12, Lemma 4.4] that is called a choice dictionary.

▶ **Theorem 1.** *Let $n \in \mathbb{N}$. A choice dictionary is a data structure that maintains an initially empty subset S of $\{1, \ldots, n\}$ under insertion, deletion, membership queries, and an operation called choice that returns an arbitrary element of S. The data structure can be initialized in $O(1)$ time using $O(n)$ bits of working space and supports each of its operations in constant time. The choice dictionary can also be extended by an operation called iteration that returns all elements in S in a time linear in $|S|$.*

We also use a simplified version of the ragged dictionary that was introduced by of Elmasry et al. [11, Lemma 2.1] and named in [12]. Missing proofs can be found in [14].

▶ **Theorem 2.** *For every fixed $n \in \mathbb{N} = \{1, 2, \ldots\}$ as well as integers $b = O(\log n)$ and $\kappa = O(n/\log n)$, there is a dictionary that can store a subset A of $\{1, \ldots, n\}$ with $|A| \leq \kappa$, each $a \in A$ with a string h_a of satellite data of b bits, in $O(n)$ bits such that the following operations all run in $O(\log \log n)$ time: h_a can be inspected for each $a \in A$ and elements with their satellite data can be inserted in and deleted from A.*

3 Cut Vertices

A *cut vertex* of a connected undirected graph G is a vertex v such that $G - v$ is disconnected. Furthermore a graph is *biconnected* if it is connected and does not have a cut vertex. We first show how to compute cut vertices in $O(n + m)$ time using $O(n + m)$ bits on an undirected graph $G = (V, E)$ with $n = |V|$ and $m = |E|$. Afterwards, we present a second algorithm that has the same running time and uses $O(n \log \log n)$ bits.

We start with the description of an algorithm, but for the time being, do not care on the running time and the amount of working space. Using a single DFS we are able to classify all edges as either tree edges or back edges. During the execution of a DFS we call a vertex white if it has not been reached by the DFS, gray if the DFS has reached the vertex, but has not yet retracted from it and black if the DFS has retracted from the vertex. W.l.o.g., we assume that all our DFS runs are deterministic such that every DFS run explores the edges of G in the same order. Let T denote the *DFS tree* of G, i.e., the subgraph of G consisting only of tree edges, which we always assume to be rooted at the start vertex of the DFS. We call a tree edge $\{u, v\}$ of T with u being the parent of v *full marked* if there is a back edge from a descendant of v to a strict ancestor of u, *half marked* if it is not full marked and there exists a back edge from a descendant of v to u, and *unmarked*, otherwise. Then one can easily prove the next lemma.

▶ **Lemma 3.** *Let T denote a DFS tree of a graph G with root r, then the following holds:*

1. *Every vertex $u \neq r$ is a cut vertex of G exactly if at least one of the edges from u to one of its children is either an unmarked edge or a half marked edge.*
2. *The vertex r is a cut vertex of G exactly if it has at least two children in T.*

Since we want to use the lemma above, we have to mark the tree edges as full marked, half marked or unmarked. Therefore, we run two DFS. In the first run, we classify all tree edges, which we initially unmark. During the second, whenever we discover a back edge $\{w, u\}$ with w being a descendant of u, it becomes evident that both u and w belong to the same biconnected component as do all vertices that are both, descendants of u and ancestors of w, since they induce a cycle C. Let v be the ancestor of w that is child of u. We mark the edge from u to v as half marked and all the other tree edges on C as full marked. If the edge $\{u, v\}$ has already been full marked in a previous step, this will not be overwritten. Note that if $\{u, v\}$ is half marked and u has a back edge to one of its ancestors such that the edge connecting u to its parent p becomes full marked, v and p do not belong to the same biconnected component, but u belongs to both, the biconnected component containing v and the biconnected component containing p. The notion to distinguish between half marked and full marked is to indicate this gap between biconnected components.

After the second DFS each cut vertex can be determined by the edge markings of the tree edges connecting it to its children. For a space-efficient implementation, the first DFS can be taken from [11]. The second one with making the markings is described in the next sections and depends on the used working space.

3.1 Cut Vertices with $O(n + m)$ bits

We use static space allocation to address $O(\deg(v))$ bits for each vertex v of degree $\deg(v)$. Within these bits, we store for every edge $\{u, v\}$ adjacent to v if (1) it is a tree or back edge, (2) u or v is closer to the root of the DFS tree, and (3) its markings in case it is a tree edge. Additionally, using $O(\log deg(v))$ bits, when a vertex v is encountered by the DFS for the first time, we store the position of the tree edge $\{u, v\}$ in the adjacency array of v, where u denotes the parent of v in the DFS tree. This information can later be used, when the DFS retreats from v, such that we can perform the DFS without explicitly having to store the DFS stack. (This idea to obtain a DFS in $O(n + m)$ time using $O(n + m)$ bits was already described by Hagerup et al. [13].) To bound the time of marking the tree edges during the second DFS by $O(n+m)$, we perform the following steps. Whenever we encounter a vertex u for the first time during the second DFS via a tree edge, we scan its entire adjacency array for back edges that lead to descendants w of u. For each such back edge $\{u, w\}$, we perform the following substep. As long as there is a tree edge from w to its parent $v \neq u$ that is not full marked, we mark $\{v, w\}$ as full marked and continue with v becoming the new w. If we encounter an edge $\{v, w\}$ that is already marked as full marked, we terminate the substep without marking any more edges. If at some point the parent v of w becomes u, we mark $\{v, w\}$ as half marked if it has not been full marked, yet, and terminate the substep thereafter.

It is easy to see that this procedure results in the correct markings for every edge. The order in which back edges are worked on assures whenever a tree edge $\{v, w\}$ is already full marked so are all tree edges between v and the child of u. Because we store, for each vertex, the position of the edge that connects it with its parent in T, the time of each substep is at most the number of tree edges that are marked plus additional constant time. Since each tree edge is marked at most twice, once as half marked and once as full marked, and the number of back edges is at most m, altogether we use $O(m)$ time.

3.2 Cut Vertices with $O(n \log \log n)$ bits

The key ideas to make the algorithm more space-efficient are to perform the classification of edges as tree edges or back edges on the fly whenever an edge is explored by the DFS, to use a second stack $\mathcal{U}$, and to apply the stack restoration technique by Elmasry et al. [11] to both stacks. We thus reconsider that paper. To obtain a DFS with a linear running time using $O(n \log \log n)$ bits, the stack is partitioned into $O(\log n)$ segments of $\Theta(n/\log n)$ entries, each consisting basically of a vertex. During the DFS, only the $O(1)$ latest segments are kept in the working memory; the remaining are thrown away. Whenever a segment that was thrown away is needed, a restoration recovers it in a time linear in the number of vertices of the segment. To restore a segment in a time linear to its size, the vertices of the ith segment have a *hue* (value) i. For a slight modification of that algorithm, one can easily see that it is not important that the segments consist of $\Omega(n/\log n)$ vertices stored in the stack, as long as their number is bounded by $O(\log n)$ and the vertices with the same hue form a connected subsequence of the sequence of elements in the stack. Therefore, we build segments not based on the entries of the stack; instead, we define the first $\Theta(n/\log n)$ vertices that are visited by the DFS as the first segment, the next $\Theta(n/\log n)$ vertices as the next segment, etc. The hue values are determined via an extra DFS at the beginning. They are stored in addition to the colors white, gray and black that are used during a "standard" DFS. The space consumption of the algorithm sums up to $O(n)$ for the segments plus $O(n \log \log n)$ bits for the hue.

A normal DFS stack $\mathcal{S}$ contains at any moment during the execution of a DFS the vertices on the path within T from its root to the vertex that is currently explored, which are all the vertices that are currently gray. The depth dp_w of a vertex w is the number of edges connecting the vertices on the path from w to the root r of T that consists solely of tree edges. The second stack $\mathcal{U}$ contains those gray vertices u that have a gray child v such that $e_u = \{u, v\}$ is not full marked. Hence, the vertices in $\mathcal{U}$ denote a subset of the vertices in $\mathcal{S}$. More exactly, each entry of $\mathcal{U}$ is a tuple of a vertex u and its depth within T. When v is explored, v is pushed onto $\mathcal{S}$ and its parent u (together with its depth) is pushed onto $\mathcal{U}$. When the DFS retreats from v, it is popped from $\mathcal{S}$ and u is popped from $\mathcal{U}$ if it is still present in $\mathcal{U}$. A second way for u to be removed from $\mathcal{U}$ is when the edge e_u becomes full marked.

Let w denote the vertex at the top of $\mathcal{S}$. Whenever the DFS tries to explore a vertex u that is gray, then $\{w, u\}$ is a back edge. Under the assumption that we know the depth of every vertex, we can mark edges as follows: While there is a vertex u' on the top of $\mathcal{U}$ whose depth is higher than dp_u, we full mark the edge $e_{u'} = \{u', v'\}$ that connects u' with its gray child v' and pop u' from $\mathcal{U}$. This loop stops if either u becomes the vertex at the top of $\mathcal{U}$ or a vertex with a lower depth than u becomes the top vertex of $\mathcal{U}$. In the first case, we half mark e_u. In the second case, we do not mark the edge $\{u, v\}$ that connects u with its gray child v since u is not on $\mathcal{U}$ because the edge $\{u, v\}$ must have been full marked during the processing of a previous back edge.

We now discuss the computation of cut vertices. When retreating from a vertex v to its parent u that is not the root of T, we check if $\{u, v\}$ is half marked or unmarked. If that is the case, we output u as a cut vertex. For the root vertex r, we maintain a counter that indicates if at least two children of r have been explored during the DFS. If so, r is outputted. Using a bit vector over the vertices, we can avoid outputting a cut vertex more than once.

For the implementation of the algorithm, the remaining problems are maintaining both stacks $\mathcal{S}$ and $\mathcal{U}$ as well as determining the depth of u, whenever processing a back edge $\{w, u\}$ with u being the ancestor of w in the DFS tree T. For the first problem, we store for every vertex its hue and use the stack restoration techniques introduced by Elmasry et al. [11], but

use it with the modified size as described above. Restorations of segments in $\mathcal{S}$ and $\mathcal{U}$ are performed independently of each other.

The second problem is more complicated and considered now. Let k be the hue of vertices that we currently process, and let Z be the set consisting of every hue $i \neq k$ that is present in $\mathcal{U}$. Our goal is to store the depth of all vertices in $\mathcal{U}$ with a hue in $\max(Z) \cup \{k\}$ such that it can be addressed by the vertex. The idea is to use one array A addressed with static space allocation to store the depth for all vertices in $\mathcal{U}$ of one segment, i.e., one hue. This allows us to build and destroy the arrays for each hue independently. However, the rank-select structure used by the static space allocation is to slow in its construction since we want a running time of $O(n/\log n)$. Therefore, we define *blocks* where block $i \in \{0, \ldots, \lceil n/\lceil(\log n)/2\rceil\rceil - 1\}$ consist of the vertices $1 + i\lceil(\log n)/2\rceil, \ldots, (i+1)\lceil(\log n)/2\rceil$—the last block may be smaller. In an auxiliary array B addressed with static space allocation, for each non-empty block b, we store a pointer to the first entry in A for a vertex in b. Within each block we use table lookup to find the exact position in A where the depth of a vertex is stored. This allows us to store the depth of the vertices in the upcoming segment in an array with static space allocation. We also restore the depth of the vertices of a segment in $\mathcal{U}$ whenever we restore it.

When processing a back edge $\{u, w\}$ (w having a hue k) with u being the ancestor of w and having a hue i there are two possibilities: If $i \in \max(Z) \cup \{k\}$, then the depth of $\mathcal{U}$ can be determined in constant time. Otherwise, $i < j = \max(Z)$ and, we iteratively restore those segments in $\{j-1, j-2, \ldots, i\}$ that have a vertex in $\mathcal{U}$ and process the vertices within these segments that are present in $\mathcal{U}$ as described above until we finally restore the segment of vertices with hue i together with their depth.

We summarize the space bound as follows. For every vertex we store in $O(n)$ bits its current color (white, gray, black), and if it yet has been outputted as a cut vertex. The set Z can be easily maintained with $O((\log n)^2)$ bits. Using $O(n \log\log n)$ bits we can store for every vertex its hue. We use additional $O(n)$ bits to store a constant number of stack segments of $O(n/\log n)$ vertices each. Since the depths of those vertices are stored compactly in arrays, each such array A together with its auxiliary array B can be implemented with $O(n)$ bits. Moreover, we use two bits for each vertex u on the stack to store if the edge connecting u and its child on the stack, if any, has been half or fully marked. Thus, the overall space bound is $O(n \log\log n)$ bits.

We finally determine the time bounds. As analyzed in [11], the DFS including the stack restorations of $\mathcal{S}$, but without the extra computations for the back edges runs in $O(n+m)$ time as do the intermediate computations in total. Assume for the moment, that we do not throw away and restore segments of $\mathcal{U}$. Then, the total time to mark the tree edges due to the back edges is $O(n+m')$ where m' is the number of back edges since each tree edge is marked at most twice using the stack $\mathcal{U}$. The tests if a vertex is a cut vertex can be performed in total time $O(m)$. It remains to bound the time for the restorations of $\mathcal{U}$. Whenever we restore a segment of $\mathcal{U}$, a hue value is removed from the set Z and never returns. This means that we have only $O(\log n)$ restorations of a segment of $\mathcal{U}$, which can be done in total time $O(n)$.

Combining the algorithms of Section 3.1 and this section, we obtain our first theorem.

▶ **Theorem 4.** *There is an algorithm that, given an n-vertex m-edge graph G in an adjacency array representation with cross pointers, runs in time $O(n+m)$ and uses $O(n + \min\{m, n\log\log n\})$ bits of working space, and determines the cut vertices of G.*

3.3 Biconnected Components

We next show that we can compute the biconnected components of an undirected graph. Recall that a graph is biconnected if it is connected and has no cut vertices. A biconnected component of a graph G is a maximal biconnected induced subgraph of G. Whenever we say in the following theorem and proof that we output an edge $\{x, y\}$, then we mean that we output the index of the edge in the adjacency array of both x and y.

▶ **Theorem 5.** *There is a data structure that, given an n-vertex m-edge graph G in an adjacency array representation with cross pointers, runs $O(n+m)$ initialization time and uses $O(n+m)$ bits of working space and that afterwards, given an edge e, computes the vertices and/or the edges of the biconnected component B of G with B containing e in a time that is linear in the number of vertices and edges that are output.*

Proof. To initialize the data structure, first run our algorithm from Section 3.1 to compute a DFS tree T with a root r, for each edge $\{u, v\}$ the ancestor-descendant relationship in T between u and v as well as the markings of the tree edges. Also store for each vertex $v \neq r$ the index of its edge connecting v to its parent. Then build a rank-select data structure for each vertex v that allows us to iterate over the fully marked tree edges and the back edges to ancestors of v in $O(1)$ time per edge. It is easy to see that the initialization runs in $O(n+m)$ time and all information can be stored with $O(n+m)$ bits using static-space allocation.

Afterwards, given an edge $e = \{u, v\}$ with u being an ancestor of v, we can output the biconnected component B containing e by running a DFS from v and traversing only tree edges (to both directions, parent and children) such that we never explore new vertices from a vertex that was reached by a half marked or unmarked edge and such that we never use a half marked or unmarked edge moving from a parent to its child via such an edge. During the DFS, we output all visited vertices as well as all traversed tree edges. Since a back edge $\{x, y\}$ with x ancestor of y always belongs to the same biconnected component as y, we output $\{x, y\}$ only if we visit y. In this case, we output the edge as an edge of y and, using cross pointers, also of x. The algorithm can easily be modified such that it only outputs the vertices/edges of the biconnected component. Using the rank-select data structures, this can be done in the time stated in the lemma.

To see that each outputted B is indeed a biconnected component observe that B is connected. Assume that B has a cut vertex v with a child u that cuts off the subtree T_u with root u, and we want to output the component containing the edge connecting v and its parent. In that case the edge $\{v, u\}$ is half marked or not marked. Hence, such an edge is not used to go from a parent to a child. On the other hand, if we want to output a component B for which a vertex v is a cut vertex disconnecting B and r, then the edge connecting v with the rest of B is half marked. Hence, we output v, but we do not visit other vertices. Finally, B is maximal by construction since all tree edges that are in B are fully marked except for the "highest" edge in T that is half marked; thus all vertices of B are found by the DFS. ◀

4 Outerplanar Graphs

Outerplanar graphs and maximal outerplanar graphs are well-studied subclasses of planar graphs. For their structural properties we refer to the monograph of Brandstädt et al. [6].

Given a biconnected outerplanar graph $G = (V, E)$, we call an edge that is incident to the outer face an *outer edge* and an edge that is incident to two inner faces an *inner edge*. We now describe a set of well-known properties for outerplanar graphs that help us to describe and prove our algorithm of Section 4.1. Every maximal outerplanar graph with at least

three vertices is biconnected and, for every biconnected outerplanar graph G, the set of outer edges induce a unique Hamiltonian cycle that contains all vertices of G. Every biconnected outerplanar graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ and $|E| > n$ contains at least one inner edge. Let the vertices of G be labeled according to their position on the Hamiltonian cycle of G, by $1, \ldots, n$, and let $\{u, v\}$ denote an inner edge that connects the vertices u and v. W.l.o.g., let $u < v$. Then the graph $G' = G[\{u, \ldots, v\}]$ is biconnected and outerplanar. Since $\{u, v\}$ is an inner edge, $1 < v - u < n - 1$ holds and there are exactly $v - u - 1 > 0$ vertices between u and v on the path part of the Hamiltonian cycle that belongs to G'. This path together with the edge $\{v, u\}$ forms the Hamiltonian cycle of G'.

Usually, one decomposes an outerplanar graph by repeatedly removing a vertex v of degree 2. However, this needs to test if the neighbors of v are connected by an edge and, if not, to add such an edge. Because the test is too time consuming and storing all such edges needs too much space, we search instead for a closed or good chain defined next.

We define a *chain* in an outerplanar graph G as either a cycle that consists solely of vertices of degree 2 or a path that contains at least three pairwise distinct vertices with the property that its first and its last vertex have a degree larger than 2 while the rest must have degree 2. We denote the first and the last vertex of a chain as its *endpoints* unless the chain C is a cycle, in which case the endpoints can be chosen arbitrarily as long as they are adjacent to each other. Furthermore, we call a cycle a *loop* if it contains one vertex of degree larger than 2 and all other vertices have a degree 2. A chain is called a *good chain* if one of its endpoints has a degree of at most 4. Let us call a face F *induced* by a chain C if the endpoints u and v of C are adjacent to each other and C together with the edge $\{u, v\}$ is the boundary of F. We denote a chain C that induces a face F as a *closed chain*. For simplicity, we sometimes consider a chain also as a set of edges.

▶ **Lemma 6.** *Let $G = (\{1, \ldots, n\}, E)$ denote a biconnected outerplanar graph with $n \geq 4$ vertices. Then G contains a good closed chain C.*

The next two lemmas are used to show the correctness of our algorithm.

▶ **Lemma 7.** *Let $G = (V, E)$ be a biconnected outerplanar graph, then the following properties hold:*

- **(i)** *For every chain C in G with endpoints v_1 and v_2, the graph $G' := G[(E \setminus C) \cup \{\{v_1, v_2\}\}]$ induced by the edge set $(E \setminus C) \cup \{\{v_1, v_2\}\}$ is biconnected and outerplanar.*
- **(ii)** *For all $v_1, v_2 \in V$, there are at most two internal vertex-disjoint paths with at least two edges each.*

Proof. To prove the first part note that G' is a minor of G. Thus, G' is outerplanar. For all vertices u_1, u_2 of G', there are two internal vertex-disjoint paths in G. Since C is a chain, it can only be a complete part of such a path and therefore replaced by the edge $\{v_1, v_2\}$. It follows there are two internal vertex-disjoint paths between u_1 and u_2 in G' and G' is biconnected. We prove the second part by contradiction. Assume that Prop. ii does not hold. Then $K_{2,3}$ is a minor of G. This is a contradiction to G being outerplanar. ◀

▶ **Lemma 8.** *Let $G = (V, E)$ be a graph for which the following properties hold.*

- **(1)** *There is a chain C in G with endpoints v_1 and v_2 such that the graph $G' := G[(E \setminus C) \cup \{\{v_1, v_2\}\}]$ is biconnected and outerplanar.*
- **(2)** *There are at most two internal vertex-disjoint paths with at least 2 edges each in G that connect v_1 and v_2.*

Then G is biconnected and outerplanar.

Proof. The existence of a chain follows from Lemma 6. G' is biconnected outerplanar because of Prop. 1. Because of Prop. 2 there is at most one internal vertex-disjoint path connecting v_1 and v_2 in $G[E \setminus C]$. Hence, the edge $\{v_1, v_2\}$ is part of the outer face of G'. It follows that we can embed C in the outer face of an embedding of G' to yield an outerplanar embedding of G. Because $\{v_1, v_2\}$ is part of the outer face of G', it is part of the Hamiltonian cycle of G'. We can extend the Hamiltonian cycle of G' by replacing $\{v_1, v_2\}$ with C to get a Hamiltonian cycle of G. Thus G is biconnected. ◀

4.1 Our Algorithm on Biconnected Outerplanar Graphs

For the time being, we assume that the given graph is biconnected. Our algorithm works in two phases and can be sketched as follows. Before our actual algorithm starts we test whether $m \leq 2n - 3$. If not, the graph is not outerplanar and we terminate immediately. Otherwise we start our algorithm that modifies the input graph into a smaller outerplanar graph as described in Lemma 8 Prop. 1 while checking Prop. 2 of Lemma 8. Lemma 7 guarantees that it does not matter which chain we take, the modification is always possible and that the check never fails unless G is not biconnected outerplanar. A main obstacle is to handle the edges replacing the chains that we call subsequently *artificial edges*.

To check Prop. 2 of Lemma 8, we keep in both phases counters P_e for every (original or artificial) edge e to count the number of internal vertex-disjoint paths with at least 2 edges that connect the endpoints of e. Whenever we remove a chain with both endpoints in e, we increment P_e. If P_e at any time exceeds 2, the graph can not be outerplanar by Lemma 7 (ii). We check this after every incrementation of an counter P_e and terminate eventually.

We start to sketch the two phases of our algorithm. The details of the phases are given subsequently. Let $G = (V, E)$ denote the input graph, which is located in read-only input space, and G' denote the subgraph of G that we are currently considering in our algorithm. Initially, $G' := G$. Thereafter, within our algorithm the edges of chains are either removed from G' if they induce a face or replaced by artificial edges that connect the endpoints of the chain directly. Consequently, G' is always a minor of G.

The purpose of the first phase is to limit the number of artificial edges that are required for the second phase to $O(n/\log n)$. The first phase consists of $\Theta(\log \log n)$ *rounds*, in each of which we iterate over all chains, but only remove the closed ones.

In the second phase, we repeatedly take a vertex of degree 2 and determine its chain. Depending on the kind of the chain, we proceed: Good closed chains are removed from G'. The edges of chains that turn out to be good, but not closed are replaced by an artificial edge connecting its endpoints. The counter of vertex-disjoint paths with the same endpoints $\{u, v\}$ for a newly created artificial edge $\{u, v\}$ is initialized with 1 to account for the chain that has been replaced by $\{u, v\}$. When processing a chain C that turns out to be not good, we implement a *shortcut* that is a pair of pointers, each one addressed by the vertex of degree 2 in C that is next to a endpoint of C and pointing to the vertex of degree 2 in C that is adjacent to the other endpoint. This way, when the degree of either one of the endpoints is lowered to 2 by the removal of adjacent chains and thus C becomes connected with an other chain, C does not have to be traversed again to check if the new chain is good.

If the input graph is outerplanar, the algorithm terminates either in Phase 1 or in Phase 2 as soon as the last good chain, which is a cycle, is processed and a single edge that is the edge between its endpoints remains. Otherwise, if the input graph is not outerplanar, at some moment one of the checks fails and the algorithm stops and answers that the input is not biconnected outerplanar. Possible conditions for a failed check are if no good chain remains and the graph has at least vertices, if the counter that counts the internal vertex-disjoint

paths with at least 2 edges between the endpoints of an edge exceeds 2 for some edge, if some loop is detected, or if a vertex turns out to be incident to at least three vertices of degree 2.

To lower the time that is required to iterate through the adjacency array of a vertex v, we initialize, for each vertex $v \in V$ with degree $deg_G(v)$ in G, a choice dictionary with universe $\{1, \ldots, deg_G(v)\}$ that represents the edges that are adjacent to v and still present within the current subgraph. The choice dictionary contains i with $i \in \{1, \ldots, deg_G(v)\}$ exactly if the ith entry of the adjacency array of v represents an edge of the input graph G that is still present in the current graph G'. Thus, future iterations through the adjacency array of v can be performed within a time that is linear in the number of edges that are incident on v in G'. Hence, the time to determine the adjacency of two endpoints v and w of a chain is bound by $O(\min\{deg_{G'}(v), deg_{G'}(w)\})$.

Phase 1: At the beginning of each round we unmark all vertices and insert all vertices of degree 2 of the current graph G' into a choice dictionary D.

For a more detailed description we subdivide the $\Theta(\log \log n)$ rounds into stages. As long as there is a vertex u in D, we extract it from D and start a new stage. Let C denote the chain that u belongs to. A stage works as follows. If C contains a vertex that is marked as *tried* or C is not closed, the stage stops. Otherwise, we increment the counter P_e for every edge e on C as well as for the edge connecting its endpoints and remove the vertices and edges of C from the graph. If the current chain C is not the first chain processed in the current stage so far, we mark every vertex of degree 2 that has been processed within the current stage and still remains in the graph as *tried*. If one or both of the endpoints of C thereby become a vertex of degree 2, the procedure is repeated immediately for the new chain C' that incorporates both endpoints of C. The stage ends if no new chain is found. At the end of the stage we remove the vertices of degree 2 that have been processed within the current stage from D. A round ends if D is empty.

Phase 2: We start to initialize a choice dictionary D consisting of one vertex of degree 2 for each good closed chain. (During Phase 2, vertices of other chains may be added into D.) Take q as the initial number of vertices in D. As we prove in Corollary 11, the number of chains that induce a face is at most $O(n/\log n)$ after Phase 1 unless G is not outerplanar. If G' contains more such chains, we can terminate immediately. Otherwise, we can store a constant number of artificial edges and shortcuts for each such chain. Whenever we process a vertex u from D, we determine the chain C that u is part of, the endpoints v and w of C, and then distinguish three cases.

- If C is good and closed, we increment the counter P_e for every edge e that is part of C and for the edge that connects its endpoints. Finally, we remove the edges of C from G'.
- If C is good and not closed, we insert an artificial edge e_a that connects the endpoints of C, increment the counter P_e for every edge e in C by 1 and remove all the edges and inner vertices of C from G'. Finally, the counter P_{e_a} of e_a is initialized with 1.
- Otherwise, C is not good. We remove all vertices of C from D and insert a shortcut between the vertices of degree 2 in C that are adjacent to the endpoints of C. If there are old shortcuts connecting other inner vertices of C, they become obsolete and are removed.

Whenever the degree of v or w decreases, we perform the following subroutine. If $deg_{G'}(v) \in \{3, 4\}$, we insert all vertices u of degree 2 that are adjacent to v into D. Note that u and w are neighbors of v on the unique Hamiltonian Cycle of G'. Thus, if G is biconnected outerplanar, there is at most one such vertex u. If $deg_{G'}(v) = 2$, we insert v into D. We proceed with w analogously. We so guarantee that each good closed chain of G' has a vertex

in D. As shown by Lemma 12, if the number of artificial edges and shortcuts that are simultaneously in use exceeds $2q$, G is not outerplanar and we stop.

Since the removal of chains is performed in accordance with Lemma 7 and Lemma 8, to show the correctness of the algorithm, it remains to verify that the checks on the counters P_e are correct and sufficient.

▶ **Lemma 9.** *The counter P_e counts for any edge $e = \{v_1, v_2\}$ that belongs at some moment during our algorithm to the current graph G' the number of internal vertex-disjoint paths with at least 2 edges between v_1 and v_2 that have been removed from G so far.*

By our counters P_e for all original and artificial edges, we count the number of internal vertex-disjoint paths between vertices that have been endpoints of a removed chain. Thus, our tests are sufficient by Lemma 8. By Lemma 7 (ii) the counting of the internal vertex-disjoint paths between the endpoints of other edges and terminating if one of these counters exceeds 2 is correct.

4.2 Space-Efficient Implementation and Space Bounds

Before any allocation of space is performed the first check of the algorithm is to verify that the number m of edges within the input graph G is at most $2n - 3$, where n denotes the number of vertices in G. From now on, we assume that $m = O(n)$. During Phase 1 we use a bit vector of n bits that allows us to mark vertices as *tried*.

The current graph G' is represented as follows. We use a choice dictionary of O(n) bits, where i is in the set represented by the choice dictionary exactly if the vertex i is still part of G'. In addition, for each vertex v of initial degree $deg_G(v)$, we store $\Theta(deg_G(v))$ bits. Within this space we use a choice dictionary C_v with universe $\{1, \ldots, deg_G(v)\}$ as already described above and a counter that maintains the current degree of v in G'. It follows that, ignoring artificial edges and shortcuts, a representation of the current graph G' with one choice dictionary for each vertex fits in $O(n)$ bits of working space.

We next want to bound the number of artificial edges and shortcuts. We start to bound the number of chains that induce a face after Phase 1. For this purpose, let us define the *dual tree* T_G of a biconnected and outerplanar graph G as the dual graph of G minus the vertex that represents the outer face of G. Since G is biconnected and outerplanar, T_G is a tree. The leaves of T_G correspond to those faces of G that are induced by chains. Thus, the removal of a closed chain C that induces a face F in G', which results in the merging of F with the outer face, corresponds to the removal of the leaf F in $T_{G'}$.

▶ **Lemma 10.** *If the input graph G with n vertices is biconnected outerplanar, then the number of chains that induce a face in the current graph G' after the tth round of Phase 1 is at most $n/2^t$.*

Proof. It is easy to see that the initial number of chains that induce a face before the first round is at most n. Recall that, in each stage of Phase 1, we consider a vertex u of degree 2 and test whether it is part of a closed chain C within our current graph G'. If so, we remove the chain C from G' and thereafter continue recursively on one endpoint v of C if the removal of C results in v becoming a vertex of degree 2. Let us analyze the modifications of the algorithm in the dual tree. Whenever we remove a chain, this means that we remove a leaf and then recursively try if its parent thereby has become a leaf and can be removed as well until a node of degree at least 2 is encountered.

Vertices of a chain C that is incident to a face F are marked as *tried* if F could not be merged with the outer face after a face that was incident to F has been successfully merged

with the outer face by the removal of the chain that induced it. The removal of F fails only if F is not induced by C and there remain at least two faces in $T_{G'}$ that are incident on F. We conclude that F had degree at least 3 at the beginning of the current round. Since in every round leaves are removed until a node is encountered that had at the beginning of the round a degree of at least 3, the number of leaves is at least halved in every round. ◀

▶ **Corollary 11.** *If the input graph G with n vertices is biconnected outerplanar, the number of closed chains after Phase 1 in the current graph G' is $O(n/\log n)$.*

We next bound the number of artificial edges and shortcuts.

▶ **Lemma 12.** *The initial number q of chains at the beginning of Round 2 that have vertices in D only doubles during Phase 2, and the number of artificial edges and shortcuts that are simultaneously in use by the algorithm is $2q = O(n/\log n)$.*

Proof. By the corollary above, $q = O(n/\log n)$ at the beginning of Phase 2. One may consider these chains cutting the Hamilton cycle of G' into q parts. Since we add new vertices into D at the end of Phase 2 only if we were in Case 1 before, a new chain C' is always neighbored to an old chain C, which is then deleted from G' before a vertex of C' is added into D. Roughly speaking, the chains at the beginning of Phase 2 are fires that can spread to new chains on the left and on the right along the Hamilton cycle, but the fire can never return and ends immediately at the moment when a new chain starts to burn. It is easy to see that at most $2q$ chains are on fire. If we define that a chain that is processed with Case 2 or 3 is still on fire, then we have to store artificial edges or shortcuts only for chains that are on fire, and thus, their number is bounded by $2q = O(n/\log n)$. ◀

We can use a bit vector of $O(n)$ bits to store for every vertex v the information whether an artificial edge or shortcut exists at v or not. As a consequence of the last lemma, we can store all artificial edges as well as all shortcuts in a ragged dictionary and the total space bound of the algorithm is $O(n)$ bits.

▶ **Lemma 13.** *There is an algorithm that, given an n-vertex biconnected graph G in an adjacency array representation with cross pointers, runs in $O(n \log\log n)$ time and uses $O(n)$ bits of working space, and determines whether G is biconnected outerplanar.*

4.3 Algorithm for General Outerplanar Graphs

We next sketch the generalization of our recognition algorithm of biconnected outerplanar graphs to general outerplanar graphs. Since a graph is outerplanar exactly if all of its biconnected components are outerplanar, we can iterate over the biconnected components of a given graph G using our framework of Section 3.3—to avoid using $\Omega(m)$ time or bits if a non-outerplanar, dense graph is given, one should initially check that $m \leq 2n - 3$. For each biconnected component, we first stream all its vertices and build an initially empty choice dictionary C_v with $\deg_G(v)$ keys for each such vertex v, then stream the edges of the biconnected component and fill in the indices of each edge in the choice dictionaries of its endpoints. In addition, we compute and store for each vertex of the biconnected component its degree.

Recall that our subgraph G' is represented by a choice dictionary that contains those vertices of G that are still part of G' and a choice dictionary C_v for every vertex v such that $i \in \{1, \ldots, deg_G(v)\}$ is present in C_v exactly if the ith entry of the adjacency array of v contains an edge that is still present G'. We initialize these choice dictionaries with the

values that are streamed from the algorithm that determines biconnected components to initialize a representation of a biconnected subgraph. The time to test if the subgraph is biconnected outerplanar is bound by the number of vertices and edges of the subgraph alone.

Finally note that a check if a outerplanar graph $G = (V, E)$ is maximal outerplanar can be easily performed by checking if $2|V| - 3 = |E|$.

▶ **Theorem 14.** *There is an algorithm that, given an n-vertex graph G in an adjacency array representation with cross pointers, runs in time $O(n \log \log n)$ and uses $O(n)$ bits of working space, and determines if G is (maximal) outerplanar.*

References

1. Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 47(3, Part B):469–479, 2014. doi:10.1016/j.comgeo.2013.11.004.
2. Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using $O(n)$ bits. In *Proc. 25th International Symposium on Algorithms and Computation (ISAAC 2014)*, volume 8889 of *LNCS*, pages 553–564. Springer, 2014. doi:10.1007/978-3-319-13075-0_44.
3. Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *J. Comput. Geom.*, 2(1):46–68, 2011.
4. Luis Barba, Matias Korman, Stefan Langerman, Rodrigo I. Silveira, and Kunihiko Sadakane. Space-time trade-offs for stack-based algorithms. In *Proc. 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *LIPIcs*, pages 281–292. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPIcs.STACS.2013.281.
5. Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. doi:10.1137/0220017.
6. A. Brandstädt, V. Le, and J. Spinrad. *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, 1999. doi:10.1137/1.9780898719796.
7. David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
9. Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.
10. Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for *st*-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6):2257–2284, 1999. doi:10.1137/S0097539795295948.
11. Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient basic graph algorithms. In *Proc. 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.288.
12. Torben Hagerup and Frank Kammer. Succinct choice dictionaries. *Computing Research Repository (CoRR)*, arXiv:1604.06058 [cs.DS], 2016.
13. Torben Hagerup, Frank Kammer, and Moritz Laudahn. Space-efficient euler partition and bipartite edge coloring. Talk given on the theory days of computer science of the Gesellschaft für Informatik in Speyer, Germany, 2015.

14 Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-efficient biconnected components and recognition of outerplanar graphs. *Computing Research Repository (CoRR)*, arXiv:1606.04679 [cs.DS], 2016.

15 Sandra L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Inf. Process. Lett.*, 9(5):229–232, 1979. doi:10.1016/0020-0190(79)90075-9.

16 J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12(3):315–323, 1980. doi:10.1016/0304-3975(80)90061-4.

17 Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1998)*, pages 264–268. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743455.

18 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, September 2008. doi:10.1145/1391289.1391291.

19 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

20 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

21 Manfred Wiegers. Recognizing outerplanar graphs in linear time. In *Proc. International Workshop on Graphtheoretic Concepts in Computer Science (WG 1986)*, volume 246 of *LNCS*, pages 165–176. Springer, 1986. doi:10.1007/3-540-17218-1_57.