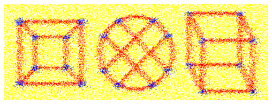
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 8, no. 1, pp. 47–87 (2004)

I/O-Optimal Algorithms for Outerplanar Graphs

Anil Maheshwari

School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada

Norbert Zeh

Faculty of Computer Science
Dalhousie University
6050 University Ave
Halifax, NS B3H 1W5, Canada

Abstract

We present linear-I/O algorithms for fundamental graph problems on embedded outerplanar graphs. We show that breadth-first search, depth-first search, single-source shortest paths, triangulation, and computing an ϵ-separator of size $O(1/\epsilon)$ take $O(\text{scan}(N))$ I/Os on embedded outerplanar graphs. We also show that it takes $O(\text{sort}(N))$ I/Os to test whether a given graph is outerplanar and to compute an outerplanar embedding of an outerplanar graph, thereby providing $O(\text{sort}(N))$-I/O algorithms for the above problems if no embedding of the graph is given. As all these problems have linear-time algorithms in internal memory, a simple simulation technique can be used to improve the I/O-complexity of our algorithms from $O(\text{sort}(N))$ to $O(\text{perm}(N))$. We prove matching lower bounds for embedding, breadth-first search, depth-first search, and single-source shortest paths if no embedding is given. Our algorithms for the above problems use a simple linear-I/O time-forward processing algorithm for rooted trees whose vertices are stored in preorder.

Article Type	Communicated by	Submitted	Revised
regular paper	M. T. Goodrich	October 2001	April 2004

Research supported by NSERC. A preliminary version of this paper was presented at ISAAC'99 [22].

1 Introduction

1.1 Motivation

External-memory graph algorithms have received considerable attention because massive graphs arise naturally in many large scale applications. Recent web crawls, for instance, produce graphs consisting of 200 million nodes and 2 billion edges. Recent work in web modeling investigates the structure of the web using depth-first search (DFS) and breadth-first search (BFS) and by computing shortest paths and connected components of the web graph. Massive graphs are also often manipulated in geographic information systems (GIS). In these applications, the graphs are derived from geometric structures; so they are often planar or even outerplanar. Yet another example of a massive graph is AT&T's 20TB phone call graph [8]. When working with such large data sets, the transfer of data between internal and external memory, and not the internal-memory computation, is often the bottleneck. Hence, I/O-efficient algorithms can lead to considerable runtime improvements.

Breadth-first search (BFS), depth-first search (DFS), single-source shortest paths (SSSP), and computing small separators are among the most fundamental problems in algorithmic graph theory; many graph algorithms use them as primitive operations to investigate the structure of the given graph. While I/O-efficient algorithms for BFS and DFS in dense graphs have been obtained [7, 9, 25], no I/O-efficient algorithms for these problems on general sparse graphs are known. In the above applications, however, the graphs are usually sparse. This is true in GIS applications due to the planarity of the graph. For web graphs and phone call graphs, sparseness is implied by the locality of references; that is, usually every node has a rather small neighborhood to which it is connected by direct links.

In this paper, we exploit the structure exhibited by outerplanar graphs to develop I/O-efficient algorithms for graph problems that are hard on general sparse graphs. Recently, a number of papers address the problems solved in this paper for the more general classes of planar graphs [3, 4, 19, 24] and graphs of bounded treewidth [23]. These algorithms can be applied to solve these problems on outerplanar graphs, as outerplanar graphs are planar and have treewidth at most two. However, the algorithms presented here are much simpler, considerably more efficient, and quite elegant. Moreover, given an embedding of the graph (represented appropriately), we can solve all problems in this paper in $O(\text{scan}(N))$ I/Os, while the algorithms of [3, 4, 19, 23, 24] perform $O(\text{sort}(N))$ I/Os.

Outerplanar graphs, though simple, have been found to have important applications for shortest-path computations in planar graphs and for network routing problems (e.g., see [14, 15, 16, 17, 28]). They can be seen as combinatorial representations of triangulated simple polygons and their subgraphs. An efficient separator algorithm for outerplanar graphs can for instance be used to develop divide-and-conquer algorithms for simple polygons. Finally, every outerplanar graph is also planar. Thus, any lower bound that we can show for

outerplanar graphs also holds for planar graphs.

In [33], an external-memory version of a shortest-path data structure for planar graphs by Djidjev [13] has been proposed. Given the improved algorithms for outerplanar graphs presented in this paper, this data structure can be constructed more efficiently for these graphs. Because of the constant size of the separator for outerplanar graphs, the space requirements of this data structure reduce to $O\left(\frac{N}{B}\log_2 N\right)$ blocks as opposed to $O\left(\frac{N^{3/2}}{B}\right)$ blocks; the query complexity becomes $O\left(\frac{\log_2 N}{DB}\right)$ I/Os as opposed to $O\left(\frac{\sqrt{N}}{DB}\right)$ I/Os for planar graphs.

1.2 Model of Computation

When the data set to be handled becomes too large to fit into the main memory of the computer, the transfer of data between fast internal memory and slow external memory (disks) becomes a significant bottleneck. Existing internal-memory algorithms usually access their data in a random fashion, thereby causing significantly more I/O-operations than necessary. Our goal in this paper is to minimize the number of I/O-operations performed by our algorithms. Several computational models for estimating the I/O-efficiency of algorithms have been developed [1, 2, 11, 12, 18, 30, 31]. We adopt the *parallel disk model* (PDM) [30] as our model of computation in this paper because it is simple and our algorithms are sequential.

In the PDM, an *external memory*, consisting of D disks, is attached to a machine whose main memory is capable of holding M data items. Each of these disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O-operation (or I/O). The complexity of an algorithm is the number of I/O-operations it performs.

Sorting, permuting, and scanning a sequence of N consecutive data items are primitive operations often used in external-memory algorithms. These operations take $\mathrm{sort}(N) = \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$, $\mathrm{perm}(N) = \Theta(\min(\mathrm{sort}(N), N))$, and $\mathrm{scan}(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os, respectively [1, 29, 30].

1.3 Previous Work

For planar graphs, Lipton and Tarjan [21] present a linear-time algorithm for finding a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$. It is well-known that every outerplanar graph has a $\frac{2}{3}$-separator of size two and that such a separator can be computed in linear time. Mitchell [27] presents a linear-time algorithm for outerplanarity testing. Embedding outerplanar graphs takes linear time. There are simple linear-time algorithms for BFS and DFS in general graphs (see [10]). Frederickson [14, 15, 16] studies outerplanar graphs in the context of shortest path and network routing problems. Although these algorithms are efficient in the RAM-model, their performance deteriorates drastically in models where ran-

dom access is expensive, such as the PDM. Refer to [17] and [28] for a good exposition of outerplanar graphs.

The currently best breadth-first search algorithm for general undirected graphs takes $O\left(\sqrt{\frac{|V||E|}{B}} + \text{sort}(|V| + |E|)\right)$ I/Os [25]; for directed graphs, the best known algorithm takes $O((V + E/B)\log_2 V)$ I/Os [7]. The best DFS-algorithms for undirected and directed graphs take $O((V + E/B)\log_2 V)$ I/Os [7, 9]. The best known SSSP-algorithm for general undirected graphs takes $O(|V| + (|E|/B)\log_2 |E|)$ I/Os [20]. For undirected graphs with edge weights in the range $[w, W]$, an $O\left(\sqrt{\frac{|V||E|\log_2(W/w)}{B}} + \text{sort}(|V| + |E|)\right)$-I/O shortest-path algorithm is presented in [26]. No I/O-efficient algorithms for SSSP in general directed graphs have been obtained. Chiang et al. [9] have developed $O(\text{sort}(N))$-I/O algorithms for computing an open ear decomposition and the connected and biconnected components of a given graph $G = (V, E)$ with $|E| = O(|V|)$ and $N = |V|$. They also propose a technique, called *time-forward processing*, that allows $O(N)$ data items to be processed through a directed acyclic graph of size N; the I/O-complexity of the algorithms is $O(\text{sort}(N))$. They apply this technique to develop an $O(\text{sort}(N))$-I/O algorithm for list-ranking.

In [24], we have shown how to compute a separator of size $O\left(N/\sqrt{h}\right)$ whose removal partitions a given planar graph into $O(N/h)$ subgraphs of size at most h and boundary size at most $\sqrt{h}$. The algorithm takes $O(\text{sort}(N))$ I/Os, provided that $h\log^2(DB) \leq M$. Together with the algorithm of [3], this leads to $O(\text{sort}(N))$-I/O algorithms for single-source shortest paths and BFS in embedded undirected planar graphs. By applying the reduction algorithm of [4], it also allows the computation of a DFS-tree of an embedded undirected planar graph in $O(\text{sort}(N))$ I/Os. Recently, the shortest-path and BFS-algorithms for undirected planar graphs have been generalized to the directed case [5]. Directed planar DFS takes $O(\text{sort}(N)\log_2(N/M))$ I/Os [6].

1.4 Our Results

We present $O(\text{scan}(N))$-I/O algorithms for the following problems on embedded outerplanar graphs:

- Breadth-first search (BFS),
- Depth-first search (DFS),
- Single-source shortest paths (SSSP),
- Triangulating the graph, and
- Computing an ϵ-separator of size $O(1/\epsilon)$ for the given graph.

Clearly, these results are optimal if no additional preprocessing is allowed. We also present an $O(\text{sort}(N))$-I/O algorithm to test whether a given graph

$G = (V, E)$ is outerplanar. The algorithm provides proof for its decision by providing an outerplanar embedding $\hat{G}$ of G if G is outerplanar, or by producing a subgraph of G which is an edge-expansion of K_4 or $K_{2,3}$ if G is not outerplanar. Together with the above results, we thus obtain $O(\text{sort}(N))$-I/O algorithms for the above problems on outerplanar graphs if no embedding is given. As there are linear-time internal-memory algorithms for all of these problems, we can use a simple simulation technique to improve the I/O-complexities of our algorithms to $O(\text{perm}(N))$. We prove matching lower bounds for BFS, DFS, SSSP, and embedding. Our linear-I/O algorithms are based on a new linear-I/O time-forward processing technique for rooted trees.

1.5 Organization of the Paper

In Section 2, we present the terminology and basic results from graph theory that we use. In Section 3, we present our time-forward processing algorithm for rooted trees. In Section 4, we describe our algorithm for testing whether a given graph is outerplanar and for computing an outerplanar embedding. In Section 5, we present an algorithm to triangulate a connected embedded outerplanar graph. In Section 6, we provide an algorithm for computing separators of embedded outerplanar graphs. In Section 7, we present algorithms for computing breadth-first search, depth-first search, and single-source shortest path trees of embedded outerplanar graphs. In Section 8, we prove lower bounds for embedding, DFS, BFS, and SSSP on outerplanar graphs.

2 Preliminaries

An (*undirected*) *graph* $G = (V, E)$ is a pair of sets, V and E; V is the *vertex set* of G; E is the *edge set* of G and consists of unordered pairs $\{v, w\}$, where $v, w \in V$. A *subgraph* of G is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The *neighborhood* of v in G is defined as $\Gamma_G(v) = \{w \in V : \{v, w\} \in E\}$. We say that a vertex $w \in \Gamma_G(v)$ is *adjacent* to v; edge $\{v, w\}$ is *incident* to v and w. The *degree* of a vertex v is defined as $\deg_G(v) = |\Gamma_G(v)|$. In a *directed graph* $G = (V, E)$, the edges in E are *ordered* pairs (v, w), $v, w \in V$. We say that edge (v, w) is *directed from v to w*, v is the *source* of edge (v, w), and w is the *target*. The *in-neighborhood* of v in G is defined as $\Gamma^-_G(v) = \{u \in V : (u, v) \in E\}$; the *out-neighborhood* of v in G is defined as $\Gamma^+_G(v) = \{w \in V : (v, w) \in E\}$. The *in-degree* and *out-degree* of a vertex v in G are defined as $\deg^-_G(v) = |\Gamma^-_G(v)|$ and $\deg^+_G(v) = |\Gamma^+_G(v)|$, respectively. The *neighborhood* of v in G is $\Gamma_G(v) = \Gamma^-_G(v) \cup \Gamma^+_G(v)$. The *degree* of v is defined as $\deg_G(v) = \deg^-_G(v) + \deg^+_G(v)$. Note that $\deg_G(v) \geq |\Gamma_G(v)|$. Adjacency of vertices and incidence of edges and vertices are defined as for undirected graphs.

For an undirected graph $G = (V, E)$, we call the graph $D(G) = (V, D(E))$ with $D(E) = \{(v, w), (w, v) : \{v, w\} \in E\}$ the *directed equivalent* of G. For a directed graph $G = (V, E)$, we call the graph $U(G) = (V, U(E))$, $U(E) = \{\{v, w\} : (v, w) \in E\}$, the *undirected equivalent* of G. $U(G)$ is also called the

underlying undirected graph of G. Note that $U(D(G)) = G$, but not necessarily $D(U(G)) = G$.

A *walk* in a directed (undirected) graph G is a subgraph P of G with vertex set $\{v_0, \ldots, v_k\}$ and edge set $\{(v_{i-1}, v_i) : 1 \leq i \leq k\}$ ($\{\{v_{i-1}, v_i\} : 1 \leq i \leq k\}$). We call v_0 and v_k the *endpoints* of P and write $P = (v_0, \ldots, v_k)$. P is a *path* if all vertices in P have degree at most two. P is a *cycle* if $v_0 = v_k$. P is a *simple cycle* if it is a cycle and a path. In this case, we write $P = (v_0, \ldots, v_{k-1})$. (Note that edge $\{v_0, v_{k-1}\}$ is represented implicitly in this list.) For a path $P = (v_0, \ldots, v_k)$ and two vertices v_i and v_j, $0 \leq i \leq j \leq k$, let $P(v_i, v_j)$ be the subpath $(v_i, \ldots, v_j)$ of P.

An undirected graph is *connected* if there is a path between any pair of vertices in G. An *articulation point* or *cutpoint* of an undirected graph G is a vertex v such that $G - v$ is disconnected. An undirected graph is *biconnected* if it does not have any articulation points. A *tree* with N vertices is a connected graph with $N - 1$ edges. The *connected components* of G are the maximal connected subgraphs of G. The *biconnected components* (bicomps) of G are the maximal biconnected subgraphs of G. A graph is *planar*, if it can be drawn in the plane so that no two edges intersect except at their endpoints. Such a drawing defines an order of the edges incident to every vertex v of G counterclockwise around v. We call G *embedded* if we are given these orders for all vertices of G. $\mathbb{R}^2 \setminus (V \cup E)$ is a set of connected regions; we call these regions the *faces* of G and denote the set of faces of G by F. A graph is *outerplanar*, if it can be drawn in the plane so that there is a face that has all vertices of G on its boundary. We assume w.l.o.g. that this is the outer face of the embedding. An outerplanar graph with N vertices has at most $2N - 3$ edges. The *dual* of an embedded planar graph $G = (V, E)$ with face set F is a graph $G^* = (V^*, E^*)$, where $V^* = F$ and E^* contains an edge between two vertices in V^* if the two corresponding faces in G share an edge. For an embedded outerplanar graph G whose interior faces are triangles, we define the *dual tree* of G to be the tree obtained from the dual of G after removing the vertex f^* dual to the outer face f and all edges incident to f^*.

An *ear decomposition* $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ of a biconnected graph G is a decomposition of G into paths $P_0, \ldots, P_k$ such that $\bigcup_{j=0}^{k} P_j = G$; P_0 consists of a single edge; the endpoints of every P_i, $i \geq 1$, are in G_{i-1}; and no other vertices of P_i are in G_{i-1}, where $G_{i-1} = \bigcup_{j=0}^{i-1} P_j$. The paths P_j are called *ears*. An *open ear decomposition* is an ear decomposition such that for every ear, the two endpoints are distinct. We denote the two endpoints of ear P_i, $0 \leq i \leq k$, as α_i and β_i.

Lemma 1 (Whitney [32]) *A graph $G = (V, E)$ is biconnected if and only if it has an open ear decomposition.*

Let $\omega : V \to \mathbb{R}$ be an assignment of weights $\omega(v)$ to the vertices of G such that $\sum_{v \in V} \omega(v) \leq 1$. The weight of a subgraph of G is the sum of the vertex weights in this subgraph. An *ϵ-separator* of G is a set S such that none of the connected components of $G - S$ has weight exceeding ϵ.

Figure 1: Illustrating the definition of K_4 and $K_{2,3}$.

The *complete graph with n vertices* (Figure 1) is defined as $K_n = (V, E)$ with $V = \{1, \ldots, n\}$ and $E = \{\{i, j\} : 1 \leq i < j \leq n\}$. The *complete bipartite graph with $m + n$ vertices* (Figure 1) is defined as $K_{m,n} = (V, E)$ with $V = \{1, \ldots, m + n\}$ and $E = \{\{i, j\} : 1 \leq i \leq m < j \leq m + n\}$. A graph G is an *edge-expansion* of another graph H, if G can be obtained from H by replacing edges in H with simple paths. Graph H is a *minor* of G if G contains a subgraph that is an edge-expansion of H. We use the following characterization of outerplanar graphs.

Theorem 1 (e.g., [17]) *A graph G is outerplanar if and only if it does not have either $K_{2,3}$ or K_4 as a minor.*

In our algorithms, we represent a graph $G = (V, E)$ as the two sets, V and E. An embedded outerplanar graph is represented as the set V of vertices sorted clockwise along the outer boundary of the graph; edges are represented as adjacency lists stored with the vertices; each adjacency list is sorted counterclockwise around the vertex, starting with the edge on the outer face of G incident to v and preceding v in the clockwise traversal of the outer boundary of the graph. In the lower bound proof for embedding, we use a slightly weaker representation of an embedding. For the lower bound proof, we represent the graph G as the two sets V and E and label every edge with its positions $n_v(e)$ and $n_w(e)$ counterclockwise around v and w, respectively. This way we guarantee that the $\Omega(\text{perm}(N))$ I/O lower bound is not just a consequence of the requirement to arrange the vertices of G in the right order.

3 Time-Forward Processing for Rooted Trees

Time-forward processing was introduced in [9] as a technique to evaluate directed acyclic graphs: Every vertex is initially labeled with a predicate $\phi(v)$. In the end, we want to compute a predicate $\psi(v)$, for every vertex v, which may only depend on the predicate $\phi(v)$ and "input" received from the in-neighbors of v. This technique requires $O(\text{sort}(N + I))$ I/Os, where N is the number of vertices in G, and I is the total amount of information sent along all edges of G. In this section, we show the following result.

Algorithm 1 Top-down time-forward processing in a rooted tree.

Input: A rooted tree T whose edges are directed from parents to children and whose vertices are stored in preorder and labeled with predicates $\phi(v)$.
Output: An assignment of labels $\psi(v)$ to the vertices of T.

```
1: Let δ(r) be some dummy input for the root r of T.
2: PUSH(S, δ(r))
3: for every node v of T, in preorder do
4:    δ(v) ← POP(S)
5:    Compute ψ(v) from δ(v) and φ(v).
6:    Let w_1, ..., w_k be the children of v sorted by increasing preorder numbers.
7:    for i = k, k − 1, ..., 1 do
8:       Compute δ(w_i) from ψ(v).
9:       PUSH(S, δ(w_i))
10:   end for
11: end for
```

Theorem 2 *Given a rooted tree $T = (V, E)$ whose edges are directed and whose vertices are sorted in preorder or postorder, T can be evaluated in $O(\text{scan}(N+I))$ I/Os and using $O((N + I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T.*

We use the following two lemmas to prove the theorem for the case when the vertices are stored in preorder. The case when the vertices are stored in postorder is similar because by reversing a postorder numbering one obtains a preorder numbering.

Lemma 2 *Given a rooted tree $T = (V, E)$ whose edges are directed from parents to children and whose vertices are sorted in preorder, T can be evaluated in $O(\text{scan}(N+I))$ I/Os and using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T.*

Proof. Denote the data sent from the parent of a node v to node v by $\delta(v)$. We use Algorithm 1 to evaluate T. This algorithm uses a stack S to implement the sending of data from the parents to their children. To prove the correctness of Algorithm 1, we need to show that $\delta(v)$ is indeed the top element of S in Line 4 of the iteration that evaluates vertex v. This follows almost immediately from the following claim.

Claim 1 *Let m be the number of elements on stack S before evaluating a vertex v of T, and let T_v be the subtree of T rooted at v. Then there are never less than $m - 1$ elements on the stack while subtree T_v is being evaluated. After the evaluation of T_v, the stack holds exactly $m - 1$ elements.*

Proof. We prove the claim by induction on the size $|T_v|$ of T_v. If $|T_v| = 1$ (i.e., v is a leaf), v removes the top entry from the stack in Line 4 and does not put any

new entries onto the stack, as the loop in Lines 7–10 is never executed. Hence, the claim is true.

So assume that $|T_v| > 1$. Then v has at least one child. Let $w_1, \ldots, w_k$ be the children of v. The evaluation of tree T_v is done by evaluating v followed by the evaluation of subtrees $T_{w_1}, \ldots, T_{w_k}$. Note that each subtree T_{w_i} has size $|T_{w_i}| < |T_v|$; so the claim holds for T_{w_i}, by the induction hypothesis.

Let $S = (s_1, \ldots, s_m)$ before the evaluation of v. After the evaluation of v and before the evaluation of w_1, $S = (\delta(w_1), \ldots, \delta(w_k), s_2, \ldots, s_m)$. Inductively, we claim that the stack S holds elements $\delta(w_i), \ldots, \delta(w_k), s_2, \ldots, s_m$ before the evaluation of node w_i. As Claim 1 holds for T_{w_i}, the evaluation of T_{w_i} never touches elements $\delta(w_{i+1}), \ldots, \delta(w_k), s_2, \ldots, s_m$ and removes the top element $\delta(w_i)$ from the stack. Hence, after the evaluation of T_{w_i}, $S = (\delta(w_{i+1}), \ldots, \delta(w_k), s_2, \ldots, s_m)$. In particular, after the evaluation of T_{w_k}, $S = (s_2, \ldots, s_m)$. But the evaluation of T_{w_k} completes the evaluation of T_v. Hence, after the evaluation of T_v, $S = (s_2, \ldots, s_m)$, as desired. Also, before the evaluation of every subtree T_{w_i}, $|S| \geq m$. By the induction hypothesis, there are never fewer than $m - 1$ elements on the stack during the evaluation of T_{w_i}. Hence, the same is true for the evaluation of T_v. □

Claim 1 implies the correctness of the lemma: If node v is the root of T, then $S = (\delta(v))$ immediately before the evaluation of node v. Otherwise, v has a parent u with children $w_1, \ldots, w_k$ such that $v = w_i$, for some i, $1 \leq i \leq k$. Immediately after the evaluation of u, $S = (\delta(w_1), \ldots, \delta(w_k), \ldots)$. By Claim 1, the evaluation of every subtree $T(w_j)$, $1 \leq j < i$, removes the topmost element from the stack and leaves the rest of S unchanged. Hence, the evaluation of subtrees $T(w_1), \ldots, T(w_{i-1})$ removes elements $\delta(w_1), \ldots, \delta(w_{i-1})$ from S, and $\delta(w_i)$ is on the top of the stack before the evaluation of $v = w_i$.

In order to see the I/O-bound, observe that we scan the vertex list of T once, which takes $O(\text{scan}(N))$ I/Os. Data is sent along the tree edges using the stack S. Every data item is pushed once and popped once, so that we perform $O(I)$ stack operations, at the cost of $O(\text{scan}(I))$ I/Os. In total, we spend $O(\text{scan}(N + I))$ I/Os. The space requirements are clearly bounded by $O((N + I)/B)$ blocks of external memory, as this is the maximum number of blocks accessible in the given number of I/Os. □

Lemma 3 *Given a rooted tree $T = (V, E)$ whose edges are directed from children to parents and whose vertices are sorted in preorder, T can be evaluated in $O(\text{scan}(N + I))$ I/Os and using $O((N + I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T.*

Proof. The proof is similar to that of Lemma 2. We simply reverse the order in which the nodes of T are processed. □

Proof (of Theorem 2). The crucial observation to prove Theorem 2 is the following: A vertex v with an edge directed from v to v's parent in T can only receive input from its children. Consider the subgraph T' of T induced by all

edges of T that are directed from children to parents. T' is a forest of rooted trees. Then the following claim holds.

Claim 2 *For every non-root vertex v in T', $\Gamma^-_{T'}(v) = \Gamma^-_T(v)$, that is, the in-neighborhood of v is the same in T and T'.*

Proof. Let $v \in T'$ be a vertex with $\Gamma^-_{T'}(v) \neq \Gamma^-_T(v)$. All edges in $\Gamma^-_T(v)$ are directed from children to parents except the edge from v's parent to v (if it is in $\Gamma^-_T(v)$). Thus, this is the only edge that may not be in $\Gamma^-_{T'}(v)$. However, if the edge from v's parent to v is directed from v's parent to v, then v is a root in T'. □

Claim 2 implies that we can use the algorithm of Lemma 3 to evaluate all non-root vertices of T'. Moreover, for every root v in T', all children have been fully evaluated, so that they can provide their input to v. Hence, every node that has not been evaluated yet is waiting only for the input from its parent. Thus, we consider the subgraph T'' of T induced by all edges directed from parents to children. Again, T'' is a forest of rooted trees. We apply Algorithm 1 to T'' to evaluate the remaining vertices.

By Lemmas 2 and 3, both phases of the algorithm take $O(\text{scan}(N+I))$ I/Os and use $O((N+I)/B)$ blocks of external memory. □

4 Outerplanarity Testing and Outerplanar Embedding

We divide the description of our algorithm for computing an outerplanar embedding of a graph into three parts. In Section 4.1, we show how to find an outerplanar embedding of a biconnected outerplanar graph G. Section 4.2 shows how to augment the algorithm of Section 4.1 to deal with the general case. In Section 4.3, we augment the algorithm of Section 4.1 so that it can test whether a given graph G is outerplanar. If G is outerplanar, the algorithm produces an outerplanar embedding of G. Otherwise, it outputs a subgraph of G that is an edge-expansion of K_4 or $K_{2,3}$. Graphs are assumed to be undirected in this section.

4.1 Outerplanar Embedding for Biconnected Graphs

We use Algorithm 2 to compute an outerplanar embedding $\hat{G}$ of a biconnected outerplanar graph G. Next we describe the three steps of this algorithm in detail.

Step 1: Computing an open ear decomposition. We use the algorithm of [9] to compute an open ear decomposition of G in $O(\text{sort}(N))$ I/Os and using $O(N/B)$ blocks of external memory.

Algorithm 2 Embedding a biconnected outerplanar graph.

Input: A biconnected outerplanar graph G.
Output: An outerplanar embedding of G represented by sorted adjacency lists.

1: Compute an open ear decomposition $\mathcal{E} = (P_0, \ldots, P_k)$ of G.
2: Compute the cycle C clockwise along the outer boundary of an outerplanar embedding $\hat{G}$ of G.
3: Compute for each vertex v of G, its adjacency list sorted counterclockwise around v, starting with the predecessor of v in C and ending with the successor of v in C.

Step 2: Computing the boundary cycle. This step computes the boundary cycle C of an outerplanar embedding $\hat{G}$ of G, that is, the boundary cycle of the outer face of the embedding, which contains all vertices of G. The following lemma shows that C is the only simple cycle containing all vertices of G; hence, we can compute C by computing *any* simple cycle with this property.

Lemma 4 *Every biconnected outerplanar graph G contains a unique simple cycle containing all vertices of G.*

Proof. The existence of cycle C follows immediately from the outerplanarity and biconnectivity of G. In particular, the boundary of the outer face of an outerplanar embedding $\hat{G}$ of G is such a simple cycle.

So assume that there exists another simple cycle C' that contains all vertices of G. Let a and b be two vertices that are consecutive in C, but not in C' (see Figure 2). Then we can break C' into two internally vertex-disjoint paths P_1 and P_2 from a to b, both of them containing at least one internal vertex. Let c be an internal vertex of P_1, and let d be an internal vertex of P_2. As a and b are connected by an edge in C, there must be a subpath P_3 of C between c and d that contains neither a nor b. Consider all vertices on P_3 that are also contained in P_1. Let c' be such a vertex that is furthest away from c along P_3. Analogously, we define d' to be the vertex closest to c which is shared by P_3 and P_2. Let P_3' be the subpath of P_3 between c' and d'. Let H be the subgraph of G defined as the union of cycle C', edge $\{a, b\}$, and path P_3'. Cycle C', edge $\{a, b\}$, and path P_3' are internally vertex-disjoint. Thus, H is an edge-expansion of K_4, which contradicts the outerplanarity of G. □

Let $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ be the open ear decomposition computed in Step 1. We call an ear P_i, $i \geq 1$, *trivial* if it consists of a single edge. Otherwise, we call it *non-trivial*. Also, ear P_0 is defined to be non-trivial. During the construction of C, we restrict our attention to the non-trivial ears $P_{i_0}, P_{i_1}, \ldots, P_{i_q}$, $0 = i_0 < i_1 < \cdots < i_q$, in $\mathcal{E}$, as a trivial ear does not contribute new vertices to the graph consisting of all previous ears. For the sake of simplicity, we write $Q_j = P_{i_j}$, $\sigma_j = \alpha_{i_j}$, and $\tau_j = \beta_{i_j}$, for $0 \leq i \leq q$. Let $G_1, \ldots, G_q$ be subgraphs of G defined

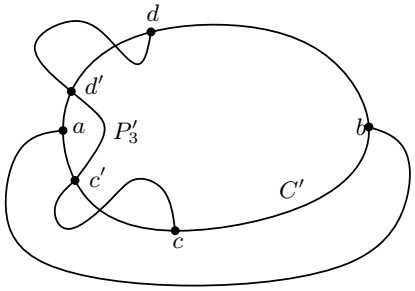

Figure 2: Proof of the uniqueness of the boundary cycle of a biconnected outerplanar graph.

as follows: $G_1 = Q_0 \cup Q_1$. For $1 < i \leq q$, G_i is obtained by removing edge $\{\sigma_i, \tau_i\}$ from G_{i-1} and adding Q_i to the resulting graph.

Lemma 5 *Graphs $G_1, \ldots, G_q$ are simple cycles.*

Proof. If $i = 1$, G_i is a simple cycle because Q_1 is a simple path and Q_0 is an edge connecting the two endpoints of Q_1. So assume that G_i is a simple cycle, for $1 \leq i < k$. We show that G_k is a simple cycle.

The crucial observation is that the endpoints σ_k and τ_k of Q_k are adjacent in G_{k-1}. Assume the contrary. Then G_{k-1} consists of two internally vertex-disjoint paths P and P' between σ_k and τ_k, each containing at least one internal vertex (see Figure 3a). Let γ be an internal vertex of P, let γ' be an internal vertex of P', and let γ_k be an internal vertex of Q_k. Vertex γ_k exists because Q_k is non-trivial and $k > 0$. Then paths $Q_k(\sigma_k, \gamma_k)$, $Q_k(\tau_k, \gamma_k)$, $P(\sigma_k, \gamma)$, $P(\tau_k, \gamma)$, $P'(\sigma_k, \gamma')$, and $P'(\tau_k, \gamma')$ are internally vertex disjoint, so that the graph $Q_k \cup G_{k-1}$, which is a subgraph of G, is an edge-expansion of $K_{2,3}$. This contradicts the outerplanarity of G.

Given that vertices σ_k and τ_k are adjacent in G_{k-1}, the removal of edge $\{\sigma_k, \tau_k\}$ from G_{k-1} creates a simple path G', so that G' and Q_k are two internally vertex disjoint simple paths sharing their endpoints (see Figure 3b). Thus, $G_k = G' \cup Q_k$ is a simple cycle. □

By Lemmas 4 and 5, $G_q = C$. It remains to show how to construct the graph G_q, in order to finish this step. Graph G_q is obtained from G by removing all trivial ears and all edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$. Given an open ear decomposition $\mathcal{E}$ of G, we scan $\mathcal{E}$ to identify all trivial ears and edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$. Given this list of edges, we sort and scan the edge list of G to remove these edges from G. This takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory. The resulting graph is C.

In order to complete this phase, we need to establish an ordering of the vertices in C along the outer boundary of $\hat{G}$. By removing an arbitrary edge from C, we obtain a simple path C'. We compute the distance of each vertex in C' from one endpoint of C' using the Euler tour technique and list-ranking [9].

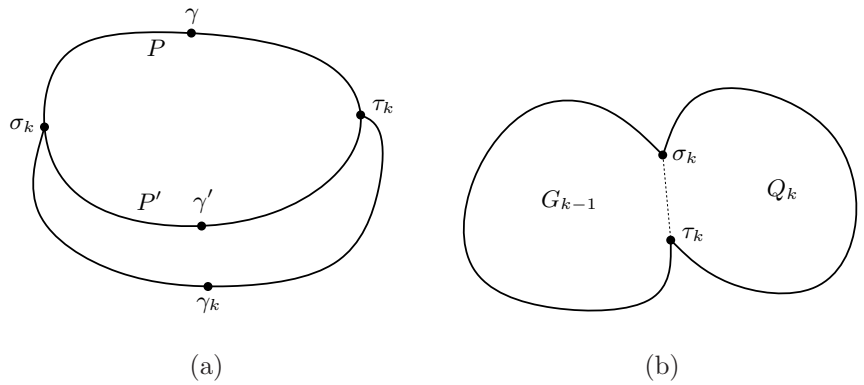

Figure 3: (a) The attachment vertices of Q_k are adjacent in G_{k-1}. (b) This implies that G_k is a simple cycle.

These distances give the desired ordering of the vertices in C along the outer boundary of $\hat{G}$. As shown in [9], these two techniques take $O(\text{sort}(N))$ I/Os and use $O(N/B)$ blocks of external memory.

Step 3: Embedding the diagonals. Let $C = (v_1, \ldots, v_N)$ be the boundary cycle computed in the previous step. In order to compute a complete embedding of G, we have to embed the diagonals of $\hat{G}$, that is, all edges of G that are not in C. Assuming that the order of the vertices in C computed in the previous step is clockwise around the outer boundary of $\hat{G}$, we compute such an embedding of the diagonals by sorting the adjacency list $A(v_i)$ of every vertex v_i, $1 \leq i \leq N$, counterclockwise around v_i, starting with v_{i-1} and ending with v_{i+1}, where we define $v_0 = v_N$ and $v_{N+1} = v_1$.

We denote $\nu(v_i) = i$ as the *index* of vertex v_i, $1 \leq i \leq N$. We sort the vertices in each list $A(v_i)$ by decreasing indices. Let $A(v_i) = \langle w_1, \ldots, w_t \rangle$ be the resulting list for vertex v_i. We find vertex $w_j = v_{i-1}$ and rearrange the vertices in $A(v_i)$ as the list $\langle w_j, \ldots, w_t, w_1, \ldots w_{j-1} \rangle$. Clearly, this step takes $O(\text{sort}(N))$ I/Os and uses $O(N/B)$ blocks of external memory. Let $A(v_i) = \langle w'_1, \ldots, w'_t \rangle$. We define $\nu_i(w'_j) = j$. The following lemma proves that the counterclockwise order of the vertices in $A(v_i)$ is computed correctly by sorting $A(v_i)$ as just described.

Lemma 6 *Let w_j and w_k be two vertices in $A(v_i)$, $1 \leq i \leq N$, with $\nu_i(w_j) < \nu_i(w_k)$. Then v_{i-1}, w_j, w_k, v_{i+1} appear in this order counterclockwise around v_i in the outerplanar embedding $\hat{G}$ of G that is defined by arranging the vertices of G in the order $v_1, \ldots, v_N$ clockwise along the outer face.*

Proof. Consider the planar subdivision D induced by C and edge $\{v_i, w_j\}$ (see Figure 4). D has three faces: the outer face, a face f_l bounded by the path from v_i to w_j clockwise along C and edge $\{v_i, w_j\}$, and a face f_r bounded by the path from v_i to w_j counterclockwise along C and edge $\{v_i, w_j\}$. If w_k appears

clockwise around v_i from w_j, then w_k is on the boundary of f_r, as edge $\{v_i, w_k\}$ and the boundary cycle of f_r cannot intersect, except in vertices v_i and w_k. Depending on where the vertex x with $\nu(x) = 0$ appears along C relative to v_{i-1}, v_i, w_j, and w_k, one of the following is true:

$$\nu(v_{i-1}) < \nu(v_i) < \nu(w_j) < \nu(w_k)$$
$$\nu(v_i) < \nu(w_j) < \nu(w_k) < \nu(v_{i-1})$$
$$\nu(w_j) < \nu(w_k) < \nu(v_{i-1}) < \nu(v_i)$$
$$\nu(w_k) < \nu(v_{i-1}) < \nu(v_i) < \nu(w_j)$$

It is easy to verify that in all four cases, $\nu_i(w_k) < \nu_i(w_j)$, contradicting the assumption made in the lemma. Thus, w_k appears counterclockwise from w_j around v_i. □

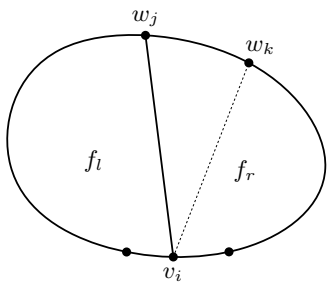

Figure 4: Proof of Lemma 6.

We represent the computed embedding $\hat{G}$ of G as a list $\mathcal{A}$ that is the concatenation of lists $A(v_i)$, $1 \leq i \leq N$, in this order. Every vertex $w \in A(v_i)$ is marked as representing a vertex adjacent to v_i, by storing it as the directed edge (v_i, w). This representation can easily be produced in $O(\text{sort}(N))$ I/Os.

4.2 Outerplanar Embedding — The General Case

If G is not connected, the connected components $G_1, \ldots, G_k$ of G can be embedded independently. An outerplanar embedding $\hat{G}_i$ of any component G_i, $1 \leq i \leq k$, can be obtained from outerplanar embeddings of its biconnected components $H_{i,1}, \ldots, H_{i,l_i}$. In particular, the order of the edges around every vertex v that is contained in only one biconnected component $H_{i,j}$ is fully determined by the embedding of $H_{i,j}$. For a cutpoint v contained in biconnected components $H_{i,j_1}, \ldots, H_{i,j_q}$, we obtain a valid ordering of the vertices around v by concatenating the sorted adjacency lists $A_1(v), \ldots, A_q(v)$ of v in $\hat{H}_{i,j_1}, \ldots, \hat{H}_{i,j_q}$.

Similar to the case where G is biconnected, we compute a list $\mathcal{A}$ which is the concatenation of adjacency lists $A(v)$, $v \in G$. List $\mathcal{A}$ is the concatenation of lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$, one for each connected component G_i, $1 \leq i \leq k$, of G. For

Algorithm 3 Embedding an outerplanar graph.

Input: An outerplanar graph G.
Output: An outerplanar embedding of G represented by sorted adjacency lists.

1: **for** every connected component G_i of G **do**
2: Compute the list C_i of vertices visited in clockwise order around the outer boundary of G_i, starting at an arbitrary vertex in G_i. (The number of appearances of a vertex $v \in G_i$ in C_i is equal to the number of biconnected components of G_i that contain v.)
3: **for** every vertex $v \in G_i$ **do**
4: Let u be the predecessor of the first appearance of v in C_i.
5: Let w be the successor of the first appearance of v in C_i.
6: Sort the vertices in adjacency list $A(v)$ in counterclockwise order around v, starting at u and ending at w.
7: Remove all but the last appearance of v from C_i.
8: **end for**
9: Let $C'_i = \langle v_1, \ldots, v_s \rangle$ be the resulting vertex list.
10: Compute list $\mathcal{A}_i$ as the concatenation of adjacency lists $A(v_1), \ldots, A(v_s)$.
11: **end for**
12: Compute list $\mathcal{A}$ as the concatenation of lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$.
13: Compute a list C as the concatenation of lists $C'_1, \ldots, C'_k$.
14: Let $C = \langle v_1, \ldots, v_k \rangle$. Then define $\nu(v_i) = i$; $\nu(v_i)$ is called the *index* of v_i.

the algorithms in Section 5 through 7 to run in $O(\text{scan}(N))$ I/Os, the adjacency lists $A(v)$ in each list $\mathcal{A}_i$ need to be arranged in an appropriate order. For the biconnected case, this order is provided by the order of the vertices along the outer boundary of G_i. In the general case, we have to do this arrangement more carefully. Our algorithm for computing list $\mathcal{A}$ is sketched in Algorithm 3.

We use algorithms of [9] to identify the connected and biconnected components of G. This takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory. Then we apply the algorithm of Section 4.1 to each of the biconnected components $H_{i,j}$ of G. This takes $O(\text{sort}(|H_{i,j}|))$ I/Os per biconnected component, $O(\text{sort}(N))$ I/Os in total.

For each connected component G_i of G, we compute list $\mathcal{A}_i$ as follows: The *bicomp-cutpoint-tree* T_i of G_i is a tree that contains all cutpoints of G_i and one vertex $v(H)$ per bicomp H of G_i. There is an edge $\{v, v(H)\}$ in T_i, if cutpoint v is contained in bicomp H. We choose one bicomp vertex $v(H_r)$ as the root of T_i. The *parent cutpoint* of a bicomp H is the cutpoint $p(v(H))$, where $p(v)$ denotes the parent of node v in T_i. The parent bicomp of bicomp H is the bicomp H' corresponding to node $v(H') = p(p(v(H)))$.

Given the biconnected components of G_i, we sort and scan the vertex lists of these biconnected components to find the cutpoints of G_i; these are the vertices that are contained in more than one biconnected component. Given the cutpoints of G_i and the bicomps containing each cutpoint, tree T_i is readily con-

structed in $O(\text{sort}(N))$ I/Os. Using the Euler tour technique and list-ranking [9], we compute the parent cutpoint and the parent bicomp for each bicomp H of G_i.

We are now ready to construct the adjacency lists of all vertices in G_i so that they can be included in $\mathcal{A}_i$. For every vertex v contained in only one bicomp H, we take the adjacency list of v as computed when embedding H. If v is a cutpoint, it is the parent cutpoint for all but one bicomp containing v. These bicomps are the children of v in T_i. Let $H_1, \ldots, H_q$ be the bicomps containing v, visited in this order by the Euler tour used to root T_i; that is, $v(H_1)$ is v's parent in T_i. Then we compute $A(v)$ as the concatenation of the adjacency lists of v computed for bicomps $H_1, H_q, H_{q-1}, \ldots, H_2$, in this order.

In order to compute C'_i, we transform each bicomp of G_i into a path by removing all edges not on its outer boundary and the edge between its parent cutpoint and its counterclockwise neighbor along the outer boundary of the bicomp. For the root bicomp, we remove an arbitrary edge on its boundary. The resulting graph is a tree T'_i. A preorder numbering of T'_i that is consistent with the previous Euler tour of T_i produces the order of the vertices in C'_i. Given the Euler tour, such a preorder numbering can easily be computed in $O(\text{sort}(N))$ I/Os using list-ranking [9]. Given the preorder numbering and adjacency lists $A(v)$, $v \in G_i$, list $\mathcal{A}_i$ can now be constructed by replacing every vertex in C'_i by its adjacency list. These computations require sorting and scanning the vertex and edge sets of G_i a constant number of times. Thus, this algorithm takes $O(\text{sort}(N))$ I/Os and uses $O(N/B)$ blocks of external memory. We conclude this subsection with a number of definitions and two observations that will be useful in proving the algorithms in Sections 5 through 7 correct.

Observation 1 *For every adjacency list $A(v) = \langle w_1, \ldots, w_k \rangle$ with $\nu(v) > 1$, there exists an index $1 \leq j \leq k$ such that $\nu(w_{j+1}) > \nu(w_{j+2}) > \cdots > \nu(w_k) > \nu(v) > \nu(w_1) > \cdots > \nu(w_j)$. If $\nu(v) = 1$, then $\nu(w_1) > \cdots > \nu(w_k) > \nu(v)$.*

We call a path $P = (v_0, \ldots, v_p)$ *monotone* if $\nu(v_0) < \cdots < \nu(v_p)$. We say that in the computed embedding of G, a monotone path $P = (s = v_0, \ldots, v_p)$ is *to the left* of another monotone path $P' = (s = v'_0, \ldots, v'_{p'})$ with the same source s if P and P' share vertices $s = w_0, \ldots, w_t$, that is, $w_k = v_{i_k} = v'_{j_k}$, for some indices i_k and j_k and $0 \leq k \leq t$, and edges $\{w_k, v_{i_k-1}\}$, $\{w_k, v_{i_k+1}\}$ and $\{w_k, v'_{j_k+1}\}$ appear in this order clockwise around w_k, for $0 \leq k \leq t$. This definition is somewhat imprecise, as vertex v_{-1} is not defined; but we will apply results based on this definition only to connected embedded outerplanar graphs and to monotone paths in these graphs that start at the vertex r with $\nu(r) = 1$. In this case, we imagine v_{-1} to be an additional vertex embedded in the outer face of the given outerplanar embedding of G and connected only to r through a single edge $\{v_{-1}, r\}$.

A *lexicographical ordering* "$\prec$" of all monotone paths with the same source is defined as follows: Let $P = (v_0, \ldots, v_p)$ and $P' = (v'_0, \ldots, v'_{p'})$ be two such paths. Then there exists an index $j > 0$ such that $v_k = v'_k$, for $0 \leq k < j$ and $v_j \neq v'_j$. We say that $P \prec P'$ if and only if $\nu(v_j) < \nu(v'_j)$.

Lemma 7

(i) Let P and P' be two monotone paths with source s, $\nu(s) = 1$. If $P \prec P'$, then P' is not to the left of P.

(ii) Let P be a monotone path from s to some vertex v. Then there exists no monotone path from s to a vertex w, $\nu(w) > \nu(v)$, that is to the left of P.

Proof. (i) Let $P = (s = v_0, \ldots, v_p)$ and $P' = (s = v'_0, \ldots, v'_{p'})$ with $P \prec P'$, and assume that P' is to the left of P. Let $j > 0$ be the index such that $v_i = v'_i$, for $0 \leq i < j$, and $\nu(v_j) < \nu(v'_j)$. As P' is to the left of P, vertices v_{j-2}, v'_j, and v_j appear in this order clockwise around v_{j-1}. As the vertices of G are numbered clockwise along the outer boundary of G, there has to exist a path P'' from s to v_j clockwise along the outer boundary that avoids v'_j. Let u be the vertex closest to v_j that is shared by P and P''. Then $P(u, v_j)$ and $P''(u, v_j)$ together define a closed Jordan curve that encloses v'_j. This contradicts the assumption that the orders of the edges around the vertices of G describe an outerplanar embedding of G. See Figure 5.

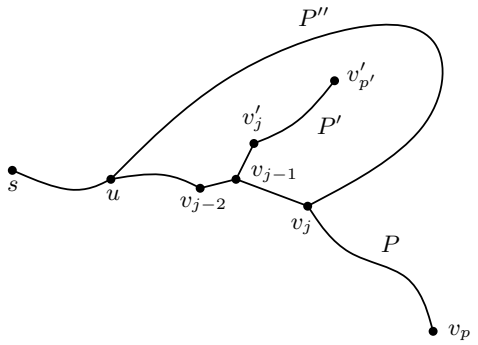

Figure 5: Proof of Lemma 7.

(ii) Let $P' = (s = v'_0, \ldots, v'_{p'} = w)$ be a monotone path from s to w, $\nu(w) > \nu(v)$, and let $P = (s = v_0, \ldots, v_p = v)$. Assume that P' is to the left of P. As $\nu(v) < \nu(w)$, the path P'' from s to v along the outer boundary of G does not contain w. Thus, similar to the proof of (i), we can construct a closed Jordan curve that encloses w, which leads to a contradiction. □

4.3 Outerplanarity Testing

We augment the embedding algorithm of the previous section in order to test whether a given graph $G = (V, E)$ is outerplanar. First we test whether $|E| \leq 2|V| - 3$. If not, G cannot be outerplanar. As a graph is outerplanar if and only if its biconnected components are outerplanar, we only have to augment the algorithm of Section 4.1, which deals with the biconnected components of G. If this algorithm produces a valid outerplanar embedding for every biconnected component of G, this is proof that G is outerplanar. Otherwise, the

Algorithm 4 Test for intersecting diagonals.

Input: The list $\mathcal{A}$ computed for bicomp H by Algorithm 2.
Output: A decision whether H is outerplanar along with a proof for the decision, either in the form of a subgraph that is an edge-expansion of K_4 or in the form of an outerplanar embedding of H.

```
1: Initialize stack S to be empty.
2: for each entry (v, w) ∈ A do
3:   if ν(w) > ν(v) then
4:     PUSH(S, {v, w})
5:   else
6:     {a, b} ← POP(S)
7:     if {a, b} ≠ {v, w} then
8:       Report the graph consisting of C augmented with edges {a, b} and
         {v, w} as proof that H is not outerplanar and stop.
9:     end if
10:  end if
11: end for
12: Report the embedding Ĥ of H represented by list A as proof for the outer-
    planarity of H.
```

algorithm fails to produce a correct outerplanar embedding for at least one of the biconnected components of G. Let H be such a bicomp.

The algorithm can fail in two ways. It may not be able to compute the boundary cycle C, or it computes the boundary cycle C and then produces an intersection between two edges when embedding the diagonals of H. We discuss both cases in detail.

Given an open ear decomposition $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ of H, the algorithm tries to compute the boundary cycle C by producing a sequence of cycles $G_0, \ldots, G_q$, where G_{i+1} is obtained from G_i by replacing edge $\{\sigma_{i+1}, \tau_{i+1}\}$ in G_i with the non-trivial ear Q_{i+1}. If G_i contains edge $\{\sigma_{i+1}, \tau_{i+1}\}$, for all $0 \leq i < q$, the algorithm successfully computes C. The only way this construction can fail is that there is a non-trivial ear Q_{i+1} such that G_i does not contain edge $\{\sigma_{i+1}, \tau_{i+1}\}$. As shown in the proof of Lemma 5, $G_i \cup Q_{i+1}$ is an edge-expansion of $K_{2,3}$ in this case. Thus, we output $G_i \cup Q_{i+1}$ as proof that G is not outerplanar.

Given the boundary cycle C, all edges of H that are not in C are diagonals of H. We compute list $\mathcal{A}$ as described in Section 4.1 and use $\mathcal{A}$ and a stack S to test for intersecting diagonals. The details are provided in Algorithm 4.

Lemma 8 *Given a cycle C and a list $\mathcal{A}$ as computed by the embedding algorithm, graph H is outerplanar if and only if Algorithm 4 confirms this.*

Proof. First assume that Algorithm 4 reports a graph H' consisting of the cycle C augmented with two edges $\{a, b\}$ and $\{v, w\}$ as proof that H is not outerplanar. This can happen only if $\nu(w) < \nu(v)$. Thus, edge $\{v, w\}$ has been

pushed on stack S before reporting H'. As edge $\{v, w\}$ cannot be successfully removed from S before visiting v, edge $\{v, w\}$ is still stored on S, below $\{a, b\}$. That is, edge $\{a, b\}$ has been pushed on the stack S after edge $\{v, w\}$, so that $\nu(w) \leq \nu(a) \leq \nu(v)$. However, $\nu(a) \neq \nu(v)$ because otherwise edge $\{v, b\}$ would succeed edge $\{v, w\}$ in the counterclockwise order around v and hence in $\mathcal{A}$; that is, the algorithm would try to pop edge $\{v, w\}$ from S before pushing edge $\{v, b\}$ on the stack. Hence, $\nu(a) < \nu(v)$. As edge $\{a, b\}$ has not been popped from the stack when the scan of C visits v, $\nu(v) < \nu(b)$. This in turn implies that $\nu(w) < \nu(a)$ because otherwise edge $\{w, v\}$ would succeed edge $\{w, b\}$ in the counterclockwise order of edges around w and hence in $\mathcal{A}$; that is, edge $\{w, b\}$ would be pushed onto S before edge $\{w, v\}$. Hence, $\nu(w) < \nu(a) < \nu(v) < \nu(b)$, so that H' is an edge-expansion of K_4, which proves that H is not outerplanar.

Now assume that H contains a subgraph that is an edge-expansion of K_4. Then there must be four vertices a, c, b, d, $\nu(a) < \nu(c) < \nu(b) < \nu(d)$ such that H contains edges $\{a, b\}$ and $\{c, d\}$. (If there are no two such edges, $\mathcal{A}$ represents a valid outerplanar embedding of H, so that H cannot have K_4 as a minor.) Edge $\{a, b\}$ is pushed on the stack before edge $\{c, d\}$ and Algorithm 4 tries to remove edge $\{a, b\}$ from the stack before removing edge $\{c, d\}$ because b is visited before d. This will lead to the reporting of a pair of intersecting diagonals. (Note that these are not necessarily $\{a, b\}$ and $\{c, d\}$, as the algorithm may find other conflicts before finding the conflict between $\{a, b\}$ and $\{c, d\}$.)

Finally, if H contains an edge-expansion H' of $K_{2,3}$, but no edge-expansion of K_4, we have to distinguish a number of cases. Let the edge-expansion of $K_{2,3}$ in H be induced by paths between vertices a, b and vertices c, d, e. W.l.o.g., $\nu(a) < \nu(c) < \nu(b) < \nu(d)$. Assume the contrary. That is, either $\nu(a) < \nu(b) < \nu(c) < \nu(d)$ or $\nu(a) < \nu(c) < \nu(d) < \nu(b)$. If $\nu(a) < \nu(b) < \nu(c) < \nu(d)$, there has to be an edge $\{v, w\}$ on the path from a to c in H' such that $\nu(v) < \nu(b) < \nu(w)$, as that path has to avoid b. Analogously, the path from b to d has to avoid v and w, so that there has to be an edge $\{x, y\}$ on this path with either $\nu(v) < \nu(x) < \nu(w) < \nu(y)$ or $\nu(y) < \nu(v) < \nu(x) < \nu(w)$. In both cases, $|\nu(v) - \nu(w)| \geq 2$ and $|\nu(x) - \nu(y)| \geq 2$. Thus, edges $\{v, w\}$ and $\{x, y\}$ cannot be part of the boundary cycle C, so that C together with edges $\{v, w\}$ and $\{x, y\}$ defines a subgraph of H that is an edge-expansion of K_4. This leads to a contradiction. The case $\nu(a) < \nu(c) < \nu(d) < \nu(b)$ is similar.

Depending on $\nu(e)$, we obtain three cases now. If $\nu(e) < \nu(a) < \nu(b)$, then $\nu(e) < \nu(a) < \nu(b) < \nu(d)$. If $\nu(a) < \nu(e) < \nu(b)$, then $\nu(a) < \nu(c), \nu(e) < \nu(b)$. If $\nu(a) < \nu(b) < \nu(e)$, then $\nu(a) < \nu(b) < \nu(d), \nu(e)$. By replacing c or d with e in the construction of the previous paragraph, we obtain a contradiction in each of these cases. Thus, the construction of cycle C would have failed if H has $K_{2,3}$ as a minor, but not K_4. □

The $K_{2,3}$-test during the construction of the boundary cycle can be incorporated in the embedding algorithm without increasing the I/O-complexity of that phase. Given list $\mathcal{A}$, the test for intersecting diagonals takes $O(\text{scan}(|\mathcal{A}|)) = O(\text{scan}(N))$ I/Os. To see this, observe that every edge $\{v, w\}$, $\nu(v) < \nu(w)$, in G is pushed on the stack at most once, namely when the traversal visits vertex v,

and removed at most once, namely when the traversal visits vertex w. Thus, we perform $O(N)$ stack operations, which takes $O(\text{scan}(N))$ I/Os. We have shown the following theorem.

Theorem 3 *It takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to test whether a graph $G = (V, E)$ of size $N = |V| + |E|$ is outerplanar and to provide proof for the decision of the algorithm by constructing an outerplanar embedding of G or extracting a subgraph of G that is an edge-expansion of $K_{2,3}$ or K_4.*

5 Triangulation

Before addressing the problems of DFS, BFS, SSSP, and computing graph separators, we show how to triangulate an embedded connected outerplanar graph G in a linear number of I/Os. This is an important (preprocessing) step in our algorithms for the above problems. Our triangulation algorithm can easily be extended to deal with disconnected graphs as follows: On encountering a vertex v that has the smallest index $\nu(v)$ in its connected component, we add an edge $\{u, v\}$, $\nu(u) = \nu(v) - 1$ to G. This can be done on the fly while triangulating G and transforms G into a connected supergraph G' whose triangulation is also a triangulation of G.

Formally, a *triangulation* of an outerplanar graph G is a biconnected outerplanar supergraph Δ of G with the same vertex set as G and all of whose interior faces are triangles.[1] We show how to compute a list $\mathcal{D}$ that represents the embedding $\hat{\Delta}$ of Δ from the list $\mathcal{A}$ that represents the embedding $\hat{G}$ of G. From now on we will not distinguish between a graph and its embedding. All graphs are considered to be embedded.

We need a few definitions to present our algorithm. An *ordered triangulation* of G is a list $\mathcal{T}$ that represents the dual tree T of a triangulation Δ of G and has the following properties: (1) The vertices $v_1, \ldots, v_N$, where $\nu(v_i) < \nu(v_{i+1})$, for $1 \leq i < N$, appear in this order in a clockwise traversal of the outer boundary of Δ. (2) A clockwise traversal of the boundary from v_1 to v_N defines an Euler tour of T, which in turn defines a postorder numbering of the vertices in T. List $\mathcal{T}$ stores the vertices of T sorted according to this postorder numbering.

Let $r \in G$ be the vertex with $\nu(r) = 1$. An *ordered partial triangulation* of G w.r.t. the shortest monotone path $P = (r = v_0, \ldots, v_p = v)$ from vertex r to some vertex v is a list $\mathcal{T} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_p$, where list $\mathcal{T}_i$ is an ordered triangulation of the subgraph of G defined by all edges $\{a, b\} \in G$, $\nu(v_{i-1}) \leq \nu(a), \nu(b) \leq \nu(v_i)$. (Note that list $\mathcal{T}_i$ is empty if $\nu(v_{i-1}) + 1 = \nu(v_i)$.)

The *fringe* $\mathcal{F}(P)$ of a monotone path $P = (v_0, \ldots, v_p)$ in graph G is a list of directed edges $\langle (v_0, w_{0,0}), \ldots, (v_0, w_{0,i_0}), (v_1, w_{1,0}), \ldots, (v_1, w_{1,i_1}), \ldots, (v_p, w_{p,0}),$

[1] This definition refers to the faces of Δ without explicitly referring to an embedding $\hat{\Delta}$ of Δ that defines these faces. The faces of a planar graph G are well-defined only if an embedding $\hat{G}$ of G is given. It is easy to show, however, that the outerplanar embedding of a triangulation as defined here is unique, except for flipping the whole graph (see Section 8).

$\ldots, (v_p, w_{p,i_p})\rangle$, where for each $0 \leq j < p$, edges $\{v_j, w_{j,k}\}$, $0 \leq k \leq i_j$, are the edges in G incident to v_j with $\nu(v_{j+1}) \leq \nu(w_{j,k})$. For v_p, we require that $\nu(v_p) < \nu(w_{p,k})$. The edges incident to every vertex v_j are sorted so that the path $P = (v_0, \ldots, v_j, w_k)$ is to the left of path $P = (v_0, \ldots, v_j, w_{k-1})$, for $0 < k \leq i_j$.

Our algorithm consists of two phases. The first phase (Algorithm 5) produces an ordered triangulation of G. A vertex α of T is labeled with the vertices u, v, w of G in clockwise order along the boundary of the triangle represented by α; that is, we denote α as the vertex (u, v, w). The second phase (Algorithm 6) uses list $\mathcal{T}$ to extract a list $\mathcal{D}$ that represents the embedding $\hat{\Delta}$ of Δ.

We say that Algorithm 5 *visits* vertex v when the for-loop inspects the first edge $(v, w) \in \mathcal{A}$ (which is the first edge in $A(v)$).

Lemma 9 *When Algorithm 5 visits vertex $v \in G$, the stack S represents the fringe of the shortest monotone path P in G from r to u, where $\nu(u) = \nu(v) - 1$. List $\mathcal{T}$ is an ordered partial triangulation of G w.r.t. P.*

Proof. We prove this claim by induction on $\nu(v)$. If $\nu(v) = 1$, $v = r$ is the first vertex to be visited, so that S is empty, and the claim of the lemma trivially holds. In this case, $\nu(w) > \nu(v)$, for all edges $(v, w) \in A(v)$. Thus, while inspecting $A(v)$, each iteration of the for loop executes Line 13, which pushes edge (v, w) on the stack. By Observation 1, $\nu(w_1) > \nu(w_2) > \cdots > \nu(w_k)$, where $A(v) = \langle (v, w_1), \ldots, (v, w_k)\rangle$. Thus, the claim of the lemma holds also for vertex v' with $\nu(v') = 2$.

So assume that $\nu(v) > 2$ and that the claim holds for u, $\nu(u) = \nu(v) - 1$. By Observation 1, there exists an index j such that $\nu(w_{j+1}) > \cdots > \nu(w_k) > \nu(u) > \nu(w_1) > \cdots > \nu(w_j)$, where $A(u) = \langle (u, w_1), \ldots, (u, w_k)\rangle$. We split the iterations inspecting $A(u)$ into three phases. The first phase inspects edge (u, w_1). The second phase inspects edges $(u, w_2), \ldots, (u, w_j)$. The third phase inspects edges $(u, w_{j+1}), \ldots, (u, w_k)$.

The iteration of the first phase executes Lines 4–10 of Algorithm 5. The iterations of the second phase execute Lines 15–19. The iterations of the third phase execute Line 13.

For the iteration of the first phase, vertex w_1 is a vertex on the path P whose fringe is stored on S. To see this, observe that P is a monotone path from r to vertex y with $\nu(y) = \nu(u) - 1$. If $w_1 = y$, we are done. So assume that $w_1 \neq y$. In this case, $\nu(w_1) < \nu(y)$ because $\nu(w_1) < \nu(u)$ and $\nu(y) = \nu(u) - 1$. Now observe that G contains a monotone path P' from r to w_1, which can be extended to a monotone path P'' from r to u by appending edge $\{w_1, u\}$. Let x be the last vertex on P which is in $P \cap P''$. If $x = w_1$, then $w_1 \in P$. Otherwise, let $P''' = P(r, x) \circ P''(x, u)$. Now either P is to the left of $P'''(r, w_1)$, or P''' is to the left of P. In both cases, we obtain a contradiction to Lemma 7(ii) because paths P, P''', and $P'''(r, w_1)$ are monotone and $\nu(w_1) < \nu(y) < \nu(u)$.

If $\nu(w_1) = \nu(u) - 1$, edge (w_1, u) is the last edge in the fringe $\mathcal{F}(P)$ represented by the current stack. Assume that this is not the case. Then let (w_1, z) be the edge succeeding (w_1, u) in $\mathcal{F}(P)$, let P_1 and P_2 be the paths obtained

Algorithm 5 Computing the dual of the triangulation.

Input: A list $\mathcal{A}$ that represents the embedding of an outerplanar graph G.
Output: A list $\mathcal{T}$ that represents an ordered triangulation of G.

Initialize stack S to be empty.
for each entry $(v, w) \in \mathcal{A}$ **do**
  if the previous entry in $\mathcal{A}$ exists and is of the form (v', w'), $v' \neq v$ **then**
    if $\nu(w) < \nu(v) - 1$ **then**
                                                {Bridge cutpoints.}
      repeat
        $(a, b) \leftarrow$ Pop(S)
        Append vertex (a, b, v) to $\mathcal{T}$.
      until $a = w$
    end if
  else
    if $\nu(v) < \nu(w)$ **then**
      Push($S, (v, w)$)
    else
      Pop(S)   {Triangulate the interior faces of G.}
      repeat
        $(a, b) \leftarrow$ Pop(S)
        Append vertex (a, b, v) to $\mathcal{T}$.
      until $a = w$
    end if
  end if
end for
Let v be the last vertex visited within the loop.
                                                {Bridge remaining cutpoints.}
Pop(S)
while S is not empty **do**
  $(a, b) \leftarrow$ Pop(S)
  Append vertex (a, b, v) to $\mathcal{T}$.
end while

by appending edges (w_1, z) and (w_1, u) to P. Path P_1 is to the left of P_2; but $\nu(z) > \nu(u)$ because $\nu(z) > \nu(w_1)$ and $\nu(u) = \nu(w_1) + 1$. This would contradict Lemma 7(i). Thus, S represents the fringe of a monotone path from r to u whose edges $\{u, w\}$, $\nu(w) > \nu(u)$, have been removed. $\mathcal{T}$ is an ordered partial triangulation of G w.r.t. this path.

If $\nu(w_1) < \nu(u) - 1$, there is no edge in $\mathcal{F}(P)$ that is incident to a vertex succeeding w_1 along P and is not part of P itself. Assume the contrary. That is, there is an edge $(w_2, v) \in \mathcal{F}(P)$ such that w_2 succeeds w_1 along P. If $w_2 = y$, then $\nu(v) > \nu(y)$. But this implies that $\nu(v) > \nu(u)$ because $\nu(u) = \nu(y) + 1$. Thus, we obtain a contradiction to Lemma 7(ii) because either $P \circ \{y, v\}$ is to the left of $P(r, w_1) \circ \{w_1, u\}$, or $P(r, w_1) \circ \{w_1, u\}$ is to the left of P. If $w_2 \neq y$, there are two cases. If $\nu(v) > \nu(y)$, we obtain a contradiction as in the case $w_2 = y$. Otherwise, let w_3 be the successor of w_2 along P. Then $P(r, w_3)$ is to the left of $P(r, w_2) \circ \{w_2, v\}$ because $\nu(v) > \nu(w_3)$. But this implies that P is to the left of $P(r, w_2) \circ \{w_2, v\}$, thereby leading to a contradiction to Lemma 7(ii) because $\nu(y) > \nu(v)$.

Thus, by adding an edge from u to the vertex y with $\nu(y) = \nu(u) - 1$ and triangulating the resulting face bounded by $P(w_1, y)$ and edges $\{y, u\}$ and $\{w_1, u\}$, we obtain a partial triangulation of G w.r.t. the path P' defined as the concatenation of $P(r, w_1)$ and edge $\{w, u\}$. The triangulation is ordered, as we add the triangles from y towards w_1 along P. Stack S now represents the fringe $\mathcal{F}(P')$ of P' after removing edges $\{u, w\}$, $\nu(w) > \nu(u)$.

For every edge (u, w) inspected in the second phase, edge (w, u) must be part of the shortened fringe of P' represented by S. This can be shown using the same argument as the one showing that vertex w_1 in the iteration of the first phase is part of P. By Observation 1, the edges inspected during the second phase are inspected according to the order of their endpoints from w_1 towards r along P. Using the same arguments as the ones showing that (w_1, u) is the last edge in the fringe $\mathcal{F}(P)$ if $\nu(w_1) = \nu(u) - 1$, it can be shown that there cannot be any edges $\{a, b\}$, $\nu(w_j) < \nu(a)$ and $\nu(b) \neq \nu(u)$, in the fringe of P', so that the top of stack S represents the subpath $P'(w_j, u)$ with dangling edges $(w_j, u), \ldots, (w_2, u)$ attached. The iterations of the second phase now triangulate the faces defined by $P'(w_j, u)$ and edges $\{w_j, u\}, \ldots, \{w_2, u\}$, so that, at the end of this phase, stack S represents a monotone path P'' from r to u, and $\mathcal{T}$ represents an ordered partial triangulation of G w.r.t. path P''. We argue as follows that path P'' is the *shortest* monotone path from r to u in G:

If there were a shorter monotone path Q from r to u, this path would have to pass through one of the biconnected components of the partial triangulation represented by $\mathcal{T}$ or through one of the vertices not inspected yet. The latter would result in a non-monotone path, as for each such vertex x, $\nu(x) > \nu(u)$. In the former case, if Q passes through the biconnected component defined by vertices $z \in G$, where $\nu(x) \leq \nu(z) \leq \nu(y)$ and $\{x, y\}$ is an edge in P'', replacing the subpath $Q(x, y)$ by edge $\{x, y\}$ in Q would result in a shorter path Q'. Thus, P'' is indeed the shortest monotone path from r to u.

The iterations of the third phase finally add edges $\{u, w_{j+1}\}, \ldots, \{u, w_k\}$ to the stack S, so that the representation of the fringe $\mathcal{F}(P'')$ of P'' is complete,

and the claim holds for v as well. □

After the for-loop is finished inspecting all edges in $\mathcal{A}$, the stack S represents the fringe $\mathcal{F}(P)$ of the shortest monotone path P in G from r to the last vertex v with $\nu(v) = N$, and $\mathcal{T}$ represents an ordered partial triangulation of G w.r.t. P. In this case, $\mathcal{F}(P) = P$, because the adjacency lists of all vertices of G have been inspected. Vertices v and r are not necessarily adjacent, so that the interior vertices of P are cutpoints of the triangulation constructed so far. To complete the triangulation, we have to make v adjacent to r and triangulate the resulting face. This is done by the while-loop in Lines 24–28. Thus, Algorithm 5 does indeed produce an ordered triangulation of G.

The following lemma shows that Algorithm 6 correctly constructs an embedding of the triangulation Δ represented by the list $\mathcal{T}$ computed by Algorithm 5. For a triangle $(a, b, c) \in \mathcal{T}$, let $\tau((a, b, c))$ be its position in $\mathcal{T}$. It follows from the fact that $\mathcal{T}$ represents a postorder traversal of tree T that triangles (a, b, c) with $\tau((a, b, c)) \geq j$, for some integer $1 \leq j \leq |\mathcal{T}|$, represent a subgraph Δ_j of Δ that is a triangulation. Let $\bar{\Delta}_j$ be the subgraph of Δ induced by all triangles (a, b, c), $\tau((a, b, c)) < j$. We denote the set of edges shared by Δ_j and $\bar{\Delta}_j$ by $\partial\Delta_j$.

Lemma 10 *After processing triangle $(a, b, c) \in \mathcal{T}$ with $\tau((a, b, c)) = j$, the concatenation of S and $\mathcal{D}$ represents an embedding of Δ_j. For every edge $\{x, y\} \in \partial\Delta_j$, stack S stores an entry (x, y), where x and y appear clockwise around the only triangle in Δ_j containing both x and y.*

Proof. We prove the claim by induction on j. If $j = |\mathcal{T}|$, triangle (a, b, c) is the first visited triangle. Thus, we execute Lines 5–11 of Algorithm 6. The concatenation of S and $\mathcal{D}$ is of the form $\langle(a, c), (a, b), (b, a), (b, c), (c, b), (c, a)\rangle$, where $\nu(a) = 1$ and $\nu(c) = |G|$. This represents an embedding of triangle (a, b, c). Moreover, stack S stores edges (a, b) and (b, c), and edge $\{a, c\}$ cannot be shared by Δ_j and $\bar{\Delta}_j$ because it is a boundary edge of Δ.

So assume that the claim holds for $j > k$. We prove the claim for $j = k$. In this case, we execute Lines 13–21. By the induction hypothesis, the concatenation of S and $\mathcal{D}$ represents an embedding of Δ_{k+1}, and S stores an edge (b, a), where edge $\{a, b\}$ is shared by triangle (a, b, c) and triangulation Δ_{k+1}. The while-loop in Lines 13–16 transfers edges from S to $\mathcal{D}$ until the top of the stack is of the form (b, a), (c, b), or (a, c). W.l.o.g. the top of the stack is (b, a). Lines 18–21 "insert" vertex c into the boundary cycle of Δ_{k+1}, by inserting entries (b, c), (c, b), (c, a), and (a, c) between entries (b, a) and (a, b) in the sequence represented by S and $\mathcal{D}$. The result is a valid embedding of Δ_k.

The edges removed from the stack during the while-loop in Lines 13–16 cannot be in $\partial\Delta_k$, as for every triangle (a', b', c') sharing one of those edges with Δ_{k+1}, $\tau((a', b', c')) > \tau((a, b, c))$. This follows from the fact that Δ is an ordered triangulation. Edge $\{a, b\}$ cannot be in $\partial\Delta_k$, as it is already shared by Δ_{k+1} and triangle (a, b, c). Thus, every edge in $\partial\Delta_k$ is either shared by Δ_{k+1} and $\bar{\Delta}_k$ or by triangle (a, b, c) and $\bar{\Delta}_k$. We have just argued that the former edges are not removed from S, and the latter edges can only be edges $\{a, c\}$ or

Algorithm 6 Extracting the embedding $\hat{\Delta}$ of Δ from the tree T.

Input: An ordered triangulation $\mathcal{T}$ of G.
Output: A list $\mathcal{D}$ representing the embedding $\hat{\Delta}$ of the triangulation Δ represented by $\mathcal{T}$.

Initialize stack S to be empty.
Initialize list $\mathcal{D}$ to be empty.
for each vertex $(a, b, c) \in \mathcal{T}$, in reverse order **do**
  if (a, b, c) is the first visited vertex **then**
    {Assume that $\nu(a) < \nu(b) < \nu(c)$.}
    Prepend entry (c, a) to $\mathcal{D}$.
    Prepend entry (c, b) to $\mathcal{D}$.
    PUSH$(S, (a, c))$
    PUSH$(S, (a, b))$
    PUSH$(S, (b, a))$
    PUSH$(S, (b, c))$
  else
    while the top of the stack S does not equal (b, a), (c, b) or (a, c) **do**
      $(d, e) \leftarrow$ POP(S)
      Prepend entry (d, e) to $\mathcal{D}$.
    end while
    {Assume w.l.o.g. that the top of the stack is (b, a).}
    Prepend entry (a, c) to $\mathcal{D}$.
    PUSH$(S, (b, c))$
    PUSH$(S, (c, b))$
    PUSH$(S, (c, a))$
  end if
end for
while S is not empty **do**
  $(a, b) \leftarrow$ POP(S)
  Prepend entry (a, b) to $\mathcal{D}$.
end while

$\{b, c\}$, whose representations we have just put on the stack. Thus, the claim also holds for $j = k$. □

Lemma 10 implies that, after the for-loop has inspected all triangles in $\mathcal{T}$, the concatenation of S and $\mathcal{D}$ represents an embedding of Δ. Thus, after prepending the entries in S to $\mathcal{D}$, as done in Lines 24–27 of Algorithm 6, $\mathcal{D}$ represents an embedding of Δ. Thus, Algorithms 5 and 6 correctly compute a list $\mathcal{D}$ representing an embedding $\hat{\Delta}$ of a triangulation Δ of G.

Theorem 4 *Given a list $\mathcal{A}$ representing an embedding $\hat{G}$ of a connected outerplanar graph G with N vertices, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a list $\mathcal{D}$ representing an embedding $\hat{\Delta}$ of a triangulation Δ of G.*

Proof. We compute list $\mathcal{D}$ using Algorithms 5 and 6. The correctness of this procedure follows from Lemmas 9 and 10.

To prove the I/O-bound, we observe that Algorithm 5 scans list $\mathcal{A}$, writes list $\mathcal{T}$, and performs a number of stack operations. Since both $\mathcal{A}$ and $\mathcal{T}$ have size $O(N)$, scanning list $\mathcal{A}$ and writing list $\mathcal{T}$ take $O(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 5 is twice the number of PUSH operations it performs. However, every entry of list $\mathcal{A}$ causes at most one PUSH operation to be performed, so that we perform $O(|\mathcal{A}|) = O(N)$ stack operations, which takes $O(\text{scan}(N))$ I/Os.

Algorithm 6 scans the list $\mathcal{T}$ and writes list $\mathcal{D}$. As both lists have size $O(N)$, this takes $O(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 6 is $O(|\mathcal{T}|) = O(N)$, as each entry in $\mathcal{T}$ causes at most four PUSH operations to be performed. Thus, Algorithm 6 also takes $O(\text{scan}(N))$ I/Os. □

6 Computing Separators of Outerplanar Graphs

In this section, we discuss the problem of finding a small ϵ-separator of an outerplanar graph. Given a graph $G = (V, E)$ and a weight function $\omega : V \to \mathbb{R}_0^+$, we define the weight $\omega(H)$ of a subgraph H of G as $\omega(H) = \sum_{v \in H} \omega(v)$. We assume that $\omega(G) \leq 1$. Then an ϵ-separator of G is a vertex set $S \subseteq V$ such that no connected component of $G - S$ has weight exceeding ϵ. We show the following result.

Theorem 5 *Given an embedded outerplanar graph $G = (V, E)$ with N vertices, represented as a list $\mathcal{A}$, a weight function $\omega : V \to \mathbb{R}_0^+$ such that $\omega(G) \leq 1$, and a constant $0 < \epsilon < 1$, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute an ϵ-separator S of size $O(1/\epsilon)$ for G.*

We assume that graph G is triangulated because this can easily be achieved using the triangulation algorithm of Section 5; every separator of the resulting triangulation is also a separator of G. Let T be its dual tree. Given an edge

$e \in T$ whose removal partitions T into two trees T_1 and T_2, trees T_1 and T_2 represent two subgraphs G_1 and G_2 of G such that G_1 and G_2 share a pair of vertices, v and w. Let $e^* = \{v, w\}$ be the edge dual to edge $e \in T$. The connected components of $G - \{v, w\}$ are graphs $G_1 - \{v, w\}$ and $G_2 - \{v, w\}$.

We choose a degree-1 vertex r of T as the root of T. For a vertex $v \in T$, let $\Delta(v)$ be the triangle of G represented by v, let V_v be the vertex set of $\Delta(v)$, let $T(v)$ be the subtree of T rooted at v, and let $G(v)$ be the subgraph of G defined as the union of all triangles $\Delta(w)$, $w \in T(v)$. Then $T(v)$ is the dual tree of $G(v)$. If $v \neq r$, we let $p(v)$ be the parent of v in T and $e_v = \{v, p(v)\}$ be the edge connecting v to its parent $p(v)$. We denote the endpoints of the dual edge e_v^* of e_v as x_v and y_v.

Our algorithms proceeds in three phases. The first phase of our algorithm computes weights $\omega(v)$ for the vertices of T such that $\omega(T) = \omega(G)$ and, for $v \neq r$, $\omega(T(v)) = \omega(G(v) - \{x_v, y_v\})$. The second phase of our algorithm computes a small edge-separator of T w.r.t. these vertex weights. The third phase of the algorithm computes the corresponding vertex separator of G. Next we discuss these three phases in detail.

Phase 1: Computing the weights of the dual vertices. We define weights $\omega(v)$ as follows: For the root r of T, we define $\omega(r) = \omega(\Delta(r))$. For every other vertex $v \in T$, we define $\omega(v) = \omega(z_v)$, where $z_v \in V_v \setminus \{x_v, y_v\}$. Note that z_v is unique. These vertex weights can be computed in $O(\text{scan}(N))$ I/Os by processing T top-down using the time-forward processing procedure of Section 3 because V_v is stored with v in T and $\{x_v, y_v\} = V_v \cap V_{p(v)}$. The next lemma, which is easy to prove by induction on the size of $T(v)$, shows that the vertex weights $\omega(v)$, $v \in T$, have the desired property.

Lemma 11 *The weight of tree T is $\omega(T) = \omega(G)$. For every vertex $v \neq r$, $\omega(T(v)) = \omega(G(v) - \{x_v, y_v\})$.*

Phase 2: Computing a small edge-separator of T. The next step of our algorithm computes a set C of edges of T such that none of the connected components $T_0, T_1, \ldots, T_k$ of $T - C$ has weight exceeding ϵ, except possibly the component T_0 that contains the root r of T; if $\omega(T_0) > \epsilon$, then $T_0 = (\{r\}, \emptyset)$. At this point, we have to assume that no vertex in T, except the root r, has weight exceeding $\epsilon/2$. We show how to ensure this condition when discussing Phase 3 of our algorithm.

In order to compute C, we process T bottom-up and apply the following rules: When visiting a leaf v of T, we define $\omega'(v) = \omega(v)$. At an internal node $v \neq r$ of T with children w_1 and w_2, we proceed as follows: If one of the children, say w_2, does not exist, we define $\omega'(w_2) = 0$. If $\omega(v) + \omega'(w_1) + \omega'(w_2) \leq \epsilon/2$, we define $\omega'(v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\epsilon/2 < \omega(v) + \omega'(w_1) + \omega'(w_2) \leq \epsilon$, we define $\omega'(v) = 0$ and add edge $\{v, p(v)\}$ to C. If $\epsilon < \omega(v) + \omega'(w_1) + \omega'(w_2)$, we define $\omega'(v) = 0$ and add edges $\{v, p(v)\}$ and $\{v, w_1\}$ to C. If $v = r$, v has a single child w. If $\omega(v) + \omega'(w) > \epsilon$, we add edge $\{v, w\}$ to C.

This procedure takes $O(\text{scan}(N))$ I/Os using the time-forward processing algorithm of Section 3. Instead of producing the list C explicitly, we label every edge in T as either being contained in C or not. This representation makes Phase 3 easier. The following two lemmas show that C is almost an ϵ-edge separator of T, whose size is $\lfloor 2/\epsilon \rfloor$.

Lemma 12 *Let $T_0, \ldots, T_k$ be the connected components of $T - C$, and let $r \in T_0$. Then $\omega(T_i) \leq \epsilon$, for $1 \leq i \leq k$. If $\omega(T_0) > \epsilon$, then $T_0 = (\{r\}, \emptyset)$.*

Proof. For every vertex $v \in T$, let T_v be the component T_i such that $v \in T_i$. We show that $\omega(T(v) \cap T_v) \leq \epsilon$, for every vertex $v \neq r$ in T. This implies that $\omega(T_i) \leq \epsilon$, for $1 \leq i \leq k$, because, for the root r_i of T_i, $T(r_i) \cap T_i = T_i$.

In order to prove this claim, we show the following stronger result: If v is the root of T_v, then $\omega(T(v) \cap T_v) \leq \epsilon$; if v is not the root of T_v, then $\omega(T(v) \cap T_v) \leq \epsilon/2$ and $\omega'(v) = \omega(T(v) \cap T_v)$. We prove this claim by induction on the size of tree $T(v)$. If $|T(v)| = 1$, v is a leaf, and $T(v) \cap T_v = (\{v\}, \emptyset)$, so that $\omega(T(v) \cap T_v) = \omega(v) \leq \epsilon/2$.

If $|T(v)| > 1$, v is an internal vertex with children w_1 and w_2. If $T_v = T_{w_1} = T_{w_2}$, then neither of w_1 and w_2 is the root of T_v. By the induction hypothesis, this implies that $\omega'(w_1) = \omega(T(w_1) \cap T_v) \leq \epsilon/2$ and $\omega'(w_2) = \omega(T(w_2) \cap T_v) \leq \epsilon/2$. This implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\omega(T(v) \cap T_v) > \epsilon$, we would have added edge $\{v, w_1\}$ to C, contradicting the assumption that $T_v = T_{w_1}$. Thus, $\omega(T(v) \cap T_v) \leq \epsilon$. If $\omega(T(v) \cap T_v) > \epsilon/2$, our algorithm adds edge $\{v, p(v)\}$ to C, so that v is the root of T_v.

If $T_v = T_{w_2} \neq T_{w_1}$, w_2 is not the root of T_v. Thus, $\omega'(w_2) = \omega(T(w_1) \cap T_v) \leq \epsilon/2$. This immediately implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega(T(w_2) \cap T_v) \leq \epsilon$. If $\omega(T(v) \cap T_v) > \epsilon/2$, we add edge $\{v, p(v)\}$ to C, so that v is the root of T_v.

Finally, if T_v, T_{w_1}, and T_{w_2} are all distinct, vertices w_1 and w_2 are the roots of trees T_{w_1} and T_{w_2}, so that our algorithm sets $\omega'(w_1) = \omega'(w_2) = 0$. This implies that $\omega'(v) = \omega(v) = \omega(T(v) \cap T_v)$.

In order to show that $T_0 = (\{r\}, \emptyset)$ if $\omega(T_0) > \epsilon$, we argue as follows: Let w be the child of r in T. If $T_r = T_w$, then vertex w is not the root of T_w, so that $\omega'(w) = \omega(T(w) \cap T_r)$. If $\omega(T_r) = \omega(r) + \omega'(w) > \epsilon$, our algorithm would have added edge $\{r, w\}$ to C. Thus, $\omega(T_r) \leq \epsilon$ if $T_r = T_w$. If $T_r \neq T_w$, $T_r = (\{r\}, \emptyset)$, because w is the only child of r. □

Lemma 13 $|C| \leq \lfloor 2\omega(T)/\epsilon \rfloor$.

Proof. In order to show the lemma, we charge the edges in C to individual subgraphs T_i of $T - C$ or to pairs of subgraphs $\{T_i, T_j\}$ of $T - C$. Every subgraph T_i is charged at most once, either individually or as part of a pair of subgraphs. Every individual graph that is charged has weight at least $\epsilon/2$ and is charged for one edge in C. Every pair of graphs that is charged has weight at least ϵ and is charged for two edges in C. Thus, on average, we charge at most one edge per $\epsilon/2$ units of weight. Thus, $|C| \leq \lfloor 2\omega(T)/\epsilon \rfloor$. We have to show how to distribute the charges.

Consider the way edges in C are added to C. An edge $\{v, p(v)\}$ that is added to C while processing v is added either alone or along with an edge $\{v, w_1\}$, where w_1 is a child of v. In the former case, $\epsilon/2 < \omega(T_v) \leq \epsilon$, and we charge edge $\{v, p(v)\}$ to graph T_v. If edge $\{v, p(v)\}$ is added to C along with edge $\{v, w_1\}$, $\epsilon < \omega(T_v) + \omega(T_{w_1})$. Then we charge these two edges to the pair $\{T_v, T_{w_1}\}$. Every subgraph T_i with root r_i is charged only for edge $\{r_i, p(r_i)\}$. Thus, every subgraph T_i is charged at most once. If edge $\{r, w\}$ is added to C while processing the child w of the root r, then this edge is already covered by this charging scheme. Otherwise, edge $\{r, w\}$ is added because $\omega(r)+\omega'(w) > \epsilon$. In this case, we charge the edge to T_0. Component T_0 is never charged for any other edges, as its root does not have a parent. □

Phase 3: Computing the vertex-separator of *G*. In order to compute an ϵ-separator of G, we have to ensure first that no vertex in the dual tree T, except possibly the root, has weight more than $\epsilon/2$. To ensure this, it is sufficient to guarantee that no vertex in G has weight exceeding $\epsilon/2$, as every vertex in T, except the root obtains its weight from a single vertex in G. Thus, we compute the vertex separator as the union of two sets S_1 and S_2. Set S_1 contains all vertices of weight more than $\epsilon/2$ in G. We set $\omega(v) = 0$, for every vertex $v \in S_1$ and compute the edge separator C of T w.r.t. these modified vertex weights. Every edge $e \in C$ corresponds to an edge $e^* = \{x, y\}$ in G. We add vertices x and y to S_2.

By Lemma 13, $|C| \leq \left\lfloor \frac{2\omega(T)}{\epsilon} \right\rfloor = \left\lfloor \frac{2(\omega(G)-\omega(S_1))}{\epsilon} \right\rfloor \leq \left\lfloor \frac{2(\omega(G)-\frac{\epsilon}{2}|S_1|)}{\epsilon} \right\rfloor = \left\lfloor \frac{2\omega(G)}{\epsilon} \right\rfloor - |S_1|$. Thus, $|S_2| \leq 4\omega(G)/\epsilon - 2|S_1|$, so that $|S| \leq |S_1| + |S_2| \leq 4\omega(G)/\epsilon$. The computation of S_1 requires $O(\text{scan}(N))$ I/Os. Given C, set S_2 is easily computed in $O(\text{scan}(N))$ I/Os using a preorder traversal of T. We have to show that S is an ϵ-separator of G.

Lemma 14 *Vertex set S is an ϵ-separator of G.*

Proof. Let $T_0, \ldots, T_k$ be the connected components of $T - C$. Let $G_0, \ldots, G_k$ be the subgraphs of G such that G_i is the union of all triangles $\Delta(v)$, $v \in T_i$. We show that every connected component of $G - S$ is completely contained in a subgraph G_i and that $\omega(G_i - S) \leq \epsilon$.

The first claim is easy to see. Indeed, all edges $e = \{v, w\} \in T$, $v \in T_i$, $w \notin T_i$, are in C, so that the endpoints of their dual edges are in S; hence, there is no path from a vertex not in G_i to a vertex in G_i that does not contain a vertex in S.

In order to prove the second claim, we observe that, for $i = 0$, $\omega(T_0) = \omega(G_0)$, by the definition of weights $\omega(v)$, $v \in T$. If $\omega(T_0) \leq \epsilon$, then $\omega(G_0 - S) \leq \epsilon$. Otherwise, $T_0 = (\{r\}, \emptyset)$ and $G_0 - S$ contains at most one vertex, whose weight is no more than $\epsilon/2$.

For $i > 0$, let r_i be the root of tree T_i. Then $\omega(T_i) = \omega(G_i - \{x_{r_i}, y_{r_i}\}) \geq \omega(G_i - S)$, as $x_{r_i}, y_{r_i} \in S$. But, $\omega(T_i) \leq \epsilon$, by Lemma 12. □

7 DFS, BFS, and Single-Source Shortest Paths

The problem of computing a DFS-tree of an embedded outerplanar graph G can easily be solved in $O(\text{scan}(N))$ I/Os, provided that the choice of the root of the tree is left to the algorithm: We choose the vertex r with $\nu(r) = 1$ as the root of the tree. A DFS-tree with this root is already encoded in the list $\mathcal{A}$ representing the embedding of G and can easily be extracted in $O(\text{scan}(N))$ I/Os. If the DFS-tree has to have a particular root r, a simple stack algorithm can be used to extract the desired DFS-tree from the embedding of G. However, while the algorithm is simple, its correctness proof is tedious.

Theorem 6 *Given a list $\mathcal{A}$ that represents an outerplanar embedding of a connected outerplanar graph G with N vertices, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a DFS-tree for G.*

In the rest of this section, we present a linear-I/O algorithm to solve the single-source shortest path problem for an embedded connected outerplanar graph G. Since breadth-first search is the same as single-source shortest paths after assigning unit weights to the edges of G, this algorithm can also be used to compute a BFS-tree for G. The algorithm presented here differs from the one presented in [22] in a number of ways. Most importantly, we consider the algorithm presented here to be much simpler than the one in [22] and the algorithm in [22] could not be used to solve the SSSP-problem. We describe our algorithm assuming that graph G is undirected. However, it generalizes in a straightforward manner to the directed case.

The first step in our algorithm is to triangulate the given graph G. Let Δ be the resulting triangulation. To ensure that the shortest path between two vertices in Δ is the same as the shortest path between these two vertices in G, we give all edges that are added to G in order to obtain Δ infinite weight. The triangulation algorithm of Section 5 can easily be augmented to maintain edge weights. Now recall that the triangulation algorithm first computes a list $\mathcal{T}$ of the triangles in Δ sorted according to a postorder traversal of the dual tree T of Δ. This representation of Δ is more useful for our SSSP-algorithm than the list $\mathcal{D}$ that represents the embedding of Δ.

We denote the weight of an edge e by $\omega(e)$. Given a source vertex s, we proceed as follows: We choose a root vertex s' of T so that the triangle $\Delta(s')$ has s as one of its vertices. For every edge e of T, let T_e be the subtree of T induced by all vertices v so that the path from s' to v in T contains edge e; that is, intuitively, tree T_e is connected to the rest of T through edge e. Let $\bar{T}_e$ be the subtree $T - T_e$ of T. Let G_e be the union of triangles $\Delta(v)$, for all vertices $v \in T_e$; graph $\bar{G}_e$ is defined analogously for $\bar{T}_e$. Then $G = G_e \cup \bar{G}_e$ and $G_e \cap \bar{G}_e = (\{x_e, y_e\}, \{e^*\})$; that is, G_e and $\bar{G}_e$ share only the dual edge e^* of e; the endpoints x_e and y_e of e^* form a separator of G. Any simple path P from s to a vertex $v \in \bar{G}_e$ either does not contain a vertex of $G_e - \{x_e, y_e\}$, or it contains both x_e and y_e. These observations suggest the following strategy:

First we compute weights $\omega'(e)$ for the edges of G. If there exists an edge $e^* \in T$ such that e is dual to e^*, we define $\omega'(e)$ as the length of the shortest

path in G_{e^*} between the endpoints x_{e^*} and y_{e^*} of e. Otherwise, we define $\omega'(e) = \omega(e)$. In the second step, let x_v be the vertex of T closest to s so that $v \in \Delta(x_v)$, for all $v \in G$. Let w_1 and w_2 be the two children of x_v. Then we compute the distance $d'(s, v)$ from s to v in the graph $\bar{G}_{\{x_v,w_1\}} \cap \bar{G}_{\{x_v,w_2\}}$ w.r.t. the weights $\omega'(e)$ computed in the first step. As we show below, $d'(s, v) = d_G(s, v)$, so that this strategy correctly computes the distances of all vertices in G from s. At the end of this section, we show how to augment the algorithm so that it computes a shortest-path tree.

Rooting *T*. In order to implement the remaining steps in the algorithm in $O(\text{scan}(N))$ I/Os, using the time-forward processing procedure from Section 3, we need to ensure that tree T is rooted at a vertex s' such that the source s of the shortest-path computation is a vertex of $\Delta(s')$, and that the vertices of T are sorted according to a preorder (or postorder) traversal of T. After applying the triangulation algorithm of Section 5, the vertices of T are stored in postorder, but the current root of T may not represent a triangle that has s on its boundary.

As the reversal of a postorder numbering of T is a preorder numbering of T, we assume w.l.o.g. that the vertices of T are stored in preorder. The first step of our algorithm is to extract an Euler-tour of T, that is, a traversal of tree T such that every edge of T is traversed exactly twice, once from the parent to the child and once the other way. Let $e_1, \ldots, e_k$ be the list of edges in the Euler tour. Then we represent the tour by a list $\mathcal{E}$ containing the source vertices of edges $e_1, \ldots, e_k$ in order. We transform list $\mathcal{E} = \langle x_1, \ldots, x_t \rangle$ into a list $\mathcal{E}' = \langle x_k \ldots, x_t, x_1, \ldots, x_{k-1} \rangle$, where $x_k = s'$. List $\mathcal{E}'$ represents an Euler tour of T starting at vertex s'. The final step of our algorithm is to extract the vertices of T in preorder. Algorithm 7 contains the details of this procedure.

In order to show the correctness of Algorithm 7, we show that the list $\mathcal{E}$ produced by Lines 1–21 describes an Euler tour of T starting at the current root r of T. This implies that list $\mathcal{E}'$ represents an Euler tour of T starting at s'. Using this fact, we show that Lines 23–32 produce a list $\mathcal{T}'$ that stores the vertices of T, when rooted at s', in preorder.

Lemma 15 *The list $\mathcal{E}$ computed by Lines 1–21 of Algorithm 7 describes an Euler tour of T that starts at vertex r.*

Proof. In this proof, we refer to each vertex $v \in T$ by its preorder number. In order to prove the lemma, we show that the for-loop of Lines 2–14 maintains the following invariant: Stack S stores the vertices on the path from the root r to the current vertex v. List $\mathcal{E}$ represents a traversal of T from s to v such that, for every vertex $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is traversed twice, once in each direction, and for every vertex $u \in S$, $u \neq r$, edge $\{u, p(u)\}$ is traversed exactly once, from $p(u)$ to u. This implies that, after the for-loop is finished, list $\mathcal{E}$ represents a tour from s' to the last vertex in T such that all edges are traversed twice, except the edges between the vertices on stack S. These edges have to be traversed once more, from children towards their parents. This is

Algorithm 7 Rooting tree T at vertex s'.

Input: A list $\mathcal{T}$ that stores the vertices of the dual tree T of Δ rooted at vertex r with $\nu(r) = 1$ in preorder.

Output: A list $\mathcal{T}'$ that stores the vertices of the dual tree T of Δ rooted at vertex s' in preorder.

Initialize stack S to be empty.
for each vertex $v \in \mathcal{T}$ **do**
  if v is the first vertex (i.e., the current root of T) **then**
    PUSH(S, v)
    Append v to $\mathcal{E}$
  else
    while the top of stack S is not equal to $p(v)$ **do**
      POP(S)
      Append the top of stack S to $\mathcal{E}$
    end while
    PUSH(S, v)
    Append v to $\mathcal{E}$
  end if
end for
while S is not empty **do**
  POP(S)
  if S is not empty **then**
    Append the top of stack S to $\mathcal{E}$
  end if
end while
Remove the last element from $\mathcal{E}$
Scan $\mathcal{E}$ to find the first index k in list $\mathcal{E} = \langle x_1, \ldots, x_t \rangle$ such that $x_k = s'$. While scanning, append elements $x_1, \ldots, x_{k-1}$ to a queue Q and delete them from $\mathcal{E}$. Once x_k has been found, move elements $x_1, \ldots, x_{k-1}$ from Q to the end of the list $\langle x_k, \ldots, x_t \rangle$. Call the resulting list $\mathcal{E}' = \langle x'_1, \ldots, x'_t \rangle$.
for each element $x'_i \in \mathcal{E}'$ **do**
  if S is empty or the top of stack S is not of the form (x'_i, x'_j) **then**
    Append the pair (x'_i, x'_{i-1}) to list $\mathcal{T}'$. $\{x'_{i-1}$ is the parent of x'_i in the tree T rooted at s'; for x'_1, we define $x'_0 =$ **nil**.$\}$
    if $x'_{i+1} \neq x'_{i-1}$ **then**
      PUSH$(S, (x'_i, x'_{i-1}))$
    end if
  else if $x'_{i+1} = x'_j$ **then**
    POP(S)
  end if
end for

accomplished in the while-loop in Lines 15–20. The root s' is added to the end of $\mathcal{E}$ by this while-loop; hence, we have to remove it, in order not to store vertex s' one more time in the list than it is visited in the Euler tour. We have to show the invariant of the for-loop.

We show the invariant by induction on the preorder number of v. If $v = 1$, then $v = r$. We execute Lines 4 and 5 of the loop. As a result, stack S stores the path from r to r. There are no vertices $v < r$, and there are no edges between vertices on S. Thus, the remainder of the invariant is trivially true.

If $v > 1$, we execute Lines 7–12. In Lines 7–9, we remove vertices from the stack and append their parents to $\mathcal{E}$ until the top of the stack stores the parent of v. This is equivalent to traversing edge $\{u, p(u)\}$, for each removed vertex u, so that we maintain the invariant that for each vertex $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is traversed twice by the tour described by $\mathcal{E}$. After Line 9, stack S represents a path from r to $p(v)$, and the tour described by $\mathcal{E}$ traverses T from r to $p(v)$ and visits all edges between vertices on stack S exactly once. Lines 11 and 12 add v to S and $\mathcal{E}$, so that stack S now represents the path from r to v, and edge $\{p(v), v\}$ is traversed by the tour described by list $\mathcal{E}$.

In order to complete the proof, we need to show that, for each vertex v, $p(v) \in S$ before the iteration for vertex v is entered. In order to show this, we have to show that $p(v)$ is an ancestor of $v - 1$. If this were not true, then $p(v) < v - 1 < v$, v is in the subtree of T rooted at $p(v)$, and $v - 1$ is not. This contradicts the fact that a preorder numbering assigns consecutive numbers to the vertices in each subtree rooted at some vertex x. □

Lemma 16 *List $\mathcal{T}'$ stores the vertices of tree T, rooted at s', sorted in preorder. Every vertex in $\mathcal{T}'$ is labeled with its parent in T.*

Proof. By definition, a preorder numbering of T is a numbering of the vertices in T according to the order of their first appearances in an Euler tour of T. Thus, we only have to show that Lines 23–32 of Algorithm 7 extract the first appearance of each vertex from $\mathcal{E}'$. Also, the first appearance of every vertex v in an Euler tour of T is preceded by an appearance of the parent of v. Thus, if we extract only the first appearance of each vertex, we also compute its parent correctly.

A vertex is extracted from $\mathcal{E}'$ if it is not equal to the top of the stack. This is true for the first appearance of each vertex in $\mathcal{E}'$, as we push a vertex on the stack only when it is visited. We have to show that every vertex of T is extracted exactly once. In order to do that, we show that each but the first appearance of every vertex v finds v on the top of the stack, so that v is not appended to $\mathcal{T}'$ again. The proof is by induction on the size of the subtree $T(v)$ of T rooted at v.

If $|T(v)| = 1$, v is a leaf, and v appears only once in any Euler tour of T. Also, v is never pushed on the stack S, because its successor in $\mathcal{E}'$ is $p(v)$.

If $|T(v)| > 1$, v is an internal node. Let $w_1, \ldots, w_k$ ($k \leq 3$) be the children of v in T. Then, by the induction hypothesis, each but the first appearance of w_i finds w_i on the top of the stack. In particular, the top of S looks like $\langle w_i, v, \ldots \rangle$

when w_i is visited, except during the first visit. Now, the first appearance of v precedes w_1, while every other appearance of v immediately succeeds the last appearance of one of v's children w_i. As each such last appearance of a child of v is succeeded by $p(w_i) = v$, w_i is removed from S when visiting the last appearance of w_i, so that before visiting the next appearance of v, v is on the top of stack S. This proves the correctness of Algorithm 7. □

Pruning subtrees. Having changed the order of the vertices in T so that they are sorted according to a preorder numbering of T starting at s', we now move to the second phase of our algorithm, which computes a weight $\omega'(e)$, for every edge $e = \{x, y\} \in G$, so that $\omega'(e) = d_{G_{e^*}}(x, y)$.

For every exterior edge e of G, we define $\omega'(e) = \omega(e)$. Next we compute the edge weights $\omega'(e)$, for all diagonals e of G. We do this by processing T bottom-up. For a vertex $v \neq s'$ of T, let $e = \{v, p(v)\}$, $e^* = \{x_e, y_e\}$, and $z \in V_v \setminus \{x_e, y_e\}$. Then we define $\omega'(e^*) = \min(\omega(e^*), \omega'(\{x_e, z\}) + \omega'(\{y_e, z\}))$.

This computation takes $O(\text{scan}(N))$ I/Os using the time-forward processing procedure of Section 3. The following lemma shows that it produces the correct result.

Lemma 17 *For every edge $e \in T$, $\omega'(e^*) = d_{G_e}(x_e, y_e)$.*

Proof. We prove this claim by induction on the size of T_e. If $|T_e| = 1$, then $e = \{v, p(v)\}$, where v is a leaf of T. In this case, $G_e = \Delta(v)$, and the shortest path from x_e to y_e in G_e is either the edge $e^* = \{x_e, y_e\}$ itself, or it is the path $P = (x_e, z, y_e)$, where z is the third vertex of $\Delta(v)$. Edges $e_1 = \{x_e, z\}$ and $e_2 = \{y_e, z\}$ are exterior edges of G, so that $\omega'(e_1) = \omega(e_1)$ and $\omega'(e_2) = \omega(e_2)$. Thus, $\omega'(e^*)$ is correctly computed as the minimum of the weights of edge e^* and path P.

If $|T_e| > 1$, let $e = \{v, p(v)\}$, let w_1 and w_2 be the children of v in T, and let e_1 and e_2 be the edges $\{v, w_1\}$ and $\{v, w_2\}$. Let $e_1^* = \{x_{e_1}, y_{e_1}\}$ and $e_2^* = \{x_{e_2}, y_{e_2}\}$, where $x_{e_1} = x_e$, $y_{e_2} = y_e$, and $y_{e_1} = x_{e_2}$. If vertex w_2 does not exist, we assume that e_2 is a tree-edge connecting v to an artificial vertex in the outer face of G and $G_{e_2} = (\{x_{e_2}, y_{e_2}\}, \{\{x_{e_2}, y_{e_2}\}\})$. Then $G_e = G_{e_1} \cup G_{e_2} \cup (\{x_e, y_e\}, e^*)$. By the induction hypothesis, $\omega'(\{x_e, y_{e_1}\}) = d_{G_{e_1}}(x_e, y_{e_1})$ and $\omega'(\{x_{e_2}, y_e\}) = d_{G_{e_2}}(x_{e_2}, y_e)$. Thus, $\omega'(e^*) = \min\{\omega(e^*), \omega'(\{x_e, y_{e_1}\}) + \omega'(\{x_{e_2}, y_e\}) = d_{G_e}(x_e, y_e)$. □

Computing distances from the source. In order to compute $d_G(s, v)$, for every vertex $v \in G$, we process tree T top-down, maintaining the invariant that, after processing vertex $v \in T$, the distances $d(s, x)$ have been computed for all vertices $x \in \Delta(v)$. At the root s' of T, we have that $s \in \Delta(s')$. For the other two vertices v and w of $\Delta(s')$, we compute $d_G(s, v) = \min(\omega'(\{s, v\}), \omega'(\{s, w\}) + \omega'(\{v, w\}))$ and $d_G(s, v) = \min(\omega'(\{s, v\}), \omega'(\{s, w\}) + \omega'(\{v, w\}))$. At any other vertex $v \in T$, let $e = \{v, p(v)\}$. Then $\Delta(v)$ has vertices x_e, y_e, and a third vertex z. After processing the parent of v, $d_G(s, x_e)$ and $d_G(s, y_e)$ are known. Thus, we compute $d_G(s, z) = \min(d_G(s, x_e) + \omega'(\{x_e, z\}), d_G(s, y_e) + \omega'(\{y_e, z\}))$.

Again, this procedure takes $O(\text{scan}(N))$ I/Os using the time-forward processing procedure from Section 3. The following lemma shows that it produces the correct result.

Lemma 18 *The above procedure computes $d_G(s,v)$ correctly, for all $v \in G$.*

Proof. Observe that the distance $d_G(s,v)$ is computed when processing a node $x_v \in T$ such that $v \in \Delta(x_v)$, and $v \notin \Delta(p(x_v))$. We prove by induction on $d_T(s', x_v)$ that $d_G(s,v)$ is computed correctly. If $d_T(s', x_v) = 0$, then $v \in \Delta(s')$. If $s = v$, then $d_G(s,v) = 0$ is computed correctly by our algorithm. Otherwise, let the vertex set of $\Delta(s')$ be $\{s, v, x\}$, and let the edges of $\Delta(s')$ be $e_1^* = \{s,v\}$, $e_2^* = \{s,x\}$, and $e_3^* = \{x,v\}$. By Lemma 17, $\omega'(e_1^*) = d_{G_{e_1}}(s,v)$, $\omega'(e_2^*) = d_{G_{e_2}}(s,x)$, and $\omega'(e_3^*) = d_{G_{e_3}}(x,v)$. Thus, $d_G(s,v) = \min(d_{G_{e_1}}(s,v), d_{G_{e_2}}(s,x) + d_{G_{e_3}}(x,v)) = \min\{\omega'(e_1^*), \omega'(e_2^*) + \omega'(e_3^*))$, as computed by our algorithm.

If $d_T(s', x_v) = k > 0$, assume that the distances are computed correctly for all vertices w with $d_T(s', x_w) < k$. Let $e = \{x_v, p(x_v)\}$. Then the vertex set of $\Delta(x_v)$ is $\{x_e, y_e, v\}$, and $d_T(s', x_{x_e}) < k$ and $d_T(s', x_{y_e}) < k$. Thus, the distances $d(s, x_e)$ and $d(s, y_e)$ have already been computed correctly. Let $e_1^* = \{x_e, v\}$, and $e_2^* = \{y_e, v\}$. The shortest path from s to v is either the concatenation of the shortest path from s to x_e, followed by the shortest path from x_e to v in G_{e_1}, or the shortest path from s to y_e, followed by the shortest path from y_e to v in G_{e_2}. Thus, $d_G(s,v) = \min(d_G(s,x_e) + d_{G_{e_1}}(x_e, v), d_G(s,y_e) + d_{G_{e_2}}(y_e, v)) = \min(d_G(s,x_e) + \omega'(e_1^*), d_G(s,y_e) + \omega'(e_2^*))$, as computed by our algorithm. □

Extracting a shortest-path tree. In order to extract a shortest-path tree from s to all other vertices in G, we augment our algorithm as follows. The first phase, which roots tree T at root s', does not need to be changed. We augment the second phase as follows:

For every tree edge $e \in T$, let $e = \{v, p(v)\}$, and $z \in \Delta(v) - \{x_e, y_e\}$. Depending on whether we computed $\omega'(e^*)$ as $\omega(e^*)$ or $\omega'(\{x_e, z\}) + \omega'(\{z, y_e\})$, the shortest path from x_e to y_e in G_e is either edge e^* or the concatenation of the shortest paths from x_e to z and from z to y_e in G_e. We store a flag with edge e distinguishing between these two possibilities.

In the third phase of our algorithm, we proceed as follows: Let $d_G(s,v) = d_G(s,x) + \omega'(\{x,v\})$, and assume that we have not computed a parent for vertex v yet. If $\{x,v\}$ is an external edge of G, we add edge $\{x,v\}$ to the shortest-path tree and set $p(v) = x$. Otherwise, there are two possibilities. If $\omega'(\{x,v\}) = \omega(\{x,v\})$, we add edge $\{x,v\}$ to the shortest-path tree, set $p(v) = x$, and inform all descendants w of x_v such that $v \in \Delta(w)$ that the parent of v has already been computed. Otherwise, we inform the descendant w of x_v such that $\{x,v\} = \{w, x_v\}^*$ that v's parent lies in $G_{\{w,x_v\}}$ and still needs to be computed.

The correctness of this procedure is easily verified, given that the above algorithm computes distances $d_G(s,v)$ correctly. Thus, we have shown the following theorem.

Theorem 7 *Given a list $\mathcal{A}$ representing an outerplanar embedding $\hat{G}$ of an outerplanar graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}^+$, it takes $O(\text{scan}(N))$ I/Os to compute a BFS-tree for G or to solve the single-source shortest path problem for G.*

Proof. The correctness of our algorithm follows from the above discussion. All but the first step of the algorithm process a tree T of size $O(N)$ using the linear-I/O time-forward processing procedure of Section 3. Thus, they take $O(\text{scan}(N))$ I/Os. As for the first step, Lines 1–21 of Algorithm 7 read list $\mathcal{T}$ and produce list $\mathcal{E}$. The size of $\mathcal{E}$ is bounded by the number of stack operations performed, as we add at most one element to $\mathcal{E}$ per stack operation. The number of POP operations is bounded by the number of PUSH operations, which in turn is bounded by $|\mathcal{T}|$, as we push each element in $\mathcal{T}$ on the stack at most once. Thus, $|\mathcal{E}| = O(N)$, and we perform $O(N)$ stack operations. Hence, Lines 1–21 take $O(\text{scan}(N))$ I/Os. Given that $|\mathcal{E}| = O(N)$, $|\mathcal{E}'| = O(N)$, and Line 22 takes $O(\text{scan}(N))$ I/Os. In Lines 23–32, list $\mathcal{E}'$ is scanned, and list $\mathcal{T}'$ is written. Every element in $\mathcal{E}'$ causes at most one element to be appended to $\mathcal{T}'$ and at most one element to be pushed on the stack. Thus, $|\mathcal{T}'| = O(N)$, and we perform $O(N)$ stack operations. Hence, Lines 23–32 also take $O(\text{scan}(N))$ I/Os. This shows that the whole algorithm takes $O(\text{scan}(N))$ I/Os to solve the single-source shortest path problem. In order to compute a BFS-tree, we give all edges in G unit weight. □

8 Lower Bounds

In this section, we address the question whether the algorithms presented in this paper are optimal. Note that all the problems in this paper require at least $\Omega(\text{scan}(N))$ I/Os. Given an outerplanar embedding, we present optimal $O(\text{scan}(N))$ I/Os algorithm for these problems. However, if no outerplanar embedding of the graph is given, we spend $O(\text{perm}(N))$ I/Os to solve any of the problems discussed in this paper, as we first embed the graph and then apply one of our linear-I/O algorithms to solve the actual problem. Now the question is whether the embedding step can be avoided, in order to obtain true linear-I/O algorithms for BFS, DFS, SSSP, triangulation, or outerplanar separators.

In order to be able to show a lower bound for any of these problems, we have to define exactly how the output of an algorithm solving the problem is to be represented. For most of the problems discussed here, such a representation of the output is far from well-defined. For instance, a graph is said to be embedded if the circular order of the edges incident to each vertex is given. How this order is to be represented is left to the particular algorithm. We may represent this order by numbering or sorting the edges clockwise around the vertex, or by having each edge store pointers to its successors clockwise around each of its endpoints. The output of a BFS-algorithm may be a labeling of every vertex with its distance to the root of the BFS-tree, or just a representation of the BFS-tree by computing for every vertex, its parent in the BFS-tree.

Even though we believe that, if no embedding of the graph is given as part of the input, $\Omega(\text{perm}(N))$ is a lower bound on the number of I/Os required to compute an embedding, BFS-tree, DFS-tree, or shortest-path tree of an outerplanar graph G, independent of which representation of the output is chosen, we are only able to prove such a lower bound, if we place certain restrictions on the output representation of the respective algorithm. These restrictions are satisfied by our algorithms. We discuss these restrictions next.

For each vertex $v \in G$, let $A(v)$ be its adjacency list. Then we require that an algorithm computing an outerplanar embedding of G either numbers the vertices in $A(v)$ from 1 to $|A(v)|$ clockwise or counterclockwise around v, or produces a representation that can be transformed into this representation of the embedding in $o(\text{perm}(N))$ I/Os. Our algorithm produces lists $A(v)$ sorted counterclockwise around v. Scanning all lists $A(v)$, we can transform such a representation into a numbering of the vertices in $A(v)$.

A BFS, DFS, or SSSP-algorithm is required to label every vertex $v \in G$ with the length of the shortest path in T from the root of T to v, or produce an output that allows the computation of such distance labels in $o(\text{perm}(N))$ I/Os. Our algorithms represent the computed spanning trees T by lists $\mathcal{T}$ storing their vertices in preorder. Using the linear-I/O time-forward processing procedure of Section 3, we can then easily compute distance labels for the vertices of T.

All our lower bounds use a rather straightforward linear-I/O reduction from list-ranking to the problem whose lower bound we want to show. This implies that the problem requires $\Omega(\text{perm}(N))$ I/Os to be solved, as list-ranking has an $\Omega(\text{perm}(N))$ I/O lower bound [9]. List-ranking is the following problem:

> Given a list $L = \langle x_1, \ldots, x_N \rangle$, where $\text{succ}(x_i) = x_{i+1}$, for $1 \leq i < N$, label the vertices of L with their distances from the tail of the list, that is, compute a labeling $\delta : L \to \mathbb{N}$, where $\delta(x_i) = N - i$. We call δ the *ranking* of list L.

Lemma 19 *Given a list L of size N, it takes $O(\text{scan}(N))$ I/Os to construct an outerplanar graph G_L of size $O(N)$ and extract a ranking δ of L from an outerplanar embedding of G_L.*

Proof. We define graph $G_L = (V_L, E_L)$ as follows: The vertex set of G_L is defined as the set $V_L = \{x_1, \ldots, x_N, y\}$. The edge set of G_L is defined as $E_L = \{\{x_i, x_{i+1} : 1 \leq i < N\} \cup \{\{y, x_i\} : 1 \leq i \leq N\}$. Graph G_L can easily be constructed from list L in $O(\text{scan}(N))$ I/Os. Figure 6 shows an outerplanar

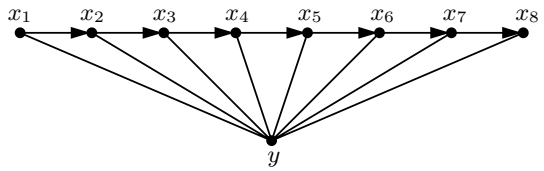

Figure 6: The proof of the lower bound for outerplanar embedding.

embedding of G_L. By Lemma 4, this is the only possible embedding of G_L, except for flipping the whole graph. Thus, an algorithm numbering the vertices in $A(y)$ clockwise around y produces a labeling $\delta' : L \to \mathbb{N}$ such that either $\delta'(x_i) = (i + c) \bmod N$, where $c \in \mathbb{N}$ is a constant, or $\delta'(x_i) = (c - i) \bmod N$. It is straightforward to decide which of the two cases applies and to determine constant c, based on the labels $\delta'(x_1)$ and $\delta'(x_N)$. Then a single scan of the vertices in L suffices to transform the labeling δ' into the desired labeling δ. □

The following simple reduction shows that any algorithm computing a BFS, DFS, or shortest-path tree for an outerplanar graph G requires $\Omega(\text{perm}(N))$ I/Os, even if we leave the choice of the root vertex to the algorithm.

Lemma 20 *Given a list L containing N elements, it takes $O(\text{scan}(N))$ I/Os to extract a ranking δ of L from a BFS-tree, DFS-tree, or shortest-path tree of L, when viewed as an undirected graph G_L.*

Proof. We represent list L as the graph $G_L = (V_L, E_L)$, $V_L = \{x_1, \ldots, x_N\}$, $E_L = \{\{x_i, x_{i+1}\} : 1 \leq i < N\}$. For the SSSP-problem we assign unit weights to the edges of G_L. If the algorithm chooses vertex x_k, for some $1 \leq k \leq N$, as the root of the spanning tree it computes, the vertices of G_L are labeled with their distances $\delta'(x_i)$ from x_k. In particular, $\delta'(x_i) = k - i$ if $1 \leq i \leq k$, and $\delta'(x_i) = i - k$ if $k < i \leq N$. The distance label $\delta'(x_1)$ is sufficient to determine index k. Then it takes a single scan of the vertices in L to transform the labeling δ' into the desired labeling δ. □

Together with the $\Omega(\text{perm}(N))$ I/O lower bound for list-ranking, Lemmas 19 and 20 imply the following result.

Theorem 8 *Given an outerplanar graph $G = (V, E)$ with N vertices, represented as vertex and edge lists, it takes $\Omega(\text{perm}(N))$ I/Os to compute an outerplanar embedding, DFS-tree, or BFS-tree of G, or to solve the single-source shortest path problem for G.*

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, September 1988.

[2] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[3] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447. Springer-Verlag, 2000.

[4] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *Journal of Graph Algorithms and Applications*, 7(2):105–129, 2003.

[5] L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.

[6] L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 261–270, 2003.

[7] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[8] A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575, 2000.

[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[11] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

[12] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 889–890, 1998.

[13] H. N. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *Proceedings of the 22nd Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 151–165. Springer-Verlag, 1996.

[14] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, January 1991.

[15] G. N. Frederickson. Using cellular embeddings in solving all pairs shortest paths problems. *Journal of Algorithms*, 19:45–85, 1995.

[16] G. N. Frederickson. Searching among intervals in compact routing tables. *Algorithmica*, 15:448–466, 1996.

[17] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[18] D. Hutchinson. *Parallel Algorithms in External Memory*. PhD thesis, School of Computer Science, Carleton University, 1999.

[19] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126:55–82, 2003.

[20] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Sumposium on Parallel and Distributed Processing*, pages 169–176, October 1996.

[21] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[22] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pages 307–316. Springer-Verlag, December 1999.

[23] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–90, 2001.

[24] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 372–381, 2002.

[25] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

[26] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th Annual European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 434–445. Springer-Verlag, 2003.

[27] S. L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, December 1979.

[28] J. van Leeuwen. *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*. MIT Press, 1990.

[29] J. S. Vitter. External memory algorithms. *Proc. ACM Symp. Principles of Database Systems*, pages 119–128, 1998.

[30] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[32] H. Whitney. Non-separable and planar graphs. *Transaction of the American Mathematical Society*, 34:339–362, 1932.

[33] N. Zeh. An external-memory data structure for shortest path queries. Master's thesis, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Germany, http://www.cs.dal.ca/∼nzeh/Publications/diplomarbeit.pdf, November 1998.