An External Memory Data Structure for Shortest Path Queries [1]

David Hutchinson [a] Anil Maheshwari [b,2] Norbert Zeh [b]

[a] *Department of Computer Science, Duke University, Durham, North Carolina*
hutchins@cs.duke.edu

[b] *School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada*
{maheshwa,nzeh}@scs.carleton.ca

Abstract

We present results related to satisfying shortest path queries on a planar graph stored in external memory. Let N denote the number of vertices in the graph and $sort(N)$ denote the number of input/output (I/O) operations required to sort an array of length N.

(1) We describe a blocking for rooted trees to support bottom-up traversals of these trees in $O(K/B)$ I/Os, where K is the length of the traversed path. The space required to store the tree is $O(N/B)$ blocks, where N is the number of vertices of the tree and B is the block size.

(2) We give an algorithm for computing a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ for a given embedded planar graph. Our algorithm takes $O(sort(N))$ I/Os, provided that a breadth-first spanning (BFS) tree is given.

(3) We give an algorithm for triangulating embedded planar graphs in $O(sort(N))$ I/Os.

We use these results to construct a data structure for answering shortest path queries on planar graphs. The data structure uses $O\left(N^{3/2}/B\right)$ blocks of external memory and allows for a shortest path query to be answered in $O\left(\left(\sqrt{N}+K\right)/DB\right)$ I/Os, where K is the number of vertices on the reported path and D is the number of parallel disks.

Key words: External-memory algorithms, graph algorithms, shortest paths, planar graphs

[1] A preliminary version appeared in [14].

[2] Research supported by NSERC and NCE GEOIDE.

1 Introduction

Answering shortest path queries in graphs is an important and intensively studied problem. It has applications in communication systems, transportation problems, scheduling, computation of network flows, and geographic information systems (GIS). Typically, an underlying geometric structure is represented by a combinatorial structure, which is often a weighted planar graph.

The motivation to study external memory shortest path problems arose in our GIS research and in particular, with an implementation of the results in [16] for shortest path problems in triangular irregular networks. In this application the given graph represents a planar map, i.e., it is planar and embedded. Quite commonly it is too large to fit into the internal memory of even a large supercomputer. In this case, and in many other large applications, the computation is forced to wait while large quantities of data are transferred between relatively slow external (disk-based) memory and fast internal memory. Thus, the classical internal memory approaches to answering shortest path queries in a planar graph (e.g., [6,8–10,15]) may not work efficiently when the data sets are too large.

1.1 Model of Computation

Unfortunately, the I/O-bottleneck is becoming more significant as parallel computing gains popularity and CPU speeds increase, since disk speeds are not keeping pace [22]. Thus, it is important to take the number of input/output (I/O) operations performed by an algorithm into consideration when estimating its efficiency. This issue is captured in the *parallel disk model* (PDM) [24], as well as a number of other external memory models [5,25]. We adopt the PDM as our model of computation for this paper due to its simplicity, and the fact that we consider only a single processor.

In the PDM, an *external memory*, consisting of D disks, is attached to a machine with an internal memory capable of holding M data items. Each of the 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. The complexity of an algorithm is the number of I/O operations it performs.

1.2 Previous Results

Shortest path problems can be divided into three general categories: (1) computing a shortest path between two given vertices of a graph, (2) computing shortest paths between a given source vertex and all other vertices of a graph (single source shortest paths (SSSP) problem), and (3) computing the shortest paths between all pairs of vertices in a graph (all pairs shortest paths (APSP) problem).

Previous results in the RAM model: In the sequential RAM model, much work has been done on shortest path problems. Dijkstra's algorithm [6], when implemented using Fibonacci heaps [11], is the best known algorithm for the SSSP-problem for general graphs (with nonnegative edge weights). It runs in $O(|E| + |V| \log |V|)$ time, where $|E|$ and $|V|$ are the numbers of edges and vertices in the graph, respectively. The APSP-problem can be solved by applying Dijkstra's algorithm to all vertices of the graph, which results in an $O(|V||E| + |V|^2 \log |V|)$ running time. For planar graphs, an $O\left(N\sqrt{\log N}\right)$-algorithm for the SSSP-problem and an $O(N^2)$-algorithm for the APSP-problem, where $N = |V|$, are given in [9]. A linear-time SSSP-algorithm for planar graphs is presented in [15].

An alternate approach is to preprocess the given graph for online shortest path queries. For graphs for which an $O\left(\sqrt{N}\right)$-separator theorem holds (e.g., planar graphs), an $O(S)$-space data structure ($N \leq S \leq N^2$) that answers distance queries in $O(N^2/S)$ time is presented in [8]. The corresponding shortest path can be reported in time proportional to the length of the reported path. (For planar graphs slightly better bounds are given.)

It is known that every tree or outerplanar graph has a $\frac{2}{3}$-separator of size $O(1)$. In [17] it is shown that every planar graph has a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$, and a linear-time algorithm for finding such a separator is given. Other results include separator algorithms for graphs of bounded genus [1] and for computing edge-separators [7].

Previous results in the PRAM model: A PRAM algorithm for computing a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ for a planar graph is presented in [12]. The algorithm runs in $O\left(\log^2 N\right)$ time and uses $O(N^{1+\epsilon})$ processors, where $\epsilon > 0$ is a constant. In [13] a PRAM algorithm is given that computes a planar separator in $O(\log N)$ time using $O(N/\log N)$ processors, provided that a BFS-tree of the graph is given.

Previous results in external memory: In the PDM, sorting, permuting, and scanning an array of size N take $sort(N) = \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$, $perm(N) = \Theta(\min\{N, sort(N)\})$, and $scan(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os [23,24]. For a comprehensive survey on external memory algorithms, refer to [23]. The only external-memory shortest path algorithm known to us is the SSSP-algorithm in [4], which takes $O\left(\frac{|V|}{D} + \frac{|E|}{DB}\log_{\frac{M}{B}}\frac{|E|}{B}\right)$ I/Os with high probability, on a random graph with random weights. We do not know of previous work on computing separators in external memory; but one can use the PRAM-simulation results in [3] together with the results of [12,13] cited above. Unfortunately, the PRAM simulation introduces $O(sort(N))$ I/Os for every PRAM step, and so the resulting I/O complexity is not attractive for this problem.

1.3 Our Results

The main results of this paper are:

(1) A blocking to store a rooted tree T in external memory so that a path of length K *towards the root* can be traversed in at most $\left\lceil\frac{K}{\tau DB}\right\rceil + 3$ I/Os, for $0 < \tau < 1$. If $\tau \geq \frac{3-\sqrt{5}}{2}$, the blocking uses at most $\left(2 + \frac{2}{1-\tau}\right)\frac{|T|}{B} + D$ blocks of external storage. For $\tau < \frac{3-\sqrt{5}}{2}$, a slight modification of the approach reduces the amount of storage to $\left(1 + \frac{1}{1-2\tau}\right)\frac{|T|}{B} + D$ blocks. For fixed τ, the tree occupies optimal $O(|T|/B)$ blocks of external storage and traversing a path takes optimal $O(K/DB)$ I/Os. Using the best previous result [20], the tree would use the same amount of space within a constant factor, but traversing a path of length K would take $O\left(K/\log_d(DB)\right)$ I/Os, where d is the maximal degree of the vertices in the tree. (See Section 3.)

(2) An external memory algorithm which computes a separator consisting of $O\left(\sqrt{N}\right)$ vertices for an embedded planar graph in $O(sort(N))$ I/Os, provided that a BFS-tree of the graph is given. Our algorithm is based on the classical planar separator algorithm in [17]. The main challenge in designing an external memory algorithm for this problem is to determine a good separator corresponding to a fundamental cycle. (See Section 4.)

(3) An external memory algorithm which triangulates an embedded planar graph in $O(sort(N))$ I/Os. (See Section 5.)

(4) An external memory data structure for answering shortest path queries online. Results 1–3, above, are the main techniques that we use to construct this data structure. Our data structure uses $O\left(N^{3/2}/B\right)$ blocks of external memory and answers online distance and shortest path queries in $O\left(\sqrt{N}/DB\right)$ and $O\left(\left(\sqrt{N}+K\right)/DB\right)$ I/Os, respectively, where K is the number of vertices on the reported path. (See Section 6.)

Our separator and triangulation algorithms may be of independent interest, since graph separators are used in the design of efficient divide-and-conquer graph algorithms and many graph algorithms assume triangulated input graphs.

2 Preliminaries

2.1 Definitions

An *undirected graph* (or *graph* for short) $G = (V, E)$ is a pair of sets V and E, where V is called the *vertex set* and E is called the *edge set* of G. Each edge in E is an unordered pair $\{v, w\}$ of vertices v and w in V. Unless stated otherwise, we use $|G|$ to denote the cardinality of V. In a *directed graph*, every edge is an ordered pair (v, w). In this case, we call v the *source vertex* and w the *target vertex* of edge (v, w). A graph G is *planar* if it can be drawn in the plane so that no two edges intersect, except possibly at their endpoints. Such a drawing defines for each vertex v of G, an order of the edges incident to v clockwise around v. We call G *embedded* if we are given this order for every vertex of G. By Euler's formula, $|E| \leq 3|V| - 6$ for planar graphs.

A *path* from a vertex v to a vertex w in G is a list $p = \langle v = v_0, v_1, \ldots, v_k = w \rangle$ of vertices, where $\{v_i, v_{i+1}\} \in E$ for $0 \leq i < k$. The *length* of path p is the number k of edges in the path. We call p a *cycle* if $v_0 = v_k$. Paths and cycles are defined analogously for directed graphs. A *directed acyclic graph* (DAG) is a directed graph that does not contain cycles of length greater than 0. A graph G is *connected* if there is a path between any two vertices in G. A *subgraph* $G' = (V', E')$ of G is a graph with $V' \subseteq V$ and $E' \subseteq E$. Given a subset $X \subseteq V$, we denote by $G[X] = (X, E[X])$ the subgraph of G *induced by* X, where $E[X] = \{\{v, w\} \in E : \{v, w\} \subseteq X\}$. The graph $G - X$ is defined as $G[V \setminus X]$. The *connected components* of G are the maximal connected subgraphs of G. A *tree* with N vertices is a connected graph with $N - 1$ edges. A *rooted tree* is a tree with a distinguished root vertex. The *level* or *depth* of a vertex v in a rooted tree is the number of edges on the path from v to the root. For an edge $\{v, w\}$ in a tree T we say that v is w's *parent* and w is v's *child* if w's depth in T is greater than v's. A vertex v is an *ancestor* of a vertex w, and w is a *descendant* of v, if v is w's parent or it is an ancestor of w's parent. A common ancestor of two vertices v and w is a vertex u which is an ancestor of v and w. The *lowest common ancestor* $\mathrm{lca}(v, w)$ of v and w is the common ancestor of v and w at maximum depth among all common ancestors of v and w. A *preorder numbering* of a rooted tree T with N vertices is an assignment of numbers 0 through $N - 1$ to the vertices of T such that every vertex has a preorder number less than any of its descendants and the

preorder numbers of each vertex and all its descendants are contiguous. Given an ordering of the children of each node, a *lexicographical numbering* of T is a preorder numbering of T such that the preorder numbers of the children of any node, sorted by the given order, are increasing. An *independent set* $I \subseteq V$ in a graph G is a set of vertices such that for every vertex $v \in I$ and every edge $\{v, w\} \in E$, $w \notin I$. That is, no two vertices in I are adjacent. A *k-coloring* of a graph G is an assignment $f : V \to \{1, \ldots, k\}$ of colors to the vertices of G such that for any edge $\{v, w\} \in E$, $f(v) \neq f(w)$.

A *spanning tree* of a graph $G = (V, E)$ is a tree $T = (V, F)$, where $F \subseteq E$. A *breadth-first spanning tree* (BFS-tree) is a rooted spanning tree T of G such that for any edge $\{v, w\}$ in G, the levels of v and w in T differ by at most 1.

Let $c : E \to \mathbb{R}^+$ be an assignment of non-negative costs to the edges of G. The *cost* $\|p\|$ of a path $p = \langle v_0, \ldots, v_k \rangle$ is defined as $\|p\| = \sum_{i=0}^{k-1} c(\{v_i, v_{i+1}\})$. A shortest path $\pi(v, w)$ is a path of minimal cost from v to w.

Let $w : V \to \mathbb{R}^+$ be an assignment of non-negative weights to the vertices of G such that $\sum_{v \in V} w(v) \leq 1$. The *weight* $w(H)$ of a subgraph H of G is the sum of the weights of the vertices in H. An *ϵ-separator*, $0 < \epsilon < 1$, of G is a subset S of V such that none of the connected components of $G - S$ has weight exceeding ϵ.

For a given DAG $G = (V, E)$, a *topological ordering* is a total order $\mathcal{O} \subseteq V \times V$ such that for every edge $(v, w) \in E$, $(v, w) \in \mathcal{O}$.

We will describe results on paths in a rooted tree which originate at an arbitrary node of the tree and proceed to the root. We will refer to such paths as *bottom-up* paths.

For a given block size B, a *blocking* of a graph $G = (V, E)$ is a decomposition of V into vertex sets $\mathcal{B}_1, \ldots, \mathcal{B}_q$ such that $V = \bigcup_{i=1}^{q} \mathcal{B}_i$ and $|\mathcal{B}_i| \leq B$. The vertex sets $\mathcal{B}_i$ do not have to be disjoint. Each such vertex set $\mathcal{B}_i$ corresponds to a block in external memory where its vertices are stored.

2.2 External Memory Techniques

First we introduce some useful algorithmic techniques in external memory which we will use in our algorithms. These include external memory stacks and queues, list ranking, time-forward processing and their applications to processing trees.

We have already observed that scanning an array of size N takes $O(scan(N))$ I/Os using at least D blocks of internal memory to hold the D blocks cur-

rently being scanned. Using $2D$ blocks of internal memory, a series of N stack operations can be executed in $O(scan(N))$ I/Os. These $2D$ blocks hold the top-most elements on the stack. Initially, we hold the whole stack in internal memory. As soon as there are $2DB$ elements on the stack, and we want to push another element on the stack, we swap the DB bottom-most elements to external memory. This takes one I/O. Now it takes at least DB stack operations (DB pushes to fill the internal buffer again or DB pops to have no elements left in internal memory) before the next I/O becomes necessary. Thus, we perform at most one I/O per DB stack operations, and N stack operations take $O(N/DB) = O(scan(N))$ I/Os. Similarly, to implement a queue we need $2D$ blocks of internal memory, D to buffer the inputs at the end of the queue and D to buffer the outputs at the head of the queue.

The list-ranking problem is the following: *Given a singly linked list L and a pointer to its head, compute for every node of L its distance to the tail of L.* A common variant is to assign weights to the nodes of L and to compute for every node the weight of the sublist of L starting at this node. In [3] a recursive list-ranking procedure was developed. If the list has size at most M, the problem can be solved in internal memory. Reading the input and writing the output take $O(scan(N))$ I/Os in this case. If the list contains more than M nodes the procedure 3-colors the list and removes the largest monochromatic set of vertices. For every removed vertex x, the weight of x's predecessor y is updated to $w(y) = w(y) + w(x)$. After recursively applying the same technique to the remaining sublist, which has size at most $\frac{2}{3}N$, the removed vertices are reintegrated into the list and their ranks computed. In [3] it is shown that the 3-coloring of the list and the removal and reintegration of the independent set can be done in $O(sort(N))$ I/Os. Thus, this procedure takes $T(N) \leq T\left(\frac{2}{3}N\right) + O(sort(N)) = O(sort(N))$ I/Os. In [26] it is shown that the 3-coloring technique can be extended to rooted trees, which allows the application of the same framework to the problem of computing the level of every node in a rooted tree. Alternatively, one can use the Euler-tour technique together with the list-ranking technique to compute these levels.

The Euler-tour technique can be described as follows: Given a rooted tree T, we replace every edge $\{v, w\}$ by two directed edges (v, w) and (w, v). For every vertex v, let $e_0, \ldots, e_{k-1}$ be the incoming edges of v and $e'_0, \ldots, e'_{k-1}$ be the outgoing edges of v, where e'_i is directed opposite to e_i. We define edge $e'_{(i+1) \bmod k}$ to be the successor of edge e_i, for $0 \leq i < k$. At the root of T we define edge e'_{k-1} to have no successor. This defines a traversal of the tree, starting at the root and traversing every edge of the tree exactly twice, once per direction. To compute the levels of all vertices in T, for instance, we assign weight 1 to all edges directed from parents to children and weight -1 to all edges directed from children to parents. Then we can apply the list-ranking technique to compute the levels of the endpoints of all edges.

If we choose the order of the edges $e_0, \ldots, e_{k-1}$ carefully, we can use the Euler-tour technique to compute a lexicographical numbering of a given rooted tree. Recall that in this case we are given a left-to-right ordering of the children of every vertex. We construct the Euler tour by choosing for every vertex v edge e_0 to be the edge connecting v to its parent. The remaining edges $e_1, \ldots, e_{k-1}$ are the edges connecting v to its children, sorted from left to right. It is easy to see that the Euler tour thus constructed traverses the tree T lexicographically. To compute the lexicographical numbers of all vertices, we assign weight 1 to edges directed from parents to children and weight 0 to all remaining edges. After applying list-ranking to this list, the lexicographical number of a vertex v is the rank of the edge with weight 1 and target vertex v.

An important ingredient of the 3-coloring technique in [3] as well as most of the algorithms presented in this paper is *time-forward processing*, which was introduced in [3]. This technique is useful for processing directed acyclic graphs (DAGs). We view the DAG as a circuit and allow sending a constant amount of information along every edge. Every node can use the information sent along its incoming edges to compute a function of these values and then send a constant amount of information along each of its outgoing edges. The technique presented for this problem in [3] has two constraints: (1) The fan-in of the vertices in the DAG has to be bounded by some constant d and (2) the ratio $m = \frac{M}{B}$ has to be large enough. These constraints have been removed in [2] using the following elegant solution: Given a DAG G, sort the vertex set topologically, which defines a total order on this set. Then evaluate the nodes in their order of appearance, thereby ensuring that all nodes u with an outgoing edge (u, v) have been evaluated before v. Every such node u inserts the information it wants to send to v into a priority queue, giving it priority v. Node v performs $d_{\text{in}}(v)$ DELETEMIN operations to retrieve its inputs, where $d_{\text{in}}(v)$ is the fan-in of v. It is easy to verify that at the time when v is evaluated, the $d_{\text{in}}(v)$ smallest elements in the priority queue have indeed priority v. After sorting the vertex and edge sets of the DAG, this technique performs $2|E|$ priority queue operations, which take $O(sort(|E|))$ I/Os [2]. Thus, this technique takes $O(sort(|V| + |E|))$ I/Os.

3 Blocking Rooted Trees

In this section we consider the following problem: Given a rooted tree T, store it in external memory so that for any query vertex $v \in T$, the path from v to the root of T can be reported I/O-efficiently. One can use redundancy to reduce the number of blocks that have to be read to report the path. However, this increases the space requirements. The following theorem gives a trade-off between the space requirements of the blocking and the I/O-efficiency of the tree traversal.

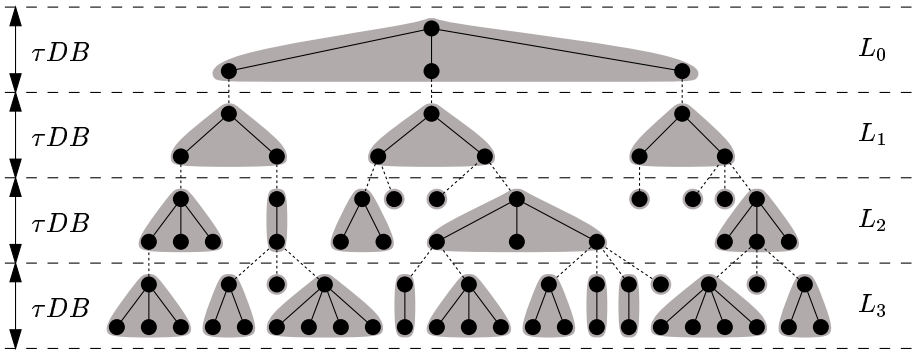

Fig. 1. A tree is cut into layers of height τDB. (Here, $\tau DB = 2$.) The tree is cut along the dashed lines. The resulting subtrees T_i are shaded.

Theorem 1 *Given a rooted tree T of size N and a constant τ, $0 < \tau < 1$, we can store T on D parallel disks so that traversing any bottom-up path of length K in T takes at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 3$ I/Os. The amount of storage used is at most $\left(2 + \frac{2}{1-\tau}\right)\frac{N}{B} + D$ blocks for $\tau \geq \frac{3-\sqrt{5}}{2}$ and at most $\left(1 + \frac{1}{1-2\tau}\right)\frac{N}{B} + D$ blocks for $\tau < \frac{3-\sqrt{5}}{2}$.*

PROOF. This follows from Lemmas 2 and 3 below. □

Intuitively, our approach is as follows. We cut T into layers of height τDB. This divides every bottom-up path of length K in T into subpaths of length τDB; each subpath stays in a particular layer. We ensure that each such subpath is stored in at most D blocks, and each of the D blocks is stored on a different disk. Thus each subpath can be traversed at the cost of a single I/O operation. This gives us the desired I/O-bound because any bottom-up path of length K crosses at most $\lceil \frac{K}{\tau DB} \rceil + 1$ layers and is thus divided into at most $\lceil \frac{K}{\tau DB} \rceil + 1$ subpaths.

More precisely, let h represent the height of T, and let $h' = \tau DB$ be the height of the layers to be created (we assume that h' is an integer). We cut T into layers $L_0, \ldots, L_{\lceil h/h' \rceil - 1}$, where layer L_i is the subgraph of T induced by the vertices on levels ih' through $(i+1)h' - 1$ (see Fig. 1). Each layer is a forest of rooted trees, whose heights are at most h'. Suppose that there are r such trees, $T_1, \ldots, T_r$, taken over all layers. We decompose each tree T_i, $1 \leq i \leq r$, into possibly overlapping subtrees $T_{i,0}, \ldots, T_{i,s}$ having the following properties:

Property 1. $|T_{i,j}| \leq DB$, for all $0 \leq j \leq s$,
Property 2. $\sum_{j=0}^{s} |T_{i,j}| \leq \left(1 + \frac{1}{1-\tau}\right)|T_i|$, and
Property 3. For every leaf l of T_i, there is a subtree $T_{i,j}$ containing the whole path from l to the root of T_i.

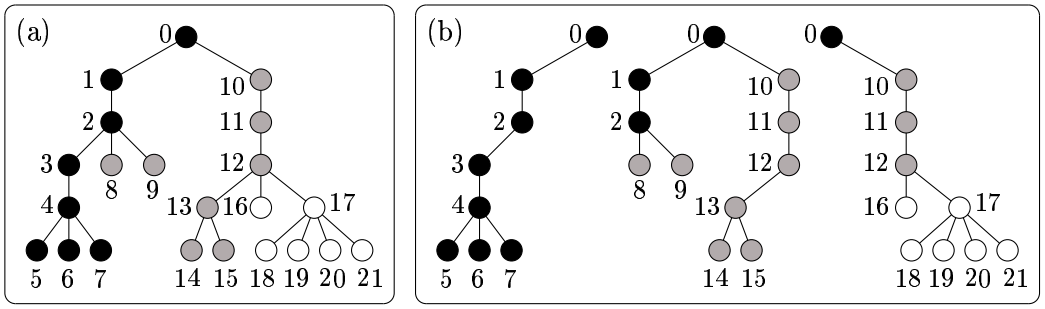

Fig. 2. (a) A rooted tree T_i with its vertices labelled with their preorder numbers. Assuming that $t = 8$, V_0 is the set of black vertices, V_1 is the set of grey vertices, and V_2 is the set of white vertices. (b) The subtrees $T_i(V_0)$, $T_i(V_1)$, and $T_i(V_2)$ from left to right.

Lemma 2 *Given a rooted tree T_i of height at most τDB, $0 < \tau < 1$, we can decompose T_i into subtrees $T_{i,0}, \ldots, T_{i,s}$ having Properties 1–3.*

PROOF. If $|T_i| \leq DB$, we "decompose" T_i into one subtree $T_{i,0} = T_i$. Then Properties 1–3 trivially hold. So assume that $|T_i| > DB$.

Given a preorder numbering of the vertices of T_i, we denote every vertex by its preorder number. Given a parameter $0 < t < DB$ to be specified later, let $s = \lceil |T_i|/t \rceil - 1$. We define vertex sets $V_0, \ldots, V_s$, where $V_j = \{jt, \ldots, (j+1)t - 1\}$ for $0 \leq j \leq s$ (see Fig. 2(a)). The vertex set V_s may contain less than t vertices. The tree $T_{i,j} = T_i(V_j)$ is the subtree of T_i consisting of all vertices in V_j and their ancestors in T_i (see Fig. 2(b)). We claim that these subtrees $T_{i,j}$ have Properties 1–3, if we choose t appropriately.

Property 3 is guaranteed because for every leaf l, the subtree $T_{i,j}$, where $l \in V_j$, contains the whole path from l to the root of T_i. To prove Property 1, let A_j be the set of vertices in $T_{i,j}$ which are not in V_j. Every such vertex x is an ancestor of some vertex $y \geq jt$. That is, $x < y$. As $x \notin V_j$ and the vertices in V_j are numbered contiguously, $x < jt$. Vertex y is in the subtree of T_i rooted at x and $x < jt \leq y$. Thus, vertex jt must be in this subtree as well because the vertices in the subtree rooted at x are numbered contiguously. Hence, every vertex in A_j is an ancestor of vertex jt. This implies that $|A_j| \leq \tau DB$ because jt has at most one ancestor at each of the at most τDB levels in T_i. Hence, $|T_{i,j}| = |A_j| + |V_j| \leq \tau DB + t$. Choosing $t = DB - \tau DB$, we guarantee that

every subtree $T_{i,j}$ has size at most DB. Now Property 2 holds because

$$\sum_{j=0}^{s} |T_{i,j}| \leq \sum_{j=0}^{\left\lceil \frac{|T_i|}{DB-\tau DB} \right\rceil - 1} DB \tag{1}$$

$$\leq \left(\frac{|T_i|}{DB - \tau DB} + 1 \right) DB \tag{2}$$

$$= \frac{1}{1-\tau} |T_i| + DB \tag{3}$$

$$< \left(\frac{1}{1-\tau} + 1 \right) |T_i|. \tag{4}$$

The step from line (3) to line (4) uses the fact that $|T_i| > DB$. □

Lemma 3 *If a rooted tree T of size N is decomposed into subtrees $T_{i,j}$ such that Properties 1–3 hold, T can be stored on D parallel disks so that any bottom-up path of length K in T can be traversed in at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 3$ I/Os, for $0 < \tau < 1$. For $\tau \geq \frac{3-\sqrt{5}}{2}$, the amount of storage used is at most $\left(2 + \frac{2}{1-\tau}\right) \frac{N}{B} + D$ blocks. For $\tau < \frac{3-\sqrt{5}}{2}$, the tree occupies at most $\left(1 + \frac{1}{1-2\tau}\right) \frac{N}{B} + D$ blocks.*

PROOF. We consider D disks of block size B as one large disk divided into *superblocks* of size DB. Thus, by Property 1, each subtree $T_{i,j}$ fits into a single superblock and can be read in a single I/O.

A bottom-up path p of length K in T crosses at most $k = \left\lceil \frac{K}{\tau DB} \right\rceil + 1$ layers. This divides p into k maximal subpaths such that none of them crosses more than one layer. Each such subpath p' is a leaf-to-root path in some subtree T_i, or a subpath thereof. Thus, by Property 3, there exists a subtree $T_{i,j}$ that contains the whole of the subpath p'. That is, each subpath p' can be accessed in a single I/O. Therefore, the traversal of p takes at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 1$ I/Os.

By property 2, all subtrees $T_{i,j}$ together use at most $\left(1 + \frac{1}{1-\tau}\right) \sum_{i=0}^{r} |T_i| = \left(1 + \frac{1}{1-\tau}\right) N$ space. Initially, we store every subtree $T_{i,j}$ in a separate superblock, which may leave many superblocks sparsely populated. As long as there are at least two superblocks that are at most half full we keep merging pairs of superblocks. Finally, if we are left with a single half full superblock we try to find a superblock that can be merged with it. If we find one, we merge the two superblocks. Otherwise, the half-full superblock together with any other superblock contains more than DB vertices. All other superblocks are at least half full. Thus, on average, each superblock is at least half full, and we use $\left\lceil \left(2 - \frac{2}{1-\tau}\right) \frac{N}{DB} \right\rceil$ superblocks, i.e., $D \left\lceil \left(2 - \frac{2}{1-\tau}\right) \frac{N}{DB} \right\rceil \leq \left(2 + \frac{2}{1-\tau}\right) \frac{N}{B} + D$ blocks to store all subtrees $T_{i,j}$.

For $\tau < \frac{3-\sqrt{5}}{2} \approx 0.38$, $1 + \frac{1}{1-2\tau} < 2 + \frac{2}{1-\tau}$. In this case, we apply the following strategy. We cut the given tree T into layers of height $2\tau DB$. We guarantee that the vertices of every subtree $T_{i,j}$ are stored contiguously, but not necessarily in a single block. As every tree $T_{i,j}$ has size at most DB, it is spread over at most two blocks. Thus, two I/Os suffice to traverse any leaf-to-root path in a subtree $T_{i,j}$, and traversing a path of length K takes at most $2\left(\left\lceil \frac{K}{2\tau DB} \right\rceil + 1\right) \leq \left\lceil \frac{K}{\tau DB} \right\rceil + 3$ I/Os. On the other hand, we guarantee that all superblocks except for the last one are full. Thus, we use $\left\lceil \left(1 - \frac{1}{1-2\tau}\right) \frac{N}{DB} \right\rceil$ superblocks, i.e., $D\left\lceil \left(1 - \frac{1}{1-2\tau}\right) \frac{N}{DB} \right\rceil \leq \left(1 + \frac{1}{1-2\tau}\right) \frac{N}{B} + D$ blocks of external memory. □

4 Separating Embedded Planar Graphs

We now present an external-memory algorithm for separating embedded planar graphs. Our algorithm is based on the classical linear-time separator algorithm of [17]. It computes a $\frac{2}{3}$-separator S of size $O\left(\sqrt{N}\right)$ for a given embedded planar graph G in $O(sort(N))$ I/Os, provided that a BFS-tree of the graph is given. [3]

The input to our algorithm is an embedded planar graph G and a spanning forest F of G. Every tree in F is a BFS-tree of the respective connected component. (In the remainder of this section, we call F a BFS-forest.) The graph G is represented by its vertex set V and its edge set E. To represent the embedding, let the edges incident to each vertex v be numbered in counterclockwise order around v, starting at an arbitrary edge as the first edge. This defines two numbers $n_v(e)$ and $n_w(e)$, for every edge $e = \{v, w\}$. Let these numbers be stored with e. The spanning forest F is given implicitly by marking every edge of G as "tree edge" or "non-tree edge" and storing with each vertex $v \in V$, the name of its parent in F.

4.1 Framework of the Algorithm

First we compute the connected components of the given graph G. If there is a component whose weight is greater than $\frac{2}{3}$ we compute a separator S of that component. Then we compute the connected components of $G - S$, which gives the desired partition of G into subgraphs of weight at most $\frac{2}{3}$ each.

[3] The currently best known algorithm for computing a BFS-tree [19] takes $O(|V| + |E|/|V|sort(|V|))$ I/Os.

The connected components can be computed in $O(sort(N))$ I/Os [3]. In the next subsection, we describe how to compute the separator S using $O(sort(N))$ I/Os, leading to the following theorem.

Theorem 4 *Given an embedded planar graph G with N vertices and a BFS-forest F of G, a $\frac{2}{3}$-separator of size at most $2\sqrt{2}\sqrt{N}$ for G can be computed in $O(sort(N))$ I/Os.*

4.2 Separating Connected Planar Graphs

In this section, we present an external memory algorithm for computing a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ for a connected embedded planar graph G of size N and weight at least $\frac{2}{3}$, provided that a BFS-tree T of G is given. We assume that G is triangulated. If it is not, it can be triangulated in $O(sort(N))$ I/Os using the algorithm in Section 5.[4] Also, we assume that no vertex has weight exceeding $\frac{1}{3}$ because otherwise $S = \{v\}$, where $w(v) > \frac{1}{3}$, is trivially a $\frac{2}{3}$-separator of G.

The algorithm (Algorithm 1) is based on the following observation: In a BFS-tree T of a given graph G, non-tree edges connect vertices on the same level or on consecutive levels. Thus, the removal of all vertices of any level in T disconnects the subgraph of G corresponding to the upper part of T from the subgraph corresponding to the lower part (see Fig. 3(a)). We will find two levels l_0 and l_2 that divide the graph into three parts G_1, G_2, and G_3, where $w(G_1) \leq \frac{2}{3}$, $w(G_3) \leq \frac{2}{3}$, and the number of vertices on levels l_0 and l_2 is at most $2\sqrt{2}\sqrt{N} - 2(l_2 - l_0 - 1)$ (Step 2; see Fig. 3(b)). Thus, G_1 and G_3 already have the desired weight, and we can afford to add up to two vertices per level between l_0 and l_2 to the separator, in order to cut G_2 into pieces whose individual weights are at most $\frac{2}{3}$ (Step 3; see Fig. 3(c)).

Lemma 5 *Algorithm 1 computes a $\frac{2}{3}$-separator S of size at most $2\sqrt{2}\sqrt{N}$ for*

[4] Note, however, that the graph has to be triangulated before computing the BFS-tree. Otherwise, T might not be a BFS-tree of the triangulation anymore.

SEPARATECONNECTED(G, T):

1: Label every vertex in T with its level in T.
2: Compute levels l_0 and l_2 cutting G into subgraphs G_1, G_2, and G_3 such that $w(G_1) \leq \frac{2}{3}$, $w(G_3) \leq \frac{2}{3}$ and $L(l_0) + L(l_2) + 2(l_2 - l_0 - 1) \leq 2\sqrt{2}\sqrt{N}$.
3: Find a $\frac{2}{3}$-separator S' of size at most $2(l_2 - l_0 - 1)$ for G_2.
4: Remove the vertices on levels l_0 and l_2 and in S' and all edges incident to these vertices from G and compute the connected components of $G - S$.

Algorithm 1: Separating a connected embedded planar graph.

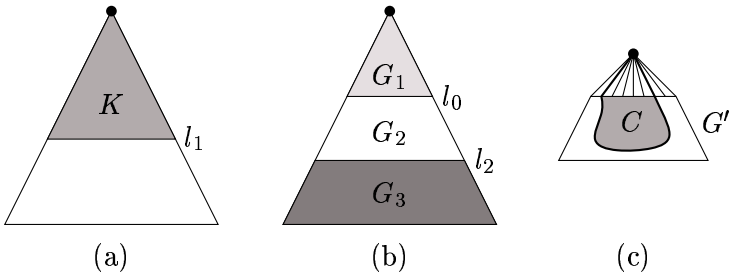

Fig. 3. Illustrating the three major steps of the separator algorithm.

an embedded connected planar graph G with N vertices in $O(sort(N))$ I/Os, provided that a BFS-tree of G is given.

PROOF. First, let us assume that Step 3 computes the desired separator of G_2 in $O(sort(N))$ I/Os. Then the major difficulty in Algorithm 1 is finding the two levels l_0 and l_2 in Step 2. For a given level l, let $L(l)$ be the number of vertices on level l and $W(l)$ be the weight of the vertices on level l. We first compute the level l_1 closest to the root such that $\sum_{l \leq l_1} W(l) > \frac{1}{2}$ (see Fig. 3(a)). Let $K = \sum_{l \leq l_1} L(l)$. Then we compute levels $l_0 \leq l_1$ and $l_2 > l_1$ such that $L(l_0) + 2(l_1 - l_0) \leq 2\sqrt{K}$ and $L(l_2) + 2(l_2 - l_1 - 1) \leq 2\sqrt{N - K}$ (see Fig. 3(b)). The existence of level l_1 is obvious. The existence of levels l_0 and l_2 has been shown in [17]. It is easy to see that levels l_0 and l_2 have the desired properties.

Now we turn to the correctness of Step 3. In this step, we shrink levels 0 through l_0 to a single root vertex r of weight 0. Next, we remove levels l_2 and below, and retriangulate the resulting graph, obtaining a triangulation G' (see Fig. 3(c)). Then we use the techniques of Section 4.3 to compute a fundamental cycle C of G' which is a $\frac{2}{3}$-separator of G'. Graph G_2 is a subgraph of G'. Thus, $G_2 \cap C$ is a $\frac{2}{3}$-separator of G_2. The fundamental cycle can have length at most $2(l_2 - l_0) - 1$. If the length is indeed $2(l_2 - l_0) - 1$, C contains the root vertex r, which is not part of G_2. Thus, $S' = G_2 \cap C$ has size at most $2(l_2 - l_0 - 1)$, as desired. When shrinking levels 0 through l_0 to a single vertex we have to be a bit careful because we have to maintain the embedding of the graph. To do this we number the vertices of T lexicographically and sort the vertices $v_1, \ldots, v_k$ on level $l_0 + 1$ by increasing numbers (i.e., from left to right). Let w_i be the parent of v_i on level l_0. Then we replace edge $\{v_i, w_i\}$ by edge $\{r, v_i\}$ and assign $n_{v_i}(\{r, v_i\}) = n_{v_i}(\{v_i, w_i\})$ and $n_r(\{r, v_i\}) = i$. This places edges $\{r, v_i\}$ counterclockwise around r and guarantees that edge $\{r, v_i\}$ is embedded between the appropriate edges incident to v_i. We construct a BFS-tree T' of G' by adding the edges $\{r, v_i\}$ to the remains of T in G'. It is easily verified that T' is a spanning tree of G'. It is not so easy to see that T' is a BFS-tree of G', which is crucial for simplifying Algorithm 2 in Section 4.3, which computes the desired separator for G' and thus G_2.

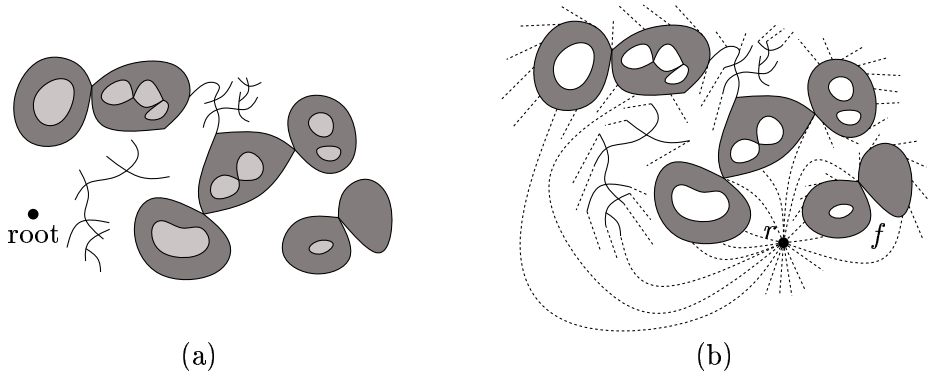

Fig. 4. (a) An embedding of a given graph G. Regions R_{l_0}, R_{l_2}, and R' are the white, light grey, and dark grey parts of the plane, respectively. The additional curves in region R_{l_0} are level $l_0 + 1$ vertices. (b) In Step 3 of Algorithm 1, we replace the graph in region R_{l_0} by a single root vertex r and connect it to all vertices at level $l_0 + 1$ (dotted curves). We also remove the interior of region R_{l_2}. All non-triangular faces like face f have to be triangulated.

Consider the embedding of G in the plane (see Fig. 4(a)). Then we define R_{l_0} to be the union of triangles that have at least one vertex at level l_0 or above in T (the outer face in Fig. 4(a)). Analogously we define R_{l_2} to be the union of triangles that have at least one vertex at level l_2 or below in T (the light gray interior faces in Fig. 4(a)). The embedding of G_2 then lies in $R' = \mathbb{R}^2 \setminus (R_{l_0} \cup R_{l_2})$ (the dark grey regions in Fig. 4(a)). The boundary of R' is a set of edge-disjoint cycles, and the interior of R' is triangulated. Also, note that no vertex at level at most l_0 can be on the boundary between R_{l_0} and R' because all incident triangles are in R_{l_0}. On the other hand, for a vertex at level at least $l_0 + 2$ to be on that boundary it has to be on the boundary of a triangle that has a vertex at level at most l_0 on its boundary, which is also impossible. Thus, the boundary between R_{l_0} and R' is formed by level $l_0 + 1$ vertices only. (Note that some level $l_0 + 1$ vertices are interior to R_{l_0} (the additional curves in region R_{l_0} in Fig. 4(a)).) Analogously, the boundary between R_{l_2} and R' is formed by level $l_2 - 1$ vertices only. When shrinking the vertices above and including level l_0 to a single root vertex, we can think of this as removing the parts of G in R_{l_0} and placing the new root vertex r into R_{l_0} (see Fig. 4(b)). The tree edges connecting r to level $l_0 + 1$ vertices are embedded in R_{l_0}. As we did not alter the triangulation of R' (i.e., G_2), the only faces of G' where the triangulation algorithm adds diagonals are in R_{l_0} or R_{l_2}. As all vertices on the boundary of R_{l_0} are already connected to r, the triangulation algorithm only adds edges connecting two level $l_0 + 1$ or two level $l_2 - 1$ vertices, thereby not destroying the BFS-property of T' with respect to G'.

Now we analyze the complexity of Algorithm 1. Step 1 takes $O(sort(N))$ I/Os (see Section 2.2). In Step 2, we sort the vertices by their levels and scan it until the total weight of the vertices seen so far exceeds $\frac{1}{2}$. We continue the

scan until we reach the end of a level. While scanning we maintain the count K of visited vertices. We scan the vertex list backward, starting at l_1, and count the number of vertices on each visited level. When we find a level l_0 with $L(l_0)+2(l_1-l_0) \leq 2\sqrt{K}$ we stop. In the same way we find l_2 scanning forward, starting at the level following l_1. As we sort and scan a constant number of times, Step 2 takes $O(sort(N))$ I/Os. Removing all levels below and including l_2 takes $O(sort(N))$ I/Os: We sort the vertex set and first sort the edges by their first endpoints; in a single scan we mark all edges that have their first endpoint on level l_2 or below; after sorting the edges by their second endpoints, we scan the edge list again to mark all edges with their second endpoints on level l_2 or below; finally, it takes a single scan to remove all marked vertices and edges. Shrinking levels 0 through l_0 to a single root vertex takes $O(sort(N))$ I/Os: $O(sort(N))$ I/Os to number the vertices lexicographically, $O(sort(N))$ I/Os to sort the vertices on level l_0+1 and another scan to replace edges $\{v_i, w_i\}$ by edges $\{r, v_i\}$. By Theorem 7, we can retriangulate the resulting graph in $O(sort(N))$ I/Os. The removal of the vertices and edges in G_1 can be done in a similar way as for G_3. The rest of Step 3 takes $O(sort(N))$ I/Os by Lemma 6. Steps 2 and 3 have marked the vertices on levels l_0 and l_2 and in S' as separator vertices. Then we use the same technique as for removing G_3 to remove all separator vertices and incident edges in $O(sort(N))$ I/Os. Computing the connected components of $G-S$ takes $O(sort(N))$ I/Os [3]. □

4.3 Finding a Small Simple Cycle Separator

Let G' be the triangulation as constructed at the beginning of Step 3 of Algorithm 1 and T' be the BFS-tree of G' constructed from T. Every non-tree edge $e = \{v, w\}$ in G' defines a fundamental cycle $C(e)$ consisting of e itself and the two paths in the tree T' from the vertices v and w to the lowest common ancestor u of v and w (see Fig. 5). (Note that in a BFS-tree, u is distinct from both v and w.) Given an embedding of G', any fundamental cycle $C(e)$ separates G' into two subgraphs $R_1(e)$ and $R_2(e)$, one induced by the vertices embedded inside $C(e)$ and the other induced by those embedded outside. In [17] it is shown that there is a non-tree edge e in G' such that $R_1(e)$ and $R_2(e)$ have weights at most $\frac{2}{3}$ each, provided that no vertex has weight exceeding $\frac{1}{3}$ and G' is triangulated. Moreover, for any non-tree edge e, the number of vertices on the fundamental cycle $C(e)$ is at most $2 \cdot \text{height}(T') - 1 = 2(l_2 - l_0) - 1$. Next we show how to find such an edge and the corresponding fundamental cycle I/O-efficiently.

We compute labels for the non-tree edges of G' so that we can compute the weights $w(R_1(e))$ and $w(R_2(e))$ of regions $R_1(e)$ and $R_2(e)$ only using the labels stored with edge e. Then we can find the edge e whose corresponding fundamental cycle is a $\frac{2}{3}$-separator in a single scan over the edge set of G'.

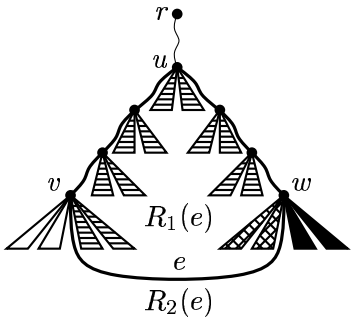

Fig. 5. A non-tree edge e and the corresponding fundamental cycle $C(e)$ shown in bold. $R_1(e)$ is the region inside the cycle and $R_2(e)$ is the region outside the cycle.

Consider Fig. 5. Given a lexicographical numbering of the vertices in T', denote the number assigned to vertex x by $n_l(x)$. For a given edge $e = \{v, w\}$, we denote by v the endpoint with smaller lexicographical number, i.e., $n_l(v) < n_l(w)$. Then the vertices in the white and striped subtrees and on the path from u to w are exactly the vertices with lexicographical numbers between $n_l(v)$ and $n_l(w)$. We compute for every vertex x a label $\nu_l(x) = \sum_{n_l(y) \leq n_l(x)} w(y)$. Then the weight of the vertices in the white and striped subtrees and on the path from u to w is $\nu_l(w) - \nu_l(v)$. Given for every vertex x the weight $\lambda(x)$ of the vertices on the path from x, inclusive, to the root r of T', the weight of the vertices on the path from u to w, including w but not u, is $\lambda(w) - \lambda(u)$, and the weight of the white and striped subtrees is $\nu_l(w) - \nu_l(v) + \lambda(w) - \lambda(u)$. It remains to add the weight of the cross-hatched trees and to subtract the weight of the white trees to obtain the weight $w(R_1(e))$ of region $R_1(e)$.

For every edge e, we compute a label $\tau(e)$. If $e = \{v, w\}$ is a tree edge and v is further away from the root of T' than w, then $\tau(e)$ is the weight of the subtree of T' rooted at v. If e is a non-tree edge, $\tau(e) = 0$. Let $e_0, \ldots, e_k$ be the list of edges incident to a vertex x, sorted counterclockwise around x and so that $e_0 = \{x, p(x)\}$, where $p(x)$ denotes x's parent in T'. Then we define $\tau_x(e_0) = 0$ and $\tau_x(e_i) = \sum_{j=1}^{i} \tau(e_j)$ for $i > 0$. Now the weight of the white subtrees in Fig. 5 is $\tau_v(e)$, and the weight of the cross-hatched subtrees is $\tau_w(e)$. Thus, we can compute the weight of region $R_1(e)$ as

$$w(R_1(e)) = \nu_l(w) - \nu_l(v) + \lambda(w) - \lambda(u) + \tau_w(e) - \tau_v(e). \tag{5}$$

It is easy to see that the weight of region $R_2(e)$ is

$$w(R_2(e)) = w(G') - w(R_1(e)) - \lambda(v) - \lambda(w) + 2\lambda(u) - w(u) \tag{6}$$

(the weight of the whole graph minus the weight of the interior region minus the weight of the fundamental cycle). Assuming that all these labels are stored with e, we can scan the edge set of G' and compute for every visited non-tree edge the weights of $R_1(e)$ and $R_2(e)$ using equations (5) and (6). It remains to show how to compute these labels. We provide the details in the proof of the following lemma.

Lemma 6 *Given a triangulated graph G' with N vertices and a BFS-tree T' of G', Algorithm 2 takes $O(sort(N))$ I/Os to compute a $\frac{2}{3}$-simple cycle separator of size at most $2 \cdot \text{height}(T') - 1$ for G'.*

PROOF. The labelling of vertices in Step 1a takes $O(sort(N))$ I/Os: The subtree weights $W(v)$ and the weighted levels $\lambda(v)$ of the vertices can be computed using time-forward processing in the tree. To compute the subtree weights, we process the tree bottom-up. To compute the weighted levels, we process the tree top-down. In Section 2.2, we have described how to compute a lexicographical numbering of T' in $O(sort(N))$ I/Os. To compute $\nu_l(x)$ for all vertices x, we sort the vertex set of T' by increasing numbers $n_l(x)$ and then scan this sorted list to compute labels $\nu_l(x)$ as the prefix sums over the weights $w(x)$.

As there are at most $2N - 5$ non-tree edges in T', we have to compute $O(N)$

CycleSeparator(G', T'):

1: Compute the vertex and edge labels required to compute the weights of regions $R_1(e)$ and $R_2(e)$, for every non-tree edge e.

a: Label every vertex x in G' with a tuple $A(x) = (W(x), n_l(x), \lambda(x), \nu_l(x))$, where

- $W(x)$ is the weight of the subtree of T' rooted at x,
- $n_l(x)$ is x's lexicographical number in T',
- $\lambda(x)$ is the total weight of all ancestors of x in T', inclusive, and
- $\nu_l(x) = \sum_{n_l(y) \leq n_l(x)} w(y)$ ("weighted lexicographical number of x").

b: For every edge $e = \{v, w\}$,

(1) Compute the lowest common ancestor u of v and w and copy the tuples $A(u)$, $A(v)$, and $A(w)$ to e, and

(2) Compute a label $\tau(e)$ defined as

$$\tau(e) = \begin{cases} 0 & \text{if } e \text{ is a non-tree edge} \\ W(v) & \text{if } e \text{ is a tree edge and } w = p(v). \end{cases}$$

c: For every vertex x let $e_0, \ldots, e_d$ be the set of edges incident to x sorted counterclockwise around x and so that $e_0 = \{x, p(x)\}$. Compute labels

$$\tau_x(e_i) = \begin{cases} 0 & \text{if } i = 0 \\ \tau_x(e_{i-1}) + \tau(e_i) & \text{if } i > 0. \end{cases}$$

2: Scan the edge list of G' and compute for every non-tree edge e, the weights of $R_1(e)$ and $R_2(e)$ using equations (5) and (6). Choose a non-tree edge e such that $w(R_1(e)) \leq \frac{2}{3}$ and $w(R_2(e)) \leq \frac{2}{3}$.

3: Report the fundamental cycle $C(e)$.

Algorithm 2: Finding a simple cycle separator in a triangulation.

lowest common ancestors in T' in Step 1b, which takes $O(sort(N))$ I/Os [3]. To copy the labels of u, v, and w to edge $e = \{v, w\}$, we first sort the vertex set by increasing vertex names. Then we sort the edge set three times, once by each of the values of u, v, and w. After each sort, we scan the vertex and edge sets to copy the tuples $A(u)$, $A(v)$, and $A(w)$, respectively, to the edges. Computing labels $\tau(e)$ then takes an additional scan over the edge set of T'.

To compute labels $\tau_x(e)$ in Step 1c, we create a list L_1 containing two tuples $(v, n_v(e), p(v), w, \tau(e))$ and $(w, n_w(e), p(w), v, \tau(e))$ for every edge $e = \{v, w\}$. We sort this list lexicographically, so that (the tuples corresponding to) the edges incident to vertex x are stored consecutively, sorted counterclockwise around x. Then we scan L_1 and compute a list L_2 of triples $(v, w, \tau_v(e))$ and $(v, w, \tau_w(e))$. Note that in L_1, the edge $(x, p(x))$ is not necessarily stored as the first edge in the sublist of edges incident to x causing us to skip some edges at the beginning of the sublist until we find edge $(x, p(x))$ in the list. The skipped edges have to be appended at the end of the sublist. We can use a queue to do this. After sorting L_2 as well as the edge list of T', it takes a single scan of the two sorted lists to copy the labels $\tau_v(e)$ and $\tau_w(e)$ to all edges e.

Step 2 searches for a non-tree edge whose corresponding fundamental cycle is a $\frac{2}{3}$-separator. As already observed, this search takes a single scan over the edge list of G', using the labels computed in Step 1 to compute the weights $w(R_1(e))$ and $w(R_2(e))$ of the interior and exterior regions.

Step 3 is based on the following observation: Given a lexicographical numbering $n_l(x)$ of the vertices of T', this numbering is also a preorder numbering. Given a vertex v with preorder number x, let the subtree rooted at v have size m. Then the vertices in this subtree have preorder numbers x through $x + m - 1$. This implies that the ancestor of a vertex v at a given level l is the vertex u such that $n_l(u) = \max\{n_l(u') : l(u') = l \wedge n_l(u') \leq n_l(v)\}$, where $l(x)$ denotes the level of vertex x. Using this observation we sort the vertices by increasing levels in T' and in each level by increasing lexicographical numbers and scan this sorted list of vertices backward, finding the ancestors of v and w at every level until we come to a level where v and w have the same ancestor, u. Thus, Step 3 also takes $O(sort(N))$ I/Os. □

5 Triangulating Embedded Planar Graphs

In this section, we present an $O(sort(N))$-algorithm to triangulate a connected embedded planar graph $G = (V, E)$. We assume the same representation of G and its embedding as in the previous section. Our algorithm consists of two phases. First we identify the faces of G. We represent each face f by a list of vertices on its boundary, sorted clockwise around the face. In the second phase

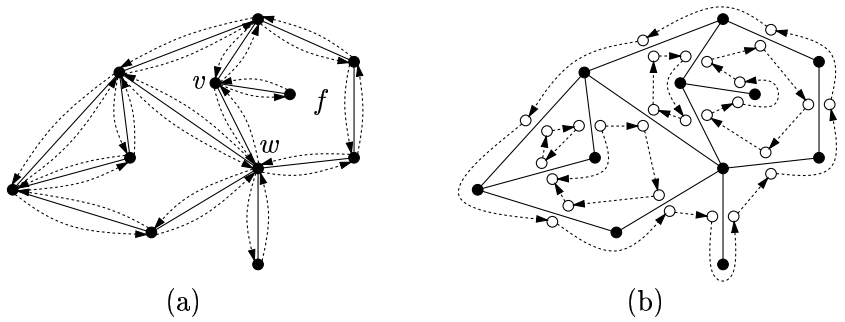

Fig. 6. (a) The directed graph D (dotted arrows) corresponding to a given graph G (solid lines). Note that vertex v appears twice on the boundary of face f, and vertex w appears twice on the boundary of the outer face. (b) The directed graph $\hat{G}$ (white vertices and dotted arrows) corresponding to D.

we use this information to triangulate the faces of G. We show the following theorem.

Theorem 7 *Given an embedded planar graph G, it can be triangulated in $O(sort(N))$ I/Os.*

PROOF. This follows from Lemmas 8 and 11. □

5.1 Identifying Faces

As mentioned above, we represent each face f by the list of vertices on its boundary, sorted clockwise around the face. Denote this list by F_f. Let F be the concatenation of the lists F_f for all faces f of G. The goal of the first step is to compute F. The idea of this step is to replace every edge $\{v, w\}$ of G by two directed edges (v, w) and (w, v) and decompose the resulting directed graph, D, into directed cycles, each representing the clockwise traversal of a face of G (see Fig. 6(a)). ("Clockwise" means that we walk along the boundary of the face with the boundary to our left. Thus, a clockwise traversal of the outer face corresponds to walking counterclockwise along the outer boundary of the graph. This somewhat confusing situation is resolved if we imagine the graph to be embedded on a sphere because then all faces are interior.) Removing one edge from each of these cycles gives us a set of paths. The vertices on such a path appear in the same order on the path as clockwise around the face represented by the path. (Note that the same vertex may appear more than once on the boundary of the same face, if it is a cutpoint (see Fig. 6(a)).) Considering the set of paths as a set of lists, we can rank the lists. This gives us the orders of the vertices around all faces, i.e., the lists F_f.

The problem with the directed cycles in D is that they are not vertex-disjoint.

Hence, we cannot apply standard graph external-memory algorithms to extract these cycles. The following modification solves this problem: Instead of building D, we build a graph $\hat{G}$ for G (see Fig. 6(b)), which is closely related to the representation of G by a doubly-connected edge list [21]. $\hat{G}$ contains a vertex $v_{(v,w)}$ for every edge (v,w) in D and an edge between two vertices $v_{(u,v)}$ and $v_{(v,w)}$ if the corresponding edges (u,v) and (v,w) are consecutive in some cycle of D representing a face of G. Graph $\hat{G}$ consists of vertex-disjoint cycles, each representing a face of G. We compute the connected components of $\hat{G}$, thereby identifying the cycles of G, and apply the same transformations to $\hat{G}$ that we wanted to apply to D. Step 1 of Algorithm 3 gives the details of the construction of $\hat{G}$. In Step 2, we use $\hat{G}$ to compute F.

Lemma 8 *Algorithm 3 takes $O(sort(N))$ I/Os to constructs the list F for a given graph G.*

PROOF. *Correctness:* Two vertices in $\hat{G}$ that are consecutive on a cycle of $\hat{G}$ represent two edges that are consecutive on the boundary of a face of G in clockwise order. Thus, these two edges are consecutive around a vertex of G in counterclockwise order. The correctness of Step 1 follows from this observation.

After removing an edge from every cycle in $\hat{G}$, the connected components of $\hat{G}$ are paths. Considering these paths as lists, we can rank these lists. Sorting the vertices in $\hat{V}$ by component labels and decreasing ranks (Step 2d) arranges the vertices representing the same face of G consecutively and sorts them clockwise around the face. Given a sublist $\hat{V}_f$ of $\hat{V}$ representing a face f of G, the vertices of $\hat{V}_f$ represent the edges of D clockwise around the face.

IDENTIFYFACES(G):

1: Construct a directed graph $\hat{G} = (\hat{V}, \hat{E})$:

- a: For each edge $e = \{v,w\} \in E$, add two vertices, $v_{(v,w)}$ and $v_{(w,v)}$ to $\hat{V}$.
- b: For each vertex $v \in V$, let $\{v,w_0\},\ldots,\{v,w_{k-1}\}$ be the edges incident to v, in counterclockwise order.
- c: Add directed edges $\left(v_{(w_i,v)}, v_{\left(v,w_{(i+1) \bmod k}\right)}\right)$, $0 \le i < k$, to $\hat{E}$.

2: Construct F:

- a: Compute the connected components of $\hat{G}$ (considering $\hat{G}$ as undirected), labelling every vertex $\hat{v}$ with a label $c(\hat{v})$ identifying its connected component.
- b: Remove one edge from each connected component of $\hat{G}$.
- c: Rank the resulting lists.
- d: Sort $\hat{V}$ by component labels $c(\hat{v})$ and decreasing ranks.
- e: Scan $\hat{V}$ and write for each vertex $v_{(v,w)}$, a copy of v with face label $f(v) = c(v_{(v,w)})$ to F.

Algorithm 3: Identifying the faces of G.

We can think of Step 2e as scanning this list of edges in D and writing the sorted sequence of source vertices of these edges to disk. This produces the desired list F_f, and the application of Step 2e to the whole of $\hat{V}$ produces the concatenation F of all such lists F_f. Note that the lists F_f are distinguished in F, as two vertices in F have the same label $f(v)$ if and only if they belong to the same list F_f.

Complexity: Step 1a requires a single scan over the edge list E of G. For step 1b, recall that we are given the embedding as two labels $n_v(e)$ and $n_w(e)$ for every edge $e = \{v, w\}$. We replace e by two triples $(v, n_v(e), w)$ and $(w, n_w(e), v)$, and sort the resulting list of triples lexicographically. In the resulting list, all (triples representing) edges incident to a vertex v are stored consecutively, sorted counterclockwise around the vertex. Thus, Step 1c requires a single scan over this sorted list of triples, and Step 1 takes $O(sort(N))$ I/Os.

Steps 2a and 2c take $O(sort(N))$ I/Os [3] as do Steps 2d and 2e. The details of Step 2b are as follows: Note that for every vertex $\hat{v}$, there is exactly one edge $(\hat{v}, \hat{w})$ in $\hat{G}$. Thus, we sort $\hat{V}$ by vertex names and $\hat{E}$ by the names of the source vertices, thereby placing edge $(\hat{v}, \hat{w})$ at the same position in $\hat{E}$ as $\hat{v}$'s position in $\hat{V}$. Scanning $\hat{V}$ and $\hat{E}$ simultaneously, we assign component labels $c((\hat{v}, \hat{w})) = c(\hat{v})$ to all edges $(\hat{v}, \hat{w})$ in $\hat{E}$. Then we sort $\hat{E}$ by these component labels. Finally we scan $\hat{E}$ again and remove every edge whose component label is different from the component label of the previous edge. Also, we remove the first edge in $\hat{E}$. As this procedure requires sorting and scanning $\hat{V}$ and $\hat{E}$ twice, the complexity of Step 2 is $O(sort(N))$ I/Os. □

5.2 Triangulating Faces

We triangulate each face f in four steps (see Algorithm 4). In Step 1, we reduce f to a *simple* face $\hat{f}$. That is, no vertex appears more than once in a clockwise traversal of $\hat{f}$'s boundary. Accordingly we reduce the list F_f to $F_{\hat{f}}$. In Step 2, we triangulate $\hat{f}$. We guarantee that there are no parallel edges in $\hat{f}$. But we may add parallel edges to different faces. (See Fig. 8(a) for an example.) Let $e_1, \ldots, e_k$ be the set of edges with endpoints v and w. In Step 3, we select one of these edges, say e_1, and mark edges $e_2, \ldots, e_k$ as *conflicting*. Each of these edges is said to be *in conflict with* e_1. In Step 4, we retriangulate all faces so that conflicts are resolved and a final triangulation is obtained.

The following lemma states the correctness of Step 1.

Lemma 9 *For each face f of G, the face $\hat{f}$ computed by Step 1 of Algorithm 4 is simple. The parts of f that are not in $\hat{f}$ are triangulated. Moreover, Step 1 does not introduce parallel edges.*

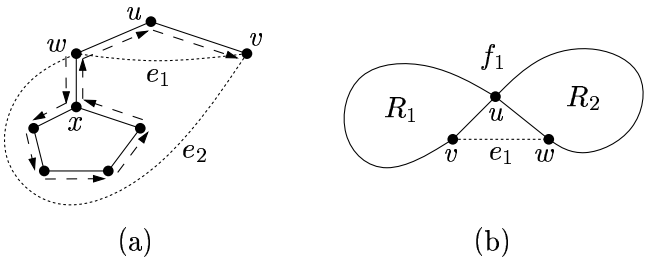

Fig. 7. Illustrating the proof of Lemma 9.

PROOF. We mark exactly one copy of every vertex on f's boundary. For the first vertex, we append a second marked copy to the end of F_f only to close the cycle. This copy is removed at the end of Step 1. We remove an unmarked vertex v by adding an edge between its predecessor u and successor w on the current boundary of the face, thereby cutting triangle (u, v, w) off the boundary of f. As we remove all unmarked vertices this way, the resulting face $\hat{f}$ is simple and the parts that have been removed from f to produce $\hat{f}$ are triangulated.

Next we show that Step 1 does not add parallel edges to the same face f. Assume for the sake of contradiction that we have added two edges with

TRIANGULATEFACES(G, F):

1: Make all faces of G simple:

For each face f, (a) mark the first appearance of each vertex v in F_f, (b) append a marked copy of the first vertex in F_f to the end of F_f, (c) scan F_f backward and remove each unmarked vertex v from f and F_f by adding a diagonal between its predecessor and successor in the current list, and (d) remove the last vertex from list F_f. Call the resulting list $F_{\hat{f}}$.

2: Triangulate the simple faces:

Let $F_{\hat{f}} = \langle v_0, \ldots, v_k \rangle$. Then add "temporary diagonals" $\{v_0, v_i\}$, $2 \leq i \leq k-1$, to $\hat{f}$.

3: Mark conflicting diagonals:

Sort E lexicographically, representing edge $\{v, w\}$ by an ordered pair (v, w), $v < w$, and so that edge $\{v, w\}$ is stored before any "temporary diagonal" $\{v, w\}$. Scan E and mark all occurrences except the first of each edge as conflicting. Restore the original order of all edges and "temporary diagonals".

4: Retriangulate conflicting faces:

a: For each face $\hat{f}$, let $D_{\hat{f}} = \langle \{v_0, v_2\}, \ldots, \{v_0, v_{k-1}\} \rangle$ be the list of "temporary diagonals".
b: Scan $D_{\hat{f}}$ until we find the first conflicting diagonal $\{v_0, v_i\}$.
c: Replace the diagonals $\{v_0, v_i\}, \ldots, \{v_0, v_{k-1}\}$ by new diagonals $\{v_{i-1}, v_{i+1}\}, \ldots, \{v_{i-1}, v_k\}$.

Algorithm 4: Triangulating the faces of G.

endpoints v and w to f (see Fig. 7(a)). We can add two such edges e_1 and e_2 only if one of the endpoints, say w, appears at least twice in a clockwise traversal of f's boundary. Then, however, there is at least one vertex x that has all its occurrences between two consecutive occurrences of w because e_1 and e_2 form a closed curve. That is, the marked copy of x also appears between these two occurrences of w. Adding e_2 to f would remove this marked copy from f's boundary; but we never remove marked vertices. Thus, e_2 cannot exist.

Now assume that we add two edges e_1 and e_2 with endpoints v and w to different faces f_1 and f_2. By adding e_1 we remove a vertex u from f_1's boundary (see Fig. 7(b)). As this copy is unmarked, there has to be another, marked, copy of u. Consider the region R_1 enclosed by the curve between the marked copy of u and the removed copy of u, following the boundary of f_1. In the same way we define a region R_2 enclosed by the curve between the removed copy of u and the marked copy of u. Any face other than f_1 that has v on its boundary must be in R_1. Any face other than f_1 that has w on its boundary must be in R_2. However, R_1 and R_2 are disjoint. Thus, we cannot add a diagonal $\{v, w\}$ to any face other than f_1. □

Step 2 triangulates all simple faces $\hat{f}$, possibly adding parallel edges to different faces. Consider all edges $e_1, \ldots, e_k$ with endpoints v and w. We have to remove at least $k-1$ of them. Also, if edge $\{v, w\}$ was already in G, we have to keep this edge and remove all diagonals that we have added later. That is, the edge $\{v, w\}$ is the edge with which all diagonals $\{v, w\}$ are in conflict, and we have to label all diagonals as conflicting while labelling edge $\{v, w\}$ as non-conflicting. If edge $\{v, w\}$ was not in G, we can choose an arbitrary diagonal $\{v, w\}$ with which all other diagonals with the same endpoints are in conflict. This strategy is realized in Step 3. The following lemma states the correctness of Step 4, thereby finishing the correctness proof for Algorithm 4.

Lemma 10 *Step 4 makes all faces $\hat{f}$ conflict-free, i.e. the triangulation obtained after Step 4 does not contain parallel edges.*

PROOF. Let $d = \{v_0, v_i\}$ be the edge found in Step 4b (see Fig. 8(a)). Then d cuts $\hat{f}$ into two halves, f_1 and f_2. All diagonals $\{v_0, v_j\}$, $j < i$ are in f_1; all diagonals $\{v_0, v_j\}$, $j > i$ are in f_2. That is, f_1 does not contain conflicting diagonals. Vertex v_{i-1} is the third vertex of the triangle in f_1 that has d on its boundary. Step 4c removes d and all edges in f_2 and retriangulates f_2 with diagonals that have v_{i-1} as one endpoint. (Intuitively it forms a star at the vertex v_{i-1}; see Fig. 8(b).)

Let d' be the edge that d is in conflict with. Then d and d' form a closed curve. Vertex v_{i-1} is outside this curve and all boundary vertices of f_2 excluding the

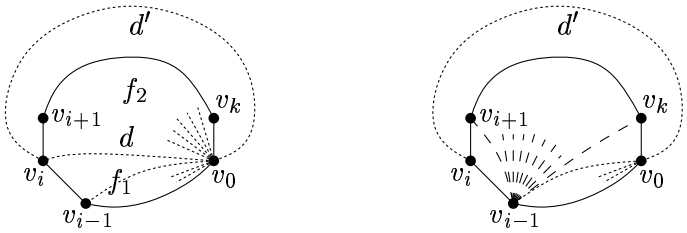

Fig. 8. (a) A simple face $\hat{f}$ with a conflicting diagonal edge $d = \{v_0, v_i\}$. Diagonal d conflicts with d' and divides $\hat{f}$ into two parts f_1 and f_2. One of them, f_1, is conflict-free. Vertex v_{i-1} is the third vertex of the triangle in f_1 that has d on its boundary. (b) The conflict-free triangulation of $\hat{f}$.

endpoints of d are inside this curve. As we keep d', no diagonal, except for the new diagonals in $\hat{f}$, can intersect this curve. Thus, the new diagonals in $\hat{f}$ are non-conflicting. The "old" diagonals in $\hat{f}$ were in f_1 and thus, by the choice of d and f_1, non-conflicting. Hence, $\hat{f}$ does not contain conflicting diagonals. □

Lemma 11 *Given the list F as computed by Algorithm 3, Algorithm 4 triangulates the given graph G in $O(sort(N))$ I/Os.*

PROOF. We have already shown the correctness of Algorithm 4. In Step 1, we first sort every list F_f by vertex names. Then it takes a single scan to mark the first occurrences of all vertices in F_f. Using another sort we restore the original order of the vertices in F_f. The rest of Step 1 requires scanning F, writing the marked vertices to $F_{\hat{f}}$, in their order of appearance, and keeping the last visited marked vertex in main memory in order to add the next diagonal. Thus, Step 1 takes $O(sort(N))$ I/Os. Step 2 requires a single scan over the list F, as modified by Step 1. Assuming that all edges in G before the execution of Step 2 were labelled as "edges", we label all edges added in Step 2 as "diagonals". Then Step 3 requires sorting the list of edges and diagonals lexicographically and scanning this sorted list to label conflicting diagonals. Note, however, that Step 4 requires the diagonals to be stored in the same order as added in Step 2. Thus, before sorting in Step 3, we label every edge with its current position in E. At the end of Step 3, we sort the edges by these position labels to restore the previous order of the edges. Of course, Step 3 still takes $O(sort(N))$ I/Os. Step 4 takes a single scan over E. Thus, the whole algorithm takes $O(sort(N))$ I/Os. □

A problem that we have ignored so far is embedding the new diagonals. Next we describe how to augment Algorithm 4 in order to maintain an embedding of G under the edge insertions performed by the algorithm. To do this, we have to modify the representation of the embedding slightly. Initially we assumed that the edges $e_1, \ldots, e_k$ incident to a vertex v are assigned labels $n_v(e_i) = i$

clockwise around v. During the triangulation process we allow labels $n_v(e)$ that are multiples of $\frac{1}{N}$. Note that this does not cause precision problems because we can represent every label $n_v(e)$ as an integer $N \cdot n_v(e)$ using at most $2 \log N$ bits (while $n_v(e)$ uses $\log N$ bits).

Let $v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}$ be the list of vertices and edges visited in a clockwise traversal of the boundary of a face f (i.e., $F_f = \langle v_0, \ldots, v_{k-1} \rangle$). During the construction of F_f, we can easily assign labels $n_1(v_i)$ and $n_2(v_i)$ to the vertices, where $n_1(v_i) = n_{v_i}(e_{(i-1) \bmod k})$ and $n_2(v_i) = n_{v_i}(e_i)$. When we add a diagonal $d = \{v_i, v_j\}$, $i < j$, we update the labels of v_i and v_j to $n_2(v_i) = n_2(v_i) - \frac{1}{N}$ and $n_1(v_j) = n_1(v_j) + \frac{1}{N}$ and embed d assigning $n_{v_i}(d) = n_2(v_i)$ and $n_{v_j}(d) = n_1(v_j)$. Assuming that $n_1(v_i) < n_2(v_i) - \frac{1}{N}$ or $n_1(v_i) \geq n_2(v_i)$ and $n_2(v_j) > n_1(v_j) + \frac{1}{N}$ or $n_2(v_j) \leq n_1(v_j)$, this embeds d between $e_{(i-1) \bmod k}$ and e_i at v_i's end and $e_{(j-1) \bmod k}$ and e_j at v_j's end. It remains to show that this assumption is always satisfied.

We maintain the following invariant for every face f: *Let $v_0, e_0, \ldots, v_{k-1}, e_{k-1}$ be the boundary description of f as given above. Then for every vertex v_i, either $n_1(v_i) + \frac{k-3}{N} < n_2(v_i)$ or $n_1(v_i) \geq n_2(v_i)$.* Initially, this is true because all labels $n_v(e)$ are integers and $k \leq N$. Adding diagonal d to f as above cuts f into two faces f_1 and f_2. For all vertices v_l, $l \notin \{i, j\}$, the labels do not change; but the sizes of f_1 and f_2 are less than the size of f. Thus, for all these vertices the invariant holds. We show that the invariant holds for v_i. A similar argument can be applied for v_j. Let f_1 be the face with vertices $v_j, \ldots, v_i$ on its boundary and f_2 be the face with vertices v_i and v_j on the boundary. Let the size of f_i be $k_i \leq k - 1$. If $n_2(v_i) \leq n_1(v_i)$, then this is also true after subtracting $\frac{1}{N}$ from $n_2(v_i)$. Otherwise, $n_2(v_i) - n_1(v_i) > \frac{k-3}{N} - \frac{1}{N} \geq \frac{k_1-3}{N}$. Thus, the invariant holds for all vertices on f_1's boundary. In f_2 we do not have any room left to add diagonals incident to v_i or v_j. However, Steps 1 and 4 of Algorithm 4 scan along the boundaries of the faces and keep cutting off triangles. Choosing the indices i and j in the above description so that f_2 is the triangle that we cut off, we never add another diagonal to f_2. (Note that it would be just as easy to maintain the embedding in Step 2; but we need not even do this because diagonals are added only temporarily in Step 2, and final diagonals are added in Step 4.)

6 The Shortest Path Data Structure

In this section, we incorporate the main ideas of the internal memory shortest path data structure in [8] and show how to use them together with the external memory techniques developed in this paper to design an efficient external memory shortest path data structure. The data structure in [8] uses $O(S)$ space and answers distance queries on graphs with separators of size $O\left(\sqrt{N}\right)$

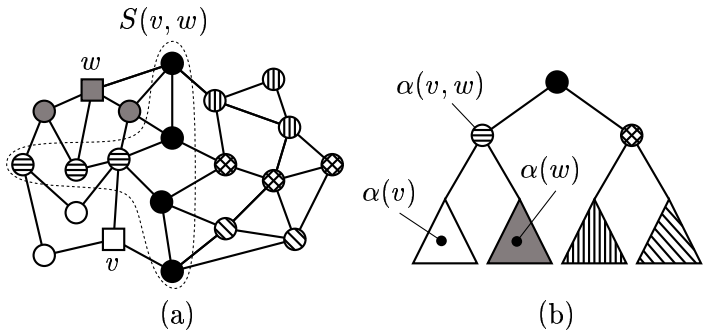

Fig. 9. A given planar graph G (a) and its separator tree $\mathrm{ST}(G)$ (b). Note that any path between vertices v and w, which are stored in the white and grey subtrees of $\mathrm{ST}(G)$, respectively, must contain a black or horizontally striped separator vertex. These separators are stored at common ancestors of $\alpha(v)$ and $\alpha(w)$.

in $O\left(N^2/S\right)$ time, where $1 \leq S \leq N^2$. Reporting the corresponding shortest path takes $O(K)$ time, where K is the number of vertices on the path. The basic structure used to obtain the above trade-off is an $O\left(N^{3/2}\right)$ size internal memory data structure that answers distance queries in $O\left(\sqrt{N}\right)$ time. Our external memory data structure is fully blocked. That is, it uses $O\left(N^{3/2}/B\right)$ blocks of external memory and answers distance queries in $O\left(\sqrt{N}/DB\right)$ I/Os. The corresponding shortest path can be reported in $O(K/DB)$ I/Os.

Given a planar graph G (see Fig. 9(a)), we compute a *separator tree* $\mathrm{ST}(G)$ for G (see Fig. 9(b)). This tree is defined recursively: We compute a $\frac{2}{3}$-separator $S(\rho)$ of size $O\left(\sqrt{|G|}\right)$ for G. Let $G_1, \ldots, G_k$ be the connected components of $G - S(\rho)$. Then we store $S(\rho)$ at the root ρ of $\mathrm{ST}(G)$ and recursively build separator trees $\mathrm{ST}(G_1), \ldots, \mathrm{ST}(G_k)$, whose roots become the children of ρ. Thus, every vertex v of G is stored at exactly one node $\alpha(v) \in \mathrm{ST}(G)$. For two vertices v and w, let $\alpha(v,w) = \mathrm{lca}(\alpha(v), \alpha(w))$. For a vertex $\alpha \in \mathrm{ST}(G)$, we define $S^-(\alpha)$ (resp. $S^+(\alpha)$) as the sets of vertices stored at α and its ancestors (resp. descendants). For convenience, we denote the sets $S(\alpha(v))$ and $S(\alpha(v,w))$ by $S(v)$ and $S(v,w)$, respectively. Sets $S^-(v)$, $S^+(v)$, $S^-(v,w)$ and $S^+(v,w)$ are defined analogously. In [8] it is shown that any path between v and w in G must contain at least one vertex in $S^-(v,w)$. For a given graph H, we denote the distance between two vertices v and w in H by $d_H(v,w)$. Then $d(v,w) = \min_{x \in S^-(v,w)}\{d_{G[S^+(x)]}(x,v) + d_{G[S^+(x)]}(x,w)\}$. If we represent all sets $S^-(v)$ and $S^-(v,w)$ as lists sorted by increasing depth in $\mathrm{ST}(G)$, then $S^-(v,w)$ is the longest common prefix of $S^-(v)$ and $S^-(w)$. Let $D(v)$ be a list of distances, where the i^{th} entry in $D(v)$ is the distance $d_{G[S^+(x)]}(v,x)$ between v and the i^{th} entry x in $S^-(v)$. Given the lists $S^-(v)$, $S^-(w)$, $D(v)$, and $D(w)$, we can compute $d(v,w)$ by scanning these four lists. We scan $S^-(v)$ and $S^-(w)$ in "lock-step" fashion and test whether $x_v = x_w$, where x_v and x_w are the current entries visited in the scans of the two lists. If $x_v = x_w$, then we proceed in the scans of $D(v)$ and $D(w)$ to compute $d_{G[S^+(x_v)]}(x_v,v) + d_{G[S^+(x_v)]}(x_v,w)$ and

compare it to the previously found minimum. If $x_v \neq x_w$, we have reached the end of $S^-(v,w)$ and stop.

For every separator vertex x, our data structure contains a shortest path tree $\mathrm{SPT}(x)$ with root x. This tree represents the shortest paths between x and all vertices in $G[S^+(x)]$. That is, every vertex v of $G[S^+(x)]$ is represented by a node in $\mathrm{SPT}(x)$, and the shortest path from v to x in $G[S^+(x)]$ corresponds to the path from v to x in $\mathrm{SPT}(x)$. Given the distance $d(v,w)$ between two vertices v and w, there must be some vertex $x_{\min} \in S^-(v,w)$ such that $d(v,w) = d_{G[S^+(x_{\min})]}(x_{\min},v) + d_{G[S^+(x_{\min})]}(x_{\min},w)$. The shortest path $\pi(v,w)$ from v to w is the concatenation of the shortest paths from v to $x_{\min}$ and from $x_{\min}$ to w. Given $\mathrm{SPT}(x_{\min})$, we can traverse the paths from v and w to the root $x_{\min}$ and concatenate the traversed paths to obtain $\pi(v,w)$. To traverse these two paths in the tree, we have to find the two (external) memory locations holding the two nodes representing v and w in $\mathrm{SPT}(x_{\min})$. We construct lists $P(v)$ for all vertices v of G holding pointers to the representatives of v in all shortest path trees. Let x be the separator vertex stored at the i^{th} position in $S^-(v)$. Then the i^{th} position of $P(v)$ holds a pointer to the node representing v in $\mathrm{SPT}(x)$. That is, if $x_{\min}$ is stored at position i in $S^-(v)$ and $S^-(w)$, the i^{th} positions of $P(v)$ and $P(w)$ hold the addresses of the representatives of v and w in $\mathrm{SPT}(x_{\min})$, giving us enough information to start traversing and reporting $\pi(v,w)$.

The size of the separator $S(\alpha)$ stored at every vertex α in the separator tree $\mathrm{ST}(G)$ is $O\left(\sqrt{|G[S^+(\alpha)]|}\right)$. From one level in $\mathrm{ST}(G)$ to the next the sizes of the subgraphs $G[S^+(\alpha)]$ associated with the vertices decrease by a factor of at least $\frac{2}{3}$. Thus, the sizes of the graphs $G[S^+(\alpha)]$ on a root-to-leaf path in $\mathrm{ST}(G)$ are bounded from above by a geometrically decreasing sequence, and the sizes of the separators $S(\alpha)$ stored at these vertices form a geometrically decreasing sequence as well. Hence, there are only $O\left(\sqrt{N}\right)$ separator vertices on any such path and each list $S^-(v)$ has size $O\left(\sqrt{N}\right)$. As we have to scan lists $S^-(v)$, $S^-(w)$, $D(v)$, and $D(w)$ to compute $d(v,w)$, this takes $O\left(\sqrt{N}/DB\right)$ I/Os. It takes two more I/Os to access the pointers in $P(v)$ and $P(w)$. Assuming that all shortest path trees have been blocked as described in Section 3, traversing the paths from v and w to $x_{\min}$ in $\mathrm{SPT}(x_{\min})$ takes $O(K/DB)$ I/Os, where K is the number of vertices on the shortest path $\pi(v,w)$.

It remains to show that the data structure can be stored in $O\left(N^{3/2}/B\right)$ blocks. As every list $S^-(v)$, $D(v)$, and $P(v)$ has size $O\left(\sqrt{N}\right)$ and there are $3N$ such lists, all lists require $O\left(N^{3/2}\right)$ space and can thus be stored in $O\left(N^{3/2}/B\right)$ blocks. There is exactly one shortest path tree for every separator vertex. Consider tree $\mathrm{SPT}(x)$ and a node v in this tree. Then this node corresponds to the entry for x in list $S^-(v)$. That is, there is a one-to-one correspondence

between entries in the lists $S^-(v)$ and shortest path tree nodes. Thus, the total size of all shortest path trees is $O\left(N^{3/2}\right)$ as well. As blocking the shortest path trees increases the space requirements by only a constant factor, the shortest path trees can also be stored in $O\left(N^{3/2}/B\right)$ blocks of external memory. We have shown the following theorem.

Theorem 12 *Given a planar graph G with N vertices, one can construct a data structure that answers distance queries between two vertices in G in $O\left(\sqrt{N}/DB\right)$ I/Os. The corresponding shortest path can be reported in $O(K/DB)$ I/Os, where K is the length of the reported path. The data structure occupies $O\left(N^{3/2}/B\right)$ blocks of external memory.*

7 Conclusions

The I/O-efficient construction of our shortest path data structure is still a problem, as there are no algorithms for BFS, embedding, and the single source shortest path problem that perform I/O-efficiently on planar graphs. The separator algorithm in Section 4 tries to address the problem of computing the separators required to build the separator tree I/O-efficiently. Also, in [26] an $O(sort(N))$ algorithm for transforming a given rooted tree of size N into the blocked form described in Section 3 is given.

A shortcoming of our separator algorithm is that it needs a BFS-tree. Most separator algorithms rely on BFS, but breadth-first search and depth-first search seem to be hard problems in external memory. Thus, it is an important open problem to develop an I/O-efficient separator algorithm that does not need BFS or DFS. The existence of an I/O-efficient planar embedding algorithm is also open.

Recently, Maheshwari and Zeh [18] presented $O(sort(N))$ algorithms for outerplanarity testing, computing an outerplanar embedding, BFS, DFS, and computing a $\frac{2}{3}$-separator of a given outerplanar graph. It is an important question whether there are other interesting classes of graphs with similar I/O-complexities for these problems. They also showed $\Omega(perm(N))$ lower bounds for computing an outerplanar embedding, and BFS and DFS in outerplanar graphs. As outerplanar graphs are also planar, the lower bounds for BFS and DFS also apply to planar graphs. A similar technique as in [18] can be used to show that planar embedding has an $\Omega(perm(N))$ lower bound.

Acknowledgements: The authors would like to thank Lyudmil Aleksandrov, Jörg-Rüdiger Sack, Hans-Dietrich Hecker, and Jana Dietel for their critical and helpful remarks.

References

[1] L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal of Discrete Mathematics*, 9:129–150, 1996.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345, 1995.

[3] 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*, January 1995.

[4] A. Crauser, K. Mehlhorn, and U. Meyer. Kürzeste-Wege-Berechnung bei sehr großen Datenmengen. In O. Spaniol, editor, *Promotion tut not: Innovationsmotor "Graduiertenkolleg"*, volume 21 of *Aachener Beiträge zur Informatik*. Verlag der Augustinus Buchhandlung, 1997.

[5] 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.

[6] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[7] K. Diks, H. N. Djidjev, O. Sykora, and I. Vrto. Edge separators of planar and outerplanar graphs with applications. *Journal of Algorithms*, 14:258–279, 1993.

[8] 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.

[9] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, December 1987.

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

[11] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[12] H. Gazit and G. L. Miller. A parallel algorithm for finding a separator in planar graphs. In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 238–248, 1987.

[13] M. T. Goodrich. Planar separators and parallel polygon triangulation. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 507–516, 1992.

[14] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proceedings of the 5th ACM-SIAM Computing and Combinatorics Conference*, volume 1627 of *Lecture Notes in Computer Science*, pages 51–60. Springer Verlag, July 1999.

[15] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 27–37, May 1994.

[16] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 274–283, June 1997.

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

[18] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *ISAAC'99 (to appear)*, Lecture Notes in Computer Science. Springer Verlag, December 1999.

[19] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999.

[20] M. Nodine, M. Goodrich, and J. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, August 1996.

[21] F. P. Preparata and M. I. Shamos. *Computational Geometry — An Introduction*. Springer Verlag, 1985.

[22] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[23] J. S. Vitter. External memory algorithms. In *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems*, June 1998.

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

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

[26] N. Zeh. *An External-Memory Data Structure for Shortest Path Queries*. Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, November 1998.