An External-Memory Data Structure for Shortest Path Queries

David Hutchinson*§  Anil Maheshwari*§  Norbert Zeh*††

Abstract

In this paper, we present results related to satisfying shortest path queries on a planar graph stored in external memory. \( N \) denotes the total number of vertices and edges in the graph and \( \text{sort}(N) \) denotes the number of input/output (I/O) operations required to sort an array of length \( N \).

1) We describe a data structure for supporting bottom-up traversal of rooted trees in external memory. A tree of size \( S \) is stored in \( O(S/B) \) blocks, and traversing a path of length \( K \) towards the root in this tree takes \( O(K/B) \) I/Os.

2) We give an algorithm for computing a separator for an embedded planar graph in \( O(\text{sort}(N)) \) I/Os, provided that a breadth-first search (BFS) tree is given.

3) We describe an algorithm for triangulating an embedded planar graph in \( O(\text{sort}(N)) \) I/Os.

Using these results, we can obtain a data structure for shortest path queries on graphs with separators of size \( O(\sqrt{N}) \) that uses \( O(N^{3/2}/B) \) blocks of external memory and allows for answering shortest path queries in \( O((\sqrt{N}+K)/DB) \) I/Os, where \( K \) is the number of vertices on the reported path.

Keywords: External-memory algorithms, graph algorithms, shortest paths, planar graphs

1 Introduction

Motivation: 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 an equivalent combinatorial structure, which is often a weighted, planar graph.

The motivation to study external memory shortest path problems arose in our GIS research (see [17]), and in particular, with an implementation of the results of Lanthier, Maheshwari, and Sack [13] for shortest path problems in triangular irregular networks. In this application, the given graph represents a planar map, i.e., 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 the fast internal memory and relatively slow external (disk-based) memory. Thus, the classical internal memory approaches to answering shortest path queries in a planar graph (e.g. [6, 7, 8]) may not work efficiently when the data sets are too large.

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 [18]. 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) [20], as well as a number of other external memory models [21, 4]. 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.

Model of Computation: In the PDM, an external memory, consisting of \( D \) disks, is attached to a machine with memory size \( M \) 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.

Previous Results: We distinguish between three kinds of shortest path problems: (1) computing a shortest path between two given vertices of the graph, (2) computing shortest paths between a given source vertex and all other vertices of the graph (single source shortest paths (SSSP) problem), and (3) computing the shortest paths between all pairs of vertices in the graph (all pairs shortest paths (APSP) problem).

In the sequential RAM model, much work has been done on these problems. Dijkstra’s algorithm [5], when implemented using Fibonacci heaps [9], is the best known algorithm for the SSSP-problem for general graphs (with nonnegative edge weights). It runs in \( O(\min \{ |E| + |V| \log |V| \}) \) time, where \(|E|\) and \(|V|\) are the number 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 \( O(|V|^2 + |V|^2 \log |V|) \) running time. For planar graphs, Frederickson [7] gave an \( O(N \sqrt{\log N}) \)-algorithm for the SSSP-problem and an \( O(N^2) \)-algorithm for the APSP-problem, where \( N = |V| \). Klein et al. [12] presented a linear-time SSSP-algorithm for planar graphs.

An alternate approach is to preprocess the given graph for online shortest path queries. For graphs for which an \( O(\sqrt{N}) \)-separator theorem holds (e.g., planar graphs), Djidjev [6] presented an \( O(S) \)-space data structure \( (N \leq S \leq N^2) \) that answers distance queries in \( O(N^2/S) \) time. The corresponding shortest path can be reported in time proportional to the length of the reported path. (For planar graphs he presents slightly better bounds.)

It is known that every tree or outerplanar graph has a \( \frac{2}{3} \)-separator of size \( O(1) \). Lipton and Tarjan [14] showed that every planar graph has a \( \frac{2}{3} \)-separator of size \( O(\sqrt{N}) \) and presented a linear-time algorithm for finding such a separator.

In the PDM, sorting an array of size \( N \) takes \( \text{sort}(N) = \Theta \left( \frac{N}{\sqrt{\log N}} \right) \) I/Os [20, 19]. Scanning an array of size \( \tilde{N} \) takes \( \text{scan}(\tilde{N}) = \Theta \left( \frac{\tilde{N}}{\sqrt{\log \tilde{N}}} \right) \) I/Os. For a comprehensive survey on external memory algorithms, refer to [19]. The only external-memory shortest path algorithm known to us is the SSSP-algorithm by Crauser, Mehlhorn, and Meyer [3], which takes \( O \left( \left| V \right| \frac{1}{D} + \left| E \right| \frac{1}{DB} \log \frac{1}{DB} \right) \) 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 of Chiang et al. [2] together with the following results. Gazit and Miller [10] gave a parallel algorithm for the PRAM model that computes a \( \frac{2}{3} \)-separator of size \( O(\sqrt{N}) \) for a planar graph. Their algorithm runs in \( O((\log^2 N) \log \log N) \) time and uses \( O(N^{1+\epsilon}) \) processors, where \( \epsilon > 0 \) is a constant. Goodrich [11] presented a PRAM algorithm for separating planar graphs that runs in \( O(\log N) \) time and uses \( O(N/\log N) \) processors, provided that a BFS-tree of the graph is given. Unfortunately, the PRAM simulation introduces \( O(\text{sort}(N)) \) I/Os for every PRAM step, and so the resulting I/O complexity is not attractive for this problem.

Our Results: The main results of this paper are:

1. An external memory data structure to store a rooted tree \( T \) of size \( N \) in at most \( \left( 2 + \frac{2}{1-\tau} \right) \frac{N}{\sqrt{B}} + D \) blocks so that a path of length \( K \) towards the root can be traversed in at most \( \left\lceil \frac{K}{DB} \right\rceil + 1 \) I/Os, for \( 0 < \tau < 1 \). For fixed \( \tau \), the tree uses optimal \( O(|T|/B) \) space and traversing a path takes optimal \( O(K/DB) \) I/Os. Using the best previous result by Nodine, Goodrich, and Vitter [16], the tree would use the same amount of space within a constant factor, but traversing
a path would take $O(K/\log_d(DB))$ 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 of size $O(\sqrt{N})$ 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 theorem of Lipton and Tarjan [14]. 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/O operations. (See Section 5.)

4. Results 1-3, above, are the main techniques that we use to construct an external memory data structure for answering shortest path queries online. Our data structure uses $O(N^{3/2}/B)$ blocks of external storage and answers online distance and shortest path queries in $O(\sqrt{N}/DB)$ and $O((\sqrt{N}+K)/DB)$ I/Os, respectively, where $K$ is the number of vertices on the path. Due to space limitations, we ask the reader to consult [22] for details of this data structure.

The 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

A graph $G = (V, E)$ 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 $V$. We can draw $G$ in the plane representing its vertices as points and its edges as continuous curves connecting their endpoints. A graph $G$ is called 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 $p = \langle v = v_0, \ldots, v_k = w \rangle$ of vertices, where $\{v_i, v_{i+1}\} \in E$ for $0 \leq i < k$. 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 $E' \subseteq E$. Connected components of $G$ are the maximal connected subgraphs of $G$.

Let $c : E \rightarrow \mathbb{R}^+$ be a mapping that assigns non-negative weights to the edges of $G$. The weight 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 from $v$ to $w$ of minimal weight.

Let $w : V \rightarrow \mathbb{R}^+$ be a mapping that assigns non-negative weights to the vertices of $G$ such that $\sum_{v \in V} w(v) \leq 1$. The weight of a subgraph $H$ of $G$ is the sum of the weights of the vertices in $H$. An $\epsilon$-separator, $0 < \epsilon < 1$, of $G$ is a subset $C$ of $V$ whose removal partitions $G$ into two subgraphs, $A$ and $B$, each of weight at most $\epsilon$, so that there is no edge in $G$ that connects any vertex in $A$ to any vertex in $B$.

We will describe results on paths in a 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.

3 Blocking Rooted Trees

In this section we describe an external memory data structure to store a rooted tree $T$ so that, given a vertex $v$ of $T$, we can traverse a bottom-up path from $v$ in an I/O-efficient manner. We assume that all accesses to $T$ are read-only. Thus, we can store each vertex an arbitrary number of times and thereby use redundancy to reduce the number of blocks that have to be read. However, this increases the space requirement. The following theorem gives a trade-off between the space requirements of the data structure and the I/O-efficiency of the tree traversal.
Theorem 3.1 Given a rooted tree $T$ of size $N$ and a constant $\tau$, $0 < \tau < 1$, we can store $T$ in at most $\left(2 + \frac{2}{1-\tau}\right)\frac{N}{\tau} + D$ blocks on $D$ parallel disks so that traversing any bottom-up path of length $K$ in $T$ takes at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 1$ I/Os.

Proof. This follows from Lemmas 3.1 and 3.2, which follow.

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

More precisely, let $h(T)$ 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). Let the level of a vertex $v$ be the number of edges in the path from $v$ to the root $r$ of $T$. Cut $T$ into layers $L_0, \ldots, L_{\lfloor h(T)/h' \rfloor}$, where layer $L_i$ is the subgraph of $T$ induced by the vertices on levels $ih'$ through $(i+1)h' - 1$ (see Figure 3). Each layer is a forest of rooted trees whose heights are at most $h'$. Suppose that there are $r$ such trees, taken over all layers. Let $T_1, \ldots, T_r$ denote these trees for all layers of $T$. We divide each tree $T_i$, for $1 \leq i \leq r$, into subtrees $T_{i,0}, \ldots, T_{i,s}$, with 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 $T_i$.

Lemma 3.1 Given a rooted tree $T_i$ of height at most $\tau DB$, we can divide $T_i$ into subtrees $T_{i,0}, \ldots, T_{i,s}$, where the collection of subtrees has Properties 1–3.

Proof. If $|T_i| \leq DB$, we “divide” $T_i$ 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 $T_i$, let $v_k$ be the vertex with preorder number $k$. Let $h' = \tau DB$, $t = DB - h'$, and $s = \left\lfloor \frac{|T_i|}{t} \right\rfloor - 1$. We define vertex sets $V_0, \ldots, V_s$, where $V_j = \{v_{jt}, \ldots, v_{j(t+1)}\}$ for $0 \leq j \leq s$ (see Figure 4(a)). The subtree $T_{i,j} = T_i(V_j)$ is the subtree of $T_i$ consisting of all vertices in $V_j$ and their ancestors in $T_i$ (see Figure 4(b)). We claim that these subtrees $T_{i,j}$ have Properties 1–3.

Property 3 is ensured by including the ancestors in $T_i$ of all vertices in $V_j$ in $T_i(V_j)$. Property 2 follows, if we can prove Property 1 because

$$\sum_{j=0}^{s} |T_{i,j}| \leq \sum_{j=0}^{s} DB \leq \left(\frac{|T_i|}{DB - \tau DB} + 1\right) DB \leq \left(\frac{1}{1 - \tau} + 1\right) |T_i|.$$

It can be shown that every vertex in $T_i(V_j)$ that is not in $V_j$ is an ancestor of $v_{jt}$. As the height of $T_i$ is at most $h'$, there can be at most $h'$ such ancestors of $v_{jt}$. Moreover, $|V_j| \leq DB - h'$, thus, $|T_i(V_j)| \leq DB$.

Lemma 3.2 If a rooted tree $T$ of size $N$ is partitioned into subtrees $T_{i,j}$ such that Properties 1–3 hold, then $T$ can be stored using $\left(2 + \frac{2}{1-\tau}\right)\frac{N}{\tau} + D$ blocks of external memory and any bottom-up path of length $K$ in $T$ can be traversed in at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 1$ I/Os.

Proof. See Appendix A.

4 Separating Embedded Planar Graphs

We now present an external-memory algorithm for separating embedded planar graphs. Our algorithm is based on Lipton and Tarjan’s [14] linear-time separator algorithm. It computes a $\frac{2}{3}$-separator $S$ of size $O(\sqrt{N})$ for a given embed-
d planar graph \( G \) in \( O(sort(N)) \) I/Os, provided that we are given a BFS-tree of the graph\(^1\).

The input to our algorithm is an embedded planar graph \( G \) and a spanning forest \( F \) of \( G \). Every tree in \( F \) is a rooted BFS-tree of the respective connected component. The graph \( G \) is represented by its vertex set \( V \) and its edge set \( E \). To represent the embedding, let the edges incident to a vertex \( v \) be numbered in a 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 or non-tree edge and storing, with each vertex \( v \) in \( V \), the name of its parent in \( F \).

**Framework of the Algorithm:** Our algorithm is modelled on the internal-memory algorithm by Lipton and Tarjan \([14]\). First we compute connected components of the given graph \( G \). If there is a component whose weight (i.e. the sum of the weights of its vertices) is greater than \( \frac{3}{4} \), we compute a separator \( S \) of that component. Finally the connected components of the resulting graph \( G - S \) are collected into two groups to obtain the desired partition.

The connected components can be computed in the PDM by using the algorithm of Chiang et al. \([2]\) in \( O(sort(N)) \) I/Os. In the next subsection, we describe how to compute the separator \( S \) using \( O(sort(N)) \) I/Os, leading to the following theorem.

**Theorem 4.1** Given an embedded planar graph \( G \) with \( N \) vertices and a BFS-tree \( T \) of \( G \), a \( \frac{3}{4} \)-separator of \( G \) of size at most \( 2\sqrt{2}\sqrt{N} \) can be computed in \( O(sort(N)) \) I/Os.

**Separating Connected Planar Graphs:** In this section we present an external memory algorithm for computing a \( \frac{3}{4} \)-separator of size \( O(\sqrt{N}) \) for a connected planar graph \( G \) of size \( N \) whose breadth first search tree \( T \) is given and the sum of weights of vertices in \( G \) is larger than \( \frac{3}{4} \). 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.

**Algorithm 1:** Lipton and Tarjan’s \([14]\) algorithm for computing planar separator.

**Lemma 4.1** Algorithm 1 computes a \( \frac{3}{4} \)-separator \( S \) of size at most \( 2\sqrt{2}\sqrt{N} \) for 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.** The separator \( S \) consists of two parts: (i) two levels \( l_0 \) and \( l_2 \) in \( T \) whose removal divides \( G \) into three subgraphs and (ii) a simple cycle separator, corresponding to a fundamental cycle.

Let \( l_1 \) be the level closest to the root of \( T \) such that the total weight of levels 0 through \( l_1 \) exceeds \( \frac{1}{2} \). Let \( K \) be the number of vertices on levels 0 through \( l_1 \). We define 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 \( L(l_2) + 2(l_2 - l_1 - 1) \leq 2\sqrt{N - K} \), where \( L(l) \) denotes the number of vertices on level \( l \). Lipton and Tarjan \([14]\) showed that such levels \( l_0 \) and \( l_2 \) exist and that the vertex set \( S \) computed by Algorithm 1 is indeed a \( \frac{3}{4} \)-separator for \( G \) of size at most \( 2\sqrt{2}\sqrt{N} \).

Step 1 of the algorithm can be implemented using a generalization of the list ranking algorithm by Chiang et al. \([2]\) (see \([22]\)). Then we sort \( V \) by level numbers and scan \( V \) to compute \( l_1, l_0, \) and \( l_2 \). Extracting the vertex set of \( G_2 \) in Step 3 takes another scan. The edge set can be extracted with a sort and a scan operation (see \([22]\) for details).

---

\(^1\)The currently best known algorithm for computing a BFS-tree \([15]\) takes \( O\left(|V| + \frac{1}{4} sort(|V|)\right) \) I/Os.
In Step 4, we first check whether $G_2$ contains a vertex $v$ of weight greater than $\frac{1}{2}$. If this is the case, $S' = \{v\}$. Otherwise, we augment $G_2$ and the restriction of $T$ to $G_2$ to a triangulation $G'$ with BFS-tree $T'$. We add a vertex $v$ with weight $w(v) = 0$ and make all vertices on level $l_0 + 1$ children of $v$. We mark the newly added edges as tree edges. Then, we triangulate the resulting graph using the algorithm in Section 5 and it requires $O(sort(N))$ I/Os. In the next subsection, we show how to compute a $\frac{2}{3}$-simple cycle separator of size at most $2(l_2 - l_0) - 1$ for $G'$. Since $G_2$ is a subgraph of $G'$, this gives us a $\frac{2}{3}$-separator of $G_2$. Moreover, at most $2(l_2 - l_0 - 1)$ vertices of the separator are in $G_2$.

**Finding a Small Simple Cycle Separator:**
We use the notation of the previous subsection. 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 Figure 1). Clearly 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. Lipton and Tarjan showed that there is a non-tree edge $e$ in $G'$ such that $R_1(e)$ and $R_2(e)$ have weights at most $\frac{1}{2}$ each. Moreover, for any non-tree edge $e$, the number of vertices on the fundamental cycle $c(e)$ is at most $2h(T') - 1 = 2(l_2 - l_0) - 1$.

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

1. Compute the vertex and edge labels required to compute the weights of vertices in $R_1(e)$ and $R_2(e)$, for every non-tree edge $e$.
   Label every vertex $v$ in $G'$ with a tuple $A(v) = (W(v), n_p(v), \delta(r, v), v_p(v))$, where
   - $W(v)$ is the weight of the subtree of $T'$ rooted at $v$,
   - $n_p(v)$ is $v$'s preorder number in $T'$,
   - $\delta(r, v)$ is the total weight of all ancestors of $v$ in $T'$, inclusive, and
   - $v_p(v) = \sum_{i=0}^{t} w_i$ ("weighted preorder number of $v$").
   For every edge $e = \{v, w\}$,
   - Compute the lowest common ancestor $u$ of $v$ and $w$ and copy the tuples $A(u)$, $A(v)$, and $A(w)$ to $e$ and
   - Compute a label $t(e)$ defined as
     \[
     t(e) = \begin{cases} 
     0 & \text{if } e \text{ is a non-tree edge} \\
     W(v) & \text{if } e \text{ is a tree edge, } w = p(v).
     \end{cases}
     \]
   For every vertex $v$, let $e_0, \ldots, e_d$ be the set of edges incident on a vertex $v$ in counterclockwise order around $v$, where $e_0 = \{v, p(v)\}$. Compute labels
     \[
     t_p^i(e_i) = \begin{cases} 
     0 & \text{if } i = 0 \\
     t_p^i(e_{i-1}) + t(e_i) & \text{if } i > 0.
     \end{cases}
     \]
   2. Scan $E'$ and compute, for every non-tree edge $e$, the sum of weights in $R_1(e)$ and $R_2(e)$ as in the proof of Lemma 4.2. Choose a non-tree edge $e$ where weights of both $R_1(e)$ and $R_2(e) \leq \frac{2}{3}$.
   3. Report the fundamental cycle $c(e)$.

**Lemma 4.2** Given a triangulated graph $G'$ with $N$ vertices and a BFS-tree $T'$ of $G'$. A $\frac{2}{3}$-simple cycle separator of size at most $2h(T') - 1$ for $G'$ can be computed in $O(sort(N))$ I/Os.

**Proof sketch.** We use Algorithm 2 to compute a simple cycle separator $c(e)$ of $G'$. Please note that some notation is defined in the statement of Algorithm 2. Assume that the preorder numbers $n_p(v)$ and $v_p(v)$ computed in Step 1 respect the embedding. That is, if $p(v)$, $w_1$, and $w_2$ appear in the counterclockwise order around the vertex.
Consider Figure 1. A non-tree edge \( e = (v, w) \) is shown and \( u \) is the lowest common ancestor of \( v \) and \( w \) in \( T' \). Let the vertex \( v \) be the lowest common ancestor of \( v \) and \( w \). As \( T' \) is a BFS-tree of \( G' \), \( v \neq u \neq w \). Four classes of subtrees are indicated by different shadings in Figure 1. The vertices in the black subtree are not important to the algorithm, but are included for completeness. The set of vertices \( R_1 \), embedded inside the cycle, are the vertices in the light grey and dark grey subtrees. The vertices in the white and light grey subtrees and on the tree path from \( u \) to \( w \) are exactly the vertices with preorder numbers between \( n_p(v) \) and \( n_p(w) \). Thus, their total weight is \( \nu_p(w) - \nu_p(v) \). The total weight of the vertices in the white trees is \( t_p^v(e) \); the total weight of the vertices on the path from \( u \) to \( w \) is \( \delta(r, w) - \delta(r, u) \); the total weight of the dark grey trees is \( t_p^e(e) \). Thus, the total weight of vertices in \( R_1 \) is \( \nu_p(w) - \nu_p(v) - t_p^v(e) + t_p^e(e) - \delta(r, w) + \delta(r, u) \).

The total weight of the vertices on \( c(e) \) is \( \delta(r, v) + \delta(r, w) - 2\delta(r, u) + w(u) \). Thus, the total weight of vertices embedded outside the cycle is \( w(G') - w(R_1) - \delta(r, v) - \delta(r, w) + 2\delta(r, u) - w(u) \). It therefore follows that Step 2 of Algorithm 2 computes the desired separator.

Now we show the I/O-complexity of Algorithm 2. The preorder numbering algorithm in [22] takes \( O(sort(N)) \) I/Os. All other vertex labels can be computed with \( O(sort(N)) \) I/Os using the time-forward processing technique in [2, 1]. The lowest common ancestors for the vertices corresponding to each non-tree edge can be computed with \( O(sort(N)) \) I/Os [2] because there are only \( O(N) \) edges. Copying the data from the vertices to the respective edges again takes a constant number of sorting and scanning passes over \( V' \) and \( E' \). Computing labels \( t(e) \) takes \( O(sort(N)) \) I/Os. Labels \( t_p^v(e) \) can then be computed by sorting and scanning \( E' \). All labels used to compute the weights of the two regions in Step 2 are already stored locally with \( e \). Thus, we only have to scan \( E' \) once to compute these weights for all non-tree edges and find the desired non-tree edge \( e \). Now \( c(e) \) can be reported with \( O(sort(N)) \) I/Os. Thus, Algorithm 2 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, we use this information to triangulate the faces of \( G \). We will show the following theorem.

**Theorem 5.1** Given an embedded planar graph \( G \), it can be triangulated with \( O(sort(N)) \) I/Os.

**Proof.** This follows from Lemmas 5.1 and 5.4.

**Identifying Faces:** Let \( F \) be the concatenation of the lists of vertices around each face of \( G \). Let \( F_f \) denote the clockwise sequence of vertices around face \( f \). The list \( F_f \) may contain more than one copy of the same vertex, depending on how often this vertex is visited in a clockwise traversal of the face boundary.

Algorithm 3 computes \( F \). The directed graph \( \tilde{G} \) is comprised of disjoint directed cycles. Each cycle represents a clockwise traversal of the boundary of a face \( f \) of \( G \). Given a cycle in \( \tilde{G} \) that represents a face \( f \) of \( G \), every vertex in this cycle represents an edge on the boundary of \( f \). We construct \( F_f \) as the list of first endpoints of these edges in clockwise order around \( f \) (see Figure 5).
**Algorithm 3: Identifying the faces of $G$.**

**Lemma 5.1** The list $F$ can be constructed in $O(\text{sort}(N))$ I/Os by Algorithm 3.

*Proof sketch.* 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. We can sort the vertices in these paths, which sorts the vertices in $\hat{G}$ in each face of $G$ clockwise around the face. The rest of Step 2 scans the sorted vertex list of $\hat{G}$ and constructs the lists $F_f$ in the way described above.

The graph $\hat{G}$ can be constructed by sorting and scanning $E$, thus in $O(\text{sort}(N))$ I/Os. We can compute the connected components of $\hat{G}$ using the algorithm by Chiang et al. [2]. To remove one edge per component, we sort $E$ by component labels and scan it to remove the first edge of each component. The resulting graph is a set of paths or lists of total length $O(N)$. We can rank these lists with $O(\text{sort}(N))$ I/Os using a generalization of the list-ranking algorithm in [2] (see [22]). The rest of Step 2 takes $O(\text{sort}(N))$ I/Os.

Thus, Step 2 takes $O(\text{sort}(N))$ I/Os.

**Triangulating Faces:** We triangulate each face $f$ in four steps (see Algorithm 4). First, we reduce $f$ to a simple face $\hat{f}$. (A face is simple if each vertex on its boundary is visited only once in a clockwise traversal of the boundary.) This reduces the list $F_f$ to $F_{\hat{f}}$. In the second step, we triangulate $\hat{f}$ and the triangulation ensures that there are no multiple edges in $\hat{f}$, but we might add conflicting edges (edges with the same endpoints) to adjacent faces. See Figure 2 for an example. In the third step, we detect all such conflicting edges. In the fourth step we retriangulate all faces $f$ so that conflicts are resolved and a final triangulation is obtained.

**Algorithm 4: Triangulating the faces of $G$.**

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 $F_f$, and (c) scan $F_f$ backward and remove each unmarked vertex $v$ from $f$ and $F_f$ by adding a diagonal edge between its predecessor and successor in the current list.

2: Triangulate the simple faces:
   Let $F_{\hat{f}} = \{v_0, \ldots, v_k\}$. Then add “temporary diagonal edges” $\{v_0, v_i\}$, $2 \leq i \leq k - 1$, to $\hat{f}$.

3: Mark conflicting diagonals:
   Sort $E$ by endpoints and so that edge $\{v, w\}$ is stored before the “temporary diagonal edge” $\{v, w\}$. Scan $E$ and mark all occurrences except the first of each edge as conflicting. Restore the original order of all edges and temporary diagonal edges.

4: Retriangulate conflicting faces:
   For each face $\hat{f}$, let $D_f = \{v_0, v_1, \ldots, v_k\}$ be the list of “temporary diagonal edges”.
   Add $D_f$ until we find the first conflicting diagonal edge $\{v_0, v_i\}$. Replace $\{v_0, v_i, \ldots, v_0, v_i, \ldots, v_0, v_{k-1}\}$ by diagonal edges $\{v_{i-1}, v_{i+1}\}, \ldots, \{v_{i-1}, v_k\}$.
Step 2 triangulates all simple faces $f$. However, we may add the same diagonal edge \{v, w\} to several faces $f_1, \ldots, f_k$. It can also happen that \{v, w\} is already an edge of $G$. If \{v, w\} \in $G$, we have to remove the diagonals \{v, w\} from all other $k$ faces where we have added it. Otherwise, we have to remove it from $k - 1$ of them. In Step 3, we mark the respective diagonal edges as conflicting. Now we need to show that the output of Step 4 is a conflict-free triangulation of $G$.

**Lemma 5.3** Step 4 makes all faces $\hat{f}$ conflict-free, i.e., the graph obtained after Step 4 is simple.

**Proof sketch.** Let $d = \{v_0, v_1\}$ (see Figure 2). Then $d$ cuts $\hat{f}$ into two halves, $f_1$ and $f_2$. All diagonal edges $\{v_0, v_j\}$, $j < i$ are in $f_1$; all diagonal edges $\{v_0, v_j\}$, $j > i$ are in $f_2$. That is, $f_1$ does not contain conflicting diagonal edges. Let $u = v_{i-1}$ be the third vertex of the triangle in $f_1$ that has $d$ as one of the edges on the boundary of the triangle. Step 4 removes $d$ and all edges in $f_2$ and retriangulates $f_2$ with diagonals that have $u$ as one endpoint (Intuitively it forms a star at the vertex $u$).

Let $d'$ be the edge that is in conflict with $d$. Then $d$ and $d'$ form a closed curve and $u$ is outside this curve. All boundary vertices of $f_2$ excluding the endpoints of $d$ are inside this curve. As no diagonal, except for the new diagonals in $\hat{f}$, can intersect this curve, the new diagonal edges in $\hat{f}$ are non-conflicting. The “old” diagonal edges 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 diagonal edges.

Marking the first appearances of vertices in each list $F_f$ can be done with $O(sort(N))$ I/Os as follows. First sort $F_f$ by vertex numbers. Then scan $F_f$ and mark the first vertex and restore the original order of $F_f$. The rest of step 1 takes $O(scan(N))$ I/Os. Steps 2–4 take $O(sort(N))$ I/Os. The following lemma summarizes our result.

**Lemma 5.4** Given the list $F$ as defined in the previous section, the graph $G$ can be triangulated in $O(sort(N))$ I/Os.

In order to use this algorithm as part of our separator algorithm, we also have to embed the diagonals in the faces.

Let $v_1, e_1, v_2, e_2, \ldots, v_k, e_k$ be the list of vertices and edges visited in a clockwise traversal of the boundary of a face $f$ (i.e., $F_f = \langle v_1, \ldots, v_k \rangle$). We define labels $n_1(v_i) = n_{v_i}(e_{(i-1)_{mod k}})$ and $n_2(v_i) = n_{v_i}(e_i)$, and store them with $v_i$ in $F_f$. When we add a diagonal $d$ incident to vertex $v_i$, we give it a label $n_{w_i}(d)$ which is a rational value between $n_1(v_i)$ and $n_2(v_i)$. (To avoid problems related to arithmetic precision, we assign the new label as an offset of $\frac{1}{2}$ from either $n_1(v_i)$ or $n_2(v_i)$.) This embeds $d$ between $e_{(i-1)_{mod k}}$ and $e_i$. After that, the labels $n_1(v_i)$ and $n_2(v_i)$ are updated to ensure that subsequent diagonals are embedded between $e_{i-1}$ and $d$.

We can maintain these labels $n_1(v)$ and $n_2(v)$ for all vertices in the lists $F_f$ while executing our triangulation algorithm without increasing the number of I/O-operations by more than a constant factor. For details we refer the reader to [22].

## 6 Conclusions

A shortcoming of our separator algorithm is that it needs a BFS-tree. All separator algorithms known to the authors rely on BFS, but breadth-first search and depth-first search seem to be difficult problems in external memory. Thus, it is an important open problem to either develop an alternative separator algorithm that does not need BFS or DFS, or show that such an alternative is not possible. Also, the existence of an I/O-efficient graph embedding algorithm is open.

**Acknowledgements:** The authors would like to thank Lyudmil Aleksandrov, Jörg-Rüdiger Sack, Hans-Dietrich Hecker, and Jana Dietel for their critical and helpful remarks.
Figure 2: (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 $f_1$ where $d$ is one of the edges of the triangle boundary. (b) The conflict-free triangulation of $\hat{f}$.

References


A Appendix

Lemma 3.2 If a rooted tree $T$ of size $N$ is partitioned into subtrees $T_{i,j}$ such that Properties 1–3 hold, then $T$ can be stored using $(2 + \frac{2}{1 - \tau}) \frac{N}{B} + D$ blocks of external memory and any bottom-up path of length $K$ in $T$ can be traversed in at most \( \left\lceil \frac{K}{DB} \right\rceil + 1 \) I/Os.

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 with a single I/O.

A bottom-up path of length $K$ in $T$ crosses at most $k = \left\lceil \frac{K}{DB} \right\rceil + 1$ layers. This divides the path 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 with a single I/O. Therefore, the traversal of the bottom-up path of length $K$ takes at most $\left\lceil \frac{K}{DB} \right\rceil + 1$ I/Os.

By property 2, all subtrees $T_{i,j}$ together use at most \( (1 + \frac{1}{1 - \tau}) \sum_{i=0}^{r} |T_i| = \left( 1 + \frac{1}{1 - \tau} \right) N \) items. On the other hand, we have to ensure that each subtree $T_{i,j}$ is stored in a single “superblock”. Using a simple block-merging argument, we can guarantee that, on average, each “superblock” is at least half-full. Thus, all subtrees $T_{i,j}$ use at most $\left\lceil (2 - \frac{2}{1 - \tau}) \frac{N}{DB} \right\rceil$ “superblocks”, i.e., $D \left\lceil (2 - \frac{2}{1 - \tau}) \frac{N}{DB} \right\rceil \leq (2 + \frac{2}{1 - \tau}) \frac{N}{B} + D$ blocks. \(\square\)
Figure 3: A tree is cut into layers of height $\tau DB$. (Here, $\tau DB = 2$.) The tree is cut along the dashed lines and the resulting subtrees $T_i$ are shaded.

(a)

(b)

Figure 4: (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 $T_i(V_1)$, $T_i(V_2)$, and $T_i(V_3)$ from left to right.

Figure 5: A given graph $G$ (black vertices and solid lines). White vertices and dotted arrows represent the graph $\bar{G}$ for $G$. Every face $f$ of $G$ is labelled with its corresponding vertex list $F_f$. 