Multiway Simple Cycle Separators and I/O-Efficient Algorithms for Planar Graphs

Lars Arge‡ Freek van Walderveen‡ Norbert Zeh‡

Abstract
We revisit I/O-efficient solutions to a number of fundamental problems on planar graphs: single-source shortest paths, topological sorting, and computing strongly connected components. Existing I/O-efficient solutions to these problems pay for I/O efficiency using excessive computation time in internal memory, thereby completely negating the performance gain achieved by minimizing the number of disk accesses. In this paper, we show how to make these algorithms simultaneously efficient in internal and external memory so they achieve I/O complexity $O(sort(N))$ and take $O(N \log N)$ time in internal memory, where $sort(N)$ is the number of I/Os needed to sort $N$ items in external memory. The key, and the main technical contribution of this paper, is a multiway version of Miller’s simple cycle separator theorem. We show how to compute these separators in linear time in internal memory, and using $O(sort(N))$ I/Os and $O(N \log N)$ (internal-memory computation) time in external memory.

1 Introduction
Large data sets beyond the size of a computer’s main memory can be processed efficiently only using I/O-efficient methods, which aim to minimize the number of disk accesses they perform to swap data between main memory and disk. This has been recognized by practitioners, most notably in the database community, and numerous algorithmic results on solving fundamental problems I/O-efficiently have been produced over the last nearly three decades. See Vitter’s survey [17] for an overview of existing results.

Given the big difference between the cost of a disk access and the cost of a main memory access, it is beneficial to improve the I/O efficiency of an algorithm even at the price of increasing its internal-memory computation slightly. As a result, most previous results on I/O-efficient algorithms focus on minimizing the number of disk accesses and mostly ignore the internal-memory computation. But when does the increase in internal-memory computation stop being “slight” and become the main bottleneck?

The main reason for the high cost of disk accesses is latency. Bandwidth to and from disk is not nearly as great a concern. Therefore, virtually all modern I/O systems are organized into blocks of a certain size $B$, and a disk access (I/O operation) transfers one such block to or from disk. The idea is to amortize the latency in addressing a disk location over a large number of transferred data items, thereby bringing the amortized time per item close to the disk’s bandwidth. In this sense, the “ideal” block size $B$ is usually one that ensures that the time it takes to transfer one block of data between memory and disk is within a constant factor of the cost of $B$ computation steps in memory. Thus, if a decrease in I/O complexity is bought at the price of increasing the internal-memory computation by a factor of $B$ or more, the running time of the algorithm does not improve at all in practice!

Motivated by applications in geographic information systems and other areas, a number of results have been obtained on solving fundamental problems on planar graphs I/O-efficiently. Some of these results [6] are obtained fairly directly from contraction-based parallel algorithms for the same problems in the PRAM model [7]. Others, and the ones we focus on in this paper, use planar separators to partition the graph into memory-sized subgraphs with only limited interaction between each other. These subgraphs are then loaded into memory one at a time and processed independently, coupled with a global computation on a compressed version of the graph that deals with the interactions between subgraphs I/O-efficiently. The problems that have been solved I/O-efficiently using this ap-
proach are single-source shortest paths [2, 4], topological sorting [3], and computing strongly connected components [5]. All these problems have linear-time solutions in internal memory [9, 16], which translate directly into algorithms with an I/O bound of $O(N)$, where $N$ is the number of vertices. The algorithms of [2–5] achieve an I/O complexity of $O(\text{sort}(N))$, where sort$(N) = \Theta(\frac{N}{B} \log \frac{M}{B} \frac{N}{B})$ is the number of I/Os needed to sort $N$ elements in external memory, $B$ is the disk block size, and $M$ is the size of the main memory. For all realistic values of $N$, $M$, and $B$, this is substantially less than $N$ I/Os. However, the time these algorithms use in internal memory is $\Omega(BN)$ in the worst case. Given the “ideal” block size defined above, this is the equivalent of $\Omega(N)$ I/Os, that is, the total running time of the algorithm is comparable to that of the linear-time internal-memory algorithms applied naively to large graphs. This negative impact of the internal-memory computation on the total running time of these algorithms has been observed in several implementations of separator-based algorithms. In order to see any speed-up over naïve algorithms, a much smaller block size had to be used than for other algorithms whose internal-memory computation is independent of the block size.

Our results. We present algorithms for computing single-source shortest paths in planar graphs with non-negative edge lengths, topological sorting of planar directed acyclic graphs (DAGs), and computing the strongly connected components of planar graphs using $O(\text{sort}(N))$ I/Os and $O(N \log N)$ internal-memory computation. In contrast to previous algorithms for these problems, which took $O(N \log N + BN)$ time in internal memory, the internal-memory computation time of our algorithms is thus independent of the block size.

The high-level idea in our algorithms is simply to follow the framework of the existing algorithms and use Klein’s multi-source shortest path algorithm for planar graphs [10] to reduce the amount of internal-memory computation to $O(N \log N)$. For this to work, however, we need a separator partition of the graph that guarantees, in addition to the standard properties of planar graph partitions, a constant number of boundary cycles per region in the partition. No existing internal- or external-memory algorithm for computing separator partitions guarantees this. Our main technical contribution therefore is an algorithm for computing a multiway variant of Miller’s simple cycle separators [15] in linear time in internal memory (Section 3) or using $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time in external memory (Section 4). These algorithms do not use RAM-specific operations and therefore also work on a pointer machine. Specifically, given a parameter $\varepsilon$, we show how to partition a graph with weights on its vertices, edges, and faces that sum to 1 and a bound $s$ on the size (number of edges on the boundary) of every face, into $O(1/\varepsilon)$ regions (collections of faces and their boundaries) such that each region has weight at most $\varepsilon$ and $O(1)$ boundary cycles of size $O(\sqrt{BN})$.

The internal-memory algorithm for computing the above separators combines existing techniques for computing separators, such as separation trees [1], simple-cycle separators [15], and recursive partitioning of regions to reduce their boundary size [8], in novel and intricate ways. The external-memory algorithm employs the bootstrapping technique of [14] to obtain an I/O-efficient separator algorithm from the internal-memory algorithm. The basic idea is to apply graph contraction to obtain a graph $G'$ from the original graph $G$ that is a constant factor smaller than $G$, recursively compute a separator partition $P'$ of $G'$, recover a suboptimal but “good enough” separator partition $P''$ of $G$ from $P'$, use $P''$ to perform BFS in $G$, and use the computed BFS tree to obtain an optimal partition $P$ of $G$. In contrast to the separator algorithm of [14] (and similar to Miller’s algorithm [15]), our separator algorithm requires a BFS tree of the face-incidence graph of $G$, which requires the bootstrapping framework to alternate between the current graph and its dual. This introduces a few complications.

Related work. The body of work on computing planar separators is too extensive to review in detail here, so we review only the results needed to put our results in context. Lipton and Tarjan [12] were the first to show that, given an $N$-vertex planar graph $G$, it takes linear time to compute a set $S$ of at most $O(\sqrt{N})$ vertices whose removal partitions $G$ into two subgraphs with at most $2N/3$ vertices and no edges between them. Such a set $S$ is called a vertex separator. With the motivation to speed up shortest path computations on planar graphs, Frederickson [8] extended this result to obtain a multway separator: given a parameter $r$, $G$ can be partitioned into $O(N/r)$ subgraphs of size at most $r$ and with boundary size $O(\sqrt{r})$, where the boundary size of a region is the number of vertices it shares with other regions. Frederickson’s algorithm to compute such a partition takes $O(N \log N)$ time, by applying Lipton and Tarjan’s algorithm recursively. Aleksandrov and Djidjev [1] showed that essentially the same partition can be computed in linear time. Their main technique is the use of a separation tree: given a spanning tree $T$ of $G$, the graph $T^*$ containing all edges dual to non-tree edges of $G$ is a spanning tree of the dual of $G$, and an appropriate partition of $T^*$ induces the desired partition of $G$. Miller [15] extended Lipton and Tarjan’s result in a different direction by showing that, assuming the
maximum face size is \( s \), it is in fact possible to compute a simple cycle \( C \) of length at most \( O(\sqrt{nN}) \) in \( G \) such that at most \( 2N/3 \) vertices lie on either side of the cycle. \( C \) is called a simple cycle separator of \( G \). To the best of our knowledge, no work has been done on multiway simple cycle separators, probably mostly because it is not clear what structural properties to require of such a separator: a single simple cycle cannot partition the graph into more than two regions, and if more than one cycle is required/allowed, what are the structural properties one should ask the separator to have?

Our definition in this paper is a natural one: Miller’s simple cycle separator partitions the graph into two subgraphs with a boundary size equivalent to the one guaranteed by Lipton and Tarjan’s algorithm, and each subgraph has exactly one boundary cycle. Our multiway separator partitions the graph into \( O(N/r) \) subgraphs of size at most \( r \) and boundary size \( O(\sqrt{sr}) \), the equivalent of Frederickson’s partition, and each region has \( O(1) \) boundary cycles. As we show, this is also the right partition to guarantee that we can solve a number of problems on planar graphs efficiently in both internal and external memory at the same time. Previous results relied on Frederickson’s graph partitioning result to compute single-source shortest paths, topological sorting, and strongly connected components using \( O(\text{sort}(N)) \) I/Os but \( O(N \log N + BN) \) time. Maheshwari and Zeh [14] showed how to compute a Frederickson-style graph partition using \( O(\text{sort}(N)) \) I/Os. By guaranteeing a bound on the number of boundary cycles of each region, we can replace the naive implementation of the internal-memory computation of these algorithms with Klein’s multi-source shortest-path algorithm (see Section 5) and thereby reduce the internal-memory computation time to \( O(N \log N) \).

## 2 Preliminaries

In this section, we introduce the notation and terminology used throughout this paper, including a formal definition of multiway simple cycle separators.

**Basic definitions.** For a graph \( G \), we use \( V(G) \) and \( E(G) \) to denote its vertex and edge sets, respectively. If \( G \) is clear from the context, we simply write \( V \) and \( E \). A **planar embedding** \( \mathcal{E}(G) \) of \( G \) is a drawing of \( G \) in the plane such that no two edges intersect, except in their endpoints. \( G \) is **planar** if it has a planar embedding. A planar graph \( G \) equipped with a particular embedding \( \mathcal{E}(G) \) is often referred to as a **plane graph**. We use \( \mathcal{E}(v) \) and \( \mathcal{E}(e) \) to denote the image of vertex \( v \) or edge \( e \) under this embedding, and define \( \mathcal{E}(S) := \bigcup_{x \in S} \mathcal{E}(x) \), for any set \( S \) of vertices and edges of \( G \). The **faces** of a plane graph \( G \) are the connected components of \( \mathbb{R}^2 \setminus (\mathcal{E}(V(G)) \cup \mathcal{E}(E(G))) \). We denote the set of faces of \( G \) by \( F(G) \). The **boundary** \( \partial f \) of a face \( f \) is the set of vertices and edges contained in \( f \)’s closure. The **size** of a face is the number of edges on its boundary. A **region** is defined as a collection \( R \) of faces and their boundaries, excluding all vertices and edges that are on the boundary of at least one face not in \( R \). The latter constitute the boundary \( \partial R \) of \( R \). We use \( \bar{R} := R \cup \partial R \) to denote the closure of a region. We call a region \( R \) **connected** if it is connected in the topological sense. Given a weight function \( w \) assigning weights to the vertices, edges, and faces of a plane graph \( G \), the weight of a region or subgraph of \( G \) is the sum of the weights of the vertices, edges, and faces it contains. Throughout this paper, when we refer to a region \( R \) as a subgraph of \( G \), we mean the subgraph of \( G \) embedded in \( R \)’s closure \( \bar{R} \), but we exclude the weight of \( R \)’s boundary from \( R \)’s weight. For a subgraph \( G' \subseteq G \), the restriction of the embedding \( \mathcal{E}(G) \) to \( G' \) induces an embedding \( \mathcal{E}(G') \) of \( G' \). We define the weight of a face \( f \) of \( G' \) to be the weight of the vertices, edges, and faces of \( G \) embedded inside \( f \) by \( \mathcal{E}(G) \). A simple cycle \( C \) is a collection of vertices and edges of \( G \) such that \( \mathcal{E}(C) \) is homeomorphic to \( S^1 \) (a closed loop). \( \mathbb{R}^2 \setminus \mathcal{E}(C) \) has two connected components, one bounded and one unbounded. We call the former the **interior** of \( C \) and the latter the **exterior** of \( C \). The **region enclosed by** a cycle \( C \) is the collection of vertices, edges, and faces enclosed in \( C \’s \) interior. The **weight enclosed by** \( C \) is the weight of the region enclosed by \( C \). The boundary \( \partial R \) of a region \( R \) is the union of a set of simple cycles, which we call **boundary cycles** of \( R \). For exactly one of these cycles \( C \subseteq \partial R \), \( R \) is in \( C \’s \) interior. We call this cycle the **outer boundary** of \( R \) and the other boundary cycles of \( R \) **holes**.

**The dual graph and related structures.** The dual \( G^* \) of a plane graph \( G \) contains one vertex \( f^* \) for each face \( f \) of \( G \). For every edge \( e \) of \( G \) on the boundary of faces \( f_1 \) and \( f_2 \), \( G^* \) contains an edge \( e^* \) with endpoints \( f_1^* \) and \( f_2^* \). We call \( f^* \) the dual of face \( f \) and \( e^* \) the dual of edge \( e \). \( G^* \) is planar, and an embedding of \( G^* \) is obtained naturally from the given embedding of \( G \). Given a spanning tree \( T \) of \( G \), the graph \( T^* \) containing every edge of \( G^* \) whose dual does not belong to \( T \) is a spanning tree of \( G^* \). We call \( T^* \) the dual of \( T \). Every non-tree edge \( e \) of \( G \) w.r.t. \( T \) defines a **fundamental cycle** \( fc(e) \), which consists of \( e \) and the path between \( e \’s \) endpoints in \( T \). If \( T^* \) is rooted in the vertex representing \( G \’s \) outer face, then the descendant nodes of \( e \’s \) dual edge \( e^* \) are the dual vertices of the faces in the region enclosed by \( fc(e) \). The **face incidence graph** \( G_t \) of \( G \) is closely related to the dual. It has one vertex \( f^* \) per face \( f \) of \( G \) and an edge \( f_1 f_2 \) if and only if the boundaries of faces \( f_1 \) and \( f_2 \) are non-disjoint. The
vertex-on-face graph $G_{f^*}$ contains every vertex of $G$ and one vertex $f^*$ per face $f$ of $G$ and contains an edge $vf^*$ if vertex $v$ is on the boundary of face $f$.

**Simple cycle $\varepsilon$-separators.** Given a parameter $0 < \varepsilon < 1$ and a function that assigns weights to the vertices, edges, and faces of $G$ such that the total weight of $G$ is 1 and any face has weight at most $\varepsilon$, a *simple cycle $\varepsilon$-separator* of a 2-edge-connected plane graph $G$ is a 2-edge-connected subgraph $S$ of $G$ together with a grouping of the faces of $G$ into $O(1/\varepsilon)$ regions with the following properties:

(i) All faces of $G$ embedded inside the same face of $S$ belong to the same region.

(ii) The weight of each region is $O(\varepsilon)$.

(iii) The boundary of each region has size $O(\sqrt{\varepsilon sN})$, where $s$ is the size of the largest face of $G$.

(iv) Every region is either connected or shares boundary edges with at most two other regions, which must be connected.

(v) The boundary of each connected portion of each region consists of $O(1)$ simple cycles.

We use $G - S$ to denote the set of connected regions into which $S$ divides $G$. A *boundary set of* $S$ is a maximal subset $V'$ of the vertex set of $S$ such that all vertices in $V'$ are on the boundaries of the same regions. We use the term *multiway (simple) cycle separator* to refer to a simple cycle $\varepsilon$-separator whenever $\varepsilon$ is clear from the context or we do not care to specify it.

### 3 Computing Multiway Simple Cycle Separators in Linear Time

This section presents our main result, stated in the following theorem.

**Theorem 3.1.** A simple cycle $\varepsilon$-separator can be computed in linear time.

Our algorithm to compute such a separator follows the standard framework used in almost all planar separator algorithms, starting with Lipton and Tarjan’s seminal result, but it has a few non-trivial twists. We proceed in three steps.

Step 1 computes a BFS tree and chooses appropriate levels of the tree as the first part $S_1$ of the separator. Just like Miller’s algorithm, we use BFS in the face-incidence graph of $G$ instead of in $G$ and then choose $S_1$ to consist of the boundaries between faces on appropriate pairs of consecutive levels. $S_1$ partitions $G$ into two types of regions: *light* ones of weight at most $\varepsilon$ and containing at most $\varepsilon N$ vertices, and *heavy* ones of weight greater than $\varepsilon$ or with more than $\varepsilon N$ vertices but of small diameter. There are two complications here. As in Miller’s algorithm, we cannot actually guarantee low diameter of the heavy regions. We can, however, identify a low-diameter subgraph $R'$ of each heavy region $R$, along with a low-diameter spanning tree $T'$ of $R'$, such that each face of $R'$ has weight at most $\varepsilon$. A multiway simple cycle separator of $R'$ is also a multiway simple cycle separator of $R$, so Step 2 focuses on partitioning each such subgraph $R'$ into subregions of weight at most $\varepsilon$ and with at most $\varepsilon^2 N$ vertices. The second complication is that $S_1$ may consist of too many simple cycles. We remove simple cycles between adjacent light regions as appropriate to reduce the number of light regions and simple cycles to $O(1/\varepsilon)$ while keeping the resulting regions light.

Step 2 partitions each heavy region $R$ into light subregions and is divided into two substeps. Step 2.1 chooses a number of fundamental cycles of the spanning tree $T'$ of $R'$ to partition $R$. We cannot yet guarantee that the resulting subregions are light because this may require too many fundamental cycles. What Step 2.1 produces is a partition of the dual spanning tree $T''$ of $T'$ into $O(1/\varepsilon)$ subtrees $T_1^*, T_2^*, \ldots, T_k^*$ such that each subtree $T_{i,j}^*$ of such a tree $T_i^*$ rooted in a child of the root of $T_i^*$ spans a subregion $R_{i,j}^*$ of $R'$ of weight at most $\varepsilon$ and with at most $\varepsilon^2 N$ vertices. Since the root of $T_{i,j}^*$ is adjacent to the root of $T_i^*$, each such subregion $R_{i,j}^*$ shares at least one edge with the face $r$ corresponding to the root of $T_i^*$. Step 2.2 computes a “nesting forest”, which captures the attachment of these subregions to $f$ and then applies a procedure very similar to Step 2.1 to partition this nesting forest into subtrees of weight at most $\varepsilon$. Once again, we cannot guarantee that this produces only $O(1/\varepsilon)$ regions, but this time the regions can be grouped naturally to reduce their number to $O(1/\varepsilon)$. This is part of what Step 3 does.

The procedure outlined so far produces a partition into the right number of regions of the right size, with the right weight, and with the right total boundary size. The boundary of an individual region, however, may be too large and may consist of too many simple cycles. Step 3 therefore subdivides regions further to ensure each individual region has a small boundary consisting of only $O(1)$ simple cycles. We do this in a manner very much like the final step in Frederickson’s algorithm, but using Miller’s simple cycle separator algorithm instead of Lipton and Tarjan’s algorithm. The final step is...
to group connected regions in a manner very similar to Frederickson’s algorithm to reduce the number of regions to $O(1/\varepsilon)$.

3.1 Step 1: Partition into small or low-diameter regions. In this section, we compute the first part $S_1$ of the separator, which partitions the graph into few regions that are either light or of low weighted diameter, as summarized in the following lemma. Given the current separator, we define the weighted diameter of a graph to be the maximum number of non-separator edges of any path in the graph.

**Lemma 3.1.** Given a parameter $\varepsilon$, $0 < \varepsilon < 1$, and a 2-edge-connected plane graph $G$ with maximum face size $s$ and maximum face weight at most $\varepsilon$, it takes linear time to compute (i) a 2-edge-connected subgraph $G'$ of $G$ whose faces have weight at most $\varepsilon$ and boundary size at most $\max(s, \sqrt{\varepsilon N})$ each and contain at most $\varepsilon N$ vertices of $G$ each, and (ii) a set $S_1 \subseteq E(G')$ consisting of $O(1/\varepsilon)$ edge-disjoint simple cycles that partition $G'$ into $O(1/\varepsilon)$ connected regions, each either of weight at most $\varepsilon$ and containing at most $\varepsilon N$ vertices or of weighted diameter at most $O(\sqrt{\varepsilon N})$. The total length of the cycles in $S_1$ is $O(\sqrt{sN}/\varepsilon)$.

As already explained, we implement this step in two substeps. First we find a superset $S_{1,1} \supseteq S_1$ of size $O(\sqrt{sN}/\varepsilon)$ that splits $G$ into regions of weighted diameter $O(\sqrt{\varepsilon N})$. Then we remove all but $O(1/\varepsilon)$ cycles from $S_{1,1}$, which may increase the weighted diameter of some regions beyond $O(\sqrt{\varepsilon N})$. For light regions, this is no problem. For heavy regions, we merge some of their faces to obtain $G'$ and ensure that the portion of $G'$ contained in each region has weighted diameter $O(\sqrt{\varepsilon N})$.

3.1.1 Step 1.1: Partition into low-diameter regions. To compute $S_{1,1}$, we construct a breadth-first search (BFS) tree $T_1$ of the face incidence graph $G_f$ of $G$, starting from $G$’s outer face. To compute this BFS tree in linear time (and I/O-efficiently in Section 4), we use the vertex-on-face graph $G_{vf}$ of $G$, which is planar and has linear size, as a proxy for $G_f$. We construct $G_{vf}$ from $G$ and compute a BFS tree $T_1$ of $G_{vf}$ rooted in the vertex of $G_{vf}$ corresponding to $G$’s outer face. $T_1$ is obtained from $T_1$ by making every face vertex a child of its grandparent in $T_1$ and then deleting all vertices of $G$ from the resulting tree. Let $L_i$ be the nodes of $T_1$ at distance $i$ from the root, and let $F_i$ be the set of vertices and edges of $G$ that are each incident to faces $f_1$ and $f_2$ with $f'_1 \in L_i$ and $f'_2 \in L_{i+1}$. $F_i$ forms a set of edge-disjoint simple cycles that separate the faces in $L_0, L_1, \ldots, L_i$ from the faces in $L_{i+1}, L_{i+2}, \ldots$ (see Figure 1). We call $F_i$ the level-$i$ frontier.

Each vertex or edge belongs to at most one frontier, so, by the pigeon hole principle, we can choose $0 \leq k < \sqrt{\varepsilon N}/s$ such that there are at most $N/\sqrt{\varepsilon N}$ edges in $S_{1,1} := F_k \cup F_{k+\varepsilon N} \cup F_{k+2\varepsilon N} \cup F_{k+3\varepsilon N} \cup \cdots$. These frontiers can clearly also be selected in linear time. It is not hard to prove that each region of $G - S_{1,1}$ has weighted diameter at most $\sqrt{\varepsilon N}$. We do not prove this here because what we really require is that every region of $G' - S_1$ has this diameter. We construct $G'$ and $S_1$ in the next step.

3.1.2 Step 1.2: Reducing the number of boundary cycles. The reduction of the number of boundary cycles is achieved in two steps. We use a nesting tree $T$ of the regions of $G - S_{1,1}$ (faces of $S_{1,1}$). This tree contains one node $R'$ per region $R$ of $G - S_{1,1}$. A node $R'_{i+1}$ is the parent of a node $R'_{i}$ if $R_{i}$ is contained in a hole of $R_{i+1}$ and there is no other region that encloses $R_{i}$ in one of its holes and is itself enclosed in a hole of $R_{i}$. In the first step, we compute $T$ and then modify it by merging leaves into their parents to ensure $T$ has only heavy leaves. This step also modifies $G$ to obtain the subgraph $G' \subseteq G$ whose regions have low weighted diameter. Since there can be only $O(1/\varepsilon)$ heavy regions, this ensures that $T$ has $O(1/\varepsilon)$ leaves and, hence, $O(1/\varepsilon)$ regions corresponding to internal nodes with at least two children. The second step merges pairs of adjacent light regions corresponding to internal nodes with only one child, in order to reduce their number to $O(1/\varepsilon)$ while keeping their weight and size bounded by $\varepsilon$ and $\varepsilon N$, respectively.

![Figure 1: Constructing $S_{1,1}$ using frontiers in Step 1.1.](image)
Computing a nesting tree with few leaves. To compute the nesting tree, we compute the dual $G^*$ of $G$ and remove all edges dual to edges in $S_{1,1}$. The regions of $G - S_{1,1}$ correspond to the connected components of the resulting subgraph of $G^*$, and we create one node in $T$ for each region. Then we add one edge to $T$ for every pair of regions that share a boundary edge. $T$ is a tree because $S_{1,1}$ is a set of edge-disjoint simple cycles. We root $T$ in the node representing the region containing the outer face of $G$. Next we assign weights to the nodes of $T$ that capture the weights of their corresponding regions. In particular, the weight of every face $f$ of $G$ is assigned to the node $R^*$ of $T$ that represents the region $R$ containing $f$, that is, the connected component of $G^* - S_{1,1}$ that contains $f^*$. For every vertex or edge $x$ of $G$, we choose an arbitrary incident face $f$ and add $x$’s weight to the node of $T$ that received $f$’s weight. We also define a region size of each node $R^*$ of $T$, which is the number of vertices of $G$ that assigned their weights to $R^*$. Note that this ensures that the total weight and the total region size of the nodes in any subtree $T' \subseteq T$ are upper bounds on the weight and the number of vertices in the region of $G$ represented by $T'$. Every edge of $T$ represents a simple cycle in $S_{1,1}$ that separates the regions corresponding to the subtrees on either side of the edge. We call an edge of $T$ heavy if the subtree below it has weight greater than $\varepsilon$ or region size greater than $\varepsilon N$. Otherwise the edge is light. First we traverse $T$ and identify all light edges whose parent edges are heavy. We call these edges critical edges and their corresponding cycles in $S_{1,1}$ critical cycles (see Figure 2). We remove all critical edges and their descendant edges from $T$, and we remove the cycles corresponding to the removed edges from $S_{1,1}$, thus obtaining a reduced separator $S_{1,2} \subseteq S_{1,1}$. This ensures that every leaf of $T$ represents a heavy region of $G - S_{1,2}$, but the weighted diameters of heavy regions may be large. Next we show how to reduce the diameter of heavy regions to $O(\sqrt{\varepsilon N})$ again while keeping the face weight, the number of vertices of $G$ embedded inside each face, and the face size bounded by $\varepsilon$, $\varepsilon N$, and $\max(s, \sqrt{\varepsilon N})$, respectively.

**Constructing $G'$ from $G$.** Consider a heavy region $R$ of $G - S_{1,2}$, and let $R''$ be the subgraph of $R$ obtained by removing all vertices and edges enclosed by a critical cycle in $R$. Each face of $R''$ is either a face of $G$ or a critical cycle. Let $C$ be such a critical cycle, and let $R_C$ be the region of $R$ enclosed by $C$. Since $R_C$ has weight at most $\varepsilon$ and contains at most $\varepsilon N$ vertices of $G$, we can arbitrarily merge faces in $R_C$ without creating a face of weight greater than $\varepsilon$ or containing more than $\varepsilon N$ vertices of $G$. Also recall that $C$ is part of a frontier $F_i$. We consider the portions of the frontiers $F_i, F_{i+1}, \ldots, F_{i+\sqrt{\varepsilon N}/\varepsilon - 1}$ that belong to $R_C$. By the pigeon hole principle and because $R_C$ has at most $\varepsilon N$ vertices, there exists one frontier $F_j$ among them that contributes at most $\sqrt{\varepsilon N}$ vertices to $R_C$. We remove all vertices and edges of $R_C$ enclosed by cycles of $F_j$ contained in $R_C$ to obtain a subgraph $R'_C \subseteq R_C$. We call the resulting faces of $R'_C$ pruning faces. By performing this transformation on each portion $R_C$ of $R$ enclosed by a critical cycle $C$ in $R$, we obtain a pruned subgraph $R' \subseteq R$. We obtain $G'$ by replacing each heavy region $R$ of $G - S_{1,2}$ with its corresponding pruned subgraph (see Figure 3).

**Lemma 3.2.** For every heavy region $R$ of $G - S_{1,2}$, the
pruned subgraph $R' \subseteq R$ has weighted diameter at most $4\sqrt{\varepsilon sN}$. Every face of $R'$ has boundary size at most $\max(s, \sqrt{\varepsilon sN})$ and weight at most $\varepsilon$, and contains at most $\varepsilon N$ vertices of $G$.

Proof. We already argued that every face of $R'$ has weight at most $\varepsilon$ and contains at most $\varepsilon N$ vertices of $G$ because either it is a face of $G$ or it is enclosed in a critical cycle. If it is a face of $G$, its size is at most $s$.

Otherwise it is a pruning face and, thus, has size at most $\sqrt{\varepsilon sN}$. To bound the diameter of $R'$, it suffices to show that every vertex of $R'$ has a path of length at most $2\sqrt{\varepsilon sN}$ to the outer boundary of $R'$ because the outer boundary of $R'$ does not count towards the weighted diameter (it is part of $S_{1,2}$). Let the outer boundary of $R'$ be part of a frontier $F_i$. Every vertex of $R'$ is on the boundary of a face $f$ of $R'$ such that $f^*$ belongs to some level $L_j$ of $T_i$ with $j - i \leq 2\sqrt{\varepsilon/N}/s$. Thus, if $(f^*)^*$ is $f^*$'s ancestor in $L_i$, we have $\text{dist}_{T_i}(f^*, (f^*)^*) \leq 2\sqrt{\varepsilon N}/s$ and $\text{dist}_{R'}(u, v) \leq s \cdot (j - i) \leq 2\sqrt{\varepsilon sN}$, for every vertex $u$ on $f$ and every vertex $v$ on $f'$, because the dual of every node on the path from $f^*$ to $(f^*)^*$ in $T_i$ is a face of $G$ and, thus, has size at most $s$. One of the vertices of $f'$ must belong to $F_i$ and, hence, to the outer boundary of $R'$. Thus, every vertex of $f$ has distance at most $2\sqrt{\varepsilon sN}$ from the outer boundary of $R'$. \hfill \Box

Even though we will further reduce $S_{1,2}$ to obtain a separator $S_1 \subseteq S_{1,2}$, this reduction will only affect light regions. Thus, Lemma 3.2 shows that the heavy regions of $G' - S_1$ have the required structure. For Step 2 it is important that we can find a particular spanning tree of each heavy region of $G' - S_1$, as stated in the following lemma.

**Lemma 3.3.** Every heavy region $R'$ of $G' - S_{1,2}$ has a spanning tree $T'$ of weighted diameter at most $4\sqrt{\varepsilon sN}$ and such that every boundary cycle of $R'$ contributes all but one of its edges to $T'$.

Proof. As shown in the proof of Lemma 3.2, every vertex of $R'$ has distance at most $2\sqrt{\varepsilon sN}$ from the outer boundary of $R'$ (or from the frontier $F_0$ if $R'$ is the region including the outer face of $G'$). Thus, if we assign length 0 to each separator edge and length 1 to each nonseparator edge, a shortest path tree of $R'$ rooted in a vertex on this outer boundary (or on $F_0$) has weighted diameter at most $4\sqrt{\varepsilon sN}$. Such a tree $T''$ can be computed in linear time using a straightforward adaptation of standard BFS because the edges have weight 0 or 1.

All the edges of $T''$ are directed away from the root. For every boundary cycle $C$ of $R'$, we remove all edges with head in $C$ from $T''$, except one edge whose tail does not belong to $C$ and whose head has the smallest distance from the root of $T''$ among these edges. If $C$ is the outer boundary of $R'$, we remove all edges with head in $C$. Then we add all edges of $C$, except one, to $T''$.

It is easily verified that the resulting graph $T'$ is a tree, has weighted diameter no greater than that of $T''$, and includes all edges of each boundary cycle, except one. \hfill $\Box$

**Merging light regions.** The final step is to merge light regions to reduce their number to $O(1/\varepsilon)$. We consider maximal paths in the nesting tree $T$ such that all nodes on such a path $P$ represent light regions and have only one child each. We traverse each such path from the end closer to $T$'s root and find the longest prefix $P'$ such that the total weight of the nodes in $P'$ is no more than $\varepsilon$ and the total vertex count of these nodes is no more than $\varepsilon N$. We remove all cycles corresponding to edges in $P'$ from $S_{1,2}$ and then proceed to the first node in $P$ after $P'$. Starting with this node, we once again find a prefix of the remainder of $P$ as above and remove the cycles corresponding to its edges from $S_{1,2}$.

We continue in this fashion until we have consumed all of $P$. The cycles not removed from $S_{1,2}$ once we are done processing all such paths $P$ in $T$ constitute the final separator $S_1$ produced by Step 1 of our algorithm.

Clearly, this reduction of $S_{1,2}$ to $S_1$ does not affect heavy regions, and we explicitly ensure that the regions produced by merging light regions are themselves light. Thus, all regions of $G' - S_1$ have the properties stated in Lemma 3.1. To bound the number of regions, consider the tree $T_1$ obtained from $T$ by replacing each subpath $P'$ in the above procedure with a single node. There are at most $4/\varepsilon$ nodes in $T_1$ that are of weight greater than $\varepsilon/2$ or have vertex count greater than $\varepsilon N/2$. Since every leaf of $T_1$ is heavy and the number of internal nodes with at least two children is bounded by the number of leaves, there are at least $8/\varepsilon$ nodes of weight greater than $\varepsilon/2$, vertex count greater than $\varepsilon N/2$ or with at least two children. We call these nodes semi-heavy and all other nodes semi-light. Every semi-light node $v$ must be the parent of a semi-heavy node because otherwise the processing of paths of degree-1 nodes above would have merged $v$ and its child. Thus, there are at most $8/\varepsilon$ semi-light nodes, and the total number of regions of $G' - S_1$ is at most $16/\varepsilon$.

**3.2 Step 2: Splitting heavy regions.** The second step of our algorithm produces a separator $S_2$ that partitions $G$ into regions of size at most $\varepsilon N$ and weight at most $\varepsilon$, as stated in the following lemma.

**Lemma 3.4.** Given a parameter $\varepsilon$, $0 < \varepsilon < 1$, and a 2-edge-connected plane graph $G$ with maximum face size...
s and maximum face weight at most $\varepsilon$, it takes linear
time to compute a subgraph $S_2 \subseteq G$ such that every
region of $G - S_2$ has at most $\varepsilon N$ vertices and weight at
most $\varepsilon$, $S_2$ has $O(\sqrt{\varepsilon N})$ edges, and every connected
component of $S_2$ is 2-edge-connected. The total number of
holes of the regions of $G - S_2$ is $O(1/\varepsilon)$.

We obtain $S_2$ by augmenting the separator $S_1$ produced by Step 1. It is sufficient to compute a separator $S_2$ as in Lemma 3.4 for $G'$ because $G' \subseteq G$ and every face of $G'$ has weight equal to the total weight of the portion of $G$ embedded in it. The light regions of $G' - S_1$ do not need to be partitioned further. We partition every heavy region $R'$ of $G' - S_1$ into light subregions in two steps. In Step 2.1, we use the separation tree technique by Aleksandrov and Djidjev [1] to partition $R'$ into $O(1/\varepsilon)$ parts composed of subregions of weight at most $\varepsilon$ and size at most $\varepsilon N$ that each share at least one edge with a “root face” $r$. This partition is obtained by adding $O(1/\varepsilon)$ fundamental cycles w.r.t. the low-diameter spanning tree produced in Step 1 to the separator. In Step 2.2, we refine the partition further to obtain a partition into small regions. This step uses a nesting forest that captures how certain subregions combined with $f$ enclose other subregions.

3.2.1 Step 2.1: Separation tree decomposition.
Consider a heavy region $R$ of $G - S_1$ and its corresponding heavy region $R'$ of $G' - S_1$. Recall that $R'$ has a low-diameter spanning tree $T'$. Let $T^*$ be the dual spanning tree of $T'$. Note that $T^*$ also includes nodes corresponding to faces bounded by the boundary cycles of $R'$ and which as such do not belong to $R'$. We assume that $T^*$ is rooted in the node corresponding to the outer boundary of $R'$. Slightly extending the separation tree technique of Aleksandrov and Djidjev [1], we assign the weights of the vertices, edges, and faces of $R'$ to the nodes of $T^*$ as follows: The weight of every face $f$ is assigned to $f^*$. For every vertex or edge $x$, we choose a face $f$ that has $x$ on its boundary and add $x$’s weight to the weight of $f^*$. This guarantees that the weight of any subtree $T''$ of $T^*$ is an upper bound on the weight of the region consisting of the faces of $R'$ dual to the vertices in $T''$. As in Step 1, we assign a vertex count $v(f^*)$ to each node $f^*$ of $T^*$, which is the number of vertices of $G$ (not $G'$) that assigned their weights to $f^*$ (either individually as a vertex of $R'$ on $f$’s boundary or through inclusion in $f$’s face weight if the vertex lies in $f$’s interior). Now we identify a set $C$ of at most $(w(T^*) + v(T^*)/N)/\varepsilon - 1$ edges of $T^*$ whose removal partitions $T^*$ into at most $(w(T^*) + v(T^*)/N)/\varepsilon = (w(R) + |R'|/N)/\varepsilon$ subtrees $T_1, T_2, \ldots, T_k$ such that, for all $1 \leq i \leq k$, any subtree of $T^*_i$ that does not include the root of $T^*_i$ has weight at most $\varepsilon$ and total vertex count at most $\varepsilon N$. Every edge $e \in C$ induces a fundamental cycle $ec(e)$ in $T'$, and the set of fundamental cycles $fc(e) = \{fe(e) \mid e \in C\}$ partitions $R'$ into connected subregions $R'_1, R'_2, \ldots, R'_{k'}$ corresponding to the trees $T'_1, T'_2, \ldots, T'_{k'}$. Since the total weight of $G$ (and hence of $G'$) is 1 and $G$ has $N$ vertices, the total number of fundamental cycles introduced in all heavy regions of $G' - S_1$ is $O(1/\varepsilon)$. By Lemma 3.3, adding these fundamental cycles to $S_1$ therefore produces a separator $S_{2.1} \supseteq S_1$ of size $|S_1| + O(\sqrt{\varepsilon s N}/\varepsilon) = O(\sqrt{\varepsilon N}/\varepsilon)$.

To compute the set $C$, we process $T$’s bottom-up (e.g., using a postorder traversal) and compute for each node $f^*$ of $T^*$ the total weight and vertex count of all its descendant nodes. When visiting a non-root node $f^*$, if the total weight of the descendants of $f^*$ exceeds $\varepsilon$ or their total vertex count exceeds $\varepsilon N$, we add the parent edge of $f^*$ to $C$. This removes the subtree below $f^*$ from $T^*$, and the nodes in this subtree do not contribute to the total weight or vertex count of any ancestor of $f^*$. This is essentially the same approach chosen by Aleksandrov and Djidjev, but, as we already observed, it does not guarantee a useful bound on the weight or size of any region yet because $T^*$ does not necessarily have constant degree. The bound on the number of edges added to $C$ is easily seen to be as claimed because every edge added to $C$ can be charged to a subtree of weight greater than $\varepsilon$ or vertex count greater than $\varepsilon N$ that is pruned from $T^*$ by adding this edge to $C$.

To identify the edges in $fe(C) \setminus S_1$ that need to be added to $S_{2.1}$, we label every node in $T^*$ with the index $i$ of the subtree $T^*_i$ it belongs to. An edge $e$ needs to be added to $S_{2.1}$ if it is not already in $S_1$ and the two endpoints of $e^*$ in $(R^*)_i$ have different labels.

For each subregion $R'_i$ of $R'$ corresponding to a tree $T^*_i$, the root face is the one corresponding to the root $r^*$ of $T^*_i$. For each child $f^*$ of $r^*$, the portion of $R'_i$ corresponding to the descendants of $f^*$ in $T^*_i$ by definition has weight at most $\varepsilon$ and size at most $\varepsilon N$ and is connected because its dual is spanned by a subtree of $T^*_i$ and all nodes in $T^*$ not corresponding to faces of $R'$ must be leaves, by Lemma 3.3. $r^*$ is a face of $G'$ and, thus, also has weight at most $\varepsilon$ and contains at most $\varepsilon N$ vertices of $G$. Thus, $S_{2.1}$ defines the partition we set out to compute in Step 2.1.

3.2.2 Step 2.2: Nesting forest decomposition.
Now consider the partition of $G'$ defined by $S_{2.1}$, and let $R'_i$ be one of its regions. If $R'_i$ is light, there is no
need to partition it further, so assume \( R_i' \) is heavy. In this case, it is one of the subregions produced from a region \( R' \) of \( G' - S_1 \) in Step 2.1. Let \( T_i^* \) be the subtree of \( T' \) corresponding to \( R_i' \), let \( r^* \) be its root, and let \( T' \) be the low-diameter spanning tree of \( R' \) computed in Step 1. Now let \( R_i'' \) be the subgraph of \( R_i' \) obtained by replacing the faces dual to the nodes in each subtree of \( T_i^* \) rooted in a child of \( r^* \) with a single face (see Figure 4(a)). By letting the weight and vertex count of each face of \( R_i'' \) be the weight and vertex count of its corresponding portion of \( T_i^* \), we ensure that the weight and vertex count of each such face is an upper bound on the weight of and the number of vertices in the portion of \( G \) embedded inside this face, and no face of \( R_i'' \) has weight greater than \( \varepsilon \) nor vertex count greater than \( \varepsilon N \). Thus, a separator partitioning \( R_i'' \) into subregions of weight at most \( \varepsilon \) and vertex count at most \( \varepsilon N \) also partitions the portion of \( G \) embedded inside \( R_i' \) into regions with these properties.

To compute such a partition of \( R_i'' \) that does not add too many edges to the separator, we construct a nesting forest \( T_i \) of \( R_i'' \). The vertex set of \( T_i \) includes one vertex \( f^* \) per face \( f \neq r \) of \( R_i'' \). To construct the edge set of \( T_i \), we number the edges on the boundary of the root face \( r \) clockwise around \( r \), starting with the edge dual to the parent edge of \( r^* \) in \( T_i^* \). Every face \( f \neq r \) of \( R_i'' \) shares at least one edge with \( r \): the dual edge of the edge connecting its corresponding subtree of \( T_i^* \) to \( r^* \). By considering the first and last edge in the ordering of edges around \( r \) shared between \( r \) and \( f \), we obtain an edge interval \( I_f \) associated with \( f \). Node \( f_1^* \) is the parent of node \( f_2^* \) in \( T_i \) if \( I_{f_2} \subseteq I_{f_1} \), and the boundaries of \( f_1 \) and \( f_2 \) share an edge. This is illustrated in Figure 4(a). We assign weights and vertex counts to the nodes of \( T_i \) in a manner analogous to Step 2.1 and call a component of \( T_i \) heavy if its weight is greater than \( \varepsilon \) or its vertex count is greater than \( \varepsilon N \), and light otherwise. Now observe that the components of \( T_i \) correspond to intervals along the boundary of \( r \) and, thus, form a sequence \( T_{i,1}, T_{i,2}, \ldots, T_{i,k} \). We obtain the desired partition of \( R_i'' \) from this sequence in four steps.

First, we add all edges on the boundary of its root face \( r \) and not already in \( S_{2,1} \) to \( S_{2,1} \).

Second, we consider each heavy component \( T_{i,j} \) in turn. There exists a unique boundary vertex \( x \) of \( r \) that belongs to the boundaries of the regions of \( R''_j \) corresponding to \( T_{i,j} \) and \( T_{i,j-1} \), and there exists a unique boundary vertex \( y \) of \( r \) that belongs to the boundaries of the regions of \( R''_j \) corresponding to \( T_{i,j} \) and \( T_{i,j+1} \). By adding shortest paths in \( T' \) from \( x \) and \( y \) to the outer boundary of \( R_i'' \) to \( S_{2,1} \), we separate the region of \( R_i'' \) corresponding to \( T_{i,j} \) from the regions corresponding to all other components of \( T_i \) (see Figure 4(b)). Moreover, once we have added such paths to \( S_{2,1} \), for every heavy component of \( T_i \), if two light components \( T_{i,h} \) and \( T_{i,j} \) are separated by a heavy component in the sequence of components of \( T_i \), their corresponding regions of \( R_i'' \) are also separated from each other by \( S_{2,1} \).

Third, we consider maximal subsequences of the sequence of components of \( T_i \) that do not include any heavy component. We greedily partition each such subsequence \( T_{i,h}, T_{i,h+1}, \ldots, T_{i,\ell} \) into maximal subsequences of total weight at most \( \varepsilon \) and total vertex count at most \( \varepsilon N \). This is similar to the merging of light regions in Step 1.2. For every pair of consecutive subsequences in this partition of \( T_{i,h}, T_{i,h+1}, \ldots, T_{i,\ell} \), let \( T_{i,j} \) and \( T_{i,j+1} \) be the last component of the first subsequence and the first component of the second subsequence, respectively. We separate \( T_{i,j} \) from \( T_{i,j+1} \) using a path in \( T' \) analogous to the step separating heavy components from the rest of \( T_i \).

After these three steps, we obtain a partition of \( R_i'' \) into three types of regions: the root face \( r \); heavy regions, each corresponding to a heavy component \( T_{i,j} \) of \( T_i \); and light regions corresponding to subsequences of light components of \( T_i \). The fourth step partitions the heavy regions into light subregions. Consider such
a region corresponding to a heavy component $T_{i,j}$ of $T_i$. We compute a set $C_{i,j}$ of edges of $T_{i,j}$ whose removal partitions $T_{i,j}$ into subtrees of weight at most $\varepsilon$ and vertex count at most $\varepsilon N$. We do this similarly to the computation of the edge set $C$ in Step 2.1, with the following modification: when we encounter a node $f^*$ whose subtree of $T_{i,j}$ has weight greater than $\varepsilon$ or vertex count greater than $\varepsilon N$, we do not only add its parent edge to $C_{i,j}$ but also all edges connecting $f^*$ to its children. Now we label every node $f^*$ of $T_{i,j}$ with the component of $T_{i,j} - C_{i,j}$ it belongs to, and once again add every edge $e$ of the region corresponding to $T_{i,j}$ to $S_{2,1}$ that does not already belong to $S_{2,1}$ and such that the endpoints of its dual edge belong to different components of $T_{i,j} - C_{i,j}$.

We obtain the final separator $S_2$ produced by Step 2 by augmenting $S_{2,1}$ in this fashion for every heavy region $R'_i$ of $G' - S_{2,1}$. Every region in the resulting partition of $R'_i$ is either the root face $r$, which is light, a region produced by grouping regions corresponding to light components of $T_i$ or a light region produced by partitioning the region corresponding to a heavy component of $T_i$. Thus, $S_2$ partitions $G$ into regions of weight at most $\varepsilon$ and vertex count at most $\varepsilon N$. Since each region of $G - S_2$ is a collection of faces of $G$, $S_2$ is obviously 2-edge-connected. It remains to prove that $S_2$ has the size claimed in Lemma 3.4 and that the total number of holes in the regions of $G - S_2$ is at most $O(1/\varepsilon)$.

**Lemma 3.5.** $|S_2| = O(\sqrt{sN}/\varepsilon)$ and the total number of holes of the regions of $G - S_2$ is $O(1/\varepsilon)$.

**Proof.** Since $|S_{2,1}| = O(\sqrt{sN}/\varepsilon)$, it suffices to prove that each of the four steps in partitioning heavy regions of $G' - S_{2,1}$ adds at most $O(\sqrt{sN}/\varepsilon)$ edges to $S_2$.

At most one heavy region of $G' - S_{2,1}$ can have a root face that is a face of $G$. All other root faces are pruning faces and, thus, have boundary size $O(\sqrt{\varepsilon sN})$. Since $G' - S_{2,1}$ has at most $2/\varepsilon$ heavy regions, the addition of the boundaries of the root faces of these regions therefore increases the size of $S_2$ by at most $O(s + \sqrt{\varepsilon sN}/\varepsilon) = O(\sqrt{sN}/\varepsilon)$.

Separating a heavy component $T_{i,j}$ of a nesting forest $T_i$ of a heavy region $R'_j$ from the remaining components in $T_i$, adds two paths in the low-diameter spanning tree $T'$ of the containing region $R'$ of $G' - S_1$ to $S_2$. By Lemma 3.3, each such path has length at most $4\sqrt{\varepsilon sN}$. Since there are at most $2/\varepsilon$ heavy components in the nesting forests of all heavy regions of $G' - S_{2,1}$, separating these components from the remaining components in their nesting forests thus adds $O(\sqrt{\varepsilon sN}/\varepsilon) = O(\sqrt{sN}/\varepsilon)$ edges to $S_2$.

Similarly, separating each pair of consecutive groups of light components of a nesting forest $T_i$ of a heavy region $R'_i$ from each other adds one path in the low-diameter spanning tree $T'$ of the containing region $R'$ of $G' - S_1$ to $S_2$. There are at most $4/\varepsilon$ such pairs in the nesting forests of all heavy regions of $G' - S_{2,1}$ because the groups in each such pair have total weight greater than $\varepsilon$ or total vertex count greater than $\varepsilon N$, and every group belongs to at most two pairs. Thus, separating these groups of light components from each other adds at most $O(\sqrt{\varepsilon sN}/\varepsilon) = O(\sqrt{sN}/\varepsilon)$ edges to $S_2$.

Finally, consider the partitioning of regions corresponding to heavy components of nesting forests. Let $T_{i,j}$ be such a component, and let $f^*$ be a node of $T_{i,j}$ such that we add the edges incident to $f^*$ to the edge set $C_{i,j}$. There are only $O(1/\varepsilon)$ such nodes in the heavy components of the nesting forests of all heavy regions of $G' - S_{2,1}$. Thus, it suffices to prove that each such node $f^*$ adds at most $O(\sqrt{\varepsilon sN})$ edges to $S_2$. The edges added to $S_2$ by $f^*$ are exactly the boundary edges of its corresponding face $f$ of $R''$. To prove that this boundary adds at most $O(\sqrt{\varepsilon sN})$ edges to $S_2$, we show that the boundary edges of $f$ not already in $S_{2,1}$ are part of at most two paths in the low-diameter spanning tree $T'$ of the containing region $R'$ of $G' - S_1$. We distinguish whether or not $f^*$ is the root of $T_{i,j}$.

If $f^*$ is the root of $T_{i,j}$, then all boundary edges of $f$ not in $S_{2,1}$ are edges between $f$ and the root face $r$ or between $f$ and faces corresponding to the children of $f^*$ in $T_{i,j}$. We prove that all but one of these edges belong to $T'$, which implies that all boundary edges of $f$ not in $S_{2,1}$ belong to at most two paths in $T'$. The root face $r$ and $f$ share an edge not in $T'$, namely the edge dual to the edge in $T'$ between $r^*$ and $f^*$. If $r$ and $f$ shared another edge not in $T'$, then $T'$ would contain two edges between $r^*$ and $f^*$, which contradicts that $T^*$ is a tree. Similarly, if $f$ and a face $f'$ corresponding to a child $(f')^*$ of $f^*$ in $T_{i,j}$ shared an edge $e$ not in $T'$, then the dual edge $e^*$ would belong to $T^*$. Since $f$ and $f'$ both correspond to subtrees of $T^*$ adjacent to $r^*$, this would imply that $T^*$ contains a cycle, again a contradiction.

If $f^*$ is not the root of $T_{i,j}$, then $f$ may also share edges not in $S_{2,1}$ with the face $p$ corresponding to the parent $p^*$ of $f^*$ in $T_{i,j}$. The same argument as for faces corresponding to children of $f^*$ in $T_{i,j}$ shows that all edges shared by $f$ and $p$ belong to $T'$. Thus, in this case, the boundary of $f$ consists of a single path of $T'$.

It remains to bound the number of holes in the regions of $G - S_2$. Since $S_1$ is composed of $O(1/\varepsilon)$ simple cycles, there cannot be more than $O(1/\varepsilon)$ holes in the regions of $G - S_1$. Step 2.1 partitions the heavy regions of $G - S_1$ using $O(1/\varepsilon)$ fundamental cycles. Every such cycle that touches the boundary of the partitioned region in one vertex or not at all increases the total
number of holes by at most one. Any cycle that touches the boundary in at least two vertices does not create any holes. In Step 2.2, the boundary of the root face \( r \) of each heavy region \( R' \) contributes at least one edge to \( S_{2,1} \). Thus, adding the whole boundary of \( r \) to \( S_2 \) amounts to adding a number of paths that start and end in distinct vertices that are already part of the separator. This cannot create any new holes. The same argument holds when adding the boundary of a face \( f \) to \( S_2 \) in Step 2.2 because \( f \) shares at least one edge with the root face \( r \) of \( R' \), whose boundary we added to \( S_2 \) previously.

### 3.3 Step 3: Limit the number of regions, boundary size, and boundary cycles.

Steps 1 and 2 of our algorithm together produce a separator \( S_2 \) of the desired total size and which partitions the graph into regions of weight at most \( \varepsilon \) and size at most \( \varepsilon N \) each, but there may be too many regions and individual regions may have too many holes or a too big boundary. The final step of our algorithm first ensures each region has only \( O(1) \) holes and boundary size \( O(\sqrt{\varepsilon sN}) \) and adds only \( O(\sqrt{sN/\varepsilon}) \) to the size of the separator to do so. This done, we merge regions to reduce their number to \( O(1/\varepsilon) \) without increasing their boundary size or number of holes.

#### Reducing boundary sizes.

While there is a region \( R \) whose boundary size is \( b > c\sqrt{\varepsilon sN} \), for an appropriate constant \( c > 0 \), we split it into two or more subregions and partition these subregions recursively to obtain regions with boundary size no more than \( c\sqrt{\varepsilon sN} \). Slightly extending Frederickson’s approach [8, Lemma 2], we triangulate the exterior and holes of \( R \) and assign weight 1 to the boundary edges of \( R \) and weight 0 to all other edges and to the vertices and faces of \( R \). We then use Miller’s simple cycle separator theorem [15] to split \( R \) into two subregions of weight at most \( 2b/3 \) each. Since each face of \( R \) has size at most \( s \) and \( R \) has size at most \( \varepsilon N \), Miller’s algorithm produces a simple cycle \( C \) of length at most \( c'\sqrt{\varepsilon sN} \). Thus, the boundary of \( R \) and \( C \), excluding edges introduced when triangulating the exterior and the holes of \( R \), together partition \( R \) into two or more subregions of boundary size at most \( 2b/3 + c'\sqrt{\varepsilon sN} \), which is at most \( 3b/4 \), for \( c' \) sufficiently larger than \( c' \). This ensures that \( O(|S_2|/\sqrt{\varepsilon N}) = O(1/\varepsilon) \) such splits suffice to produce a partition into regions of boundary size at most \( c\sqrt{\varepsilon sN} \) each. Each application of Miller’s algorithm to a region of size \( O(\varepsilon N) \) takes \( O(\varepsilon N) \) time. Since we do this \( O(1/\varepsilon) \) times, the total cost of this step is linear.

#### Reducing the number of holes.

We use a similar approach to limit the number of holes. We recursively split regions with more than \( a \) holes, for an appropriate constant \( a > 0 \), using Miller’s simple cycle separator theorem. For such a region \( R \), we once again triangulate its exterior and its holes. If the outer boundary of \( R \) has \( k \) edges, this splits the exterior of \( R \) into \( k - 2 \) triangles, and we assign weight \( 1/(k - 2) \) to each of them. We similarly weight the triangles of each hole so that their total weight is 1. We assign weight 0 to all other faces of \( R \) and to the vertices and edges of \( R \).

The two regions produced by the simple cycle \( C \) obtained using Miller’s algorithm have at most two thirds of the holes of \( R \). Thus, \( R \)’s boundary and \( C \), once again excluding edges that were introduced by triangulating the exterior and holes of \( C \), define a partition of \( R \) into subregions that each contain at most two thirds of the holes of \( R \). Note that every hole of \( R \) not completely on one side of \( C \) merges with \( C \) as part of the subregions’ boundaries and thus does not contribute to the hole counts of the subregions. Therefore, if \( R \) has \( h \) holes, every region in the partition of \( R \) defined by \( R \)’s boundary and \( C \) has at most \( 2(h + 1)/3 + 1 \) boundary cycles. The addition of \( C \) to the separator may result in some subregions having boundary size greater than \( c\sqrt{\varepsilon sN} \). For each such region, a constant number of splits as in the previous paragraph suffice to split it into subregions with boundary size at most \( c\sqrt{\varepsilon sN} \) and with at most \( 2h/3 + O(1) \) holes. By choosing \( a \) large enough, we can once again ensure that only \( O(1/\varepsilon) \) such splits are necessary to reduce the number of holes per region to \( O(1) \) while maintaining the bound on the boundary size of each region and maintaining a bound of \( O(\sqrt{\varepsilon sN}/\varepsilon) \) on the total size of the separator. Since each of these \( O(1/\varepsilon) \) splits once again operates on a region of size \( O(\varepsilon N) \), the total cost of these splitting steps is \( O(N) \).

#### Merging regions.

Let \( S_{3,1} \) be the separator obtained so far. \( S_{3,1} \) has all the properties of a simple cycle \( \varepsilon \)-separator, except that we may have created too many regions. The total number of holes is \( O(1/\varepsilon) \) because this was true for the regions of \( G - S_{2,2} \) and each of the \( O(1/\varepsilon) \) splits we have performed to reduce the boundary size or number of holes in a region can create at most one new hole. In this final step, we merge regions to reduce their number to \( O(1/\varepsilon) \) while maintaining the bounds on each region’s size, weight, number of holes, and boundary size.

We merge regions using the uniform graph contraction procedure of Maheshwari and Zeh [14, Section 3]. First we construct the dual \( S_{3,1}^* \) of \( S_{3,1} \) and eliminate parallel edges. The resulting graph \( H \) has one vertex per face of \( S_{3,1} \) (region of \( G - S_{3,1} \)) and an edge between two vertices if their corresponding faces share a boundary edge. We assign four types of weights to the vertices of \( H \) as follows: The weight \( w_1(v) \) of a vertex \( v \in H \) is the weight of the region of \( G - S_{3,1} \) represented by \( v \). In
addition, every boundary vertex or edge \( x \) of \( G - S_{3,1} \) adds its weight to \( w_1(v) \), for an arbitrary vertex \( v \) representing a region incident to \( x \). Weights \( w_2(v), w_3(v) \), and \( w_4(v) \) respectively are the boundary size, number of holes, and vertex count of the region represented by \( v \). Again, every boundary vertex of \( G - S_{3,1} \) increases \( w_4(v) \) by one, for an arbitrary vertex \( v \) representing an incident region.

The uniform graph contraction procedure proceeds in two phases. The contraction phase repeatedly contracts edges in \( H \) to reduce \( H \)'s size. When contracting an edge \( vw \), the weights of the resulting vertex are the sums of the corresponding weights of \( v \) and \( w \). At the end of this contraction phase, every vertex is either heavy or light (see below for a definition), and every light vertex is adjacent only to heavy vertices. The bundling phase greedily merges pairs of light degree-1 and degree-2 vertices with the same set of neighbours. Again, the weights of the vertex resulting from such a merge are the sums of the corresponding weights of the merged vertices. Both phases ensure that the weights of the vertices they produce do not exceed certain upper bounds given as parameters to the algorithm. In our application of this technique, we require that \( w_1(v) \leq \varepsilon, w_2(v) \leq c\varepsilon sN, w_3(v) \leq a, \) and \( w_4(v) \leq \varepsilon N, \) for appropriate constants \( a > 0 \) and \( c > 0 \). We call a vertex heavy if \( w_1(v) > \varepsilon/2, w_2(v) > (c/2)\varepsilon sN, w_3(v) > a/2 \) or \( w_4(v) > \varepsilon N/2 \). Otherwise the vertex is light. Let \( H^C \) be the graph produced from \( H \) using the uniform contraction procedure. The total number of heavy vertices in \( H^C \) is \( O(1/\varepsilon) \) because their total \( w_1-, w_2-, w_3-, \) and \( w_4- \) weights respectively are \( 1, O(\sqrt{sN/\varepsilon}), O(1/\varepsilon), \) and \( N \). Maheshwari and Zeh proved that the total number of light vertices in \( H^C \) is at most six times the number of heavy vertices in \( H^C \). Thus, \( H^C \) has \( O(1/\varepsilon) \) vertices.

The final partition contains one region per vertex in \( H^C \), which immediately implies that it has \( O(1/\varepsilon) \) regions. Each vertex of \( H^C \) represents a group of vertices of \( H \) that were merged into it in the two phases of the contraction procedure. Its corresponding region is the union of the regions of \( G - S_{3,1} \) represented by these vertices of \( H \). The separator consists of all vertices and edges of \( G \) whose incident faces do not all belong to the same region of the partition. The weights assigned to the vertices of \( H \) ensure that each region in the final partition has weight no more than \( \varepsilon \), boundary size \( O(\sqrt{sN}) \), \( O(1) \) holes, and size at most \( \varepsilon N \). Finally, observe that merging the regions of \( G - S_{3,1} \) corresponding to vertices of \( H \) that are merged into a single vertex during the contraction phase produces a connected region because these vertices induce a connected subgraph of the dual of \( G - S_{3,1} \). The bundling phase merges regions only if they are adjacent to the same set of at most two neighbours. Thus, the bundling phase produces potentially disconnected regions, but all connected components of these regions are adjacent to at most two other regions. Since these neighbouring regions correspond to heavy vertices produced by the contraction phase, they are connected. Thus, the regions in the final partition are either connected or are adjacent to at most two other regions, which are connected. It is easily verified that the construction of the graph \( H \), the application of the uniform contraction procedure, and the construction of the final partition from \( H^C \) can each be carried out in linear time. Since we have already argued that the remainder of the algorithm also takes linear time, we have proved Theorem 3.1.

### 4 Computing Multiway Simple Cycle Separators I/O-Efficiently

In this section, we discuss how to combine the separator algorithm from Section 3 with the contraction procedure of Maheshwari and Zeh [14] to obtain an I/O-efficient algorithm for computing multiway simple cycle separators.

**Theorem 4.1.** A simple cycle \( \varepsilon \)-separator can be computed using \( O(\text{sort}(N)) \) I/Os and \( O(N \log N) \) time.

It is easy but tedious to verify that each step of the algorithm in Section 3 can be implemented using \( O(\text{sort}(N)) \) I/Os and \( O(N \log N) \) time using sorting and scanning primitives, as well as primitives to compute the connected components of a planar graph in these bounds [6] and to manipulate the dual, faces, etc. of an embedded planar graph [18]. The only exception is the computation of BFS trees in Step 1 and in all applications of Miller’s simple cycle separator algorithm in Step 3. The only known method to compute BFS or shortest paths in the same I/O bound is the algorithm of [2]. In Section 5.1, we discuss how to reduce its internal-memory computation time to \( O(N \log N) \), but the algorithm requires the input graph to be planar, which the face incidence graph is not, and it requires a multiway (simple cycle) separator to be given as part of the input. We overcome these two problems as follows.

BFS in the face incidence graph \( G_t \) of \( G \) reduces to BFS in the vertex-on-face graph \( G_{vf} \) of \( G \) (see Section 2): a BFS tree \( T_{vf} \) of \( G_{vf} \) can be obtained from a BFS tree \( T^* \) of \( G^* \) by making every face vertex \( f^* \) a child of its grandparent in \( T_{vf} \) (which is also a face vertex).

To overcome the circular dependency between computing separators and computing BFS or shortest paths, we use the bootstrapping approach from [14, Section 7].
The basic idea is to apply graph contraction to $G$ to obtain a compressed graph $G_2$, recursively compute a separator partition $P_2$ of $G_2$ and then undo the contraction to obtain a partition $P'$ of $G$. Due to the expansion, this partition guarantees only a region size and a region boundary size a constant factor larger than we could guarantee by computing a partition for $G$ directly, but this is good enough for computing shortest paths in $G$. Once we have a shortest path tree for $G$, we can use it to compute our final partition $P$ of $G$ using an I/O-efficient implementation of the algorithm from Section 3.

Applying this approach to compute multiway simple cycle separators requires two non-trivial changes, which we describe in detail in this section. First, the bootstrapping procedure from [14] only maintains a vertex separator, a set of vertices whose removal breaks the graph into connected components of the desired size. We need to carry out the expansion of $G_2$ to $G$ more carefully to ensure we recover the regions of a multiway simple cycle separator from $P_2$. Second, our multiway simple cycle separator algorithm requires a BFS tree not of $G$ but of the face-on-vertex graph $G_{vf}$ of $G$. Thus, the partition $P'$ we compute needs to be a partition of $G_{vf}$, not of $G$. Since $G_{vf}$ is a constant factor larger than $G$, the parameters of the contraction procedure need to be adjusted to ensure the recursive call operates on a graph that is a constant factor smaller than $G$.

The notion of simple cycle separators is meaningful only if the graph is embedded. This is why we have so far assumed the input graph is embedded. The applications we discuss in Section 5, on the other hand, make sense even in the absence of an embedding. Thus, if no embedding of the graph is given, the goal of the algorithm in this section is to compute an arbitrary embedding of $G$ along with a multiway simple cycle separator based on this embedding. Since the I/O-efficient planar embedding algorithm of [13] requires a vertex separator, we cannot simply compute an embedding before computing separators, but the computation of the embedding needs to be incorporated in the recursive computation of the separator as in [14, Section 7].

In the remainder of this section, we recall the contraction procedure from [14], discuss how to recover a multiway simple cycle separator when undoing this contraction, and finally put things together to obtain a planar embedding and a multiway simple cycle separator in the I/O and time bounds stated in Theorem 4.1.

**Graph contraction.** The graph contraction procedure of [14] proceeds in two phases. Both phases are driven by a bound $\beta$ on the weight of each vertex in the contracted graph, which is the number of vertices in $G$ it represents. Initially, we set the contracted graph to be $G$, and every vertex has weight 1. The first phase repeatedly computes a matching of the current graph and contracts the edges in the matching. The vertex resulting from the contraction of a matching edge $uv$ represents the vertices in $G$ represented by $u$ and $v$. Thus, its weight is the sum of the weights of $u$ and $v$. The first phase ensures that no vertex in the current graph has weight greater than $\beta$ and continues contracting edges until each vertex of weight less than $\beta/2$ is adjacent only to vertices of weight at least $\beta/2$. The second phase groups degree-1 and degree-2 vertices based on the vertices they are adjacent to. Each such group of vertices with the same set of neighbours is greedily divided into maximal subgroups such that the total weight of the vertices in each group is at most $\beta$. The vertices in each such subgroup are merged into a single vertex of weight equal to the total weight of the subgroup.

As shown in [14], these two contraction phases can be implemented using $O(sort(N))$ I/Os and $O(N \log N)$ computation time in memory. The final graph $G'$ obtained using this procedure is planar, has no vertex of weight greater than $\beta$, and has at most $14N/\beta$ vertices, where $N$ is the number of vertices in $G$.

**Expanding the separator.** Since a multiway cycle separator is essentially defined by grouping the faces of $G$ into regions, it is tempting to apply the above contraction procedure to the dual of $G$, compute a multiway simple cycle separator of the graph that has the contracted graph as its dual, and then replace each face in the resulting partition with the corresponding faces of $G$. The reason we do not choose this strategy is that edge contraction does not preserve vertex degrees, that is, face sizes in the primal, a property of the graph every simple cycle separator algorithm relies on. Instead, we contract the primal (which cannot increase face sizes) and then recover the boundary cycles of the regions in $P'$ from those of the regions in $P_2$. Here we discuss how to construct $P'$ from $P_2$.

Let $E$ be an embedding of $G$ with respect to which we want to obtain a multiway simple cycle separator, let $G_2$ be the graph obtained by applying the contraction procedure to $G$, and let $E_2$ be the embedding of $G_2$ constructed as part of the contraction. Recall that $G_2$ is obtained from $G$ in two phases. The contraction phase obtains a graph $G_1$ and an embedding $E_1$ of $G_1$ by repeatedly contracting matching edges. The bundling phase groups degree-1 and degree-2 vertices with the same neighbours in $G_1$ to obtain $G_2$. We first construct a simple cycle separator $P_1$ of $G_1$ from the given simple cycle separator $P_2$ of $G_2$ and then construct the simple cycle separator $P'$ of $G$ from $P_1$.

We assume that $P_2$ is represented by labelling each boundary edge of $P_2$ with its two incident regions. Our goal is to first compute the same representation of $P'$
G and then use it to label the faces of G with the regions of P' they belong to.

P_1 has the same regions as P. The bundling of degree-1 and degree-2 vertices in G_1 to obtain G_2 amounts to removing such vertices and their incident edges from G_1. In constructing P_1, we re-introduce these vertices and edges and embed them according to ε_1. This splits each face f of G_2 into a collection of faces of G_1 but does not alter the boundaries of regions. Thus, the labelling of the boundary edges of P_2 in G_2 ⊆ G_1 also represents P_1.

To obtain the required labelling of the boundary edges of P', we observe that every boundary edge e of P_1 corresponds to a unique edge of P'. We assign e's region labels to this edge, thereby making it a boundary edge of P'. Every boundary vertex v of G_1 represents a connected subgraph G_v of G of constant size because v's weight is at most β and G_1 is obtained from G using only edge contractions. We need to ensure the replacement of v with G_v leaves all boundary cycles that included v intact. We do this by labelling the edges in G_v as follows. Let e_1, e_2,..., e_k be the edges in G corresponding to boundary edges in G_1 incident to v. We compute a spanning tree T_v of G_v and let T'_v be the tree obtained by adding edges e_1, e_2,..., e_k to T_v. We compute an Euler tour of T'_v that respects the embedding of G. Note that each edge in the Euler tour represents a side of an edge of T'_v. We traverse this Euler tour, starting with one side of edge e_1. Every edge that is not a side of an edge e_i inherits its region label from the last side of an edge e_i visited by the Euler tour. This labels the boundary edges of P' that belong to G_v with their incident regions.

To construct the labelling of the faces of G with the regions of P' they belong to, we compute G's dual G* and, for every boundary edge e of G with incident regions R_1 and R_2, label the endpoints of e* as belonging to R_1 and R_2, respectively. Now we remove all edges dual to boundary edges of P' from G* and compute the connected components of the resulting graph. Each such component includes at least one vertex representing a face incident to a boundary edge, and all such vertices received the same region label. We copy this region label to all other vertices in the same connected component. Again, using standard primitives, such as the computation of Euler tours, the computation of the dual of a planar graph, etc., this computation of P' from P_1 takes O(sort(N)) I/Os and O(N log N) time.

†It also labels some edges of G_v interior to regions of P', but these edges are easy to distinguish from boundary edges because both their sides receive the same region label.

**Bootstrapping.** Now consider a planar graph G for which we want to compute a planar embedding E and a multiway cycle separator P. First we compute E unless an embedding of G is already given. This can be done using O(sort(N)) I/Os and O(N log N) time, given an O(B^2/N)-vertex separator of G [13]. To compute such a vertex separator P', we apply the above contraction procedure with a parameter β = O(1) to be determined later. We recursively compute an embedding and a simple cycle B^2/N-separator of the resulting graph G_C, use the algorithm from Section 5.1 to compute a shortest path tree T^C of G_C, and then use this shortest path tree T^C together with the vertex separator algorithm of [14] to obtain a B^2/N-vertex separator P_C of G_C. P' now contains one region per region of P_C. The region R' in P' corresponding to a region R in P_C contains all vertices of G represented by vertices in R, that is, that were contracted into vertices in R during the contraction step. The separator vertices of P' are all vertices of G represented by separator vertices of P_C. Since each vertex of G_C expands into at most β vertices of G, P' is an O(B^2/N)-vertex separator of G. The cost of this procedure is O(sort(N)) I/Os and O(N log N) internal-memory computation time plus the cost of recursively computing an embedding and a cycle separator of G_C. If I(N) and T(N) respectively denote the I/O and time bounds of our algorithm on an N-vertex graph, this recursive call takes at most I(14N/β) I/Os and T(14N/β) time because G_C has at most 14N/β vertices.

With an embedding E of G in hand, we now use a similar approach to obtain a simple cycle ε-separator of G. As already stated, this takes O(sort(N)) I/Os and O(N log N) time with the exception of the cost of computing BFS trees. We need to compute BFS trees of the vertex-on-face graph G_vf of G, of the regions produced in Step 1 of our algorithm, and of the vertex-on-face graphs of the regions we partition in Step 3. For each such graph H, we apply the contraction procedure to obtain a compressed graph H^C along with an embedding E_C of H^C consistent with E. We use the same parameter β as when constructing G_C from G. We compute a simple cycle B^2/N-separator P_C of H^C w.r.t. embedding E_C by invoking our algorithm recursively on H^C. Then we use the expansion procedure above to obtain a simple cycle O(B^2/N)-separator P'' of H and use this cycle separator to compute a BFS tree T of H. Thus, the cost of the recursive call involved in computing a BFS tree for H is I(14H/β) I/Os and T(14H/β) time. Since I(N) and T(N) are convex functions, we have ∑_H I(14H/β) ≤ I(∑_H 14H/β) and ∑_H T(14H/β) ≤ T(∑_H 14H/β). Now it suffices to observe that the total size of all graphs for which
we need to compute BFS trees in Steps 1 and 3 of our algorithm is $O(N)$. Thus, we can choose $\beta$ large enough so that $14N/\beta \leq N/4$ (the size of $G^C$ in computing an embedding $E$ of $G$) and $\sum_{H} 14|H|/\beta \leq N/4$. For this choice of $\beta$, we obtain recurrences for the I/O complexity and running time of our algorithm bounded by $I(N) \leq 2I(N/4) + O(sort(N))$ and $T(N) \leq 2T(N/4) + O(N \log N)$, which solve to $I(N) = O(sort(N))$ and $T(N) = O(N \log N)$. This proves Theorem 4.1.

5 Applications

In this section, we discuss how to use a separator obtained using the algorithm from Section 4 to design algorithms for planar graphs that are efficient in internal and external memory at the same time. In particular, we prove the following result.

Theorem 5.1. The single-source shortest path problem on a planar graph with non-negative edge lengths, topological sorting of a planar DAG, and computing the strongly connected components of a planar graph can be solved using $O(sort(N))$ I/Os and $O(N \log N)$ time, where $N$ is the number of vertices in the graph.

Algorithms for these problems with the same I/O bounds as in Theorem 5.1 were presented in [2,3,5], but these algorithms were not efficient in internal memory. Here we exploit the structure of the graph partition from Section 3 to obtain variants of these algorithms with the same I/O bounds and which are also efficient in internal memory. All three algorithms use the same four-step procedure and assume that $M \geq cB^2$, for some constant $c > 0$. We recall this procedure in our discussion of the shortest path algorithm in Section 5.1 and then discuss the changes necessary to use this approach to compute a topological ordering (Section 5.2) or strongly connected components (Section 5.3). Since the algorithms require a separator partition with a small number of boundary sets, we need to assume the input graphs have bounded degree. In the case of shortest paths and strong connectivity, this assumption is easily satisfied by replacing each high-degree vertex with a directed cycle of degree-3 vertices. This does not change the strong connectivity of the graph and, if the edges in each cycle are chosen to be of length 0, neither the distances between vertices. In the case of topological sorting, we are not aware of any such simple transformation that preserves planarity and acyclicity and reduces the degree of each vertex to $O(1)$. We discuss at the end of Section 5.2 how to obtain a topological sorting algorithm for planar DAGs without a bound on their vertex degrees.

5.1 Single-source shortest paths. The shortest-path algorithm of [2] proceeds in four steps. In the first step, it computes a partition $P$ of the graph into $O(N/B^2)$ subgraphs (or regions) $G_1, G_2, \ldots, G_r$ of size at most $cB^2$, boundary size $O(B)$, and with $O(N/B^2)$ boundary sets. Using our algorithm from Section 4, such a separator partition can be obtained using $O(sort(N))$ I/Os and $O(N \log N)$ time. In contrast to previous separator algorithms, our algorithm also ensures that each connected subregion of each region $G_i$ has only $O(1)$ boundary cycles. This is the key to making the shortest path algorithm efficient in internal memory.

Step 2 computes a compressed graph $G^R$ on the boundary vertices of all regions. This graph is obtained by replacing each region $G_i$ with a graph $G^R_i$ over its boundary vertices. Let $G_{i1}, G_{i2}, \ldots, G_{ik_i}$ be the connected subregions of $G_i$. Then $G^R_i$ contains an edge $uv$ if and only if $u$ and $v$ are on the boundary of the same subregion $G_{ij}$ and its length is $\ell(uv) := dist_{G_{ij}}(u,v)$ in this case. It is easily verified that this implies that $dist_{G^R_i}(u,v) = dist_G(u,v)$, for any two vertices $u, v \in G^R_i$.

Step 3 computes $dist_{G^R_i}(s,v) = dist_G(s,v)$, for all vertices $v \in G^R_i$. As shown in [2], this can be done using $O(sort(N))$ I/Os and $O(N \log N)$ time because $G^R_i$ has $O(N/B)$ vertices and $O(N)$ edges, and the partition of $G$ was derived from has $O(N/B^2)$ boundary sets.

Step 4 finally computes the distances $dist_{G_i}(s,v)$, for all vertices $v \in G$. To do this, it inspects each region $G_i$ in turn and computes $dist_G(s,v) = \min_{u \in G_i}(dist_G(s,u) + dist_G(u,v))$, for each non-boundary vertex $v \in G_i$, where the minimum is taken over all boundary vertices $u$ of $G_i$. Since each region fits in memory, this can be done by loading each region into memory in turn and running Dijkstra’s algorithm on this region in memory. Thus, this step takes $O(N/B)$ I/Os and $O(N \log N)$ time.

It remains to discuss the cost of Step 2 above. Its I/O complexity is $O(N/B)$, for the same reason Step 4 achieves this I/O complexity: it suffices to load each region $G_i$ into memory, generate $G_i^R$ in memory and then write $G_i^R$ back to disk. By summing the I/O complexities of the four steps, we obtain an I/O bound of $O(sort(N))$ I/Os for the whole algorithm. This is the same argument as in [2]. The construction of $G^R_i$ from $G_i$ in memory requires the computation of distances between all boundary vertices of each connected subregion $G_{ij}$ of $G_i$. A naive implementation runs $O(B)$ single-source shortest path computations on $G_i$, one per boundary vertex. Since $G_i$ has $O(B^2)$ vertices, this takes $O(B^3)$ time using the linear time single-source shortest path algorithm of [9]. Using an all-pairs shortest path algorithm instead is no faster.
Since we have to repeat this step for each of the $O(N/B^2)$ regions in $P$, the total time used in Step 2 is $O(N/B^2 \cdot B^3) = O(NB)$. Next we discuss how to reduce the internal-memory cost of constructing $G_i^R$ from $G_i$ to $O(B^2 \log B)$. This reduces the internal-memory cost of Step 2 to $O(N \log B)$ and the internal-memory cost of the whole algorithm to $O(N \log N)$, which proves the cost of computing shortest paths claimed in Theorem 5.1.

To compute the distances between all boundary vertices of each connected subregion $G_{i,j}$, we use the fact that $G_{i,j}$ has only $O(1)$ boundary cycles, each of which bounds a face of $G_{i,j}$. For each such cycle $C$, we can use an algorithm by Klein [10] to compute a shortest path tree $T_u$ for each vertex $u \in C$. This algorithm takes $O(n_{i,j} \log n_{i,j})$ time to construct all these trees, where $n_{i,j}$ is the number of vertices in $G_{i,j}$. The computed representation of each such shortest path tree $T_u$ supports distance queries between $u$ and any vertex $v \in G_{i,j}$ in $O(\log n_{i,j})$ time. Since $G_{i,j}$ has $O(1)$ boundary cycles, it thus takes $O((n_{i,j} + b_{i,j}^2) \log n_{i,j})$ time to compute the distances between all boundary vertices of $G_{i,j}$, where $b_{i,j}$ is the number of these boundary vertices. Now it suffices to observe that $\sum_{j=1}^{k_{i,j}} n_{i,j} = O(B^2)$ and $\sum_{j=1}^{k_{i,j}} b_{i,j} = O(B)$ to conclude that the total cost of computing $G_i^R$ is $O(B^2 \log B)$, as claimed.

5.2 Topological sorting. To obtain a topological ordering of the vertices of a planar DAG $G$, Arge and Toma [3] start with the following observation.$^3$ Let $G^*$ be a DAG obtained by adding a new source vertex $s$ to $G$ and connecting $s$ to each source $v$ of $G$ using an edge $sv$. Let the level $\ell_{G^*}(v)$ of each vertex $v$ of $G$ be the length of the longest path from $s$ to $v$ in $G^*$. A topological ordering of $G$ can be obtained by sorting the vertices of $G$ by their levels. To compute these levels, Arge and Toma employ the same 4-step strategy as in the shortest path algorithm discussed in Section 5.1. Step 1 computes a graph partition of $G$ with the same properties as in Section 5.1. Step 2 constructs a graph $G^R$ whose vertex set contains the source $s$ of $G^*$ and all boundary vertices of $G$. To construct $G^R$, each region $G_i$ in the partition is replaced with a graph $G_i^R$ whose vertex set contains $s$ and the boundary vertices of $G_i$. For every pair of boundary vertices $(u,v)$ of $G_i$, $G_i^R$ contains an edge $uv$ if and only if there exists a connected subregion $G_{i,j}$ that has $u$ and $v$ on its boundary and there exists a path from $u$ to $v$ in $G_{i,j}$. The length of edge $uv$ is the length of the longest path from $u$ to $v$ in $G_{i,j}$ in this case. In addition, let $G_i^s$ be the subgraph of $G^s$ containing $s$ and all vertices of $G_i$. Then $G_i^R$ contains an edge $sv$, for every boundary vertex $v$ of $G_i$ that is reachable from $s$ in $G_i^s$. The length of this edge is the length of the longest path from $s$ to $v$ in $G_i^s$. Once again, it is not difficult to see that $\ell_{G^R}(v) = \ell_{G^s}(v)$, for every vertex $v \in G_i^R$. Step 3 computes $\ell_{G^R}(v)$, for all $v \in G_i^R$, by iteratively removing sinks and updating the levels of their out-neighbours. Step 4 computes $\ell_{G^s}(v)$, for all non-boundary vertices of $G$. To this end, it inspects each region $G_i$ in turn and computes the levels of all vertices in a weighted supergraph $\bar{G}_i^s$ of $G_i^s$. In addition to the vertices and edges of $G_i^s$, $\bar{G}_i^s$ contains an edge $sv$, for each boundary vertex $v$ of $G_i$. All edges of $G_i^s$ have weight 1 in $\bar{G}_i^s$, while each edge $sv$ not in $G_i^s$ has weight $\ell_{G^R}(v) = \ell_{G^s}(v)$. These weights ensure that $\ell_{G_i^R}(v) = \ell_{G_i^s}(v)$, for every vertex $v \in G_i^R$. These levels can be computed for all vertices in $\bar{G}_i^s$ by once again iteratively removing sinks from $\bar{G}_i^s$ and updating the levels of their out-neighbours.

Arge and Toma [3] argued that the I/O complexity of this algorithm is $O(\text{sort}(N))$. Step 1 takes $O(N \log N)$ time in internal memory, while Steps 3 and 4 take $O(1)$ time per vertex and edge and operate on graphs of total size $O(N)$. Thus, it remains to bound the internal-memory computation time of Step 2. In particular, we argue once again that the construction of $G_i^R$ from $G_i$ takes $O(B^2 \log B)$ time.

The addition of edges $sv$ to $G_i^R$, for all boundary vertices of $G_i$ reachable from $s$ in $G_i^s$, and the computation of the weights of these edges is yet another level computation on $G_i^s$ and hence takes $O(B^2)$ time. To add edges between the boundary vertices of each connected subregion $G_{i,j}$, we observe that the length of the longest path from $u$ to $v$ in $G_{i,j}$ is the negation of the length of the shortest path from $u$ to $v$ in $G_{i,j}$ after assigning length $-1$ to each edge. As noted by Klein et al. [11, Section 2.5], the algorithm of Klein [10] can be used to compute shortest paths from all vertices on the same boundary cycle $C$ even in the presence of negative-length edges, provided the distances from some vertex $v \in C$ to all other vertices in $G_{i,j}$ are known. Computing these distances requires a longest-path computation (with the original edge lengths) similar to the addition of edges $sv$ to $G_i^R$. Only this time we operate on a graph $G_{i,j}$, which is obtained from $G_{i,j}$ by adding a single length-0 edge $sv$. We repeat this distance computation once per boundary cycle of $G_{i,j}$. Since $G_{i,j}$ has $O(1)$ boundary cycles, this takes $O(n_{i,j})$ time. Computing the distances between all pairs of boundary vertices of $G_{i,j}$ using Klein’s algorithm takes $O((n_{i,j} + b_{i,j}^2) \log n_{i,j})$ time again, which dominates the cost of processing the subregion $G_{i,j}$. Thus, by summing these costs for all
subregion of $G_i$ as in Section 5.1, we obtain a bound of $O(B^2 \log B)$ for the internal-memory cost of constructing $G^R_i$ from $G_i$ and a bound of $O(N \log B)$ for the total internal-memory cost of Step 2.

One limitation of the algorithm just described is that it applies only to planar graphs of bounded degree. In [4], a more complicated topological sorting algorithm is presented that does not make this assumption but uses an I/O-efficient shortest path algorithm to compute two rooted spanning trees of the dual of the given DAG. Using the algorithm discussed in Section 5.1, these spanning trees can be computed using $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. It is easy but tedious to verify that the remainder of the algorithm of [4] performs only $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. Thus, we obtain the complexity of topologically sorting planar DAGs claimed in Theorem 5.1.

5.3 Strongly Connected Components. To compute the strongly connected components of a planar graph, Arge and Zeh [5] use the following algorithm, once again based on the 4-step framework from Section 5.1. Step 1 computes a graph partition with the same properties as in Section 5.1. Step 2 computes the strongly connected components of a planar graph, Arge and Zeh [5] use the following algorithm, once again based on the 4-step framework from Section 5.1. Step 2 computes a graph $G^R$ from $G$ by replacing each region $G_i$ of the partition with a graph $G^R_i$ over its boundary vertices. $G^R_i$ contains an edge $uv$ if and only if there exists a connected subregion $G_{i,j}$ of $G_i$ that has $u$ and $v$ on its boundary and there exists a path from $u$ to $v$ in $G_{i,j}$. It is easy to verify that two vertices of $G^R_i$ belong to the same strongly connected component of $G_i$. Step 3 assigns a component label to each vertex $u$ of $G^R_i$, which uniquely identifies the strongly connected component of $G^R_i$ and hence of $G$ that contains $u$. Step 4 extends this labelling by assigning component labels to the non-boundary vertices of $G$. To this end, it processes each region $G_i$ in turn. Region $G_i$ is augmented with cycles connecting the boundary vertices of $G_i$ that belong to the same strongly connected component of $G$ according to the component labels computed in Step 3. Let $G'_i$ be the resulting graph. Next the algorithm assigns component labels to the vertices of $G'_i$ such that every boundary vertex of $G'_i$ receives the same component label as in Step 3 and two vertices in the same strongly connected component of $G'_i$ receive the same label. It is easy to verify that this ensures that the final labelling assigns the same component label to two vertices if and only if they belong to the same strongly connected component of $G$.

Once again, we use our separator algorithm from Section 4 to compute the graph partition in Step 1 using $O(\text{sort}(N))$ I/Os and $O(N \log N)$ time. Arge and Zeh showed that the remainder of the algorithm takes $O(\text{sort}(N))$ I/Os, and it is easy to verify that the implementations of Steps 3 and 4 provided in [5] take $O(N)$ time. To implement Step 2, it suffices to compute the distances between the boundary vertices of each connected subregion $G_{i,j}$ as discussed in Section 5.1. If this distance is finite, a path between the two vertices exists, and we add the corresponding edge to $G^R_i$; otherwise we do not. Thus, Step 2 takes $O(N \log B)$ time as in Section 5.1, and we obtain a strong connectivity algorithm with the complexity claimed in Theorem 5.1.

6 Conclusions

In this paper, we have presented algorithms to compute a natural extension of Miller’s simple cycle separators that partitions the graph into more than two pieces, and we have shown that these multiway simple cycle separators can be used to reduce the internal-memory computation cost of I/O-efficient algorithms for planar graphs.

Our analysis in this paper did not focus on bounding the constants in the size of the computed separator, and we believe that doing so using the arguments used in our proofs results in rather large constants. However, a much more careful analysis of the algorithm we presented here is likely to produce much more reasonable constants.

Miller’s simple cycle separator partitions the graph into two connected regions, while some of the regions our algorithm produces may be disconnected. The reason for obtaining such disconnected regions is the grouping of small regions in Step 3 of our separator algorithm. It seems possible that a (significantly more complicated) version of Step 3, which modifies the separator obtained in Steps 1 and 2 to keep these grouped regions connected to each other, results in a partition that consists of only connected regions. Such a partition would be an even closer multiway equivalent of Miller’s two-way simple cycle separators. We did not focus on obtaining such a partition here because this would have complicated Step 3 substantially and the partition we obtained in this paper is good enough for the applications we considered.

References


