SIAM J. COMPUT.
Vol. 38, No. 3, pp. 767–801

I/O-EFFICIENT PLANAR SEPARATORS*

ANIL MAHESHWARI† AND NORBERT ZEH‡

Abstract. We present I/O-efficient algorithms for computing optimal separator partitions of planar graphs. Our main result shows that, given a planar graph G with N vertices and an integer $r > 0$, a vertex separator of size $O(N/\sqrt{r})$ that partitions G into $O(N/r)$ subgraphs of size at most r and boundary size $O(\sqrt{r})$ can be computed in $O(\text{sort}(N))$ I/Os. This bound holds provided that $M \geq 56r \log^2 B$. Together with an I/O-efficient planar embedding algorithm presented in [N. Zeh, *I/O-Efficient Algorithms for Shortest Path Related Problems*, Ph.D. thesis, School of Computer Science, Carleton University, Ottawa, ON, Canada, 2002], this result is the basis for I/O-efficient solutions to many other fundamental problems on planar graphs, including breadth-first search and shortest paths [L. Arge, G. S. Brodal, and L. Toma, *J. Algorithms*, 53 (2004), pp. 186–206; L. Arge, L. Toma, and N. Zeh, *I/O-efficient algorithms for planar digraphs*, in Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, 2003, pp. 85–93], depth-first search [L. Arge et al., *J. Graph Algorithms Appl.*, 7 (2003), pp. 105–129; L. Arge and N. Zeh, *I/O-efficient strong connectivity and depth-first search for directed planar graphs*, in Proceedings of the 44th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 2003, pp. 261–270], strong connectivity [L. Arge and N. Zeh, *I/O-efficient strong connectivity and depth-first search for directed planar graphs*, in Proceedings of the 44th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 2003, pp. 261–270], and topological sorting [L. Arge and L. Toma, *Simplified external memory algorithms for planar DAGs*, in Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 3111, Springer-Verlag, Berlin, New York, 2004, pp. 493–503; L. Arge, L. Toma, and N. Zeh, *I/O-efficient algorithms for planar digraphs*, in Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, 2003, pp. 85–93]. Our second result shows that, given I/O-efficient solutions to these problems, a general separator algorithm for graphs with costs and weights on their vertices [L. Aleksandrov et al., *Partitioning planar graphs with costs and weights*, in Proceedings of the 4th Workshop on Algorithm Engineering and Experiments, Lecture Notes in Comput. Sci. 2409, Springer-Verlag, Berlin, New York, 2002, pp. 98–107] can be made I/O-efficient. Many classical separator theorems are special cases of this result. In particular, our I/O-efficient version allows the computation of a separator as produced by our first separator algorithm, but without placing any constraints on r in relation to the memory size.

Key words. I/O-efficient algorithms, memory hierarchies, graph algorithms, planar graphs, graph separators

AMS subject classifications. 49M27, 68Q25, 90C06, 90C35

DOI. 10.1137/S0097539705446925

1. Introduction.

I/O-efficient graph algorithms have received considerable attention because massive graphs arise naturally in many applications. Recent Web crawls, for example, produced graphs of on the order of 200 million nodes and 2 billion edges. Recent work in Web modeling uses depth-first search (DFS), breadth-first search (BFS), and the computation of shortest paths and connected components as primitive operations for investigating the structure of the Web [12]. Massive graphs

*Received by the editors February 4, 2005; accepted for publication (in revised form) January 11, 2008; published electronically May 28, 2008. An extended abstract of this paper appeared in the *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco), ACM, New York, SIAM, Philadelphia, 2002, pp. 382–389. This research was supported by NSERC and NCE GEOIDE.

http://www.siam.org/journals/sicomp/38-3/44692.html

†School of Computer Science, Carleton University, Ottawa, ON, K1S 5B6, Canada (maheshwa@scs.carleton.ca).

‡Faculty of Computer Science, Dalhousie University, Halifax, NS, B3H 1W5, Canada (nzeh@cs.dal.ca).

are also often manipulated in geographic information systems (GIS), where many fundamental problems can be formulated as basic graph problems. Yet another example of a massive graph is AT&T's phone call graph [13]. When working with such large data sets, the transfer of data between internal and external memory, and not the internal-memory computation, is often the bottleneck. Thus, I/O-efficient algorithms can lead to considerable run time improvements.

Planar graphs are a natural abstraction of many real-world problems. For example, the graphs arising in GIS are often planar or "almost planar." On the theoretical side, planar graphs are among the fundamental combinatorial structures used in algorithmic graph theory. Planar separators have played the key role in designing divide-and-conquer algorithms for planar graphs. The classical separator theorem for planar graphs by Lipton and Tarjan [27], coupled with linear-time planar embedding algorithms [10, 18, 24, 26], has led to phenomenal developments in algorithmic graph theory. Numerous research results that followed describe efficient algorithms for computing a variety of separators of other sparse graphs and discuss applications of separators such as lower bounds on the size of Boolean circuits, approximation algorithms for NP-complete problems, nested dissection of sparse systems of linear equations, load balancing in parallel numerical simulations based on the finite element method, partitioning triangular irregular networks in the field of GIS, and encoding graphs.

In external memory, planar separators have been the key to obtaining I/O-efficient algorithms for a variety of problems on embedded planar graphs, including BFS and single-source shortest paths [5, 8], DFS [6, 9], strong connectivity [9], and topological sorting [8, 7].

Existing internal-memory algorithms for computing planar separators (see, e.g., [2, 3, 27]) are not I/O-efficient because they use BFS to gather structural information about the graph, and BFS and DFS are among those fundamental graph problems for which truly I/O-efficient solutions on general graphs are still lacking. The existing I/O-efficient BFS-algorithms for planar graphs [5, 8] are separator-based; hence, using them in a separator algorithm creates a circular dependency. In this paper, we present a new algorithm that applies BFS only to a compressed version of the given graph and combines this with graph contraction techniques to obtain an optimal separator partition in an I/O-efficient manner. An added benefit of our algorithm is that it does not rely on a planar embedding of the given graph.

1.1. Model of computation and previous work. The algorithms in this paper are designed and analyzed in the I/O-model of Aggarwal and Vitter [1]. This model quite accurately captures the characteristics of current hard drives and yet is simple enough to allow the I/O-complexity of complex algorithms to be analyzed. In the I/O-model, the computer is equipped with two levels of memory. The *internal memory* (or *memory* for short) is of limited size, capable of holding M data items. The disk-based *external memory* is of conceptually unlimited size and is divided into blocks of B consecutive data items. All computation has to be performed on data in internal memory. The transfer of data between internal and external memory happens by means of *I/O-operations* (or *I/Os* for short), each of which transfers one block of data to or from the disk. The complexity of an algorithm is the number of I/O-operations it performs.

For surveys of results obtained in the I/O-model and its extensions, we refer the reader to [28, 32]. The following results are relevant to the work presented in this paper.

It has been shown in [1] that sorting and permuting an array of N data items

take $\mathrm{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ and $\mathrm{perm}(N) = \Theta(\min\{N, \mathrm{sort}(N)\})$ I/Os, respectively; scanning an array of size N takes $\Theta(N/B)$ I/Os.

In internal memory, that is, in the RAM model, the problem of computing planar separators is well studied. Lipton and Tarjan [27] were the first to show that every planar graph with N vertices has a $\frac{2}{3}$-separator of size $\mathrm{O}(\sqrt{N})$, that is, a vertex set of size $\mathrm{O}(\sqrt{N})$ whose removal partitions G into two subgraphs containing at most $2N/3$ vertices each. They also presented a linear-time algorithm to compute such a separator. In [21], it has been shown that every graph of genus g has a $\frac{2}{3}$-separator of size $\mathrm{O}(\sqrt{gN})$ and that such a separator can be computed in linear time. In [2], a linear-time algorithm for computing a t-separator of size $\mathrm{O}(\sqrt{(g+1/t)N})$ for an embedded graph of genus g was given; for $0 < t < 1$, a *t-separator* is a vertex set whose removal partitions G into subgraphs of size at most tN. Other results deal with computing small simple-cycle separators of planar graphs [29], edge separators of planar graphs [15], separators of planar graphs with negative or multiple vertex weights [17], and separators of low cost [3, 16].

In [25], the first I/O-efficient algorithm for computing planar separators was presented. This algorithm is an external-memory version of Lipton and Tarjan's algorithm and computes a $\frac{2}{3}$-separator of size $\mathrm{O}(\sqrt{N})$; its I/O-complexity is $\mathrm{O}(\mathrm{sort}(N))$, provided that a planar embedding and a BFS-tree of the graph are given. In [5], an external version of Goodrich's multiway separator algorithm [22] has been developed. This algorithm computes a t-separator of size $\mathrm{O}((N/B)\log_{M/B}(N/B) + \sqrt{N/t})$ and takes $\mathrm{O}(\mathrm{sort}(N))$ I/Os, again assuming that an embedding and a BFS-tree are given.

A number of subsequent papers develop a hierarchy of reductions that lead to $\mathrm{O}(\mathrm{sort}(N))$-I/O algorithms for a variety of fundamental problems on embedded planar graphs if an optimal separator decomposition can be obtained in $\mathrm{O}(\mathrm{sort}(N))$ I/Os: In [5, 8], separator-based ideas first pioneered in [19] were used to obtain I/O-efficient shortest-path algorithms for undirected and directed planar graphs. These algorithms can, of course, also be used to compute BFS-trees of planar graphs. In [6], a DFS-algorithm for undirected planar graphs was presented; this algorithm uses BFS in a planar graph derived from the dual and, hence, also depends on planar separators. The directed DFS-algorithm of [9] needs to compute directed spanning trees of the graph; currently, the only I/O-efficient algorithm for this problem is the shortest-path algorithm of [8]. Other I/O-efficient algorithms for planar graphs that rely on separators are the strong connectivity algorithm of [9] and the algorithms of [7, 8] for topologically sorting planar directed acyclic graphs.

1.2. New results. The two main results of our paper are the following:

(i) Given a planar graph G with N vertices and an integer $r > 0$, it takes $\mathrm{O}(\mathrm{sort}(N))$ I/Os to compute a vertex separator S of size $\mathrm{O}(N/\sqrt{r})$ whose removal partitions G into $\mathrm{O}(N/r)$ subgraphs of size at most r and boundary size $\mathrm{O}(\sqrt{r})$. The bound on the I/O-complexity of the algorithm holds as long as the internal memory has size at least $56r\log^2 B$.

(ii) Using a bootstrapping approach based on our second algorithm, discussed below, the memory requirements of the algorithm can be reduced from $M = \Omega(B^2\log^2 B)$ to $M = \Omega(B^2)$ for the special case when $r = B^2$. This special case is important because $r = B^2$ is the granularity of the partition required by all separator-based I/O-efficient algorithms for planar graphs that have been developed so far.

Thanks to our algorithm, a wide variety of problems, as discussed in the previous section, can be solved in $\mathrm{O}(\mathrm{sort}(N))$ I/Os if $M = \Omega(B^2)$. In particular, a shortest-path tree of a planar graph can be obtained in this complexity. Using this fact, we

show the following I/O-efficient versions of results from [3]:

(iii) Let $G = (V, E)$ be a planar graph with N vertices, let $w : V \to \mathbb{R}^+$ and $c : V \to \mathbb{R}^+$ be assignments of nonnegative weights and costs to the vertices of G, and let $0 < t < 1$ be an arbitrary constant. Let $w(G) = \sum_{x \in V} w(x)$ and $C(G) = \sum_{x \in V} (c(x))^2$. If the size of the internal memory is $\Omega(B^2)$, it takes $O(\mathrm{sort}(N))$ I/Os to compute a vertex separator S of cost $c(S) \leq 4\sqrt{2 \cdot C(G)/t}$ such that no connected component of $G - S$ has weight exceeding $tw(G)$.

If all vertices have weight and cost equal to 1 and we choose $t = r/N$, we obtain a separator of size $O(N/\sqrt{r})$ that partitions the graph into pieces of size at most r. This matches the result produced by our first algorithm, but without requiring that $M \geq 56r \log^2 B$. More precisely, it allows the computation of arbitrarily coarse partitions, while our first algorithm is restricted to computing partitions into pieces of size $O(M/\log^2 B)$ if an I/O-complexity of $O(\mathrm{sort}(N))$ is desired.

(iv) Let $G = (V, E)$ be a planar graph with N vertices, let $w : V \to \mathbb{R}^+$ be an assignment of nonnegative weights to the vertices of G, let $0 < t < 1$ be an arbitrary constant, and let $w(G) = \sum_{x \in V} w(x)$. If $w(x) \leq tw(G)$ for every vertex $x \in V$, there exists an edge separator $S \subseteq E$ of size $|S| \leq 4\sqrt{2(\sum_{x \in V} (\deg(x))^2)/t}$ such that no connected component of $G - S$ has weight exceeding $tw(G)$. Such a separator can be computed in $O(\mathrm{sort}(N))$ I/Os, provided that $M = \Omega(B^2)$.

The algorithm in result (i) performs $O(N \log N)$ work in internal memory. Result (iii) and, thus, also results (ii) and (iv) rely on an I/O-efficient shortest-path algorithm for planar graphs. Currently, the best such algorithm performing $O(\mathrm{sort}(N))$ I/Os is the algorithm of [5], which performs $O(N \log N + BN)$ work in internal memory. The algorithms in result (ii)–(iv) inherit this computational bound, but their computational bound would decrease to $O(N \log N + T(N))$ with the development of a planar shortest-path algorithm that performs $O(\mathrm{sort}(N))$ I/Os and $T(N) = o(N \log N + BN)$ computation in internal memory.

The presentation of our results is organized as follows. In section 2, we introduce the necessary terminology and notation and discuss some technical results that will be useful in our algorithms. In section 3, we discuss a graph contraction procedure that is used many times in our algorithms. Our algorithm for partitioning an unweighted planar graph is presented in section 4. The partition produced by the algorithm does not have all the properties required by the shortest-path algorithm of [5] or by any of the other separator-based I/O-efficient algorithms for planar graphs [7, 8, 9]. In section 5, we explain how to ensure the required additional properties. In section 6, we discuss our I/O-efficient algorithm for computing separators of planar graphs with costs and weights. In section 7, we explain how to combine the algorithms from sections 4 and 6 to reduce the memory requirements of our unweighted separator algorithm to $M = \Omega(B^2)$. We present concluding remarks in section 8.

2. Preliminaries.

2.1. Graphs and planarity.

We assume that the reader is familiar with standard graph-theoretic terms and notation as defined, for example, in [23, 31]. In this paper, all graphs are simple, that is, do not contain parallel edges or loops, even though the results are easy to generalize to multigraphs. Let $G = (V, E)$ be a graph. For a set $W \subseteq V$ of vertices, we use $G[W]$ to denote the subgraph of G whose vertex set is W and whose edge set consists of all edges in G that have both endpoints in W; we call $G[W]$ the subgraph of G *induced* by vertex set W. For a set of vertices $W \subseteq V$, let $G - W = G[V \setminus W]$. For a vertex $x \in V$, let $G - x = G - \{x\}$. For a set

of edges $F \subseteq E$, let $G - F = (V, E - F)$.

For a vertex $x \in G$, the *(open) neighborhood* $\mathcal{N}(x)$ of x is the set of vertices adjacent to x, that is, $\mathcal{N}(x) = \{y \in V : xy \in E\}$; the *closed neighborhood* of x is $\mathcal{N}[x] = \{x\} \cup \mathcal{N}(x)$. We generalize this to vertex sets and subgraphs by defining $\mathcal{N}[V'] = \bigcup_{x \in V'} \mathcal{N}[x]$, $\mathcal{N}(V') = \mathcal{N}[V'] \setminus V'$, $\mathcal{N}[G'] = G[\mathcal{N}[V']]$, and $\mathcal{N}(G') = G[\mathcal{N}(V')]$, where $G' = (V', E')$. We often call $\mathcal{N}(G')$ the *boundary* of G'. The *degree* $\deg(x)$ of a vertex x is the number of edges incident to x. Since we assume that G is simple, we have $\deg(x) = |\mathcal{N}(x)|$.

A graph G is *planar* if it can be drawn in the plane so that the edges of G do not intersect, except at their endpoints. Such a drawing of G is called a *topological embedding* of G; we denote it by $\mathcal{E}(G)$. Every topological embedding of G defines an order of the edges incident to each vertex $x \in G$ clockwise around x. A representation of these orders for all vertices $x \in G$ is called a *combinatorial embedding* of G, which we denote by $\hat{G}$. Given a topological embedding $\mathcal{E}(G)$ consistent with a combinatorial embedding $\hat{G}$ of G, we call the connected regions of $\mathbb{R}^2 \setminus \mathcal{E}(G)$ the *faces* of $\hat{G}$. The *size* of a face of $\hat{G}$ is the number of edges on its boundary. Let F denote the set of faces of $\hat{G}$. By Euler's formula, $|V| + |F| - |E| = 2$. In particular, $|E| \leq 3|V| - 6$. We define the *size* $|G|$ of a planar graph $G = (V, E)$ to be the number of vertices in G.

For an embedded planar graph $G = (V, E)$ and an edge $e \in E$, the *dual* e^* of e is the edge $f_1 f_2$, where f_1 and f_2 are the two faces of $\hat{G}$ that have edge e on their boundaries. The dual of G is the multigraph $G^* = (F, E^*)$, where $E^* = \{e^* : e \in E\}$.

A *partition* of a set S is a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subsets of S so that every element of S belongs to exactly one set S_i, that is, $S_i \cap S_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{k} S_i = S$.

Given a graph $G = (V, E)$ and a partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the vertex set of G, the *contraction* of $\mathcal{V}$ in G is the graph $G/\mathcal{V} = (\mathcal{V}, E')$, where $E' = \{V_i V_j : \exists\, xy \in E$ such that $x \in V_i$ and $y \in V_j\}$. If the vertices of G have weights, we define $w(V_i) = \sum_{x \in V_i} w(x)$ for all $1 \leq i \leq k$. If the graph $G[V_i]$ is connected for all $1 \leq i \leq k$, we call $G/\mathcal{V}$ an *edge contraction*. If there exist vertex sets $X_1, \ldots, X_k$ such that, for all $1 \leq i \leq k$ and all $x \in V_i$, $\mathcal{N}(x) \setminus V_i = X_i$, that is, if all vertices in V_i have the same neighbors in $V(G) \setminus V_i$, we call $G/\mathcal{V}$ a *vertex bundling*. We will use the following facts. The first one states that edge contractions and vertex bundlings preserve planarity. Intuitively, the second one says that undoing any contraction preserves separators.

FACT 2.1. *If G is planar, then every edge contraction or vertex bundling of G is planar.*

FACT 2.2. *Let $\mathcal{S} \subseteq \mathcal{V}$; $S = \bigcup_{V_i \in \mathcal{S}} V_i$; $V_i, V_j \notin \mathcal{S}$; and $x \in V_i$ and $y \in V_j$. If V_i and V_j belong to different connected components of $(G/\mathcal{V}) - \mathcal{S}$, then x and y belong to different connected components of $G - S$.*

2.2. Graph separators. Given an assignment $w : V \to \mathbb{R}^+$ of weights to the vertices of a graph $G = (V, E)$ and a parameter $0 < t < 1$, we call a set $S \subseteq V$ of vertices a *t-vertex separator* of G if no connected component of $G - S$ has weight greater than $tw(G)$, where $w(G) = \sum_{x \in V} w(x)$ is the weight of G. Similarly, a *t-edge separator* of G is a set $S \subseteq E$ of edges so that no connected component of the graph $G - S$ has weight exceeding $tw(G)$. Since we are mainly interested in vertex separators in this paper, we refer to them simply as *separators*. If G is unweighted, we give every vertex of G weight one and define separators w.r.t. these weights.

In our separator algorithm for unweighted graphs, we apply the following result to a compressed version of the graph we want to partition. This produces a first separator that is then refined during a sequence of refinement steps.

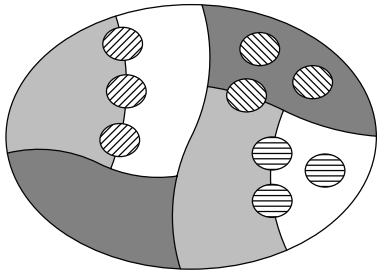

FIG. 2.1. *A regular partition. The solid regions are connected. Each of the hatched regions is disconnected but shares its boundary with at most two solid regions and no hatched region.*

THEOREM 2.1 (Aleksandrov and Djidjev [2]). *Given a planar graph G of size N, an assignment $w : V \to \mathbb{R}^+$ of weights to the vertices of G, and a constant $0 < t < 1$, a t-vertex separator of size at most $4\sqrt{N/t}$ for G can be computed in linear time.*

By grouping the connected components of $G-S$, where G is an unweighted graph and S is a t-separator of G, we obtain a tN-partition of G. More precisely, let $r > 0$ be an integer, let $t = r/N$, and let S be a t-separator of G. Then an *r-partition* of G is a pair $\mathcal{P} = (S, \{G_1, \ldots, G_q\})$ with the following properties:

(i) $G_1, \ldots, G_q$ are vertex-induced subgraphs of G;
(ii) the set $\{V(G_1), \ldots, V(G_q)\}$ is a partition of $V(G) \setminus S$;
(iii) $\mathcal{N}(V(G_i)) \subseteq S$ for all $1 \le i \le q$; and
(iv) $|G_i| \le r$ for all $1 \le i \le q$.

The third condition captures that every subgraph G_i is a collection of connected components of $G - S$; that is, no connected component of $G - S$ has vertices in two such subgraphs.

If G is a planar graph, we call an r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_q\})$ *normal* if $|S| = \mathrm{O}(N/\sqrt{r})$, $q = \mathrm{O}(N/r)$, and $\sum_{i=1}^{q} |\mathcal{N}(G_i)| = \mathrm{O}(N/\sqrt{r})$. A normal r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_q\})$ is *c-proper* for some constant $c > 0$ if $|\mathcal{N}(G_i)| \le c\sqrt{r}$ for all $1 \le i \le q$. If we do not want to specify c, we say that $\mathcal{P}$ is *proper*. A proper r-partition is stronger than a normal r-partition because the latter may contain subgraphs with a large boundary, as long as the total size of all subgraph boundaries is small; in the former, every individual subgraph has to have a small boundary. Finally, we call an r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_q\})$ *regular* if, for every $1 \le i \le q$, one of the following conditions holds:

(i) The graph $\mathcal{N}[G_i]$ is connected, or
(ii) there are at most two indices $1 \le j < k \le q$, $i \notin \{j, k\}$, such that $\mathcal{N}(V(G_i)) \cap \mathcal{N}(V(G_j)) \neq \emptyset$ and $\mathcal{N}(V(G_i)) \cap \mathcal{N}(V(G_k)) \neq \emptyset$. $\mathcal{N}[G_j]$ and $\mathcal{N}[G_k]$ are connected in this case.

This concept is visualized in Figure 2.1, which is reproduced from Frederickson [19], who shows that every planar graph has a proper r-partition and that every planar graph of bounded degree has a regular proper r-partition. The following result states that a proper r-partition of a planar graph can be obtained efficiently.

THEOREM 2.2 (Frederickson [19]). *Given a planar graph G of size N and a normal r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_p\})$ of G, a proper r-partition $\mathcal{P}' = (S', \{G'_1, \ldots, G'_q\})$ of G such that $S \subseteq S'$ can be computed in $\mathrm{O}(N \log N)$ time.*

Frederickson also shows how to obtain a normal r-partition of a planar graph in $\mathrm{O}(N \log N)$ time. Together with Theorem 2.2, this implies that a proper r-partition of a planar graph can be computed in $\mathrm{O}(N \log N)$ time. To prove Theorem 2.2, Fred-

erickson considers each graph G_i in turn and partitions it into subgraphs of boundary size $O(\sqrt{r})$. When partitioning G_i, the algorithm requires knowledge of only $\mathcal{N}[G_i]$. Thus, if $|\mathcal{N}[G_i]| \leq M$ for all $1 \leq i \leq p$, we can implement Frederickson's procedure by loading each graph $\mathcal{N}[G_i]$ into internal memory and partitioning G_i without incurring any further I/Os. Assuming that each graph $\mathcal{N}[G_i]$ is stored in consecutive memory locations, we thus have the following.

COROLLARY 2.3. *Given a planar graph G of size N and a normal r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_p\})$ of G, a proper r-partition $\mathcal{P}' = (S', \{G'_1, \ldots, G'_q\})$ of G such that $S \subseteq S'$ can be computed in $O(N/B)$ I/Os, provided that $|\mathcal{N}[G_i]| \leq M$ for all $1 \leq i \leq p$.*

2.3. Bipartite planar graphs. A graph $G = (V, E)$ is *bipartite* if the vertex set V can be partitioned into two sets U and W such that $x \in U$ and $y \in W$ for every edge $xy \in E$. In this case, we write $G = (U, W, E)$. We use the following two simple results to bound the sizes of certain bipartite planar graphs in different parts of our algorithms.

LEMMA 2.4. *Let $G = (U, W, E)$ be a simple connected bipartite planar graph such that every vertex in W has degree at least three. Then $|W| < 2|U|$.*

Proof. Consider a planar embedding $\hat{G}$ of G. Since G is bipartite, every face of $\hat{G}$ has size at least four and, thus, $|F| \leq |E|/2$. By Euler's formula, $|V| + |F| - |E| = 2$, that is,

$$\begin{aligned} 2 &= |V| + |F| - |E| \\ &\leq |V| - |E|/2, \\ |E| &< 2|V|. \end{aligned}$$

On the other hand, $|E| \geq 3|W|$, so that

$$\begin{aligned} 3|W| &< 2|V| \\ &= 2(|U| + |W|), \\ |W| &< 2|U|. \end{aligned}$$ □

COROLLARY 2.5. *Let $G = (U, W, E)$ be a simple connected bipartite planar graph such that no two vertices $x, y \in W$ of degree at most two have the same open neighborhood. Then $|G| < 7|U|$.*

Proof. We have to prove that $|W| < 6|U|$. To this end, we divide W into three subsets and bound the size of each of these sets. Let W_1 be the set of degree-1 vertices in W, W_2 the set of degree-2 vertices in W, and W_3 the set of vertices of degree at least three in W, that is, $W_3 = W \setminus (W_1 \cup W_2)$.

Since there are no two vertices $x, y \in W$ with $\deg(x) = \deg(y) = 1$ and $\mathcal{N}(x) = \mathcal{N}(y)$, W_1 contains at most $|U|$ vertices, that is, one per vertex in U; see Figure 2.2(a).

To count the vertices in W_3, we consider the bipartite planar graph $G_3 = (U_3, W_3, E_3)$ induced by all edges incident to vertices in W_3; see Figure 2.2(b). By Lemma 2.4 and since $U_3 \subseteq U$, we have $|W_3| < 2|U_3| \leq 2|U|$.

To count the vertices in W_2, consider the graph $H_2 = (U_2, E_2)$, where $U_2 = \mathcal{N}(W_2) \subseteq U$ and $E_2 = \{xz :$ there exists a vertex $y \in W_2$ with $\mathcal{N}(y) = \{x, z\}\}$; see Figure 2.2(d). Since there are no two vertices $x, y \in W_2$ with $\mathcal{N}(x) = \mathcal{N}(y)$, H_2 contains exactly one edge per vertex in W_2, that is, $|E_2| = |W_2|$. Next we argue that H_2 is planar, which, by Euler's formula, implies that $|W_2| = |E_2| < 3|U_2| \leq 3|U|$.

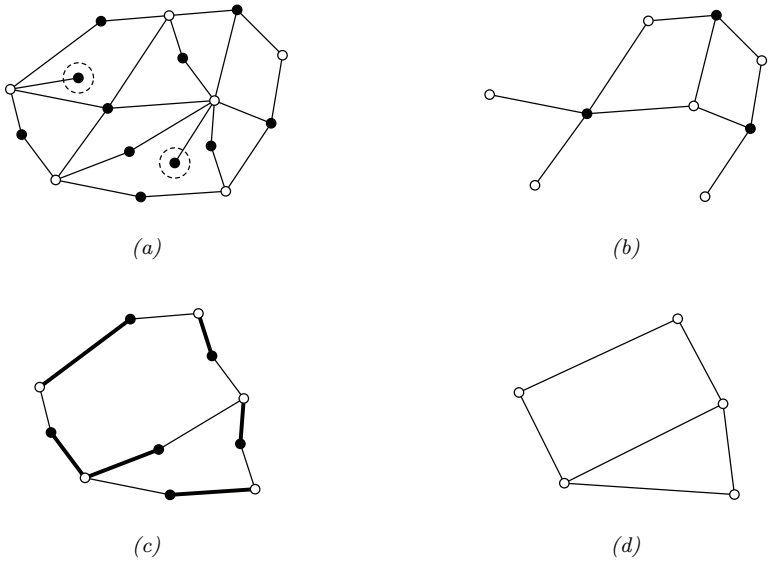

FIG. 2.2. *Illustration of the proof of Corollary* 2.5. (a) *A bipartite planar graph* $G = (U, W, E)$*; the white vertices are in* U*, and the black ones are in* W*. The circled vertices are in* W_1*.* (b) *The subgraph* G_3 *of* G *induced by all edges incident to vertices in* W_3*.* (c) *The subgraph* G_2 *of* G *induced by all edges incident to vertices in* W_2*.* (d) *The graph* H_2 *obtained from* G_2 *by contracting the bold edges in* (c).

To see that H_2 is planar, consider the subgraph G_2 of G induced by all edges incident to vertices in W_2; see Figure 2.2(c). Since G is planar, G_2 is planar. H_2 can be obtained from G_2 by contracting one of the edges incident to each vertex $y \in W_2$. By Fact 2.1, this implies that H_2 is planar.

In summary, we have $|W| = |W_1| + |W_2| + |W_3| < |U| + 3|U| + 2|U| = 6|U|$, that is, $|G| = |U| + |W| < 7|U|$. □

2.4. Primitive operations. Our algorithms make frequent use of a number of primitive operations. In order to avoid repeatedly discussing their implementations, we discuss them here and then refer to this section whenever we make use of such an operation.

Set operations. All elementary operations on two sets A and B—union, intersection, difference, etc.—can be carried out in $O(\mathrm{sort}(N))$ I/Os if the two sets A and B are represented as unordered sequences of elements sort A and B (assuming that every element is represented by a unique integer ID). Then scan the two sorted lists to produce $C = A \odot B$, where $\odot \in \{\cap, \cup, \setminus\}$.

Even though it is not a set operation as such, we want to mention duplicate removal here. Some operations may produce multisets. In order to obtain a proper representation of the set of elements in such a multiset, we need to remove duplicates. This can be done in $O(\mathrm{sort}(N))$ I/Os by sorting and scanning the multiset.

Copying labels. Since pointers are mostly useless in I/O-efficient graph algorithms, graphs are often represented as an (unsorted) list of vertices, each with a unique vertex ID, and as an (unsorted) list of edges, each labeled with the IDs of its two endpoints. Edges do not store any pointers to their endpoints. Thus, if certain labels are assigned to the vertices of the graph, the edges have no knowledge of the labels of their endpoints. However, it takes $O(\mathrm{sort}(N))$ I/Os to label all edges with the labels of

their endpoints. Sort the vertices by their IDs. Designate one endpoint of every edge as being the first endpoint. Then sort the edges by the IDs of their first endpoints. Now scan the sorted vertex and edge sets to label every edge with the label of its first endpoint. To label the edges with the labels of their second endpoints, sort the edges by the IDs of their second endpoints, and repeat the copying process.

Graph contraction. Given a labeling of the vertices of graph G that represents a partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the vertex set of G, that is, assigns the same label to two vertices if and only if they belong to the same set $V_i \in \mathcal{V}$, the graph $G/\mathcal{V}$ can be computed in $O(\mathrm{sort}(N))$ I/Os as follows: First, label the edges of G with the labels of their endpoints. Now replace every vertex with its label and every edge xy with the edge ab, where a is the label of x and b is the label of y. Finally, remove duplicates from the resulting vertex and edge sets.

In this paper, we choose the label of a set $V_i \in \mathcal{V}$ to be the ID of a *representative* $x \in V_i$. We will then often refer to the vertex V_i in $G/\mathcal{V}$ as the vertex x, taking the point of view that x has survived the contraction and that all other vertices in V_i have been *contracted into* x.

Connected components. Chiang et al. [14] proved that it takes $O(\mathrm{sort}(N))$ I/Os to compute the connected components of an N-vertex planar graph G, that is, to compute a labeling of the vertices of G such that two vertices have the same label if and only if they are connected by a path in G.

3. Uniform graph contraction. In this section, we discuss a general contraction procedure for planar graphs that is used repeatedly in our algorithms. In general, when using graph contraction, one repeatedly contracts edges until some goal is reached (usually a sufficient reduction of the size of the graph). Since it would not be I/O-efficient to contract edges one at a time, I/O-efficient algorithms based on graph contraction usually contract many edges simultaneously. More precisely, these algorithms compute an edge contraction $G/\mathcal{V}$ of G such that $|\mathcal{V}| \leq c|V|$, for some $c < 1$; that is, $G/\mathcal{V}$ has only a constant fraction of the vertices of G. The algorithms of [5, 14, 30] for computing connected components and minimum spanning trees are based on exactly this idea. However, the partition $\mathcal{V}$ used in these algorithms is nonuniform in the sense that some sets $V_i \in \mathcal{V}$ may be large, while others may be small; that is, some vertices in $G/\mathcal{V}$ represent many vertices in G, and others represent only few. As we will see, our separator algorithm for unweighted graphs relies heavily on the compression being uniform, that is, on every set $V_i \in \mathcal{V}$ having constant size. Our goal in this section is to compute a partition $\mathcal{V}$ such that $|\mathcal{V}| \leq c|V|$ and that every set in $\mathcal{V}$ has constant size. In order to achieve this, we compute $G/\mathcal{V}$ in two phases: First, we compute an edge contraction $G_1 = G/\mathcal{V}_1$ and then a vertex bundling $G_2 = G_1/\mathcal{V}_2$.

In section 3.1, we give a high-level description of our uniform contraction procedure and prove a general bound on the size of the compressed graph it produces. In section 3.2, we show how to implement this procedure in $O(\mathrm{sort}(N))$ I/Os on an N-vertex planar graph.

3.1. The high-level procedure. Our separator algorithm requires not only that $\mathcal{V}$ be a partition of V into subsets of constant size, but also that each subset be of bounded weight according to appropriately chosen weights assigned to the vertices in V. In this section, we assume more generally that the input to our contraction procedure consists of a planar graph G, a constant number of real-valued functions $w_1, \ldots, w_k$ assigning weights to the vertices of G, and a set of weight thresholds $u_1, \ldots, u_k$. Initially, every vertex x satisfies $w_j(x) \leq u_j$ for all $1 \leq j \leq k$; we say that vertex x is *within bounds*. Our goal is to compute a contraction $G/\mathcal{V}$ of G such

that each set V_i in $\mathcal{V}$ is within bounds, that is, $w_j(V_i) = \sum_{x \in V_i} w_j(x) \leq u_j$ for all $1 \leq j \leq k$. We say that a vertex or set x that is within bounds is *light* if $w_j(x) \leq u_j/2$ for all $1 \leq j \leq k$; otherwise, we call it *heavy*. The main result of this section is stated in the following theorem.

THEOREM 3.1. *Let G be a planar graph, $w_1, \ldots, w_k$ real-valued functions assigning weights to the vertices of G, and $u_1, \ldots, u_k$ weight thresholds such that every vertex $x \in G$ satisfies $w_j(x) \leq u_j$ for all $1 \leq j \leq k$. Then it takes $\mathrm{O}(\mathrm{sort}(N))$ I/Os to compute a planar graph G_2 such that G_2 is a contraction of G, all vertices of G_2 are within bounds, and more than $|G_2|/7$ vertices in G_2 are heavy.*

The contraction procedure consists of two phases: The *edge contraction phase* computes an edge contraction $G_1 = G/\mathcal{V}_1$ of G such that all vertices of G_1 are within bounds and every edge of G_1 has at least one heavy endpoint. The *bundling phase* computes a vertex bundling $G_2 = G_1/\mathcal{V}_2$ of G_1 such that all vertices of G_2 are within bounds and there are no two light vertices $x, y \in G_2$ of degree at most two that have the same open neighborhood. Graph G_2 is the final graph we return.

In section 3.2, we prove that the two phases of this procedure can be implemented in $\mathrm{O}(\mathrm{sort}(N))$ I/Os. Here we prove that G_2 has the properties claimed in Theorem 3.1.

By the description of the contraction procedure, the vertices of G_2 are explicitly ensured to be within bounds. Graph G_1 is an edge contraction of G, and G_2 is a vertex bundling of G_1. Hence, G_2 is a contraction of G and, by Fact 2.1, planar. We have to prove that at least every seventh vertex in G_2 is heavy.

LEMMA 3.2. *If h is the number of heavy vertices in G_2, then $|G_2| < 7h$.*

Proof. Consider the subgraph G' of G_2 induced by all edges incident to light vertices in G_2. Since G' contains all light vertices of G_2, it suffices to prove that $|G'| < 7h'$, where $h' \leq h$ is the number of heavy vertices in G'. We use Corollary 2.5 to do so.

First, observe that G' is bipartite: By the definition of G', every edge in G' has at least one light endpoint. If there was an edge with two light endpoints in $G' \subseteq G_2$, then G_1 would have to contain an edge with two light endpoints because G_2 is a vertex bundling of G_1, but we ensure explicitly that G_1 contains no such edge.

Observe also that no two light vertices of degree at most two in G_2 have the same neighbors. Hence, the same is true in G', and G' satisfies the conditions of Corollary 2.5 with U being the set of heavy vertices and W being the set of light vertices in G'. Thus, by Corollary 2.5, $|G'| < 7h'$. ∎

3.2. I/O-efficient implementation. In this section, we discuss how to implement the two phases of the contraction procedure in $\mathrm{O}(\mathrm{sort}(N))$ I/Os.

3.2.1. Edge contraction phase. To implement the edge contraction phase, we compute a set F of edges and define $\mathcal{V}_1$ to be the partition of V corresponding to the connected components of (V, F). We use $\mathcal{V}_F$ to denote this partition. We choose the edges in F so that the graph (V, F) is a forest, which makes the computation of connected components and, thus, the computation of the partition $\mathcal{V}_F$, easy [14].

The set F is not necessarily a subset of E. However, whenever we add an edge xy to F, there exists an edge $x'y' \in E$ such that x and x' belong to the same set V_1, and y and y' belong to the same set V_2 in $\mathcal{V}_F$. Hence, if $G[V_1]$ and $G[V_2]$ are connected, so is $G[V_1 \cup V_2]$, and the addition of edge xy to F maintains that $G/\mathcal{V}_F$ is an edge contraction of G.

We compute the edge set F iteratively, starting with $F = \emptyset$. Each iteration computes a set F' of edges such that $G/\mathcal{V}_{F \cup F'}$ is an edge contraction of G and each of its vertices is within bounds. The edges in F' are then added to F. This iterative

process stops as soon as every edge in $G/\mathcal{V}_F$ has at least one heavy endpoint. At this point, we define $G_1 = G/\mathcal{V}_F$.

To compute the set F' efficiently, each iteration consists of three steps: The first step extracts the subgraph H_1 of $G/\mathcal{V}_F$ induced by all edges in $G/\mathcal{V}_F$ whose endpoints are both light. We call these edges *contractible* and call H_1 the *contractible subgraph* of $G/\mathcal{V}_F$. The second step computes a maximal matching F_1' of H_1 and contracts the edges in F_1', producing the graph $H_2 = H_1/\mathcal{V}_{F_1'}$. We call a vertex in H_2 *matched* if it represents the two endpoints of an edge in F_1'; otherwise, the vertex represents only one vertex in H_1, and we call it *unmatched*. Note that all neighbors of an unmatched vertex are matched because F_1' is maximal. The third step adds edges between matched and unmatched vertices to a set F_2' so that every nonsingleton set in $\mathcal{V}_{F_2'}$ contains exactly one matched vertex. To determine which edges to add to F_2', we inspect each unmatched vertex x in turn. If there is a matched neighbor y of x that is contained in a light set V_y in $\mathcal{V}_{F_2'}$, we add the edge xy to F_2', thereby adding x to V_y. After this third step, we define $F' = F_1' \cup F_2'$ and add the edges in F' to F. This finishes the iteration.

Note that testing the loop condition—whether or not $G/\mathcal{V}_F$ contains two adjacent light vertices—is easy because, by the definition of H_1, this is the case if and only if H_1 is nonempty. It is also unnecessary to compute $G/\mathcal{V}_F$ from scratch after each iteration: Each iteration is interested only in the contractible subgraph of $G/\mathcal{V}_F$. If H_1 and H_1' are the contractible subgraphs of $G/\mathcal{V}_F$ and $G/\mathcal{V}_{F\cup F'}$, respectively, it is easy to see that $H_1' \subseteq H_1/\mathcal{V}_{F'} = H_2/\mathcal{V}_{F_2'}$. Thus, each iteration has to compute only $H_2 = H_1/\mathcal{V}_{F_1'}$ and $H = H_2/\mathcal{V}_{F_2'}$, and the next iteration can extract its contractible subgraph from H. This leads to the contraction procedure shown in Algorithm 1.

Before providing the implementation details and analyzing the I/O-complexity of this procedure, we show that it correctly implements the edge contraction phase of our uniform graph contraction procedure.

LEMMA 3.3. *Let G_1 be the graph produced by procedure* CONTRACTEDGES. *Then G_1 is an edge contraction of G, every vertex of G_1 is within bounds, and every edge of G_1 has at least one heavy endpoint.*

Proof. We prove by induction on the number of iterations that the graph $G/\mathcal{V}_F$ is an edge contraction of G and that all its vertices are within bounds. Since we exit with $G_1 = G/\mathcal{V}_F$ only when every edge in $G/\mathcal{V}_F$ has a heavy endpoint, this proves the lemma.

Before the first iteration, $G/\mathcal{V}_F = G$ is a trivial edge contraction of G because $F = \emptyset$. Moreover, all vertices of G are assumed to be within bounds.

So assume that, before the current iteration, $G/\mathcal{V}_F$ is an edge contraction of G and all its vertices are within bounds. We want to prove that the same is true for $G/\mathcal{V}_{F\cup F'}$.

First, we prove that $G/\mathcal{V}_{F\cup F'}$ is an edge contraction of G. Since $G/\mathcal{V}_F$ is an edge contraction of G, every set in $\mathcal{V}_F$ induces a connected subgraph of G. Every edge xy added to F_1' is an edge of $H_1 \subseteq G/\mathcal{V}_F$. Hence, there exist two sets $V_1, V_2 \in \mathcal{V}_F$ and an edge $x'y' \in E(G)$ such that $x, x' \in V_1$ and $y, y' \in V_2$. By adding edge xy to F_1', the sets V_1 and V_2 are merged into the set $V_1 \cup V_2$ in $\mathcal{V}_{F\cup F_1'}$. Since $G[V_1]$ and $G[V_2]$ are connected, the existence of edge $x'y' \in E(G)$ implies that $G[V_1 \cup V_2]$ is connected, and $G/\mathcal{V}_{F\cup F_1'}$ remains an edge contraction of G after adding edge xy to F_1'.

As for Step 3, every edge added to F_2' is an edge of $H_2 = H_1/\mathcal{V}_{F_1'} \subseteq G/\mathcal{V}_{F\cup F_1'}$. Since we have just argued that $G/\mathcal{V}_{F\cup F_1'}$ is an edge contraction of G, the same argument as in the previous paragraph implies that $G/\mathcal{V}_{F\cup F_1'\cup F_2'}$ is an edge contraction of G.

Algorithm 1 The edge contraction phase.

Procedure ContractEdges

Input: A weighted graph $G = (V, E)$ and a set of weight thresholds based on which every vertex in G is classified as light or heavy.

Output: An edge contraction $G_1 = G/\mathcal{V}_F$ of G.

1: $H \leftarrow G$
2: $F \leftarrow \emptyset$
3: **while** H is not empty **do**
4: Step 1: Extract the contractible subgraph
$H_1 \leftarrow$ the contractible subgraph of H
5: Step 2: Contract a maximal matching
Compute a maximal matching F_1' of H_1
$H_2 \leftarrow H_1/\mathcal{V}_{F_1'}$
6: Step 3: Contract edges incident to unmatched vertices
$F_2' \leftarrow \emptyset$
Let $\mathcal{V}_{F_2'}$ be the partition of $V(H_2)$ defined by F_2'.
for every unmatched vertex $x \in H_2$ **do**
if x has a (matched) neighbor y contained in a light set $V_y \in \mathcal{V}_{F_2'}$ **then**
Add edge xy to F_2' and increase the weights of V_y by the corresponding weights of x.
end if
end for
$F \leftarrow F \cup F_1' \cup F_2'$
$H \leftarrow H_2/\mathcal{V}_{F_2'}$
7: **end while**
8: $G_1 \leftarrow G/\mathcal{V}_F$

To see that all vertices of $G/\mathcal{V}_{F\cup F'}$ are within bounds, we first observe that all vertices of $G/\mathcal{V}_{F\cup F_1'}$ are within bounds. Indeed, each vertex $x \in G/\mathcal{V}_{F\cup F_1'}$ represents one or two vertices in $G/\mathcal{V}_F$. In the former case, x is obviously within bounds. In the latter case, the two vertices represented by x both belong to H_1 and are thus light; hence, x is within bounds in this case as well.

In the third step, when adding an edge xy to F_2', x is an unmatched vertex of H_2, that is, represents a single vertex in H_1 and is thus light. The set V_y containing y is light because otherwise we would not add edge xy to F_2'. Thus, after adding xy to F_2', and thereby x to V_y, the vertex in $G/\mathcal{V}_{F\cup F_1'\cup F_2'}$ representing V_y remains within bounds. Arguing inductively over all edges added to F_2' in Step 3, we obtain that all vertices in $G/\mathcal{V}_{F\cup F_1'\cup F_2'}$ are within bounds. This finishes the proof of the lemma. ∎

Procedure ContractEdges is fairly easy to implement in O(sort(N)) I/Os. To prove this, we show how to implement every iteration of the while-loop in lines 3–7 in O(sort($|H|$)) I/Os, show that $|H|$ decreases by a factor of at least two from one iteration to the next, and that the rest of the algorithm takes O(sort(N)) I/Os. We start by analyzing the I/O-complexity of one iteration of the while-loop.

Extracting the contractible subgraph. The contractible subgraph of H can be extracted in O(sort($|H|$)) I/Os: First, we label all edges with the weights of their endpoints, and then we scan the edge set to discard all edges that have at least one heavy

endpoint. The result is the edge list of the contractible subgraph. Now we scan this edge list and create a list of the edges' endpoints. To produce the vertex list of H_1, we remove duplicates from the resulting list. As discussed in section 2.4, all steps of this construction take $O(\text{sort}(|H|))$ I/Os.

Computing and contracting a matching. Zeh [33] presents an algorithm that computes a maximal matching of a graph $G = (V, E)$ in $O(\text{sort}(N))$ I/Os, where $N = |V| + |E|$. Since H_1 is planar, we have $N = O(|H_1|)$ when applying this procedure to H_1. Hence, the matching can be computed in $O(\text{sort}(|H_1|))$ I/Os. Once the matching is given, its edges can be contracted in $O(\text{sort}(|H_1|))$ I/Os by using the contraction procedure from section 2.4.

Contracting edges between matched and unmatched vertices. To contract edges between matched and unmatched vertices, that is, to implement Step 3 of the iteration, we start by extracting all edges in H_2 that are incident to unmatched vertices. This is easily done in $O(\text{sort}(|H_2|)) = O(\text{sort}(|H_1|))$ I/Os by labeling all edges with the statuses of their endpoints and discarding all edges that have two matched endpoints. (Recall that every edge in H_2 has at least one matched endpoint.)

For the resulting bipartite graph H', we compute the following information: Let V_m be the set of matched vertices, and let V_u be the set of unmatched vertices in H'. We arbitrarily number the vertices in V_m as $y_1, \ldots, y_r$ and the vertices in V_u as $x_1, \ldots, x_s$. We sort the edges in H' by their unmatched endpoints as primary keys and by their matched endpoints as secondary keys. For every edge $x_i y_j$, we store the unmatched endpoint $x_{i'}$ of the next edge incident to y_j; that is, if y_j is adjacent to vertices $x_{i_1}, \ldots, x_{i_t}$ with $i_1 < \cdots < i_t$, then, for $1 \le h < t$, edge $x_{i_h} y_j$ stores the ID of vertex $x_{i_{h+1}}$. Edge $x_{i_t} y_j$ stores **nil** to indicate that it is the last edge incident to y_j.

This information can easily be computed in $O(\text{sort}(|H'|))$ I/Os: We sort the edges by their matched endpoints as primary keys and by their unmatched endpoints as secondary keys. Then we scan the sorted edge list to compute, for every edge $x_i y_j$, the endpoint of the next edge incident to y_j. (If the next edge in the sorted sequence is $x_{i'} y_j$, then this endpoint is $x_{i'}$. If the next edge is $x_{i'} y_{j'}$ with $j' \neq j$, then edge $x_i y_j$ stores **nil**.) Now we sort the edges by their unmatched endpoints as primary keys and by their matched endpoints as secondary keys.

Given that graph H' has been prepared in this manner, we find the edges in F_2' as follows: We scan the sorted edge list, which is equivalent to inspecting the unmatched vertices in sorted order and scanning their adjacency lists. For every vertex x_i, as soon as we find an edge $x_i y_j$ such that the set V_{y_j} is light, we add edge $x_i y_j$ to F_2' and increase each weight $w_h(V_{y_j})$ by the corresponding weight $w_h(x_i)$ of x_i. The remaining edges in x_i's adjacency list are then ignored. If all neighbors of x_i are contained in heavy sets, no edge incident to x_i is added to F_2'.

In order to implement this procedure correctly, we need a mechanism to inform every edge incident to a vertex y_j about the current weights of the set V_{y_j} at the time when this edge is inspected. We use a priority queue Q to do this. Initially, we have $w_h(V_{y_j}) = w_h(y_j)$ for all h, because $F_2' = \emptyset$ and, hence, $V_{y_j} = \{y_j\}$. For every matched vertex y_j, we insert the initial weights of V_{y_j} into Q, with priority equal to the ID of the first edge incident to y_j. For every edge $x_i y_j$ inspected during the scan, we perform a Delete-Min operation on Q to retrieve the current weights of V_{y_j}. If edge $x_i y_j$ is added to F_2', the weights of V_{y_j} are updated; otherwise, they remain unchanged. Then, if edge $x_i y_j$ stores a vertex $x_{i'} \neq$ **nil** as the next unmatched vertex incident to y_j, we insert the current weights of V_{y_j} into Q, but now with priority $x_{i'} y_j$.

The whole procedure requires one scan of the sorted edge list of H', and two

priority queue operations per edge, once the edges of H' are stored in the right order and store the appropriate successor pointers as defined above. If we use a buffer tree [4] to implement the priority queue, every priority queue operation takes $O((1/B)\log_{M/B}(r/B)) = O((1/B)\log_{M/B}(|H'|/B))$ I/Os amortized. Thus, the construction of F_2' takes $O(\mathrm{sort}(|H'|)) = O(\mathrm{sort}(|H_1|))$ I/Os.

Total cost of the loop. The previous three paragraphs establish that each iteration of the while-loop takes $O(\mathrm{sort}(|H|))$ I/Os. Also observe that $|H| = N$ before the first iteration. To prove that the cost of the whole loop is $O(\mathrm{sort}(N))$ I/Os, it is therefore sufficient to show that $|H|$ decreases by a factor of at least two from one iteration to the next.

LEMMA 3.4. *The loop in lines* 3–7 *of procedure* CONTRACTEDGES *takes* $O(\mathrm{sort}(N))$ *I/Os.*

Proof. Assume that there are k iterations. For $1 \le i \le k$, let $H^{(i)}$ and $H_1^{(i)}$ be snapshots of H and H_1 at the beginning of the ith iteration and after Step 1 of the ith iteration, respectively. Then the total cost of Step 1 over all iterations is $O\big(\sum_{i=1}^{k} \mathrm{sort}\big(\big|H^{(i)}\big|\big)\big)$, and the cost of Steps 2 and 3 over all iterations is $O\big(\sum_{i=1}^{k} \mathrm{sort}\big(\big|H_1^{(i)}\big|\big)\big)$. It is easy to see that $\big|H^{(i+1)}\big| \le \big|H_1^{(i)}\big|$ for all $1 \le i < k$. Hence, the total cost of all iterations is $O\big(\mathrm{sort}\big(\big|H^{(1)}\big|\big) + \sum_{i=1}^{k} \mathrm{sort}\big(\big|H_1^{(i)}\big|\big)\big)$, which is $O(\mathrm{sort}(N) + \sum_{i=1}^{k} \mathrm{sort}(|H_1^{(i)}|))$ because $H^{(1)} = G$. Next we prove that, for $1 \le i < k$, $\big|H_1^{(i+1)}\big| \le \big|H_1^{(i)}\big|/2$. Hence, $\sum_{i=1}^{k}\big|H_1^{(i)}\big| \le 2\big|H_1^{(1)}\big| \le 2N$, and the total cost of the loop is $O(\mathrm{sort}(N))$ I/Os.

Consider the ith iteration. After Step 2, there are at most $|H_1^{(i)}|/2$ matched vertices in H_2 because each represents two vertices in $H_1^{(i)}$. To bound the number of vertices in $H_1^{(i+1)}$ by $|H_1^{(i)}|/2$, we argue that every vertex in $H_1^{(i+1)}$ represents a set V' in $\mathcal{V}_{F_2'}$ that contains a matched vertex. In particular, we argue that every vertex in $H^{(i+1)} = H_2/\mathcal{V}_{F_2'}$ that represents a singleton set in $\mathcal{V}_{F_2'}$ containing only an unmatched vertex has only heavy neighbors and, thus, does not belong to the contractible subgraph of $H^{(i+1)}$. To see that this is true, observe that we inspect every unmatched vertex x in H_2 to check whether it has a matched neighbor y whose containing set $V_y \in \mathcal{V}_{F_2'}$ is light; if so, we add the edge xy to F_2', thereby adding x to V_y. Thus, if x remains in a singleton set, all its neighbors in $H^{(i+1)}$ are heavy. This finishes the proof. ∎

I/O-complexity of the edge contraction phase. To complete the analysis, we have to consider the costs of lines 1, 2, and 8 of the algorithm. Lines 1 and 2 are easy to implement in $O(N/B)$ I/Os. Line 8 requires computing the connected components of the graph $G_F = (V, F)$. This can be done in $O(\mathrm{sort}(N))$ I/Os (see section 2.4). The connected components algorithm identifies components by labeling every vertex in a connected component of G_F with a representative. As argued in section 2.4, the graph $G/\mathcal{V}_F$ can then be computed in $O(\mathrm{sort}(N))$ I/Os. Together with Lemmas 3.3 and 3.4, this proves the following.

LEMMA 3.5. *The edge contraction phase of the uniform graph contraction procedure takes* $O(\mathrm{sort}(N))$ *I/Os.*

3.2.2. Bundling phase. The bundling phase assumes that no two light vertices in the input graph G_1 are adjacent. By Lemma 3.3, this is true for the graph produced by the edge contraction phase. We first extract all light vertices of degree at most two and represent each such vertex x as a triple $(x, y_1, \mathbf{nil})$ or (x, y_1, y_2), depending on whether it has one or two (heavy) neighbors. We sort these triples by their last two

components, thereby storing all vertices with the same neighbors consecutively. Then we scan this sorted list; that is, we scan each group C of vertices with the same neighbors. For each such group, we form subgroups, starting with the first vertex in C. For every subgroup, we record its total weights $W_1, \ldots, W_k$, which are the sums of the corresponding weights of the vertices in the group. For every inspected vertex x, we add it to the current group if $W_h \leq u_h/2$ for all $1 \leq h \leq k$. Otherwise, vertex x starts a new group. Groups are represented by labeling every vertex in a group with the ID of the first vertex in the group. The final grouping represents the partition $\mathcal{V}_2$ of $V(G_1)$, and we compute the graph $G_2 = G_1/\mathcal{V}_2$ using the graph contraction procedure from section 2.4.

LEMMA 3.6. *Given an N-vertex planar graph G_1 that has no two adjacent light vertices and all of whose vertices are within bounds, the bundling phase takes* $\mathrm{O}(\mathrm{sort}(N))$ *I/Os and produces a vertex bundling $G_2 = G_1/\mathcal{V}_2$ of G_1 that contains no two light vertices of degree at most two that have the same open neighborhood. All vertices in G_2 are within bounds.*

Proof. The I/O-bound of the procedure is obvious because we sort and scan the vertex and edge sets of G_1 a constant number of times and then apply the contraction procedure from section 2.4. It is also obvious that $G_1/\mathcal{V}_2$ is a vertex bundling because we add two vertices to the same set in $\mathcal{V}_2$ only if they have the same neighbors.

Next assume that G_2 contains two light vertices x and y of degree at most two and such that $\mathcal{N}(x) = \mathcal{N}(y)$. Then x and y are the representatives of two sets V_1 and V_2 in $\mathcal{V}_2$ that are subsets of the set C of all vertices with neighborhood $\mathcal{N}(x)$, and both sets are light. Assume w.l.o.g. that V_1 is formed before V_2, and let z be the first vertex in C that is not in V_1. Since V_1 is light, z would have been added to V_1, a contradiction.

Now assume that G_2 contains a vertex x that is out of bounds. Since all vertices of G_1 are within bounds, x must represent some set V_1 formed by collecting vertices that belong to a set C of light vertices with the same neighbors. Let y be the last vertex in C that is added to V_1. Since we add y to V_1, V_1 is light before the addition of y. Since y is light, this implies that V_1 remains within bounds after adding y to it, a contradiction. ∎

Lemmas 3.5 and 3.6 together establish the I/O-complexity of the uniform graph contraction procedure claimed in Theorem 3.1 and, thus, finish the proof of Theorem 3.1.

4. An algorithm for partitioning unweighted graphs. In this section, we present our main result: an I/O-efficient algorithm for computing a proper r-partition of an unweighted planar graph G. In particular, we prove the following.

THEOREM 4.1. *Given a planar graph G and an integer $r > 0$, a proper r-partition of G can be computed in* $\mathrm{O}(\mathrm{sort}(N))$ *I/Os, provided that $M \geq 56r \log^2 B$.*

Our algorithm (Algorithm 2) consists of three steps. The first two compute a separator S_0 of size $\mathrm{O}(N/\sqrt{r})$ that defines an $\left(r \log^2 B\right)$-partition of G. The last step then refines this partition to an r-partition by adding at most $\mathrm{O}(N/\sqrt{r})$ more vertices to the separator. The reason for not computing an r-partition immediately in the first two steps is that Step 2 consists of $\log B$ iterations, each of which adds $\mathrm{O}(N/\sqrt{r'})$ vertices to the separator if an r'-partition is desired. By choosing $r' = r \log^2 B$, we ensure that the total number of separator vertices chosen in the first two steps is $\mathrm{O}(N/\sqrt{r})$, and the refinement in Step 3 increases this number by only a constant factor.

Algorithm 2 Computing a separator for an unweighted planar graph.

Procedure SEPARATOR

Input: A planar graph $G = (V, E)$ and an integer $r > 0$.
Output: A proper r-partition $\mathcal{P} = (S, \mathcal{R})$ of G.

1: $\ell \leftarrow \lfloor \log B \rfloor - 1$

2: Step 1: Compute the graph hierarchy

$G_0 \leftarrow G$
for $i = 1, \ldots, \ell$ **do**
Compute a graph G_i that satisfies properties (GH3)–(GH6) from G_{i-1}.
end for

3: Step 2: Compute the separator S_0

Apply Theorem 2.1 to compute a separator $S_\ell \subseteq V(G_\ell)$ of size $O(|G_\ell|/(\sqrt{r} \log B))$ and whose removal partitions G_ℓ into connected components of size at most $r \log^2 B$.
for $i = \ell - 1, \ldots, 0$ **do**
Compute a separator S_i for G_i. Separator S_i consists of two sets S_i' and S_i''. S_i' is the set of vertices represented by the vertices in S_{i+1}. S_i'' is the set of separator vertices introduced in order to partition the connected components of $G_i - S_i'$ into subgraphs of size at most $r \log^2 B$.
end for

4: Step 3: Compute the final partition

Compute the partition $\mathcal{P} = (S, \mathcal{R})$ by dividing the connected components of $G - S_0$ into $O(N/r)$ subgraphs of size at most r and boundary size $O(\sqrt{r})$ and adding the required separator vertices to S_0.

Step 3 is easy to implement I/O-efficiently: Given the assumption that $M \geq 56r \log^2 B$, every connected component of $G - S_0$ fits in internal memory. Hence, we can compute the desired r-partition by loading each component of $G - S_0$ into memory and applying Theorem 2.1 to it without incurring any further I/Os.

In Steps 1 and 2, we apply the same idea iteratively. In Step 1, we construct a hierarchy of $\ell = \lfloor \log B \rfloor - 1$ planar graphs $G_0, \ldots, G_\ell$, where $G_0 = G$ and $|G_\ell| = O(N/B)$. We obtain each graph G_i from the previous graph G_{i-1} using the uniform graph contraction procedure from section 3. Given the reduced size of G_ℓ, we can compute an $\big(r \log^2 B\big)$-partition of G_ℓ in $O(N/B)$ I/Os using Theorem 2.1. Let S_ℓ be the set of separator vertices used in this partition. In Step 2, we undo the contraction steps that produced graphs $G_1, \ldots, G_\ell$ from G_0, one graph at a time. In each iteration, we derive a separator S_i for G_i from the separator S_{i+1} computed for G_{i+1} in the previous iteration. We do this as follows: Since G_{i+1} is obtained from G_i using the uniform graph contraction procedure, every vertex in G_{i+1} represents a set of vertices in G_i. Let S_i' be the set of vertices in G_i represented by the vertices in S_{i+1}. In order to obtain an $\big(r \log^2 B\big)$-partition of G_i, we partition every component of $G_i - S_i'$ into subgraphs of size at most $r \log^2 B$ and add the separator vertices used in this partition to a set S_i''. The separator S_i is the union of sets S_i' and S_i''. S_0 is the separator of $G_0 = G$ obtained at the end of this process.

In order to carry out Step 2 efficiently, and in order to obtain a small separator S_0 at the end of Step 2, the graph hierarchy $G_0, \ldots, G_\ell$ computed in Step 1 has to satisfy a number of properties.

First, we want every graph G_{i+1} to have roughly half as many vertices as G_i. This guarantees that the cost of Steps 1 and 2 is dominated by the cost of the computation on $G_0 = G$ and, since $\ell = \lfloor \log B \rfloor - 1$, that G_ℓ has size $O(N/B)$.

Second, during the construction of the separator S_i from S'_i, we would like to use an internal-memory algorithm to partition each connected component of $G_i - S'_i$; that is, we want each such component to fit in memory. Since S_{i+1} defines an $\left(r \log^2 B\right)$-partition of G_{i+1} and $M \geq 56r \log^2 B$, every connected component of $G_i - S'_i$ fits in memory if every vertex in G_{i+1} represents at most 56 vertices in G_i.

Finally, observe that, once we add a vertex x in G_i to S_i, all vertices in G represented by x belong to S_0. Thus, to guarantee that S_0 is small, we have to ensure that no vertex in G_i represents too many vertices of G.

These conditions are formalized in the following properties which we require the graph hierarchy $G_0, \ldots, G_\ell$ to have:

(GH1) $\ell = \lfloor \log B \rfloor - 1$,
(GH2) $G = G_0$,
(GH3) For $i = 0, \ldots, \ell$, G_i is planar,
(GH4) For $i = 0, \ldots, \ell$, $|G_i| \leq 7N/2^{i-1}$,
(GH5) For $i = 1, \ldots, \ell$, every vertex in G_i represents at most 56 vertices in G_{i-1}, and
(GH6) For $i = 0, \ldots, \ell$, every vertex in G_i represents at most 2^{i+1} vertices in G.

In section 4.1, we show how to compute a graph hierarchy with these properties in $O(\mathrm{sort}(N))$ I/Os. In section 4.2, we show that the desired separator S_0 of G can be computed from this graph hierarchy in $O(\mathrm{sort}(N))$ I/Os. In section 4.3, we provide the details of Step 3 and show that this step can be carried out in the same I/O bound as Steps 1 and 2, thereby establishing the complexity of Algorithm 2 claimed in Theorem 4.1.

4.1. The graph hierarchy. The first step of our algorithm is the computation of a hierarchy of graphs $G_0, \ldots, G_\ell$ that satisfy properties (GH1)–(GH6). We start by setting $G_0 = G$. Then we compute each graph G_i from the previous graph G_{i-1} by using the uniform graph contraction procedure from section 3. To ensure properties (GH5) and (GH6), we assign a *weight* $w(x)$ and a *size* $s(x)$ to every vertex in G_{i-1}; the former is equal to the number of vertices in G represented by x, and the latter is equal to the number of vertices in G_{i-1} represented by x, that is, equal to 1. We then pass a weight threshold $u_w = 2^{i+1}$ and a size threshold $u_s = 56$ to the contraction procedure.

Note that the assignment of weights and sizes to the vertices of G_{i-1} before the construction of G_i is easily accomplished. Initially, we set $w(x) = s(x) = 1$ for every vertex x in $G_0 = G$. Subsequently, before computing G_i from G_{i-1}, the size of every vertex in G_{i-1} can be reset to 1 in a single scan over the vertex set of G_{i-1}; as a result of the construction of G_{i-1} from G_{i-2}, every vertex in G_{i-1} already stores its correct weight.

LEMMA 4.2. *A graph hierarchy $G_0, \ldots, G_q$ with properties* (GH1)–(GH6) *can be constructed in* $O(\mathrm{sort}(N))$ *I/Os.*

Proof. First, we prove that the graph hierarchy computed by the procedure we have just described has the desired properties. Properties (GH1) and (GH2) are trivially satisfied. Property (GH3) is easy to prove by induction: For $i = 0$, $G_0 = G$ and is thus planar. For $i > 0$, the planarity of G_i follows from the planarity of G_{i-1} because, by Theorem 3.1, the uniform contraction procedure preserves planarity.

Properties (GH5) and (GH6) are also easy to show by induction. In particular,

$w(x) = 1 \leq 2$ for every vertex $x \in G_0$, and property (GH5) holds vacuously in this case. When constructing G_i from G_{i-1}, every vertex in G_{i-1} is within bounds because, by the induction hypothesis, it has weight at most 2^i and size 1 (after resetting its size to 1). Hence, by Theorem 3.1, every vertex in G_i is within bounds and, thus, has weight at most 2^{i+1} and size at most 56.

It remains to show property (GH4). For graph G_0, property (GH4) holds because $|G_0| = |G| = N \leq 7N/2^{-1}$. To prove the claim for graphs $G_1, \ldots, G_\ell$, let h_i be the number of heavy vertices in G_i. By Theorem 3.1, each graph G_i has size less than $7h_i$. To prove property (GH4), it is therefore sufficient to prove that $h_i \leq N/2^{i-1}$.

We prove this claim by induction. We partition the heavy vertices into two categories. A heavy vertex of type I has weight exceeding 2^i. A type-II vertex has weight at most 2^i and size greater than 28. Graph G_i contains less than $N/2^i$ type-I vertices and less than $|G_{i-1}|/28$ type-II vertices; that is, $h_i < N/2^i + |G_{i-1}|/28$.

For $i = 1$, we obtain $h_1 < N/2 + N/28 < N/2^0$. For $i > 1$, we obtain

$$h_i < \frac{N}{2^i} + \frac{|G_{i-1}|}{28} \tag{1}$$

$$\leq \frac{N}{2^i} + \frac{h_{i-1}}{4} \tag{2}$$

$$\leq \frac{N}{2^i} + \frac{N}{2^i} \tag{3}$$

$$= \frac{N}{2^{i-1}}. \tag{4}$$

Equation (2) follows from (1) because $|G_{i-1}| \leq 7h_{i-1}$, as argued above. Equation (3) follows from (2) by the induction hypothesis.

To bound the I/O-complexity, we recall that, by Theorem 3.1, the construction of graph G_i from graph G_{i-1} takes $\mathrm{O}(\mathrm{sort}(|G_{i-1}|))$ I/Os. By property (GH4), we have $\sum_{i=0}^{\ell} |G_i| = \mathrm{O}(N)$. Thus, the total I/O-complexity is $\sum_{i=1}^{\ell} \mathrm{O}(\mathrm{sort}(|G_{i-1}|)) = \mathrm{O}(\mathrm{sort}(N))$. ∎

4.2. The separator hierarchy. In Step 2 of Algorithm 2, we use the graph hierarchy computed in the first step to construct a relatively coarse partition of G. In particular, we compute a separator S_0 of size $\mathrm{O}(N/\sqrt{r})$ whose removal partitions G into connected components of size at most $r \log^2 B$.

First, we compute a partition of G_ℓ into subgraphs of size at most $r \log^2 B$. To do so, we use an arbitrary linear-time planar embedding algorithm (see, e.g., [11]) to compute a planar embedding of G_ℓ, and then we apply Theorem 2.1 to compute the desired partition. Let $S_\ell = S''_\ell$ be the computed separator.

In the loop in Step 2, we apply the following iterative strategy to compute separators $S_{\ell-1}, \ldots, S_0$ for graphs $G_{\ell-1}, \ldots, G_0$: Given the separator S_{i+1} computed for graph G_{i+1} in the previous iteration, we construct the set S'_i of vertices in G_i represented by the vertices in S_{i+1}. Then we apply Theorem 2.1 to each connected component of $G_i - S'_i$ whose size exceeds $r \log^2 B$, in order to partition it into subgraphs of size at most $r \log^2 B$. Let S''_i be the set of separator vertices introduced by partitioning the connected components of $G_i - S'_i$ in this manner. Then $S_i = S'_i \cup S''_i$.

LEMMA 4.3. *The separator S_0 of G computed in Step 2 of Algorithm 2 has size $\mathrm{O}(N/\sqrt{r})$. The connected components of $G - S_0$ have size at most $r \log^2 B$.*

Proof. The bound on the size of the connected components of $G - S_0$ is explicitly guaranteed by the construction. We bound the size of S_0 as follows: For every vertex

$x \in G_i$, let $R_0(x)$ be the set of vertices in G represented by x. From our computation of S_0 it follows that, for every vertex $y \in S_0$, there exists a unique set S_i'' and a unique vertex $x \in S_i''$ such that $y \in R_0(x)$. Hence, we have

$$|S_0| = \sum_{i=0}^{\ell} \sum_{x \in S_i''} |R_0(x)|.$$

By property (GH6), we have $|R_0(x)| = w(x) \leq 2^{i+1}$ for all $x \in S_i''$. Hence,

$$|S_0| \leq \sum_{i=0}^{\ell} 2^{i+1} |S_i''|.$$

Since we compute S_i'' by applying Theorem 2.1 to disjoint subgraphs of G_i, partitioning each into pieces of size at most $r \log^2 B$, we have $|S_i''| \leq 4|G_i|/(\sqrt{r} \log B)$. By property (GH4), this implies that $|S_i''| \leq 28N/(2^{i-1}\sqrt{r} \log B)$. Thus,

$$|S_0| \leq \sum_{i=0}^{\ell} 2^{i+1} \frac{28N}{2^{i-1}\sqrt{r} \log B} = \sum_{i=0}^{\ell} \frac{112N}{\sqrt{r} \log B} = \frac{112N}{\sqrt{r}}. \qquad \square$$

LEMMA 4.4. *Step* 2 *of Algorithm* 2 *takes* $O(\mathrm{sort}(N))$ *I/Os to compute the separator* S_0*, provided that* $M \geq 56r \log^2 B$.

Proof. The computation of the separator S_ℓ takes $O(N/B)$ I/Os by Theorem 2.1 and because graph G_ℓ has size at most $7N/2^{\ell-1} = O(N/B)$. Since the sizes of graphs $G_0, \ldots, G_\ell$ are geometrically decreasing, it suffices to show that the separator S_i can be constructed from S_{i+1} in $O(\mathrm{sort}(|G_i|))$ I/Os. This implies then that Step 2 takes $O(\mathrm{sort}(|G_0|)) = O(\mathrm{sort}(N))$ I/Os.

Since the uniform graph contraction procedure labels every vertex x in G_i with the vertex in G_{i+1} representing x, we can compute the separator S_i' in $O(\mathrm{sort}(|G_i|))$ I/Os: First, we sort the vertices in S_{i+1} by their IDs, and then we sort the vertices in G_i by their representatives in G_{i+1}. Then we scan the two lists and mark all those vertices in G_i as being in S_i' whose representatives in G_{i+1} belong to S_{i+1}. Now it takes $O(\mathrm{sort}(|G_i|))$ I/Os to compute the connected components of $G_i - S_i'$ (see section 2.4).

Since every connected component of $G_{i+1} - S_{i+1}$ has size at most $r \log^2 B$, it follows from property (GH5) and Fact 2.2 that every connected component of $G_i - S_i'$ has size at most $56r \log^2 B \leq M$; that is, each such component fits in memory. Thus, we can load each connected component of $G_i - S_i'$ whose size exceeds $r \log^2 B$ into memory and apply Theorem 2.1 to partition it into connected components of size at most $r \log^2 B$. As the computation of Theorem 2.1 is carried out in internal memory, partitioning the connected components of $G_i - S_i'$ into subgraphs of size at most $r \log^2 B$ takes $O(|G_i|/B)$ I/Os. Thus, the computation of S_i takes $O(\mathrm{sort}(|G_i|))$ I/Os, and the total I/O-bound follows. $\square$

4.3. Computing the final partition. In order to obtain the final partition in Step 3 of Algorithm 2, we have to partition the connected components of $G - S_0$ into subgraphs of size at most r. We also have to merge subgraphs to reduce their number to $O(N/r)$, while maintaining the bounds on their size and boundary size. We do this as follows: First, we use Theorem 2.1 to partition each connected component of $G - S_0$ into pieces of size at most r. This adds $O(N/\sqrt{r})$ vertices to the separator and, thus, increases the separator size by only a constant factor. The resulting partition may

contain more than $O(N/r)$ subgraphs, and the total boundary size of its subgraphs may exceed $O(N/\sqrt{r})$. We group the subgraphs in the current partition to reduce their number to $O(N/r)$ and their total boundary size to $O(N/\sqrt{r})$. Finally, we partition each subgraph in the resulting partition into subgraphs of boundary size $O(\sqrt{r})$. This is similar to Frederickson's approach [19] and, as argued below, increases both the number of subgraphs in the partition and the total boundary size by only a constant factor.

Since every connected component of $G - S_0$ has size at most $r \log^2 B \leq M$, the partition of $G - S_0$ into connected components of size at most r can be computed by loading each connected component of $G - S_0$ into internal memory and applying Theorem 2.1 to it. Let $S' \supseteq S_0$ be the separator produced by this step. Section 4.3.1 discusses how to obtain a normal r-partition $\mathcal{P}' = (S', \{G'_1, \ldots, G'_r\})$ of G from S'. Section 4.3.2 then refines this partition to make it proper.

4.3.1. Grouping components. Intuitively, we compute the partition $\mathcal{P}'$ in two phases: The first phase groups connected components of boundary size at most two with other components that have the same boundary, while ensuring that none of the resulting subgraphs has size greater than r. This phase reduces the total boundary size of all subgraphs to $O(N/\sqrt{r})$. The second phase merges subgraphs that share boundary vertices until no two subgraphs sharing boundary vertices can be merged without producing a subgraph of size greater than r. This reduces the number of subgraphs to $O(N/r)$. Note that merging subgraphs in this manner cannot increase the total boundary size; that is, the total boundary size of all subgraphs in the partition remains $O(N/\sqrt{r})$, and the resulting partition $\mathcal{P}'$ is normal.

To determine which connected components to group in these two phases, we use an auxiliary graph H_0 whose vertices represent separator vertices and connected components of $G - S'$. Both phases operate on H_0, grouping component vertices rather than actual components. After the two phases have been applied to H_0, we obtain $\mathcal{P}'$ by merging the components in the partition that correspond to merged component vertices in H_0.

Graph H_0 contains all vertices in S' and one vertex per connected component of $G - S'$. There is an edge between two separator vertices in H_0 if such an edge exists in G. There is an edge between a separator vertex x and a component vertex representing a component G' of $G - S'$ if $x \in \mathcal{N}(G')$. Finally, every vertex in H_0 has a weight equal to the number of vertices in G it represents.

Graph H_0 is easily constructed from G and S' in $O(\mathrm{sort}(N))$ I/Os: First, we compute the connected components of $G - S'$, thereby labeling every vertex in $G - S'$ with the ID of the component it belongs to; we also label every vertex in S' with its own ID. Then we apply the contraction procedure from section 2.4 to G.

Reducing the total boundary size. Merging components of boundary size at most two that have the same boundary is equivalent to merging component vertices in H_0 that have the same neighbors and degree at most two. The latter is easily achieved using the uniform graph contraction procedure (in fact, only the bundling phase is sufficient). For the purpose of applying this procedure, we change the weight of every separator vertex to r, leave the weights of all component vertices unchanged, and set the weight threshold to r. Then the edge contraction phase does nothing because every edge of H_0 has at least one endpoint that is a separator vertex, that is, is heavy. The bundling phase merges light component vertices that have the same neighbors and degree at most two. By Theorem 3.1, the application of the uniform contraction procedure takes $O(\mathrm{sort}(|H_0|)) = O(\mathrm{sort}(N))$ I/Os. The next lemma proves that this

produces a graph H_1 from H_0 whose component vertices represent subgraphs of $G-S'$ of size at most r each and sufficiently small total boundary size. Note that the bound on the size of H_1 stated in the lemma implies the claimed bound on the boundary size of the corresponding partition of G because the total boundary size is equal to the number of edges between component vertices and separator vertices in H_1. Since H_1 is planar and has size $O(N/\sqrt{r})$, there are only $O(N/\sqrt{r})$ edges in H_1.

LEMMA 4.5. *Applying the uniform contraction procedure to H_0 produces a planar graph H_1 of size $O(N/\sqrt{r})$. Every vertex in H_1 has weight at most r.*

Proof. Since H_0 is an edge contraction of G, Fact 2.1 implies that H_0 is planar. By Theorem 3.1, this implies that H_1 is planar. Before applying the contraction procedure to H_0, all vertices are within bounds, that is, have weight at most r. Hence, by Theorem 3.1, every vertex in H_1 has weight at most r. Finally, to bound the size of H_1, observe that H_1 contains only $O(N/\sqrt{r})$ heavy vertices: $O(N/\sqrt{r})$ separator vertices and $O(N/r)$ heavy component vertices; the latter is true because the total weight of all component vertices in H_1 is at most N. By Theorem 3.1, this implies that H_1 has $O(N/\sqrt{r})$ vertices. ∎

Reducing the number of subgraphs. To reduce the size of H_1 to $O(N/r)$ by further merging vertices, we first reset the weight of every separator vertex to 1 and then apply the uniform contraction procedure to H_1, again with weight threshold r. Since $|H_1| \leq N$, this takes $O(\mathrm{sort}(N))$ I/Os by Theorem 3.1. Since a vertex is heavy if its weight exceeds $r/2$, and the total weight of all vertices in the resulting graph H_2 is N, there are at most $2N/r$ heavy vertices in H_2. By Theorem 3.1, this implies that the total size of H_2 is $O(N/r)$. Moreover, since no vertex in H_1 has weight exceeding r, Theorem 3.1 implies that no vertex in H_2 has weight exceeding r. Thus, we have the following.

LEMMA 4.6. *Applying the uniform contraction procedure to H_1 produces a planar graph H_2 of size $O(N/r)$. Every vertex in H_2 has weight at most r.*

The final grouping. Every component vertex in H_2 now represents a subgraph in the partition $\mathcal{P}'$. We finish the computation of $\mathcal{P}'$ by labeling every nonseparator vertex with the ID of the subgraph it belongs to. This is easily achieved by sorting and scanning the vertex sets of G, H_0, H_1, and H_2 a constant number of times. Indeed, every vertex in $G - S'$ is initially labeled with the connected component of $G - S'$ that contains it, that is, with its representative in H_0. Similarly, every vertex in H_0 is labeled with its representative in H_1, and every vertex in H_1 is labeled with its representative in H_2. Thus, sorting and scanning suffices to label every vertex in H_1, and subsequently every vertex in H_0 and G, with its representative in H_2. This labeling represents the subgraphs in $\mathcal{P}'$.

LEMMA 4.7. *Given the separator S_0, a normal r-partition $\mathcal{P}'$ of G can be computed in $O(\mathrm{sort}(N))$ I/Os.*

Proof. The I/O-bound of computing $\mathcal{P}'$ from S_0 follows from the above discussion of the different steps required to obtain $\mathcal{P}'$ from G and S_0.

There are only $O(N/r)$ subgraphs in $\mathcal{P}'$ because there are only $O(N/r)$ component vertices in H_2 and each of them defines one subgraph in $\mathcal{P}'$. Every subgraph in the partition has size equal to the weight of its representative in H_2; by Lemma 4.6, no vertex in H_2 has weight exceeding r. Finally, by Lemma 4.5, there are only $O(N/\sqrt{r})$ edges in H_1. Each such edge represents an adjacency between a separator vertex and a subgraph in the partition $\mathcal{P}''$ of G represented by H_1. Thus, $\mathcal{P}''$ has total boundary size $O(N/\sqrt{r})$. Since partition $\mathcal{P}'$ is obtained by merging subgraphs in $\mathcal{P}''$, the total boundary size of the subgraphs in $\mathcal{P}'$ cannot be greater than the total boundary size of the subgraphs in $\mathcal{P}''$. ∎

4.3.2. Ensuring small boundary size. In order to obtain a proper r-partition from the normal r-partition $\mathcal{P}'$, we have to reduce the boundary size of each individual subgraph to $O(\sqrt{r})$ by further partitioning each subgraph in $\mathcal{P}'$ whose boundary size exceeds this bound. In order to do so, we apply Corollary 2.3. This, however, requires that each graph $\mathcal{N}[G'_i]$ fit in memory. While $|G'_i| \leq r \leq M$, the graph $\mathcal{N}[G'_i]$ may be big and may not fit in memory.

We solve this problem by first augmenting the separator so that its size increases by only a constant factor, and every graph G''_i in the resulting partition satisfies $|\mathcal{N}[G''_i]| = O(r) \leq M$. This then allows us to apply Corollary 2.3 to obtain the final partition.

To augment the separator, we consider each graph G'_i in the partition $\mathcal{P}'$ in turn. Let $\tilde{G}_i$ be the graph obtained from $\mathcal{N}[G'_i]$ as follows: First, we remove all edges between vertices in $\mathcal{N}(G'_i)$. Then we merge all vertices in $\mathcal{N}(G'_i)$ whose resulting degree is at most 2 that have the same set of neighbors (which all belong to G'_i). For every vertex in $\tilde{G}_i$ that represents more than one vertex in $\mathcal{N}(G'_i)$, we add its neighbors in G'_i to a set S''_i. Then we define $\mathcal{P}'' = (S'', \{G''_1, \ldots, G''_p\})$, where $G''_i = G[V(G'_i) \setminus S''_i]$ and $S'' = S' \cup \bigcup_{i=1}^{p} S''_i$; that is, in $\mathcal{P}''$, the vertices in S''_i are removed from the graph G'_i and are added to the separator.

LEMMA 4.8. *Let $\mathcal{P}'' = (S'', \{G''_1, \ldots, G''_p\})$ be the partition obtained from $\mathcal{P}'$ using the above transformation. Then $\mathcal{P}''$ is normal, and every graph $\mathcal{N}[G''_i]$, $1 \leq i \leq p$, has size $O(r)$.*

Proof. Let $\mathcal{P}' = (S', \{G'_1, \ldots, G'_p\})$. Then $|S'| = O(N/\sqrt{r})$ and $\sum_{i=1}^{p} |\mathcal{N}(G_i)| = O(N/\sqrt{r})$ because $\mathcal{P}'$ is normal. Now consider subgraphs G'_i and G''_i, and let $\mathcal{N}_{\text{sh}}(G'_i) = \mathcal{N}(G'_i) \cap \mathcal{N}(G''_i)$ and $\mathcal{N}_{\text{dis}}(G'_i) = \mathcal{N}(G'_i) \setminus \mathcal{N}(G''_i)$.

We start by proving that $|S''_i| \leq |\mathcal{N}_{\text{dis}}(G'_i)|$. Consider a vertex $x \in S''_i$. This vertex belongs to S''_i because there exists a vertex $y \in \tilde{G}_i$ that represents a set V_y of $h \geq 2$ vertices in $\mathcal{N}(G'_i)$ that are adjacent to x. We charge each such vertex $1/h \leq 1/2$ for the addition of x to S''_i. Since each vertex in V_y is adjacent to at most two vertices in G'_i, each vertex in V_y is charged for at most two vertices, and the charge to each vertex is at most 1. Thus, the total number of charged vertices is an upper bound on $|S''_i|$. Note, however, that all neighbors of a charged vertex that belong to G'_i are added to S''_i. Hence, every charged vertex belongs to $\mathcal{N}_{\text{dis}}(G'_i)$ and $|S''_i| \leq |\mathcal{N}_{\text{dis}}(G'_i)|$. This immediately implies that partition $\mathcal{P}''$ is normal: The size of S'' is $|S''| = |S'| + \sum_{i=1}^{p} |S''_i| \leq |S'| + \sum_{i=1}^{p} |\mathcal{N}_{\text{dis}}(G'_i)| \leq |S'| + \sum_{i=1}^{p} |\mathcal{N}(G'_i)| = O(N/\sqrt{r})$. For each graph G''_i, we have $|\mathcal{N}(G''_i)| \leq |\mathcal{N}_{\text{sh}}(G'_i)| + |S''_i| \leq |\mathcal{N}_{\text{sh}}(G'_i)| + |\mathcal{N}_{\text{dis}}(G'_i)| = |\mathcal{N}(G'_i)|$. Therefore, $\sum_{i=1}^{p} |\mathcal{N}(G''_i)| \leq \sum_{i=1}^{p} |\mathcal{N}(G'_i)| = O(N/\sqrt{r})$.

To bound the size of each graph $\mathcal{N}[G''_i]$, we partition its vertices into two groups: those in G'_i and those in $\mathcal{N}_{\text{sh}}(G'_i)$. Since $|G'_i| \leq r$, $\mathcal{N}[G''_i]$ can contain at most r vertices that belong to G'_i. Next, observe that there are no two vertices x and y of degree at most two in $\mathcal{N}_{\text{sh}}(G'_i)$ such that $\mathcal{N}(x) = \mathcal{N}(y)$: if there were two such vertices, their neighbors in G'_i would have been added to S''_i. Thus, the subgraph of G induced by the edges between vertices in $\mathcal{N}_{\text{sh}}(G'_i)$ and vertices in G'_i satisfies the conditions of Corollary 2.5, and the number of vertices in $\mathcal{N}_{\text{sh}}(G'_i)$ is less than $6|G'_i| \leq 6r$. The size of $\mathcal{N}[G''_i]$ is therefore at most $7r$. ∎

The computation of partition $\mathcal{P}''$ takes $O(\text{sort}(N))$ I/Os. The computation of graph $\tilde{G}_i$, for every graph $\mathcal{N}[G'_i]$, is easily carried out using the uniform graph contraction procedure after assigning weight 1 to every separator vertex and weight $2N$ to every vertex in G'_i; the weight threshold is $2N$. The construction of set S''_i now requires scanning the vertex set of $\tilde{G}_i$ and adding the neighbors of all separator vertices

of weight at least two and degree at most two to S_i''. Thus, the computation for each graph $\mathcal{N}[G_i']$ takes $O(\text{sort}(|\mathcal{N}[G_i']|))$ I/Os, and the total cost is $\sum_{i=1}^{p} O(\text{sort}(|\mathcal{N}[G_i']|))= O(\text{sort}(N))$.

To obtain the final partition, we apply Theorem 2.2 to every subgraph G_i'' in partition $\mathcal{P}''$ whose boundary size exceeds $c\sqrt{r}$, for an appropriate constant $c > 0$. By Corollary 2.3, this can be done in $O(N/B)$ I/Os and increases the size of the separator and the number of graphs in the partition by only a constant factor; that is, this final step produces a proper r-partition of G.

Since we have shown that all three steps of Algorithm 2 can be carried out in $O(\text{sort}(N))$ I/Os and that the final partition we obtain is proper, we have thus shown Theorem 4.1.

5. Regular partitions. Our result from the previous section provides an algorithm for computing proper r-partitions of planar graphs, as long as r is small; but the computed partitions are not necessarily regular. In general, without a bound on the degrees of the vertices in the graph, a regular proper r-partition may not exist. For planar graphs of degree three, however, regular proper r-partitions do exist [19], and the algorithms of [5, 7, 8, 9] rely on the existence of such partitions. In this section, we show how to modify the partition produced by Algorithm 2 to obtain a regular proper r-partition for a planar graph of degree three.

Given such a graph G, we use Algorithm 2 to compute a proper r-partition $\mathcal{P} = (S, \{G_1, \ldots, G_p\})$ of G. We are, however, interested only in the separator S and discard the grouping of the connected components $G - S$ into subgraphs. Next we regroup the connected components of $G - S$ to obtain the desired regular proper r-partition $\mathcal{P}' = (S, \{G_1', \ldots, G_q'\})$. This grouping is again similar to [19].

We use an auxiliary graph H to compute the desired grouping. Graph H contains one vertex per connected component of $G - S$. There is an edge between two vertices in H if the two corresponding components of $G - S$ share a boundary vertex. Since every vertex in G has degree at most three, graph H is planar. We give two weights $s(x)$ and $b(x)$ to each vertex x in H: $s(x)$ is the size, that is, the number of vertices in the connected component represented by x; $b(x)$ is the size of the component's boundary. Note that $s(x) \leq r$ and $b(x) \leq c\sqrt{r}$, for some $c > 0$, because $\mathcal{P}$ is a proper r-partition.

Now we apply the uniform graph contraction procedure to H, with thresholds $u_s = r$ and $u_b = c\sqrt{r}$. This produces a graph H', each of whose vertices represents a set of vertices in H and, thus, a set of connected components of $G - S$. The graphs $G_1', \ldots, G_q'$ in partition $\mathcal{P}'$ are defined as the graphs represented by the vertices in H'. By the arguments in section 4.3, this partition of $G - S$ can be computed in $O(\text{sort}(N))$ I/Os.

THEOREM 5.1. *Given an N-vertex planar graph G none of whose vertices has degree greater than three, and an integer $r > 0$, a regular proper r-partition of G can be computed in $O(\text{sort}(N))$ I/Os, provided that $M \geq 56r \log^2 B$.*

Proof. The I/O-complexity of the procedure follows immediately from Theorem 4.1 and our discussion above. Next, we argue that the produced partition $\mathcal{P}'$ is proper. Since the partition $\mathcal{P}$ produced by Algorithm 2 is proper, we have $|S| = O(N/\sqrt{r})$. This implies that the total boundary size of the connected components of $G - S$ cannot exceed $O(N/\sqrt{r})$ because every vertex in G has degree at most three, and, thus, every vertex in S is adjacent to at most three connected components of $G - S$. Therefore, there are only $O(N/r)$ heavy vertices in H': at most $2N/r$ vertices of size greater than $r/2$ and $O(N/r)$ vertices of boundary size greater than $c\sqrt{r}/2$.

By Theorem 3.1, this implies that $|H'| = O(N/r)$, that is, that partition $\mathcal{P}'$ has $O(N/r)$ subgraphs. Since every vertex x in H has weights $s(x) \leq r$ and $b(x) \leq c\sqrt{r}$, Theorem 3.1 implies that the same is true for every vertex in H'. Thus, no subgraph in partition $\mathcal{P}'$ has size exceeding r or boundary size exceeding $c\sqrt{r}$, and partition $\mathcal{P}'$ is proper.

In order to prove that $\mathcal{P}'$ is regular, we analyze the two phases of the computation of H' from H. The edge contraction phase of the uniform contraction procedure merges only vertices that are adjacent. In G, this corresponds to merging connected components of $G - S$ that share boundary vertices. Thus, for every graph G_i'' in the resulting partition, the graph $\mathcal{N}[G_i'']$ is connected. The bundling phase merges only vertices of degree at most two that have the same neighbors. Moreover, these neighbors are heavy, that is, are not merged with any other vertices during the bundling phase. Thus, the merging of vertices during the bundling phase corresponds to producing merged subgraphs that are potentially disconnected but share boundary vertices with at most two other subgraphs G_j' and G_k', which satisfy that $\mathcal{N}[G_j']$ and $\mathcal{N}[G_k']$ are connected. □

6. Low-cost separators and edge separators. In this section, we show that the results from sections 4 and 5 can be used to obtain an I/O-efficient version of the following theorem by Aleksandrov et al. [3].

THEOREM 6.1 (Aleksandrov et al. [3]). *Given a planar graph $G = (V, E)$, a cost function $c : V \to \mathbb{R}^+$, a weight function $w : V \to \mathbb{R}^+$, and a real number $0 < t < 1$, there exists a t-vertex separator S of cost $c(S) \leq 4\sqrt{2C(G)/t}$ for G, where $c(S) = \sum_{x \in S} c(x)$ and $C(G) = \sum_{x \in V} (c(x))^2$. Such a separator can be computed in linear time.*

In this theorem, the sizes of the subgraphs in the computed partition are measured in terms of the total *weight* assigned to their vertices by a weight function w. Similarly, the size of the separator S is measured in terms of the total *cost* assigned to the vertices in S by a cost function c. The former is a fairly standard notion already considered in the classical paper by Lipton and Tarjan [27]. The latter is a more recent concept that allows a number of separator theorems to be seen as special cases of Theorem 6.1. Theorem 2.1, for example, can be obtained from Theorem 6.1 by choosing $c(x) = 1$ for all $x \in V$, and Aleksandrov et al. have shown how to obtain a generalization of the edge separator theorem of [15] from Theorem 6.1 (see also Theorem 6.3 in section 6.3). The main result of this section is as follows.

THEOREM 6.2. *Given a planar graph $G = (V, E)$, a cost function $c : V \to \mathbb{R}^+$, and a weight function $w : V \to \mathbb{R}^+$, a separator S as in Theorem* 6.1 *can be computed in* $O(\text{sort}(N))$ *I/Os, provided that* $M = \Omega\big(B^2 \log^2 B\big)$.

Theorem 6.2 is more general than Theorem 4.1 because it takes vertex costs and weights into account; moreover, even in the unweighted case, Theorem 4.1 requires that $r = tN = O\big(M/\log^2 B\big)$, while Theorem 6.2 places no such restriction on r.

Our exposition of the algorithm that proves Theorem 6.2 is organized as follows. In section 6.1, we review the algorithm by Aleksandrov et al., as it forms the basis for our I/O-efficient version. In section 6.2, we provide I/O-efficient implementations of the three main steps of this algorithm, thereby obtaining an I/O-efficient version of the algorithm. This proves Theorem 6.2. In section 6.3, we briefly argue that this also leads to an I/O-efficient version of the edge separator algorithm of [3].

We assume throughout sections 6.1 and 6.2 that the given graph is triangulated; since any planar graph can be triangulated in $O(\text{sort}(N))$ I/Os [25], and a separator

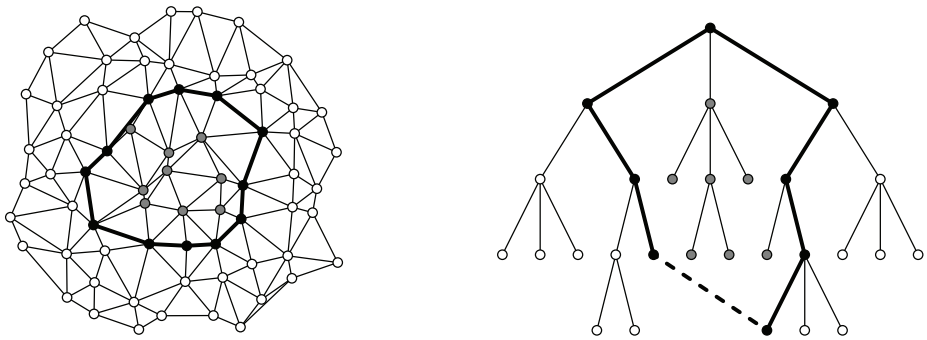

FIG. 6.1. (a) *The bold cycle separates the white vertices on its outside from the grey ones on its inside.* (b) *The dashed nontree edge defines the bold fundamental cycle.*

of the resulting triangulation is also a separator of the original graph, this is not a restriction.

6.1. The algorithm of Aleksandrov et al. The algorithm of [3] can be seen as a nontrivial extension of Lipton and Tarjan's algorithm [27]. The first observation is that every simple cycle C in G separates the vertices inside C from those outside C in the given embedding of G; see Figure 6.1(a). The central goal then is to compute a collection of cycles that partition G into regions of the desired weight. These cycles are chosen from the set of *fundamental cycles* w.r.t. a spanning tree T of G, where a fundamental cycle $F(e)$ consists of an edge $e \in E(G) \setminus E(T)$ and the path in T connecting the two endpoints of e; see Figure 6.1(b). We refer to an edge $e \in E(G) \setminus E(T)$ as a *nontree edge*, while every edge in T is a *tree edge*.

In order to obtain a separator of low cost in this manner, it is necessary to bound the number of fundamental cycles comprising the separator, as well as the total cost of the vertices on each fundamental cycle. As we will see below, the former is easy, as the number of required cycles is inversely proportional to t. To ensure that each fundamental cycle is of low cost, Aleksandrov et al. compute T as a shortest-path tree w.r.t. appropriate edge weights that guarantee that the depth of every vertex x in T (that is, the weighted distance of x from the root of T) is equal to the total cost of the vertices on the path from the root of T to x; the cost of any fundamental cycle is then at most twice the *radius* of T, where the radius of the tree is the maximum depth of any of its vertices. By itself, this does not yet guarantee low cost of each fundamental cycle because T may have a large radius. To fix this, Aleksandrov et al. first find a set of vertices of low total cost whose removal partitions T into subgraphs $G_0, \ldots, G_p$ of low depth, where the *depth* of a graph G_i is the maximal difference between the depths (in T) of any two vertices in G_i; see Figure 6.2(a). We call these graphs $G_0, \ldots, G_p$ *layers*. They then triangulate each layer and obtain a spanning tree for the resulting triangulation whose radius is bounded by the depth of the layer. Hence, each fundamental cycle w.r.t. this spanning tree has low cost, and the layer can be partitioned using fundamental cycles; see Figure 6.2(b).

In summary, the algorithm therefore consists of two phases. The first phase computes the shortest-path tree T and partitions G into shallow layers by removing an appropriate set of vertices of low total cost. The second phase partitions each layer by

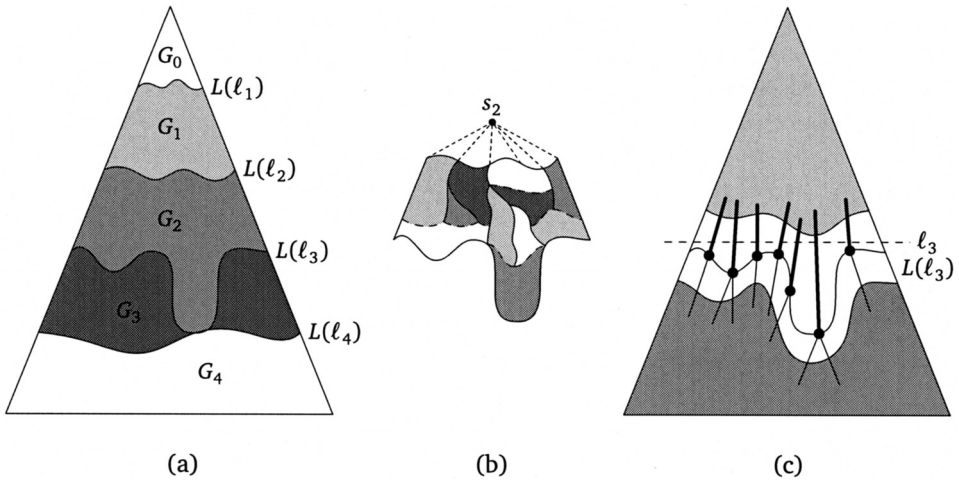

FIG. 6.2. (a) *The partition of G into layers $G_0, \ldots, G_4$ using levels $L(\ell_1), \ldots, L(\ell_4)$.* (b) *The partition of G_2 into subgraphs of weight at most $tw(G)$ using fundamental cycles. The dotted edges are the ones introduced to connect all vertices that were adjacent to vertices in $L(\ell_2)$ to the new root vertex s_2. The dashed edges are the nontree edges defining the fundamental cycles that are used to partition G_2.* (c) *The definition of level $L(\ell_3)$. The bold edges are all edges in T spanning depth ℓ_3. Their bottom endpoints are in $L(\ell_3)$. The sets $V^-(\ell_3)$ and $V^+(\ell_3)$ are shown in light and dark grey, respectively. Note that there are no edges between $V^-(\ell_3)$ and $V^+(\ell_3)$.*

removing the vertices belonging to a small set of fundamental cycles. The separator consists of all vertices removed in these two phases.

Next, we provide more details on the three parts of the algorithm: the computation of the shortest-path tree T, the partitioning of G into shallow layers, and the partitioning of each layer using fundamental cycles.

6.1.1. The shortest-path tree. To compute the spanning tree T used in the separator algorithm, recall that we assume that G is triangulated. The algorithm starts by choosing one face f, adding a new vertex s of cost and weight zero inside this face, and connecting s to the three vertices on the boundary of f.

Next, every edge xy of G is replaced with two directed edges xy and yx, and each directed edge xy is assigned a weight $w'(xy) = c(y)$; that is, the weight of every edge is equal to the cost of its head. Tree T is the shortest-path tree obtained by computing single-source shortest paths from s w.r.t. edge weights w'. It is easy to see that the distance of a vertex x from s in T equals the cost of x's ancestors in T, including x itself. For every vertex $x \in G$, let its *depth* be $d(x) = \text{dist}_T(s, x)$, and let the *radius* of T be $r(T) = \max\{d(x) : x \in V\}$.

6.1.2. Cutting G into layers. The set of vertices used to partition G into shallow layers is the union of a set of *levels* $L(\ell_1), \ldots, L(\ell_p)$, where each level $L(\ell_i)$ is the set of bottom endpoints of all edges spanning depth ℓ_i: $L(\ell_i) = \{y : e = xy \in T \text{ and } d(x) < \ell_i \leq d(y)\}$. See Figure 6.2(c) for an illustration.

Every level $L(\ell_i)$ is a separator of G whose removal partitions the vertex set V of G into two sets $V^-(\ell_i) = \{x \in V : d(x) < \ell_i\}$ and $V^+(\ell_i) = \{x \in V : d(x) \geq \ell_i \text{ and } x \notin L(\ell_i)\}$ so that no vertex in $V^-(\ell_i)$ is adjacent to a vertex in $V^+(\ell_i)$. The set of levels $L(\ell_1), \ldots, L(\ell_p)$ then partitions G into $p+1$ subgraphs $G_0, \ldots, G_p$, where $G_0 = G[V^-(\ell_1)]$, $G_p = G[V^+(\ell_p)]$, and, for $0 < i < p$, $G_i = G[V^+(\ell_i) \cap V^-(\ell_{i+1})]$.

Graphs $G_0, \ldots, G_p$ are the layers we want to compute.

There is obviously a trade-off between the cost of levels $L(\ell_1), \ldots, L(\ell_p)$ and the cost of partitioning the layers $G_0, \ldots, G_p$ using fundamental cycles. By choosing more levels $\ell_1, \ldots, \ell_p$, the layers can be made more shallow, thereby reducing the cost of the fundamental cycles used to partition them. This, however, increases the total cost of levels $L(\ell_1), \ldots, L(\ell_p)$.

As we will see, this trade-off is balanced by partitioning G into $p+1 = \lfloor r(T)/h \rfloor + 1$ layers of depth at most $2h$, where $h = \sqrt{tC(G)/8}$. This is achieved by choosing the value ℓ_i defining each level $L(\ell_i)$ from the interval $((i-1)h, ih]$ so that the level $L(\ell_i)$ has minimal cost among all levels $L(\ell)$ with $(i-1)h < \ell \leq ih$. Indeed, this ensures that two consecutive values ℓ_i and ℓ_{i+1} differ by at most $2h$; that is, every layer has depth at most $2h$. As shown by Aleksandrov et al., it also ensures that the cost of the union S_1 of levels $L(\ell_1), \ldots, L(\ell_p)$ is $c(S_1) \leq C(G)/h = 2\sqrt{2C(G)/t}$.

6.1.3. Partitioning the layers. The removal of levels $L(\ell_1), \ldots, L(\ell_p)$ partitions G into layers $G_0, \ldots, G_p$. If a layer G_i has weight at most $tw(G)$, it does not have to be partitioned further. In general, however, G_i may have a weight exceeding $tw(G)$ and needs to be partitioned into subgraphs of weight at most $tw(G)$. This is done by augmenting G_i to obtain a triangulation that has a spanning tree T_i of diameter not exceeding the depth of G_i; this augmented version of G_i is then partitioned using fundamental cycles w.r.t. T_i.

The augmentation of G_0 involves simply triangulating it. For $i > 0$, graph G_i is augmented in two steps: First, a vertex s_i of weight and cost zero is added to G_i and connected to all vertices in G_i that, in G, are adjacent to vertices in $L(\ell_i)$. The resulting graph is then triangulated, and the edges of G_i are assigned weights as in section 6.1.1. Tree T_i is now chosen to be a shortest-path tree of G_i with root s_i, where $s_0 = s$. See Figure 6.2(b) for an illustration.

The approach for finding the fundamental cycles used to partition G_i into subgraphs of weight at most $tw(G)$ can be explained as follows: Let T_i^* be the dual of T_i. This tree is obtained from the dual G_i^* of G_i by removing all those edges that are dual to edges in T_i. Thus, every edge e^* in T_i^* corresponds to a nontree edge e of G_i and, hence, represents a fundamental cycle $F(e)$ in G_i. If the vertex corresponding to the outer face of G_i is chosen as the root of T_i^*, the descendant vertices and edges of e^* in T_i^* represent the region enclosed by $F(e)$.

The goal now is to assign weights $w^*(e^*)$ to the edges of T_i^* so that the total weight of the edges in T_i^* equals the total weight of the vertices in G_i, and the total weight of the descendant edges of an edge e^* in T_i^* is an upper bound on the weight of the vertices in G_i enclosed by $F(e)$. Given such an assignment of edge weights, it suffices to partition T_i^* into a small number of subtrees of weight at most $tw(G)$ by removing a set of edges from T_i^*; the vertices on the corresponding fundamental cycles then form a separator partitioning G_i into subgraphs of weight at most $tw(G)$.

The weight function w^* is obtained by charging the weight of every vertex x in G_i to some edge e^* of T_i^*: If x has at least one incident nontree edge, edge e^* is chosen to be the dual of one of these edges. Otherwise, e^* is chosen to be the dual of a nontree edge e both of whose endpoints are neighbors of x in T_i. It is easy to see that such an edge always exists.

The weight function is easily seen to have the two properties above: Since every vertex of G_i is charged to exactly one edge of T_i^*, the total weight of the edges in T_i^* equals the total weight of the vertices in G_i. A vertex x in the region enclosed by a fundamental cycle $F(e)$ must have been charged to an edge e_1^* in T_i^* such that e_1 is

also contained in the region enclosed by $F(e)$. Thus, e_1^* is a descendant edge of e^* in T_i^*, and the weight of the descendant edges of e^* is an upper bound on the weight of the vertices in G_i enclosed by $F(e)$. See [2] for a more rigorous argument.

In order to partition T_i^* into subtrees of weight at most $tw(G)$ by removing a set of edges X_i, a leaf of T_i^* is chosen as the root of T_i^*, and the edges of T_i^* are then inspected from the bottom up. For every edge e^*, if the total weight of all its descendant edges, including e^* itself, exceeds $tw(G)/2$, the subtree below e^* is pruned from T_i^* by adding e^* to the edge separator X_i. The edges in the pruned subtree are then no longer counted when determining the total weight of the descendant edges of any ancestor of e^*.

Since the vertices in the dual of a planar triangulation have degree at most three and the root of T_i^* has degree one, T_i^* is a binary tree. Thus, the above procedure ensures that each subtree in the produced partition has weight at most $tw(G)$ and, hence, that the fundamental cycles in the set $\mathcal{F}(X_i) = \{F(e) : e^* \in X_i\}$ partition G_i into subgraphs of weight at most $tw(G)$. To bound the number of fundamental cycles in $\mathcal{F}(X_i)$, observe that every edge e^* in X_i has descendant edges of total weight at least $tw(G)/2$ and that every edge of T_i^* is counted as a descendant edge of at most one edge in X_i. Hence $|\mathcal{F}(X_i)| = |X_i| \leq \frac{2w^*(T_i^*)}{tw(G)} = \frac{2w(G_i)}{tw(G)}$. The total number of fundamental cycles used to partition the layers $G_0, \ldots, G_p$ is therefore at most $\frac{2w(G)}{tw(G)} = 2/t$. Since each layer has depth at most $2h$, the cost of the vertices on one fundamental cycle is at most $4h$ and, hence, the total cost of all fundamental cycles in $\mathcal{F}(X_0) \cup \cdots \cup \mathcal{F}(X_p)$ is at most $8h/t = 2\sqrt{2C(G)/t}$.

Let S_2 be the set of vertices on the fundamental cycles in $\mathcal{F}(X_0) \cup \cdots \cup \mathcal{F}(X_p)$. The final separator $S = S_1 \cup S_2$ partitions G into subgraphs of weight at most $tw(G)$ and has cost $c(S_1) + c(S_2) \leq 4\sqrt{2C(G)/t}$.

Aleksandrov et al. [3] showed how to implement this procedure in linear time. In the next section, we show how to carry out the three steps of the algorithm in O(sort(N)) I/Os.

6.2. An I/O-efficient algorithm.

6.2.1. Computing T. In order to compute the shortest-path tree T, we need to compute a planar embedding of G, triangulate G, add a new vertex of cost and weight 0 inside one of its faces, assign weights $w'(e)$ to the edges of G, and finally, compute single-source shortest paths w.r.t. these edge weights.

A planar embedding of G can be computed in O(sort(N)) I/Os [33]; an embedded planar graph can be triangulated in the same I/O-bound [25]. Next, we extract a description of the faces of the triangulation as lists of vertices, each containing the boundary vertices of one face sorted clockwise around that face; this can also be done in O(sort(N)) I/Os. We add a vertex s to G and traverse the vertex list representing one of the faces of G to add edges between s and the vertices on the boundary of this face to G. Now it takes O(sort(N)) I/Os to label every edge of G with the costs of its endpoints, replace every edge of G with its corresponding directed edges, and assign weights as defined in section 6.1.1 to these edges (see section 2.4). The shortest-path tree T can now be computed in O(sort(N)) I/Os using the shortest-path algorithm of [8].

This procedure for computing T is where we depend on Theorems 4.1 and 5.1. The embedding algorithm of [33] relies on a proper B^2-partition of G; the shortest-path algorithm of [8] requires a regular proper B^2-partition.

6.2.2. Cutting T into layers. To compute the levels $L(\ell_1), \ldots, L(\ell_p)$ used to partition G into layers $G_0, \ldots, G_p$, we first need to compute the values $\ell_1, \ldots, \ell_p$ and then extract the vertices belonging to $S_1 = L(\ell_1) \cup \cdots \cup L(\ell_p)$.

To compute values $\ell_1, \ldots, \ell_p$, we label both endpoints of every edge in T with their costs and their distances from s. Then we sort the edges of T by the distances of their tails from s and scan the sorted edge list to simulate a sweep from $\ell = -\infty$ to $\ell = +\infty$. During this sweep, we maintain the cost $c(L(\ell))$ of the current level $L(\ell)$ and keep track of the value i such that $(i-1)h < \ell \leq ih$. We also maintain the minimum cost $c_{\min}(i)$ of all levels $L(\ell')$ with $(i-1)h < \ell' \leq ih$ we have seen so far, as well as the level ℓ_i such that $(i-1)h < \ell_i \leq ih$ and $c(L(\ell_i)) = c_{\min}(i)$. When $\ell = d(x_j)$, for some vertex x_j, we perform the following operations: First, we test whether $d(x_{j+1}) > ih$. If so, we have finished processing all levels $L(\ell')$ with $(i-1)h < \ell' \leq ih$, so we report ℓ_i, increase i by one, and initialize $c_{\min}(i) = +\infty$. Then we decrease $c(L(\ell))$ by $c(x_j)$ and increase $c(L(\ell))$ by the total cost of the heads of all edges having x_j as their tail. This produces $c(L(d(x_{j+1})))$. If $c(L(d(x_{j+1}))) < c_{\min}(i)$, we set $c_{\min}(i) = c(L(d(x_{j+1})))$ and $\ell_i = d(x_{j+1})$.

Given values $\ell_1, \ldots, \ell_p$ and the edge set of T as sorted above, the set $S_1 = L(\ell_1) \cup \cdots \cup L(\ell_p)$ can be extracted as follows: We scan the list of values $\ell_1, \ldots, \ell_p$ and the edge set of T, again to simulate a sweep from $\ell = -\infty$ to $\ell = +\infty$. During the sweep we maintain the index i of the next level ℓ_i to be passed by the sweep; initially, $i = 1$. When the sweep passes the tail of an edge xy, its tail is at a depth less than ℓ_i. Thus, we add its head y to $L(\ell_i)$ if $d(y) \geq \ell_i$. When the sweep passes level ℓ_i, we increase i by one and, thus, start constructing the next level $L(\ell_{i+1})$. Since this computation of set S_1 requires sorting and scanning the edge set of T a constant number of times, its I/O-complexity is $O(\mathrm{sort}(N))$.

6.2.3. Partitioning the layers. The final step of the algorithm extracts graphs $G_0, \ldots, G_p$, computes shortest-path trees $T_0, \ldots, T_p$ for these graphs, and partitions each graph G_i, $0 \leq i \leq p$, into subgraphs of weight at most $tw(G)$ using fundamental cycles w.r.t. T_i.

Computing the layers. To compute graphs $G_0, \ldots, G_p$, we first compute the set $V - S_1$ and sort the vertices in $V - S_1$ by their distances from s. We scan the vertices in $V - S_1$ and the values $-\infty = \ell_0, \ldots, \ell_{p+1} = r(T)$ to partition $V - S_1$ into sets $V_0, \ldots, V_p$, where $V_i = \{x \in V - S_1 : \ell_i < d(x) \leq \ell_{i+1}\}$. For $1 \leq i \leq p$, we add a new vertex s_i to V_i. This produces the vertex sets of graphs $G_0, \ldots, G_p$.

Next we partition E into sets $E_0, \ldots, E_p$, $E_1^-, \ldots, E_p^-$, and E^+ such that every edge in E_i has both endpoints in V_i; every edge xy in E_i^- has one endpoint, say y, in V_i, and the other endpoint, x, satisfies $x \in S_1$ and $d(x) < d(y)$; and set E^+ contains the remaining edges. This partition is easily computed in $O(\mathrm{sort}(N))$ I/Os: We label every edge with the membership of its endpoints in $V_0, \ldots, V_p$ or S_1 and with their distances from s. Every edge can then determine its membership in one of the sets based on its local information, and we can sort E to obtain the desired partition. Graph G_i is now defined as $G_i = (V_i, E_i \cup E_i')$, where $E_i' = \{s_i y, y s_i : xy \in E_i^- \text{ and } d(x) < d(y)\}$. Finally, we triangulate G_i using the algorithm of [25].

This procedure requires sorting and scanning the vertex and edge sets of G a constant number of times. In addition, we invoke the $O(\mathrm{sort}(N))$-I/O triangulation algorithm of [25] on graphs $G_0, \ldots, G_p$, whose total size is $O(N)$. Hence, the construction of graphs $G_0, \ldots, G_p$ takes $O(\mathrm{sort}(N))$ I/Os.

Computing shortest-path trees and their duals. Each shortest-path tree T_i can be computed in $O(\mathrm{sort}(|G_i|))$ I/Os using the procedure described in section 6.2.1.

To construct T_i^*, we compute the dual $G_i^* = (F_i, E_i^*)$ of G_i, which can be done in $O(\text{sort}(|G_i|))$ I/Os [25]. Then we remove all edges dual to edges in T_i from E_i^*. This takes another $O(\text{sort}(|G_i|))$ I/Os (see section 2.4). Thus, in total, the construction of trees $T_0, \ldots, T_p$ and $T_0^*, \ldots, T_p^*$ takes $O(\text{sort}(N))$ I/Os.

Computing dual edge weights. Before computing an edge separator of T_i^* and the corresponding set of fundamental cycles, we have to assign weights $w^*(e^*)$ as defined in section 6.1.3 to the edges of T_i^*. We do this in two phases. First, we partition V_i into two sets V_i' and V_i'' such that every vertex in V_i' has an incident nontree edge, while all edges incident to a vertex in V_i'' are tree edges. While doing this, we also identify a nontree edge e incident to each vertex $x \in V_i'$ and add $w(x)$ to $w^*(e^*)$. In the second phase, we find a nontree edge e for every vertex $x \in V_i''$ such that both endpoints of e are neighbors of x in T_i; we add $w(x)$ to $w^*(e^*)$. The details follow.

To implement the first phase, we create a list Y_i of nontree edges of G_i. More precisely, Y_i contains directed edges xy and yx for every nontree edge xy of G_i. We sort the edges in Y_i by their tails and sort the vertices in V_i by their IDs. Now a single scan of lists V_i and Y_i suffices to identify all vertices x in V_i such that Y_i contains at least one edge with tail x. These are the vertices in V_i'; all other vertices belong to V_i''. During this scan, we also extract, for every vertex $x \in V_i'$, the first edge e with tail x from Y_i and add a pair $(e^*, w(x))$ to a list W. This list will be used after the second phase to compute the weights of the edges in T_i^*.

To implement the second phase, we observe that, for every vertex $x \in V_i''$ and every nontree edge yz such that y and z are both neighbors of x in T_i, one endpoint of yz, say y, must be a child of x in T_i, and the other, z, must be x's parent in T_i or another child of x. This implies in particular that, for every nontree edge yz, there exists at most one vertex $x \in V_i''$ such that y and z are both neighbors of x in T_i. Our goal, therefore, is to partition the nontree edges of G_i into sets $E(x)$ such that both endpoints of each edge in $E(x)$ are neighbors of x in T_i; for every vertex $x \in V_i''$, we then choose one edge from $E(x)$ and add the pair $(e^*, w(x))$ to W. To obtain the partition into sets $E(x)$, we first label every vertex $x \in T_i$ with its grandparent in T_i: We create a second copy P_i of V_i, sort the vertices in P_i by their IDs, and sort the vertices in V_i by the IDs of their parents. This ensures that the vertices in V_i are stored in the same order as their parents in P_i. Since each vertex in P_i also stores the ID of its parent, a single scan of V_i and P_i now suffices to label every vertex in V_i with the ID of its grandparent. Now we label every nontree edge of G_i with the parents and grandparents of its endpoints. A nontree edge yz belongs to $E(x)$ if and only if x is the parent of both y and z or, w.l.o.g., x is the parent of y and z is the grandparent of y. This can now be tested based on the local information stored with edge yz. If edge yz satisfies this condition, we label it as belonging to $E(x)$. We sort the nontree edges by their membership in sets $E(x)$ and scan V_i'' and the sorted edge list to add a pair $(e^*, w(x))$ to W for every vertex $x \in V_i''$ and the first edge $e \in E(x)$.

To finish the computation of the weights of the edges in T_i^*, we sort the edges of T_i^* by their IDs and the pairs in W by their first components. A single scan of these two sorted lists now suffices to add $w(x)$ to $w^*(e^*)$, for every pair $(e^*, w(x)) \in W$.

Since this procedure sorts and scans lists of size $O(|G_i|)$ a constant number of times, the assignment of weights to the edges of T_i^* takes $O(\text{sort}(|G_i|))$ I/Os. The total cost for all graphs $G_0, \ldots, G_p$ is therefore $O(\text{sort}(N))$.

Computing the edge separator and fundamental cycles. To obtain the edge separator X_i of T_i^*, we root T_i^* in an arbitrary leaf, compute a preorder numbering of T_i^* w.r.t. the chosen root, and direct all edges in T_i^* from children to parents. This

can be done using the Euler tour technique and list ranking [14]. The construction of the edge separator X_i as described in section 6.1.3 can now be implemented using the time-forward processing technique of [14]. Given the edge separator X_i of T_i^* produced by this procedure, we mark the endpoints of all edges $e \in G_i$ such that $e^* \in X_i$ and then process T_i from the bottom up (using time-forward processing again) to identify all vertices of G_i that belong to the fundamental cycles defined by the edges in X_i. We add these vertices to S_2.

This procedure applies the Euler tour technique, list ranking, and time-forward processing to T_i and T_i^*, both of which have size $O(|G_i|)$. Hence, this takes $O(\text{sort}(|G_i|))$ I/Os. Apart from this, we sort and scan lists of size $O(|G_i|)$. Thus, partitioning each graph G_i takes $O(\text{sort}(|G_i|))$ I/Os, and the computation of the whole separator S_2 takes $O(\text{sort}(N))$ I/Os.

6.2.4. Final remarks. Since we have shown that the computation of both S_1 and S_2 takes $O(\text{sort}(N))$ I/Os, Theorem 6.2 is proved. Since the algorithm relies on the separator algorithm of section 4 and the shortest-path algorithm of [8], it inherits their memory requirements. In particular, the latter requires a proper $\Theta\big(B^2\big)$-partition as part of the input and uses $\Theta\big(B^2\big)$ main memory to carry out its computation. The algorithm from section 4 can be used to produce the desired partition in $O(\text{sort}(N))$ I/Os, provided that $M = \Omega\big(B^2 \log^2 B\big)$.

As a final comment, note that the shortest-path algorithm of [8] relies on a *regular* proper $\Theta\big(B^2\big)$-partition, which is guaranteed to exist only if the graph has bounded degree. The triangulations in which we need to compute shortest paths may not satisfy this constraint, but given an embedding, each planar graph G can be transformed into a planar graph G' of size $O(|G|)$ such that every vertex in G' has degree at most three. This is done by replacing every vertex x of degree $\deg(x) > 3$ with a cycle of $\deg(x)$ vertices and making every edge incident to x incident to a different vertex on this cycle. This takes $O(\text{sort}(|G|))$ I/Os. Moreover, if the edges in each cycle replacing a high-degree vertex are given weight 0, this transformation preserves the distances between vertices. Thus, the algorithm of [8] can be used to compute shortest paths in G and in the layers $G_0, \ldots, G_p$.

6.3. Edge separators. The final result of this section is an I/O-efficient edge separator algorithm. Aleksandrov et al. [3] showed that Theorem 6.1 can also be used to compute optimal edge separators of planar graphs as follows: Define the cost of each vertex to be equal to its degree. Then compute a vertex separator S of cost at most $4\sqrt{2(\sum_{x \in V}(\deg(x))^2)/t}$ and add all edges incident to a vertex in S to the edge separator.

It is easy to verify that the computation of the vertex costs and the extraction of the edge separator from the computed vertex separator can be carried out in $O(\text{sort}(N))$ I/Os. Hence, the following result is an immediate consequence of Theorem 6.2.

THEOREM 6.3. *Let $G = (V, E)$ be a planar graph, let $0 < t < 1$ be a real number, and let $w : V \to \mathbb{R}^+$ be a weight function so that $w(x) \leq tw(G)$ for all $x \in V$. Then there exists a set S of at most $4\sqrt{2(\sum_{v \in V}(\deg(v))^2)/t}$ edges so that no connected component of $G - S$ has weight exceeding $tw(G)$. Such an edge separator S can be found in $O(\text{sort}(N))$ I/Os, provided that $M = \Omega\big(B^2 \log^2 B\big)$.*

7. Improving the memory requirements. In this final section of the paper, we show how to reduce the memory requirements of our algorithm from section 4 to $M \geq \max\big(196B^2, 7r\big)$. The resulting algorithm also produces a separator significantly

smaller than the one produced by the algorithm in section 4. However, these two improvements come at the expense of increasing the internal-memory computation of the algorithm from $O(N \log N)$ to $O(N \log N + NB)$.

Recall the reason why $M \geq 56r \log^2 B$ is required for the algorithm in section 4: If we choose to compute an r'-partition of each graph G_i in the graph hierarchy, then the separator vertices introduced at each level correspond to $O\big(N/\sqrt{r'}\big)$ vertices in G; no better upper bound is known. Since there are $\lfloor \log B \rfloor$ levels in the hierarchy, and we want a separator of size $O(N/\sqrt{r})$, we have to ensure that $O\big(\big(N/\sqrt{r'}\big) \log B\big) = O(N/\sqrt{r})$, which we achieve by choosing $r' = r \log^2 B$. This now forces us to use $56r \log^2 B$ main memory because, as argued in section 4, every piece of $G_i - S'_i$ has size at most $56r' = 56r \log^2 B$, and we need to load each such piece into memory to partition it into smaller pieces.

So the central problem is that, if we were to choose $r' = r$, then every level in the hierarchy adds $O(N/\sqrt{r})$ vertices to the final separator of G, that is, we would obtain a separator that is too big by a factor of $\log B$. Next we explain how to avoid this problem by using a recursive bootstrapping approach.

The centerpiece of the algorithm is the separator algorithm from section 6, but now using vertex costs and weights equal to 1. This algorithm takes $O(\mathrm{sort}(N))$ I/Os using only $\Theta(B)$ main memory if we ignore the costs and memory requirements of computing an embedding of G, computing the shortest-path tree T of G, and computing the shortest-path trees $T_0, \ldots, T_p$ for layers $G_0, \ldots, G_p$. Given appropriate separator decompositions, the computation of the embedding and the shortest-path computations take $O(\mathrm{sort}(N))$ I/Os and require $\Theta\big(B^2\big)$ main memory [5, 33]. Our strategy is to obtain these separator decompositions by recursive application of our algorithm.

Embedding G. To compute a planar embedding of G, we require a proper $\Theta\big(B^2\big)$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_q\})$ of G. We obtain this partition as follows: First, we apply the uniform graph contraction procedure to G and recursively compute a proper B^2-partition $\tilde{\mathcal{P}} = \big(\tilde{S}, \big\{\tilde{G}_1, \ldots, \tilde{G}_q\big\}\big)$ of the resulting graph $\tilde{G}$. Then we choose S to be the set of vertices in G represented by the vertices in $\tilde{S}$, and each graph G_i in $\mathcal{P}$ to be the subgraph of G represented by the vertices in $\tilde{G}_i$. To bound the number of vertices in G represented by each vertex in $\tilde{G}$, we assign weight 1 to every vertex in G and provide a weight threshold u, to be specified later, to the uniform graph contraction procedure. This guarantees that $|S| \leq u\big|\tilde{S}\big| = O(N/B)$, $|G_i| \leq u\big|\tilde{G}_i\big| = O\big(B^2\big)$, and $|\mathcal{N}(G_i)| = O(B)$ for all $1 \leq i \leq q$. It also ensures that $\sum_{i=1}^{q} |\mathcal{N}(G_i)| \leq u \sum_{i=1}^{q} \big|\mathcal{N}\big(\tilde{G}_i\big)\big| = O(N/B)$. Thus, $\mathcal{P}$ is a proper $\Theta\big(B^2\big)$-partition of G. An embedding of G can now be obtained from $\mathcal{P}$ in $O(\mathrm{sort}(N))$ I/Os [33]. In total, the cost of computing a planar embedding of G is $O(\mathrm{sort}(N))$ I/Os plus the cost of the recursive call on $\tilde{G}$.

Computing $T, T_0, \ldots, T_p$. Given a planar embedding of G, we use the procedure from section 6.2.4 to transform G into a planar graph G' of degree at most three and so that the distances between vertices in G are the same as the distances between their representatives in G'. This takes $O(\mathrm{sort}(N))$ I/Os. Now we apply the procedure from the previous paragraph to obtain a proper $\Theta\big(B^2\big)$-partition of G' and then use the procedures from sections 4.3 and 5 to augment this partition to obtain a regular proper B^2-partition of G'. The construction of G' takes $O(\mathrm{sort}(N))$ I/Os and, as discussed in the previous paragraph, the cost of computing a proper $\Theta(B^2)$-partition of G' is $O(\mathrm{sort}(|G'|))$ plus the cost of the recursive call on a compressed version $\tilde{G}'$ of G'. As discussed in sections 4.3 and 5, augmenting the computed partition to a regular proper

$\Theta(B^2)$-partition also takes $\mathrm{O}(\mathrm{sort}(|G'|))$ I/Os. Given such a partition, we use the single-source shortest-path algorithm of [8] to obtain a shortest-path tree T' of G' in $\mathrm{O}(\mathrm{sort}(|G'|))$ I/Os, which is easily transformed into a shortest-path tree T of G in the same number of I/Os. Thus, the total cost of computing T is $\mathrm{O}(\mathrm{sort}(N) + \mathrm{sort}(|G'|))$ plus the cost of the recursive call on $\tilde{G}'$.

The shortest-path trees $T_0, \ldots, T_p$ are obtained by applying the same procedure to graphs $G_0, \ldots, G_p$. We denote the degree-3 graphs obtained from $G_0, \ldots, G_p$ by $G'_0, \ldots, G'_p$ and the compressed versions of $G'_0, \ldots, G'_p$ for which we recursively compute proper B^2-partitions by $\tilde{G}'_0, \ldots, G'_p$. Using this notation, our discussion above implies that the computation of each tree T_i takes $\mathrm{O}(\mathrm{sort}(|G_i|) + \mathrm{sort}(|G'_i|))$ I/Os plus the cost of a recursive call on $\tilde{G}'_i$.

Analysis. Summing up the costs of the individual steps of our algorithm, the cost of computing a proper B^2-partition of G is

$$T(N) = \mathrm{O}(\mathrm{sort}(N)) + \mathrm{O}\big(\mathrm{sort}\big(\big|G'\big|\big)\big) + T\big(\big|\tilde{G}\big|\big) + T\big(\big|\tilde{G}'\big|\big) + \sum_{i=0}^{q} \Big(\mathrm{sort}(|G_i|) + \mathrm{sort}\big(\big|G'_i\big|\big) + T\big(\big|\tilde{G}'_i\big|\big)\Big).$$

The first $\mathrm{O}(\mathrm{sort}(N))$ term includes the cost of computing the separator once the trees $T, T_0, \ldots, T_p$ have been computed.

To bound this recurrence by $\mathrm{O}(\mathrm{sort}(N))$, we first bound the sizes of the different graphs involved in the computation as follows: Graph G has N vertices. Graph G' has at most $6N$ vertices, at most two per edge in G. Graphs $G_0, \ldots, G_p$ contain at most N vertices, as they are vertex-disjoint subgraphs of G. Thus, graphs $G'_0, \ldots, G'_p$ also contain at most $6N$ vertices. The total size of graphs $G, G', G_0, \ldots, G_p, G'_0, \ldots, G'_p$ is therefore $\mathrm{O}(N)$, which implies that the sort$(\cdot)$-terms in the above recurrence sum to $\mathrm{O}(\mathrm{sort}(N))$, simplifying the recurrence to

$$T(N) = \mathrm{O}(\mathrm{sort}(N)) + T\big(\big|\tilde{G}\big|\big) + T\big(\big|\tilde{G}'\big|\big) + \sum_{i=0}^{p} T\big(\big|\tilde{G}'_i\big|\big).$$

Now, if we choose $u = 196$, graph $\tilde{G}$ contains at most $|G|/98$ heavy vertices and, by Theorem 3.1, has size at most $|G|/14$. Similarly, $\big|\tilde{G}'\big| \le |G'|/14$ and, for $0 \le i \le p$, $\big|\tilde{G}'_i\big| \le |G'_i|/14$. Thus, we have

$$\begin{aligned}\big|\tilde{G}\big| + \big|\tilde{G}'\big| + \sum_{i=0}^{p}\big|\tilde{G}'_i\big| &\le \frac{|G|}{14} + \frac{|G'|}{14} + \sum_{i=0}^{p}\frac{|G'_i|}{14} \\ &\le \frac{N}{14} + \frac{6N}{14} + \frac{6N}{14} \\ &= \frac{13N}{14},\end{aligned}$$

and the recurrence solves to $T(N) = \mathrm{O}(\mathrm{sort}(N))$. The memory requirements of the embedding and shortest-path algorithms are at most $uB^2 = 196B^2$, as we provide them with uB^2-partitions. This proves that we can compute a t-vertex separator and, thus, a t-edge separator, for any $0 < t < 1$, in $\mathrm{O}(\mathrm{sort}(N))$ I/Os, provided that $M \ge 196B^2$. In order to obtain a (regular) proper r-partition from an (r/N)-separator, we still have to be able to load subgraphs of size at most $7r$ into internal memory. This leads to the following result.

THEOREM 7.1. *The partitions in Theorems* 4.1 *and* 5.1 *can be computed in* $O(\mathrm{sort}(N))$ *I/Os, provided that* $M \geq \max\bigl(196B^2, 7r\bigr)$. *The separators in Theorems* 6.2 *and* 6.3 *can be computed in* $O(\mathrm{sort}(N))$ *I/Os, provided that* $M \geq 196B^2$.

The memory requirements in Theorem 7.1 can be reduced further to $M \geq \max\bigl(cB^2, 7r\bigr)$ and $M \geq cB^2$, for an arbitrarily small constant $c > 0$. Indeed, $196B^2$ memory is required because we recursively compute B^2-partitions of the compressed graphs, which correspond to $\bigl(196B^2\bigr)$-partitions in the uncompressed graphs. Instead, we can compute $\bigl(cB^2/196\bigr)$-partitions of the compressed graphs, thereby obtaining $\bigl(cB^2\bigr)$-partitions of the uncompressed graphs. This affects the sizes of the produced separators—and, thus, the performance of the embedding and shortest-path algorithms—by only a constant factor but reduces the memory requirements.

8. Conclusions. In this paper, we have demonstrated that different types of separator decompositions of planar graphs can be computed I/O-efficiently. Using these partitions, a wide variety of fundamental problems on planar graphs can be solved I/O-efficiently.

A number of open questions remain, however. The constant factors in our algorithms—in terms of the size of the produced separator, the memory requirements, and the efficiency—are big. In order for the algorithms to be of practical value, these constant factors have to be reduced. Furthermore, even though the individual steps of the algorithm are fairly simple, the algorithm consists of too many of them. This makes the algorithm tedious to implement and impacts the efficiency of the algorithm; for example, only a small number of sorting steps is affordable in practice. From a practical point of view, it would therefore be desirable to have a simpler—even possibly a theoretically suboptimal—algorithm for computing separators I/O-efficiently. On the theoretical side, the most important open questions are whether separators can be computed in $O(\mathrm{sort}(N))$ I/Os using $o\bigl(B^2\bigr)$ main memory and whether they can be computed in $O(\mathrm{sort}(N))$ I/Os cache-obliviously. See [20] for a discussion of cache-obliviousness.

Acknowledgments. We would like to thank Richard Cole and the anonymous referees for helpful comments on how to improve the presentation of the results in this paper.

REFERENCES

[1] A. AGGARWAL AND J. S. VITTER, *The input/output complexity of sorting and related problems*, Comm. ACM, (1988), pp. 1116–1127.

[2] L. ALEKSANDROV AND H. DJIDJEV, *Linear algorithms for partitioning embedded graphs of bounded genus*, SIAM J. Discrete Math., 9 (1996), pp. 129–150.

[3] L. ALEKSANDROV, H. DJIDJEV, H. GUO, AND A. MAHESHWARI, *Partitioning planar graphs with costs and weights*, in Proceedings of the 4th Workshop on Algorithm Engineering and Experiments, Lecture Notes in Comput. Sci. 2409, Springer-Verlag, Berlin, New York, 2002, pp. 98–107.

[4] L. ARGE, *The buffer tree: A technique for designing batched external data structures*, Algorithmica, 37 (2003), pp. 1–24.

[5] L. ARGE, G. S. BRODAL, AND L. TOMA, *On external-memory MST, SSSP and multi-way planar graph separation*, J. Algorithms, 53 (2004), pp. 186–206.

[6] L. ARGE, U. MEYER, L. TOMA, AND N. ZEH, *On external-memory planar depth first search*, J. Graph Algorithms Appl., 7 (2003), pp. 105–129.

[7] L. ARGE AND L. TOMA, *Simplified external memory algorithms for planar DAGs*, in Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 3111, Springer-Verlag, Berlin, New York, 2004, pp. 493–503.

[8] L. Arge, L. Toma, and N. Zeh, *I/O-efficient algorithms for planar digraphs*, in Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, 2003, pp. 85–93.
[9] L. Arge and N. Zeh, *I/O-efficient strong connectivity and depth-first search for directed planar graphs*, in Proceedings of the 44th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 2003, pp. 261–270.
[10] K. Booth and G. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.
[11] J. Boyer and W. Myrvold, *Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm*, in Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1999, pp. 140–146.
[12] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, *Graph structure in the web*, Computer Networks, 33 (2000), pp. 309–320.
[13] A. L. Buchsbaum and J. R. Westbrook, *Maintaining hierarchical graph views*, in Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2000, pp. 566–575.
[14] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, *External-memory graph algorithms*, in Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1995, pp. 139–149.
[15] K. Diks, H. N. Djidjev, O. Sykora, and I. Vrto, *Edge separators of planar and outerplanar graphs with applications*, J. Algorithms, 14 (1993), pp. 258–279.
[16] H. N. Djidjev, *Partitioning graphs with costs and weights on vertices: Algorithms and applications*, Algorithmica, 28 (2000), pp. 51–75.
[17] H. N. Djidjev and J. R. Gilbert, *Separators in graphs with negative and multiple vertex weights*, Algorithmica, 23 (1999), pp. 57–71.
[18] S. Even and R. E. Tarjan, *Computing an st-numbering*, Theoret. Comput. Sci., 2 (1976), pp. 339–344.
[19] G. N. Frederickson, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.
[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, in Proceedings of the 40th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1999, pp. 285–297.
[21] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan, *A separator theorem for graphs of bounded genus*, J. Algorithms, 5 (1984), pp. 391–407.
[22] M. T. Goodrich, *Planar separators and parallel polygon triangulation*, J. Comput. System Sci., 51 (1995), pp. 374–389.
[23] F. Harary, *Graph Theory*, Addison-Wesley, New York, 1969.
[24] J. Hopcroft and R. E. Tarjan, *Efficient planarity testing*, J. ACM, 21 (1974), pp. 549–568.
[25] D. Hutchinson, A. Maheshwari, and N. Zeh, *An external memory data structure for shortest path queries*, Discrete Appl. Math., 126 (2003), pp. 55–82.
[26] A. Lempel, S. Even, and I. Cederbaum, *An algorithm for planarity testing of graphs*, in Theory of Graphs: International Symposium (Rome, 1966), Gordon and Breach, New York, 1967, pp. 215–232.
[27] R. J. Lipton and R. E. Tarjan, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.
[28] U. Meyer, P. Sanders, and J. Sibeyn, eds., *Algorithms for Memory Hierarchies: Advanced Lectures*, Lecture Notes in Comput. Sci. 2625, Springer-Verlag, Berlin, New York, 2003.
[29] G. L. Miller, *Finding small simple cycle separators for 2-connected planar graphs*, J. Comput. System Sci., 32 (1986), pp. 265–279.
[30] K. Munagala and A. Ranade, *I/O-complexity of graph algorithms*, in Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1999, pp. 687–694.
[31] W. T. Tutte, *Graph Theory*, Cambridge University Press, Cambridge, UK, 2001.
[32] J. S. Vitter, *External memory algorithms and data structures: Dealing with massive data*, ACM Comput. Surveys, 33 (2001), pp. 209–271.
[33] N. Zeh, *I/O-Efficient Algorithms for Shortest Path Related Problems*, Ph.D. thesis, School of Computer Science, Carleton University, Ottawa, ON, Canada, 2002.