Journal of Graph Algorithms and Applications

http://jgaa.info/

vol. 7, no. 2, pp. 105–129 (2003)

On External-Memory Planar Depth-First Search

Lars Arge[1], *Ulrich Meyer*[2], *Laura Toma*[1], *Norbert Zeh*[3]

[1]Department of Computer Science
Duke University, Durham, NC 27708, USA
{large,laura }@cs.duke.edu

[2]Max-Planck-Institut für Informatik
Saarbrücken, Germany
umeyer@mpi-sb.mpg.de

[3]School of Computer Science
Carleton University, Ottawa, Canada
nzeh@scs.carleton.ca

Abstract

Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems still remain open. For example, no space- and I/O-efficient algorithms are known for depth-first search or breath-first search in sparse graphs. In this paper, we present two new results on I/O-efficient depth-first search in an important class of sparse graphs, namely undirected embedded planar graphs. We develop a new depth-first search algorithm that uses $O(\mathrm{sort}(N)\log(N/M))$ I/Os, and show how planar depth-first search can be reduced to planar breadth-first search in $O(\mathrm{sort}(N))$ I/Os. As part of the first result, we develop the first I/O-efficient algorithm for finding a simple cycle separator of an embedded biconnected planar graph. This algorithm uses $O(\mathrm{sort}(N))$ I/Os.

Communicated by Giuseppe Liotta and Ioannis G. Tollis: submitted November 2002; revised January 2003.

Lars Arge and Laura Toma supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879 and CAREER grant CCR–9984099. Ulrich Meyer supported in part by the IST Programme of the EU under contract number IST–1999–14186 (ALCOM–FT). Norbert Zeh supported in part by NSERC and NCE GEOIDE research grants.

1 Introduction

External memory graph algorithms have received considerable attention lately because massive graphs arise naturally in many applications. Recent web crawls, for example, produce graphs with on the order of 200 million vertices and 2 billion edges [11]. Recent work in web modeling uses depth-first search, breadth-first search, shortest path and connected component computations as primitive routines for investigating the structure of the web [9]. Massive graphs are also often manipulated in Geographic Information Systems (GIS), where many common problems can be formulated as basic graph problems. Yet another example of a massive graph is AT&T's 20 TB phone-call data graph [11]. When working with such massive data sets, the I/O-communication, and not the internal memory computation, is often the bottleneck. I/O-efficient algorithms can thus lead to considerable run-time improvements.

Breadth-first search (BFS) and depth-first search (DFS) are the two most fundamental graph searching strategies. They are extensively used in many graph algorithms. The reason is that in internal memory both strategies are easy to implement in linear time; yet they reveal important information about the structure of the given graph. Unfortunately no I/O-efficient BFS or DFS-algorithms are known for arbitrary sparse graphs, while known algorithms perform reasonably well on dense graphs. The problem with the standard implementations of DFS and BFS is that they decide which vertex to visit next one vertex at a time, instead of predicting the sequence of vertices to be visited. As a result, vertices are visited in a random fashion, which may cause the algorithm to spend one I/O per vertex. Unfortunately it seems that in order to predict the order in which vertices are visited, one essentially has to solve the searching problem at hand. For dense graphs, the I/Os spent on accessing vertices in a random fashion can be charged to the large number of edges in the graph; for sparse graphs, such an amortization argument cannot be applied.

In this paper, we consider an important class of sparse graphs, namely *undirected embedded planar graphs*: A graph G is *planar* if it can be drawn in the plane so that its edges intersect only at their endpoints. Such a drawing is called a *planar embedding* of G. If graph G is given together with an embedding, we call it *embedded*. The class of planar graphs is restricted enough, and the structural information provided by a planar embedding is rich enough, to hope for more efficient algorithms than for arbitrary sparse graphs. Several such algorithms have indeed been obtained recently [6, 16, 22, 24]. We develop an improved DFS-algorithm for embedded planar graphs and show that planar DFS can be reduced to planar BFS in an I/O-efficient manner.

1.1 I/O-Model and Previous Results

We work in the standard disk model proposed in [3]. The model defines the following parameters:

$$\begin{aligned} N &= \text{number of vertices and edges } (N = |V| + |E|), \\ M &= \text{number of vertices/edges that can fit into internal memory, and} \\ B &= \text{number of vertices/edges per disk block,} \end{aligned}$$

where $2B < M < N$. In an *Input/Output* operation (or simply *I/O*) one block of data is transferred between disk and internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta\left(\frac{N}{B}\right)$ (the *linear* or *scanning* bound). The number of I/Os required to sort N items is $\text{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ (the *sorting* bound) [3]. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$. Therefore the difference between the running times of an algorithm performing N I/Os and one performing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os can be considerable [8].

I/O-efficient graph algorithms have been considered by a number of authors [1, 2, 4, 5, 6, 10, 12, 14, 16, 19, 20, 21, 22, 23, 24, 26, 30]. We review the previous results most relevant to our work (see Table 1). The best known general DFS-algorithms on undirected graphs use $O\left(|V| + \text{scan}(|E|)\right) \cdot \log_2 |V|)$ [19] or $O\left(|V| + \frac{|V|}{M} \cdot \text{scan}(E)\right)$ I/Os [12]. Since the best known BFS-algorithm for general graphs uses only $O\left(\sqrt{\frac{|V|(|V|+|E|)}{B}} + \text{sort}(|V| + |E|)\right)$ I/Os [23], this suggests that on undirected graphs, DFS may be harder than BFS. For directed graphs, the best known algorithms for both problems use $O\left(\left(|V| + \frac{|E|}{B}\right) \cdot \log_2 \frac{|V|}{B} + \text{sort}(|E|)\right)$ I/Os [10]. For most graph problems $\Omega(\min\{|V|, \text{sort}(|V|)\})$ is a lower bound [5, 12], and, as discussed above, this is $\Omega(\text{sort}(|V|))$ in all practical cases. Still, all of the above algorithms, except the recent BFS-algorithm of [23], use $\Omega(|V|)$ I/Os. For sparse graphs, the same I/O-complexity can

Problem	General graphs		Planar graphs	
DFS	$O\left(\|V\| + \frac{\|V\|}{M} \cdot \text{scan}(E)\right)$	[12]	$O(N)$	
	$O\left((\|V\| + \text{scan}(\|E\|)) \cdot \log_2 \|V\|\right)$	[19]		
BFS	$O\left(\sqrt{\frac{\|V\|(\|V\|+\|E\|)}{B}} + \text{sort}(\|V\| + \|E\|)\right)$	[23]	$O(N/\sqrt{B})$	[23]

Table 1: Best known upper bounds for BFS and DFS on undirected graphs (and linear space).

be achieved much easier using the standard internal memory algorithm. Improved algorithms have been developed for special classes of planar graphs. For undirected planar graphs the first $o(N)$ DFS and BFS algorithms were developed by [24]. These algorithms use $O(\frac{N}{\gamma \log B} + \text{sort}(NB^\gamma))$ I/Os and $O(NB^\gamma)$ space, for any $0 < \gamma \leq 1/2$. BFS and DFS can be solved in $O(\text{sort}(N))$ I/Os on trees [10, 12] and outerplanar graphs [20]. BFS can also be solved in $O(\text{sort}(N))$ I/Os on k-outerplanar graphs [21].

1.2 Our Results

The contribution of this paper is two-fold. In Section 3, we present a new DFS-algorithm for undirected embedded planar graphs that uses $O(\text{sort}(N)\log(N/M))$ I/Os and linear space. For most practical values of B, M and N this algorithm uses $o(N)$ I/Os and is the first algorithm to do so using linear space. The algorithm is based on a divide-and-conquer approach first proposed in [27]. It utilizes a new $O(\text{sort}(N))$ I/O algorithm for finding a simple cycle in a biconnected planar graph such that neither the subgraph inside nor the one outside the cycle contains more than a constant fraction of the vertices of the graph. Previously, no such algorithm was known.

In Section 4 we obtain an $O(\text{sort}(N))$ I/O reduction from DFS to BFS on undirected embedded planar graphs using ideas similar to the ones in [15]. Contrary to what has been conjectured for general graphs, this shows that for planar graphs, BFS is as hard as DFS. Together with two recent results [6, 22], this implies that planar DFS can be solved in $O(\text{sort}(N))$ I/Os. In particular, Arge *et al.* [6] show that BFS and the single source shortest path problem can be solved in $O(\text{sort}(N))$ I/Os, given a multi-way separator of a planar graph. Maheshwari and Zeh [22] show that such a separator can be computed in $O(\text{sort}(N))$ I/Os.

A preliminary version of this paper appeared in [7].

2 Basic Graph Operations

In the algorithms described in Sections 3 and 4 we make use of previously developed $O(\text{sort}(N))$ I/O solutions for a number of basic graph problems. We review these problems below. Most of the basic computations we use require a total order on the vertex set V and on the edge set E of the graph $G = (V, E)$. For the vertex set V, such a total order is trivially provided by a unique numbering of the vertices in G. For the edge set E, we assume that an edge $\{v, w\}$ is stored as the pair (v, w), $v < w$, and we define $(v, w) < (x, y)$ for edges (v, w) and (x, y) in E if either $v < x$ or, $v = x$ and $w < y$. We call this ordering the *lexicographical order* of E. Another ordering, which we call the *inverted lexicographical order* of E, defines $(v, w) < (x, y)$ if either $w < y$, or $w = y$ and $v < x$.

Set difference: Even though strictly speaking set difference is not a graph operation, we often apply it to the vertex and edge sets of a graph. To compute the difference $X \setminus Y$ of two sets X and Y drawn from a total order, we first sort X and Y. Then we scan the two resulting sorted lists simultaneously, in a way similar to merging them into one sorted list. However, elements from Y are not copied to the output list, and an element from X is copied only if it does not match the current element in Y. This clearly takes $O(\text{sort}(N))$ I/Os, where $N = |X| + |Y|$. We use SetDifference as a shorthand for this operation.

Duplicate removal: Given a list $X = \langle x_1, \ldots, x_N \rangle$ with some entries potentially occurring more than once, the DuplicateRemoval operation computes a list $Y = \langle y_1, \ldots, y_q \rangle$ such that $\{x_1, \ldots, x_N\} = \{y_1, \ldots, y_q\}$, $y_j = x_{i_j}$, for indices $i_1 < \cdots < i_q$, and $x_l \neq y_j$, for $1 \leq l < i_j$. That is, list Y contains the first occurrences of all elements in X in sorted order. (Alternatively we may require list Y to store the last occurrences of all elements in X.) To compute Y in $O(\text{sort}(N))$ I/Os, we scan X and replace every element x_i with the pair (x_i, i). We sort the resulting list X' lexicographically. Now we scan list X' and discard for every x, all pairs that have x as their first component, except the first such pair. List Y can now be obtained by sorting the remaining pairs (x, y) by their indices y and scanning the resulting list to replace every pair (x, y) with the single element x.

Computing incident edges: Given a set V of vertices and a set E of edges, the IncidentEdges operation computes the set E' of edges $\{v, w\} \in E$ such that $v \in V$ and $w \notin V$. To compute E' in $O(\text{sort}(N))$ I/Os where $N = |V| + |E|$, we sort V in increasing order and E in lexicographical order. We scan V and E and mark every edge in E that has its first endpoint in V. We sort E in inverted lexicographical order and scan V and E again to mark every edge in E that has its second endpoint in V. Finally we scan E and remove all edges that have not been marked or have been marked twice.

Copying labels from edges to vertices: Given a graph $G = (V, E)$ and a labeling $\lambda : E \to X$ of the edges in E, the SumEdgeLabels operation computes a labeling $\lambda' : V \to X$ of the vertices in V, where $\lambda'(v) = \bigoplus_{e \in E_v} \lambda(e)$, E_v is the set of edges incident to v, and $\oplus$ is any given associative and commutative operator on X. To compute labeling λ' in $O(\text{sort}(N))$ I/Os, we sort V in increasing order and E lexicographically. We scan V and E and compute a label $\lambda''(v) = \bigoplus_{e \in E'_v} \lambda(e)$, for each v, where E'_v is the set of edges that have v as their first endpoint. Then we sort E in inverted lexicographical order and scan V and E to compute the label $\lambda'(v) = \lambda''(v) + \bigoplus_{e \in E''_v} \lambda(e)$, for each v, where E''_v is the set of edges that have v as their second endpoint.

Copying labels from vertices to edges: Given a graph $G = (V, E)$ and a labeling $\lambda : V \to X$ of the vertices in V, the CopyVertexLabels operation computes a labeling $\lambda' : E \to X \times X$, where $\lambda'(\{v, w\}) = (\lambda(v), \lambda(w))$. We can compute this labeling in $O(\text{sort}(N))$ I/Os using a procedure similar to the one implementing operation SumEdgeLabels.

Algorithms for lists and trees: Given a list stored as an unordered sequence of edges $\{(u, next(u))\}$, *list ranking* is the problem of determining for every vertex u in the list, the number of edges from u to the end of the list. List ranking can be solved in $O(\text{sort}(N))$ I/Os [4, 12] using techniques similar to the ones used in efficient parallel list ranking algorithms [18]. Using list ranking and PRAM techniques, $O(\text{sort}(N))$ I/O algorithms can also be developed for most problems on trees, including Euler tour computation, BFS and DFS-numbering, and lowest common ancestor queries (Q queries can be answered in $O(\text{sort}(Q + N))$ I/Os) [12]. Any computation that can be expressed as a "level-by-level" traversal of a tree, where the value of every vertex is computed either from the values of its children or from the value of its parent, can also be carried out in $O(\text{sort}(N))$ I/Os [12].

Algorithms for planar graphs: Even though no $O(\text{sort}(N))$ I/O algorithms for BFS or DFS in planar graphs have been developed, there exist $O(\text{sort}(N))$ I/O solutions for a few other problems on planar graphs, namely computing the connected and biconnected components, spanning trees and minimum spanning trees [12]. All these algorithms are based on edge-contraction, similar to the PRAM algorithms for these problems [13, 29]. We make extensive use of these algorithms in our DFS-algorithms.

3 Depth-First Search using Simple Cycle Separators

3.1 Outline of the Algorithm

Our new algorithm for computing a DFS-tree of an embedded planar graph in $O(\text{sort}(N)\log(N/M))$ I/Os and linear space is based on a divide-and-conquer approach first proposed in [27]. First we introduce some terminology used in this section.

A *cutpoint* of a graph G is a vertex whose removal disconnects G. A connected graph G is *biconnected* if it does not have any cutpoints. The *biconnected components* or *bicomps* of a graph are its maximal biconnected subgraphs. A *simple cycle α-separator* C of an embedded planar graph G is a simple cycle such that neither the subgraph inside nor the one outside the cycle contains more than $\alpha|V|$ vertices. Such a cycle is guaranteed to exist only if G is biconnected.

The main idea of our algorithm is to partition G using a simple cycle α-separator C, recursively compute DFS-trees for the connected components of $G \setminus C$, and combine them to obtain a DFS-tree for G. If each recursive step can be carried out in $O(\text{sort}(N))$ I/Os, it follows that the whole algorithm takes $O(\text{sort}(N)\log(N/M))$ I/Os because the sizes of the subgraphs of G we recurse on are geometrically decreasing, and we can stop the recursion as soon as the current graph fits into main memory. Below we discuss our algorithm in more detail, first assuming that the graph is biconnected.

Given a biconnected embedded planar graph G and some vertex $s \in G$, we construct a DFS-tree T of G rooted at s as follows (see Figure 1):

1. *Compute a simple cycle $\frac{2}{3}$-separator C of G.*

   In Section 3.2, we show how to do this in $O(\text{sort}(N))$ I/Os.

2. *Find a path P from s to some vertex v in C.*

   To do this, we compute an arbitrary spanning tree T' of G, rooted at s, and find a vertex $v \in C$ whose distance to s in T' is minimal. Path P is the path from s to v in T'. The spanning tree T' can be computed in $O(\text{sort}(N))$ I/Os [12]. Given tree T', vertex v can easily be found in $O(\text{sort}(N))$ I/Os using a BFS-traversal [12] of T'. Path P can then be identified by extracting all ancestors of v in T'. This takes $O(\text{sort}(N))$ I/Os using standard tree computations [12].

3. *Extend P to a path P' containing all vertices in P and C.*

   To do this, we identify one of the two neighbors of v in C. Let w be this neighbor, and let C' be the path obtained by removing edge $\{v, w\}$ from C. Then path P' is the concatenation of paths P and C'. This computation can easily be carried out in $O(\text{scan}(N))$ I/Os: First we scan the edge list of C and remove the first edge we find that has v as an endpoint. Then we concatenate the resulting edge list of C' and the edge list of P.

4. *Compute the connected components $H_1, \ldots, H_k$ of $G \setminus P'$. For each component H_i, find the vertex $v_i \in P'$ furthest away from s along P' such that there is an edge $\{u_i, v_i\}$, $u_i \in H_i$.*

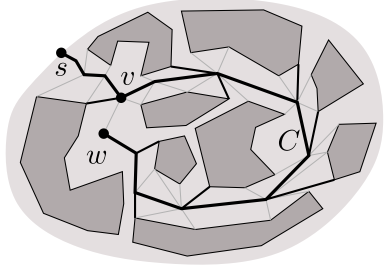

Figure 1: The path P' is shown in bold. The connected components of $G \setminus P'$ are shaded dark gray. Medium edges are edges $\{u_i, v_i\}$. Light edges are non-tree edges.

The connected components $H_1, \ldots, H_k$ can be computed in $O(\text{sort}(N))$ I/Os [12]. We find vertices $v_1, \ldots, v_k$ in $O(\text{sort}(N))$ I/Os as follows: First we mark every vertex in P' with its distance from s along P'. These distances can be computed in $O(\text{sort}(N))$ I/Os using the Euler tour technique and list ranking [12]. Then we apply operation IncidentEdges to $V(P')$ and $E(G)$, to find all edges in $E(G) \setminus E(P')$ incident to P'. We sort the resulting edge set so that edge $\{v, w\}$, $v \in H_i$, $w \in P'$, precedes edge $\{x, y\}$, $x \in H_j$, $y \in P'$, if either $i < j$ or $i = j$ and the distance from s to w is no larger than the distance from s to y. Ties are broken arbitrarily. We scan the resulting list and extract for every H_i the last edge $\{u_i, v_i\}$, $u_i \in H_i$, in this list.

5. *Recursively compute DFS-trees $T_1, \ldots, T_k$ for components $H_1, \ldots, H_k$, rooted at vertices $u_1, \ldots, u_k$, and construct a DFS-tree T for G as the union of trees $T_1, \ldots, T_k$, path P', and edges $\{u_i, v_i\}$, $1 \leq i \leq k$. Note that components $H_1, \ldots, H_k$ are not necessarily biconnected. Below we show how to deal with this case.*

To prove the correctness of our algorithm, we have to show that T is indeed a DFS-tree for G. To do this, the following classification of the edges in $E(G) \setminus E(T)$ is useful: An edge $e = (u, v)$ in $E(G) \setminus E(T)$ is called a *back-edge* if u is an ancestor of v in T, or vice versa; otherwise e is called a *cross-edge*. In [28] it is shown that a spanning tree T of a graph G is a DFS-tree of G if and only if all edges in $E(G) \setminus E(T)$ are back-edges.

Lemma 1 *The tree T computed by the above algorithm is a DFS-tree of G.*

Proof: It is easy to see that T is a spanning tree of G. To prove that T is a DFS-tree, we have to show that all non-tree edges in G are back-edges. First note that there are no edges between components $H_1, \ldots, H_k$. All non-tree edges with both endpoints in a component H_i are back-edges because tree T_i is a DFS-tree of H_i. All non-tree edges with both endpoints in P' are back-edges because P' is a path. For every non-tree edge $\{v, w\}$ with $v \in P'$ and $w \in H_i$, w is a descendant of the root u_i of the DFS-tree T_i. Tree T_i is connected to P' through edge $\{v_i, u_i\}$. By the choice of vertex v_i, v is an ancestor of v_i and thus an ancestor of u_i and w. Hence, edge $\{v, w\}$ is a back-edge. □

In the above description of our algorithm we assume that G is biconnected. If this is not the case, we find the bicomps of G, compute DFS-trees for all bicomps, and join these trees at the cutpoints of G. More precisely, we compute the *bicomp-cutpoint-tree* T_G of G containing all cutpoints of G and one vertex $v(C)$ per bicomp C (see Figure 2). There is an edge between a cutpoint v and a bicomp vertex $v(C)$ if v is contained in C. We choose the bicomp vertex $v(C_r)$ corresponding to a bicomp C_r that contains vertex s as the root of T_G. The *parent cutpoint* of a bicomp $C \neq C_r$ is the parent $p(v(C))$ of $v(C)$ in T_G. T_G can be constructed in $O(\text{sort}(N))$ I/Os, using the algorithms discussed in Section 2. We compute a DFS-tree of C_r rooted at vertex s. For every bicomp $C \neq C_r$,

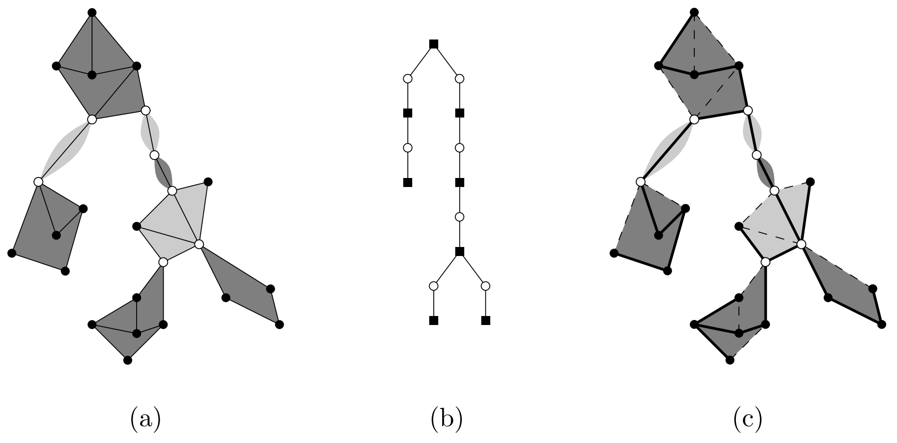

Figure 2: (a) A connected graph G with its bicomps shaded. Cutpoints are hollow. Other vertices are solid. (b) The bicomp-cutpoint-tree of G. Bicomp vertices are squares. (c) The DFS tree of G obtained by "gluing together" DFS-trees of its bicomps.

we compute a DFS-tree rooted at the parent cutpoint of C. The union of the resulting DFS-trees (see Figure 2c) is a DFS-tree for G rooted at s, since there are no edges between different bicomps. Thus, we obtain our first main result.

Theorem 1 *A DFS-tree of an embedded planar graph can be computed in* $O(sort(N)\log(N/M))$ *I/O operations and linear space.*

3.2 Finding a Simple Cycle Separator

In this section, we show how to compute a simple cycle $\frac{2}{3}$-separator of an embedded biconnected planar graph, utilizing ideas similar to the ones used in [17, 25]. As in the previous section, we start by introducing the necessary terminology.

Given an embedded planar graph G, the *faces* of G are the connected regions of $\mathbb{R}^2 \setminus G$. We use F to denote the set of faces of G. The *boundary* of a face f is the set of edges contained in the closure of f. For a set F' of faces of G, let $G_{F'}$ be the subgraph of G defined as the union of the boundaries of the faces in F' (see Figure 3a). The *complement* $\overline{G}_{F'}$ of $G_{F'}$ is the graph obtained as the union of the boundaries of all faces in $F \setminus F'$ (see Figure 3b). The *boundary* of $G_{F'}$ is the intersection between $G_{F'}$ and its complement $\overline{G}_{F'}$ (see Figure 3c). The *dual* G^* of G is the graph containing one vertex f^* per face $f \in F$, and an edge between two vertices f_1^* and f_2^* if faces f_1 and f_2 share an edge (see Figure 3d). We use v^*, e^*, and f^* to refer to the face, edge, and vertex that is dual to vertex v, edge e, and face f, respectively. The dual G^* of a planar graph G is planar and can be computed in $O(\text{sort}(N))$ I/Os [16].

The idea in our algorithm is to find a set of faces $F' \subset F$ such that the

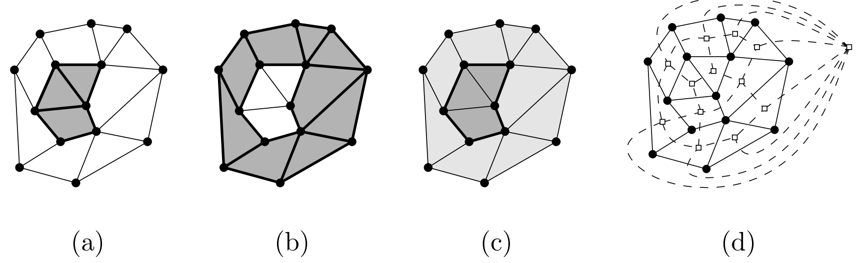

Figure 3: (a) A graph G and a set F' of faces shaded gray. The edges in $G_{F'}$ are shown in bold. (b) The shaded faces are the faces in $F \setminus F'$. The bold edges are in $\bar{G}_{F'}$. (c) The boundary of $G_{F'}$ shown in bold. (d) The dual of G represented by hollow squares and dashed edges.

boundary of $G_{F'}$ is a simple cycle $\frac{2}{3}$-separator. The main difficulty is to ensure that the boundary of $G_{F'}$ is a simple cycle. We compute F' as follows:

1. **Checking for heavy faces:** We check whether there is a single face whose boundary has size at least $\frac{|V|}{3}$ (Figure 4a). If we find such a face, we report its boundary as the separator C, as there are no vertices inside C and at most $\frac{2}{3}|V|$ vertices outside C.

2. **Checking for heavy subtrees:** If there is no heavy face, we compute a spanning tree T^* of the dual G^* of G, and choose an arbitrary vertex r as its root. Every vertex $v \in T^*$ defines a subtree $T^*(v)$ of T^* that contains v and all its descendants. The vertices in this subtree correspond to a set of faces in G whose boundaries define a graph $G(v)$. Below we show that the boundary of $G(v)$ is a simple cycle. We try to find a vertex v such that $\frac{1}{3}|V| \leq |G(v)| \leq \frac{2}{3}|V|$, where $|G(v)|$ is the number of vertices in $G(v)$ (Figure 4b). If we succeed, we report the boundary of $G(v)$ as the separator C.

3. **Splitting a heavy subtree:** If Steps 1 and 2 fail to produce a simple cycle $\frac{2}{3}$-separator of G, we are left in a situation where for every leaf $l \in T^*$ (face in G), we have $|G(l)| < \frac{1}{3}|V|$; for the root r of T^*, we have $|G(r)| = |V|$; and for every other vertex $v \in T^*$, either $|G(v)| < \frac{1}{3}|V|$ or $|G(v)| > \frac{2}{3}|V|$. Thus, there has to be a vertex v with $|G(v)| > \frac{2}{3}|V|$ and $|G(w_i)| < \frac{1}{3}|V|$, for all children $w_1, \ldots, w_k$ of v. We show how to compute a subgraph G' of $G(v)$ consisting of the boundary of the face v^* and a subset of the graphs $G(w_1), \ldots, G(w_k)$ such that $\frac{1}{3}|V| \leq |G'| \leq \frac{2}{3}|V|$, and the boundary of G' is a simple cycle (Figure 4c).

Below we describe our algorithm in detail and show that all of the above steps can be carried out in $O(\text{sort}(N))$ I/Os. This proves the following theorem.

Theorem 2 *A simple cycle $\frac{2}{3}$-separator of an embedded biconnected planar graph can be computed in $O(sort(N))$ I/O operations and linear space.*

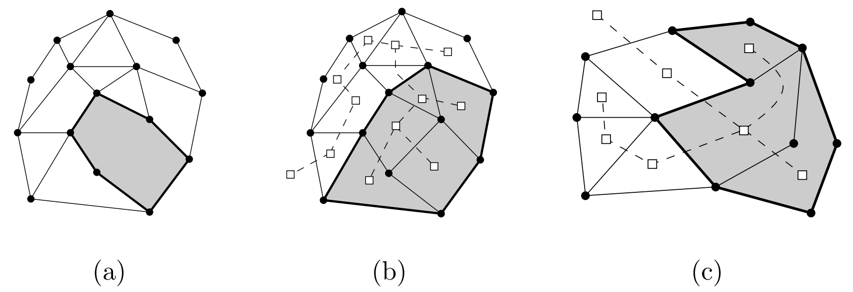

Figure 4: (a) A heavy face. (b) A heavy subtree. (c) Splitting a heavy subtree.

3.2.1 Checking for Heavy Faces

In order to check whether there exists a face f in G with a boundary of size at least $\frac{1}{3}|V|$, we represent each face of G as a list of vertices along its boundary. Computing such a representation takes $O(\text{sort}(N))$ I/Os [16]. Then we scan these lists to see whether any of them has length at least $\frac{1}{3}|V|$. In total, this step uses $O(\text{sort}(N))$ I/Os.

3.2.2 Checking for Heavy Subtrees

First we prove that the boundary of $G(v)$ defined by the vertices in $T^*(v)$ is a simple cycle. Consider a subset F' of the faces of an embedded planar graph G, and let H be the subgraph of G that is the union of the boundaries of the faces in F'. Let H^* be the subgraph of the dual G^* of G induced by the vertices that are dual to the faces in F'. We call H^* the *dual* of H. We call graph H *uniform* if H^* is connected. Since for every vertex $v \in T^*$, $T^*(v)$ and $T^* \setminus T^*(v)$ are both connected, $G(v)$ and its complement $\overline{G(v)}$ are both uniform. Using the following lemma, this implies that the boundary of $G(v)$ is a simple cycle.

Lemma 2 (Smith [27]) *Let G' be a subgraph of a biconnected planar graph G. The boundary of G' is a simple cycle if and only if G' and its complement are both uniform.*

The main difficulty in finding a vertex $v \in T^*$ such that $\frac{1}{3}|V| \leq |G(v)| \leq \frac{2}{3}|V|$ is the computation of the sizes $|G(v)|$ of graphs $G(v)$ for all vertices $v \in T^*$. Once this information has been computed, a single scan of the vertex set of T^* is sufficient to decide whether there is a vertex $v \in T^*$ with $\frac{1}{3}|V| \leq |G(v)| \leq \frac{2}{3}|V|$. As $|T^*| = O(N)$, this takes $O(\text{scan}(N))$ I/Os. Given vertex v, the vertices in $T^*(v)$ can be reported in $O(\text{sort}(N))$ I/Os using standard tree computations [12]. Given these vertices, we can apply operation IncidentEdges to find the set E' of edges in G^* with exactly one endpoint in $T^*(v)$. The set $\{e^* : e \in E'\}$ is the boundary of $G(v)$. All that remains is to describe how to compute the sizes of graphs $G(v)$ I/O-efficiently.

Assume that every vertex $v \in T^*$ stores the number $|v^*|$ of vertices on the boundary of face v^*. The basic idea in our algorithm for computing $|G(v)|$ is to sum $|w^*|$ for all descendants w of v in T^*. This can be done by processing T^* level-by-level, bottom-up, and computing for every vertex v, the value $|G(v)| = |v^*| + \sum_{i=1}^{k} |G(w_i)|$, where $w_1, \dots w_k$ are the children of v. By doing this, however, we count certain vertices several times. Below we discuss how to modify the above idea in order to make sure that every vertex is counted only once.

We define the lowest common ancestor $LCA(e)$ of an edge $e \in G$ to be the lowest common ancestor of the endpoints of its dual edge e^* in T^*. For a vertex $v \in T^*$, we define $E(v)$ to be the set of edges in G whose duals have v as their lowest common ancestor. For a vertex v with children $w_1, w_2, \dots w_k$, $E(v)$ consists of all edges on the boundary between v^* and graphs $G(w_1), G(w_2), \dots, G(w_k)$, as well as the edges on the boundary between graphs $G(w_1), G(w_2), \dots, G(w_k)$. Every endpoint of such an edge is contained in more than one subgraph of $G(v)$, and thus counted more than once by the above procedure. The idea in our modification is to define an *overcount* $c_{v,u}$, for every endpoint u of an edge in $E(v)$, which is one less than the number of times vertex u is counted in the sum $S = |v^*| + \sum_{i=1}^{k} |G(w_i)|$. The sum of these overcounts is then subtracted from S to obtain the correct value of $|G(v)|$.

Let $V(v)$ denote the set of endpoints of edges in $E(v)$. A vertex $u \in V(v)$ is counted once for each subgraph in $\{v^*, G(w_1), G(w_2), \dots, G(w_k)\}$ having u on its boundary. Let l be the number of edges in $E(v)$ incident to u. Each such edge is part of the boundary between two of the subgraphs $v^*, G(w_1), \dots, G(w_k)$. Thus, if u is an internal vertex of $G(v)$ (i.e., not on its boundary), there are l such subgraphs, and u is counted l times (see vertex u_1 in Figure 5). Otherwise, if u is on the boundary of $G(v)$, it follows from the uniformity of $G(v)$ and $\bar{G}(v)$ that two of the edges in $G(v)$ incident to v are on the boundary of $G(v)$ (see vertex u_2 in Figure 5). Hence, $l+1$ of the subgraphs $v^*, G(w_1), \dots, G(w_k)$ contain u, and u is counted $l+1$ times. Therefore the overcount $c_{v,u}$ for vertex $u \in V(v)$ is defined as follows:

$$c_{v,u} = \begin{cases} l-1 & \text{if all edges incident to } u \text{ have their LCA in } T^*(v) \\ l & \text{otherwise} \end{cases}$$

We can now compute $|G(v)|$ using the following lemma.

Lemma 3 *For every vertex $v \in T^*$,*

$$|G(v)| = \begin{cases} |v^*| + \sum_{i=1}^{k} |G(w_i)| - \sum_{u \in V(v)} c_{v,u} & \textit{if } v \textit{ is an internal vertex} \\ & \textit{with children } w_1, \dots, w_k. \\ |v^*| & \textit{if } v \textit{ is a leaf} \end{cases}$$

Proof: The lemma obviously holds for the leaves of T^*. In order to prove the lemma for an internal vertex v of T^*, we have to show that we count every

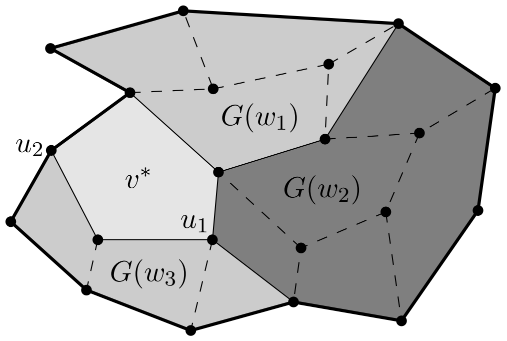

Figure 5: The boundary of graph $G(v)$ is shown in bold. The LCAs of these edges are ancestors of v in T^*. Thin solid edges are those on the boundary between graphs v^*, $G(w_1)$, $G(w_2)$, and $G(w_3)$. The LCA of these edges in T^* is v. The LCAs of dashed edges are descendants of v^*. Vertex u_1 is counted three times in the sum $S = |v^*| + |G(w_1)| + |G(w_2)| + |G(w_3)|$ because it is in v^*, $G(w_2)$, and $G(w_3)$. It has three incident edges with LCA v, and all edges incident to u_1 have their LCA in $T^*(v)$. Hence, its overcount c_{v,u_1} is 2, so that by subtracting c_{v,u_1} from S, vertex u_1 is counted only once. Vertex u_2 is counted twice in S, because it is in v^* and $G(w_3)$. It has one incident edge with LCA v, but not all of its incident edges have their LCA in $T^*(v)$ (it is on the boundary of $G(v)$). Hence, its overcount c_{v,u_2} is one, so that by subtracting c_{v,u_2} from S, vertex u_2 is counted only once.

vertex in $G(v)$ exactly once in the sum $|v^*| + \sum_{i=1}^{k} |G(w_i)| - \sum_{u \in V(v)} c_{v,u}$. A vertex in $G(v) \setminus V(v)$ is counted once, since it is contained in only one of the graphs $v^*, G(w_1), \ldots, G(w_k)$. A vertex $u \in V(v)$ is included in the sum $|v^*| + \sum_{i=1}^{k} |G(w_i)|$ once for every graph v^* or $G(w_i)$ containing it. If all edges incident to u have their LCA in $T^*(v)$, then all faces around u are in $G(v)$. That is, $G(v)$ is an internal vertex of $G(v)$. As argued above, u is counted l times in this case, where l is the number of edges in $E(v)$ incident to u. Thus, it is overcounted $l-1$ times, and we obtain the exact count by subtracting $c_{v,u} = l-1$. Otherwise, u is on the boundary of $G(v)$ and, as argued above, it is counted $l+1$ times. Thus, we obtain the correct count by subtracting $c_{v,u} = l$. □

We are now ready to show how to compute $|G(v)|$, for all $v \in T^*$, I/O-efficiently. Assuming that every vertex $v \in T^*$ stores $|v^*|$ and $c_v = \sum_{u \in V(v)} c_{v,u}$, the graph sizes $|G(v)|$, $v \in T^*$, can be computed in $O(\text{sort}(N))$ I/Os basically as described earlier: For the leaves of T^*, we initialize $|G(v)| = |v^*|$. Then we process T^* level by level, from the leaves towards the root, and compute for every internal vertex v with children $w_1, \ldots, w_k$, $|G(v)| = |v^*| + \sum_{i=1}^{k} |G(w_i)| - c_v$. It remains to show how to compute c_v, $v \in T^*$.

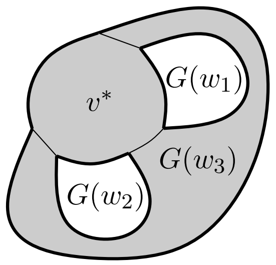

Figure 6: The boundary of $v^* \cup G(w_3)$ is not a simple cycle.

By the definition of overcounts $c_{v,u}$, $c_v = 2|E(v)| - |V'(v)|$, where $V'(v)$ is the set of vertices $u \in V(v)$ so that all edges in G incident to u have their LCAs in $T^*(v)$. To compute sets $E(v)$, for all $v \in T^*$, we compute the LCAs of all edges in G. As discussed in Section 2, we can do this in $O(\text{sort}(N))$ I/Os [12] because there are $O(N)$ edges in G and $O(N)$ vertices in T^*. By sorting the edges of G by their LCAs, we obtain the concatenation of lists $E(v)$, $v \in T^*$, which we scan to determine $|E(v)|$, for all $v \in T^*$. To compute sets $V'(v)$, for all $v \in T^*$, we apply operation SumEdgeLabels to find for every vertex $u \in G$, the edge incident to u whose LCA is closest to the root. We call the LCA of this edge the MAX-LCA of u. By sorting the vertices in G by their MAX-LCAs, we obtain the concatenation of lists $V'(v)$, $v \in T^*$, which we scan to determine $|V'(v)|$, for all $v \in T^*$.

3.2.3 Splitting a Heavy Subtree

If the previous two steps did not produce a simple cycle $\frac{2}{3}$-separator of G, we have to deal with the case where no vertex $v \in T^*$ satisfies $\frac{1}{3}|V| \le |G(v)| \le \frac{2}{3}|V|$. In this case, there must be a vertex $v \in T^*$ with children $w_1, \dots, w_k$ such that $|G(v)| > \frac{2}{3}|V|$ and $|G(w_i)| < \frac{1}{3}|V|$, for $1 \le i \le k$. Our goal is to compute a subgraph of $G(v)$, consisting of the boundary of v^* and a subset of the graphs $G(w_i)$, whose size is between $\frac{1}{3}|V|$ and $\frac{2}{3}|V|$ and whose boundary is a simple cycle C.

In [17] it is claimed that the boundary of the graph defined by v^* and any subset of graphs $G(w_i)$ is a simple cycle. Unfortunately, as illustrated in Figure 6, this is not true in general. However, as we show below, we can compute a permutation $\sigma : [1, k] \to [1, k]$ such that the boundary of each of the graphs obtained by incrementally "gluing" graphs $G(w_{\sigma(1)}), \dots, G(w_{\sigma(k)})$ onto face v^* is a simple cycle. More formally, we define graphs $H_\sigma(1), \dots, H_\sigma(k)$ as $H_\sigma(i) = v^* \cup \bigcup_{j=1}^{i} G(w_{\sigma(j)})$. Then we show that $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform, for all $1 \le i \le k$. This implies that the boundary of $H_\sigma(i)$ is a simple cycle, by Lemma 2. Given the size $|v^*|$ of face v^* and the sizes $|G(w_1)|, \dots, |G(w_k)|$ of graphs $G(w_1), \dots, G(w_k)$, the sizes $|H_\sigma(1)|, \dots, |H_\sigma(k)|$

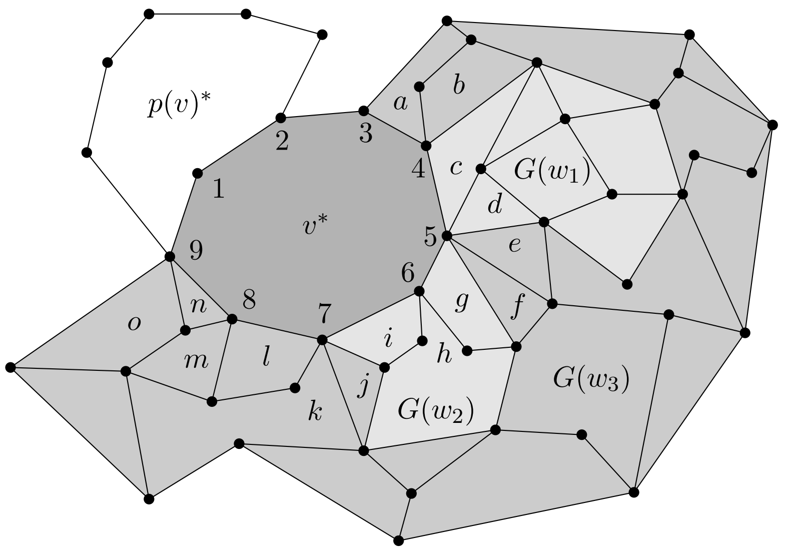

Figure 7: The graph $G(v)$ is shaded. Let $\sigma(i) = i$. Different shades represent the different subgraphs $G(w_1)$, $G(w_2)$ and $G(w_3)$ of $G(v)$. The vertices on the boundary of v^* are numbered clockwise around v^*, starting at the endpoint of an edge shared by v^* and $p(v)^*$. The faces in $G(v)$ incident to the boundary of v^* are labeled with small letters.

of graphs $H_\sigma(1), \ldots, H_\sigma(k)$ can be computed in $O(\text{sort}(N))$ I/Os using a procedure similar to the one applied in the previous section for computing the sizes $|G(v)|$ of graphs $G(v)$, $v \in T^*$. Since $|G(v)| > \frac{2}{3}|V|$ and $|G(w_i)| < \frac{1}{3}|V|$ for all $1 \leq i \leq k$, there must exist a graph $H_\sigma(i)$ such that $\frac{1}{3}|V| \leq |H_\sigma(i)| \leq \frac{2}{3}|V|$. It remains to show how to compute the permutation σ I/O-efficiently.

To construct σ, we extract $G(v)$ from G, label every face in $G(w_i)$ with i, and all other faces of $G(v)$ with 0. This labeling can be computed in $O(\text{sort}(N))$ I/Os by processing $T^*(v)$ from the root towards the leaves. Next we label every edge in $G(v)$ with the labels of the two faces on each side of it. Given the above labeling of the faces in $G(v)$ (or vertices in $T^*(v)$), this labeling of the edges in $G(v)$ can be computed in $O(\text{sort}(N))$ I/Os by applying operation Copy-VertexLabels to the dual graph $G^*(v)$ of $G(v)$. Now consider the vertices $v_1, \ldots, v_t$ on the boundary of v^* in their order of appearance clockwise around v^*, starting at an endpoint of an edge shared by v^* and the face corresponding to v's parent $p(v)$ in T^* (see Figure 7). As in Section 3.1, we can compute this order in $O(\text{sort}(N))$ I/Os using the Euler tour technique and list ranking [12]. For every vertex v_i, we construct a list L_i of edges around v_i in clockwise order, starting with edge $\{v_{i-1}, v_i\}$ and ending with edge $\{v_i, v_{i+1}\}$. These lists can be extracted from the embedding of G in $O(\text{sort}(N))$ I/Os. Let L be the concate-

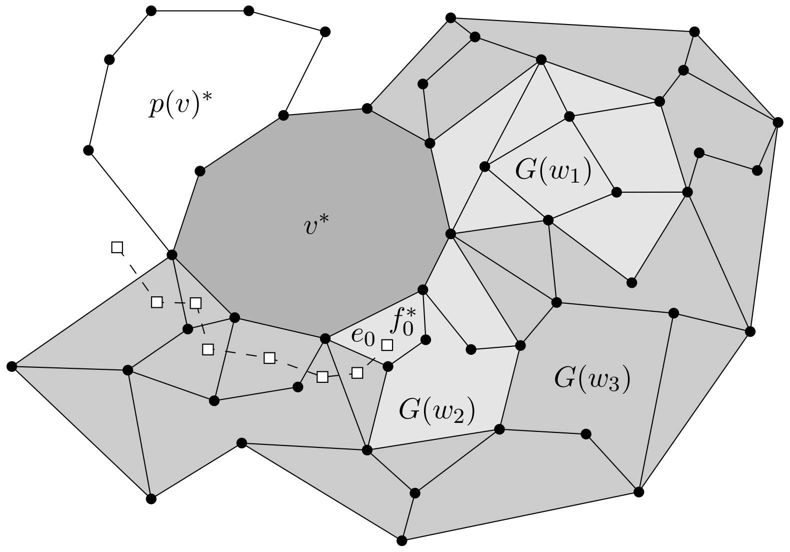

Figure 8: The dashed path is the path from vertex f_0^* to a vertex in the dual of $\overline{G(v)}$ constructed in the proof of Lemma 4, assuming that $j = 2$.

nation of lists $L_1, L_2, \ldots, L_t$. For an edge e in L incident to a vertex v_i, let f_1 and f_2 be the two faces on each side of e, where f_1 precedes f_2 in clockwise order around v_i. We construct a list F of face labels from L by considering the edges in L in their order of appearance and appending the non-zero labels of faces f_1 and f_2 in this order to F. (Recall that faces in $G(w_i)$ are labeled with number i.) This takes $O(\text{scan}(N))$ I/Os. List F consists of integers between 1 and k. Some integers may appear more than once, and the occurrences of integer i are not necessarily consecutive. (This happens if the union of v^* with a subgraph $G(w_i)$ encloses another subgraph $G(w_j)$.) For the graph $G(v)$ in Figure 7,

$$\begin{aligned} F \quad = \quad & \langle \underbrace{3,3}_{\text{v. 3}}, \underbrace{3,3,3,3,1,1}_{\text{vertex 4}}, \underbrace{1,1,1,1,3,3,3,3,2,2}_{\text{vertex 5}}, \underbrace{2,2,2,2,2,2}_{\text{vertex 6}}, \underbrace{2,2,3,3,3,3,3,3}_{\text{vertex 7}}, \\ & \underbrace{3,3,3,3,3,3}_{\text{vertex 8}}, \underbrace{3,3,3,3}_{\text{vertex 9}} \rangle. \end{aligned}$$

We construct a final list S by removing all but the last occurrence of each integer from F. (Intuitively, this ensures that if the union of v^* and $G(w_i)$ encloses another subgraph $G(w_j)$, then j appears before i in S; for the graph in Figure 7, $S = \langle 1, 2, 3 \rangle$.) List S can be computed from list F in $O(\text{sort}(N))$ I/Os using operation DuplicateRemoval. List S contains each of the integers 1 through k exactly once and thus defines a permutation $\sigma : [1, k] \to [1, k]$, where $\sigma(i)$ equals the i-th element in S. It remains to show the following lemma.

Lemma 4 *For all $1 \leq i \leq k$, graphs $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform.*

Proof: First note that every graph $H_\sigma(i)$ is uniform because every subgraph $G(w_j)$ is uniform and there is an edge between v and w_j in G^*, for $1 \leq j \leq k$. To show that every graph $\overline{H_\sigma(i)}$ is uniform, i.e., that its dual is connected., we first observe that $\overline{G(v)}$ is uniform and every subgraph $G(w_j)$, $1 \leq j \leq k$, is uniform. Graph $\overline{H_\sigma(i)}$ is the union of a subset of these graphs. Hence, if its dual is disconnected, there has to be a subgraph $G(w_j) \subseteq \overline{H_\sigma(i)}$ so that its dual and the dual of $\overline{G(v)}$ are in different connected components of the dual of $\overline{H_\sigma(i)}$. Since $G(w_j) \subseteq \overline{H_\sigma(i)}$, $j = \sigma(h)$, for some $h > i$.

Now recall the computation of permutation σ (see Figure 8). Let L be the list of edges clockwise around face v^*, as in our construction, let e_0 be the last edge of $G(w_j)$ in L, and let f_0 be the face of $G(w_j)$ that precedes edge e_0 in the clockwise order around v^*. Then for every subgraph $G(w_{j'})$ that contains an edge that succeeds e_0 in L, $j' = \sigma(h')$, for some $h' > h$. Hence, the following path in G^* from f_0^* to a vertex in the dual of $\overline{G(v)}$ is completely contained in the dual of $\overline{H_\sigma(i)}$: We start at vertex f_0^* and follow the edge e_0^* dual to e_0. For every subsequent vertex f^* that has been reached through an edge e^*, where $e \in L$, either f is a face of $\overline{G(v)}$, and we are done, or we follow the dual of the edge e' that succeeds e in L. This traversal of G^* finds a vertex in the dual of $\overline{G(v)}$ because if it does not encounter a vertex in the dual of $\overline{G(v)}$ before, it will ultimately reach vertex $p(v)$, which is in the dual of $\overline{G(v)}$.

This shows that the duals of $\overline{G(v)}$ and $G(w_j)$ are in the same connected component of the dual of $\overline{H_\sigma(i)}$, for every graph $G(w_j) \subseteq \overline{H_\sigma(i)}$, so that the dual of $\overline{H_\sigma(i)}$ is connected and $\overline{H_\sigma(i)}$ is uniform. □

4 Reducing Depth-First Search to Breadth-First Search

In this section, we give an I/O-efficient reduction from DFS in an embedded planar graph G to BFS in its "vertex-on-face graph", using ideas from [15]. The idea is to use BFS to partition the faces of G into levels around a source face that has the source s of the DFS on its boundary, and then "grow" the DFS-tree level by level around that face.

In order to obtain a partition of the faces of G into levels around the source face, we define a graph which we call the *vertex-on-face graph* $G^\dagger$ of G. As before, let $G^* = (V^*, E^*)$ denote the dual of graph G; recall that each vertex f^* in V^* corresponds to a face f in G. The vertex set of the vertex-on-face graph $G^\dagger$ is $V \cup V^*$; the edge set contains an edge (v, f^*) if vertex v is on the boundary of face f (see Figure 10a). We will show how a BFS-tree of $G^\dagger$ can be used to obtain a partition of the faces in G such that the source face is at level 0, all faces sharing a vertex with the source face are at level 1, all faces sharing a vertex with a level-1 face—but not with the source face—are at level 2, and so on (Figure 9a). Let G_i be the subgraph of G defined as the union of the

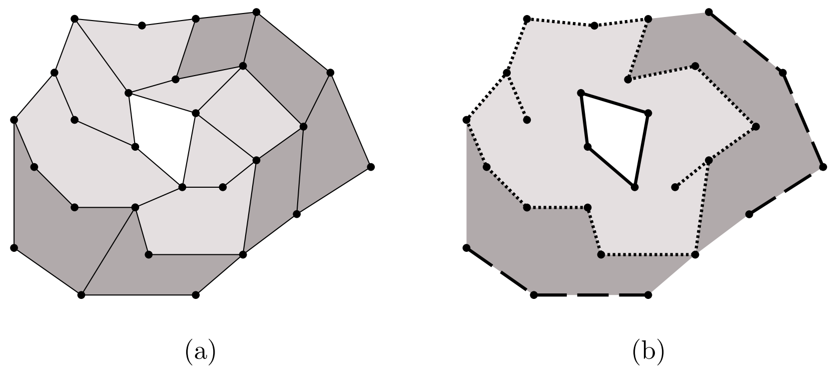
Figure 9: (a) A graph G with its faces colored according to their levels; The level-0 face is white, level-1 faces are light gray, level-2 faces are dark gray. (b) Graphs H_0 (solid), H_1 (dotted), and H_2 (dashed).

boundaries of faces at level at most i, and let $H_i = G_i \setminus G_{i-1}$, for $i > 0$. (The difference $G_i \setminus G_{i-1}$ of graphs G_i and G_{i-1} is the subgraph of G_i with vertex set $V(G_i) \setminus V(G_{i-1})$ and whose edge set contains all edges of G_i that have both endpoints in $V(G_i) \setminus V(G_{i-1})$; see Figure 9b.) For $i = 0$, we define $H_0 = G_0$. We call the vertices and edges of H_i *level-i vertices* and *edges*. An edge $\{v, w\}$ connecting two vertices $v \in H_i$ and $w \in G_{i-1}$ is called an *attachment edge* of H_i. The edges of G_{i-1} and H_i together with the attachment edges of H_i form a partition of the edges of G_i. The basic idea in our algorithm is to grow the DFS-tree by walking clockwise[1] from s around the level-0 face G_0 until we reach the counterclockwise neighbor of s. The resulting path is a DFS tree T_0 for G_0. Next we build a DFS-tree for H_1 and attach it to T_0 through an attachment edge of H_1 in a way that does not introduce cross-edges. Hence, the result is a DFS-tree T_1 for G_1. We repeat this process until we have processed all levels $H_0, \ldots, H_r$ obtaining a DFS-tree T for G (see Figure 11). The key to the efficiency of the algorithm lies in the simple structure of graphs $H_0, \ldots, H_r$. Below we give the details of our algorithm and prove the following theorem.

Theorem 3 *Let G be an undirected embedded planar graph, $G^\dagger$ its vertex-on-face graph, and f_s a face of G containing the source vertex s. Given a BFS-tree of $G^\dagger$ rooted at f_s^*, a DFS tree of G rooted at s can be computed in $O(sort(N))$ I/Os and linear space.*

First consider the computation of graphs $G_1, \ldots, G_r$ and $H_1, \ldots, H_r$. We start by computing graph $G^\dagger$ in $O(\text{sort}(N))$ I/Os as follows: First we compute a representation of G consisting of a list of vertices clockwise around each face of G. Such a representation can be computed in $O(\text{sort}(N))$ I/Os [16]. Then we add a face vertex f^*, for every face f of G, and connect f^* to all vertices

[1]A *clockwise* walk on the boundary of a face means walking so that the face is to our right.

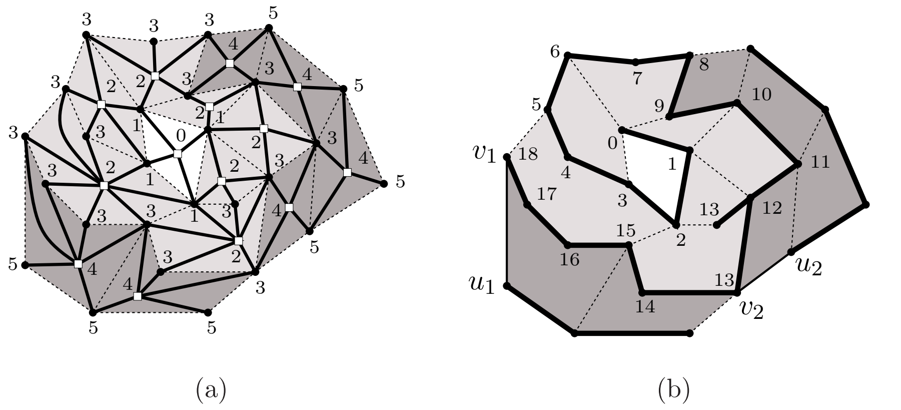

Figure 10: (a) $G^\dagger$ shown in bold; numbers represent BFS-depths in $G^\dagger$. (b) T_1, H_2 and attachment edges $\{u_i, v_i\}$.Vertices in T_1 are labeled with their DFS-depths.

in the vertex list representing f. This requires a single scan of the vertex lists representing the faces of G. The levels of the faces of G can now be obtained from a BFS-tree of the vertex-on-face graph $G^\dagger$, rooted at the dual vertex f_s^* of a face f_s that contains s: Every vertex of G is at an odd level in the BFS-tree; every dual vertex corresponding to a face of G is at an even level (Figure 10a). The level of a face is the level of the corresponding vertex in the BFS-tree divided by two. The vertex set $V(H_i)$ of graph H_i contains all vertices of G at distance $2i+1$ from f_s^* in $G^\dagger$. Hence, we can obtain a partition of $V(G)$ into vertex sets $V(H_1), \dots, V(H_r)$ by sorting the vertices in $V(G)$ by their distances from f_s^* in $G^\dagger$. The vertex set $V(G_i)$ of graph G_i is $V(G_i) = \bigcup_{j=0}^{i} V(H_i)$. An edge $e \in G$ is in H_i if both its endpoints are at distance $2i+1$ from f_s^* in $G^\dagger$. For an attachment edge $\{v, w\}$ of H_i, v is at distance $2i+1$, and w is at distance $2i-1$ from f_s^* in $G^\dagger$. Thus, we can obtain a partition of $E(G)$ into sets $E(H_0), \dots, H(H_r)$ and the sets of attachment edges of graphs $H_1, \dots, H_r$ by sorting the edges in $E(G)$ in inverted lexicographical order defined by the distances of their endpoints from f_s^* in $G^\dagger$.

Next we discuss a few simple properties of graphs G_i and H_i, which we use to prove the correctness of our algorithm. For every edge in G_{i-2}, as well as for every attachment edge of H_{i-1}, the two faces on both sides of the edge are at level at most $i-1$. Thus, they cannot be boundary edges for G_{i-1}. It follows that the boundary edges of G_{i-1} are in $E(H_{i-1})$. Consequently, all boundary vertices of G_{i-1} are in $V(H_{i-1})$. As G_{i-1} is a union of faces, its boundary consists of a set of cycles, called the *boundary cycles* of G_{i-1}. Graph H_i lies entirely "outside" the boundary of G_{i-1}, i.e., in $\overline{G_{i-1}}$. Hence, all attachment edges of H_i are connected only to boundary vertices of G_{i-1}, i.e., vertices of H_{i-1}. Finally, note that graph G_i is uniform. This can be shown as follows:

Graph G_i corresponds to the first $2i$ levels of the BFS-tree of $G^\dagger$. For a level-$(i-1)$ face f_1 and a level-i face f_2 that share a vertex v, graph G_i contains all faces incident to v. Hence, there is a path from f_1^* to f_2^* in G_i^*. Applying this argument inductively, we obtain that there is a path in G_i^* from f_s^* to every vertex of G_i^*, which shows that G_i is uniform. On the other hand, graph $\overline{G_{i-1}}$, and thus H_i, is not necessarily uniform.

We are now ready to describe the details of our algorithm for constructing a DFS-tree for G by repeatedly growing a DFS-tree T_i for G_i from a DFS-tree T_{i-1} for G_{i-1}, starting with the DFS-tree T_0 for G_0. During the algorithm we maintain the following two invariants (see Figure 10b):

(i) Every boundary cycle C of G_{i-1} contains exactly one edge e not in T_{i-1}. One of the two endpoints of that edge is an ancestor in T_{i-1} of all other vertices in C.

(ii) The depth of each vertex in G_{i-1}, defined as the distance from s in T_{i-1}, is known.

Assume we have computed a DFS-tree T_{i-1} for G_{i-1}. Our goal is to compute a DFS-forest for H_i and link it to T_{i-1} through attachment edges of H_i without introducing cross-edges, in order to obtain a DFS-tree T_i for G_i. If we can compute a DFS-forest of H_i in $O(\text{sort}(|H_i|))$ I/Os and link it to T_{i-1} in $O(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os, the overall computation of a DFS-tree T for G uses $O\left(\text{sort}(|H_0|) + \sum_{i=1}^{r} \text{sort}(|H_{i-1}| + |H_i|)\right) = O\left(\sum_{i=0}^{r} \frac{2|H_i|}{B} \log_{M/B} \frac{N}{B}\right) = O(\text{sort}(N))$ I/Os. Next we show how to perform both computations in the desired number of I/Os.

Let $H_1', \dots, H_k'$ be the connected components of H_i. They can be computed in $O(\text{sort}(|H_i|))$ I/Os [12]. For every component H_j', we find the deepest vertex v_j on the boundary of G_{i-1} such that there is an attachment edge $\{u_j, v_j\}$ of H_i with $u_j \in H_j'$. Then we compute a DFS-tree T_j' of H_j' rooted at u_j and attach T_j' to T_{i-1} using edge $\{u_j, v_j\}$. Let T_i be the resulting tree.

Lemma 5 *Tree T_i is a DFS-tree of G_i.*

Proof: Tree T_i is a spanning tree of G_i, since T_{i-1} is a DFS-tree for G_{i-1}, trees $T_1', \dots, T_k'$ are DFS-trees of the connected components of H_i, and each tree T_j' is connected to T_{i-1} by a single edge. Now let $\{v, w\}$ be a non-tree edge of G_i. As there are no edges between different connected components of H_i in G_i, either $v, w \in H_j'$, for some $1 \le j \le k$, $v, w \in G_{i-1}$, or w.l.o.g. $v \in H_j'$, for some $1 \le j \le k$, and $w \in G_{i-1}$. In the first two cases, edge $\{v, w\}$ is a back edge, since trees T_{i-1} and T_j' are DFS-trees for G_{i-1} and H_j', respectively. In the latter case, $\{v, w\}$ is a back-edge because v is a descendant of u_j, and, by Invariant (i), w must be an ancestor of v_j on the boundary cycle of G_{i-1} enclosing H_j'. □

We can compute tree T_i from tree T_{i-1} in $O(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os: First we find the attachment edges $\{u_1, v_1\}, \dots, \{u_k, v_k\}$ connecting graphs $H_1', \dots, H_k'$ to G_{i-1}. This can be done using a procedure similar to the one used

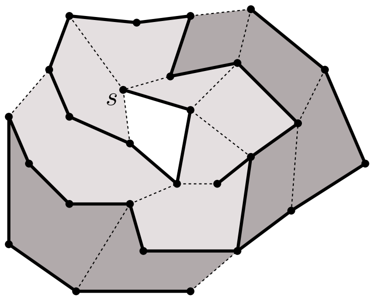

Figure 11: The DFS-tree of G (Figure 9 and Figure 10)

in Section 3.1. As the attachment edges of H_i are the edges of a planar graph with vertex set $V(H_{i-1}) \cup V(H_i)$, this procedure takes $O(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os. All that remains is to show how to compute a DFS-tree T'_j rooted at u_j, for each connected component H'_j of H_i. The key to doing this I/O-efficiently is the following lemma, which shows that H_i has a simple structure.

Lemma 6 *The non-trivial bicomps of H_i are the boundary cycles of G_i.*

Proof: Consider a cycle C in H_i. All faces incident to C are at level i or greater. Thus, since G_{i-1} is uniform, all its faces are either inside or outside C. Assume w.l.o.g. that G_{i-1} is inside C. Then none of the faces outside C shares a vertex with a level-$(i-1)$ face. That is, all faces outside C must be at level at least $i+1$, which means that C is a boundary cycle of G_i.

Every bicomp that is not a cycle contains at least three internally vertex-disjoint paths P_1, P_2, and P_3 with the same endpoints v and w. As we have just shown, the graph $C_1 = P_1 \cup P_3$ is a boundary cycle of G_i, as is the graph $C_2 = P_1 \cup P_2$. Let $\{v, x\}$ be the first edge of P_2, and $\{y, w\}$ be the last edge of P_2. Since C_1 is a boundary cycle of G_i, G_i is either completely inside or completely outside C_1. Since C_1 is a subgraph of H_i, all faces incident to C_1 that are on the same side of C_1 as G_i are at level i because all faces on the other side of C_1 are at level at least $i+1$. Hence, if P_2 is on the same side of C_1 as G_i, the four faces incident to edges $\{v, x\}$ and $\{y, w\}$ are at level i, which contradicts the fact that C_2 is a boundary cycle of G_i. If P_2 is on the other side of C_1, the four faces incident to edges $\{v, x\}$ and $\{y, w\}$ are at level at least $i+1$, which contradicts the fact that edges $\{v, x\}$ and $\{y, w\}$ are at level i. Thus, every bicomp of H_i consists of a single boundary cycle. □

In order to compute a DFS-tree of H'_j rooted at u_j, we first partition H'_j into its bicomps. This takes $O(\text{sort}(|H'_j|))$ I/Os [12]. Then, as in Section 3, we construct the bicomp-cutpoint-tree of H'_j, rooted at the bicomp containing u_j. For each bicomp K, we determine the parent cutpoint x. If K is a trivial bicomp (i.e., consists of a single edge), the DFS-tree T_K of K consists of the single edge in K. Otherwise, by Lemma 6, K is a cycle. Let y be a neighbor of x in K. This neighbor can be computed in a single scan of the edge set of K. To obtain a DFS-tree T_K of K rooted at x, we remove edge $\{x, y\}$ from K. The DFS-tree T'_j of H'_j is the union of DFS-trees T_K of all bicomps K of H'_j. Note that T_K is a path from x to y, and all vertices along this path are descendants of x. Since the non-trivial bicomps of H_i are the boundary cycles of G_i, Invariant (i) is hence maintained after attaching the resulting DFS-trees $T'_1, \ldots, T'_k$ to T_{i-1}.

Finally, to maintain Invariant (ii), we have to determine the depth of each vertex in T_i. The depth of vertices in $T_{i-1} \subseteq T_i$ do not change by adding trees $T'_j, \ldots, T'_k$ to T_{i-1}. The depths of the vertices in H_i can be computed as follows: Every vertex u_j has depth one more than the depth of $v_j \in T_{i-1}$. The depths of all other vertices in T'_j can be computed from the depth of u_j in $O(\text{sort}(|T'_j|))$ I/Os by performing a DFS-traversal of T'_j. Hence, this computation takes $O(\text{sort}(|H_i|))$ I/Os, for all trees in the DFS-forest of H_i.

This concludes the description of our reduction from planar DFS to planar BFS, and thus the proof of Theorem 3. The following corollary is an immediate consequence of Theorem 3 and recent results of [6, 22].

Corollary 1 *A DFS-tree of an embedded planar graph can be computed in $O(sort(N))$ I/O operations and linear space.*

5 Conclusions

In this paper, we have developed the first $o(N)$ I/O and linear space algorithm for DFS in embedded planar graphs. We have also designed an $O(\text{sort}(N))$ I/O reduction from planar DFS to planar BFS, proving that external memory planar DFS is not harder than planar BFS. Together with recent results of [6, 22], this leads to an algorithm that computes a DFS-tree of an embedded planar graph in $O(\text{sort}(N))$ I/Os.

References

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[2] P. Agarwal, L. Arge, M. Murali, K. Varadarajan, and J. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 117–126, 1998.

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

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

[5] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 1004 of *Lecture Notes in Computer Science*, pages 82–91, 1995.

[6] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proc. Scandinavian Workshop on Algorithms Theory 2000*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447, 2000.

[7] L. Arge and U. Meyer and L. Toma and N. Zeh. On External-Memory Planar Depth First Search, In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 471–482, 2001.

[8] L. Arge, L. Toma, and J. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proceedings of the Workshop on Algorithm Engineering and Experimentation*, 2000.

[9] A. Broder, R. Kumar, F. Manghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web: Experiments and models. In *Proceedings of the ninth WWW Conference*, 2000. Available at http://www9.org/w9cdrom/index.html.

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

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

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

[13] F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graphs problems. *Communications of the ACM*, 25(9):659–665, 1982.

[14] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 416–425, 1993.

[15] T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, 1990.

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

[17] J. JáJá and R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, 1988.

[18] J. F. JáJá. *An introduction to parallel algorithms*, chapter 5, pages 222–227. Addison-Wesley, Reading, MA, 1992.

[19] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Sumposium on Parallel and Distributed Computing*, pages 169–177, 1996.

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

[21] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 89–90, 2001.

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

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

[24] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88, 2001.

[25] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32:265–279, 1986.

[26] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[27] J. R. Smith. Parallel algorithms for depth-first searches I. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, 1986.

[28] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–159, 1972.

[29] R. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[30] U. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellligence*, 3:331–360, 1991.