Alcom-FT Technical Report Series
ALCOMFT-TR-01-93

On External-Memory Planar Depth First Search

Lars Arge[1,⋆], Ulrich Meyer[2,⋆⋆], Laura Toma[1,⋆⋆⋆], and 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 breadth-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 efficient depth-first search algorithm and show how planar depth-first search in general can be reduced to planar breadth-first search. As part of the first result we develop the first I/O-efficient algorithm for finding a simple cycle separator of a biconnected planar graph. Together with other recent reducibility results, the second result provides further evidence that external memory breadth-first search is among the hardest problems on planar graphs.

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 nodes and 2 billion edges, and recent work in web modeling uses depth-first search, breadth-first search, shortest path computation and connected component computation as primitive routines for investigating the structure of the web [5]. 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 20TB phone-call data graph [7].

⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879 and CAREER grant EIA–9984099.

⋆⋆ Supported in part by the IST Programme of the EU under contract number IST–1999–14186 (ALCOM–FT). Part of this work was done while visiting Duke University.

⋆⋆⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734 and CAREER grant EIA–9984099.

† Supported in part by NSERC and NCE GEOIDE research grants. Part of this work was done while visiting Duke University.

When working with such massive graphs the I/O-communication, and not the internal memory computation time, is often the bottleneck. Designing I/O-efficient algorithms can thus lead to considerable runtime 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 both strategies can be implemented in linear time in internal memory; still they reveal important information about the structure of the input 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. In this paper we consider an important class of sparse graphs, namely *undirected embedded planar graphs*. This class is restricted enough to hope for more efficient algorithms than for arbitrary sparse graphs. Several such algorithms have indeed been obtained recently [3, 15]. We develop an improved DFS algorithm for planar graphs and show how planar DFS can be reduced to planar BFS. Since several other problems on planar graphs have also been shown to be reducible to BFS, this provides further evidence that in external memory BFS is among the hardest problems on planar graphs.

1.1 I/O-Model and Previous Results

We work in the standard two-level I/O model with one (logical) disk [1] (our results work in a D-disk model; but for brevity we only consider one disk in this extended abstract). The model defines the following parameters:

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

where $2B < M < N$. An *Input/Output* operation (or simply *I/O*) involves reading (or writing) a block from (to) disk into (from) internal memory. Our 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 $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ (the *sorting* bound) [1]. 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 very significant [8, 4].

I/O-efficient graph algorithms have been considered by a number of authors. For a review see [19] and the references therein. We review the previous results most relevant to our work. The best previously known general DFS algorithms on undirected graphs use $O\big((|V| + (|E|/B)) \log_2 |V|\big)$ I/Os [12] or $O\big(|V| + (|V|/M) \cdot (|E|/B)\big)$ I/Os [8]. Since the best known general BFS algorithm uses only $O(|V| + (|E|/|V|)\text{sort}(|V|)) = O(|V| + \text{sort}(|E|))$ I/Os [17], this suggests that on undirected graphs DFS might be harder than BFS. For directed graphs the best known algorithms for BFS and DFS both use $O((|V| + |E|/B) \cdot \log(|V|/B) + \text{sort}(|E|))$ I/Os [6]. In general we cannot hope to design algorithms that perform less than $\Omega(\min(|V|, \mathrm{sort}(|V|)))$ I/Os for either of the two problems [2, 8, 17]. As mentioned above, in practice $O(\min(|V|, \mathrm{sort}(|V|))) = O(\mathrm{sort}(|V|))$. Still, all of the above algorithms use $\Omega(|V|)$ I/Os. For planar graphs this bound is matched by the standard internal memory algorithms.

Recently, the first $o(N)$ DFS and BFS algorithms for undirected planar graphs were developed [15]. These algorithms use $O(\frac{N}{\gamma \log B} + \mathrm{sort}(NB^{\gamma}))$ I/Os and $O(NB^{\gamma})$ space, for any $0 < \gamma \leq 1/2$. Further improved algorithms have been developed for special classes of planar graphs. For trees, $O(\mathrm{sort}(N))$ I/O algorithms are known for both BFS and DFS—as well as for Euler tour computation, expression tree evaluation, topological sorting, and several other problems [6, 8, 3]. BFS and DFS can also be solved in $O(\mathrm{sort}(N))$ I/Os on outerplanar graphs [13] and on k-outerplanar graphs [14]. Developing $O(\mathrm{sort}(N))$ I/O DFS and BFS algorithms for arbitrary planar graphs remains a challenging open problem.

1.2 Our Results

The contribution of this paper is two-fold. In Sec. 2 we present a new DFS algorithm for undirected embedded planar graphs that uses $O(\mathrm{sort}(N) \log N)$ 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 such algorithm using linear space. The algorithm is based on a divide-and-conquer approach first proposed in [18]. It utilizes a new $O(\mathrm{sort}(N))$ 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 edges of the graph. Previously, no such algorithm was known.

In Sec. 3 we use ideas similar to the ones utilized in [9] to obtain an $O(\mathrm{sort}(N))$ I/O reduction from DFS to BFS on undirected embedded planar graphs. Contrary to what has been conjectured for general graphs, this shows that for planar graphs BFS is as hard as DFS. A recent paper shows that given a BFS-tree of a planar graph, the single source shortest path problem as well as the multi-way separation problems can be solved in $O(\mathrm{sort}(N))$ I/Os [3]. Together, these results suggest that BFS may indeed be a universally hard problem for planar graphs. That is, if planar BFS can be performed I/O-efficiently, most other problems on planar graphs can also be solved I/O-efficiently.

2 DFS using Simple Cycle Separators

2.1 Outline of the Algorithm

Our new $O(\mathrm{sort}(N) \log N)$ I/O and linear space algorithm for computing a DFS tree of a planar graph is based on a divide-and-conquer approach first proposed in [18].

A *cutpoint* of a graph G is a vertex whose removal disconnects G. We first consider the case where G is *biconnected*, i.e., does not contain any cutpoints. In

Sec. 2.2 we show that for a biconnected planar graph G we can compute a *simple cycle α-separator* in $O(\text{sort}(N))$ I/Os (Thm. 2). A simple cycle α-separator C of G is a simple cycle such that neither the subgraph inside nor outside the cycle contains more than $\alpha|E|$ edges. The main idea of our algorithm is to partition G using a simple cycle α-separator, for some constant $0 < \alpha < 1$, recursively compute DFS-trees for the connected components of $G \setminus C$, and combine them to obtain a DFS-tree for G. Given that each recursive step can be realized in $O(\text{sort}(N))$ I/Os, the whole algorithm takes $O(\text{sort}(N) \log N)$ I/Os.

In more detail, we construct a DFS tree T of a biconnected embedded planar graph G, rooted at some vertex s as follows (see Fig. 1): First we compute a simple cycle α-separator C of G in $O(\text{sort}(N))$ I/Os. Then we find a path P from s to some vertex v in C by computing a spanning tree T' of G and finding the closest vertex to s in C along T'. This also takes $O(\text{sort}(N))$ I/Os [8]. Next we extend P to a path P' containing all vertices in P and C. To do so we identify the counter-clockwise neighbor $w \in C$ of v, relative to the last edge on P, remove edge $\{v, w\}$ from C, rank the resulting path to obtain the clockwise order of the vertices in C, and finally concatenate P with the resulting path. All these steps can be performed in $O(\text{sort}(N))$ I/Os [8]. We compute the connected components $H_1, \ldots, H_k$ of $G \setminus P'$ in $O(\text{sort}(N))$ I/Os [8]. For each component H_i, we find the vertex $v_i \in P'$ furthest away from s along P' such that there is an edge $\{u_i, v_i\}$, $u_i \in H_i$. This can easily be done in $O(\text{sort}(N))$ I/Os. Next we recursively compute DFS trees $T_1, \ldots, T_k$ for components $H_1, \ldots, H_k$ and obtain a DFS tree T for G as the union of trees $T_1, \ldots, T_k$, path P', and edges $\{u_i, v_i\}$, $1 \le i \le k$. Note that components $H_1, \ldots, H_k$ are not necessarily biconnected. Below we show how to deal with this case.

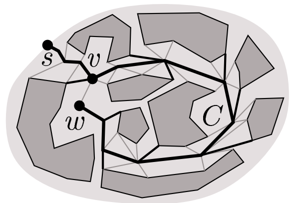


Fig. 1. The path P' is shown in bold. Components are shaded dark grey. Medium edges are edges $\{u_i, v_i\}$. Light edges are non-tree edges.

To see that T is indeed a DFS tree, first note that there are no edges between components $H_1, \ldots, H_k$. For every non-tree edge $\{v, w\}$ connecting a vertex v in a component H_i with a vertex w in P', v is a descendant of u_i and, by the choice of v_i, w is an ancestor of v_i. Thus all non-tree edges in G are back-edges, and T is a DFS tree.

We handle the case where G is not biconnected by finding the *biconnected components* or *bicomps* (i.e., the maximal biconnected subgraphs) of G, computing a DFS tree for each bicomp and joining them at the cutpoints. 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. There is an edge between a cutpoint v and a bicomp vertex $v(C)$ if v is contained in C. We choose a bicomp C_r containing vertex s as the root of T_G. The *parent cutpoint* of a bicomp C is the parent $p(v(C))$ of $v(C)$ in T_G. The *parent bicomp* of C is the bicomp C' corresponding to $v(C') = p(p(v(C)))$. T_G can be constructed in $O(\text{sort}(N))$ I/Os [8]. We compute a DFS tree of C_r rooted at vertex s. In all other bicomps C, we compute a DFS tree rooted at the parent cutpoint of C. The union of the resulting DFS trees is a DFS tree for G rooted at s, as 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 $O(sort(N) \log N)$ I/O operations and linear space.*

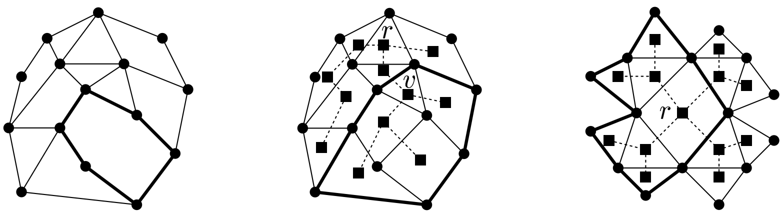
2.2 Finding a Simple Cycle Separator

Utilizing ideas similar to the ones used in [11, 16] we now show how to compute a simple cycle $\frac{5}{6}$-separator for a planar biconnected graph.

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'. The *complement* $\overline{G}_{F'}$ of $G_{F'}$ is the graph obtained as the union of boundaries of all faces in $F \setminus F'$. The *boundary* of $G_{F'}$ is the intersection between $G_{F'}$ and its complement $\overline{G}_{F'}$. 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. We use v^*, e^*, and f^* to refer to the face, edge, and vertex which 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 [10].

The main idea in our algorithm is to find a set of faces $F' \subset F$ such that the boundary of $G_{F'}$ is a simple cycle $\frac{5}{6}$-separator. The main difficulty is to ensure that the boundary of $G_{F'}$ is a simple cycle. We compute F' as follows: First we check whether there is a single face whose boundary has size at least $\frac{|E|}{6}$ (Fig. 2a). If we find such a face, we report its boundary as the separator C. Otherwise, we compute a spanning tree T^* of the dual G^* of G rooted at an arbitrary node r. Every node $v \in T^*$ defines a maximal subtree $T^*(v)$ of T^* rooted at v. The nodes 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 in G. We try to find a node v such that $\frac{1}{6}|E| \le |G(v)| \le \frac{5}{6}|E|$, where $|G(v)|$ is the number of edges in $G(v)$ (Fig. 2b). If we succeed, we report the boundary of $G(v)$. Otherwise, we are left in a situation where for every leaf $l \in T^*$ (face in G^*) we have $|G(l)| < \frac{1}{6}|E|$, for the root r of T^* we have $|G(r)| = |E|$, and for every other vertex $v \in T^*$ either $|G(v)| < \frac{1}{6}|E|$ or $|G(v)| > \frac{5}{6}|E|$. Thus, there has to be a node v with $|G(v)| > \frac{5}{6}|E|$ and $|G(w_i)| < \frac{1}{6}|E|$, for all children $w_1, \dots, w_k$ 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 $G(w_1), \dots, G(w_k)$ such that $\frac{1}{6}|E| \le |G'| \le \frac{5}{6}|E|$, and the boundary of G' is a simple cycle (Fig. 2c). Below we describe our algorithm in detail and show that all of the above steps can be performed in $O(\text{sort}(N))$ I/Os. This proves the following theorem.

Theorem 2. *A simple cycle $\frac{5}{6}$-separator of an embedded biconnected planar graph can be computed in $O(sort(N))$ I/O operations and linear space.*



(a) (b) (c)

Fig. 2. (a) A heavy face. (b) A heavy subtree. (c) Splitting a heavy subtree.

Checking for heavy faces. In order to check if there exists a face f in G with a boundary of size at least $\frac{1}{6}|E|$, 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 [10]. Then we scan these lists to see whether any of them has length at least $\frac{1}{6}|E|$.

Checking for heavy subtrees. First we prove that the boundary of $G(v)$ defined by the nodes in $T^*(v)$ is a simple cycle. A planar graph is *uniform* if its dual is connected. Since for every $v \in T^*$, $T^*(v)$ 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 1 (Smith [18]). *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.*

Next we show how to find a node $v \in T^*$ such that $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$. G^* and T^* can both be computed in $O(\text{sort}(N))$ I/Os [10, 8]. For every node $v \in T^*$, let $|v^*|$ be the number of edges on the boundary of face v^*. Let the *weight* $\omega(G(v))$ of subgraph $G(v)$ be defined as $\omega(G(v)) = \sum_{w \in T^*(v)} |w^*|$. As $\omega(G(v)) = |v^*| + \sum_{i=1}^{k} \omega(G(w_i))$, where $w_1, \ldots, w_k$ are the children of v in T^*, we can process T^* bottom-up to compute the weights of all subgraphs $G(v)$. Using time-forward processing [8], this takes $O(\text{sort}(N))$ I/Os. For a node v in T^* every boundary edge of $G(v)$ is counted once in $\omega(G(v))$; every interior edge is counted twice. This implies that if $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$, then $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$. Thus, we can find a node $v \in T^*$ with $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$ in $O(\text{scan}(N))$ I/Os by scanning through the list of nodes T^* and finding a node v such that $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$, if such a node exists.[1]

[1] Note that even if a node v with $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$ exists in T^*, the algorithm might not find it since it does not follow that $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$. This is not a problem, however, since in this case a simple cycle $\frac{5}{6}$-separator will still be found in the final phase or our algorithm. In the full paper we discuss how to modify the algorithm in order to compute $|G(v)|$ exactly for each node $v \in T^*$. This allows us to find even a simple cycle $\frac{2}{3}$-separator.

Splitting a heavy subtree. We are now in a situation where no vertex $v \in T^*$ satisfies $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$. Thus, there must be a vertex $v \in T^*$ with children $w_1, \ldots, w_k$ such that $\omega(G(v)) > \frac{2}{3}|E|$ and $\omega(G(w_i)) < \frac{1}{3}|E|$, for $1 \leq i \leq k$. Our goal is to compute a subgraph of $G(v)$ consisting of the boundary v^* and a subset of the graphs $G(w_i)$ whose weight is between $\frac{1}{3}|E|$ and $\frac{2}{3}|E|$ and whose boundary is a simple cycle C.

In [11] 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 this is not true in general, as illustrated in Fig. 3. However, as we show below, we can compute a permutation $\sigma : \{1 \ldots k\} \to \{1 \ldots k\}$ such that if we start with v^* and incrementally "glue" $G(w_{\sigma(1)})$, $G(w_{\sigma(2)})$, $\ldots$, $G(w_{\sigma(k)})$ onto face v^*, the boundary of each of the obtained graphs is a simple cycle. More formally, we show that if we define $H_\sigma(i) = v^* \cup \bigcup_{j=1}^{i} G(w_{\sigma(j)})$ then $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform for all $1 \leq i \leq k$. This implies that the boundary of $H_\sigma(i)$ is a simple cycle by Lemma 1. Since we have already computed the sizes $|v^*|$ of faces v^* and the weights $\omega(G(v))$ of all graphs $G(v)$, it takes $O(\text{scan}(N))$ I/Os to compute weights $\omega(H_\sigma(i))$, $1 \leq i \leq k$, and find index i such that $\frac{1}{3}|E| \leq \omega(H_\sigma(i)) \leq \frac{2}{3}|E|$. It remains to show how to compute the permutation σ I/O-efficiently.

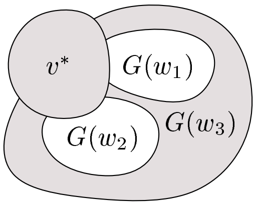


Fig. 3. The boundary of $v^* \cup G(w_3)$ is not a simple cycle.

To construct σ, we extract $G(v)$ from G, label v^* with 0, and label every face in $G(w_i)$ with i. Next we label every edge in $G(v)$ with the labels of the two faces on each side of it. We perform the labeling in $O(\text{sort}(N))$ I/Os using the previously computed representations of G and G^* and a post-order traversal of T^*. Details will appear in the full paper. Now consider the vertices $v_1, \ldots, v_t$ on the boundary of v^* in the order they appear clockwise around v^*, starting at the common endpoint of an edge shared by v^* and the face corresponding to v's parent $p(v)$ in T^*. As in Sec. 2, we can compute this order in $O(\text{sort}(N))$ I/Os using list ranking. For each 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 are easily computed in $O(\text{sort}(N))$ I/Os from the embedding of G. Let L be the concatenation 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 incident to e, where f_1 precedes f_2 in clockwise order around v_i. We construct a list F of face labels from L in $O(\text{scan}(N))$ I/Os by considering the edges in L in order and appending the labels of f_1 and f_2 in this order to F. List F consists of integers between 1 and k. Some integers may appear more than once, and the occurrences of some integer i are not necessarily consecutive. (This happens if the union of v^* with a subgraph $G(w_i)$ encloses another subgraph $G(w_j)$; see Fig. 3.) 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). This takes $O(\text{sort}(N))$ I/Os by sorting and scanning F twice. Again details will appear in the full paper. S contains each of the integers

1 through k exactly once and thus defines a permutation σ. All that remains is to prove the following lemma.

Lemma 2. *For all $1 \le i \le k$, $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform.*

Proof. Every graph $H_\sigma(i)$ is uniform because every subgraph $G(w_j)$ is uniform and w_j is connected to v by an edge in G^*. Next we show that every $\overline{H_\sigma(i)}$ is uniform. To do this we must show that every $\overline{H_\sigma(i)}^*$ is connected. Note that $\overline{G(v)} \subseteq \overline{H_\sigma(i)}$, $\overline{G(v)}$ is uniform, and each graph $G(w_j)$ is uniform. Hence, in order to prove that $\overline{H_\sigma(i)}^*$ is connected, it suffices to show that for all $i < j \le k$, there is a path in $\overline{H_\sigma(i)}^*$ connecting a vertex in $G(w_j)^*$ to a vertex in $\overline{G(v)}^*$. So assume for the sake of contradiction that there is a graph $G(w_j)$, $i < j \le k$, such that there is no such path from a vertex in $G(w_j)^*$ to a vertex in $\overline{G(v)}^*$ in $\overline{H_\sigma(i)}^*$ (Fig. 4). Let $\tilde{G}$ be the uniform component of $\overline{H_\sigma(i)}$ containing $G(w_j)$, and C be the boundary cycle of $\tilde{G}$. Let P be the path obtained by removing the edges shared by v^* and $p(v)^*$ from the boundary cycle of v^*. Let v_1 be the first vertex of C encountered during a clockwise walk along P; let v_2 be the last such vertex. We define P' to be the path obtained by walking clockwise around $\tilde{G}$ starting at v_1 and ending at v_2. Let e_1 be the first and e_2 be the last edge on P'. Edge e_1 separates two faces $f_1 \in H_\sigma(i)$ and $f_2 \in \overline{H_\sigma(i)}$. Similarly, edge e_2 separates two faces $f_3 \in \overline{H_\sigma(i)}$ and $f_4 \in H_\sigma(i)$. Let j_1, j_2, j_3 and j_4 be the labels of these faces. We show that label j_2 appears before label j_4 in S: Assume that label j_2 appears after label j_4 in S. Then there has to be a face f' with label j_2 occurring after face f_4 clockwise around v^*. In particular, face f' is outside cycle C, while face f_2 is inside. As $G^*(w_{j_2})$ is connected there has to be a path from f'^* to f_2^* in $G^*(w_{j_2})$. But this is not possible since every path from f_2^* to f'^* must contain an edge e^*, for some edge $e \in C$, and edge e^* cannot be in $G^*(w_{j_2})$ because one of its endpoints is in $H_\sigma^*(i)$. Therefore it follows that label j_2 appears before label j_4 in S. But this means means that f_2 is being added to $H_\sigma(i)$ before f_4, contradicting the assumption that $f_2 \in \overline{H_\sigma(i)}$ and $f_4 \in H_\sigma(i)$. $\square$

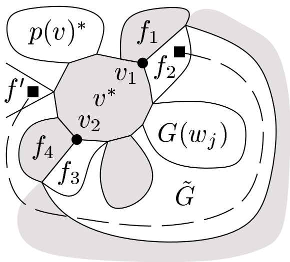


Fig. 4. Grey regions are in $H_\sigma(i)$.

3 Reducing DFS to BFS

This section gives an I/O-efficient reduction from DFS in an embedded planar graph G to BFS in its *vertex-on-face graph*, using ideas from [9]. The vertex-on-face graph $G^\dagger$ of G is defined as follows: The vertex set of $G^\dagger$ is $V \cup V^*$; there is an edge (v, f^*) in $G^\dagger$ if v is on the boundary of face f. The graph $G^\dagger$ can be computed from G in $O(\text{sort}(N))$ I/Os in a way similar to the computation of the dual G^* of G. We use the vertex-on-face graph instead of the graph used in [9],

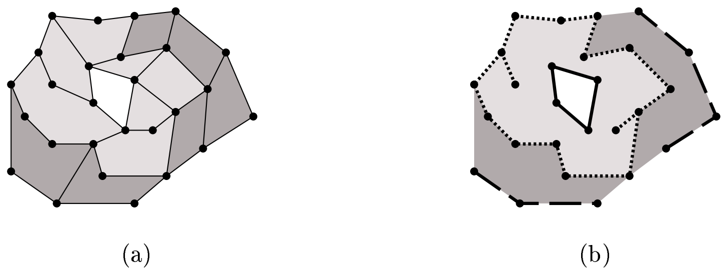


Fig. 5. (a) A graph G with its faces colored according to their levels; level 0 white, level 1 light grey, level 2 dark grey. (b) H_0 (solid), H_1 (dotted), H_2 (dashed).

because the vertex-on-face graph of an embedded planar graph G is planar. This could be important in case planar BFS turns out to be easier than general BFS.

The basic idea in our algorithm is to partition the faces of G into levels around a source face with the source s of the DFS tree on its boundary, and then grow a DFS tree level-by-level; Let the source face be at level 0. We partition the remaining faces of G into levels so that all faces at level 1 share a vertex with the level-0 face, all faces at level 2 share a vertex with some level-1 face but not with the level-0 face, and so on (Fig. 5a). Let G_i be the subgraph of G defined by the union of the boundaries of faces at level at most i, and let $H_i = G_i \setminus G_{i-1}$ (Fig. 5b). We call the vertices of H_i *level-i vertices*. To grow the DFS tree we start by walking clockwise[2] around the level-0 face G_0 until we reach the counterclockwise neighbor of s on G_0. 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 in a way that does not introduce cross-edges, thereby obtaining a DFS tree T_1 for G_1. We repeat this process until we have processed all layers H_i. The key to the efficiency of the algorithm lies in the simple structure of the graphs H_i. Below we give the details of our algorithm and prove the following theorem.

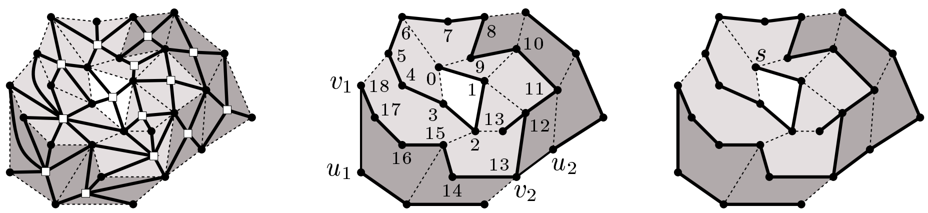
Theorem 3. *Let G be an undirected embedded planar graph, $G^\dagger$ its vertex-on-face graph, and f a face of G^* containing the source vertex s. Given a BFS tree of $G^\dagger$ rooted at f^*, a DFS tree of G rooted at s can be computed in $O(sort(N))$ I/Os.*

Corollary 1. *If there is an algorithm that computes a BFS tree of a planar graph in $\mathcal{I}(N)$ I/Os using $S(N)$ space, then DFS on planar graphs takes $O(\mathcal{I}(N))$ I/Os and $O(S(N))$ space.*

First consider the computation of graphs G_i and H_i. The level of all faces can be obtained from a BFS tree for the vertex-on-face graph $G^\dagger$ rooted at a face containing s (Fig. 6(a)). 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. The *level of a*

---

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



(a) (b) (c)

Fig. 6. (a) $G^\dagger$ shown in bold. (b) T_1, H_2 and attachment edges $\{u_i, v_i\}$.Vertices in T_1 are labeled with their DFS-depths. (c) The DFS tree.

face is the level of the corresponding vertex in the BFS tree divided by two. Given the levels of all faces, the graphs G_i and H_i can be computed in $O(\text{sort}(N))$ I/Os using standard techniques similar to the ones used in computing G^* from G.

Now assume that 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} without introducing cross-edges. If we can do so in $O(\text{sort}(|H_i|))$ I/Os we obtain an $O(\text{sort}(N))$ I/O reduction from planar DFS to planar BFS. Note that the entire graph H_i lies "outside" the boundary of G_{i-1}, i.e., in $\overline{G_{i-1}}$. The boundary of G_{i-1} is in H_{i-1} and consists of cycles, called the *boundary cycles* of G_{i-1}. The graph G_{i-1} is uniform; but $\overline{G_{i-1}}$ may not be uniform. Graph H_i may consist of several connected components. The following lemma shows that H_i has a simple structure, which allows us to compute its DFS tree efficiently.

Lemma 3. *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. 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. Thus any cycle in H_i is a boundary cycle of G_i. Every bicomp that is not a cycle consists of at least two cycles sharing at least two vertices; but the cycles must be boundary cycles, and two boundary cycles of a uniform graph cannot share two vertices. Hence every bicomp is a cycle and thus a boundary cycle. □

Assume for the sake of simplicity that the boundary of G_{i-1} is a simple cycle, so that $\overline{G_{i-1}}$ is uniform. During the construction of the DFS tree for G we maintain the following invariant used to prove the correctness of the algorithm: For every boundary cycle C of G_{i-1}, there is a vertex v on C such that the path traversed by walking clockwise along C is a path in T_{i-1}, and v is an ancestor of all vertices in C (Figure 6b). The *depth* of a vertex in G_{i-1} is its distance from s in T_{i-1}. Let $H'_1, \ldots, H'_k$ be the connected components of H_i. They can be found in $O(\text{sort}(|H_i|))$ I/Os [8]. For every component H'_j, we find the deepest vertex v_j on the boundary of G_{i-1} such that there is an edge $\{u_j, v_j\} \in G$ with

$u_j \in H'_j$. We find these vertices using a procedure similar to the one used in Sec. 2. Below we show how to compute DFS trees T'_j for components H'_j rooted at nodes u_j in $O(\text{sort}(|H'_j|))$ I/Os. Let T_i be the spanning tree of G_i obtained by adding these DFS trees and all edges $\{u_j, v_j\}$ to T_{i-1}. T_i is a DFS tree for G_i: Let $\{v, w\}$ be a non tree edge with $v \in H'_j$. Then either $w \in H'_j$, or w is a boundary vertex of G_{i-1} because $H_i \subseteq G \setminus G_{i-1}$. In the former case, $\{v, w\}$ is a back-edge, as T'_j is a DFS tree for H'_j. In the latter case, $\{v, w\}$ is a back-edge because v is a descendant of u_j, and w is an ancestor of v_j, by the choice of v_j and by our invariant.

All that remains to show is how to compute the DFS tree rooted at u_j for each connected component H'_j of H_i. If we can compute DFS trees for the biconnected components of H'_j, we obtain a DFS tree for H'_j using the bicomp-cutpoint tree as in Sec. 2. By Lemma 3 the non-trivial biconnected components of H_i are cycles. Let C be such a cycle in H'_j, and v be the chosen root for the DFS tree of C. The path obtained after removing the edge between v and its counterclockwise neighbor w along C is a DFS tree for C. We find w using techniques similar to those applied in Sec. 2. In total we compute the DFS tree for H'_j in $O(\text{sort}(|H'_j|))$ I/Os. As this adds simple paths along the boundary cycles of G_i to T_i, the above invariant is preserved.

For the sake of simplicity all the previous arguments were based on the assumption that the boundary of G_{i-1} is a simple cycle. In the general case we compute the boundary cycles $C_1, \dots, C_k$ of G_{i-1} and apply the above algorithm to every C_j. Each cycle C_j is the boundary of a uniform component G'_j of $\overline{G_{i-1}}$. Thus, cycles $C_1, \dots, C_k$ separate subgraphs $H_{i,j} = H_i \cap G'_j$ from each other, and we can deal with each of them separately because the DFS trees computed for the connected components of each of these subgraphs cannot interfere with each other. This concludes the proof of Thm. 3.

4 Conclusions

In this paper we developed the first $o(N)$ I/O and linear space algorithm for DFS in planar graphs. We also designed an $O(\text{sort}(N))$ reduction from planar DFS to planar BFS, proving that in external memory planar DFS is not harder than planar BFS and thus providing further evidence that BFS is among the hardest problems for planar graphs.

Adding the single source shortest path algorithm of [4] as an intermediate reduction step, we can modify our reduction algorithm in order to reduce planar DFS to BFS on either a planar triangulated graph or a planar 3-regular graph. Developing an efficient BFS algorithm for one of these classes of graphs remains an open problem.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

2. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995.
3. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.
4. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.
5. 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. *Computer Networks and ISDN Systems.*
6. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.
7. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 566–575, 2000.
8. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.
9. T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.
10. D. Hutchinson, A. Maheshwari, and N. Zeh. An external-memory data structure for shortest path queries. In *Proc. Annual Combinatorics and Computing Conference, LNCS 1627*, pages 51–60, 1999.
11. J. JáJá and R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.
12. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.
13. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1741*, pages 307–316, 1999.
14. 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.
15. U. Meyer. External memory bfs on undirected graphs with bounded degree. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88, 2001.
16. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.
17. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
18. J. R. Smith. Parallel algorithms for depth-first searches I. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.
19. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.