I/O-Efficient Strong Connectivity and Depth-First Search for Directed Planar Graphs

Lars Arge*
Department of Computer Science
Duke University
Durham, NC 27708
USA
large@cs.duke.edu

Norbert Zeh†
Faculty of Computer Science
Dalhousie University
Halifax, NS B3H 2Y5
Canada
nzeh@cs.dal.ca

Abstract

We present the first I/O-efficient algorithms for the following fundamental problems on directed planar graphs: finding the strongly connected components, finding a simple-path $\frac{2}{3}$-separator, and computing a depth-first spanning (DFS) tree. Our algorithms for the first two problems perform $O(\mathrm{sort}(N))$ I/Os, where $N = V + E$ and $\mathrm{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ is the number of I/Os required to sort N elements. The DFS-algorithm performs $O(\mathrm{sort}(N)\log(N/M))$ I/Os, where M is the number of elements that fit into main memory.

1. Introduction

Recently, external-memory graph algorithms have received considerable attention, because massive graphs arise naturally in a number of applications such as web modeling and geographic information systems (GIS). When working with massive graphs, the I/O-communication, and not the internal memory computation, is often the bottleneck. Efficient external-memory (or I/O-efficient) algorithms can thus lead to considerable runtime improvements.

Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems on general graphs remain open. For planar graphs, on the other hand, significant progress has been made recently. A large number of fundamental problems on *undirected* planar graphs have been solved I/O-efficiently [3, 4, 7, 10, 14]; for *directed* planar graphs, an important first step has been taken with the recent development of an I/O-efficient algorithm for topologically sorting planar directed acyclic graphs (along with algorithms for a few other problems) [5].

In this paper, we develop the first I/O-efficient algorithm for the fundamental problem of computing a depth-first spanning tree of a planar directed graph. Even though the algorithm is non-optimal, it constitutes a major improvement over previous (trivial) algorithms. To obtain this algorithm, we develop I/O-optimal algorithms for computing the strongly connected components of a planar directed graph and for finding a simple-path $\frac{2}{3}$-separator of a strongly connected planar graph. These algorithms may be of independent interest. Our results are a big step towards completely understanding the I/O-complexity of fundamental problems on planar graph.

I/O-Model and previous work. We work in the standard two-level I/O-model with one (logical) disk [2]. This model defines the following parameters:

N	=	number of vertices and edges in the graph ($N = V + E$);
M	=	number of vertices/edges that fit into memory;
B	=	number of vertices/edges that fit into a disk block.

We assume that $2B^2 \leq M < N$.[1] An *Input/Output* operation (or *I/O* for short) transfers one block of consecutive elements 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 $\Theta(\mathrm{scan}(N)) = \Theta(N/B)$. The number of I/Os required to sort N items is $\Theta(\mathrm{sort}(N)) =$

*Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182.

†Supported in part by the National Science Foundation through grant CCR–9984099. This work was done while a postdoctoral fellow at Duke University.

[1] Often, it is assumed only that $2B \leq M$; but sometimes, as in this paper, the very realistic assumption is made that the memory is large enough to hold $\Omega(B^2)$ elements.

$\Theta((N/B)\log_{M/B}(N/B))$ [2]. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$; so the difference in running time between an algorithm performing N I/Os and one performing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os is often substantial.

Despite considerable efforts, I/O-efficient algorithms for many fundamental problems on general graphs have yet to be discovered—see the surveys in [16, 17]. For example, while $\Omega(\min\{V, \text{sort}(V)\})$ (which is $\Omega(\text{sort}(V))$ in practice) is a lower bound for the I/O-complexity of most graph problems [7], the best known algorithms for depth-first search perform $\Omega(V)$ I/Os [6, 7, 12]. As a result, many algorithms have been developed for special classes of graphs. For trees, for example, $O(\text{sort}(N))$-I/O algorithms are known for BFS- and DFS-numbering, Euler tour computation, expression tree evaluation, topological sorting, and several other problems [6, 7]. Most fundamental problems on planar *undirected* graphs also have been solved in $O(\text{sort}(N))$ I/Os [3, 4, 7, 10, 14]. Most of these algorithms exploit the existence of small separators for planar graphs. More precisely, they use that for any planar graph G and any integer $h > 0$, there exists a set of $O\big(N/\sqrt{h}\big)$ vertices whose removal partitions G into $O(N/h)$ subgraphs of size at most h. Such a set of vertices can be computed in $O(\text{sort}(N))$ I/Os [14].[2] For planar *directed* graphs, Arge et al. [5] recently took an important first step with the development of an I/O-efficient algorithm for topologically sorting planar directed acyclic graphs. To obtain this algorithm, they also develop I/O-efficient shortest-path, breadth-first search, and ear decomposition algorithms for planar directed graphs, which in turn use the I/O-efficient planar separator algorithm of [14]. In spite of these developments, the internal-memory DFS-algorithm, which performs $\Omega(V)$ I/Os, remained the best known algorithm for depth-first search in planar directed graphs.

Many external-memory graph algorithms have been obtained using ideas from the corresponding PRAM-algorithms. In some cases, it is even possible to "simulate" a PRAM-algorithm in a standard way and obtain an I/O-efficient algorithm (the so-called *PRAM-simulation* [7]). Relevant to this paper, Kao [11] shows how an efficient algorithm for planar strong connectivity can be used in efficient PRAM-algorithms for computing a simple-path $\frac{2}{3}$-separator for a planar directed graph G (that is, a simple directed path S so that every strongly connected component in $G - S$ has size at most $\frac{2}{3}N$) as well as for computing a DFS-tree of G. However, direct simulation of these algorithms, even given an I/O-efficient strong connectivity algorithm, does not lead to I/O-efficient algorithms.

Our results. In this paper, we develop the first I/O-efficient algorithm for computing a depth-first spanning tree of a planar directed graph. The algorithm performs $O(\text{sort}(N)\log(N/M))$ I/Os, which is a major improvement over the $\Omega(N)$ I/O-bound of previous (trivial) algorithms. To obtain this algorithm, we develop $O(\text{sort}(N))$-I/O algorithms for computing the strongly connected components of a planar directed graph and for finding a simple-path $\frac{2}{3}$-separator of a strongly connected planar graph. These algorithms are optimal and may be of independent interest. Our algorithms utilize a host of I/O-efficient algorithms and techniques previously developed for planar graphs.

Our strong connectivity algorithm for planar directed graphs, presented in Section 2, computes a small vertex separator of the given graph G and derives a compressed version G^c of G from the resulting separator decomposition. Using a modified and I/O-efficient version of the internal-memory strong connectivity algorithm of [8, Chapter 25], it identifies the strongly connected components of G^c and then uses this information to identify the strongly connected components of G.

Our algorithm for computing a simple-path $\frac{2}{3}$-separator of a strongly connected planar graph is presented in Section 3. It uses ideas from Kao's PRAM-algorithm for the same problem [11]. However, the central computation of the algorithm is carried out in a novel and non-trivial manner that exploits the sequential nature of the I/O-model.

Finally, in Section 4, we present our DFS-algorithm for directed planar graphs. The algorithm performs $O(\text{sort}(N)\log(N/M))$ I/Os. It is based on a recursive partition of the graph into smaller subgraphs. At the bottom of this recursion are the strongly connected components of the graph. We perform DFS in these components using ideas from the PRAM-algorithm of [11].

2. Strong Connectivity

Our algorithm for computing the strongly connected components of a planar directed graph $G = (V, E)$ relies on our ability to compute small separators of planar graphs I/O-efficiently. More precisely, we use the $O(\text{sort}(N))$-I/O algorithm by Maheshwari and Zeh [14] to compute a set $S \subseteq V$ of $O(N/B)$ vertices, called a *separator*, whose removal partitions G into $O\big(N/B^2\big)$ subgraphs $G_1, \ldots, G_k$ with the following properties: (1) Every graph G_i has size at most B^2. (2) For every graph G_i, the set ∂G_i of separator vertices adjacent to vertices in G_i has size at most B. We call this set the *boundary* of G_i. (3) The partition has $O\big(N/B^2\big)$ *boundary sets*, defined as the maximal subsets of the separator S so that the vertices in each subset are adjacent to the same set of subgraphs G_i [9]. To guarantee that a partition with the last property exists, we have to assume that G has bounded degree; more precisely, we assume that G has degree at most three. This is not a restriction because it is easy to reduce the computation of the strongly connected components of any

[2] The algorithm as described in [14] requires that $M \geq B^2 \log^2 B$, but a simple improvement reduces this requirement to $M \geq 2B^2$.

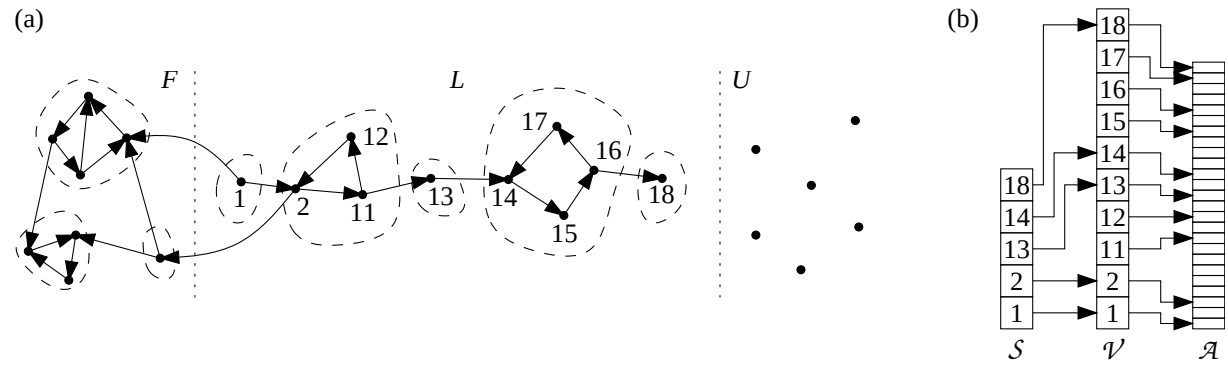

Figure 1. (a) The partition of the strongly connected components of G' into three sets F (finished), L (live), and U (untouched). (b) The three stacks representing the strongly connected components in L and their unexplored out-edges.

planar graph of size N to the computation of the strongly connected components of another planar graph of degree at most three and size $O(N)$.

From the above partition of G, we construct a graph G^c that encodes the reachability between the vertices in S succinctly. Graph G^c has vertex set S; the edge set of G^c is defined so that two separator vertices are in the same strongly connected component of G if and only if this is true in G^c. To achieve this, we add an edge (v,w) to graph G^c if and only if this edge exists in G or v and w are on the boundary ∂G_i of the same subgraph G_i and there is a directed path from v to w in the subgraph R_i of G induced by the vertices in $V(G_i) \cup \partial G_i$. Graph G^c has $O(N/B)$ vertices and $O\big(N/B^2 \cdot B^2\big) = O(N)$ edges. Since $M \geq 2B^2 \geq |R_i|$, for all $1 \leq i \leq k$, the construction of G^c can be carried out by loading graphs $R_1, \ldots, R_k$ into memory, one at a time, and adding the appropriate edges to G^c. This takes $O(N/B)$ I/Os.

Below we argue that the strongly connected components of G^c can be computed in $O(\mathrm{sort}(N))$ I/Os. In the full paper, we show that the strongly connected components of G can be derived from the strongly connected components of graphs G^c and $R_1, \ldots, R_k$. We also show that this can be done by loading each graph R_i into memory for a second time and finishing the computation in main memory. Hence, this takes another $O(N/B)$ I/Os. By putting the three steps together, we obtain the following result.

Theorem 1 *The strongly connected components of a planar directed graph G with N vertices can be computed in* $O(\mathrm{sort}(N))$ *I/Os.*

In the remainder of this section, we argue that a modified version of the internal-memory algorithm of [8, Chapter 25] can be used to compute the strongly connected components of the compressed graph G^c in $O(\mathrm{sort}(N))$ I/Os. We explore the edges of G^c in a depth-first manner and maintain the strongly connected components of the graph $G' = (S, E')$, where E' is the set of edges that have been explored so far. The current set of strongly connected components is partitioned into three sets F, L, and U (see Figure 1a). The components in F are *finished*; that is, they are in fact strongly connected components of G. The components in L are *live*, meaning that they are potentially part of larger strongly connected components of G. We maintain the invariant that the components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ in L form a "path of strongly connected components"; that is, for $1 < i \leq q$, graph G' contains an edge (v,w) with $v \in \mathcal{C}_{i-1}$ and $w \in \mathcal{C}_i$. Finally, the set U contains all isolated vertices of G'. These vertices are *untouched* in the sense that no edges incident to them have been explored up to this point.

Given the current sets F, L, and U, we choose an out-edge (v,w) of the last component $\mathcal{C}_q$ in L (that is, $v \in \mathcal{C}_q$) as the next edge to be explored. Depending on the "location" of vertex w, we update the current set of components of G': If $w \in U$, we add a new strongly connected component $\mathcal{C}_{q+1} = (\{w\}, \emptyset)$ to L and remove w from U. If $w \in \mathcal{C}_q$ or $w \in \mathcal{C}'$, for some component $\mathcal{C}' \in F$, we take no further action, since the addition of edge (v,w) to G' does not create or merge any strongly connected components. If $w \in \mathcal{C}_i$, for $i < q$, we merge components $\mathcal{C}_i, \ldots, \mathcal{C}_q$ into a larger strongly connected component $\mathcal{C}'_i$. Finally, if component $\mathcal{C}_q$ does not have any unexplored out-edges, we move it to F, as it cannot be merged with any other strongly connected components (because all unexplored edges with endpoints in $\mathcal{C}_q$ are in-edges of $\mathcal{C}_q$). The correctness of the algorithm is easily established [8, Chapter 25].

In order to carry out the above algorithm I/O-efficiently, we preprocess G^c as follows: First we arrange the vertex set S of G^c so that the vertices in each boundary set are stored consecutively. We also sort the adjacency list of every vertex so that edges whose targets are in the same boundary set are stored consecutively. This takes $O(\mathrm{sort}(N))$ I/Os. Dur-

ing the algorithm, we label every vertex as belonging to F, L, or U; initially, every vertex belongs to U. When moving a vertex v from U to L, we time-stamp it. When moving v from L to F as part of a finished component, we assign a label to v that identifies this component. We represent the live components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ using three stacks $\mathcal{V}$, $\mathcal{A}$, and $\mathcal{S}$ (see Figure 1b). Stack $\mathcal{V}$ stores all live vertices in their order of discovery, that is, by increasing time stamps. Stack $\mathcal{A}$ stores their adjacency lists in the same order. Due to the preprocessing, the out-edges of a vertex $v \in \mathcal{V}$ that are stored on $\mathcal{A}$ are ordered by the boundary sets containing their target vertices. This defines a natural partition of $\mathcal{A}$ into sequences of edges whose targets are in the same boundary set. We call these sequences $\mathcal{A}_1, \ldots, \mathcal{A}_s$ *stack segments* and maintain the invariant that all edges (v,w) in the last stack segment $\mathcal{A}_s$ store the correct labels of their target vertices $w \in S$; that is, we do not need to access w to determine whether it is in F, L, or U. We call this the *stack segment invariant*. Finally, stack $\mathcal{S}$ stores one entry per component. The entry for component $\mathcal{C}_i$ "points" to the first vertex in $\mathcal{C}_i$ on stack $\mathcal{V}$, by storing its time stamp.

Using the above representation of graph G^c and its live components, we perform the computation of the algorithm as follows: Let (v,w) be the topmost edge on stack $\mathcal{A}$. If v is not in the last component $\mathcal{C}_q$ of L—that is, its time stamp t_v is smaller than the time stamp t_q of the topmost entry on stack $\mathcal{S}$—component $\mathcal{C}_q$ is finished and has to be moved to F. To do this, we perform the following computation until the time stamp t_u of the topmost vertex u on stack $\mathcal{V}$ is less than t_q: We remove u from stack $\mathcal{V}$, change its label to t_q, and mark it as finished. When $t_u < t_q$, we are done labeling the vertices of $\mathcal{C}_q$; so we remove t_q from stack $\mathcal{S}$. We repeat this procedure, moving finished components to F, until the topmost entry on stack $\mathcal{S}$ has a time stamp less than or equal to t_v. Now that v is in the last component $\mathcal{C}_q$ of L, we proceed as follows: If vertex w is finished, we discard edge (v,w). If w is in U, we mark it as live, time-stamp it, and push it onto $\mathcal{V}$. We also push the edges in the adjacency list of w onto stack $\mathcal{A}$ and a new entry with w's time stamp onto stack $\mathcal{S}$. If w is live and has time stamp t_w, it is contained in some live component $\mathcal{C}_i$. We achieve the required merging of components $\mathcal{C}_i, \ldots, \mathcal{C}_q$ by removing all entries from the top of stack $\mathcal{S}$ whose time stamps are greater than t_w.

In addition to maintaining the connected components of $G' = (V, E')$ after adding edge (v,w) to E', we have to guarantee that the stack segment invariant remains valid. This invariant may be violated in three different ways by the above procedure for processing the current edge (v,w): (1) Target vertices of edges in the topmost segment of stack $\mathcal{A}$ may become finished as the result of moving components from L to F. (2) When moving a vertex x from U to L, its adjacency list is pushed onto $\mathcal{A}$; so a new segment becomes the topmost stack segment. (3) Edge (v,w) was the last edge in the current stack segment. In either of these cases, we re-establish the invariant by loading the topmost segment of stack $\mathcal{A}$ and the corresponding boundary set of S into memory, updating the labels of the edges in the topmost stack segment, and writing this segment back to $\mathcal{A}$. This takes $O(1)$ I/Os, because every segment and every boundary set has size at most B and the vertices of every boundary set are stored consecutively on disk.

To analyze the I/O-complexity of our algorithm, we first count the number of I/Os we spend on maintaining the stack segment invariant. As we have just argued, we spend $O(1)$ I/Os to re-establish the invariant whenever a vertex becomes live or finished and when the last entry of a stack segment is removed from $\mathcal{A}$. Both events occur only $O(N/B)$ times. For the former, this is obvious, because each of the $O(N/B)$ vertices in S becomes live and is finished only once. To see that the latter happens only $O(N/B)$ times, observe that every vertex in S has degree at most $3B$ in G^c (because G has degree at most three) and, hence, every boundary set of the partition can give rise to only $O(B)$ stack segments. Since there are $O(N/B^2)$ boundary sets, this implies that there are only $O(N/B)$ stack segments. This shows that we spend $O(N/B)$ I/Os in total on maintaining the stack segment invariant. Besides that, we spend $O(N/B)$ I/Os on stack operations, because each of the $O(N/B)$ vertices and $O(N)$ edges is pushed onto and popped from a stack exactly once, and one stack operation takes $O(1/B)$ I/Os amortized. We touch every vertex in S twice, once when it becomes live and again when it becomes finished. This takes $O(1)$ I/Os per vertex, $O(N/B)$ I/Os in total. Finally, we spend $O(N/B + \text{scan}(N)) = O(N/B)$ I/Os on copying the adjacency lists of all vertices to stack $\mathcal{A}$. Including the preprocessing, our algorithm hence takes $O(\text{sort}(N))$ I/Os to compute the strongly connected components of G^c.

3. Finding a Directed Path Separator

Kao [11] shows that every strongly connected planar graph has a simple-path $\frac{2}{3}$-separator, that is, a simple directed path S starting at a specified vertex s so that no strongly connected component of $G - S$ has size more than $\frac{2}{3}N$. Our algorithm for computing such a separator uses ideas of Kao's parallel algorithm for this problem [11]; but, in order to obtain an $O(\text{sort}(N))$-I/O algorithm, we exploit the sequential nature of the I/O-model and present a novel and non-trivial way to carry out the central computation of Kao's algorithm.

The high-level description of the algorithm is as follows: First we use the shortest path algorithm of [5] to compute, in $O(\text{sort}(N))$ I/Os, a directed spanning tree T of G rooted at s. Every edge e of G that is not an edge of T defines a *fundamental cycle* consisting of edge e and the (undirected)

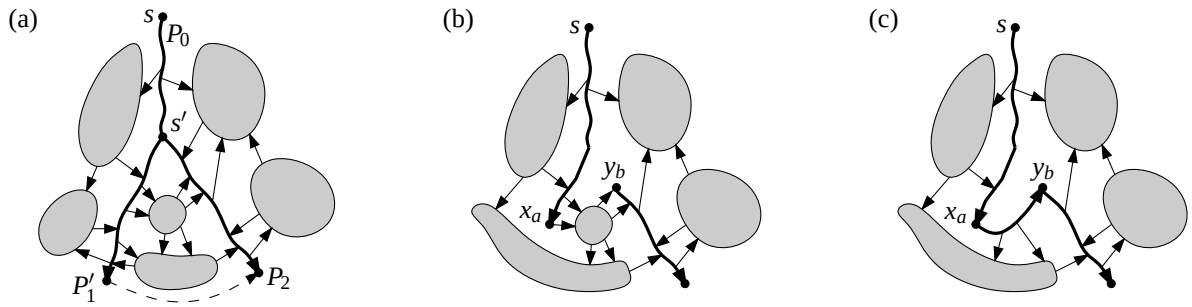

Figure 2. The three steps of finding a simple-path separator: (a) Find a $\frac{2}{3}$-separating fundamental cycle in a directed spanning tree of G. (None of the strongly connected components has size more than $\frac{2}{3}$N.) (b) Find minimal subpaths of the two paths computed in Step (a) so that these two paths still form a $\frac{2}{3}$-separator. (c) Join these two paths using a simple path from the sink of the first path to the source of the second path.

path in T connecting the two endpoints of e. If G is triangulated, at least one of these cycles has the property that at most $\frac{2}{3}N$ vertices are either inside or outside the cycle [13]. We triangulate G, using an algorithm of [10], and use an I/O-efficient version of Lipton and Tarjan's algorithm [13], presented in [10], to find such a cycle in the resulting graph (see Figure 2a); this takes $O(\text{sort}(N))$ I/Os. The computed cycle consists of edge e and two paths P_1' and P_2 in T from a vertex s' to the two endpoints of e. Let P_0 be the path in T from s to s'. Note that, even though the fundamental cycle induced by edge e may no exist in G (because edge e may not be in G), paths P_0, P_1', and P_2 exist in G (because $T \subseteq G$). We use the strong connectivity algorithm from Section 2 to test, in $O(\text{sort}(N))$ I/Os, whether $P_0 \cup P_1'$ or $P_0 \cup P_2$ is a $\frac{2}{3}$-separator of G, that is, whether all strongly connected components of $G-(P_0 \cup P_1')$ or $G-(P_0 \cup P_2)$ have weight at most $\frac{2}{3}N$. If one of these paths is a $\frac{2}{3}$-separator, we report it as the desired separator S; otherwise, let $P_1 = P_0 \cup P_1'$.

Let $P_1 = (s = x_0, \ldots, x_q)$ and $P_2 = (y_0, \ldots, y_r)$. Since $P_0 \circ P_2$ is not a $\frac{2}{3}$-separator, there exists a vertex $x_a \in P_1'$ so that $(x_0, \ldots, x_a) \cup P_2$ is a $\frac{2}{3}$-separator, but $(x_0, \ldots, x_{a-1}) \cup P_2$ is not. Similarly, there exists a vertex $y_b \in P_2$ so that $(x_0, \ldots, x_a) \cup (y_b, \ldots, y_r)$ is a $\frac{2}{3}$-separator, but $(x_0, \ldots, x_a) \cup (y_{b+1}, \ldots, y_r)$ is not (see Figure 2b). Below we argue that these two vertices can be found in $O(\text{sort}(N))$ I/Os. Kao [11] shows that there exists a simple path P' from x_a to y_b in $G-((x_0, \ldots, x_{a-1}) \cup (y_{b+1}, \ldots, y_r))$. We compute such a path P' and report the path $S = (x_0, \ldots, x_a) \cup P' \cup (y_b, \ldots, y_r)$ as the desired simple-path $\frac{2}{3}$-separator (see Figure 2c); the computation of P' can be carried out in $O(\text{sort}(N))$ I/Os by using the shortest-path algorithm of [5] to compute a directed spanning tree T' of $G-((x_0, \ldots, x_{a-1}) \cup (y_{b+1}, \ldots, y_r))$ rooted at x_a and then applying standard tree computations to extract the path from x_a to y_b in T'. Since all parts of the algorithm can be carried out in $O(\text{sort}(N))$ I/Os, we obtain the following theorem.

Theorem 2 *For any strongly connected planar graph G with N vertices and any vertex $s \in G$, a simple path $\frac{2}{3}$-separator of G rooted at s can be found in $O(\text{sort}(N))$ I/Os.*

We have to show how to compute vertices x_a and y_b. We find both vertices in a similar manner; so we describe only the computation of vertex x_a: We start by computing the strongly connected components of $G-(P_1 \cup P_2)$ and contracting each such component $\mathcal{C}$ into a single vertex v of weight $\omega(v) = |\mathcal{C}|$. All vertices in $P_1 \cup P_2$ have weight one. Let G' be the resulting graph. The weight of a subgraph of G' is the total weight of its vertices. Vertex x_a is now the vertex in P_1 so that no strongly connected component of $G'-((x_0, \ldots, x_a)) \cup P_2)$ has *weight* exceeding $\frac{2}{3}N$, but graph $G'-((x_0, \ldots, x_{a-1}) \cup P_2)$ contains such a component. Let $G'' = G' - P_2$.

The goal of our algorithm is to walk along path P_1, from x_q to x_0, and find the first vertex x_a so that one strongly connected component of $G''-(x_0, \ldots, x_{a-1})$ has weight more than $\frac{2}{3}N$. Since x_a is the first such vertex, this component does in fact contain x_a. Hence, it suffices to compute, for every visited vertex x_i, the weight of the strongly connected component $\mathcal{C}_i$ of $G''-(x_0, \ldots, x_{i-1})$ that contains x_i.

To perform this computation efficiently, we first characterize the set of vertices in $\mathcal{C}_i$. First observe that every descendant of x_i in P_1 that can reach x_i in $G''-(x_0, \ldots, x_{i-1})$ is in $\mathcal{C}_i$. Let P-LOW(x_i) be the lowest such descendant of x_i.[3] Then component $\mathcal{C}_i$ contains vertices $x_i, \ldots,$ P-LOW(x_i) but no ancestor of x_i or descendant of P-LOW(x_i) in P_1. To

[3] We define the lowest or highest vertex with a certain property to be the vertex with this property that is furthest away from s or closest to s, respectively (see Figure 3); but we compare vertices and compute minima and maxima of vertices x_i w.r.t. their indices i.

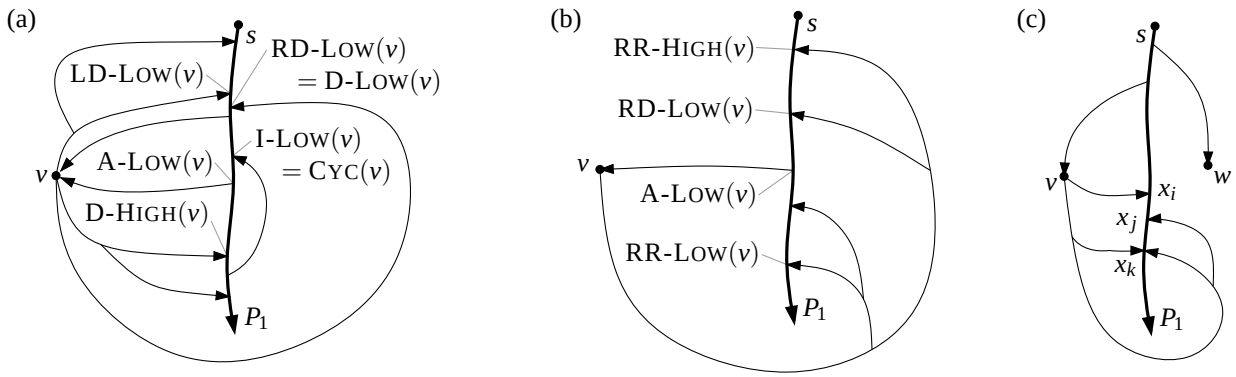

Figure 3. Figures (a) and (b) illustrate the definitions of important vertices on path P_1. Figure (c) illustrates the concept of left- and right-attachment and left- and right-reachability: Vertex v is left-attached; vertex w is right-attached. Vertex x_i is left-reachable from v; vertex x_j is right-reachable from v; and vertex x_k is both.

identify the set of vertices in $G'' - P_1$ that are in $\mathcal{C}_i$, we define a vertex $\text{CYC}(v)$, for every $v \in G'' - P_1$: $\text{CYC}(v)$ is the lowest vertex $x_h \in P_1$ so that there exists a directed cycle in $G'' - (x_0, \ldots, x_{h-1})$ that contains both v and x_h. In the full paper, we show that $v \in \mathcal{C}_i$ if and only if $\text{CYC}(v) \in \mathcal{C}_i$. Thus, we can characterize component $\mathcal{C}_i$ using the following lemma.

Lemma 1 *A vertex $v \in G''$ is in $\mathcal{C}_i$ if and only if (i) $v \in P_1$ and $x_i \leq v \leq \text{P-LOW}(x_i)$ or (ii) $v \in G'' - P_1$ and $x_i \leq \text{CYC}(v) \leq \text{P-LOW}(x_i)$.*

By Lemma 1, we can find vertex x_a using the following simple algorithm, once vertices $\text{CYC}(v)$, $v \in G'' - P_1$, and $\text{P-LOW}(x_i)$, $x_i \in P_1$, are given: We modify the weight of every vertex $x_i \in P_1$ so that it represents the total weight of x_i and all vertices $v \in G'' - P_1$ so that $\text{CYC}(v) = x_i$; that is, we compute $\omega(x_i) = 1 + \sum_{x_i = \text{CYC}(v)} \omega(v)$. This can be done by sorting and scanning the vertex set of G''. Next we scan vertices $x_q, \ldots, x_0$ and maintain an initially empty stack $\mathcal{S}$ that stores the weights of the strongly connected components of the current graph $G'' - (x_0, \ldots, x_{i-1})$ that include vertices in P_1. For every vertex x_i, we perform the following computation: While the vertex x_j on the top of stack $\mathcal{S}$ is an ancestor of $\text{P-LOW}(x_i)$, we remove x_j from $\mathcal{S}$ and add $\omega(x_j)$ to $\omega(x_i)$. When x_j is a proper descendant of $\text{P-LOW}(x_i)$, or $\mathcal{S}$ is empty, we check whether $\omega(x_i) > \frac{2}{3}N$. If so, we report $x_a = x_i$; otherwise, we push x_i onto $\mathcal{S}$ and proceed to x_{i-1}. This requires $O(q) = O(N)$ stack operations and hence takes $O(N/B)$ I/Os.

We have to show how to compute vertices $\text{CYC}(v)$, $v \in G'' - P_1$, and $\text{P-LOW}(x_i)$, $x_i \in P_1$, in $O(\text{sort}(N))$ I/Os. First, let us characterize these vertices (see Figure 3a). Since vertex $\text{CYC}(v) = x_i$ can reach v in $G'' - (x_0, \ldots, x_{i-1})$, $\text{CYC}(v)$ has to be an ancestor of the lowest vertex $\text{A-LOW}(v) \in P_1$ so that there exists a path from $\text{A-LOW}(v)$ to v in G'' that has no internal vertex in P_1. We call such a path a *direct* path from $\text{A-LOW}(v)$ to x_i and say that $\text{A-LOW}(v)$ can reach v *directly*. Since every ancestor x_i of $\text{A-LOW}(v)$ can reach v in $G'' - (x_0, \ldots, x_{i-1})$, $\text{CYC}(v)$ can be characterized as the lowest ancestor x_i of $\text{A-LOW}(v)$ so that there exists a path from v to x_i in $G'' - (x_0, \ldots, x_{i-1})$; this path may or may not be direct. Let $\text{D-LOW}(v)$ be the lowest ancestor of $\text{A-LOW}(v)$ that can be reached directly from v. If $\text{CYC}(v)$ is a descendant of $\text{D-LOW}(v)$, there has to be a path P from v to $\text{CYC}(v)$ whose internal vertices that are in P_1 are descendants of $\text{A-LOW}(v)$. Let $\text{D-HIGH}(v)$ be the highest descendant of $\text{A-LOW}(v)$ that can be reached directly from v, and let $\text{I-LOW}(v)$ be the lowest ancestor of $\text{A-LOW}(v)$ that can be reached from $\text{D-HIGH}(v)$ through a path all of whose internal vertices in P_1 are descendants of $\text{I-LOW}(v)$; that is, $\text{I-LOW}(v)$ is the lowest ancestor x_i of $\text{A-LOW}(v)$ so that $\text{P-LOW}(x_i)$ is a descendant of $\text{D-HIGH}(v)$. Then $\text{CYC}(v) = \max(\text{D-LOW}(v), \text{I-LOW}(v))$.

After extending the above definition of $\text{A-LOW}(v)$ to vertices $v \in P_1$, the following lemma, which we prove in the full paper, provides the characterization of vertex $\text{P-LOW}(x_i)$, for every $x_i \in P_1$.

Lemma 2 *For every vertex $x_i \in P_1$, let x_j be the vertex in $(x_{i+1}, \ldots, \text{A-LOW}(x_i))$ so that $\text{A-LOW}(x_j)$ is maximized. Then $\text{P-LOW}(x_i) = \max(\text{A-LOW}(x_i), \text{P-LOW}(x_j))$.*

In the rest of this section, we argue that vertices $\text{P-LOW}(x_i)$, $x_i \in P_1$, and $\text{A-LOW}(v)$, $\text{D-LOW}(v)$, $\text{I-LOW}(v)$, and $\text{D-HIGH}(v)$, for all $v \in G'' - P_1$, can be computed I/O-efficiently. Vertices $\text{CYC}(v)$, for all $v \in G'' - P_1$, can then be found in a single scan of the vertex set of G''.

Finding A-LOW($\mathbf{v}$). Note that graph $G'' - P_1$ is a planar DAG. We extend this DAG to a DAG G_1 by adding the vertices in P_1 and their out-edges to $G'' - P_1$. For every vertex $v \in G'' - P_1$, A-LOW(v) is the maximal vertex x_i that can reach v in G_1. This vertex can be found, for all $v \in G'' - P_1$, by processing G_1 from the sources towards the sinks. The sources forward their own identities to their out-neighbors; every non-source vertex v chooses A-LOW(v) as the maximal vertex received from its in-neighbors and forwards A-LOW(v) to its out-neighbors. Using the time-forward processing technique of [7], this computation can be carried out in $O(\text{sort}(N))$ I/Os. (Details appear in the full paper.) The application of this technique requires G_1 to be topologically sorted; since G_1 is planar, this can be done in $O(\text{sort}(N))$ I/Os [5].

For every $x_i \in P_1$, we compute vertex A-LOW(x_i) as the maximum of vertices A-LOW(v), where v iterates over the in-neighbors of x_i in $G'' - P_1$. This computation can be carried out, for all vertices in P_1, by sorting and scanning the vertex and edge sets of G'' and hence takes $O(\text{sort}(N))$ I/Os.

Finding P-LOW($\mathbf{x_i}$). To compute P-LOW(x_i), for every $x_i \in P_1$, we find the vertex x_{j_i} in $(x_{i+1}, \ldots, \text{A-LOW}(x_i))$ so that A-LOW(x_{j_i}) is maximized. This reduces to a range-maxima query over the point set $(x_j, \text{A-LOW}(x_j))$, $x_j \in P_1$, with query interval $I_{x_i} = (x_i, \text{A-LOW}(x_i)]$; that is, we find the point with largest y-coordinate whose x-coordinate is in the interval I_{x_i}. As shown in [7], these queries can be answered in $O(\text{sort}(N))$ I/Os, for all $v \in P_1$. Now we define a rooted forest F by making every vertex x_i the child of vertex x_{j_i}. We process F from the root towards the leaves, using standard tree computations, and compute, for every vertex x_i, P-LOW(x_i) = $\max(\text{A-LOW}(x_i), \text{P-LOW}(x_{j_i}))$. This takes another $O(\text{sort}(N))$ I/Os.

Finding D-LOW($\mathbf{v}$) and D-HIGH($\mathbf{v}$). The computations of D-LOW(v) and D-HIGH(v) are similar; so we describe only the computation of D-LOW(v). For every vertex $v \in G'' - P_1$ so that A-LOW(v) exists, we fix a direct path P_v from A-LOW(v) to v. We call v *left-* or *right-attached* (to P_1) if path P_v leaves P_1 to the left or right, respectively (see Figure 3c). We call a vertex $x_i \in P_1$ that can be reached directly from v *left-* or *right-reachable* from v if there exists a direct path from v to x_i that enters P_1 from the left or right, respectively.

We compute, for every vertex $v \in G'' - P_1$, the lowest ancestors LD-LOW(v) and RD-LOW(v) of A-LOW(v) that are left or right-reachable from v, respectively (see Figure 3a). Clearly, D-LOW(v) is the lower of LD-LOW(v) and RD-LOW(v). We deal with left- and right-attached vertices separately. Since the computations are similar, we describe only how to compute LD-LOW(v) and RD-LOW(v) for every left-attached vertex v.

Let V_L be the set of left-attached vertices. If LD-LOW(v) exists, for a vertex $v \in V_L$, it can be reached by walking along P_v from A-LOW(v) to v and then choosing the "most clockwise" path back to P_1. More precisely, at every vertex y on the path from v back to P_1, let x be the predecessor of y on the path; then the next vertex z on the path is the vertex so that edge (y,z) is the first edge after (x,y) in counterclockwise order around y so that z can reach a vertex in P_1.

Based on this observation, we can find LD-LOW(v), for all $v \in V_L$, as follows: We construct a DAG G_2 that is obtained by changing the directions of all edges in G'' and then removing all edges whose targets are in P_1 or that leave P_1 to the right. For every source $v \in P_1$ of DAG G_2, we define LD-LOW(v) = v. For all sources in $G'' - P_1$, LD-LOW(v) is not defined. Now we process DAG G_2 from the sources to the sinks and perform the following computation, for every vertex: Every source v forwards its own value LD-LOW(v) to all its out-neighbors. For every non-source vertex v, let u be the out-neighbor of v in G_2 that precedes v in P_v. Then LD-LOW(v) is chosen as the label received along the first edge (x,v) after edge (v,u) in counterclockwise order around v so that there is a label being sent along edge (x,v). (Some edges do not carry any label.) Similarly, for every out-edge (v,w) of v, the label sent to w is chosen as the label received along the first edge (x,v) after edge (v,w) in counterclockwise order around v so that there is a label being sent along edge (x,v). To carry out this computation in $O(\text{sort}(N))$ I/Os, we topologically sort G_2 and apply the time-forward processing technique of [7]. (Details appear in the full paper.)

To find RD-LOW(v), for all $v \in V_L$, we make the following observation: Let V_R be the set of vertices that are right-reachable from vertices in V_L, and let RR-LOW(v) and RR-HIGH(v) be the lowest and highest vertices that are right-reachable from a vertex $v \in V_L$ (see Figure 3b). Then every vertex in V_R that is a descendant of RR-HIGH(v) and an ancestor of RR-LOW(v) is right-reachable from v. In the full paper, we formally prove this fact; intuitively, this follows from the planarity of G''. The following lemma is an immediate consequence of this observation.

Lemma 3 RD-LOW(v) = $\max\{w \in V_R : \text{RR-HIGH}(v) \leq w \leq \min(\text{RR-LOW}(v), \text{A-LOW}(v))\}$.

Using this lemma, finding vertices RD-LOW(v) translates into a set of range-maxima queries with query intervals $I_v = [\text{RR-HIGH}(v), \min(\text{RR-LOW}(v), \text{A-LOW}(v))]$, for all $v \in G'' - P_1$, over the point set $\{(x_i, x_i) : x_i \in V_R\}$. As argued before, these queries can be answered in $O(\text{sort}(N))$ I/Os [7]. The computation of vertices RR-HIGH(v) and RR-LOW(v), for all $v \in V_L$, is similar to the computation of vertices A-LOW(v), for all $v \in G'' - P_1$.

Finding I-LOW(v). To compute I-LOW(v), for all $v \in G'' - P_1$, we represent every vertex $v \in G'' - P_1$ as an interval $I_v = [\text{A-LOW}(v), \text{D-HIGH}(v)]$ and every vertex $x_i \in P_1$ as an interval $J_{x_i} = [x_i, \text{P-LOW}(x_i)]$. From the definition of I-LOW(v), we obtain that I-LOW$(v) = \max\{x_i \in P_1 : I_v \subseteq J_{x_i}\}$. We sort the intervals by their left endpoints, in decreasing order, and scan the list of intervals to simulate a sweep from $+\infty$ to $-\infty$. When the sweep passes the left endpoint of an interval $I_v = [x_i, x_j]$, we insert v into a priority queue Q and give it priority j. When the sweep passes the left endpoint of an interval $J_{x_i} = [x_i, x_j]$, we use a sequence of DELETEMIN operations to remove all vertices of priority at most j from Q and report I-LOW$(v) = x_i$, for each such vertex. For every vertex v that remains in Q at the end of the sweep, I-LOW(v) does not exist. This computation requires sorting and scanning the set of intervals and involves $O(N)$ priority queue operations. Using the priority queue of [1], this takes $O(\text{sort}(N))$ I/Os.

4. Depth-First Search

In this section, we present an algorithm that constructs a DFS-tree of a planar graph G in $O(\text{sort}(N)\log(N/M))$ I/Os. More precisely, the algorithm constructs a spanning tree T of G that can be obtained by performing a depth-first traversal of G and adding an edge (v,w) to T whenever the algorithm follows edge (v,w). For an undirected graph G, a DFS-tree T of G is any spanning tree so that graph G does not contain any cross edges w.r.t. T; that is, there is no edge $(v,w) \in G$ so that neither v nor w is an ancestor of the other in T. Any depth-first traversal of T is a depth-first traversal of G. For directed graphs, a DFS-tree may contain cross edges, and not every depth-first traversal of a DFS-tree is a depth-first traversal of G. In particular, if ν is the postorder numbering of T defined by the traversal, then the traversal is a depth-first traversal of G if and only if $\nu(v) > \nu(w)$, for every cross edge (v,w). Hence, we require that a directed DFS-algorithm constructs T along with a postorder numbering ν that has the above property. We call (T,ν) a *DFS-pair* of G. We call a pair (T',ν'), where T' is a subtree of G, a *partial DFS-pair* of G if it is a DFS-pair for the subgraph G' of G induced by the vertices in T'. The *anchor* $\sigma(v)$ of a vertex $v \in G - T'$ is the vertex $u \in T'$ with minimal number $\nu'(u)$ so that there is a path from u to v in G that has no internal vertex in T'.

4.1. Arbitrary Directed Planar Graphs

The general idea of our algorithm for computing a DFS-pair of a planar directed graph G is to number and sort the strongly connected components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ of G so that no vertex in $\mathcal{C}_i$ can reach a vertex in $\mathcal{C}_j$, for $j < i$. (In a sense, we "topologically sort" the strongly connected components of G.) We assume that the source s of the DFS is contained in component $\mathcal{C}_1$, because otherwise no vertex in $\mathcal{C}_1$ is reachable from s. Now we process components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ in topologically sorted order, compute a DFS-tree of $\mathcal{C}_1$, and augment it with appropriate DFS-trees of subsequent components.

To obtain an I/O-efficient DFS-algorithm using this idea, we cannot literally process components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ one at a time in their order of appearance; rather, we use a recursive algorithm that partitions the components into two sets $\mathcal{C}_1, \ldots, \mathcal{C}_p$ and $\mathcal{C}_{p+1}, \ldots, \mathcal{C}_q$ so that the components in both sets contain approximately the same number of vertices. We recursively compute a DFS-tree for the subgraph of G induced by the vertices in components $\mathcal{C}_1, \ldots, \mathcal{C}_p$ and then augment it with DFS-trees of appropriate subgraphs of the graph induced by the vertices in components $\mathcal{C}_{p+1}, \ldots, \mathcal{C}_q$.

The details of our algorithm are as follows: If $|G| \leq M$, we load G into internal memory and compute a DFS-pair (T,ν) of G using the internal-memory DFS-algorithm. If $|G| > M$, but G is strongly connected, we show in Section 4.2 that a DFS-pair of G can be computed in $O(\text{sort}(N)\log(N/M))$ I/Os. If neither of these two cases applies, we spend $O(\text{sort}(N))$ I/Os to compute the strongly connected components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ of G, using the strong connectivity algorithm from Section 2. We construct a planar DAG G' by contracting every component $\mathcal{C}_i$ into a single vertex v_i and spend $O(\text{sort}(N))$ I/Os to topologically sort G', using the algorithm of [5]. Now we arrange the strongly connected components of G in the same order as their corresponding vertices in the topological order of G'. Assume w.l.o.g. that $\mathcal{C}_1, \ldots, \mathcal{C}_q$ is the order of the strongly connected components of G at the end of this computation. We scan components $\mathcal{C}_1, \ldots, \mathcal{C}_q$ and find the index p so that the difference between the sizes of graphs $\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_p$ and $\mathcal{C}_{p+1} \cup \cdots \cup \mathcal{C}_q$ is minimized. We denote the vertex set of components $\mathcal{C}_1, \ldots, \mathcal{C}_p$ by V_1 and the vertex set of components $\mathcal{C}_{p+1}, \ldots, \mathcal{C}_q$ by V_2. Let G_1 and G_2 be the subgraphs of G induced by the vertices in V_1 and V_2, and let $\tilde{E}$ be the set of edges connecting vertices in G_1 with vertices in G_2. We also split graph G' into the two subgraphs G'_1 and G'_2 induced by vertices $v_1, \ldots, v_p$ and $v_{p+1}, \ldots, v_q$, respectively. In total, the partition of G into graphs G_1 and G_2 and edge set $\tilde{E}$, as well as the partition of G' into subgraphs G'_1 and G'_2, can be obtained in $O(\text{sort}(N))$ I/Os.

Now we use graphs G_1 and G'_1 to recursively compute a DFS-pair (T_1,ν_1) of G_1. In order to augment (T_1,ν_1) to a DFS-pair of G, we partition graph G_2 into so-called *dangling subgraphs* G_v, defined below, for the target vertices v of the edges in $\tilde{E}$. For every non-empty graph G_v, we recursively compute a DFS-pair (T_v,ν_v) and join T_v to T_1 using edge $(\sigma(v),v)$, where $\sigma(v)$ is the anchor of v. In the full paper, we show how to derive the final postorder numbering ν of the resulting tree T.

To make sure that tree T is a DFS-tree of G, we define the dangling subgraphs as follows: Let the edges $e_1, \ldots, e_k$ in $\tilde{E}$ be sorted by increasing postorder numbers of their sources. A DFS-traversal of G that is consistent with the partial DFS-pair (T_1, ν_1) computed for G_1 explores the edges in $\tilde{E}$ in this order. Hence, we add a vertex $w \in G_2$ to a dangling subgraph G_v if v is the target of the first edge (u,v) in this order so that v can reach w in G_2. Intuitively, the DFS would explore all vertices reachable from the target of the first edge (that is, the vertices in the first dangling subgraph). Then it would backtrack, explore the vertices reachable from the target of the next edge, and so on.

Since the computation of DFS-pairs (T_1, ν_1) and (T_v, ν_v), for the dangling subgraphs G_v, is carried out recursively, all that remains to be described is the computation of the dangling subgraphs: First we sort and scan the vertex set of components $\mathcal{C}_{p+1}, \ldots, \mathcal{C}_q$ and the edge set $\tilde{E}$ to find, for every component $\mathcal{C}_i \in G_2$, the first edge $e_{\lambda(i)} \in \tilde{E}$ whose target is in $\mathcal{C}_i$, if any. Now we spend $O(\mathrm{sort}(N))$ I/Os to topologically sort DAG G_2' and process its vertices in topologically sorted order to compute, for every vertex $v_i \in G_2'$, the vertex $v_{\rho(i)}$ that can reach v_i in G_2' and so that $\lambda(\rho(i))$ is minimized. We sort the strongly connected components of G_2 by the labels $\rho(i)$ of their corresponding vertices $v_i \in G_2'$ and define the dangling subgraph G_v, for the target v of an edge $e_{\lambda(i)}$, to be the subgraph of G_2 induced by the vertices in all components $\mathcal{C}_j$ with $\rho(j) = i$. This takes another $O(\mathrm{sort}(N))$ I/Os.

The correctness of this construction follows from the following two facts: (1) A vertex $v_i \in G_2'$ can reach a vertex $v_j \in G_2'$ if and only if all vertices $v \in \mathcal{C}_i$ can reach all vertices $w \in \mathcal{C}_j$. (2) All vertices of a component $\mathcal{C}_i$ are in the same dangling subgraph.

From the above discussion, we obtain that one recursive step of our algorithm takes $O(\mathrm{sort}(N))$ I/Os, so that the I/O-complexity of our algorithm is given by the recurrence $I(N) = O(\mathrm{sort}(N)) + I(|G_1|) + \sum_{G_v \neq \emptyset} I(|G_v|)$. Now recall that the recursion stops as soon as the current graph fits into memory or is strongly connected. In the former case, the computation can be finished in $O(N'/B)$ I/Os, where N' is the size of the current graph; below we show that, in the latter case, the computation can be finished in $O(\mathrm{sort}(N')\log(N'/M))$ I/Os. If we recurse, then graphs G_1 and G_2 are of approximately the same size, unless one of them contains a large strongly connected component. This is sufficient to show the following theorem.

Theorem 3 *Depth-first search in a planar graph G with N vertices takes $O(\mathrm{sort}(N)\log(N/M))$ I/Os.*

4.2. Strongly Connected and Bubble Graphs

To compute a DFS-tree T of a strongly connected planar graph G rooted at a given vertex s, we follow ideas from the parallel algorithm for this problem by Kao [11]. Kao's algorithm does in fact solve DFS in "bubble graphs", defined as follows: Let G be an embedded directed planar graph. We call a strongly connected component $\mathcal{C}$ of G a *source* or *sink component* if all edges with exactly one endpoint in $\mathcal{C}$ are out-edges or in-edges of $\mathcal{C}$, respectively. Graph G is a *bubble graph* if it has exactly one source component and there is a face f of G so that every source or sink component of G has at least one vertex on the boundary of f.

In our algorithm, we treat strongly connected graphs and bubble graphs differently, exploiting that we do not need the full machinery for bubble graphs to perform DFS in strongly connected graphs and then using the DFS-algorithm for strongly connected graphs to deal with the strongly connected components of bubble graphs less conservatively than Kao does. While this idea does not seem to lead to an improvement of the running time of the PRAM-algorithm, it saves a log-factor in the I/O-complexity of our algorithm.

Strongly connected graphs. To perform DFS in a strongly connected graph, we apply the algorithm from Section 3 to compute a simple-path $\frac{2}{3}$-separator S rooted at the given source vertex s. We consider path S to be a partial DFS-tree of G and compute the only possible postorder numbering ν_S of S. Then (S, ν_S) is a partial DFS-pair of G. Computing S and ν_S takes $O(\mathrm{sort}(N))$ I/Os. Now we compute dangling subgraphs G_v, for all vertices $v \in G - S$ that are out-neighbors of vertices in S. These subgraphs are defined w.r.t. (S, ν_S) in the same way as the dangling subgraphs of (T_1, ν_1) in Section 4.1. For each graph G_v, we compute a DFS-pair (T_v, ν_v) rooted at v and attach T_v to S using edge $(\sigma(v), v)$. Similar to the algorithm in Section 4.1, this produces a DFS-tree T of G; a postorder numbering ν so that (T, ν) is a DFS-pair is easily derived from numberings ν_S and ν_v, for all graphs G_v.

The two crucial observations are: (1) Since S is a $\frac{2}{3}$-separator, no graph G_v contains a strongly connected component of size more than $\frac{2}{3}N$. (2) Graphs G_v are bubble graphs [11]. Hence, we can use the DFS-algorithm for bubble graphs, outlined below, to compute DFS-pairs (T_v, ν_v), for all graphs G_v. The I/O-complexity $I_S(N)$ of our DFS-algorithm for strongly connected planar graphs is therefore given by the following recurrence, where $I_B(N,k)$ denotes the I/O-complexity of DFS in a bubble graph whose strongly connected components have size at most k:

$$I_S(N) = O(\mathrm{sort}(N)) + \textstyle\sum_{G_v \neq \emptyset} I_B\big(|G_v|, \frac{2}{3}N\big)$$

Bubble graphs. To perform DFS in a bubble graph G, where the source s of the DFS is contained in the source component of G, we start by computing a *splitting component* of G, that is, a strongly connected component $\mathcal{C}$ of G

so that every vertex $v \in \mathcal{C}$ can reach at least $N/2$ vertices of G, while no vertex $w \notin \mathcal{C}$ that is reachable from the vertices in $\mathcal{C}$ can reach $N/2$ vertices. We use the shortest path algorithm of [5] to compute a path P from s to the first vertex s' in $\mathcal{C}$, use the DFS-algorithm for strongly connected planar graphs to compute a DFS-pair (T_C, ν_C) of $\mathcal{C}$ rooted at s', and derive a partial DFS-pair (T', ν') with $T' = P \cup T_C$. Then we compute the dangling subgraphs G_v of T', compute a DFS-pair (T_v, ν_v), for each dangling subgraph G_v, and attach T_v to T' using edge $(\sigma(v), v)$. Again, graphs G_v are bubble graphs [11], so that DFS-pairs (T_v, ν_v) can be found by applying the algorithm recursively. The choice of component $\mathcal{C}$ ensures that no dangling subgraph has size more than $N/2$.

The central part of the algorithm is finding a splitting component $\mathcal{C}$. We do this as in the parallel algorithm of [11]: First we compress every strongly connected component $\mathcal{C}$ of G into a single vertex of weight $|\mathcal{C}|$. Our goal now is to find a vertex in the resulting DAG G' that can reach vertices of total weight at least $N/2$ and so that none of its out-neighbors has this property. This task is trivial, once we have computed the total weight of the vertices reachable from every vertex in G'. The latter can be achieved using the reachability counting algorithm of [15], after transforming G' into a planar st-graph G'' by adding a new sink t of weight zero to G' and adding an edge from every sink of G' to t. This is where it is important that G is a bubble graph, because it ensures that this construction does indeed produce a planar st-graph. The construction of graph G'' requires sorting and scanning the vertex and edge sets of G' in order to identify the sinks of G' and add an edge from each such sink to t. The reachability counting algorithm of [15] can be carried out in $O(\text{sort}(N))$ I/Os, using the time-forward processing technique and standard range-searching techniques. (Details appear in the full paper.) Hence, the I/O-complexity $I_B(N,k)$ of the DFS-algorithm for a bubble graph whose largest strongly connected component has size k is given by the following recurrence:

$$I_B(N,k) \leq O(\text{sort}(N)) + I_S(k) + \textstyle\sum_{G_v \neq \emptyset} I_B(|G_v|, |G_v|),$$

where $|G_v| \leq N/2$, for all G_v. Using substitution, we can show now that $I_S(N) \leq c \cdot \text{sort}(N)(4\log_{3/2}(N/M) - 1)$ and $I_B(N,k) \leq c \cdot \text{sort}(N)(2\log_{3/2}(N/M) + 2\log_{3/2}(\max(k, \frac{2}{3}N)/M))$, for some constant $c > 0$. This proves the following theorem.

Theorem 4 *DFS in a strongly connected planar graph G with N vertices takes $O(\text{sort}(N)\log(N/M))$ I/Os.*

References

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

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

[3] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447. Springer-Verlag, 2000.

[4] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 471–482. Springer-Verlag, 2001. To appear in *Journal of Graph Algorithms and Applications*, 7(2), 2003.

[5] L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.

[6] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[7] 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.

[8] E. D. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[9] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[10] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126:55–82, 2003.

[11] M.-Y. Kao. Planar strong connectivity helps in parallel depth-first search. *SIAM Journal on Computing*, 24:46–62, 1995.

[12] 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 Processing*, pages 169–176, 1996.

[13] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[14] 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.

[15] R. Tamassia and J. S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM Journal on Computing*, 20(4):708–725, 1991.

[16] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[17] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.