I/O-Efficient Topological Sorting of Planar DAGs*

Extended Abstract

Lars Arge
Dept. of Computer Science
Duke University
Durham, NC 27708-0129
large@cs.duke.edu

Laura Toma
Dept. of Computer Science
Duke University
Durham, NC 27708-0129
laura@cs.duke.edu

Norbert Zeh
Dept. of Computer Science
Duke University
Durham, NC 27708-0129
nzeh@cs.duke.edu

ABSTRACT

We present algorithms that solve a number of fundamental problems on planar directed graphs (planar digraphs) in $\mathcal{O}(\text{sort}(N))$ I/Os, where $\text{sort}(N)$ is the number of I/Os needed to sort N elements. The problems we consider are breadth-first search, the single-source shortest path problem, computing a directed ear decomposition of a strongly connected planar digraph, computing an open directed ear decomposition of a strongly connected biconnected planar digraph, and topologically sorting a planar directed acyclic graph.

1. INTRODUCTION

Recently external memory graph algorithms have received much 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 in recent years, a number of basic and important problems on general graphs still remain open. Since even basic problems seem hard for general graphs, several authors have considered special classes of graphs, and tremendous progress has been made especially for *planar undirected* graphs. Apart from being among the most fundamental combinatorial structures used in algorithmic graph theory, planar graphs arise naturally in many real life applications. For example, graphs encountered in GIS are often planar or "almost planar".

*The work in this paper was 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.

SPAA'03, June 7–9, 2003, San Diego, California, USA.

For planar *directed* graphs (planar digraphs), no I/O-efficient algorithms have been developed so far. In this paper we present the first I/O-efficient algorithms for several fundamental problems on planar digraphs. Our main result is an algorithm for topologically sorting a planar directed acyclic graph (planar DAG). In order to obtain this algorithm we develop algorithms for the single source shortest path problem and for computing a directed ear decomposition of a strongly connected (strong) planar digraph. We also consider the problem of computing an *open* directed ear decomposition of a biconnected strong planar digraph. Our results are a step towards understanding the I/O-complexity of problems on directed graphs and the first major progress on the longstanding open problem of I/O-efficient topological sorting.

1.1 Problem statement

Given a weighted digraph G, the *single source shortest path* (SSSP) problem is the well-known problem of finding the shortest paths from a given source vertex to all other vertices in G (the length of a path is the sum of the weights of the edges on the path). In this paper we assume that the weights are non-negative. The problem of computing a *breadth-first spanning tree* (BFS-tree) of an unweighted digraph is equivalent to solving the SSSP problem on the graph obtained by assigning weight one to every edge.

A *directed ear decomposition* $\mathcal{E} = (P_0, \ldots, P_k)$ of a digraph $G = (V, E)$ is a partition of E into simple directed paths $P_0, \ldots, P_k$, called *ears*, with the following properties: (1) The endpoints s_0 and t_0 of P_0 are the same vertex (P_0 is a simple directed cycle). (2) The endpoints s_i and t_i of each ear P_i, $i > 0$, are in two ears P_j and $P_{j'}$ with $j, j' < i$; but the internal vertices of P_i are not in any ear P_j with $j < i$. Ear P_i (and every edge $e \in P_i$) is said to have *index* i. A graph G has a directed ear decomposition *if and only if* it is strongly connected (strong) [11], that is, if for any pair of vertices $u, v \in V$, there is a directed cycle containing both u and v. An ear P_i is called *open* if its endpoints are distinct (i.e., P_i is not a cycle), and an ear decomposition $\mathcal{E}$ is *open* if P_0 is the only cycle in the decomposition.

Topological sorting is the problem of ordering the vertices of a graph $G = (V, E)$ so that for any edge $(u, v) \in E$, vertex u comes before vertex v. A graph can be topologically sorted *if and only if* it is acyclic.

1.2 I/O-Model and previous work

We work in the standard two-level I/O-model with one

(logical) disk [1]. The model defines the following parameters:[1]

N = number of vertices and edges ($N = V + E$),
M = number of vertices/edges that fit into internal memory,
B = number of vertices/edges that fit into a disk block,

where $B^2 \log^2 B \leq M < N$.[2] An *Input/Output* operation (or simply *I/O*) transfers one block of consecutive elements from (to) disk to (from) internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta(N/B)$ (the *linear* or *scanning* bound). The number of I/Os required to sort N items is $\text{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ (the *sorting* bound) [1]. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$, so that the difference in running time between an algorithm performing N I/Os and one performing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os can be very significant.

Most previous work on I/O-efficient graph algorithms has focused on *undirected* graphs. (See the surveys in [17, 18].) Despite considerable efforts, many fundamental problems on general undirected graphs remain open. For example, while $\Omega(\min\{V, \text{sort}(V)\})$ (which is $\Omega(\text{sort}(V))$ in practice) is a lower bound for the number of I/Os required to solve most graph problems, the best known algorithms for depth-first search (DFS) and SSSP perform $\Omega(V)$ I/Os [13, 6]. For BFS, an algorithm performing $o(V)$ I/Os has been developed only recently [16]. For directed graphs, even fewer results are known. The best know algorithms for directed SSSP, BFS, and topological sorting all use $\Omega(V)$ I/Os. More precisely, their I/O-complexity is $\mathcal{O}(\min\{(V+\frac{E}{B})\cdot \log V + \text{sort}(E), V + \frac{V}{M}\frac{E}{B}\})$ [5, 6, 13].

A number of improved algorithms have been developed for several special classes of graphs. For trees for example, $\mathcal{O}(\text{sort}(N))$ I/O algorithms are known for BFS and DFS-numbering, Euler tour computation, expression tree evaluation, topological sorting, as well as several other problems [5, 6]. Most problems on planar *undirected* graphs, including SSSP, BFS, and DFS, can also be solved in $\mathcal{O}(\text{sort}(N))$ I/Os [3, 4, 6, 14]. Almost all of these algorithms exploit the existence of small separators for planar graphs. More precisely, they use that for every planar graph G and any integer $h > 0$, there exists a set of $\mathcal{O}\left(N/\sqrt{h}\right)$ separator vertices whose removal partitions G into $\mathcal{O}(N/h)$ subgraphs of size at most h. Such a partition of G can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os [14]. While many $\mathcal{O}(\text{sort}(N))$ I/O algorithms are known for planar undirected graphs, similar results have not been obtained for planar *directed* graphs. Topological sorting has only been solved in $O(\text{sort}(N))$ I/Os for the very special case of planar *st*-graphs [6]. For general planar digraphs all known algorithms (the ones developed for general digraphs) perform $\Omega(N)$ I/Os, which can also be achieved by using the much simpler internal memory algorithms for these problems.

Many external memory graph algorithms use ideas from the corresponding PRAM-algorithms. In some cases it is even possible to obtain an I/O-efficient algorithm by "simulating" a PRAM-algorithm in a standard way (the so-called *PRAM simulation* [6]). Efficient PRAM algorithms have been developed for many problems on directed planar graphs (e.g. [10, 11, 12]); but simulation of these algorithms does not lead to I/O-efficient algorithms.

[1] For convenience we use use the name of a set to denote the actual set as well as its cardinality.

[2] Often it is only assumed that $2B \leq M$; but sometimes, as in this paper, the very realistic assumption is made that the main memory is big enough to hold $B^{2+\epsilon}$ elements.

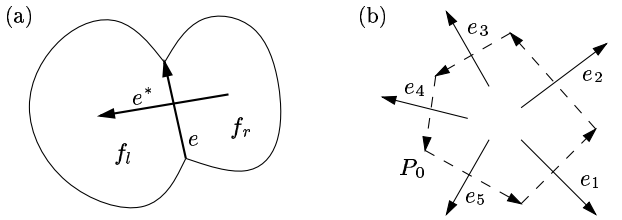

Figure 1: (a) The direction of a dual edge. (b) The relationship between a cycle in the dual and a topological ordering of the primal—all vertices inside P_0 precede all vertices outside P_0.

Based on an efficient algorithm for computing a directed spanning tree, Kao and Klein [11] showed how a directed ear decomposition of a strong planar digraph can be computed efficiently. Using this algorithm they develop an efficient algorithm for topologically sorting planar DAGs [10, 11]. The topological sorting algorithm is based on an important structural relationship between a planar embedded DAG $G = (V, E)$ and its *directed dual* $G^* = (V^*, E^*)$, which is defined similar to the dual of an undirected embedded planar graph: The *faces* of G are the connected regions of $\mathbb{R}^2 \setminus G$. A face f is said to be to the *left* (resp. *right*) of edge (u, v) if it is to the left (resp. right) when walking from u to v along edge (u, v). The directed dual of G contains a vertex $f^* \in V^*$ for every face f of G. For every edge $(v, w) \in E$, let f_l and f_r be the two faces to the left and right of edge $(v, w) \in E$. Then there is an edge $(f_r^*, f_l^*) \in E^*$ (see Figure 1a). As usual, we use v^*, e^*, and f^* to refer to the face, edge, and vertex that is dual to vertex v, edge e, and face f, respectively. The following lemma provides the connection between a DAG and its directed dual.

LEMMA 1 (KAO/KLEIN [10, 11]). *The directed dual of an embedded planar DAG is strong, that is, it has a directed ear decomposition.*

Given a planar DAG G and a directed ear decomposition $\mathcal{E} = (P_0, \ldots, P_k)$ of its dual G^*, the topological sorting algorithm by Kao and Klein [11] uses that because P_0 is a directed cycle, all edges of G crossing ear P_0 cross it in the same direction (see Figure 1b.) Hence, a topological ordering of G can be obtained by topologically sorting the subgraphs of G to the left and right of P_0 separately and placing all vertices to the left of P_0 before all vertices to the right of P_0. Since all subsequent ears partition the regions defined by previous ears in a similar manner, topological orderings of the subgraphs of G to the left and right of P_0 can be obtained recursively. To obtain an efficient PRAM algorithm (which halves the number of ears in each recursive step), Kao and Klein [11] modify this basic idea to consider the first $k/2$ ears simultaneously. Since the non-recursive part of this algorithm can be performed in $\mathcal{O}(\text{sort}(N))$ I/Os using PRAM simulation, this immediately leads to an $\mathcal{O}(\text{sort}(N) \cdot \log N)$ I/O algorithm.

1.3 Our results

In Section 2 we develop an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for the SSSP problem on planar digraphs. This algorithm is based on the recent I/O-efficient planar separator algorithm of [14] and utilizes ideas from the I/O-efficient shortest path algorithm for undirected planar graphs [3]. Of course, the algorithm can also be used to construct a BFS-tree of a planar digraph.

In Section 3 we develop an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for computing a directed ear decomposition of a strong planar digraph. This algorithm is based on Kao and Klein's algorithm [11], but is not obtained using PRAM simulation. It uses the SSSP algorithms developed in Section 2. We also consider the problem of computing an *open* directed ear decomposition. An efficient PRAM algorithm is know for the undirected version of this problem [15], but to our knowledge the directed version of the problem has not previously been considered. We show that a graph has an open directed ear decomposition if and only if it is strong and biconnected, and develop an $O(\text{sort}(N))$ I/O algorithm for such graphs.

In Section 4 we present the main result of our paper, namely an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for topologically sorting planar DAGs. As the PRAM algorithm by Kao and Klein [11], our algorithm utilizes the directed ear decomposition of the directed dual of the graph. However, in order to reduce the I/O-complexity to $\mathcal{O}(\text{sort}(N))$, we introduce the *ordered ear decomposition tree* and show how a topological ordering can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os using a traversal of this tree. The main contribution is then to prove a number of structural properties of the tree that allow us to construct it I/O-efficiently.

2. SINGLE SOURCE SHORTEST PATHS

In this section we sketch our algorithm for solving the SSSP problem on a planar digraph. This algorithm utilizes ideas from the I/O-efficient algorithm for the undirected version of the problem [3].

Given a planar digraph $G = (V, E)$ with $\mathcal{O}(N)$ vertices and edges, the first step of our algorithm is to compute a smaller graph G^R such that solving the SSSP problem on G reduces to solving it on G^R. In particular, we compute a small separator S of G that contains the source vertex s and build a graph G^R with vertex set S so that the distances from s to any vertex in S is the same in G and G^R. The separator S has size $\mathcal{O}(N/B)$ and partitions G into $\mathcal{O}(N/B^2)$ subgraphs $G_1, \ldots, G_q$ of size at most B^2 and boundary size at most B. The boundary ∂G_i of G_i is the set of separator vertices adjacent to vertices in G_i. We can obtain this partition in $\mathcal{O}(\text{sort}(N))$ I/Os by ignoring the directions of the edges in E and using the planar separator algorithm of [14]. For each graph G_i, we solve the all pairs shortest path problem on the subgraph R_i of G induced by the vertices in $V(G_i) \cup \partial G_i$. To obtain G^R we replace each such graph R_i with a complete digraph R_i' over the vertex set ∂G_i. The weight of an edge (v, w) in R_i' is the length of the shortest path from v to w in R_i. If there is no path from v to w in R_i, edge (v, w) has weight ∞. The resulting graph G^R has $O(N/B)$ vertices and $O(N)$ edges. G^R can be computed in $O(N/B)$ I/Os after computing the partition of G, as each graph R_i has at most $B^2 + B$ vertices and thus fits into main memory. It is easily verified that for any two vertices $v, w \in G^R$, the distance from v to w in G^R is the same as in G. In particular, since $s \in S$, the distances from s to all separator vertices can be computed by solving the SSSP problem on G^R. Since the shortest path from s to any vertex in a graph G_i has to contain a boundary vertex of G_i, the distances to all non-separator vertices of G can then be computed in $\mathcal{O}(N/B)$ I/Os by once more processing each graph R_i in turn and applying the formula $\text{dist}_G(s, v) = \min\{\text{dist}_G(s, u) + \text{dist}_{R_i}(u, v) : u \in \partial G_i\}$ for every vertex $v \in G_i$.

All that remains to be done is describe how to solve the SSSP problem on G^R. To do this we use a directed version of the algorithm used for the undirected case [3], i.e., a modified version of Dijkstra's algorithm [7]. Recall that Dijkstra's algorithm explores the graph using a priority queue that stores the vertices of G whose distances from s have not been determined yet. The priority of vertex v is equal to its tentative distance from s, i.e., the length of the shortest path from s to v found so far. The vertices are retrieved one by one from the priority queue, using DeleteMin operations. For each retrieved vertex, its priority is recorded as its distance from s, its adjacency list is retrieved, and the priorities of its out-neighbors are updated using DecreaseKey operations. In general, an I/O-efficient implementation of Dijkstra's algorithm has to address two problems: (1) The vertices of the graph are retrieved from the priority queue in an order that is hard to predict without solving the SSSP problem. Hence, it seems difficult to avoid spending at least one I/O to access each adjacency list, so that accessing all adjacency lists takes $\mathcal{O}(V + E/B)$ I/Os. In the case of graph $G^R = (S, E^R)$, however, we have $|S| = \mathcal{O}(N/B)$ and $|E^R| = \mathcal{O}(N)$, so that accessing all adjacency lists takes $\mathcal{O}(N/B)$ I/Os. Essentially, graph G^R is dense enough to amortize the I/Os spent on accessing adjacency lists at random over the I/Os required to read all edges in sequential order. (2) There is no known priority queue that supports Insert, DeleteMin, and DecreaseKey operations in $\mathcal{O}((1/B)\log_{M/B}(N/B))$ I/Os amortized, while for instance the buffer tree [2] supports Insert, Delete, and DeleteMin operations in this number of I/Os. The Delete operation requires the element to be deleted as well as its current priority as arguments, which is the only obstacle to simulating DecreaseKey operations by pairs of Delete and Insert operations. When running Dijkstra's algorithm on G^R, we can exploit the structure of G^R to provide every DecreaseKey operation with the current priority of the affected vertex, thereby facilitating the simulation of these operations using Delete and Insert operations, and we can do so in an I/O-efficient manner. In addition to the priority queue, we maintain a list L that stores the tentative distances of all vertices from s. When extracting a vertex v from the priority queue, we retrieve the tentative distances of its out-neighbors from L. For each out-neighbor w of v, we test whether its tentative distance is greater than the distance of v from s plus the weight of edge (v, w). If so, we update the distance of v in L, delete the old entry for w from the priority queue and insert a new entry for w with the updated distance into the priority queue. In total, we perform $\mathcal{O}(N)$ operations on the priority queue, for a total of $\mathcal{O}(\text{sort}(N))$ I/Os [2]. Even though we access list L $\mathcal{O}(N)$ times, $\mathcal{O}(1)$ times per edge in G^R, these accesses can be performed using only $\mathcal{O}(N/B)$ I/Os in total. The key to proving this is the notion of *boundary sets*: Two separator vertices are said to be in the same boundary set if they are

adjacent to the same set of subgraphs G_i. The boundary of each subgraph G_i has size at most B, so that each boundary set has size at most B. Hence, if we store the vertices in each boundary set consecutively in L, we can access these vertices in $\mathcal{O}(1)$ I/Os instead of spending one I/O per vertex. (Note that all vertices in a boundary set have the same neighbors in G^R, so that if one of them needs to be retrieved from L, the other vertices in the boundary set need to be retrieved as well.) Using a standard transformation, we can ensure that graph G has bounded degree. Under this assumption, we can ensure that the computed partition has only $\mathcal{O}(N/B^2)$ boundary sets [14]. Since every vertex in S has constant degree in G, it is adjacent to $\mathcal{O}(1)$ subgraphs G_i in G and hence has degree $\mathcal{O}(B)$ in G^R. Thus, every boundary set is accessed $\mathcal{O}(B)$ times in L, and the total cost of retrieving tentative distances from L is $\mathcal{O}(B \cdot N/B^2) = \mathcal{O}(N/B)$. We have shown the following result.

THEOREM 1. *The single source shortest path problem on a planar digraph with non-negative edge weights can be solved in $\mathcal{O}(sort(N))$ I/Os.*

3. DIRECTED EAR DECOMPOSITION

In this section we sketch how to compute directed ear decompositions of strong planar digraphs in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. In Section 3.1 we sketch the basic algorithm, which follows the PRAM-algorithm of [11]. In Section 3.2 we show how to modify the computed decomposition to obtain an *open* directed ear decomposition, provided that the graph is also biconnected.

3.1 Ear decomposition

An ear decomposition of an *undirected* graph G can be obtained as a collection of appropriate subpaths of fundamental cycles of a spanning tree of G [15]. The idea in the PRAM algorithm of Kao and Klein [11] for obtaining a *directed* ear decomposition of a strong digraph G is similar, but the construction makes use of two spanning trees T_c and T_d rooted in the same vertex r. The edges in T_c are directed towards r (T_c is called *convergent*); the edges in T_d are directed away from r (T_d is called *divergent*). We call pair (T_c, T_d) a *CD-pair*. For a strong planar digraph, trees T_c and T_d can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using our shortest path algorithm (Theorem 1).

Given the CD-pair, the algorithm of Kao and Klein consists of four main steps: Step 1 uses trees T_c and T_d to construct a sequence of directed (but not necessarily simple) paths $P_0^1, \ldots, P_{l_1}^1$ so that P_0^1 is a directed cycle and every subsequent path has its endpoints in paths of lower index. In Step 2 each of the paths $P_0^1, \ldots, P_{l_1}^1$ is broken into simple paths. Each such simple path may still share internal vertices and even edges with paths of lower index. In Step 3 every duplicate edge is removed from all but the ear of lowest index containing it. Every path sharing internal vertices with paths of lower index is broken into shorter paths at these internal vertices. This produces a directed ear decomposition of a strong subgraph of G that contains all vertices of G, but possibly not all edges. Step 4 adds each of these edges to the ear decomposition as a separate path, thereby producing an ear decomposition of G.

Steps 3 and 4 of Kao and Klein's algorithm can easily be performed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using a few sorting and scanning steps. The details of this will appear in the full paper. To see that Steps 1 and 2 can be performed in the same number of I/Os, we need to consider them in more detail.

To produce the initial collection $P_0^1, \ldots, P_{l_1}^1$ of paths in Step 1, consider the set $x_0, \ldots, x_{l_1}$ of leaves of T_c. For every leaf, we compute two paths A_i and B_i. Path A_i is the path from a_i to x_i in T_d, where a_i is the lowest ancestor of x_i that is on the path from r to a leaf x_j with $j < i$. Similarly, B_i is the path in T_c from x_i to the lowest ancestor b_i that is on a path from another vertex x_j, $j < i$, to the root r of T_c. Path P_i^1 is the concatenation of paths A_i and B_i. Each path P_i^1 starts at vertex a_i and ends at vertex b_i, so that P_0^1 is a directed cycle ($a_0 = b_0 = r$), and every path P_i^1, $i > 0$, has its endpoints in two paths with lower indices. Paths $A_0, \ldots, A_{l_1}$ and $B_0, \ldots, B_{l_1}$ can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using standard I/O-efficient tree algorithms [5, 6] (processing trees T_c and T_d from the leaves towards the root). In particular, it suffices to tag every vertex v of T_c or T_d with the lowest index x_j so that v is on the path from x_j to r in T_c or T_d, respectively.

To carry out Step 2, i.e., to break paths $P_0^1, \ldots, P_{l_1}^1$ into simple paths $P_0^2, \ldots, P_{l_2}^2$, we partition each path P_i^1 into simple paths $Q_{i,1}, \ldots, Q_{i,k_i}$ such that $Q_{i,1}$ has the same endpoints as P_i^1, and the endpoints of each path $Q_{i,j}$ are in paths $Q_{i,j'}$ with lower indices. Sequence $(P_0^2, \ldots, P_{l_2}^2)$ is the concatenation of sequences $(Q_{i,1}, \ldots, Q_{i,k_i})$, for $0 \le i \le l_1$: To construct paths $Q_{i,1}, \ldots, Q_{i,k_i}$ we compute the vertices $y_1, \ldots, y_{k_i} = x_i$ shared by A_i and B_i, in their order of appearance along A_i. Then $Q_{i,1}$ is the concatenation of the subpath of A_i from a_i to y_1 and the subpath of B_i from y_1 to b_i. For $j > 1$, $Q_{i,j}$ is the concatenation of the subpath of A_i from y_{j-1} to y_j and the subpath of B_i from y_j to the first vertex already contained in a path $Q_{i,j'}$ of lower index. Computing vertices $y_1, \ldots, y_{k_i}$ requires sorting and scanning the vertex sets of A_i and B_i. After that, paths $Q_{i,1}, \ldots, Q_{i,k_i}$ can be computed by considering paths A_i and B_i to be trees rooted at a_i and b_i, respectively, and repeating the computation of Step 1 using vertices $y_1, \ldots, y_{k_i}$ in place of the leaves of T_c. This takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, for all $P_0^1, \ldots, P_{l_1}^1$.

Since all four steps of the construction can be carried out in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, we obtain the following result.

THEOREM 2. *A directed ear decomposition of a strong planar digraph can be computed in $\mathcal{O}(sort(N))$ I/Os.*

It is interesting to observe that the only place where the planarity of G is used is in computing trees T_c and T_d.

3.2 Open ear decomposition

Even though our algorithm for topologically sorting a planar DAG presented in Section 4 only requires us to compute a directed ear decomposition, we sketch in this section how to compute an *open* directed ear decomposition for a strong and biconnected planar digraph. Details will appear in the full version of this paper, where we also prove that (similar to undirected graphs) a strong planar digraph has an open directed ear decomposition *if and only if* it is biconnected.

Our algorithm mimics the PRAM-algorithm for computing an open ear decomposition algorithm of an *undirected* graph [15]. First we compute a particularly well-structured CD-pair $(\tilde{T}_c, \tilde{T}_d)$ of G from a directed ear decomposition $\mathcal{E}$ obtained using Theorem 2: Tree $\tilde{T}_c$ is obtained by removing

the first edge f_i from every ear P_i in $\mathcal{E}$, and similarly $\tilde{T}_d$ is obtained by removing the last edge l_i from every ear P_i. Both trees are rooted at vertex $s_0 = t_0$. The trees can easily be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os. Then we use $\mathcal{E}$, $\tilde{T}_c$, and $\tilde{T}_d$ to derive an *open* ear decomposition $\mathcal{E}' = (P'_0, \ldots, P'_k)$ of G in $\mathcal{O}(\text{sort}(N))$ I/Os. This will show the following result.

THEOREM 3. *Given a strong biconnected planar digraph G, an open directed ear decomposition of G can be computed in $\mathcal{O}(sort(N))$ I/Os.*

Below we discuss how to obtain the open ear decomposition $\mathcal{E}'$ from $\mathcal{E}$, $\tilde{T}_c$, and $\tilde{T}_d$. First we present an algorithm that only guarantees that $\mathcal{E}'$ is an ear decomposition of G. Then we show how to modify the construction in order to guarantee that $\mathcal{E}'$ is open.

3.2.1 Computing Ear Decomposition $\mathcal{E}'$

$\mathcal{E}'$ is derived from a collection of simple paths and cycles $P_0 = \tilde{P}_0, \ldots, \tilde{P}_k$ that may share edges. Path $\tilde{P}_i$, $i > 0$, is obtained by extending ear P_i at end vertices s_i and t_i: We walk up $\tilde{T}_c$ and $\tilde{T}_d$ from s_i and t_i, respectively, until we find the first two vertices $\tilde{s}_i$ and $\tilde{t}_i$ that are in the same ear P_{ρ_i}, $\rho_i < i$. We denote the (possibly empty) paths from $\tilde{s}_i$ to s_i and from t_i to $\tilde{t}_i$ by $\tilde{A}_i$ and $\tilde{B}_i$, respectively. To eliminate edges that appear in more than one path $\tilde{P}_i$, we then define the priority of ear P_i as the pair $pr(P_i) = (\rho_i, i)$ and remove all duplicates of an edge except the one in the path $\tilde{P}_i$ defined by the ear P_i of lowest priority. This may split a path $\tilde{P}_i$ into more than one subpath; some subpaths may share internal vertices with subpaths of a path $\tilde{P}_j$ of lower priority. We split the paths at these internal vertices and make each resulting subpath of $\tilde{P}_i$ a separate ear in $\mathcal{E}'$. The order of the ears in $\mathcal{E}'$ is defined by the lexicographical order of the priorities of their defining ears P_i. It is not hard to see that every edge of G is in some ear of $\mathcal{E}'$, every ear is a simple path or cycle, and every ear, except $\tilde{P}_0$, has its endpoints in ears of lower index. Hence, $\mathcal{E}'$ is a directed ear decomposition of G.

Given indices $\rho_1, \ldots, \rho_k$, we can construct ear decomposition $\mathcal{E}'$ without constructing paths $\tilde{P}_0, \ldots, \tilde{P}_k$ explicitly. In particular, we process trees $\tilde{T}_c$ and $\tilde{T}_d$ from the leaves towards the root to compute priorities $pr_c(e) = \min\{pr(P_i) : \text{edge } e \text{ is on the path from the root to } t_i \text{ in } T_c\}$ and $pr_d(e) = \min\{pr(P_i) : \text{edge } e \text{ is on the path from the root to } s_i \text{ in } T_d\}$, for every edge $e \in G$. The priority $pr(e)$ of an edge $e \in P_i$ is then chosen to be $pr(e) = \min\{pr(P_i), pr_c(e), pr_d(e)\}$. The edges with priority $pr(P_i)$ define a collection of subpaths of $\tilde{P}_i$. The priorities of all edges, and thus these subpaths, can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os by applying standard I/O-efficient tree algorithms to $\tilde{T}_c$ and $\tilde{T}_d$. Finally, the ears of $\mathcal{E}'$ are obtained as described above by splitting these subpaths at internal vertices. This can be done in $\mathcal{O}(\text{sort}(N))$ I/Os using a few sorting and scanning steps. Hence, ear decomposition $\mathcal{E}'$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os if indices $\rho_1, \ldots, \rho_k$ can be found in $\mathcal{O}(\text{sort}(N))$ I/Os.

To find indices $\rho_1, \ldots, \rho_k$ we utilize two compressed versions T_c^* and T_d^* of $\tilde{T}_c$ and $\tilde{T}_d$. Both trees have vertex set $\{\alpha_0, \ldots, \alpha_k\}$, where vertex α_0 is the root of both trees, and such that the parent of a vertex α_i, $i > 0$, in T_c^* (resp. T_d^*) is the vertex α_j so that t_i (resp. s_i) is an internal vertex of ear P_j. These two trees can easily be constructed from $\mathcal{E}$ in $\mathcal{O}(\text{sort}(N))$ I/Os. Now ρ_i is the maximal index of all vertices α_j that are proper ancestors of α_i in T_c^* and T_d^*. To find these ancestors, for all vertices α_i, we represent every vertex α_i as a rectangle $R_i = [\nu_c(\alpha(i), \nu_c(\alpha_i) + |T_c^*(\alpha_i)|] \times [\nu_d(\alpha(i), \nu_d(\alpha_i) + |T_d^*(\alpha_i)|]$, where ν_c and ν_d are preorder numberings of T_c^* and T_d^*, respectively, and $|T(v)|$ denotes the number of descendants of vertex v in tree T. In the full paper we argue that these rectangles are non-intersecting and that R_{ρ_i} is the smallest rectangle containing R_i. Hence, finding indices $\rho_1, \ldots, \rho_k$ reduces to answering $k = \mathcal{O}(N)$ point location queries in a planar subdivision of size $\mathcal{O}(N)$. These queries can be answered in $\mathcal{O}(\text{sort}(N))$ I/Os [8]. Hence, we can compute ear decomposition $\mathcal{E}'$ in $\mathcal{O}(\text{sort}(N))$ I/Os.

3.2.2 Making $\mathcal{E}'$ Open

In order to guarantee that ear decomposition $\mathcal{E}'$ is open, we modify the second components of the priorities of the ears in $\mathcal{E}$ in the above computation. The purpose of this change of priorities is to guarantee that every currently closed ear in $\mathcal{E}'$ loses at least one edge; but no open ear gains enough edges to become closed. In particular, observe that a closed ear in $\mathcal{E}'$ is equal to some path $\tilde{P}_i$ so that $\tilde{s}_i = \tilde{t}_i$. Let $\tilde{f}_i$ and $\tilde{l}_i$ be the first and last edges of this path. We choose the new priorities so that edges $\tilde{f}_i$ and $\tilde{l}_i$ are guaranteed to be in different ears in $\mathcal{E}'$ unless $\tilde{s}_i \neq \tilde{t}_i$.

To compute the new priorities, we partition $\mathcal{E}$ into subsets $\mathcal{E}_i$, $0 \leq i \leq k$, so that for all ears $P_j \in \mathcal{E}_i$, $\rho_j = i$. We now define an undirected auxiliary graph H_i that contains one vertex $v(P_j)$ per ear $P_j \in \mathcal{E}_i$ as well as one vertex $v(e)$ per edge e in the set $\bigcup_{P_j \in \mathcal{E}_i} \{\tilde{f}_j, \tilde{l}_j\}$. Every vertex $v(P_j)$ has the two vertices $v(\tilde{f}_j)$ and $v(\tilde{l}_j)$ as neighbors in H_i. We call an ear P_j in $\mathcal{E}_i$ as well as vertex $v(P_j)$ *good* if either $\tilde{s}_j \neq \tilde{t}_j$, or one of edges $\tilde{f}_j$ or $\tilde{l}_j$ is contained in a path A_h or B_h, for some ear P_h with $\rho_h < \rho_j = i$. Intuitively, an ear P_j is good if either $\tilde{P}_j$ is not a cycle or one of the edges $\tilde{f}_i$ and $\tilde{l}_i$ is guaranteed to end up in an ear that is a subpath of $\tilde{P}_h$, where $\rho_h < \rho_j$. In the full paper, we prove the following lemma.

LEMMA 2. *Every connected component of H_i contains at least one good vertex.*

The proof idea for the lemma is to argue that if the lemma did not hold, then there would be a vertex $\tilde{s}_j = \tilde{t}_j$, for some j, so that $\tilde{s}_j$ is a cutpoint of G, thereby contradicting the fact that G is biconnected. Using this lemma, we can now define the priorities of the ears corresponding to the vertices in a connected component of H_i so that the ear with lowest priority is good and every subsequent path $\tilde{P}_i$ is guaranteed to lose at least one edge to a previous path $\tilde{P}_j$. This guarantees that all ears in $\mathcal{E}'$ except $\tilde{P}_0$ are open. To compute these priorities, we identify the connected components of each graph H_i, find a good vertex in each component, and perform BFS from the chosen good vertex in each component. Then the priority of ear P_j is (ρ_j, c_j, d_j), where c_j is a label identifying the component of H_{ρ_j} that contains $v(P_j)$, and d_j is the BFS-depth of $v(P_j)$ in this component.

Given the first and last edges $\tilde{f}_i$ and $\tilde{l}_i$ of every path $\tilde{P}_i$, the construction of graphs H_i, finding their connected components, and performing BFS take $\mathcal{O}(\text{sort}(N))$ I/Os because these graphs are easily seen to be planar. Hence, the only

difficult part of the algorithm is finding these edges and deciding which ear $P_i \in \mathcal{E}$ is good.

To compute edges $\tilde{f}_i$ and $\tilde{l}_i$, we observe that $\tilde{f}_i = f_x$ and $\tilde{l}_i = l_y$, where α_x is the child of α_{ρ_i} on the path from α_{ρ_i} to α_i in T_d^*, and α_y is the child of α_{ρ_i} on the path from α_{ρ_i} to α_i in T_c^*. These children can be identified in $\mathcal{O}(\text{sort}(N))$ I/Os by applying standard tree computations to T_c^* and T_d^*. Then we have to sort and scan the set of edges of the ears in $\mathcal{E}$ to extract edges $\tilde{f}_i$ and $\tilde{l}_i$, for all $P_i \in \mathcal{E}$.

Given edges $\tilde{f}_i$ and $\tilde{l}_i$, for all ears P_i, we can immediately identify all good ears P_i with $\tilde{s}_i \neq \tilde{t}_i$. To identify the second kind of good ears, it suffices to compute for every edge e, the lowest index ρ_i so that edge e is on the path from s_0 to s_i in T_d or from t_0 to t_i in T_c. If for an edge $e \in \{\tilde{f}_i, \tilde{l}_i\}$, this index is less than ρ_i, then ear P_i is good. The computation of this indices for all edges of G can again be carried out using standard tree algorithms. Hence, the whole algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os.

4. TOPOLOGICAL SORTING

As discussed in the introduction, the topological sorting algorithm of Kao and Klein [11] uses the following basic idea. By Lemma 1, the dual G^* of a DAG G is strongly connected and hence has a directed ear decomposition $\mathcal{E} = (P_0, \ldots, P_k)$. This decomposition can be used to define a total order σ of the vertices of G: If we consider the planar subdivision defined by ears $P_0, \ldots, P_{i-1}$, ear P_i splits a region of this subdivision into two. In σ the vertices in the left region precede all vertices in the right region. This way σ is consistent with the partial order defined by the edges of G because P_i is a directed path and hence all edges of G crossing P_i cross it in the same direction (see Figure 2). Thus σ is a topological ordering of G.

Our topological sorting algorithm uses the same general idea as the algorithm of Kao and Klein; but in order to obtain an $\mathcal{O}(\text{sort}(N))$ I/O algorithm, we utilize a tree structure, called the *ordered ear decomposition tree*, that describes the recursive partitioning of the plane using the ears in $\mathcal{E}$ and the resulting total order of the vertices of G. In Section Section 4.1 we define this tree and show that it can be used to compute σ in $\mathcal{O}(\text{sort}(N))$ I/Os. In Section Section 4.2 we describe how to compute this tree in $\mathcal{O}(\text{sort}(N))$ I/Os from a directed ear decomposition. Since the strongly connected dual of a DAG can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os [9], and the directed ear decomposition of this graph can also be computed in $\mathcal{O}(\text{sort}(N))$ I/Os (Theorem 2), this will prove the following result.

THEOREM 4. *A topological numbering of the vertices of a planar DAG can be computed in $\mathcal{O}(sort(N))$ I/Os.*

4.1 Computing a Topological Numbering from an Ordered Ear Decomposition Tree

The (unordered) *ear decomposition tree* (EDT) $T_{\mathcal{E}}$ of a directed ear decomposition $\mathcal{E} = (P_0, \ldots, P_k)$ of G^* (the dual of a DAG G) is a binary tree that represents the recursive partition of the plane obtained by incrementally adding ears $P_0, \ldots, P_k$ to the current subdivision. Tree $T_{\mathcal{E}}$ is defined recursively as follows: We start with the root ρ of $T_{\mathcal{E}}$ and define the region $R(\rho)$ represented by ρ as $R(\rho) = \mathbb{R}^2$. Given a node α representing a region $R(\alpha)$, α is a leaf if $R(\alpha)$ is a face of G^*. Otherwise let $E(\alpha)$ be the ear P_j of lowest index that is embedded inside $R(\alpha)$. Ear $E(\alpha)$ partitions $R(\alpha)$ into two regions R_1 and R_2. Then α has two children β and γ with $R(\beta) = R_1$ and $R(\gamma) = R_2$. As a result of this definition, every leaf α represents a face $R(\alpha)$ of G^*, and every internal node α represents a region $R(\alpha)$ and an ear $E(\alpha)$ splitting this region.

To obtain an *ordered ear decomposition tree* (OEDT), we impose the following ordering on the children of every node α in $T_{\mathcal{E}}$ (see Figure 3a): Let β and γ be the two children of α. Then β is the *left child* and γ is the *right child* of α if $R(\beta)$ is to the left of $E(\alpha)$ and $R(\gamma)$ is to the right of $E(\alpha)$. More precisely, let (u, v) be an edge dual to an edge in $E(\alpha)$. Node β is the left child of α if $u \in R(\beta)$ and the right child otherwise. In what follows we will use $T_{\mathcal{E}}$ to refer to both the unordered and ordered versions of the EDT—the meaning will be clear from the context.

Recall that each leaf in the OEDT $T_{\mathcal{E}}$ for G^* corresponds to a vertex in G. It is easy to see that the topological ordering σ of the vertices in G defined above is identical to the order in which the leaves of $T_{\mathcal{E}}$ are visited in a recursive traversal that starts at the root and visits the left child of every vertex before its right child. Such a traversal (Euler tour) can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os [6]. Ordering σ can then be computed by applying the list-ranking algorithm of [6] in order to number the leaves of T in their order of appearance in the Euler tour. This shows the following lemma.

LEMMA 3. *Let G^* be the strongly connected dual of a planar DAG G and $\mathcal{E}$ a directed ear decomposition of G^*. Given an OEDT $T_{\mathcal{E}}$ of $\mathcal{E}$, a topological numbering of the vertices of G can be computed in $\mathcal{O}(sort(N))$ I/Os.*

4.2 Computing an Ordered Ear Decomposition Tree

Given an unordered EDT $T_{\mathcal{E}}$, it can be ordered as follows: First we arbitrarily choose an ordering of the two children of every internal node. Based on this ordering, we number the leaves of $T_{\mathcal{E}}$ from left to right using the procedure described in Section 4.1. For every node α, we select an edge (v, w) of G whose index is i, where $E(\alpha) = P_i$. If v precedes w in σ, the order of the children of α was chosen correctly. Otherwise we reverse the order of these children. Since this test can be performed for all nodes simultaneously, we can order an unordered EDT in $\mathcal{O}(\text{sort}(N))$ I/Os.

In the remainder of this section we show how to compute an unordered EDT $T_{\mathcal{E}}$ of G^*, given an ear decomposition

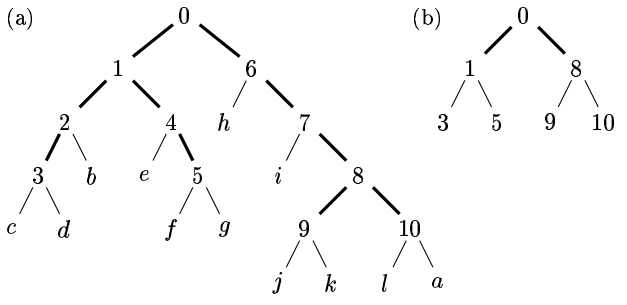

Figure 3: (a) The ordered ear decomposition tree $T_{\mathcal{E}}$ of the ear decomposition in Figure 2b. Subtree $T'_{\mathcal{E}}$ is shown in bold. (b) The compressed version $T^c_{\mathcal{E}}$ of $T'_{\mathcal{E}}$.

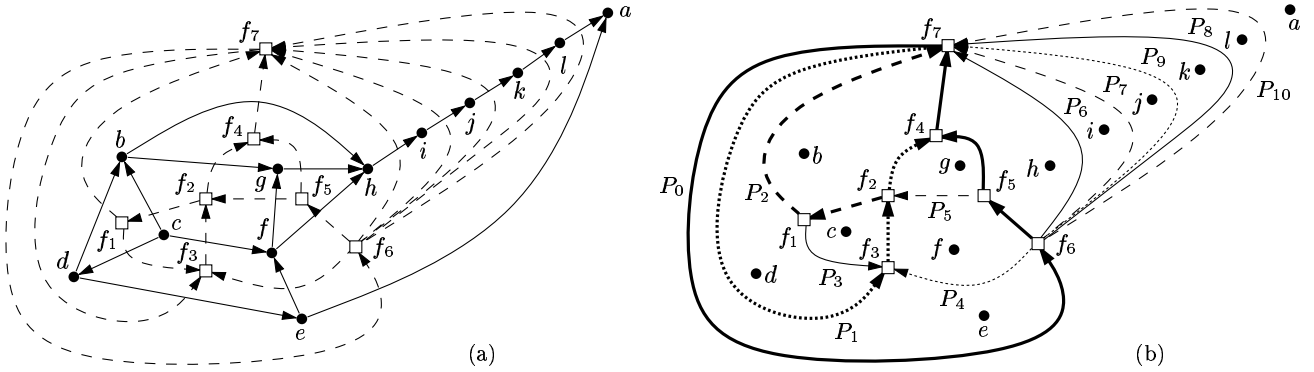

Figure 2: (a) A DAG and its dual. (b) A directed ear decomposition $\mathcal{E} = (P_0, \ldots, P_{10})$ of the dual. All edges of G crossing P_0 are directed from the left (inside) of P_0 to the right (outside) of P_0. Therefore vertices b, c, d, e, f, g to the left of P_0 precede vertices a, h, i, j, k, l to the right of P_0 in the topological ordering we construct. Similarly, for ear P_1, vertices b, c, d to the left of P_1 precede vertices e, f, g to the right of P_1 in the ordering.

$\mathcal{E}$. Since $T_\mathcal{E}$ is unordered we can ignore the directions of all edges in G, G^*, and $\mathcal{E}$. The vertex set of $T_\mathcal{E}$ is easily generated, as it contains one leaf per vertex of G and one internal vertex per ear $P_i \in \mathcal{E}$. To compute the edge set of $T_\mathcal{E}$, we identify the parent of every node in $T_\mathcal{E}$, except the root (which of course does not have a parent). The root of $T_\mathcal{E}$ is easily identified, as it is the node corresponding to ear P_0. Since the parent of every non-root node α is an internal node, which represents some ear P_{j_α}, our goal is hence to compute the index j_α of the ear associated with the parent of α.

To compute j_α we first observe that j_α is the maximal index of the edges on the boundary of region $R(\alpha)$. To compute these indices efficiently, we use a well-chosen spanning tree T of G, which we call the *spanning tree of ear decomposition* $\mathcal{E}$ (see Figure 4a): We define the index of an edge e in G to be equal to the index of edge e^* in G^*. For all $0 \leq i \leq k$, tree T contains exactly one edge e_i of G with index i. Apart from that, the choice of the edge set of T is arbitrary. Intuitively spanning tree T is chosen so that it contains exactly one edge crossing every ear in the ear decomposition of the dual G^* of G. For any node $\alpha \in T_\mathcal{E}$ we can prove that the leaves below α induce a subtree in T, which we denote by $T(\alpha)$. We use j'_α to denote the maximal index of all edges in T with exactly one endpoint in $T(\alpha)$.

Now we make the following two observations: (1) The edges in T with exactly one endpoint in $T(\alpha)$ are a subset of the edges in G with exactly one endpoint in $R(\alpha)$. (2) All edges in G with index j_α (including e_{j_α}) have exactly one endpoint in $R(\alpha)$. The following lemma is an immediate consequence of these two observations.

LEMMA 4. *For every node $\alpha \in T_\mathcal{E}$, except the root, $j_\alpha = j'_\alpha$.*

The main problem with the characterization provided by Lemma 4 is that region $R(\alpha)$ and hence tree $T(\alpha)$ is only known if α is a leaf of $T_\mathcal{E}$. Nevertheless, we can use it to design an algorithm for computing $T_\mathcal{E}$: Below we show that a *leafless* version $T'_\mathcal{E}$ of $T_\mathcal{E}$ (i.e., $T_\mathcal{E}$ with all its leaves removed; see Figure 3a) can be computed by recursive application of our algorithm. Since every leaf α corresponds to a vertex $v \in G$, Lemma 4 enables us to find the parent of every leaf in $T_\mathcal{E}$ by inspecting the edges in T incident to v. This takes $\mathcal{O}(\text{sort}(N))$ I/Os for all leaves of $T_\mathcal{E}$.

To compute $T'_\mathcal{E}$ we define a *compressed* version $T^c_\mathcal{E}$ of $T'_\mathcal{E}$ as the tree obtained by replacing every maximal path in $T'_\mathcal{E}$ whose internal nodes have degree two (one child and one parent) with a single edge (see Figure 3b). We also define a *compressed* version T^c of T. To do this we partition the edges of T into two categories (see Figure 4a); We say that an edge is *large* if it is the edge with highest index incident to one of its endpoints. Otherwise it is *small*. Tree T^c is obtained from T by contracting all large edges (see Figure 4b). In particular, tree T^c contains all small edges of T. Every vertex of T^c represents a connected component of the subgraph T^+ of T induced by all large edges. T^c can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using a few sorting and scanning steps.

In Section 4.2.1 we show how $T^c_\mathcal{E}$ can be computed recursively from T^c, by showing that there exists an ear decomposition $\mathcal{F}$ that has T^c as spanning tree and $T^c_\mathcal{E}$ as ear decomposition tree. We also show that $|T^c| \leq N/2$. In Section 4.2.2 we show how to obtain $T'_\mathcal{E}$ from $T^c_\mathcal{E}$ in $\mathcal{O}(\text{sort}(N))$ I/Os. In total, the I/O-complexity of our algorithm is given by the recurrence $\mathcal{I}(N) = \mathcal{I}(N/2) + \mathcal{O}(\text{sort}(N))$, which solves to $\mathcal{I}(N) = \mathcal{O}(\text{sort}(N))$. Thus we obtain the following lemma.

LEMMA 5. *An unordered EDT $T_\mathcal{E}$ of ear decomposition $\mathcal{E}$ can be computed in $\mathcal{O}(sort(N))$ I/Os.*

4.2.1 Recursive Computation of the Compressed Ear Decomposition Tree

The first step towards showing that an ear decomposition $\mathcal{F}$ with spanning tree T^c and ear decomposition $T^c_\mathcal{E}$ exists is proving that there exists a natural bijection between the internal vertices of $T^c_\mathcal{E}$ and the edges of T^c.

LEMMA 6. *For every internal node $\alpha \in T^c_\mathcal{E}$ with $E(\alpha) = P_i$, edge e_i is an edge of T^c, and vice versa.*

PROOF. The internal nodes of $T^c_\mathcal{E}$ are exactly the nodes with two children in $T'_\mathcal{E}$. For such a node α, neither of its two children β and γ in $T_\mathcal{E}$ is a leaf. Hence, neither $R(\beta)$ nor $R(\gamma)$ is a face of G^*. This implies that both faces corresponding to the endpoints of edge e_i, where $E(\alpha) = P_i$, have an edge of index larger than i on their boundaries. By

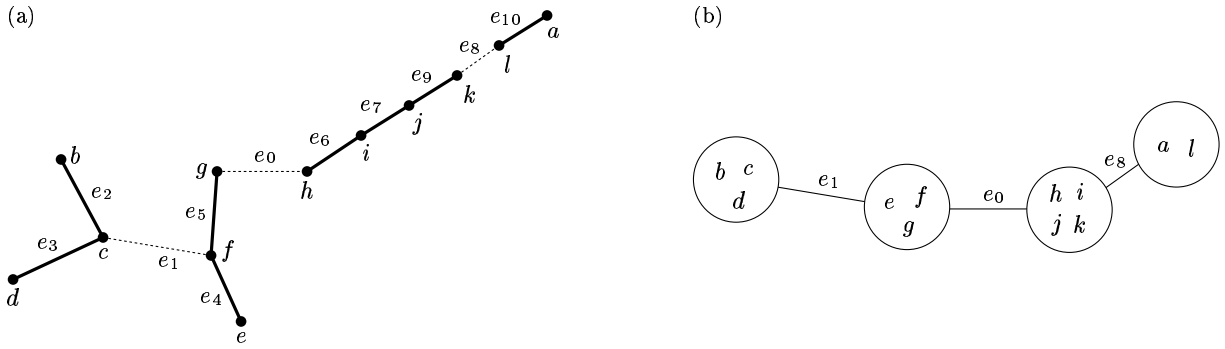

Figure 4: (a) Tree T. The large edges are fat. Small edges are dotted. The fat edges are in T^+. (b) Tree T^c. The vertices are labeled with the vertices in T they represent.

Lemma 4, this implies that edge e_i is not large for either of its two endpoints, so that $e_i \in T^c$.

A vertex α with less than two children in $T'_\mathcal{E}$ has at least one child β in $T_\mathcal{E}$ that is a leaf, so that $R(\beta)$ is a face v^* of G^*. Face v^* is a face of the subdivision defined by ears $P_0, \ldots, P_i$, but not of the subdivision defined by ears $P_0, \ldots, P_{i-1}$. Hence, the edge with largest index on the boundary of v^* has index i. By Lemma 4, this implies that edge e_i is the largest edge incident to vertex v in T, so that e_i is large, and $e_i \notin T^c$. □

Using Lemma 6, we can now prove the existence of an ear decomposition with spanning tree T^c and ear decomposition tree $T^c_\mathcal{E}$, which allows the recursive construction of $T^c_\mathcal{E}$ from T^c.

LEMMA 7. *There exists an ear decomposition $\mathcal{F}$ of the dual H^* of an embedded planar graph H that has T^c as its spanning tree and tree $T^c_\mathcal{E}$ as its ear decomposition tree.*

PROOF. Let α be a node in $T'_\mathcal{E}$ with children β and γ, and let $E(\alpha) = P_i$. Then the removal of edge e_i divides $T(\alpha)$ into $T(\beta)$ and $T(\gamma)$. The nodes in $T'_\mathcal{E}(\beta)$ and $T'_\mathcal{E}(\gamma)$ correspond to the edges in $T(\beta)$ and $T(\gamma)$, respectively. In particular, by Lemma 6, the nodes with two children in $T'_\mathcal{E}(\beta)$ and $T'_\mathcal{E}(\gamma)$ correspond to the small edges in $T(\beta)$ and $T(\gamma)$. Hence, if $\alpha \in T^c_\mathcal{E}$, then edge e_i divides $T^c(\alpha)$ into the subtrees $T^c(\beta')$ and $T^c(\gamma')$, where β' and γ' are the children of α in $T^c_\mathcal{E}$. This allows the following recursive definition of $\mathcal{F}$:

Consider a drawing of T^c in the plane. For the root ρ' of $T^c_\mathcal{E}$ with $E(\rho') = P_i$, we draw a cycle that intersects only edge e_i, thereby defining two regions $R(\alpha)$ and $R(\beta)$ for the two children α and β of ρ' in $T^c_\mathcal{E}$ so that $T^c(\alpha)$ is embedded inside $R(\alpha)$ and $T^c(\beta)$ is embedded inside $R(\beta)$.

For every non-root node α with children β and γ in $T^c_\mathcal{E}$, let $E(\alpha) = P_j$. Since e_j partitions $T^c(\alpha)$ into subtrees $T^c(\beta)$ and $T^c(\gamma)$, we can split region $R(\alpha)$ into two regions $R(\beta)$ and $R(\gamma)$ by adding an ear $E(\alpha)$ with both endpoints on the boundary of $R(\alpha)$ and intersecting only e_j. Then $T^c(\beta)$ is contained in $R(\beta)$, and $T^c(\gamma)$ is contained in $R(\gamma)$. Continuing this construction to the leaves of $T^c_\mathcal{E}$ produces the desired ear decomposition $\mathcal{F}$. □

By Lemma 7, we can construct $T^c_\mathcal{E}$ recursively from T^c. Now observe that every vertex in T has one incident edge that is large. On the other hand, every edge is large for at most two vertices: its two endpoints. Hence, there are at least $N/2$ large edges in T, and $|T^c| \leq N/2$.

4.2.2 *Deriving the Leafless Ear Decomposition Tree*

To construct $T'_\mathcal{E}$ from $T^c_\mathcal{E}$, we start by mapping missing vertices of $T'_\mathcal{E}$ to the leaves of $T^c_\mathcal{E}$. In particular, there exists a natural bijection between the leaves of $T^c_\mathcal{E}$ and the connected components of T^+. Every edge in such a connected component corresponds to a missing vertex of $T'_\mathcal{E}$; we map this vertex to the leaf of $T^c_\mathcal{E}$ corresponding to this component. Given this mapping of missing vertices to the leaves of $T^c_\mathcal{E}$, we show that tree $T'_\mathcal{E}$ can be obtained by inserting every missing vertex β on an edge of $T^c_\mathcal{E}$ connecting two appropriate ancestors of α, where α is the leaf β was mapped to.

The bijection between the connected components of T^+ and the leaves of $T^c_\mathcal{E}$ is provided by the following lemma, whose proof we provide in the full paper.

LEMMA 8. *Consider a connected component T' of T^+. Then the set of nodes in $T'_\mathcal{E}$ corresponding to the edges in T' contains exactly one leaf α of $T^c_\mathcal{E}$, and all other nodes in this set are ancestors of α in $T'_\mathcal{E}$.*

By Lemma 8, we can obtain $T'_\mathcal{E}$ by inserting every node β mapped to a leaf α on an edge connecting two appropriate ancestors of α. To find the correct edge for every node β and to arrange the nodes inserted on the same edge in the right order, we use the monotonicity of root-to-leaf paths in $T'_\mathcal{E}$. In particular, observe that for a node α with $E(\alpha) = P_i$ and any proper ancestor β of α in $T'_\mathcal{E}$ with $E(\beta) = P_j$, $j < i$. Hence, we can use the following three-step approach to construct $T'_\mathcal{E}$ from $T^c_\mathcal{E}$:

First we map the connected components of T^+ (and the corresponding nodes of $T'_\mathcal{E}$) to the leaves of $T^c_\mathcal{E}$. This requires sorting and scanning the set of leaves of $T^c_\mathcal{E}$ and the edge set of T^+ to find for every connected component T' of T^+, the leaf α of $T^c_\mathcal{E}$ with $E(\alpha) = P_i$so that $e_i \in T'$. Next, to map every node β currently mapped to a leaf α of $T^c_\mathcal{E}$ to the appropriate edge (γ, δ) of $T^c_\mathcal{E}$, we find the two ancestors γ and δ of α so that $h < i < j$, where $E(\gamma) = P_h$, $E(\beta) = i$, and $E(\delta) = j$. In the full paper, we argue that finding vertices γ and δ is an oblivious search query on tree $T^c_\mathcal{E}$ in the sense defined in [18]. Hence, these queries can be answered in $\mathcal{O}(\text{sort}(N))$ I/Os for all missing nodes of $T'_\mathcal{E}$, using the topology buffer tree [18]. Finally, to replace every edge e of $T^c_\mathcal{E}$ with the correct path in $T'_\mathcal{E}$, we sort the nodes mapped to edge e by increasing indices of their associated ears and replace edge e with a path having these nodes as internal vertices, in this order. This takes another $\mathcal{O}(\text{sort}(N))$ I/Os.

5. REFERENCES

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

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

[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.

[5] 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.

[6] 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, January 1995.

[7] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[8] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

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

[10] M.-Y. Kao. Linear-processor NC algorithms for planar directed graphs. I. strongly connected components. *SIAM Journal on Computing*, 22(3):431–459, 1993.

[11] M.-Y. Kao and P. N. Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. *Journal of Computer and System Sciences*, 47(3):459–500, 1993.

[12] M.-Y. Kao and G. E. Shannon. Linear-processor NC algorithms for planar directed graphs. II. directed spanning trees. *SIAM Journal on Computing*, 22(3):460–481, 1993.

[13] 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, October 1996.

[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] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[16] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*. Springer-Verlag, 2002. To appear.

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

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