I/O-Efficient Algorithms for Sparse Graphs

Laura Toma and Norbert Zeh*

Department of Computer Science, Duke University, Durham, NC 27708, USA
{laura, nzeh}@cs.duke.edu

Abstract. We survey I/O-efficient algorithms for problems on sparse graphs. We call a graph \( G \) sparse if \( |E(H)| = O(|V(H)|) \) for any graph \( H \) that can be obtained from \( G \) by a series of edge contractions followed by removing duplicate edges.

We emphasize graph contraction and time-forward processing as the two fundamental algorithmic techniques used in these algorithms and point out the structural properties of the graph classes we consider that can be exploited in I/O-efficient algorithms for these graphs.

We start the discussion with an overview of algorithms for connectivity problems such as finding the connected components (CC), biconnected components (BCC), a minimum spanning tree (MST), or an ear-decomposition of a graph. The algorithms used to solve these problems are the ones presented in Chapter ??, however, due to the above sparseness property these algorithms perform much better on sparse graphs than on general graphs.

In the second part of this chapter we consider searching problems such as breadth-first search (BFS), depth-first search (DFS), and single source shortest paths (SSSP). The performance of the general algorithms for these problems breaks down on sparse graphs. The algorithms for these problems presented in this chapter exploit structural properties of special classes of sparse graphs such as outerplanar graphs, planar graphs, grid graphs, and graphs of bounded treewidth in order to achieve good and often optimal performance on these graph classes. The SSSP and BFS-algorithms are based on the existence of small separators for these classes of graphs, while the DFS-algorithms for the first three graph classes exploit their geometric structure.

In order for the algorithms to exploit structural information about the given graph, efficient algorithms are needed that compute this information explicitly. We discuss I/O-efficient algorithms for computing planar and outerplanar embeddings for planar and outerplanar graphs, respectively, tree-decompositions of graphs of bounded treewidth, and small separators for graphs in any of these classes.

1 Introduction

Massive graphs arise naturally in many applications. Recent web crawls, for example, produce graphs with on the order of 200 million nodes and 2 billion edges.

* Part of this work was done when the second author was a Ph.D. student at the School of Computer Science of Carleton University, Ottawa, Canada.
Recent research in web modeling uses depth-first search, breadth-first search, and
the computation of shortest paths and connected components as primitive rou-
tines for investigating the structure of the web [14]. Massive graphs are also often
manipulated in Geographic Information Systems (GIS), where many problems
can be formulated as fundamental graph problems. When working with such
massive data sets, only a fraction of the data can be held in the main memory
of a state-of-the-art computer. Thus, the transfer of data between main memory
and secondary, disk-based memory, and not the internal memory computation,
is often the bottleneck. A number of models have been developed for the pur-
pose of analyzing this bottleneck and designing algorithms that minimize the
traffic between main memory and disk. The algorithms discussed in this chap-
ter are designed and analyzed in the parallel disk model (PDM) of Vitter and
Shriver [36]. For a definition and discussion of this model, the reader may refer
to Chapter ??.

Despite the efforts of many researchers [1, 2, 5–7, 15, 16, 19, 24, 25, 27, 28, 32,
33, 35], the design of I/O-efficient algorithms for basic graph problems is still a
research area with many challenging open problems. For most graph problems,
\( \Omega(\text{perm}(|V|)) \) or \( \Omega(\text{sort}(|V|)) \) are lower bounds on the number of I/Os required
to solve them [6, 16], while the best known algorithms for these problems on
general graphs perform considerably more I/Os. For example, the best known
algorithms for DFS and SSSP perform \( \Omega(|V|) \) I/Os in the worst case; for BFS an
type algorithm performing \( o(|V|) \) I/Os has been proposed only recently (see Table 1).
While these algorithms are I/O-efficient for graphs with at least \( B \cdot |V| \) edges,
they are inefficient for sparse graphs.

In this chapter we focus on algorithms that solve a number of fundamental
graph problems I/O-efficiently on sparse graphs. The algorithms we discuss,
besides exploiting the combinatorial and geometric properties of special classes
of sparse graphs, demonstrate the power of two general techniques applied in I/O-
efficient graph algorithms: graph contraction and time-forward processing. The
problems we consider are computing the connected and biconnected components
(CC and BCC), a minimum spanning tree (MST), or an ear decomposition of

<table>
<thead>
<tr>
<th>Problem</th>
<th>General undirected graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC, MST</td>
<td>( O(\text{sort}(</td>
</tr>
<tr>
<td>SSSP</td>
<td>( O(</td>
</tr>
<tr>
<td>DFS</td>
<td>( O(</td>
</tr>
<tr>
<td></td>
<td>( O((</td>
</tr>
<tr>
<td>BFS</td>
<td>( O\left(\sqrt{</td>
</tr>
</tbody>
</table>
Table 2. The problems that can be solved in $O(\text{sort}(N))$ I/Os and linear space on sparse graphs and the sections where they are discussed. A left-arrow indicates that the problem can be solved using the more general algorithm to the left.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Graph class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sparse</td>
</tr>
<tr>
<td>CC, BCC, MST</td>
<td>5</td>
</tr>
<tr>
<td>Ear decomposition</td>
<td>5</td>
</tr>
<tr>
<td>BFS + SSSP</td>
<td>open</td>
</tr>
<tr>
<td>DFS</td>
<td>open</td>
</tr>
<tr>
<td>Graph partition</td>
<td>N/A</td>
</tr>
<tr>
<td>Embedding</td>
<td>N/A</td>
</tr>
<tr>
<td>Tree-decomposition</td>
<td>N/A</td>
</tr>
</tbody>
</table>

the given graph, breadth-first search (BFS), depth-first search (DFS), and single source shortest paths (SSSP). The first four problems can be categorized as connectivity problems, while the latter three can be considered graph searching problems.

In our discussion we emphasize that graph contraction almost immediately leads to I/O-efficient solutions for connectivity problems on sparse graphs because edge contractions preserve the connectivity of the graph. For graph searching problems, the algorithms we discuss exploit structural properties of the graph classes we consider such as having small separators, outerplanar or planar embeddings, or tree-decompositions of constant width. The graph classes we consider are outerplanar graphs, planar graphs, grid graphs and graphs of bounded treewidth. A crucial condition for exploiting the structural properties of these graph classes in algorithms is the ability to compute such information explicitly. We discuss I/O-efficient algorithms that compute outerplanar and planar embeddings of outerplanar and planar graphs, tree-decompositions of graphs of bounded treewidth, and small separators of graphs in any of these classes. Table 2 gives an overview of the algorithms for sparse graphs discussed in this chapter.

2 Overview of the Paper

In Section 3 we define the graph classes we consider. In Section 4 we describe the two fundamental algorithmic techniques used to solve graph problems I/O-efficiently on sparse graphs. In each of Sections 5 through 9 we discuss a different graph problem. Sections 6 through 9 are further divided into subsections describing the different solutions of the considered problem for different classes of sparse
graphs. While we present these solutions separately, we emphasize the common ideas behind these solutions. We conclude in Section 10 with a summary and a discussion of some open problems.

3 Definitions and Graph Classes

The notation and terminology used in this chapter are quite standard. The reader may refer to [18, 23] for definitions of basic graph-theoretic concepts. For clarity, we review a few basic definitions and define the graph classes considered in this chapter.

Given a graph $G = (V, E)$ and an edge $(v, w) \in E$, the **contraction** of edge $(v, w)$ is the operation of replacing vertices $v$ and $w$ with a new vertex $x$ and every edge $(u, y)$, where $u \in \{v, w\}$ and $y \notin \{v, w\}$, with an edge $(x, y)$. This may introduce duplicate edges into the edge set of $G$. These edges are removed. We call graph $G$ sparse if $|E'| = O(|V'|)$ for any graph $H = (V', E')$ that can be obtained from $G$ through a series of edge contractions.\(^1\)

A **planar embedding** $G$ of a graph $G = (V, E)$ is a drawing of $G$ in the plane so that every vertex is represented as a unique point, every edge is represented as a contiguous curve connecting its two endpoints, and no two edges intersect, except possibly at their endpoints. A graph $G$ is **planar** if it has a planar embedding.

Given an embedded planar graph, the **faces** of $G$ are the connected components of $\mathbb{R}^2 \setminus \tilde{G}$. The **boundary** of a face $f$ is the set of vertices and edges contained in the closure of $f$.

A graph $G = (V, E)$ is **outerplanar** if it has a planar embedding one of whose faces has all vertices of $G$ on its boundary. We call this face the **outer face** of $G$.

A **grid graph** is a graph whose vertices are a subset of the vertices of a $\sqrt{N} \times \sqrt{N}$ regular grid. Every vertex is denoted by its coordinates $(i, j)$ and can be connected only to those vertices whose coordinates differ by at most one from its own coordinates. Note that a grid graph is not necessarily planar because diagonal edges may intersect.

A **tree-decomposition** of a graph $G = (V, E)$ is a pair $\mathcal{D} = (T, \mathcal{X})$, where $T = (I, F)$ is a tree and $\mathcal{X} = \{X_i\}_{i \in I}$ is a collection of vertex sets satisfying the following properties (see Fig. 1):

(i) $\bigcup_{i \in I} X_i = V$,
(ii) For every edge $(v, w) \in E$, there exists a node $i \in I$ so that $v, w \in X_i$,
(iii) For two nodes $i, k \in T$ and any node $j$ on the path from $i$ to $k$ in $T$, $X_i \cap X_k \subseteq X_j$.

The **width** of tree-decomposition $\mathcal{D}$ is defined as $\max \{|X_i| : i \in I\} - 1$. The **treewidth** of graph $G$ is the minimum width of all its tree-decompositions. Intuitively, the treewidth of a graph measures how close the graph is to being a tree.

\(^1\) The authors of [16] call these graphs “sparse under edge contraction”, thereby emphasizing the fact that the condition $|E| = O(|V|)$ is not sufficient for a graph to belong to this class.
Fig. 1. A graph of treewidth three and a tree-decomposition of width three for the graph.

A class $C$ of graphs is said to have bounded treewidth if there exists a constant $k$ so that all graphs in $C$ have treewidth at most $k$. Outerplanar graphs, for example, have treewidth two. For algorithmic purposes, a particular type of tree-decomposition is especially useful: A nice tree-decomposition of a graph $G$ is a tree-decomposition $D = (T, \mathcal{X})$ with the following additional properties:

(iv) Tree $T$ is a rooted binary tree, and

(v) Every internal node of $T$ is of one of the following types: A forget node $i \in T$ has one child $j$, and $X_i = X_j \setminus \{x\}$, for some vertex $x \in X_j$. An introduce node $i \in T$ has one child $j$, and $X_i = X_j \cup \{x\}$, for some vertex $x \notin X_j$. A join node $i \in T$ has two children $j$ and $k$, and $X_i = X_j = X_k$.

The leaves of $T$ are also referred to as start nodes.

Bodlaender and Kloks [11] show that every graph of treewidth $k$ has a nice tree-decomposition of width $k$ and size $O(N)$, where the size of a tree-decomposition is the number of nodes in $T$.

The relationships between the different graph classes are visualized in Fig. 2. In general, the more classes a graph belongs to, the more restrictive is its structure, so that it is not surprising that outerplanar graphs allow very simple solutions to the problems discussed in this chapter.

4 Techniques

Before discussing the particular algorithms in this survey, we sketch the two fundamental algorithmic techniques used in these algorithms: graph contraction and time-forward processing.
Fig. 2. The relationships between the different graph classes considered in this survey.

4.1 Graph Contraction

At a very abstract level, graph contraction is simple and elegant: Identify a number of edge-disjoint subgraphs of $G$ so that representing each such subgraph by a graph of smaller size reduces the size of $G$ by a constant factor and preserves the properties of interest. This approach is often taken in parallel graph algorithms, where the contraction procedure is applied recursively $q = \mathcal{O}(\log N)$ times, in order to produce a sequence $G = G_0, G_1, \ldots, G_q$ of graphs with $|G_q| = \mathcal{O}(1)$.

Then the problem is solved in $\mathcal{O}(1)$ time on $G_q$. A solution for graph $G$ is constructed by undoing the contraction steps and at each step deriving a solution for $G_i$ from the given solution for graph $G_{i+1}$. When designing I/O-efficient algorithms, the contraction can usually stop after $q = \mathcal{O}(\log B)$ contraction levels. At that point, the resulting graph is guaranteed to have $\mathcal{O}(|V|/B)$ vertices, so that the algorithm can afford to spend $\mathcal{O}(1)$ I/Os per vertex to solve the problem on the contracted graph. The edges can usually be handled using I/O-efficient data structures.

4.2 Time-Forward Processing

Time-forward processing is a very elegant technique for evaluating directed acyclic graphs. This technique has been proposed in [16] and improved in [5]. Formally, the following problem can be solved using this technique:

Let $G$ be a directed acyclic graph (DAG) whose vertices are numbered so that every edge in $G$ leads from a vertex with lower number to a vertex with higher number. That is, this numbering is a topological numbering of $G$. Let every vertex $v$ of $G$ store a label $\phi(v)$, and let $f$ be a function to be applied in order to compute for every vertex $v$, a new label $\psi(v) = f(\phi(v), \lambda(u_1), \ldots, \lambda(u_k))$, where $u_1, \ldots, u_k$ are the in-neighbors of $v$ in $G$, and $\lambda(u_i)$ is some piece of information "sent" from $u_i$ to $v$ after computing $\psi(u_i)$. The goal is to "evaluate" $G$, i.e., to compute $\psi(v)$, for all vertices $v \in G$. 

6
While time-forward processing does not solve the problem of computing $\psi(v)$ I/O-efficiently in the case where the input data $\phi(v)$ and $\lambda(u_1), \ldots, \lambda(u_k)$ do not fit into internal memory, it provides an elegant way to supply vertex $v$ with this information at the time when $\psi(v)$ is computed. The idea is to process the vertices in $G$ by increasing numbers. This guarantees that all in-neighbors of vertex $v$ are evaluated before $v$. Thus, if these in-neighbors “send” their outputs $\lambda(u_1), \ldots, \lambda(u_k)$ to $v$, $v$ has these inputs and its own label $\phi(v)$ at its disposal to compute $\psi(v)$. After computing $\psi(v)$, $v$ sends its output $\lambda(v)$ “forward in time” to its out-neighbors, which guarantees that these out-neighbors have $\lambda(v)$ at their disposal when it is their turn to be evaluated.

The implementation of this technique due to Arge [5] uses a priority queue $Q$ to realize the “sending” of information along the edges of $G$. When a vertex $v$ wants to send its output $\lambda(v)$ to another vertex $w$, it inserts $\lambda(v)$ into priority queue $Q$ and gives it priority $w$. When vertex $w$ is being evaluated, it removes all entries with priority $w$ from $Q$. As every in-neighbor of $w$ sends its output to $w$ by queuing it with priority $w$, this provides $w$ with the required inputs. Moreover, every vertex removes its inputs from the priority queue when it is evaluated, and all vertices with smaller numbers are evaluated before $w$. Thus, the entries in $Q$ with priority $w$ are in fact those with lowest priority in $Q$ at the time when $w$ is evaluated. Therefore they can be removed using a sequence of DELETEMIN operations. Since this procedure involves $O(|V| + |E|)$ priority queue operations, graph $G$ can be evaluated in $O(sort(|V| + |E|))$ I/Os using an I/O-efficient priority queue (see Chapter ??).

The power of time-forward processing lies in the fact that many problems on undirected graphs can be expressed as evaluation problems of DAGs derived from these graphs.

5 Connectivity Problems

5.1 Connected Components

The I/O-efficient connectivity algorithm of [16] uses ideas from the PRAM algorithm of Chin et al. [17] for this problem. First the graph contraction technique from Section 4.1 is applied in order to compute a sequence $G = G_0, \ldots, G_q$ of graphs whose vertex sets have geometrically decreasing sizes and so that the vertex set of graph $G_q$ fits into main memory. The latter implies that the connected components of $G_q$ can be computed using a simple semi-external connectivity algorithm as outlined below. Given the connected components of $G_q$, the connected components of $G$ are computed by undoing the contraction steps used to construct graphs $G_1, \ldots, G_q$ one by one and in each step computing the connected components of $G_i$ from those of $G_{i+1}$. The details of the algorithm are as follows:

In order to compute graph $G_{i+1}$ from graph $G_i$ during the contraction phase, every vertex in $G_i = (V_i, E_i)$ selects its incident edge leading to its neighbor with smallest number. The selected edges form a forest $F_i$ each of whose trees
contains at least two vertices. Every tree in \( F_i \) is then contracted into a single vertex, which produces a new graph \( G_{i+1} = (V_{i+1}, E_{i+1}) \) with at most half as many vertices as \( G_i \). In particular, for \( 0 \leq i \leq q \), \( |V_i| \leq \frac{|V|}{2^i} \). Choosing \( q = \log_2(|V|/M) \), this implies that \( |V_i| \leq M \), i.e., the vertex set of \( G_q \) fits into main memory. Hence, the connected components of \( G_q \) can be computed using the following simple algorithm:

- Load the vertices of \( G_q \) into main memory and label each of them as being in a separate connected component. Now scan the edge set of \( G_q \) and merge connected components whenever the endpoints of an edge are found to be in different connected components. The computation of this algorithm is carried out in main memory, so that computing the connected components of \( G_q \) takes \( O(\text{scan}(|V_q| + |E_q|)) \) I/0s.

- To construct the connected components of graph \( G_i \) from those of graph \( G_{i+1} \) when undoing the contraction steps, all that is required is to replace each vertex \( v \) of \( G_i \) with the tree in \( F_i \) it represents and assign \( v \)'s component label to all vertices in this tree.

In [16] it is shown that the construction of \( G_{i+1} \) from \( G_i \) as well as computing the connected components of \( G_i \) from those of \( G_{i+1} \) takes \( O(\text{sort}(|E_i|)) \) I/0s. Hence, the whole connectivity algorithm takes \( \sum_{i=0}^{\log_2(|V|/M)} O(\text{sort}(|E_i|)) \) I/0s. Since the graphs we consider are sparse, \( |E_i| = O(|V_i|) = O(|V|/2^i) \), so that \( \sum_{i=0}^{\log_2(|V|/M)} O(\text{sort}(|E_i|)) = O(\text{sort}(|V|)) \). That is, the contraction-based connectivity algorithm computes the connected components of a sparse graph in \( O(\text{sort}(|V|)) \) I/0s.

### 5.2 Minimum Spanning Tree

The algorithm outlined in Section 5.1 can be modified so that it computes an MST (a minimum spanning forest if the graph is disconnected). Instead of selecting for every vertex, the edge connecting it to its neighbor with smallest number, the MST-algorithm chooses the edge of minimum weight incident to each vertex. The weight of an edge in \( G_i \) is the minimum weight of its corresponding edges in \( G \). When an edge in \( G_i \) is selected, its corresponding minimum weight edge in \( G \) is added to the MST. For details and a correctness proof see [16, 17]. Clearly, these modifications do not increase the I/O-complexity of the connectivity algorithm. Hence, the algorithm takes \( O(\text{sort}(|V|)) \) I/0s on sparse graphs.

### 5.3 Biconnected Components

Tarjan and Vishkin [34] present an elegant parallel algorithm to compute the biconnected components of a graph \( G \) by computing the connected components of an auxiliary graph \( H \). Given a spanning tree \( T \) of \( G \), every non-tree edge \((v, w)\) of \( G \) (i.e., \((v, w) \notin E(G) \setminus E(T)\)) defines a fundamental cycle, which consists of the path from \( v \) to \( w \) in \( T \) and edge \((v, w)\) itself. The auxiliary graph \( H \) contains one vertex per edge of \( G \). Two vertices in \( H \) are adjacent if the corresponding edges appear consecutively on a fundamental cycle in \( G \). Using this definition of \( H \), it is
5.4 Ear Decomposition

An ear decomposition $\mathcal{E} = (P_0, P_1, \ldots, P_k)$ of a graph $G$ is an ordered partition of $G$ into edge-disjoint simple paths $P_i$ with endpoints $s_i$ and $t_i$. Ear $P_0$ is an edge. For $1 \leq i \leq k$, ear $P_i$ shares its two endpoints $s_i$ and $t_i$, but none of its internal vertices, with the union $P_0 \cup \cdots \cup P_{i-1}$ of all previous ears. An ear $P_i$ is open if $s_i \neq t_i$. Ear decomposition $\mathcal{E}$ is open if all its ears are open.

For a graph to have an ear decomposition, it has to be 2-edge connected. That is, there cannot be an edge whose removal disconnects the graph. For the graph to have an open ear decomposition, it has to be 2-vertex connected (biconnected). That is, there cannot be a vertex whose removal disconnects the graph. So let $G$ be a 2-edge connected graph, and let $\mathcal{E}$ be an ear decomposition of $G$. Removing an arbitrary edge from each ear in $\mathcal{E}$ results in a spanning tree of $G$. Conversely, an ear decomposition of $G$ can be obtained from any spanning tree $T$ of $G$ as follows: Consider the fundamental cycles defined by the non-tree edges of $G$ and sort them by their distances in $T$ from the root $r$ of $T$, where the distance of a cycle from $r$ is the minimum distance of its vertices from $r$. Remove from each fundamental cycle all those edges that are already contained in a previous cycle. It can be shown that the resulting sorted set of cycles and paths is an ear decomposition of $G$.

In order to obtain a parallel ear decomposition algorithm, Maon et al. [30] propose the following implementation of the above idea. Their algorithm assigns labels to the edges of $G$ so that two edges are in the same ear if and only if they have the same label. For a non-tree edge $e = (v, w)$, let $\text{LCA}(e)$ be the lowest common ancestor of $v$ and $w$ in $T$, and let $\text{depth}(e)$ be the distance of $\text{LCA}(e)$ from the root of $T$. Then $\text{label}(e) = (\text{depth}(e), e)$. For a tree-edge $e$, label($e$) is the minimum label of all non-tree edges so that edge $e$ is on the fundamental cycles defined by these edges. Now every non-tree edge $e$ defines a cycle or simple path $P_e$ with edge set $\{e' \in G : \text{label}(e') = \text{label}(e)\}$. Maon et al. show that the collection of these cycles and paths, sorted by their labels, is an ear decomposition of $G$. This ear decomposition is not necessarily open, even if $G$ is biconnected; but the computation can be modified to produce an open ear decomposition in this case. See [30] for details.

Given spanning tree $T$, the computation of edge labels as described above involves only standard tree computations such as answering LCA queries or computing the depth of a vertex in $T$. In [16] it is shown that these computations can be carried out in $O(\text{sort}(|V|))$ I/Os. Since we consider sparse graphs, tree $T$ can be computed in $O(\text{sort}(|V|))$ I/Os using for example the minimum spanning tree
algorithm of Section 5.2. Hence, for sparse graphs an (open) ear decomposition can be computed in $O(sort(|V|))$ I/Os.

6 Breadth-First Search and Single Source Shortest Paths

After covering connectivity problems, we now turn to the first two graph searching problems: breadth-first search (BFS) and the single source shortest path (SSSP) problem. Since BFS is the same as the SSSP problem if all edges in the graph have unit weight, and both problems have an $O(sort(|V|))$ I/O lower bound, we restrict our attention to SSSP-algorithms. Even though the details of the SSSP-algorithms for different graph classes differ, their efficiency is based on the fact that the considered graph classes have small separators. In particular, a separator decomposition of a graph in each such class can be obtained I/O-efficiently (see Section 8), and the shortest path algorithms apply dynamic programming to such a decomposition in order to solve the SSSP problem.

6.1 Planar Graphs and Grid Graphs

Every planar graph with $N$ vertices or grid graph embedded into a $\sqrt{N} \times \sqrt{N}$ grid contains a set $S$ of $O(N/B)$ separator vertices whose removal partitions the graph into $O(N/B^2)$ subgraphs of size at most $B^2$ and boundary size at most $B$, where the boundary $\partial G_i$ of a subgraph $G_i$ is the set of separator vertices adjacent to vertices in $G_i$ [20]. The set of separator vertices can be partitioned into maximal subsets so that the vertices in each subset are adjacent to the same set of subgraphs $G_i$. These sets are called the boundary sets of the partition (see Fig. 3). If the graph has bounded degree, which is true for grid graphs and can be ensured for planar graphs using a simple transformation, there exists a partition that, in addition to the above properties, has only $O(N/B^2)$ boundary sets [20].
Given such a partition of $G$ into subgraphs $G_1, \ldots, G_q$, the single source shortest path algorithm first computes a graph $G_R$ with vertex set $S$ so that the distances between two separator vertices in $G$ and $G_R$ are the same. Assuming that $s \in S$, the distances from $s$ to all separator vertices can hence be computed by solving the single source shortest path problem on $G_R$, which can be done $I/O$-efficiently due to the reduced size of the vertex set of $G_R$. Given the distances from $s$ to all separator vertices, the distances from $s$ to all vertices in a graph $G_i$ can be computed as $\text{dist}_G(s, v) = \min \{ \text{dist}_G(s, u) + \text{dist}_{R_i}(u, v) : u \in \partial G_i \}$, where $R_i$ is the subgraph of $G$ induced by the vertices in $V(G_i) \cup \partial G_i$.

The construction of graph $G_R$ from graph $G$ is done as follows: For each graph $R_i$ as defined above, compute the distances in $R_i$ between all pairs of vertices in $\partial G_i$. Then construct a complete graph $R'_i$ with vertex set $\partial G_i$ and assign weight $\text{dist}_{R_i}(v, w)$ to every edge $(v, w) \in R'_i$. Graph $G_R$ is the union of graphs $R'_1, \ldots, R'_q$.

Assuming that $M = \Omega(B^2)$, there is enough room in main memory to store one graph $R_i$ and its compressed version $R'_i$. Hence, graph $G_R$ can be computed from graph $G$ by loading graphs $R_1, \ldots, R_q$ into main memory, one at a time, computing for each graph $R_i$ the compressed version $R'_i$ without incurring any $I/O$s and writing $R'_i$ to disk. As this procedure requires a single scan of the list of graphs $R_1, \ldots, R_q$, and these graphs have a total size of $\mathcal{O}(N)$, graph $G_R$ can be constructed in $\mathcal{O}(\text{scan}(N))$ $I/O$s. Similarly, once the distances from $s$ to all separator vertices are known, the computation of the distances from $s$ to all non-separator vertices can be carried out in another scan of the list of graphs $R_1, \ldots, R_q$ because the computation for the vertices in $R_i$ is local to $R_i$.

From the above discussion it follows that the SSSP problem can be solved in $\mathcal{O}(\text{sort}(N))$ $I/O$s on $G$ if it can be solved in that many $I/O$s on $G_R$. Since $G_R$ has only $\mathcal{O}(N/B)$ vertices and $\mathcal{O}\left(\frac{(N/B)^2 \cdot B^2}{B} = \mathcal{O}(N) \right)$ edges, the SSSP problem on $G_R$ can be solved in $\mathcal{O}(\text{sort}(N))$ $I/O$s using the shortest path algorithm described in Chapter ??.

In order to reduce the I/O-complexity of this step to $\mathcal{O}(\text{sort}(N))$, Arge et al. [7, 9] propose a modified version of Dijkstra's algorithm, which avoids the use of a DecreaseKey operation. This is necessary because the best known external priority queue that supports this operation [25] takes $\mathcal{O}(\text{sort}(N))$ $I/O$s to process a sequence of $N$ priority queue operations, while there are priority queues that do not support this operation, but can process a sequence of $N$ Insert, Delete, and DeleteMin operations in $\mathcal{O}(\text{sort}(N))$ $I/O$s [5, 13].

In addition to a priority queue $Q$ storing the unvisited vertices of $G_R$, the algorithm of Arge et al. maintains a list $L$ of the vertices of $G_R$, each labeled with its tentative distance from $s$. That is, for every vertex stored in $Q$, its label in $L$ is the same as its priority in $Q$. For a vertex not in $Q$, list $L$ stores its final distance from $s$. Initially, all distances, except that of $s$, are $\infty$. Vertex $s$ has distance 0. Now the algorithm repeatedly performs DeleteMin operations on $Q$ to obtain the next vertex to process. For every retrieved vertex $v$, the algorithm loads the adjacency list of $v$ into main memory and updates the distances from $s$ to $v$'s neighbors as necessary. (The adjacency list of $v$ fits into main memory because
every vertex in \( S \) has degree \( O(B) \) in \( G_R \). To see this, observe that each vertex
in \( S \) is on the boundary of \( O(1) \) subgraphs \( G_i \) because graph \( G \) has bounded
degree, and each subgraph has at most \( B \) boundary vertices. In order to update
these distances, the algorithm retrieves the entries corresponding to \( v \)’s neighbors
from \( L \) and compares the current tentative distance of each neighbor \( w \) of \( v \) to
the length of the path from \( s \) to \( w \) through \( v \). If the path through \( v \) is shorter,
the distance from \( s \) to \( w \) is updated in \( L \) and \( Q \). Since the old tentative distance
from \( s \) to \( w \) is known, the update on \( Q \) can be performed by deleting the old copy
of \( w \) and inserting a new copy with the updated distance as priority. That is,
the required \textsc{DecreaseKey} operation is replaced by a \textsc{Delete} and an \textsc{Insert}
operation.

Since graph \( G_R \) has \( O(N/B) \) vertices and \( O(N) \) edges, retrieving all adjacency lists takes \( O(N/B + \text{scan}(N)) = O(\text{scan}(N)) \) I/Os. For the same reason,
the algorithm performs only \( O(N) \) priority operations on \( Q \), which takes
\( O(\text{sort}(N)) \) I/Os. It remains to analyze the number of I/Os spent on accessing
list \( L \). If the vertices in \( L \) are not arranged carefully, the algorithm may spend one
I/O per access to a vertex in \( L \), \( O(N) \) I/Os in total. In order to reduce this I/O-
bound to \( O(N/B) \), Arge et al. use the fact that there are only \( O(N/B^2) \) boundary sets, each of size \( O(B) \). If the vertices in each boundary set are stored consecu-
tively in \( L \), the bound on the size of each boundary set implies that the vertices
in the set can be accessed in \( O(1) \) I/Os. Moreover, every boundary set is accessed
only \( O(B) \) times, once per vertex on the boundary of the subgraphs defining this
boundary set. Since there are \( O(N/B^2) \) boundary sets, the total number of I/Os
spent on loading boundary sets from \( L \) is hence \( O(B \cdot N/B^2) = O(N/B) \).

The algorithm described above computes only the distances from \( s \) to all
vertices in \( G \). However, it is easy to augment the algorithm so that it computes
shortest paths in \( O(\text{sort}(N)) \) I/Os using an additional post-processing step.

### 6.2 Graphs of Bounded Treewidth and Outerplanar Graphs

The SSSP algorithm for planar graphs and grid graphs computes shortest paths
in three steps: First it encodes the distances between separator vertices in a com-
pressed graph. Then it computes the distances from the source to all separator
vertices in this compressed graph. And finally it computes the distances from
the source to all non-separator vertices using the distance information computed
for the separator vertices on the boundary of the subgraph \( G_i \) containing each
such vertex. The shortest path algorithm for outerplanar graphs and graphs of
bounded treewidth [28, 37] applies this approach iteratively, using the fact that
a tree-decomposition of the graph provides a hierarchical decomposition of the
graph using separators of constant size.

In particular, assume that the given tree-decomposition \( \mathcal{D} = (T, \mathcal{X}) \) of \( G \) is
nice in the sense defined in Section 3 and that \( s \in X_v \), for all \( v \in T \).

\footnote{Explicitly adding \( s \) to all sets \( X_v \) to ensure the latter assumption increases the width
of the decomposition by at most one.}
subtree of \( T \) rooted at some node \( v \in T \) represents a subgraph \( G(v) \) of \( G \), which shares only the vertices in \( X_v \) with the rest of \( G \).

The first phase of the algorithm processes \( T \) from the leaves towards the root and computes for every node \( v \in T \) and every pair of vertices \( x, y \in X_v \), the distance from \( x \) to \( y \) in \( G(v) \). Since \( G(r) = G \), for the root \( r \) of \( T \), this produces the distances in \( G \) between all vertices in \( X_r \). In particular, the distances from \( s \) to all other vertices in \( X_r \) are known at the end of the first phase. The second phase processes tree \( T \) from the root towards the leaves to compute for every node \( v \in T \), the distances from \( s \) to all vertices in \( X_v \).

During the first phase, the computation at a node \( v \) uses only the weights of the edges between vertices in \( X_v \) and distance information computed for the vertices stored at \( v \)’s children. During the second phase, the computation at node \( v \) uses the distance information computed for the vertices in \( X_v \) during the first phase of the algorithm and the distances from \( s \) to all vertices in \( X_{p(v)} \), where \( p(v) \) denotes \( v \)’s parent in \( T \). Since the computation at every node involves only a constant amount of information, it can be carried out in main memory. All that is required is passing distance information from children to parents in the first phase of the algorithm and from parents to children in the second phase. This can be done in \( O(\text{sort}(N)) \) I/Os using time-forward processing because tree \( T \) has size \( O(N) \), and \( O(1) \) information is sent along every edge.

To provide at least some insight into the computation carried out at the nodes of \( T \), we discuss the first phase of the algorithm. For a leaf \( v \), \( G(v) \) is the graph induced by the vertices in \( X_v \). In particular, \( |G(v)| = O(1) \), and the distances in \( G(v) \) between all vertices in \( X_v \) can be computed in main memory. For a root \( v \) with child \( w \), \( G(v) = G(w) \) and \( X_v \subseteq X_w \), so that the distance information for the vertices in \( X_v \) has already been computed at node \( w \) and can easily be copied to node \( v \). For an introduce node \( v \) with child \( w \), \( X_v = X_w \cup \{x\} \). A shortest path in \( G(v) \) between two vertices in \( X_v \) consists of shortest paths in \( G(w) \) between vertices in \( X_w \) and edges between \( x \) and vertices in \( X_w \). Hence, the distances between vertices in \( X_v \) are the same in \( G(v) \) and in a complete graph \( G'(v) \) with vertex set \( X_v \) whose edges have the following weights:

If \( y, z \in X_w \), then edge \((y, z)\) has weight \( \text{dist}_{G(w)}(y, z) \). Otherwise assume w.l.o.g. that \( y = x \). Then the weight of edge \((x, z)\) is the same in \( G'(v) \) as in \( G \). The distances in \( G(v) \) between all vertices in \( X_v \) can now be computed by solving all pairs shortest paths on \( G'(v) \). This can be done in main memory because \( G'(v) = O(1) \). For a join node \( u \) with children \( v \) and \( w \), a similar graph \( G'(u) \) of constant size is computed, which captures the lengths of the shortest paths between all vertices in \( X_u = X_v = X_w \) that stay either completely in \( G(v) \) or completely in \( G(w) \). The distances in \( G(u) \) between all vertices in \( X_u \) are again computed in main memory by solving all pairs shortest paths on \( G'(u) \).

The second phase of the algorithm proceeds in a similar fashion, using the fact that a shortest path from \( s \) to a vertex \( x \) in \( X_v \) either stays completely inside \( G(v) \), in which case the shortest path information between \( s \) and \( x \) computed in the first phase is correct, or it consists of a shortest path from \( s \) to a vertex \( y \) in \( X_{p(v)} \) followed by a shortest path from \( y \) to \( x \) in \( G(p(v)) \).
Fig. 4. (a) A planar graph $G$ with its faces colored according to their levels. Level-0 faces are white. Level-1 faces are light grey. Level-2 faces are dark grey. (b) The corresponding partition of the graph into outerplanar subgraphs $H_0$ (solid), $H_1$ (dotted), and $H_2$ (dashed).

Since outerplanar graphs have treewidth 2, the algorithm sketched above can be used to solve SSSP on outerplanar graphs in $O(sort(N))$ I/Os. Alternatively, one can derive a separator decomposition of an outerplanar graph directly from an outerplanar embedding of the graph. This separator decomposition has a somewhat simpler structure, which allows the algorithm to be simplified and the constants to be improved. However, the overall structure of the algorithm remains the same. The interested reader may refer to [37] for details.

7 Depth-First Search

The algorithms of the previous section use small separators to compute shortest paths I/O-efficiently. In this section we discuss algorithms that construct DFS-trees of planar graphs, grid graphs, and outerplanar graphs. These algorithms exploit the geometric structure of these graphs to carry out their task in an I/O-efficient manner. Since graphs of bounded treewidth do not exhibit such a geometric structure in general, the techniques used in these algorithms fail on graphs of bounded treewidth, so that the problem of computing a DFS-tree of a graph of bounded treewidth I/O-efficiently is open.

7.1 Planar Graphs

For the sake of simplicity, assume that the given planar graph $G$ is biconnected. If this is not the case, a DFS-tree of $G$ can be obtained in $O(sort(N))$ I/Os by identifying the biconnected components of $G$ using the biconnectivity algorithm from Section 5.3 and merging appropriate DFS-trees computed separately for each of these biconnected components.

In order to perform DFS in an embedded biconnected planar graph $G$, the algorithm of [8], which follows ideas from [22], uses the following approach: First the faces of $G$ are partitioned into layers around a central face that has the source of the DFS on its boundary (see Fig. 4a). The partition of the faces of $G$ into layers induces a partition of $G$ into outerplanar graphs of a particularly simple
structure, so that DFS-trees of these graphs can be computed I/O-efficiently (see Fig. 4b). A DFS-tree of $G$ is then obtained by merging appropriate DFS-trees of these layer graphs.

Formally, the layers are defined as follows: The first layer consists of a single face $r$ that has the source $s$ of the DFS on its boundary. Given the first $i$ layers, the $(i+1)$-st layer contains all faces that share a vertex with a face in the $i$-th layer and are not contained in layers 0 through $i$. Such a partition of the faces of $G$ into layers can be obtained using BFS in the face-on-vertex graph $G_F$ of $G$. This graph contains all vertices of $G$ as well as one vertex $f^*$ per face $f$ of $G$. There is an edge $(v, f^*)$ in $G_F$ if and only if vertex $v$ is on the boundary of face $f$ in $G$. Given the BFS-distances of all vertices in $G_F$ from $r^*$, a face $f$ is in the $i$-th layer if the BFS-distance of vertex $f^*$ from $r^*$ is $2i$ (see Fig. 5a).

The computed layer partition of the faces of $G$ defines the following partition of $G$ into outerplanar subgraphs $H_0, H_1, \ldots, H_k$ (see Fig. 4b). Let $G_i$ be the subgraph of $G$ induced by all faces at levels 0 through $i$, and let $E_i = E(G_i) \setminus E(G_{i-1})$. Then edge set $E_i$ can be partitioned into two sets $E_i'$ and $E_i''$. An edge is in $E_i'$ if none of its endpoints is on the boundary of a level-$(i-1)$ face. Otherwise it is in $E_i''$. Now graph $H_i$ is defined as the subgraph of $G$ induced by the edges in $E_i'$. The edges in $E_i''$ are the attachment edges of $H_i$, since they form the connection between $H_i$ and $G_{i-1}$.

In order to compute a DFS-tree of $G$, the algorithm now computes DFS-trees $T_0, \ldots, T_k$ of graphs $H_0 = G_0, \ldots, G_k = G$, where for all $1 \leq i \leq k$, tree $T_i$ is obtained by augmenting tree $T_{i-1}$ appropriately. To facilitate this incremental construction, the algorithm maintains the invariant that the vertices on each boundary cycle of $G_i$ appear on a root-to-leaf path in $T_i$. We call this the boundary cycle invariant. Using this invariant, a DFS-tree $T_{i+1}$ of $G_{i+1}$ can be obtained as follows: For every connected component $H'$ of $H_{i+1}$, find the attachment edge in $E_i''$ that connects a vertex $v$ in $H'$ to a vertex $u$ of maximal depth in $T_i$. Compute a DFS-tree of $H'$ rooted at $v$ and join it to $T_i$ using edge $(u, v)$. Since $H'$ is enclosed by exactly one boundary cycle of $G_i$, the choice of edge $(u, v)$ and the boundary cycle invariant guarantee that for any other attachment edge $(x, y)$ of $H'$, its endpoint $x \in G_i$ is an ancestor of $u$ in $T_i$. 

---

Fig. 5. (a) The face-on-vertex graph $G_F$ shown in bold. (b) Spanning tree $T_1$ and layer graph $H_2$ are shown in bold. Attachment edges $(u, v)$ are thin solid edges. The vertices in $T_1$ are labeled with their DFS-depths. (c) The final DFS tree of $G$. 

---

15
The endpoint $y \in H'$ is a descendant of $w$, so that $(x, y)$ is a back edge. Hence, $T_{i+1}$ is a DFS-tree of $G_{i+1}$.

We have to discuss how to compute a DFS-tree of $H'$ in a manner that maintains the boundary cycle invariant. This is where the simple structure of $H_{i+1}$ comes into play. In particular, it can be shown that graph $H_{i+1}$ is a "forest of cycles". That is, every non-trivial biconnected component of $H_{i+1}$ is a simple cycle. Now consider the biconnected components of $H'$. These components form a tree of biconnected components, which can be rooted at a component containing vertex $v$. For every biconnected component $B$, except the root component, its parent cutpoint is the cutpoint shared by $B$ and its parent component in the tree of biconnected components. The parent cutpoint of the root component is defined to be $v$, even though $v$ is not necessarily a cutpoint of $H'$. A DFS-tree of $H'$ is now obtained by removing from every non-trivial biconnected component of $H'$ one of the two edges incident to its parent cutpoint. Since the boundary cycles of $G_{i+1}$ are cycles in $H_{i+1}$, and each such cycle is a non-trivial biconnected component of $H_{i+1}$, this maintains the boundary cycle invariant for $T_{i+1}$.

The computation of a DFS-tree for $H'$ involves only computing the biconnected components of $H'$ and the removal of appropriate edges from the non-trivial biconnected components. The former can be done in $O(\text{sort}(|H'|))$ I/Os using the biconnectivity algorithm from Section 5.3. The latter can be done in a constant number of sorting and scanning steps. Hence, computing DFS-trees for all connected components of $H_{i+1}$ takes $O(\text{sort}(|H_{i+1}|))$ I/Os. Finding attachment edges $(u, v)$ for all connected components of $H_{i+1}$ requires sorting and scanning the vertex set of $H_i$ and the set $E_{i+1}$ of attachment edges, which takes $O(\text{sort}(|H_i| + |H_{i+1}|))$ I/Os. Summing these complexities over all layers of $G$, the whole DFS-algorithm takes $O(\text{sort}(|V|))$ I/Os because graphs $H_0, \ldots, H_k$ are disjoint.

7.2 Grid Graphs

Finding an I/O-efficient algorithm for DFS in grid graphs is still an open problem, even though the standard internal memory DFS-algorithm performs only $O(N/\sqrt{B})$ I/Os if carried out carefully. In particular, Meyer [32] made the following observation: Consider a grid of size $\sqrt{N}$ by $\sqrt{N}$ divided into subgrids of size $\sqrt{B}$ by $\sqrt{B}$. Then a DFS-tree of $G$ can be computed using the standard internal memory algorithm: Start from the block (subgrid) containing the source vertex, perform DFS until the algorithm visits a vertex that is not in the block, load the block containing this vertex into main memory, and continue in this fashion until the DFS-tree is complete. A block may be loaded several times during a run of the algorithm, each time to compute a different part of the DFS tree that lies within this block. However, the DFS tree enters a block through a boundary vertex, and leaves it through a boundary vertex. Every vertex is visited $O(1)$ times by the DFS-algorithm. Since a block has $O(\sqrt{B})$ boundary vertices, this implies that every block is loaded $O(\sqrt{B})$ times, so that the algorithm takes $O(N/B \cdot \sqrt{B}) = O(N/\sqrt{B})$ I/Os.
7.3 Outerplanar Graphs

The DFS-algorithm for outerplanar graphs is hardly worth being called an algorithm. It merely exploits the fact that a DFS-tree of the graph is already encoded in an outerplanar embedding of the graph. In particular, if the graph is biconnected, the path obtained by removing an edge from the boundary cycle of the outer face of $G$ is a DFS-tree of $G$. This path can be extracted in $O(\text{scan}(N))$ I/Os from an appropriate representation of the outerplanar embedding.

If the graph is not biconnected, the basic idea of the algorithm is to walk along the outer boundary of $G$ and keep track of the number of times a vertex has been visited so far. If a vertex is visited for the first time, its predecessor along the outer boundary is made its parent in the DFS-tree. Otherwise nothing is done for the vertex. This strategy can easily be realized in $O(\text{scan}(N))$ I/Os using a stack, again assuming an appropriate representation of the outerplanar embedding. The interested reader may refer to [37] for details.

8 Graph Partitions

In this section we review algorithms for partitioning sparse graphs into smaller subgraphs. In particular, given a graph $G$ belonging to a class $\mathcal{C}$ of sparse graphs and an integer $h > 0$, the goal is to compute a small set $S$ of vertices whose removal partitions $G$ into $k = O(N/h)$ subgraphs $G_1, \ldots, G_k$ of size at most $h$.

We refer to the pair $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ as an $h$-partition of $G$. The vertices in $S$ are referred to as separator vertices. Set $S$ as a whole is referred to as the separator that induces partition $\mathcal{P}$. Finally, let $\sigma(\mathcal{C}, N, h) = \max_{\mathcal{G} \in \mathcal{C}} \min_{S} \{|S| : S \text{ separator of } G\}$, where the maximum is taken over all $N$-vertex graphs in class $\mathcal{C}$. We call partition $\mathcal{P}$ optimal if $|S| = O(\sigma(\mathcal{C}, G, h))$.

Among the algorithms presented so far, only the shortest path algorithm for planar graphs requires an optimal $h$-partition (for $h = B^2$), while the algorithm for outerplanar graphs and graphs of bounded treewidth only relies on the fact that a recursive partition of the graph using separators of constant size is encoded in the tree-decomposition of the graph. Nevertheless graph partitions have been applied to design efficient sequential and parallel algorithms for a wide range of problems and may prove useful for designing I/O-efficient algorithms for these problems as well.

8.1 Planar Graphs

A number of researchers have put considerable effort into designing algorithms that compute optimal partitions of planar graphs I/O-efficiently. The first step towards a solution has been made in [24], where it is shown that an optimal $\frac{1}{4}N$-partition of a planar graph $G$ can be computed in $O(\text{sort}(N))$ I/Os using an I/O-efficient version of Lipton and Tarjan's algorithm [26]. Unfortunately the I/O-bound holds only if a BFS-tree and a planar embedding of $G$ are
Fig. 6. $O(sort(N))$ I/O reductions between fundamental problems on planar graphs. An arrow indicates that the pointing problem can be solved in $O(sort(N))$ I/Os if the problem the arrow points to can be solved in that many I/Os.

given. Arge et al. [7] present an I/O-efficient variant of an algorithm due to Goodrich [21]. Given a planar graph $G$ and an integer $h > 0$, the algorithm takes $O(sort(N))$ I/Os to compute a separator of size $O(sort(N) + N/\sqrt{h})$ that induces an $h$-partition of $G$. Again, a BFS-tree and a planar embedding of $G$ are required. In addition, the amount of available memory is required to be $M > B^{2+\alpha}$, for some $\alpha > 0$. Together with the shortest path algorithm from Section 6.1, the DFS-algorithm from Section 7.1, and the observation that BFS can be solved using an SSSP-algorithm, the latter result leads to circular dependencies between different fundamental problems on embedded planar graphs as shown in Fig. 6. In particular, if any of the three problems in the cycle—i.e., computing optimal $h$-partitions, BFS, or SSSP—can be solved in $O(sort(N))$ I/Os, all problems in Fig. 6 can be solved in this number of I/Os on embedded planar graphs.

The break-through has been achieved in [29], where it is shown that an optimal $h$-partition of a planar graph $G$ can be computed in $O(sort(N))$ I/Os without using a solution to any of the other problems, provided that the amount of available main memory is $O(h \log^2 B)$. Since the shortest path algorithm from Section 6.1 requires an optimal $B^2$-partition, this implies that BFS, DFS, and SSSP can be solved in $O(sort(N))$ I/Os, provided that $M = \Omega(B^2 \log^2 B)$.

In [37], it is shown that using these results and an I/O-efficient version of a recent result of [4], optimal partitions of planar graphs with costs and weights on their vertices and optimal edge separators of weighted planar graphs can be obtained in $O(sort(N))$ I/Os. We sketch here the result of [29] on computing unweighted partitions because it is the key to the efficiency of the other algorithms, and refer the reader to [37] for the details of the more general separator algorithm.

The algorithm of [29] obtains an optimal $h$-partition of $G$ by careful application of the graph contraction technique, combined with a linear-time internal memory algorithm for this problem. In particular, it first constructs a hierarchy of planar graphs $G = H_0, \ldots, H_r$ whose sizes are geometrically decreasing and so that $|H_r| = O(N/B)$. The latter implies that applying the internal memory algorithm to $H_r$ in order to compute an optimal partition of $H_r$ takes $O(N/B) = O(sort(N))$ I/Os. Given the partition of $H_r$, the algorithm now iter-
ates over graphs $H_{r-1}, \ldots, H_0$, in each iteration deriving a partition of $H_i$ from
the partition of $H_{i+1}$ computed in the previous iteration. The construction of a
separator $S_i$ for $H_i$ starts with the set $S'_i$ of vertices in $G_i$ that were contracted
into the vertices in $S_{i+1}$ during the construction of $H_{i+1}$ from $H_i$. Set $S'_i$ induces
a preliminary partition of $H_i$, which is then refined by adding new separator
vertices to $S'_i$. The resulting set is $S_i$.

The efficiency of the procedure and the quality of its output depend heavily
on the properties of the computed graph hierarchy. In [29] it is shown that a graph
hierarchy $G = H_0, \ldots, H_r$ with the following properties can be constructed in
$O(\text{sort}(N))$ I/Os:

(i) For all $0 \leq i \leq r$, graph $H_i$ is planar,
(ii) For all $1 \leq i \leq r$, every vertex in $H_i$ represents a constant number of
vertices in $H_{i-1}$ and at most $2^i$ vertices in $G$, and
(iii) For all $0 \leq i \leq r$, $|H_i| = O(N/2^i)$.

Choosing $r = \log B$, Property (iii) implies that $|H_r| = O(N/B)$, as required
by the algorithm. Properties (ii) and (iii) can be combined with an appropriate
choice of the size of the subgraphs in the partitions of graphs $H_r, \ldots, H_1$ in order
to guarantee that the final partition of $G$ is optimal. In particular, the algorithm
makes sure that for $1 \leq i \leq r$, separator $S_i$ induces an $(h\log^2 B)$-partition
of $H_i$, and only the refinement step computing $S = S_0$ from $S_0'$ has the goal
of producing an $h$-partition of $G = H_0$. Aleksandrov and Djidjev [3] show that
for any graph of size $N$ and any $h' > 0$, their algorithm computes a separator
of size $O(N/\sqrt{h'})$ that induces an $h'$-partition of the graph. Hence, since we
use the algorithm of [3] to compute $S_r$ and to derive separator $S_i$ from $S'_i$,
for $0 \leq i < r$, $|S_i| = O(|H_r|/\sqrt{\log B})$, and for $i > 0$, the construction
of $S_i$ from $S'_i$ adds $O(|H_i|/\sqrt{\log B})$ separator vertices to $S'_i$. By Properties
(ii) and (iii), this implies that $|S'_i| = O(N/\sqrt{h})$. In order to obtain an $h$-partition
of $G$, the algorithm of [3] adds another $O(N/\sqrt{h})$ separator vertices to $S'_0$, so
that $S$ induces an optimal $h$-partition of $G$.

The efficiency of the algorithm also follows from Properties (ii) and (iii). We
have already argued that for $r = \log B$, $|H_r| = O(N/B)$, so that the linear-time
separator algorithm takes $O(\text{scan}(N))$ I/Os to compute the initial $(h\log^2 B)$-
partition of $H_r$. Property (ii) implies that separator $S'_r$ induces a $(ch\log^2 B)$-
partition of $H_r$, for some constant $c \geq 1$. Under the assumption that $M \geq
ch \log^2 B$, this implies that every connected component of $H_r - S'_r$ fits into main
memory. Hence, the algorithm of [29] computes the connected components of
$H_r - S'_r$, loads each of them into main memory and applies the internal memory
algorithm of [3] to partition it into subgraphs of size at most $h\log^2 B$ (or $h$, if
$i = 0$).

Since $S_r$ can be computed in $O(\text{scan}(N))$ I/Os and the only external memory
computation required to derive $S_i$ from $S'_i$ is computing the connected compo-
nents of $H_i - S'_i$, the whole algorithm takes $O(\text{scan}(N)) + \sum_{i=0}^{r-1} O(\text{sort}(|H_i|)) =
\sum_{i=0}^{r-1} O\left(\text{sort}\left(N/2^i\right)\right) = O(\text{sort}(N))$ I/Os.
In order to use the computed partition in the SSSP algorithm from Section 6.1, it has to satisfy a few more stringent properties than optimality in the above sense. In particular, it has to be guaranteed that each of the $O(N/h)$ subgraphs in the partition is adjacent to at most $\sqrt{h}$ separator vertices and that there are only $O(N/h)$ boundary sets as defined in Section 6.1. In [37], it is shown that these properties can be ensured using a post-processing that takes $O(sort(N))$ I/Os and increases the size of the computed separator by at most a constant factor. The construction is based on ideas from [20].

8.2 Grid Graphs

For a grid graph $G$, the geometric information associated with its vertices makes it very easy to compute an $h$-partition of $G$. In particular, every vertex stores its coordinates $(i, j)$ in the grid. Then the separator $S$ is chosen to contain all vertices in rows and columns $\sqrt{h}, 2\sqrt{h}, 3\sqrt{h}, \ldots$. Separator $S$ has size $O(N/\sqrt{h})$ and partitions $G$ into subgraphs of size at most $h$. That is, the computed partition is optimal. Since every vertex in a grid graph can be connected only to its eight neighboring grid vertices, each subgraph in the computed partition is adjacent to at most $4\sqrt{h}$ separator vertices. The number of boundary sets in the partition is $O(N/h)$. Hence, this partition can be used in the shortest path algorithm from Section 6.1.

8.3 Graphs of Bounded Treewidth and Outerplanar Graphs

As in the case of the single-source shortest path problem, the computation of $h$-partitions for graphs of bounded treewidth and outerplanar graphs is similar. Again, for outerplanar graphs, a simplified algorithm producing a slightly better partition is presented in [37]; but the basic approach is the same as in the general algorithm for graphs of bounded treewidth, which we describe next.

As the shortest path algorithm, the algorithm starts by computing a nice tree-decomposition $D = (T, \mathcal{X})$ of $G$ using either the tree-decomposition algorithm from Section 9.2 or the outerplanar embedding algorithm from Section 9.3. The goal of the algorithm is to use the tree-decomposition to compute an optimal $h$-partition of $G$, i.e., a partition of $G$ into subgraphs of size at most $h$ using a separator of size $O(kN/h)$, where $k$ is the treewidth of $G$. To do this, tree $T$ is processed from the leaves towards the root, starting with an empty separator $S$.

For every node $i \in T$, a weight $\omega(i)$ is computed, which equals the size of the connected component of $G(i) - S$ containing the vertices in $X_i$. At a leaf, $\omega(i)$ is computed as follows: If $\omega(G(i)) > h/2$, all vertices in $X_i$ are added to $S$, and $\omega(i) = 0$. Otherwise $\omega(i) = \omega(G(i))$. At a forget node, $\omega(i) = \omega(j)$. At an introduce node, let $\omega'(i) = \omega(j) + 1$. If $\omega'(i) > h/2$, the vertices in $X_i$ are added to $S$, and $\omega(i) = 0$. Otherwise $\omega(i) = \omega'(i)$. Finally, at a join node, let $\omega'(i) = \omega(j) + \omega(k)$. Then $\omega(i)$ is computed using the same rules as for an introduce node. It is easily verified that the computed vertex set $S$ induces an $h$-partition of $G$. To see that $S$ has size $O(kN/h)$, observe that every group of $k + 1$ vertices added to $S$ can be charged to a set of at least $h/2$ vertices.
in $G$ and that these groups of charged vertices are disjoint. Hence, at most
$(k + 1)N/(h/2) = \mathcal{O}(kN/h)$ vertices are added to $S$.

9 Gathering Structural Information

Having small separators is a structural property that all the graph classes we
consider have in common. In this section we review I/O-efficient algorithms
to gather more specific information about each class. In particular, we sketch
algorithms for computing outerplanar and planar embeddings of outerplanar and
planar graphs and tree-decompositions of graphs of bounded treewidth.
These algorithms are essential, at least from a theoretical point of view, as all of the
algorithms presented in previous sections, except the separator algorithm for
planar graphs, require an embedding or tree-decomposition to be given as part of
the input.

9.1 Planarity Testing and Planar Embedding

In order to test whether a given graph is planar, the algorithm of [29] exploits
the fact that the separator algorithm from Section 8.1 does not require a planar
embedding to be given as part of the input. In fact, the algorithm can be applied
even without knowing whether $G$ is planar. The strategy of the planar embedding
algorithm is to use the separator algorithm and try to compute an optimal $B^2$-
partition of $G$ whose subgraphs $G_1, \ldots, G_q$ have boundary size at most $B$. If the
separator algorithm fails to produce the desired partition in $O(sort(N))$ I/Os,
the planar embedding algorithm terminates and reports that $G$ is not planar.
Otherwise the algorithm first tests whether each of the graphs $G_1, \ldots, G_q$ is
planar. If one of these graphs is non-planar, graph $G$ cannot be planar. If graphs
$G_1, \ldots, G_q$ are planar, each graph $G_i$ is replaced with a constraint graph $C_i$ of
size $O(B)$. These constraint graphs have the property that graph $G$ is planar if
and only if the approximate graph $A$ obtained by replacing each subgraph $G_i$
with its constraint graph $C_i$ is planar. If $A$ is planar, a planar embedding of $G$
is obtained from a planar embedding of $A$ by locally replacing the embedding of
each constraint graph $C_i$ with a consistent planar embedding of $G_i$.

This approach leads to an I/O-efficient algorithm using the following observations: (1) Graphs $G_1, \ldots, G_q$ have size at most $B^2$, and graphs $C_1, \ldots, C_q$ have
size $O(B)$ each. Thus, the test of each graph $G_i$ for planarity and the construction
of the constraint graph $C_i$ from $G_i$ can be carried out in main memory,
provided that $M \geq B^2$, which has to be true already in order to apply the separator algorithm. (2) Graph $A$ has size $O(N/B)$ because it is constructed from
$O(N/B^2)$ constraint graphs of size $O(B)$, so that a linear time planarity testing
and planar embedding algorithm (e.g., [12]) takes $O(sort(N))$ I/Os to test
whether $A$ is planar and if so, produce a planar embedding of $A$. The construction
of consistent planar embeddings of graphs $G_1, \ldots, G_q$ from the embeddings
of graphs $C_1, \ldots, C_q$ can again be carried out in main memory.
This seemingly simple approach involves a few technicalities that are discussed in detail in [37]. At the core of the algorithm is the construction of the constraint graph \( C_i \) of a graph \( G_i \). This construction is based on a careful analysis of the structure of graph \( G_i \) and beyond the scope of this survey. We refer the reader to [37] for details. However, we sketch the main ideas.

The construction is based on the fact that triconnected planar graphs are rigid in the sense that they have only two different planar embeddings, which can be obtained from each other by “flipping” the whole graph. The construction of constraint graph \( C_i \) partitions graph \( G_i \) into its connected components, each connected component into its biconnected components, and each biconnected component into its triconnected components. The connected components can be handled separately, as they do not interact with each other. The constraint graph of a connected component is constructed bottom-up from constraint graphs of its biconnected components, which in turn are constructed from constraint graphs of their triconnected components.

The constraint graph of a triconnected component is triconnected, and its embedding contains all faces of the triconnected component so that other parts of \( G \) may be embedded in these faces. The rest of the triconnected component is compressed as far as possible while preserving triconnectivity and planarity.

The constraint graph of a biconnected component is constructed from the constraint graphs of its triconnected components by analyzing the amount of interaction of these triconnected components with the rest of \( G \). Depending on these interactions, the constraint graph of each triconnected component is either (1) preserved in the constraint graph of the biconnected component, (2) grouped with the constraint graphs of a number of other triconnected components, or (3) does not appear in the constraint graph of the biconnected component at all because it has no influence on the embedding of any part of \( G \) that is not in this biconnected component. In the second case, the group of constraint graphs is replaced with a new constraint graph of constant size. The constraint graph of a connected component is constructed in a similar manner from the constraint graphs of its biconnected components.

### 9.2 Computing a Tree-Decomposition

The algorithm of [28] for computing a tree-decomposition of width \( k \) for a graph \( G \) of treewidth \( k \) follows the framework of the linear-time algorithm for this problem by Bodlaender and Kloks [10,11]. The algorithm can also be used to test whether a given graph \( G \) has treewidth at most \( k \), as long as \( k \) is constant.

The details of the algorithm are rather involved, so that we sketch only the main ideas here. In [10] it is shown that for every \( k > 0 \), there exist two constants \( c_1, c_2 > 0 \) so that for every graph \( G \) of treewidth \( k \) one of the following is true: (1) Every maximal matching of \( G \) contains at least \( c_1 N \) edges. (2) \( G \) contains a set \( X \) of at least \( c_2 N \) vertices so that a tree-decomposition of width \( k \) for \( G \) can be obtained by attaching one node of size at most \( k + 1 \) per vertex in \( X \) to a tree-decomposition of width \( k \) for the graph \( G - X \). In the latter case, the algorithm of [28] recursively computes a tree-decomposition of \( G - X \) and then attaches
the additional nodes in $O(sort(N))$ I/Os. In the former case, it computes a tree-decomposition $D'$ of width at most $k$ for the graph $G'$ obtained by contracting the edges in a maximal matching. The maximal matching can be computed in $O(sort(N))$ I/Os using an algorithm of [37]. By replacing every vertex in $G'$ that corresponds to an edge in the matching by the two endpoints of this edge, one immediately obtains a tree-decomposition of width at most $2k + 1$ for $G$. In order to obtain a tree-decomposition of width at most $k$, an I/O-efficient version of the algorithm of [11] is used, which computes the desired tree-decomposition in $O(sort(N))$ I/Os.

This algorithm starts by transforming the given tree-decomposition $D'$ of width at most $2k + 1$ into a nice tree-decomposition of width at most $2k + 1$ and size $O(N)$. Then dynamic programming is applied to this tree-decomposition, from the leaves towards the root, in order to compute an implicit representation of a tree-decomposition of width $k$ for graph $G$. In a second pass of time-forward processing from the leaves towards the root, the tree-decomposition is extracted from this implicit representation. The details of this algorithm are complex and beyond the scope of this survey.

Given that each recursive step of the algorithm takes $O(sort(N))$ I/Os, the I/O-complexity of the whole algorithm is $O(sort(N))$, since graphs $G'$ and $G - X$ passed to recursive invocations of the algorithm contain only a constant fraction of the vertices and edges of $G$.

### 9.3 Outerplanarity Testing and Outerplanar Embedding

In order to compute an outerplanar embedding of an outerplanar graph, the algorithm of [37]\(^3\) exploits the following two observations: (i) An outerplanar embedding of an outerplanar graph can be obtained from outerplanar embeddings of the biconnected components of the graph, by making sure that no biconnected component is embedded in an interior face of another biconnected component. (ii) The boundary of the outer face of an outerplanar embedding of a biconnected outerplanar graph $G$ is the only cycle in $G$ that contains all vertices of $G$.

Observation (i) can be used to reduce the problem of computing an outerplanar embedding of $G$ to that of computing outerplanar embeddings of its biconnected components. These components can be computed in $O(sort(N))$ I/Os using the biconnectivity algorithm from Section 5.3, so that any outerplanar graph can be embedded in $O(sort(N))$ I/Os if a biconnected outerplanar graph can be embedded in this number of I/Os. To achieve the latter, Observation (ii) is exploited.

The algorithm for embedding a biconnected outerplanar graph computes the boundary cycle $C$ of $G$, numbers the vertices along $C$, and uses this numbering of the vertices to derive the final embedding of $G$. Assume for now that cycle $C$ is given. Then the desired numbering of the vertices of $G$ can be obtained by

---

\(^3\) The algorithm for this problem originally presented in [27] is more complicated, so that we present the simplified version from [37] here.
removing an arbitrary edge from cycle $C$ and applying the Euler-tour technique
and list-ranking to the resulting path in order to compute the distances of all
vertices in this path from one of the endpoints of the removed edge. Given the
numbering of the vertices along $C$, the edges incident to every vertex in $G$ can be
ordered clockwise around this vertex using the observation that these edges ap-
pear in the same order clockwise around $v$ as their endpoints clockwise along $C$.
Hence, an outerplanar embedding of $G$ can be computed in $O(\text{sort}(N))$ I/Os if
cycle $C$ can be identified in $O(\text{sort}(N))$ I/Os.

To compute cycle $C$, the algorithm exploits the fact that this cycle is unique.
In particular, an algorithm that computes any cycle containing all vertices in $G$
must produce cycle $C$. A cycle containing all vertices of $G$ can be computed
as follows from an open ear-decomposition of $G$. Let $P_0, \ldots, P_q$ be the ears in
the ear-decomposition. Then remove every ear $P_i$, $i > 0$, that consists of a
single edge. The resulting graph is a biconnected subgraph of $G$ that contains
all vertices of $G$. For every remaining ear $P_i$, $i > 0$, remove edge $(a, b)$ from $G$,
where $a$ and $b$ are the endpoints of $P_i$. This procedure can easily be carried out
using a constant number of sorts and scans, so that it takes $O(\text{sort}(N))$ I/Os.
The following argument shows that the resulting graph is the desired cycle $C$.

Let $P'_0, \ldots, P'_r$ be the set of ears remaining after removing all ears $P_i$, $i > 0,$
consisting of a single edge. Then the above construction is equivalent to the
following construction of graphs $G_1, \ldots, G_r$. Graph $G_1$ is the union of ears $P'_0$
and $P'_1$. Since ear $P'_0$ consists of a single edge, and the endpoints of ear $P'_1$ are
in $P'_0$, graph $G_1$ is a cycle. In order to construct graph $G_i$ from cycle $G_{i-1}$,
remove the edge connecting the endpoints of ear $P'_i$ from $G_{i-1}$ and add $P'_i$ to
the resulting graph. If the endpoints of ear $P'_i$ are adjacent in $G_{i-1}$, graph $G_i$ is
again a cycle. But if the endpoints of $P_i$ are not adjacent in $G_{i-1}$, it follows from
the biconnectivity of $G_{i-1}$ and the fact that $P_i$ contains at least one internal
vertex that graph $G_i$ contains a subgraph that is homeomorphic to $K_{2,3}$, so that
$G_i$ and hence $G$ cannot be outerplanar. Applying this argument inductively, we
obtain that $G_r = C$.

The algorithm sketched above can easily be augmented to test whether a
given graph is outerplanar. For details, we refer the reader to [37].

10 Conclusions and Open Problems

The algorithms for BFS, DFS, and SSSP on special classes of sparse graphs are
a major step towards solving these problems on sparse graphs in general. In
particular, the results on planar graphs have answered the long standing ques-
tion whether these graphs allow $O(\text{sort}(N))$ I/O solutions for these problems.
However, all these algorithms are complex because they are based on computing
separators. Thus, the presented results pose a new challenge, namely that of
finding simpler, practical algorithms for these problems.

Since the currently best known separator algorithm for planar graphs requires
that $M = \Omega(B^2 \log^3 B)$, the algorithms for BFS, DFS, and SSSP on planar
graphs inherit this constraint. It seems that this memory requirement of
separator algorithm (and hence of the other algorithms as well) can be removed
or at least reduced if the semi-external single source shortest path problem can
be solved in $\mathcal{O}(\text{sort}(|E|))$ I/Os on arbitrary graphs. ("Semi-external" means that
the vertices of the graph fit into main memory, but the edges do not.)

For graphs of bounded treewidth, the main open problem is finding an I/O-
efficient DFS-algorithm. Practicality is not an issue here, as the chances to obtain
practical algorithms for these graphs are minimal, as soon as the algorithms rely
on a tree-decomposition.

For grid graphs, the presented shortest path algorithm uses a partition of
the graph into a number of cells that depends on the size of the grid. This
may be non-optimal if the graph is an extremely sparse subgraph of the grid.
An interesting question here is whether it is possible to exploit the geometric
information provided by the grid to obtain a partition of the same quality as the
one obtained by the separator algorithm for planar graphs, but with much less
effort, i.e., in a way that leads to a practical algorithm.

References

graph algorithms. In Proceedings of the 6th European Symposium on Algorithms,
volume 1461 of Lecture Notes in Computer Science, pages 332–343. Springer-
Verlag, 1998.
algorithms for contour-line extraction and planar graph blocking. In Proceedings
of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 117–126,
1998.
3. L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded
1996.
In Proceedings of the 4th Workshop on Algorithm Engineering and Experiments,
2002.
5. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In Pro-
cedings of the 4th Workshop on Algorithms and Data Structures, volume 955 of
6. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In
Proceedings of the 5th International Symposium on Algorithms and Computation,
volume 1004 of Lecture Notes in Computer Science, pages 82–91. Springer-Verlag,
1995.
7. 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 Al-
gorithm Theory, volume 1851 of Lecture Notes in Computer Science, pages 433–447.
search. In Proceedings of the 7th International Workshop on Algorithms and Data
Structures, volume 2125 of Lecture Notes in Computer Science, pages 471–482.
9. L. Arge, L. Toma, and J.S. Vitter. I/O-efficient algorithms for problems on grid-
based terrains. In *Proceedings of the 2nd Workshop on Algorithm Engineering and
10. H.L. Bodlaender. A linear time algorithm for finding tree-decompositions of small
11. H.L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the path-
12. J. Boyer and W. Myrvold. Stop minding your P’s and Q’s: A simplified $O(n)$ planar
13. G.S. Brodal and J. Katajainen. Worst-case efficient external-memory priority
14. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata,
A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th
On external memory graph traversal. In *Proceedings of the 11th Annual ACM-
17. F.Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graphs
19. E. Feierstein and A. Marchetti-Spaccamela. Memory paging for connectivity and
path problems in graphs. In *Proceedings of the International Symposium on Algo-
rithms and Computation*, volume 762 of *Lecture Notes in Computer Science*, pages
20. G.N. Frederickson. Fast algorithms for shortest paths in planar graphs, with ap-
24. 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
25. V. Kumar and E.J. Schwabe. Improved algorithms and data structures for solving
graph problems in external memory. In *Proceedings of the 8th IEEE Symposium
27. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs.
Verlag, December 1999.

26