Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Algorithms 53 (2004) 186–206
Journal of Algorithms
www.elsevier.com/locate/jalgor

On external-memory MST, SSSP and multi-way planar graph separation ☆

Lars Arge [a,*,1], Gerth Stølting Brodal [b,2], Laura Toma [a,1]

[a] *Department of Computer Science, Duke University, USA*
[b] *BRICS (Basic Research in Computer Science, Center of Danish National Research Foundation), University of Aarhus, Denmark*

Received 15 November 2002

Abstract

Recently external memory graph problems have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems still remain open.

The results in this paper fall in two main classes. First we develop an improved algorithm for the problem of computing a minimum spanning tree (MST) of a general undirected graph. Second we show that on planar undirected graphs the problems of computing a multi-way graph separation and single source shortest paths (SSSP) can be reduced I/O-efficiently to planar breadth-first search (BFS). Since BFS can be trivially reduced to SSSP by assigning all edges weight one, it follows that in external memory planar BFS, SSSP, and multi-way separation are equivalent. That is, if any of these problems can be solved I/O-efficiently, then all of them can be solved I/O-efficiently in the same bound. Our planar graph results have subsequently been used to obtain I/O-efficient algorithms for all fundamental problems on planar undirected graphs.

☆ An extended abstract version of this paper was presented at the Seventh Scandinavian Workshop on Algorithm Theory (SWAT 2000) [Lecture Notes in Comput. Sci., vol 1851, 2000, pp. 433–447].

* Corresponding author.
E-mail addresses: large@cs.duke.edu (L. Arge), gerth@brics.dk (G.S. Brodal), laura@cs.duke.edu (L. Toma).

1 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.

2 Supported in part by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the Carlsberg foundation under contract number ANS-0257/20.

doi:10.1016/j.jalgor.2004.04.001

1. Introduction

Recently external memory graph problems have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. One example of a massive graph is AT&T's 20TB phone-call data graph [12]. Other examples of massive graphs arise in Geographic Information Systems (GIS). For instance, GIS terrains are often represented using planar graphs and many common GIS problems can be formulated as standard graph problems (Arc/Info [4], the most commonly used GIS package, contains functions that correspond to computing depth-first, breadth-first, and minimum spanning trees, as well as shortest paths and connected components). When working with such massive graphs the I/O-communication, and not the internal memory computation time, is often the bottleneck. Designing efficient external memory algorithms for such problems can thus lead to considerable runtime improvements (see, e.g., [8]).

Even though a large number of I/O-efficient graph algorithms have been developed in recent years, a number of important problems still remain open, especially for sparse graphs. In this paper we develop an improved I/O-efficient algorithm for the problem of computing a minimum spanning tree of a general undirected graph. We also show that on embedded planar undirected graphs, multi-way planar graph separation and single source shortest path can be reduced to breadth-first search. Our planar graph results have subsequently been used to obtain I/O-efficient algorithms for all fundamental problems on planar undirected graphs.

1.1. Problem statement

Given a weighted graph $G = (V, E)$ the minimum spanning tree (MST) problem is the problem of finding a spanning tree for G of minimum weight. The single-source shortest path (SSSP) problem is the problem of finding the shortest paths from a given source vertex in G to all other vertices in G (the length of a path is the sum of the weights of the edges on the path). For an undirected graph $G = (V, E)$ and a function $f : N \to N$, an $f(V)$-*separator*[3] of G is a subset S of V of size $f(V)$ such that the removal of S disconnects G into two subgraphs G_1 and G_2, each of size at most $2V/3$. The vertices in S are called *separator vertices*. Lipton and Tarjan [25] proved that any planar graph (a graph that can be embedded in the plane so that no two edges cross except at the endpoints) has an $O(\sqrt{V})$-separator. For any parameter R, this result can be used recursively to partition a planar graph into $\Theta(V/R)$ subgraphs G_i with $O(R)$ vertices each using $O(V/\sqrt{R})$ separator vertices, such that there is no edge between a vertex in G_i and a vertex in G_j for $i \neq j$. We call a partition of a graph G into $O(V/R)$ subgraphs with $O(R)$ vertices each using a set of separator vertices S a *multi-way planar graph separation* of G. Graph separation is often used in the design of divide-and-conquer graph algorithms.

[3] For convenience we will use the name of a set to denote both the actual set and its cardinality.

Throughout this paper we assume that G is given as a list of edges ordered by vertex. We also assume without loss of generality that G is connected and that no two edges have the same weight. In our algorithms for planar graphs we assume that a planar embedding is given. When a BFS tree T of G is given, we assume that T is represented implicitly by storing with each vertex in G its parent in T, and that each edge of G is marked as being either a tree or a non-tree edge.

1.2. Previous results on I/O-efficient graph algorithms

We work in the standard two-level I/O model with one (logical) disk [3,22]. The model defines the following parameters:

$$N = V + E,$$

$$M = \text{number of vertices/edges that can fit into internal memory},$$

$$B = \text{number of vertices/edges per disk block},$$

where $M < N$ and $1 \leqslant B \leqslant M^{1/(2+\varepsilon)}$, for some $\varepsilon > 0$.[4] An *Input/Output* (or simply *I/O*) involves reading (or writing) a block of consecutive elements from (to) disk into (from) internal memory. The measure of performance of an algorithm in this model 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 *scanning* or *linear* bound), and the number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ [3] (the *sorting* bound). For all realistic values of N, B and M, $\text{scan}(N) < \text{sort}(N) \ll N$. In practice the difference between an algorithm doing N I/Os and one doing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os can be very significant [8].

I/O-efficient graph algorithms have been considered by a number of authors [1,2,5,6,11, 13,17,20,24,26,29–31,35]. Table 1 reviews the best known algorithms for basic graph theoretic problems on general undirected graphs. For directed graphs the best known algorithms for breadth-first search (BFS) and depth-first search (DFS) use $O((V + \text{scan}(E)) \cdot \log V + \text{sort}(E))$ I/Os [11]. In general, $\Omega(\min\{V, \text{sort}(V)\})$ (which is $\Omega(\text{sort})$ in all practical cases) is a lower bound on the number of I/Os needed to solve most graph problems [5,13,30]. Note that no $O(\text{sort}(E))$ (deterministic) algorithm is known for *any* of the fundamental graph problems, and that, except for the very recent undirected BFS algorithm of [29], the best known algorithms for DFS, BFS and SSSP require $\Omega(V)$ I/Os. MST and connected components (CC) can be solved in $O(\text{sort}(E))$ I/Os with randomized algorithms [1,13].

Improved algorithms have been developed for several special classes of (sparse) graphs. See [34] for a complete reference. For trees, $O(\text{sort}(N))$ algorithms are known for BFS and DFS numbering, Euler tour computation, expression tree evaluation, topological sorting, as well as several other problems [11,13]. For grid graphs $O(\text{sort}(N))$ algorithms are known for BFS and SSSP, and an $O(\text{scan}(N))$ algorithm for CC [8]. Outerplanar graphs and bounded treewidth graphs are considered in [26,27]. For planar graphs, $O(\text{sort}(N))$ algorithms are known for CC and MST [13].

---

[4] Often it is only assumed that $B \leqslant M/2$ but sometimes, as in this paper, the realistic assumption that the main memory is capable of holding B^2 (or as here, $B^{2+\varepsilon}$) elements is made.

Table 1
Best known upper bounds for basic problems on general undirected graphs: depth-first-search (DFS), breadth-first-search (BFS), connected-components (CC), minimal spanning tree (MST), and single-source-shortest-paths (SSSP)

Problem	Best know algorithm	
DFS	$O\big(V + \frac{V}{M}\frac{E}{B}\big)$	[13]
	$O((V + \text{scan}(E)) \cdot \log V + \text{sort}(E))$	[24]
BFS	$O\big(V + \frac{E}{V} \cdot \text{sort}(V)\big)$	[30]
	$O\Big(\sqrt{\frac{V \cdot E}{B}} + \text{sort}(E)\Big)$	[29]
CC	$O\big(\text{sort}(E) \cdot \log\log \frac{VB}{E}\big)$	[30]
MST	$O\big(\text{sort}(E) \cdot \log \frac{V}{M}\big)$	[13]
	$O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$	[24]
SSSP	$O\big(V + \frac{E}{B} \cdot \log \frac{V}{B}\big)$	[24]

1.3. Our results

In the first part of this paper, Section 2, we give an $O(\text{sort}(E) \cdot \log\log(VB/E)) = O(\text{sort}(E) \cdot \log\log B)$ algorithm for the MST problem on general undirected weighted graphs, improving the previous bound of $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$ [24]. The algorithm uses the same general idea as the CC algorithm of Munagala and Ranade [30] and consists of two phases: first an edge-contraction algorithm is used to reduce the number of vertices to $O(E/B)$, and then an $O(V + \text{sort}(E))$ MST algorithm is used on the reduced graph. The new contraction algorithm uses ideas similar to the ones used in [9,15,30], as well as a simplified algorithm for the basic contraction step used in previous MST algorithms [9,13–15,24,30,33]. The new $O(V + \text{sort}(E))$ MST algorithm is a modified version of Prim's algorithm. It remains a challenging open problem to develop an $O(\text{sort}(E))$ MST algorithm.

Given that even very basic graph problems seem hard to externalize, it is natural to try to reduce the problems to one another. In the second part of this paper, we show how to reduce two problems on planar graph to planar BFS. Initial work in this direction was done by Hutchinson et al. [20] who showed how to reduce the problem of finding an $O(\sqrt{N})$-separator of a planar graph to planar BFS in $O(\text{sort}(N))$ I/Os. In Section 3, we give an $O(\text{sort}(N))$ reduction from the multi-way planar graph separation problem to planar BFS. More specifically, we show how, given a BFS-tree, G can be partitioned into $O(N/R)$ subgraphs of size $O(R)$ using $O(\text{sort}(N) + N/\sqrt{R})$ separator vertices in $O(\text{sort}(N))$ I/Os. This result improves on the straightforward I/O-bound of $O(\log(N/R) \cdot \text{sort}(N))$ I/Os obtained by recursive use of the result from [20]. Our reduction uses a divide-and-conquer approach and uses ideas from [19]. In Section 4, we then show how the multi-way separation of a planar graph can be used to solve the SSSP problem in $O(\text{sort}(N))$ I/Os. The algorithm is a generalization of an I/O-efficient SSSP algorithm for grid graphs [8] and uses ideas similar to the ones utilized by Frederickson [18].

Since BFS can be trivially reduced to SSSP by assigning all edges weight one, our results show that in external memory planar BFS, SSSP and multi-way separation are essentially equivalent; if any of the problems can be solved I/O-efficiently, then all of them can be solved I/O-efficiently. In a recent paper, Arge et al. [7] also showed that planar DFS can be reduced to planar BFS in $O(\text{sort}(N))$ I/Os. Recently, Maheshwari and Zeh [28] developed an $O(\text{sort}(N))$ algorithm for computing a multi-way separation of a planar graph (without assuming that a BFS tree is given) provided that $M \geqslant R \cdot \log^2 B$.[5] In combination, the results in [7,28] and this paper show that all fundamental problems on planar undirected graphs can be solved in $O(\text{sort}(N))$ I/Os.

2. General graph minimum spanning tree

In this section we describe our MST algorithm for general undirected weighted graphs. The basic idea is to use an $O(\text{sort}(E) \cdot \log\log(VB/E))$ algorithm to reduce the number of vertices to $O(E/B)$, and then use an $O(V + \text{sort}(E))$ MST algorithm on the resulting graph. The overall I/O complexity will thus be $O(\text{sort}(E) \cdot \log\log(VB/E) + E/B + \text{sort}(E)) = O(\text{sort}(E) \cdot \log\log(VB/E))$. In Section 2.1 we first describe the $O(V + \text{sort}(E))$ MST algorithm, and in Section 2.2 we then describe the reduction algorithm. Our result is summarized in the following theorem.

Theorem 1. *A minimum spanning tree of an undirected weighted graph $G = (V, E)$ can be found in $O(\text{sort}(E) \cdot \log\log(VB/E))$ I/Os.*

2.1. An $O(V + \text{sort}(E))$ MST algorithm

Our algorithm is a modified version of Prim's internal memory algorithm [16]. The idea of Prim's algorithm is to grow the MST iteratively from a source vertex while maintaining a priority queue on the vertices not included in the MST so far; the priority of a vertex is the weight of the minimum weight edge connecting it to the current MST. The algorithm repeatedly extracts the minimum priority vertex v, adds it to the MST, and updates the priority of the vertices u adjacent to v. Specifically, the weight w of edge (v, u) is compared with the priority of vertex u in the priority queue, and a priority update is performed if w is smaller than the current priority. Prim's algorithm cannot be implemented efficiently in external memory, mainly because the current priority of a given vertex cannot in general be obtained without doing an I/O. A direct implementation would thus lead to an $\Omega(E)$ I/O algorithm. Previously known algorithms [13,24] rely instead on edge-contraction methods [9,14,15].

Our modification of Prim's algorithm consists of storing *edges* in the priority queue instead of vertices. During the algorithm the priority queue contains (at least) all edges connecting vertices in the current MST with vertices not in the tree; it can also contain

[5] Even though their algorithm computes a multi-way separation without the use of an efficient BFS algorithm, we believe our algorithm is of independent interest since the two algorithms use fundamentally different approaches.

edges between two vertices in the MST. The queue is initialized to contain all edges incident to the source vertex. The algorithm works as follows: The minimum weight edge (u, v) is repeatedly extracted from the priority queue. If v is already in the MST the edge is discarded. Otherwise v is included in the MST and all edges incident to v, except (v, u), are inserted in the priority queue. The correctness of the algorithm follows directly from the correctness of Prim's algorithm. The key to its I/O-efficiency is that we have a simple way of determining if v is already included in the MST—if both u and v are in the MST when processing an edge $e = (u, v)$, the edge e must have been inserted in the priority queue twice. Thus we can determine if v is already included in the MST by simply checking if the next minimal weight edge in the priority queue is identical to e. For this to work we use our assumption that any two edges have distinct weights.

Our algorithm performs at least one I/O for each vertex in order to read its adjacent vertices (traverse its adjacency list) when it is included in the MST. Thus in total, processing all vertices and edges takes $O(V + E/B)$ I/Os. The algorithm also performs $O(E)$ operations on the priority queue. Using an external priority queue [6,10] supporting $O(N)$ operations in $O(\text{sort}(N))$ I/Os we obtain:

Lemma 1. *The minimum spanning tree of an undirected weighted graph $G = (V, E)$ can be computed in $O(V + \text{sort}(E))$ I/Os.*

2.2. MST vertex reduction algorithm

Our MST vertex reduction algorithm is obtained using ideas from the connected-component algorithm of Munagala and Ranade [30] (which is based on edge-contraction), as well as the notion of "blocking values". The standard MST algorithm based on edge-contraction proceeds in $O(\log V)$ phases (or *Boruvka steps* [9]). In each phase the minimum weight edge adjacent to each vertex v is selected and output as part of the MST. Then the vertices connected by the selected edges are contracted to supervertices. Proofs of the correctness of this approach can, e.g., be found in [9,13–15,24,30]. Let the size of a supervertex be the number of vertices it contains from the original graph. After the ith phase the size of every supervertex is at least 2^i and thus after $O(\log(V/M))$ phases the contracted graph fits in memory. In Section 2.2.1 below we discuss how one contraction phase (a Boruvka step) can be performed in $O(\text{sort}(E))$ I/Os, resulting in an $O(\text{sort}(E) \cdot \log(V/M))$ algorithm [13]. Kumar and Schwabe [24] obtained an improved $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$ algorithm by utilizing that after $\Theta(\log B)$ phases, when the number of vertices has decreased to $O(V/B)$, a contraction phase can be performed more efficiently.

As discussed, we will use a contraction algorithm to reduce the number of supervertices to E/B (and then utilize the algorithms presented in the previous section). To do so we need to do $\Theta(\log(VB/E))$ contraction phases, and in Section 2.2.2 we show how to perform these phases in $O(\text{sort}(E) \cdot \log\log(VB/E))$ I/Os (as opposed to $O(\text{sort}(E) \cdot \log(VB/E)))$ by dividing the $\Theta(\log(VB/E))$ phases into $\Theta(\log\log(VB/E))$ *superphases* requiring $O(\text{sort}(E))$ I/Os each. This way we obtain the following Lemma, which together with Lemma 1 proves Theorem 1.

Lemma 2. *The minimum spanning tree of an undirected weighted graph $G = (V, E)$ can be reduced to the same problem on a graph with $O(E/B)$ vertices and $O(E)$ edges in $O(\text{sort}(E) \cdot \log\log(VB/E))$ I/Os.*

2.2.1. $O(\text{sort}(E))$ vertex contraction algorithm

Recall that in one contraction step (or Boruvka step [9]) on a graph $G = (V, E)$ the lightest edge incident to each vertex is selected and contracted to create supervertices. The relevant lightest incident edges can easily be collected in $O(E/B)$ I/Os in a simple scan of the edge-list representation of G, and several $O(\text{sort}(E))$ algorithms for performing the actual contraction have been developed [13,24,30]. In this section we describe an algorithm that we believe is simpler than previously developed algorithms.

For each vertex v let $C(v)$ denote the lightest vertex adjacent to v (i.e., the edge $(v, C(v))$ is the lightest edge incident to v). Let G' be the graph obtained by selecting the edge $(v, C(v))$ for each vertex v. Our goal is to contract each connected component in G', that is, to identify a unique representative vertex in each component and replace each edge (v, u) in G with the edge (v_r, u_r), where v_r and v_u are the unique representatives of the components containing v and u, respectively.

To compute the unique representatives we consider the directed graph G'_d obtained by directing the edge $(v, C(v))$ in G' from $C(v)$ to v. Note that each vertex in G'_d has indegree one. The connected components of G'_d (G') consist of "pseudo trees" [21]; in each component two edges $e_1 = (u, v)$ and $e_2 = (v, u)$ must have the same (minimal) weight and form a cycle (e_1 and e_2 correspond to the same edge e in G and e is the minimal weight edge incident to both u and v). If one of these two edges is removed the resulting component must form a tree (since the number of edges is one less than the number of vertices) with root v or u. In this tree each vertex is on a directed path from the root to a leaf (since each vertex has indegree one) and the weights of edges along a directed path are strictly increasing. Refer to Fig. 1.

The structure of the connected components of G'_d allows us to compute unique representatives I/O-efficiently; we can easily construct G'_d and identify all the cycles in $O(\text{sort}(N))$ I/Os using a few sorting and scanning steps. After removing one of the edges in each cycle we are left with a collection of trees where we have identified the roots. In each tree we choose the root as the unique representative and distribute this information to the rest of the nodes in the tree using an idea similar to "time forward processing"

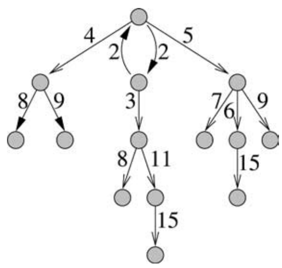

Fig. 1. Pseudo tree. Tree contains one cycle (the "root") consisting of the minimal weight edge, and the weight of edges along any root-leaf path is increasing.

[6,13]:[6] Let L be the list of edges in G'_d sorted in increasing order of weight, where we after each edge (u, v) store a copy of all other edges incident to v. The list L contains each edge in G'_d twice and can easily be constructed in $O(\text{sort}(V))$ I/Os in a few sorting and scanning steps. We process edges in order from L while maintaining a priority queue *PQ*; this queue is conceptually used to send the representative of already processed vertices "forward in time" to their immediate successors. More precisely, we maintain the following invariant: *PQ* contains each vertex v for which the incoming edge (u, v) has not yet been processed but where the incoming edge of u has. The priority of vertex v in *PQ* is equal to the weight of the incoming edge (u, v) and v is augmented with information about the unique representative of u. Note that when v is inserted in *PQ* we have identified its unique representative and we can therefore also output this information to an output list. We initialize *PQ* to contain each vertex that is an immediate successor of a root vertex in G'_d; if v is an immediate successor of root u, that is, the edge $e = (u, v)$ exists, we insert v with priority equal to the weight of e and labeled with u. We can easily perform this initialization in $O(\text{sort}(V))$ I/Os by scanning through the edges of G'_d while inserting the relevant vertices. To process the next edge $e = (v, w)$ from L we first extract the minimal priority vertex from *PQ*. Since the edges are processed in increasing order of weight, v must already have been processed and w inserted in *PQ* with priority equal to the weight of e. Since all edges with weight smaller than e have already been processed, w must be the vertex extracted from *PQ*. Thus we have obtained the unique representative for w. We can reestablish the invariant by inserting an element for each successor of w in *PQ* with the appropriate priority (obtained from the information associated with e in L) and marked with the unique representative of v (and w). Since we perform $O(V)$ operations on *PQ*, we in total use $O(\text{sort}(V))$ I/Os to perform the priority queue operations [6,10]. Thus we have identified the unique representatives in $O(E/B + \text{sort}(V))$ I/Os.

To finish the contraction we need to replace each edge (v, u) in G with the edge (v_r, u_r) between the representatives v_r of v and v_u of u. Given a list E of edges (v, u) and a list R of representatives (v, v_r), we can easily do so in $O(\text{sort}(E))$ I/Os as follows; we first sort E and R by the first component. Then we scan the two lists simultaneously, replacing each edge (v, u) in E with (v_r, u). Next we replace the second component of E with their representatives in a similarly way by sorting E by second component and scanning E and R simultaneously. Finally, we remove duplicate edges in a simple sorting and scanning step.

Lemma 3. *Given an undirected weighted graph $G = (V, E)$, the lightest edge incident to each vertex can be contracted in $O(\text{sort}(E))$ I/Os.*

2.2.2. Superphase algorithm

In this section we show how to perform $\Theta(\log(VB/E))$ contraction phases on a graph $G = (V, E)$, reducing the number of vertices to E/B, in $O(\log\log(VB/E) \cdot \text{sort}(E))$ I/Os. We do so by performing the $\Theta(\log(VB/E))$ phases in $\Theta(\log\log(VB/E))$ *superphases* requiring $O(\text{sort}(E))$ I/Os each.

[6] The distribution can be done using standard I/O-efficient tree algorithms [11,13] but since the weights along any root-leaf path are increasing we can use the somewhat simpler algorithm described here.

Let $N_i = 2^{(3/2)^i}$, i.e., $N_{i+1} = N_i\sqrt{N_i}$. Superphase i consists of $\lceil \log\sqrt{N_i}\,\rceil$ phases. We will maintain the invariant that before superphase i the number of supervertices is at most $2V/N_i$. Let $G_i = (V_i, E_i)$ be the graph just prior to superphase i. To be efficient, the phases in superphase i only work on a subset E_i' of the edges in E_i. For each vertex v, E_i' contains the $\lceil\sqrt{N_i}\,\rceil$ lightest edges incident to v. Heavier edges $e = (v, u)$ incident to v are only included in E_i' if e is among the $\lceil\sqrt{N_i}\,\rceil$ lightest edges incident to u. Furthermore, we define the *blocking value* of v to be the weight of the $(\lceil\sqrt{N_i}\,\rceil + 1)$th lightest edge incident to v. Note that $E_i' \leqslant 2V_i\lceil\sqrt{N_i}\,\rceil$ and since $V_i \leqslant 2V/N_i$ we have $E_i' \leqslant V_i\lceil\sqrt{N_i}\,\rceil < 2V/\sqrt{N_i}$. The set E_i' and blocking values can be computed in $O(\text{sort}(E_i))$ I/Os using a few scanning and sorting steps.

We now perform $\lceil\log\sqrt{N_i}\,\rceil$ contraction phases on the reduced graph $G_i' = (V_i, E_i')$. A phase is performed as in the basic vertex reduction algorithm: For each vertex v we consider the incident edge $e = (v, u)$ in E_i' with minimum weight. If the weight of e is smaller than the blocking value of v, we select e for contraction. If the weight of e is larger than the blocking value no edge is selected for v (since there might be a lighter edge adjacent to v in $E_i - E_i'$). The selected edges are contracted in $O(\text{sort}(E_i'))$ I/Os using the algorithms described in Section 2.2.1 (Lemma 3); after the contraction we define the blocking value of a supervertex to be the minimum of the blocking values of the contracted vertices. By induction the remaining edges of E_i' contain all edges of E_i adjacent to supervertex v with weight smaller than the blocking value of v. Thus the algorithm correctly contract only edges that actually belong to the MST of G.

That the number of supervertices after the $\lceil\log\sqrt{N_i}\,\rceil$ phases is at most $2V/N_{i+1}$ can be seen as follows: if in superphase i the blocking value of a supervertex v prevents us from selecting an edge for v, then v must be the contraction of at least $\sqrt{N_i}$ vertices from V_i. This follows from the fact that the blocking value of v corresponds to the blocking value of some vertex u in V_i and v must contain the $\lceil\sqrt{N_i}\,\rceil$ vertices adjacent to u in E_i'. If no blocking value prevents us from selecting an edges for v, then after $\lceil\log\sqrt{N_i}\,\rceil$ phases v must have size at least $2^{\log\sqrt{N_i}} = \sqrt{N_i}$. Thus using $O(\text{sort}(E_i) + \text{sort}(E_i') \cdot \log\sqrt{N_i}\,) = O(\text{sort}(E) + \text{sort}(V/\sqrt{N_i}\,) \cdot \log\sqrt{N_i}\,) = O(\text{sort}(E))$ I/Os the number of vertices is reduced by a factor of at least $\sqrt{N_i}$, i.e., the number of vertices after the $\lceil\log\sqrt{N_i}\,\rceil$ contraction phases is at most $V_i/\sqrt{N_i} \leqslant 2V/(N_i\sqrt{N_i}) = 2V/N_{i+1}$.

After performing the $\lceil\log\sqrt{N_i}\,\rceil$ contraction phases on G' (that is, considering only the sampled edges E_i'), we need to reincorporate the edges $(E_i - E_i')$ in order to finish superphase i; the edge (v, u) should be replaced with (v_s, u_s), where v_s and u_s are the supervertices containing v and u, respectively. To do so we maintain during the contraction phases a list C containing for each vertex v the current supervertex containing v, that is, C contains pairs of the form (v, v_s). After each phase (the algorithm in Section 2.2.1) we obtain a similar list L of vertex-representative pairs and need to update C accordingly. We can easily do so in $O(\text{sort}(V_i))$ I/Os by sorting C by second component and L by first component, and then scan the two lists simultaneously while replacing each pair $(v, v_s), (v_s, v_{s'})$ with $(v, v_{s'})$. In total we use $O(\log\sqrt{N_i} \cdot \text{sort}(V_i)) = O(\log\sqrt{N_i} \cdot \text{sort}(V/N_i)) = O(\text{sort}(V))$ I/Os to maintain L. Given L we can reincorporate (update) the edges in $(E_i - E_i')$ in $O(\text{sort}(E))$ in the same way we updated the edges after a single contraction in Section 2.2.1.

Finally, to reduce the number of vertices in G to $O(E/B)$ it is sufficient to perform i superphases such that $V/N_i \leqslant E/B$. Thus it is sufficient to perform $O(\log\log(VB/E))$ superphases using $O(\text{sort}(E))$ I/Os each, for a total of $O(\text{sort}(E) \cdot \log\log(VB/E))$ I/Os. This proves Lemma 2 and concludes the description of our MST algorithm.

3. Multi-way planar graph separation

Given a BFS tree T of a planar graph $G = (V, E)$, Hutchinson et al. [20] showed how to compute an $O(\sqrt{N})$-separator for G in $O(\text{sort}(N))$ I/Os. Their algorithm closely follows the algorithm by Lipton and Tarjan [25]: the BFS tree T has the property that no edge in G crosses two or more levels, and hence every level in T is a separator in G. The "middle" level ℓ_1 in T (the level containing the vertex with number $N/2$ in the BFS numbering) has the property that the total number of vertices on levels above ℓ_1, as well as on levels below ℓ_1, is less than $N/2$. The problem is that ℓ_1 may contain more than $\sqrt{N}$ vertices. However, there exists a level ℓ_0 above ℓ_1 and a level ℓ_2 below ℓ_1 with $\sqrt{N}$ vertices each, such that $\ell_2 - \ell_0 \leqslant \sqrt{N}$ (that is, ℓ_0 and ℓ_2 are not too far away from ℓ_1). Levels ℓ_0 and ℓ_2 divide G into three subgraphs G_0, G_1, and G_2 consisting of the vertices on the levels above ℓ_0, between ℓ_0 and ℓ_2, and below ℓ_2, respectively, with the property that G_0 and G_2 contain less than $N/2$ vertices and G_1 has a spanning tree of bounded height $\sqrt{N}$. Refer to Fig. 2.

It can be shown that in order to find a separator for G it is sufficient to find a separator in G_1 [25]. Such a separator can be found using properties of the *dual* graph of G_1. The dual graph $G_1^\star = (V^\star, E^\star)$ of a planar graph G_1 is a planar graph obtained by placing a vertex in each face of G_1 and connecting two faces f_i and f_j adjacent to a common edge $e = (u, v)$ of G_1 with an edge $e^* = (f_i, f_j)$ in $E^\star$. The edge e^* in $G_1^\star$ is called the *dual edge* of e in G_1. Let T' be a subset of the edges in G_1. It is well known that T' is a spanning tree of G_1 *if and only if* $(E - T')^\star$ is a spanning tree in $G^\star$ [23]. Refer to Fig. 3(a). If T' is a

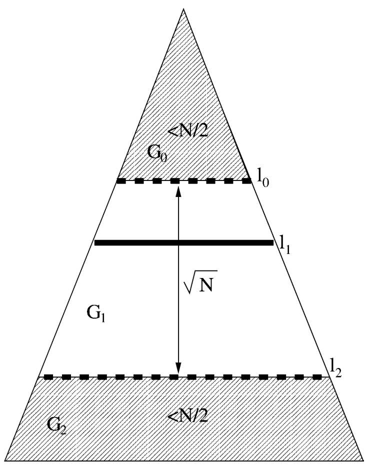

Fig. 2. Planar separator algorithm [25]. G_0 and G_2 have size less than $N/2$ and G_1 has a spanning tree of height $\sqrt{N}$.

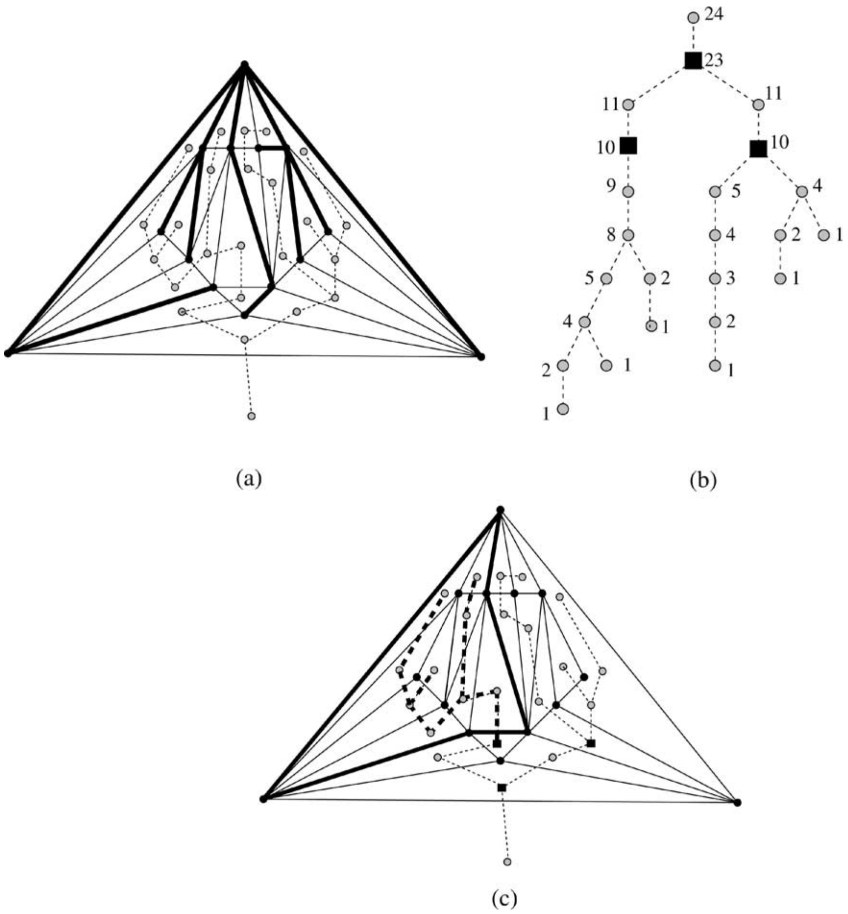

Fig. 3. (a) A triangulated graph G (solid lines), with spanning tree T (solid thick lines), and dual spanning tree $T^\dagger$ (dotted lines). (b) The weight of each vertex of $T^\dagger$ with the attachment vertices of 10-bridges marked. (c) Subtree of $T^\dagger$ and the induced cycle in G.

spanning tree of bounded height $\sqrt{N}$ then adding any edge in $(E - T)$ to T' creates a cycle with at most $2\sqrt{N}$ vertices. Assuming (without loss of generality) that G is triangulated, Lipton and Tarjan [25] proved that there exists an edge $e \in (E - T)$ such that the number of vertices inside and outside the cycle defined by e is $\leqslant 2N/3$, and showed how it can be computed efficiently using a bottom-up traversal of the dual spanning tree $(E - T')^*$. Hutchinson et al. [20] showed how to perform all these operations using $O(\mathrm{sort}(N))$ I/Os provided that a BFS tree of G is given.

Recall that the multi-way planar graph separation problem is the problem of partitioning a planar graph G into $\Theta(N/R)$ subgraphs with $O(R)$ vertices using a set S of separator vertices. The (two-way) $O(\sqrt{N})$-separator algorithm of Hutchinson et al. [20] can be used to develop a recursive $O(\log(N/R) \cdot \mathrm{sort}(N))$ I/O multi-way separator algorithm in a straightforward way. In this section we show how to improve this to $O(\mathrm{sort}(N))$ I/Os by partitioning G into (roughly) M/B subgraphs (instead of two) in each recursive step. We do so using ideas similar to the ones utilized by Goodrich [19]: We identify (roughly) M/B levels in T dividing G into subgraphs of size $O(N/(M/B))$. We then use these levels to find a set of levels with few vertices that divide G into subgraphs such that each subgraph is either of size $O(N/(M/B))$ or has a spanning tree of bounded height. We subdivide the subgraphs with bounded height spanning trees using properties of the dual graphs and recursively subdivide the subgraphs of size $O(N/(M/B))$. In Section 3.1 below we first discuss how to subdivide the bounded height subgraphs I/O-efficiently, and in Section 3.2 we then provide all the details of our algorithm.

3.1. Partitioning a planar graph with bounded height spanning tree

In this section we describe how we in $O(\text{sort}(N))$ I/Os can partition a planar graph $G = (V, E)$ with a spanning tree T of height H into $\Theta(N/R)$ subgraphs of size $O(R)$ each using $O((N/R) \cdot H)$ separator vertices.

Assume for simplicity that G is triangulated. (If this is not the case, we can triangulate it using $O(\text{sort}(N))$ I/Os [20] and mark the added edges so that they can be removed at the end of the partitioning. Note that T remains a spanning tree after the triangulation.) Let $G^\star$ be the dual of G and let $T^\dagger = (E - T)^\star$ be the spanning tree in $G^\star$. Refer to Fig. 3(a). An edge in $T^\dagger$ is the dual of an edge $e = (u, v)$ in $(E - T)$ and since there exists a unique path from u to v in T, adding e to T creates a cycle. Since T has bounded height H this cycle contains at most $2H - 1$ vertices. This way we can think of each edge in $T^\dagger$ as defining a cycle in G of size $O(H)$, which partitions G into the vertices inside the cycle and the vertices outside the cycle. The main idea in our algorithm is to find $O(N/R)$ edges/cycles that partition G into subgraphs of size $O(R)$. Below we discuss how to find $O(N/R)$ edges in $T^\dagger$ such that their removal divides $T^\dagger$ into subtrees of size $O(R)$, and then we discuss how the duals of these edges define $O(N/R)$ cycles in G with the desired properties.

Parallel algorithms for partitioning a tree into subtrees of approximately equal size waere studied by Gazit et al. [32]. We briefly review their notations and results. Let $T^\dagger$ be a tree and define the weight $w(v)$ of a vertex v in $T^\dagger$ to be the number of vertices in the subtree rooted at v. A vertex v is called *R-critical* if v is not a leaf and $\lceil w(v)/R \rceil > \lceil w(v')/R \rceil$ for all children v' of v. Let C be a subset of the vertices in $T^\dagger$. Two edges e and e' of $T^\dagger$ are called *C-equivalent* if there exists a path from e to e' that avoids the vertices C. The graphs induced by the equivalence classes of the C-equivalent edges are called the *bridges* of C. The *attachments vertices* of a bridge I are the vertices in I that are also in C. The *R-bridges* of $T^\dagger$ are the bridges of the set of R-critical vertices of $T^\dagger$. Refer to Fig. 3(b). Gazit et al. [32] prove the following important properties of R-bridges of an N vertex tree $T^\dagger$:

1. The number of R-critical vertices in $T^\dagger$ is at most $2N/R - 1$.
2. If $T^\dagger$ has bounded degree d the number of R-bridges is at most $d(2N/R - 1)$.
3. The number of vertices of an R-bridge is at most $R + 1$.
4. An R-bridge has at most two attachment vertices.

Using the above properties we can easily find $O(N/R)$ edges such that their removal divides $T^\dagger$ into $O(N/R)$ subtrees of size $O(R)$: $T^\dagger$ is a binary tree since G is a triangulated graph, and thus it has at most $4N/R$ R-bridges of size $R+1$ each. Thus if the fewer than $2 \cdot 4N/R$ attachments vertices (or the at most $3 \cdot 2 \cdot 4N/R = O(N/R)$ edges incident to these vertices) are removed, the graph breaks into $O(N/R)$ subgraphs (the R-bridges) of size $O(R)$. That these subgraphs can be used to partition G can be seen as follows. Consider the (at most) two attachment vertices defining an R-bridge I. The two edges in I incident to these vertices define two cycles in G, and the faces inside one of these cycles but outside the other are exactly the faces corresponding to the vertices in I. Since I contains at most $R+1$ vertices (faces in G), the two edges (cycles in G) define a subgraph of G of size at most $3(R+1)$. Overall, since each cycle contains $O(H)$ vertices, the $O(N/R)$ R-bridges and the corresponding adjacent edges define $O((N/R) \cdot H)$ separator vertices partitioning G into $O(N/R)$ subgraphs of $O(R)$ vertices.

To compute the partition of G using T we first compute G^*, and thus $T^\dagger$, in $O(\mathrm{sort}(N))$ I/Os [20]. Then we compute the attachment vertices of the R-bridges of $T^\dagger$. To do so the only problem we need to solve is the computation of the weight of each vertex in $T^\dagger$. This problem, like most other problems on trees, can be solved in $O(\mathrm{sort}(N))$ I/Os [11, 13]. The R-bridges and therefore the partition of $T^\dagger$ can also be computed in $O(\mathrm{sort}(N))$ I/Os using a simple tree traversal. To compute the $O((N/R) \cdot H)$ separator vertices and $O(N/R)$ subgraphs in the partition we scan though the R-bridges and for each vertex v we output the three vertices in G defining the face dual to v to a list L, with each vertex marked with a unique identifier for the R-bridge it corresponds to. This way each vertex in G can appear many times in L and the vertices that appear with at least two distinct identifiers are the separator vertices. All copies of a vertex in a given subgraph are marked with the same R-bridge identifier. Thus we can compute the partition by first identifying and remove all vertices that appear in L with more than one identifier, and then remove duplicate vertices from the resulting list. This can easily be done in $O(\mathrm{sort}(N))$ I/Os using a few sorting and scanning steps.

Lemma 4. *A planar graph G with a spanning tree T of height H can be partitioned into $\Theta(N/R)$ subgraphs of size $O(R)$ using $O((N/R) \cdot H)$ separator vertices in $O(\mathrm{sort}(N))$ I/Os.*

3.2. Separating planar graphs

We are now ready to describe our multi-way separation algorithm in detail. Let $G = (V, E)$ be a planar graph with BFS tree T, and let $L(i)$ be the total number of vertices on levels 0 through i of T. Given a parameter $X < N$, we define the *starter levels* to be the levels i such that the interval $(L(i), L(i+1)]$ contains a multiple of $\lceil N/X \rceil$. It is easy to see that there are at most X starter levels and the number of vertices between consecutive starter levels is smaller than $\lceil N/X \rceil$. Just like the ℓ_1 level in Lipton and Tarjan's algorithm [25], the starter levels divide G in subgraphs of "small" size. However, as previously, the starter levels can contain many vertices. Therefore we consider the first level above each starter level, as well as the first level below each starter level, containing at most Y vertices for a given parameter $Y < N$. We call these levels the *cutter levels*. Now consider the partition of

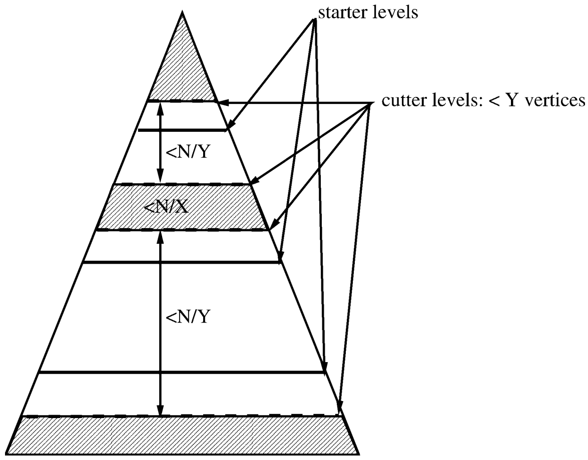

Fig. 4. Starter and cutter levels in T. Graphs between two consecutive cutter levels either have size less than N/X or a spanning tree of height smaller than N/Y.

G into $O(X)$ subgraphs G_i obtained by grouping vertices between two consecutive cutter levels together. If the two cutter levels defining G_i are within two (consecutive) starter levels then G_i has size $O(N/X)$. Otherwise G_i has a spanning tree of height $O(N/Y)$ since each of the levels of T in G_i have more than Y vertices (note that this is not the case for a graph G_i defined by two cutter levels between the same starter levels). Refer to Fig. 4.

In order to compute a multi-way-separation of G we partition the subgraphs of bounded height using the algorithm in Section 3.1 (Lemma 4), and recursively partition the subgraphs of size $O(N/X)$. To do so we need a BFS tree for each subgraph G_i; the part of T in that falls within G_i is not a BFS tree for G_i, since it is not connected. However, we can easily produce a BFS tree for G_i by introducing a "fake" root v_i and connecting it with "fake" edges to all vertices just below the top cutter level defining G_i. Note that if T is given level-by-level the BFS trees for all subgraphs G_i can easily be computed in $O(N/B)$ I/Os. Since v_i replaces at least one vertex on the cutter level, the total size of the subgraphs on any level of the recursion remains $O(N)$. The fake vertices and edges are marked and removed from the final partitioned graph. This can easily be done in $O(N/B)$ I/Os.

To obtain a partition with $O(\text{sort}(N))$ separator vertices we choose $Y = N/\sqrt{R}$. Each bounded height subgraph G_i of size N_i has height $\sqrt{R}$, and can thus be partitioned using Lemma 4 into $\Theta(N_i/R)$ subgraphs of size $O(R)$ using $O((N_i/R)\cdot\sqrt{R}) = O(N_i/\sqrt{R})$ separator vertices. Apart from the $O(N/\sqrt{R})$ separator vertices used to partition each of the at most X bounded height subgraphs, the at most X cutter levels contribute $O(X\cdot Y) = O(X\cdot N/\sqrt{R})$ separator vertices. Thus the total number of separator vertices is given by $S(N) \leqslant O(XN/\sqrt{R}) + O(N/\sqrt{R}) + X\cdot S(N/X)$ (and $S(R)=0$). If we choose $X = (M/B^2)^{1/4}$ and assume $R > B\sqrt{M}$, we get that $XN/\sqrt{R} = O(N/B)$ and therefore $S(N) = O(N/B) + (M/B^2)^{1/4}\cdot S(N/(M/B^2)^{1/4})$. This solves to $O(N/B)\log_{(M/B^2)^{1/4}}(N/R)$, which is $O(\text{sort}(N))$ under the assumption that $M > B^{2+\varepsilon}$.

That our algorithm uses $O(\mathrm{sort}(N))$ I/Os can be seen as follows. The preprocessing step of representing T level by level, and thus also computing the BFS level for each vertex, can easily be performed in $O(\mathrm{sort}(N))$ I/Os using standard tree algorithms [11,13]. Not counting the I/Os used to partition the subgraphs with bounded height, one recursion step can be performed in $O(N/B)$ I/Os, and the recurrence for the number of I/Os is then $T(N) \leqslant N/B + X \cdot T(N/X) = O(\mathrm{sort}(N))$. Since we do not recurse on subgraphs G_i with bounded height but immediately subdivide them using $O(\mathrm{sort}(G_i))$ I/Os, the total cost of partitioning all such subgraphs over all levels of the recursion adds up to $O(\mathrm{sort}(N))$.

So far we assumed $R > B\sqrt{M}$. If we want to partition a graph G into subgraphs of size $R \leqslant B\sqrt{M} < M$ we can first use the algorithm above to partition G into subgraphs of size $O(M)$ and then load each subgraph into internal memory in turn and apply the algorithm of Lipton and Tarjan [25] recursively until all subgraphs have size $O(R)$. This only requires an extra $O(N/B)$ I/Os and introduces $O(M/\sqrt{R})$ separator vertices in each of the $O(N/M)$ subgraphs, for a total of $O(N/\sqrt{R})$ vertices. Thus we have the following.

Theorem 2. *Let $G = (V, E)$ be a planar graph and T a breadth-first search tree for G. For any value of R, the graph can be partitioned into $O(N/R)$ subgraphs G_i of size $O(R)$ using a set S of $O(\mathrm{sort}(N) + N/\sqrt{R})$ separator vertices in $O(\mathrm{sort}(N))$ I/Os.*

For every subgraph G_i in a multi-way separation, we call the separator vertices adjacent to G_i the *boundary vertices* of G_i or, in short, the *boundary* ∂G_i of G_i (the union of a graph G_i and its boundary ∂G_i is sometimes called a *region*). Frederickson [18] developed an algorithm for modifying a partitioning of a *bounded degree*[7] planar graph into $S = O(N/\sqrt{R})$ separator vertices and $O(N/R)$ subgraphs of size $O(R)$, such that each subgraph only has $O(S/(N/R)) = O(\sqrt{R})$ boundary vertices. The algorithm works by computing a weighted version of multi-way separation in each subgraph $\partial G_i \cup G_i$. Using Theorem 2 and choosing R such that $\mathrm{sort}(N) = O(N/\sqrt{R})$ we obtain a partitioning with $S = O(N/\sqrt{R})$ separator vertices. Since we in this case have

$$R = O\left(\frac{B^2}{\log^2_{M/B} \frac{N}{B}}\right) = O\left(B^2\right) = O(M)$$

(and since ∂G_i also has $O(M)$ vertices because of the bounded degree) we can directly apply Fredrickson's algorithm (that is, load each subgraph and its boundary in main memory in turn and apply a weighted separator algorithm) to obtain a separation with each subgraph having $O(\sqrt{R})$ boundary vertices. Since this takes $O((N/R) \cdot (R/B)) = O(N/B)$ I/Os we have the following:

Lemma 5. *Let $G = (V, E)$ be a bounded degree planar graph and T a breadth-first search tree for G. For $R = O(B^2/\log^2_{M/B}(N/B))$, G can be partitioned in $O(\mathrm{sort}(N))$ I/Os into $O(N/R)$ subgraphs G_i of size $O(R)$ using a set S of $O(N/\sqrt{R})$ separator vertices, such that each subgraph G_i has $O(\sqrt{R})$ boundary vertices.*

[7] Any graph can easily be transformed into a graph with each vertex having degree at most 3 [18].

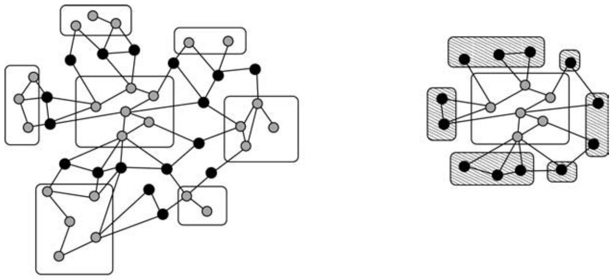

Fig. 5. (a) Separation of a graph into subgraphs (boxed) and separators (black). (b) A subgraph in the partition with the boundary sets of its boundary vertices.

A *boundary set* in a multi-way separation is a maximal subset of separator vertices such that all vertices in the subset are adjacent to exactly the same subgraphs. Refer to Fig. 5. Frederickson [18] developed an algorithm for modifying a partition of a bounded degree planar graph into $S = O(N/\sqrt{R})$ separator vertices and $O(N/R)$ subgraphs G_i of size $O(R)$,[8] such that the number of boundary sets is $O(N/R)$. The algorithm considers the connected components in the graph G^- obtained by removing the vertices in S from G and groups them together appropriately. It only utilizes connected component adjacency information, that is, it works on the graph G_c obtained from G by contracting the vertices in each connected component of G^- (and removing duplicate edges). Since G_c is connected and of bounded degree it has $O(S)$ vertices and edges; using G_c Frederickson's algorithm can be used to compute the modified partitioning in $O(S)$ I/Os. Using Theorem 2 and choosing R such that $N/\sqrt{R} = O(\text{sort}(N))$, that is $R = \Omega(B^2/\log^2_{M/B}(N/B))$, we obtain a partitioning with $S = O(\text{sort}(N))$ separator vertices. Using an $O(\text{sort}(N))$ connected component algorithm [13] and a few scanning and sorting steps we can then easily compute G_c in $O(\text{sort}(N))$ I/Os. Since G_c has $S = O(\text{sort}(N))$ vertices and edges we can then directly apply Fredrickson's algorithm [18] and in $O(\text{sort}(N))$ I/Os obtain a separation with $O(N/R)$ boundary sets.

Lemma 6. *Let $G = (V, E)$ be a bounded degree planar graph and T a breadth-first search tree for G. For $R = \Omega(B^2/\log^2_{M/B}(N/B))$, G can be partitioned in $O(\text{sort}(N))$ I/Os into $O(N/R)$ subgraphs G_i of size $O(R)$ using a set S of $O(N/\sqrt{R})$ separator vertices, such that the number of boundary sets is $O(N/R)$.*

Combining Lemma 5 and Lemma 6 (choosing $R = \Theta(B^2/\log^2_{M/B}(N/B)))$ we obtain following:

Theorem 3. *Let $G = (V, E)$ be a bounded degree planar graph and T a breadth-first search tree for G. Then G can be partitioned in $O(\text{sort}(N))$ I/Os into $O((N/B^2) \cdot$*

[8] Note that a subgraph G_i is not necessarily connected.

$\log^2_{M/B}(N/B))$ subgraphs G_i of size $O(B^2/\log^2_{M/B}(N/B))$ using $S = O(\mathrm{sort}(N))$ separator vertices such that:

1. *The number of boundary vertices of each subgraph G_i is $O(B/\log_{M/B}(N/B))$.*
2. *The number of boundary sets is $O((N/B^2) \cdot \log^2_{M/B}(N/B))$.*

4. Single source shortest paths on planar graphs

Dijkstra's algorithm [16] is probably the most well-know single source shortest path algorithm. The algorithm iteratively grows a shortest path tree using a priority queue on the vertices not yet included in the tree. This is very similar to the way Prim's MST algorithm [16] grows a minimal spanning tree (and as in the case of Prim's algorithm, a direct implementation of Dijkstra's algorithms is not I/O-efficient). In this section we show how to use Theorem 3 to obtain a modified and I/O-efficient version of Dijkstra's algorithm for planar graphs of bounded degree. The main idea in our algorithm is to use multi-way separation to reduce a single source shortest path problem on a (non-planar) graph G with $O(N)$ vertices and edges to the same problem on a graph with $O(\mathrm{sort}(N))$ vertices and $O(N)$ edges, and to utilize that each subgraph is adjacent to a small number of separator vertices to process the $O(N)$ edges I/O-efficiently. These ideas are similar to the ones utilized by Frederickson [18].

Let $\{G_i = (V_i, E_i)\}$ be the $O(N/R)$ subgraphs of size $O(R)$ obtained by partitioning G using the algorithm in Theorem 3. Consider a shortest path P between the source vertex s and a vertex t in G, and let $\{s_0, s_1, \ldots\}$ be the set of separator vertices in P in the order they appear along the path. The part of P between s_i and s_{i+1} is completely within some subgraph G_i and it must be the shortest path between s_i and s_{i+1} within G_i. Thus we can find the shortest path from s to all separator vertices by solving the SSSP problem on the graph G^R obtained by replacing each subgraph G_i with a complete graph on its boundary vertices, where the weight of an edge (u, v) is equal to the weight of the shortest path between u and v in G_i. If the source s is not a separator vertex, it is also included in G^R along with edges to the boundary vertices of the subgraph containing it. The graph G^R has $O(\mathrm{sort}(N))$ vertices, and since the partition of G consists of $O((N/B^2) \cdot \log^2_{M/B}(N/B))$ subgraphs with $O(B/\log_{M/B}(N/B))$ boundary vertices each it has $O((N/B^2) \cdot \log^2_{M/B}(N/B) \cdot (B/\log_{M/B}(N/B))^2) = O(N)$ edges.

After computing the partition of G using $O(\mathrm{sort}(N))$ I/Os, G^R can be computed by loading each subgraph G_i and its boundary vertices into main memory in turn, use an internal memory all-pair-shortest-paths algorithm to compute the weights of the new edges corresponding to G_i, and write these edges back to disk. Since each of the S separator vertices is a boundary vertex for at most $O(1)$ subgraphs (because of the bounded degree), we use $O(N/B + S) = O(\mathrm{sort}(N))$ I/Os to load all the subgraphs and their boundary vertices. We also use $O(N/B)$ I/Os to write the new edges, and thus G^R can be computed in $O(\mathrm{sort}(N))$ I/Os in total. Similarly to the way G^R is computed from G in $O(\mathrm{sort}(N)$ I/Os, the lengths of the shortest paths from s to all vertices in G can be computed in $O(\text{sort}(N))$ I/Os once the lengths of the shortest paths in G^R have been computed; we simply load each subgraph G_i and its boundary vertices (now marked with shortest path lengths) into main memory in turn, and use an internal memory algorithm to compute the shortest path $\delta(s,t)$ from s to each vertex $t \in V_i$ using the formula $\delta(s,t) = \min_v\{\delta(s,v) + \delta_{G_i}(v,t)\}$, where v ranges over all boundary vertices of G_i.

To solve the the SSSP problem on G^R in $O(\text{sort}(N))$ I/Os we use a modified version of Dijkstra's algorithm. The idea of Dijkstra's algorithm is to grow a SSSP tree T_G incrementally while maintaining a priority queue on the vertices not yet included in the tree; the priority of a vertex v is the weight of the shortest path from the source s to v such that all but the last edge is in T_G. The algorithm repeatedly extracts the minimum priority vertex v, adds it (and the relevant edge incident to it) to T_G, and updates the priority of each vertex u adjacent to v. Specifically, if w_e is the weight of the edge $e = (v,u)$, the weight $\delta(s,v) + w_e$ of the path from s to u thorough v is compared to the currently priority of u (weight of the current shortest path to u), and an update is performed if the new weight is smaller. Even though G^R only has $O(\text{sort}(N))$ vertices, a direct implementation of Dijkstra's algorithm does not lead to an I/O-efficient algorithm, mainly because the current priority of a given vertex cannot be obtained without doing an I/O. Thus processing the $O(N)$ edges leads to an $\Omega(N)$ algorithm.[9]

To be able to obtain the priority of a vertex I/O-efficiently, and thus be able to perform $O(N)$ update/decrease-priority in $O(\text{sort}(N))$ I/Os using a delete and insert operation on the external priority queues of [6,10], we exploit the grouping of boundary vertices into boundary sets. The boundary sets allow us to implement Dijkstra's algorithm I/O-efficiently as follows: Apart from the priority queue PQ on the vertices, we maintain a list L of the current priorities of the vertices, that is, we maintain the same priority information in PQ and L. We store vertices in the same boundary set consecutively in L. The algorithm now repeatedly extracts the minimal priority vertex v from PQ and loads the $O(B/\log_{M/B}(N/B)) = O(B)$ edges incident to v into main memory. Next the priorities of the $O(B/\log_{M/B}(N/B))$ boundary vertices adjacent to v are retrieved from L, and it is determined (without further I/Os) which of these vertices need to have their priorities updated in PQ and L. Finally the relevant updates are performed on PQ (using a delete and an insert per update) and the boundary vertices (with updated priorities) are written back to L.

For each of the $O(\text{sort}(N))$ vertices v in G^R our algorithm use $O(1)$ I/Os to load the edges incident to v. The number of I/Os needed to load (and write) the $O(B/\log_{M/B}(N/B))$ boundary vertices adjacent to v from L can be analyzed as follows. Since each vertex is adjacent to $O(B/\log_{M/B}(N/B)) = O(B)$ vertices, each boundary set also contains $O(B)$ vertices. Since they are stored consecutively in L, a boundary set

[9] This can be improved to $O((N/B)\log_2(N/B))$ I/Os, or $O((\log_2(N/B))/B)$ I/Os per edge, using a priority queue by Kumar and Schwabe [24] that supports a decrease-priority operation where the current priority p of an element does not need to be know when the operation is performed—the update is only actually made if the new priority is smaller than p.

can be loaded in $O(1)$ I/Os. During the whole algorithm, each boundary set is accessed $O(B/\log_{M/B}(N/B))$ times (once by each of its adjacent vertices), and thus we use

$$O\left(\frac{B}{\log_{M/B}\frac{N}{B}}\cdot\frac{N}{B^2}\cdot\log^2_{M/B}\frac{N}{B}\right)=O\big(\mathrm{sort}(N)\big)$$

I/Os in total to access the $O((N/B^2)\cdot\log^2_{M/B}(N/B))$ boundary sets in L. (Note that if the boundary sets were not stored consecutively in L we would use $O(B/\log_{M/B}(N/B))$ I/Os to load the vertices adjacent to v, for a total of $O(B/\log_{M/B}N/B\cdot\mathrm{sort}(N))=O(N)$ I/Os for the $O(\mathrm{sort}(N))$ vertices). Finally, our algorithm performs $O(N)$ operations on PQ using $O(\mathrm{sort}(N))$ I/Os in total [6,10]. We have obtained the following.

Theorem 4. *Let G be a bounded degree planar graph and T a BFS tree for G. The weights of the shortest paths from a given source vertex s to all vertices in G can be computed in $O(\mathrm{sort}(N))$ I/Os.*

In the above algorithm we focused on computing the weights of the shortest paths in G. If we are interested in the actual paths, that is, in the shortest path tree T_G, standard techniques can easily be used to augment the algorithm so it outputs the edges in T_G. Given T_G, Hutchinson et al. [20] showed how to store it such that for any vertex t, the shortest path between the source s and t can be returned in P/B I/Os, where P is the number of vertices on the path.

Corollary 1. *Let G be a bounded degree planar graph and T a BFS tree for G. A data structure can be constructed in $O(\mathrm{sort}(N))$ I/Os such that the shortest path from a given source vertex s and any vertex t can be found in $O(P/B)$ I/Os, where P is the number of vertices on the path.*

5. Conclusions and open problems

In this paper we developed an improved $O(\mathrm{sort}(N)\cdot\log\log(VB/E))$ algorithm for MST on general undirected graphs. It remains a challenging open problem to develop an $O(\mathrm{sort}(N))$ I/O algorithm. We also showed that planar BFS, multi-way graph separation, and SSSP are essentially equivalent by providing $O(\mathrm{sort}(N))$ reductions between them. Recently, it was also shown how to reduce planar undirected DFS to BFS [7]. Very recently, these reductions lead to $O(\mathrm{sort}(N))$ I/O algorithms for all fundamental problems on planar undirected graphs [28]. It remains an open problem to develop $O(\mathrm{sort}(N))$ algorithms for planar directed graphs.

References

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

[2] P.K. Agarwal, L. Arge, T.M. Murali, K. Varadarajan, J.S. Vitter, I/O-efficient algorithms for contour line extraction and planar graph blocking, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 1998, pp. 117–126.

[3] A. Aggarwal, J.S. Vitter, The Input/Output complexity of sorting and related problems, Comm. ACM 31 (9) (1988) 1116–1127.

[4] ARC/INFO, Understanding GIS—the ARC/INFO method, ARC/INFO, 1993, Rev. 6 for workstations.

[5] L. Arge, The I/O-complexity of ordered binary-decision diagram manipulation, in: Proc. International Symposium on Algorithms and Computation, in: Lecture Notes in Comput. Sci., vol. 1004, 1995, pp. 82–91. A complete version appears as BRICS Technical Report RS-96-29, University of Aarhus.

[6] L. Arge, The buffer tree: a technique for designing batched external data structures, Algorithmica 37 (1) (2003) 1–24.

[7] L. Arge, U. Meyer, L. Toma, N. Zeh, On external-memory planar depth first search, J. Graph Algorithms Appl. 7 (2) (2003) 105–129.

[8] L. Arge, L. Toma, J.S. Vitter, I/O-efficient algorithms for problems on grid-based terrains, ACM J. Exp. Algorithmics 6 (1) (2001).

[9] O. Borůvka, O jistém problému minimálním, Práca Moravské Přírodovědecké Společnosti 3 (1926) 37–58.

[10] G.S. Brodal, J. Katajainen, Worst-case efficient external-memory priority queues, in: Proc. Scandinavian Workshop on Algorithms Theory, in: Lecture Notes in Comput. Sci., vol. 1432, 1998, pp. 107–118.

[11] A.L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, J.R. Westbrook, On external memory graph traversal, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 2000, pp. 859–860.

[12] A.L. Buchsbaum, J.R. Westbrook, Maintaining hierarchical graph views, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 2000, pp. 566–575.

[13] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter, External-memory graph algorithms, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 1995, pp. 139–149.

[14] F. Chin, J. Lam, I. Chen, Efficient parallel algorithms for some graph problems, Comm. ACM 25 (1982) 659–665.

[15] R. Cole, U. Vishkin, Approximate parallel scheduling, II. Applications to logarithmic-time optimal parallel graph algorithms, Inform. and Comput. 92 (1) (1991) 1–47.

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[17] E. Feuerstein, A. Marchetti-Spaccamela, Memory paging for connectivity and path problems in graphs, in: Proc. International Symposium on Algorithms and Computation, in: Lecture Notes in Comput. Sci., vol. 762, 1993, pp. 416–425.

[18] G.N. Frederickson, Fast algorithms for shortest paths in planar graphs, with applications, SIAM J. Comput. 16 (1987) 1004–1022.

[19] M. Goodrich, Planar separators and parallel polygon triangulation, J. Comput. System Sci. 51 (3) (1995) 374–389.

[20] D. Hutchinson, A. Maheshwari, N. Zeh, An external-memory data structure for shortest path queries, in: Proc. Annual Combinatorics and Computing Conference, in: Lecture Notes in Comput. Sci., vol. 1627, 1999, pp. 51–60.

[21] J.F. JáJá, An Introduction to Parallel Algorithms, Addison–Wesley, Reading, MA, 1992, Chapter 5, pp. 222–227.

[22] D.E. Knuth, Sorting and Searching, The Art of Computer Programming, vol. 3, second ed., Addison–Wesley, Reading, MA, 1998.

[23] D. Kozen, The Design and Analysis of Algorithms, Springer, Berlin, 1992.

[24] V. Kumar, E. Schwabe, Improved algorithms and data structures for solving graph problems in external memory, in: Proc. IEEE Symposium on Parallel and Distributed Processing, 1996, pp. 169–177.

[25] R.J. Lipton, R.E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36 (1979) 177–189.

[26] A. Maheshwari, N. Zeh, External memory algorithms for outerplanar graphs, in: Proc. International Symposium on Algorithms and Computation, in: Lecture Notes in Comput. Sci., vol. 1741, 1999, pp. 307–316.

[27] A. Maheshwari, N. Zeh, I/O-efficient algorithms for graphs of bounded treewidth, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 2001, pp. 89–90.

[28] A. Maheshwari, N. Zeh, I/O-optimal algorithms for planar graphs using separators, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 2002, pp. 372–381.

[29] K. Mehlhorn, U. Meyer, External-memory breadth-first search with sublinear I/O, in: Proc. European Symposium on Algorithms, in: Lecture Notes in Comput. Sci., vol. 2461, 2002, pp. 723–735.
[30] K. Munagala, A. Ranade, I/O-complexity of graph algorithms, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 1999, pp. 687–694.
[31] M.H. Nodine, M.T. Goodrich, J.S. Vitter, Blocking for external graph searching, Algorithmica 16 (2) (1996) 181–214.
[32] J.H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, 1993.
[33] R.E. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, PA, 1983.
[34] L. Toma, External memory graph algorithms and applications to geographic information systems, PhD thesis, Duke University, 2003.
[35] J.D. Ullman, M. Yannakakis, The input/output complexity of transitive closure, Ann. Math. Artif. Intell. 3 (1991) 331–360.

Further reading

[36] L. Arge, G.S. Brodal, L. Toma, On external memory MST, SSSP and multi-way planar graph separation, in: Proc. Scandinavian Workshop on Algorithms Theory, in: Lecture Notes in Comput. Sci., vol. 1851, 2000, pp. 433–447.