A Faster Cache-Oblivious Shortest-Path Algorithm for Undirected Graphs with Bounded Edge Lengths

Luca Allulli* Peter Lichodzijewski† Norbert Zeh‡

Abstract

We present a cache-oblivious algorithm for computing single-source shortest paths in undirected graphs with non-negative edge lengths. The algorithm incurs $O\big(\sqrt{(nm \log W)/B} + (m/B) \log n + \mathrm{MST}(n, m)\big)$ memory transfers on a graph with n vertices, m edges, and real edge lengths between 1 and W; B denotes the cache block size, and $\mathrm{MST}(n, m)$ denotes the number of memory transfers required to compute a minimum spanning tree of a graph with n vertices and m edges. Our algorithm is the first cache-oblivious shortest-path algorithm incurring less than one memory transfer per vertex if the graph is sparse ($m = O(n)$) and $W = 2^{o(B)}$.

1 Introduction

Let $G = (V, E)$ be a graph with vertex set V and edge set E, let s be a vertex of G, called the *source vertex*, and let $\ell : E \to \mathbb{R}^+$ be an assignment of non-negative real *lengths* to the edges of G. The *single-source shortest-path* (SSSP) problem is to find, for every vertex $v \in V$, the distance $D(v)$ from s to v, that is, the length of a shortest path from s to v in G.

The classical SSSP-algorithm for graphs with non-negative edge lengths is Dijkstra's algorithm [9], which has seen many improvements for planar graphs [12], undirected graphs with integer or float edge lengths [19, 20], and undirected graphs with real edge lengths [18]. Unfortunately, only the algorithm of [12], if implemented appropriately [11], makes good use of cache memory by avoiding random memory accesses.

Much previous work has focused on developing cache-efficient shortest-path algorithms, most of them *external-memory algorithms*. These algorithms are designed and analyzed in the *I/O-model* [1], which assumes that the computer has an *internal memory* that can hold M vertices or edges and that the graph is stored in *external memory* (on disk). Computation can occur only on items in internal memory. Therefore, in order to process the graph, the algorithm has to swap pieces of it between internal and external memory, which it does in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* or *memory transfer* (MT). The complexity of an algorithm in this model is the number of memory transfers it performs. An external-memory algorithm requires knowledge of M and B to initiate memory transfers explicitly.

The *cache-oblivious model* [10] prescribes that an algorithm be designed in the RAM-model, that is, without using the parameters M and B in the algorithm, but analyzed in the I/O-model. For this analysis, a paging algorithm has to perform the memory transfers necessary to bring accessed items into internal memory. To make room for the loaded block, another block must be evicted from internal memory. The cache-oblivious model assumes that the paging algorithm is optimal in the sense that it chooses blocks to be evicted so that the total number of memory transfers necessary to serve all memory accesses performed by the algorithm is minimized. For a justification of this ideal-cache model, see [10]. Since the parameters M and B are used only in the analysis, but not by the algorithm, the analysis applies to any level in a multi-level memory hierarchy, that is, the resulting algorithm is adaptive to any such hierarchy. For many cache-oblivious algorithms, a *tall-cache assumption* is made, which means that it is assumed that $M = \Omega\big(B^{1+\varepsilon}\big)$, for some $\varepsilon > 0$. For example, Brodal and Fagerberg [4, 5] show that this assumption is sufficient and necessary to match the external sorting bound of $\mathrm{sort}(N) = \Theta\big((N/B) \log_{M/B}(N/B)\big)$ MT's using a cache-oblivious sorting algorithm. Our algorithm also assumes that $M = \Omega\big(B^{1+\varepsilon}\big)$.

A number of papers propose algorithms for solv-

*Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. Email: allulli@dis.uniroma1.it. Supported in part by the research project "Algoritmi evoluti per Internet e per il Web" sponsored by the University of Rome "La Sapienza", and by the Natural Sciences and Engineering Research Council of Canada.

†Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Canada. Email: piotr@cs.dal.ca. Supported by the Natural Sciences and Engineering Research Council of Canada.

‡Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Canada. Email: nzeh@cs.dal.ca. Supported by the Natural Sciences and Engineering Research Council of Canada and the Canadian Foundation for Innovation.

ing variants of SSSP on undirected graphs in a cache-efficient manner [3, 6, 8, 13, 14, 15, 16, 17]. For *unweighted* graphs, that is, for breadth-first search (BFS), the best external-memory algorithm is that of [14], which performs $O\big(\sqrt{nm/B} + \mathrm{ST}(n,m)\big)$ MT's, where $n = |V|$, $m = |E|$, and $\mathrm{ST}(n,m)$ is the cost of computing a spanning tree.[1] In [6], a cache-oblivious version of this algorithm has been obtained whose complexity is $O\big(\sqrt{nm/B} + (m/B)\log n + \mathrm{ST}(n,m)\big)$ MT's. We call this algorithm CO-BFS. The best general external-memory shortest-path algorithm performs $O(n + (m/B)\log_2(n/B))$ MT's [13]. This bound has been matched in the cache-oblivious model [6, 8] by developing a cache-oblivious priority queue to replace the external one used in [13]. For graphs with real edge lengths between 1 and W, an external-memory shortest-path algorithm with complexity $O\big(\sqrt{(nm\log W)/B} + \mathrm{MST}(n,m)\big)$ is presented in [15]; $\mathrm{MST}(n,m)$ denotes the cost of computing a minimum spanning tree.[1] We call this algorithm MZ-SSSP. MZ-SSSP is better than the algorithm of [13] for sparse graphs with small edge lengths. Recently, a version of MZ-SSSP has been obtained whose complexity is $O\big(\sqrt{nm/B}\log n + \mathrm{MST}(n,m)\big)$ MT's [16], that is, is independent of the edge lengths.

All cache-efficient SSSP-algorithms discussed in the previous paragraph are variants of Dijkstra's algorithm, implemented using cache-efficient priority queues. The main bottleneck in the algorithms of [6, 8, 13, 17] is that retrieving the adjacency lists of visited vertices, in order to relax their incident edges, requires at least one MT per vertex because the order in which vertices are visited is hard to predict. The improved algorithms of [6, 14, 15, 16] overcome this bottleneck using the following idea: Instead of accessing one adjacency list at a time, group vertices appropriately and form *edge groups* by concatenating the adjacency lists of the vertices in each vertex group; when the first vertex in a vertex group is visited, load the whole corresponding edge group into a *hot pool*. Subsequent visits to vertices in the group find their adjacency lists in the hot pool. Thus, the number of memory transfers caused by random accesses to adjacency lists is reduced, but at the expense of increasing the cost per edge because an edge may be inspected by many searches of the hot pool before it is finally relaxed (and removed from the hot pool). In this paper, we present a cache-oblivious shortest-path algorithm based on MZ-SSSP. The following theorem summarizes our result.

THEOREM 1.1. *Given an undirected graph G with n vertices, m edges, and edge lengths between 1 and W, and provided that $M = \Omega\big(B^{1+\varepsilon}\big)$, the single-source shortest-path problem on G can be solved by a cache-oblivious algorithm that incurs $O\big(\sqrt{(nm\log W)/B} + (m/B)\log n + \mathrm{MST}(n,m)\big)$ memory transfers.*

The complexity of our algorithm can be seen as the perfect marriage between the complexities of MZ-SSSP and CO-BFS. In particular, our algorithm matches the cost of cache-oblivious BFS if the edge lengths come from a range of constant size.

Section 2 gives an overview of our algorithm. At a high level, our algorithm is almost identical to MZ-SSSP. The central challenge, addressed by our paper, is the design of a cache-oblivious replacement for the hot pool structure used in MZ-SSSP. MZ-SSSP uses a *one-dimensional* hot pool hierarchy to cope with edges of different lengths, while making decisive use of its knowledge of B in computing *disjoint* edge groups of the right diameter so that edges do not linger in the hot pool too long. CO-BFS also uses a one-dimensional hot pool hierarchy, but partitions the graph into *nested* groups of increasing diameter, in order to be adaptive to any block size. In computing these groups and in organizing the hot pool, it uses that all edges have the same length. Our algorithm uses a nested group partition that combines the crucial characteristics of the two partitions used in MZ-SSSP and CO-BFS and uses a *two-dimensional* hot pool whose rows play the role of the hot pools in MZ-SSSP, while the columns play the role of the hot pools in CO-BFS. Section 3 discusses the group partition used by our algorithm. Section 4 discusses the hot pool.

2 Overview

At a high level, our algorithm is almost identical to MZ-SSSP, which in turn is an I/O-efficient implementation of Dijkstra's algorithm. It maintains a *tentative distance* $d(v)$ for every vertex $v \in G$, which is an upper bound on its true distance from s. Initially, $d(s) = 0$ and $d(v) = +\infty$, for all $v \neq s$. This upper bound is improved iteratively until $d(v) = D(v)$. At this point v is *visited*, that is, every edge vw in v's adjacency list $A(v)$ is *relaxed*, which means that $d(w)$ is replaced with $\min(d(w), d(v) + \ell(vw))$.

As MZ-SSSP, our algorithm differs from Dijkstra's algorithm in that it may visit more than one vertex at a time, as long as the distances of all simultaneously visited vertices from s differ by less than the minimum edge weight. Our algorithm differs from MZ-SSSP in that it loads *the whole graph* into the hot pool at the

[1] The currently best bounds for computing a (minimum) spanning tree are $\mathrm{ST}(n,m) = \mathrm{MST}(n,m) = O(\mathrm{sort}(m)\log\log(nB/m))$ in the I/O-model [3, 7] and $\mathrm{ST}(n,m) = \mathrm{MST}(n,m) = O(\mathrm{sort}(m)\log\log n)$ in the cache-oblivious model [2].

```
CO-SSSP(G)
1  Q ← an empty priority queue
2  H ← a hot pool structure storing the adjacency lists of all vertices in G
3  Update(Q, s, 0)
4  while Q ≠ ∅
5        do L ← BatchedDeleteMin(Q)
6           Retrieve all adjacency lists of vertices in L from H, and place them into a list S.
7           For every edge vw ∈ S, perform an Update(Q, w, d(v) + ℓ(vw)) operation.
```

Figure 1: Outline of the algorithm.

beginning of the algorithm, just as CO-BFS does. The high-level procedure is shown in Figure 1. The priority queue and the hot pool ensure that the queries in Lines 5 and 6 output the vertices in L sorted by their IDs and the edges in S sorted by the IDs of their tails. Thus, a single scan of L and S suffices to implement Line 7.

The priority queue Q used in the algorithm supports two operations: An Update(x, p) operation inserts vertex x into Q, with priority p, if it is not currently in Q; if x is in Q and has priority p', its priority is replaced with $\min(p, p')$; that is, Update is a combined Insert/DecreaseKey operation. A BatchedDeleteMin operation deletes and returns a set L of vertices with two properties: every vertex in L has a priority less than that of any vertex that remains in Q, and the priorities of two vertices in L differ by at most 1.

Note that Line 7 may re-insert already visited vertices into Q because it performs an Update operation on w, for *every* edge $vw \in S$, without checking w's status. The algorithms of [6, 8, 13, 15, 16] face the same problem and address it by using a second priority queue to eliminate re-inserted vertices before they can be visited for a second time. As we discuss below, our algorithm can deal with this problem in a much more straightforward manner because it loads the whole graph into the hot pool at the beginning. The correctness of the algorithm then follows from the correctness of Dijkstra's algorithm because none of the vertices in L can be on the shortest path to another vertex in L (see [15] for a detailed argument). This proves the following lemma.

Lemma 2.1. *Procedure CO-SSSP correctly solves the SSSP problem on undirected graphs with non-negative edge weights.*

In the remainder of this section, we review the priority queue used in our algorithm and provide a high-level description of the hot pool structure. The rest of the paper is then concerned with providing the details of the latter structure.

2.1 Priority queue. Our algorithm utilizes the bucket priority queue of [15], which is easily implemented in a cache-oblivious manner without changing the cost of the priority queue operations. Here we give an overview and review the properties of this structure relevant to our algorithm. The priority queue consists of $r = \lfloor \log W \rfloor + 1$ buckets $\mathcal{B}_1, \ldots, \mathcal{B}_r$, each of which is an array of vertices, sorted by vertex IDs. Buckets $\mathcal{B}_1, \ldots, \mathcal{B}_r$ are stored consecutively in an array $\mathcal{B} = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_r$. The buckets are defined by priorities $p_0 \le p_1 \le \cdots \le p_r = +\infty$ in the sense that all priority queue entries (x, p) with priority $p_{i-1} \le p < p_i$ are stored in bucket $\mathcal{B}_i$; in particular, no entry has priority less than p_0. Priorities $p_0, \ldots, p_r$ satisfy that, for $1 < i < r$, either $p_i = p_{i-1}$ or $2^{i-2}/3 \le p_i - p_{i-1} \le 2^{i-2}$; for $i = 1$, we have $0 < p_1 - p_0 \le 1$, except at the very beginning of the algorithm. The priorities are initialized to $p_0 = \cdots = p_{r-1} = 0$ and $p_r = +\infty$.

Update(x, p) operations are implemented by inserting Update(x, p) signals into an update buffer $\mathcal{U}_1$. More precisely, there is one such buffer $\mathcal{U}_i$ associated with each bucket $\mathcal{B}_i$, and buffers $\mathcal{U}_1, \ldots, \mathcal{U}_r$ are concatenated to form an array $\mathcal{U} = \mathcal{U}_1 \circ \cdots \circ \mathcal{U}_r$. Over time, signals move to higher buffers until they find the bucket where the new element is to be inserted. This happens during BatchedDeleteMin operations.

A BatchedDeleteMin operation first ensures that $\mathcal{B}_1$ is non-empty. To do so, it scans buckets $\mathcal{B}_1, \ldots, \mathcal{B}_i$, for each $1 \le j \le i$ applying the signals in $\mathcal{U}_j$ to $\mathcal{B}_j$ and then merging them into $\mathcal{U}_{j+1}$. It stops when it finds the first bucket $\mathcal{B}_i$ that is non-empty after the updates in $\mathcal{U}_i$ have been applied to it. It then sets $p_0 = p_{\min}$, where $p_{\min}$ is the minimum priority of any element in $\mathcal{B}_i$ and updates priorities $p_1, \ldots, p_{i-1}$ so that the above constraints on these priorities are satisfied; p_{i-1} equals p_i at the end of this procedure. The elements of $\mathcal{B}_i$ are then distributed over $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$ according to their priorities, which is achieved by moving all elements with priority less than p_{i-1} from $\mathcal{B}_i$ to $\mathcal{B}_{i-1}$, then moving all elements with priority less than p_{i-2} from $\mathcal{B}_{i-1}$ to $\mathcal{B}_{i-2}$, and so on. When moving elements from $\mathcal{B}_j$ to $\mathcal{B}_{j-1}$, both buckets

are scanned, in order to merge the moved elements into $\mathcal{B}_{j-1}$.

Meyer and Zeh prove the following two lemmas, where a BatchedDeleteMin operation is said to *empty* bucket $\mathcal{B}_i$ if it distributes the contents of $\mathcal{B}_i$ over buckets $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$.

LEMMA 2.2. (MEYER/ZEH [15]) *Let p be the minimum priority of the entries retrieved by a BatchedDeleteMin operation, let $i \geq 2$, and consider the sequence of all subsequent BatchedDeleteMin operations that empty buckets $\mathcal{B}_h$ with $h \geq i$. Let q_k be the minimum priority of the entries retrieved by the k'th operation in this sequence. Then $q_k - p \geq (k-4)2^{i-2}/3$.*

LEMMA 2.3. (MEYER/ZEH [15]) *The total cost of all priority queue operations performed by CO-SSSP is $O((m/B)\log W)$.*[2]

2.2 Hot pool. Conceptually, the hot pool consists of r sets $\mathcal{H}_1, \ldots, \mathcal{H}_r$, called *rows*, that are closely tied to the buckets in the priority queue. To define this connection, some terminology is required. We define the *category* of an edge e to be the integer i such that $2^{i-1} \leq \ell(v) < 2^i$. An *$i$-component*, then, is a connected component of the graph obtained from G by removing all edges of categories greater than i.

Our algorithm maintains the following invariant: *An adjacency list $A(v)$ is stored in the highest row $\mathcal{H}_i$ (row with highest index i) such that the $(i-1)$-component containing v does not contain a vertex z that has been visited already or is stored in a bucket $\mathcal{B}_j$ with $j < i$.* Intuitively, the goal is to store every adjacency list $A(v)$ in as high a row as possible because higher rows are inspected less frequently by our algorithm. It has to be guaranteed, however, that the row where $A(v)$ is stored is inspected again before v is visited. This is captured by the condition that no vertex in the $(i-1)$-component containing v is allowed to be visited or stored in a bucket $\mathcal{B}_j$ with $j < i$.

The invariant implies that, for every vertex $v \in \mathcal{B}_1$, $A(v) \subseteq \mathcal{H}_1$, unless v has already been visited. Thus, the edge set S in each iteration can be retrieved by querying $\mathcal{H}_1$, and vertices in L that have already been visited are easily detected (and ignored) because their adjacency lists are simply not found in $\mathcal{H}_1$.

In order to maintain the above invariant about where adjacency lists are stored, and in order to retrieve the adjacency lists of the vertices returned by a BatchedDeleteMin operation, every BatchedDeleteMin operation does the following when moving vertices from bucket $\mathcal{B}_i$ to bucket $\mathcal{B}_{i-1}$: Denote the set of moved vertices by L. For every vertex $v \in L$, the $(i-1)$-component containing v is retrieved from $\mathcal{H}_i$, if it is still stored in $\mathcal{H}_i$, and then inserted into $\mathcal{H}_{i-1}$. The implementation of the procedure to move $(i-1)$-components to $\mathcal{H}_{i-1}$ is discussed in detail in Section 4. When finally returning the contents of bucket $\mathcal{B}_1$, the BatchedDeleteMin operation retrieves the adjacency lists of all returned vertices from $\mathcal{H}_1$.

For BatchedDeleteMin operations to maintain the above invariant, it is necessary and sufficient that, if a vertex in L does not find its $(i-1)$-component in $\mathcal{H}_i$, this component is already stored in row $\mathcal{H}_{i-1}$ or below. The next lemma states that this is the case. We omit the proof because it is essentially the same as in [15].

LEMMA 2.4. *For a vertex $v \in \mathcal{B}_i$, the i-component containing v is stored in row $\mathcal{H}_i$ or below.*

The next two sections provide the details of the hot pool structure. Section 3 discusses the group partition required by our algorithm. Section 4 then discusses how to use this partition to efficiently maintain the hot pool structure.

3 Group Partition

The implementation of the hot pool requires a finer partition of the graph than the one into category components used in the high-level description of the algorithm. Inspired by the partition into groups of exponentially increasing diameter used in CO-BFS, we use a partition of the vertex set into (i,j)-groups, $1 \leq i \leq r = \lfloor \log W \rfloor + 1$, $0 \leq j \leq s = \lceil \log n \rceil + 2$; each such group is a subset of the vertices of an i-component and has diameter $O(2^{i+j})$, where the diameter of a vertex set is the maximal distance between any two vertices in the set.

More precisely, a *nested group partition* has the following properties, where we refer to an (i,j)-group as a $(\cdot,j)$-group or $(i,\cdot)$-group if we do not want to specify i or j.

(P1) Every $(i,\cdot)$-group is completely contained in an i-component.

(P2) Every i-component is an $(i+1,j)$-group and an (i,j')-group, for some j and $j' \leq j+3$. The i-component then contains no $(i+1,j'')$-groups with $j'' < j$ and no (i,j'')-group with $j'' > j'$.

(P3) For every pair (i,j) and any two distinct (i,j)-groups $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_1 \cap \mathcal{G}_2 = \emptyset$.

(P4) Every (i,j)-group is either equal to an i-component or completely contained in an $(i,j+1)$-group.

[2] Meyer and Zeh prove the lemma for MZ-SSSP, which performs the same priority queue operations as CO-SSSP.

(P5) Every (i,j)-group has diameter less than 2^{i+j-1}, where the diameter of a group is the maximal distance between any two of its vertices in G.[3]

(P6) For every $j > 0$, there are $O(n/2^j)$ $(\cdot, j)$-groups.

We say that an (i,j)-group is *trivial* if it consists of a single $(i-1)$-component or $(i, j-1)$-group; otherwise, it is *non-trivial*. Now observe that the nesting relationship defines a tree-like hierarchy of the groups whose root is the whole graph G and whose leaves are the vertices of G. We number the vertices in G in the order they are visited by a preorder traversal of this group tree. This implies that the vertices in every (i,j)-group are numbered consecutively. The *representation* of the partition consists of the adjacency lists $A(v)$ of all vertices, sorted by the IDs of vertices v; the adjacency list of vertex v is preceded by descriptors of all non-trivial groups in the partition that have v as their minimum vertex. These descriptors are sorted by decreasing numbers of vertices contained in their corresponding groups. Each descriptor describing a non-trivial (i,j)-group stores the pair (i,j), the number of edges in the adjacency lists of all vertices in its group, plus the number of smaller non-trivial groups nested in it. For every (i,j)-group, the corresponding *edge group* is the subsequence of the representation that starts with the group's descriptor and ends with the adjacency list of its last vertex. Since we store only descriptors of non-trivial groups, the total number of descriptors in each group is bounded by the number of edges in the group, so that the total length of this representation is still $O(m)$. The descriptors are used when splitting groups into smaller groups. Due to lack of space, we do not discuss this in detail here. In the full paper, we prove the following lemma, which is obtained using an extension of the clustering algorithm used in MZ-SSSP.

LEMMA 3.1. *A nested group partition of an undirected graph with n vertices and m edges can be computed in $O(\mathrm{MST}(n,m) + (n/B)(\log n + \log W))$ MT's.*

4 Hot Pools

The remainder of this paper is dedicated to describing the hot pool structure. The hot pool consists of a two-dimensional array of *buckets* $\mathcal{H}_{i,j}$, $1 \le i \le r = \lfloor \log W \rfloor + 1$ and $0 \le j \le s = \lceil \log n \rceil + 2$; bucket $\mathcal{H}_{i,j}$ stores (i,j)-groups. Initially, the whole graph is viewed as an (r, j^*)-group, for some j^*, which is stored in $\mathcal{H}_{r,j^*}$. We refer to buckets $\mathcal{H}_{i,0}, \ldots, \mathcal{H}_{i,s}$ as *row* i and to buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{r,j}$ as *column* j.

In order to use the hot pool in our shortest-path algorithm, we have to support the following *down-propagation operation*: When moving a list L of vertices from bucket $\mathcal{B}_i$ to bucket $\mathcal{B}_{i-1}$ in the priority queue, we have to move all $(i-1)$-components containing vertices in L from row i to row $i-1$. A special case of this is retrieving (from row 1) the adjacency lists of the vertices returned by a BatchedDeleteMin operation.

Conceptually, we do the following for every vertex $v \in L$. First we locate the (i,j)-group $\mathcal{G}$ in the i'th row that contains v.[4] If $\mathcal{G}$ is the $(i-1)$-component containing v, we move it to row $i-1$: $\mathcal{G}$ is also an $(i-1, j')$-group, for some $j' \le j+3$; so we insert it into $\mathcal{H}_{i-1,j'}$. If $\mathcal{G}$ is not an $(i-1)$-component, we split $\mathcal{G}$ into $(i, j-1)$-groups, which we insert into $\mathcal{H}_{i,j-1}$. We keep doing this until there are no more groups in row i containing vertices in L.

To implement this procedure efficiently, we need an appropriate layout of the hot pool buckets in memory, as well as a number of indexing structures to locate groups in the hot pool. The hot pool is represented by *column structures* $\mathcal{C}_0, \ldots, \mathcal{C}_r$, each storing the buckets of a column. These structures are discussed in detail in Section 4.2. For every row, we have a *row index* $\mathcal{I}_i$, which represents the groups stored in row i. In particular, a group $\mathcal{G}$ in $\mathcal{H}_{i,j}$ containing vertices with IDs between a and b and being the k'th group ever inserted into the hot pool is represented by a group descriptor (i, j, a, b, k) in $\mathcal{I}_i$. The descriptors in $\mathcal{I}_i$ are sorted by their a-components. Since the intervals of vertex IDs in the groups in row i are disjoint, $\mathcal{I}_i$ is thus a sorted list of disjoint vertex intervals.

The moving of groups between rows is achieved using a *down-propagation buffer* $\mathcal{D}$, which is a concatenation of buckets $\mathcal{D}_1, \ldots, \mathcal{D}_r$; bucket $\mathcal{D}_i$ stores $(i-1)$-components that are moving to row $i-1$ soon. In particular, each such component contains a vertex in $\mathcal{B}_i$.

The down-propagation of $(i-1)$-components containing vertices in L is now implemented as follows: First we locate all groups in row i that contain vertices in L. Since the vertices in L are given sorted by their IDs, a single scan of L and $\mathcal{I}_i$ suffices to retrieve the list P of descriptors of all groups in row i that contain vertices in L; we remove these descriptors from $\mathcal{I}_i$. We sort the descriptors in P by their j- and k-components. For each set of descriptors with the same j-component, we retrieve the corresponding groups from the appropriate

[3] Determining the exact group diameters efficiently is difficult. As in [15], our clustering algorithm uses an upper bound on the diameter that is easy to compute and tight enough to ensure that the graph can be partitioned into few groups with a small upper bound on their diameters. Details appear in the full paper.

[4] We do not distinguish between vertex groups and their corresponding edge groups. So we may talk about the containment of a vertex in an edge group or about an i-component (which is an (i,j)-group) being equal to an edge group.

column structure $\mathcal{C}_j$ and append them to an array $\mathcal{L}$. For each such group, we add a descriptor (i, j, a, b, p) to an index $\mathcal{J}$, where p is a pointer to the location of the group in $\mathcal{L}$.

We now iteratively perform the following procedure until L is empty: We discard all vertices in L that are not contained in any group in $\mathcal{L}$. For every (i, j)-group $\mathcal{G}$ in $\mathcal{L}$, if it does not contain a vertex in L, we insert it into bucket $\mathcal{H}_{i,j}$ and add a corresponding group descriptor to an index $\mathcal{I}'_i$ to be merged with $\mathcal{I}_i$ later. If $\mathcal{G}$ contains a vertex in L and it is an $(i-1)$-component, we identify the vertex in $\mathcal{G}$ with minimum priority (which must belong to L), store its priority with $\mathcal{G}$, and then insert $\mathcal{G}$ into $\mathcal{D}_i$. If $\mathcal{G}$ is not an $(i-1)$-component, we split it into its constituent $(i, j-1)$-groups,[5] which we add to a list $\mathcal{L}'$; for these groups, we add a group descriptor as those in $\mathcal{J}$ to an index $\mathcal{J}'$. Once all groups in $\mathcal{L}$ have been processed, $\mathcal{L}'$ and $\mathcal{J}'$ replace $\mathcal{L}$ and $\mathcal{J}$.

Each iteration of this loop can be implemented by sorting and scanning indexes $\mathcal{J}$ and $\mathcal{J}'$ and by scanning arrays L, $\mathcal{L}$, and $\mathcal{L}'$.

The actual down-propagation is now implemented by inspecting $\mathcal{D}_i$. A group $\mathcal{G}$ remains in $\mathcal{D}_i$ if its priority is at least p_{i-1}. If its priority is less than p_{i-1}, it is moved to row $i-1$. If $\mathcal{G}$ is an $(i-2)$-component, we simply insert it into $\mathcal{D}_{i-1}$. Otherwise, it is a non-trivial $(i-1, j)$-group, for some j. In this case, we insert $\mathcal{G}$ into bucket $\mathcal{H}_{i-1,j}$ and add a corresponding index entry to an index $\mathcal{I}'_{i-1}$. Once we have processed all groups in $\mathcal{D}_i$, we sort the group descriptors in $\mathcal{I}'_{i-1}$ and $\mathcal{I}'_i$ by their a-components and merge them into $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$, respectively.

Next we analyze the cost of manipulating lists L, $\mathcal{L}$, and $\mathcal{L}'$, as well as indexes $\mathcal{I}_i$, $\mathcal{I}'_i$, $\mathcal{J}$, and $\mathcal{J}'$. The cost of manipulating the down-propagation buffer is analyzed in Section 4.1, which also proves the correctness of our criterion for moving groups from $\mathcal{D}_i$ to row $i-1$. The cost of manipulating the column structures is analyzed in Section 4.2. The following two lemmas will be used throughout the analysis of the hot pool.

LEMMA 4.1. *Every edge is involved in at most* $\mathrm{O}(\log n + \log W)$ *insertions into buckets or arrays* $\mathcal{L}$ *and* $\mathcal{L}'$. *After it enters a bucket* $\mathcal{H}_{i,j}$, *it remains in the hot pool for* $\mathrm{O}\big(2^{i+j}\big)$ *distance steps, where a distance step is the increase of the minimum priority of the vertices in the priority queue by one.*

Proof sketch. Consider an edge e. Whenever edge e is inserted into array $\mathcal{L}$ or $\mathcal{L}'$ as part of an (i, j)-group, let us think of e as inserted into bucket $\mathcal{H}_{i,j}$. Then the number of insertions of e into buckets is easily seen to be as stated because the hot pool has $\lceil \log n \rceil + 2$ columns and $\lfloor \log W \rfloor + 1$ rows, and every edge, when it moves, moves left, down, left and down, or one down and at most three right, where we view the rows as numbered bottom to top and the columns as numbered left to right.

An edge stored in $\mathcal{H}_{i,j}$ belongs to an (i, j)-group $\mathcal{G}$. This group is stored in bucket $\mathcal{H}_{i,j}$ because the $(i, j+1)$-group $\mathcal{G}'$ containing $\mathcal{G}$ contains a visited vertex or a vertex that is visited within the next 2^i distance steps. Since the diameter of $\mathcal{G}'$ is less than 2^{i+j}, this implies that every vertex in $\mathcal{G}$ is visited within the next $\mathrm{O}\big(2^{i+j}\big)$ distance steps, which means that every edge in $\mathcal{G}$ disappears from the hot pool by this time. □

LEMMA 4.2. *The number of* $(\cdot, 0)$*-groups inserted into column 0 is* $\mathrm{O}(n)$.

Proof sketch. First note that there are only $\mathrm{O}(n)$ unique $(\cdot, 0)$-groups, as any two such groups are either disjoint or properly nested. Each such group is inserted into at most one bucket $\mathcal{H}_{i,0}$. To see this, observe that every $(i, 0)$-group is an $(i-1)$-component. Hence, when an $(i, 0)$-group $\mathcal{G}$ leaves bucket $\mathcal{H}_{i,0}$ it is inserted into $\mathcal{D}$; $\mathcal{G}$ does not leave $\mathcal{D}$ until it disappears completely from the hot pool structure or reaches a row i' where it is a non-trivial (i', j)-group with $j > 0$. □

The following lemma now follows immediately from Lemmas 2.2, 4.1, and 4.2 and Properties (P5) and (P6).

LEMMA 4.3. *Manipulating lists* L, $\mathcal{L}$, $\mathcal{L}'$ *and indexes* $\mathcal{I}_i$, $\mathcal{I}'_i$, $\mathcal{J}$, $\mathcal{J}'$ *costs* $\mathrm{O}((m/B)(\log n + \log W))$ *MT's.*

4.1 Down-propagation buffer. First we prove that our method for deciding when to move a group $\mathcal{G}$ in bucket $\mathcal{D}_i$ to a lower row is correct: precisely, that the priority stored with $\mathcal{G}$ is the minimum priority p of all vertices in $\mathcal{G}$, which implies that a vertex in $\mathcal{G}$ moves to row $i-1$ exactly when $p < p_{i-1}$.

LEMMA 4.4. *The minimum priority stored with each group in the down-propagation buffer* $\mathcal{D}$ *is correct at all times.*

Proof. When a group $\mathcal{G}$ moves into bucket $\mathcal{D}_i$ from a bucket $\mathcal{H}_{i,j}$, we label $\mathcal{G}$ correctly with the minimum priority of any vertex in $\mathcal{G}$ because this vertex must belong to L. Let p be the priority assigned to $\mathcal{G}$ at this point. We claim that no vertex in $\mathcal{G}$ can *ever* have a priority less than p, which implies that p remains the minimum priority of the vertices in $\mathcal{G}$ throughout $\mathcal{G}$'s life span. Let p^* be the minimum priority in L (and thus in

[5] Note that these groups may be trivial. In this case, we still do not move them to columns to the left of column $j-1$ unless they contain vertices in L.

$\mathcal{B}_i$) at the time when p is assigned to $\mathcal{G}$. If a subsequent relaxation of an edge uv changes the priority of a vertex $v \in \mathcal{G}$, then $d(u) \geq p^*$. If $u \in \mathcal{G}$, then $d(u) \geq p$, and $d(v) = d(u) + \ell(uv) \geq p$. So consider the case that $u \notin \mathcal{G}$. Since group $\mathcal{G}$ is an $(i-1)$-component, edge uv must have category at least i, that is, length at least 2^{i-1}. Thus, $d(u) + \ell(uv) \geq p^* + 2^{i-1}$. By the choice of priorities p_{i-1} and p_i, however, we have $p - p^* < p_i - p_{i-1} \leq 2^{i-2}$, that is, $p < d(u) + \ell(uv) = d(v)$. □

The implementation of the down-propagation buffer is fairly straightforward: We maintain $\mathcal{D}$ as the concatenation of buckets $\mathcal{D}_1, \ldots, \mathcal{D}_i$. Each such bucket is a collection of groups, in no particular order. Since $\mathcal{D}_i \neq \emptyset$ only if $\mathcal{B}_i \neq \emptyset$ and we inspect rows only up to the first non-empty bucket $\mathcal{B}_i$, we are always interested in only the first non-empty bucket in $\mathcal{D}$, if any. Hence, manipulating the current bucket $\mathcal{D}_i$ always reduces to scanning an appropriate prefix of $\mathcal{D}$.

LEMMA 4.5. *The cost of manipulating the down-propagation buffer is* $O((m/B) \log W)$.

Proof. Every group $\mathcal{G}$ in a bucket $\mathcal{D}_i$ contains a vertex v in $\mathcal{B}_i$. The next inspection of row i moves v to row $i-1$. Hence, by Lemma 4.4, the priority of $\mathcal{G}$ is less than p_{i-1}, and group $\mathcal{G}$ moves to row $i-1$. Thus, every edge is inspected at most once as part of a bucket $\mathcal{D}_i$. Summing this scanning cost over all $\lfloor \log W \rfloor + 1$ buckets in $\mathcal{D}$ gives the bound claimed in the lemma. □

4.2 Column structure. Each column of the hot pool is represented as a column structure consisting of an array $\mathcal{C}_j$ equipped with a number of indexes to support group insertions and extractions. This array consists of $r = \lfloor \log W \rfloor + 1$ chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,r}$, arranged in this order. Chunk $\mathcal{C}_{j,i}$ is associated with bucket $\mathcal{H}_{i,j}$ and stores only elements from buckets $\mathcal{H}_{i',j}$ with $i' \geq i$. Initially, all chunks have size 0 and are empty.

Every memory location in a chunk $\mathcal{C}_{j,i}$ can be in three different states: An *occupied* location stores an element of some bucket $\mathcal{H}_{i',j}$. A *clean* location does not store anything and never has since the time of the creation of $\mathcal{C}_{j,i}$. A location that is neither occupied nor clean is *unoccupied*.

We keep track of the chunks using a *chunk index*, which is an array of size r. The i'th entry stores the following information about chunk $\mathcal{C}_{j,i}$: the physical address $p_{\mathcal{C}_{j,i}}$ of $\mathcal{C}_{j,i}$ in $\mathcal{C}_j$, the size $s_{\mathcal{C}_{j,i}}$ of $\mathcal{C}_{j,i}$, the number $c_{\mathcal{C}_{j,i}}$ of clean memory locations in $\mathcal{C}_{j,i}$ (which are at the beginning of $\mathcal{C}_{j,i}$), and the number $|\mathcal{H}_{i,j}|$ of elements in bucket $\mathcal{H}_{i,j}$.

We keep track of groups stored in the chunks using a *group index*. Each index entry is a triple (i, k, p), which signifies that the corresponding group belongs to bucket $\mathcal{H}_{i,j}$, is the k'th group ever inserted into the hot pool, and is currently stored at position p in $\mathcal{C}_j$. The entries in the group index are sorted by increasing i-components and by decreasing k-components.

When inserting groups into column j in a down-propagation step, these groups are inserted into an *insertion buffer* $\mathcal{Z}_j$. At the end of each down-propagation step, we incorporate the groups in $\mathcal{Z}_j$ into $\mathcal{C}_j$.

Operations. To insert an (i, j)-group $\mathcal{G}$ into $\mathcal{C}_j$, we append $\mathcal{G}$ to $\mathcal{Z}_j$. We increment the insertion count, store this count with $\mathcal{G}$, and return a group descriptor (a, b, i, j, k) for insertion into the row index $\mathcal{I}_i$, where k is the current insertion count and $[a, b]$ is the range of vertex IDs in $\mathcal{G}$. We call this an InsertGroup operation.

At the end of each down-propagation step from row i to row $i-1$, we flush each non-empty insertion buffer $\mathcal{Z}_j$, inserting the groups in $\mathcal{Z}_j$ into $\mathcal{C}_j$. We call this a FlushGroups operation. Note that buffer $\mathcal{Z}_j$ can hold only groups to be inserted into $\mathcal{H}_{i-1,j}$ or $\mathcal{H}_{i,j}$ at this point. The FlushGroups operation scans $\mathcal{Z}_j$ twice. During the first scan, we insert the groups to be inserted into $\mathcal{H}_{i,j}$. The second scan inserts the remaining groups into $\mathcal{H}_{i-1,j}$. Consider the first scan, the second one being similar. To insert groups into $\mathcal{H}_{i,j}$, we increase $|\mathcal{H}_{i,j}|$ by the total size of these groups and scan the chunk index to find the chunk $\mathcal{C}_{j,x}$ with maximal index $x \leq i$ that has clean positions. Let h be the number of edges to be inserted into $\mathcal{H}_{i,j}$. If $h \leq c_{\mathcal{C}_{j,x}}$, we insert these edges into positions $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - h$ through $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - 1$ and then decrease $c_{\mathcal{C}_{j,x}}$ by h. If $h > c_{\mathcal{C}_{j,x}}$, we insert $c_{\mathcal{C}_{j,x}}$ elements into positions $p_{\mathcal{C}_{j,x}}$ through $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - 1$, set $c_{\mathcal{C}_{j,x}} = 0$, find the next lower chunk $\mathcal{C}_{j,x'}$ that has clean positions and repeat the whole procedure for the remaining $h - c_{\mathcal{C}_{j,x}}$ elements in the group. As a result, a group may be distributed over more than one chunk. We refer to the part of a group stored in a chunk as a *group segment*. For every group, we form a linked list of its segments by storing with every group segment a pointer to the next segment of the same group, by increasing address.

Finally, we need to update the group index. When we insert a new group $\mathcal{G}$ with number k into $\mathcal{H}_{i,j}$, let p be the position of the first segment of $\mathcal{G}$ in $\mathcal{C}_j$. Then we add a triple (i, k, p) to a list A. Once the processing of all insertions into $\mathcal{H}_{i,j}$ is done, we merge the contents of list A into the group index.

When inserting groups of in total h elements into $\mathcal{H}_{i,j}$, it is possible that $\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} < h$, that is, there is not enough room for inserting the groups into $\mathcal{C}_j$. In this case, we temporarily grow the first chunk $\mathcal{C}_{j,1}$ and increase $c_{\mathcal{C}_{j,1}}$ accordingly to obtain $\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} = h$. Then we perform the insertion as above, leaving

$\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} = 0$. This is followed by rebuilding a prefix of chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i'}$ with $i' \geq i$, as described below.

At the beginning of each down-propagation step from row i to row $i-1$, we extract from $\mathcal{C}_j$ some of the groups in bucket $\mathcal{H}_{i,j}$. We call this an ExtractGroups operation, which is given the descriptors of all groups to be extracted, sorted by decreasing k-values. This operation scans the list of group descriptors and the group index to translate every tuple (a, b, i, j, k) into a tuple (a, b, i, j, p), where p is the physical position of the requested group. Then we scan this pointer list, retrieve each group indexed by a tuple in this list, and add its elements to a list L. In particular, for each group, its corresponding group index entry points to its first segment. We retrieve this segment and then follow pointers between the segments to traverse the list of group segments and thereby collect all the elements in the group. Note that every ExtractGroups operation retrieves groups from exactly one bucket $\mathcal{H}_{i,j}$, in the order in which they are stored in $\mathcal{C}_j$. So no sorting is required. Finally, we decrease $|\mathcal{H}_{i,j}|$ by the total size of the extracted groups, and we return list L.

The removal of the elements in L from $\mathcal{C}_j$ may leave a prefix of $\mathcal{C}_j$ too sparsely populated. Thus, before returning the elements in L, we check whether there exists an index i such that $\sum_{x=1}^{i} |\mathcal{H}_{x,j}| < \frac{1}{4} \sum_{x=1}^{i} s_{\mathcal{C}_{j,x}}$. If so, we rebuild some prefix $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i'}$ with $i' \geq i$.

Rebuilding. When a FlushGroups or ExtractGroups operation triggers the rebuilding of a prefix of $\mathcal{C}_j$, we first identify the set of chunks that need to be rebuilt. We find the maximal index i' such that $\sum_{x=1}^{i'} |\mathcal{H}_{x,j}| < \frac{1}{4} \sum_{x=1}^{i'} s_{\mathcal{C}_{j,x}}$ and $|\mathcal{H}_{i',j}| < \frac{1}{2} s_{\mathcal{C}_{j,i'}}$. (After rebuilding chunk $\mathcal{C}_{j,i'}$, we have $|\mathcal{H}_{i',j}| = \frac{1}{2} s_{\mathcal{C}_{j,i'}}$. Hence, the second condition implies that bucket $\mathcal{H}_{i',j}$ has lost at least one element. Without this condition, Observation 4.1 below would not hold.) If there is no such index, we set $i' = 0$. Next we find the maximal index $i > i'$ satisfying the following two conditions: (i) Bucket $\mathcal{H}_{i,j}$ is non-empty or $s_{\mathcal{C}_{j,i}} > 0$ and (ii) $c_{\mathcal{C}_{j,x}} = 0$, for all $i' < x \leq i$. If there is no such index i, we choose $i = i'$. Now we rebuild chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$.

Note that, by the choice of index i, the groups in chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ belong to buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$ because $c_{\mathcal{C}_{j,i+1}} > 0$ and any insertion into a bucket $\mathcal{H}_{x,j}$ with $x > i$ would have completely filled $\mathcal{C}_{j,i+1}$ before inserting elements into one of $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$. We perform the following operations for $i' = 1, \ldots, i$: We scan the part of the group index storing pointers to the groups in $\mathcal{H}_{i',j}$, retrieve the corresponding groups from $\mathcal{C}_j$, and append them to a list L. Once we have done this, L stores all groups in buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$, sorted in the same order as their corresponding group index entries; and the elements of each group are stored consecutively, that is, none of these groups is segmented.

We now scan backwards over L, creating chunks $\mathcal{C}_{j,i}, \mathcal{C}_{j,i-1}, \ldots, \mathcal{C}_{j,1}$. For each chunk $\mathcal{C}_{j,x}$, we set $|\mathcal{H}_{x,j}|$ to be the number of elements in $\mathcal{H}_{x,j}$, $c_{\mathcal{C}_{j,x}} = |\mathcal{H}_{x,j}|$ and $s_{\mathcal{C}_{j,x}} = 2|\mathcal{H}_{x,j}|$. Then we allocate $s_{\mathcal{C}_{j,x}}$ memory locations to $\mathcal{C}_{j,x}$ and store the elements in $\mathcal{H}_{x,j}$ in the highest $|\mathcal{H}_{x,j}|$ memory locations of $\mathcal{C}_{j,x}$. Since groups are stored in the same order in L as their corresponding group index entries, and we place the groups in the same order into the new chunks, we can also update the pointers of the corresponding group index entries in a single scan of the corresponding prefix of the group index.

It is easy to prove the following two lemmas. The first one shows that no prefix of $\mathcal{C}_j$ is ever too sparsely populated (except immediately before rebuilding it). The second one establishes that, whenever we rebuild a prefix, we have enough insertions or deletions into or from $\mathcal{C}_j$ that can pay for the rebuilding cost.

LEMMA 4.6. *For all $1 \leq i \leq r$, we have $\sum_{x=1}^{i} s_{\mathcal{C}_{j,x}} \leq 4 \sum_{x=1}^{i} |\mathcal{H}_{i,j}|$.*

LEMMA 4.7. *For every chunk $\mathcal{C}_{j,x}$, let $u_{\mathcal{C}_{j,x}}$ be the number of elements inserted into or deleted from $\mathcal{C}_{j,x}$ since the last time this chunk was rebuilt. When a prefix $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ is rebuilt, we have $\sum_{x=1}^{i} u_{\mathcal{C}_{j,x}} > \frac{1}{4} \sum_{x=1}^{i} s_{\mathcal{C}_{j,x}}$.*

Analysis. It remains to analyze the costs of the different operations. In this analysis, we make use of our tall-cache assumption by assuming that, for some parameter $t \leq \log B$ to be chosen later and all $0 \leq j \leq t$, we can keep the following four blocks of the column structure $\mathcal{C}_j$ in cache: the first blocks of the chunk and group indexes, the first block of array $\mathcal{C}_j$, and the last block of the buffer array $\mathcal{Z}_j$. This requires $\mathrm{O}(B \log B) = \mathrm{O}\big(B^{1+\varepsilon}\big)$ cache space.

First let us analyze the cost of maintaining the chunk indexes. We assume that every chunk index access scans the whole index. Then the cost of scanning chunk indexes is $\mathrm{O}((n/B) \log W)$: the size of each chunk index is $\lfloor \log W \rfloor + 1$; the chunk index is scanned at most once per FlushGroups, ExtractGroups, or rebuilding operation, of which there are $\mathrm{O}(n)$, by Property (P6) and Lemma 4.2. Chunk index accesses for columns $j > t$ may incur another $\mathrm{O}(n/2^t)$ MT's because, when accessing a chunk of a column $j > t$, we may have to load the first block of the chunk index into cache. By Property (P6), this happens at most $\mathrm{O}(n/2^t)$ times. Before bounding the cost of accessing group indexes, we need the following observation.

OBSERVATION 4.1. *Between every two accesses to a chunk $\mathcal{C}_{j,i}$ in array $\mathcal{C}_j$, there is at least one access to a bucket $\mathcal{H}_{i',j}$ with $i' \geq i$.*

When accessing a group index, it is to access the entries of (i,j)-groups in one bucket $\mathcal{H}_{i,j}$ or to access the entries of all groups in chunks $\mathcal{C}_{j,1},\ldots,\mathcal{C}_{j,i}$ during a rebuild operation. In both cases, we bound the cost of accessing the group index by scanning the prefix of the group index up to the last group in $\mathcal{C}_{j,i}$. By Observation 4.1 and Lemmas 2.2 and 4.1, every group index entry in the j'th column is scanned $\mathrm{O}\big(2^j\big)$ times. By Property (P6) and Lemma 4.2, there are $\mathrm{O}\big(n/2^j\big)$ groups in column j. Hence, the scanning cost of group index entries in a single column is $\mathrm{O}(n/B)$. Summing over all columns, the scanning cost is $\mathrm{O}((n/B)\log n)$. Every access to a group index of a column $j > t$ may cost one extra MT. Since there are $\mathrm{O}(n/2^t)$ accesses to columns $j > t$, this amounts to an extra cost of $\mathrm{O}(n/2^t)$ MT's.

The accesses to insertion buffers cost $\mathrm{O}\big(n/2^t + (m/B)(\log n + \log W)\big)$: The first term is the cost of loading the last block of buffer $\mathcal{Z}_j$ if $j > t$. The second term is the scanning cost because every edge is scanned $\mathrm{O}(1)$ times per insertion into a column, and an edge is inserted into $\mathrm{O}(\log n + \log W)$ buckets, by Lemma 4.1.

The final part of the analysis concerns the cost of accessing arrays $\mathcal{C}_j$. We distinguish the two cases $j \leq t$ and $j > t$. For $j \leq t$, we can bound the cost of every access to chunk $\mathcal{C}_{j,i}$, or chunks $\mathcal{C}_{j,1},\ldots,\mathcal{C}_{j,i}$ if rebuilding is required, by the cost of a *staggered scan* of these chunks, defined as follows: We first scan chunk $\mathcal{C}_{j,1}$, then chunks $\mathcal{C}_{j,1},\mathcal{C}_{j,2}$ (scanning $\mathcal{C}_{j,1}$ again!), then chunks $\mathcal{C}_{j,1},\mathcal{C}_{j,2},\mathcal{C}_{j,3}$, and so on until we finally scan $\mathcal{C}_{j,1},\ldots,\mathcal{C}_{j,i}$. The following two lemmas analyze the costs of accessing the two different kinds of columns.

LEMMA 4.8. *The total cost of manipulating arrays $\mathcal{C}_j$, $0 \leq j \leq t$, is $\mathrm{O}((m2^t/B)\log W)$ MT's.*

Proof. By Lemma 4.6, every staggered scan reads at most a constant factor more memory locations than there are elements in the corresponding buckets. Thus, for this analysis we can think of every chunk $\mathcal{C}_{j,i}$ as being the same as bucket $\mathcal{H}_{i,j}$. We prove that every edge e is accessed at most $\mathrm{O}(2^t \log W)$ times. Summing over all edges proves the lemma. In this proof, we consider edge e to be in row i if it is stored in a bucket $\mathcal{H}_{i,j}$ with $j \leq t$. In particular, we do not consider it to be in row i if it is contained in a bucket $\mathcal{H}_{i,j}$ with $j > t$. This captures that we are interested only in the cost edge e incurs in the first t columns.

We now consider priorities $q_0 \geq q_1 \geq \cdots \geq q_r$, which are defined as follows: q_0 is the minimum priority of any vertex retrieved by the BatchedDeleteMin operation that also retrieves edge e. For $i > 0$, we define $q_i = q_{i-1}$ if edge e never enters row i (in the above sense); otherwise, let q_i be the minimum priority of any vertex retrieved by the BatchedDeleteMin operation that puts e into row i.

Now let $a_{i,i'}$ be the number of accesses to a row $i' \geq i$ while edge e is in row i, let $a^*_{i,i'}$ be the number of accesses to rows i' and above while edge e is in row i, and let $A_{i'}$ be the number of accesses to rows i' and above while edge e is in row i' or below. The total number of accesses to edge e in the first t rows is at most $\sum_{i=1}^{r}\sum_{i'\geq i}(i'-i+1)a_{i,i'} = \sum_{i=1}^{r}\sum_{i'\geq i} a^*_{i,i'} = \sum_{i'=1}^{r}\sum_{i\leq i'} a^*_{i,i'} = \sum_{i'=1}^{r} A_{i'}$. By Lemma 2.2, we have $A_{i'} \leq 4 + 12(q_0 - q_{i'})/2^{i'}$. By Lemma 4.1, we have $q_0 - q_{i'} = \mathrm{O}\Big(2^{i'+t}\Big)$. Thus, $A_{i'} = \mathrm{O}(1+2^t) = \mathrm{O}(2^t)$. Inserting into the above summation, we obtain that edge e is scanned at most $\mathrm{O}(2^t \log W)$ times as part of columns 1 through t. This implies the lemma. □

LEMMA 4.9. *The total cost of manipulating arrays $\mathcal{C}_j$, $j > t$, is $\mathrm{O}(n/2^t + (m/B)(\log n + \log W))$.*

Proof. The first term in the bound accounts for the cost of having to access the first segment of each group; by Property (P6), there are at most $\mathrm{O}(n/2^t)$ groups in columns $j > t$. The second term accounts for the cost of scanning edges while inserting or extracting them; the bound follows because, by Lemma 4.1, every edge is involved in at most $\mathrm{O}(\log n + \log W)$ insertions or extractions. Note that, by Lemma 4.7, we can charge the cost of scanning edges during rebuilding to the inserted or deleted edges that triggered the rebuilding.

What remains is to bound the number of memory transfers incurred by accessing groups that have more than one segment, that is, are distributed over multiple chunks. We call following the pointer from one group segment to the next a *jump*. A jump is *short* if it is from a chunk $\mathcal{C}_{j,x}$ to a chunk $\mathcal{C}_{j,z}$ with $z > x$ and $s_{\mathcal{C}_{j,y}} = 0$ for all $x < y < z$; otherwise, it is *long*.

The cost of every long jump is at most one MT. Consider a group $\mathcal{G}$. For a long jump from a chunk $\mathcal{C}_{j,x}$ to a chunk $\mathcal{C}_{j,z}$, there must be a chunk $\mathcal{C}_{j,y}$, $x < y < z$, with $s_{\mathcal{C}_{j,y}} > 0$. We choose y maximally so. Since the elements of $\mathcal{G}$ were not inserted into $\mathcal{C}_{j,y}$ at the time of $\mathcal{G}$'s insertion, we must have had $c_{\mathcal{C}_{j,y}} = 0$ at that time. However, since $s_{\mathcal{C}_{j,y}} > 0$, we must have had $c_{\mathcal{C}_{j,y}} > 0$ immediately after $\mathcal{C}_{j,y}$ was rebuilt. Hence, there must have been an insertion into $\mathcal{C}_{j,y}$ between the rebuilding of $\mathcal{C}_{j,y}$ and the insertion of $\mathcal{G}$. We charge the first group $\mathcal{G}'$ inserted into $\mathcal{C}_{j,y}$ after the last time $\mathcal{C}_{j,y}$ was rebuilt for the cost of the long jump. Observe that $\mathcal{G}'$ can be charged only once: It can obviously be charged only until the next time $\mathcal{C}_{j,y}$ is rebuilt; and before rebuilding

$\mathcal{C}_{j,y}$ again, there can be only one group $\mathcal{G}$ that jumps over chunk $\mathcal{C}_{j,y}$ and charges $\mathcal{G}'$.

The cost of a short jump is bounded by the cost of scanning the whole chunk $\mathcal{C}_{j,x}$ from which it originates. Since there can be at most one jump out of $\mathcal{C}_{j,x}$ before $\mathcal{C}_{j,x}$ is rebuilt again, we can charge this cost to the update operations that cause the next rebuilding of $\mathcal{C}_{j,x}$, increasing the scanning cost of $\mathcal{C}_{j,x}$ incurred while rebuilding chunks by a constant factor. If $\mathcal{C}_{j,x}$ is never rebuilt again after this short jump, we bound the cost of the short jump by one MT and charge any of the groups in $\mathcal{C}_{j,x}$ at the time it was rebuilt last for this jump. This charges every group at most once. □

By summing the bounds in Lemmas 4.3, 4.5, 4.8, and 4.9 and choosing $t = \left\lceil \log \sqrt{nB/(m \log W)} \right\rceil \leq \lceil \log B \rceil$, we obtain the following corollary, which together with Lemmas 2.1, 2.3, and 3.1 implies Theorem 1.1.

COROLLARY 4.1. *The total cost of hot pool manipulations is* $\mathrm{O}\Big(\sqrt{(nm \log W)/B} + (m/B) \log n\Big)$.

References

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

[2] L. Arge, M. A. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[3] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[4] G. S. Brodal and R. Fagerberg. Funnel heap—a cache oblivious priority queue. In *Proceedings of 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2002.

[5] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, 2003.

[6] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer-Verlag, 2004.

[7] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[8] R. A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–254, 2004.

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

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[11] H. Jampala and N. Zeh. Cache-oblivious planar shortest paths. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 563–575. Springer-Verlag, 2005.

[12] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55:3–23, 1997.

[13] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, October 1996.

[14] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

[15] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th Annual European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 434–445. Springer-Verlag, 2003.

[16] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the 14th Annual European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 540–551. Springer-Verlag, 2006.

[17] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, January 1999.

[18] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 267–276, 2002.

[19] M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[20] M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35:189–201, 2000.