# I/O-Efficient Undirected Shortest Paths with Unbounded Weights

**Ulrich Meyer**
**Norbert Zeh**

Technical Report CS-2006-04

June 15, 2006

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

# I/O-Efficient Undirected Shortest Paths with Unbounded Weights

Ulrich Meyer[1,⋆] and Norbert Zeh[2,⋆⋆]

[1] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken, 66123, Germany.
Email: `umeyer@mpi-sb.mpg.de`

[2] Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Canada.
Email: `nzeh@cs.dal.ca`

**Abstract.** We present an algorithm for computing single-source shortest paths in undirected graphs with non-negative edge weights in $O(\sqrt{nm/B}\log n + MST(n,m))$ I/Os, where $n$ is the number of vertices, $m$ is the number of edges, $B$ is the disk block size, and $MST(n,m)$ is the I/O-cost of computing a minimum spanning tree. Our algorithm is based on our previous algorithm for graphs with bounded edge weights. Our contribution is the removal of the algorithm's dependence on the edge weights.

## 1 Introduction

The *singlesource shortestpath* (SSSP) problem is a fundamental combinatorial optimization problem with numerous applications. 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 weights to the edges of $G$. The SSSP problem is to find, for every vertex $v \in V$, the distance, $\mathrm{dist}_G(s,v)$, from $s$ to $v$, that is, the weight of a minimum-weight (shortest) path from $s$ to $v$ in $G$.

In this paper, it will be convenient to consider the following problem, which we call the *closest-source shortestpath* (CSSP) problem: In addition to the input discussed above, let $w : V \to \mathbb{R}^+$ be an assignment of non-negative *source weights* to the vertices of $G$. Then compute for every vertex $x \in G$, its distance $D(x) = \min(w(y) + \mathrm{dist}_G(y,x) \mid y \in G\}$. If $y$ is a vertex such that $w(y) + \mathrm{dist}_G(y,x) = D(x)$, a *shortest path to* $x$, denoted $\pi(x)$, is a path of length $\mathrm{dist}_G(y,x)$ from $y$ to $x$. It is easy to transform an SSSP instance into an equivalent CSSP instance and vice versa.

The classical SSSP-algorithm is Dijkstra's algorithm [6], which has seen many improvements, culminating in linear-time or almost linear-time algorithms for planar graphs [7], undirected graphs with integer or float edge weights [14, 15], and undirected graphs with real edge weights [13]. Unfortunately, all of these algorithms perform poorly when applied to massive graphs that do not fit into memory and are stored on disk. The reason is that they access the data in a random fashion, which is in spite of the fact that the algorithms of [13–15] are similar in spirit to our algorithm.

Much previous work has focused on solving SSSP on massive graphs. These algorithms are analyzed in the I/O-model [1], where it is assumed that the computer has a main memory that can hold $M$ vertices or edges and that the graph is stored on disk. In order to process the graph, pieces of it have to be loaded into memory, which happens in blocks of $B$ consecutive data items. Such a transfer is referred to as an *I/Ooperation* (I/O). The complexity of an algorithm is the number of I/O-operations it performs. The equivalent of the internal-memory time bound of $O(n \log n)$ required to sort $n$ numbers is $\mathrm{sort}(n) = O((n/B)\log_{M/B}(n/B))$ I/Os [1].

While the I/O-bound of $O((n + m/B)\log n)$ achievable by Dijkstra's algorithm (when implemented using the buffered repository tree of [4]) is essentially still the best bound for SSSP on

⋆ Research supported by DFG grant ME 2088/1-3.
⋆⋆ Research supported by NSERC and CFI.

general *directed* graphs, much progress has been made on *undirected* graphs [8, 11, 12] and on special graph classes [2, 9, 10]. The algorithm of Kumar and Schwabe (*KSSSSP* ) [8] performs $O(n + (m/B)\log(n/B))$ I/Os. For dense graphs, the second term dominates; but for sparse graphs, the I/O-bound becomes $O(n)$. The algorithm of [11] improves on this bound for the special case of breadth-first search (BFS). The SSSP-algorithm by Meyer and Zeh (*MZSSSP* ) [12] extends the ideas of [11] to graphs with edge weights between 1 and $W$, leading to an $O(\sqrt{nm\log W/B} + MST(n,m))$ bound, where $MST(n,m)$ is the cost of computing a minimum spanning tree of a graph with $n$ vertices and $m$ edges.[1] In this paper, we extend MZ-SSSP significantly to remove the algorithm's dependence on $W$, leading to the following result:

**Theorem 1.** *The SSSP problem in an undirected graph with n vertices, m edges, and nonnegative* [2] *edge weights can be solved in* $O(\sqrt{nm/B}\log n + MST(n,m))$ *I/Os.*

For sparse graphs, the cost of our algorithm is $O((n/\sqrt{B})\log n)$. The algorithm is fairly complicated and, for realistic values of $n$ and $B$, unlikely to outperform KS-SSSP. It demonstrates, however, that it is in general possible to avoid performing one I/O per vertex, without regard for the edge weights, and, we believe, provides further insight into the I/O-complexity of the SSSP-problem.

We start the presentation in Section 2 with a discussion of the central ideas of KS-SSSP and MZ-SSSP, which will be reused in our algorithm. Section 3 shows how to augment MZ-SSSP to make it independent of the edge weights, provided that the graph can be partitioned into appropriate subgraphs. The complexity of the algorithm, instead of depending on $\log W$, will depend on a parameter of the partition, called its depth. Section 4 discusses a recursive shortest-path algorithm that uses another partition of the graph into "well-separated" subgraphs, which allows the computation of shortest paths in the whole graph using nearly independent shortest-path computations on these subgraphs. Section 5 then argues that any graph can be partitioned into such well-separated subgraphs such that each has a partition of small depth in the sense of Section 3. This allows us to combine the two approaches from Sections 3 and 4 to obtain an efficient CSSP-algorithm. Section 6 puts the bits and pieces together and analyzes the complexity of the final algorithm.

## 2 Previous Work: KS-SSSP and MZ-SSSP

*KSSSSP.* KS-SSSP [8] is an I/O-efficient version of Dijkstra's algorithm. It uses a priority queue $Q$ to maintain the tentative distances of all vertices. It retrieves the vertices one by one from $Q$, by increasing tentative distance. After retrieving a vertex $x$, it is *visited*, that is, its incident edges $xy$ are *relaxed*, where the relaxation of an edge $xy$ replaces the tentative distance $d(y)$ of $y$ with $\min(d(y), d(x) + \ell(xy))$. This update is reflected by updating the priority of $y$ in $Q$.

The main contribution of KS-SSSP is the development of an I/O-efficient priority queue that supports operations Update$(x, p)$, Delete$(x)$, and DeleteMin, each in $O((1/B)\log(n/B))$ I/Os amortized. The latter two behave as on any priority queue, deleting $x$ or the item with minimum priority, respectively. The former replaces $x$'s current priority $p_x$ with $\min(p_x, p)$ if $x$ is in the priority queue. If $x$ is not in the priority queue and has never been in it, it is inserted with priority $p$. If $x$ has been in the priority queue before, but has been deleted, the operation does nothing. This particular behaviour

[1] The current bounds for $MST(n,m)$ are $O(\mathrm{sort}(m)\log\log(nB/m))$ deterministically [2] and $O(\mathrm{sort}(m))$ randomized [5].

[2] In the discussion, it is assumed that edge weights are strictly positive. Weight-0 edges can be handled by treating each connected component of the subgraph they induce as a single vertex.

of the Update operation is supported only for undirected shortest-path computations, by recording updates vertices perform on each other's distances and using this information to prevent re-inserted vertices from being retrieved for a second time using a DeleteMin operation. See [8] for details.

This method to ensure the peculiar behaviour of the Update operation relies on the vertices being visited by increasing distance. This is not true for our algorithm, but we will argue in the full paper that, when we retrieve a vertex twice from the priority queue, due to a re-insertion, we still visit it only once, due to the delayed edge relaxations employed by our algorithm. Thus, we assume in the remainder of the paper that we have a priority queue whose Update operation behaves as above.

Given this fairly powerful priority queue, visiting a vertex $x$ reduces to scanning its adjacency list $E(x)$ and performing an Update$(y, d(x) + \ell(xy))$ operation on $Q$ for every edge $xy \in E(x)$. Thus, KS-SSSP performs $O(m)$ priority queue operations, which cost $O((m/B)\log(n/B))$ I/Os, and it spends $O(1 + \deg(x)/B)$ I/Os to retrieve the adjacency list of each vertex $x$, $O(n + m/B)$ I/Os in total. This leads to the claimed I/O-complexity. The bottleneck of KS-SSSP is thus the random accesses to the adjacency lists. This is the problem addressed by MZ-SSSP and by our new algorithm.

*MZSSSP.* For the case of BFS—that is, SSSP with unit edge weights—the bottleneck created by random accesses to adjacency lists has been addressed in [11] using a clustering-based approach. MZ-SSSP extends this approach to allow real edge weights between 1 and $W$. The basic idea can be described as follows: First, the vertex set of $G$ is partitioned into $q = O(n/\mu)$ carefully chosen sets $V_1, \dots, V_q$, called *vertex clusters*; $1 \le \mu \le \sqrt{B}$ is a parameter specified later. For each vertex cluster $V_i$, the adjacency lists of the vertices in $V_i$ are concatenated to form an *edge cluster* $E_i$. The edges in each edge cluster $E_i$ are stored consecutively on disk.

The shortest-path computation now proceeds as in KS-SSSP, except that a *hot pool* $\mathcal{H}$ acts as an intermediary between the priority queue and the adjacency lists. In particular, when a vertex $x$ is retrieved from $Q$, it is only *released*, which means that the hot pool $\mathcal{H}$ is instructed to visit $x$. $\mathcal{H}$ may delay visiting $x$, but not too long, as formalized by the following property:

(SP) Every vertex $x$ is visited before a vertex $y$ with $D(y) > D(x) + \text{dist}_G(x,y)/2$ is visited.

It is straightforward to prove that any SSSP/CSSP-algorithm that has this property is correct.

The hot pool $\mathcal{H}$ is a buffer space storing adjacency lists. When a vertex $x$ needs to be visited and $E(x)$ is in $\mathcal{H}$, all edges in $E(x)$ are relaxed. If $E(x)$ is not in $\mathcal{H}$, then the entire edge cluster containing $E(x)$ is loaded into $\mathcal{H}$ before $x$ is visited. This ensures that only $O(n/\mu + m/B)$ I/Os are performed to load edges into the hot pool, because every edge cluster is loaded only once. The difficult part is looking for adjacency lists in $\mathcal{H}$ efficiently, which can be done in $O(\mu \log W/B)$ I/Os amortized per edge, provided that the cluster partition has certain properties, discussed in the next section. A partition with these properties can be computed in $O(MST(n,m) + (n/B)\log W)$ I/Os. By using a priority queue that exploits the bounded range of edge weights to support Update, Delete, and DeleteMin operations in $O((1/B)\log W)$ I/Os amortized, the complexity of the algorithm thus becomes $O(n/\mu + (m\mu \log W)/B + MST(n,m))$, which is $O(\sqrt{nm \log W/B} + MST(n,m))$ if we choose $\mu = \sqrt{nB/(m\log W)}$.

## 3 A Weight-Independent MZ-SSSP

In this section, we review MZ-SSSP and present a new implementation, which we call MZ-SSSP$^*$. The cost of MZ-SSSP$^*$ is $O((n/\mu)\log n + m(\mu d + \log n)/B)$ I/Os, where $d$ is a parameter of the used cluster partition, called the *depth* of the partition. This parameter will always be bounded by $\log W$. In Section 5, we will be concerned with ensuring that $d = O(\log n)$, which, after choosing $\mu = \sqrt{nB/m}$, leads to the desired complexity of $O(\sqrt{nm/B}\log n)$ I/Os, plus the cost for computing the partition, which will be $O(MST(n,m))$. In our discussion, we will emphasize whenever MZ-SSSP$^*$ deviates significantly from MZ-SSSP.

### 3.1 $\mu$-Partitions

The efficient implementation of the hot pool in MZ-SSSP$^*$ requires that the used cluster partition has a number of properties. To define these, we need some notation. For an edge $e \in G$, the *category* of $e$ is the integer $c$ such that $2^{c-1} \leq \ell(e) < 2^c$. A *c-component* of $G$ is a maximal connected subgraph of $G$ all of whose edges have category $c$ or less. A *category component* is a $c$-component, for some $c$. A *c-cluster* is a *vertex cluster* $V_i$ that is contained in a $c$-component, but not in a $(c-1)$-component. We also refer to the corresponding edge cluster $E_i$ as a $c$-cluster. The *diameter* $\operatorname{diam}(V')$ of a vertex set $V'$ is equal to $\max\{\operatorname{dist}_G(x,y) \mid x,y \in V'\}$. Now the partition required by MZ-SSSP$^*$ is a *μ-partition* of $G$, which is a partition of $V$ into $q = O(n/\mu)$ vertex clusters $V_1,\ldots,V_q$ with the following properties:

(C1) Every cluster $V_i$ contains at most $\mu$ vertices,
(C2) Every $c$-cluster $V_i$ has diameter at most $\mu 2^c$,
(C3) No $(c-1)$-component contributes vertices to two $c$-clusters, and
(C4) Every $c$-component contributing a vertex to a $c'$-cluster with $c' > c$ has diameter at most $2^c\mu$.

The first property was not required by MZ-SSSP. We define the *depth* of a cluster as the difference between the category of the cluster and the category of the shortest edge with exactly one endpoint in the cluster. The depth of the partition is the maximal depth of its clusters. A *$(\mu,d)$-partition* is a $\mu$-partition of depth $d$. Note that $d \leq \log W$.

Even though every edge with exactly one endpoint in a $c$-cluster of a $(\mu,d)$-partition has category at least $c-d$, edges between vertices in the same $c$-cluster may be arbitrarily short. We define a *mini-cluster* to be a $(c-d)$-component contained in a $c$-cluster. (Note that, in a $(\mu,d)$-partition, any $(c-d)$-component is either completely contained in or disjoint from a given $c$-cluster.) The shortest-path computation will have to deal with non-trivial mini-clusters in a special way to ensure correctness of the algorithm. In MZ-SSSP, all mini-clusters were trivial because it only used the upper bound of $\log W$ on the depth of the partition; so this complication did not arise.

In order to facilitate the efficient retrieval of edges from the hot pool, the shortest-path computation requires information about which vertices of which cluster are contained in which category component. This information is provided by a *cluster tree* $T_i$ associated with each cluster $V_i$. To define these cluster trees, let us define the *component tree* $T_c$ of $G$ first: We say that a $c$-component is *unique* if it is properly contained in a $(c+1)$-component or it is equal to $G$; in addition, all vertices of $G$ are unique 0-components. The vertex set of $T_c$ consists of all unique category components. Component $C$ is the parent of component $C'$ if $C' \subset C$ and $C \subseteq C''$ for all $C'' \supset C'$. Now the cluster tree $T_i$ of a $c$-cluster $V_i$ is the subtree of $T_c$ containing all nodes of category $c$ or less that are ancestors of vertices in $V_i$ (which are leaves of $T_c$).

### 3.2 Shortest Paths

The shortest-path computation proceeds in iterations; each iteration releases a vertex $x$ from the priority queue $Q$ and inserts a $\mathrm{Visit}(x,d(x))$ signal into the hot pool to induce the relaxation of all edges incident of $x$. Before releasing $x$ from $Q$, however, a Scan operation is invoked on the hot pool to ensure that all edges that need to be relaxed before releasing the next vertex are relaxed. This operation is described in the next section, which discusses the implementation of the hot pool.

## 3.3 Hot Pool

*The structure.* The hot pool consists of a hierarchy of $r = \lceil \log W \rceil$ *edge buffers* $\mathrm{EB}_1, \ldots, \mathrm{EB}_r$, a hierarchy of *r tree buffers* $\mathrm{TB}_1, \ldots, \mathrm{TB}_r$, and a hierarchy of *r signal buffers* $\mathrm{SB}_1, \ldots, \mathrm{SB}_r$. Each of these hierarchies is implemented as a single stack with markers indicating the boundaries between individual buffers. We refer to the triple $(\mathrm{EB}_i, \mathrm{TB}_i, \mathrm{SB}_i)$ as *level i*.

The edge buffers hold edges that have been loaded into the hot pool. Edge buffer $\mathrm{EB}_{i+1}$ will be inspected by Scan operations about half as often as $\mathrm{EB}_i$. If an edge $xy$ is stored in $\mathrm{EB}_i$, then $\mathrm{TB}_i$ stores all ancestors of $x$ in $T_c$ that have category at most $i$. The signal buffers store signals used to trigger edge relaxations and movements of edges between different edge buffers. The purpose of moving edges between different edge buffers is to initially store edges in buffers that are inspected infrequently and later, when the time of their relaxation approaches, move them to buffers that are inspected more frequently, in order to avoid delaying their relaxation for too long.

The inspection of edge buffers is controlled by *due times* $t_1 \le t_2 \le \cdots \le t_{r+1} = +\infty$ associated with these buffers. These due times satisfy the following condition:

(DT) For $1 \le i < r$, $t_{i+1} = t_i$ or $2^{i-3} \le t_{i+1} - t_i \le 2^{i-2}$.

Initially, we set $t_i = \min\{w(x) \mid x \in G\} + 2^{i-2}$, for $1 \le i \le r$. Due time $t_i$ indicates that we have to inspect $\mathrm{EB}_i$ for edges to be relaxed or moved to lower buffers before the first vertex with tentative distance $d(x) \ge t_i$ is released from $Q$. We will maintain the following invariant:

(HP) After loading a $c$-edge cluster $E_j$ into the hot pool, an edge $xy$ is stored in the lowest edge buffer $\mathrm{EB}_i$ such that $i \ge c - d$ and the $i$-component $C$ containing vertex $x$ satisfies $d(C) < t_{i+1}$.

The tree buffers are used to check this condition. In particular, component $C$ is stored as a node of $T_i$ in the tree buffer $\mathrm{TB}_i$, and we ensure that this copy of $C$ in $\mathrm{TB}_i$ always stores the correct value of $d(C) = \min\{d(x) \mid x \in C\}$.[3] To achieve this, we insert an $\mathrm{Update}(y, d(x) + \ell(xy))$ signal into an appropriate signal buffer whenever an edge $xy$ is relaxed; this signal updates the tentative distance of every ancestor of $y$ in $T_c$ that is stored in a tree buffer to which this signal is applied.

As discussed in the previous subsection, the shortest-path algorithm inserts $\mathrm{Visit}(x, d(x))$ signals into the hot pool to trigger the relaxation of edges incident to $x$. In MZ-SSSP, these signals are inserted into $\mathrm{SB}_1$ and then move through all signal buffers as higher and higher levels are scanned. Thus, each signal can visit $\log W$ levels. In order to reduce the number of levels through which a signal travels, we insert a $\mathrm{Visit}(x, d(x))$ signal into $\mathrm{SB}_{c_x}$, where $c_x = c - d$ if $x$ is contained in a $c$-cluster, and let it travel only up to level $c + O(\log n)$. For now, let us say that we *send* this signal to level $c_x$. We discuss how to do this efficiently after discussing the Scan operation on the hot pool.

*Scanning the hot pool.* The Scan operation scans a prefix $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ of edge buffers for edges that need to be relaxed or moved to other levels. Let $f$ be the minimum priority of the vertices in $Q$, called the *frontier*. (Note that $Q$ can easily maintain this value.) Since $f$ is the priority of the next vertex to

[3] This is not quite correct; $C$ will store only an upper bound $d^*(C)$ on $d(C)$; but this upper bound will be good enough to move edges to lower buffers when their relaxation approaches.

be released from $Q$, we need to scan all edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ such that $t_1 \leq \cdots \leq t_j \leq f < t_{j+1}$. The scanning of an edge buffer $\mathrm{EB}_i$ may decrease $f$. Then we use the updated value of $f$ to decide whether to continue the scan with $\mathrm{EB}_{i+1}$.

The Scan operation can be divided into two separate phases: The *upphase* inspects edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$, relaxes edges, and moves edges whose relaxation is not imminent to higher levels. The *downphase* inspects $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ in reverse order, assigns new due times to $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$, and moves edges to lower levels if the maintenance of property (HP) requires it. These two phases perform the following operations on each inspected level $i$:

**Up-phase:**

- Insert all signals sent to level $i$ since the last scan of this level into $\mathrm{SB}_i$.
- For every Visit$(x, d(x))$ signal in $\mathrm{SB}_i$ such that $x \notin \mathrm{TB}_i$, load the edge cluster $E_h$ containing $E(x)$ into $\mathrm{EB}_i$; load the corresponding cluster tree $T_h$ into $\mathrm{TB}_i$ and retrieve the tentative distances of all nodes in $T_h$ from a distance repository REP, which is discussed in Section 3.4. If there are Visit$(x, d(x))$ signals in $\mathrm{SB}_i$ for more than one vertex in $V_h$, $E_h$ is loaded only once.
- For every cluster tree node $C$ in $\mathrm{TB}_i$ and every Update$(x, d)$ signal in $\mathrm{SB}_i$ such that $x \in C$, replace $d(C)$ with $\min(d(C), d)$.
- For every Visit$(x, d(x))$ signal in $\mathrm{SB}_i$, process the mini-cluster containing $x$. The details are explained below. For every category-$c$ edge $xy$ with $c \geq i$ relaxed during this process, send an Update$(y, d(x) + \ell(xy))$ signal to level $\max(i+1, c - \log n - d - 1)$ and, if $c \leq i + \log n + d + 1$, to level $i$. We say that such an Update signal has category $c$.
- Move all cluster tree nodes $C$ in $\mathrm{TB}_i$ to $\mathrm{TB}_{i+1}$ such that either the category of $C$ is greater than $i$ or the tentative distance of the $i$-component containing $C$ is at least $t_{i+1}$. For every cluster tree leaf (vertex) $x$ moved to $\mathrm{TB}_{i+1}$, move $E(x)$ to $\mathrm{EB}_{i+1}$.
- Move update signals with category greater than $i - d$ to $\mathrm{SB}_{i+1}$. Discard all other signals in $\mathrm{SB}_i$.
- Test whether $f \leq t_{i+1}$ and, if so, continue to level $i+1$.

**Down-phase:**

- Update the due time $t_i$: If $f + 2^{i-1} \geq t_{i+1}$, then $t_i = t_{i+1}$. Otherwise, let $t_i = (t_{i+1} + f)/2$. It is easy to check that this maintains Property (DT).
- Insert all signals sent to level $i$ into $\mathrm{SB}_i$. At this point, they will all be Update signals sent during scans of higher levels. As in the up-phase, apply these Update signals to the nodes stored in $\mathrm{TB}_i$.
- Move all cluster tree nodes $C$ in $\mathrm{TB}_i$ to $\mathrm{TB}_{i-1}$ such that the category of $C$ is less than $i$ and the $(i-1)$-component containing $C$ has tentative distance less than $t_i$. Discard all cluster tree nodes of category $i$. For every cluster tree leaf $x$ moved to $\mathrm{TB}_{i-1}$, move $E(x)$ to $\mathrm{EB}_{i-1}$.
- Move all signals of category less than $i + \log n + d + 1$ to $\mathrm{SB}_{i-1}$. Discard all other signals in $\mathrm{SB}_i$.

In order to implement these different steps efficiently, we number the nodes in $T_c$ in preorder. Then we keep the cluster tree nodes sorted by their preorder numbers; we keep the signals in $\mathrm{SB}_i$ sorted by the preorder numbers of the vertices they affect; and we keep the edges in $\mathrm{EB}_i$ sorted by the preorder numbers of their first endpoints. Given this ordering, it is easy to show that all the above steps can be implemented by scanning the involved buffers a constant number of times, except that the signals inserted into $\mathrm{SB}_i$ have to be sorted before merging them into $\mathrm{SB}_i$.

We also argue in the full paper that the due times of empty levels can be represented implicitly using the due times of the two closest non-empty levels. This is necessary to avoid spending I/Os on accessing due times of empty levels. Accesses to due times of non-empty levels can be amortized over the accesses to the elements in these levels.

*Processing miniclusters.* The processing of a mini-cluster $C$ involves visiting all vertices in $C$ that have Visit$(x, d(x))$ signals in SB$_i$. Since the vertices in the mini-cluster are connected by potentially very short edges, it may also be necessary to immediately visit vertices in the same mini-cluster whose tentative distances decrease dramatically as a result of the relaxation of these edges. In particular, starting with their current tentative distances, we run a bounded version of Dijkstra's algorithm on the mini-cluster. This can be done in internal memory because the mini-cluster has at most $\mu \leq \sqrt{B}$ vertices and, thus, at most $B$ edges. When Dijkstra's algorithm is about to visit a vertex $x$, the vertex is visited if $d(x) \leq t_i$. Otherwise, the algorithm terminates. Once Dijkstra's algorithm terminates, it updates the tentative distances of all vertices in the mini-cluster that have not been visited, that is, for each such vertex, Update$(x, d(x))$ operations are performed on $Q$ and the distance repository, and $d(x)$ is updated in TB$_i$. For every visited vertex $x$ and every category-$c$ edge $xy$ in $E(x)$ with $c \geq i$, Update$(y, d(x) + \ell(xy))$ signals are sent to the levels specified in the discussion of the up-phase. Finally, a Delete$(x)$ operation is performed on $Q$ for every visited vertex $x$. This is necessary to ensure that $x$ is not visited again because, during the processing of the mini-cluster, we may visit vertices that have not been released from $Q$ yet.

*Sending signals to levels.* The sending of signals to specific levels in the hierarchy is easily accomplished using two priority queues SQ$^+$ and SQ$^-$. A Visit signal to be sent to level $i$ is inserted into SQ$^+$ with priority $i$. The next time level $i$ is inspected, this signal is retrieved from SQ$^+$ and inserted into SB$_i$. Since levels $1, \ldots, i-1$ are inspected before level $i$, the minimum priority in SQ$^+$ is $i$ at this point, so that these signals can be retrieved using DeleteMin operations. Update signals sent to other levels when visiting level $i$ are inserted into SQ$^+$ if the target level is greater than $i$, and into SQ$^-$ if the target level is at most $i$. In the down phase, signals sent to each level are retrieved from SQ$^-$ using DeleteMax operations.

## 3.4 Distance Repository

MZ-SSSP$^*$ needs a *distance repository* REP, which stores the tentative distances of all vertices in $G$ and allows us to retrieve the tentative distances of all nodes in a cluster tree $T_i$. In particular, the repository supports Update$(x, d(x))$ and ClusterQuery$(T_i)$ operations. MZ-SSSP does not need such a repository because of the way the priority queue in MZ-SSSP is implemented and coupled with the hot pool. Since we use the priority queue of KS-SSSP instead, to avoid the $O((1/B)\log W)$ cost per Update operation on the priority queue in MZ-SSSP, we need the repository to query the tentative distances of all nodes in a cluster tree when its cluster is loaded into the hot pool. The relaxation of an edge $xy$, in addition to performing an Update$(y, d(x) + \ell(xy))$ operation on $Q$, now also performs such an operation on REP.

## 3.5 Analysis

What remains is to prove the correctness of MZ-SSSP$^*$ and analyze its I/O-complexity. The proof will rely on the following lemma, which is proved in Appendix A. In this lemma, $d(x)$ and $d^*(x)$ denote the tentative distances of $x$ stored in REP and with the cluster tree node $x$ in a tree buffer, respectively. It is possible that $d(x) < d^*(x)$ because the cluster tree node may not be informed about all updates of $d(x)$. We will argue, however, that this does not affect the correctness of the algorithm.

**Lemma 1.** *A vertex $x$ visited during a scan of level $i$ satisfies $t_i - 2^{i-2} \leq d(x) \leq d^*(x) \leq t_i$.*

The correctness of the algorithm now follows from the following two lemmas.

**Lemma 2.** *If the vertices on $\pi(x)$ are visited by increasing distance, then $d^*(x) = d(x) = D(x)$ when $x$ is visited.*

*Proof.* $d(x) = D(x)$ follows by induction along $\pi(x)$. Now let $y$ be the predecessor of $x$ on $\pi(x)$, which is visited before $x$. If $x$'s cluster is loaded after $y$ is visited, we have $d^*(x) = d(x)$ because $d(x)$ is read from REP at this point. So assume that $y$ is visited after $x$ has been loaded into the hot pool. Note that, at any time, $x$ is stored somewhere between levels $c_x$ and $c_x + d$. Hence, it suffices to prove that either $x$ and $y$ belong to the same mini-cluster or the Update$(x, D(x))$ signal issued by $y$ has category between $c_x$ and $c_x + d + \log n + 1$. Indeed, in the former case, the update of $d(x)$ to $D(x)$ is immediately recorded in $\mathrm{TB}_{c_x}$. In the latter case, the Update signal is applied to all levels between $c_x$ and $c_x + d$, that is, $x$ sees this update.

So assume that $y$ and $x$ do not belong to the same mini-cluster. Then edge $yx$ has category at least $c_x$, which proves the lower bound on the category of the update. If edge $yx$ had category greater than $c_x + d + \log n + 1$, then $\ell(yx) \geq n \cdot 2^{c_x+d+1} + 2^{c_y-1}$. On the other hand, since $x$ is loaded before this update happens, a vertex $z$ in the same $(c_x + d)$-component as $x$ is visited before $y$. This implies that $d(z) \leq D(y) + 2^{c_y-2}$. Since $x$ and $z$ belong to the same $(c_x + d)$-component, we have $D(x) < d(z) + n \cdot 2^{c_x+d} \leq D(y) + 2^{c_y-2} + 2^{c_x+d} < D(y) + \ell(yx)$, a contradiction. □

**Lemma 3.** *Let $y$ be $x$'s predecessor on $\pi(x)$. If $d(y) = D(y)$ when $y$ is visited, then $y$ is visited before $x$.*

*Proof.* Assume that $x$ is visited before $y$, and let $I_x$ and $I_y$ be the two invocations of procedure Scan during which $x$ and $y$ are visited. If $I_x = I_y$ and $x$ and $y$ belong to the same mini-cluster, $y$ must be visited before $x$ because $D(y) < D(x) \leq d(x)$. So $x$ and $y$ belong to different mini-clusters. Then $c_x \leq c_y$. Hence, $d(x) \leq t_{c_x} \leq t_{c_y} \leq D(y) + 2^{c_y-2} < D(y) + \ell(yx)$, a contradiction.

If $I_x$ precedes $I_y$, then we have $d(x) \leq t_{c_x} \leq f_x$, where $f_x$ is the value of $f$ at the end of invocation $I_x$. At the end of $I_x$, $t_{c_y} \geq f_x$; see Lemma 8 in Appendix A. Hence, we have $D(y) \geq t_{c_y} - 2^{c_y-2} > t_{c_y} - \ell(yx) \geq d(x) - \ell(yx)$, again a contradiction. □

**Lemma 4.** *MZSSSP $^*$ is correct and has property (SP).*

*Proof.* Correctness is immediate from Lemmas 2 and 3. To prove property (SP), assume that there exist two vertices $x$ and $y$ such that $D(y) > D(x) + \mathrm{dist}_G(x,y)/2$, but $y$ is visited before $x$. Vertices $x$ and $y$ cannot be in the same mini-cluster because it is easily verified that all vertices in a mini-cluster are visited by increasing distance. Thus, $\mathrm{dist}_G(x,y) \geq 2^{\max(c_x,c_y)-1} \geq 2^{c_x-1}$. Now consider the two invocations $I_x$ and $I_y$ that visit vertices $x$ and $y$.

If $I_x = I_y$, then we must have $c_y \leq c_x$ because $y$ is visited before $x$. Since $x$ and $y$ are visited in this invocation, we have $D(y) \leq t_{c_y} \leq t_{c_x} \leq D(x) + 2^{c_x-2} \leq D(x) + \mathrm{dist}_G(x,y)/2$, a contradiction.

If $I_y$ precedes $I_x$, let $f_y$ be the value of the frontier at the end of invocation $I_y$. Then $D(y) \leq t_{c_y} \leq f_y \leq t_{c_x} \leq D(x) + 2^{c_x-2} \leq D(x) + \mathrm{dist}_G(x,y)/2$, again a contradiction. □

**Lemma 5.** *Excluding the cost of computing the $(\mu, d)$-partition, the I/O-complexity of MZSSSP $^*$ is $O((n/\mu)\log n + m(\mu d + \log n)/B)$.*

*Proof sketch.* First observe that we perform $O(m)$ priority queue operations and Update operations on REP. The cost per Update on the repository is $O((1/B)\log n)$ amortized, as discussed in Section 6, which gives a cost of $O((m/B)\log n)$ for these operations. We also send only $O(m)$ signals to the different levels of the hot pool, which costs $O(\mathrm{sort}(m))$ I/Os for the involved operations on $\mathrm{SQ}^+$ and $\mathrm{SQ}^-$, as well as sorting these signals before insertion into the signal buffers.

The remainder of the complexity analysis hinges on two claims: (1) Every cluster is loaded into the hot pool only once. This would result in a cost of $O(n/\mu + m/B)$ for reading edge clusters and tree buffers, plus $O((n/\mu)\log n + n/B)$ for answering cluster queries on REP (see Section 6 for the cost of these queries). (2) Every signal traverses at most $d + \log n + 2$ levels in the hot pool; every edge and cluster tree node traverses at most $d$ levels in the hierarchy, remaining at each level for only $O(\mu)$ scans of this level. This would imply a cost of $O((m/B)(d + \log n + 2))$ for scanning the signals and $O(md\mu/B)$ for scanning edges and cluster tree nodes. Summing up the different costs, we obtain the claimed cost of the algorithm.

The number of levels traversed by each edge or signal is easily seen to be as claimed. The number of scans of a level during which an edge remains at a given level follows from properties (C2) and (C4) and the fact that $t_i$ increases by at least $2^{i-3}$ every time level $i$ is scanned, which is easy to prove. So we have to prove that we do not load a cluster more than once. To prove this, it suffices to show that, after loading a cluster containing a vertex $x$ into the hot pool, a $\mathrm{Visit}(x, d(x))$ signal finds $x$ at level $c_x$. This follows from Lemma 10 in Appendix A. $\square$

## 4 A Recursive Shortest-Path Algorithm

In this section, we describe a CSSP-algorithm that uses in a sense the exact opposite of a $\mu$-partition of low depth. We start by defining the partition required by the algorithm and then argue that shortest paths in the whole graph can be computed by solving nearly independent CSSP-problems on the graphs in the partition. We only prove the correctness of the algorithm here. We analyze its complexity in Section 6, where we combine it with MZ-SSSP* to obtain our final algorithm.

### 4.1 Barrier Decomposition

Our algorithm uses a *barrier decomposition* of $G$, which consists of a number of multigraphs $G_0, \dots, G_q$ and vertex sets $\emptyset = B_0, \dots, B_q$, called *barriers*, with the following properties:

(B1) Every graph $G_i$ represents a connected vertex-induced subgraph $H_i$ of $G$; $H_0 = G$.

(B2) For $i < j$, $H_i \cap H_j = \emptyset$ or $H_j \subset H_i$. If $H_j \subset H_i$ and $H_i \subseteq H_k$ for all $H_k \supset H_j$, we call $G_i$ the *parent* of $G_j$ (and $G_j$ a *child* of $G_i$).

(B3) For all $i$, graph $G_i$ is obtained from $H_i$ by contracting each graph $H_j$ such that $G_j$ is a child of $G_i$ into a single vertex $r(G_j)$, which we call the *representative* of $G_j$. For a vertex $x \in H_j$, we consider $r(G_j)$ the representative of $x$ in $G_i$ and denote it by $r_x$. For $x \in G_i$, let $r_x = x$.

(B4) For a given graph $G_j$ with parent $G_i$, $B_j$ is the set of vertices in $(V(H_i) \cup B_i) \setminus V(H_j)$ that are reachable from $H_j$ using edges of length at most $2n\ell_{\max}(H_j)$, where $\ell_{\max}(H_j)$ is the length of the longest edge in $H_j$.

(B5) No set $B_i$ contains a graph representative.

Intuitively, for every graph $G_j$, the set $B_j$ forms a barrier between $H_j$ and the rest of $G$ in the sense that a shortest-path between two vertices in $H_j$ cannot contain a vertex not in $V(H_j) \cup B_j$.

## 4.2 The Algorithm

Assume we are given a Dijkstra-like CSSP algorithm $\mathcal{A}$, that is, an algorithm that visits every vertex exactly once and, when it does, relaxes all edges incident to $x$. Assume also that algorithm $\mathcal{A}$ has property (SP). Given a barrier decomposition of $G$, we can then solve the CSSP problem on $G$ recursively. The resulting algorithm will require the use of the distance repository REP, augmented to support a GraphQuery$(G_i)$ operation, which returns the tentative distances of all vertices in $G_i$; for a graph representative $x = r(G_j)$, let $d(x) = \min\{d(y) \mid y \in H_j\}$. The algorithms looks as follows:

**ShortestPaths**$(G_i)$**:** Run a modified version of algorithm $\mathcal{A}$ on the graph graph $G_i \cup B_i$ obtained from $G[V(H_i) \cup B_i]$ by contracting each graph $H_j$ such that $G_j$ is a child of $G_i$ into a single vertex $r(G_j)$. The modifications are as follows:

- Terminate $\mathcal{A}$ as soon as all vertices in $G_i$ have been visited. In particular, it is not required to visit all vertices in $B_i$.
- When $\mathcal{A}$ visits a vertex $x$ that is not a graph representative, relax all its incident edges. In particular, for such an edge $xr_y$, where $r_y$ may or may not be a graph representative, replace $d(y)$ with $\min(d(y), d(x) + \ell(xy))$ in REP and $d(r_y)$ with $\min(d(r_y), d(x) + \ell(xy))$ in $\mathcal{A}$'s data structures.
- When $\mathcal{A}$ visits a graph representative $r(G_j)$:
  - Recursively invoke ShortestPaths$(G_j)$ with the source weights of all vertices in $V(G_j) \cup B_j$ initialized to their current tentative distances. (These distances are retrieved from REP.)
  - If the recursive call visits vertices in $B_j$, reflect this in the data structures of the current invocation to ensure that these vertices are not visited again. (E.g., if $\mathcal{A}$ is Dijkstra's algorithm or MZ-SSSP*, remove these vertices from the priority queue.)
  - If the recursive call updates the tentative distances of vertices in $B_j$, reflect this in the data structures of the current invocation. (E.g., if $\mathcal{A}$ is MZ-SSSP*, update their priorities in the priority queue and send corresponding Update signals to the hot pool.)
  - Relax all edges with exactly one endpoint in $H_j$, that is, for each edge $xy$ such that $x \in H_j$ and $y \in H_i \setminus H_j$, replace $d(r_y)$ with $\min(d(r_y), d(x) + \ell(xy))$.

The initial invocation is on graph $G_0$, which ensures that all vertices in $G$ are visited. The following lemma shows that this solves the CSSP problem.

**Lemma 6.** *For every vertex $x \in H_i \cup B_i$ visited by ShortestPaths$(G_i)$, we have $d(x) = D(x)$ at the time when $x$ is visited.*

*Proof.* The proof is by induction on the number of descendants of $G_i$. If there is none, the algorithm behaves like $\mathcal{A}$ and the claim follows because, by (SP), all vertices on $\pi(x)$ are visited in order.

So assume that $G_i$ has at least one child $G_j$, that there exists a vertex $x \in H_i \cup B_i$ such that $d(x) > D(x)$ when $x$ is visited, and that every vertex $z$ preceding $x$ on $\pi(x)$ satisfies $d(z) = D(z)$ when it is visited. First assume that $x$ is not visited in a recursive call Shortest-Path$(G_j)$, where $G_j$ is a child of $G_i$. Let $y$ be $x$'s predecessor on $\pi(x)$, and let $r_y$ be its representative in $G_i$. $r_y$ must be visited after $x$ because otherwise $d(x) = D(x)$ when $x$ is visited. Hence, by (SP), $D(y) \geq D(r_y) \geq D(x) - \mathrm{dist}_{G_i \cup B_i}(r_y, x)/2 \geq D(x) - \mathrm{dist}_G(y, x)/2$, a contradiction because $y \in \pi(x)$.

Now assume that $x$ is visited during a recursive call ShortestPaths$(G_j)$. Then the claim follows by induction if we can prove that $D_{H_j \cup B_j}(x) = D(x)$. So assume the contrary. If $\pi(x) \subseteq H_j \cup B_j$, this is impossible. So $\pi(x)$ contains at least one vertex outside $H_j \cup B_j$. Let $z$ be the last such vertex on $\pi(x)$, and let $y$ be its successor on $\pi(x)$, which is in $H_j \cup B_j$. We need to prove that $w(y) = D(y)$.

Assume the contrary. Then $r = r(G_i)$ must be visited before $r_z$, that is, by (SP), $D(r) \le D(r_z) + \mathrm{dist}_{G_i \cup B_i}(r_z, r)/2$. However, we have $D(u) < D(r) + n \cdot \ell_{\max}(H_j) \le D(r) + \mathrm{dist}_{G_i \cup B_i}(r_z, r)/2$, for all $u \in H_j$, because $z \notin B_j$. Hence, $D(u) < D(r_z) + \mathrm{dist}_{G_i \cup B_i}(r_z, r) \le D(z) + \mathrm{dist}_G(z, u)$. Thus, $z$ cannot belong to $\pi(u)$, for any $u \in H_j$, and $x \notin H_j$. Then, however, $x$ is visited only if $D_{H_j \cup B_j}(x) \le D_{H_j \cup B_j}(u)$, for some $u \in H_j$. Since $D(x) \le D_{H_j \cup B_j}(x)$ and $D_{H_i \cup B_i}(u) = D(u)$, this implies again that $z \notin \pi(x)$, a contradiction. □

## 5 Computing the Partition

Our final algorithm is based on the recursive framework of Section 4 and uses MZ-SSSP* or, on small graphs, Dijkstra's algorithm to compute shortest paths on the different graphs in the barrier decomposition. To achieve the desired I/O-complexity, we need the following properties of the barrier decomposition of $G$.

(P1) The barrier decomposition consists of $O(n/\mu)$ multigraphs $G_0, \ldots, G_q$.
(P2) Each graph $G_i$ has at most $\sqrt{B}$ vertices or is equipped with a $(\mu, \log n + 2)$-partition. In the former case, we call it *atomic*; in the latter *compound*.
(P3) If the parent $G_i$ of $G_j$ is atomic, then $G_j$ is $G_i$'s only child. If the parent $G_i$ of a graph $G_j$ is compound, then $B_j$ is a subset of a vertex cluster of $G_i$, and this vertex cluster contains only one graph representative. This implies in particular, that $|B_j| \le \mu \le \sqrt{B}$, for all $j$.

Next we sketch the construction of such a partition, essentially proving its existence. Due to space limitations, we omit the proof that the construction can be carried out in $O(MST(n,m))$ I/Os. Details are provided in Appendix B. The construction consists of three phases: The first two compute a $\mu$-partition of $G$ whose deep clusters have a particularly simple structure. In the third phase, every deep cluster is split into up to three parts, two of which are shallow; the third defines an atomic graph that separates the two graphs containing the two shallow parts.

To obtain a $\mu$-partition, we compute $T_c$ from a minimum spanning tree of $G$ using a variation of an algorithm from [3]. Then we process $T_c$ bottom-up. For every $c$-component $C$, if it contains more than $\mu$ vertices not assigned to clusters yet or has diameter greater than $2^c\mu$, we group (the unassigned vertices of) its children to form clusters of diameter at most $2^c\mu$ and containing at most $\mu$ vertices. This phase essentially simulates the construction in [12], but does so in $O(\mathrm{sort}(n))$ I/Os using the information provided by $T_c$.

In the second phase, we call a cluster $V_i$ *deep* if it has depth greater than $\log n + 2$. We call a node $C$ in $T_c$ *internal*, *mixed*, or *external* depending on whether all, some, or none of the vertices in $C$ belong to $V_i$. Our goal is to ensure that the mixed nodes in each cluster tree $T_i$ form a top-down path and that the only mixed node in $T_i$ with external children is the bottom-most node of the path. This can be achieved by processing $T_i$ bottom-up. When reaching a mixed node $C$ in $T_i$ with at least one mixed child and at least two non-internal children in $T_c$, we remove the subtrees of $T_i$ rooted at mixed children of $C$ from $T_i$ and form a new cluster containing the leaves of each of these trees. It is easy to verify that all deep clusters that remain after this processing have the desired structure and that the construction at most triples the number of clusters.

The final partition is now obtained as follows: For the cluster tree $T_i$ of each deep cluster $V_i$, let $C_i^1$ be the highest mixed node in $T_i$ that has category no greater than $c - \log n - 2$, where $c$ is the category of $V_i$; let $C_i^2$ be the lowest mixed node in $T_i$. Let $\mathcal{C}$ be the set of all these nodes $C_i^1$ and $C_i^2$ for all deep clusters in the partition. Then each node $C_i^h$ in $\mathcal{C}$ defines a subgraph $H_j$ consisting of all descendant leaves of $C_i^h$ in $T_c$. Consequently, the vertex set of each graph $G_j$ consists of the leaves of one of the subtrees of $T_c$ obtained by splitting each node $C_i^h$ into a top and a bottom copy and making all children of $C_i^h$ children of the bottom copy.

This procedure splits every deep cluster $V_i$ into up to three parts. The parts corresponding to the top and bottom parts of $T_i$ are easily verified to have depth at most $\log n + 2$. The middle part defines an atomic graph. Note that every graph's barrier is contained in a cluster of the original $\mu$-partition and, thus, has size at most $\mu$. It is also easily verified that every graph $G_i$ satisfies Property (P3). The number of graphs is at most twice the number of clusters in the $\mu$-partition produced by the second phase, which is $O(n/\mu)$. Hence, we obtain

**Lemma 7.** *It takes $O(MST(n,m))$ I/Os to compute a barrier decomposition of an undirected graph $G$ that has properties (P1)–(P3).*

## 6 Complexity of the Final Algorithm

By Lemma 7, it takes $O(MST(n,m))$ I/Os to compute the desired decomposition of the graph. Using MZ-SSSP* to solve CSSP in a compound graph $G_i$ in the computed barrier decomposition takes $O(((n_i + |B_i|)/\mu)\log n + (m_i\mu\log n)/B)$ I/Os, where $n_i$ is the number of vertices in $G_i$ and $m_i$ is the number of edges in $G_i$. if graph $G_i$ is atomic, we use Dijkstra's algorithm to solve CSSP in $G_i$, which incurs $O(1 + m_i/B)$ I/Os: Load all vertices and, for every pair of vertices, the shortest edge between these two vertices into memory. Then run Dijkstra's algorithm. Before visiting the graph representative $r(G_j)$ in $G_i$, relax *all* edges incident to visited vertices to ensure that all vertices in $H_j$ know their correct tentative distances.

It is easy to see that $\sum_{i=1}^{q}(n_i + |B_i|) = O(n)$ and $\sum_{i=1}^{q} m_i = O(m)$. Hence, the cost of all CSSP-computations on graphs $G_i$ is $O((n/\mu)\log n + (m\mu\log n)/B) = O(\sqrt{nm/B}\log n)$, for $\mu = \sqrt{nB/m}$.

The distance repository can be implemented as an augmented buffered repository tree [4] (see Appendix C), which supports the required operations in the following I/O-bounds: Update$(x,d)$ operations take $O((1/B)\log n)$ I/Os amortized. A ClusterQuery$(V_j)$ operation takes $O((1+r_j)\log n + |T_j|/B)$ I/Os, where $r_j$ is the number of cluster tree roots that belong to or are adjacent to $T_j$. A GraphQuery$(G_i)$ takes $O((1+c_i)\log n + |V(G_i)|/B)$ I/Os, where $c_i$ is the number of children of $G_i$. Now observe that our algorithm performs exactly one subgraph query per graph $G_i$ and at most two cluster queries per cluster: once when the cluster is loaded into the hot pool and a second time when the graph representative $r(G_j)$ in the cluster is visited, namely to retrieve the tentative distances of all vertices in $B_j$. Moreover, it is easy to show that the sum of the $r_j$ and $c_i$ is $O(n/\mu)$, so that the cost of all queries on the repository is $O((n/\mu)\log n + n/B) = O((n/\mu)\log n)$. Since we perform only $O(m)$ edge relaxations, the cost of all Update operations is $O((m/B)\log n)$.

The final issue is the relaxation of edges between vertices in $H_j$ and $H_i \setminus H_j$ whenever the recursive call on a child $G_j$ of the current graph $G_i$ returns. This is easily accomplished as follows: For every edge $xy$, we store the index of the smallest graph $H_i$ such that $x,y \in H_i$. When vertex $x$ is visited during the invocation of ShortestPaths on a descendant $G_j$ of $G_i$, we insert an Update$(y, d(x) + \ell(xy))$ signal into a priority queue XQ, with priority $i$. When the recursive call later returns to $G_i$, we retrieve all Update signals with priority $i$ from XQ and update the tentative distances of all representatives $r_y$ accordingly. This adds an extra cost of $O(\text{sort}(m))$ for all edges.

Summing the costs of all parts of the algorithm, we obtain the I/O-bound claimed in Theorem 1.

## 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, 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.
3. L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.
4. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACMSIAM Symposium on Discrete Algorithms* , pages 859–860, 2000.
5. 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 ACMSIAM Symposium on Discrete Algorithms* , pages 139–149, January 1995.
6. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.
7. 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.
8. 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.
9. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th Annual ACMSIAM Symposium on Discrete Algorithms* , pages 89–90, 2001.
10. A. Maheshwari and N. Zeh. I/O-optimal algorithms for outerplanar graphs. *Journal of Graph Algorithms and Applications*, 8(1):47–87, 2004.
11. 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.
12. 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.
13. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th Annual ACMSIAM Symposium on Discrete Algorithms* , pages 267–276, 2002.
14. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
15. M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35:189–201, 2000.

## A Analysis of MZ-SSSP

This appendix provides omitted proofs of lemmas used in the analysis of MZ-SSSP$^*$.

**Lemma 8.** *At the end of an invocation of procedure Scan, $t_i \geq f$, for all $i$. The value of $t_i$ never decreases.*

*Proof.* Let $i$ be the highest level scanned in the invocation. Then, for $j \leq i$, $t_j$ is changed to a value no less than $f$. Level $i+1$ is not scanned because $t_{i+1} > f$, that is, $f \leq t_j$, for all $j > i$.

To see that the due time $t_i$ never decreases, observe that $t_i$ changes only when level $i$ is inspected. In this case, we have $f \geq t_i$ and $t_i$ is changed to a value no less than $f$. □

**Lemma 9 (Lemma 1).** *A vertex $x$ visited during a scan of level $i$ satisfies $t_i - 2^{i-2} \leq d(x) \leq d^*(x) \leq t_i$.*

*Proof.* The rightmost inequality is trivial because $x$ is visited only if $d^*(x) \leq t_i$. The middle inequality is also trivial because, when $x$ is loaded into a tree buffer, the current value of $d(x)$ is retrieved from REP, that is, $d^*(x) = d(x)$ at this time. Subsequently, the node $x$ in the tree buffer sees a subset of the updates of $d(x)$ performed on REP.

We prove the first inequality by induction on $d(x)$. Initially, we have $t_i = w_{\min} + 2^{i-2}$, where $w_{\min}$ is the minimum source weight of any vertex in $G$. Since no update can change $d(x)$ below $w_{\min}$, we have $d(x) \geq w_{\min} = t_i - 2^{i-2}$ if $x$ is visited before $t_i$ is changed for the first time.

If $x$ is visited after $t_i$ is changed, let $I$ be the invocation of procedure Scan that visits vertex $x$, and let $I'$ be the last invocation before $I$ that changed $t_i$. Since $x$ is not visited when iteration $I'$ inspects level $i$, we must have one of two cases: (1) Vertex $x$ is not released from $Q$ before invocation $I'$ or (2) it is released before invocation $I'$, but $d(x) > t_i$.

Case (2) immediately leads to a contradiction because we would have released a vertex with tentative distance greater than $t_i$ before scanning level $i$. So we have to consider Case (1). First we observe that, at the end of invocation $I'$, we have $d(x) \geq f'$, where $f'$ is the value of the frontier at the end of invocation $I'$. This follows because $x \in Q$ at the end of the invocation. If $d(x) \geq f'$ at the beginning of invocation $I$, we are done because invocation $I'$ changes $t_i$ to a value no greater than $f' + 2^{i-2}$, that is, $d(x) \geq t_i - 2^{i-2}$. If $d(x) < f'$ at the beginning of invocation $I$, there must be an edge $yx$ that is relaxed after invocation $I'$ and whose relaxation decreases $d(x)$ to a value less than $f'$. Since $d(y) < d(x)$, the induction hypothesis implies that, when vertex $y$ is visited, we have $d(y) \geq t_{c_y} - 2^{c_y-2}$. If $c_y = i$, then vertex $y$ is in fact visited in invocation $I$, and we have $d(x) > d(y) \geq t_i - 2^{i-2}$, as claimed. If $c_y \neq i$, vertices $x$ and $y$ cannot belong to the same mini-cluster, and edge $yx$ has length $\ell(yx) \geq 2^{c_y-2} + 2^{i-2}$, that is, $d(x) = d(y) + \ell(yx) \geq t_{c_y} - 2^{c_y-2} + 2^{c_y-2} + 2^{i-2} \geq f' + 2^{i-2}$, by Lemma 8, which is a contradiction because we assumed that $d(x) < f'$. □

The following lemma proves that a vertex $x$, once loaded into the hot pool, reaches level $c_x$ before a Visit$(x, d(x))$ signal does so, even if vertex $x$ is temporarily moved to a higher level. In particular, the lemma implies that, if $D(x) < t_{c_x+1}$, then the whole $c_x$-component containing $x$ is stored at level $c_x$. In the proof of the lemma, we make use of Lemmas 2 and 3. In particular, we use that, when a vertex $x$ is visited, we have $d(x) = D(x)$.

**Lemma 10.** *Let $C$ be a category-$j$ component. If $C$ is stored in a tree buffer at level $j+1$, then $D(C) \geq t_{j+1}$.*

*Proof.* Let $x$ be a vertex in $C$ such that $D(C) = D(x)$, and let $y$ be $x$'s predecessor on $\pi(x)$. Then $y \notin C$ and, hence, edge $yx$ has category at least $\max(c_y, j+1)$, that is, length at least $2^{c_y-2} + 2^{j-1}$. As long as vertex $y$ has not been visited yet, we either have $y \in Q$ or there is a Visit$(y, D(y))$ signal pending.

In the former case, we have $t_{c_y} \geq f$, by Lemma 8, where we use the value of $t_{c_y}$ at the time when $y$ is visited and the current value of $f$. By Lemma 9, we have $D(y) \geq t_{c_y} - 2^{c_y-2}$ when $y$ is visited. Hence, we have $D(y) \geq f - 2^{c_y-2}$.

In the latter case, since this signal has not been processed yet, we have $t_{c_y} \geq f$, using their current values and, thus, by Lemma 8, $t_{c_y} \geq f$ if we use the value of $t_{c_y}$ at the time when $y$ is visited and the current value of $f$. Thus, as in the previous case, $D(y) \geq f - 2^{c_y-2}$.

This implies, in both cases, that $D(C) \geq D(x) = D(y) + \ell(yx) \geq f - 2^{c_y-2} + 2^{c_y-2} + 2^{j-1} = f + 2^{j-1}$. Whenever level $j+1$ is inspected, $t_{j+1}$ is updated to a value no greater than $f + 2^{j-1}$. Hence, we have $D(C) > t_{j+1}$.

If $y$ has been visited already, then $d(C) = D(C)$. Since component $C$ is stored at a level greater than $j$, we have $d^*(C) \geq t_{j+1}$. Next we argue that $d^*(C) = D(C)$, so that $D(C) \geq t_{j+1}$.

Arguments analogous to the proof of Lemma 2 prove that the update of $d(C)$ to $D(C)$ is applied to level $j+1$ the next time level $j+1$ is inspected after vertex $y$ is visited. Let $I$ be this invocation, and let $I'$ be the previous invocation that inspected level $j+1$. Then, by our above argument, the fact that $C$ is stored at level $j+1$ implies that $D(C) \geq t_{j+1}$ after invocation $I'$ and before invocation $I$. Invocation $I$ updates $d^*(C)$ to $D(C)$, so that, if $d^*(C) \geq t_{j+1}$ at any time after invocation $I$, this implies that $D(C) \geq t_{j+1}$. □

## B Computing the Graph Decomposition

### B.1 High-Level Description

To describe the computation of a barrier decomposition with properties (P1)–(P3), a little more notation is required. Let $T$ be a minimum spanning tree of $G$. For a $c$-component $C$, let $T[C]$ be the subtree of $T$ induced by the vertices of $C$, and let $\tilde{T}[C]$ be the tree obtained from $T[C]$ by contracting all edges of category less than $c$. Let $\mathcal{E} = (e_1, \dots, e_{2n-2})$ be an Euler tour of $T$, let $\mathcal{E}[C]$ be the tour obtained from $\mathcal{E}$ by removing all edges that do not belong to $T[C]$, and let $\tilde{\mathcal{E}}[C]$ be the tour obtained from $\mathcal{E}[C]$ by removing all edges of category less than $c$, that is, $\mathcal{E}[C]$ is an Euler tour of $T[C]$ and $\tilde{\mathcal{E}}[C]$ is an Euler tour of $\tilde{T}[C]$. Let the *Euler diameter* $\operatorname{diam}_{\mathcal{E}}(C)$ of a component be equal to the weight of all edges in $\mathcal{E}[C]$. For two category-$c$ edges $e$ and $f$ in $C$, let $\operatorname{dist}_{\mathcal{E}}(e,f)$ be the weighted length of the subtour $(e, \dots, f)$ of $\mathcal{E}[C]$.

Let $B_T$ be the bottleneck tree of $T$, which is defined recursively as follows: If $T$ has one vertex, then $B_T$ consists of this vertex. If $T$ has more than one vertex, let $e$ be the heaviest edge in $T$, and let $T_1$ and $T_2$ be the two connected components of $T - e$. Then $B_T$ has a root node representing $e$, and the two children of $e$ are the roots of bottleneck trees for $T_1$ and $T_2$. The component tree $T_c$ can be computed from $B_T$ as follows: Assign category 0 to every leaf of $B_T$, and let the category of every internal node be equal to the category of the corresponding edge in $T$. Then contract all edges in $B_T$ whose endpoints have the same category. The result is $T_c$.

*Computing a μpartition.* A $\mu$-partition of $G$ can now be obtained as follows: Process $T_c$ bottom-up. For every component $C$, compute the number $n_C$ of vertices in $C$ that have not been assigned to clusters yet. For a leaf, let $n_C = 1$. For an internal node $C$ with children $C_1, \dots, C_k$, let $n_C = \sum_{i=1}^{k} n_{C_i}$. If $n_C \leq \mu$ and $\operatorname{diam}_{\mathcal{E}}(C) \leq 2^c\mu$, where $c$ is the category of $C$, label $C$ as not assigned to a cluster and proceed to $C$'s parent. Otherwise, let $n_C = 0$ and use $\tilde{\mathcal{E}}[H]$ to partition $C$ into clusters: First observe that every edge in $\tilde{\mathcal{E}}[H]$ connects two $(c-1)$-components contained in $C$. Thus, we can consider $\tilde{\mathcal{E}}[H]$ to be a sequence $\mathcal{F} = (C_{i_1}, f_1, C_{i_2}, f_2, \dots, f_{t-1}, C_{i_t})$ of alternating components and edges such that, for $1 \leq j \leq t-1$, the first endpoint of $f_j$ is in $C_{i_j}$ and the second endpoint is in $C_{i_{j+1}}$. Now perform a scan of this sequence to group components $C_1, \dots, C_k$ into clusters. During the scan, maintain an upper bound $D$ on the diameter of the cluster currently under construction, as well as the number $S$ of vertices in the cluster. Initially, $D = 0$ and $S = 0$. For each item in $\mathcal{F}$, perform the following actions:

- When processing a component $C_{i_j}$, distinguish two cases:
  - If this is the first occurrence of $C_{i_j}$ in $\mathcal{F}$ and $n_{C_{i_j}} > 0$, then add $\operatorname{diam}_{\mathcal{E}}(C_{i_j})$ to $D$ and increase $S$ by $n_{C_{i_j}}$. If now $D > 2^c\mu$ or $S > \mu$, start a new cluster consisting of component $C_{i_j}$, and define $D = \operatorname{diam}_{\mathcal{E}}(C_{i_j})$ and $S = n_{C_{i_j}}$. Label $C_{i_j}$ with the ID of this cluster. Otherwise, add $C_{i_j}$ to the current cluster, and label $C_{i_j}$ with its ID.
  - If this is not the first occurrence of $C_{i_j}$ or $n_{C_{i_j}} = 0$, there are two subcases: If $D = 0$, do nothing. If $D > 0$, add $\operatorname{dist}_{\mathcal{E}}(f_{j-1}, f_j) - \ell(f_{j-1}) - \ell(f_j)$ to $D$.

– When processing an edge $f_j$, if $D = 0$, do nothing; otherwise, add $\ell(f_j)$ to $D$.

At the end, if $n_C > 0$, where $C$ is the root of $T_c$, label $C$ with the ID of a unique cluster to collect all vertices that have not been assigned to clusters yet. Once $T_c$ has been processed like this, find for every leaf $x$ the lowest labelled ancestor in $T_c$ and label $x$ with the cluster ID of that ancestor. The final $\mu$-partition is defined as the collection of maximal vertex sets $V_1, \ldots, V_q$ such that all vertices in a set $V_i$ have the same cluster ID.

**Lemma 11.** *The above procedure produces a $\mu$-partition of $G$.*

*Proof.* First let us prove that every cluster has properties (C1)–(C4): Consider a cluster $V_i$ formed when partitioning a $c$-component. If this cluster is a $c$-cluster, its diameter is bounded by $2^c\mu$ because otherwise its last $(c-1)$-component in the Euler tour would have started a new cluster. (Note that a single $(c-1)$-component contributing to $V_i$ cannot have diameter greater than $2^c\mu$ because otherwise it would have been partitioned into clusters of category less than $c$.) For the same reason, it cannot contain more than $\mu$ vertices. If $V_i$ is a $c'$-cluster with $c' < c$, then the $c'$-component containing $V_i$ is not partitioned (because $V_i$ is formed when partitioning the category-$c$ ancestor of this component), which implies that the whole component, and thus $V_i$, has diameter at most $2^{c'}\mu$ and contains at most $\mu$ vertices that have not been assigned to clusters yet. Thus, in either case, $V_i$ has properties (C1) and (C2).

A $(c-1)$-component cannot contribute vertices to more than one $c$-cluster because the partitioning of $c$-components assigns complete $(c-1)$-components, that is, more precisely, all unassigned vertices in these components to clusters. This proves property (C3).

Finally, it is easy to see that every $c'$-component that contributes to a $c$-cluster with $c > c'$ cannot have diameter greater than $2^{c'}\mu$ because otherwise this component would have been partitioned into clusters of category at most $c'$. Thus, $V_i$ has property (C4).

It remains to bound the number of clusters: Clusters are formed by partitioning category components. We count two types of clusters separately: For every component $C$ that is partitioned into clusters by our construction, the first cluster is of type I. All remaining clusters are of type II. To pay for the creation of the type-I cluster when partitioning a $c$-component $C$, we add $\mathrm{diam}_{\mathcal{E}}(C)/(2^c\mu) + n_C/\mu$ to a charge account $\Phi_1$. To pay for the creation of any type-II cluster, we add $D/(2^c\mu) + S/\mu$ to a charge account $\Phi_2$, where we use the values of $D$ and $S$ at the time when the cluster is created. Since $n_C > \mu$ or $\mathrm{diam}_{\mathcal{E}}(C) > 2^c\mu$ if $C$ is partitioned, $\Phi_1$ increases by at least one for every type-I cluster. Similarly, $\Phi_2$ increases by at least one for every subsequent cluster because this cluster is formed whenever $D > 2^c\mu$ or $S > \mu$. To bound the number of clusters, it therefore suffices to bound $\Phi_1$ and $\Phi_2$.

To bound $\Phi_1$, we observe that $n_C = 0$ after partitioning a component $C$. Hence, every vertex contributes exactly $1/\mu$ to $\Phi_1$, and the total contribution of all $n_C/\mu$ terms to $\Phi_1$ is $n/\mu$. A category-$c$ edge $xy$ in the Euler tour contributes only to the diameters of category-$c'$ components with $c' \geq c$. Hence, its total contribution to $\Phi_1$ is at most $\sum_{c'=c}^{+\infty} \ell(xy)/(2^{c'}\mu) < \sum_{i=0}^{+\infty} 1/(2^i\mu) = 2/\mu$. Since the Euler tour contains less than $2n$ edges, the total contribution of all Euler tour edges to $\Phi_1$ is less than $4n/\mu$. Thus, there are at most $5n/\mu$ type-I clusters in total.

The contribution of vertices to $\Phi_2$ can be bounded by $2n/\mu$ as follows: Once a vertex $x$ is assigned to a cluster, it does not contribute to $S$ in any subsequent partitioning step. It may, however, be charged for the creation of two clusters, namely if it is in a $(c-1)$-component that starts a new cluster. In this case, it contributes to the values of $S$ charged for the creation of the cluster containing

$x$, as well as the following cluster. Thus, every vertex contributes at most $2/\mu$ to $\Phi_2$, and the total contribution of all vertices in at most $2n/\mu$.

Similarly, an edge contributes at most $8/\mu$ to $\Phi_2$. Consider the partitioning of a $c$-component $C$. If edge $e$ has category $c$, it contributes $\ell(e)/(2^c\mu)$ to $D$ at most once. Otherwise, let $C'$ be the $(c-1)$-component that contains $e$. Then $e$ contributes up to two times to $\Phi_2$, for exactly the same reasons why a vertex may contribute to $\Phi_2$ twice. As in the analysis of edge contributions to $\Phi_1$, this sums up to a total contribution of $8n/\mu$ by all the edges. Thus, there are at most $10n/\mu$ type-II clusters. □

*Making deep clusters skinny.* The next step is to refine the $\mu$-partition so that every deep cluster $V_i$ (that is, cluster of depth greater than $d = \log n + 2$) has the following properties:

(DC1) For every category $c'$, there exists at most one $c'$-component that shares some, but not all, of its vertices with $V_i$.

(DC2) Let $c$ be the category of $V_i$, let $C$ bet the $c$-component containing $V_i$, and let $c''$ be the minimal category such that there exists a $c''$-component $C'$ that shares some, but not all vertices with $V_i$. Then every child of $C'$ is either disjoint from $V_i$ or completely contained in $V_i$, and every vertex in $C \setminus V_i$ is contained in $C'$.

Consider a deep $c$-cluster $V_i$, and let $T_i$ be its cluster tree. We call a node $C$ of $T_c$ *internal*, *mixed*, or *external*, depending on whether all, some, or none of the vertices in $C$ belong to $V_i$. Now process $T_i$ bottom-up. For a given node $C$, apply the following rules:

- If $C$ is internal or external, proceed to $C$'s parent.
- If $C$ is mixed, let $C_1, \dots, C_k$ be its children in $T_c$. If exactly one of them is mixed and the rest are internal or $C$ has no mixed child, proceed to $C$'s parent. If there is at least one mixed child and at least two non-internal (mixed or external) children, then iterate over the mixed children and, for each such child, create a new cluster that contains all vertices in the subtree of $T_i$ rooted at this child. Declare this child to be external. If $C$ now has only external children, declare $C$ to be external.

**Lemma 12.** *Given a $\mu$partition $\mathcal{P} = (V_1, \dots, V_q)$ of $G$, the above procedure produces a $\mu$partition $\mathcal{P}' = (V_1', \dots, V_s')$ of $G$ that has properties (DC1) and (DC2).*

*Proof.* It is easy to verify that all clusters in partition $\mathcal{P}'$ have properties (C1)–(C4). We bound the number of clusters and prove that each deep cluster has properties (DC1) and (DC2).

When creating a cluster containing all vertices in $V_i$ that belong to a mixed child $C'$ of a node $C$ in $T_i$, we can think of this as cutting the edge $CC_j$. Let $T_i'$ be the tree obtained by contracting all edges in $T_i$ that are *not* cut. The total number of new clusters we create is equal to the number of edges in $T_i'$, which is at most twice the number of leaves of $T_i'$. We argue that we can charge each leaf in all these trees $T_i'$ to a unique cluster $V_j$ in $\mathcal{P}$, which proves that the number of clusters at most triples.

We start by observing that the cluster trees $T_i$ are edge-disjoint. Every leaf of $T_i'$ corresponds to a category component $C$ that contributes some, but not all, of its vertices to $V_i$. This component must contain another cluster whose cluster tree root $C'$ is equal to $C$ or is a descendant of $C$ such that every node on the path from $C$ to $C'$ in $T_c$ is not a cluster tree root. We then charge $C$ to $C'$, which ensures that every cluster tree root is charged at most once. This proves our claim.

Now observe that the partitioning procedure explicitly guarantees that every cluster tree $T'_j$ corresponding to a cluster $V'_j$ in $\mathcal{P}'$ contains at most one node corresponding to a mixed component per level, that is, cluster $V'_j$ has property (DC1). To prove property (DC2), let $c$ be the category of $V'_j$, let $C$ be the $c$-component containing $V'_j$, and let $C'$ be the lowest mixed node in $T'_j$. Now assume that not every node in $C \setminus V'_j$ is contained in $C'$. Then there exists a proper ancestor $C''$ of $C'$ such that at least two of its children have descendant leaves that do not belong to $V'_j$, one of which is an ancestor of $C'$. Since the ancestor of $C'$ is mixed, $C''$ satisfies the condition for removing all of its mixed descendants from $V'_j$. This is a contradiction. □

*Computing the barrier decomposition.* To finish the construction, we consider each deep cluster $V'_i$ in the current partition in turn. Let $\tilde{T}_i$ be the subtree of $T_c$ induced by nodes that are mixed w.r.t. $V'_i$. Then, by property (DC1), $\tilde{T}_i$ is a single path $(C_1, \ldots, C_k)$, where $C_1$ is the root of $\tilde{T}_i$, and the categories of $C_1$ and $C_k$ differ by at least $d$. Every child $C' \notin \tilde{T}_i$ of a node $C_j \in \tilde{T}_i$ is either internal or external. All children of nodes $C_1, \ldots, C_{k-1}$ are internal, by property (DC2). Let $C_j$ be the deepest node whose category is at least $c - d$, where $c$ is the category of $C_1$, that is, the category of $V'_i$. Then split $\tilde{T}_i$ into three subpaths $(C_1, \ldots, C_j)$, $(C_{j+1}, \ldots, C_{k-1})$, and $(C_k)$. In the case when $j = k-1$, the middle path is empty. This partition of $\tilde{T}_i$ into subpaths corresponds to a partition of $V'_i$ into subclusters $V_i^1$, $V_i^2$, and $V_i^3$, where $V_i^3$ contains all descendant leaves of $C_k$ that belong to $V'_i$, $V_i^2$ contains all descendant leaves of $C_{j+1}$ that are in $V'_i \setminus V_i^3$, and $V_i^1$ contains all the remaining vertices of $V'_i$.

We obtain our barrier decomposition by making every vertex $C_{j+1}$ or $C_k$ in the above partition a graph $H_l$ and then defining graphs $G_l$ as in Section 4.1. Note that these graphs $G_l$ have as their vertex sets the leaves of the trees obtained by splitting each node $C_{j+1}$ or $C_k$ into a top and a bottom copy and making its children children of the bottom copy. It is then easy to see that every cluster tree corresponding to a cluster produced in the above partitioning of deep clusters is completely contained in such a tree corresponding to a graph $G_l$, that is, the cluster trees naturally define a partition of each graph $G_l$ into clusters.

We define a graph $G_l$ to be atomic if it corresponds to a tree that contains a path $(C_{j+1}, \ldots, C_{k-1})$ of the tree $\tilde{T}_i$ of a deep cluster $V'_i$. All other graphs are compound.

**Lemma 13.** *The above procedure produces a barrier decomposition with properties (P1)–(P3).*

*Proof.* First observe that there are only $O(n/\mu)$ graphs in the barrier decomposition because $\mathcal{P}'$ contains $O(n/\mu)$ clusters and each cluster causes the creation of at most two additional subgraphs in the partition. This proves property (P1).

It is easy to verify that the clusters in each graph $G_l$ form a $\mu$-partition. Now consider a compound graph $G_l$ and a cluster $V'$ in the $\mu$-partition of $G_l$. If $V'$ is a cluster of $\mathcal{P}'$, then it is not partitioned and, thus, has depth at most $\log n + 2$. Otherwise, it is a cluster $V_i^1$ or $V_i^3$ because the cluster $V_i^2$ belongs to an atomic graph. It is easily verified that each cluster $V_i^1$ or $V_i^3$ has depth at most $\log n + 2$. Thus, the partition has property (P2).

Now consider a graph $G_l$ whose parent is atomic. Then $H_l = C_k$ in the partition of a deep cluster $V'_i$, and its parent is $C_{j+1}$. But $C_{j+1}$ is easily seen to have only one child. If $G_l$'s parent is compound, then $H_l = C_{j+1}$ in the partition of a deep cluster $V'_i$. The cluster in $G_l$'s parent that contains $r(G_k)$ is then $V_i^1$. Note that this cluster contains only one graph representative and that the difference between the category of $C_{j+1}$ and $C_1$ is at least $\log n + 2$, which implies that $B_l$ is contained in $V_i^1$. This proves property (P3). □

## B.2 I/O-Efficient Implementation

The three steps of the above procedure are rather easy to implement in an I/O-efficient manner.

*Computing the μpartition.* The construction of the minimum spanning tree $T$ takes $O(MST(n,m))$ I/Os. As shown in [3], the extraction of the bottleneck tree $B_T$ from $T$ can be carried out in $O(\text{sort}(n))$ I/Os. This construction is easy to augment so that it labels every node in $B_T$ with the category of its corresponding edge. The contraction of the edges connecting nodes with the same category can then be carried out in $O(\text{sort}(n))$ I/Os using the Euler tour technique and list ranking [5].

An Euler tour $\mathcal{E}$ of $T$ can be computed in $O(\text{sort}(n))$ I/Os [5]. Given Euler tour $\mathcal{E}$, we have to compute $\text{diam}_{\mathcal{E}}(C)$, for each node $C$ in $T_c$, as well Euler tours $\tilde{\mathcal{E}}[C]$ and the distances $\text{dist}_{\mathcal{E}}(e,f)$, for consecutive edges $e$ and $f$ in $\tilde{\mathcal{E}}[C]$.

First the construction of $\tilde{\mathcal{E}}[C]$, for all $C \in T_c$: Label all the nodes of $T_c$ in preorder. Now sort the vertices of $T_c$ by their categories as primary keys and by their preorder numbers as secondary keys. Sort the edges in $\mathcal{E}$ by their categories as primary keys and by the minimum preorder numbers of their endpoints as secondary keys. It is not hard to see that this results in a partition of the edges in $\mathcal{E}$ into sets $\tilde{\mathcal{E}}[C]$ and that these sets are stored in the same order as their corresponding nodes $C$ in $T_c$. Rearranging the edges in each list $\tilde{\mathcal{E}}[C]$ in their order of appearance in $\mathcal{E}$ creates a correct Euler tour of $\tilde{T}[C]$.

Now observe that $\mathcal{E}[C]$ consists of all edges in lists $\tilde{\mathcal{E}}[C']$, where $C'$ is a descendant of $C$ in $T_c$. Hence, we can process $T_c$ bottom-up, computing for every node $C$, $\text{diam}_{\mathcal{E}}(C) = \sum_{e \in \tilde{\mathcal{E}}[C]} \ell(e) + \sum_{C'} \text{diam}_{\mathcal{E}}(C')$, where the second sum is over all children of $C$ in $T_c$.

The computation of distances $\text{dist}_{\mathcal{E}}(e,f)$, for consecutive edges $e$ and $f$ in $\tilde{\mathcal{E}}[C]$ is discussed in Appendix B.3. Note that we need to answer only $O(n)$ such distance queries. By Corollary 1, this takes $O(\text{sort}(n))$ I/Os.

Given this information, the initial $\mu$-partition is easy to compute because it requires processing $T_c$ bottom-up and, for every node, traversing the Euler tour $\tilde{\mathcal{E}}[C]$. Since $T_c$ has size $O(n)$ and the total size of all tours $\tilde{\mathcal{E}}[C]$ is $O(n)$, the total cost of this procedure is $O(\text{sort}(n))$.

Once the nodes of $T_c$ have been labelled with cluster IDs in this processing phase, it suffices to process $T_c$ top-down to propagate cluster IDs from labelled nodes to the leaves in the corresponding cluster. This takes another $O(\text{sort}(n))$ I/Os [5].

*Refining the μpartition.* Given the initial cluster partition, it suffices to sort the adjacency lists of the vertices in $G$ so that the adjacency lists of all vertices in the same cluster are stored consecutively. Now a single scan of these edge lists suffices to identify all deep clusters, that is, all clusters whose category is more than $d$ more than the category of the shortest edge with exactly one endpoint in the cluster.

The next task is identifying $T_i$, for every deep cluster $V_i$. This is done quite easily by marking all leaves of $T_c$ that belong to a deep cluster. In addition, label each such leaf with the smallest interval of preorder numbers that contains all preorder numbers of nodes in $V_i$. This is useful for identifying the root of $T_i$. Now process the nodes of $T_c$ bottom-up. For every node $C$ with children $C_1, \dots, C_k$, if there is such a child marked as belonging to a cluster $V_i$ and this child is not the root of $T_i$, mark $C$ as belonging to $T_i$. If $C$'s preorder interval includes the preorder interval of $V_i$, then mark $C$ as being the root of $T_i$.

The cost of this procedure is $O(\text{sort}(n))$ for processing $T_c$ in this fashion. The labelling of nodes in $T_c$ as belonging to trees $T_i$ can be computed using the above procedure because every non-root node belongs to exactly one tree $T_i$ (otherwise, the corresponding component would contribute vertices to two clusters of a higher category). For every root node, it is easy to create a copy per tree $T_i$ of which it is a root. Since there are only $O(n/\mu)$ clusters, only $O(n/\mu)$ copies of nodes in $T_c$ are made. Now sort the nodes by their membership in trees $T_i$. Then the procedure for partitioning each tree $T_i$ into subtrees that induce the refined $\mu$-partition can be performed by processing each tree $T_i$ bottom-up.

*Computing the barrier decomposition.* To carry out the third step, we again partition $T_c$ into subtrees corresponding to clusters in the refined partition. Splitting each tree into the three pieces is then easily done in $O(\text{sort}(n))$ I/Os. Once $T_c$ has been partitioned in this fashion, the construction of graphs $G_1, \dots, G_q$ and arranging the clusters in their cluster partitions is trivially implemented in $O(\text{sort}(n))$ I/Os.

Thus, we obtain a barrier decomposition of $G$ with properties (P1)–(P3) in $O(MST(n,m) + \text{sort}(n)) = O(MST(n,m))$ I/Os, which proves Lemma 7.

## B.3 Computing Euler Distances

In this section, we assume that we are given the Euler tour $\mathcal{E}$, the component tree $T_c$, and $O(n)$ queries to determine distances $\text{dist}_{\mathcal{E}}(e,e')$. We assume that the edges in $\mathcal{E}$ are numbered 1 through $2n-2$—we refer to these numbers as the *indices* of the edges—and that every query to compute $\text{dist}_{\mathcal{E}}(e,e')$ is given as the interval $[e+1, e'-1]$, annotated with the ID of the component $C$ such that $e$ and $e'$ are consecutive in $\tilde{E}[C]$. For two edges $e, e' \in \mathcal{E}$, let $\mathcal{E}(e,e')$ be the subtour of $\mathcal{E}$ including all edges with indices in the range $[e,e']$. $\mathcal{E}[C](e,e')$ is defined similarly w.r.t. $\mathcal{E}[C]$. Note that edges $e$ and $e'$ are not necessarily part of $\mathcal{E}[C](e,e')$.

Now consider a query $[e,e']$ generated by two edges in the tour $\tilde{E}[C]$ of a $c$-component $C$. If $\mathcal{E}(e,e')$ does not contain an edge of category greater than $c$, then $\mathcal{E}[C](e,e') = \mathcal{E}(e,e')$, and we can answer query $[e,e']$ by answering a range weight query over $\mathcal{E}$, that is, by summing the weights of all edges in $\mathcal{E}$ with indices in the range $[e,e']$ and returning the resulting total weight. If there is an edge of category greater than $c$ in $\mathcal{E}(e,e')$, then $\mathcal{E}(e,e')$ leaves $C$ through an edge $f = xy$ of category greater than $c$ and returns to $c$ through the opposite edge $f^r = yx$. $\mathcal{E}(e,e')$ may leave $C$ and return to $C$ several times. For every edge $f \in \mathcal{E}(e,e')$ through which $\mathcal{E}(e,e')$ leaves $C$, the edges in $\mathcal{E}(f,f^r)$ do not belong to $\mathcal{E}[C](e,e')$.

Our goal now is to start with the whole Euler tour $\mathcal{E}$ and to delete the edges from $\mathcal{E}$ in such an order that, for any query $[e,e']$, there exists a point in time when $\mathcal{E}(e,e') = \mathcal{E}[C](e,e')$; this is the point at which we answer query $[e,e']$ by answering a range weight query over $\mathcal{E}$. More precisely, if we say that processing an edge $e$ means deleting $e$ from $\mathcal{E}$ and processing a query means answering it over the subsequence of $\mathcal{E}$ produced by all edge deletions performed so far, we want to find an ordering of the edges and the queries so that processing them in this order gives the correct answer for each query.

This ordering is obtained rather easily from the component tree $T_c$. First we extend $T_c$ by adding a child $[e,e']$ to a node $C$ in $T_c$, for every query $[e,e']$ over $\mathcal{E}[C]$; we also add a child $e$ to $C$, for every edge $e$ in $\tilde{\mathcal{E}}[C]$. Let $T_c^*$ denote this extended version of $T_c$. Now let the index of a component $C$ be the minimum index of any edge contained in $C$. We sort the children of each component in $T_c^*$ so that components and edges succeed queries and the components and edges are sorted by decreasing indices. Now we compute the postorder numbering of the nodes of $T_c^*$ defined by the depth-first traversal that visits the children of each node in the given order. The ordering of queries and edges defined by this numbering is the order in which we process them. The next lemma proves that this gives the correct result for every query.

**Lemma 14.** *Processing queries and edges in the order described above makes every query $[e,e']$ over $\mathcal{E}[C]$ return the correct weight of $\mathcal{E}[C](e,e')$.*

*Proof.* Consider a query $[e,e']$ and an edge $f$ in $\mathcal{E}(e,e')$, and let $c$ be the category of edges $e$ and $e'$. We say that $f$ is *separated from* query $[e,e']$ if there exists an edge $g$ of category greater than $c$ such that $f \in [g,g^r]$ and $[g,g^r] \subset [e,e']$. Then an edge in $\mathcal{E}(e,e')$ belongs to $\mathcal{E}[C](e,e')$ if and only if it is not separated from query $[e,e']$. Therefore, we need to prove that, by the time we answer query $[e,e']$, an edge $f \in \mathcal{E}(e,e')$ is still present in $\mathcal{E}$ if and only if it is not separated from $[e,e']$.

First assume that $f$ is separated from $[e,e']$, and let $C'$ be the LCA of $f$ and $[e,e']$ in $T_c^*$. Since $f$ and $e$ do not belong to the same $c$-component, $C'$ is a proper ancestor of $C$ and $f$. Let $x$ and $y$ be the two children of $C'$ that are ancestors of $C$ and $f$, respectively. Since $f \in [e,e']$, the index of $x$ must be less than the index of $y$. Hence, $[e,e']$ succeeds $f$ in the processing order, that is, $f$ is removed from $\mathcal{E}$ before $[e,e']$ is answered.

Now assume that $f$ is not separated from $[e,e']$. Then $f$ must belong to the same $c$-component $C$ as $e$ and $e'$. Thus, $f$ must have category at most $c$, that is, it is either contained in a child component of $C$ or it is itself a child of $C$. Since the components and edges that are children of $C$ succeed the queries that are children of $C$ in the processing order, we process query $[e,e']$ before deleting edge $f$. □

The above ordering can easily be computed in $O(\text{sort}(n))$ I/Os using standard techniques. The next lemma states that the resulting sequence of edge deletions and range weight queries over the Euler tour can be processed in $O(\text{sort}(n))$ I/Os using a buffer tree. Again, the required buffer tree operations are standard.

**Lemma 15.** *A sequence of $N$ deletions and range weight queries can be answered in $O(\text{sort}(N))$ I/Os using a buffer tree.*

Since $\mathcal{E}$ has length $O(n)$, that is, we perform only $O(n)$ deletions, and since we have to determine only $O(n)$ distances $\text{dist}_{\mathcal{E}}(e,e')$, Lemmas 14 and 15 immediately imply

**Corollary 1.** *The Euler distances $\text{dist}_{\mathcal{E}}(e,e')$ required in the computation of a $\mu$-partition of an undirected graph $G$ can be computed in $O(\text{sort}(n))$ I/Os.*

## C The Distance Repository

The distance repository is very similar to a buffered repository tree (BRT) [4]. In particular, we number the vertices of $G$ in the order in which they are visited by a depth-first traversal of $T_c$, and we assign them to the leaves of a balanced binary tree in this order, $B$ vertices per leaf. Every internal node of $T_c$ is assigned the smallest interval containing the numbers of all its descendant leaves.

Initially, every leaf stores the source weights of all its vertices as their tentative distances, and every internal node stores the minimal tentative distance of all vertices stored in its descendant leaves.

### C.1 Updates

An Update$(x,d)$ operation traverses the path from the root to the leaf containing $x$, updating $x$'s tentative distance as well as the tentative distance of every ancestor of the leaf storing $x$. As in a BRT, this is implemented in a lazy fashion, by associating a buffer of size $B$ with every internal node and inserting Update$(x,d)$ signals into the root buffer. When the buffer of a node $v$ overflows, we update $v$'s tentative distance to the minimum of its current tentative distance and the tentative distances of all Update signals in the buffer. We then distribute the signals to the two children of $v$.

In particular, if these children are internal nodes, the signals are appended to their buffers, and the buffers are emptied recursively if this causes them to overflow. If the children are leaves, they are loaded into internal memory, and the Update signals are applied to the vertices they store. The cost per buffer emptying process is $O(1/B)$ amortized per signal. Since every signal participates in $O(\log(n/B))$ buffer emptying processes, the cost per Update operation is $O((1/B)\log(n/B))$ amortized, as claimed.

### C.2 Graph Queries

A GraphQuery$(G_i)$ operation now proceeds as follows: Let $V_i^1$ be the set of vertices in $G_i$ that are not graph representatives, and let $V_i^2$ be the set of graph representatives. First we empty all buffers of ancestors of the vertices in $V_i^1$, and then we inspect all leaves storing vertices in $V_i^1$ to retrieve their current tentative distances. For every vertex in $V_i^2$, we observe that, since it represents a subgraph $H_j$ of $G$, it corresponds to a consecutive range $[a,b]$ of vertex IDs, and the tentative distance of $r(G_j)$ is the minimum tentative distance of the vertices in this range. This can be queried by emptying the buffers along the paths from the root to the leaves storing vertices $a$ and $b$. Then we query the tentative distances of $a$ and all vertices in the same leaf as $a$ that are to the right of $a$, the tentative distances of $b$ and all vertices in the same leaf as $b$ that are to the left of $b$, as well as the tentative distances of all children of nodes on the two traversed paths whose intervals are completely contained in $[a,b]$.

The total cost of processing the nodes in $V_i^1$ is $O((1+|V_i^2|)\log(n/B)+L_1)$ amortized, where $L_1$ is the number of leaves that store vertices belonging to $V_i^1$. Indeed, if $|V_i^2| = 0$, then the leaves in $L_1$ form a consecutive range of leaves, and the query bound would be $O(\log(n/B)+L_1)$; every node in $V_i^2$ creates a gap in the range of the leaves in $L_1$, and it may be necessary to traverse a path of length $\log(n/B)$ to reach each of the leaves in $L_1$ immediately before and after the gap.

The total cost of processing the nodes in $V_i^2$ is $O(|V_i^2|\log n)$ amortized. Now observe that $L_1 \leq 2+|V_i^1|/B+2|V_i^2|$: If there were no gaps in the sequence of vertex IDs of the vertices in $V_i^1$, then these vertices would be stored in at most $2+|V_i^1|/B$ leaves of the BRT. Every gap creates at most two nodes that are visited, but which contribute less than $B$ nodes to the output. Each such gap, however, corresponds to a node in $V_i^2$. Hence, the total cost is $O((1+|V_i^2|)\log(n/B)+|V_i|/B)$, as claimed.

### C.3 Cluster Queries

A ClusterQuery$(V_i)$ operation is similar to a GraphQuery, but needs to take into account that the intervals of vertex IDs of the nodes in the queried cluster tree $T_i$ are non-disjoint. Let $C$ be the root of $T_i$. Then we partition the range of IDs of vertices in $C$ into three sets of intervals. Let $V_i^1$ be the set of leaves of $T_i$ that are not graph representatives. Let $V_i^2$ be the set of leaves of $T_i$ that are graph representatives, and let $I_i^2$ be the set of their corresponding intervals of vertex IDs. Finally, for every node $C' \in T_i$ such that not all vertices in $C'$ belong to $V_i$, partition the set of vertices in $C'$ that belong neither to $V_i$ nor to a child of $C'$ in $T_i$ into intervals of consecutive vertex IDs and add these intervals to a set $I_i^3$.

We start by retrieving the tentative distances of all vertices in $V_i^1$ as in the case of a graph query. The intervals in $I_i^2$ and $I_i^3$ are processed as the vertices in $V_i^2$ in the case of a graph query. Now store the tentative distance of every vertex in $V_i^1$ with its corresponding leaf in $T_i$; store the result of every query in $I_i^2$ with its corresponding leaf in $T_i$; and, for every node $C' \in T_i$ that added intervals to $I_i^3$, store the minimum tentative distance of all its intervals with $C'$. Now process $T_i$ bottom-up and update the tentative distance of every node to be the minimum of its own current tentative distance and the tentative distances of its children. It is obvious that this produces the correct tentative distances for all nodes in $T_i$.

As for the I/O-complexity, let us first consider the cost of the queries in the three sets. Every query in $I_i^2$ or $I_i^3$ costs $O(\log(n/B))$ I/Os and corresponds to a cluster tree $T_j$ whose root either belongs to $T_i$ or is a child of a node in $T_i$. The queries for the vertices in $V_i^1$ cost $O((1+|I_i^2|+|I_i^3|)\log(n/B)+|V_i^1|/B)$ I/Os because, as in the case of a GraphQuery, the vertices in $V_i^1$ would be stored in consecutive leaves, except for the gaps introduced by graph representatives and clusters other than $V_i$ that contain vertices of $C$. Each such gap, adds at most two leaves to the set of leaves to be inspected. Hence, in total, the cost of the queries on the BRT equals $O((1+r_i)\log(n/B)+|V_i|/B)$, where $r_i$ is the number of cluster tree roots in or adjacent to $T_i$. (Note that $r_i$ accounts for the queries in both $|I_i^2|$ and $|I_i^3|$ because every graph representative is also a cluster tree root.)

Once the queries in the three sets have been answered, assigning them to the correct nodes in $T_i$ and processing $T_i$ bottom-up can be accomplished in $O((|T_i|+r_i)/B)$ I/Os, provided that the nodes of $T_i$ are stored in preorder, which is easily ensured when preparing the cluster trees. Hence, this increases the cost of the query by only a constant factor, and the cost of a ClusterQuery is as claimed. This proves

**Lemma 16.** *There exists a data structure supporting Update$(x,d)$ operations in $O((1/B)\log(n/B))$ I/Os, GraphQuery$(G_i)$ operations in $O((1+c_i)\log(n/B)+|G_i|/B)$ I/Os, and ClusterQuery$(V_j)$ operations in $O((1+r_j)\log(n/B)+|T_j|/B)$ I/Os, where $c_i$ is the number of children of $G_i$ and $r_j$ is the number of cluster tree roots in or adjacent to $T_j$. All I/Obounds are amortized.*