I/O-Efficient Undirected Shortest Paths with Unbounded Edge Lengths (Extended Abstract*)

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 show how to compute single-source shortest paths in undirected graphs with non-negative edge lengths in $\mathcal{O}(\sqrt{nm/B}\log n + MST(n,m))$ 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. For sparse graphs, the new algorithm performs $\mathcal{O}((n/\sqrt{B})\log n)$ I/Os. This result removes our previous algorithm's dependence on the edge lengths in the graph.

1 Introduction

Let $G = (V, E)$ be a graph, let s be a vertex of G, called the *source vertex*, and let $\ell : E \to \mathbb{R}^+$ be an assignment of non-negative real lengths to the edges of G. The *single-source shortest-path* (SSSP) problem is to find, for every vertex $v \in V$, the distance, $\text{dist}_G(s, v)$, from s to v, that is, the length of a shortest path from s to v in G. We focus mostly on the equivalent *closest-source shortest-path* (CSSP) problem: In addition to the input for SSSP, let $w : V \to \mathbb{R}^+$ be an assignment of non-negative *weights* to the vertices of G. Then compute for every vertex $x \in G$, its distance $D(x) = \min\{w(y) + \text{dist}_G(y, x) \mid y \in G\}$ from the closest source. If y is a vertex such that $w(y) + \text{dist}_G(y, x) = D(x)$, a *shortest path to* x, denoted $\pi(x)$, is a path of length $\text{dist}_G(y, x)$ from y to x. The classical SSSP-algorithm for general graphs is Dijkstra's algorithm [6], which has seen many improvements, particularly for undirected graphs with integer or float edge lengths [12, 13], and undirected graphs with real edge lengths [11]. When applied to massive graphs that do not fit in memory and are stored on disk, however, Dijkstra's algorithm and its improved variants perform poorly. This is because they access the data in a random fashion.

More recently, much work has focused on SSSP in massive graphs. These algorithms are analyzed in the *I/O-model* [1], which assumes 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

* For more details, see [10].

** Research supported by DFG grant ME 2088/1-3.

*** Research supported by NSERC and CFI.

memory, which happens in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* (I/O). The complexity of an algorithm is the number of I/Os it performs; e.g., $\mathrm{sort}(n) = \Theta((n/B)\log_{M/B}(n/B))$ I/Os to sort n numbers [1].

Little is known about solving SSSP in general *directed* graphs I/O-efficiently. For *undirected* graphs, the algorithm of Kumar and Schwabe (*KS-SSSP*) [7] performs $\mathcal{O}(n+(m/B)\log(n/B))$ I/Os. For dense graphs, the second term dominates; but for sparse graphs, the I/O-bound becomes $\mathcal{O}(n)$. The SSSP-algorithm of Meyer and Zeh (*MZ-SSSP*) [9] extends the ideas of [8] for breadth-first search (BFS) to graphs with edge lengths between 1 and W, leading to an I/O-bound of $\mathcal{O}(\sqrt{nm\log W/B}+MST(n,m))$, where $MST(n,m)$ is the I/O-cost of computing a minimum spanning tree.[1] This paper removes the algorithm's dependence on W using a number of new ideas, proving the following result:

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

Note that for sparse graphs, the cost of our algorithm is $\mathcal{O}((n/\sqrt{B})\log n)$ I/Os. The rest of the paper is organized as follows: Section 2 discusses the ideas of KS-SSSP and MZ-SSSP that are reused in our algorithm. Section 3 augments MZ-SSSP to make it independent of the edge lengths, assuming an appropriate graph partition. The algorithm's complexity now depends on a parameter of the partition, called its depth. Section 4 describes a recursive shortest-path algorithm that uses another partition into "well-separated" subgraphs, allowing the computation of shortest paths in the whole graph using nearly independent computations on these subgraphs. Section 5 argues that any graph can be partitioned into such well-separated subgraphs, while at the same time ensuring that each has a partition of small depth in the sense of Sect. 3. This allows the two approaches from Sects. 3 and 4 to be combined to obtain an efficient CSSP-algorithm (which also solves SSSP) as stated in Thm. 1 above.

2 Previous Work: KS-SSSP and MZ-SSSP

KS-SSSP. KS-SSSP [7] is an I/O-efficient version of Dijkstra's algorithm. It uses a priority queue Q to maintain the tentative distances of all vertices and retrieves the vertices one by one from Q, by increasing tentative distances. After retrieving a vertex x from Q, x is *visited*, that is, its incident edges 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 is reflected by updating the priority of y in Q.

The main contribution of KS-SSSP is an I/O-efficient priority queue that supports Update(x,p), Delete(x), and DeleteMin operations, each in amortized

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

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

$\mathcal{O}((1/B)\log(n/B))$ I/Os. The latter two respectively delete x or the item with minimum priority from the priority queue. The former replaces x's current priority p_x with $\min(p_x, p)$ if $x \in Q$. If $x \notin Q$ and x has never been in Q, it is inserted with priority p. If x has been in Q before, but has been deleted, the operation does nothing! This particular behaviour of update operations is supported only for SSSP-computations in undirected graphs that visit vertices by increasing distances, because information about the structure of the resulting update sequence is used to ensure this behaviour (see [7] for details). As our algorithms visit vertices out of order, more effort is required to ensure this behaviour in our algorithms. Details appear in the full paper.

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 $\mathcal{O}(m)$ priority queue operations, which cost $\mathcal{O}((m/B)\log(n/B))$ I/Os, and it spends $\mathcal{O}(1+\deg(x)/B)$ I/Os to retrieve the adjacency list of each vertex x, $\mathcal{O}(n + m/B)$ I/Os in total. For sparse graphs, the bottleneck of KS-SSSP is thus the random accesses to the adjacency lists. This problem is addressed by MZ-SSSP and by our new algorithm.

MZ-SSSP. MZ-SSSP partitions the vertex set of G into $q = \mathcal{O}(n/\mu)$ carefully chosen sets $V_1, \ldots, V_q$, 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. 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 long enough to compute incorrect distances, as formalized by the following property:

(SP) If vertex y is visited before vertex x, then $D(y) \le D(x) + \mathrm{dist}_G(x, y)/2$.

This implies in particular that the vertices along any shortest path are visited by increasing distances, which immediately implies the correctness of the algorithm.

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 $\mathcal{H}$, the 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 $\mathcal{O}(n/\mu + m/B)$ 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 amortized $\mathcal{O}(\mu \log W/B)$ I/Os per edge, provided that the cluster partition has certain properties, discussed in the next section. A partition with these properties can be computed in $\mathcal{O}(MST(n, m) + (n/B)\log W)$ I/Os. By using a priority queue that exploits the bounded range of the edge lengths to support Update, Delete, and DeleteMin operations in amortized $\mathcal{O}((1/B)\log W)$ I/Os, the complexity of the algorithm thus becomes $\mathcal{O}(n/\mu + (m\mu \log W)/B + MST(n, m))$ I/Os, which is $\mathcal{O}(\sqrt{nm \log W/B} + MST(n, m))$ if $\mu = \sqrt{nB/(m \log W)}$ is chosen.

3 A Length-Independent MZ-SSSP

This section presents a new implementation of MZ-SSSP, called MZ-SSSP*. The cost of MZ-SSSP* is $\mathcal{O}((n/\mu)\log n + m(\mu d + \log n)/B)$ I/Os, where d is a parameter of the used cluster partition, called the *depth* of the partition. Section 5 will be concerned with ensuring that $d = \mathcal{O}(\log n)$, which, after choosing $\mu = \sqrt{nB/m}$, leads to the desired complexity of $\mathcal{O}(\sqrt{nm/B}\log n)$ I/Os, plus the cost for computing the partition, which will be $\mathcal{O}(MST(n,m))$ I/Os.

3.1 μ-Partitions

The efficient implementation of the hot pool in MZ-SSSP* requires that the used cluster partition has a number of properties. For an edge $e \in G$, the *category* of e is the integer c such that $2^{c-1} \le \ell(e) < 2^c$. 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. The corresponding edge cluster E_i is also referred to as a c-cluster. The *diameter* of a vertex set V' is equal to $\max\{\mathrm{dist}_G(x,y) \mid x,y \in V'\}$. Now the partition required by MZ-SSSP* is a *μ-partition* of G, which is a partition of V into $q = \mathcal{O}(n/\mu)$ vertex clusters $V_1, \ldots, V_q$ with the following properties:

(C1) Every cluster V_i contains at most μ 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 *depth* of a cluster is 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 *(μ, d)-partition* is a μ-partition of depth d. Note that $d \le \log W$.

Even though every edge with exactly one endpoint in a c-cluster of a (μ, d)-partition has category at least $c-d$, edges between vertices in the same c-cluster may be arbitrarily short. A *mini-cluster* is a $(c-d)$-component contained in a c-cluster. (Note that, in a (μ, d)-partition, any $(c-d)$-component is either completely contained in or disjoint from a given c-cluster.) Mini-clusters have to be treated specially, in order to ensure correctness of the algorithm.

The hot pool also 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, the *component tree* T_c of G needs to be defined first: A c-component is *maximal* if it is properly contained in a $(c+1)$-component or it is equal to G; all vertices of G are maximal 0-components. The vertex set of T_c consists of all maximal 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 Visit$(x, d(x))$ signal into the hot pool to induce the relaxation of all edges incident to x. Before releasing x from Q, a Scan operation is invoked on the hot pool to ensure that all edges that need to be relaxed before releasing x are relaxed. This operation is described in the next section, which discusses the implementation of the hot pool.

Our algorithm uses the same priority queue as KS-SSSP. It also requires a distance repository REP, which stores the tentative distances of all vertices in G, maintained using Update$(x, d(x))$ operations, and allows the retrieval of the tentative distances of all nodes in a cluster tree T_i using a ClusterQuery(T_i) operation. Every edge relaxation in our algorithm performs an update on the priority queue *and* on the repository.

The repository can be implemented as an augmented buffered repository tree (BRT) [4], which supports Update$(x, d(x))$ operations in amortized $\mathcal{O}(\log n/B)$ I/Os and ClusterQuery(T_i) operations in amortized $\mathcal{O}((1 + r_i) \log n + |T_i|/B)$ I/Os, where r_i is the number of cluster tree roots that are contained in or adjacent to T_i. It is easy to prove that $\sum_{i=1}^{q} r_i = \mathcal{O}(n/\mu)$. Details appear in the full paper.

3.3 Hot Pool

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 consecutive buffers. Buffers EB_i, TB_i, and SB_i form *level* i of the hot pool.

The edge buffers hold edges that have been loaded into the hot pool. Edge buffer EB_{i+1} is inspected by Scan operations about half as often as EB_i. If an edge xy is stored in EB_i, then TB_i stores all ancestors of x in T_c that have category at most i. The signal buffers store signals that are 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* $t_1 \leq t_2 \leq \cdots \leq t_{r+1} = +\infty$ associated with these buffers. These due times satisfy the following condition:

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

The due times are initialized as $t_i = \min\{w(x) \mid x \in G\} + 2^{i-2}$, for $1 \leq i \leq r$. Due time t_i indicates that EB_i has to be inspected for edges to be relaxed or moved to lower buffers before the first vertex with tentative distance $d(x) \geq t_i$ is released from Q. The hot pool maintains the following invariant:

(HP) After loading a c-edge cluster E_j into the hot pool, an edge $xy \in E_j$ is stored in the lowest edge buffer EB_i such that $i \geq c - d$ 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_j in the tree buffer TB_i, and this copy of C in TB_i always stores the correct value of $d(C) = \min\{d(x) \mid x \in C\}$.[3] To achieve this, an $\mathrm{Update}(y, d(x) + \ell(xy))$ signal is inserted 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 a $\mathrm{Visit}(x, d(x))$ signal into the hot pool to trigger the relaxation of edges incident to x. This signal is inserted into SB_{c_x}, where $c_x = c - d$ if x is contained in a c-cluster. From there, it travels only up to level $c + O(\log n)$ and is then discarded. In order to achieve this insertion into SB_{c_x} without performing a random access, the signal *is sent to level* c_x by inserting it, with priority c_x, into a priority queue SQ^+. Priority queue SQ^+ is used to send signals to higher levels. Another priority queue SQ^- is used to send signals to lower levels.

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. Since f is the priority of the next vertex to be released from Q, edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ such that $t_1 \leq \cdots \leq t_j \leq f < t_{j+1}$ need to be scanned. The scanning of an edge buffer EB_i may decrease f. Then the *updated* value of f is used to decide whether to include EB_{i+1} in the scan.

The Scan operation can be divided into two phases: The *up-phase* inspects edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$, relaxes edges, and moves edges whose relaxation is not imminent to higher levels. The *down-phase* 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:

- Retrieve all signals sent to level i from SQ^+ and insert them into SB_i.
- For every $\mathrm{Visit}(x, d(x))$ signal in SB_i such that $x \notin \mathrm{TB}_i$, load the edge cluster E_h containing $E(x)$ into EB_i; load the corresponding cluster tree T_h into TB_i and retrieve the tentative distances of all nodes in T_h from REP. E_h is loaded only once, even if more than one vertex in V_h is to be visited.
- For every cluster tree node C in TB_i and every $\mathrm{Update}(x, d)$ signal in SB_i such that $x \in C$, replace $d(C)$ with $\min(d(C), d)$.
- For every $\mathrm{Visit}(x, d(x))$ signal in 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 $\mathrm{Update}(y, d(x) + \ell(xy))$ signal to level $\max(i+1, c - \log n - d - 1)$ (using SQ^+) and, if $c \leq i + \log n + d + 1$, to level i (using SQ^-). Such an Update signal is said to have category c.
- Move all cluster tree nodes C in TB_i to TB_{i+1} for which either the category of C is greater than i or the tentative distance of the i-component containing

[3] This is not quite correct; C stores only an upper bound $d^*(C) \geq d(C)$; but this upper bound suffices to move edges to lower buffers in time for their relaxation.

C is at least t_{i+1}. For every cluster tree leaf (vertex) x moved to TB_{i+1}, move $E(x)$ from EB_i to EB_{i+1}.
- Move update signals with category greater than $i - d$ to SB_{i+1}. Discard all other signals in SB_i.
- Test whether $f \geq t_{i+1}$ 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).
- Retrieve all signals sent to level i from SQ^- and insert them into SB_i. At this point, they will all be Update signals sent during scans of higher levels. As in the up-phase, apply these signals to the nodes stored in TB_i.
- Move all cluster tree nodes C in TB_i to TB_{i-1} for which 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 TB_{i-1}, move $E(x)$ from EB_i to EB_{i-1}.
- Move all signals of category less than $i + \log n + d + 1$ to SB_{i-1}. Discard all other signals in SB_i.

To implement these different steps efficiently, the nodes in T_c are numbered in preorder. The algorithm then keeps the cluster tree nodes in TB_i sorted by their preorder numbers, the signals in SB_i sorted by the preorder numbers of the vertices they affect, and the edges in EB_i sorted by the preorder numbers of their first endpoints. It is easy to show then that both phases can be implemented by scanning the involved buffers a constant number of times, except that the signals retrieved from SQ^+ and SQ^- have to be sorted before merging them into SB_i.

We show in the full paper that due times of empty levels can be represented implicitly using the due times of the two closest non-empty levels. This avoids spending I/Os on accessing due times of empty levels. Accesses to due times of non-empty levels can be charged to accesses to elements in these levels.

Processing mini-clusters: The processing of a mini-cluster C involves visiting all vertices in C that have $\mathrm{Visit}(x, d(x))$ 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 other vertices in the same mini-cluster. In particular, starting with their current tentative distances, we apply a bounded version of Dijkstra's algorithm to 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, the tentative distances of all vertices in the mini-cluster that have not been visited are updated, that is, for each such vertex, $\mathrm{Update}(x, d(x))$ operations are performed on Q and REP, and $d(x)$ is updated in TB_i. For every visited vertex x and every category-c edge xy in $E(x)$ with $c \geq i$, $\mathrm{Update}(y, d(x)+\ell(xy))$ signals are sent to the levels specified in the discussion of the up-phase. Finally, a $\mathrm{Delete}(x)$ 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, vertices not yet released from Q may be visited.

3.4 Analysis

The lengthy and technical correctness proof of our algorithm is omitted from this extended abstract due to lack of space. The main idea is to prove the following lemma, which immediately implies the algorithm's correctness.

Lemma 1. *MZ-SSSP* has property (SP).*

The key to proving this is the following lemma.

Lemma 2. *A vertex x visited during a scan of level i satisfies $t_i - 2^{i-2} \leq d(x) \leq d^*(x) \leq t_i$, where $d^*(x)$ is the tentative distance stored with x in* TB_i.

From Lem. 2, Property (SP) follows almost immediately, ignoring a few technical details: Consider a vertex y that is visited before the current scan of level i. Then one can show that this vertex satisfies $d(y) < t_i$ because otherwise, level i would have been scanned before visiting y. Thus, if x and y do not belong to the same mini-cluster, then, because the path from x to y must include a category-i edge and by Lem. 2, $d(y) \leq d(x) + \mathrm{dist}_G(x, y)/2$, which implies Property (SP). If x and y belong to the same mini-cluster, it can be shown that they are visited by increasing tentative distances, that is, $d(y) \leq d(x)$, which again implies Property (SP).

The key to the analysis of the I/O-complexity is to prove that the hot pool maintains Property (HP), which we do in the full paper. Given this, we obtain

Lemma 3. *Excluding the cost of computing the (μ, d)-partition, MZ-SSSP* performs $\mathcal{O}((n/\mu)\log n + m(\mu d + \log n)/B)$ I/Os.*

Proof sketch. Observe that the algorithm performs $\mathcal{O}(m)$ priority queue operations and Update operations on REP. All these operations have an amortized cost of $\mathcal{O}((1/B)\log n)$ I/Os, which gives a cost of $\mathcal{O}((m/B)\log n)$ I/Os for these operations. Only $\mathcal{O}(m)$ signals are sent to the different levels of the hot pool, which costs $\mathcal{O}(\mathrm{sort}(m))$ I/Os for the involved operations on SQ^+ and SQ^- and for 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 results in a cost of $\mathcal{O}(n/\mu + m/B)$ I/Os for reading edge clusters and cluster trees, plus $\mathcal{O}((n/\mu)\log n + n/B)$ I/Os for answering cluster queries on REP. (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 at most $\mathcal{O}(\mu)$ scans of this level. This implies a cost of $\mathcal{O}((m/B)(d + \log n + 2))$ I/Os for scanning the signals and $\mathcal{O}(md\mu/B)$ I/Os for scanning edges and cluster tree nodes. Summing up the different costs proves the lemma.

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. Finally, Property (HP) implies immediately that every cluster is loaded only once because an edge xy, once loaded, reaches level c_x in time for its relaxation. □

4 A Recursive Shortest-Path Algorithm

This section describes a CSSP-algorithm that uses in a sense the exact opposite of a μ-partition of low depth. Section 4.1 defines the partition required by the algorithm. Section 4.2 shows that shortest paths in the whole graph can be computed by solving nearly independent CSSP-problems on the graphs in the partition. This section proves only the correctness of the algorithm. Its complexity is analyzed in Sect. 5, where it is combined with MZ-SSSP* to obtain the final algorithm.

4.1 Barrier Decomposition

The algorithm uses a *barrier decomposition* of G, which consists of a number of multigraphs $G_0, \ldots, G_q$ and vertex sets $\emptyset = B_0, \ldots, B_q$, 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 $H_k \supset H_j$, G_i is 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 $r(G_j)$, called the *representative* of G_j. For a vertex $x \in H_j$, $r(G_j)$ is considered the representative of x in G_i and denoted 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

Now assume that a Dijkstra-like CSSP algorithm $\mathcal{A}$ is given, 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, the CSSP problem in G can then be solved using the following recursive algorithm. The algorithm requires the use of the distance repository REP from Sect. 3, 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 $d(x) = \min\{d(y) \mid y \in H_j\}$. In the full paper, we show how to perform such an operation in amortized $\mathcal{O}((1 + c_i) \log n + |V(G_i)|/B)$ I/Os, where c_i is the number of children of G_i.

ShortestPaths(G_i)**:** Run a modified version of algorithm $\mathcal{A}$ on the 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 necessary 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 each such edge xy, 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 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 4. *For every vertex $x \in H_i \cup B_i$ visited by ShortestPaths(G_i), $d(x) = D(x)$ 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), $D(y) \geq D(r_y) \geq D(x) - \text{dist}_{G_i \cup B_i}(r_y, x)/2 \geq D(x) - \text{dist}_G(y, x)/2$, 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 $D_{H_j \cup B_j}(x) = D(x)$. If $\pi(x) \subseteq H_j \cup B_j$, this is trivial. So assume that $\pi(x)$ 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 $r = r(G_j)$ must be visited before r_z, that is, by (SP), $D(r) \leq D(r_z) + \text{dist}_{G_i \cup B_i}(r_z, r)/2$. However, $D(u) < D(r) + n \cdot \ell_{\max}(H_j) \leq D(r) + \text{dist}_{G_i \cup B_i}(r_z, r)/2$, for all $u \in H_j$, because $z \notin H_j \cup B_j$. Hence, $D(u) < D(r_z) + \text{dist}_{G_i \cup B_i}(r_z, r) \leq D(z) + \text{dist}_G(z, u)$. 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 $D_{H_j \cup B_j}(x) \leq D_{H_j \cup B_j}(u)$, for some $u \in H_j$. Since $D(x) \leq D_{H_j \cup B_j}(x)$ and $D_{H_i \cup B_i}(u) = D(u)$, this implies again that $z \notin \pi(x)$, a contradiction. □

5 The Final Algorithm

Our final algorithm is based on the recursive framework of Sect. 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 claimed I/O-complexity, the following properties of the barrier decomposition are required:

(P1) The barrier decomposition consists of $\mathcal{O}(n/\mu)$ 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, it is called *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 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, namely $r(G_j)$. This implies in particular that $|B_j| \leq \mu \leq \sqrt{B}$, for all j.

In the full paper, we prove the following lemma.

Lemma 5. *It takes $\mathcal{O}(MST(n, m))$ I/Os to compute a barrier decomposition of an undirected graph G that has properties (P1)–(P3).*

In a nutshell, such a decomposition can be obtained as follows: In [9], a procedure is described that computes a μ-partition of a graph in $\mathcal{O}(MST(n, m) + (n/B) \log W)$ I/Os, by computing a minimum spanning tree T and then computing c-clusters iteratively using $\log W$ scans of an Euler tour of T. Using an algorithm from [3], the component tree T_c can be computed from T in $\mathcal{O}(\text{sort}(n))$ I/Os; the $\log W$ scans of the Euler tour can then be simulated in $\mathcal{O}(\text{sort}(n))$ I/Os using one traversal of T_c. Once this μ-partition is given, two more traversals of T_c are needed. The first one refines the partition so that all clusters of depth greater than $\log n + 2$ have a particularly simple structure. The second one splits each of these deep clusters into three parts: a top, middle, and bottom part, which correspond to top, middle, and bottom parts of its cluster tree. The top and bottom parts define clusters in $(\mu, \log n + 2)$-partitions of two graphs G_i and G_k in the barrier decomposition. The middle part G_j defines an atomic graph in the barrier decomposition that is a child of G_i and the parent of G_k.

By Lem. 5, it takes $\mathcal{O}(MST(n, m))$ 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 $\mathcal{O}(((n_i + |B_i|)/\mu) \log n + (m_i \mu \log n)/B)$ 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 G_i is atomic, Dijkstra's algorithm can be used to solve CSSP in G_i, which incurs $\mathcal{O}(1 + m_i/B)$ I/Os because G_i fits in memory.

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

The cost of all repository operations can be bounded as follows: The algorithm performs exactly one subgraph query per graph G_i and at most two cluster queries per cluster: one when the cluster is loaded into the hot pool and another one when the graph representative $r(G_j)$ in the cluster is visited. Moreover, it is easy to show that the sum of the r_j and c_i is $\mathcal{O}(n/\mu)$, so that the cost of all queries on the repository is $\mathcal{O}((n/\mu) \log n + n/B) = \mathcal{O}((n/\mu) \log n)$ I/Os. Since the algorithm performs only $\mathcal{O}(m)$ edge relaxations, the cost of all Update operations on the repository is $\mathcal{O}((m/B) \log n)$ I/Os.

Summing the costs of all parts of the algorithm yields an I/O-complexity of $\mathcal{O}((n/\mu) \log n + (m\mu \log n)/B + MST(n, m))$, which is $\mathcal{O}(\sqrt{nm/B} \log n + MST(n, m))$ for $\mu = \sqrt{nB/m}$. This proves Thm. 1.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, pp. 1116–1127, 1988.
2. L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Alg.*, 53(2):186–206, 2004.
3. L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proc. 15th SPAA*, pp. 85–93, 2003.
4. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th SODA*, pp. 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 *Proc. 6th SODA*, pp. 139–149, 1995.
6. E. W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.
7. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th SPDP*, pp. 169–176, 1996.
8. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th ESA*, LNCS 2461, pp. 723–735. Springer-Verlag, 2002.
9. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. 11th ESA*, LNCS 2832, pp. 434–445. Springer-Verlag, 2003.
10. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded weights. Tech. Report CS-2006-04, Faculty of Comp. Sci., Dalhousie Univ., 2006.
11. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. 13th SODA*, pp. 267–276, 2002.
12. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.
13. M. Thorup. Floats, integers, and single source shortest paths. *J. Alg.*, 35:189–201, 2000.