Alcom-FT Technical Report Series
ALCOMFT-TR-03-61

I/O-Efficient Undirected Shortest Paths

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

[1] Max-Planck-Institut für Informatik, Stuhlsatzhausenweg 85, 66123 Saarbrücken, Germany.
[2] Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Canada.

Abstract. We present an I/O-efficient algorithm for the single-source shortest path problem on undirected graphs $G = (V, E)$. Our algorithm performs $\mathcal{O}(\sqrt{(VE/B)\log_2(W/w)} + \mathrm{sort}(V + E)\log\log(VB/E))$ I/Os[1], where $w \in \mathbb{R}^+$ and $W \in \mathbb{R}^+$ are the minimal and maximal edge weights in G, respectively. For uniform random edge weights in $(0, 1]$, the expected I/O-complexity of our algorithm is $\mathcal{O}(\sqrt{VE/B} + ((V + E)/B)\log_2 B + \mathrm{sort}(V + E))$.

1 Introduction

The *single-source shortest path (SSSP) problem* is a fundamental combinatorial optimization problem with numerous applications. It is defined as follows: Let G be a graph, let s be a distinguished vertex of G, and let ω be an assignment of non-negative real *weights* to the edges of G. The weight of a path is the sum of the weights of its edges. We want to find for every vertex v that is reachable from s, the weight $\mathrm{dist}(s, v)$ of a minimum-weight ("shortest") path from s to v.

The SSSP-problem is well-understood as long as the whole problem fits into *internal memory*. For larger data sets, however, classical SSSP-algorithms perform poorly, at least on sparse graphs: Due to the unpredictable order in which vertices are visited, the data is moved frequently between fast internal memory and slow *external memory*; the I/O-communication becomes the bottleneck.

I/O-model and previous results. We work in the standard I/O-model with one (logical) disk [1]. This model defines the following parameters:[2] N is the number of vertices and edges of the graph ($N = V + E$), M is the number of vertices/edges that fit into internal memory, and B is the number of vertices/edges that fit into a disk block. We assume that $2B < M < N$. In an *Input/Output* operation (or *I/O* for short), one block of data is transferred between disk and internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\mathrm{scan}(N) = \Theta(N/B)$. The number of I/Os required to

⋆ Partially supported by EU programme IST-1999-14186 and DFG grant SA 933/1-1.
⋆⋆ Part of this work was done while visiting the Max-Planck-Institut in Saarbrücken.
[1] $\mathrm{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ is the I/O-complexity of sorting N data items.
[2] We use V and E to denote the vertex and edge sets of G as well as their sizes.

sort N items is $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ [1]. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$.

External-memory graph algorithms have received considerable attention in recent years; see the surveys of [10, 13]. Despite these efforts, only little progress has been made on the SSSP-problem: The best known lower bound is $\Omega(\text{sort}(V+E))$ I/Os, while the currently best algorithm, by Kumar and Schwabe [8], performs $\mathcal{O}(V + (E/B)\log_2(V/B))$ I/Os. For $E = \mathcal{O}(V)$, this is hardly better than naïvely running Dijkstra's internal-memory algorithm [6, 7] in external memory, which would take $\mathcal{O}(V \log_2 V + E)$ I/Os. Improved external-memory SSSP-algorithms exist for restricted graph classes such as planar graphs, grid graphs, and graphs of bounded treewidth; see [14] for an overview.

A number of improved internal-memory SSSP-algorithms have been proposed for bounded integer/float weights or bounded ratio W/w, where $w \in \mathbb{R}^+$ and $W \in \mathbb{R}^+$ are the minimal and maximal edge weights in G, respectively; see [11, 12] for an overview. For $W/w = 1$, SSSP becomes *breadth-first search* (BFS). A simple extension of the first $o(V)$-I/O algorithm for undirected BFS [9] yields an SSSP-algorithm that performs $\mathcal{O}(\sqrt{VEW/B} + W \cdot \text{sort}(V+E))$ I/Os for integer weights in $[1, W]$. Obviously, W must be significantly smaller than B for this algorithm to be efficient. Furthermore, the algorithm requires that $BW < M$.

The only paper addressing the *average-case* I/O-complexity of SSSP [5] is restricted to random graphs with random edge weights. It reduces the I/O-complexity exclusively by exploiting the power of independent parallel disks; on a single disk, the performance of the algorithm is no better than that of [8].

New Results. We propose a new SSSP-algorithm for undirected graphs. The I/O-complexity of our algorithm is $\mathcal{O}(\sqrt{(VE/B)\log_2(W/w)} + \text{sort}(V+E))$ with high probability or $\mathcal{O}(\sqrt{(VE/B)\log_2(W/w)} + \text{sort}(V+E)\log\log(VB/E))$ deterministically, where $w \in \mathbb{R}^+$ and $W \in \mathbb{R}^+$ are the minimal and maximal edge weights in G, respectively. Compared to the solution of [9], the new algorithm exponentially increases the range of efficiently usable edge weights, while only requiring that $M = \Omega(B)$.[3] These results hold for arbitrary graph structures and edge weights between w and W. For uniform random edge weights in $(0, 1]$, the average-case I/O-complexity of our algorithm reduces to $\mathcal{O}(\sqrt{VE/B} + ((V+E)/B)\log_2 B + \text{sort}(V+E))$. For sparse graphs, this matches the I/O-bound of the currently best BFS-algorithm.

2 Preliminaries and Outline

As previous I/O-efficient SSSP-algorithms [8, 9], our algorithm is an I/O-efficient version of Dijkstra's algorithm [6]. Dijkstra's algorithm uses a priority queue Q to store all vertices of G that have not been settled yet (a vertex is said to be *settled* when its distance from s has been determined); the priority of a vertex v in Q is the length of the currently shortest known path from s to v. Vertices

[3] In this extended abstract, we assume that $M = \Omega(B \log_2(W/w))$, to simplify the exposition.

are settled one-by-one by increasing distance from s. The next vertex v to be settled is retrieved from Q using a DeleteMin operation. Then the algorithm relaxes the edges between v and all its non-settled neighbors, that is, performs a DecreaseKey$(w, \mathrm{dist}(s,v)+\omega(v,w))$ operation for each such neighbor w whose priority is greater than $\mathrm{dist}(s,v)+\omega(v,w)$.

An I/O-efficient version of Dijkstra's algorithm has to (a) avoid accessing adjacency lists at random, (b) deal with the lack of optimal DecreaseKey operations in current external-memory priority queues, and (c) efficiently remember settled vertices. The previous SSSP-algorithms of Kumar and Schwabe [8], **KS** for short, and Mehlhorn and Meyer [9], **MM**, address these issues as follows: **KS** ignores (a) and spends $\Omega(1)$ I/Os on retrieving the adjacency list of each settled vertex. **MM**, on the other hand, forms clusters of vertices and loads the adjacency lists of all vertices in a cluster into a "hot pool" of edges as soon as the first vertex in the cluster is settled. In order to relax the edges incident to settled vertices, the hot pool is scanned and all relevant edges are relaxed.

As for (b), **KS** uses a tournament tree, whereas **MM** applies a cyclic bucket queue composed of $2W+1$ lists. Both support batched processing and emulate Insert and DecreaseKey operations using a weaker Update operation, which decreases the priority of the element if it is already stored in the priority queue and otherwise inserts the element into the priority queue.

As for (c), **KS** performs an Update operation for *every* neighbor of a settled vertex, which eliminates the need to remember previously settled vertices, but may re-insert settled vertices into the priority queue Q. Kumar and Schwabe call the latter a *spurious update*. Using a second priority queue Q^*, these re-inserted vertices are removed from Q before they can be settled for a second time.[4] In contrast, **MM** deals with (c) by using half of its lists to identify settled vertices; Update operations are performed only for non-settled vertices.

Our new approach inherits ideas from both algorithms: As **KS**, we use a second priority queue to eliminate the effect of spurious updates. But we replace the tournament tree used by **KS** with a hierarchical bucket queue (Section 3), which, in a way, is an I/O-efficient version of the integer priority queue of [2]. Next we observe that the relaxation of edges of large weight can be delayed because if such an edge is on a shortest path, it takes some time before its other endpoint is settled. Hence, we extend **MM**'s combination of clustering and hot pools to use a hierarchy of hot pools and gather long edges in hot pools that are touched much less frequently than the pools containing short edges. As we show in the full paper, already this idea alone works well on graphs with random edge weights. We obtain a worst-case guarantee for the I/O-complexity of our algorithm by storing even short edges in pools that are touched infrequently; we shift these edges to pools that are touched more and more frequently the closer the time of their relaxation draws. To make this work, we form clusters in a locality-preserving manner, essentially guaranteeing that a vertex is closer to its neighbors in the same cluster than to its neighbors in other clusters (Section 4.1).

[4] The algorithm of [8] does not handle adjacent vertices with the same distance from s correctly. In the full paper, we provide a correct solution to this problem.

To predict the time of relaxation of every edge during the shortest path phase of the algorithm (Section 4.2), we use an explicit representation of the structure of each cluster, which is computed during the clustering phase.

Our clustering approach is similar to the one used in Thorup's linear-time SSSP-algorithm [12]. However, the precise definition of the clusters and their use during the shortest path phase of the algorithm differ from Thorup's, mainly because our goals are different. While we try to avoid random accesses by clustering nodes and treating their adjacency lists as one big list, Thorup's goal is to beat the sorting bound inherent in Dijkstra's algorithm by relaxing the order in which vertices are visited. Arguably, this makes the order in which the vertices are visited even more random.

3 An Efficient Batched Integer Priority Queue

In this section, we describe a simple batched integer priority queue Q, which can be seen as an I/O-efficient version of the integer priority queue of [2]. It supports UPDATE, DELETE, and BATCHEDDELETEMIN operations. The first two operations behave as on a tournament tree; the latter retrieves *all* elements with minimal priority from Q. For the correctness of our data structure, the priority of an inserted or updated element has to be greater than the priority of the elements retrieved by the last BATCHEDDELETEMIN operation. Let C be a bound so that, at all times, the difference between the minimum and maximum priorities of the elements in Q is at most C. Then Q supports the above operations in $\mathcal{O}((\log_2 C + \log_{M/B}(N/B))/B)$ I/Os amortized.

Q consists of $r = 1 + \log_2 C$ buckets. Each such bucket is represented by two sub-buckets $\mathcal{B}_i$ and $\mathcal{U}_i$. The buckets are defined by splitter elements $s_0 \leq s_1 \leq \cdots \leq s_r = \infty$. Every entry (x, p_x) in $\mathcal{B}_i$, representing an element x with priority p_x, satisfies $s_{i-1} \leq p_x < s_i$. Initially, we set $s_0 = 0$ and, for $1 \leq i < r$, $s_i = 2^{i-1}$. We refer to $s_i - s_{i-1}$ as the *size* of bucket $\mathcal{B}_i$. These bucket sizes may change during the algorithm; but we enforce that, at all times, bucket $\mathcal{B}_1$ has size at most 1, and bucket $\mathcal{B}_i$, $1 < i < r$, has size 0 or a size between $2^{i-2}/3$ and 2^{i-2}. We use buckets $\mathcal{U}_1, \ldots, \mathcal{U}_r$ to perform updates in a batched manner. In particular, bucket $\mathcal{U}_i$ stores updates to be performed on buckets $\mathcal{B}_i, \ldots, \mathcal{B}_r$.

An UPDATE or DELETE operation inserts itself into $\mathcal{U}_1$, augmented with a time stamp. A BATCHEDDELETEMIN operation reports the contents of $\mathcal{B}_1$, after filling it with elements from $\mathcal{B}_2, \ldots, \mathcal{B}_r$ as follows: We iterate over buckets $\mathcal{B}_1, \ldots, \mathcal{B}_i$, applying the updates in $\mathcal{U}_1, \ldots, \mathcal{U}_i$ to $\mathcal{B}_1, \ldots, \mathcal{B}_i$, until we find the first bucket $\mathcal{B}_i$ that is non-empty after these updates. We split the priority interval of $\mathcal{B}_i$ into intervals for $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$, assign an empty interval to $\mathcal{B}_i$, and distribute the elements of $\mathcal{B}_i$ over $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$, according to their priorities.

To incorporate the updates in $\mathcal{U}_1, \ldots, \mathcal{U}_i$ into $\mathcal{B}_1, \ldots, \mathcal{B}_i$, we sort the updates in $\mathcal{U}_1$ by their target elements and time stamps and repeat the following for $1 \leq j \leq i$: We scan $\mathcal{U}_j$ and $\mathcal{B}_j$, to update the contents of $\mathcal{B}_j$. If a deletion in $\mathcal{U}_j$ matches an existing element in $\mathcal{B}_j$, we remove this element from $\mathcal{B}_j$. If an UPDATE(x, p_x) operation in $\mathcal{U}_j$ matches an element (x, p'_x) in $\mathcal{B}_j$ and $p_x < p'_x$, we replace (x, p'_x) with (x, p_x) in $\mathcal{B}_j$. If element x is not in $\mathcal{B}_j$, but $s_{j-1} \le p_x < s_j$, we insert (x, p_x) into $\mathcal{B}_j$. If there are UPDATE and DELETE operations matching the same element in $\mathcal{B}_j$, we decide by the time stamps which action is to be taken. After these updates, we copy appropriate entries to $\mathcal{U}_{j+1}$, maintaining their sorted order: We scan $\mathcal{U}_j$ and $\mathcal{U}_{j+1}$ and insert every UPDATE(x, p_x) operation in $\mathcal{U}_j$ with $p_x \ge s_j$ into $\mathcal{U}_{j+1}$; for every DELETE(x) or UPDATE(x, p_x) operation with $p_x < s_j$, we insert a DELETE(x) operation into $\mathcal{U}_{j+1}$. (The latter ensures that UPDATE operations do not re-insert elements already in Q.)

To compute the new priority intervals for $\mathcal{B}_1, \dots, \mathcal{B}_{i-1}$, we scan $\mathcal{B}_i$ and find the smallest priority p of the elements in $\mathcal{B}_i$; we define $s_0 = p$ and, for $1 \le j \le i-1$, $s_j = \min\{p + 2^{j-1}, s_i\}$. Note that every $\mathcal{B}_j$, $1 \le j < i$, of non-zero size has size 2^{j-2}, except the last such $\mathcal{B}_h$, whose size can be as small as 1. If the size of $\mathcal{B}_h$ is less than $2^{h-2}/3$, we redefine $s_{h-1} = s_h - 2^{h-2}/3$; this increases the size of $\mathcal{B}_h$ to $2^{h-2}/3$ and leaves $\mathcal{B}_{h-1}$ with a size between $2^{h-3}/3$ and 2^{h-3}.

To distribute the elements of $\mathcal{B}_i$ over $\mathcal{B}_1, \dots, \mathcal{B}_{i-1}$, we repeat the following for $j = i, i-1, \dots, 2$: We scan $\mathcal{B}_j$, remove all elements that are less than s_{j-1} from $\mathcal{B}_j$, and insert them into $\mathcal{B}_{j-1}$.

The I/O-complexity of an UPDATE or DELETE operation is $\mathcal{O}(1/B)$ amortized, because these operations only insert themselves into $\mathcal{U}_1$. To analyze the I/O-complexity of a BATCHEDDELETEMIN operation, we observe that every element is involved in the sorting of $\mathcal{U}_1$ only once; this costs $\mathcal{O}(\log_{M/B}(N/B)/B)$ I/Os amortized per element. When filling empty buckets $\mathcal{B}_1, \dots, \mathcal{B}_{i-1}$ with the elements of $\mathcal{B}_i$, every element in $\mathcal{B}_i$ moves down at least one bucket and will never move to a higher bucket again. If an element from $\mathcal{B}_i$ moves down x buckets, it is scanned $1 + x$ times. Therefore, the total number of times an element from $\mathcal{B}_1, \dots, \mathcal{B}_r$ can be scanned before it reaches $\mathcal{B}_1$ is at most $2r = \mathcal{O}(\log_2 C)$. This costs $\mathcal{O}((\log_2 C)/B)$ I/Os amortized per element.

Emptying bucket $\mathcal{U}_i$ involves the scanning of buckets $\mathcal{U}_i, \mathcal{U}_{i+1}$, and $\mathcal{B}_i$. In the full paper, we prove that every element in $\mathcal{U}_i$ and $\mathcal{U}_{i+1}$ is involved in at most two such emptying processes of $\mathcal{U}_i$ before it moves to a higher bucket; every element in $\mathcal{B}_i$ is involved in only one such emptying process before it moves to a lower bucket. By combining our observations that every element is involved in the sorting of $\mathcal{U}_1$ at most once and that every element is touched only $\mathcal{O}(1)$ times per level in the bucket hierarchy, we obtain the following lemma.

Lemma 1. *There exists an integer priority queue Q that processes a sequence of N UPDATE, DELETE, and BATCHEDDELETEMIN operations in $\mathcal{O}(\text{sort}(N) + (N/B)\log_2 C)$ I/Os, where C is the maximal difference between the priorities of any two elements stored simultaneously in the priority queue.*

The following lemma, proved in the full paper, follows from the lower bound on the sizes of non-empty buckets. It is needed by our shortest path algorithm.

Lemma 2. *Let p be the priority of the entries retrieved by a BATCHEDDELETEMIN operation, and consider the sequence of all subsequent BATCHEDDELETEMIN operations that empty buckets $\mathcal{B}_h$, $h \ge i$, for some $i \ge 2$. Let $p_1, p_2, \dots$ be the priorities of the entries retrieved by these operations. Then $p_j - p \ge (j-4)2^{i-2}/3$.*

Note that we do not use that the priorities of the elements in Q are integers. Rather, we exploit that, if these priorities are integers, then $p > p'$ implies $p \geq p'+1$, which in turn implies that after removing elements from $\mathcal{B}_1$, all subsequent insertions go into $\mathcal{B}_2, \ldots, \mathcal{B}_r$. Hence, we can also use Q for elements with real priorities, as long as BatchedDeleteMin operations are allowed to produce a weaker output and Update operations satisfy a more stringent constraint on their priorities. In particular, it has to be sufficient that the elements retrieved by a BatchedDeleteMin operation include all elements with smallest priority $p_{\min}$, their priorities are smaller than the priorities of all elements that remain in Q, and the priorities of any two retrieved elements differ by at most 1. Every subsequent Update operation has to have priority at least $p_{\min} + 1$.

4 The Shortest Path Algorithm

Similar to the BFS-algorithm of [9], our algorithm consists of two phases: The *clustering phase* computes a partition of the vertex set of G into $o(V)$ vertex clusters $V_1, \ldots, V_q$ and groups the adjacency lists of the vertices in these clusters into *cluster files* $\mathcal{F}_1, \ldots, \mathcal{F}_q$. During the *shortest path phase*, when a vertex v is settled, we do not only retrieve its adjacency list but the whole cluster file from disk and store it in a collection of hot pools $\mathcal{H}_1, \ldots, \mathcal{H}_r$. Thus, whenever another vertex in the same cluster as v is settled, it suffices to search the hot pools for its adjacency list. Using this approach, we perform only one random access per cluster instead of performing one random access per vertex to retrieve adjacency lists. The efficiency of our algorithm depends on how efficiently the edges incident to a settled vertex can be located in the hot pools and relaxed.

In Section 4.1, we show how to compute a *well-structured cluster partition*, whose properties help to make the shortest path phase, described in Section 4.2, efficient. In Section 4.3, we analyze the average-case complexity of our algorithm.

4.1 The Clustering Phase

In this section, we define a well-structured cluster partition $\mathcal{P} = (V_1, \ldots, V_q)$ of G and show how to compute it I/O-efficiently. We assume w.l.o.g. that the minimum edge weight in G is $w = 1$. We group the edges of G into $r = \lceil \log_2 W \rceil$ *categories* so that the edges in category i have weight between 2^{i-1} and 2^i. The category of a vertex is the minimum of the categories of its incident edges. Let $G_0, \ldots, G_r$ be a sequence of graphs defined as $G_0 = (V, \emptyset)$ and, for $1 \leq i \leq r$, $G_i = (V, E_i)$ with $E_i = \{e \in E : e \text{ is in category } j \leq i\}$. We call the connected components of G_i *category-i components*. The category of a cluster V_j is the smallest integer i so that V_j is completely contained in a category-i component. The *diameter* of V_j is the maximal distance in G between any two vertices in V_j. For some $\mu \geq 1$ to be fixed later, we call $\mathcal{P} = (V_1, \ldots, V_q)$ *well-structured* if (P1) $q = \mathcal{O}(V/\mu)$, (P2) no vertex v in a category-i cluster V_j has an incident category-k edge (v, u) with $k < i$ and $u \notin V_j$, and (P3) no category-i cluster has diameter greater than $2^i \mu$.

The goal of the clustering phase is to compute a well-structured cluster partition $\mathcal{P} = (V_1, \ldots, V_q)$ along with *cluster trees* $\tilde{T}_1, \ldots, \tilde{T}_q$ and *cluster files* $\mathcal{F}_1, \ldots, \mathcal{F}_q$; the cluster trees capture the containment of the vertices in clusters $V_1, \ldots, V_q$ in the connected components of graphs $G_0, \ldots, G_r$; the cluster files are the concatenations of the adjacency lists of the vertices in the clusters.

Computing the cluster partition. We use a minimum spanning tree T of G to construct a well-structured cluster partition of G. For $0 \leq i \leq r$, let T_i be the subgraph of T that contains all vertices of T and all tree edges in categories $1, \ldots, i$. Then two vertices are in the same connected component of T_i if and only if they are in the same connected component of G_i. Hence, a well-structured cluster partition of T is also a well-structured cluster partition of G. We show how to compute the former. For any set $X \subseteq V$, we define its *tree diameter* as the total weight of the edges in the smallest subtree of T that contains all vertices in X. We guarantee in fact that every category-i cluster in the computed partition has tree diameter at most $2^i\mu$. Since the tree diameter of a cluster may be much larger than its diameter, we may generate more clusters than necessary; but their number is still $\mathcal{O}(V/\mu)$.

We iterate over graphs $T_0, \ldots, T_r$. In the i-th iteration, we partition the connected components of T_i into clusters. To bound the number of clusters we generate, we partition a component of T_i only if its tree diameter is at least $2^i\mu$ and it contains vertices that have not been added to any cluster in the first $i-1$ iterations. We call these vertices *active*; a component is active if it contains at least one active vertex; an active category-i component is *heavy* if its tree diameter is at least $2^i\mu$. To partition a heavy component C of T_i into clusters, we traverse an Euler tour of C, forming clusters as we go. When we visit an active category-$(i-1)$ component in C for the first time, we test whether adding this component to the current cluster would increase its tree diameter beyond $2^i\mu$. If so, we start a new cluster consisting of the active vertices in this component; otherwise, we add all active vertices in the component to the current cluster.

This computation takes $\mathcal{O}(\text{sort}(V + E) + (V/B)\log_2(W/w))$ I/Os w.h.p.: A minimum spanning tree T of G can be computed in $\mathcal{O}(\text{sort}(V + E))$ I/Os w.h.p. [4]. An Euler tour L of T can be computed in $\mathcal{O}(\text{sort}(V))$ I/Os [4]. The heavy components of a graph T_i can be identified and partitioned using two scans of L and using three stacks to keep track of the necessary information as we advance along L. Hence, one iteration of the clustering algorithm takes $\mathcal{O}(V/B)$ I/Os; all $r = \log_2 W$ iterations take $\mathcal{O}((V/B)\log_2 W)$ I/Os.

It remains to be argued that the computed partition is well-structured. Clearly, every category-i cluster has diameter at most $2^i\mu$. Such a cluster is completely contained in a category-i component, and no category-$(i-1)$ component has vertices in two different category-i clusters. Hence, all clusters have Property (P2). In the full paper, we show that their number is $\mathcal{O}(V/\mu)$.

Lemma 3. *A well-structured cluster partition of a weighted graph $G = (V, E)$ can be computed in $\mathcal{O}(\text{sort}(V + E) + (V/B)\log_2(W/w))$ I/Os w.h.p.*

Computing the cluster trees. In order to decide in which hot pool to store an edge (v, w) during the shortest path phase, we must be able to find the smallest i so that the category-i component containing v includes a settled vertex. Next we define *cluster trees* as the tool to determine category i efficiently.

The nesting of the components of graphs $T_0, \ldots, T_r$ can be captured in a tree $\tilde{T}$. The nodes of $\tilde{T}$ represent the connected components of $T_0, \ldots, T_r$. A node representing a category-i component C is the child of a node representing a category-$(i+1)$ component C' if $C \subseteq C'$. We ensure that every internal node of $\tilde{T}$ has at least two children; that is, a subgraph C of T that is a component of more than one graph T_i is represented only once in $\tilde{T}$. We define the category of such a component as the largest integer i so that C is a component of T_i.

Now we define the cluster tree $\tilde{T}_j$ for a category-i cluster V_j: Let C be the category-i component containing V_j, and let v be the node in $\tilde{T}$ that represents C; $\tilde{T}_j$ consists of the paths in $\tilde{T}$ from v to all the leaves that represent vertices in V_j.

Tree $\tilde{T}$ can be computed in r scans of Euler tour L, similar to the construction of the clusters; this takes $\mathcal{O}((V/B)\log_2 W)$ I/Os. Trees $\tilde{T}_1, \ldots, T_q$ can be computed in $\mathcal{O}(\mathrm{sort}(V))$ I/Os, using a DFS-traversal of $\tilde{T}$. In the full paper, we show that their total size is $\mathcal{O}(V)$.

Computing the cluster files. The last missing piece of information about clusters $V_1, \ldots, V_q$ is their *cluster files* $\mathcal{F}_1, \ldots, \mathcal{F}_q$. File $\mathcal{F}_j$ is the concatenation of the adjacency lists of the vertices in V_j. Clearly, files $\mathcal{F}_1, \ldots, \mathcal{F}_q$ can be computed in $\mathcal{O}(\mathrm{sort}(V+E))$ I/Os, by sorting the edge set of G appropriately.

4.2 The Shortest Path Phase

At a very high level, the shortest path phase is similar to Dijkstra's algorithm. We use the integer priority queue from Section 3 to store all non-settled vertices; their priorities equal their tentative distances from s. We proceed in iterations: Each iteration starts with a BatchedDeleteMin operation, which retrieves the vertices to be settled in this iteration. The priorities of the retrieved vertices are recorded as their final distances from s, which is correct because all edges in G have weight at least 1 and the priorities of the retrieved vertices differ by at most 1. Finally, we relax the edges incident to the retrieved vertices.

We use the clusters built in the previous phase to avoid spending one I/O per vertex on retrieving adjacency lists. When the first vertex in a cluster V_j is settled, we load the whole cluster file $\mathcal{F}_j$ into a set of hot pools $\mathcal{H}_1, \ldots, \mathcal{H}_r$. When we subsequently settle a vertex $v \in V_k$, we scan the hot pools to see whether they contain v's adjacency list. If so, we relax the edges incident to v; otherwise, we have to load file $\mathcal{F}_k$ first. Since we load every cluster file only once, we spend only $\mathcal{O}(V/\mu + E/B)$ I/Os on retrieving adjacency lists. Since we scan the hot pools in each iteration, to decide which cluster files need to be loaded, the challenge is to avoid touching an edge too often during these scans. We solve this problem by using a hierarchy of hot pools $\mathcal{H}_1, \ldots, \mathcal{H}_r$ and inspecting only a subset $\mathcal{H}_1, \ldots, \mathcal{H}_i$ of these pools in each iteration. We choose the pool where to store every edge to be the highest pool that is scanned at least once before this edge has to be relaxed. The precise choice of this pool is based on the following two observations: (1) It suffices to relax a category-i

edge incident to a settled vertex v any time before the first vertex at distance at least $\mathrm{dist}(s,v) + 2^{i-1}$ from s is settled. (2) An edge in a category-i component cannot be up for relaxation before the first vertex in the component is about to be settled. The first observation allows us to store all category-i edges in pool $\mathcal{H}_i$, as long as we guarantee that $\mathcal{H}_i$ is scanned at least once between the settling of two vertices whose distances from s differ by at least 2^{i-1}. The second observation allows us to store even category-j edges, $j < i$, in pool $\mathcal{H}_i$, as long as we move these edges to lower pools as the time of their relaxation approaches.

The second observation is harder to exploit than the first one because it requires some mechanism to identify for every vertex v, the smallest category i so that the category-i component containing v contains a settled vertex or a vertex to be settled soon. We provide such a mechanism using four additional pools $\mathcal{V}_i$, $\mathcal{T}_i$, $\mathcal{H}'_i$, and $\mathcal{T}'_i$ per category. Pool $\mathcal{V}_i$ contains settled vertices whose catergory-j edges, $j < i$, have been relaxed. Pools $\mathcal{T}_1, \dots, \mathcal{T}_r$ store the nodes of the cluster trees corresponding to the cluster files loaded into pools $\mathcal{H}_1, \dots, \mathcal{H}_r$. A cluster tree node is stored in pool $\mathcal{T}_i$ if its corresponding component C is in category i or it is in category $j < i$ and the smallest component containing C and at least one settled vertex or vertex in $\mathcal{B}_1, \dots, \mathcal{B}_i$ is in category i. We store an edge (v, w) in pool $\mathcal{H}_i$ if its category is i or it is less than i and the cluster tree node corresponding to vertex v resides in pool $\mathcal{T}_i$. Pools $\mathcal{H}'_1, \dots, \mathcal{H}'_r$ and $\mathcal{T}'_1, \dots, \mathcal{T}'_r$ are auxiliary pools that are used as temporary storage after loading new cluster files and trees and before we determine the correct pools where to store their contents. We maintain a labeling of the cluster tree nodes in pools $\mathcal{T}'_1, \dots, \mathcal{T}'_r$ that helps us to identify the pool $\mathcal{T}_i$ where each node in these pools is to be stored: A node is either marked or unmarked; a marked node in $\mathcal{T}'_i$ corresponds to a component that contains a settled vertex or a vertex in $\mathcal{B}_1, \dots, \mathcal{B}_i$. In addition, every node stores the category of (the component corresponding to) its lowest marked ancestor in the cluster tree.

To determine the proper subset of pools to be inspected in each iteration, we tie the updates of the hot pools and the relaxation of edges to the updates of priority queue buckets performed by the BATCHEDDELETEMIN operation. Every such operation can be divided into two phases: The *up-phase* incorporates the updates in buckets $\mathcal{U}_1, \dots, \mathcal{U}_i$ into buckets $\mathcal{B}_1, \dots, \mathcal{B}_i$; the *down-phase* distributes the contents of bucket $\mathcal{B}_i$ over buckets $\mathcal{B}_1, \dots, \mathcal{B}_{i-1}$. We augment the up-phase so that it loads cluster files and relaxes the edges in pools $\mathcal{H}_1, \dots, \mathcal{H}_i$ that are incident to settled vertices. In the down-phase, we shift edges from pool $\mathcal{H}_i$ to pools $\mathcal{H}_1, \dots, \mathcal{H}_{i-1}$ as necessary. The details of these two phases are as follows:

The up-phase. We update the contents of the category-j pools and relax the edges in $\mathcal{H}_j$ after applying the updates from $\mathcal{U}_j$ to $\mathcal{B}_j$. We mark every node in $\mathcal{T}'_j$ whose corresponding component contains a vertex in $\mathcal{B}_j \cup \mathcal{V}_j$ and identify, for every node, the category of its lowest marked ancestor in $\mathcal{T}'_j$. We move every node whose lowest marked ancestor is in a category greater than j to $\mathcal{T}'_{j+1}$ and insert the other nodes into $\mathcal{T}_j$. For every leaf of a cluster tree that was moved to $\mathcal{T}'_{j+1}$, we move the whole adjacency list of the corresponding vertex from $\mathcal{H}'_j$ to $\mathcal{H}'_{j+1}$. Any other edge in $\mathcal{H}'_j$ is moved to $\mathcal{H}'_{j+1}$ if its category is greater than j;

otherwise, we insert it into $\mathcal{H}_j$. We scan $\mathcal{V}_j$ and $\mathcal{H}_j$ to identify all category-j vertices in $\mathcal{V}_j$ that do not have incident edges in $\mathcal{H}_j$, load the corresponding cluster files and trees into $\mathcal{H}'_j$ and $\mathcal{T}'_j$, respectively, and sort $\mathcal{H}'_j$ and $\mathcal{T}'_j$. We proceed as above to decide where to move the nodes and edges in $\mathcal{T}'_j$ and $\mathcal{H}'_j$. We scan $\mathcal{V}_j$ and $\mathcal{H}_j$ again, this time to relax all edges in $\mathcal{H}_j$ incident to vertices in $\mathcal{V}_j$. As we argue below, the resulting UPDATE operations affect only $\mathcal{B}_{j+1}, \ldots, \mathcal{B}_r$; so we insert these updates into $\mathcal{U}_{j+1}$. Finally, we move all vertices in $\mathcal{V}_j$ to $\mathcal{V}_{j+1}$ and either proceed to $\mathcal{B}_{j+1}$ or enter the down-phase with $i = j$, depending on whether or not $\mathcal{B}_j$ is empty.

The down-phase. We move edges and cluster tree nodes from $\mathcal{H}_j$ and $\mathcal{T}_j$ to $\mathcal{H}_{j-1}$ and $\mathcal{T}_{j-1}$ while moving vertices from bucket $\mathcal{B}_j$ to bucket $\mathcal{B}_{j-1}$. First we identify all nodes in $\mathcal{T}_j$ whose corresponding components contain vertices that are pushed to $\mathcal{B}_{j-1}$. If the category of such a node v is less than j, we push the whole subtree rooted at v to $\mathcal{T}_{j-1}$. For every leaf that is pushed to $\mathcal{T}_{j-1}$, we push all its incident edges of category less than j from $\mathcal{H}_j$ to $\mathcal{H}_{j-1}$. Finally, we remove all nodes of $\tilde{T}$ from $\mathcal{T}_j$ that have no descendent leaves left in $\mathcal{T}_j$.

Correctness. We need to prove the following: (1) The relaxation of a category-i edge can only affect buckets $\mathcal{B}_{i+1}, \ldots, \mathcal{B}_r$. (2) Every category-$i$ edge (v, w) is relaxed before a vertex at distance at least $\text{dist}(s, v) + 2^{i-1}$ from s is settled.

To see that the first claim is true, observe that a vertex v that is settled between the last and the current relaxation of edges in $\mathcal{H}_i$ has distance at least $l - 2^{i-2}$ from s, where $[l, u)$ is the priority interval of bucket $\mathcal{B}_i$, i.e., $u \le l + 2^{i-2}$. Since an edge $(v, w) \in \mathcal{H}_i$ has weight at least 2^{i-1}, we have $\text{dist}(s, v) + \omega(v, w) > l + 2^{i-2} = u$; hence, vertex w will be inserted into one of buckets $\mathcal{B}_{i+1}, \ldots, \mathcal{B}_r$.

The second claim follows immediately if we can show that when vertex v reaches pool $\mathcal{V}_i$, edge (v, w) either is in $\mathcal{H}_i$ or is loaded into $\mathcal{H}_i$. This is sufficient because we have to empty at least one bucket $\mathcal{B}_j$, $j \ge i$, between the settling of vertex v and the settling of a vertex at distance at least $\text{dist}(s, v) + 2^{i-1}$. Since edge (v, w) is in category i, the category of vertex v is $h \le i$. When $v \in \mathcal{V}_h$, the cluster file containing v's adjacency list is loaded into pool $\mathcal{H}'_h$, and all category-h edges incident to v are moved to $\mathcal{H}_h$, unless pool $\mathcal{H}_h$ already contains a category-h edge incident to v. It is easy to verify that in the latter case, $\mathcal{H}_h$ must contain *all* category-h edges incident to v. This proves the claim for $i = h$. For $i > h$, we observe that the adjacency list of v is loaded at the latest when $v \in \mathcal{V}_h$. If this is the case, edge (v, w) is moved to pool $\mathcal{H}_i$ at the same time when vertex v reaches pool $\mathcal{V}_i$. If vertex v finds an incident category-h edge in $\mathcal{H}_h$, then edge (v, w) is either in one of pools $\mathcal{H}'_{h+1}, \ldots, \mathcal{H}'_i$ or in one of pools $\mathcal{H}_i, \ldots, \mathcal{H}_r$. In the former case, edge (v, w) is placed into pool $\mathcal{H}_i$ when vertex v reaches pool $\mathcal{V}_i$. In the latter case, edge (v, w) is in fact in pool $\mathcal{H}_i$ because, otherwise, pool $\mathcal{H}_h$ could not contain any edge incident to v. This proves the claim for $i > h$.

I/O-complexity. The analysis is based on the following two claims proved below: (1) Every cluster file is loaded exactly once. (2) Every edge is involved in $\mathcal{O}(\mu)$ updates of a pool $\mathcal{H}_i$ before it moves to a pool of lower category; the same is true for the cluster tree nodes in $\mathcal{T}_i$. In the full paper, we show that all the updates of the hot pools can be performed using a constant number of scans. Also note that every vertex is involved in the sorting of pool $\mathcal{V}_1$ only once, and every edge or cluster tree node is involved in the sorting of a pool $\mathcal{H}'_i$ or $\mathcal{T}'_j$ only once. These observations together establish that our algorithm spends $\mathcal{O}(V/\mu + (V + E)/B)$ I/Os on loading cluster files and cluster trees; $\mathcal{O}(\text{sort}(V + E))$ I/Os on sorting pools $\mathcal{V}_1, \mathcal{H}'_1, \dots, \mathcal{H}'_r$, and $\mathcal{T}'_1, \dots, \mathcal{T}'_r$; and $\mathcal{O}((\mu E/B) \log_2 W)$ I/Os on all remaining updates of priority queue buckets and hot pools. Hence, the total I/O-complexity of the shortest path phase is $\mathcal{O}(V/\mu + (\mu E/B) \log_2 W + \text{sort}(V + E))$.

To show that every cluster file is loaded exactly once, we have to prove that once a cluster file containing the adjacency list of a category-i vertex v has been loaded, vertex v finds an incident category-i edge (v, w) in $\mathcal{H}_i$. The only circumstance possibly preventing this is if $(v, w) \in \mathcal{H}_j$, $j > i$, at the time when $v \in \mathcal{V}_i$. However, at the time when edge (v, w) was moved to $\mathcal{H}_j$, no vertex in the category-$(j - 1)$ component C that contains v had been settled or was in one of $\mathcal{B}_1, \dots, \mathcal{B}_{j-1}$. Every vertex in C that is subsequently inserted into the priority queue is inserted into a bucket $\mathcal{B}_{h+1}$, $h \geq j$, because this happens as the result of the relaxation of a category-h edge. Hence, before any vertex in C can be settled, such a vertex has to be moved from $\mathcal{B}_j$ to $\mathcal{B}_{j-1}$, which causes edge (v, w) to move to $\mathcal{H}_{j-1}$. This proves that vertex v finds edge (v, w) in $\mathcal{H}_i$.

To prove that every edge is involved in at most $\mathcal{O}(\mu)$ scans of pool $\mathcal{H}_i$, observe that at the time when an edge (v, w) is moved to pool $\mathcal{H}_i$, there has to be a vertex x in the same category-i component C as v that either has been settled already or is contained in one of buckets $\mathcal{B}_1, \dots, \mathcal{B}_i$ and hence will be settled before pool $\mathcal{H}_i$ is scanned for the next time; moreover, there has to be such a vertex whose distance from v is at most $2^i \mu$. By Lemma 2, the algorithm makes progress at least $2^{j-2}/3$ every time pool $\mathcal{H}_i$ is scanned. Hence, after $\mathcal{O}(\mu)$ scans, vertex v is settled, so that edge (v, w) is relaxed before or during the next scan of pool $\mathcal{H}_i$. This proves the second claim.

Summing the I/O-complexities of the two phases, we obtain that the I/O-complexity of our algorithm is $\mathcal{O}(V/\mu + (\mu(V + E)/B) \log_2 W + \text{sort}(V + E))$ w.h.p. By choosing $\mu = \sqrt{VB/(E \log_2 W)}$, we obtain the following result.

Theorem 1. *The single source shortest path problem on an undirected graph* $G = (V, E)$ *can be solved in* $\mathcal{O}(\sqrt{(VE/B) \log_2(W/w)} + \text{sort}(V + E))$ *I/Os w.h.p., where* w *and* W *are the minimal and maximal edge weights in* G.

Observe that the only place in the algorithm where randomization is used is in the computation of the minimum spanning tree. In [3], it is shown that a minimum spanning tree can be computed in $\mathcal{O}(\text{sort}(V + E) \log \log(VB/E))$ I/Os deterministically. Hence, we can obtain a deterministic version of our algorithm that takes $\mathcal{O}(\sqrt{(VE/B) \log_2(W/w)} + \text{sort}(V + E) \log \log(VB/E))$ I/Os.

4.3 An Average-Case Analysis

Next we analyze the average-case complexity of our algorithm. We assume uniform random edge weights in $(0, 1]$, but make no randomness assumption about the structure of the graph. In the full paper, we show that we can deal with "short edges", that is, edges whose weight is at most $1/B$, in expected $\mathcal{O}(\mathrm{sort}(E))$ I/Os, because their expected number is E/B. We deal with long edges using the algorithm described in this section. Now we observe that the expected number of category-i edges in G and category-i nodes in $\tilde{T}$ is $\mathcal{O}(2^{i-1}E/B)$. Each such edge or node moves up through the hierarchy of pools $\mathcal{H}'_1, \ldots, \mathcal{H}'_r$ or $\mathcal{T}'_1, \ldots, \mathcal{T}'_r$, being touched $\mathcal{O}(1)$ times per category. Then it moves down through pools $\mathcal{H}_r, \mathcal{H}_{r-1}, \ldots, \mathcal{H}_i$ or $\mathcal{T}_r, \mathcal{T}_{r-1}, \ldots, \mathcal{T}_i$, being touched $\mathcal{O}(\mu)$ times per category. Hence, the total cost of scanning pools $\mathcal{H}'_1, \ldots, \mathcal{H}'_r$ and $\mathcal{T}'_1, \ldots, \mathcal{T}'_r$ is $\mathcal{O}(E \log_2 B/B)$, and the expected total cost of scanning pools $\mathcal{H}_1, \ldots, \mathcal{H}_r$ and $\mathcal{T}_1, \ldots, \mathcal{T}_r$ is $\mathcal{O}((\mu E/B^2) \sum_{i=1}^{r} 2^{i-1}(r-i+1)) = \mathcal{O}(\mu E/B)$. Thus, the expected I/O-complexity of our algorithm is $\mathcal{O}(V/\mu + \mu E/B + ((V+E)/B)\log_2 B + \mathrm{sort}(V+E))$. By choosing $\mu = \sqrt{VB/E}$, we obtain the following result.

Theorem 2. *The single source shortest path problem on an undirected graph $G = (V, E)$ whose edge weights are drawn uniformly at random from $(0, 1]$ can be solved in expected $\mathcal{O}(\sqrt{VE/B} + ((V+E)/B)\log_2 B + \mathrm{sort}(V+E))$ I/Os.*

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. of the ACM*, pp. 1116–1127, 1988.
2. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–233, 1990.
3. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. *Proc. 7th SWAT*, LNCS 1851, pp. 433–447. Springer, 2000.
4. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. *Proc. 6th ACM-SIAM SODA*, pp. 139–149, 1995.
5. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. *Proc. 23rd MFCS*, LNCS 1450, pp. 722–731. Springer, 1998.
6. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.
7. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.
8. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. *Proc. 8th IEEE SPDP*, pp. 169–176, 1996.
9. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. *Proc. 10th ESA*, LNCS 2461, pp. 723–73. Springer, 2002.
10. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, LNCS 2625. Springer, 2003.
11. R. Raman. Recent results on the single-source shortest paths problem. *ACM SIGACT News*, 28(2):81–87, June 1997.
12. M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
13. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
14. N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, School of Computer Science, Carleton University, 2002.