Cache-Oblivious Shortest Paths in Graphs Using Buffer Heap

Rezaul Alam Chowdhury*
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
shaikat@cs.utexas.edu

Vijaya Ramachandran†
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
vlr@cs.utexas.edu

ABSTRACT

We present the *Buffer Heap (BH)*, a cache-oblivious priority queue that supports *Delete-Min*, *Delete*, and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized block transfers from external memory, where B is the (unknown) block-size and N is the maximum number of elements in the queue. As is common in cache-oblivious algorithms, we assume a 'tall cache' (i.e., $M = \Omega(B^{1+\epsilon})$, where M is the size of the main memory). We also assume the *Decrease-Key* operation only verifies that the element does not exist in the priority queue with a smaller key value, hence it also supports the insert operation in the same amortized bound. The amortized time bound for each operation is $\mathcal{O}(\log N)$. We also present a *Cache-Oblivious Tournament Tree (COTT)*, which is simpler than the Buffer Heap, but has weaker bounds.

Using the Buffer Heap we present cache-oblivious algorithms for undirected and directed single-source shortest path (SSSP) problems for graphs with non-negative edge-weights. On a graph with V vertices and E edges, our algorithm for the undirected case performs $\mathcal{O}(V + \frac{E}{B}\log_2\frac{V}{B})$ block transfers and for the directed case performs $\mathcal{O}((V + \frac{E}{B})\cdot\log_2\frac{V}{B})$ block transfers. The running time of both algorithms is $\mathcal{O}((V+E)\cdot\log V)$.

For both priority queues with *Decrease-Key* operation, and for shortest path problems on general graphs, our results appear to give the first non-trivial cache-oblivious bounds.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Discrete Mathematics—*Graph Theory*; E.1 [**Data Structures**]: Lists, stacks, and queues; B.3.2 [**Hardware**]: Memory Structures—*Design Styles*

General Terms

Algorithms, Theory, Performance

Keywords

Cache-oblivious model, cache-aware model, priority queue, decrease-key, buffer heap, shortest paths, tournament tree

*Supported in part by an MCD Graduate Fellowship and NSF Grant CCR-9988160.
†Supported in part by NSF Grant CCR-9988160.

SPAA'04, June 27–30, 2004, Barcelona, Spain.

1. INTRODUCTION

1.1 The SSSP Problem

The *single-source shortest path* (SSSP) problem is one of the most fundamental and important combinatorial optimization problems from both a theoretical and a practical point of view. Given a (directed or undirected) graph G with vertex set $V[G]$, edge set $E[G]$, a non-negative real-valued weight function w over $E[G]$, and a distinguished vertex $s \in V[G]$, the SSSP problem seeks to find a path of minimum total edge-weight from s to every reachable vertex $v \in V[G]$. By V and E we denote the size of $V[G]$ and $E[G]$, respectively.

As long as the whole problem fits in *internal memory*, the SSSP problem can be solved efficiently by Dijkstra's algorithm [7] implemented using a *Fibonacci heap* [8] in $\mathcal{O}(E + V\log V)$ time for directed graphs. For undirected graphs the problem can also be solved in $\mathcal{O}(E\alpha(E,V) + V\min(\log V, \log\log\rho))$ time [13], where ρ is the ratio of the maximum and the minimum edge-weights in G, and $\alpha(E,V)$ is a certain natural inverse of Ackermann's function. Faster algorithms exist for special classes of graphs and graphs with restricted edge-weights. All of these algorithms, however, perform poorly on large data sets when data needs to be swapped between the faster internal memory and the slower *external memory*. Since most real world applications work with huge data sets, the large number of I/O operations performed by these algorithms becomes a bottleneck which necessitates the design of I/O efficient SSSP algorithms.

1.2 Cache-Aware Algorithms

Memory in modern computers is typically organized in a hierarchy with registers in the lowest level followed by level 1 cache, level 2 cache, level 3 cache, main memory, and disk. Access time of a memory level increases with its level, i.e., registers have the smallest access time while disks have the largest. To capture the influence of the memory access pattern of an algorithm on its running time Aggarwal and Vitter

[1] introduced the *two-level I/O model* (or *external memory model*). This model consists of a memory hierarchy with an internal memory of size M, and an arbitrarily large external memory partitioned into blocks of size B. The *I/O complexity* of an algorithm in this model is measured in terms of the number of blocks transferred between these two levels. This is a simple model that successfully models the situation where I/O operations between two levels of the memory hierarchy dominate the running time of the algorithm. Two basic I/O bounds are known for this model: the number of I/Os needed to read N contiguous data items from the disk is $scan(N) = \Theta(\frac{N}{B})$ and the number of I/Os required to sort N data items is $sort(N) = \Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ [1]. For most realistic values of M, B and N, $scan(N) < sort(N) \ll N$.

In recent years there has been considerable research on developing efficient external memory graph algorithms (see [15, 10] for recent surveys). Several I/O-efficient versions of internal memory SSSP algorithms have been developed [6, 11, 10, 12]. Virtually all internal memory SSSP algorithms work by maintaining an upper bound on the shortest distance (a *tentative distance*) to every vertex from s and visiting the vertices one by one (or in groups) in non-decreasing order of tentative distances. The next vertex (or group of vertices) to be visited is the one with the smallest tentative distance extracted from the set of unvisited vertices kept in a priority queue Q. After a vertex (or a group of vertices) has been extracted from Q each of its unvisited neighbors is either inserted into Q with a finite tentative distance or has its tentative distance updated if it already resides in Q. As pointed out in [10], the key problems in making these algorithms I/O efficient are:

(a) unstructured indexed accesses to adjacency lists.

(b) remembering visited vertices.

(c) the lack of external memory priority queues supporting *Decrease-Key* operations.

Virtually all external memory SSSP algorithms require $\Theta(V + \frac{E}{B})$ I/Os to access the adjacency lists (problem (a)). Recently, it has been shown in [12] that for undirected graphs adjacency lists can be retrieved in $\mathcal{O}(\sqrt{\frac{VE}{B}\log_2\rho} + \frac{E}{B})$ I/Os by an appropriate clustering of vertices and by loading an entire cluster in a *hot pool* at appropriate times. However, there are some I/O overheads in the preprocessing step that may dominate depending on the values of the parameters.

Problem (b) has been addressed in [6] by keeping the visited vertices in a dictionary in internal memory and periodically (when the internal memory becomes full) scanning the adjacency lists in external memory to remove the edges leading to these visited vertices and then emptying the dictionary. If this approach (the *phase approach*) is used problem (c) can also be avoided. Problem (b) has also been solved by using a *buffered repository tree* [5] in order to store the visited neighbors of each vertex.

Problem (c) has been addressed in [11] by using the I/O-efficient *tournament tree* that supports a sequence of k *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\frac{k}{B}\log_2\frac{V}{B})$ I/Os.

Major known results for the external memory SSSP problem are summarized in Table 2 under the column named "Cache-Aware Results". Kumar & Schwabe [11] were the first to develop an I/O-efficient version of Dijkstra's SSSP algorithm for undirected graphs. They use a tournament tree as a priority queue and perform some extra book-keeping using an auxiliary priority queue in order to handle visited vertices. Their algorithm requires $\mathcal{O}(V + \frac{E}{B}\log_2\frac{V}{B})$ block transfers. Using the *phase approach* [6] undirected SSSP can be solved in $\mathcal{O}(V + \frac{VE}{BM} + sort(E))$ I/Os. Recently, Meyer & Zeh developed another undirected SSSP algorithm that works on graphs with positive edge-weights and requires $\mathcal{O}(\sqrt{\frac{VE}{B}\log_2\rho} + sort(V+E)\log_2\log_2\frac{VB}{E})$ memory accesses [12]. For directed graphs the survey paper [15] claims a bound of $\mathcal{O}((V+\frac{E}{B})\cdot\log_2\frac{V}{B})$ I/Os for SSSP. Using the phase approach directed SSSP can be solved in $\mathcal{O}(V + \frac{VE}{BM}\log_2\frac{V}{B})$ memory transfers [6, 10].

1.3 The Cache-Oblivious Model

The main disadvantage of the two-level model is that algorithms often crucially depend on the knowledge of the parameters of two particular levels of the memory hierarchy and thus do not adapt well when the parameters change. In order to remove this inflexibility Frigo et al. introduced the *cache-oblivious model* [9]. As before, this model consists of a two-level memory hierarchy, but algorithms are designed and analyzed without using the parameters M and B in the algorithm description. It is assumed that whenever a data item is requested that is not stored in the internal memory, the external memory block containing that item is automatically transferred to the internal memory. When the internal memory is full and a block needs to be fetched from the external memory, the block in internal memory chosen for eviction is the one that will be accessed farthest in the future, i.e., an *optimal offline cache replacement strategy* is assumed. Since the analysis of an algorithm in this model holds for any memory and block size, it holds for all levels of a multi-level memory hierarchy [9]. Thus by reasoning about a simple two-level memory model we can, in fact, prove results about a multi-level memory model. Another advantage of this model is that the resulting algorithms are more flexible and portable. However, often these algorithms require a *tall cache* (i.e., $M = \Omega(B^{1+\epsilon})$, for some $\epsilon > 0$) for I/O-efficiency.

The cache-oblivious priority queue introduced by Arge et al. [2] and the *Funnel Heap* introduced by Brodal & Fagerberg [3] support *Insert-Key* and *Delete-Min* in optimal $\mathcal{O}(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os, where N is the number of elements in the queue, but they do not support *Decrease-Keys*. It appears that prior to our work, no nontrivial cache-oblivious results were known for priority queue with *Decrease-Keys* or for SSSP on graphs.

1.4 Our Results

Our results are tabulated in Tables 1 and 2. Our main contribution is the cache-oblivious Buffer Heap (BH) in Table 1, and its application to the cache-oblivious results for undirected and directed SSSP listed in Table 2. The BH supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os each under the tall cache assumption; without the tall cache assumption the bound is $\mathcal{O}(\frac{1}{B}\log_2 N)$ amortized. We assume that each element in the heap has an associated identifier, and these identifiers are drawn from a totally ordered set. We use the Buffer Heap to obtain cache-oblivious SSSP algorithms for undirected and directed graphs requiring $\mathcal{O}(V + \frac{E}{B}\log_2\frac{V}{B})$ and $\mathcal{O}((V + \frac{E}{B})\cdot\log_2\frac{V}{B})$ I/Os, respectively. We also present a cache-aware version of the Buffer Heap that supports each operation in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$ amortized I/Os without assum-

I/O Model	Priority Queue	*Decrease-Key*	*Delete*	*Delete-Min*
Cache-Aware	*Tournament Tree* [11]	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$
	Buffer Heap (cache-aware version) **(this paper)**	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$
Cache-Oblivious	*Buffer Heap* ($M=\Omega(B^{1+\epsilon})$) **(this paper)**	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$	$\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$

Table 1: Amortized I/O bounds for priority queues with *Decrease-Keys*. (N = number of elements)

Graph Type	Cache-Aware Results	Cache-Oblivious Results ($M=\Omega(B^{1+\epsilon})$)
Undirected	$\mathcal{O}(V+\frac{E}{B}\log_2\frac{V}{B})$ [11]	$\mathcal{O}(V+\frac{E}{B}\log_2\frac{V}{B})$ **(this paper)**
	$\mathcal{O}(V+\frac{VE}{BM}+sort(E))$ [6, 10]	
	$\mathcal{O}(\sqrt{\frac{VE}{B}\log_2\rho}+sort(V+E)\log_2\log_2\frac{VB}{E})$ [12]	
	$\mathcal{O}(V+\frac{E}{B}\log_2\frac{V}{M})$ **(this paper)**	
Directed	$\mathcal{O}((V+\frac{E}{B})\cdot\log_2\frac{V}{B})$ [15]	$\mathcal{O}((V+\frac{E}{B})\cdot\log_2\frac{V}{B})$ **(this paper)**
	$\mathcal{O}(V+\frac{VE}{BM}\log_2\frac{V}{B})$ [6, 10]	

Table 2: I/O bounds for the SSSP problem on weighted graphs. ($V=|V[G]|$, $E=|E[G]|$)

ing a tall cache, and use it to obtain a cache-aware undirected SSSP algorithm requiring $\mathcal{O}(V+\frac{E}{B}\log_2\frac{V}{M})$ I/Os. Soon after our conference submission, Brodal et al. [4] proposed another cache-oblivious priority queue (which they call the *Bucket Heap*) supporting *Decrease-Key* operations. The Bucket Heap differs slightly from our Buffer Heap. It supports the same operations in similar I/O bounds, but does not assume a tall cache. The tall cache assumption can be removed from our Buffer Heap as well by a small modification using a strategy from [4] (we describe this near the end of section 2.1.5). The Bucket Heap is used in [4] in exactly the same way as we do to obtain a cache-oblivious undirected SSSP algorithm.

We also present a cache-oblivious variant of the tournament tree [11], which we call COTT. Although our bounds for COTT are weaker than those for Buffer Heap, COTT is a simpler data structure, and may be more amenable to practical implementation. Our directed SSSP algorithm runs just as efficiently with COTT as with BH.

Our I/O bounds for shortest paths are not very impressive for sparse graphs, but they do provide dramatic improvements for moderately dense graphs. For example, for undirected graphs, if $E \geq \frac{VB}{\log_2 V/B}$ our algorithm reduces the number of I/Os by a factor of $\frac{B}{\log_2 V/B}$ over the naïve method. For directed graphs, we obtain the same improvement if $E \geq VB$.

2. CACHE-OBLIVIOUS PRIORITY QUEUES

In this section we describe two cache-oblivious priority queues: the *Buffer Heap* (BH) and the *Cache-Oblivious Tournament Tree* (COTT).

BH builds on the integer priority queue described by Meyer and Zeh in [12]. It supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized memory transfers each assuming a tall cache, where N is the maximum number of elements in the queue. A *Delete*(x) operation deletes the element x from the queue if it exists and a *Delete-Min*() operation retrieves and deletes the element with the minimum key from the queue. A *Decrease-Key* (x,k_x) operation inserts the element x with key k_x into the queue if x does not already exist in the queue, otherwise it replaces the key k'_x of x in the queue with k_x provided $k_x < k'_x$. Therefore, a new element x with key k_x can be inserted into the queue by performing a *Decrease-Key*(x,k_x) operation. In section 3, we will use this priority queue to design cache-oblivious shortest path algorithms for directed and undirected graphs.

COTT is a cache-oblivious version of the cache-aware tournament tree introduced by Kumar and Schwabe in [11]. This structure can contain only a pre-determined set of elements which are initially inserted into fixed positions in the structure with $+\infty$ key value. While the cache-aware tournament tree of [11] with N elements supports a sequence of k *Delete*, *Delete-Min* and *Decrease-Key* operations in at most $\mathcal{O}(\frac{k}{B}\log_2\frac{N}{B})$ I/Os, our cache-oblivious version supports *Delete* and *Delete-Min* in $\mathcal{O}(\log_2\frac{N}{B})$, and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os, respectively, under the tall cache assumption. Although COTT is not as I/O-efficient as BH, it is a simpler structure and gives the same I/O bound for directed SSSP (section 3.2) as does BH.

2.1 The Buffer Heap

In this section we describe the Buffer Heap (BH). We assume, for convenience, that all keys are distinct (although BH can be modified to run with the same bounds when keys are not distinct).

2.1.1 Structure

The Buffer Heap consists of $r = 1 + \lceil \log_2 N \rceil$ levels. For $0 \leq i \leq r-1$, level i consists of two buffers: the *element buffer* B_i and the *update buffer* U_i. The buffers are defined by splitter elements $s_0, s_1, \ldots, s_r$. Each element in an element buffer is of the form (x,k_x), where x is the element id and k_x is its key. Each update buffer U_i stores updates to be applied in a batched manner on the elements in element buffers $B_i, B_{i+1}, \ldots, B_{r-1}$. When a *Delete* or *Decrease-Key* operation arrives, it is inserted into U_0 augmented with the current time stamp. These operations move lazily level by level from U_0 to U_{r-1} and take necessary actions (if any) on their target elements in the element buffer in each level.

At any time, the following invariants are maintained:

INVARIANT 1. *(a)* $-\infty = s_0 \leq s_1 \leq \ldots \leq s_r = +\infty$ *(b) Each* B_i *contains only elements with keys in the range* $(s_i, s_{i+1}]$. *(c) Each* U_i *contains only updates with associated key value (if any) in the range* $(s_i, s_r]$.

INVARIANT 2. *For each element* x *in any* B_i, *all updates applicable to* x *that are not yet applied, reside in* $U_0, U_1, \ldots, U_i$.

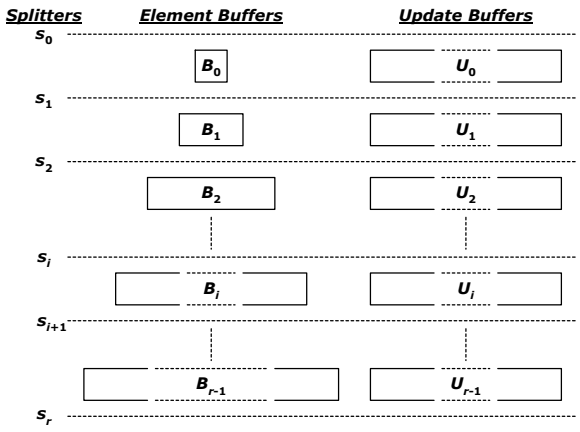

Figure 1: The structure of the Buffer Heap: Each B_i stores elements with key values in $(s_i, s_{i+1}]$, and each U_i stores updates with key values (if any) in $(s_i, s_r]$. We also have $|B_i| \leq 2^i$ for each B_i, but there is no such restriction on $|U_i|$.

INVARIANT 3. *Each B_i contains at most 2^i elements.*

INVARIANT 4. *(a) Elements in each B_i are kept sorted in ascending order by element id. (b) Updates in each U_i except U_0 are divided into (at most three) segments with updates in each segment sorted in ascending order by element id and time stamp.*

Initially all buffers are empty and $s_0 = -\infty, s_1 = s_2 = \ldots = s_r = +\infty$.

2.1.2 Layout

The element buffers are stored in a stack S_B with elements of lower-numbered levels placed above elements of higher-numbered levels. Elements of the same level occupy contiguous space in the stack with elements having lower element ids placed above elements having higher element ids. Similarly, update buffers are placed in another stack S_U where buffers of lower-numbered levels are placed above the ones of higher-numbered levels. Updates in a single buffer occupy a contiguous region in the stack. For $1 \leq i \leq r-1$, the segments of U_i are stored one above another in the stack, and updates in each segment are stored sorted from top to bottom first by element id and then by time stamp. However, updates in U_0 are placed as they arrive and hence are in sorted order descending by time stamp from top to bottom. We maintain an array A_s of size $1+r$ to maintain information on the buffers and splitter elements. For $0 \leq i \leq r-1$, $A_s[i]$ contains the value of s_i, the number of elements in B_i, and the number of segments in U_i along with the number of updates in each segment; $A_s[r]$ contains only the value of s_r. We also use a constant number of arrays $(B', B'', B''', U'$ and $U'')$ of size $\mathcal{O}(N)$ each for temporary usage. These arrays will be accessed or modified sequentially. The Buffer Heap uses $\mathcal{O}(N)$ space.

2.1.3 Operations

A *Delete* or *Decrease-Key* operation inserts itself into U_0 (by pushing itself into S_U) augmented with the current time stamp. Further processing is deferred to the next *Delete-Min* operation. The *Delete-Min* operation uses the following four functions: **Delete-Min**, **Apply-Updates**, **Split** and **Redistribute-Elements** (descriptions follow).

We also reconstruct the data structure any time the number of operations in update buffers exceeds the number of elements in element buffers, using the following *Reconstruct* operation.

Function Reconstruct()

Step 1: Sort the elements in S_B by element id and level.

Step 2: Sort the operations in S_U by element id and time stamp.

Step 3: Scan S_B and S_U simultaneously, and apply the updates in S_U on the elements of S_B.

Step 4: Reconstruct the data structure by filling its shallowest levels with the elements of S_B, and emptying S_U.

Function Delete-Min()

Function: Deletes and returns the element X with the smallest key k_X in BH.

Steps:

Step 1: Set B' to empty and $k \leftarrow -1$.

Step 2: While $k < r-1$ and $|B'| = 0$ do:

Step 2(a): Set $k \leftarrow k+1$.
Step 2(b): **Apply-Updates**(k, B')

Step 3: If $|B'| = 0$ then return "Queue Empty" otherwise perform the following steps:

Step 3(a): $(X, k_X) \leftarrow$ **Redistribute-Elements**(k, B')
Step 3(b): Return (X, k_X).

Function Apply-Updates(k, B')

Function: Returns all elements (if any) in the priority queue with key value at most s_{k+1} (more precisely in the range $(s_k, s_{k+1}]$) sorted in ascending order by element id. The elements are returned in the array B'.

Pre-Conditions:

(1) $0 \leq k \leq r-1$.
(2) Invariants 1, 2, 3 and 4 are satisfied.
(3) For $0 \leq i < k$, all B_i and U_i are empty.

Post-Conditions:

(1) Invariants 1, 2 and 4 are satisfied, but 3 may be violated for B_k, i.e., B_k may contain more than 2^k elements.
(2) For $0 \leq i < k$, all B_i and for $0 \leq i \leq k$, all U_i are empty.

Steps:

Step 1: If $k = 0$, pop the contents of U_k from S_U and insert them sequentially into the initially empty array U', otherwise use the temporary arrays U' and U'' to merge the segments of U_k from S_U, and store the merged updates sorted by element id and time-stamp in U'.

Step 2: If $k = 0$, sort the updates in U' cache-obliviously in ascending order by element id and time stamp.

Step 3: Scan U' and B_k (by popping the elements of B_k from S_B) simultaneously to generate updated elements into an initially empty array B'. Elements in B' are generated in ascending order by element id. Use the following rules to generate B':

Rule 3(a): If an element (x, k_x) in B_k has no matching update in U' copy (x, k_x) to B'.

Rule 3(b): If (x, k_x) in B_k has exactly one matching update in U' and that update is:

Delete(x)**:** do not copy (x, k_x) to B'.

Decrease-Key(x, k'_x) **/** ***Sink***(x, k'_x)**:** copy $(x, \min(k_x, k'_x))$ to B'.

Rule 3(c): If (x, k_x) in B_k has multiple matching updates in U', then decide by time stamp and rule 3(b), which action is to be taken.

Rule 3(d): If an update in U' has no matching element in B_k, and the update is:

Decrease-Key(x, k'_x) **/** ***Sink***(x, k'_x)**:** copy (x, k'_x) to B' if $s_k < k'_x \leq s_{k+1}$.

Update $A_s[k]$ for the new sizes of level k buffers.

Step 4: If $k < r-1$, sequentially scan U' and push appropriate updates into S_U forming a new segment of U_{k+1}. We use the following rules to find the updates to push into S_U:

Rule 4(a): Push each $Sink(x, k_x)$ and $Decrease\text{-}Key(x, k_x)$ in U' with $k_x > s_{k+1}$ into S_U.

Rule 4(b): For every $Delete(x)$ and $Decrease\text{-}Key(x, k_x)$ operation with $k_x \leq s_{k+1}$ in U', push a $Delete(x)$ operation into S_U.

Update $A_s[k+1]$ to add information on the new segment added to U_{k+1}.

Step 5: Return B'.

Function Split(B', B'', B''', t)

Function: B' contains elements of the form (element, key) sorted by element id, and $|B'| \geq t$. This function squeezes t elements of B' having the smallest t keys to the beginning of B' and copies the remaining elements to B'' retaining their order by element id. It also returns the t-th largest key value in B'. Uses B''' as a temporary storage area.

Steps:

Step 1: Empty both B'' and B'''.

Step 2: Copy the contents of B' to B'''.

Step 3: Find the element with the t-th largest key p in B''' using a cache-oblivious selection algorithm.

Step 4: Scan the contents of B' and squeeze the elements with key value at most p to the beginning of B' and move the remaining elements to B'' retaining their order in B'.

Step 5: Return p.

Function Redistribute-Elements(k, B')

Function: Removes and returns the element with the minimum key value from B_k as (X, k_X) and redistributes the remaining elements of B_k to $B_0, B_2, \ldots, B_{k-1}$, and to U_{k+1} if $k < r-1$. The elements of B_k are assumed to be in the array B'.

Pre-Conditions:

(1) $0 \leq k \leq r-1$.

(2) Invariants 1, 2 and 4 are satisfied, but 3 may be violated for B_k.

(3) For $0 \leq i < k$, all B_i and for $0 \leq i \leq k$, all U_i are empty.

(4) B_k is non-empty.

Post-Conditions:

(1) Invariants 1, 2, 3 and 4 are satisfied.

(2) The element with the minimum key in the original B_k is removed from the priority queue.

(3) Splitters $s_1, s_2, \ldots, s_{k+1}$ are redefined. If s_{k+1} had a finite value at the start of the function, then $s_{k+2}, s_{k+3}, \ldots, s_l$ are set to the new value of s_{k+1}, where $l > k$ is the smallest level such that $(s_l, s_{l+1}]$ was non-empty at the start of the function.

(4) For some $j \leq k$, each element buffer from B_0 to B_{j-1} contains some elements, but B_j is empty. For $0 \leq i \leq j-1$, each B_i must be full (contains 2^i elements) except possibly B_{j-1} which may be non-full. For $j \leq i \leq k$ each B_i is empty and assigned an empty range ($s_i = s_{i+1}$).

(5) For $0 \leq i \leq k$, all U_i are empty.

Steps:

Step 1: If $|B'| > 2^k$ then perform the followings:

Step 1(a): Set $p \leftarrow$ **Split**$(B', B'', B''', 2^k)$

Step 1(b): Consider each element of B'' as a *Sink* operation with the current time stamp. Push them sorted by element id and time-stamp into S_U as a new segment of U_{k+1}.

Step 1(c): Update $A_s[k+1]$ to include information on the new segment of U_{k+1} and $A_s[k]$ to set the number of elements in B_k to 0.

Step 1(d): Set $t \leftarrow s_{k+1}$. Set $s_{k+1} \leftarrow p$. Update $A_s[k+1]$.

Step 1(e): If $t \neq +\infty$ then for all s_{i+1} with $i > k$ and $s_{i+1} = t$, set $s_{i+1} \leftarrow p$ and update $A_s[i+1]$.

Step 2: Set $s_k \leftarrow s_{k+1}$. Update $A_s[k]$.

Step 3: For $i \leftarrow k-1$ downto 0 do:

Step 3(a): If $|B'| \leq 2^i$, set $s_i \leftarrow s_{i+1}$, otherwise perform the following steps:

Step 3(a_1): Set $p \leftarrow$ **Split**$(B', B'', B''', 2^i)$

Step 3(a_2): Push the contents of B'' in appropriate order to S_B.

Step 3(a_3): If $i \neq 0$ then set $s_i \leftarrow p$

Step 3(b): Update $A_s[i]$ to reflect the number of elements in B_i (this number is 0 if $s_i = s_{i+1}$) and the new value of s_i.

Step 4: Now B' contains only one element, and it is assigned to (X, k_X) and returned.

2.1.4 Correctness

A *Delete* or *Decrease-Key* operation only inserts itself into U_0 and this insertion does not violate any invariant. Subsequent *Delete-Min* and *Reconstruct* operations are responsible for actual application of these *Delete*/*Decrease-Key* operations on BH elements. The correctness of the *Reconstruct* operation is straight-forward. Therefore, we only need to prove the following lemma:

LEMMA 1. *A* Delete-Min *operation finds and deletes the element with the smallest key value from BH, while correctly applying all relevant* Decrease-Key *and* Delete *operations, and maintaining all invariants.*

PROOF. Observe that if invariants 1 and 2 hold, the smallest level k such that B_k is non-empty after applying all updates in $U_0, U_1, \ldots, U_k$ on B_k will certainly contain the element with the smallest key in the entire BH. The **Delete-Min** function utilizes this idea. However, the way *Delete-Min* is implemented (more specifically **Apply-Updates**) it needs invariant 4 to hold, too, in order to correctly apply the updates in a level in a constant number of passes of both the update and the element buffers. We will prove that all invariants are always maintained. However, note that invariant 3 is needed only for efficiency, not for correctness.

Starting with level $k = 0$, **Delete-Min** function finds the smallest level that has a non-empty element buffer after the application of updates. It calls **Apply-Updates** to apply the updates in U_k (assuming that $U_0, U_1, \ldots, U_{k-1}$ are all emptied and collapsed to U_k by previous calls of **Apply-Updates**) on B_k. A *Delete* operation in U_k is applied by deleting any matching element from B_k, and a *Decrease-Key*(x, k_x) operation applicable to B_k (i.e., with $s_k < k_x \leq s_{k+1}$) is applied by inserting (x, k_x) into B_k if x does not already exist in B_k, otherwise updating the key value of x in B_k if necessary. **Apply-Updates** empties U_k after applying the updates on B_k and copying appropriate updates from U_k to U_{k+1} (if $k < r - 1$). It copies all *Delete* operations from U_k to U_{k+1} along with every *Decrease-Key* operation that is not applicable to B_k. However, for every *Decrease-Key*(x, k_x) operation applied on B_k, it copies a *Delete*(x) operation to U_{k+1} in order to remove from BH any occurrence of x with a larger key value. During this process **Apply-Updates** does not change the splitter elements. Therefore, clearly it never violates invariants 1, 2 and 4. However, invariant 3 may be violated if B_k contains more than 2^k elements after the updates. But as soon as $|B_k| > 0$ is detected after returning from **Apply-Updates** for some k the **Delete-Min** function enters the redistribution step. It redistributes the elements in B_k to lower-level element buffers and to U_{k+1} (if exists) by calling the function **Redistribute-Elements**. **Redistribute-Elements** checks whether invariant 3 is violated, and if so, it fixes it by keeping in B_k only 2^k elements having the smallest 2^k keys and moving the rest to U_{k+1} as *Sink* operations with the current time-stamp. It appropriately lowers the boundaries of all levels with empty ranges after level k including the left boundary of the first level with a non-empty range, but this fixation does not violate any invariant. Starting with $i = k-1$ downto $i = 0$, in iteration i it finds 2^i elements in B_i (all if B_i contains at most 2^i elements) having the smallest 2^i keys and moves them to B_{i-1} (moves out of BH if $i = 0$), and leaves the rest to B_i and fixes the level boundaries appropriately (and assigning empty ranges to empty buffers). At the termination of this loop only the element with the smallest key in the entire BH is left undistributed which is returned. This loop does not violate invariants 1, 2 and 4 and terminates with invariant 3 restored. Therefore, all invariants hold again at termination of **Delete-Min**(). □

2.1.5 I/O Complexity

First we prove the following lemma:

LEMMA 2. *For $1 \leq k \leq r - 1$, every empty U_k receives at most 3 batches of updates before U_k is applied on B_k and emptied again.*

PROOF. When level k is initialized (only once when BH was first created) or reinitialized (during a *Delete-Min* operation), U_k is set to be empty. Let us consider U_k after such an initialization/reinitialization. The first time this empty U_k receives a batch of updates is when **Apply-Updates** for level $k-1$ is called. This call to **Apply-Updates** may leave B_{k-1} in either of the following two states:

(1) B_{k-1} empty: **Apply-Updates** for level k will be called immediately which will empty U_k.

(2) B_{k-1} non-empty: **Redistribute-Elements** for level $k - 1$ will be called immediately which may push a new batch of updates (*Sink* operations) into U_k. It will empty B_{k-1} and also assign an empty range to level $k - 1$. The next time U_k will be accessed when **Apply-Updates** for level $k - 1$ is called again. This time **Apply-Updates** may push another batch of updates into U_k and will definitely leave B_{k-1} empty (since level $k - 1$ has an empty range now). This leaves us in a situation exactly as in case (1).

Therefore, for $1 \leq k \leq r - 1$, every empty U_k receives at most 3 batches of updates before U_k is emptied again. □

This lemma has the following implications:

- Each entry of A_s has constant size and thus sequential access of A_s will incur $\mathcal{O}(\frac{1}{B})$ amortized cache-misses per access per entry.
- Merging the segments of U_k (step 1 of **Apply-Updates**) before applying them on B_k will incur only $\mathcal{O}(\frac{1}{B})$ amortized cache-misses per update in U_k.

We also have the following three observations:

(1) Whenever a *Decrease-Key*/*Sink* operation inserts an element into an element buffer B_i during **Apply-Updates**, it may cause the buffer to overflow, i.e., the buffer may contain more than 2^i elements after the updates are applied. Since B_i contained at most 2^i elements before the updates and each *Decrease-Key*/*Sink* operation adds at most one element to B_i, for each excess element (x, k_x) we can find a distinct *Decrease-Key*/*Sink* operation that can be held responsible for the eviction of (x, k_x) from B_i. Each eviction from B_i causes a *Sink* operation to be inserted into U_{i+1}. Suppose a *Decrease-Key*(x, k_x)/*Sink*(x, k_x) operation in U_i is held responsible for the eviction of $(x', k_{x'})$ from B_i which generates a *Sink*$(x', k_{x'})$ operation in U_{i+1}. Then that *Decrease-Key*(x, k_x)/*Sink*(x, k_x) operation in level i can be thought of as generating the *Sink*$(x', k_{x'})$ operation in level $i + 1$. Observe that when the *Sink*$(x', k_{x'})$ is generated its generator no longer exists in the queue: if it was a *Decrease-Key* operation it was converted to a *Delete* operation, and it was simply discarded if it was a *Sink* operation. Thus every existing *Sink* operation in the queue can be traced back to a *Decrease-Key* operation following a chain of discarded *Sink*s. Therefore, we can say that a *Decrease-Key* is converted to a (*Delete*, *Sink*) pair after it inserts an element into a level that causes an eviction, and that pair proceeds level by level with possibly a different *Sink* operation in each level until at some level the *Sink* operation fails to evict an element at which point the pair turns into a single *Delete*.

(2) The ith largest element in an array of X elements can be found cache-obliviously in $\mathcal{O}(1 + \frac{X}{B})$ I/Os, where $1 \geq i \geq |X|$ [14]. If $|X| \leq B$, selection can be performed without

any extra I/Os, provided that the elements already reside in internal memory. In general, if at least the first $\Theta(B)$ elements of the array are always kept in internal memory, selection requires $\mathcal{O}(1+\frac{X}{B}) = O(\frac{X}{B})$ I/Os.

(3) For $M = \Omega(B)$, a list of X elements can be sorted cache-obliviously in $\mathcal{O}(\frac{X}{B}\log_2 X)$ I/Os using a simple I/O-efficient version of ordinary *binary mergesort*.

We will now prove the I/O bounds for BH operations.

LEMMA 3. *A BH containing at most N elements at any time, supports* Delete, Delete-Min *and* Decrease-Key *operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os each under the tall cache assumption, and in $\mathcal{O}(\frac{1}{B}\log_2 N)$ amortized I/Os each without the tall cache assumption.*

PROOF. For simplicity, we will assume that a *Decrease-Key* operation is inserted into U_0 as an ordered pair of operations $\langle$*Decrease-Key*, *Dummy*$\rangle$. After the successful application of that *Decrease-Key* operation on some B_i, the *Decrease-Key* operation in the ordered pair moves to U_{i+1} as a *Delete* operation, and the *Dummy* operation either turns into an element in B_i, or moves to U_{i+1} as a *Sink* operation (see observation (1) above for the validity of this assertion). Thus a *Decrease-Key* operation will be counted as two operations until it is applied on some element buffer.

For $0 \le i \le r-1$, let u_i be the number of operations in U_i, and b_i be the number of elements in B_i. If H is the current state of BH, we define the *potential* of H as

$$\Phi(H) = 4ru_0 + \sum_{i=1}^{r-1}(3r-i)u_i + \sum_{i=0}^{r-1}(i+1)b_i$$

Note that each component of the actual cost that we will calculate will have a $\Theta(\frac{1}{B})$ factor associated with it, which we will drop for simplicity. Therefore, the calculated amortized cost will have to be multiplied by the largest such dropped constant $\alpha = \Theta(\frac{1}{B})$ in order to obtain the final amortized cost.

First let us consider the amortized cost of a *Reconstruct* operation. At the start of the operation $\sum_{i=0}^{r-1} u_i \ge \sum_{i=0}^{r-1} b_i$, and thus the actual cost of performing steps 1 to 4 is $r\sum_{i=0}^{r-1} u_i$. After this reconstruction process the update buffers are left empty with each update contributing at most one element to the element buffers. This results in a drop of potential which is at least $r\sum_{i=0}^{r-1} u_i$. Therefore, the amortized cost of a *Reconstruct* operation is at most $r\sum_{i=0}^{r-1} u_i - r\sum_{i=0}^{r-1} u_i = 0$.

The amortized cost of a *Delete* operation is $= 1 + 4r = O(\log_2 N)$, and that of a *Decrease-Key* operation is $= 2(1+4r) = O(\log_2 N)$. We now consider the amortized cost of a *Delete-Min* operation. Let us calculate the actual cost of the operation first:

- The cost of sorting the updates in U_0 is at most ru_0.
- Let $k \ge 0$ be the smallest index such that B_k is non-empty after applying updates on it. Then the total cost for examining (read/copy/move/change) the updates is $= \sum_{i=0}^{k}(k-i+1)u_i$.

Note that this cost includes the cost of moving the excess elements (if any) from B_k to U_{k+1} as *Sink* operations since each of these moves can be assigned to a *Decrease-Key*/*Sink* operation from a lower-level update buffer.

- The cost of examining the elements in $B_0, B_1, \ldots, B_{k-1}$ during updates is $= \sum_{i=0}^{k-1} b_i$.
- The cost of examining the elements of B_k during updates is $\Theta(b_k)$. Let B_k contain $b'_k (\le 2^k)$ elements after applying the updates and removing the excess elements if necessary. During the removal of excess elements these b'_k elements incur a cost of $\Theta(b'_k)$. Now consider the process of redistributing these elements to lower-level buffers. The cost of applying the first selection step is $\Theta(b'_k)$. But the total cost of applying the remaining selection steps is also at most $\Theta(b'_k)$ since in this case the number of elements on which selection is applied is halved in every step. Thus the total cost incurred is $= \max(b_k, b'_k)$.
- The cost of accessing the A_s stack is at most r.

Thus, the actual cost $c \le ru_0 + \sum_{i=0}^{k}(k-i+1)u_i + \sum_{i=0}^{k-1} b_i + \max(b_k, b'_k) + r$

Let us now calculate the change in potential of H:

- Change in potential due to changes in update buffers is $\le -4ru_0 - \sum_{i=1}^{k}(3r-i)u_i + (3r-k-1)\sum_{i=0}^{k} u_i$, since the minimum decrease in potential of an element in $U_i, i \le k$ occurs if that element is moved to U_{k+1} (rather than being moved to some B_j or being deleted from BH).
- To calculate the change in potential due to changes in element buffers first note that before redistribution all buffers $b_0, b_1, \ldots, b_{k-1}$ were emptied which caused a potential drop of $\sum_{i=0}^{k-1} b_i$ at that point. During redistribution, however, the b'_k elements in B_k were all moved to lower-level element buffers. Since before this *Delete-Min* operation each of these elements either belonged to B_k or was in one of the update buffers as an update, the redistribution will cause a potential drop of at least 1 for each element, and thus a total potential drop of at least b'_k. But if $b_k > b'_k$, the potential drop is at least b_k.

Thus, $\Phi(H_{after}) - \Phi(H_{before}) \le -ru_0 - \sum_{i=0}^{k}(k-i+1)u_i - \sum_{i=0}^{k-1} b_i - \max(b_k, b'_k)$, hence, amortized cost, $\widehat{c} = c + (\Phi(H_{after}) - \Phi(H_{before})) \le r = O(\log_2 N)$, and the amortized number of I/O's per operation is $\mathcal{O}(\frac{1}{B}\log_2 N)$.

If we assume a tall cache, we have $N \gg M = \Omega(B^{1+\epsilon})$ for some $\epsilon > 0$, implying $\log_2 N = O(\log_2\frac{N}{B})$. Therefore, BH supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os each under the tall cache assumption. □

Some Extensions. In addition to the amortized I/O bounds, we observe that amortized *time* cost of each BH operation is $\mathcal{O}(\log N)$. The *Find-Min* operation can be supported in constant time with no I/O, and *Increase-Key* can be supported in the same bounds as the other operations.

Removing the Tall Cache Assumption. We can remove the need for a tall cache in BH while still supporting each operation in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os by using a strategy from [4], namely restricting the size of each U_i. We sketch this modification: We maintain an additional invariant that the number of updates in U_i $(0 \le i \le r-1)$ is at most 2^i. If U_i overflows, the following function (**Fix-U**) is called with parameter i in order to restore the invariant.

Function Fix-U(i)

Step 1: Pop from S_B elements residing above the elements of B_i, and push them into a global stack S'_B.

Step 2: Apply the updates in U_i on B_i.

Step 3: If $i < r-1$, move appropriate updates from U_i and elements from B_i to U_{i+1}. Empty U_i.

Step 4: If $i < r-1$ and $|U_{i+1}| > 2^{i+1}$, call **Fix-U**$(i+1)$.

Step 5: Pop the elements from S'_B that were pushed into it in step 1, and push them back to S_B.

After inserting each *Delete* or *Decrease-Key* operation into U_0, if $|U_0| > 2^0 = 1$, we call **Fix-U**(0). After performing a *Delete-Min* operation, we call **Fix-U**(k) if $|U_k| > 2^k$, where k is the smallest index of a non-empty update buffer.

Cache-Aware Model. BH can be modified to support *Decrease-Key*, *Delete* and *Delete-Min* operations in $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$ amortized I/Os each in the cache-aware model as follows. For some $q = \log_2(\beta M)$, where $\beta < 1$ is a constant, BH will now consist of only $r - q$ levels: $q, q+1, \ldots, r-1$. The element buffer B_q (containing at most $2^q = \beta M$ elements) will be kept in internal memory and all new *Decrease-Key*/*Delete* operations will be inserted into U_q. The first q entries of A_s will also be kept in internal memory. Each operation will be charged for $\mathcal{O}(\frac{1}{B})$ amortized I/Os per level for $r - q = \log_2\frac{N}{M}$ levels, i.e., $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{M})$ amortized I/Os in total. This version of BH does not require a tall cache.

2.2 Cache-Oblivious Tournament Tree

A Cache-Oblivious Tournament Tree (COTT) with N elements is a static binary tree with N leaves numbered 1 through N from left to right. The root of the tree is denoted by R, and T_v denotes the subtree rooted at any node v. Each node v stores an ordered pair (x_v, k_v), where x_v is an element corresponding to a leaf in T_v and k_v is a key of x_v. Each internal node v has an associated stack S_v.

The supported operations are propagated lazily from the root to appropriate leaves and the following two invariants are maintained:

INVARIANT 5. *For each internal node v, each entry in S_v is a* Decrease-Key *operation to be performed on a leaf of T_v. Thus the stacks associated with the nodes on the path from a leaf to the root together contain all* Decrease-Keys *to be performed on that leaf.*

INVARIANT 6. *The ordered pair (x_v, k_v) stored in node v refers to the element x_v corresponding to a leaf in T_v, having the minimum key k_v taking into account only the operations (*Delete*,* Delete-Min *and* Decrease-Key*) seen by v so far. Thus, at any time, the root of the tree stores the element with the minimum key value in the whole tree.*

Initially, the elements in all leaves are assigned a key value of $+\infty$. All stacks are empty and the ordered pair in each internal node refers to the leftmost leaf in the corresponding subtree. Thus both invariants hold initially.

Decrease-Keys are performed lazily. Whenever a *Decrease-Key* operation arrives, it is pushed on to the stack S_R of the root node and the minimum value at the root is updated if necessary. Thus, invariants 5 and 6 are maintained.

A *Delete-Min* operation is a special case of the *Delete* operation: it reads the ordered pair (x_R, k_R) from the root of the tree and performs a *Delete*(x_R) operation. A *Delete*(x) operation is performed as follows:

Function Delete(x)

Step 1 (Distribution Step): We follow the path from the root to the leaf corresponding to x. At each internal node v with children v_1 and v_2 on this path we pop all *Decrease-Key* operations from S_v and distribute them to S_{v_1} and S_{v_2}, and also update (x_{v_1}, k_{v_1}) and (x_{v_2}, k_{v_2}) if necessary. However, if a child node is a leaf, we discard any *Decrease-Key* operation it receives after updating the ordered pair stored in that leaf.

Step 2 (Fixing Step): We set the key value of element x at the leaf to $+\infty$ and propagate this change along the leaf to root path from x. At each internal node v with children v_1 and v_2 on this path, we set (x_v, k_v) to (x_{v_1}, k_{v_1}) if $k_{v_1} \le k_{v_2}$, otherwise we set it to (x_{v_2}, k_{v_2}).

LEMMA 4. *A COTT with N elements supports* Delete/Delete-Min *and* Decrease-Key *operations:*

(a) in $\mathcal{O}(\log_2 N)$ and $\mathcal{O}(\frac{1}{B}\log_2 N)$ amortized I/Os, respectively, in cache-oblivious model without a tall cache.

(b) in $\mathcal{O}(\log_2\frac{N}{B})$ and $\mathcal{O}(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os, respectively, in cache-oblivious model with a tall cache.

(c) in $\mathcal{O}(\log_2\frac{NB}{M})$ and $\mathcal{O}(\frac{1}{B}\log_2\frac{NB}{M})$ amortized I/Os, respectively, in cache-aware model.

PROOF. **(a)** During a *Delete* or *Delete-Min* operation 2 nodes are accessed at each of the $\mathcal{O}(\log_2 N)$ levels during the root to leaf path traversal in step 1 and the same holds for step 2. We charge the $\mathcal{O}(1)$ cache-misses for retrieving the node information and the stacks in each level to the current *Delete*/*Delete-Min* operation. Thus each *Delete*/*Delete-Min* is charged for $\mathcal{O}(\log_2 N)$ external memory accesses in total. On the other hand, each *Decrease-Key* operation is pushed and popped to a stack only once in each level which incurs $\mathcal{O}(\frac{1}{B})$ cache-misses per level. Each *Decrease-Key* operation also incurs $\mathcal{O}(\frac{1}{B})$ amortized cache-misses during its insertion into S_R. Thus each *Decrease-Key* operation is charged for $\mathcal{O}(\frac{1}{B}\log_2 N)$ amortized external memory accesses in total. **(b)** The claim follows from the observation that since $N \gg M$, the tall cache assumption implies $\log_2 N = O(\log_2\frac{N}{B})$. **(c)** We modify COTT so that, for some $q = \log_2\beta\frac{M}{B}$, where $\beta < 1$ is a constant, nodes in the first $q - 1$ levels do not have any stacks, and the topmost block of each of the stacks in level q are kept in the internal memory. □

3. CACHE-OBLIVIOUS SSSP

3.1 Undirected SSSP

The cache-aware undirected SSSP algorithm by Kumar & Schwabe [11] (see [10] for a description) can be made cache-oblivious by replacing both the primary and the auxiliary cache-aware priority queues used in that algorithm with buffer heaps. The primary priority queue is used to perform the standard operations for shortest path computation. Whenever a vertex with final distance $d[u]$ is settled, for each $(u, v) \in E[G]$, we will perform a *Decrease-Key*$((u, v), d[u] + w(u, v))$ operation on the auxiliary buffer heap to correct for spurious updates, i.e., the auxiliary buffer heap will treat edges, instead of vertices, as its elements.

The algorithm is given below (correctness proof is in [10]).

Step 1: Perform the following initializations:

(a): Initialize Q and Q' to be two empty BH. Each item in Q will be of the form (x, k_x) denoting an element x with key k_x, while each item in Q' will have the form $((x, y), k_{x,y})$ denoting an element (x, y) with key $k_{x,y}$.

(b): For each $v \in V[G]$, set $d[v] \leftarrow +\infty$.

(c): Perform *Decrease-Key*$_{(Q)}(s, 0)$.

Step 2: While Q is non-empty do:

(a): Set $(u, k) \leftarrow$ *Find-Min*$_{(Q)}()$.
Set $((u', v'), k') \leftarrow$ *Find-Min*$_{(Q')}()$.

(b): If $k \leq k'$ then:

(b_1): Perform $Delete_{(Q)}(u)$. Set $d[u] \leftarrow k$.
(b_2): For each neighbor v of u do:
$Decrease\text{-}Key_{(Q)}(v, d[u]+w(u,v))$.
$Decrease\text{-}Key_{(Q')}((u,v), d[u]+w(u,v))$.

Otherwise (if $k > k'$) do:

(b'_1): Perform $Delete_{(Q')}((u', v'))$.
Perform $Delete_{(Q)}(u')$.

I/O Complexity. If we use the cache-oblivious Buffer Heap, the algorithm requires $\mathcal{O}(V + \frac{E}{B}\log_2 \frac{V}{B})$ I/Os. If we use the cache-aware BH, we obtain a cache-aware undirected SSSP algorithm requiring $\mathcal{O}(V + \frac{E}{B}\log_2 \frac{V}{M})$ I/Os.

3.2 Directed SSSP

The SSSP algorithm we will describe in this section is a cache-oblivious implementation of Dijkstra's SSSP algorithm [7] with a Buffer Heap (or a COTT) Q. Additionally, we will use a cache-oblivious *Buffered Repository Tree* (BRT) structure D described in [2], in order to prevent a vertex whose shortest distance from the source vertex s has already been determined, from being reinserted into Q. The BRT structure maintains $\mathcal{O}(E)$ elements with keys in the range $[1 \ldots V]$ under the operations $Insert(v,u)$ and $Extract(u)$. The $Insert(v,u)$ operation inserts a new element v with key u, and the $Extract(u)$ operation reports and deletes from the structure all elements v with key u. The *Insert* and *Extract* operations are supported in $\mathcal{O}(\frac{1}{B}\log_2 V)$ and $\mathcal{O}(\log_2 V)$ amortized I/Os, respectively (or in $\mathcal{O}(\frac{1}{B}\log_2 \frac{V}{B})$ and $\mathcal{O}(\log_2 \frac{V}{B})$ amortized I/Os, respectively, assuming a tall cache).

The cache-oblivious directed SSSP algorithm consists of the following steps and for each vertex $v \in V[G]$, it stores the shortest distance from s to v in $d[v]$:

Step 1: For each vertex $v \in V[G]$, find the set $L_v = \{u|(u,v) \in E[G]\}$ by sorting the edges of G.

Step 2: Perform the following initializations:

(a): Initialize Q to be an empty BH (or a COTT).
(b): Initialize D to be an empty BRT capable of supporting key values in the range $[1 \ldots V]$.
(c): For each $v \in V[G]$, set $d[v] \leftarrow +\infty$.
(d): Perform a $Decrease\text{-}Key(s, 0)$ operation on Q.

Step 3: While Q is non-empty do:

(a): Perform a *Delete-Min* on Q to extract the vertex u with minimum key d_u. Set $d[u] \leftarrow d_u$.
(b): Let L' be the set of vertices to which u has an outgoing edge (obtained from the adjacency list of u). The list is augmented to contain the weight of the corresponding edge with each vertex. Sort L' in ascending order by vertex number.
(c): Perform $Extract(u)$ on D to obtain a set L'' of neighbors of u (to which u has an outgoing edge) whose shortest distance from s has already been determined. Sort L'' by vertex number.
(d): Scan L' and L'' simultaneously and for each vertex v in L' that does not occur in L'' perform a $Decrease\text{-}Key(v, d_u + w(u,v))$ operation on Q.
(e): For each vertex $v \in L_u$ (determined in step 1) perform an $Insert(u,v)$ operation on D.

3.2.1 Correctness

A standard implementation of Dijkstra's directed SSSP algorithm is through the use of a priority-queue Q with *Decrease-Key*. Priority-queue Q stores all vertices that are not yet *settled* (i.e., vertices whose shortest path length from s has not yet been finalized), and in each iteration of the algorithm, a vertex u is extracted from Q with a *Delete-Min* operation. The vertex u is provably settled at this point, and for each edge (u,v) such that v is not settled, i.e., such that v is on Q, a suitable *Decrease-Key* operation is performed on v in Q.

Our directed SSSP algorithm implements Dijkstra's algorithm as described above. However, if we had used one of our two cache-oblivious priority queues with *Decrease-Key* in the straightforward manner, we would have needed to make one external memory I/O for each edge (u,v) to determine whether or not v is settled. This is too expensive. Instead, we improve the performance using the BRT structure (the BRT has been used for breadth-first search and depth-first search in the cache-aware setting in [5] and in the cache-oblivious setting in [2]).

Informally, the BRT structure is used in our algorithm as follows. For every unsettled vertex u, the BRT structure stores an element v with key value u for each settled vertex v to which u has an outgoing edge. Thus when u itself settles (i.e., u is deleted from Q), one can identify all settled neighbors of u by an $Extract(u)$ operation on the BRT.

Since our directed SSSP algorithm is an an implementation of Dijkstra's algorithm, in order to prove the correctness it suffices to show that our use of BRT results in the *Decrease-Key* operation being performed on exactly the unsettled vertices in step 3(d).

LEMMA 5. *The cache-oblivious directed SSSP algorithm correctly implements Dijkstra's directed SSSP algorithm.*

PROOF. As described above, we only need to show that the *Decrease-Key* operation is performed on the correct set of vertices in each iteration of step 3.

At any time, let S denote the set of vertices extracted from Q so far in step 3(a) of the algorithm. Let $P(S_1, S_2)$ denote the set $\{(v,w)|v \in S_1 \wedge w \in S_2 \wedge (w,v) \in E[G]\}$.

We claim that the following invariant holds at the start of each iteration of the loop in step 3:

INVARIANT 7. *D contains only elements from the set S, and for each $v \in S$ and $u \in V[G] \setminus S$, such that $(u,v) \in E[G]$, there is an item (v,u) in D, where v is an element and u is its key. Thus, $P(S, V[G]) \supseteq D \supseteq P(S, V[G] \setminus S)$.*

Before the first iteration of the loop in step 3, $S = \emptyset$, and D is initialized to be empty. Thus the invariant holds trivially. We now need to prove that it is maintained correctly.

In step 3(a), vertex u is included in set S. Let S' denote the set S before this inclusion. Thus $S = S' \cup \{u\}$.

Since this invariant holds for S' up to and including step 3(b), we have,

$$P(S', V[G]) \supseteq D \supseteq P(S', V[G] \setminus S')$$

Step 3(c) deletes the set $\{(v,u)|v \in S' \wedge (u,v) \in E[G]\}$ from D. Thus after this deletion,

$$P(S', V[G]) \supseteq D \supseteq P(S', V[G] \setminus S)$$

Step 3(e) adds the set $\{(u,w)|w \in V[G] \wedge (w,u) \in E[G]\}$ to D. Thus, after this addition,

$$P(S, V[G]) \supseteq D \supseteq P(S, V[G] \setminus S)$$

Thus the invariant holds before every iteration of the loop in step 3. Hence step 3(c) correctly extracts from D all settled vertices to which vertex u has an outgoing edge, and step 3(d) avoids performing *Decrease-Key* operations on them.

Therefore the algorithm is a correct implementation of Dijkstra's algorithm. □

3.2.2 I/O Complexity

LEMMA 6. *Single source shortest paths in a directed graph can be computed cache-obliviously in* $\mathcal{O}((V + \frac{E}{B}) \cdot \log_2 \frac{V}{B})$ *I/Os using a BH/COTT under the tall cache assumption.*

PROOF. We will prove the claim for COTT only. The proof for BH is then straight-forward.

Step 1 requires $\mathcal{O}(V + \frac{E}{B})$ I/Os to copy the edges from the adjacency lists, $\mathcal{O}(\frac{E}{B} \log_2 \frac{E}{B})$ I/Os to sort them and $\mathcal{O}(V + \frac{E}{B})$ I/Os to create a separate list for each vertex.

Steps 2(a) and 2(b) require $\mathcal{O}(1 + \frac{1}{B} \log_2 \frac{V}{B})$ and $\mathcal{O}(V)$ I/Os, respectively. Step 2(c) requires $\mathcal{O}(1 + \frac{V}{B})$ I/Os. We will include the I/O cost of *Decrease-Key*$(s, 0)$ in step 2(d) in the total I/O cost of all *Decrease-Keys* in the algorithm.

Step 3(a) requires $\mathcal{O}(\log_2 \frac{V}{B})$ amortized I/Os to perform the *Delete-Min* and $\mathcal{O}(1)$ I/O to store/report the shortest distance/path information for the extracted vertex u. Step 3(b) requires $\mathcal{O}(1 + \frac{out(u)}{B})$ I/Os to read the adjacency list of u, where $out(u)$ denotes the out-degree of vertex u. Sorting L' requires $\mathcal{O}(\frac{out(u)}{B} \log_2 \frac{out(u)}{B})$ I/Os. Step 3(c) requires $\mathcal{O}(\log_2 \frac{V}{B})$ amortized I/Os to perform the *Extract* operation and $\mathcal{O}(\frac{out(u)}{B} \log_2 \frac{out(u)}{B})$ I/Os to do the sorting. Step 3(d) requires $\mathcal{O}(\frac{out(u)}{B})$ I/Os to read L' and L'', and $\mathcal{O}(\frac{out(u)}{B} \log_2 \frac{V}{B})$ amortized I/Os for the *Decrease-Keys*. Step 3(e) requires $\mathcal{O}(1 + \frac{in(u)}{B})$ I/Os to read the list L_u, where $in(u)$ is the in-degree of u. The *Insert* operations require at total of $\mathcal{O}(\frac{in(u)}{B} \log_2 \frac{V}{B})$ amortized I/Os.

Hence, a single iteration of step 3 on a vertex u requires $\mathcal{O}(\log_2 \frac{V}{B} + \frac{in(u)+out(u)}{B} \log_2 \frac{V}{B})$ amortized I/Os, which sums to $\mathcal{O}((V + \frac{E}{B}) \cdot \log_2 \frac{V}{B})$ over all iterations of step 3. The cost of this step dominates the cost of steps 1 and 2. □

4. REFERENCES

[1] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

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

[3] G.S. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation, LNCS 2518*, pp. 219–228, Nov. 2002.

[4] G.S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths. To appear in *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, July 2004.

[5] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 859–860, 2000.

[6] 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 ACM-SIAM Symposium on Discrete Algorithms*, pp. 139–149, 1995.

[7] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

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

[10] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In U. Meyer, P. Sanders, and J.F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, *LNCS* 2625. Springer, 2003.

[11] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pp. 169–177, 1996.

[12] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the European Symposium on Algorithms, LNCS 2832*, pp. 434–445, Sep. 2003.

[13] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pp. 713–722, San Francisco, CA, 2002.

[14] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, June 1999.

[15] J.S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.