External-Memory Exact and Approximate All-Pairs Shortest-Paths in Undirected Graphs *

Rezaul Alam Chowdhury   Vijaya Ramachandran

UTCS Technical Report TR-04-38

August 31, 2004

Abstract

We present several new external-memory algorithms for finding all-pairs shortest paths in a V-node, E-edge undirected graph. Our results include the following, where B is the block-size and M is the size of internal memory. We present cache-oblivious algorithms with $\mathcal{O}(V \cdot \frac{E}{B} \log_{\frac{M}{B}} \frac{E}{B})$ I/Os for all-pairs shortest paths and diameter in unweighted undirected graphs. For weighted undirected graphs we present a cache-aware APSP algorithm that performs $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + \frac{E}{B} \log \frac{V}{B}))$ I/Os. We also present efficient cache-aware algorithms that find paths between all pairs of vertices in an unweighted graph whose lengths are within a small additive constant of the shortest path length.

All of our results improve earlier results known for these problems. For approximate APSP we provide the first nontrivial results. Our diameter result uses $\mathcal{O}(V + E)$ extra space, and all of our other algorithms use $\mathcal{O}(V^2)$ space. In our work on external-memory algorithm for APSP in weighted undirected graphs we develop the notion of a *slim data structure* that might have other applications in external-memory computations.

1 Introduction

1.1 The APSP Problem

The *all-pairs shortest paths* (APSP) 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 $E[G]$, and a non-negative real-valued weight function w over $E[G]$, the APSP problem seeks to find a path of minimum total edge-weight between every pair of vertices in $V[G]$. For any pair of vertices $u, v \in V$, the path from u to v having the minimum total edge-weight is called the *shortest path* from u to v, and the sum of all edge-weights along that path is the *shortest distance* from u to v. The *diameter* of G is the longest shortest distance between any pair of vertices in G. For unweighted graphs the APSP problem is also called the all-pairs breadth-first-search (AP-BFS) problem. By V and E we denote the size of $V[G]$ and $E[G]$, respectively.

Considerable research has been devoted to developing efficient internal-memory approximate and exact APSP algorithms [18]. 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 APSP algorithms.

*Dept of Comp Sci, UT-Austin, Austin, TX 78712. Email: {shaikat,vlr}@cs.utexas.edu.

This work was supported in part by NSF CCR-9988160.

1.2 Cache-Aware Algorithms

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. 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].

A straight-forward method of computing AP-BFS (or APSP) is to simply run a BFS (or *single source shortest path* (SSSP) algorithm, respectively) from each of the V vertices of the graph. External BFS on an unweighted undirected graph can be solved using either $(V + sort(E))$ I/Os [15] or $\mathcal{O}(\sqrt{\frac{VE}{B}} + sort(E))$ I/Os [13]. External SSSP on an undirected graph with general non-negative edge-weights can be computed in $\mathcal{O}(V + \frac{E}{B}\log\frac{V}{M})$ I/Os using the cache-aware *Buffer Heap* [8]. There are also some results known for external SSSP on undirected graphs with restricted edge-weights [14]. The I/O complexity for external AP-BFS (or APSP) is obtained by multiplying the I/O complexity of external BFS (or SSSP) by V.

Very recently Arge et al. [6] proposed an $\mathcal{O}(V \cdot sort(E))$ I/O cache-aware algorithm for AP-BFS on undirected graphs. Their algorithm works by clustering nearby vertices in the graph, and running concurrent BFS from all vertices of the same cluster. This same algorithm can be used to compute unweighted diameter of the graph in the same I/O bound and $\mathcal{O}(\sqrt{VEB})$ additional space. They also present another algorithm for computing the unweighted diameter of sparse graphs ($E = \mathcal{O}(V)$) in $\mathcal{O}(sort(kV^2B^{\frac{1}{k}}))$ I/Os and $\mathcal{O}(kV)$ space for any integer k, $3 \le k \le \log B$.

For undirected graphs with general non-negative edge-weights Arge et al. [6] proposed an APSP algorithm requiring $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}\log V} + sort(E)))$ I/Os, whenever $E \le \frac{VB}{\log V}$. They use a priority queue structure called the *Multi-Tournament-Tree* which is created by bundling together a number of I/O-efficient *Tournament Trees* [12]. The use of this structure reduces unstructured accesses to adjacency lists at the expense of increasing the cost of each priority queue operation.

1.3 The Cache-Oblivious Model

The main disadvantage of the two-level I/O 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* [11]. 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, and it is assumed that an optimal cache-replacement strategy is used.

No non-trivial algorithm is known for the AP-BFS and the APSP problems in the cache-oblivious model except for the method of running single BFS and SSSP, respectively, from each of the V vertices. In this model, BFS on an undirected graph can be performed using $\mathcal{O}(\sqrt{\frac{VE}{B}} + \frac{E}{B}\log V + MST(E))$ I/Os [7], and SSSP on an undirected graph with non-negative real-valued edge-weights can be solved in $\mathcal{O}(V + \frac{E}{B}\log\frac{V}{M})$ I/Os using the cache-oblivious Buffer Heap [8] or Bucket Heap [7]. (The result is stated as $\mathcal{O}(V + \frac{E}{B}\log\frac{V}{B})$ I/Os in both [8] and [7], but it was observed by the current authors and the second author in [7] that the amortized I/O cost is actually $\mathcal{O}(V + \frac{E}{B}\log\frac{V}{M})$ [17, 10].) The I/O complexity of the corresponding all-pairs version of the problem is obtained by multiplying the I/O complexity of the single-source version by V.

1.4 Our Results

In section 2 we present a simple cache-oblivious algorithm for computing AP-BFS on unweighted undirected graphs in $\mathcal{O}(V \cdot sort(E))$ I/Os, matching the I/O complexity of its cache-aware counterpart [6]. We use this algorithm to compute the diameter of an unweighted undirected graph in the same I/O bound and $\mathcal{O}(V+E)$ space. Our cache-oblivious algorithm is arguably simpler than the cache-aware algorithm in [6] and it has a better space bound for computing the diameter.

In section 3 we present the first nontrivial external-memory algorithm to compute approximate APSP on unweighted undirected graphs with small additive error. The algorithm is cache-aware, it uses $\mathcal{O}(\frac{1}{B^{\frac{2}{3}}} V^{2-\frac{2}{3k}} E^{\frac{2}{3k}} \log^{\frac{2}{3}(1-\frac{1}{k})} V + \frac{k}{B} V^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-\frac{1}{k}} V)$ I/Os, and it produces estimated distances with an additive error of at most $2(k-1)$, where $2 \leq k \leq \log V$ is an integer, and $E > V \log V$. The number of I/Os performed by our algorithm is close to being a factor of B smaller than the running time of the best internal-memory algorithm known for this problem [9]. For the special case $k = 2$, we present an alternate algorithm that performs better for large values of B; this algorithm builds on the internal-memory algorithm in [2].

In section 4 we introduce the notion of a *Slim Data Structure* for external-memory computation. This notion captures the scenario where only a limited portion of the internal memory is available to store data from the data structure; it is assumed, however, that while executing an individual operation of the data structure, the entire internal memory of size M is available for the computation. We describe and analyze the *Slim Buffer Heap* which is a slim data structure based on the Buffer Heap [8]. We use Slim Buffer Heaps in a *Multi-Buffer Heap* to solve the cache-aware exact APSP problem for undirected graphs with general non-negative edge-weights in $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + sort(E)))$ I/Os and $\mathcal{O}(V^2)$ space, whenever $E \leq \frac{VB}{\log^2 V}$. This improves on the result in [6] for weighted undirected APSP. We also believe that the notion of a Slim Data Structure is of independent interest and is likely to have other applications in external-memory computation.

2 Cache-Oblivious APSP and Diameter for Unweighted Undirected Graphs

In this section we present a cache-oblivious algorithm for computing all-pairs shortest paths and diameter in an unweighted undirected graph.

2.1 The Cache-Oblivious BFS Algorithm of Munagala and Ranade

Given a source node s, the algorithm of Munagala & Ranade [15] computes the BFS level of each node with respect to s. Let $L(i)$ denote the set of nodes in BFS level i. For $i < 0$, $L(i)$ is defined to be empty. Let $N(v)$ denote the set of vertices adjacent to vertex v, and for a set of vertices S, let $N(S)$ denote the multiset formed by concatenating $N(v)$ for all $v \in S$.

Algorithm MR-BFS(G)

The algorithm starts by setting $L(0) = \{s\}$. Then starting from $i = 1$, for each $i < V$, the algorithm computes $L(i)$ assuming that $L(i-1)$ and $L(i-2)$ have already been computed. Each $L(i)$ is computed in the following three steps:

Step 1: Construct $N(L(i-1))$ by $|L(i-1)|$ accesses to the adjacency lists, once for each $v \in L(i-1)$. This step requires $\mathcal{O}(|L(i-1)| + \frac{1}{B}|N(L(i-1))|)$ I/Os.

Step 2: Remove duplicates from $N(L(i-1))$ by sorting the nodes in $N(L(i-1))$ by node indices, followed by a scan and a compaction phase. Let us denote the resulting set by $L'(i)$. This step requires $\mathcal{O}(sort(|N(L(i-1))|))$ I/Os.

Step 3: Remove from $L'(i)$ the nodes occurring in $L(i-1) \cup L(i-2)$ by parallel scanning of $L'(i)$, $L(i-1)$ and $L(i-2)$. Since all these three sets are sorted by node indices the I/O complexity of this step is $\mathcal{O}(\frac{1}{B}(|N(L(i-1))| + |L(i-1)| + |L(i-2)|))$. The resulting set is the required set $L(i)$.

Since $\sum_i |L(i)| = \mathcal{O}(V)$ and $\sum_i |N(L(i))| = \mathcal{O}(E)$, total I/O complexity of this algorithm is $\mathcal{O}(\sum_i(|L(i)| + sort(|N(L(i))|) + \frac{1}{B}(|N(L(i))| + L(i)))) = \mathcal{O}(V + sort(E))$.

2.2 Cache-Oblivious APSP for Unweighted Undirected Graphs

In this section we describe a cache-oblivious APSP algorithm for unweighted undirected graphs using $\mathcal{O}(V \cdot sort(E))$ I/Os. Let $G = (V[G], E[G])$ be an unweighted undirected graph. By $d(u, v)$ we denote the shortest distance between two vertices u and v in G.

Our algorithm is based on the following observation which follows from triangle inequality and the fact that $d(u, v) = d(v, u)$ in an undirected graph:

Observation 1. *For any three vertices u, v and w in G, $d(u, w) - d(u, v) \leq d(v, w) \leq d(u, w) + d(u, v)$.*

Suppose for some $u \in V[G]$ we have already computed $d(u, w)$ for all $w \in V[G]$. We sort the adjacency lists in non-decreasing order by $d(u, \cdot)$, and by $A(j)$ we denote the portion of this sorted list containing adjacency lists of vertices w with $d(u, w) = j$. Now if v is another vertex in $V[G]$ then observation 1 implies that the adjacency list of any vertex w with $d(v, w) = i$, must reside in some $A(j)$ where $i - d(u, v) \leq j \leq i + d(u, v)$. Therefore, we can use observation 1 to compute $d(v, w)$ for all $w \in V[G]$ as follows:

Algorithm Incremental-BFS$(G, u, v, d(u, \cdot))$

Function: Given an unweighted undirected graph G, two vertices $u, v \in V[G]$, and $d(u, w)$ for all $w \in V[G]$, this algorithm computes $d(v, w)$ for all $w \in V[G]$. It is assumed that $E[G]$ is given as a set of adjacency lists.

Steps:

Step 1: Sort the adjacency lists of G so that adjacency list of a vertex x is placed before that of another vertex y provided $d(u, x) < d(u, y)$ or $d(u, x) = d(u, y) \wedge x < y$. Let $A(i)$, $0 \leq i < |V|$, denote the portion of this sorted list that contains adjacency lists of vertices lying exactly at distance i from u.

Step 2: To compute $d(v, w)$ for all $w \in V[G]$, run Munagala and Ranade's BFS algorithm with source vertex v. But step (1) of that algorithm is modified so that instead of finding the adjacency lists of the vertices in $L(i-1)$ by $|L(i-1)|$ independent accesses, they are found by scanning $L(i-1)$ and $A(j)$ in parallel for $\max\{0, i-1-d(u, v)\} \leq j \leq \min\{|V|-1, i-1+d(u, v)\}$.

Step 1 of **Incremental-BFS** requires $\mathcal{O}(sort(E))$ I/Os. In step 2 each $A(j)$ is scanned $\mathcal{O}(d(u, v))$ times. Since $\sum_j |A(j)| = \mathcal{O}(E)$, this step requires $\mathcal{O}(\frac{E}{B} d(u, v) + sort(E))$ I/Os. Thus the I/O complexity of **Incremental-BFS** is $\mathcal{O}(\frac{E}{B} d(u, v) + sort(E))$.

Since **Incremental-BFS** is actually an implementation of Munagala and Ranade's algorithm, its correctness follows from the correctness of that algorithm, and from observation 1 which guarantees that the set of $A(j)$'s scanned to find the adjacency lists of the vertices in $L(i-1)$ in step 2 of **Incremental-BFS** contains all adjacency lists sought.

We can use **Incremental-BFS** to perform BFS I/O-efficiently from all vertices of G. The following observation each part of which follows in a straight-forward manner from the properties of spanning trees, Euler Tours and shortest paths, is central to this extension:

Observation 2. *If ET is an Euler Tour of a spanning tree of an unweighted undirected graph G, then*

(a) *the number of edges between any two vertices x and y on ET is an upper bound on $d(x, y)$ in G,*

(b) *ET has $\mathcal{O}(V)$ edges, and*

(c) *each vertex of $V[G]$ appears at least once in ET.*

The extended algorithm (**AP-BFS**) is as follows:

Algorithm AP-BFS(G)

Steps:

Step 1:

(a) Find a spanning tree T of G.

(b) Construct an *Euler Tour* ET for T.

(c) Mark the first occurrence of each vertex on ET, and let $v_1, v_2, \ldots, v_{|V|}$ be the marked vertices in the order they appear on ET.

Step 2: Run Munagala and Ranade's original BFS algorithm with v_1 as the source vertex, and compute $d(v_1, w)$ for all $w \in V[G]$.

Step 3: For $i \leftarrow 2$ to $|V|$ do:

Call **Incremental-BFS** $(G, v_{i-1}, v_i, d(v_{i-1}, \cdot))$ to compute $d(v_i, w)$ for all $w \in V[G]$.

Correctness. Correctness of **AP-BFS** follows from the correctness of Munagala and Ranade's BFS algorithm and that of **Incremental-BFS**. Moreover, observation 2(c) ensures that BFS will be performed for each $v \in V[G]$.

I/O Complexity. Step 1(a) can be performed cache-obliviously in $\mathcal{O}(\min\{V + sort(E), sort(E) \cdot \log_2 \log_2 V\})$ I/Os [4]. In step 1(b) ET can also be constructed cache-obliviously using $\mathcal{O}(sort(V))$ I/Os [4]. Step 1(c) requires $\mathcal{O}(sort(E))$ I/Os. Step 2 requires $\mathcal{O}(V + sort(E))$ I/Os. Iteration i of step 3 requires $\mathcal{O}(\frac{E}{B} d(v_{i-1}, v_i) + sort(E))$ I/Os. Total number of I/O operations required by the entire algorithm is thus $\mathcal{O}(\frac{E}{B} \sum_{i=2}^{|V|} d(v_{i-1}, v_i) + V \cdot sort(E))$. Since by observation 2(a) and 2(b) we have $\sum_{i=2}^{|V|} d(v_{i-1}, v_i) = \mathcal{O}(V)$, the I/O complexity of **AP-BFS** reduces to $\mathcal{O}(V \cdot sort(E))$.

Space Complexity. Since the algorithm requires to output all $\Theta(V^2)$ pairwise distances its space requirement is $\Theta(V^2)$.

2.3 Cache-Oblivious Unweighted Diameter for Undirected Graphs

The **AP-BFS** algorithm can be used to find the unweighted diameter of an undirected graph cache-obliviously in $\mathcal{O}(V \cdot sort(E))$ I/Os. We no longer need to output all $\Theta(V^2)$ pairwise distances, and each iteration of step 3 of **AP-BFS** only requires the $\Theta(V)$ distances computed in the previous iteration or in step 2. Thus the space requirement is only $\Theta(V)$ in addition to the $\mathcal{O}(E)$ space required to handle the adjacency lists.

3 Cache-Aware Approximate APSP for Unweighted Undirected Graphs

In this section we present a family of cache-aware external-memory algorithms **Approx-AP-BFS**$_k$ for approximating all distances in an unweighted undirected graph with an additive error of at most $2(k-1)$, where $2 \leq k \leq \log V$ is an integer. The error is one sided. If $\delta(u, v)$ denotes the shortest distance between any two vertices u and v in the graph, and $\widehat{\delta}(u, v)$ denotes the estimated distance between u and v produced by the algorithm, then $\delta(u, v) \leq \widehat{\delta}(u, v) \leq \delta(u, v) + 2(k-1)$. Provided $E > V \log V$, **Approx-AP-BFS**$_k$ runs in $\mathcal{O}(kV^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-1/k} V)$ time, and triggers $\mathcal{O}(\frac{1}{B^{\frac{2}{3}}} V^{2-\frac{2}{3k}} E^{\frac{2}{3k}} \log^{\frac{2}{3}(1-\frac{1}{k})} V + \frac{k}{B} V^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-\frac{1}{k}} V)$ I/Os. This family of algorithms is the external-memory version of the family of $\mathcal{O}(kV^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-1/k} V)$ time internal-memory approximate shortest paths algorithms (**apasp**$_k$) introduced by Dor et al. [9] which is the most efficient algorithm available for solving the problem in internal memory.

The second term in the I/O complexity of **Approx-AP-BFS**$_k$ is exactly $(1/B)$ times the running time of the Dor et al. algorithm [9]. Though the first term in the I/O complexity of **Approx-AP-BFS**$_k$ has a smaller denominator ($B^{\frac{2}{3}}$), its numerator is smaller than the numerator of the second term when $E > V \log V$, thus reducing the impact of the first term in the overall I/O complexity.

3.1 The Internal-Memory Approximate AP-BFS Algorithm by Dor et al.

The internal-memory approximate APSP algorithm (**apasp**$_k$) in [9] receives an unweighted undirected graph $G = (V[G], E[G])$ as input, and outputs an approximate distance $\widehat{\delta}(u, v)$ between every pair of vertices $u, v \in V[G]$ with a positive additive error of at most $2(k - 1)$. Recall that a set of vertices D is said to dominate a set U if every vertex in U has a neighbor in D.

A high level overview of the algorithm is given below:

Algorithm DHZ-Approx-AP-BFS$_k(G)$

Step 1: For $i \leftarrow 1$ to $k - 1$ do:

(a) Set $s_i \leftarrow \frac{E}{V}(\frac{V \log V}{E})^{\frac{i}{k}}$

Step 2: Decompose G to produce the following sets:

(a) A sequence of vertex sets $D_1, D_2, \ldots, D_k$ of increasing sizes with $D_k = V[G]$. For $1 \leq i \leq k - 1$, D_i dominates all vertices of degree at least s_i in G.

(b) A decreasing sequence of edge sets $E_1 \supseteq E_2 \supseteq \ldots \supseteq E_k$, where $E_1 = E[G]$ and for $1 < i \leq k$ the set E_i contains edges that touch vertices of degree at most s_{i-1}.

(c) A set $E^* \subseteq E[G]$ which bears witness that each D_i dominates the vertices of degree at least s_i in G.

Step 3: For $i \leftarrow 1$ to k do:

(a) For each $u \in D_i$ do:

(a$_1$) Run SSSP from u on $G_i(u) = (V[G], E_i \cup E^* \cup (\{u\} \times V[G]))$

In each $G_i(u)$ the edges $E_i \cup E^*$ are unweighted edges of the input graph, but the edges $\{u\} \times V[G]$ are weighted, and to each such edge (u, v) an weight is attached which is equal to the current known best upper bound on the shortest distance from u to v.

Step 4: Return the smallest distance computed between every pair of vertices in step 2.

The algorithm maintains the invariant that after the ith iteration in step 2, the approximate distance computed by the algorithm from each $u \in D_i$ to each $v \in V[G]$ has an additive error of at most $2(i - 1)$. Thus after the kth iteration a surplus $2(k - 1)$ distance is computed between every pair of vertices in G.

3.2 Our Algorithm

Our algorithm adapts the Dor et al. algorithm (**DHZ-Approx-AP-BFS**$_k$) to obtain a cache-efficient implementation. In our adaptation we do not modify step 1 of **DHZ-Approx-AP-BFS**$_k$, and use the same sequence of values for $\langle s_1, s_2, \ldots, s_{k-1} \rangle$. In section 3.3 we describe an external-memory implementation of step 2 of **DHZ-Approx-AP-BFS**$_k$.

It turns out that the I/O-complexity of **DHZ-Approx-AP-BFS**$_k$ depends on the I/O-efficiency of the SSSP algorithm used in step 3(a$_1$). Therefore, we replace each SSSP algorithm with a more I/O-efficient BFS algorithm by transforming each $G_i(u)$ to an unweighted graph $G'_i(u)$ of comparable size. But in order to preserve the shortest distances from u to other vertices in $G_i(u)$, the weighted edges of $G_i(u)$ need to be replaced with a set of *directed* unweighted edges. This makes the graph $G'_i(u)$ partially directed, and we need to modify existing external undirected BFS algorithms to handle the partial directedness in $G'_i(u)$ efficiently. This is described in section 3.4.

There are two ways to apply the BFS: either we can run an independent BFS from each $u \in D_i$ as in step 3 of **DHZ-Approx-AP-BFS**$_k$, or we can run BFS incrementally from the vertices of D_i as in section 2.2. It turns out that running independent BFS is more I/O-efficient when $|D_i|$ is smaller (i.e., value of i is smaller), and incremental BFS is more I/O-efficient when $G'_i(u)$ is sparser (i.e., value of i is larger). Therefore, we choose a value of i at which switching from independent BFS to incremental BFS minimizes the I/O-complexity of the entire algorithm. The overall algorithm is described in section 3.5.

3.3 External-Memory Implementation of Step 2

A set of vertices D is said to *dominate* a set U if every vertex in U has a neighbor in D. It has been shown by Aingworth et al. [2] that there is always a set of size $\mathcal{O}(\frac{V \log V}{s})$ that dominates all the vertices of degree at least s in an undirected graph, and in [9] it has been shown that this set can be found deterministically in $\mathcal{O}(V + E)$ time. In this section we present an external-memory implementation of the internal-memory greedy algorithm described in [9] for computing this set. The external-memory version, which we call **Dominate**, requires $\mathcal{O}(V + \frac{V^2}{B} + \mathit{sort}(E))$ I/Os and $\mathcal{O}(V^2 + E \log V)$ time, which is sufficient for our purposes. The internal-memory algorithm uses a priority queue that supports *Delete-Max* and *Decrease-Key* (for implementing steps 2(a) and 2(e) in **Dominate**). But due to the lack of any such I/O-efficient priority queue we use linear scans to simulate those two operations leading to the $\frac{V^2}{B}$ term, and thus the I/O-complexity of **Dominate** is worse than what one would typically expect from an external-memory implementation of an $\mathcal{O}(V + E)$ time internal-memory algorithm.

The **Dominate** function receives an undirected graph $G = (V[G], E[G])$ and a *degree threshold* s as inputs, and outputs a pair (D, E^*), where D is a set of size $\mathcal{O}(\frac{V \log V}{s})$ dominating the set of vertices of degree at least s in G, and $E^* \subseteq E[G]$ is a set of size $\mathcal{O}(V)$ such that for every $u \in V[G]$ with degree at least s, there is an edge $(u, v) \in E^*$ with $v \in D$.

Algorithm Dominate(G, s)

Function: Given an undirected graph $G = (V[G], E[G])$ and a *degree threshold* s, this algorithm outputs a pair (D, E^*), where D is a set of size $\mathcal{O}(\frac{V \log V}{s})$ that dominates the vertices of degree at least s in G, and $E^* \subseteq E[G]$ is a set of size $\mathcal{O}(V)$ such that for every $u \in V[G]$ with degree at least s, there is an edge $(u, v) \in E^*$ with $v \in D$.

Steps:

Step 1: Perform the following initializations:

(a) Sort the adjacency lists of G by their corresponding vertex indices, and the vertices in each adjacency list by their own indices.

(b) Scan the sorted adjacency lists to compute the degree of each vertex, and collect the vertices of degree at least s in sorted order (according to vertex indices) into an initially empty list L_1. Each vertex in L_1 will be accompanied by its degree.

(c) Set $D \leftarrow \emptyset$, $E^* \leftarrow \emptyset$, and $L_2 \leftarrow \emptyset$. The list L_2 will be used to collect the dominated vertices in sorted order (by vertex indices).

Step 2: While $L_1 \neq \emptyset$ do:

(a) Scan L_1 to find and remove a vertex with the largest degree. Let this vertex be u and A_u be its adjacency list.

(b) Add u to D and L_2 maintaining the sorted order of L_2.

(c) Scan A_u and L_2 in parallel and remove from A_u any vertex appearing in L_2.

(d) Add the vertices in A_u to L_2 by scanning both lists in parallel.

(e) Scan L_1 and A_u in parallel and decrease the degree of each vertex in L_1 that appears in A_u. Remove the vertices with degree zero from L_1.

(f) For each $v \in A_u$ do:

- Add (v, u) to E^*.
- Scan L_1 and v's adjacency list A_v in parallel, and decrease the degree of each vertex in L_1 that appears in A_v. Remove the vertices with degree zero from L_1.

Correctness of Dominate. Since **Dominate** is a straight-forward external-memory implementation of the internal-memory greedy algorithm for finding dominating sets described in [9], its correctness directly follows from the correctness of that algorithm.

I/O Complexity of Dominate. Step 1(a) requires $\mathcal{O}(sort(E))$ I/Os and 2(a) requires $\mathcal{O}(\frac{E}{B})$ I/Os. Thus step 1 requires at most $\mathcal{O}(sort(E))$ I/Os. In step 2, the adjacency list of each vertex in G is loaded at most twice, and scanned $\mathcal{O}(1)$ times. In each iteration of step 2, L_1 and L_2 are also scanned only a constant number of times. Thus step 2 requires $\mathcal{O}(V + \frac{V^2}{B})$ I/Os. Therefore, I/O complexity of **Dominate**(G, s) is $\mathcal{O}(V + \frac{V^2}{B} + sort(E))$.

We describe another function, called **Decompose**, which is an external-memory version of an internal-memory function with the same name described in [9], and uses the **Dominate** function as a subroutine. The function receives an undirected graph $G = (V[G], E[G])$, and a decreasing sequence $s_1 > s_2 > \ldots > s_{k-1}$ of degree thresholds as inputs. It produces a decreasing sequence of edge sets $E_1 \supseteq E_2 \supseteq \ldots \supseteq E_k$, where $E_1 = E[G]$ and for $1 < i \le k$ the set E_i contains edges that touch vertices of degree at most s_{i-1}. Clearly, $|E_i| \le V s_{i-1}$ for $1 < i \le k$. This function also produces a sequence of dominating sets $D_1, D_2, \ldots, D_k$, and an edge set E^*. For $1 \le i < k$ the set D_i dominates all vertices of degree greater than s_i, while D_k is simply $V[G]$. The set $E^* \subseteq E$ is a set of edges such that if the degree of a vertex u is greater than s_i then there exists an edge $(u, v) \in E^*$ with $v \in D_i$. Clearly $|E^*| \le kV$.

Algorithm Decompose$(G, \langle s_1, s_2, \ldots, s_{k-1} \rangle)$

Function: Given an undirected graph $G = (V[G], E[G])$ and a decreasing sequence $s_1, s_2, \ldots, s_{k-1}$ of degree thresholds, this algorithm outputs a sequence of edge sets $E_1 \supseteq E_2 \supseteq \ldots \supseteq E_k$, where $E_1 = E[G]$ and for $1 < i \le k$ the set E_i contains edges that touch vertices of degree at most s_{i-1}. It also outputs dominating sets $D_1, D_2, \ldots, D_k$, and an edge set E^*. For $1 \le i < k$ the set D_i dominates all vertices of degree greater than s_i, while D_k is simply $V[G]$. The set $E^* \subseteq E$ is such that if $deg(u) > s_i$ then there exists an edge $(u, v) \in E^*$ with $v \in D_i$, where $deg(u)$ denotes the degree of vertex u.

Steps:

Step 1: Perform the following initializations:

(a) Sort the adjacency lists of G by their corresponding vertex indices, and the vertices in each adjacency list by their own indices.

(b) Scan the sorted adjacency lists to compute the degree of each vertex.

Step 2:

(a) For $i \leftarrow 2$ to k do:

Scan the adjacency lists to produce the set
$E_i \leftarrow \{(u, v) \in E[G] \mid deg(u) \le s_{i-1} \vee deg(v) \le s_{i-1}\}$.

(b) For $i \leftarrow 1$ to $k - 1$ do:

$(D_i, E_i^*) \leftarrow Dominate(G, s_i)$

(c) Set $E_1 \leftarrow E$, $D_k \leftarrow V$, and $E^* \leftarrow \bigcup_{i=1}^{k-1} E_i^*$

I/O Correctness of Decompose. The correctness of this function directly follows from the internal-memory **Decompose** function in [9].

I/O Complexity of Decompose. The I/O cost of step 1 is $\mathcal{O}(sort(E))$. Step 2(a) requires $\mathcal{O}(k\frac{E}{B})$ I/Os. Step 2(b) requires $\mathcal{O}(k(V + \frac{V^2}{B}))$ I/Os in total since step 1(a) of **Dominate** can now be eliminated. Step 2(c) can be implemented in $\mathcal{O}(\frac{kV}{B} + \frac{E}{B})$ I/Os. Thus the I/O complexity of **Decompose** is $\mathcal{O}(k(V + \frac{V^2}{B}) + sort(E))$.

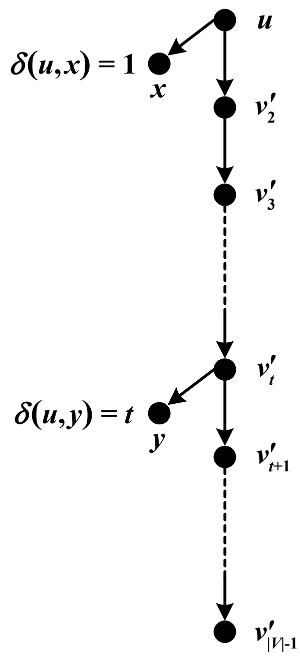

Figure 1: The directed unweighted edges that replace the undirected weighted edges of $G_i(u)$.

3.4 Replacing SSSP by BFS in Step 3(a_1)

For $i = 1, 2, \ldots, k$, in step 3(a_1) **DHZ-Approx-AP-BFS**$_k$ runs an SSSP algorithm from each $u \in D_i$ on a graph $G_i(u) = (V, E_i(u))$, where $E_i(u) = E_i \cup E^* \cup (\{u\} \times V)$. The edges $E_i \cup E^*$ are the original edges of the graph. But the edges $\{u\} \times V$ are not necessarily the edges of the input graph, and to such an edge (u, v) an weight of $\widehat{\delta}(u, v)$ is attached, where $\widehat{\delta}(u, v)$ is the current best known upper bound on the shortest distance from u to v in G. Initially, $\widehat{\delta}(u, v) = 1$ if $(u, v) \in E[G]$ and $\widehat{\delta}(u, v) = \infty$ otherwise.

Since external-memory BFS is more I/O-efficient than external-memory SSSP, we replace the SSSP in step 3(a_1) with a BFS algorithm. But this requires us to transform the weighted graph $G_i(u)$ into an unweighted graph of comparable size.

Transforming $G_i(u)$ into an Unweighted Graph. Since the distances we compute are non-negative integers smaller than $|V|$, we can, in fact, transform $G_i(u)$ into an unweighted graph $G'_i(u)$ by introducing $|V| - 2$ new vertices along with at most $2|V| - 3$ new unweighted directed edges instead of the weighted undirected edges of $\{u\} \times V$ while preserving the shortest distances from u to all other vertices in V. We introduce $|V| - 2$ new vertices $v'_2, v'_3, \ldots, v'_{|V|-1}$, and introduce the directed edges $(u, v'_2), (v'_2, v'_3), (v'_3, v'_4), \ldots, (v'_{|V|-2}, v'_{|V|-1})$. For each $v \in V[G]$ with $\widehat{\delta}(u, v) = 1$, we add a directed edge (u, v), and for each $v \in V[G]$ with $2 \le \widehat{\delta}(u, v) = t \le |V| - 1$, we add a directed edge (v'_t, v) (see Figure 1). The resulting graph $G'_i(u)$ is partially directed.

We have the following lemma:

Lemma 3.1. *The unweighted partially directed graph $G'_i(u)$ obtained from the weighted undirected graph $G_i(u) = (V, E_i(u))$ preserves the shortest distances from u to all other vertices in V.*

Proof. We observe that for every $v \in V$ to which a finite upper bound on the shortest distance from u is known, we introduce exactly one u to v path containing only directed edges. Let us call a path that contains no directed edges *Old Path*, and a path that contains at least one directed edge *New Path*. Now suppose $G'_i(u)$ contains a shortest path from u to some vertex $v \in V$ that is shorter than the shortest u to v path in $G_i(u)$. This path must be of the form $P_1 \cdot P_2$, where P_1 is a subpath (new or old) from u to some vertex $w \in V (w \neq u, v)$, and P_2 is a subpath (new) from w to v. But since u is the only entry point to the new directed edges introduced in $G'_i(u)$, P_2 must contain a path from u to v. Thus the path $P_1 \cdot P_2$ is not simple, and so cannot be a shortest path. ☐

Handling the Partial Directedness in $G'_i(u)$. We can modify the **MR-BFS** algorithm in section 2.1 to correctly handle the partial directedness in $G'_i(u)$ with only $\mathcal{O}(scan(E) + sort(V))$ I/O overhead, and thus without changing its I/O complexity. The algorithm will receive $G'_i(u)$ as an undirected graph, and will implicitly handle the edges that are intended to be directed. It must ensure the following

(a) $L(i)$ must not contain any other v'_j's except v'_{i+1}, and

(b) if the BFS level of a vertex v is less than i, then any edge (v'_{i+1}, v) must not force v to be included in $L(i)$.

Ensuring **(a)** is straight-forward, but in order to ensure **(b)** we use an optimal external-memory priority queue supporting *Insert* and *Delete-Min* [3] that keeps track of the visited vertices connected to the v'_j's. The modifications are detailed in **Modified-MR-BFS**. It performs at most one *Insert* and one *Delete-Min* for each edge of the form (v'_j, v), and thus causing $\mathcal{O}(sort(V))$ extra I/Os [3]. An additional $\mathcal{O}(scan(E))$ I/O overhead results from scanning the adjacency lists.

Algorithm Modified-MR-BFS$(G'_i(u), u)$

Function: The input graph $G'_i(u)$ is given as an undirected graph but with implicit directed edges as discussed in section 3.5. This algorithm is a version of Munagala & Ranade's BFS algorithm modified to perform BFS on this implicitly partially directed graph from the source vertex u.

Steps:

Step 1: Perform the following initializations:

- **(a)** Set $L(0) \leftarrow \{u\}$
- **(b)** Set $Q \leftarrow \emptyset$, where Q is an optimal external-memory priority queue supporting *Insert* and *Delete-Min*

Step 2: For $i \leftarrow 1$ to $V - 1$ do:

- **(a)** Scan the adjacency lists of vertices in $L(i-1)$, and for each edge (v, v'_{j+1}) with $j \geq i$, set $Q \leftarrow Q \cup \{(v, j)\}$ (*Insert*)
- **(b)** Set $P \leftarrow \{v \mid (v, i) \in Q\}$ (*Delete-Min*)
- **(c)** Construct $N(L(i-1))$
- **(d)** Remove duplicates and all v'_j's from $N(L(i-1))$
- **(e)** Set $L(i) \leftarrow \{N(L(i-1)) \setminus \{L(i-1) \cup L(i-2) \cup P\}\} \cup \{v'_{i+1}\}$

Correctness of Modified-MR-BFS. Since the correctness of **MR-BFS** has already been proved in section 2.1, in order to prove the correctness of **Modified-MR-BFS** we only need to show that it correctly handles the partial directedness implicit in its input graph. In show this we need to prove that if the BFS level of a vertex v is less than i, then no edge of the form (v'_{i+1}, v) can force v to be included in $L(i)$. In iteration i, we insert each vertex $v \in L(i-1)$ into Q with a key $j \geq i$ provided the edge (v'_{j+1}, v) exists. Then we extract from Q each vertex v that has an incoming edge from v'_{i+1}. These vertices must have BFS level less than i and are excluded from $L(i)$. Vertices can be extracted from Q using *Delete-Min* operations because initially (before iteration 1) Q is empty, and iteration i never inserts into Q any vertex with a key value less than i, and these insertions always precede the *Delete-Min* operations in the same iteration.

3.5 External-Memory Approximate AP-BFS

As pointed out in section 3.2, there are two ways to apply the BFS in step 3(a_1) of **DHZ-Approx-AP-BFS$_k$**: either we can run BFS independently from each vertex in D_i as in **DHZ-Approx-AP-BFS$_k$**, or we can run BFS incrementally from the vertices of D_i using the strategy used in **AP-BFS** (see section 2.2).

We present the algorithm **Independent-BFS** which when called with D_i as a parameter constructs the partially directed unweighted graph $G_i'(u)$ for each $u \in D_i$ and runs Mehlhorn & Meyer's sublinear I/O BFS algorithm [13] on $G_i'(u)$ from u. The I/O-complexity of Mehlhorn & Meyer's algorithm is $\mathcal{O}(\sqrt{\frac{VE}{B}}+\frac{E}{B}\log V)$ as opposed to the $\mathcal{O}(V+sort(E))$ I/O-complexity of Munagala & Ranade's algorithm (**MR-BFS** in section 2.1), and thus it performs better on sparse graphs. Mehlhorn & Meyer's algorithm is based on **MR-BFS**, and can be modified in exactly the same way to handle the partial directedness in $G_i'(u)$. The I/O-complexity of **Independent-BFS** is thus $\mathcal{O}(D_i(\sqrt{\frac{VE_i}{B}} + \frac{E_i}{B}\log V))$.

The algorithm **Interdependent-BFS** when called with parameter D_i, constructs $G_i'(u)$ for each $u \in D_i$, and then runs **Modified-MR-BFS** (section 3.4) incrementally on $G_i'(u)$ from each u using the technique used in **AP-BFS** (section 2.2). The main differences between **Interdependent-BFS** and **AP-BFS** are: **Interdependent-BFS** uses a different range for locating the adjacency lists, works on a slightly different graph in each iteration, each graph it works on is partially directed, and runs BFS only from the vertices in D_i. The I/O-complexity of **Interdependent-BFS** is $\mathcal{O}(\frac{E_i}{B}(V+iD_i)+D_i sort(E_i))$.

We observe that running **Independent-BFS** in step 3(a) of **DHZ-Approx-AP-BFS$_k$** is more I/O-efficient when $|D_i|$ is smaller and $G_i'(u)$ is denser (i.e., value of i is smaller), and **Interdependent-BFS** is more I/O-efficient when $|D_i|$ is larger and $G_i'(u)$ is sparser (i.e., value of i is larger). If we use **Independent-BFS** for all values of i, it will cause a total of $\mathcal{O}(\frac{V^2}{\sqrt{B}} + \frac{k}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ I/Os, and running **Interdependent-BFS** for all values of i requires a total of $\mathcal{O}(\frac{VE}{B}+\frac{k}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ I/Os. Therefore, we can do better if we take a hybrid approach: starting from $i=1$ we run **Independent-BFS** up to some value l of i, and then we switch to **Interdependent-BFS**. We call this parameter l a *switching parameter*, and choose its value in order to minimizes the I/O-complexity of the entire algorithm. The overall algorithm is given in **Approx-AP-BFS$_k$**

Algorithm Independent-BFS(V, E, D_i, E_i, E^*, L)

<u>*Function:*</u> Perform BFS independently from each vertex $u \in D_i$ on a graph constructed from V, E_i, E^* and the information in the list L of current best upper bounds on all-pairs shortest distances in the original graph (V, E). It updates L with the computed distances. Invoked by **Approx-AP-BFS**. See **Approx-AP-BFS** for the definition of the parameters.

<u>*Steps:*</u>

Step 1: Set $L' \leftarrow \emptyset$

Step 2: Sort the vertices in D_i by vertex indices.

Step 3: For each $u \in D_i$ do:

- **(a)** Set $V' \leftarrow V$, and $E' \leftarrow E_i \cup E^*$
- **(b)** Retrieve from L the current best upper bound $\widehat{\delta}(u,v)$ on the shortest distances from u to each $v \in V$. Collect only the finite bounds.
- **(c)** Add $|V|-2$ new vertices $v_2', v_3', \ldots, v_{|V|-1}'$ to V'.
- **(d)** Add the following undirected edges to E':
  - (u, v_2')
  - (u, v) for each $v \in V$ with $\widehat{\delta}(u,v) = 1$
  - (v_t', v_{t+1}') for $2 \le t < |V|-1$
  - (v_t', v) for each $v \in V$ with $\widehat{\delta}(u,v) = t$
- **(e)** Sort the edges in E' to convert it into adjacency list format.
- **(f)** Run the sublinear I/O BFS algorithm in [13] on the graph (V', E'), and append the computed shortest distances to L'. The algorithm must be modified to handle the implicit partial directedness in (V', E').

Step 4: Update the entries in L by sorting L' appropriately and scanning the two lists in parallel.

Algorithm Interdependent-BFS$(V, E, D_i, E_i, E^*, \langle v_1, v_2, \dots, v_{|V|}\rangle, L)$

<u>*Function:*</u> Perform BFS from each vertex $u \in D_i$ on a graph constructed from V, E_i, E^* and the information in the list L of current best upper bounds on all-pairs shortest distances in the original graph (V, E). BFS is performed on the vertices of D_i in the order they appear in $\langle v_1, v_2, \dots, v_{|V|}\rangle$, and distance information obtained from the BFS run immediately preceding the current run is used to reduce I/O overhead. This algorithm updates L with the computed distances. Invoked by **Approx-AP-BFS**. See **Approx-AP-BFS** for the definition of the parameters.

<u>*Steps:*</u>

Step 1: Set $L' \leftarrow \emptyset$

Step 2: Arrange the vertices in D_i in the order they appear in $\langle v_1, v_2, \dots, v_{|V|}\rangle$ Let $\langle u_1, u_2, \dots, u_t\rangle$ be the sequence of vertices in D_i after the ordering.

Step 3:

- **(a)-(e)** Same as the steps **3(a)** to **3(e)** in **Independent-BFS**, but performed with u_1 instead of u. Let (V'_1, E'_1) be the graph constructed.
- **(f)** Run Munagala and Ranade's BFS algorithm (**Modified-MR-BFS**) with u_1 as the source vertex to compute the distances $d(u_1, w)$ for all $w \in V$. Append the computed distances to L'.

Step 4: For $j \leftarrow 2$ to t do:

- **(a)-(e)** Same as the steps **3(a)** to **3(e)** in **Independent-BFS**, but performed with u_j instead of u. Let (V'_j, E'_j) be the graph constructed.
- **(f)** Sort the adjacency lists of the vertices $v'_2, v'_3, \dots, v'_{|V|-1}$ so that for $2 \le p < |V| - 1$, adjacency list of v'_p is placed ahead of that of v'_{p+1}. Let A' be this sorted list of adjacency lists.
- **(g)** Sort the remaining adjacency lists so that adjacency list of a vertex x is placed before that of another vertex y provided $d(u_{j-1}, x) < d(u_{j-1}, y)$ or $d(u_{j-1}, x) = d(u_{j-1}, y) \wedge x < y$. Let $A(p)$, $0 \le i < |V|$, denote the portion of this sorted list that contains adjacency lists of vertices lying exactly at distance p from u_{j-1}.
- **(h)** To compute $d(u_j, w)$ for all $w \in V'$, run Munagala and Ranade's BFS algorithm (**Modified-MR-BFS**) with source vertex u_j. But step (2) of that algorithm is modified so that instead of finding the adjacency lists of the vertices in $L(q-1)$ by $|L(q-1)|$ independent accesses, they are found by scanning $L(q-1)$ and $A(p)$ in parallel for $\max\{0, q-1-d(u_{j-1}, u_j) - 2(i-1)\} \le p \le \min\{|V| - 1, q - 1 + d(u_{j-1}, u_j) + 2(i-1)\}$. If $v'_q \in L(q-1)$ load its adjacency list from A'. Append the computed distances to L'.

Step 4: Update the entries in L by sorting L' appropriately and scanning the two lists in parallel.

Algorithm Approx-AP-BFS$_k(G, l)$

<u>*Function:*</u> Given an undirected graph $G = (V[G], E[G])$ and a switching parameter l, this algorithm computes the shortest distance between every pair of vertices in G with additive error of at most $2(k-1)$.

<u>*Steps:*</u>

Step 1: Perform the following initializations:

- **(a)** For $i \leftarrow 1$ to $k-1$ do:

  Set $s_i \leftarrow \frac{E}{V}(\frac{V \log V}{E})^{\frac{i}{k}}$
- **(b)** Set $(\langle E_1, E_2, \dots, E_k, E^*\rangle, \langle D_1, D_2, \dots, D_k\rangle) \leftarrow$ **Decompose**$(G, \langle s_1, s_2, \dots, s_{k-1}\rangle)$
- **(c)** Sort the edges in $E[G]$ so that an edge (u_1, v_1) is placed ahead of another edge (u_2, v_2) provided $(u_1 < u_2) \vee ((u_1 = u_2) \wedge (v_1 < v_2))$. Scan $E[G]$ to produce a sorted (in the same order that is used for sorting $E[G]$) list L of approximate distances $\widehat{\delta}(u, v)$, where $u, v \in V[G]$, and $\widehat{\delta}(u, v) \leftarrow 1$ provided $(u, v) \in E[G]$, $\widehat{\delta}(u, v) \leftarrow \infty$ otherwise.

Step 2:

- **(a)** For $i \leftarrow 1$ to l do:

  Independent-BFS(V, E, D_i, E_i, E^*, L)
- **(b)** Find a spanning tree T of G, and construct an *Euler Tour ET* of T. Mark the first occurrence of each vertex on ET, and let let $v_1, v_2, \dots, v_{|V|}$ be the marked vertices in the order they appear on ET.
- **(c)** For $i \leftarrow l + 1$ to k do:

  Interdependent-BFS$(V, E, D_i, E_i, E^*, \langle v_1, v_2, \dots, v_{|V|}\rangle, L)$

Step 3: Return the output of step 2(c).

Correctness of Approx-AP-BFS$_k$. The correctness of **Approx-AP-BFS$_k$** follows from the following:

(a) correctness of **DHZ-Approx-AP-BFS$_k$**,

(b) lemma 3.1,

(c) correctness of **Modified-MR-BFS**,

(d) correctness of Mehlhorn & Meyer's BFS algorithm [13] modified to handle the type partial directedness in the input graph as described in section 3.4, and

(e) the guarantee that in step 4(h) of **Interdependent-BFS**, all adjacency lists are found within the range searched.

Proof of **(a)** can be found in [9]. Proofs of **(b)** and **(c)** are given in section 3.4. Proof of **(d)** follows from the proof of **(c)** since Mehlhorn & Meyer's algorithm builds on Munagala & Ranade's BFS algorithm [15] (**MR-BFS** in section 2.1), and the modifications required are exactly the same..

We only need to prove **(e)**. For $1 \leq i \leq k$, and $u, v \in V[G]$, let $\delta_i(u,v)$ be the value of $\widehat{\delta}(u,v)$ after running BFS from all vertices of D_i. It has been shown in [9] that if $u \in D_i$ and $v \in V[G]$, **DHZ-Approx-AP-BFS$_k$** maintains $\delta(u,v) \leq \delta_i(u,v) \leq \delta(u,v) + 2(i-1)$. The algorithm **Approx-AP-BFS$_k$** also clearly maintains this invariant up to level l. The first vertex in D_{l+1} also computes its distances with surplus error of at most $2l$. This is the base case. Now suppose all distances were calculated with surplus error of at most $2(i-1)$ up to the $(j-1)$-th vertex u_{j-1} of some D_i, where $1 < j \leq |D_i|$ and $l < i \leq k$. We will now prove that for $u_j \in D_i$ the adjacency lists will be found within the range in step 4(h) of **Interdependent-BFS**. Let v be any vertex in $V[G]$. We have the following three invariants from **DHZ-Approx-AP-BFS$_k$**: $\delta(u_{j-1}, u_j) \leq \delta_i(u_{j-1}, u_j) \leq \delta(u_{j-1}, u_j) + 2(i-1)$, $\delta(u_{j-1}, v) \leq \delta_i(u_{j-1}, v) \leq \delta(u_{j-1}, v) + 2(i-1)$, and $\delta(u_j, v) \leq \delta_i(u_j, v) \leq \delta(u_j, v) + 2(i-1)$. We also have the following two triangle inequalities: $\delta(u_{j-1}, v) \leq \delta(u_{j-1}, u_j) + \delta(u_j, v)$ and $\delta(u_j, v) \leq \delta(u_j, u_{j-1}) + \delta(u_{j-1}, v)$. From the five inequalities above, we get $\delta_i(u_{j-1}, v) - \delta_i(u_{j-1}, u_j) - 2(i-1) \leq \delta_i(u_j, v) \leq \delta_i(u_{j-1}, v) + \delta_i(u_{j-1}, u_j) + 2(i-1)$. Therefore the range used in step 4(h) of **Interdependent-BFS** is sufficient for finding the corresponding adjacency lists. Note that distances for the first vertex in any D_i, $l < i \leq k$, are computed correctly (with surplus error of at most $2(i-1)$) assuming that distances for all vertices in all D_j's with $1 \leq j \leq l$ are already computed correctly (with surplus error of at most $2(j-1)$).

I/O Complexity of Approx-AP-BFS$_k$. We note that $D_i = \mathcal{O}(\frac{V \log V}{s_i})$ for $1 \leq i < k$, $D_k = V$, $E_1 = E$, and $E_i \leq V s_{i-1}$ for $2 \leq i \leq k$. We also have $s_i = \frac{E}{V}\alpha^i$ for $1 \leq i < k$, where $\alpha = (\frac{V \log V}{E})^{\frac{1}{k}}$.

I/O cost of step 1 is dominated by the I/O cost of **Decompose**, and so this step requires $\mathcal{O}(k(V + \frac{V^2}{B}) + sort(E))$ I/Os.

For $i = 1$ to l, iteration i of step 2(a) calls **Independent-BFS** with D_i as a parameter which in turns runs the $\mathcal{O}(\sqrt{\frac{VE_i}{B}} + \frac{E_i}{B}\log V)$ I/O BSF algorithm by Mehlhorn & Meyer [13] (modified as outlined in **Modified-MR-BFS** to handle the partial directedness implicit in the input graph) from each vertex $u \in D_i$ on the graph $G'_i(u)$. Thus this step requires $\mathcal{O}(\sum_{i=1}^{l} D_i(\sqrt{\frac{VE_i}{B}} + \frac{E_i}{B}\log V)) = \mathcal{O}(V^2\sqrt{\frac{V}{BE\alpha^{l+1}}}\log V + \frac{l}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ I/Os.

Step 2(b) requires $\mathcal{O}(sort(E) \cdot \log_2 \log_2 \frac{VB}{E})$ I/Os [5].

For $i = l+1$ to k, iteration i of step 2(c) calls **Interdependent-BFS** with D_i as a parameter requiring $\mathcal{O}(\frac{E_i}{B}(V + iD_i) + D_i sort(E_i))$ I/Os. Thus the I/O-complexity of step 2(c) is $\mathcal{O}(\sum_{i=l+1}^{k}\{\frac{E_i}{B}(V + iD_i) + D_i \cdot sort(E_i)\}) = \mathcal{O}(\frac{VE\alpha^{l-1}}{B} + \frac{k-l}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$.

Therefore the total I/O cost of **Approx-AP-BFS$_k$** is $\mathcal{O}(V^2\sqrt{\frac{V}{BE\alpha^{l+1}}}\log V + \frac{VE\alpha^{l-1}}{B} +$

$\frac{k}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}}V)$. We determine the value of l by equating the first two terms of this expression. We get $l = \frac{\log(V^3 B\log^2 V)-\log(E^3\alpha)}{3\log\alpha}+1$, and the I/O complexity reduces to $\mathcal{O}(\frac{1}{B^{\frac{2}{3}}}V^{2-\frac{2}{3k}}E^{\frac{2}{3k}}\log^{\frac{2}{3}(1-\frac{1}{k})}V + \frac{k}{B}V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}}V)$.

3.6 An Alternate Algorithm for $k = 2$

For $k = 2$, **Approx-AP-BFS**$_k$ causes $\mathcal{O}(\frac{1}{B^{\frac{2}{3}}}V^{\frac{5}{3}}E^{\frac{1}{3}}\log^{\frac{1}{3}}V + \frac{2}{B}V^{\frac{3}{2}}E^{\frac{1}{2}}\log^{\frac{1}{2}}V)$ I/Os and produces estimated distances with an additive error of at most 2. We can, however, externalize the $\mathcal{O}(V^2 s + \frac{V^3\log V}{s})$-time internal-memory approximation algorithm (**Approx-APSP**) by Aingworth et al. [2] (where s is a degree threshold whose value can be chosen to optimize performance) to compute all pairwise distances in an unweighted undirected graph with an additive one-sided error of at most 2 incurring $\mathcal{O}(\frac{1}{B^{\frac{3}{4}}}V^{\frac{7}{4}}E^{\frac{1}{4}}\log V + \frac{1}{B}V^{\frac{3}{2}}E^{\frac{1}{2}}\log^{\frac{3}{2}}V + \frac{1}{B}V^{\frac{5}{2}}\log V)$ I/Os. The resulting algorithm (described below as **Alternate-Approx-AP-BFS**$_2$) outperforms **Approx-AP-BFS**$_2$ whenever $B > \frac{V^{\frac{5}{2}}\log^2 V}{E}$ assuming $V \geq \log^4 V$ and $E \leq \frac{V^2}{\log V}$.

Given a graph $G = (V, E)$ and the value of s, **Alternate-Approx-AP-BFS**$_2$ computes a small set D of vertices dominating all vertices of G having degree at least s, and finds the graph G_L induced by the vertices of G having degree less than s. It performs a single **AP-BFS** on G_L, but performs one BFS on G from each vertex in D. Then it combines the computed distances appropriately to produce surplus a 2 distance between every pair of vertices in G.

The **Alternate-Approx-AP-BFS**$_2$ algorithm is a fairly straight-forward external-memory implementation of Aingworth et al.'s internal-memory algorithm (**Approx-APSP**) [2]. However, the main difference between our implementation and **Approx-APSP** is that we run the **AP-BFS** algorithm from section 2 on G_L (see steps 1(c) of **Alternate-Approx-AP-BFS**$_2$ for the definition of G_L) instead of running an independent BFS from each vertex of G_L, and also we choose a value of s different from that chosen by **Approx-APSP**. We also use a different algorithm for finding dominating sets; the **Dominate** function (see section 3.3) we use is based on Dor et al.'s internal-memory implementation [9] of the greedy algorithm for computing dominating sets, which is more efficient than the internal-memory implementation of the same algorithm by Aingworth et al. [2].

Algorithm Alternate-Approx-AP-BFS$_2(G, s)$

Function: Given an undirected graph $G = (V[G], E[G])$, and degree threshold s, this algorithm computes the shortest distance between every pair of vertices in G with additive error at most 2.

Steps:

Step 1: Perform the following initializations:

- **(a)** Let $\widehat{d}$ be a two-dimensional array of dimension $V \times V$ which is laid out in a row-major order. For $1 \leq u, v \leq V$, $\widehat{d}[u, v]$ contains the current best known upper bound on the shortest distance from u to v.
  For $1 \leq u, v \leq V$, set $\widehat{d}[u, v] \leftarrow +\infty$
- **(b)** Set $V_L \leftarrow \{v \in V | deg(v) < s\}$
- **(c)** Set $G_L \leftarrow$ subgraph of G induced by V_L

Step 2: Run **Dominate**(G, s) to compute a set of vertices D dominating all vertices of G of degree at least s

Step 3:

- **(a)** Set $d_1 \leftarrow \emptyset$ $\{d_1$ *is a list to store distance values*$\}$
- **(b)** For each $w \in D$ do:
  Run the sublinear I/O BFS algorithm in [13] on G from w, and append the calculated distances to d_1.
- **(c)** Sort the distances in d_1 so that the distance between vertices u_1 and v_1 is placed before that of u_2 and v_2 provided $(u_1 < u_2) \vee ((u_1 = u_2) \wedge (v_1 < v_2))$.

(d) Update $\widehat{d}$ by scanning $\widehat{d}$ and d_1 simultaneously.

Step 4:

(a) Set $d_2 \leftarrow \emptyset$ $\{d_2$ *is a list to store distance values*$\}$
(b) Run the **AP-BFS** algorithm in section 2 on G_L and store the calculated distances in d_2.
(c) Sort the distances in d_2 so that the distance between vertices u_1 and v_1 is placed before that of u_2 and v_2 provided $(u_1 < u_2) \vee ((u_1 = u_2) \wedge (v_1 < v_2))$.
(d) Update $\widehat{d}$ by scanning $\widehat{d}$ and d_2 simultaneously.

Step 5: For each $w \in D$ do:

(a) Set $d_3 \leftarrow \emptyset$ $\{d_3$ *is a list to store distance values*$\}$
(b) Scan $\widehat{d}$ to generate $\widehat{d}(w,u) + \widehat{d}(w,v)$ $(= \widehat{d}(u,w) + \widehat{d}(w,v) =$ distance from u to v through $w)$ for all $u, v \in V$, and append these distance values to d_3.
(c) Update $\widehat{d}$ by scanning $\widehat{d}$ and d_3 simultaneously.

Correctness. Since **Alternate-Approx-AP-BFS$_2$** is a straight-forward implementation of **Approx-APSP** in [2], its correctness follows from the correctness of **Approx-APSP** [2], and from the correctness of **Dominate** in section 3.3, **AP-BFS** in section 2.2 and the BFS algorithm in [13].

I/O Complexity. The initializations in step 1 requires only a constant number of scanning and sorting steps incurring $\mathcal{O}(\frac{V^2}{B} + sort(E))$ I/Os. I/O complexity of step 2 is also $\mathcal{O}(\frac{V^2}{B} + sort(E))$. Since $|D| = \mathcal{O}(\frac{V \log V}{s})$, step 3 requires $\mathcal{O}(\frac{V \log V}{s}(\sqrt{\frac{VE}{B}} + \frac{E}{B}\log V))$ I/Os. Since the number of edges in G_L is $\mathcal{O}(Vs)$, the I/O complexity of step 4 is $\mathcal{O}(\frac{V^2 s}{B}\log V)$. Step 5 requires $\mathcal{O}(\frac{V \log V}{s}\frac{V^2}{B})$ I/Os. Thus the total I/O complexity is $\mathcal{O}(\frac{V \log V}{s}(\sqrt{\frac{VE}{B}} + \frac{E}{B}\log V + \frac{V^2}{B}) + \frac{V^2 s}{B}\log V)$. This expression is minimized for $s = (\sqrt{\frac{EB}{V}} + \frac{E}{V}\log V + V)^{\frac{1}{2}}$, and the I/O complexity reduces to $\mathcal{O}(\frac{1}{B^{\frac{3}{4}}}V^{\frac{7}{4}}E^{\frac{1}{4}}\log V + \frac{1}{B}V^{\frac{3}{2}}E^{\frac{1}{2}}\log^{\frac{3}{2}} V + \frac{1}{B}V^{\frac{5}{2}}\log V)$.

4 Cache-Aware APSP for Weighted Undirected Graphs

In this section we present a cache-aware algorithm for computing all-pairs shortest paths in a weighted undirected graph. In [6], Arge et al. introduce the *Multi-Tournament-Tree* data structure to obtain an $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}}\log V + sort(E)))$ I/O cache-aware algorithm for computing APSP on general weighted undirected graphs whenever $E \leq \frac{VB}{\log V}$. In this section we introduce the *Multi-Buffer-Heap* data structure, and use it to obtain an $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + sort(E)))$ I/O cache-aware algorithm for solving the same problem assuming $E \leq \frac{VB}{(\log V)^2}$. This leads to an $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + \frac{E}{B}\log\frac{V}{B}))$ I/O algorithm for any edge density. Our algorithm uses $O(V^2)$ space.

4.1 Slim Data Structures

We introduce here the notion of a *slim data structure* which is an external-memory data structure in which a fixed-sized portion is kept in internal memory. The area in the internal memory that holds that specific portion is called the *slim cache*. By $DS(\lambda)$ we denote an external-memory data structure DS, in which a portion of size λ is kept in the slim cache. We continue to assume the behavior of the two-level I/O model, namely **(a)** the size of the internal memory is M and **(b)** data is transferred between the two levels of memory in blocks of size B. Thus $1 \leq \lambda \leq M$; and the data structure operations must assume that the portion of the data structure that is not stored in the slim cache is stored in an external memory divided into blocks of size B, and thus accessing anything outside the slim cache causes I/Os. While executing a data structural operation the operation can use all free internal memory for temporary computation, but after the operation completes only the data in the slim cache is preserved for reuse by the next operation on the data structure.

Some existing external-memory data structures can be viewed trivially as slim data structures. For example, Arge et al. [6] analyzed each component Tournament Tree of the *Multi-Tournament-Tree* as supporting *Decrease-Key*, *Delete* and *Delete-Min* operations in $\mathcal{O}(\frac{1}{\lambda}\log N)$ amortized I/Os each for $1 \leq \lambda \leq \frac{B}{2}$; this can be viewed as a slim data structure for this range of values for λ. The cache-oblivious and cache-aware Buffer Heaps without the tall cache assumption, described briefly in Chowdhury & Ramachandran [8], can be analyzed as slim data structures that support the same three operations with the following bounds: for $B \leq \lambda \leq M$, the cost of each operation is $\mathcal{O}(\frac{1}{B}\log\frac{N}{\lambda})$ amortized I/Os, and for $1 \leq \lambda \leq B$ the cost of each operation is $\mathcal{O}(\frac{1}{\lambda}\log\frac{N}{\lambda})$ amortized I/Os.

In the next section we present a slim data structure based on the Buffer Heap, which we call a *Slim Buffer Heap*, $SBH(\lambda)$, which supports *Decrease-Key*, *Delete* and *Delete-Min* with the amortized cost of $\mathcal{O}(\frac{1}{\lambda}+\frac{1}{B}\log\frac{N}{\lambda})$ I/Os each. This improves on the results mentioned above for the component Tournament Trees of the Multi-Tournament-Tree and for the Buffer Heap when $B < \lambda$. In section 4.3 we use a collection of Slim Buffer Heaps in a *Multi-Buffer-Heap* to obtain an improved cache-aware APSP algorithm for undirected graphs with general non-negative edge-weights.

Although our main motivation behind introducing the notion of slim data structures was to obtain the result in section 4.3, we believe that the need for slim data structures could arise in other applications. A typical application would be one in which a number of data structures need to be kept in internal memory simultaneously, and thus only a limited portion of the internal memory can be dedicated to each data structure.

4.2 The Slim Buffer Heap

In this section we extend the cache-oblivious priority queue known as the Buffer Heap [8] (developed recently by the authors) to a slim data structure with an arbitrary parameter λ. We call this data structure a *Slim Buffer Heap* (*SBH*), and for an SBH with parameter λ ($1 \leq \lambda \leq M$), denoted by $SBH(\lambda)$, it is assumed that an initial segment of $\Theta(\lambda)$ elements in the data structure resides in internal memory and no I/O is required to access the elements in this segment. A Buffer Heap supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\frac{1}{B}\log\frac{N}{B})$ I/Os each. We show in this section that an $SBH(\lambda)$ supports each of these three operations in $\mathcal{O}(\frac{1}{\lambda}+\frac{1}{B}\log\frac{N}{\lambda})$ amortized I/Os, where N is the number of elements. A *Delete*(x) operation deletes element x from the queue if it exists and a *Delete-Min*() operation retrieves and deletes the element with 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 $k_x < k'_x$.

4.2.1 Structure

The structure is the same as that of a 'Buffer Heap without a tall cache' which was described briefly in [8]. It consists of $r = 1 + \lceil \log_2 N \rceil$ levels. For $0 \leq i \leq r-1$, level i consists of an *element buffer* B_i and an *update buffer* U_i. Each element in B_i is of the form (x, k_x), where x is the element id and k_x is its key. Each update in U_i is augmented with a time stamp indicating the time of its insertion into the structure.

At any time, the following invariants are maintained:

Invariant 4.1.

(a) *Each B_i contains at most 2^i elements.*

(b) *Each U_i contains at most 2^i updates.*

Invariant 4.2.

(a) *For $0 \leq i < r-1$, key of every element in B_i is no larger than the key of any element in B_{i+1}.*

(b) *For $0 \leq i < r-1$, for each element x in B_i, all updates applicable to x that are not yet applied, reside in $U_0, U_1, \ldots, U_i$.*

Invariant 4.3.

(a) *Elements in each B_i are kept sorted in ascending order by element id.*

(b) *Updates in each U_i are divided into (a constant number of) segments with updates in each segment sorted in ascending order by element id and time stamp.*

All buffers are initially empty.

4.2.2 Layout

As in [8] we use a stack S_B to store the element buffers, and another stack S_U to store the update buffers. An array A_s of size r to stores information on the buffers. For $0 \leq i \leq r-1$, $A_s[i]$ contains the number of elements in B_i, and the number of segments in U_i along with the number of updates in each segment. We assume the existence of a slim cache of size $\Theta(\lambda)$, large enough to store $B_0, B_1, \ldots, B_t$, $U_0, U_1, \ldots, U_{t+1}$, and the first λ entries of A_s, where $t = \log(\lambda+1) - 1$. The remaining portions of S_B, S_U and A_s are kept in external memory.

4.2.3 Operations

In this section we describe how *Delete*, *Delete-Min* and *Decrease-Key* operations are implemented. 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 except that the **Fix-U** function may be called to restore invariant 4.1(b) for the structure. If needed, the *Delete*/*Decrease-Key* operation collects enough elements from higher level element buffers to fill the slim cache.

The **Fix-U** function uses a function called **Apply-Updates**. When called with a parameter i, **Apply-Updates** applies the updates in U_i on the elements of B_i, and empties U_i by moving the updates from U_i to U_{i+1}. It also moves any overflowing elements from B_i to U_{i+1} as *Sink* operations. A *Sink*(x, k_x) operation is used to move an element (x, k_x) from B_i to B_{i+1} through U_{i+1}.

The *Delete-Min* function works by finding the shallowest element buffer B_i that is left non-empty after applying the updates in U_i (by calling **Apply-Updates**(i)). The **Fix-U** function is then called to fix overflowing update buffers if any, and to collect enough elements to fill the slim cache. The minimum element is then extracted from all elements collected so far and the remaining elements are distributed to the shallowest element buffers.

After each operation the *Reconstruct* function is called. This function reconstructs the entire data structure periodically. It remembers the number of elements N_e in the structure immediately after the last reconstruction, and keeps track of the number of new operations N_o performed since then. Initially N_e is set to 0. When $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$, the data structure is rebuilt by calling **Apply-Updates** for each level, emptying the update buffers and distributing the remaining elements to the shallowest possible levels. The objective of the function is to ensure that the number of levels r in the structure is always within ± 1 of $\log_2 N$, where N is the current number of elements in the structure. This invariant is maintained because r can decrease by at most 1 since the last reconstruction (this happens if all $\lfloor \frac{N_e}{2} \rfloor + 1$ operations are *Delete* or *Delete-Min* operations), and can increase by at most 1 (if all those operations are *Decrease-Keys*).

Function Decrease-Key(x, k_x)/Delete(x)

<u>*Function:*</u> Inserts a *Decrease-Key/ Delete* operation into the structure.

<u>*Steps:*</u>

Step 1: Insert the operation into U_0 augmented with the current time stamp

Step 2:
- **(a)** Set $B' \leftarrow \emptyset$, $i \leftarrow 0$ $\{B'$ *is a temporary list to collect elements returned by* **Fix-U**$\}$
- **(b)** **Fix-U**(i, B')

Step 3: Move the contents of B' to the shallowest possible element buffers maintaining invariants 4.1(a), 4.2(a) and 4.3(a)

Step 4: **Reconstruct**()

Function Fix-U(i, B')

<u>*Function:*</u> Fixes all overflowing update buffers in levels i and up. An update buffer U_i overflows if $|U^i| > 2^i$. For each overflowing U_i collects the contents of B_i in B' after applying U_i on B_i.

<u>*Steps:*</u>

Step 1:
While $i < r$ AND ($|U_i| > 2^i$ OR ($i = t + 1$ AND $|B'| = 0$) OR ($i > t + 1$ AND $|B'| < \lambda$)) do:
- **(a)** **Apply-Updates**(i)
- **(b)** Append the elements of B_i to B'
- **(c)** Set $i \leftarrow i + 1$

Step 2: If $i < r$ then merge the segments of U_i

Function Apply-Updates(i)

<u>*Function:*</u> Apply the updates in U_i on the elements in B_i, move remaining updates from U_i to U_{i+1} if $i < r - 1$, and after applying the updates move overflowing elements from B_i to U_{i+1} as *Sink* operations.

<u>*Steps:*</u>

Step 1: If $|B_i| = 0$ and $i < r - 1$ then:
- **(a)** Merge the segments of U_i
- **(b)** Move the contents of U_i as a new segment of U_{i+1}
- **(c)** Set $U_i \leftarrow \emptyset$

Step 2: Else ($|B_i| > 0$ or $i = r - 1$) do:
- **(a)** Merge the segments of U_i
- **(b)** If $i = r - 1$ then set $k \leftarrow +\infty$ else set $k \leftarrow$ largest key of elements in B_i
- **(c)** Scan B_i and U_i simultaneously, and for each operation in U_i if the operation is:

  Delete(x) then remove any element (x, k_x) from B_i if exists

  Decrease-Key(x, k_x) **/** ***Sink***(x, k_x) then if any element (x, k'_x) exists in B_i replace it with $(x, \min(k_x, k'_x))$, otherwise copy (x, k_x) to B_i if $k_x \leq k$
- **(d)** If $i < r - 1$ then do the following:
  - copy each *Decrease-Key*(x, k_x) / *Sink*(x, k_x) in U_i with $k_x > k$ to U_{i+1}
  - for each *Delete*(x) and *Decrease-Key*(x, k_x) with $k_x \leq k$ in U_i copy a *Delete*(x) to U_{i+1}
- **(e)** If $|B_i| > 2^{i+1}$ then do:
  - if $i = r - 1$ then set $r \leftarrow r + 1$
  - keep the 2^{i+1} elements with the smallest 2^{i+1} keys in B_i and insert each remaining element (x, k_x) into U_{i+1} as *Sink*(x, k_x)
- **(f)** Set $U_i \leftarrow \emptyset$

Function Delete-Min()

Function: Deletes and returns the element with the smallest key in the structure.

Steps:

Step 1:
Set $i \leftarrow -1$
While $i < r-1$ do:

(a) Set $i \leftarrow i+1$
(b) **Apply-Updates**(i)
(c) If B_i is non-empty then exit loop

Step 2:
(a) Set $B' \leftarrow B_i$, $i \leftarrow i+1$
(b) **Fix-U**(i, B')

Step 3:
(a) Extract the element with minimum key from B' to return
(b) Move remaining elements from B' to the shallowest possible element buffers maintaining invariants 4.1(a), 4.2(a) and 4.3(a)

Step 4: **Reconstruct**()

Function Reconstruct() (Reconstruct the data structure periodically.)

Function: Reconstructs the data structure when $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$, where N_e is the number of elements in SBH immediately after the last reconstruction ($N_e = 0$ initially), and N_o is the number of operations since the last reconstruction or the initialization of SBH.

Steps:

Step 1:
If $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$ then:

(a) For $i \leftarrow 0$ to $r-1$ do:

Apply-Updates(i)

(b) Distribute the elements remaining in SBH to the shallowest possible element buffers

---

Correctness. We need to prove that each of the three operations, namely *Decrease-Key*, *Delete* and *Delete-Min*, maintains all invariants.

First we observe that **Apply-Updates**(i) can cause U_{i+1} to overflow in step 2(e), and thus violate invariant 4.1(b). It does not violate any other invariants.

The function **Fix-U** when called with parameter i fixes 4.1(b) for all overflowing U_j with $j \geq i$. It calls **Apply-Updates** in each iteration. When **Apply-Updates**(j) is called for some $j \geq i$, it may violate invariant 4.1(b) for U_{j+1}. But in that case, **Fix-U** calls **Apply-Updates**($j+1$) which leaves U_j empty, and thus trivially fixes the invariant for U_j. The **Fix-U** function keeps applying **Apply-Updates** as long as there is an overflowing update buffer in the structure. Since **Fix-U** does not violate any other invariants, at termination all invariants hold.

The **Reconstruct** function periodically rebuilds the data structure, and trivially maintains all invariants.

A **Decrease-Key/Delete** function may cause U_0 to overflow in step 1 which is fixed by **Fix-U** in step 2. In step 3, the elements collected by **Fix-U** are distributed to the shallowest possible element buffers maintaining all invariants. The **Reconstruct** function in step 4 does not violate any invariant. Thus at termination of a **Decrease-Key/Delete** function all invariants continue to hold.

Since the **Delete-Min** function calls **Apply-Updates** in step 1, at the end of that step invariant 4.1(b) may no longer hold for U_{i+1}. But the call to **Fix-U** in step 2, restores all invariants. In step 3 the element with minimum key is extracted from the elements collected in steps 1 and 2, and the remaining elements are distributed to shallowest possible element buffers maintaining all invariants. The call to the **Reconstruct** function in step 4 does not violate any invariant. Thus **Deelete-Min** maintains all invariants.

4.2.4 I/O Complexity

We begin with a preliminary lemma whose proof of correctness is an extension to that of lemma 2 in [8], and is included in the Appendix.

LEMMA 4.1. *For $1 \leq i \leq r-1$, every empty U_i receives batches of updates a constant number of times before U_i is applied on B_i and 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_i (in **Apply-Updates**) incur only $\mathcal{O}(\frac{1}{B})$ amortized cache-misses per update in U_i.

The following lemma gives the I/O complexities of the operations supported by a Slim Buffer Heap:

LEMMA 4.2. *A Slim Buffer Heap supports* Delete*,* Delete-Min *and* Decrease-Key *operations in $\mathcal{O}(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda})$ amortized I/Os each using $O(N)$ space, where N is the current number of elements in the structure.*

Proof. As in [8], we assume that a *Decrease-Key* operation is inserted into U_0 as an ordered pair $\langle$*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. Thus a *Decrease-Key* operation will be counted as two operations until it is applied on some element buffer.

For $0 \leq i \leq r-1$, let u_i be the number of operations in U_i and b_i the number in B_i. Let Δ denote the number of new *Decrease-Key*, *Delete* and *Delete-Min* operations since the last time any part of the data structure outside the slim cache was accessed, and let Δ_o be the number of operations since the last construction/reconstruction of the data structure. If H is the current state of $SBH(\lambda)$, we define the *potential* of H as follows:

$$\Phi(H) = \frac{2}{B}\sum_{i=0}^{r-1}\{(2r-i)\cdot u_i \;+\; (i+1)\cdot b_i\} \;+\; \frac{r}{B}\cdot\Delta_o \;+\; \frac{2}{\lambda}\cdot(\Delta+\Delta_o)$$

As in the analysis of the I/O-complexities of the Buffer Heap operations in [8], the key observation is that operations always move downward in the U buffer and elements generally move upward in the B buffer.

First let us consider the amortized cost of the *Reconstruct* function. At the time of reconstruction $\Delta_0 = \lfloor\frac{N_e}{2}\rfloor + 1$, where N_e is the number of elements in the structure immediately after the last reconstruction. Thus $\lceil\frac{N_e}{2}\rceil - 1 \leq \sum_0^{r-1} b_i \leq \lfloor\frac{3N_e}{2}\rfloor + 1$ implying $\Delta_0 = \Theta(\sum_0^{r-1} b_i)$. If during the reconstruction operation no buffer outside the slim cache is accessed then no I/O occurs. Therefore, we will only consider the case in which some element buffer above level t is accessed. In that case $\Delta_o = \Omega(\lambda)$. The actual cost of the reconstruction operation is $\mathcal{O}(1 + \frac{1}{B}\Delta_o r)$. Since all update buffers are emptied the potential drop is $\Omega(1 + \frac{1}{B}\Delta_o r)$. Thus the amortized cost of the **Reconstruct** function is $(1 + \frac{1}{B}\Delta_o r) - (1 + \frac{1}{B}\Delta_o r) = 0$.

Now let us calculate the amortized cost of a *Decrease-Key* or *Delete* operation. The increase in potential due to the insertion of a *Decrease-Key* into U_0 is $2 \cdot (\frac{4}{\lambda} + \frac{5}{B} \cdot r)$, and due to insertion of a *Delete* is $\frac{4}{\lambda} + \frac{5}{B} \cdot r$. If no element buffer of level higher than t is accessed in step 2 of the **Decrease-Key/Delete** function then no I/O occurs. So we only need to consider the case when a B_i with $i > t$ is accessed. We observe that

• B_{t+1} can be accessed only due to the overflow of U_t. The overflow of U_t implies that $\Omega(\lambda)$ new operations have been inserted into the structure since B_{t+1} was accessed last time. Thus $\Delta = \Omega(\lambda)$, and the potential drop caused by these Δ updates is sufficient to pay for the extra I/O needed to access B_{t+1}.

• For $i > t + 1$, let j be the largest value of i for which **Apply-Updates** was called in step 1 of **Fix-U**. If U_j was full before **Apply-Updates** was called then the drop of potential due to the movement of these $|U_j| \geq 2^j$ updates to U_{j+1} is enough to pay for the actual cost of examining all elements and updates. This is because during the downward movement of the updates each element buffer encountered is scanned only a constant number of times causing a total of $\mathcal{O}(\frac{2^j}{B})$ I/Os, and the contents of B' in step 3 of the **Decrease-Key/Delete** function can be redistributed to the shallowest possible element buffers in $\mathcal{O}(\frac{2^j}{B})$ I/Os using a linear I/O selection algorithm [16]. Let k be the largest integer such that $|B'| \geq 2^k - 1$. The first selection step finds $2^k - 1$ elements with $2^k - 1$ smallest keys in $\mathcal{O}(\frac{|B'|}{B})$ I/Os, and keeps them in B' leaving the remaining elements in B_k. The second selection step finds 2^{k-1} elements from the $2^k - 1$ elements in B' to leave in B_{k-1}, the third one finds 2^{k-2} elements from the remaining $2^{k-1} - 1$ elements in B' to leave in B_{k-2}, and so on. Thus the total I/O-complexity of all selection steps starting from the second one is $\mathcal{O}(\frac{|B'|}{B})$, too. But $|B'| = \mathcal{O}(2^j)$. Therefore, the total I/O-complexity of redistributing the elements in B' is $\mathcal{O}(\frac{|B'|}{B}) = \mathcal{O}(\frac{2^j}{B})$.

• If U_j was not full before **Apply-Updates** was called then the cost of examining all updates is compensated by the drop of potential due to the downward movement of the updates. All but at most λ elements examined move upwards. The cost of examining each element that move upwards is paid by the potential drop due to the upward movement of that element. If some elements are left behind in B_j then $\Omega(\lambda)$ elements must have moved upward by at least 1 level, and the potential drop due to the upward movement of those elements pays for the cost of examining the elements that are left behind.

Thus the amortized cost of a *Decrease-Key/Delete* operation is $\mathcal{O}(\frac{1}{\lambda} + \frac{1}{B} \log_2 N)$. But since accessing the first t levels does not cause any external I/O, the amortized cost is $\mathcal{O}(\frac{1}{\lambda} + \frac{1}{B}\{\log_2 N - t\}) = \mathcal{O}(\frac{1}{\lambda} + \frac{1}{B} \log_2 \frac{N}{\lambda})$.

A *Delete-Min* operation increases the potential by $\frac{2}{\lambda}$. In the case of a *Delete-Min* operation B_{t+1} is accessed for either of the following two reasons:

• B_t underflows, which means $\Omega(\lambda)$ *Delete*/*Delete-Min* operations must have occurred since last time B_{t+1} was accessed, and thus $\Delta = \Omega(\lambda)$, and the potential drop caused by these Δ operations will pay for the extra I/O operation to access B_{t+1}.

• U_t overflows, and this case is the same as the one that occurs during a *Decrease-Key*/*Delete* operation.

The analysis of the actual cost of examining elements and updates, and the source of compensating potential drops is similar to that for a *Decrease-Key*/*Delete* operation. Thus the amortized cost of a *Delete-Min* operation is $\mathcal{O}(\frac{1}{\lambda} + \frac{1}{B} \log_2 \frac{N}{\lambda})$. □

4.3 Multi-Buffer-Heap and External-Memory APSP

In this section we introduce a compound priority queue structure based on Slim Buffer Heap, called the *Multi-Buffer-Heap*, and use this structure for efficient computation of external-memory APSP on an undirected graph with general non-negative edge-weights.

A Multi-Buffer-Heap is constructed as follows. Let $\lambda < B$ and let $L = \frac{B}{\lambda}$. We pack the slim caches of $\Theta(L)$ $SBH(\lambda)$ into a single memory block. We call this block the *multi-slim-cache* and the resulting structure a *Multi-Buffer-Heap*. By the analysis in section 4.2.4 this structure supports *Delete*, *Delete-Min* and *Decrease-Key* operations on each of its component Slim Buffer Heaps in $\mathcal{O}(\frac{L}{B} + \frac{1}{B}\log_2 \frac{NL}{B})$ amortized I/Os each.

For computing APSP we take the approach described in [6]. It solves APSP by working on all V underlying SSSP problems simultaneously, and each individual SSSP problem is solved using Kumar & Schwabe's algorithm for weighted undirected graphs [12]. For $1 \leq i \leq V$, this approach requires a priority queue pair (Q_i, Q'_i), where the ith pair belong to the ith SSSP problem. These V priority queue pairs are implemented using $\Theta(\frac{V}{L})$ Multi-Buffer-Heaps. The algorithm proceeds in V rounds. In each round it loads the multi-slim-cache of each MBH, and for each MBH extracts a settled vertex with minimum distance from each of the $\Theta(L)$ priority queue pairs it stores. It sorts the extracted vertices by vertex indices. It then scans this sorted vertex list and the sorted sequences of adjacency lists in parallel to retrieve the adjacency lists of the settled vertices of this round. Another sorting phase moves all adjacency lists to be applied to the same MBH together. Then all necessary *Decrease-Key* operations are performed by cycling through the Multi-Buffer-Heaps once again. At the end of the algorithm the extracted vertices along with their computed distance values are sorted to produce the final distance matrix.

I/O Complexity. In each round $\mathcal{O}(\frac{V}{L})$ I/Os are required to load the multi-slim-caches of all Multi-Buffer-Heaps. Accessing all required adjacency lists over $\mathcal{O}(V)$ rounds requires $\mathcal{O}(V \cdot sort(E))$ I/Os. A total of $\mathcal{O}(VE \cdot (\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{V}{\lambda}))$ I/Os are required by all $\mathcal{O}(VE)$ priority queue operations performed by this algorithm. Sorting the final distance matrix requires $\mathcal{O}(V \cdot sort(V))$ I/Os. Thus the I/O complexity of this algorithm is $\mathcal{O}(V \cdot (\frac{V}{L} + \frac{E}{\lambda} + \frac{E}{B}\log_2 \frac{V}{\lambda} + sort(E)))$. Using $L = \sqrt{\frac{VB}{E}} \geq 1$, we obtain the following:

Theorem 4.1. *Using Multi-Buffer-Heaps, APSP on undirected graphs with non-negative real edge weights can be solved using $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + sort(E)))$ I/Os and $\mathcal{O}(V^2)$ space whenever $E \leq \frac{VB}{(\log V)^2}$.*

In conjunction with the I/O efficient APSP algorithm for sufficiently dense graphs implied by the SSSP results in [12, 8] we obtain the following corollary.

Corollary 4.1. *APSP on an undirected graph with non-negative real edge weights can be solved using $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + \frac{E}{B}\log\frac{V}{B}))$ I/Os and $\mathcal{O}(V^2)$ space. The number of I/Os is reduced to $\mathcal{O}(\frac{VE}{B}\log\frac{V}{B})$ when $E \geq \frac{VB}{(\log\frac{V}{B})^2}$.*

References

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

[2] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal of Computing*, 28:1167–1181, 1999.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[4] 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*, pages 268–276, Montreal, Quebec, Canada, May 2002.

[5] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP, and multi-way planar graph separation. In *Proceedings of the Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.

[6] L. Arge, U. Meyer, and L. Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, pages 146–157, Turku, Finland, July 2004.

[7] G.S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, pages 480–492, Humlebaek, Denmark, July 2004.

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

[9] D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM Journal of Computing*, 29(5):1740–1759, 2000.

[10] R. Fagerberg. Dagstuhl Wokshop on Cache-Oblivious and Cache-Aware Algorithms. Seminar talk, July 22, 2004.

[11] 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*, pages 285–297, 1999.

[12] 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*, pages 169–177, 1996.

[13] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the European Symposium on Algorithms, LNCS 2461*, pages 723–735, 2002.

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

[15] K. Munagala and A. Ranade. I/o-complexity of graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

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

[17] V. Ramachandran. Dagstuhl Wokshop on Cache-Oblivious and Cache-Aware Algorithms. Seminar talk, July 22, 2004.

[18] U. Zwick. Exact and approximate distances in graphs – a survey. updated version at http://www.cs.tau.ac.il/~zwick. In *Proceedings of the 9th European Symposium on Algorithms, LNCS 2161*, pages 33–48, 2001.

APPENDIX

A Proof of Lemma 4.1

LEMMA 4.1. *For $1 \leq i \leq r-1$, every empty U_i receives batches of updates a constant number of times before U_i is applied on B_i and emptied again.*

Proof. Update buffer U_i receives at most two batches of updates whenever the execution of a **Decrease-Key/Delete/Delete-Min** function reaches level $i-1$. If the execution continues and reaches level i then U_i is applied on B_i, and thus emptied. Therefore, we only need to compute the number of times U_i receives a batch of updates between two successive times the execution of any of these functions reaches level i. Thus it suffices to consider only executions that terminate at level $i-1$. We will also ignore the case in which the function leaves B_{i-1} empty since an empty B_{i-1} ensures that the next time a function reaches level $i-1$ it will definitely proceed to level i, and empty U_i.

First let us consider the execution of a **Decrease-Key/Delete** function that terminates at level $i-1$. We have three cases based on the value of i.

Case 1 ($i \leq t+1$): Update buffer U_{i-1} was full before it was applied on B_{i-1}, and so U_i receives $\Omega(\frac{2^i}{2})$ updates.

Case 2 ($t+2 \leq i \leq t+3$): The execution can reach a level $i-1 > t$ if U_t overflows. Thus $\Omega(\frac{2^i}{8})$ updates reach U_i.

Case 3 ($i \geq t+4$): If U_{i-1} was full before it was applied on B_{i-1} then U_i receives $\Omega(\frac{2^i}{2})$ updates. We will now consider a maximal sequence of executions each with a non-full U_{i-1} before application on B_{i-1}. Before the first such execution reaches level $i-1$, B_{i-1} may contain as many as 2^{i-1} elements. But the first execution (and any subsequent execution ending at level $i-1$) will leave at most $\lambda - 1$ elements in B_{i-1}, and contribute at most two batches of updates to U_i. We will now calculate the number of updates pushed into U_i by any subsequent execution ending at level $i-1$. An execution can reach level $i-1$ if the total number of elements left in $B_0, B_1, \ldots, B_{i-2}$ is less than λ. If the number of elements in B_{i-1} is less than λ before the updates, and greater than 0 after the updates, then the number of updates reaching U_i is at least $2^{i-1} - 2\lambda \geq 2^{i-1} - 2 \cdot 2^{t+1} \geq 2^{i-1} - 2 \cdot 2^{i-3} = \frac{2^i}{4}$.

Next we consider the execution of a **Delete-Min** function that terminates at level $i-1$. Here, too, we have three cases based on the value of i. In each of these cases we will assume that the first non-empty element buffer after the updates lies in a level smaller than $i-1$, since if B_{i-1} is the first non-empty element buffer it will be immediately emptied by the **Delete-Min** function ensuring that the next time any function reaches level $i-1$ will proceed to level i and empty U_i.

Case 1 ($i \leq t+2$): Similar to case 1 for **Decrease-Key/Delete** function.

Case 2 ($i = t+3$): If the first non-empty element buffer is in a level smaller than $t+1$ then U_i receives $\Omega(\frac{2^i}{4})$ updates. If B_{t+1} is the first non-empty element buffer and U_{t+1} was full before applying it on B_{t+1} then, too, $\Omega(\frac{2^i}{4})$ updates are pushed into U_i. We will now consider a maximal sequence of executions each with B_{t+1} as the first non-empty element buffer, and a non-full U_{t+1} before application on B_{t+1}. The first such execution (and any subsequent ones, too) will leave at most $\lambda - 1$ elements in B_{t+2}. Starting from the second execution we partition the sequence into pairs of successive executions. Since $B_0, B_1, \ldots, B_t$ are all empty, and $\lambda = 2^{t+1} - 1$, every such pair of executions will push $\Omega(\frac{2^i}{4})$ updates into U_i.

Case 3 ($i \geq t+4$): Similar to case 3 for **Decrease-Key/Delete** function.

Thus between two successive emptyings of U_i, every batch of updates (except at most a constant number of them) received by U_i fills up a constant fraction of its capacity. If U_i overflows it is immediately emptied by **Fix-U**. Therefore, for $1 \leq i \leq r-1$, every empty U_i receives batches of updates a constant number of times before it is emptied again. $\square$