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

Rezaul Alam Chowdhury Vijaya Ramachandran

Abstract

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

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 [17]. 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.

1.2 Cache-Aware Algorithms. The *two-level I/O model* (or *cache-aware model*) was introduced in [1]. 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: to read N contiguous data items from the disk one needs $scan(N) = \Theta(\frac{N}{B})$ I/Os and to sort N items, $sort(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os [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 $\mathcal{O}(\sqrt{VE/B} + sort(E))$ I/Os [13]. External SSSP on an undirected graph with general non-negative edge-weights is computed in $\mathcal{O}(V + \frac{E}{B} \log \frac{E}{M})$ I/Os using the cache-aware *Buffer Heap* in [8]. There are also some results known for external SSSP on undirected graphs with restricted edge-weights [14]. The I/O complexity of external AP-BFS (or APSP) is obtained by multiplying the I/O complexity of external BFS (or SSSP) by V.

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

*Dept of Comp Sci, University of Texas, Austin, TX 78712. Email: {shaikat,vlr}@cs.utexas.edu. This work was supported in part by NSF CCR-9988160.

Results	Unweighted APSP	Approximate unweighted APSP with additive error $2(k-1)$ (for integer $k \in [2, \log V]$)	Weighted APSP
Known	$\mathcal{O}(V \cdot sort(E))$ I/Os, $\mathcal{O}(\sqrt{VEB})$ extra space [6]	$\mathcal{O}(\frac{1}{B^{\frac{1}{2}}} V^2 \log^{\frac{1}{2}} V + \frac{k}{B} V^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-\frac{1}{k}} V \log\log \frac{VB}{E})$ (trivial using [10, 14])	$\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B} \log V} + sort(E)))$ for $E \le \frac{VB}{\log V}$ [6]
New (this paper)	**cache-oblivious**, $\mathcal{O}(V \cdot sort(E))$ I/Os, $\mathcal{O}(V)$ extra space	$\mathcal{O}(\frac{1}{B^{\frac{2}{3}}} V^{2-\frac{2}{3k}} E^{\frac{2}{3k}} \log^{\frac{2}{3}(1-\frac{1}{k})} V + \frac{k}{B} V^{2-\frac{1}{k}} E^{\frac{1}{k}} \log^{1-\frac{1}{k}} V)$	$\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + sort(E)))$ for $E \le \frac{VB}{\log^2 \frac{VE}{B}}$ $\mathcal{O}(V \cdot (\sqrt{\frac{VE}{B}} + \frac{E}{B} \log \frac{E}{B}))$ always

Table 1: I/O bounds for APSP problems on undirected graphs. ($V = |V[G]|, E = |E[G]|$, and all algorithms are cache-aware unless explicitly specified)

unweighted diameter of sparse graphs ($E = \mathcal{O}(V)$) in $\mathcal{O}(sort(kV^2 B^{\frac{1}{k}}))$ 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 $\mathcal{O}(V \cdot (\sqrt{(VE/B) \cdot \log V} + sort(E)))$ I/Os, whenever $E \le VB/\log V$. 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]. This 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 $\mathcal{O}(\sqrt{VE/B} + (E/B) \cdot \log V + MST(E))$ I/Os [7], and SSSP on an undirected graph with non-negative real-valued edge-weights can be solved in $\mathcal{O}(V + \frac{E}{B} \log \frac{E}{M})$ I/Os using the cache-oblivious Buffer Heap [8] or Bucket Heap [7].

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 produces estimated distances with an additive error of at most $2(k-1)$, where $2 \le k \le \log V$ is an integer, and $E > V \log V$. Our algorithm is based on an internal-memory algorithm in [10], and the number of I/Os performed by our algorithm is close to being a factor of B smaller than the running time of that algorithm. Our approximate algorithm performs fewer I/O operations than the $\mathcal{O}(V \cdot sort(E))$ I/O exact AP-BFS algorithm when $E > \max\{k^{\frac{k}{k-1}}, (\frac{B}{\log V})^{\frac{k}{3k-2}}\} \cdot V \log V$. For $k = 2$, 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 $\mathcal{O}(V \cdot (\sqrt{VE/B} + sort(E)))$ I/Os and $\mathcal{O}(V^2)$ space, whenever $E \le VB/\log^2(VE/B)$ (or $E = \mathcal{O}(VB/\log^2 V)$). 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.

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 2.1. **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:

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.

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.

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 $\sum_i |N(L(i))| = \mathcal{O}(E)$, this algorithm performs $\mathcal{O}(\sum_i(|L(i)| + sort(|N(L(i))|) + \frac{1}{B}(|N(L(i))| + L(i)))) = \mathcal{O}(V + sort(E))$ I/Os.

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

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

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 2.1 implies that the adjacency list of any vertex w with $d(v, w) = i$, must reside in some $A(j)$ where $i - d(u, v) \le j \le i + d(u, v)$. Therefore, we can use observation 2.1 to compute $d(v, w)$ for all $w \in V[G]$ as follows:

ALGORITHM 2.2. **Incremental-BFS**$(G, u, v, d(u, \cdot))$
(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.)

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 $A(i)$, $0 \le i < |V|$, denote the portion of this sorted list that contains adjacency lists of vertices lying exactly at distance i from u.

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 as follows:
For $j \leftarrow \max\{0, i-1-d(u, v)\}$ to $\min\{|V|-1, i-1+d(u, v)\}$ do:
Extract the adjacency list of each $w \in V[G]$ that appears in $L(i-1)$ and whose adjacency list appears in $A(j)$ by scanning $L(i-1)$ and $A(j)$ simultaneously.

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 2.1 which guarantees that the adjacency lists of all $w \in L(i-1)$ in step 2 of **Incremental-BFS** are found in the set of $A(j)$'s scanned.

We can use **Incremental-BFS** to perform BFS I/O-efficiently from all $v \in V[G]$. The following observation each part of which follows trivially from the properties of spanning trees, Euler Tours and shortest paths, is central to this extension:

OBSERVATION 2.2. *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.*

This extension is outlined in algorithm 2.3 (**AP-BFS**).

Correctness. Correctness of **AP-BFS** follows from the correctness of **MR-BFS** and **Incremental-BFS**. Moreover, observation 2.2(c) ensures that BFS will be performed from each $v \in V[G]$.

Space Complexity. Since the algorithm outputs all $\Theta(V^2)$ pairwise distances it requires $\Theta(V^2)$ space.

Algorithm 2.3. **AP-BFS**(G)

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.

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

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

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

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 2.2(a) and 2.2(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 $\mathcal{O}(V \cdot sort(E))$.

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 $2 \le k \le \log V$ 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 $\delta(u, v) \le \widehat{\delta}(u, v) \le \delta(u, v) + 2(k-1)$. 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 by Dor et al. [10] 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 [10]. Though the first term 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 [10] 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 follows:

Algorithm 3.1. **DHZ-Approx-AP-BFS**$_k$(G)

1. For $i \leftarrow 1$ to $k-1$ do: set $s_i \leftarrow \frac{E}{V} (\frac{V \log V}{E})^{\frac{i}{k}}$

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 $1 \le i \le k-1$, 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 $1 < i \le k$ 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.

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.

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 3, the distance computed 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 $u, v \in V[G]$.

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. Running independent BFS is more I/O-efficient when $|D_i|$ is smaller (i.e., i is smaller), and incremental BFS is more I/O-efficient when $G'_i(u)$ is sparser (i.e., 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. 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 vertices of degree at least s in an undirected graph, and in [10] it has been shown that this set can be found deterministically in $\mathcal{O}(V + E)$ time. We describe an external-memory version of this construction, which we call **Dominate**, that requires $\mathcal{O}(V + \frac{V^2}{B} + sort(E))$ 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*. 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 in the I/O-complexity of **Dominate**. Details of this construction are in the full paper [9].

We need another function, called **Decompose**, which is an external-memory version of an internal-memory function with the same name described in [10], and uses **Dominate** 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 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}. Clearly, $|E_i| \leq V s_{i-1}$ for $1 < i \leq k$. This function also produces dominating sets $D_1, D_2, \ldots, D_k$, and an edge set E^*. For $1 \leq i < k$, 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 $|E^*| \leq kV$. Details of **Decompose** and the analysis of its I/O complexity of is $\mathcal{O}(k(V + \frac{V^2}{B}) + sort(E))$ are in [9].

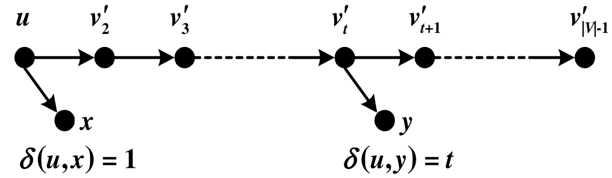

Figure 1: The directed unweighted edges that replace the undirected weighted edges of $G_i(u)$.

3.4 Replacing SSSP with 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 so, 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 $\delta(u, v)$ 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 $2 \leq \widehat{\delta}(u, v) = t \leq |V| - 1$, we add a directed edge (v'_t, v) (see Figure 1). The resulting graph $G'_i(u)$ is partially directed. The following lemma has been proved in the full paper [9] for $G'_i(u)$:

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.*

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 v'_j except v'_{i+1}, and

(b) for a vertex v with BFS level less than i, 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. Correctness of this algorithm appears in the full paper [9].

ALGORITHM 3.2. **Modified-MR-BFS**$(G'_i(u), u)$
(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.)

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*

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 $L(i) \leftarrow \{N(L(i-1)) \setminus \{L(i-1) \cup L(i-2) \cup P\}\} \cup \{v'_{i+1}\}$

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 BFS algorithm [13] on $G'_i(u)$ from u. The I/O-complexity of Mehlhorn & Meyer's algorithm is $\mathcal{O}(\sqrt{VE/B} + (E/B)\log V)$, and thus it performs better than Munagala & Ranade's algorithm (**MR-BFS** in section 2.1) 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 $\mathcal{O}(D_i(\sqrt{VE_i/B} + (E_i/B)\log V))$.

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 $\mathcal{O}((E_i/B)(V + iD_i) + D_i sort(E_i))$.

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., i is smaller), and **Interdependent-BFS** is more I/O-efficient when $|D_i|$ is larger and $G'_i(u)$ is sparser (i.e., i is larger). If we use **Independent-BFS** for all values of i, it will cause a total of $\mathcal{O}(V^2/\sqrt{B} + (k/B)V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ I/Os, and running **Interdependent-BFS** for all values of i requires a total of $\mathcal{O}(VE/B + (k/B)V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ 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**$_k$, and its proof of correctness is in [9].

ALGORITHM 3.3. **Independent-BFS**(V, E, D_i, E_i, E^*, L)
(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.)

1. Set $L' \leftarrow \emptyset$

2. Sort the vertices in D_i by vertex indices.

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 distance from u to each $v \in V$. Collect only 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': ***(i)*** (u, v'_2), ***(ii)*** (u, v) for each $v \in V$ with $\widehat{\delta}(u, v) = 1$, ***(iii)*** (v'_t, v'_{t+1}) for $2 \leq t < |V| - 1$, and ***(iv)*** (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 Mehlhorn & Meyer's BFS [13] on (V', E'), and append the computed distances to L'. The algorithm must be modified to handle the implicit partial directedness in (V', E').

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

ALGORITHM 3.4. **Interdependent-BFS**(V, E, D_i, E_i, E^*, $\langle v_1, v_2, \ldots, v_{|V|}\rangle$, L)
(Perform BFS from each $u \in D_i$ 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 graph (V, E). BFS is performed on the vertices of D_i in the order they appear in $\langle v_1, v_2, \ldots, v_{|V|}\rangle$, and distance information obtained from the last (most recent) BFS is used to reduce I/O overhead. List L is updated with the computed distances. Invoked by **Approx-AP-BFS**. See **Approx-AP-BFS** for the definition of the parameters.)

1. Set $L' \leftarrow \emptyset$

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

3. (a)-(e) Same as steps **3(a)-(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 algorithm (**Modified-MR-BFS**) with u_1 as the source to compute $d(u_1, w)$ for all $w \in V$. Append the computed distances to L'.

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 $v_2', v_3', \ldots, v_{|V|-1}'$ 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 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'.

5. Update the entries in L by sorting L' appropriately and scanning the two lists in parallel.

ALGORITHM 3.5. **Approx-AP-BFS**$_k(G, l)$
(Given an undirected graph $G = (V[G], E[G])$ and a switching parameter l, computes the shortest distance between every pair of vertices in G with additive error of at most $2(k-1)$.)

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 $(\langle E_1, E_2, \ldots, E_k, E^*\rangle, \langle D_1, D_2, \ldots, D_k\rangle) \leftarrow$ **Decompose**$(G, \langle s_1, s_2, \ldots, s_{k-1}\rangle)$
(c) Sort the edges in $E[G]$ so that edge (u_1, v_1) is placed ahead 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.

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 an *Euler Tour ET* of T. Mark the first occurrence of each vertex on ET; let $v_1, v_2, \ldots, v_{|V|}$ be the marked vertices in the order they appear on ET.
(c) For $i \leftarrow l+1$ to k do: **Interdependent-BFS**$(V, E, D_i, E_i, E^*, \langle v_1, v_2, \ldots, v_{|V|}\rangle, L)$

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

I/O Complexity of Approx-AP-BFS$_k$. I/O cost of step 1 is dominated by that of **Decompose** which is $\mathcal{O}(k(V + V^2/B) + sort(E))$. Step 2(a) requires $\mathcal{O}(\sum_{i=1}^{l} D_i(\sqrt{VE_i/B} + (E_i/B)\log V)) = \mathcal{O}(V^2\sqrt{V/(BE\alpha^{l+1})}\log V + (l/B)V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$ I/Os, where $\alpha = (V \log V/E)^{\frac{1}{k}}$. Step 2(b) incurs $\mathcal{O}(sort(E) \cdot \log_2\log_2 \frac{VB}{E})$ I/Os [5]. The I/O-complexity of step 2(c) is $\mathcal{O}(\sum_{i=l+1}^{k}\{(E_i/B)(V + iD_i) + D_i \cdot sort(E_i)\}) = \mathcal{O}(VE\alpha^{l-1}/B + ((k-l)/B)V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$.

Therefore the total I/O cost of **Approx-AP-BFS$_k$** is $\mathcal{O}(V^2\sqrt{V/(BE\alpha^{l+1})}\log V + VE\alpha^{l-1}/B + (k/B)V^{2-\frac{1}{k}}E^{\frac{1}{k}}\log^{1-\frac{1}{k}} V)$. This expression is minimized for $l = (\log(V^3B\log^2 V) - \log(E^3\alpha))/(3\log\alpha) + 1$, and thus 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 $k = 2$.

We can externalize the internal-memory approximation algorithm by Aingworth et al. [2] 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 is described in detail in [9] and outperforms **Approx-AP-BFS$_2$** whenever $B > \frac{V^{\frac{5}{2}}\log^2 V}{E}$ assuming $V \ge \log^4 V$ and $E \le \frac{V^2}{\log V}$.

4 Cache-Aware APSP for Weighted Undirected Graphs

In [6], Arge et al. introduce the *Multi-Tournament-Tree* to obtain an $\mathcal{O}(V \cdot (\sqrt{(VE/B)\log V} + sort(E)))$ I/O cache-aware algorithm for computing APSP on general weighted undirected graphs with $E \le VB/\log V$. In this section we introduce the *Multi-Buffer-Heap*, and use it to obtain an $\mathcal{O}(V \cdot (\sqrt{VE/B} + sort(E)))$ I/O cache-aware algorithm for solving the same problem assuming $E \le VB/\log^2(VE/B)$ or $E = \mathcal{O}(VB/\log^2 V)$. This leads to an $\mathcal{O}(V \cdot (\sqrt{VE/B} + (E/B)\log E/B))$ I/O algorithm for any edge density using $O(V^2)$ 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)** 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.

In the next section we present a slim data structure based on the Buffer Heap [8], 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. In section 4.3 we use a collection of Slim Buffer Heaps in a *Multi-Buffer-Heap*.

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 structure.

4.2 The Slim Buffer Heap. In this section we extend the cache-oblivious Buffer Heap [8] 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. 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 $k_x < k'_x$. 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 operations in $\mathcal{O}(\frac{1}{\lambda}+\frac{1}{B}\log\frac{N}{\lambda})$ amortized I/Os, where N is the number of elements.

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 queue. 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-Min/Delete/ Decrease-Key* operation collects enough elements from higher level element buffers to fill the slim cache.

After each operation the *Reconstruct* function is called which reconstructs the entire data structure periodically. 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.

FUNCTION 4.1. **Decrease-Key**(x, k_x)**/Delete**(x)
(Inserts a *Decrease-Key/Delete* operation into the structure.)

1. Push the operation into U_0 augmented with current time stamp

2.
- Set $B' \leftarrow \emptyset$, $i \leftarrow 0$ {*List B' stores elems returned by* **Fix-U**}
- **Fix-U**(i, B')

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

4. Reconstruct()

FUNCTION 4.2. **Fix-U**(i, B')
(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.)

1. While $i < r$ AND ($|U_i| > 2^i$ OR ($i = t+1$ AND $|B'| = 0$) OR ($i > t+1$ AND $|B'| < \lambda$)) do:

• **Apply-Updates**(i)
• Append the elements of B_i to B'
• Set $i \leftarrow i+1$
2. If $i < r$ then merge the segments of U_i

FUNCTION 4.3. **Apply-Updates**(i)
(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*s.)

1. If $|B_i| = 0$ and $i < r-1$ then:
• Merge the segments of U_i
• Empty U_i by moving contents of U_i as a new segment of U_{i+1}

2. Else ($|B_i| > 0$ or $i = r-1$) do:
• Merge the segments of U_i
• If $i = r-1$ then set $k \leftarrow +\infty$ else set $k \leftarrow$ largest key in B_i
• 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 $k_x \leq k$
• 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}
• 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)
• Set $U_i \leftarrow \emptyset$

FUNCTION 4.4. **Delete-Min**()
(Extracts the element with the smallest key from the structure.)

1. Set $i \leftarrow 0$
Repeat
 – **Apply-Updates**(i)
 – Set $i \leftarrow i+1$
Until B_i is non-empty or $i = r$
2.
• Set $B' \leftarrow B_i$, $i \leftarrow i+1$
• **Fix-U**(i, B')
3.
• Extract the minimum-key element from B'
• Move rest of B' to the shallowest possible element buffers maintaining invariants 4.1(a), 4.2(a) and 4.3(a)
4. Reconstruct()

FUNCTION 4.5. **Reconstruct**()
(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/initialization of SBH.)

1. If $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$ then:
• For $i \leftarrow 0$ to $r-1$ do **Apply-Updates**(i)
• Distribute remaining elements to shallowest element buffers

---

4.2.4 Analysis.

Correctness of the operations is straight-forward, and the proof is in the full paper [9].

The proof of the following lemma is also in [9].

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**) incurs only $\mathcal{O}(\frac{1}{B})$ amortized I/Os per update in U_i.

We now state the main lemma of this section.

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. (Sketch - see [9] for details) 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. Further, any time a U buffer is examined, it is emptied and its contents moved down to the next lower buffer, and between two successive emptyings it never receives more than a constant number of batches of updates. Similarly, any time a B buffer is examined, each element in it is either moved up to a higher B buffer or is moved to a lower U buffer as a *Sink* operation. The one exception is when a B buffer is examined during **Fix-U**, and the cost of this is paid by the drop in potential due to the upward movement of $\Omega(\lambda)$ elements in element buffers (this is the reason for the factor 2 that appears before the summation part in the potential function). Ignoring the *Sink* operation for the moment, all other costs are paid for by the corresponding drop in potential. One unit of $\frac{1}{\lambda}$ on $\Theta(\lambda)$ entries in the top t levels pays for the cost of bringing in a new block when an access is made to an entry in level $t+1$. Finally the cost of the *Sink* operations is handled in the same manner as in [8], namely by the drop in potential incurred by the removal of the *Decrease-Key* operation that triggered the *Sink*. The Δ_o terms appearing in the potential function ensures enough potential drop to pay for the cost of periodic reconstruction of the data structure. □

4.3 Multi-Buffer-Heap and External-Memory APSP. 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 in [6]. We work on all V underlying SSSP problems simultaneously, and solve each individual SSSP problem using Kumar & Schwabe's algorithm for weighted undirected graphs [12]. For $1 \le i \le V$, we require 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 we load the multi-slim-cache of each MBH, and for each MBH extract a settled vertex with minimum distance from each of the $\Theta(L)$ priority queue pairs it stores. We sort the extracted vertices by vertex indices, and scan 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 $\mathcal{O}(VE \cdot (\frac{1}{\lambda} + \frac{1}{B}\log_2\frac{E}{\lambda}))$ 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 $\mathcal{O}(V \cdot (\frac{V}{L} + \frac{E}{\lambda} + \frac{E}{B}\log_2\frac{E}{\lambda} + sort(E)))$. Using $L = \sqrt{VB/E} \ge 1$, 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* $\mathcal{O}(V \cdot (\sqrt{VE/B} + sort(E)))$ *I/Os and* $\mathcal{O}(V^2)$ *space whenever* $E \le \frac{VB}{\log^2 VE/B}$ *(or* $E = \mathcal{O}(\frac{VB}{\log^2 V})$*).*

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* $\mathcal{O}(V \cdot (\sqrt{VE/B} + (E/B)\log E/B))$ *I/Os and* $\mathcal{O}(V^2)$ *space.*

References

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

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

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th WADS*, pp. 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 *Proc. STOC*, pp. 268–276, 2002.

[5] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP, and multi-way planar graph separation. In *Proc. 7th SWAT*, pp. 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 *Proc. 31st ICALP*, pp. 146–157, 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 *Proc. 9th SWAT*, pp. 480–492, 2004.

[8] R. A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proc. 16th SPAA*, pp. 245–254, 2004.

[9] R. A. Chowdhury and V. Ramachandran. External-Memory Exact and Approximate All-Pairs Shortest-Paths in Undirected Graphs. Tech. Rep. TR-04-38, UT Austin, 2004.

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

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th FOCS*, pp. 285–297, 1999.

[12] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th SPDP*, pp. 169–177, 1996.

[13] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th ESA, LNCS 2461*, pp. 723–735, 2002.

[14] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. 11th ESA, LNCS 2832*, pp. 434–445, 2003.

[15] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th SODA*, pp. 687–694, 1999.

[16] H. Prokop. Cache-oblivious algorithms. Master's thesis, Dept. of EECS, MIT, June 1999.

[17] U. Zwick. Exact and approximate distances in graphs – a survey. In *Proc. 9th ESA, LNCS 2161*, pp. 33–48, 2001. Updated version at http://www.cs.tau.ac.il/~zwick.