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 $\mathcal{O}(sort(kV^2 B^{\frac{1}{k}}))$ I/Os and $\mathcal{O}(kV)$ space for any integer k, $3 \leq k \leq \log B$.

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 $E \leq \frac{VB}{\log V}$. 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, $d(u,w) - d(u,v) \le d(v,w) \le d(u,w) + d(u,v)$.*

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 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 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 $A(i)$, $0 \le i < |V|$, 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 $\max\{0, i-1-d(u,v)\} \le j \le \min\{|V|-1, i-1+d(u,v)\}$.

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 $2 \le k \le \log V$ 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 (**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 $1 \le i \le k-1$, $D_i$ 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 $G$.

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 $\mathcal{O}(V + \frac{V^2}{B} + sort(E))$ 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 $\mathcal{O}(k(V + \frac{V^2}{B}) + sort(E))$.