Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths

Gerth Stølting Brodal[1,⋆,⋆⋆], Rolf Fagerberg[1,⋆⋆], Ulrich Meyer[2,⋆⋆,⋆⋆⋆], and Norbert Zeh[3]

[1] BRICS, Department of Computer Science, University of Aarhus, DK-8000 Århus C, Denmark
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
[3] Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 1W5, Canada

Abstract. We present improved cache-oblivious data structures and algorithms for breadth-first search and the single-source shortest path problem on undirected graphs with non-negative edge weights. Our results close the performance gap between the currently best *cache-aware* algorithms for these problems and their *cache-oblivious* counterparts. Our shortest-path algorithm relies on a new data structure, called *bucket heap*, which is the first cache-oblivious priority queue to efficiently support a weak DecreaseKey operation.

1 Introduction

Breadth-first search (BFS) and the *single-source shortest path* (SSSP) problem are fundamental combinatorial optimization problems with numerous applications. SSSP is defined as follows: Let $G = (V, E)$ be a graph with V vertices and E edges,[1] let s be a distinguished vertex of G, and let ω be an assignment of non-negative real *weights* to the edges of G. The weight of a path is the sum of the weights of its edges. We want to find for every vertex v that is reachable from s, the weight $\mathrm{dist}(s, v)$ of a minimum-weight ("shortest") path from s to v. BFS can be seen as the unweighted version of SSSP.

Both problems are well understood in the RAM model, where the cost of a memory access is assumed to be independent of the accessed memory location. However, modern computers contain a hierarchy of memory levels; the cost of a memory access depends on the currently lowest memory level that contains the accessed element. This is not accounted for in the RAM model, and current BFS and SSSP-algorithms, when run in memory hierarchies, turn out to be

⋆ Supported by the Carlsberg Foundation (contract number ANS-0257/20).
⋆⋆ Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
⋆⋆⋆ Partially supported by DFG grant SA 933/1-1.

[1] For convenience, we use the name of a set to denote both the set and its cardinality. Furthermore, we will assume that $E = \Omega(V)$, in order to simplify notation.

notoriously inefficient for sparse input graphs. The purpose of this paper is to provide improved data structures and algorithms for BFS and SSSP under the currently most powerful model for multi-level memory hierarchies.

Models for memory hierarchies. The most widely used model for the design of cache-aware algorithms is the *I/O-model* of Aggarwal and Vitter [2]. This model assumes a memory hierarchy consisting of two levels; the lower level has size M; data is transferred between the two levels in blocks of B consecutive data items. The complexity of an algorithm is the number of blocks transferred (I/Os). The parameters M and B are assumed to be known to the algorithm. The strength of the I/O-model is its simplicity, while it still adequately models the situation when the I/Os between two levels of the memory hierarchy dominate the running time of the algorithm; this is often the case when the size of the data significantly exceeds the size of main memory. *Cache-oblivious* algorithms are designed to be I/O-efficient without knowing M or B; that is, they are formulated in the RAM model and analyzed in the I/O-model, assuming that the memory transfers are performed by an optimal offline paging algorithm. Since the analysis holds for any block and memory sizes, it holds for *all* levels of a multi-level memory hierarchy (see [15] for details). Thus, the cache-oblivious model elegantly combines the simplicity of the I/O-model with a coverage of the entire memory hierarchy.

A comprehensive list of results for the I/O-model have been obtained—see [3, 18, 21] and the references therein. One of the fundamental facts is that, in the I/O-model, comparison-based sorting of N elements takes $\Theta(\text{Sort}(N))$ I/Os in the worst case, where $\text{Sort}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}$.

For the cache-oblivious model, Frigo et al. developed optimal cache-oblivious algorithms for matrix multiplication, matrix transposition, fast Fourier transform, and sorting [15]. The cache-oblivious sorting bound matches that for the I/O-model: $\mathcal{O}(\text{Sort}(N))$ I/Os. After the publication of [15], a number of results for the cache-oblivious model have appeared; see [12, 18] for recent surveys.

Some results in the cache-oblivious model, in particular those concerning sorting and algorithms and data structures that can be used to sort, such as priority queues, are proved under the assumption that $M \geq B^2$. This is also known as the *tall-cache assumption*. In particular, this assumption is made in the FUNNELSORT algorithm of Frigo et al. [15]. A variant termed LAZY-FUNNELSORT [6] works under the weaker tall-cache assumption that $M \geq B^{1+\varepsilon}$, for any fixed $\varepsilon > 0$. Recently, it has been shown [8] that a tall-cache assumption is necessary for cache-oblivious comparison-based sorting algorithms.

Previous and related work. Graph algorithms for the I/O-model have received considerable attention in recent years. Most efficient cache-aware algorithms do have a cache-oblivious counterpart that achieves the same performance; see Table 1. Despite these efforts, only little progress has been made on the solution of the SSSP-problem with general non-negative edge weights using either cache-aware or cache-oblivious algorithms: The best known lower bound is $\Omega(\text{Sort}(E))$ I/Os, which can be obtained through a reduction from list ranking; but the

Problem	Best cache-oblivious result		Best cache-aware result	
List ranking	$\mathcal{O}(\mathrm{Sort}(V))$	[4]	$\mathcal{O}(\mathrm{Sort}(V))$	[10]
Euler Tour	$\mathcal{O}(\mathrm{Sort}(V))$	[4]	$\mathcal{O}(\mathrm{Sort}(V))$	[10]
Spanning tree/MST	$\mathcal{O}(\mathrm{Sort}(E) \cdot \log\log V)$ $\mathcal{O}(\mathrm{Sort}(E))$ (randomized)	[4] [1]	$\mathcal{O}(\mathrm{Sort}(E) \cdot \log\log(V \cdot B/E))$ $\mathcal{O}(\mathrm{Sort}(E))$ (randomized)	[5] [1]
Undirected BFS	$\mathcal{O}(V + \mathrm{Sort}(E))$ $\mathcal{O}(\mathrm{ST}(E) + \mathrm{Sort}(E) + \frac{E}{B}\log V + \sqrt{VE/B})$ $\mathcal{O}(\mathrm{ST}(E) + \mathrm{Sort}(E) + \frac{E}{B} \cdot \frac{1}{\varepsilon} \cdot \log\log V + \sqrt{VE/B} \cdot \sqrt{VB/E}^{\varepsilon})$	[20] **New** **New**	$\mathcal{O}(\sqrt{VE/B} + \mathrm{Sort}(E) + \mathrm{ST}(E))$	[17]
Directed BFS & DFS	$\mathcal{O}((V + E/B) \cdot \log V + \mathrm{Sort}(E))$	[4]	$\mathcal{O}((V + E/B) \cdot \log V + \mathrm{Sort}(E))$	[9]
Undirected SSSP	$\mathcal{O}(V + (E/B)\log(E/B))$	**New**	$\mathcal{O}(V + (E/B)\log(E/B))$	[16]

Table 1. I/O-bounds for some fundamental graph problems.

currently best algorithm [16] performs $\mathcal{O}(V + (E/B)\log_2(E/B))$ I/Os on undirected graphs. For $E = \mathcal{O}(V)$, this is hardly better than naïvely running Dijkstra's internal-memory algorithm [13, 14] in external memory, which would take $\mathcal{O}(V \log_2 V + E)$ I/Os. On dense graphs, however, the algorithm is efficient. The algorithm of [16] is not cache-oblivious, because the applied external-memory priority queue based on the tournament tree is not cache-oblivious. Cache-oblivious priority queues exist [4, 7]; but none of them efficiently supports a DecreaseKey operation. Indeed, the tournament tree is also the only cache-aware priority queue that supports at least a weak form of this operation.

For bounded edge weights, an improved external-memory SSSP-algorithm has been developed recently [19]. This algorithm is an extension of the currently best external-memory BFS-algorithm [17], Fast-BFS, which performs $\mathcal{O}(\sqrt{V \cdot E/B} + \mathrm{Sort}(E) + \mathrm{ST}(E))$ I/Os, where $\mathrm{ST}(E)$ is the number of I/Os required to compute a spanning tree (see Table 1). Again, the key data structure used in Fast-BFS is not cache-oblivious, which is why the currently best cache-oblivious BFS-algorithm is that of [20].

Our results. In Section 2, we develop the first cache-oblivious priority queue, called *bucket heap*, that supports an Update operation, which is a combined Insert and DecreaseKey operation. The amortized cost of operations Update, Delete, and DeleteMin is $\mathcal{O}((1/B)\log_2(N/B))$ where N is the number of distinct elements in the priority-queue. Using the bucket heap, we obtain a cache-oblivious shortest-path algorithm for undirected graphs with non-negative edge weights that matches the performance of the best cache-aware algorithm for this problem: $\mathcal{O}(V + (E/B)\log_2(E/B))$ I/Os. Independently of our work, the bucket heap and a cache-oblivious version of the tournament tree have been developed also by Chowdhury and Ramachandran [11].

In Section 3, we develop a new cache-oblivious algorithm for undirected BFS. The algorithm comes in two variants: The first variant performs $\mathcal{O}(\mathrm{ST}(E) +$

$\text{Sort}(E) + \frac{E}{B} \log V + \sqrt{VE/B})$ I/Os; the second variant performs $\mathcal{O}(\text{ST}(E) + \text{Sort}(E) + \frac{E}{B} \cdot \frac{1}{\varepsilon} \cdot \log \log V + \sqrt{VE/B} \cdot \sqrt{VB/E}^{\varepsilon})$ I/Os, for any $\varepsilon > 0$. Here, $\text{ST}(E)$ denotes the cost of cache-obliviously finding a spanning tree.

2 The Bucket Heap and Undirected Shortest Paths

In this section, we describe the *bucket heap*, which is a priority queue that supports an Update (a weak DecreaseKey) operation and does so in the same I/O-bound as the tournament tree of [16]. Using the bucket heap, the SSSP-algorithm of [16] becomes cache-oblivious. Similar to the tournament tree, the bucket heap supports the following three operations, where we refer to an element x with priority p as element (x, p):

Update(x, p) inserts element (x, p) into the priority queue if x is not in the priority queue; otherwise, it replaces the current element (x, p') in the priority queue with $(x, \min(p, p'))$.

Delete(x) removes element x from the priority queue.

DeleteMin removes and reports the minimal element in the priority queue.

The bucket heap consists of q *buckets* $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_q$ and $q+1$ *signal buffers* $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{q+1}$, where q varies over time, but is always at most $\lceil \log_4 N \rceil$. The *capacity* of bucket $\mathcal{B}_i$ is 2^{2i}; buffer $\mathcal{S}_i$ has capacity 2^{2i-1}. In order to allow for temporary overflow, we allocate 2^{2i+1} memory locations for bucket $\mathcal{B}_i$ and 2^{2i} memory locations for buffer $\mathcal{S}_i$. We store all buckets and buffers consecutively in memory, in the following order: $\mathcal{S}_1, \mathcal{B}_1, \mathcal{S}_2, \mathcal{B}_2, \ldots, \mathcal{S}_q, \mathcal{B}_q, \mathcal{S}_{q+1}$.

Buckets $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_q$ store the elements currently in the priority queue. We maintain the invariant that for any two buckets $\mathcal{B}_i$ and $\mathcal{B}_j$ with $i < j$ and any two elements $(x, p) \in \mathcal{B}_i$ and $(y, q) \in \mathcal{B}_j$, $p \leq q$.

Buffers $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{q+1}$ store three types of signals, which we use to update the bucket contents in a lazy manner: Update(x, p) and Delete(x) signals are used to implement Update(x, p) and Delete(x) operations. A Push(x, p) signal is used to push elements from a bucket $\mathcal{B}_i$ to a bucket $\mathcal{B}_{i+1}$ when $\mathcal{B}_i$ overflows. Every signal has a time stamp corresponding to the time when the operation posting this signal was performed. The time stamp of an element in a bucket is the time stamp of the Update signal that led to its insertion.

The three priority queue operations are implemented as follows: A DeleteMin operation uses the Fill operation below to make sure that bucket $\mathcal{B}_1$ is non-empty and, hence, contains the element with lowest priority. This element is removed and returned. An Update(x, p) or Delete(x) operation inserts the corresponding signal into $\mathcal{S}_1$ and empties $\mathcal{S}_1$ using the Empty operation below. Essentially, all the work to update the contents of the bucket heap is delegated to two auxiliary procedures: Procedure Empty$(\mathcal{S}_i)$ empties the signal buffer $\mathcal{S}_i$, applies these signals to bucket $\mathcal{B}_i$, and inserts appropriate signals into buffer $\mathcal{S}_{i+1}$. If this leads to an overflow of buffer $\mathcal{S}_{i+1}$, the procedure is applied recursively to $\mathcal{S}_{i+1}$. Procedure Fill$(\mathcal{B}_i)$ fills an underfull bucket $\mathcal{B}_i$ with the smallest 2^{2i} elements in buckets $\mathcal{B}_i, \ldots, \mathcal{B}_q$. The details of these procedures are as follows:

$\textsc{Empty}(\boldsymbol{\mathcal{S}_i})$

1 If $i = q + 1$, then increase q by one and create two new arrays $\mathcal{B}_q$ and S_{q+1}.
2 Scan bucket $\mathcal{B}_i$ to determine the maximal priority p' of the elements in $\mathcal{B}_i$. (If $i = q$ and $\mathcal{S}_{q+1} = \emptyset$, then $p' = \infty$. Otherwise, if $\mathcal{B}_i = \emptyset$, then $p' = -\infty$.)
3 Scan buckets $\mathcal{S}_i$ and $\mathcal{B}_i$ simultaneously and perform the following operations for each signal in $\mathcal{S}_i$:
   $\textsc{Update}(x, p)$: If $\mathcal{B}_i$ contains an element (x, p''), replace this element with $(x, \min(p, p''))$ and mark the $\textsc{Update}(x, p)$ signal in $\mathcal{S}_i$ as handled. If x is not in $\mathcal{B}_i$, but $p \leq p'$, insert (x, p) into $\mathcal{B}_i$ and replace the $\textsc{Update}(x, p)$ signal with a $\textsc{Delete}(x)$ signal. If x is not in $\mathcal{B}_i$ and $p > p'$, do nothing.
   $\textsc{Push}(x, p)$: If there is an element (x, p'') in $\mathcal{B}_i$, replace it with (x, p). Otherwise, insert (x, p) into $\mathcal{B}_i$. Mark the $\textsc{Push}(x, p)$ signal as handled.
   $\textsc{Delete}(x)$: If element x is in $\mathcal{B}_i$, delete it.
4 If $i < q$ or $\mathcal{S}_{i+1}$ is non-empty, then scan buffers $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ and insert all unhandled signals in $\mathcal{S}_i$ into $\mathcal{S}_{i+1}$. $\mathcal{S}_i \leftarrow \emptyset$
5 If $|\mathcal{B}_i| > 2^{2i}$, then find the 2^{2i}-th smallest priority p in $\mathcal{B}_i$. Scan bucket $\mathcal{B}_i$ and buffer $\mathcal{S}_{i+1}$ twice: The first scan removes all elements with priority greater than p from $\mathcal{B}_i$ and inserts corresponding $\textsc{Push}$ signals into $\mathcal{S}_{i+1}$. The second scan removes $|\mathcal{B}_i| - 2^{2i}$ elements with priority p from $\mathcal{B}_i$ and inserts corresponding $\textsc{Push}$ signals into $\mathcal{S}_{i+1}$.
6 If $|\mathcal{S}_{i+1}| > 2^{2i+1}$, then $\textsc{Empty}(\mathcal{S}_{i+1})$

$\textsc{Fill}(\boldsymbol{\mathcal{B}_i})$

1 $\textsc{Empty}(\mathcal{S}_i)$
2 If $|\mathcal{B}_{i+1}| < 2^{2i}$ and $i < q$, then $\textsc{Fill}(\mathcal{B}_{i+1})$
3 Find the $(2^{2i} - |\mathcal{B}_i|)$-th smallest priority p in $\mathcal{B}_{i+1}$. Scan $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$ twice. The first scan moves all elements with priority less than p from $\mathcal{B}_{i+1}$ to $\mathcal{B}_i$. The second scan moves the correct number of elements with priority p from $\mathcal{B}_{i+1}$ to $\mathcal{B}_i$ so that $\mathcal{B}_i$ contains 2^{2i} elements or $\mathcal{B}_{i+1}$ is empty at the end.
4 $q \leftarrow \max\{j : \mathcal{B}_j \text{ or } \mathcal{S}_{j+1} \text{ is non-empty}\}$

2.1 Correctness

A priority queue is *correct* if, given a sequence $o_1, o_2, \ldots, o_t$ of priority queue operations, every $\textsc{DeleteMin}$ operation o_i returns the smallest element in the set $\mathcal{O}_{i-1}$ constructed by operations $o_1, \ldots, o_{i-1}$ according to their definitions at the beginning of this section. In the following we use the term "element at level j" to refer to an element in bucket $\mathcal{B}_j$ or to an $\textsc{Update}$ signal in buffer $\mathcal{S}_j$. We say that a $\textsc{Delete}(x)$ signal in a buffer $\mathcal{S}_i$ *hides* an element (x, p) at level j if $i \leq j$ and the time stamp of the $\textsc{Delete}(x)$ signal is greater than the time stamp of element (x, p). An element (x, p') hides an element (x, p) if it is not hidden by a $\textsc{Delete}(x)$ signal and $p' < p$. An element that is not hidden is *visible*.

Observe that a hidden element can never be returned by a $\textsc{DeleteMin}$ operation. Hence, it suffices to show that the set $\mathcal{V}_i$ of elements that are visible after applying operations $o_1, o_2, \ldots, o_i$ equals the set $\mathcal{O}_i$ and that every $\textsc{DeleteMin}$ operation returns the minimal element in the current set of visible elements.

Lemma 1. *A* DeleteMin *operation o_i returns the minimal element in $\mathcal{V}_{i-1}$.*

Proof. We have already observed that the returned element (y, q) is in $\mathcal{V}_{i-1}$. If (x, p) is the smallest visible element and $q > p$, we distinguish a number of cases: Let i be the current level of element (x, p), and let j be the current level of element (y, q). If $i \le j$, then element (x, p) will move to lower buckets before or at the same time as (y, q). Hence, (x, p) would have to be returned before (y, q), a contradiction. If $j < i$, we observe that element (y, q) can only be represented by an Update(y, q) signal in $\mathcal{S}_j$ because neither bucket $\mathcal{B}_i$ nor buffer $\mathcal{S}_i$ can contain elements with priorities less than those of the elements in $\mathcal{B}_j$. Hence, before (y, q) can be moved to a bucket and ultimately returned, we must reach the case $i \le j$, in which case we would return (x, p) first, as argued above. □

Lemma 2. *For any sequence $o_1, o_2, \ldots, o_t$ and any $1 \le i \le t$, $\mathcal{V}_i = \mathcal{O}_i$.*

Proof. The proof is by induction on i. For $i = 0$, the claim holds because $\mathcal{O}_i$ and $\mathcal{V}_i$ are empty. So assume that $i > 0$ and that $\mathcal{V}_{i-1} = \mathcal{O}_{i-1}$. If o_i is a DeleteMin operation, it removes and returns the minimum element (x, p) in $\mathcal{V}_{i-1}$, so that $\mathcal{V}_i = \mathcal{V}_{i-1} \setminus \{(x, p)\} = \mathcal{O}_{i-1} \setminus \{(x, p)\} = \mathcal{O}_i$. If o_i is a Delete(x) operation, its insertion into $\mathcal{S}_1$ hides all copies of x in the priority queue. Hence, $\mathcal{V}_i = \mathcal{V}_{i-1} \setminus \{x\} = \mathcal{O}_{i-1} \setminus \{x\} = \mathcal{O}_i$. If o_i is an Update(x, p) operation, we distinguish three cases: If x is not in $\mathcal{O}_{i-1}$, there is no element (x, p') that hides the Update(x, p) signal. Hence, $\mathcal{V}_i = \mathcal{V}_{i-1} \cup \{(x, p)\} = \mathcal{O}_{i-1} \cup \{(x, p)\} = \mathcal{O}_i$. If there is an element (x, p') in $\mathcal{O}_{i-1}$ and $p' < p$, element (x, p') hides the Update(x, p) signal, and $\mathcal{V}_i = \mathcal{V}_{i-1} = \mathcal{O}_{i-1} = \mathcal{O}_i$. If $p' > p$, the Update(x, p) signal hides element (x, p'), and $\mathcal{V}_i = (\mathcal{V}_{i-1} \setminus \{(x, p')\}) \cup \{(x, p)\} = (\mathcal{O}_{i-1} \setminus \{(x, p')\}) \cup \{(x, p)\} = \mathcal{O}_i$. □

2.2 Analysis

We assume that every element has a unique ID drawn from a total order and keep the elements in each bucket or buffer sorted by their IDs. This invariant allows us to perform updates by scanning buckets and buffers as in the description of procedures Empty and Fill. The amortized cost per scanned element is hence $\mathcal{O}(1/B)$. In our analysis of the amortized complexity of the priority queue operations, we assume that $M = \Omega(B)$, large enough to hold the first $\log_4 B$ buckets and buffers plus one cache block per stream that we scan. Under this assumption, operations Update, Delete, and DeleteMin, excluding the calls to Empty and Fill, do not cause any I/Os because bucket $\mathcal{B}_1$ and buffer $\mathcal{S}_1$ are always in cache. We have to charge the I/Os incurred by Empty and Fill operations to Update and Delete operations in such a manner that no operation is charged for more than $\mathcal{O}((1/B)\log_2(N/B))$ I/Os. To achieve this, we define the following potential function, where U, D, and P are the numbers of Update, Delete, and Push signals in buffers $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_q$:

$$\Phi = (3U + D + 2P)(\log_4(N/B) + 3) + \sum_{i=\log_4 B}^{q} \left((|\mathcal{B}_i| - |S_i|) \cdot (i - \log_4 B) + 2^{2i}\right)$$

Since the actual cost of an UPDATE or DELETE operation is 0 and each of them increases the potential by $\mathcal{O}(\log_2(N/B))$, the amortized cost of these operations is $\mathcal{O}((1/B)\log_2(N/B))$ if we can show that all I/Os incurred by FILL and EMPTY operations can be paid by a sufficient potential decrease, where a potential decrease of $\Omega(B)$ is necessary to pay for a single I/O.

We distinguish two types of EMPTY operations: A *regular* EMPTY$(\mathcal{S}_i)$ operation is one with $|\mathcal{S}_i| > 2^{2i-1}$. If $|\mathcal{S}_i| \leq 2^{2i-1}$, the operation is *early.* The latter type of EMPTY$(\mathcal{S}_i)$ operation may by triggered by a FILL$(\mathcal{B}_i)$ operation.

First consider a regular EMPTY$(\mathcal{S}_i)$ operation, where $i \leq q$. If $i < \log_4 B$, the operation causes no I/Os—because $\mathcal{S}_i$, $\mathcal{S}_{i+1}$, and $\mathcal{B}_i$ are in cache—and the potential does not increase. If $i \geq \log_4 B$, the cost is bounded by $\mathcal{O}(2^{2i-1}/B)$ because only buffers $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ and bucket $\mathcal{B}_i$ are scanned. Let k be the increase of the size of bucket $\mathcal{B}_i$, let u, p, and d be the number of UPDATE, PUSH, and DELETE operations in $\mathcal{S}_i$, and let u', p', and d' be the number of such operations inserted into $\mathcal{S}_{i+1}$. Then we have $u+d+p = |\mathcal{S}_i| > 2^{2i-1}$. From the description of the EMPTY operation, we obtain that $u+d = u'+d'$ and $k+u'+p' \leq u+p$. The change of potential is $\Delta\Phi \leq (3u'+d'+2p'-3u-d-2p)(\log_4(N/B)+3)+(u+d+p+k)(i-\log_4 B)-(u'+d'+p')(i+1-\log_4 B)$. Using elementary transformations, this gives $\Delta\Phi \leq -2^{2i-1}$.

If $i = q+1$, the actual cost of a regular EMPTY$(\mathcal{S}_i)$ operation remains the same, but the change of potential is $\Delta\Phi \leq -(3u+d+2p)(\log_4(N/B)+3)+(u+d+p+k)(i-\log_4 B)+2^{2i}$. Again, this can be bounded by $\Delta\Phi \leq -2^{2i-1}$.

Next we show that the cost of FILL and early EMPTY operations can be paid by a sufficient decrease of the potential Φ. Consider a FILL$(\mathcal{B}_1)$ operation, and let j be the highest index such that a FILL$(\mathcal{B}_j)$ operation is triggered by this FILL$(\mathcal{B}_1)$ operation. Then the cost of all FILL and early EMPTY operations triggered by this FILL$(\mathcal{B}_1)$ operation is $\mathcal{O}(2^{2j}/B)$. If there are new elements inserted into the buckets during the EMPTY operations, a similar analysis as for regular EMPTY operations shows that the resulting potential change is zero or negative. Hence, it suffices to show that, excluding these insertions, the potential decreases by $\Omega(2^{2j})$. We distinguish two cases. if q does not change, then the FILL$(\mathcal{B}_j)$ operation moves at least $3 \cdot 2^{2j-2}$ elements from $\mathcal{B}_{j+1}$ to $\mathcal{B}_j$, which results in the desired potential decrease. If q decreases, then $q > j$ before the FILL$(\mathcal{B}_1)$ operation and q decreases by at least one. This results in a potential decrease of at least $2^{2q} \geq 2^{2j}$. Hence, we obtain the following result.

Theorem 1. *The bucket heap supports the operations* UPDATE, DELETE, *and* DELETEMIN *at an amortized cost of* $\mathcal{O}((1/B)\log_2(N/B))$ *I/Os.*

The shortest path algorithm of [16] is cache-oblivious, except for its use of a cache-aware priority queue: the tournament tree. Since the bucket heap is cache-oblivious and achieves the same I/O-bound as the tournament tree, we obtain the following result by replacing the tournament tree with the bucket heap.

Corollary 1. *There exists a cache-oblivious algorithm that solves the single-source shortest path problem on an undirected graph* $G = (V, E)$ *with non-negative edge weights and incurs at most* $\mathcal{O}(V + (E/B)\log_2(E/B))$ *I/Os.*

3 Cache-Oblivious Breadth-First Search

In this section, we develop a cache-oblivious version of the undirected BFS-algorithm from [17]. As in [17], the actual BFS-algorithm is the algorithm from [20], which generates the BFS levels L_i one by one, exploiting that, in an undirected graph, $L_{i+1} = \mathcal{N}(L_i) \setminus (L_i \cup L_{i-1})$, where $\mathcal{N}(S)$ denotes the set of nodes[2] that are neighbours of nodes in S, and L_i denotes the nodes of the i'th level of the BFS traversal, that is, the nodes at distance i from the source vertex s. The algorithm from [20] relies only on sorting and scanning and, hence, is straightforward to implement cache-obliviously; this gives the cache-oblivious $\mathcal{O}(V + \mathrm{Sort}(E))$ result mentioned in Table 1.

The speed-up in [17] compared to [20] is achieved using a data structure which, for a query set S, returns $\mathcal{N}(S)$ in an I/O-efficient manner. To do so, the data structure exploits the fact that the query sets are the levels $L_0, L_1, L_2, \ldots$ of a BFS traversal. Our contribution is to develop a cache-obliviously version of this data structure.

3.1 The Data Structure

To construct the data structure, we first build a spanning tree for G and then construct an Euler tour for the tree (which is a list containing the edges of the Euler tour in the order they appear in the tour) using the algorithm described in [4]. Next, we assign to each node v the rank in the Euler tour of the first occurrence of the node, which is done by a traversal of the Euler tour and a sorting step. We denote this value as $r(v)$. The Euler tour has length $2V-1$; so $r(v) \in [0; 2V-2]$. A central observation used in [17] as well as in this paper is the following:

Observation 1. *If for two nodes u and v, the values $r(v)$ and $r(u)$ differ by d, then a section of the Euler tour is a path in G of length d that connects u and v; hence, d is an upper bound on the distance between their BFS levels.*

Let $g_0 < g_1 < g_2 < \cdots < g_h$ be an increasing sequence of $h+1$ integers where $g_0 = 1$, $g_{h-1} < 2V-2 \le g_h$, and g_i divides g_{i+1}. We will later consider two specific sequences, namely $g_i = 2^i$ and one for which $g_i = \Theta(2^{(1+\varepsilon)^i})$. For each integer g_i, we can partition the nodes into groups of at most g_i nodes each by letting the k'th group V_{ki} be all nodes v for which $kg_i \le r(v) < (k+1)g_i$. We call a group V_{ki} of nodes a *g_i-node-group* and call its set of adjacency lists $\mathcal{N}(V_{ki})$ a *g_i-edge-group*. Since g_i divides g_{i+1}, the groups form a hierarchy of $h+1$ levels, with level h containing one group with all nodes and level 0 containing $2E-1$
1) figure groups of at most one node.-1- Note that there is no strong relation between g_i and the number of nodes in a g_i-node-group or the number of edges in a g_i-edge-group. In particular, g_i-edge-groups may be much larger than g_i.

The data structure consists of h levels $G_1, \ldots, G_h$, where each level stores a subset of the adjacency lists of the graph G. Each adjacency list $\mathcal{N}(v)$ will appear

[2] As shorthand for $\mathcal{N}(\{v\})$ we will use $\mathcal{N}(v)$.

in exactly one of the levels, unless it has been removed from the structure. Recall that the query sets of the BFS-algorithm are the BFS-levels $L_0, L_1, L_2, \ldots$; so each node v is part of a query set S from the BFS-algorithm exactly once. Its adjacency list $\mathcal{N}(v)$ will leave the data structure when this happens. Initially, all adjacency lists are in level h. Over time, the query algorithm for the data structure moves the adjacency list of each node from higher numbered to lower numbered levels, until the adjacency list eventually reaches level G_1 and is removed.

GetEdgeLists(S,i)
$S' = S \setminus \{v \in V \mid \mathcal{N}(v)$ is stored in level $G_i\}$
if $S' \neq \emptyset$:
    $X =$ GetEdgeLists(S', $i+1$)
    for each g_i-edge-group g in X
        insert g in G_i
for each g_{i-1}-edge-group γ in G_i containing $\mathcal{N}(v)$ for some $v \in S$
    remove γ from G_i
    include γ in the output set

Fig. 1. The query algorithm.

A high-level description of our query algorithm is shown in Figure 1. It is a recursive procedure that takes as input a query set S of nodes and a level number i to query. The output consists of the g_{i-1}-edge-groups stored at level G_i for which the corresponding g_{i-1}-node-group contains one or more nodes in S. The BFS-algorithm will query the data structure by calling GetEdgeLists(S, 1), which will return $\mathcal{N}(v)$, for all v in S. (Recall that $g_0 = 1$; so non-empty g_0-edge-groups are single edge lists). -2- 2) discussion of correctness

Next we describe how we represent a level G_i so that GetEdgeLists can be performed efficiently. First of all, it follows from Figure 1 that at any time, the edges stored in level G_i constitute a set of g_i-edge-groups from each of which zero or more g_{i-1}-edge-groups have been removed. Since g_{i-1} divides g_i, the part of a g_i-edge-group residing at level G_i is a collection of g_{i-1}-edge-groups.

We store the adjacency lists of level G_i in an array B_i. Each g_{i-1}-edge-group of the level is stored in consecutive locations of B_i, and the adjacency lists $\mathcal{N}(v)$ of a group occupy these locations in order of increasing ranks $r(v)$. The g_{i-1}-edge-groups of each g_i-edge-group are also stored in order of increasing ranks of the nodes involved, but empty location may exist between the g_{i-1}-edge-groups. However, the entire array B_i has a number of locations which is at most a constant times the number of edges it contains. This will require B_i to shrink and grow appropriately. The arrays B_1, B_2,..., B_h will be laid out in $\mathcal{O}(E)$ consecutive memory locations. For purposes of exposition, we defer the discussion of how this shrinking and growing can be achieved at sufficiently low cost while maintaining this layout.

Note that there are no restrictions on the order in which the g_i-edge-groups appear in among each other in B_i. To locate these groups, we keep an index A_i at every level, which is an array of entries (k, p), one for every g_i-edge-group present in G_i. The value k is the number of the corresponding g_i-node-group V_{ki}, and p is a pointer to the start of the g_i-edge-group in B_i. The entries of A_i are sorted by their first components. The indexes A_1, A_2,..., A_h occupy consecutive locations of one of two arrays A' and A'' of size $\mathcal{O}(V)$.

Finally, every g_i-edge-group g of a level will contain an index of the g_{i-1}-edge-groups it presently contains. This index consists of the first and last edge of each g_{i-1}-edge-group γ together with pointers to the first and last locations of the rest of γ. These edges are kept at the front of g, in the same order as the g_{i-1}-edge-groups to which they belong. For simplicity, we assume that the data type for an edge has room for such a pointer.

We now describe how each step of GetEdgeLists is performed. We assume that every query set S of nodes is sorted by assigned rank $r(v)$, which can be ensured by sorting the initial query set before the first call. In Line 1 of the algorithm, we find S' by simultaneously scanning S and A_i, using that, if (k, p) is an entry of A_i, all nodes $v \in S$ for which $kg_i \leq r(v) < (k+1)g_i$ will have $\mathcal{N}(v)$ residing in the g_i-edge-group pointed to by p (otherwise, $\mathcal{N}(v)$ would have been found at an earlier stage of the recursion). In other words, S' is the subset of S not covered by the entries in A_i.

In line 5, when a g_i-edge-group g is to be inserted into level G_i, the index of the g_{i-1}-edge-groups of g is generated by scanning g, and g (now with the index) is appended to B_i. An entry for A_i is generated. When the for-loop in Line 4 is finished, the set of new A_i entries are sorted by their first components and merged with the current A_i. Specifically, the merging writes A_i to A'' if A_i currently occupies A', and vice versa, implying that the location of the entire set of A_i's alternates between A' and A'' for each call GetEdgeLists(S, 1).

In Line 6, we scan S and the updated A_i to find the entries (k, p) pointing to the relevant g_i-edge-groups of the updated B_i. During the scan, we generate for each of these groups g each of the g_{i-1}-edge-groups γ inside g that contains one or more nodes from S, a pair (v, p), where v is the first node in the group. These pairs are now sorted in reverse lexicographic order (the second component is most significant), so that the g_i-edge-groups can be accessed in the same order as they are located in B_i. For each such group g, we scan its index to find the relevant g_{i-1}-edge-groups and access these in the order of the index. Each g_{i-1}-edge-group is removed from its location in B_i (leaving empty positions) and placed in the output set. We also remove its entry in the index of g. The I/O-bound of this process in the minimum I/O-bound of a scan of B_i and a scan of each of the moved g_{i-1}-edge-groups.

3.2 Analysis

In the following we analyze the number of I/Os performed by our cache-oblivious BFS-algorithm (assuming that the management of the layout of the B_i's can be done efficiently, which will be easier to discuss once we have the analysis of the main structure available).

The basic BFS-algorithm from [20] scans each BFS-level L_i twice: once while constructing L_{i+1} and once while constructing L_{i+2}, causing a total of $\mathcal{O}(V/B)$ I/Os for all lists L_i. Edges extracted from the data structure storing the adjacency lists are sorted once and scanned once for filtering out duplicates and already discovered nodes, causing a total of $\mathcal{O}(\mathrm{Sort}(E) + E/B)$ I/Os. It follows that the basic algorithm requires a total of $\mathcal{O}(\mathrm{Sort}(E))$ I/Os.

We now turn to the I/Os performed during queries of the data structure storing the adjacency lists. The cost for constructing the initial spanning tree is $\mathcal{O}(\mathrm{MST}(E))$; the Euler tour can be constructed in $\mathcal{O}(\mathrm{Sort}(V))$ I/Os [4]. Assigning ranks to nodes and labelling the edges with the assigned ranks requires further $\mathcal{O}(\mathrm{Sort}(E))$ I/Os. The total preprocessing of the data structure hence costs $\mathcal{O}(\mathrm{MST}(E) + \mathrm{Sort}(E))$ I/Os.

For each query from the basic BFS-algorithm, the query algorithm for the data structure accesses the A_i and B_i lists. We first consider the number of I/Os for handling the A_i lists. During a query, the algorithm scans each A_i list at most a constant number of times: to identify which g_i-edge-groups to extract recursively from B_{i+1}; to merge A_i with new entries extracted recursively; and to identify the g_{i-1}-edge-groups to extract from B_i. The number of distinct g_i-edge-groups is $2V/g_i$. Each group is inserted into level G_i at most once. By Observation 1, when a g_i-edge-group is inserted into level G_i, it will become part of an initial query set S within g_i queries from the basic BFS-algorithm, that is, within the next g_i BFS-levels; at this time, it will be removed from the structure. In particular, it will reside in level G_i for at most g_i queries. We conclude that the total cost of scanning A_i during the run of the algorithm is $\mathcal{O}((2V/g_i) \cdot g_i/B)$, implying a total number of $\mathcal{O}(h \cdot V/B)$ I/Os for scanning all A_i lists. This bound holds because the A_i's are stored in consecutive memory locations, which can be considered to be scanned in a single scan during a query from the basic BFS-algorithm. Since each node is part of exactly one query set of the basic BFS-algorithm, the total I/O cost for scanning the S sets during all recursive calls is also $\mathcal{O}(h \cdot V/B)$.

We now bound the sorting cost caused during the recursive extraction of groups. The pointer to each g_i-edge-group participates in two sorting steps: When the group is moved from level $i+1$ to level i, the pointer to the group participates in the sorting of A_{i+1} before accessing B_{i+1}; when the g_i-edge-group has been extracted from B_{i+1}, the pointer is involved in the sorting of the pointers to all extracted groups before they are merged into A_i. We conclude that the total sorting cost is bounded by $\mathcal{O}(\sum_{i=0}^{h} \mathrm{Sort}(2V/g_i))$ which is $\mathcal{O}(\mathrm{Sort}(V))$, since g_i is at least exponentially increasing for both of the two sequences considered.

Finally, we need to argue about the I/O cost of accessing the B_i lists. For each query of the basic BFS-algorithm, these will be accessed in the order $B_h, B_{h-1}, \ldots, B_1$. Let t be an integer, where $1 \leq t \leq h$. The cost of accessing $B_t, \ldots, B_1$ during a query is bounded by the cost of scanning $B_t, \ldots, B_1$. Since an edge in B_i can only remain in B_i for g_i queries from the basic BFS-algorithm, we get a bound on the total I/O cost for $B_1, \ldots, B_t$ of $\mathcal{O}(\sum_{i=1}^{t} g_i \cdot E/B)$, which is $\mathcal{O}(g_t \cdot E/B)$ since g_i is at least exponentially increasing.

To bound the cost of accessing $B_h, \ldots, B_{t+1}$, we note that the number of I/Os for moving a g_i-edge-group list containing k edges from B_{i+1} to B_i is bounded by $\mathcal{O}(1 + k/B + g_{i+1}/(g_i B))$, where g_{i+1}/g_i is the bound of the size of the index of a g_{i+1}-edge-group. Since the number of g_i-edge-groups is bounded by $2V/g_i$, it follows that the I/O cost for accessing $B_h, \ldots, B_{t+1}$ is bounded by $\mathcal{O}(\sum_{i=t}^{h-1}(2V/g_i + E/B + (2V/g_i) \cdot g_{i+1}/(g_i B)) = \mathcal{O}(V/g_t + h \cdot E/B)$, since $g_{i+1} \leq g_i^2$ for both of the two sequences considered. The total cost of accessing all B_i is, hence, $\mathcal{O}(g_t \cdot E/B + V/g_t + h \cdot E/B)$, for all $1 \leq t \leq h$.

Adding all the bounds above gives a bound of $\mathcal{O}(\mathrm{MST}(E) + \mathrm{Sort}(E) + g_t \cdot E/B + V/g_t + h \cdot E/B)$ on the total number of of I/Os incurred by the query algorithm, for all $1 \leq t \leq h$.

For $g_i = 2^i$, we select $g_t = \Theta(\sqrt{VB/E})$ and have $h = \Theta(\log V)$, so the I/O-bound becomes $\mathcal{O}(\mathrm{ST}(E) + \mathrm{Sort}(E) + \frac{E}{B}\log V + \sqrt{VE/B})$. For $g_i = \Theta(2^{(1+\varepsilon)^i})$, we select the smallest $g_t \geq \sqrt{VB/E}$, i.e. $g_t \leq \sqrt{VB/E}^{1+\varepsilon}$, and have $h = \Theta(\frac{1}{\varepsilon}\log\log V)$, so the I/O-bound becomes $\mathcal{O}(\mathrm{ST}(E) + \mathrm{Sort}(E) + \frac{E}{B} \cdot \frac{1}{\varepsilon} \cdot \log\log V + \sqrt{VE/B} \cdot \sqrt{VB/E}^{\varepsilon})$.

3.3 Layout Management

In this section, we describe how to efficiently maintain a layout of $B_1, \ldots, B_h$ in a single array of size $2E$. We will maintain a sequence of pointers $0 = p_1 \leq s_1 \leq p_2 \leq s_2 \leq p_3 \leq \cdots \leq p_h \leq s_h \leq p_{h+1} = 2E$ into the array, where $p_2 \geq 2|B_1|$ and $p_{i+1} - p_i = 2|B_i|$ for $1 \leq i \leq h$. Here, $|B_i|$ denotes the total number of edges stored in B_i.

We maintain the invariant that all adjacency lists within B_i are stored within positions $[s_i; p_{i+1})$. The g_{i-1}-edge-groups will always be stored as a consecutive list of positions within B_i; but between the individual g_{i-1}-edge-groups, there can be unused array positions. In Figure 2, we show the memory layout of the sets $B_1, \ldots, B_h$. Each of the shaded areas within B_i represents a g_{i-1}-edge-group list.

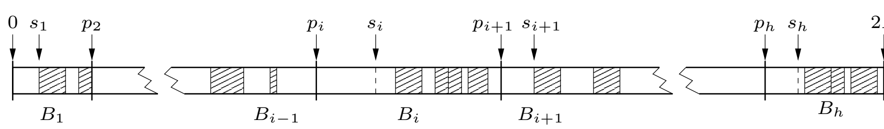

Fig. 2. The memory layout of $B_1, \ldots, B_h$

In the initial state $0 = p_1 = s_1 = p_2 = s_2 = \cdots = p_h$, $s_h = E$ and $p_{h+1} = 2E$, and all adjacency lists are contained within B_h are stored consecutively within array positions $[E; 2E)$.

During a query, g_{i-1}-edge-groups will be moved from B_i to B_{i-1}, leaving unused array positions in B_i. When moving a total of x edges from B_i to B_{i-1}, we place these consecutively at array positions $[p_i; p_i + x)$. We then increase p_i by $2x$ to maintain that $p_{j+1} - p_j = 2|B_j|$ for all j. If this makes p_i larger than s_i, we compress B_i by scanning $[s_i; p_{i+1})$, moving B_i into the consecutive positions $[p_{i+1} - |B_i|; p_{i+1})$, and setting $s_i = p_{i+1} - |B_i|$. Since for the previous value of s_i, we had $s_i < p_i = p_{i+1} - 2|B_i|$, the value of s_i has increased by at least half of the scanned area. Charging the scanning cost to the advance of s_i, and using that each of the s_i, for $i = 1, 2, \ldots, h$, can increase by $2E$ in total, this cost is bounded by $\mathcal{O}(hE/B)$ I/Os in total. Since each compression is triggered by the movement of x edges into $[p_i; p_i + x)$, and the area to be scanned starts below $p_i + 2x$, the single I/O initiating the scan can be included in the cost of moving the x edges, a cost which was bounded in the previous section.

After the compression of B_i, we will need to update A_i. This can be done efficiently if we add another index C_i to the structure. For level G_i, index C_i stores the pairs (k, p) previously discussed for the A_i array, but stores them in the same order as the g_i-edge-groups appear in B_i. The pointers in the pairs in A_i now point to the entries with the same k value in C_i. In short, information about the permutation of the g_i-edge-groups is stored in A_i, while information about their locations is stored in C_i. The C_i index can be updated during compression of B_i using a scan (whereas updating A_i in the absence of C_i would require sorting). Since C_i and A_i have the same size and the C_i's are accessed in the order $h, h-1, \ldots, 2, 1$ during a query, the analysis of the cost of accessing the A_i's from the previous section applies also to the C_i's.

In summary, the desired layout of the B_i's can be achieved without changing the I/O-bounds from the previous section.

In the described layout, we have chosen to store $B_0, \ldots, B_h$ in an array of size $2E$. We could also have chosen a smaller array of size $(1 + \varepsilon)E$, for some constant $\varepsilon > 0$, such that $p_{i+1} - p_i = (1+\varepsilon)|B_i|$. The result would be an increased number of compression steps, implying a factor $1/\varepsilon$ more I/Os for the repeated compression of the B_i's.

References

1. J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.
3. L. Arge. External memory data structures. In *Proc. 9th Annual European Symposium on Algorithms*, volume 2161 of *LNCS*, pages 1–29. Springer Verlag, 2001.
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. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276. ACM Press, 2002.
5. L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proc. 8th Scandinavian Workshop on Algorithmic Theory*, volume 1851 of *LNCS*, pages 433–447. Springer Verlag, 2000.

6. G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 426–438. Springer Verlag, 2002.
7. G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer Verlag, 2002.
8. G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 307–315. ACM Press, 2003.
9. A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860. ACM-SIAM, 2000.
10. Y. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149. ACM-SIAM, 1995.
11. 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*, 2004. To appear.
12. E. D. Demaine. Cache-oblivious data structures and algorithms. In *Proc. EFF summer school on massive data sets*, LNCS. Springer Verlag, 2004, to appear.
13. E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.
14. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.
15. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.
16. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th Symposium on Parallel and Distrib. Processing*, pages 169–177. IEEE Computer Society Press, 1996.
17. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 723–735. Springer Verlag, 2002.
18. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer Verlag, 2003.
19. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. 11th Annual European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 434–445. Springer Verlag, 2003.
20. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.
21. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.