Chapter 3

Cache-oblivious Buffer Heap and its Applications

The distance is nothing;
it is only the first step that is difficult.
(Marie Anne du Deffand)

In this chapter we present the *buffer heap*, a cache-oblivious priority queue that supports *Delete*, *Delete-Min*, and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{M}\right)$ amortized block transfers from main memory, where M and B are the (unknown) cache size and block-size, respectively, and N is the number of elements in the queue. We assume that the *Decrease-Key* operation only verifies that the element does not exist in the priority queue with a smaller key value, and hence it also supports the *Insert* operation in the same amortized bound. The amortized time bound for each operation is $\mathcal{O}(\log N)$.

Using the buffer heap we present cache-oblivious algorithms for undirected and directed single-source shortest path (SSSP) problems for graphs with non-negative real edge-weights. On a graph with n vertices and m edges, our algorithm for the undirected case performs $\mathcal{O}\left(n+\frac{m}{B}\log_2\frac{n}{M}\right)$ block transfers and for the directed case performs $\mathcal{O}\left(\left(n+\frac{m}{B}\right)\cdot\log_2\frac{n}{B}\right)$ block transfers. Running time of both algorithms is $\mathcal{O}\big((m+n)\cdot\log n\big)$.

For both priority queues with *Decrease-Key* operation, and for SSSP problems on general graphs, our results give the first non-trivial cache-oblivious bounds. Our results, though not known to be optimal, provide substantial improvements over known trivial bounds.

We also introduce the notion of a *slim data structure* which captures the situation when only a limited portion of the cache which we call a *slim cache*, is available to the data structure to retain data between data structural operations. We show that a buffer heap automatically adapts to such an environment and supports all operations in $\mathcal{O}\left(\frac{1}{\lambda}+\frac{1}{B}\log_2\frac{N}{\lambda}\right)$ amortized block transfers each when the size of the slim cache is λ. We use buffer heaps in this setting to improve the cache complexity of the cache-aware all-pairs shortest path (APSP) problem on weighted undirected graphs.

3.1 Introduction

The single-source shortest path (SSSP) and the all-pairs shortest path (APSP) problems are among the most important combinatorial optimization problems with numerous practical applications (see Chapter 1 for definitions). Under the traditional *von Neumann Model* of computation which assumes a single layer of memory with uniform access cost, the SSSP problem on a directed graph can be solved efficiently in $\mathcal{O}(m + n \log n)$ time by Dijkstra's algorithm [43] implemented using a *Fibonacci heap* [51]. For undirected graphs the problem can also be solved in $\mathcal{O}(m\alpha(m, n) + n \min(\log n, \log\log \rho))$ time [99], where ρ is the ratio of the maximum and the minimum edge-weights in G, and $\alpha(m, n)$ is a certain natural inverse of Ackermann's function that evaluates to a small constant for all practical values of m and n. Faster algorithms exist for special classes of graphs and graphs with restricted edge-weights. Efficient APSP algorithms have also been developed for this model [136].

As explained in Chapter 1, modern computers with deep memory hierarchies differ significantly from the original von Neumann architecture, and demand cache-efficient algorithms.

3.1.1 Cache-aware Shortest Path Algorithms

In recent years there has been considerable research on developing cache-efficient graph algorithms (see [127, 77] for recent surveys). Several cache-efficient SSSP algorithms have been developed [31, 83, 77, 89]. As explained in Section 2.1.3 of Chapter 2, in addition to a mechanism to remember visited vertices, cache-efficient implementations of virtually all SSSP algorithms require cache-efficient priority queues supporting *Decrease-Key* operations.

Major known SSSP results for the two-level I/O model are summarized in Table 3.3 under the caption "Cache-aware Results". Kumar & Schwabe [83] were the first to develop a cache-efficient version of Dijkstra's SSSP algorithm for undirected graphs. They use a tournament tree as a priority queue and perform some extra book-keeping using an auxiliary priority queue in order to handle visited vertices. A cache-efficient tournament tree supports a sequence of k *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{k}{B}\log_2\frac{n}{M}\right)$ block transfers leading to an SSSP algorithm incurring $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$ cache-misses. The *phase approach* used in [31] implements a priority queue with *Decrease-Key*s indirectly and results in an undirected SSSP algorithm that beats Kumar & Schwabe's algorithm when $n = \mathcal{O}\left(M\log_2\frac{n}{M}\right)$, i.e., the set of vertices is not too large compared to the size of the cache. In [89] Meyer & Zeh developed another undirected SSSP algorithm that works on graphs with real edge-weights, but its performance depends on ρ, the ratio of the largest and the smallest edge-weights in the graph. This algorithm outperforms Kumar &

Schwabe's algorithm for sparse graphs, i.e., when $m = \mathcal{O}\left(\frac{B}{\log_2 \rho} \cdot n\right)$. This algorithm uses a hierarchical decomposition technique to reduce random accesses to adjacency lists, and a priority queue called the *bucket heap* that is specifically designed for this purpose. The bucket heap supports a sequence of k *Delete*, *Delete-Min* (*Batched-Delete-Min*) and *Decrease-Key* operations in $\mathcal{O}\left(sort(k) + \frac{k}{B}\log_2 \rho\right)$ cache-misses.

For directed graphs the survey paper [127] mentions a cache complexity of $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log_2 \frac{n}{B}\right)$ for SSSP using a tournament tree. Using the phase approach directed SSSP can be solved in $\mathcal{O}\left(n + \frac{mn}{BM}\log_2 \frac{n}{B}\right)$ block transfers [31, 77].

A straight-forward method of computing APSP is to simply run an SSSP algorithm from each of the n vertices of the graph. Arge et al. [13] proposed a cache-aware APSP algorithm for undirected graphs with general non-negative edge-weights that performs $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}\log n} + sort(m)\right)\right)$ block transfers when $m = \mathcal{O}\left(\frac{B}{\log n} \cdot n\right)$. They use a priority queue structure called the *multi-tournament-tree* which is created by bundling together a number of cache-efficient tournament trees. The use of this structure reduces unstructured accesses to adjacency lists at the expense of increasing the cost of each priority queue operation.

3.1.2 Cache-oblivious Shortest Path Algorithms

The cache-oblivious priority queue introduced by Arge et al. [11] and the *funnel heap* introduced by Brodal & Fagerberg [22] support *Insert* and *Delete-Min* in amortized optimal $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$ cache-misses, where N is the number of elements in the queue, but they do not support *Decrease-Keys*. Prior to our work no non-trivial cache-oblivious results were known for priority queue with *Decrease-Key*s or for SSSP on graphs. Very recently, however, Allulli et al. [7] obtained a cache-oblivious SSSP algorithm for undirected sparse graphs with bounded edge-weights by extending the cache-aware algorithm in [89] which outperforms our algorithm when $m = \mathcal{O}\left(\frac{B}{\log_2 \rho} \cdot n\right)$ and $\rho = 2^{o(B)}$, where ρ is the ratio of the largest and the smallest edge-weights.

I/O Model	Priority Queue	*Decrease-Key*	*Delete*	*Delete-Min*
Cache-aware	Tournament Tree [83]	$\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$		
Cache-oblivious	Buffer Heap **(our result)** (see also [24])	$\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$		

Table 3.1: Amortized cache complexities for priority queues with *Decrease-Keys*. (N = number of items in the queue)

I/O Model	Slim Priority Queue	*Decrease-Key*	*Delete*	*Delete-Min*
Cache-aware	Slim Tournament Tree $\left[1 \leq \lambda \leq \frac{B}{2}\right]$ (component of multi-tournament-tree [13])	$\mathcal{O}\left(\frac{1}{\lambda}\log_2 N\right)$		
Cache-oblivious	Slim Buffer Heap $[1 \leq \lambda \leq M]$ **(our result)**	$\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$		

Table 3.2: Amortized cache complexities for slim priority queues with *Decrease-Keys*. (λ = slim cache size, N = # items)

3.1.3 Our Results

Majority of the results included in this chapter were presented in two conference papers [32, 33].

We introduce the *buffer heap*, the first cache-oblivious priority queue to support *Decrease-Key* operations. Independently of our work a similar data structure was also presented in [24]. The buffer heap matches the cache complexity of the cache-aware tournament tree (see Table 3.1), and we use it to obtain the first cache-oblivious SSSP algorithms for weighted undirected and directed graphs matching the cache performance of their cache-aware counterparts (see Table 3.3). Our cache-miss bounds for SSSP problems are not very impressive for sparse graphs, but they do provide dramatic improvements for moderately dense graphs. For example, for undirected graphs, if $m \geq \frac{nB}{\log_2\left(\frac{n}{B}\right)}$ our algorithm reduces the number of cache-misses by a factor of $\frac{B}{\log_2\left(\frac{n}{B}\right)}$ over the naïve method. For directed graphs, we obtain the same improvement if $m \geq nB$.

We also introduce the notion of a *slim data structure*. This notion captures the scenario where only a limited portion of the cache 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 cache is available for the computation. We describe and analyze the *slim buffer heap* which is a slim data structure based on the buffer heap (see Table 3.2 for a comparison with the only other similar data structure known), and use it to improve the cache performance of the cache-aware APSP algorithm for undirected graphs with general non-negative edge-weights given in [13] to $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ when $m = \mathcal{O}\left(\frac{nB}{\log^2 n}\right)$ (see Table 3.3). Recall that $sort(m)$ is the cache complexity of sorting m data items. For general values of m our algorithm performs $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + \frac{m}{B}\log\frac{m}{B}\right)\right)$ block transfers. We also believe that the notion of a slim data structure is of independent interest.

In this chapter we show that the slim buffer heap can be made oblivious of the slim cache size without sacrificing its performance. In fact, we show that when a regular buffer heap (i.e., a buffer heap which is not restricted to using a slim cache) is run in an environment that limits the amount λ of cache space available to it to store data between data structural operations, it automatically adapts to this new environment and matches the performance bounds of a slim buffer heap with a slim cache of size λ.

Problem	Cache-aware Results	Cache-oblivious Results
Weighted Undirected SSSP	$\mathcal{O}\left(n + \frac{m}{B}\log_2 \frac{n}{M}\right)$ [83] $\mathcal{O}\left(n + \frac{mn}{BM} + sort(m)\right)$ [31, 77] $\mathcal{O}\left(\sqrt{\frac{mn}{B}\log_2 \rho} + sort(m+n)\log_2\log_2 \frac{nB}{m}\right)$ [89]	$\mathcal{O}\left(n + \frac{m}{B}\log_2 \frac{n}{M}\right)$ **(our result)** (see also [24])
Weighted Directed SSSP	$\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot \log_2 \frac{n}{B}\right)$ [127] $\mathcal{O}\left(n + \frac{mn}{BM}\log_2 \frac{n}{B}\right)$ [31, 77]	$\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot \log_2 \frac{n}{B}\right)$ $\left[M = \Omega\left(B^2\right)\right]$ **(our result)**
Weighted Undirected APSP	$\mathcal{O}\left(n\cdot\left(\sqrt{\frac{mn}{B}\log_2 n} + sort(m)\right)\right)$ [13] $\mathcal{O}\left(n\cdot\left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ **(our result)**	$\mathcal{O}\left(n\cdot\left(n + \frac{m}{B}\log_2 \frac{n}{M}\right)\right)$ **(derived from our undirected SSSP result above)**

Table 3.3: Cache complexities for SSSP and APSP problems on weighted graphs. ($n = |V|$, $m = |E|$)

3.1.4 Organization of the Chapter

In Section 3.2, we define a slim data structure. In Section 3.3, we present the cache-oblivious buffer heap as a slim data structure, prove the correctness of its implementation and analyze its cache and time complexities. In Section 3.4, we discuss three major applications of buffer heap. In Sections 3.4.1 and 3.4.2 we use the buffer heap to obtain cache-oblivious SSSP algorithms for weighted undirected and directed graphs, respectively. In Section 3.4.3 we describe the application of buffer heap in obtaining an improved cache-aware APSP algorithm for weighted undirected graphs. Finally, we present some concluding remarks in Section 3.5.

3.2 Slim Data Structures

A *slim data structure* is a data structure with a fixed-size footprint in the cache. The area in the cache that holds the footprint is called the *slim cache*. By $DS(\lambda)$ we denote a data structure DS, in which a portion of size $\Theta(\lambda)$ 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 cache is M and **(b)** data is transferred between the cache and the main memory in blocks of size B. Thus $1 \leq \lambda \leq M$; and the data structural operations must assume that the portion of the data structure that is not stored in the slim cache is stored in a main memory divided into blocks of size B, and thus accessing anything outside the slim cache may cause cache-misses. While executing a data structural operation the operation can use all free cache space for temporary computation, but after the operation completes only the data in the slim cache is preserved for reuse by the next operation on the data structure.

Some existing data structures can be viewed trivially as slim data structures. For example, Arge et al. [13] analyzed each component tournament tree of the *multi-tournament-tree* as supporting *Decrease-Key*, *Delete* and *Delete-Min* operations in $\mathcal{O}\left(\frac{1}{\lambda}\log N\right)$ amortized cache-misses each for $1 \leq \lambda \leq \frac{B}{2}$; this can be viewed as a slim data structure for this range of values for λ.

Although our main motivation behind introducing the notion of slim data structures was to obtain the APSP result in Section 3.4.3, we believe that the need for slim data structures could arise in other applications. A typical application would be one in which a number of data structures need to be kept in the cache simultaneously, and thus only a limited portion of the cache can be dedicated to each data structure.

In the next section we present our cache-oblivious buffer heap, and analyze its performance as a slim data structure.

3.3 The Buffer Heap

In this section we present the *Buffer Heap*, a cache-oblivious priority queue that supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$ amortized cache-misses each, where N is the number of items in the priority queue. A *Delete*(x) operation deletes element x from the queue if it exists and a *Delete-Min*$()$ operation retrieves and deletes an element with the minimum key from the queue. A *Decrease-Key*(x, k_x) operation inserts the element x with key k_x into the queue if x does not already exist in the queue, otherwise it replaces the smallest key k'_x of x in the queue with k_x provided $k_x < k'_x$, and deletes all remaining keys of x in the queue. For simplicity of exposition, we assume that all keys in the data structure are distinct.

When analyzed as a slim data structure with a slim cache of size λ, we show that a buffer heap supports each of its three operations in $\mathcal{O}\left(\frac{1}{\lambda}+\frac{1}{B}\log_2\frac{N}{\lambda}\right)$ amortized cache-misses. The buffer heap, however, remains oblivious of the parameter λ; the external application using the data structure may choose to maintain a slim cache, i.e., impose a restriction on the value of λ. When a buffer heap is restricted to use a slim cache, we call it a *Slim Buffer Heap* and denote it by $SBH(\lambda)$, otherwise we call it a *Regular Buffer Heap*. Note that since a buffer heap is not aware of the existence of a slim cache, both types of buffer heap (slim and regular) have exactly the same implementation, the only difference is in their analysis. A regular buffer heap can be viewed as a slim buffer heap with a slim cache of size $\lambda = \Theta(M) = \Omega(B)$.

A regular buffer heap matches the cache complexity of a tournament tree [83], its only cache-aware counterpart that supports the same operations. It has been shown in [13] that a slim version of the tournament tree (a component of the multi-tournament-tree introduced in [13]) supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{\lambda}\log N\right)$ amortized cache-misses each when restricted to use a slim cache of size $\lambda \in \left[1, \frac{B}{2}\right]$. Hence, a slim buffer heap improves over the cache complexity of a slim tournament tree.

3.3.1 Structure

A buffer heap on N items consists of $r = 1 + \lceil \log_2 N \rceil$ levels. For $0 \le i \le r-1$, 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 or operation in U_i is augmented with a timestamp indicating the time of its insertion into the data structure.
At any time, the following invariants are maintained:

Invariant 3.3.1.

(*a*) *Each* B_i $(0 \le i < r)$ *contains at most* 2^i *elements.*
(*b*) *Each* U_i $(0 \le i < r)$ *contains at most* 2^i *updates.*

Invariant 3.3.2.

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

(*b*) *All updates applicable to* B_i $(0 \le i < r-1)$ *that are not yet applied, reside in* $U_0, U_1, \ldots, U_i$.

Invariant 3.3.3.

(*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 timestamp.*

All buffers are initially empty.

3.3.2 Layout

The element buffers are stored in a stack S_B with elements of B_i placed above elements of B_j for all $i < j$. Elements of the same B_i occupy contiguous space in the stack with an element (x_1, k_1) stored above another element (x_2, k_2) provided $x_1 < x_2$. Similarly, update buffers are placed in another stack S_U where updates in any U_i are stored above those in all U_j with $j > i$. Updates in a single buffer occupy a contiguous region in the stack. For $0 \leq i \leq r-1$, the segments of U_i are stored one above another in the stack, and updates in each segment are stored sorted from top to bottom first by element id and then by timestamp. An array A_s of size r 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.

The buffer heap uses $\mathcal{O}(N)$ space.

3.3.3 Operations

In this section we describe how *Delete*, *Delete-Min* and *Decrease-Key* operations are implemented.

A *Decrease-Key* operation is performed by the DECREASE-KEY function (i.e., Function 3.3.1) which inserts it into U_0 augmented with the current timestamp. Further processing is deferred to the next *Delete-Min* operation except that the FIX-U function may be called to restore invariant 3.3.1(b) (i.e., overflowing update buffers) for the structure. A *Delete* operation is performed by the DELETE function (i.e., Function 3.3.2) in exactly the same way.

The FIX-U function uses a function called APPLY-UPDATES. When called with a parameter i, APPLY-UPDATES (i.e., Function 3.3.5) applies the updates in U_i on the elements of B_i, and empties U_i by moving the updates from U_i to U_{i+1}. It also moves any overflowing elements from B_i to U_{i+1} as *Sink* operations. A *Sink*(x, k_x) operation is used to move an element (x, k_x) from B_i to B_{i+1} through U_{i+1}.

The FIX-U function (i.e., Function 3.3.6) is called with parameter i when U_i overflows. This function starts at level i and continues calling APPLY-UPDATES on each successive level until it reaches a level j such that U_{j+1} does not overflow when APPLY-UPDATES(j) completes execution. It collects all elements left in $B_i, B_{i+2}, \ldots, B_j$ in a temporary buffer B' and returns B' leaving these element buffers empty.

Every call to FIX-U is followed by a call to the REDISTRIBUTE function (i.e., Function 3.3.7) which redistributes the elements returned by FIX-U to the shallowest element buffers.

The DELETE-MIN function (i.e., Function 3.3.3) executes a *Delete-Min* operation by first calling the FIND-MIN function to find an element with the minimum key in the data structure, and then calling the DELETE function to delete this element.

The FIND-MIN function (i.e., Function 3.3.4) works by finding the shallowest element buffer B_k that is left non-empty after applying the updates in U_k (by calling APPLY-UPDATES). The FIX-U function is then called to fix overflowing update buffers, if any. The elements left in B_k along with the elements returned by FIX-U are distributed to the shallowest element buffers by calling REDISTRIBUTE.

After each operation the RECONSTRUCT function (i.e., Function 3.3.8) is called. This function reconstructs the entire data structure periodically. It remembers the number of elements N_e in the structure immediately after the last reconstruction, and keeps track of the number of new operations N_o performed since then. Initially N_e is set to 0. When $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$, the data structure is rebuilt by calling APPLY-UPDATES for each level, emptying the update buffers and distributing the remaining elements to the shallowest possible levels. The objective of the function is to ensure that the number of levels r in the structure is always within ± 1 of $\log_2 N$, where N is the current number of elements in the structure. This invariant is maintained because r can decrease by at most 1 since the last reconstruction (this happens if all $\lfloor \frac{N_e}{2} \rfloor + 1$ operations are *Delete* or *Delete-Min* operations), and can increase by at most 1 (if all those operations are *Decrease-Key*s).

Correctness

We prove the correctness of all buffer heap operations below.

Lemma 3.3.1. *Buffer heap correctly supports three external-memory priority queue operations, namely,* Decrease-Key*,* Delete *and* Delete-Min *operations, on its elements.*

Proof. We will prove that the DECREASE-KEY/DELETE function correctly inserts the corresponding *Decrease-Key*/*Delete* operation into the buffer heap, and the DELETE-MIN function correctly extracts the element with the minimum key from the buffer heap, while correctly applying all relevant *Decrease-Key* and *Delete* operations, and maintaining all invariants.

Before proving the correctness of the three functions mentioned above we must establish the correctness of APPLY-UPDATES and FIX-U which are called as subroutines by all of them. The APPLY-UPDATES function is at the core of all buffer heap functionality.

APPLY-UPDATES. When called with parameter i, APPLY-UPDATES applies all updates in U_i on the elements in B_i under the assumption that all invariants hold initially except possibly invariant 3.3.1(b) for U_i. All U_j for $0 \leq j < i$ are assumed to be empty.

FUNCTION **3.3.1.** DECREASE-KEY(x, k_x)

[Inserts a *Decrease-Key* operation into the structure, that decreases the key of element x to k_x. If x does not already exist in the structure, this operation results in the insertion of x with key k_x.**]**

1. insert the operation into U_0 augmented with current timestamp maintaining inv. 3.3.3(b)

2. $B' \leftarrow \emptyset$, $i \leftarrow 0$ {*list B' stores elements returned by* FIX-U}
FIX-U(i, B') {*fix U_i (i.e, restore invariant 3.3.1(b)) in case of overflow*}

3. REDISTRIBUTE(B') {*redistribute elements in B' to shallowest element buffers*}

4. RECONSTRUCT( ) {*reconstruct the data structure periodically*}

DECREASE-KEY ENDS

FUNCTION **3.3.2.** DELETE(x)

[Inserts a *Delete* operation into the structure, that deletes element x from the structure if exists.**]**

Same as Function 3.3.1 (DECREASE-KEY) above

DELETE ENDS

FUNCTION **3.3.3.** DELETE-MIN( ) **[**Extracts element with the smallest key from the structure.**]**

1. $(x, k_x) \leftarrow$ FIND-MIN( ) {*find the element with the minimum key*}

2. ***if*** $k_x \neq +\infty$ ***then*** DELETE(x) {*delete x from the data structure if nonempty*}

3. ***return*** (x, k_x)

DELETE-MIN ENDS

FUNCTION **3.3.4.** FIND-MIN( ) **[**Returns the element with the smallest key in the structure.**]**

1. $i \leftarrow -1$
repeat
(*i*) $i \leftarrow i + 1$
(*ii*) APPLY-UPDATES(i) {*apply the updates in U_i on the elements in B_i*}
until ($|B_i| > 0$) $\vee$ ($i = r - 1$)

2. ***if*** $|B_i| = 0$ ***then*** {*the data structure has become empty*}
(*i*) $(x, k_x) \leftarrow (\_\ , +\infty)$, $r \leftarrow 1$ {*will return $+\infty$ as the minimum key*}

3. ***else*** {*the data structure is nonempty*}
(*i*) $B' \leftarrow B_i$, $i \leftarrow i + 1$
FIX-U(i, B') {*fix U_i (i.e, restore invariant 3.3.1(b)) in case of overflow*}
(*ii*) REDISTRIBUTE(B') {*redistribute elements in B' to shallowest element buffers*}
(*iii*) $(x, k_x) \leftarrow$ the element in B_0 {*B_0 has the element with the minimum key*}

4. ***return*** (x, k_x)

FIND-MIN ENDS

Function **3.3.5.** Apply-Updates(i)

[Applies 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 moves overflowing elements from B_i to U_{i+1} as *Sink*s.

Preconditions: All invariants hold except possibly 3.3.1(b) for U_i. All U_j, $j \in [0, i-1]$ are empty.

Postconditions: All invariants hold except possibly 3.3.1(b) for U_{i+1}. All U_j, $j \in [0, i]$ are empty.]

1. merge the segments of U_i

2. ***if*** ($|B_i| = 0$) $\wedge$ ($i < r-1$) ***then*** {*if i is not the last level and B_i is empty*}

(i) empty U_i by moving the contents of U_i as a new segment of U_{i+1}

3. ***else***

(i) ***if*** $i = r-1$ ***then*** $k \leftarrow +\infty$ ***else*** $k \leftarrow$ largest key in B_i

(ii) scan B_i and U_i simultaneously, and for each $op \in U_i$: {*apply the updates in U_i on B_i*}

(a) ***if*** $op =$ *Delete*(x) ***then*** remove any element (x, k_x) from B_i if exists

(b) ***if*** $op =$ *Decrease-Key*($x,\ k_x$)/*Sink*($x,\ k_x$) ***then***

- replace any $(x, k'_x) \in B_i$ with $(x, \min(k_x, k'_x))$
- copy (x, k_x) to B_i if no (x, k'_x) exists in B_i and $k_x \leq k$

(iii) ***if*** $i < r-1$ ***then*** {*move appropriate updates from U_i to U_{i+1}*}

(a) copy each *Decrease-Key*($x,\ k_x$) in U_i, not applied in step 3(ii)(b) to U_{i+1}

(b) for each *Delete*(x) and each *Decrease-Key*($x,\ k_x$) in U_i that was applied in step 3(ii)(b) copy a *Delete*(x) to U_{i+1}

(iv) ***if*** $|B_i| > 2^i$ ***then*** {*restore invariant 3.3.1(a) if violated*}

(a) ***if*** $i = r-1$ ***then*** $r \leftarrow r+1$

(b) keep the 2^i elements with the smallest 2^i keys in B_i and move each remaining element (x, k_x) to U_{i+1} as *Sink*($x,\ k_x$)

(v) $U_i \leftarrow \emptyset$

Apply-Updates Ends

Function **3.3.6.** Fix-U($i,\ B'$)

[Fixes all overflowing update buffers in levels i and up. Update buffer U_i overflows if $|U_i| > 2^i$ (see invariant 3.3.1(b)). For each overflowing U_i collects contents of B_i in B' after applying U_i on B_i.

Preconditions: All invariants hold except invariant 3.3.1(b) for U_i. All U_j for $0 \leq j < i$ are empty.

Postconditions: All invariants hold. If k is the largest index for which the ***while*** loop in line 1 was executed, then all U_j for $0 \leq j \leq k$ are empty. The contents of all B_j for $i \leq j \leq k$ after applying all applicable updates on them are collected in B' leaving those buffers empty.]

1. ***while*** ($i < r$) $\wedge$ $\left(|U_i| > 2^i \right)$ ***do***

(i) Apply-Updates(i) {*apply the updates in U_i on the elements in B_i*}

(ii) empty B_i by merging it with B' {*collect in B' the elements remaining in B_i*}

(iii) $i \leftarrow i+1$

Fix-U Ends

Function **3.3.7.** Redistribute(B')

[Distributes the elements in B' to the shallowest element buffers maintaining invariants 3.3.1(a), 3.3.2(a) and 3.3.3(a).

Preconditions: All invariants hold. All B_i and U_i with $0 \le i \le k$ are empty, where k is the smallest integer such that $2^{k+1} - 1 \ge |B'|$. No key value in the data structure is smaller than any key value in B'.

Postconditions: All invariants hold. All update buffers remain unchanged, but $\bigcup_{i=0}^{k} B_i = B'$.]

1. $i \leftarrow$ largest integer such that $2^i - 1 < |B'|$

2. ***while*** $i \ge 0$ ***do***

(i) move $|B'| - 2^i + 1$ elements with the largest $|B'| - 2^i + 1$ keys from B' to B_i maintaining invariant 3.3.3(a)

(ii) $i \leftarrow i - 1$

Redistribute Ends

Function **3.3.8.** Reconstruct( )

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

1. ***if*** $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$ ***then***

(i) $B' \leftarrow \emptyset$

for $i \leftarrow 0$ ***to*** $r - 1$ ***do***

(a) Apply-Updates(i) {*apply the updates in* U_i *on the elements in* B_i}

(b) merge B_i with B' {*collect in* B' *the elements remaining in* B_i}

$B_i \leftarrow \emptyset$

(ii) Redistribute(B') {*redistribute elements in* B' *to shallowest element buffers*}

(iii) $r \leftarrow i$, where i is the largest level such that $|B_i| > 0$

Reconstruct Ends

Observe that since invariant 3.3.2(b) holds initially and for $0 \le j < i$, $|U_j| = 0$, all updates applicable to B_i must reside in U_i. For each element x, this function considers all updates in U_i that are applicable to x in increasing order of timestamp, i.e., in the order in which they were inserted into the data structure. For each such $op \in U_i$ taken in order Apply-Updates does the following.

- $op = Delete(\ x\ )$: If any (x, k_x) exists in B_i it is deleted. If this element did not exist in B_i initially then it must have been inserted into B_i by a *Decrease-Key*(x, k_x)/*Sink*(x, k_x) operation in U_i earlier in the order. Irrespective of whether this *Delete*(x) operation was able to delete an element from B_i or not, it is moved to U_{i+1} without changing its timestamp which ensures that any remaining occurrence of x in the data structure inserted by operations with earlier timestamps is deleted.

• $op = \textit{Decrease-Key}(\ x,\ k_x\ )$: If some (x, k'_x) appears in B_i it is replaced with $(x, \min(k_x, k'_x))$. However, if element x does not appear in B_i, (x, k_x) is inserted into B_i provided $k_x \leq k$, where k is the largest key in B_i ($k = +\infty$ if i is the last level). Observe that if x initially existed in B_i but does not exist now, then it must have been deleted by some *Delete*(x) operation in U_i earlier in the order. Since each *Decrease-Key*(x, k_x) operation that cannot be applied to B_i must have $k_x > k$, it must be applicable to some element buffer in $B_{i+1}, B_{i+2}, \ldots, B_{r-1}$, and so it is moved to U_{i+1} in order to ensure that it is applied to the appropriate element buffer. For each *Decrease-Key*(x, k_x) operation that is applied to B_i, we copy a *Delete*(x) operation with the same timestamp to U_{i+1} so that all occurrences of x in $B_{i+1}, B_{i+2}, \ldots, B_{r-1}$ inserted by *Decrease-Key*(x, k_x) / *Sink*(x, k_x) operations with earlier timestamps are deleted.

• $op = \textit{Sink}(\ x,\ k_x\ )$: If some (x, k'_x) appears in B_i it is replaced with $(x, \min(k_x, k'_x))$, otherwise (x, k_x) is inserted into B_i. Since a *Sink*(x, k_x) operation is used to move element (x, k_x) from B_{i-1} to B_i, we will always have $k_x \leq k$, where k is the largest key in B_i ($k = +\infty$ if i is the last level), and so these updates are not applicable to element buffers in higher levels, i.e., Apply-Updates does not need to carry these updates to U_{i+1}.

Clearly, Apply-Updates never violates invariants 3.3.2 and 3.3.3. However, it can violate invariant 3.3.1(a) if $|B_i| > 2^i$ holds after the updates. It fixes this violation by keeping only the 2^i elements with the smallest 2^i keys in B_i and moving the remaining elements to U_{i+1} as *Sink* operations. Each such overflowing item (x, k_x) is moved to U_{i+1} as a *Sink*(x, k_x) operation with the current timestamp so that existing operations in the data structure cannot prevent this operation from inserting (x, k_x) into B_{i+1}.

Thus after the function terminates all invariants continue to hold except possibly invariant 3.3.1(b) for U_{i+1}. Since all updates from U_i are either moved to U_{i+1} or discarded, $|U_j| = 0$ holds for $j \in [0, i]$.

Fix-U. This function is called with parameter i when U_i overflows. It makes the same assumptions as Apply-Updates. Starting from level i it continues to call Apply-Updates for each level until it reaches a level j such that U_{j+1} does not overflow when Apply-Updates(j) terminates, i.e., the data structure does not have any overflowing update buffers and thus all invariants hold. For $i \leq k \leq j$, this function collects in a temporary buffer B' the contents of each B_j after applying U_j to it leaving B_j empty, and returns B'. The correctness of Fix-U follows directly from the correctness of Apply-Updates.

Decrease-Key(x, k_x)/Delete(x). The function inserts the corresponding *Decrease-Key*(x, k_x) / *Delete*(x) operation into U_0 augmented with the current timestamp so that it is treated by the data structure as the most recent operation. This insertion does not violate any invariants except possibly invariant 3.3.1(b) for U_0, i.e., U_0 overflows. This violation is fixed by calling Fix-U with parameter $i = 0$. Upon return from Fix-U all invariants hold. The set B' of elements returned by Fix-U does not have any key value larger than any key in the data structure, and Fix-U leaves enough empty element buffers at the shallowest possible levels so that the elements in B' can be distributed to those buffers without violating any invariant. The Redistribute function performs this distribution. The Reconstruct function reconstructs the entire data structure periodically. Thus the correctness of Decrease-Key/Delete follows from the correctness of Fix-U, Redistribute and Reconstruct. We have already argued the correctness of Fix-U. The proofs of correctness of Redistribute and Reconstruct are straight-forward and hence are omitted.

Delete-Min( ). The Delete-Min function first calls Find-Min in order to find the element with the minimum key in the entire data structure, and then calls Delete in order to delete this element. We have already argued correctness of Delete, and hence we only need to prove Find-Min correct.

Observe that if invariant 3.3.2 holds, the smallest level k such that B_k is non-empty after applying all updates in $U_0, U_1, \ldots, U_k$ on B_k will contain the element with the smallest key in the entire data structure. The Find-Min function builds on this observation. Starting from level 0 it calls Apply-Updates for each level until it reaches the first level k with $|B_k| \neq 0$ upon return from Apply-Updates. At this point all invariants hold except possibly invariant 3.3.1(b) for U_{k+1}. The overflowing U_{k+1} is fixed by calling Fix-U for level $k+1$. All elements returned by Fix-U along with the contents of B_k are distributed to the shallowest possible element buffers by Redistribute. At this point all invariants hold, B_0 contains exactly one element and U_0 is empty. Therefore, the element in B_0 which is returned by Find-Min is, indeed, the element with the smallest key. ■

Cache Complexity

In this section we will view the buffer heap as a slim data structure with a slim cache of size $\Theta(\lambda)$ and denote it by $SBH(\lambda)$. The slim cache is assumed to be large enough to store $B_0, B_1, \ldots, B_t$ and $U_0, U_1, \ldots, U_{t+1}$, where $t = \log(\lambda + 1) - 1$. The remaining buffers reside in external memory.

The following two observations will be useful in our analyses.

Observation 3.3.1. *For* $i \in [1, r-1]$,

(a) *Each* Sink *operation in* U_i *can be mapped to a unique* Decrease-Key/Sink *operation that existed in* U_{i-1} *but does not exist in* U_i*; and*

(b) U_i *cannot contain more* Sink *operations than* Delete *operations.*

It is not difficult to see that Observation 3.3.1(a) is valid since each *Sink* operation in U_i is generated by an element evicted from B_{i-1} due to overflow, and each eviction from B_{i-1} can be viewed as caused by a unique *Decrease-Key*/*Sink* operation in U_{i-1} that inserted an element into B_{i-1}. After the insertion the responsible *Decrease-Key*/*Sink* operation ceases to exist: if it is a *Decrease-Key* operation it is converted to a *Delete* operation, and if it is a *Sink* operation it is simply discarded. The implication of Observation 3.3.1(a) is that every existing *Sink* operation in the queue can be traced back to a unique *Decrease-Key* operation following a chain of *Sink*s.

We know that the unique *Decrease-Key* operation responsible for the generation of any given *Sink* operation in U_i was converted to a *Delete* at the time it was applied on an element buffer, and it is not difficult to see that this *Delete* operation must now reside in U_i. Thus each *Sink* operation in U_i maps to a unique *Delete* operation in U_i, and Observation 3.3.1(b) follows.

The following lemma which implies that merging the segments of U_i (in line 1 of Apply-Updates) incurs only $\mathcal{O}\left(\frac{1}{B}\right)$ amortized cache-misses per operation in U_i, will be crucial in proving the cache-complexity of buffer heap operations.

Lemma 3.3.2. *For* $1 \leq i \leq r-1$*, every empty* U_i *receives batches of updates at most a constant number of times before* U_i *is applied on* B_i *and emptied again.*

Proof. Since $|U_1| \leq 2$, U_1 cannot receive more than two batches of updates before it overflows, and thus the lemma holds for $i = 1$. Hence, for the rest of proof we will assume $i > 1$.

Update buffer U_i receives at most two batches of updates whenever the execution of a Decrease-Key/Delete/Delete-Min function reaches level $i-1$. If the execution continues and reaches level i then U_i is applied on B_i, and thus emptied. If the execution terminates at level $i-1$ but leaves B_{i-1} empty, the next time an execution reaches level $i-1$ will continue to level i and empty U_i. Therefore, it suffices to consider only executions that terminate at level $i-1$ and leave B_{i-1} nonempty. Let $\mathcal{E}$ be such an execution. We will show that $\mathcal{E}$ increases the number of updates in U_i by at least 2^{i-2} which implies that executions can terminate at level $i-1$ at most four times without emptying B_{i-1} before U_i overflows (since $|U_i| \leq 2^i$) and is thus emptied by Fix-U.

For $j \in [0, r-1]$, let u_j and u'_j denote the number of updates in U_j immediately before the start of $\mathcal{E}$ and immediately after the termination of $\mathcal{E}$, respectively, and let $\delta u_j = u'_j - u_j$. For $j \in [0, i-1]$, we denote by u''_j the number of updates in U_j immediately before $\mathcal{E}$ reaches level j (i.e., $\mathcal{E}$ has already pushed all updates and overflowing elements from level $j-1$ to level j if $j > 0$). Let b'_j ($j \in [0, r-1]$) be the number of elements in B_j immediately after $\mathcal{E}$ terminates. We will prove the following.

$$(\ i > 1\ ) \wedge (\ b'_{i-1} \neq 0\ ) \Rightarrow (\ \delta u_i \geq 2^{i-2}\ ) \qquad (3.3.1)$$

Now in order to establish equation 3.3.1 we consider the following two cases.

Case 1 $\left(u''_{i-1} < 2^{i-1}\right)$**:** Let $\mathcal{E}'$ be the last execution before $\mathcal{E}$ that reached level $i-1$ (and possibly continued to higher levels). Execution $\mathcal{E}$ has reached level $i-1$ because all B_j, $j \in [0, i-2]$ have become empty which were left full by $\mathcal{E}'$. Hence, at least $\sum_{j=0}^{i-2} 2^j = 2^{i-1} - 1$ elements have been deleted from the structure since $\mathcal{E}'$ completed execution, i.e., u''_{i-1} includes at least $2^{i-1} - 1 \geq 2^{i-2}$ *Delete* operations all of which will be moved to U_i and thus $\delta u_i \geq 2^{i-2}$.

Case 2 $\left(u''_{i-1} \geq 2^{i-1}\right)$**:** Since an update buffer cannot contain more *Sink* operations than *Delete* operations (see Observation 3.3.1(b)), u''_{i-1} includes at least $\frac{2^{i-1}}{2} = 2^{i-2}$ *Delete*/*Decrease-Key* operations and thus $\delta u_i \geq 2^{i-2}$.

Hence, equation 3.3.1 and consequently the lemma follow. ■

The following lemma gives the cache complexity of the operations supported by a slim buffer heap:

Lemma 3.3.3. *A slim buffer heap with a slim cache of size* λ *(i.e.,* $SBH(\lambda)$*) supports* Delete, Delete-Min *and* Decrease-Key *operations in* $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$ *amortized cache-misses each, where* N *is the number of elements in the structure.*

Proof. For $0 \leq i \leq r-1$, let u_i be the number of operations in U_i and let d_i be the number of *Decrease-Key* operations among them. By Δ we denote the number of *Decrease-Key*, *Delete* and *Delete-Min* operations performed on the data structure since its last construction/reconstruction. If H is the current state of $SBH(\lambda)$, we define the *potential* of H as follows:

$$\Phi(H) = \sum_{i=0}^{r-1} \left(\frac{1}{B} \cdot (r-i) \;+\; \frac{2}{\lambda} \cdot \frac{1}{2^{\max(i-t,0)}} \right) \cdot (u_i + d_i) \;+\; \left(\frac{r}{B} + \frac{1}{\lambda} \right) \cdot \Delta,$$

where $t = \log(\lambda + 1) - 1$.

As in the original I/O analysis of Buffer Heap operations in [32], the key observation is that operations in update buffers always move downward and at each level they participate in a constant number of scans. The first term under the summation in $\Phi(H)$ captures this flow of data. The main reason for adding the second term is to ensure that after every $\Theta(\lambda)$ new operations enough potential is accumulated to account for the extra cache-miss in accessing data outside the slim cache. Also $\Phi(H)$ has been designed so that the potential gain due to a new *Decrease-Key* operation is more than that for a new *Delete* operation. This uneven distribution of potential is based on the observation that after a *Decrease-Key* operation has been applied successfully on some B_i it turns into a *Delete* operation and possibly generates an additional *Sink* operation in U_{i+1} (see Observation 3.3.1 and its implications). The last term in $\Phi(H)$ gathers potentials for the next reconstruction of the data structure.

We compute the amortized cost of each buffer heap operation below.

Reconstruction. Let us first consider the amortized cost of reconstruction (i.e., the RECONSTRUCT function). At the time of reconstruction $\Delta = \left\lfloor \frac{N_e}{2} \right\rfloor + 1$, where N_e is the number of elements in the structure immediately after the last reconstruction. Thus $\left\lceil \frac{N_e}{2} \right\rceil - 1 \leq \sum_{i=0}^{r-1} |B_i| \leq \left\lfloor \frac{3N_e}{2} \right\rfloor + 1$ implying $\Delta = \Theta\left(\sum_{i=0}^{r-1} |B_i|\right)$. If during the reconstruction operation no buffer outside the slim cache is accessed then no cache-miss occurs. Therefore, we will only consider the case in which some element buffer above level t is accessed. In that case $\Delta = \Omega(\lambda)$.

Accessing the first data outside the slim cache incurs $\mathcal{O}(1)$ cache-misses. The actual cache complexity of APPLY-UPDATES when called with a parameter i in step $1(i)(a)$ of RECONSTRUCT is $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right) = \mathcal{O}\left(\frac{\Delta}{B}\right)$, since the merge operations in step 1 of APPLY-UPDATES can be performed in $\mathcal{O}\left(\frac{|U_i|}{B}\right)$ cache-misses (implied by Lemma 3.3.2); steps $2(i)$, $3(i)$, $3(ii)$ and $3(iii)$ involve a constant number of scans of B_i and U_i incurring $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right)$ cache-misses; and step $3(iv)$ can be performed in $\mathcal{O}\left(\frac{|B_i|}{B}\right)$ cache-misses using a linear I/O selection algorithm [104]. The buffer B_i can be merged with B' in step $1(i)(b)$ of RECONSTRUCT in $\mathcal{O}\left(\frac{|B_i|+|B'|}{B}\right) = \mathcal{O}\left(\frac{\Delta}{B}\right)$ cache-misses. Therefore, the actual cache complexity of step $1(i)$ of RECONSTRUCT is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right)$. The actual cost of the REDISTRIBUTE function in step $1(ii)$ of RECONSTRUCT is $\mathcal{O}\left(\frac{r}{B} \cdot \Delta\right)$ since the ***while*** loop in step 2 of REDISTRIBUTE iterates $\mathcal{O}(r)$ times and in each iteration scans each element of B' at most a constant number of times if a linear I/O selection algorithm is used. Thus the actual cache complexity of reconstruction is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right)$.

Since all update buffers are emptied during reconstruction and $\Delta = \Omega(\lambda)$, the potential drop is $\Omega\left(\left(\frac{1}{\lambda} + \frac{r}{B}\right) \cdot \Delta\right) = \Omega\left(1 + \frac{r}{B} \cdot \Delta\right)$. Thus the amortized cost of reconstruction is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right) - \Omega\left(1 + \frac{r}{B} \cdot \Delta\right) \leq 0$.

Decrease-Key/Delete. The increase in potential due to the insertion of a *Decrease-Key* operation into U_0 is $\frac{5}{\lambda} + \frac{3}{B} \cdot r$, and due to the insertion of a *Delete* operation is $\frac{3}{\lambda} + \frac{2}{B} \cdot r$. If no element buffer of level higher than t is accessed in step 2 of the DECREASE-KEY/DELETE function then no cache-miss occurs (except in the RECONSTRUCT function in step 4 whose amortized cost has already been shown to be ≤ 0). So we only need to consider the case when a B_i with $i > t$ is accessed.

Let j be the largest value of i for which the ***while*** loop in step 1 of FIX-U was executed. The actual cost of APPLY-UPDATES when called with a parameter i in step $1(i)$ of FIX-U is $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right) = \mathcal{O}\left(\frac{2^i}{B}\right)$. The buffer B_i can be merged with B' in step $1(ii)$ of FIX-U in $\mathcal{O}\left(\frac{|B_i|+|B'|}{B}\right) = \mathcal{O}\left(\frac{2^i}{B}\right)$ cache-misses. Therefore, FIX-U incurs at most $\sum_{i=0}^{j}\mathcal{O}\left(\frac{2^i}{B}\right) = \mathcal{O}\left(\frac{2^j}{B}\right)$ cache-misses in total. Also $|B'| = \mathcal{O}\left(2^j\right)$ when FIX-U returns. Hence, the actual number of cache-misses incurred by REDISTRIBUTE in step 3 of DECREASE-KEY/DELETE for redistributing the elements in B' is at most $\mathcal{O}\left(\frac{2^j}{B}\right) + \sum_{i=0}^{j}\mathcal{O}\left(\frac{2^i}{B}\right) = \mathcal{O}\left(\frac{2^j}{B}\right)$ assuming a linear I/O selection algorithm is used. Therefore, including the $\mathcal{O}(1)$ cache-misses incurred for accessing the first data outside the slim cache, the actual cost of steps 1–3 of a *Decrease-Key/Delete* operation is $\mathcal{O}\left(1 + \frac{2^j}{B}\right)$.

Since U_j was full before APPLY-UPDATES was called in step $1(i)$ of FIX-U, the drop of potential due to the movement of these $|U_j| \geq 2^j$ updates to U_{j+1} is $\Omega\left(2^j \cdot \left(\frac{2}{\lambda} \cdot \frac{1}{2^{j+1-t}} + \frac{1}{B}\right)\right) = \Omega\left(1 + \frac{2^j}{B}\right)$. Therefore, this potential drop can compensate for the actual cost of executing steps 1–3 of DECREASE-KEY/DELETE.

Thus the amortized cost of a *Decrease-Key/Delete* operation is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{r}{B}\right) = \mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 N\right)$. But since accessing the first t levels incurs no cache-misses, the amortized cost is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\{\log_2 N - t\}\right) = \mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$.

***Delete-Min*.** The DELETE-MIN function calls the FIND-MIN function followed by a possible call to the DELETE function. We have already shown that the amortized cost of a *Delete* operation is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$. We will show below that the amortized cost of finding the minumum is ≤ 0.

Let j be the largest value of i for which APPLY-UPDATES(i) was called by FIND-MIN. If $|U_j| \geq 2^j$ immediately before APPLY-UPDATES(j) was called (i.e., called inside FIX-U in step $3(i)$ of FIND-MIN), then the analysis is similar to that for *Decrease-Key/Delete* operation. Hence, here we will only consider the case when $|U_j| < 2^j$, i.e., APPLY-UPDATES(j) was called in step $1(ii)$ of FIND-MIN.

As before, we will assume that $j > t$. In this case, using an analysis similar to that for *Decrease-Key/Delete*, one can show that the actual cache complexity of FIND-MIN is $\mathcal{O}\left(1 + \frac{2^j}{B}\right)$.

Let b_j be the number of elements in B_j before APPLY-UPDATES(j) was called. Then in order to compute the potential drop we need to consider the following two cases.

(i) $b_j > 0$: Observe that in this case the last REDISTRIBUTE function call that distributed elements from level j or higher must have left $B_0, B_1, \ldots, B_{j-1}$ completely full, and hence at least $\sum_{i=0}^{j-1}2^i = 2^j - 1$ elements have been deleted from the structure since last time B_j was accessed. Therefore, immediately before the current call to APPLY-UPDATES(j), U_j must have included at least $2^j - 1$ *Delete* operations, all of which were moved to U_{j+1}. Hence, the potential drop due to the movement of these operations is $\Omega\left((2^j - 1) \cdot \left(\frac{2}{\lambda} \cdot \frac{1}{2^{j+1-t}} + \frac{1}{B}\right)\right) = \Omega\left(1 + \frac{2^j}{B}\right)$.

(*ii*) $b_j = 0$: This can only happen when $j = r-1$. Observe that level j was created due to an overflow in B_{j-1} and the overflowing elements from B_{j-1} was pushed into U_j as *Sink* operations. Therefore, at least 2^{j-1} elements have been deleted from the structure since this level was created, and as in case (*i*) this implies a potential drop of $\Omega\left(1+\frac{2^j}{B}\right)$.

The amortized cost of FIND-MIN is thus $\mathcal{O}\left(1+\frac{2^j}{B}\right) - \Omega\left(1+\frac{2^j}{B}\right) \leq 0$.

Therefore, a *Delete-Min* operation incurs $\mathcal{O}\left(\frac{1}{\lambda}+\frac{1}{B}\log_2\frac{N}{\lambda}\right)$ amortized cache-misses. ■

The following corollary follows by replacing λ with $\Theta(M) = \Omega(B)$ in Lemma 3.3.3.

Corollary 3.3.1. *A buffer heap supports* Delete*,* Delete-Min *and* Decrease-Key *operations in* $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{M}\right)$ *amortized cache-misses each using* $O(N)$ *space, where* N *is the current number of elements in the structure.*

Time Complexity

The internal memory time complexities of slim buffer heap operations turn out to be independent of M, B and the slim cache size λ, and are given by the following lemma.

Lemma 3.3.4. *A slim buffer heap supports* Delete*,* Delete-Min *and* Decrease-Key *operations in* $\mathcal{O}(\log N)$ *amortized time each, where* N *is the number of elements in the structure.*

Proof. The proof uses the following potential function:

$$\Phi'(H) = \sum_{i=0}^{r-1}(r-i)\cdot(u_i+d_i) \; + \; r\cdot\Delta,$$

where H is the current state of the data structure, and u_i, d_i and Δ are as defined in the proof of Lemma 3.3.3.

The rest of the proof is similar to that of Lemma 3.3.3 but is simpler, and hence is omitted. ■

Additional Priority Queue Operations

It is straight-forward to augment a slim buffer heap with the following priority queue operations without changing its performance bounds.

Change-Key(x, k_x). This operation changes the key value of element x to k_x, and is implemented by performing a *Delete*(x) operation immediately followed by a *Decrease-Key*(x, k_x) operation. If $k_x \leq k'_x$, where k'_x is the old key of x, then the *Delete* operation acts simply like the *Delete* operation generated by the *Decrease-Key* operation immediately after its application, and thus works correctly. If $k_x > k'_x$, then the *Delete* operation first deletes x, after which the *Decrease-Key* operation reinserts it with the new key value. Since the *Delete* operation has a smaller timestamp than the *Decrease-Key* it cannot delete the new key value inserted by the *Decrease-Key*, and hence works correctly.

Relative-Increase(x, δ_x). This operation increases the key value of x by δ_x if it exists in the priority queue. It is implemented in the same way as the *Change-Key* operation above, but the *Decrease-Key* operation does not know the key value k_x initially and instead knows δ_x. However, as soon as the *Delete*(x) operation preceding the *Decrease-Key* finds the element x, k_x is updated to $k'_x + \delta_x$, where k'_x is the old key value of x discovered by the *Delete*. The *Decrease-Key* operation is then applied as usual.

3.4 Buffer Heap Applications

In this section we discuss three major applications of buffer heap. In Sections 3.4.1 and 3.4.2 we consider cache-oblivious SSSP algorithms for weighted undirected and directed graphs, respectively. These algorithms use regular buffer heaps, that is they do not impose any restriction on the size of the slim cache (i.e., assume slim cache size, $\lambda = \Theta(M) = \Omega(B)$). In Section 3.4.3 we discuss a cache-aware APSP algorithm for weighted undirected graphs. This algorithm uses a data structure built on slim buffer heaps.

3.4.1 Cache-oblivious Undirected SSSP

The cache-aware undirected SSSP algorithm by Kumar & Schwabe [83] (see [77] for a description and proof of correctness) can be made cache-oblivious by replacing both the primary and the auxiliary cache-aware priority queues used in that algorithm with buffer heaps. The primary priority queue is used to perform the standard operations for shortest path computation, and the auxiliary priority queue is used to correct for spurious updates performed on the primary priority queue. The auxiliary priority queue treats edges, instead of vertices, as its elements, and whenever a vertex with final distance $d[u]$ is settled, for each $(u, v) \in E$, a *Decrease-Key*$((u, v), d[u] + w(u, v))$ operation is performed on the auxiliary priority queue. The resulting cache-oblivious algorithm, i.e., Kumar & Schwabe's algorithm with buffer heaps, is given in Function 3.4.1 (Undirected-SSSP).

Function **3.4.1.** Undirected-SSSP($G,\ w,\ s,\ d$)

{*Kumar & Schwabe's algorithm [83] with buffer heap*}

[Given an undirected graph G with vertex set V (each vertex is identified with a unique integer in $[1, |V|)$, edge set E, a weight function $w : E \to \Re$ and a source vertex $s \in V$, this function cache-obliviously computes the shortest distance from s to each vertex $v \in V$ and stores it in $d[v]$.**]**

1. perform the following initializations:

- (*i*) $Q \leftarrow \emptyset$, $Q' \leftarrow \emptyset$ {*Q and Q' are both regular buffer heaps; Q contains items of the form (x, k_x) and Q' contains items of the form $((x, y), k_{x,y})$*}
- (*ii*) ***for*** each $v \in V$ ***do*** $d[v] \leftarrow +\infty$
- (*iii*) Decrease-Key$_{(Q)}$($s,\ 0$) {*insert vertex s with key (i.e., distance) 0 into Q*}

2. ***while*** $Q \neq \emptyset$ ***do***

- (*i*) $(u, k) \leftarrow$ Find-Min$_{(Q)}$( ), $((u', v'), k') \leftarrow$ Find-Min$_{(Q')}$( )
- (*ii*) ***if*** $k \leq k'$ ***then*** {*a new shortest distance (k) has been found* }
  - (*a*) Delete$_{(Q)}$(u), $d[u] \leftarrow k$ {*k is the shortest distance from s to u*}
  - (*b*) ***for*** each $(u, v) \in E$ ***do***
    - Decrease-Key$_{(Q)}$($v,\ d[u] + w(u, v)$) {*relax edge (u, v)*}
    - Decrease-Key$_{(Q')}$($(u, v),\ d[u] + w(u, v)$) {*guard for spurious update on u*}

  else {*$k > k'$: shortest distance to u' has already been computed*}
  - (*a*) Delete$_{(Q)}$(u'), Delete$_{(Q')}$((u', v')) {*remove spurious vertex u'*}

Undirected-SSSP Ends

Cache Complexity. The algorithm incurs $\mathcal{O}\left(\frac{m}{B}\log_2\frac{n}{M}\right)$ cache-misses for the $\mathcal{O}(m)$ priority queue operations it performs. In addition to that it incurs $\mathcal{O}\left(n + \frac{m}{B}\right)$ cache-misses for accessing $\mathcal{O}(n)$ adjacency lists. The cache complexity of the algorithm is thus $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$.

3.4.2 Cache-oblivious Directed SSSP

In this section we describe a cache-oblivious implementation of Dijkstra's directed SSSP algorithm [43] with a regular buffer heap used as a priority queue. Additionally, we use a cache-oblivious *Buffered Repository Tree*[1] (BRT) described in [11], in order to prevent any vertex whose shortest distance from the source vertex has already been determined, from being reinserted into the priority queue. A BRT maintains $\mathcal{O}(m)$ elements with keys in the range $[1 \ldots n]$ under the operations *Insert*(v, u) and *Extract*(u). An *Insert*(v, u) operation inserts a new element v with key u into the BRT, while an *Extract*(u) operation reports and deletes from the data structure

[1]Buffered Repository Trees have been used for breadth-first search and depth-first search in the cache-aware setting in [25] and in the cache-oblivious setting in [11]

all elements v with key u. The *Insert* and *Extract* operations are supported in $\mathcal{O}\left(\frac{1}{B}\log_2 n\right)$ and $\mathcal{O}\left(\log_2 n\right)$ amortized cache-misses, respectively (or in $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{n}{B}\right)$ and $\mathcal{O}\left(\log_2 \frac{n}{B}\right)$ amortized cache-misses, respectively, assuming a tall cache).

The resulting cache-oblivious implementation of Dijkstra's algorithm is given in Function 3.4.2 (Directed-SSSP).

Function **3.4.2.** Directed-SSSP(G, w, s, d)

[Given a directed graph G with vertex set V (each vertex is identified with a unique integer in $[1, |V|]$), edge set E, a weight function $w : E \to \Re$ and a source vertex $s \in V$, this function cache-obliviously computes the shortest distance from s to each vertex $v \in V$ and stores it in $d[v]$.**]**

1. ***for*** each $v \in V$ ***do***
  $L_v \leftarrow \{\ u \mid (u, v) \in E\ \}$ {*L_v is the set of vertices from which v has an incoming edge*}
  $L'_v \leftarrow \{\ \langle u, w(v, u)\rangle \mid (v, u) \in E\ \}$ {*L'_v is the set of vertices to which v has an outgoing edge*}
  sort the items in both L_v and L'_v by vertex number

2. perform the following initializations:

  (*i*) $Q \leftarrow \emptyset$, $D \leftarrow \emptyset$ {*Q is a regular buffer heap that contains items of the form (x, k_x) and D is a BRT capable of containing key values in the range $[1 \ldots |V|]$*}
  (*ii*) ***for*** each $v \in V$ ***do*** $d[v] \leftarrow +\infty$
  (*iii*) Decrease-Key$_{(Q)}$(s, 0) {*insert vertex s with key (i.e., distance) 0 into Q*}

3. ***while*** $Q \neq \emptyset$ ***do***

  (*i*) $(u, k) \leftarrow$ Delete-Min$_{(Q)}$( ), $d[u] \leftarrow k$ {*k is the shortest distance from s to u*}
  (*ii*) $L''_u \leftarrow$ Extract$_{(D)}$(u) {*set of settled vertices to which u has an outgoing edge*}
  sort L''_u by vertex number
  (*iii*) scan L'_u and L''_u simultaneously and ***for*** each $v \in L'_u$ such that $v \notin L''_u$ ***do***
    Decrease-Key$_{(Q)}$(v, $k + w(u, v)$) {*relax edge (u, v) to the yet-to-settle vertex v*}
  (*iv*) ***for*** each $v \in L_u$ ***do***
    Insert$_{(D)}$(u, v) {*mark neighbor u of v as settled*}

Directed-SSSP Ends

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

Our implementation of Dijkstra's algorithm (Directed-SSSP) differs from the standard implementation in two ways, both with an eye to improving cache-efficiency. Firstly, we use a regular buffer heap instead of a standard priority queue. Secondly, instead of accessing a vertex directly in order to determine whether it is settled or not, we use a BRT D to perform these operations cache-efficiently and thus avoid a potential cache-miss during each such operation.

Since we have already proved the correctness of buffer heap (see Lemma 3.3.1), if we simply replace Q with a regular buffer heap in the standard implementation of Dijkstra's algorithm the implementation will still be correct. For $i \in [1, n]$, let u'_i denote the i-th vertex extracted from the priority queue in this implementation, and let V'_i be the set of vertices on which *Decrease-Key* operations are performed immediately after this extraction. Let u_i and V_i have similar definitions for Directed-SSSP. Therefore, assuming the correctness of BRT operations (see [11]), correctness of Directed-SSSP will follow if we can prove the following claim.

Claim 3.4.1. *For $i \in [1, n]$, $u_i = u'_i$ and $V_i = V'_i$.*

Proof. Let $S_i = \{ u_j \mid 1 \leq j \leq i \}$ for $i \in [0, n]$. Then clearly $V'_i = \{ v \mid (u'_i, v) \in E \wedge v \notin S_{i-1} \}$.

Since $u_1 = u'_1 = s$ and D is initially empty, the claim trivially holds for $i = 1$. Now suppose it holds up to some value $j \in [0, n-1]$ of i. We will show that it holds for $i = j + 1$.

Since the claim holds for all $i \leq j$, immediately before the extraction of the $(j+1)$-th vertex from the priority queue, the state of the priority queue in both implementations, i.e., the standard implementation with buffer heap and Directed-SSSP, are exactly the same. Hence, $u_{j+1} = u'_{j+1}$.

Let U_{j+1} be the set of vertices extracted from D in iteration $j + 1$ of the ***while*** loop in Directed-SSSP. Since the claim was true up to iteration j, for each $v \in S_j$ with $(u_{j+1}, v) \in E$, an element u_{j+1} with key value v was inserted into D in step 3(*iv*) at some point during the first j iterations. Hence, $U_{j+1} \supseteq \{ v \mid (u_{j+1}, v) \in E \wedge v \in S_j \}$. Again since D was initially empty and only settled vertices insert items into it, $U_{j+1} = \{ v \mid (u_{j+1}, v) \in E \wedge v \in S_j \}$. Therefore, $V_{j+1} = \{ v \mid (u_{j+1}, v) \in E \} \setminus U_{j+1} = V'_{j+1}$.

Hence, the claim holds for all $i \in [1, n]$. ■

Therefore, Directed-SSSP is a correct implementation of Dijkstra's algorithm.

Cache Complexity. The following lemma gives the cache-complexity of Directed-SSSP.

Lemma 3.4.1. *Single source shortest paths in a directed graph can be computed cache-obliviously in $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log_2 \frac{m}{B}\right)$ cache-misses using a buffer heap under the tall cache assumption.*

Proof. In step 1, all sets L_v and L'_v can be generated with their items in appropriately sorted order after a constant number of sorting and scanning phases incurring $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{m}{B}\right)$ cache-misses.

In step 3, the algorithm performs n *Delete-Min* and m *Decrease-Key* operations on Q, and n *Extract* and m *Insert* operations on D incurring $\mathcal{O}\left(\frac{m+n}{B}\log_2\frac{n}{M}\right)$ and $\mathcal{O}\left(n\log_2\frac{n}{B} + \frac{m}{B}\log_2\frac{n}{B}\right)$ cache-misses, respectively. All lists in step $3(ii)$ can be sorted in $\mathcal{O}\left(\frac{m}{B}\log_2\frac{n}{B}\right)$ cache-misses in total, and the total cache-misses incurred by all scans in steps $3(iii)$ and $3(iv)$ is $\mathcal{O}\left(n + \frac{m}{B}\right)$.

Therefore, overall cache complexity of Directed-SSSP is $\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot\log_2\frac{m}{B}\right)$. ■

Directed SSSP with Cache-oblivious Tournament Tree. In Appendix A we present the *cache-oblivious tournament tree* (COTT) which supports the same set of operations (*Delete*, *Delete-Min* and *Decrease-Key*) as the buffer heap. Although COTT has weaker bounds than buffer heap, it is a simpler data structure, and can be used instead of buffer heap in the directed SSSP algorithm to achieve the same level of cache-efficiency as with buffer heap.

3.4.3 Cache-aware Undirected APSP

In this section we introduce a compound priority queue data structure based on slim buffer heap, called the *Multi-Buffer-Heap* (MBH), and use this structure for cache-efficient computation of APSP on an undirected graph with general non-negative edge-weights.

A multi-buffer-heap is constructed as follows. Let $\lambda < B$ and let $L = \frac{B}{\lambda}$. We pack the slim caches of $\Theta(L)$ slim buffer heaps $SBH(\lambda)$ into a single cache block. We call this block the *multi-slim-cache* and the resulting structure a *multi-buffer-heap*. By the analysis in section 3.3.3 this structure supports *Delete*, *Delete-Min* and *Decrease-Key* operations on each of its component slim buffer heaps in $\mathcal{O}\left(\frac{L}{B} + \frac{1}{B}\log_2\frac{NL}{B}\right)$ amortized cache-misses each.

For computing APSP we take the approach described in [13]. It solves APSP by working on all n underlying SSSP problems simultaneously, and each individual SSSP problem is solved using Kumar & Schwabe's algorithm for weighted undirected graphs [83]. For $1 \le i \le n$, this approach requires a priority queue pair (Q_i, Q'_i), where the i-th pair belongs to the i-th SSSP problem. These n priority queue pairs are implemented using $\Theta(\frac{n}{L})$ multi-buffer-heaps. The algorithm proceeds in n rounds. In each round it loads the multi-slim-cache of each MBH, and for each MBH extracts a settled vertex with minimum distance from each of the $\Theta(L)$ priority queue pairs it stores. It sorts the extracted vertices by vertex indices. It then scans this sorted vertex list and the sorted sequences of adjacency lists in parallel to retrieve the adjacency lists of the settled vertices of this round. Another sorting phase moves all adjacency lists to be applied to the same MBH together. Then all necessary *Decrease-Key* operations are performed by cycling through the multi-buffer-heaps once again. At the end of the algorithm the extracted vertices along with their computed distance values are sorted to produce the final distance matrix.

Cache Complexity. In each round the multi-slim-caches of all multi-buffer-heaps are loaded into the cache in $\mathcal{O}\left(\frac{n}{L}\right)$ cache-misses. Accessing all required adjacency lists over $\mathcal{O}(n)$ rounds incurs $\mathcal{O}(n \cdot sort(m))$ cache-misses, and a total of $\mathcal{O}\big(mn \cdot \left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{n}{\lambda}\right)\big)$ cache-misses are incurred by all $\mathcal{O}(mn)$ priority queue operations performed by this algorithm. The final distance matrix can be sorted in $\mathcal{O}(n \cdot sort(n))$ cache-misses. Thus the total cache complexity of this algorithm is $\mathcal{O}\Big(n \cdot \Big(\frac{n}{L} + \frac{m}{\lambda} + \frac{m}{B}\log_2 \frac{n}{\lambda} + sort(m)\Big)\Big)$. Using $L = \sqrt{\frac{nB}{m}} \geq 1$, we obtain the following:

Lemma 3.4.2. *Using multi-buffer-heaps, APSP on undirected graphs with non-negative real edge weights can be solved in* $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ *cache-misses and* $\mathcal{O}\left(n^2\right)$ *space when* $m \leq \frac{nB}{(\log n)^2}$.

In conjunction with the cache-efficient APSP algorithm for sufficiently dense graphs implied by the SSSP results in [83, 32] we obtain the following corollary.

Corollary 3.4.1. *APSP on an undirected graph with non-negative real edge weights can be solved in* $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + \frac{m}{B}\log \frac{n}{B}\right)\right)$ *cache-misses and* $\mathcal{O}\left(n^2\right)$ *space. The number of cache-misses is reduced to* $\mathcal{O}\left(\frac{mn}{B}\log \frac{n}{B}\right)$ *when* $m \geq \frac{nB}{\left(\log \frac{n}{B}\right)^2}$.

3.5 Conclusion

In this chapter we presented the buffer heap, the first cache-oblivious priority queue that supports *Decrease-Key* operations and used it to obtain the first cache-oblivious SSSP algorithms for weighted undirected and directed graphs, and an improved cache-aware APSP algorithm for weighted undirected graphs. All our cache-oblivious results match the cache complexity of their best cache-aware counterparts. However, open questions still remain. For example:

1. The only known lower bound on the cache complexity of cache-oblivious priority queue operations is $\Omega\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{M}\right)$ amortized which is trivially derived from the sorting lower bound. The buffer heap improves the upper bound from trivial $\mathcal{O}(\log N)$ to $\mathcal{O}\left(\frac{1}{B}\log \frac{N}{M}\right)$ amortized. But there is still a gap between this new upper bound and known lower bound. An open problem is to eliminate this gap.

2. The known cache-miss lower bound for the SSSP problem is $\Omega\left(\frac{m}{n} \cdot sort(n)\right)$ [92]. Though our SSSP algorithms improve significantly over known upper bounds, they are not known to be optimal.

- The n term in the cache complexity of our cache-oblivious undirected SSSP algorithm results from unstructured accesses to adjacency lists. Though some progress has been made in reducing this overhead for bounded-weight graphs [7, 89], nothing is known for graphs with general edge-weights.
- The $n \log n$ term in the cache complexity of our cache-oblivious directed SSSP algorithm results from the overhead of remembering visited vertices. Perhaps a completely new technique for handling this problem will be able to reduce this overhead significantly.

3. The $n\sqrt{\frac{mn}{B}}$ term in the cache complexity of the weighted undirected APSP algorithm described in Section 3.4.3 arises from unstructured accesses to adjacency lists. Though we show in Chapter 5 that we can get rid of this term completely for unweighted undirected graphs, achieving the same for weighted graphs still remains an open question.

Bibliography

[1] 9th DIMACS implementation challenge - shortest paths. url: http://www.dis.uniroma1.it/~challenge9/.

[2] Fujitsu MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP disk drives product/maintenance manual.

[3] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*.

[4] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[6] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[7] L. Allulli, P. Lichodzijewski, and N. Zeh. A faster cache-oblivious shortest-path algorithms for undirected graphs with bounded edge lengths. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 910–919, New Orleans, Louisiana, 2007.

[8] S. Altschul and B. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

[9] ARC/INFO. *Understanding GIS – the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.

[10] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, LNCS 955, pages 334–345. Springer-Verlag, 1995.

[11] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 24th ACM Symposium on Theory of Computing*.

[12] L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP, and multi-way planar graph separation. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, LNCS 1851, pages 433–447. Springer-Verlag, 2000.

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

[14] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 622–627, San Jose, California, 1994.

[15] D. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. url: http://www-static.cc.gatech.edu/~kamesh/GTgraph/.

[16] V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *Proceedings of the 7th Annual International Conference on Research in Computational Molecular Biology*, pages 9–18, Berlin, Germany, 2003.

[17] R. Bellman. *Dynamic Programming*. The Princeton University Press, Princeton, New Jersey, 1957.

[18] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, Barcelona, Spain, 2004.

[19] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[20] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[21] G. Brodal. Cache-oblivious algorithms and data structures. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, LNCS 3111, pages 3–13, Humlebæk, Denmark, 2004. Springer-Verlag.

[22] G. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, LNCS 2518, Vancouver, BC, Canada. Springer-Verlag.

[23] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, San Diego, California, 2003.

[24] G. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, pages 480–492, Humlebæk, Denmark, July 2004.

[25] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[26] A. Buchsbaum and J. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575, 2000.

[27] J. Cannone, S. Subramanian, M. Schnare, J. Collett, L. D'Souza, Y. Du, B. Feng, N. Lin, L. Madabusi, K. Muller, N. Pande, Z. Shang, N. Yu, and R. Gutell. The comparative RNA web (CRW) site: An online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BioMed Central Bioinformatics*, 3:2, 2002. url: http://www.rna.icmb.utexas.edu/.

[28] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining*, Orlando, Florida, 2004.

[29] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thotethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[30] C. Cherng and R. Ladner. Cache efficient simple dynamic programming. In *Proceedings of the International Conference on the Analysis of Algorithms*, pages 49–58, Barcelona, Spain, 2005.

[31] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External-memory graph algorithms. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

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

[33] R. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest paths in undirected graphs. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 735–744, Vancouver, BC, Canada, 2005. More details can be found in the technical report with the same title, TR-04-38, CS Dept., UT Austin, August 2004.

[34] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, Miami, Florida, 2006.

[35] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 71–80, San Diego, California, 2007.

[36] T. Cormen. *Virtual Memory for Data Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[37] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[38] R. Cromp. An intelligent information fusion system for handling the archiving and querying of terabyte–sized spatial databases. In S. Tate, editor, *Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community*, pages 75–84. CESDIS Technical Report Series, 1993.

[39] P. D'Alberto and A. Nicolau. R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.

[40] R. Dementiev. STXXL homepage, documentation and tutorial. url: http://stxxl.sourceforge.net/.

[41] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. In *Proceedings of the 13th Annual European Symposium on Algorithms*, LNCS 1004, pages 640–651. Springer-Verlag, 2005.

[42] T. DeSantis, I. Dubosarskiy, S. Murray, and G. Andersen. Comprehensive aligned sequence construction for automated design of effective probes

(cascade-p) using 16S rDNA. *Bioinformatics*, 19:1461–1468, 2003. url: http://greengenes.llnl.gov/16S/.

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

[44] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[45] S. Dreyfus and A. Law. *The Art and Theory of Dynamic Programming*. Academic Press Inc., 1977.

[46] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[47] P. Erdös and A. Rényi. On the evolution of random graphs. *Mat. Kuttató. Int. Közl.*, 5:17–60, 1960.

[48] R. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[49] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles & Practice*. Addison-Wesley, 1999.

[50] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

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

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

[53] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, 1998.

[54] M. Frigo and V. Strumpen. Cache-oblivious stencil computations. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 361–366, Cambridge, Massachusetts, 2005.

[55] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, Cambridge, Massachusetts, 2006.

[56] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[57] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $o(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[58] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[59] G. Golub and C. Van Loan. *Matrix Computations*. The John Hopkins University Press, third edition, 1996.

[60] K. Goto. GotoBLAS, 2005. url: http://www.tacc.utexas.edu/resources/software.

[61] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[62] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, second edition, 1994.

[63] J. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *Computer Applications in the Biosciences*, 13(1):45–53, 1997.

[64] R. Grossi and G. Italiano. Efficient cross-trees for external memory. In J. Abello and J. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society Press, Providence, RI, 1999.

[65] R. Grossi and G. Italiano. Revised version of "Efficient cross-trees for external memory". Technical Report TR-00-16, Dipartimento di Informatica, Università de Pisa, Pisa, Italy, 2000.

[66] J. Gunnels, F. Gustavson, G. Henry, and R. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[67] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, 1997.

[68] L. Haas and W. Cody. Exploiting extensible DBMS in integrated geographic information systems. In *Proceedings of the 2nd International Symposium on Advances in Spatial Databases*, LNCS 525, pages 423–450. Springer-Verlag, 1991.

[69] P. Hayes, D. Joyce, and P. Pathak. Ubiquitous learning – an application of mobile technology in education. In *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications*, volume 1, Lugano, Switzerland.

[70] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[71] D. Hirschberg and L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

[72] C. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[73] C. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[74] J. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[75] K. Iversion. *A Programming Language*. Wiley, 1962.

[76] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for data models with constraints and classes. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pages 233–243, 1993.

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

[78] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesely, 2005.

[79] B. Knudsen. Multiple parsimony alignment with "affalign". Software package `multalign.tar`.

[80] B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *Proceedings of Workshop on Algorithms in Bioinformatics*, pages 433–446, 2003.

[81] D. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[82] D. Knuth. Two notes on notation. *American Mathematical Monthly*, 99:403–422, 1992.

[83] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–177, 1996.

[84] A. LaMarca and L. R. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, 1:4, 1996.

[85] D. Lan Roche. Experimental study of high performance priority queues, 2007. Undergraduate Honors Thesis, CS-TR-07-34, The University of Texas at Austin, Department of Computer Sciences.

[86] R. Laurini and A. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[87] H. Le. Algorithms for identification of patterns in biogeography and median alignment of three sequences in bioinformatics, 2006. Undergraduate Honors Thesis, CS-TR-06-29, The University of Texas at Austin, Department of Computer Sciences.

[88] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, LNCS 2461, pages 723–735. Springer-Verlag, 2002.

[89] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th European Symposium on Algorithms*, LNCS 2832, pages 434–445. Springer-Verlag, 2003.

[90] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. 1994.

[91] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[92] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[93] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[94] S. Pan, C. Cherng, K. Dick, and R. Ladner. Algorithms to take advantage of hardware prefetching. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pages 91–98, 2007.

[95] J. Park, M. Penner, and V. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[96] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the USA*, volume 85, pages 2444–2448, 1988.

[97] S. Pettie. Towards a final analysis for pairing heaps. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–183, 2005.

[98] S. Pettie and V. Ramachandran. Command line tools generating various families of random graphs. url: http://www.dis.uniroma1.it/~challenge9/code/Randgraph.tar.gz.

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

[100] D. Powell. Software package `align3str_checkp.tar.gz`.

[101] D. Powell, L. Allison, and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *Journal of Theoretical Biology*, 207(3):325–336, 2000.

[102] D. Powell, L. Allison, and T. Dix. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. url: http://math-atlas.sourceforge.net.

[103] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

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

[105] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, pages 25–35, Vancouver, BC, Canada, 1994.

[106] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. 285(5):2053–2068, 1999.

[107] W. Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59:297–307, 1988.

[108] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison-Wesley, 1989.

[109] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics*, 5:1–25, 2000.

[110] P. Sanders. Memory hierarchies – models and lower bounds. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, LNCS 2625. Springer-Verlag, 2003.

[111] P. Sanders and D. Schultes. United states road networks (tiger/line). Data Source: U.S. Census Bureau, Washington, DC, url: http://www.dis.uniroma1.it/~challenge9/data/tiger/.

[112] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-Linux programs). url: http://valgrind.kde.org/index.html.

[113] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[114] M. Sniedovich. *Dynamic Programming*. The Marcel Dekker, Inc., New York, NY, 1992.

[115] J. Stasko and J. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30:234–249, 1987.

[116] G. Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, third edition, 1988.

[117] G. Tan, S. Feng, and S. Ninghui. Cache oblivious algorithms for nonserial polyadic programming. *The Journal of Supercomputing*, 39(2):227–249, 2007.

[118] G. Tan, S. Ninghui, and G. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 135–144, San Diego, California, 2007.

[119] J. Thomas, J. Touchman, R. Blakesley, G. Bouffard, S. Beckstrom-Sternberg, E. Margulies, M. Blanchette, A. Siepel, P. Thomas, J. McDowell, B. Maskeri, N. Hansen, M. Schwartz, R. Weber, W. Kent, D. Karolchik, T. Bruen, R. Bevan, D. Cutler, S. Schwartz, L. Elnitski, J. Idol, A. Prasad, S. Lee-Lin, V. Maduro, T. Summers, M. Portnoy, N. Dietrich, N. Akhter, K. Ayele, B. Benjamin, K. Cariaga, C. Brinkley, S. Brooks, S. Granite, X. Guan, J. Gupta, P. Haghihi, S. Ho, M. Huang, E. Karlins, P. Laric, R. Legaspi, M. Lim, Q. Maduro, C. Masiello, S. Mastrian, J. McCloskey, R. Pearson, S. Stantripop,

E. Tiongson, J. Tran, C. Tsurgeon, J. Vogt, M. Walker, K. Wetherby, L. Wiggins, A. Young, L. Zhang, K. Osoegawa, B. Zhu, B. Zhao, C. Shu, P. De Jong, C. Lawrence, A. Smit, A. Chakravarti, D. Haussler, P. Green, W. Miller, and E. Green. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, 424:788–793, 2003.

[120] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[121] L. Toma and N. Zeh. I/O-efficient algorithms for sparse graphs. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, LNCS 2625. Springer-Verlag, 2003.

[122] L. Tong. Implementation and experimental evaluation of the cache-oblivious buffer heap, 2006. Undergraduate Honors Thesis, CS-TR-06-21, The University of Texas at Austin, Department of Computer Sciences.

[123] J. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360.

[124] L. Valiant. General context-free recognition in less than cubic time. *Journal of Compute and System Sciences*, 10:308–315, 1975.

[125] D. Vengroff and J. Vitter. I/O–efficient scientific computation using TPIE. In *Proceedings of IEEE Symposium on Parallel and Distributed Computing*, pages 74–77, 1995.

[126] V. Viswanathan, S. Huang, and H. Liu. Parallel dynamic programming. In *Proceedings of IEEE Conference on Parallel Processing*, pages 497–500, 1990.

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

[128] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[129] M. Waterman. *Introduction to Computational Biology.* Chapman & Hall, London, UK, 1995.

[130] J. Watson. The human genome project: Past, present and future. *Science*, 248:44–49, 1990.

[131] I. Wegner. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[132] J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.

[133] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[134] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, 1993.

[135] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-aware programs. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 93–104, San Diego, California, 2007.

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