Cache-Oblivious Planar Shortest Paths

Hema Jampala[⋆] and Norbert Zeh[⋆]

Faculty of Computer Science, Dalhousie University,
6050 University Ave, Halifax, NS B3H 1W5, Canada
Email: {jampala,nzeh}@cs.dal.ca

Abstract. We present an efficient cache-oblivious implementation of the shortest-path algorithm for planar graphs by Klein et al., and prove that it incurs no more than $\mathcal{O}\left(\frac{N}{B^{1/2-\epsilon}} + \frac{N}{B}\log N\right)$ block transfers on a graph with N vertices. This is the first cache-oblivious algorithm for this problem that incurs $o(N)$ block transfers.

1 Introduction

The *single-source shortest-path* (SSSP) problem is a fundamental combinatorial optimization problem with numerous applications. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E; let $s \in V$ be a distinguished vertex, called the *source vertex*; and let $\omega : E \to \mathbb{R}^+$ be an assignment of non-negative real weights to the edges of G. The SSSP-problem is to find, for every vertex $v \in V$, the distance $d(v)$ from s to v, that is, the weight of a minimum-weight (shortest) path from s to v. This problem is well-studied in the RAM-model. The classical algorithm for this problem is Dijkstra's algorithm [15], which has seen many improvements (e.g.,[19, 21–23]). In particular, on planar graphs, much progress has been made: Frederickson [17] proposes an algorithm that takes $\mathcal{O}(N\sqrt{\log N})$ time, pioneering the idea to use separator decompositions to speed up shortest-path computations. Klein et al. [19] present a non-trivial refinement of Frederickson's approach that uses a hierarchy of nested separator decompositions to solve SSSP in planar directed graphs in linear time.

More recently, the SSSP-problem has been studied in memory hierarchy models, which take the varying access times of different levels of cache, main memory, and disk into account. Such algorithms can be *cache-aware* or *cache-oblivious*. The former require knowledge of the parameters of the different levels of memory and often explicitly transfer data between the different levels; the latter are oblivious of these parameters, but help the default paging algorithm by laying out the data appropriately and accessing it in a local fashion.

The most widely used model for the design of *cache-aware* algorithms is the I/O-model of Aggarwal and Vitter [1]. This model assumes a memory hierarchy consisting of two levels: the lower level has size M; data is transferred between the two levels in blocks of B consecutive data items. The complexity

⋆ Research supported by the Natural Sciences and Engineering Research Council of Canada and by the Canadian Foundation for Innovation.

of an algorithm is the number of blocks transferred (*I/Os*). Algorithms that perform a small number of I/Os are usually referred to as *I/O-efficient* algorithms. The strength of the I/O-model is its simplicity, while it still adequately models the situation when the I/Os between two levels of the memory hierarchy dominate the running time of the algorithm, which is often the case in large-scale applications. Complexities that arise often in I/O-efficient algorithms are the sorting bound, $\text{sort}(N) = \Theta\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\big)$ I/Os [1, 18], the permutation bound, $\text{perm}(N) = \Theta(\min(N, \text{sort}(N)))$ I/Os [1], and the scanning bound, $\text{scan}(N) = \Theta(N/B)$ I/Os.

Solving interesting problems using cache-aware algorithms for *multi-level* hierarchies is cumbersome, because it is necessary to tune the algorithms to the sizes and block sizes of all levels of memory. *Cache-oblivious* algorithms provide an elegant solution to this problem. They are designed to be I/O-efficient without knowing M or B; that is, they are formulated in the RAM-model and analyzed in the I/O-model, assuming that the memory transfers are performed by an optimal offline paging algorithm. Since the analysis holds for any block and memory sizes, it holds for *all* levels of a multi-level memory hierarchy (see [18] for details). Thus, the cache-oblivious model elegantly combines the simplicity of the I/O-model with a coverage of the entire memory hierarchy. The bounds for sorting and scanning are the same as in the I/O-model [9, 18], whereas $\text{perm}(N) = \text{sort}(N)$ in the cache-oblivious model [11]. Since any internal-memory algorithm is by definition cache-oblivious, but usually incurs a substantial number of block transfers, we refer to an algorithm as *cache-oblivious* in this paper if it is cache-oblivious in the sense of the definition and incurs $o(T(N))$ block transfers, where $T(N)$ is the computation time of the best internal-memory algorithm.

Previous work on graph algorithms for memory hierarchies has focused mainly on *I/O-efficient* algorithms, motivated by a number of large-scale applications that have to deal with massive graphs, such as geographic information systems, web modelling, and data mining of phone call databases. The obtained results include a large number of algorithms for planar graphs, such as $\mathcal{O}(\text{sort}(N))$-I/O algorithms for computing connected components [13], minimum spanning trees [13], and strongly connected components [8]; breadth-first search (BFS) [3] and undirected depth-first search (DFS) [5]; single-source shortest paths [3]; and topological sorting [6, 7]. Directed planar DFS is studied in [8], and an optimal $\mathcal{O}(N^2/B)$-I/O all-pairs shortest path algorithm is presented in [4].

Recently, a number of cache-oblivious graph algorithms have been obtained for general graphs, including algorithms for computing connected components and minimum spanning trees [2], directed breadth-first search and depth-first search [2], undirected breadth-first search [12], and undirected shortest paths [12, 14]. All these algorithms are obtained from I/O-efficient algorithms for these problems by designing cache-oblivious data structures that can replace the cache-aware ones in these algorithms. This strategy does not seem to work for most of the specialized algorithms for planar graphs, mentioned above: Their dependence on B is not hidden in data structures; and many of them exploit that computation in internal memory is free in the I/O-model, by performing $\Omega(BN)$

computation in main memory. In a multi-level hierarchy, this extra computation may incur block transfers at lower cache levels, thereby leading to an excessive number of block transfers. The internal-memory computation can often be reduced. For example, for shortest paths, it can be reduced to $\mathcal{O}(n \cdot \operatorname{polylog}(B))$, using results from [16]; but these algorithms are not easily made cache-oblivious. Nevertheless, a number of cache-oblivious algorithms for planar graphs exist. Using cache-oblivious data structures from [2, 10], the I/O-efficient algorithms for connectivity, biconnectivity, and minimum spanning trees [13], and for topologically sorting planar directed acyclic graphs [7] can be made cache-oblivious, without asymptotically increasing their complexities.

2 New Result

We make the first progress towards solving the SSSP-problem cache-obliviously in planar graphs, by analyzing the number of block transfers incurred by a cache-efficient implementation of the algorithm of [19]. We assume that a suitable multi-level separator decomposition of the graph is given. We consider finding such a decomposition the central open problem in the design of cache-oblivious algorithms for planar graphs, because separators also have played a central role in the design of I/O-efficient algorithms for planar graphs. In Sec. 7, we suggest one approach that may lead to a cache-oblivious algorithm for this problem. Our result is summarized in the following theorem.

Theorem 1. *The SSSP-problem in a planar directed graph G with N vertices and non-negative edge weights can be solved using a cache-oblivious algorithm that incurs $\mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}} + \frac{N}{B}\log N\big)$ block transfers, for any constant $\epsilon > 0$, provided that a suitable multi-level separator decomposition of G is given.*

In this paper, we assume that every vertex in G has in-degree at most 2 and out-degree at most 2. A simple transformation described, for instance, in [20] achieves this. The discussion of the algorithm is divided into several sections. In Sec. 3, we discuss the necessary terminology regarding graph separators. We outline the linear-time SSSP-algorithm of [19] in Sec. 4. We show in Sec. 5 how to implement this algorithm in a cache-efficient manner. The analysis of the algorithm is provided in Sec. 6. Open problems are discussed in Sec. 7.

3 Separator Decompositions

The algorithm of [19], as many other SSSP-algorithms for planar graphs, uses a separator decomposition to organize its computation. Next we define the required partition and discuss its representation.

Definition. For a planar graph $G = (V, E)$ and an integer parameter $r > 0$, an *r-partition* of G is a pair $(S, \{G_1, G_2, \ldots, G_h\})$ with the following properties: (i) S is a subset of V of size $\mathcal{O}(N/\sqrt{r})$. (ii) The graphs $G_1, G_2, \ldots, G_h$ are edge-disjoint subgraphs of G whose union is G and such that any two such graphs

G_i and G_j share only vertices in S. (iii) Every graph G_i, $1 \leq i \leq h$, contains at most r edges and at most $\sqrt{r}$ vertices from S. We call the vertices in S *separator vertices*. The set of vertices in S that are contained in a graph G_i are the *boundary* of G_i.

Given a vector $\mathbf{r} = (r_0, r_2, \ldots, r_k)$, a *recursive* $\mathbf{r}$*-partition* $\mathcal{P}$ of G consists of a sequence of partitions of G, $(\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k)$, with the following properties: (i) Each $\mathcal{P}_i$ is an r_i-partition of G. (ii) For two consecutive partitions $\mathcal{P}_i = (S_i, \{G_1, G_2, \ldots, G_s\})$ and $\mathcal{P}_{i+1} = (S_{i+1}, \{G'_1, G'_2, \ldots, G'_t\})$, $S_i \supset S_{i+1}$ and, for every graph G_j in $\mathcal{P}_i$, there exists a graph G'_ℓ in $\mathcal{P}_{i+1}$ such that $G_j \subseteq G'_\ell$.

In this paper, a recursive $\mathbf{r}$-partition will always satisfy $r_0 = 1$ and $r_k = |E|$; that is, the lowest level of the partition splits G into its edges, and the highest level consists of the entire graph.

Representation. A recursive $\mathbf{r}$-partition $\mathcal{P}$ can be represented quite naturally by a rooted tree. The root of the tree is the single subgraph in $\mathcal{P}_k$, which is G. For a node corresponding to a graph R in $\mathcal{P}_i$, $i > 0$, its children represent the graphs in $\mathcal{P}_{i-1}$ that are subgraphs of R. The leaves represent the edges of G. We call this tree the *partition tree* of $\mathcal{P}$. The subgraphs of G corresponding to the nodes in this tree are referred to as *regions*. The edges of G, which correspond to the leaves of the tree, are *atomic regions*, as they cannot be split any further. The *level* of a region R is the index i such that R is a graph in $\mathcal{P}_i$. We define ancestry of regions to be the ancestry of the corresponding nodes in the partition tree.

4 Outline of the Algorithm

The algorithm of [19] is a variant of Dijkstra's algorithm, implemented using a hierarchy of priority queues associated with the regions of a recursive $\mathbf{r}$-partition of G (see Alg. 1). The entries in the priority queue $Q(R)$ of region R are the children of R; the priority of such a child R' is the minimum tentative distance of the source vertices of all unrelaxed edges in R'. Initially, the priorities of the regions containing out-edges of s are 0 and those of the rest are $+\infty$. The algorithm chooses the child with minimal priority from $Q(G)$ and recurses on this child. If the child is non-atomic, it repeats this process; otherwise it relaxes the edge (u, v) in this region and updates the priorities of all regions containing the out-edges of v. Procedure Shortest-Paths is called on G until all edges are relaxed, that is, the priority of G itself is $+\infty$.

If every invocation of this procedure on a non-atomic region were to return after the first recursive call it makes on one of its children, this algorithm would be Dijkstra's algorithm. However, the procedure returns from a recursive call on a level-i region R only after all edges in the region have been relaxed or α_i recursive calls have been made on children of R, for a suitable parameter α_i.

By keeping the algorithm focused on a region for some time, once a recursive call has been made on this region, Klein et al. limit the number of priority queue operations on large priority queues (that is, priority queues attached to regions at higher levels) and obtain a linear time bound for their algorithm. The other

```
Shortest-Paths(R)
 1 ▷ R is the region on which the current invocation operates.
 2 $j \leftarrow \text{level}(R)$
 3 if $j = 0$
 4   then Relax the edge $(u, v)$ in $R$, that is, set $d(v) = \min(d(v), d(u) + \omega(u, v))$.
 5        if this changes the distance of $v$ from $s$
 6          then Change the priority of every out-edge $(v, w)$ of $v$ to $d(v)$.
 7               Change the priority of every ancestor of $(v, w)$ whose priority is
                 greater than $d(v)$ to $d(v)$.
 8        Change the priority of $(u, v)$ to $\infty$.
 9   else $i \leftarrow 0$
10        while the minimum entry in $Q(R)$ is not $\infty$ and $i < \alpha_j$
11          do $R' \leftarrow$ Delete-Min$(Q(R))$
12             Shortest-Paths$(R')$
13             Let the priority of $R'$ in $Q(R)$ be the minimum priority in $Q(R')$.
14             $i \leftarrow i + 1$
```

Algorithm 1: Outline of the shortest-path algorithm. This procedure is called repeatedly with the whole graph G as the argument until all edges are relaxed.

advantage this approach has is that it avoids "jumping around randomly"; that is, once the algorithm focuses on a subgraph that fits into cache, it stays focused on this subgraph for a while. The computation inside the subgraph does not incur any block transfers after the whole subgraph has been loaded into the cache. To take full advantage of this, however, the algorithm has to be implemented using cache-oblivious data structures, and the parameters in the algorithm have to be chosen differently, which increases the amount of computation to $\mathcal{O}(N \log N)$. In Sec. 7, we argue that, while a reduction of the log-factor may be possible, it cannot be eliminated entirely.

5 Cache-Oblivious Implementation

In [19], Algorithm 1 is used more or less verbatim to obtain a linear running time. The updates of the priorities of all relevant regions in Lines 6 and 7 are performed using a so-called GlobalUpdate operation. This operation traverses the path from region (v, w) to the lowest common ancestor of regions (u, v) and (v, w) in the partition tree and performs the updates described in Lines 6 and 7.

A cache-efficient algorithm cannot use this strategy because the updates of atomic regions alone would require $\Theta(N)$ block transfers in the worst case. But we can exploit that the recursive calls in the algorithm correspond to a traversal of the partition tree: The correct priority of a region R has to be known only by the time the traversal visits its parent R' because only then this information is required to extract the correct child to be visited from $Q(R')$. In order to reach R', we have to traverse the path from the current region (u, v) to R'. Thus, if the relaxation of edge (u, v) affects the priority of R, we can carry this update along the path from (u, v) to R' and apply it to $Q(R')$ immediately before the next Delete-Min operation is to be performed on $Q(R')$. The details are as follows:

Every invocation I on a region R collects all updates on atomic regions outside of R that are triggered by descendants of I and passes them to its parent invocation I' when it returns. Let R' be the region of I'. Then I' inspects the updates received from I. Every update on an atomic region outside of R' is scheduled to be passed to the parent of I'. Updates on atomic regions inside R' are scheduled to be sent to the appropriate children of R' when the next invocations on these children are made. When an invocation on R' makes a recursive call on one of these children, R'', the updates scheduled to be sent to R'' are applied to $Q(R'')$ and then scheduled to be sent to the appropriate children of R'', based on which child contains the edge affected by each update.

Line 5 presents a similar problem, with a similar solution: We cannot afford to access vertex v to test whether the relaxation of edge (u, v) decreases $d(v)$. To avoid this, we store $d(v)$ with both in-edges (u_1, v) and (u_2, v) of v. When one in-edge, say (u_1, v), decreases the distance, it informs the other in-edge, (u_2, v), by sending an Update-Target message to (u_2, v). Again, the correct distance of v has to be known to region (u_2, v) only when the next invocation on this region is made; that is, Update-Target messages can be delivered in the same fashion as Update messages for the out-edges of v.

We implement this strategy using the following data structures: We use a stack S to collect and pass updates to ancestors of the current region. We associate a cache-oblivious buffered repository tree (BRT) $B(R)$ [2] with every region, which we use to collect all updates to be sent from R to its children. The tree $B(R)$ serves yet another purpose: Since the updates to be performed on $Q(R)$ are in fact updates on atomic regions, and it is too costly to identify the child of R affected by every update, we store atomic regions in $Q(R)$. When retrieving the minimum entry (u, v) from $Q(R)$, we have to (1) determine the child R' of R that contains edge (u, v), in order to make a recursive call on R', and (2) remove all edges in R' from $Q(R)$, in order to effectively set the priority of R' to $+\infty$. Using the BRT, we can achieve both: Assume that the leaves of the partition tree are numbered left to right and that every internal node has been labelled with the interval of numbers of its descendant leaves. (This can be achieved in a preprocessing step.) We associate with every leaf of a BRT $B(R)$, which corresponds to a region R', the interval associated with R' and with every internal node the union of the intervals of its children. This is sufficient to decide, for every update on an atomic region, to which leaf in the BRT it should be sent and to use any atomic region contained in a child R' as the key for an Extract operation that identifies R' and retrieves all updates on descendants of R'. Once the updates on R' are retrieved, they can be deleted from $Q(R)$. The details of the cache-oblivious implementation are shown in Alg. 2.

From our discussion, it follows that the modifications do not change the sequence of edge relaxations performed by the algorithm. Hence, the algorithm remains correct. We summarize this in the following lemma:

Lemma 1. *Procedure CO-Shortest-Paths terminates with the label $d(v)$ of every vertex v set to its distance from s in G.*

```
CO-Shortest-Paths($R$, $U$)
1  ▷ $R$ is the region on which the current invocation operates; $U$ is an array of Update
   operations provided by the parent invocation.
2  $j \leftarrow$ level($R$)
3  if $j = 0$
4     then Update $d(u)$ and $d(v)$ using the minimum priorities of the Update and
          Update-Target operations in $U$, respectively.
5          Relax the edge $(u, v)$ in $R$, that is, set $d(v) = \min(d(v), d(u) + \omega(u, v))$.
6          if Line 5 changes the distance of $v$ from $s$
7             then Insert an Update($(v, w), d(v)$) operation into $S$, for every out-
                  edge $(v, w)$ of $v$; mark this operation as inserted by region $R$.
8                  Insert an Update-Target($(u', v), d(v)$) operation into $S$, where
                  $(u', v)$ is the other in-edge of $v$; mark this operation as inserted
                  by region $R$.
9                  Change the priority of $(u, v)$ to $\infty$ and return region $(u, v)$ with
                  priority $\infty$, for insertion into the parent's priority queue.
10    else Perform the updates in $U$ on $Q(R)$ and insert them into $B(R)$.
11         $i \leftarrow 0$
12         while the minimum entry in $Q(R)$ is not $\infty$ and $i < \alpha_j$
13            do $(u, v) \leftarrow$ Delete-Min($Q(R)$)
14               Identify the child $R'$ of $R$ that contains edge $(u, v)$ and extract all
                 updates on descendants of $R'$ from $B(R)$.
15               Store these updates in an array $U'$ and delete all regions affected
                 by these updates from $Q(R)$.
16               CO-Shortest-Paths($R', U'$); let $(x, y)$ be the edge returned by this
                 recursive call, and let $p$ be its priority.
17               Insert edge $(x, y)$ into $Q(R)$, with priority $p$, and into $B(R)$.
18               Extract all updates from $S$ that have been inserted by $R'$. Re-insert
                 the updates on edges outside of $R$ and mark them as inserted by
                 $R$. Apply all other updates to $Q(R)$ and insert them into $B(R)$.
19               $i \leftarrow i + 1$
20         Return the minimum-priority entry in $Q(R)$ for insertion into the parent's
           priority queue.
```

Algorithm 2: Cache-oblivious implementation of the shortest-path algorithm.

6 Analysis

In this section, we analyze the number of block transfers incurred by the algorithm. One of the keys to minimizing the number of block transfers is an appropriate layout of the data structures in memory. We describe this layout in Sec. 6.1. We also describe the paging strategy we use in our analysis; the optimal paging strategy cannot do worse than ours. In Sec. 6.2, we show that our paging strategy incurs at most the number of block transfers stated in Thm. 1.

6.1 Memory Layout and Space Bound

The partition, including the associated data structures, is laid out in memory as follows: We number the nodes of the partition tree in a depth-first manner.

For every node representing a region R of size r, we allocate space for $Q(R)$, $B(R)$, and a buffer space $H(R)$ of size $3a^{1/2a}r^{1-1/2a}$, for some constant $1 < a \leq 2$ defined in Sec. 6.2, which holds the buffers associated with the nodes in $B(R)$. If we refer to every region by its number, we arrange these structures in the order $Q(1), B(1), H(1), Q(2), B(2), H(2), \ldots$. The space for stack S succeeds these data structures and buffer spaces for the regions in the partition.

The space allocated for every data structure is fixed, that is, the memory layout is static. Preallocating space for the BRT's and priority queues is easy because we can bound their sizes by $\mathcal{O}(r^{1/a})$ and $\mathcal{O}(r^{1-1/2a})$, respectively. For the BRT, excluding buffers, this follows because we will show in Sec. 6.2 that the number of children of R is $\mathcal{O}(r^{1-1/a})$, and $B(R)$ contains one leaf per child. For the priority queue, we observe that an edge (atomic region) stored in $Q(R)$ is either incident to a boundary vertex of a child of R or it is the value returned by the last recursive call to a child of R. The number of entries of the first type is bounded by the number of edges incident to boundary vertices of children of R, which is $\mathcal{O}(r^{1-1/2a})$. For entries of the second type, we observe that each such entry is removed before the next recursive call to the child that inserted the entry. Hence, there can be at most one such entry per child of R in $Q(R)$, that is, at most $\mathcal{O}(r^{1-1/a}) = \mathcal{O}(r^{1-1/2a})$ entries. In order to bound the buffer space required by the BRT, we exploit that, similar to $Q(R)$, $B(R)$ cannot store updates on more than $\mathcal{O}(r^{1-1/2a})$ atomic regions, which allows us to use a variant of the BRT with limited buffer space. This variant is discussed in the next paragraph.

Another way to look at the above space bound is that every level uses space proportional to the number of boundary vertices at the next lower level. Since the number of boundary vertices at level 0 is $\mathcal{O}(N)$, and the number of boundary vertices per level decreases by at least a constant factor as we proceed towards the root (see Sec. 6.2), we obtain the following lemma:

Lemma 2. *The layout of the partition tree, including priority queues, BRT's, and buffer spaces associated with its nodes, uses linear space.*

A BRT with limited buffer space. In order to limit the buffer space for every BRT, we exploit that, if there is more than one update pending on an atomic region, it suffices to perform the update with minimum priority. We partition the buffer space for the BRT of R into three regions of size $t = a^{1/2a}r^{1-1/2a}$, called the *sorted*, *unsorted*, and *root* buffer space. The root buffer space holds the root buffer. The unsorted buffer space holds buffers for all non-root nodes as in [2]. The sorted buffer space stores additional buffers, one per leaf, whose entries are sorted by the atomic regions they affect. As will become clear from the following discussion, the sorted buffer space can never overflow. As long as neither the root buffer nor the unsorted buffer space overflows, we operate on the BRT as described in [2], except that an Extract operation needs to read out an additional leaf buffer in the sorted buffer space.

When either the root buffer or the unsorted buffer space overflows, we sort the contents of these buffer spaces and then merge them with the contents of the sorted buffer space. If there are duplicate entries in the resulting list, we keep only the one with minimum priority. Hence, the resulting list has size at most t and completely fits into the sorted buffer space.

The cost of all BRT operations, excluding the cost of compacting the buffers when the root buffer or unsorted buffer space overflows, remains $\mathcal{O}\left(\frac{\log N}{B}\right)$ for Insert operations and $\mathcal{O}(\log N + K/B)$ for Extract operations. Next we argue that the cost of buffer compaction is $\mathcal{O}\left(\frac{\log t}{B}\right)$ amortized per element, if $t \geq B$: The cost of sorting the contents of the unsorted and root buffer space is $\mathcal{O}(\mathrm{sort}(t'))$, where t' is the number of elements in these two buffer spaces. The rest of the compaction takes two scans of $\mathcal{O}(t)$ data, which requires $\mathcal{O}(t/B)$ block transfers. This is $\mathcal{O}\left(\frac{\log t}{B}\right)$ amortized per element if we can prove that $t' \geq t/\log t$. To do so, observe that every element, as it is propagated down the tree, requires us to allocate one memory cell at each level in the BRT, $\log t$ in total. Hence, it takes $t' \geq t/\log t$ elements to make the unsorted or root buffer space overflow.

A paging strategy. We permanently keep the top two blocks of S in cache. For every invocation on a region of size at most B, we load the part of the data structure into memory that corresponds to the region and its descendants. We call such an invocation *small*. The same argument that establishes the linear space bound in Lem. 2 implies that the size of this portion of the data structure is $\mathcal{O}(B)$; and it is stored consecutively in memory. Hence, such an invocation costs $\mathcal{O}(1)$ block transfers, and the descendant invocations do not incur any further block transfers, except for stack operations. For any invocation on a region of size greater than B, we load the first block of $Q(R)$ and the last block of the root buffer of $B(R)$ into cache. Thus, excluding the cost of priority queue, BRT, and stack operations, each such invocation costs $\mathcal{O}(1)$ I/Os. We call such an invocation *large*.

6.2 Counting Block Transfers

It remains to analyze the number of block transfers performed by our paging algorithm. We start by choosing suitable region sizes $r_1, r_2, \ldots, r_k$ and parameters $\alpha_1, \alpha_2, \ldots, \alpha_k$. In order to obtain the complexity stated in Thm. 1, we choose

$$r_j = a^j \cdot 2^{a^j/(a-1)} \quad \text{and} \quad \alpha_j = 2^{a^{j-1}/2},$$

where $a = 1/(1-\epsilon) > 1$. The height of the recursive partition is the minimum k such that $r_k \geq 3N$; thus, the height of the partition is no more than $\log_a((a-1)\log 3N) = \mathcal{O}(\log\log N)$. We can divide the block transfers incurred by the algorithm into 4 groups, depending on which part of the algorithm triggers them:

$T_1(N)$: The cost of loading buffer blocks for large invocations and complete data structures for small invocations. We call these block transfers *invocation swaps*.
$T_2(N)$: The cost of priority queue operations and Insert operations on BRT's.
$T_3(N)$: The cost of Extract operations on BRT's.
$T_4(N)$: The cost of stack operations.

We prove that $T_1(N) = \mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}}\big)$, $T_2(N) = \mathcal{O}\big(\frac{N}{B}\log N\big)$, $T_3(N) = \mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}} + \frac{N}{B}\log\log N\big)$, and $T_4(N) = \mathcal{O}\big(\frac{N}{B}\log\log N\big)$. Summing these four terms, we obtain the following lemma:

Lemma 3. *Algorithm 2 incurs $\mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}} + \frac{N}{B}\log N\big)$ block transfers.*

Next we sketch how to derive the above bounds for $T_1(N)$ through $T_4(N)$. Details will be provided in the full paper.

Invocation swaps. It suffices to count the number of invocations whose parent regions have size at least B because each such invocation incurs $\mathcal{O}(1)$ block transfers, and these are the only invocations that incur block transfers. Klein et al. [19] classify invocations as *truncated* or *non-truncated*. The former is an invocation that returns because all edges in the region have been relaxed. The latter returns because α_j recursive calls on children of the current region have been made. We denote the number of truncated level-j invocations by S'_j and the total number of level-j invocations by S_j. Using a non-trivial charging scheme, Klein et al. prove that the number of truncated level-j invocations is

$$S'_j \le \sum_{i \ge j} \mathcal{O}(\beta_{ij} \cdot N/\sqrt{r_i}),^1$$

where $\beta_{ij} = \prod_{k=j+1}^{i} \alpha_k$. Each non-truncated invocation at level j gives rise to α_j invocations at level $j-1$. Hence, the total number of invocations at level j is

$$S_j \le \frac{S_{j-1}}{\alpha_j} + S'_j.$$

Note that all level-0 invocations are truncated, that is, $S_0 = S'_0$. Using our choice of parameters α_i, we have $\beta_{ij} = \frac{2^{a^i/(2(a-1))}}{2^{a^j/(2(a-1))}}$. Substituting this in the expressions for S'_j and S_j, we obtain that $S'_j = \mathcal{O}(N/\sqrt{r_j})$ and $S_j = \mathcal{O}((j+1)a^{j/2}N/\sqrt{r_j})$. Since the r_j increase doubly exponentially, we have $\sum_{i \ge j} S_i = \mathcal{O}(S_j)$. Hence, it suffices to argue that, for the maximum j_0 such that $r_{j_0} \le B$, we obtain $S_{j_0} = \mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}}\big)$. For this j_0, we have $r_{j_0+1} = a^{j_0+1} \cdot 2^{a^{j_0+1}/(a-1)} > B$, that is, $r_{j_0} \ge (r_{j_0+1}/a)^{1/a} = \Omega(B^{1/a})$. On the other hand, we have $j_0 \le \log_a((a-1)\log B) = \mathcal{O}(\log\log B)$ because $r_{j_0} \le B$. Hence, $T_1(N) = \mathcal{O}(\sum_{i \ge j_0} S_i) = \mathcal{O}\left(\frac{N\sqrt{\log B}\log\log B}{B^{1/2a}}\right) = \mathcal{O}\left(\frac{N}{B^{1/2-\epsilon}}\right)$.

Priority queue operations and insertions into BRT's. The cost of a priority queue operation or an Insert operation on a BRT of a level-j region is $\mathcal{O}\big(\frac{\log r_j}{B}\big)$. The number of priority queue operations is bounded by the number of insertions into the BRT. Hence, it suffices to bound the cost of the latter.

^{1} They prove $S'_j \le \sum_{i \ge j} \mathcal{O}(\beta_{ij} \cdot Nf(r_i)/r_i)$, where $f(r_i)$ is a bound on the boundary size of every level-i region. We use optimal planar separators; hence, $f(r_i) = \mathcal{O}(\sqrt{r_i})$.

Every insertion into a BRT is the result of the relaxation of an edge (u, v). Such a relaxation triggers updates of the priorities of both out-edges of v and their ancestors and an Update-Target operation on the other in-edge of v. We argue about the cost of updates on the out-edges; the cost of the Update-Target operations can be bounded in a similar manner. In the worst case, the lowest common ancestor of edge (u, v) and an out-edge (v, w) of v is the root. Then the update of (v, w) traverses all levels in the partition and triggers one insertion at every level. Hence, the cost per level-0 invocation is at most $\sum_{j\geq 0} \mathcal{O}\big(\frac{\log r_j}{B}\big)$, which is $\mathcal{O}\big(\frac{\log r_k}{B}\big) = \mathcal{O}\big(\frac{\log N}{B}\big)$ because the graph sizes r_j increase doubly exponentially. Hence, the total cost of priority queue operations and BRT insertions is $T_2(N) = \mathcal{O}\big(\frac{S_0}{B} \log N\big) = \mathcal{O}\big(\frac{N}{B} \log N\big)$.

Extractions from BRT's. Every Extract operation on the BRT of a level-j region costs $\mathcal{O}(\log r_i + K/B)$ block transfers, where K is the number of extracted elements. Every extracted element must have been inserted before, and we have argued that every level-0 invocation triggers at most three insertions per level. The number of extract operations at level j equals the number of invocations at level $j-1$. Hence, $T_3(N) = \mathcal{O}(\sum_{i>j_0} S_{i-1} \log r_i) + \mathcal{O}\big(\frac{S_0}{B} \log\log N\big)$. Using similar arguments as the ones we used to bound the cost of invocation swaps, this simplifies to $T_3(N) = \mathcal{O}(S_{j_0} \log B^a) + \mathcal{O}\big(\frac{N}{B} \log\log N\big) = \mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}} + \frac{N}{B} \log\log N\big)$.

Stack operations. Every update incurs at most a constant number of stack operations at every level. Hence, the cost of all stack operations is $T_4(N) = \mathcal{O}\big(\frac{S_0}{B} \log\log N\big) = \mathcal{O}\big(\frac{N}{B} \log\log N\big)$.

7 Open Problems

The most interesting open questions are the following: (1) Can the required separator decomposition be computed cache-obliviously? (2) Can the $\mathcal{O}\big(\frac{N}{B^{1/2-\epsilon}}\big)$-term in the complexity of the algorithm be reduced to $o(N/B^{1/2})$? (3) Can the $\mathcal{O}\big(\frac{N}{B} \log N\big)$-term be reduced to $\mathcal{O}(N/B)$ (the equivalent of the linear time bound obtained in [19]).

To answer question (1), we believe that the contraction-based separator algorithm of [20] can be extended to compute the desired partition: At every contraction level, use the (by a constant factor suboptimal) separator produced at the previous level to compute a BFS-tree of the current level. Then use this BFS-tree to obtain an optimal separator for the current level (which is constant-factor suboptimal for the next level), and iterate.

The answer to question (2) may be yes, but not using purely separator-based ideas because these algorithms trade-off the number of times a region is loaded into cache against a certain amount of wasteful computation in each region. While the I/O-model can ignore the latter, a cache-oblivious algorithm cannot and must therefore balance this trade-off, which is what our choice of parameters in the algorithm of [19] achieves. The answer to question (3) is most definitely no because this would violate the $\Omega(\mathrm{perm}(N))$ lower bound for shortest paths [13], which is $\Omega(\mathrm{sort}(N))$ in the cache-oblivious model.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, pp. 1116–1127, 1988.
2. L. Arge, M. A. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. *Proc. 34th STOC*, pp. 268–276, 2002.
3. L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Alg.*, 53:186–206, 2004.
4. L. Arge, U. Meyer, and L. Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. *Proc. 31st ICALP*, LNCS 3142, pp. 146–157. Springer-Verlag, 2004.
5. L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *J. Graph Alg. and Appl.*, 7(2):105–129, 2003.
6. L. Arge and L. Toma. Simplified external memory algorithms for planar DAGs. *Proc. 9th SWAT*, LNCS 3111, pp. 493–503. Springer-Verlag, 2004.
7. L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. *Proc. 15th SPAA*, pp. 85–93. 2003.
8. L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. *Proc. 44th FOCS*, pp. 261–270, 2003.
9. G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. *Proc. 29th ICALP*, LNCS 2380, pp. 426–438. Springer-Verlag, 2002.
10. G. S. Brodal and R. Fagerberg. Funnel heap—a cache oblivious priority queue. *Proc. 13th ISAAC*, LNCS 2518, pp. 219–228. Springer-Verlag, 2002.
11. G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. *Proc. 35th STOC*, pp. 307–315, 2003.
12. G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. *Proc. 9th SWAT*, LNCS 3111, pp. 480–492. Springer-Verlag, 2004.
13. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. *Proc. 6th SODA*, pp. 139–149, 1995.
14. R. A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. *Proc. 16th SPAA*, pp. 245–254, 2004.
15. E. W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.
16. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, near linear time. *Proc. 42nd FOCS*, pp. 232–241, 2001.
17. G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comp.*, 16:1004–1022, 1987.
18. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *Proc. 40th FOCS*, pp. 285–297, 1999.
19. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest path algorithms for planar graphs. *J. Comp. Sys. Sci.*, 55:3–23, 1997.
20. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. *Proc. 13th SODA*, pp. 372–381, 2002.
21. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. *Proc. 13th SODA*, pp. 267–276, 2002.
22. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.
23. M. Thorup. Floats, integers, and single source shortest paths. *J. Alg.*, 35:189–201, 2000.