On External Memory Graph Traversal

Adam L. Buchsbaum* Michael Goldwasser† Suresh Venkatasubramanian*
Jeffery R. Westbrook*

Abstract

We describe a new external memory data structure, the *buffered repository tree*, and use it to provide the first non-trivial external memory algorithm for directed breadth-first search (BFS) and an improved external algorithm for directed depth-first search. We also demonstrate the equivalence of various formulations of external undirected BFS, and we use these to give the first I/O-optimal BFS algorithm for undirected trees.

1 Introduction

We use the standard I/O model [1], which counts disk accesses incurred by an algorithm, using the following parameters: M is the memory size, B is the block size, and we assume that $B \leq M/2$. Define $sort(N) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$, the number of I/Os needed to sort N items, and $scan(N) = \lceil N/B \rceil$, the number of I/Os needed to transfer N contiguous items between disk and internal memory.

Given a graph with V vertices and E edges, the model applies when $M < V \leq E$. For undirected graphs, Kameshwar and Ranade [5] give an $O(V + \frac{E}{V}sort(V))$ I/O algorithm for breadth-first search (BFS); Kumar and Schwabe, in an unpublished fuller version of their conference paper [6], give an $O((V + \frac{E}{B})\log_2\frac{V}{B} + sort(E))$ I/O algorithm for depth-first search (DFS). Chiang et al. [3] give an $O(V + \frac{V}{M}scan(E) + sort(E))$ I/O directed DFS algorithm.

1.1 Our Results. We present the first non-trivial algorithm for external *directed* BFS and an improved scheme for external *directed* DFS, each using $O((V + \frac{E}{B})\log_2\frac{V}{B} + sort(E))$ I/Os. Our algorithms outperform the naive $O(E)$ I/O algorithms for all but sparse graphs, and the Chiang et al. [3] DFS algorithm for all graphs when $M = o(\frac{V}{B}/\log_2\frac{V}{B})$.

We assume the graph is given as an unordered list of arcs. The output is an ordering of the vertices by BFS or DFS number. We demonstrate the equivalence of this to other formulations in the case of undirected BFS. Using these reductions, we give the first optimal ($O(sort(V))$ I/Os) BFS algorithm for undirected trees.

*AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932; {alb,jeffw,suresh}@research.att.com.

†Dept. of Math and Comp. Science, Loyola University Chicago, 6525 N. Sheridan Rd., Chicago, IL 60626; mhg@cs.luc.edu. Research completed while at Princeton University and supported by DIMACS, an NSF Science and Technology Center.

2 Buffered repository trees

Our new data structure is the *buffered repository tree* (BRT), which improves on an auxiliary data structure used by Kumar and Schwabe [6, unpublished version]. A BRT, T, stores a multi-set of items from an ordered universe. We equate items with their keys. T is subject to the operations $insert(T, x)$, which adds item x to T, and $extract(T, k)$, which reports and deletes from T all items with key k.

We implement T as an augmented $(2, 4)$-tree. Each leaf holds up to B items. Each internal node has the standard search key(s), plus a buffer holding up to B items. We maintain the invariant that all items stored in descendent leaves and buffers of a node x obey the standard search criteria with respect to the rank of x among its sibling(s).

We maintain the root node, r, in internal memory. To insert item x, we add x to r's buffer. If the buffer overflows, we distribute B of its items to r's children appropriately, recursively distributing overflowing buffers down the tree. To perform $extract(T, k)$, we search to the leaves in T that delimit the range of items with key k. We extract all items with key k from the delimited contiguous range of leaves, plus all such items from buffers in ancestors of these leaves.

LEMMA 2.1. *A BRT, T, uses $O(N/B)$ space and admits $insert(T, x)$ in $O(\frac{1}{B}\log_2\frac{N}{B})$ amortized I/Os and $extract(T, k)$ in $O(\log_2\frac{N}{B} + S/B)$ I/Os, where N is the total number of items added over the lifetime of T and S is the size of the set returned by $extract(T, k)$.*

PROOF (SKETCH): For any node x that is the parent of leaf children, at least one child of x had at least $B/4$ items at one time, yielding the space bound. Any time a buffer is distributed to children nodes during an insert, we apportion the $O(1)$ I/Os uniformly to the B items that are distributed. Each item is thus charged $O(\frac{1}{B})$ I/Os per level in T, yielding the insert bound. Extract visits $O(\log_2\frac{N}{B} + S/B)$ nodes in the tree, spending $O(1)$ I/Os per node. □

3 External DFS and BFS

Our external directed DFS algorithm resembles the undirected BFS algorithm of Kumar and Schwabe [6], except that we use BRTs in lieu of their tournament trees to maintain a

list of arcs that should not be traversed.

The basic idea is to simulate the internal directed DFS algorithm: examine the top vertex on the stack, examine the next unexplored arc out of that vertex, and if the arc points to an undiscovered vertex, push the new vertex on the stack and iterate. The key difficulty is determining whether the next unexplored arc points to an undiscovered vertex, without doing $\Omega(1)$ I/Os per arc. To do this, when a vertex v is first discovered, we "premark" incoming arcs (x, v) by putting them, keyed by x, in a BRT, D. We also maintain a priority queue, $P(v)$, initialized to contain all outgoing arcs (v, x), each keyed by its rank in the adjacency list of v, i.e., the order that DFS scans the adjacency list.

Suppose vertex u is currently at the top of the stack, and it is time to find the next arc out of u to an undiscovered vertex. First, extract all arcs of the form (u, x) from D, i.e., all arcs with key u. Such an arc will only be in D if vertex x has been discovered between the current time and the last time u was at the top of the stack, or since the start of the algorithm if this is the first time u is at the top of the stack. Each arc (u, x) so extracted engenders a *delete*(x) operation on $P(u)$. Finally, a *delete-min* operation is applied to $P(u)$ to find the next arc (u, y) to scan. The deletions maintain the invariant that, prior to each delete-min, $P(u)$ contains precisely the arcs from u to its unexplored successors, so we can push y onto the stack and iterate, assigning y the next DFS number in the process. If $P(u)$ is empty, then u is popped and we iterate.

BFS is similar, except that the stack is replaced by a queue; y (the successor of u) is injected, not pushed; and $P(\cdot)$ can be implemented as a simple queue.

THEOREM 3.1. *DFS and BFS numbers can be assigned to a directed graph G in $O((V + \frac{E}{B}) \log_2 \frac{V}{B} + sort(E))$ I/Os.*

PROOF: We arrange the arcs of G into adjacency and reverse adjacency lists in $O(sort(E))$ I/Os. We use Arge's buffer trees [2] for the priority queues. There are E inserts and deletes and $2V$ delete-mins overall, for an I/O complexity of $O(sort(E))$ for priority queue operations. There are E inserts and $2V$ extracts in the BRT, for an overall I/O complexity of $O((V + \frac{E}{B}) \log_2 \frac{V}{B})$ for BRT operations. □

3.1 Equivalence of external BFS formulations. Kameshwar and Ranade [5] give a construction that can be extended to show an $\Omega(\frac{E}{V} sort(V))$-I/O lower bound for solving undirected BFS ordering. The following shows that various common formulations of undirected BFS are equivalent up to an edge-list sort. Similar reductions for the directed case remain open.

THEOREM 3.2. *For any undirected graph $G = (V, E)$ and designated root, the following problems are reducible to each other in $O(sort(E))$ I/Os.* **Order:** *compute a BFS order on the vertices.* **Unorient:** *compute the undirected BFS tree edges.* **Orient:** *compute the BFS tree edges, oriented from child to parent.* **Level:** *compute the BFS level of each vertex.*

PROOF (SKETCH): **Order→Orient.** Let $b : V \to [1, |V|]$ be the BFS ordering. For each vertex v, its parent in the BFS tree is the node v' such that $b(v') = \min_{(v,w) \in E} b(w)$. We can thus compute all parents in $O(sort(E))$ I/Os. **Orient→Unorient.** Trivial. **Unorient→Orient.** Duplicate each undirected BFS edge, swapping the two vertices. Compute an Euler tour on the resulting directed graph. Break the tour at the root. Apply list ranking to the resulting list, assigning a rank to each arc. For each vertex v, the least ranked incident arc comes from v's parent. Using Chiang et al.'s results [3], we can construct the Euler tour and list ranking in $O(sort(V))$ I/Os, and so the entire procedure takes $O(sort(V))$ I/Os. **Orient→Level.** This can be accomplished using an Euler tour on the vertices [4]. **Level→Order.** Given the correct order for vertices in level i of the tree and all edges between levels i and $i + 1$, we can compute the correct order for vertices in level $i + 1$ by noting that each vertex in level $i + 1$ must be a child (in a BFS tree) of the adjacent level-i vertex of least rank in the order. Each level can be processed in $O(sort(E_i) + sort(V_i))$ I/Os, where V_i is the number of vertices at level i, and E_i is the number of edges between levels i and $i + 1$. □

Theorem 3.2 immediately implies the first I/O-optimal BFS algorithm for undirected trees. We simply orient each edge to its parent, and, if desired, assign BFS numbers.

THEOREM 3.3. *BFS numbers can be assigned to an undirected, rooted tree in $O(sort(V))$ I/Os.*

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *C. ACM*, 31(8):1116–27, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O algorithms. In *Proc. 4th WADS*, volume 955 of *LNCS*, pages 334–45. Springer-Verlag, 1995.

[3] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM SODA*, pages 139–49, 1995.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[5] M. V. Kameshwar and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th ACM-SIAM SODA*, pages 687–94, 1999.

[6] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE SPDP*, pages 169–76, 1996.