Efficient Tree Layout in a Multilevel Memory Hierarchy

Michael A. Bender[1]*, Erik D. Demaine[2]**, and Martin Farach-Colton[3]***

[1] Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. bender@cs.sunysb.edu.
[2] MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA. edemaine@mit.edu.
[3] Google Inc., 2400 Bayshore Parkway, Mountain View, CA 94043, USA, and Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. farach@cs.rutgers.edu.

Abstract. We consider the problem of laying out a tree or trie in a hierarchical memory, where the tree/trie has a fixed parent/child structure. The goal is to minimize the expected number of block transfers performed during a search operation, subject to a given probability distribution on the leaves. This problem was previously considered by Gil and Itai, who show optimal but high-complexity algorithms when the block-transfer size is known. We propose a simple greedy algorithm that is within an additive constant strictly less than 1 of optimal. We also present a relaxed greedy algorithm that permits more flexibility in the layout while decreasing performance (increasing the expected number of block transfers) by only a constant factor. Finally, we extend this latter algorithm to the *cache-oblivious* setting in which the block-transfer size is unknown to the algorithm; in particular this extension solves the problem for a multilevel memory hierarchy. The query performance of the cache-oblivious layout is within a constant factor of the query performance of the optimal layout with known block size.

1 Introduction

The B-tree [4] is the classic optimal search tree for external memory, but it is only optimal when accesses are uniformly distributed. In practice, however, most distributions are nonuniform, e.g., distributions with heavy tails arise almost universally throughout computer science. Examples of nonuniform distributions include access distributions in file systems [3, 16, 19, 22].

Consequently, there is a large body of work on optimizing search trees for nonuniform distributions in a variety of contexts:

* Supported in part by HRL Laboratories, NSF Grant EIA-0112849, and Sandia National Laboratories.
** Supported in part by NSF Grant EIA-0112849.
*** Supported in part by NSF Grant CCR-9820879.

1. *Known distribution on a RAM* — optimal binary search trees [1,15] and variations [12], and Huffman codes [13].
2. *Unknown distribution on a RAM* — splay trees [14,20].
3. *Known distribution in external memory* — optimal binary search trees in the HMM model [21].
4. *Unknown distribution in external memory* — alternatives to splay trees [14].[4]

Fixed Tree Topology. Search trees frequently encode decision trees that cannot be rebalanced because the operations lack associativity. Such trees naturally arise in the context of string or geometric data, where each node represents a character in the string or a geometric predicate on the data. Examples of such structures include tries, suffix trees, Cartesian trees, k-d trees and other BSP trees, quadtrees, etc. These data structures are among the most practical in computer science. Almost always their contents are not uniformly distributed, and often these search trees are unbalanced.

How can we optimize these fixed-topology trees when the access distribution is known? On a RAM there is nothing to optimize because there is nothing to vary. In external memory, however, we can choose the layout of the tree structure in memory, that is, which nodes of the tree are stored in which blocks in memory. This problem was proposed by Gil and Itai [10,11] at ESA'95. Among other results described below, they presented a dynamic-programming algorithm for optimizing the partition of the N nodes into blocks of size B, given the probability distribution on the leaves. The algorithm runs in $O(NB^2 \log \Delta)$ time, where Δ is the maximum degree of a node, and uses $O(B \log N)$ space.

This problem is so crucial because when trees are unbalanced or distributions are skewed, there is even more advantage to a good layout. Whereas uniform distributions lead to B-trees, which save a factor of only $\lg B$ over the standard $\lg N$, the savings grow with nonuniformity in the tree. In the extreme case of a linear-height tree or a very skewed distribution we obtain a dramatic factor of B savings over a naïve memory layout.

Recently, there has been a surge of interest in data structures for multilevel memory hierarchies. In particular, Frigo, Leiserson, Prokop, and Ramachandran [9,17] introduced the notion of *cache-oblivious algorithms*, which have asymptotically optimal memory performance for all possible values of the memory-hierarchy parameters (block size and memory-level size). As a consequence, such algorithms tune automatically to arbitrary memory hierarchies and exploit arbitrarily many memory levels. Examples of cache-oblivious *data structures* include cache-oblivious B-trees [6] and its simplifications [7,8,18], cache-oblivious persistent trees [5], and cache-oblivious priority queues [2]. However, all of these data structures assume a uniform distribution on operations.

Our Results. In this paper, we design simple greedy algorithms for laying out a fixed tree structure in a hierarchical memory. The objective is to minimize the

[4] Although [14] does not explicitly state its results in the external-memory model, its approach easily applies to this scenario.

expected number of blocks visited on a root-to-leaf path, for a given probability distribution on the leaves. Our results are as follows:

1. We give a greedy algorithm whose running time is $O(N \log B)$ and whose query performance is within an additive constant strictly less than 1 of optimal.
2. We show that this algorithm is robust even when the greedy choices are within a constant factor of the locally best choices.
3. Using this greedy approach, we develop a simple cache-oblivious layout whose query performance is within a small constant factor of optimal at every level of the memory hierarchy.

Related Work. In addition to the result mentioned above, Gil and Itai [10,11] consider other algorithmic questions on tree layouts. They prove that minimizing the number of distinct blocks visited by each query is equivalent to minimizing the number of block transfers over several queries; in other words, caching blocks over multiple queries does not change the optimization criteria. Gil and Itai also consider the situation in which the total number of blocks must be minimized (the external store is expensive) and prove that optimizing the tree layout subject to this constraint is NP-hard. In contrast, with the same constraint, it is possible optimize the expected query cost within an additive 1/2 in $O(N \log N)$ time and $O(B)$ space. This algorithm is obtained by taking their polynomial-time dynamic program for the unconstrained problem and tuning it.

2 Tree Layout with Known Block Size

Define the *probability of an internal node* to be the probability that the node is on a root-to-leaf path for a randomly chosen leaf, that is, the probability of an internal node is the sum of the probabilities of the leaves in its subtree. These probabilities can be computed and stored at the nodes in linear time.

All of our algorithms are based on the following structural lemma:

Lemma 1. *There exists an optimal layout of a tree T such that within the block containing the root of T (the* root block*) the nodes form a connected subtree.*

Proof. The proof follows an exchange argument. To obtain a contradiction, suppose that in all optimal layouts the block containing the root does not form a connected subtree. Let r be the root of the tree T. Define the *root block* of a given layout to be the block containing r. Define the *root-block tree* to be the maximal-size (connected) tree rooted at r and entirely contained within the root block. Consider the optimal layout $\mathcal{L}^*$ such that the root-block tree is largest. We will exhibit a layout $\hat{\mathcal{L}}$ whose search cost is no larger and where the root-block tree contains at least one additional node. Thus, we will obtain a contradiction.

Consider nodes $u, v, w \in T$, where node u is the parent of node v, and node w is a descendant of node v; furthermore, node u is in the root-block tree (and therefore is stored in the root block), node v is stored in a different block, and node w is stored in the root block (but by definition is not in the root-block tree). Such nodes u, v, and w must exist by the inductive hypothesis.

To obtain the new layout $\hat{\mathcal{L}}$ from $\mathcal{L}^*$, we exchange the positions of v and w in memory. This exchange increases the number of nodes in the root-block tree by at least one. We show that the search cost does not increase; there are three cases. (1) The search visits neither v nor w. The expected search cost is unchanged in this case. (2) The search visits both v and w. Again the expected search cost is unchanged in this case. (3) The search visits v but not w. The search cost in this case stays the same or decreases because the block that originally housed v may no longer need to be transferred. Therefore, the search cost does not increase, and we obtain a contradiction. □

2.1 Greedy Algorithm

The *greedy algorithm* chooses the root block that maximizes the sum of the probabilities of the nodes within the block. To compute this root block, the algorithm starts with the root node, and progressively adds maximum-probability nodes adjacent to nodes already in the root block. Then the algorithm conceptually removes the root block and recurses on the remaining subtrees. The base case is reached when a subtree has at most B nodes; these nodes are all stored in a single block.

Equivalently, the greedy algorithm maximizes the expected depth of the leaves in the root block. Let p_i denote the probability of a leaf node in the root block, and let d_i denote the depth of that leaf. If we write out the sum of the probabilities of every node in the root block, the term p_i will occur for each ancestor of the corresponding leaf, that is, d_i times. Thus, we are choosing the root block to maximize $\sum_{i=1}^{\ell} p_i \cdot d_i$, which is the expected depth of a leaf.

Because the expected depth of the leaves in the root block plus the expected depth of the leaves in the remaining subtrees equals the total expected depth of the leaves in the tree, the greedy algorithm is equivalent to minimizing the expected depth of leaves in the remaining subtrees.

The greedy algorithm has the following performance:

Theorem 1. *The expected block cost of the greedy algorithm is at most the optimal expected block cost plus* $(B-1)/B < 1$.

Proof. Consider a *smooth* cost model for evaluation, in which the cost of accessing a block containing j elements is j/B instead of 1. By the properties of the greedy algorithm, the only blocks containing $j < B$ elements are *leaf blocks* (those containing all descendents of all contained nodes). This cost model only decreases the cost, so that the optimal expected smooth cost is a lower bound on the optimal expected block cost. We claim that the greedy algorithm produces a layout having the optimal expected smooth cost. Hence, the expected smooth cost of the greedy algorithm is at most the optimal expected block cost. Therefore, the expected block cost of the greedy algorithm is at most $(B-1)/B$ plus the optimal expected block cost.

Now we prove the claim. Consider an optimal layout according to the expected smooth cost. Assume by induction on the number of nodes in the tree that only the root block differs from the greedy layout. Pick for removal or *demotion* a minimum-probability leaf d in the root block, and pick for addition or *promotion* a maximum-probability child p of another leaf in the root block. Assuming the root block differs from the greedy layout, the probability of d is less than the probability of p. We claim that an "exchange," consisting of demoting d and promoting p (as described below) strictly decreases the expected smooth cost.

First consider demoting d from the root block. We push d into one of its child blocks. (Recall that d is a leaf of the root block, so its children are in different blocks.) This push causes a child block to overflow, and we recursively remove the smallest-probability leaf in that block, which has smaller probability than d. In the end, we either push a node d' into a nonempty leaf block, in which case we simply add d' to that block, or we reach a full leaf block, so that an excess node d' cannot be pushed into a child block because there are no child blocks. In the latter case, we create a new child block that just stores d'. In either case, the increase in the expected smooth cost is $\Pr[d']/B \le \Pr[d]/B$. Our re-arrangement could only be worse than the optimal, which is equal to greedy on these subproblems by induction.

Second consider promoting p to the root block. This corresponds to removing p from its previous block b. We claim that this will decrease the expected smooth cost of accessing a leaf by at least $\Pr[p]/B$. If a child of p is not in block b, then accessing every leaf below that child now costs one block access less than before, because they no longer have to route through block b (and because by the greedy strategy, block b is full). Thus, we only need to consider children of p that are in block b.

If k of p's children are in block b, then the promotion of p up out of block b effectively partitions the block into k sub-blocks. In any case, any child previously in block b ends up in a block that stores strictly less than B nodes. Thus, we can recursively add to that block (at least) the maximum-probability child of a leaf in the block. In this way, the additions propagate to all leaf blocks below p. In the end, we move the maximum-probability leaf node from every leaf block into its parent block. This move decreases the smooth cost of that leaf by at least $1/B$, because it no longer has to access the leaf block which had size at least 1. The move also decreases the smooth cost of every leaf remaining in that leaf block by at least $1/B$, because the leaf block got smaller by one element.

Thus, the total decrease in expected smooth cost from promoting p to the root block is at least $\Pr[p]/B$. Again our re-arrangement could only be worse than the optimal, which is equal to greedy on these subproblems by induction.

In total the expected smooth cost decreases by at least $(\Pr[p] - \Pr[d])/B > 0$, contradicting the assumption that we started with an optimal layout according to expected smooth cost. Hence the optimal smooth-cost layout is in fact greedy. $\square$

The greedy algorithm can be implemented in $O(N \log B)$ time on a RAM as follows. Initialize each root-block-selection phase by creating a priority queue containing only the root node. Until the root block has size B, remove the maximum-probability node from the priority queue and "add" that node. To add a node, place it into the root block, compute an order statistic to find the B maximum-probability children of the node, and add those children into the priority queue. We can get away with keeping track of only the top B children because we will select only B nodes total. Any remaining children become the roots of separate blocks. Hence, the priority queue has size at most B^2 at any moment, so accessing it costs $O(\log B)$ time. We compute order statistics only once for each node, so the total cost for computing order statistics is $O(N)$.

An interesting open problem is how to implement the greedy algorithm efficiently in a hierarchical memory given a tree in some prescribed form (e.g., a graph with pointers stored in a contiguous segment of $O(N)$ memory cells). The bound of $\Theta(N)$ follows trivially by implementing the priority queue of size B^2 as a B-tree of height 2. Can we reduce the number of memory transfers below $\Theta(N)$?

2.2 Relaxed Greedy Algorithm

For a constant $0 < \varepsilon < 1$, the *relaxed greedy algorithm* chooses nodes for the root block in a less greedy manner: at each step, it picks any node whose probability is at least ε times the highest-probability node that could be picked. Once the root block is chosen, the remaining subtrees are laid out recursively as before.

Theorem 2. *The relaxed greedy algorithm has an expected block cost of at most $1/\varepsilon$ times optimal, plus $(B-1)/B < 1$.*

Proof. In our analysis we start with a greedy layout and gradually transform it to a relaxed greedy layout. As we perform the transformation, we bound the increase in cost. In the greedy layout, we work up from the bottom of the tree, and change each block into the relaxed-greedy choice for the block. The expected smooth cost of leaf nodes is the same as before in Theorem 1.

The proof of the claim follows the exchange argument in the proof of Theorem 1. By induction suppose that only the root block of the tree has not yet been converted into the relaxed-greedy layout. Consider exchanging nodes, one at a time, from the greedy layout to the relaxed-greedy layout. Each exchange causes one highest-probability node d to be demoted from the root block, and causes one node p to be promoted, where p has probability at least ε times the highest probability.

By the argument in the proof of Theorem 1, the demotion of d causes the expected smooth cost to increase by at most $1/B$ times the probability of d; and the promotion of p causes the expected smooth cost to decrease by at least $1/B$ times the probability of p. In particular, the increase in expected smooth cost at most $\Pr[d]/B - \Pr[p]/B$. Because $\varepsilon \Pr[d] < \Pr[p]$, this quantity is at most $\left(\frac{1}{\varepsilon} - 1\right) \frac{\Pr[p]}{B} \leq \left(\frac{1}{\varepsilon} - 1\right) \frac{1}{B}$. Because there are at most B exchanges in the root block, the expected smooth cost increases by at most $(\frac{1}{\varepsilon} - 1)$. Since the original smooth cost of accessing the (entirely filled) root block was 1, there is a $1/\varepsilon$ factor increase in the smooth cost in order to change the root block from the greedy layout to the relaxed greedy layout. This proves the claim. □

3 Cache-Oblivious Tree Layout

We now present the cache-oblivious layout. We follow the greedy algorithm with unspecified block size B, repeatedly adding the highest-probability node to the root block, until the number of nodes in the root block is roughly equal to the expected number of nodes in the child subtrees (rounding so that the root block may be slightly larger). Then we recursively lay out the root block and each of the child subtrees and concatenate the resulting recursive layouts (in any order). At any level of detail, we distinguish a node as either a *root block* or a *leaf block*, depending on the rôle it played during the last split.

Theorem 3. *The cache-oblivious layout has an expected block cost of at most* 4 *times optimal, plus* 4*.*

Proof. Consider the level of detail at which every block has more than B nodes and such that, at the next finer level of detail, a further refinement of each block induces a *refined root block* with at most B nodes. Thus, the *refined child block* that is visited by a random search has an expected number of nodes that is at most B. That is, if we multiply the size of each refined child block within a block by the probability of entering that refined child block (the probability of its root), then the aggregate (expectation) is at most B. Therefore, the expected number of memory transfers within each block is at most 4: at most 2 for the refined root block (depending on the alignment with block boundaries) and at most 2 for a randomly visited refined child block.

Thus, each block has size more than B and has expected visiting cost of at most 4 memory transfers. The partition into blocks can be considered an execution of the greedy algorithm in which blocks have varying maximum size that is always more than B. The expected number of blocks visited can thus only be better than the expected block cost of greedy in which every block has size exactly B, which by Theorem 1 is at most optimal plus 1. Hence, the total expected number of memory transfers is at most 4 times optimal plus 4. □

Acknowledgment

We thank Ian Munro for many helpful discussions.

References

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

2. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.
3. M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198–212, 1991.
4. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.
5. M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, 2002. To appear.
6. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.
7. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.
8. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height (extended abstract). In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.
9. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, October 1999.
10. J. Gil and A. Itai. Packing trees. In *Proc. 3rd Annual European Symposium on Algorithms)*, pages 113–127, 1995.
11. J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.
12. T. C. Hu and P. A. Tucker. Optimal alphabetic trees for binary search. *Information Processing Letters*, 67(3):137–140, 1998.
13. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40(9):1098–1101, 1952.
14. J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proc. 11th Symposium on Discrete Algorithms*, pages 516–522, 2001.
15. D. E. Knuth. *The Art of Computer Programming, V. 3: Sorting and Searching*. Addison-Wesley, Reading, 1973.
16. J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15–24, 1985.
17. H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.
18. N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *Proc. 5th Workshop on Algorithms Engineering*, pages 67–78, 2001.
19. D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proc. 2000 USENIX Annual Technical Conference*, pages 41–54, 2000.
20. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.
21. S. Thite. Optimum binary search trees on the hierarchical memory model. Master's thesis, Department of Computer Science, University of Illinios at Urbana-Champaign, 2001.
22. W. Vogels. File system usage in Windows NT 4.0. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 93–109, 1999.