6.897: Advanced Data Structures	Spring 2003
Lecture 18 — April 30, 2003	
Prof. Erik Demaine	*Scribes: Rui Fan and Alan Leung*

1 Tree Layout Problem

We saw in Lecture 16 that B-trees allow us to perform efficient search and insert operations and minimize the number of memory transfers from external storage. But how do we minimize the number of memory transfers if we are given not only the elements in the tree, but the tree's topology also? More specifically, suppose we are given a linear piece of memory, such that using one memory access, we can read in a consecutive block of size B from the memory and subsequently work with this block free of cost. How should we store a tree of a given topology in this memory in order minimize the number of memory transfers along a root-to-leaf path? We will consider two ways to measure the number of memory transfers. The first is the maximum number of memory transfers from the root of the tree to any leaf. We denote this quantity by OPT_M. The second is to assume a distribution P on the probability of accessing any leaf, and to minimize the expected cost of a root-to-leaf path over the distribution. We denote this quantity by OPT_P. Note that, for a given tree, the optimal layout of the tree under the two cost measures may be different.

2 Basic Properties of an Optimal Layout

The problem of optimally laying out a tree in memory is the problem of how we divide up the storage of the tree into blocks. Before we attack the problem, let us consider some basic properties which the blocks of an optimal layout should have. Note that while not all optimal layouts may have these properties, there is always *some* optimal layout which does.

2.1 Convexity

The first property that an optimal layout should have is *convexity*: every block stores a connected subtree of the given tree. We will informally prove why the property holds. Suppose that it does not hold. Then there are two ways in which a block of memory can store disconnected portions of a tree, as shown in Figures 1 and 2.

The first way is if the block stores two portions of the tree, X and Y, such that X is an ancestor of Y, but the block does not store the path connecting X and Y. Let y be some node in Y, and let x be a node in the path from X to Y. Suppose we modify the block by removing y and adding x to the block. This procedure does not change the total number of nodes in the block. But we claim the number of memory transfers along any root-to-leaf path does not increase using the new blocking structure. This is because every root-to-leaf path which goes through y also goes through x. This means it takes possibly one extra memory transfer to access any leaf which is a descendant of x in the old blocking structure as compared to the new. Since this argument holds for every pair of nodes in Y and the path from X to Y, we see that in some optimal blocking, we remove enough

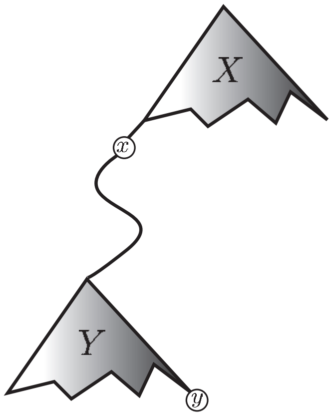

Figure 1: A block stores X and Y, but not the path between them. We can reduce the cost by removing y and adding x to the block.

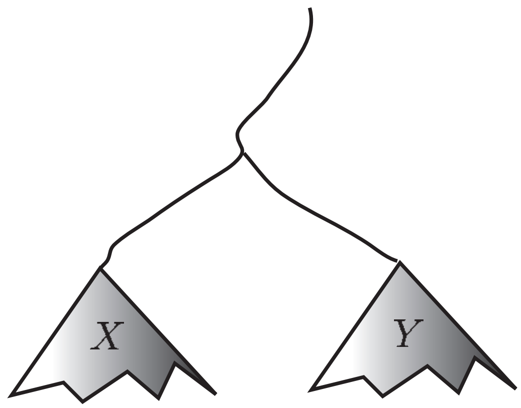

Figure 2: A block stores siblings X and Y. We can store X and Y in separate blocks without increasing the cost, since no root to leaf path goes through both X and Y.

nodes from Y to fill up the entire path from X to Y, which means we should store a connected subtree.

The other way for the block to store two disconnected portions is if X and Y are "sibling" subtrees. But in this case, we might as well store X and Y in two different memory blocks. This is because any root-to-leaf path never goes through both X and Y, so that it never saves any memory transfers to read in a block containing both X and Y.

The previous two cases show that it never costs more for blocks to store a connected subtree of the given tree, which shows there is an optimal layout with the convexity property.

2.2 Monotonicity

The next property regarding the optimal layout is *monotonicity*, which says that given a layout of the tree, if we increase B, the size of the block, both OPT_M and OPT_P do not increase. This property is obvious, since any operation using the larger B can "emulate" the one using the smaller B, just by ignoring the appropriate elements in the larger blocks.

2.3 Smoothness

The last property is *smoothness*. It says that if we increase B by a factor of 2, then both OPT_M and OPT_P can decrease by a factor of at most 2. This property will be useful when we consider the optimal tree layout in the cache-oblivious model. To see the property, consider Figure 3. Suppose we are given an optimal layout using blocks of size $2B$. By the convexity property, the block contains a subtree of the given tree (left). We choose half the nodes from this block, such that every node in the block has an ancestor in the block. Now, we put the B chosen nodes in a block, and put the unchosen nodes in some other blocks of size B (right). Each time we access the size $2B$ block, we access at most 2 of the size B blocks. This shows that the optimum cost using blocks of size B is at most twice the optimum cost using blocks of size $2B$.

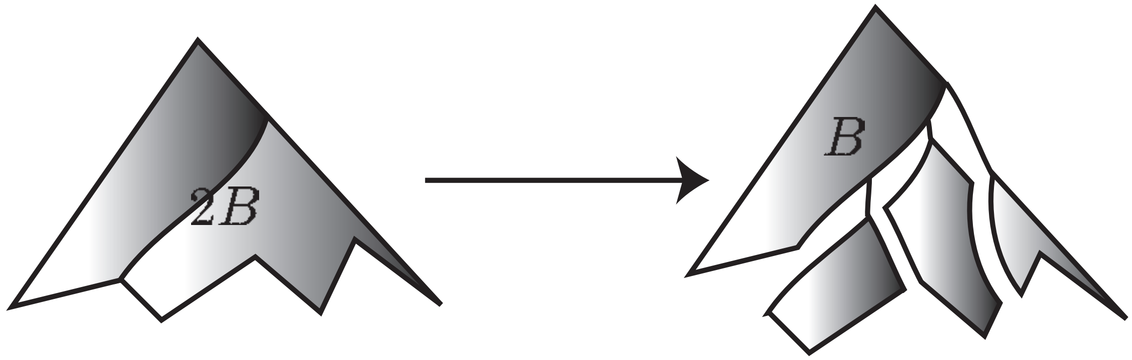

Figure 3: We break the left block into the blocks on the right. Each time we read the left block, we read at most 2 blocks on the right, so the cost increases by at most a factor of 2.

3 Optimum Layout in the External Memory Model

We first consider some algorithms for the optimum tree layout in the external memory model, where the value of B is known.

3.1 Minimizing the Maximum Cost

We now describe an algorithm for minimizing the maximum number of external memory transfers, due to [2]. The algorithm greedily builds the blocks from bottom up. First, we put each leaf into its own block, and set the *depth* of all those blocks to be 1. Then we look at the parent of some sibling blocks, and try to merge the sibling blocks and their parent into one block. We use the following rules when merging:

1. If the sibling blocks have the same depth, and they can be merged with their parent into a block of size B, then merge, and set the depth of the merged block to be the depth of the siblings.

2. If the siblings have the same depth but cannot be merged into one block, then we close off the sibling blocks, and start a new block containing only the parent. The depth of the new block is one more than the depth of the siblings.

3. If the siblings have different depths, then we close off the sibling with the least depth, and try to merge the remaining blocks with the parent.

    (a) If the merge is possible (i.e. they fit in a block of size B), then we set the depth of the merged block to be the maximum depth of any sibling.

    (b) If the merge is not possible, we close off the sibling of second least depth, and retry the merge.

4. Finally, if the parent cannot be merged with any of its child blocks, we start a new block containing only the parent, with depth equal to the maximum depth of any child sibling block.

The running time of this algorithm is $O(N)$, where N is the number of nodes in the tree. It can be shown with induction that this produces a layout which minimizes the maximum cost.

3.2 Minimizing the Expected Cost

We now describe a recent algorithm by Gil and Itai for minimizing the expected cost of accessing a leaf given the probability of accessing each leaf. The algorithm uses dynamic programming. Each subproblem has the following forms: suppose we are given the root r to a tree and a set of probabilities for accessing its leaves. What is the best layout of the tree subject to the condition that the block containing the root contains at most $C \leq B$ elements, and the other blocks contain at most B elements? We call each such problem an (r, C) problem. We will first solve these problems for the case when the input tree is binary, and later generalize to arbitrary trees.

Recall that by the convexity property for an optimal layout, each block should contain a subtree of the given tree. Given an (r, C) problem for a binary tree, this means that the block containing the root should also contain i nodes from the left subtree of r, and $C-i-1$ nodes from the right, where $0 \leq i \leq C-1$. Thus, each (r, C) problem leads to $2(C-1)$ subproblems, $\{(r_L, i), (r_R, C-i-1) : 0 \leq i \leq C-1\}$, where r_L is the left child of r, and r_R is the right child of r. We choose the i which minimizes $cost(r_L, i) + cost(r_R, C-i-1)$, and then put r in a block along with i nodes from the left subtree, and $C-i-1$ nodes from the right subtree. For the base subproblem (r', C') where the entire subtree rooted at r' contains at most C' elements, we put all those elements in one block, and define the cost of the block to be the sum of the probabilities of accessing the leaves in the block. We claim there are $O(BN)$ problems of the form (r, C). This is because there are $O(N)$ choices for r, where N is the total number of leaves in the given tree, and there are $O(B)$ choices for the value of C. Each subproblem involves a minimization over $O(B)$ subproblems. Therefore, the total running time of the algorithm is $O(B^2N)$.

We wish to extend the above algorithm to work for non-binary trees as well. Let Δ be the maximum degree of any node in the tree. We cannot just break each (r, C) problem into subproblems $(r_1, i_1), \ldots, (r_\Delta, i_\Delta)$, where $\sum_{1 \leq i \leq \Delta} r_i = C - 1$. This is because the number of such subproblems is the partition number of C, which is $O(2^{\sqrt{C}})$. Instead, we use a standard trick from dynamic programming. For each node of degree $d > 2$, we replace the node and its d children by a complete binary tree with d leaves, and give the internal nodes in the binary tree zero cost. Then we run the dynamic programming algorithm for binary trees on the transformed tree. The running time goes up only by a factor of Δ, to $O(\Delta B^2 N)$.

4 Optimal Layout in the Cache-Oblivious Model

Recall in *Lecture 14* when the Cache-Oblivious Model was first introduced, the algorithm is not aware of the values of M and B and is not allowed to find it by sampling or any other techniques. It was remarked that this model "*work well with multiple level of hierarchy.*"

We will use the "*Split-and-Refine*" techniques as introduced in [1], decreasing B through powers of 2 and applying a black-box to get the optimal external-memory-model layout for $B = \lceil\lceil N \rceil\rceil, \frac{\lceil\lceil N \rceil\rceil}{2}, \ldots, 4, 2, 1$. We notice that in this way the cost of solution changes only by a constant factor. With the *smoothness property* in section 2, it is tempted to say that at each level of detail

i, the cost is twice the cost at level of detail $i-1$. But because of the subtleties between blocks, the actual factor should be between 2 and 4. Therefore at level of detail 0, $B = \lceil\lceil N \rceil\rceil \Rightarrow$ cost $=$ 1; and at level of detail 1, the cost is between 2 and 4. In general, $2\times$ cost at level of detail $i-1 \leq$ cost at level of detail $i \leq 4\times$ cost at level of detail $i-1$

We refine the structure by cutting each block in level of detail i according to the partition at levels of detail i or lower such that these levels altogether form a recursive structure. We then store these chunks consecutively in memory.

Finally, we sort the nodes, in descending order of significance, by the keys $(L_1, L_2, \ldots, L_{\bigcirc(\log N)})$ where L_i is the block label at level i. (i.e. L_1 is the most significant key while $L_{\bigcirc(\log N)}$ is least significant.)

5 Cost Analysis

5.1 Unrefined level of detail

Suppose $L-1$ is the highest level of detail such that block size is greater than the true value of B, i.e. block size is less than the true value of B at level of detail L or higher. We then obtain *a lower bound for the optimal layout cost* by using *the monotonicity property*: true value of B $\leq$ block size at level of detail $L-1 \Rightarrow$ optimal layout cost for true value of $B \geq$ cost at level of detail $L-1$.

As explained in the previous section, cost at level of detail $L \leq 4\times$ cost at level $L-1$. Here however, because of alignment problems, an extra factor of 2 is introduced: cost at level of detail $L \leq 8\times$ cost at level of detail $L-1 \leq 8\times$ optimal layout cost for true value of B

5.2 Refined level of detail(expected cost)

The analysis turns out to be easier if we consider the edges between different blocks, and we can define 2 random variables:

X_i denotes the number of memory transfers in layout from undefined level of detail i along random root-to-leaf path. This is the same as 1 plus the number of edges straddling different blocks at level of detail i in random root-to-leaf path.

X denotes the number of memory transfers in cache-oblivious layout for random root-to-leaf path.

Total cost $= X = (X_1 - 1) + (X_2 - 1) + \cdots + (X_L - 1) + 1 = X_1 + X_2 + \cdots + X_L - (L-1)$

By linearity of expectation, $E[X] = E[X_1] + E[X_2] + \cdots + E[X_l] - (L-1) \leq E[X_1] + \cdots + E[X_L]$

The sum is a geometric serious because by construction, $E[X_i] \geq 2 \times E[X_{i-1}]$

$\Rightarrow E[X] \leq 2 \times E[X_L] \leq 2 \times (8 \times E[\text{optimal layout cost}])$ *(from unrefined analysis)*

$= 16 \times E[\text{optimal layout cost}]$

5.3 Maximum Cost

The analysis is very similar to the above except we need to replace X_i with non-random variable and $E[X_i]$ with X_i. As a result, we do not have linearity of expectation but the rest is identical.

5.4 Direction for Further Investigation

The first possibility is we could dynamize the problem, namely analyzing the cost when the structure is made dynamic. The second possibility is we could use self-adjusting techniques to guess the distribution of the structure so that we can get a better estimate of the number of memory transfers in the cost analysis.

References

[1] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010, November 2002. `http://www.arXiv.org/abs/cs.DS/0211010`.

[2] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391. ACM Press, 1996.