Worst-Case Optimal Tree Layout in a Memory Hierarchy

Erik D. Demaine∗† John Iacono‡† Stefan Langerman§

Abstract

Consider laying out a fixed-topology tree of \(N\) nodes into external memory with block size \(B\) so as to minimize the worst-case number of block memory transfers required to traverse a path from the root to a node of depth \(D\). We prove that the optimal number of memory transfers is

\[
\Theta \left( \begin{cases} 
\frac{D}{\lg(1+B)} & \text{when } D = O(\lg N) \\
\frac{\lg N}{\lg \left(1 + \frac{B \lg N}{D}\right)} & \text{when } D = \Omega(\lg N) \text{ and } D = O(B \lg N) \\
\frac{D}{B} & \text{when } D = \Omega(B \lg N)
\end{cases} \right),
\]

This bound can be achieved even when \(B\) is unknown to the (cache-oblivious) layout algorithm.

1 Introduction

Trees can have a meaningful topology in the sense that edges carry a specific meaning—such as letters from an alphabet in a suffix tree or trie—and consequently nodes cannot be freely rebalanced. Nontrivial trees also do not fit in the cache closest to the processor, so a natural problem is to lay out (store) a tree in a way that minimizes the cost of a root-to-node traversal in a multilevel memory hierarchy. Here we consider efficient algorithms for laying out a static fixed-topology binary tree in the external-memory and cache-oblivious memory-hierarchy models.

The \emph{external-memory model} [AV88] (or I/O model or Disk Access Model) defines a memory hierarchy of two levels: one level is fast but has limited size, \(M\), and the other level is...
slow but has unlimited size. Data can be transferred between the two levels in aligned blocks of size $B$, and an algorithm’s performance in this model is the number of such memory transfers. An external-memory algorithm may be parameterized by $B$ and $M$. The cache-oblivious model \cite{FLPR99} requires the additional property that the algorithm is not parameterized by $B$ or $M$, though the number of memory transfers (and analysis) still depend on these parameters. One consequence of this property is that an optimal cache-oblivious algorithm is simultaneously optimal between every pair of levels in every possible memory hierarchy.

The general objective in a tree-layout problem is to store the $N$ nodes of a static fixed-topology tree in a linear array so as to minimize the number of memory transfers incurred by visiting the nodes in order along a path starting at the root of the tree. Each node must be stored exactly once. The specific goal in a tree-layout problem varies depending on the relative importance of the memory-transfer cost of different root-to-node paths. (It is impossible to minimize the number of memory transfers along every root-to-node path simultaneously.)

Tree-layout problems have been considered before. Clark and Munro \cite{CM96} give a linear-time algorithm to find an external-memory tree layout with the minimum worst-case number of memory transfers along all root-to-leaf paths. Gil and Itai \cite{GI99} give a polynomial-time algorithm to find an external-memory tree layout with the minimum expected number of memory transfers among a randomly selected root-to-leaf path, given a fixed independent probability distribution on the leaves. Alstrup et al. \cite{ABD04} give a general transformation from external-memory tree layouts to cache-oblivious tree layouts. In particular, they obtain polynomial-time algorithms to find a cache-oblivious layout with minimum worst-case or expected number of memory transfers along a root-to-leaf path, up to constant factors.

We consider the natural parameterization of the tree-layout problem by the length $D$ of the root-to-node path, i.e., the maximum depth $D$ of the accessed nodes. Without such a parameterization, the best worst-case bound that can be stated over all trees is $O(\lceil N/B \rceil)$ memory transfers, because the tree might be a path. Parameterized by $D$, the worst-case cost can be substantially better, depending on the relationship between $N$, $B$, and $D$. We characterize the worst-case number of memory transfers incurred by a root-to-node path in a binary tree, over all possible values of these parameters, as

$$\Theta \left( \begin{array}{c} D \\ \frac{1}{\lg (1+B)} \\ \frac{\lg N}{\lg (1+B)} \\ \frac{D}{B} \end{array} \right) \begin{cases} \\ \text{when } D = O(\lg N) \\ \text{when } D = \Omega(\lg N) \text{ and } D = O(B \lg N) \\ \text{when } D = \Omega(B \lg N) \end{cases}.$$  

This characterization consists of an external-memory and a cache-oblivious layout algorithm, and a matching worst-case lower bound. The external-memory layout algorithm runs in $O(N)$ time, and the cache-oblivious layout algorithm runs in $\tilde{O}(N \lg N)$ time. These construction times are measured as CPU time on a RAM; the same upper bounds also hold on the number of memory transfers. As in previous work, we do not know how to guarantee a substantially smaller number of memory transfers during construction, because on input the tree might be scattered throughout memory.
2 Upper Bound

Our layout algorithm consists of two phases. The first phase is simple and achieves the desired bound for $D = O(\lg N)$ without significantly raising the cost for larger $D$. The second phase is more complicated, particularly in the analysis, and achieves the desired bound for $D = \Omega(\lg N)$. Both phases run in $O(N)$ time.

2.1 Phase 1

The first part of our layout simply stores the first $\Theta(\lg N)$ levels according to a B-tree clustering, as if those levels contained a perfect binary tree. More precisely, the first block in the layout consists of the $\leq B$ nodes in the topmost $\lfloor \lg(B + 1) \rfloor$ levels of the binary tree. Conceptually removing these nodes from the tree leaves $O(B)$ disjoint trees which we lay out recursively, stopping once the topmost $c \lg N$ levels have been laid out, for any fixed $c > 0$.

This phase defines a layout for a subtree of the tree, which we call the phase-1 tree. The remaining nodes form a forest of nodes to be laid out in the second phase. We call each connected tree of this forest a phase-2 tree.

The number of memory blocks along any root-to-node path within the phase-1 tree, i.e., of length $D \leq c \lg N$, is $\Theta(D/\lg(B + 1))$. More generally, any root-to-node path incurs a cost of $\Theta(\min\{D, \lg N\}/\lg(B + 1))$ within the phase-1 tree, i.e., for the first $c \lg N$ nodes.

2.2 Phase 2: Layout Algorithm

The second phase defines a layout for each phase-2 tree, i.e., for each connected tree of nodes not laid out during the first phase.

For a node $x$ in the tree, let $w(x)$ be the weight of $x$, i.e., the number of nodes in the subtree rooted at node $x$. Let $\ell(x)$ and $r(x)$ be the left and right children of node $x$, respectively. If $x$ lacks a child, $\ell(x)$ or $r(x)$ is a null node whose weight is defined to be 0.

For a simpler recursion, we consider a generalized form of the layout problem where the goal is to lay out the subtree rooted at a node $x$ into blocks such that the block containing the root of the tree is constrained to have at most $A$ nodes, for some nonnegative integer $A \leq B$, while all other blocks can have up to $B$ nodes. This restriction represents the situation when $B - A$ nodes have already been placed in the root block (in the caller to the recursion), so space for only $A$ nodes remains.

Our algorithm chooses a set $K(x, A)$ of nodes to store into the root block by placing the root $x$ and divvying up the remaining $A - 1$ nodes of space among the two children subtrees of $x$ proportionally according to weight. More precisely, $K(x, A)$ is defined recursively as follows:

$$K(x, A) = \begin{cases} \emptyset & \text{if } A < 1, \\ \{x\} \cup K[\ell(x), (A - 1) \cdot w(\ell(x))/w(x)] \\ \cup K[r(x), (A - 1) \cdot w(r(x))/w(x)] & \text{otherwise.} \end{cases}$$

Because $w(x) = 1 + w(\ell(x)) + w(r(x))$, $|K(x, A)| \leq A$. Also, for positive $A$, $K(x, A)$ always includes the root node $x$ itself.

At the top level of recursion, the algorithm creates a root block $K(r, B)$, where $r$ is the root of the phase-2 tree $T$, as the first node in the layout of that tree $T$. Then the algorithm
recursively lays out the trees in the forest \( T - K(r, B) \), starting with root blocks of \( K(r', B) \) for each child \( r' \) of a node in \( K(r, B) \) that is not in \( K(r, B) \).

### 2.3 Phase 2: Analysis

Within this analysis, let \( D \) denote the depth of the path within this phase-2 tree \( T (\Theta(\lg N) \) less than the global notion of \( D ) \). Define the density \( p(x) \) of a node \( x \) to be \( w(x)/w(r) \) where \( r \) is the root of the phase-2 tree \( T \). In other words, the density of \( x \) measures the fraction of the entire tree within the subtree rooted at the node \( x \). Let \( T_x \) denote the subtree rooted at \( x \).

Consider a (downward) root-to-node path \( x_0, x_1, \ldots, x_k \) where \( x_0 \) is the root of the tree. Define \( p_i = p(x_i) \) for \( 0 \leq i \leq k \), and define \( q_i = p_i/p_{i-1} \) for \( 1 \leq i \leq k \). Thus \( p_i = p_0 q_1 q_2 \cdots q_i = q_1 q_2 \cdots q_i \) because \( p_0 = 1 \). If \( x_k \) is in the block containing the root \( x_0 \), then the number \( m_k \) of nodes from \( T_{x_k} \) that the algorithm places in that block is given by the recurrence

\[
\begin{align*}
m_0 &= B \\
m_k &= (m_{k-1} - 1) q_k
\end{align*}
\]

which solves to

\[
\begin{align*}
m_k &= (\cdots((B - 1)q_1 - 1)q_2 - 1)q_3 \cdots - 1)q_{k-1} - 1)q_k \\
&= (Bq_1q_2 \cdots q_k) - (q_1q_2 \cdots q_k) - (q_2q_3 \cdots q_k) - \cdots - (q_{k-1}q_k) - (q_k) \\
&= Bp_k - p_k - \frac{p_k}{p_1} - \cdots - \frac{p_k}{p_{k-1}}.
\end{align*}
\]

This number is at least 1 precisely when there is room for \( x_k \) in the block containing the root \( x_0 \). Thus, if \( x_k \) is not in the block containing the root \( x_0 \), then we must have the opposite:

\[
Bp_k - p_k - \frac{p_k}{p_1} - \cdots - \frac{p_k}{p_{k-1}} - \frac{p_k}{p_{k-1}} < 1,
\]

i.e.,

\[
p_k + \frac{p_k}{p_1} + \cdots + \frac{p_k}{p_{k-2}} + \frac{p_k}{p_{k-1}} > Bp_k - 1.
\]

Because \( p_0 \geq p_1 \geq \cdots \geq p_k \), each term \( p_k/p_i \) on the left-hand side is at most 1, so the left-hand side is at most \( k \). Therefore \( k > Bp_k - 1 \).

Let \( \text{cost}_B(N, D) = \text{cost}(N, D) \) denote the number of memory blocks of size \( B \) visited along a worst-case root-to-node path of length \( D \) in a tree of \( N \) nodes laid out according to our algorithm. Certainly \( \text{cost}(N, D) \) is nondecreasing in \( N \) and \( D \). Suppose the root-to-node path visits nodes in the order \( x_0, x_1, \ldots, x_k, \ldots \), with \( x_k \) being the first node outside the block containing the root node. By the analysis above,

\[
\begin{align*}
\text{cost}(N, D) &= \text{cost}(Np_k, D - k) + 1 \\
&\leq \text{cost}(Np_k, D - p_k B + 1) + 1.
\end{align*}
\]
This inequality is a recurrence that provides an upper bound on cost\((N, D)\). The base cases are cost\((1, D) = 1\) and cost\((N, 0) = 1\). In the remainder of this section, we solve this recurrence.

Define \(x_k, x_{k+1}, x_{k+2}, \ldots, x_k\) to be the first node within each memory block visited along the root-to-node path. Thus \(x_{kj}\) is the root of the subtree that formed the \(j\)th block, so \(x_{k0}\) is the root of the tree, and \(k_1 = k\). As before, define \(p_{kj} = p(x_{kj})\). Now we can expand the recurrence \(t\) times:

\[
\text{cost}(N, D) \leq \text{cost} \left( N \prod_{i=1}^{t} p_{ki}, D - B \sum_{i=1}^{t} p_{ki} + t \right) + t.
\]

So the cost\((N, D)\) recursion terminates when

\[
\prod_{i=1}^{t} p_{ki} \leq \frac{1}{N} \quad \text{or} \quad \sum_{i=1}^{t} p_{ki} \geq \frac{D + t}{B},
\]

whichever comes first. Because \(t \leq D\), the recursion must terminate once

\[
\prod_{i=1}^{t} p_{ki} \leq \frac{1}{N} \quad \text{or} \quad \sum_{i=1}^{t} p_{ki} \geq \frac{2D}{B},
\]

whichever comes first.

Our goal is to find an upper bound on the maximum value of \(t\) at which the recursion could terminate, because \(t+1\) is the number of memory transfers incurred. Define \(p\) to be the average of the \(p_{ki}\)'s, \((p_{k1} + \cdots + p_{kt})/t\). In the termination condition, the product \(\prod_{i=1}^{t} p_{ki}\) is at most \(\prod_{i=1}^{t} p\) because the product of terms with a fixed sum is maximized when the terms are equal; and the sum \(\sum_{i=1}^{t} p_{ki}\) is equal to \(\sum_{i=1}^{t} p\). Thus the following termination condition happens later than the original termination condition:

\[
\prod_{i=1}^{t} p \leq \frac{1}{N} \quad \text{or} \quad \sum_{i=1}^{t} p \geq \frac{2D}{B}.
\]

Therefore, by obtaining a worst-case upper bound on \(t\) with this termination condition, we also obtain a worst-case upper bound on \(t\) with the original termination condition.

Now the cost\((N, D)\) recursion terminates when

\[
p^t \leq \frac{1}{N} \quad \text{or} \quad tp \geq \frac{2D}{B},
\]

i.e., when

\[
t \geq \frac{\lg N}{\lg(1/p)} \quad \text{or} \quad t \geq \frac{2D}{Bp}.
\]

Thus we obtain the following upper bound on the number of memory transfers along this path:

\[
t + 1 \leq \min \left\{ \frac{\lg N}{\lg(1/p)}, \frac{2D}{Bp} \right\} + 1.
\]
Maximizing this bound with respect to $p$ gives us an upper bound irrespective of $p$. The
maximum value is achieved when either $p = 0$, $p = 1$, or the two terms in the min are equal.
At $p = 0$, the bound is 0, so this is never the maximum. At $p = 1$, the bound is $2D/B$. The
two terms in the min are equal when, by cross-multiplying,

$$Bp \lg N = 2D \lg(1/p),$$

i.e.,

$$\frac{1}{p} \lg \frac{1}{p} = \frac{B \lg N}{2D},$$

or asymptotically

$$\frac{1}{p} = \Theta \left( \frac{B \lg N}{\lg \left(2 + \frac{B \lg N}{D}\right)} \right).$$

In this case, the min terms are

$$\Theta \left( \frac{\lg N}{\lg \left(2 + \frac{B \lg N}{D}\right)} \right).$$

Therefore, the upper bound is

$$\max \left\{ O \left( \frac{\lg N}{\lg \left(2 + \frac{B \lg N}{D}\right)} \right), \frac{D}{B} \right\},$$

or

$$O \left( \frac{\lg N}{\lg \left(2 + \frac{B \lg N}{D}\right)} + \frac{D}{B} \right).$$

2.4 Putting It Together

The total number of memory transfers is the sum over the first and second phases. If
$D \leq c \lg N$, only the first phase plays a role, and the cost is $O(D/\lg(B+1))$. If $D > c \lg N$,
the cost is the sum

$$O \left( \frac{c \lg N}{\lg(B+1)} + \frac{\lg N}{\lg \left(2 + \frac{B \lg N}{D-c \lg N}\right)} + \frac{D-c \lg N}{B} \right),$$

which is at most

$$O \left( \frac{c \lg N}{\lg(B+1)} + \frac{\lg N}{\lg \left(2 + \frac{B \lg N}{B}\right)} + \frac{D}{B} \right).$$

Because $D = \Omega(\lg N)$, the denominator of the second term is at most $\lg(B+1)$, so the first
term is always at most the second term up to constant factors. Thus we focus on the second
and third terms. If $D = X \lg N$, then the second term is $O((\lg N)/\lg(2 + B/X))$ and the
third term is $O((X \lg N)/B) = O((\lg N)/(B/X))$. For $X = O(B)$, the second term divides
$\lg N$ by $\Theta(\lg(B/X))$, while the third term divides $\lg N$ by $\Theta(B/X)$. Thus the second term
is larger up to constant factors for $X = O(B)$. For $X = Ω(B)$, the second term is $O(\lg N)$, while the third term is $O((X/B) \lg N)$, which is larger up to constant factors.

In summary, the first term dominates when $D = O(\lg N)$, the second term dominates when $D = Ω(\lg N)$ and $D = O(B \lg N)$, and the third term dominates when $D = Ω(B \lg N)$. Therefore we obtain the following overall bound:

**Theorem 1** Given $B$ and a fixed-topology tree on $N$ nodes, we can compute in $O(N)$ time an external-memory tree layout with block size $B$ in which the number of memory transfers incurred along a root-to-node path of length $D$ is

$$
O \left( \begin{cases} 
\frac{D}{\lg(1+B)} & \text{when } D = O(\lg N) \\
\frac{\lg N}{\lg (1+\frac{B\lg N}{D})} & \text{when } D = Ω(\lg N) \text{ and } D = O(B \lg N) \\
\frac{D}{B} & \text{when } D = Ω(B \lg N)
\end{cases} \right).
$$

### 2.5 Cache-Oblivious Layout

Our external-memory layout is parameterized by $B$. This layout can be transformed into a single cache-oblivious layout that is independent of $B$.

We use a general transformation from external-memory tree layouts to cache-oblivious tree layouts by Alstrup et al. [ABD+04]. This transformation takes as input an arbitrary external-memory tree-layout algorithm as a black box, and produces a cache-oblivious layout with approximately equal performance. More precisely, the number of memory transfers incurred along any root-to-node path in the cache-oblivious layout is at most a constant factor larger than the external-memory layout constructed with the machine’s true value of $B$. The cache-oblivious layout algorithm applies the external-memory layout black box for $Θ(\lg N)$ values of $B$, and combines these layouts into a single linear order of the nodes.

If we apply this general transformation to our external-memory layout algorithm, we obtain a cache-oblivious layout algorithm with the desired properties. Because the number of memory transfers along every root-to-node path increases by at most a constant factor, our worst-case bound also applies to the cache-oblivious layout. Because our external-memory layout requires $O(N)$ construction time for a particular value of $B$, the total construction time of the resulting cache-oblivious layout is $O(N \lg N)$.

**Theorem 2** Given a fixed-topology tree on $N$ nodes, we can compute in $O(N \lg N)$ time a cache-oblivious tree layout in which the number of memory transfers incurred along a root-to-node path of length $D$ satisfies the same bound as Theorem 1.

---

1. A minor detail is that Alstrup et al. [ABD+04] assume that all accesses are to leaves instead of arbitrary nodes. This difference is not essential: adding a leaf child to every nonleaf node and treating these two nodes as equivalent allows us to assume that all accesses are to leaves.
Figure 1: The recursive lower-bound construction: a complete binary tree with $1/p$ leaves attached to $1/p$ paths of length $pB$, each attached to a recursive construction.

3 Lower Bound

For $D \leq \lg(N + 1)$, the perfectly balanced binary tree on $N$ nodes gives a worst-case lower bound of $\Omega(D/\lg B)$ memory transfers. For all $D$, any root-to-node path of length $D$ requires at least $D/B$ memory transfers just to read the $D$ nodes along the path. Thus we are left with proving a lower bound for the case when $D = \Omega(\lg N)$ and $D = O(B \lg N)$.

This lower-bound construction essentially mimics the worst-case behavior predicted in Section 2.3. We set $p$ to be the solution to Equation 1, i.e., to $Bp \lg N = D \lg(1/p)$. Because $D = \Omega(\lg N)$, this equation implies that $Bp = \Omega(\lg(1/p))$.

The asymptotic solution for $1/p$ is given by Equation 2

$$\frac{1}{p} = \Theta\left(\frac{B \lg N}{D} \cdot \frac{\lg(1/p)}{\lg(2 + \frac{B \lg N}{D})}\right).$$

Using this value of $p$, we build a tree of slightly more than $B$ nodes, as shown in Figure 1, that partitions the space of nodes into $1/p$ fractions of $p$. We repeat this tree construction recursively in each of the children subtrees stopping at the height that results in $N$ nodes.

Consider any external-memory layout of the tree. Because each tree construction has more than $B$ nodes, it cannot fit in a block. Thus every tree construction has at least one leaf that is not in the same block as the root. Hence, for any $k \geq 1$, there is a root-to-node path that incurs at least $k$ memory transfers by visiting $k$ tree constructions. Such a path has length $D = O(k [pB + \lg(1/p)])$, which is $O(kpB)$ by Equation 3. Therefore

$$k = \Omega\left(\frac{D}{pB}\right) = \Omega\left(\frac{\lg N}{\lg(2 + \frac{B \lg N}{D})}\right).$$
Theorem 3 For any values of $N$, $B$, and $D$, there is a fixed-topology tree on $N$ nodes in which every external-memory layout with block size $B$ incurs

$$\Omega \begin{cases} 
\frac{D}{\lg(1+B)} & \text{when } D = O(\lg N) \\
\frac{\lg N}{\lg \left(1+\frac{B \lg N}{D}\right)} & \text{when } D = \Omega(\lg N) \text{ and } D = O(B \lg N) \\
\frac{D}{B} & \text{when } D = \Omega(B \lg N)
\end{cases}$$

memory transfers along some root-to-node path of length $D$.

Acknowledgments

This research was initiated during the Seminar on Cache-Oblivious and Cache-Aware Algorithms held at Schloss Dagstuhl in July 2004. We thank Jeff Erickson and J. Ian Munro for many helpful discussions during that seminar. We also thank Paulo Ferragina for early discussions on this problem.

References

[ABD+04] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010, 2004. [http://www.arXiv.org/abs/cs.DS/0211010]

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. Communications of the ACM, 31(9):1116–1127, September 1988.

[CM96] 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, Atlanta, January 1996.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pages 285–297, New York, October 1999.

[GI99] Joseph Gil and Alon Itai. How to pack trees. Journal of Algorithms, 32(2):108–132, 1999.