DIMACS Technical Report 93-30
May 1993

Lazy Structure Sharing for Query Optimization

by

Adam L. Buchsbaum[1,2]
Dept. of Computer Science
Princeton University

Rajamani Sundar[3]
Dept. of Computer Science and Automation
Indian Institute of Science

Robert E. Tarjan[4,5]
Dept. of Computer Science
Princeton University
and
NEC Research Institute

[1]Affiliated Graduate Student Member.
[2]Supported by a Fannie and John Hertz Foundation fellowship and National Science Foundation Grant No. CCR-8920505.
[3]Work completed while at Princeton University and DIMACS.
[4]Permanent Member.
[5]Research at Princeton University partially supported by the National Science Foundation, Grant No. CCR-8920505, the Office of Naval Research, Contract No. N00014-91-J-1463.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

We study *lazy structure sharing* as a tool for optimizing equivalence testing on complex data types. We investigate a number of strategies for a restricted case of the problem and provide upper and lower bounds on their performance (how quickly they effect ideal configurations of our data structure). In most cases, the bounds provide nontrivial improvements over the naïve linear-time equivalence-testing strategy that employs no optimization.

1 Introduction

In his Ph.D. thesis, Pugh [8] introduces *lazy structure sharing*. Upon finding two complex data types to be equivalent, the underlying data structures are updated to "share structure." In this fashion, future equivalence queries involving the same values take less time. For example, if two linear lists are determined to be "equivalent," the head of one list can be made to point to the first element of the other. Successive queries on the two heads will terminate immediately after finding that they both point to the same physical element. Further, each element in the first list can be made to point to the corresponding element's successor in the second, thus optimizing future sublist equivalence tests. If instead of linear lists one considers a data structure of nodes of outdegree two, the problem becomes akin to testing S-expression equivalence in LISP [1]. Pugh [8] considers lazy structure sharing as an alternative to *hashed consing* [1, 10] for optimizing S-expression equivalence testing.

We consider some strategies proposed by Pugh on a restricted case of lazy structure sharing, in which all nodes are of outdegree zero or one—this is the *list equivalence problem*. We also investigate a new strategy on the general case of the problem. Pugh [8] states that the theoretical properties of lazy structure sharing are not well understood. Indeed, while lazy structure sharing has been used to implement unification of first-order terms [3], theorem provers [2, 6], and computational linguistics applications [4, 5, 7], almost all of these works provide only empirical (if any) observations about its effects. Only Cordin and Bidoit [3] provide any asymptotic analysis of the effect of lazy structure sharing; they consider a variant similar to our *total order strategy* (see below). We are thus motivated to consider the simplest case of the problem.

In the following sections, we explore four strategies for lazy structure sharing: the *total order*, *reference count*, *ancestor count*, and *phantom pointer* methods. We show that each strategy, except the total order method for the general list case, yields nontrivial improvements over the naïve linear-time equivalence test that effects no optimization. In contrast, we provide nontrivial lower bounds showing that neither of the first two strategies achieves constant amortized time per query.

Finally, we mention previous research into the similar problem of testing equivalence in sets [9, 11, 12, 14, 15] and also subset and set intersection testing [16].

2 Definitions and Problem Statement

The general case of the lazy structure sharing problem is as follows. We are given a collection of *nodes*, each containing zero or more *pointers* to other nodes. The nodes to which the pointers of x point are called *children of* x; further, the children are ordered. We freely interchange the notion of a pointer and the child to which it points; the meaning will be clear from the context. We assume the induced graph is acyclic.

Two nodes x and y are *equivalent* if they each have the same number $j \geq 0$ of children and, for each $1 \leq i \leq j$, the ith child of x is equivalent to the ith child of y. Given acyclicity, equivalence can be tested by a depth-first search of the induced graph. We can dispense with

Figure 1: (a) x and y and their children before a query. (b) After finding x and y, and their respective pairs of children, equivalent.

acyclicity if we can mark nodes during the search and allow pointers to point to previously marked nodes, assuming they were marked "in an equivalent fashion." We can add keys to the nodes and add the requirement that equivalent nodes contain equal keys.

A *query* tests two nodes for equivalence. We allow no insertions or deletions of nodes or pointers. Our data structure is thus static except for the following type of modification: if a query determines that two nodes x and y are equivalent, we may change any pointer to x (y) to point to y (x) instead. In particular, if we do this for each child of x (y), then an immediately succeeding query on x and y will terminate after examining only the immediate children of the nodes, for each node's ith pointer will point to the same child. See Figure 1. (In all figures in this paper, all the pointers point downwards.) An *ideal configuration* of our data structure is one in which equivalent nodes point to physically equal children; i.e., the ith pointer of x and the ith pointer of y point to the same node if x and y are equivalent. Once an ideal configuration is attained, all successive queries are of constant cost if the nodes are of bounded outdegree.

In the Sections 3–5, we assume all the nodes are of outdegree zero or one. In this case we have a set of lists, and two nodes are equivalent if they point to sublists of equal length. We denote by n the number of lists and k the size of the lists (the number of pointers in each list—one less than the number of nodes in each list); we assume all lists are the same size. Thus, $p = nk$ is the total number of pointers. In Section 6 we make no assumptions concerning the outdegrees of the nodes.

We expect the data structure to reach an ideal configuration over a number of successful queries; we are interested in measuring how long this takes under a variety of strategies. Let q be the total number of successful queries (those that return equivalence; we do not count unsuccessful queries, since they effect no optimization), and let w be the total amount of *work* done. We charge one unit of work for each pointer traversed plus one unit for comparing two distinct nodes of outdegree zero; this subsumes the number of pointer changes performed. We wish to bound the ratio $R = w/(q + p)$, the work amortized over the number of queries and pointers. We must allow each pointer to be moved at least once, thus accounting for the p term in the denominator. Where possible, we express R in terms of q and p. An *optimal strategy* achieves $R = O(1)$.

We examine various strategies for choosing which pointer to move when given the choice between two. Our results show that the *reference count* method (changing the pointer pointing to the node of lower indegree to point to the one of higher indegree) performs better than the *total order* method (a priori determining for each pair of nodes which one

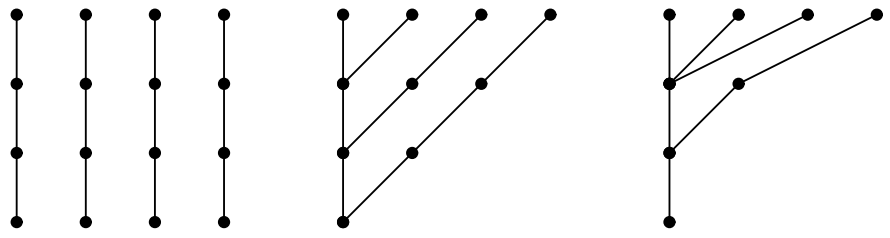

Figure 2: (a) An initial collection of four lists of length three. (b) After phase 0. (c) After phase 1.

will receive the other's pointer). Except for one special case, however, even the reference count method is not optimal. The *ancestor count* method (like reference count, but using ancestor count instead of indegree) in turn outperforms the reference count method. We also discuss a different strategy involving "phantom pointers" and disjoint set union that performs better than the other three strategies.

3 Total Order

The *total order strategy* assigns a priori a total order $\lessdot$ to all the nodes. If x and y are equivalent and $x(i)$ ($y(i)$) is the (pointer to the) ith child of x (y), then if $x(i) \lessdot y(i)$ we update $y(i)$ to point to $x(i)$; otherwise we update $x(i)$ to point to $y(i)$. For clarity, all relevant figures in this paper assume the total order is a "move left" order; i.e., we always move the right pointer to point to the left node.

3.1 Lower Bound for Top Querying

Here we examine the special case in which we have n lists of length $n-1$ and we perform queries only on top nodes, continuing until the data structure reaches an ideal configuration. Embedding the lists in the plane with the pointers pointing down, we only query the heads of the lists. Number the lists left-to-right from 0 through $n-1$, and denote by *node* i the head of list i. We perform $n-1$ phases of queries; each phase consists of querying, in order, nodes $n-2$ and $n-1$, then $n-3$ and $n-2$, and so on through nodes 0 and 1. To simplify later equations, we call the first phase the 0th phase and remark trivially that each 0th-phase query costs n, for a total 0th-phase cost of $n(n-1)$. See Figure 2.

Lemma 3.1 *Querying x and $x+1$ in phase $i > 0$ costs $\left\lceil \frac{x+1}{i} \right\rceil$.*

PROOF: After phase 0, all paths in the data structure terminate at a list 0 node. Therefore, the cost of a query in a phase $i > 0$ is the length of the longer path from a list head to a node in list 0. The above sequence of queries causes each pointer change to make the affected pointer point to the next list to the left; i.e., if it pointed to a list j node, it changes to point

to a list $j-1$ node. Thus the path from node x to a node in list 0 is of length $\left\lceil \frac{x}{i} \right\rceil$ at the beginning of phase i. The path is not affected until x is queried with $x-1$. □

We modify each phase, without affecting the validity of Lemma 3.1, by removing queries of unit cost (those that effect no pointer changes). Thus, phase 1 begins by querying $n-2$ and $n-1$ and ends by querying 1 and 2; in general, phase i begins by querying $n-2$ and $n-1$ and ends by querying i and $i+1$. Call a sequence of queries of only top nodes that effects an ideal configuration of the data structure a *top querying problem.*

Theorem 3.2 *$R = \Omega(\log p)$ for the top querying problem using the total order strategy on n lists of length $n-1$.*

PROOF: Using Lemma 3.1 we have

$$\begin{aligned} w &= n(n-1) + \sum_{i=1}^{n-2} \sum_{x=i+1}^{n-1} \left\lceil \frac{x}{i} \right\rceil \\ &\geq n(n-1) + \sum_{i=1}^{n-2} \frac{1}{i} \left(\frac{n(n-1) - i(i+1)}{2} \right) \\ &= n(n-1) + \frac{n(n-1)}{2} H_{n-2} - \frac{1}{2} \left(\frac{n(n-1)}{2} - 1 \right) \end{aligned}$$

Given that $q = \sum_{i=1}^{n-1} i$ and $p = n(n-1)$, we derive $R = w/(q+p) = \Omega(\log n) = \Omega(\log p)$. □

3.2 Upper Bound for the General Case

For the general case, we have only been able to derive the trivial upper bound.

Theorem 3.3 *$R = O(n)$ for a sequence of queries using the total order strategy. In particular, if $k = n^e$ for some constant e, then $R = O(p^{\frac{1}{1+e}})$.*

PROOF: The set of pointers pointing to equivalent nodes forms an equivalence class of size n. Consider the pointers (and their target nodes) in any class. Since each pointer only moves to successively lower nodes in the total order, the total number of pointer changes that occurs in the class is $O(n^2)$. There are k distinct classes, so the total number of pointer changes possible under any sequence of queries is $O(kn^2)$. The bounds follow since $p = nk$. □

Note that if $k \gg n$, the upper bound on R becomes very small with respect to p. This is because after only $n-1$ queries of cost k, no further query is of cost greater than $n-1$. For example, the 0th phase from the top querying section achieves this state.

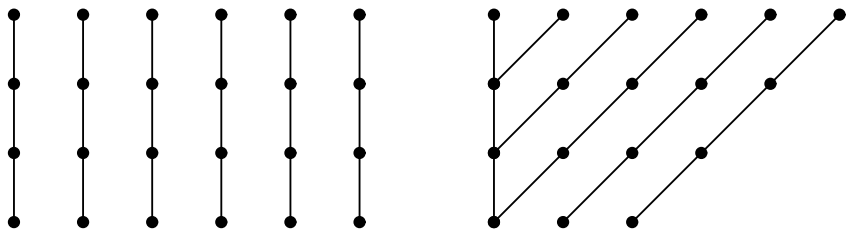

Figure 3: (a) An initial collection of six lists of length three. (b) After phase 0.

3.3 Lower Bound for the General Case

For the general case, we derive a nontrivial lower bound as follows. Consider n initial lists of length k such that $n = ck$ for some positive integer c. As before, number the lists left-to-right from 0 through $n-1$. Proceed once more in phases, querying heads of lists. In the 0th phase, query the heads of lists $n-2$ and $n-1$, then $n-3$ and $n-2$, and so on through 0 and 1. We thus reduce the data structure to $n-k$ lists of length k, plus k lists of length less than k, which we ignore. See Figure 3. Repeat the sequence in phase 1, using only the length-k lists, and so on. Thus, each phase $0 \leq i < n/k$ performs $n-1-ik$ queries of cost $k+1$.

Theorem 3.4 *$R = \Omega(p^{1/3})$ for a sequence of queries using the total order strategy.*

PROOF: We perform $\sum_{i=0}^{n/k-1}(n-1-ik)$ queries of cost $k+1$. Evaluating this sum yields $q = \frac{n(n+k-2)}{2k}$. We also have $w = q(k+1)$ and $p = nk$. The result maximizes $R = w/(q+p)$ with respect to k at $k \approx \sqrt{n}$. At this value, $R \approx \sqrt{n}$. Since $p = nk$, the result follows. □

Note that the lower bound is an explicit power of p, whereas in the upper bound the power of p depends on the value of k. This is not contradictory, however, since for the lower bound argument we assumed $k \leq n$. If we make this assumption, then the upper bound becomes no better than $O(p^{1/2})$. The assumption is valid, furthermore, because of the previous observation that if $k \gg n$, the data structure can quickly devolve into a state in which all queries cost no more than $n-1$.

4 Reference Counts

The total order strategy suffers from the drawback that once a node acquires a lot of incoming pointers, it can still lose them during later queries. The *reference count strategy* alleviates this problem as follows: given pointers to two equivalent nodes x and y, if the indegree of x is greater than that of y, the pointer to y is changed to point to x, and vice-versa, breaking ties arbitrarily. Note that under this strategy, the data structure would reach an ideal configuration after one pass of phase 0 from above.

We prove that while the reference count strategy has better upper and lower bounds on R than the total order strategy, it still does not achieve $R = O(1)$ except in the top querying problem.

4.1 Tight Bound for Top Querying

For the top querying problem, we show that the reference count strategy is optimal. Define the *level* of a node to be its distance from the head of the list, and define the level of a pointer to be the level of its source node. A *top node* is a node of level 0.

Lemma 4.1 *If there are n pointers in a given level, the total number of pointer changes to all the pointers in that level is $O(n^2)$.*

PROOF: Let $x_1, \ldots, x_n$ be the indegrees of the nodes in the next level. Define a potential $\phi = \sum_{i=1}^{n} x_i^2$. Clearly $n \leq \phi \leq n^2$. Consider a query passing through pointers on this level to nodes i and j, and assume without loss of generality that $x_i \leq x_j$. By the reference count strategy, we change the pointer to i to point to j, yielding $\Delta\phi = (x_j+1)^2+(x_i-1)^2-x_j^2-x_i^2 \geq 2$. Thus, the total number of pointer updates is no more than $\frac{1}{2}(n^2 - n)$. □

Now we show that we only have to consider about half the level i pointers at level $i + 1$. Call two top nodes a *short pair* if querying them costs less than n—i.e., if a query on them terminates before reaching the tail of a list. Note that if x and y form a short pair and y and z form a short pair, then x and z also form a short pair. Therefore the following is well-defined: a *short set* is a maximal set of top nodes such that any two nodes in the set form a short pair. For any short set, define a *short structure* to be the short set together with all the nodes (and pointers) that are reachable from nodes in the set. Note that the level of a node does not change due to query optimization, since a level i node has incoming pointers from only level $i-1$ nodes. Define the *level* of a short structure to be the maximum level of a node of indegree greater than one in the structure; there is only one such node unless the structure is a list, in which case define its level to be 0.

Lemma 4.2 *A short structure of level l has at least 2^l top nodes.*

PROOF: By induction. The base case is a list. Querying two nodes in the same short set cannot increase the level of the associated short structure. Consider a query of two top nodes in different short sets, of levels i and j. The new short set is the union of the two old ones. If $i < j$ $(i > j)$ the new short structure is of level j (i), and the invariant is maintained. Otherwise, $i = j$, and the new short structure is of level $i + 1$; by the induction hypothesis, there were at least 2^i nodes in each old short structure. □

Lemma 4.3 *In a short structure with n top nodes, the number of level i nodes is no more than* $\max\{n/2^i, 1\}$.

PROOF: By induction. Initially, we have a group of lists, each of which is a short structure, and all levels within any short structure contain 1 node. Querying two top nodes in the same short set cannot increase the number of nodes in any level in the underlying short structure. Consider a query of two nodes in different short sets: one with n top nodes and of level i, the other with m top nodes and of level j. The resulting short structure has $n + m$ top nodes. Assume without loss of generality that $i \geq j$, and consider level l in the resulting structure. There are three cases:

1. $l < j$. There are no more than $n/2^l + m/2^l$ nodes in level l.
2. $l = j$. There are no more than $\max\left\{n/2^l, 1\right\} + 1$ nodes in level l, but by Lemma 4.2, $m \geq 2^j$, so this allows the additional node.
3. $l > j$. There are no more than $\max\left\{n/2^l, 1\right\}$ nodes in level l.

□

Theorem 4.4 *$R = O(1)$ for the top querying problem using the reference count strategy on n lists of length $n - 1$.*

PROOF: We can perform no more than $n-1$ queries of cost n. The remaining queries are on short pairs. Lemma 4.3 bounds the number of pointers in level i of a short structure with x top nodes by $\max\{x/2^i, 1\}$. Lemma 4.1 says that in a level with y pointers, we can perform only $O(y^2)$ pointer changes. Since we can only perform pointer changes up through pointer level $l - 1$ during a query on two top nodes in a short structure of level l, the total number of pointer changes due to intra-short-structure queries in a short structure of level l with x top nodes is no more than $\sum_{i=0}^{l-1} \left(\frac{x}{2^i}\right)^2 = O(x^2)$. When we combine two short structures via a long (cost n) query, the potentials for their respective levels combine in the straightforward manner.

We thus have $w = q + O(n^2)$ and $p = n(n-1)$, giving $R = w/(q+p) = O(1)$. □

4.2 Upper Bound for the General Case

The analysis for the top querying problem ignores unreachable nodes. In the general case, though, any query is allowed, so there are no such nodes. We can refine the potential function, however, to achieve a nontrivial upper bound for the general case. Note that the proof of Lemma 4.1 relies only on the fact that the potential function increases monotonically. In fact, it can increase by big ($\Omega(n)$) amounts; this happens if $x_j - x_i = \Omega(n)$. The analysis ignores these large leaps, however.

We address this problem by breaking the potential function into two pieces: a small piece and a large piece. Intuitively, we consider the small nodes and large nodes on a given level. The small nodes have indegrees not exceeding $\sqrt{n}$, and the large nodes are the rest. The original potential function suffices to bound the pointer changes affecting small nodes. For big nodes, however, note that those incoming pointers in excess of the $\sqrt{n}$ threshold move

only to nodes of higher and higher indegree (until the big node becomes a small node). But there are only about $\sqrt{n}$ big nodes, so this "rightward progression" must be short. This notion of considering small and big parts is also utilized by Yellin [16] to analyze a data structure that performs subset and set intersection queries.

Now consider the following game on piles of stones. Initially, there n piles of 1 stone each. At any turn, pick two piles and move a stone from the smaller to the larger pile (breaking ties arbitrarily). We bound the number of possible turns—the game ends when there is one pile of n stones. Observing that we can substitute any pile of the same degree for a pile actually picked in a turn without affecting the result, we show the following.

Lemma 4.5 *The number of turns possible in the above game is $O(n\sqrt{n})$.*

PROOF: Let there be k piles remaining at any point in the game, and let their sizes (numbers of stones) be $x_1 \leq \cdots \leq x_k$. Define the following potential function:

$$\begin{aligned} \phi(x_i) &= \min\left\{x_i^2, n\right\} + \max\left\{x_i - \sqrt{n}, 0\right\} \cdot (c\sqrt{n} - k + i) \\ \Phi &= \sum_{i=1}^{k} \phi(x_i) \end{aligned}$$

c is some constant to be determined later. As before, we show that Φ monotonically increases, but (assuming that for the moment) now note that $n \leq \Phi \leq n(c\sqrt{n} - c + 1) = O(n\sqrt{n})$. The min term is the "small potential," which bounds the pointer changes for the small piles, whereas the max term bounds the "rightward progression" of "big pointers" (those excess pointers in the big piles).

Consider a turn that moves a stone from pile i to pile j. From the above discussion, we know $x_i \leq x_j$ and, in fact, $i < j$ (since the x_is are ordered). Further, by the previous observation, we assume without loss of generality that (1) $i = 1$ or $x_{i-1} < x_i$, and (2) $j = k$ or $x_j < x_{j+1}$. Assuming for the moment that $x_i > 1$, there are five cases to consider (the x_ls below are before the operation):

1. $x_j < \sqrt{n}$. $\Delta\Phi = (x_j + 1)^2 + (x_i - 1)^2 - x_j^2 - x_i^2 \geq 2$.

2. $x_j = \sqrt{n}$. $\Delta\Phi = n + c\sqrt{n} - k + j + (x_i - 1)^2 - x_j^2 - x_i^2 = c\sqrt{n} - k + j - 2x_i + 1$. But since there are only $\sqrt{n}$ big piles, we have $-\sqrt{n} + 1 \leq j - k \leq 0$; also, we know $x_i \leq \sqrt{n}$. Therefore, $\Delta\Phi \geq 2$ as long as $c \geq 3$.

3. $x_i = \sqrt{n} + 1$. $\Delta\Phi = n + (x_j + 1 - \sqrt{n})(c\sqrt{n} - k + j) + n - n - (x_j - \sqrt{n})(c\sqrt{n} - k + j) - n - c\sqrt{n} + k - i = j - i \geq 1$.

4. $x_i > \sqrt{n} + 1$. $\Delta\Phi = n + (x_j + 1 - \sqrt{n})(c\sqrt{n} - k + j) + n + (x_i - 1 - \sqrt{n})(c\sqrt{n} - k + i) - n - (x_j - \sqrt{n})(c\sqrt{n} - k + j) - n - (x_i - \sqrt{n})(c\sqrt{n} - k + i) = j - i \geq 1$.

5. $x_i \leq \sqrt{n}$, $x_j > \sqrt{n}$. $\Delta\Phi = n + (x_j + 1 - \sqrt{n})(c\sqrt{n} - k + j) + (x_i - 1)^2 - n - (x_j - \sqrt{n})(c\sqrt{n} - k + j) - x_i^2 = c\sqrt{n} - k + j - 2x_i + 1$. Again, if $c \geq 3$, then $\Delta\Phi \geq 2$.

The only other matter affecting Φ is the disappearance of a pile (if $x_i = 1$). In this case, we must consider the effect on $l - k$ in the max term of $\phi(x_l)$ for $2 \leq l \leq k$. We note that there is no change, for all the piles "shift left" by one. That is, the lth pile becomes the $(l-1)$th pile, and k becomes $k-1$, so $l-k$ remains unchanged. Choosing $c = 3$ completes the proof. □

We now prove the following for n lists of length k.

Theorem 4.6 *$R = O(\sqrt{n})$ for a sequence of queries using the reference count strategy. In particular, if $k = n^e$ for some constant e, then $R = O(p^{\frac{1}{2(1+e)}})$.*

PROOF: Lemma 4.5 says that we can perform $O(n\sqrt{n})$ pointer changes per level. There are k levels, so we get $w = q + O(kn\sqrt{n})$. Since $p = nk$, we have $R = w/(q+p) = O(\sqrt{n})$. □

In the next section, we prove a lower bound that, among other things, says that we can in fact perform $\Omega(n\sqrt{n})$ turns in our stone game. Thus, the above is the best upper bound derivable by bounding the number of pointer changes on a per-level basis. Finding a better upper bound will require analyzing the effects of a query on a number of levels simultaneously.

4.3 Lower Bound for the General Case

In this section, we construct a sequence of p queries on a family of lists consisting of p nodes that causes the reference count algorithm to incur $\Omega(p^{6/5})$ cost.

We need a few definitions. A *degree-d length-k fan-list* is the graph constructed by successively querying the heads of $(d-1)$ disjoint lists of length k with the head of a single starting list of length k; in other words, a fan-list is simply a list of internal nodes of indegree d together with a set of leaf nodes (of indegree 0) that point to these internal nodes. A *(k,n)-fan-grid* is the graph that is the union of n disjoint length-k fan-lists of respective degrees $3, 5, \ldots, 2n+1$. See Figure 4a.

We construct the following sequence of $O(n^3/k + kn^2)$ queries performed on $O(n^2)$ disjoint length-k lists, containing $\Omega(n^2 + n^3/k)$ cost-k queries, for the reference count algorithm:

1. Construct a (k,n)-fan-grid from $n(n+2)$ length-k lists. Let $v_{i,j}$, for $1 \leq i \leq k$ and $1 \leq j \leq n$, denote the internal nodes of the fan-grid. The indegree of $v_{i,j}$ equals $2j+1$; an *entry* to $v_{i,j}$ is a leaf node pointing to $v_{i,j}$; $v_{i,j}$ points to $v_{i+1,j}$ for all $1 \leq i < k$, $1 \leq j \leq n$.

2. Repeat the following n times:

   (a) Repeatedly perturb the fan-grid as follows $\lfloor n/(2k) \rfloor$ times (assume $\lfloor n/(2k) \rfloor \geq 1$): Let $\vec{u} = (u_0, u_1, \ldots, u_k)$ be a new list, and let j be the maximum integer such that the chain of pointers that starts in $v_{1,j}$ ends in $v_{k,n}$; initially $j = n$. Query u_0 with an entry to $v_{1,i}$, in turn, for $1 \leq i \leq j$. Each such "percolation" of a $\vec{u}$ causes the chain subtended by a node $v_{1,l}$ to end in a fan of degree k higher than before if it still ends at a node $v_{k,l'}$, and thus j decreases by k. See Figure 4b.

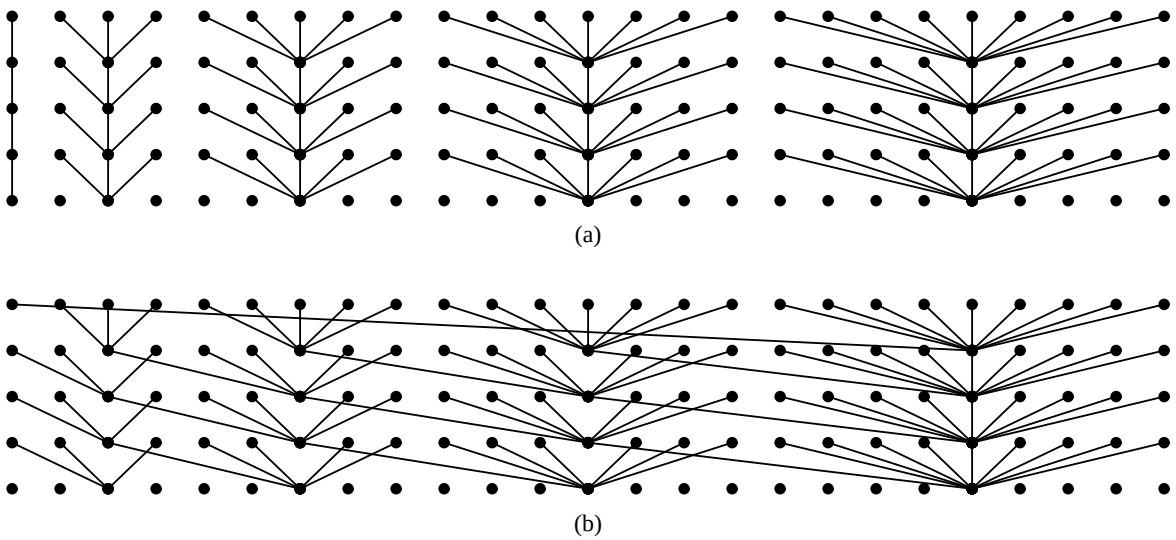

Figure 4: (a) A (4,4)-fan-grid and a single list of length 4 to its left. (b) After one percolation of the list through the grid; note that k is too large for this to be a real case of our construction, but the figure nonetheless illustrates the process.

(b) Reset the perturbed fan-grid to its initial state as follows:

i. Create a degree-$(2n+3)$ length-k fan-list, called the *buffer fan-list,* and a degree-$(3n + 4 + \lfloor n/(2k) \rfloor)$ length-k fan-list, called the *garbage fan-list.* The internal nodes of these fan-lists form respective chains $(w_1, \ldots, w_k)$ and $(x_1, \ldots, x_k)$.

ii. For j from 1 to n and i from $k-1$ to 1 (left-to-right, bottom-up), query $v_{i,j}$ and w_j; this makes $v_{i,j}$ point to w_{j+1}. The w_js thus increase in indegree by n each.

iii. For any node $v_{1,j}$ such that the indegree of $v_{1,j}$ is greater than $2j+1$, reduce the indegree of $v_{1,j}$ to $2j+1$ by querying sufficiently many of its entries, in turn, with an entry to w_1. Reduce the indegree of all the nodes $v_{i,j}$ such that $2 \leq i \leq k$ to $2j$ analogously. The w_js increase in indegree by no more than $\lfloor n/(2k) \rfloor$ each. This follows from the fact that only $\lfloor n/(2k) \rfloor$ extra entries accrue per level, since an extra entry is obtained by a fan only via a list percolation from Step 2a, and each percolation creates only one extra entry per level.

iv. Reduce the indegree of w_j, for $2 \leq j \leq k$, to n by querying all of its entries, in turn, with an entry to x_j; i.e., w_j is left with incoming pointers only from the $v_{j-1,l}$ nodes. That x_j is of sufficient initial indegree follows from the above discussions of the increase in indegree of w_j.

v. For j from n down to 1 and for i from 1 to $k-1$, query $v_{i,j}$ with an entry to $v_{i+1,j}$. Since $v_{i,j}$ points to w_{j+1}, this resets $v_{i,j}$ to point to $v_{i+1,j}$. The indegrees of the w_js decrease fast enough as this step progresses to the left through the fan-grid. After this step, the fan-grid has been returned to its initial (unperturbed) state.

Theorem 4.7 *$R = \Omega(p^{1/5})$ for a sequence of queries using the reference count strategy.*

PROOF: Let us calculate the number of queries performed during the above query sequence. Step 1 performs $n(n+1)$ queries of cost k each (actually of cost $k+1$, but we can consider the costs to be k without affecting the asymptotic results). From the discussion in Step 2a we see that $j \geq \lfloor n/2 \rfloor$, so each application of Step 2a consists of $\Theta(n^2/k)$ queries of cost k.

Now consider the queries incurred by Step 2b. Part (i) performs $O(n)$ queries. Note that while the buffer list must be reconstructed each time, the garbage list need only be constructed once. Part (ii) performs $O(nk)$ queries. In Part (iii), the indegree of a node must be reduced only if it has acquired new entries. This happens only during the percolation of one of the $\vec{u}$ lists, so there are only $\lfloor n/(2k) \rfloor$ extra entries per level. Therefore, Part (iii) requires only $O(n)$ queries. Parts (iv) and (v) perform $O(nk)$ queries. Thus, each application of Step 2b performs $O(nk)$ queries.

Since Step 2 is repeated n times, the above query sequence contains $q = O(n^3/k + kn^2)$ queries, of which $\Omega(n^2 + n^3/k)$ are of cost k. Only $O(n^2)$ disjoint lists are used during the entire sequence, so $p = O(kn^2)$. $R = w/(q+p)$ is maximized with respect to k at $k \approx \sqrt{n}$. □

As in Section 3.3, note that the lower bound is an explicit power of p, whereas the upper bound depends upon k. Again, however, the lower bound construction assumed that $k \leq n$, and under this assumption the upper bound is no better than $R = O(p^{1/4})$. As before, the quick degeneration of the problem from the case where $k > n$ to a set of problems with $k \leq n$ justifies this assumption.

Finally, note that a variant of the above construction shows that Lemma 4.5 is tight. Start with $O(n)$ piles of 1 stone. First create piles of $2, 4, \ldots, 2\sqrt{n}$ stones, leaving $O(n)$ piles of 1 stone. Percolate each unit pile through the sequence of increasing piles, resulting in $\Omega(n\sqrt{n})$ moves. Thus, Theorem 4.6 is the best we can prove by bounding the number of pointer changes per level.

5 Ancestor Counts

The *ancestor count strategy* refines the total order strategy by determining which pointer to move based upon the total number of *ancestors* of the target nodes rather than their indegrees. We define *ancestor* in the usual way: y is an *ancestor* of x if there is a path from y to x; x is an ancestor of itself. We denote by $ac(x)$ the number of ancestors of x. We show that the ancestor count strategy outperforms the reference count strategy.

5.1 Tight Bound for Top Querying

Theorem 5.1 *$R = O(1)$ for the top querying problem using the ancestor count strategy on n lists of length $n-1$.*

PROOF: Apply the same analysis used in Section 4.1. In Lemma 4.1, let $x_1, \ldots, x_n$ be the ancestor counts of the nodes in some level i. Let $\phi = \sum_{j=1}^{n} \left(\frac{x_j}{i}\right)^2$. □

5.2 Upper Bound for the General Case

Here we use a different type of argument to prove our bound. We define a potential function on a node x to be $\phi(x) = \log ac(x)$, but we only use the potential function for part of the analysis. For n lists of length k ($p = nk$), note that initially $\Phi = \sum_x \phi(x) = n \sum_{i=1}^{k+1} \log i = O(p \log p)$, and we always have $\Phi > 0$.

First we show that Φ monotonically decreases during queries. When querying two lists $(x_1, \ldots, x_l)$ and $(y_1, \ldots, y_l)$ for some $l \leq k$, we consider the pointer changes bottom-up; i.e., first we swap one of the pointers to x_l and y_l, then to x_{l-1} and y_{l-1}, etc. In this way, a pointer swap affects the ancestor counts only of its old and new target nodes. Consider changing the pointer pointing to x_i to point to y_i instead. For clarity, we use x in place of $ac(x)$ in formulas. We have

$$\begin{aligned}
\Delta\Phi &= \log(x_i - x_{i-1}) - \log x_i + \log(y_i + x_{i-1}) - \log y_i \\
&= \log \frac{(x_i - x_{i-1})(y_i + x_{i-1})}{x_i y_i} \\
&= \log \left(\left(\frac{1}{y_i} - \frac{x_{i-1}}{x_i y_i} \right) \left(\frac{1}{x_i} + \frac{x_{i-1}}{x_i y_i} \right) \right)
\end{aligned}$$

Since we know that (1) $y_i \geq x_i$, (2) $x_i \geq x_{i-1} + 1$, and (3) $y_i \geq y_{i-1} + 1$, we derive $\left(\frac{1}{y_i} - \frac{x_{i-1}}{x_i y_i} \right) < 1$ and $\left(\frac{1}{x_i} + \frac{x_{i-1}}{x_i y_i} \right) \leq 1$, yielding $\Delta\Phi < 0$. The following analysis shows that for some pointer changes, the potential drop is at least a bound constant, which can happen only $O(p \log p)$ times, and that there will not be too many other pointer changes.

Theorem 5.2 *$R = O(\log p)$ for a sequence of queries using the ancestor count strategy.*

PROOF: Consider a query on two lists $(x_1, \ldots, x_l)$ and $(y_1, \ldots, y_l)$ for some $l \leq k$. Consider any pointer swap in the query, say for level i, and assume without loss of generality that the pointer to x_i changes to point to y_i. There are two cases:

1. $x_i > 2x_{i-1}$ or $y_i > 2y_{i-1}$. This can happen $O(\log p)$ times per query, since each time the ancestor count of a node in one of the lists at least doubles.

2. $x_i \leq 2x_{i-1}$ and $y_i \leq 2y_{i-1}$. Recall that

$$\begin{aligned}
\Delta\Phi &= \log(x_i - x_{i-1}) - \log x_i + \log(y_i + x_{i-1}) - \log y_i \\
&= \log \frac{x_i - x_{i-1}}{x_i} + \log \frac{y_i + x_{i-1}}{y_i}
\end{aligned}$$

   Knowing that $x_i \leq 2x_{i-1}$ and $x_i \leq y_i$, we derive that $\Delta\Phi$ is maximized when $x_i = 2x_{i-1}$ at $\Delta\Phi \leq -2 + \log 3$. Since this is a bound constant less than zero, this case can occur a total of $O(p \log p)$ times throughout the lifetime of the query sequence.

Therefore, $w = O((q + p) \log p)$. □

Thus, the ancestor count method performs strictly better than the reference count method. Unfortunately, we have been unable to find a lower bound for the ancestor count method.

6 Phantom Pointers

We now consider a different type of strategy, the *phantom pointer strategy*, and study its performance for the general case of lazy structure sharing, in which we make no assumptions about the outdegrees of the nodes. For this section, redefine p to be the total number of nodes in our data structure.

The phantom pointer strategy maintains a disjoint set union data structure [13] to record equivalent nodes. Associate with each node x a phantom pointer $p(x)$ and a size $s(x)$. Initially, $p(x) = x$ and $s(x) = 1$ for all nodes x. Define the operation *find*(x) as follows:

1. Let $y_1 \leftarrow x$, $y_2 \leftarrow x$.
2. While $y_1 \neq p(y_1)$ set $y_1 \leftarrow p(y_1)$.
3. While $y_2 \neq p(y_2)$ set $t \leftarrow p(y_2)$, $p(y_2) \leftarrow y_1$, $y_2 \leftarrow t$.
4. Return y_1.

That is, find(x) returns the end x' of the chain induced by the phantom pointers starting at x, and as a side effect compresses this chain so that all the phantom pointers on it are set to x'. Maintaining the invariant that $p(x) = y$ implies that x and y are equivalent, the path compression is a valid modification as defined in Section 2. Define a *set* to be a maximal collection X of nodes such that finds on any two nodes in X return the same value.

Now, any time a query passes through a node x, substitute find(x) for x. If two nodes x and y (assume x and y are already the returns of two find operations) are in different sets and via a query determined to be equivalent, set $p(x) \leftarrow y$ and $s(y) \leftarrow s(x) + s(y)$ if $s(x) < s(y)$; otherwise set $p(y) \leftarrow x$ and $s(x) \leftarrow s(x) + s(y)$. The work w now includes the number of phantom pointers updated. We mention the result before discussing the practicality of this implementation.

Theorem 6.1 *$R = O(\alpha(p + q, p))$, where α is a functional inverse of the Ackermann function, for a sequence of queries using the phantom pointer strategy.*

PROOF: The queries form an instance of disjoint set union. A pair of phantom pointer traversals either causes a union or terminates the query; since there can be at most $p - 1$ unions on p nodes, there are therefore $O(p+q)$ finds. From Tarjan and van Leeuwen [13] the total work done is $O((p + q)\alpha(p + q, p))$, where α is a functional inverse of the Ackermann function. □

Note that the nonlinear lower bound from Tarjan and van Leeuwen [13] might not apply here, since we employ a special case of disjoint set union in which (1) we always perform finds on two elements at a time, and (2) if the two elements are in separate sets, we always union the sets. Finding a lower bound for this strategy remains open.

One might reasonably question the fairness of this strategy, since it necessitates an extra pointer and count per node. If some real system is being designed to use lazy structure

sharing as a means of query optimization, this requirement may be unrealistic. We can reduce the space requirement by realizing that only the *roots* of the set trees (those nodes x in our data structures such that $p(x) = x$) must keep track of the sizes of the sets; further, the roots need no phantom pointers. So assuming that we can store a count in the space required by a pointer, the phantom pointer strategy requires only an extra pointer and an additional bit per node. Continuing, note that the reference and ancestor count strategies themselves require an extra count per node. With no extra information allowed per node, even these strategies become unimplementable. In this case, one is left with either the total order strategy (using, perhaps, the memory locations of the nodes as the total order) or some randomized strategy (see Section 7).

7 Conclusion and Open Problems

We have contrasted various strategies for implementing lazy structure sharing as a query optimization technique. In all the cases we describe, there remain gaps between the upper and lower bounds for the general problem. Narrowing these gaps is an interesting open problem. In particular, is there a nontrivial upper bound for the total order strategy? For the reference count strategy, finding some way to analyze the effect of a query on multiple levels of the data structure is necessary to improve the upper bound (if it is improvable). For the ancestor count strategy, we have no lower bound. For the phantom pointer strategy, finding a lower bound (or improving the upper bound) for the special case of the disjoint set union problem we employ has long been open.

There are interesting alternative strategies to consider. One idea is a randomized strategy, in which, given the choice between two pointers to move, the algorithm picks one at random. This benefits from the fact that, like the total order strategy, it requires no extra space per node in the data structure. As the randomized strategy seems unlikely to encounter the pathological case that makes total order perform poorly, we conjecture that it would be an improvement over the total order strategy.

To apply lazy structure sharing to testing S-expression equivalence in LISP, we need to generalize the above analyses for the case in which each node can have as many as two outgoing pointers. Initial work in this direction is promising; for example, the top querying problem using reference counts on n complete m-ary trees (fixed m) has similarly good ratios R as for the list case. In particular, the total number of pointer changes is $O(n^2)$ for $1 \leq m \leq 3$, $O(n^2 \log n)$ for $m = 4$, and $O(mn^{\log m})$ for $m > 4$. We can also consider more general cases in which each node has fixed, bounded, or arbitrary outdegree. The phantom pointer strategy already applies to the general case.

Finally, the similarity between complex data type equivalence testing and set equivalence testing deserves further attention to see if the techniques used to address the two problems can be unified.

8 Acknowledgements

We thank Danny Sleator and Peter Winkler for many helpful discussions. In particular, Danny introduced us to this problem; Peter was instrumental in the development of the initial sum of squares potential.

References

[1] J. Allen. *Anatomy of LISP*. McGraw-Hill Computer Science Series. McGraw-Hill Book Co., New York, 1978.

[2] R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Machine Intelligence*, 7:101–16, 1972.

[3] J. Cordin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *Proc. 9th IFIP World Computer Cong.*, pages 909–14, 1983.

[4] D. Flickinger, C. Pollard, and T. Wasow. Structure-sharing in lexical representation. In *Proc. 23rd Ann. Mtg. of the Asso. for Computation Linguistics*, pages 262–7, 1985.

[5] L. Karttunen and M. Kay. Structure sharing with binary trees. In *Proc. 23rd Ann. Mtg. of the Asso. for Computation Linguistics*, pages 133–6a, 1985.

[6] A. Martelli and U. Montanari. Theorem proving with structure sharing and efficient unification. In *Proc. 5th Int'l. Joint Conf. on Artificial Intelligence*, page 543, 1977.

[7] F. C. N. Pereira. A structure-sharing representation for unification-based grammar functions. In *Proc. 23rd Ann. Mtg. of the Asso. for Computation Linguistics*, pages 137–44, 1985.

[8] W. W. Pugh. *Incremental Computation and the Incremental Evaluation of Functional Programs*. PhD thesis, Department of Computer Science, Cornell University, 1988.

[9] W. W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 315–28, 1989.

[10] J. M. Spitzer, K. N. Levitt, and L. Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–75, 1978.

[11] R. Sundar and R. E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 18–25, 1990.

[12] M. Sussa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(2):31–4, 1976.

[13] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[14] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–79, 1981.

[15] D. M. Yellin. Representing sets with constant time equality testing. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 64–73, 1990.

[16] D. M. Yellin. Algorithms for subset testing and finding maximal sets. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 386–92, 1992.