

Salvage-Embeddings of Complete Trees^{*}

Sandeep N. Bhatt
Yale University
New Haven, Conn.

Fan R. K. Chung
Bell Communications Research
Morristown, N.J.

F. Thomson Leighton
MIT
Cambridge, Mass.

Arnold L. Rosenberg
University of Massachusetts
Amherst, Mass.

Abstract. A *salvage-embedding* (*S-embedding*) maps a complete binary tree \mathcal{G} into a larger complete binary tree \mathcal{H} some fraction of whose leaves have been labeled GOOD. The S-embedding maps leaves of \mathcal{G} one-to-one to GOOD leaves of \mathcal{H} ; it may be many-to-one on internal nodes. We measure the quality of an S-embedding by its *harvest*, the fraction H of GOOD leaves of \mathcal{H} that are images of leaves of \mathcal{G} , and its *congestion*, the largest number of edges of \mathcal{G} that get “routed” across the same edge of \mathcal{H} . We consider three scenarios for an N -leaf tree \mathcal{H} . In the *worst-case* scenario: if the fraction G of \mathcal{H} 's leaves are labeled GOOD, then one can S-embed an HGN -leaf \mathcal{G} in \mathcal{H} with congestion $\log \log N +$ a constant depending only on G and H , no matter how the GOOD leaves are distributed; this congestion cannot be lowered by more than a small constant factor. In the *expected-case* scenario, where leaves of \mathcal{H} are labeled GOOD or not, independently, with fixed probability: with probability exceeding $1 - N^{-\Omega(1)}$, for any $0 < H \leq 1/8$, one can S-embed an HN -leaf \mathcal{G} in \mathcal{H} with congestion $O(\log \log \log N)$; we do not know if this congestion can be lowered. In the *salvaging* scenario: we present an algorithm that, in time $O(N(\log N)^{3C+2})$, S-embeds in a given a leaf-labeled \mathcal{H} the largest possible \mathcal{G} , subject to the prespecified bound C on congestion. This work is motivated by the problem of salvaging a fault-free subnetwork of a *leaf-tree machine* — a tree architecture whose leaves hold “full-power” PEs and whose nonleaf nodes hold “rudimentary” PEs that route messages and perform simple combining tasks.

^{*} A preliminary version of this paper was presented at *CONPAR/92*, Lyon, France, September 1-4, 1992.

Authors' Mailing Addresses:

S.N. Bhatt: Dept. of Computer Science, Yale University, New Haven, CT 06520

F.R.K. Chung: Bell Communications Research, 435 South St., Morristown, NJ 07960

F.T. Leighton: Dept. of Mathematics and Lab. for Computer Science, MIT, Cambridge, MA 02139

A.L. Rosenberg: Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003

1 Introduction

This paper studies *salvage-embeddings* (*S-embeddings*) of small complete binary trees into large ones, motivated by the problem of tolerating faults in a type of tree-structured parallel architecture that we call a *leaf-tree machine*. Our study is among the first that concentrates solely on the congestion of embeddings.

1.1 Basic Notions

A. Trees and Forests

The *height- n complete binary tree* \mathcal{T}_n is the graph whose $2^{n+1} - 1$ nodes comprise the set of all binary words of length at most n and whose edges connect each node x of length less than n with its single-letter *children* $x0$ and $x1$. For each $\ell \in \{0, 1, \dots, n\}$, the 2^ℓ words/nodes of length ℓ form *level* $n - \ell$ of \mathcal{T}_n ; the unique node at level n is the *root* of \mathcal{T}_n , and the 2^n nodes at level 0 are the *leaves* of \mathcal{T}_n . We say that node x is a (*proper*) *ancestor* of node y , or, equivalently, that node y is a (*proper*) *descendant* of node x , just when the string x is a (proper) prefix of the string y . For each node x of \mathcal{T}_n , the *subtree of \mathcal{T}_n rooted at x* is the induced subgraph of \mathcal{T}_n on the nodes $\{xy : 0 \leq |y| \leq n - |x|\}$, i.e., the set of descendants of x .

Finally, our study calls for a nonstandard notion of forest. For our purposes, a *forest* is a nonempty set of complete binary trees of distinct sizes.

Our focus on complete *binary* trees simplifies notation and calculations. The ideas in our study translate readily to complete trees of arbitrary arity.

B. Salvage-Embeddings

Let us be given a complete binary tree \mathcal{T}_n , the fraction $0 < G \leq 1$ of whose 2^n leaves have been labeled GOOD; call the fraction G the *yield* of the labeling. Let us further be given a complete binary tree \mathcal{T}_k , where $2^k \leq G2^n$. We wish to embed \mathcal{T}_k into \mathcal{T}_n , using only the GOOD leaves of the latter; for mnemonic emphasis, we henceforth denote by \mathcal{G}_k (for *guest*) the copy of \mathcal{T}_k , and by \mathcal{H}_n (for *host*) the (leaf-labeled) copy of \mathcal{T}_n .

A *salvage-embedding* (*S-embedding*, for short) of \mathcal{G}_k in \mathcal{H}_n , where $2^k \leq G2^n$, is given by:

- an assignment α of the nodes of \mathcal{G}_k to nodes of \mathcal{H}_n that
 - maps each leaf of \mathcal{G}_k to a unique GOOD leaf of \mathcal{H}_n (so is one-to-one on the leaves of \mathcal{G}_k)

- is *progressive*, in that it preserves all ancestor-descendant relations among nodes of \mathcal{G}_k .
- a routing function ρ that assigns to each edge (x, y) of \mathcal{G}_k the shortest path in \mathcal{H}_n that connects nodes $\alpha(x)$ and $\alpha(y)$.

By extension, an *S-embedding of a forest of trees in \mathcal{H}_n* is a set of S-embeddings of the trees in the forest, *whose leaf assignments are node disjoint*. This node disjointness guarantees that if any subset of the trees in the forest are grown and combined into a single tree, an S-embedding of that single tree in \mathcal{H}_n can use the leaf assignments of the S-embedding of the forest in \mathcal{H}_n .

C. The Costs of an S-Embedding

Let us be given an S-embedding $\langle \alpha, \rho \rangle$ of the m -tree forest $\{\mathcal{G}_{k_1}, \mathcal{G}_{k_2}, \dots, \mathcal{G}_{k_i}\}$ in \mathcal{H}_n , where each $k_j < k_{j+1}$. We are interested in two costs of the embedding:

The *harvest* of the embedding is the ratio of the number of leaves in the largest tree in the forest (namely, \mathcal{G}_{k_m}) to the number of GOOD leaves in \mathcal{H}_n ; symbolically,

$$\text{Harvest}(\langle \alpha, \rho \rangle) = \frac{2^{k_m - n}}{G}.$$

The *congestion* of the embedding is measured by focussing just on those ρ -routing paths that are used to S-embed \mathcal{G}_{k_m} in \mathcal{H}_n : it is the maximum number of such paths that cross any single edge of \mathcal{H}_n . Symbolically,

$$\text{Congestion}(\langle \alpha, \rho \rangle) = \max_{(x,y) \in \text{Edges}(\mathcal{H}_n)} |\{(u, v) \in \text{Edges}(\mathcal{G}_{k_m}) \mid \rho(u, v) \text{ contains edge } (x, y)\}|.$$

1.2 Accomplishments

Throughout we consider the tree \mathcal{H}_n with $N = 2^n$ leaves. We denote the yield of \mathcal{H}_n by G ; i.e., we assume that the fraction $0 < G \leq 1$ of \mathcal{H}_n 's leaves have been labeled GOOD.

A. The Problems of Interest

The Congestion-Harvest Tradeoff Problems. *Determine, as a function of n , G , and the desired harvest¹ $0 < H \leq 1/2$, the smallest congestion $C = C(n, G, H)$ for which there is a congestion- C S-embedding of $\mathcal{G}_{\lfloor \log HG2^n \rfloor}$ into \mathcal{H}_n :*

(a) *in the worst-case scenario, i.e., no matter how the GOOD leaves are distributed in*

¹We cannot aim at harvests exceeding $1/2$: the number of harvested leaves must be a power of 2, but the largest power of 2 not exceeding $G2^n$, namely, $2^{\lfloor \log HG2^n \rfloor}$, may be close to $G2^{n-1}$.

\mathcal{H}_n ;

(b) in the expected scenario, i.e., with probability $\geq 1 - 2^{-\Omega(n)}$, when leaves of \mathcal{H}_n are labeled GOOD or not, independently, with some fixed probability.

In both of the **Congestion-Harvest Tradeoff** Problems, we assume that the harvest fraction $H = \Theta(1)$; i.e., we aim for a harvest that is a fixed fraction of the number of GOOD leaves.

The Harvest-Maximization Problem. *Given an allowable congestion (i.e., an integer) $C < n$, find the largest k for which there is an S -embedding of \mathcal{G}_k in \mathcal{H}_n with congestion $\leq C$.*

In this last problem, we insist that $C < n$, since the problem of S -embedding trivializes when $C \geq n$.

B. The Results Obtained

The Worst-Case Congestion-Harvest Tradeoff. *For any yield $0 < G \leq 1$ and harvest $0 < H \leq 1/2$, one can S -embed $\mathcal{G}_{\lfloor \log HG2^n \rfloor}$ into \mathcal{H}_n with congestion² ³*

$$\log^{(2)} N + \kappa(F, G),$$

no matter how the GOOD leaves are distributed in \mathcal{H}_n (Theorem 2.1).

Moreover, in general, this amount of congestion is necessary, to within a constant factor; i.e., there exist yields G with associated patterns of GN GOOD leaves for which any S -embedding of $\mathcal{G}_{\lfloor \log HG2^n \rfloor}$ into \mathcal{H}_n has congestion $\kappa(F, G) \log^{(2)} N$ (Theorem 2.2).

The Expected-Case Congestion-Harvest Tradeoff. *For any harvest $0 < H \leq 1/8$: if leaves of \mathcal{H}_n are labeled GOOD or not, independently, with probability $1/2$ (an arbitrary fixed constant), then, with probability exceeding $1 - N^{-\Omega(1)}$, one can embed $\mathcal{G}_{\lfloor \log H2^n \rfloor}$ into \mathcal{H}_n with congestion*

$$\log^{(3)} N + \kappa(H).$$

(Theorem 3.1). It remains an inviting challenge to determine whether or not a smaller congestion suffices.

The Harvest-Maximization Algorithm. *For any congestion bound C and any labeling of the leaves of \mathcal{H}_n , one can determine in time $O(n^{3C+22^n})$ the largest \mathcal{G}_k that can*

²All logarithms are to the base 2. The iterated logarithm $\log^{(k)}$ is defined by:

$$\log^{(1)} N = \log N; \quad \log^{(k+1)} N = \log \log^{(k)} N.$$

³For given parameters A, B, \dots, Z , we denote by $\kappa(A, B, \dots, Z)$ a constant that depends only on the parameters; instances of “ $\kappa(A, B, \dots, Z)$ ” in different expressions may denote different constants. Thus, we use the κ -notation in very much the same way as the big- O notation.

be S -embedded into \mathcal{H}_n with congestion $\leq C$ (Theorem 4.1). Note that this time is a low-degree polynomial in $N = 2^n$ even when C is as large as $\log(N/\log N)$.

We study the worst-case congestion-harvest tradeoff problem in Section 2, the expected-case congestion-harvest tradeoff problem in Section 3, and the harvest-maximization problem in Section 4. It is worth stressing here that our study focusses on developing algorithms that effect S -embeddings with certain costs, rather than merely on proving that such S -embeddings exist.

1.3 The Inspiration for Our Machine Model

The idea of studying fault tolerance using S -embeddings was suggested by a genre of tree-machine that has appeared several times in the literature.

A *leaf-tree machine (LTM)* is a parallel architecture consisting of N “full-power” processing elements (PEs) and $N - 1$ “rudimentary” PEs. The $2N - 1$ PEs are interconnected by a network having the topology of a complete binary tree: the leaf-nodes of the tree hold the “full-power” PEs of the LTM that do the “real” computing; the nonleaf-nodes hold the “rudimentary” PEs that do simple auxiliary tasks such as routing and broadcasting messages and performing simple combining and accumulating tasks.

The leaf-nodes of \mathcal{H}_n represent the “full-power” PEs of the LTM; the nonleaf-nodes represent the LTM’s “rudimentary” PEs.

While interconnection networks with the topology of a tree are inherently inefficient due to the presence of communication bottlenecks, research has shown that an LTM can be a useful *auxiliary network* when adjoined to a processor network having a richer topology (say, a mesh or hypercube): A variety of computations were shown in [4] to yield to the simple, fast combining mechanism that is inherent in the structure of trees; these abilities of trees are exploited in [3] and [11], where LTMs are adjoined to data-processing machines for speedy searching, selection, and combining tasks; most recently, in [5, 6], LTMs have been adjoined to MIMD hypercubes with the end of using the trees’ fast combining and broadcasting capabilities for processor synchronization, as well as other simple combining and broadcasting tasks.

The problem we study here arises from the vulnerability of aggressive VLSI designs to fabrication defects which almost certainly disable some positive fraction of an architecture’s “full-power” PEs. (Wires and “rudimentary” PEs, being considerably smaller than “full-power” PEs, are commensurately less vulnerable to both defects and faults [12].) Our study of S -embeddings abstracts one approach to the problem of rendering

an LTM tolerant to defects in its “full-power,” leaf-PEs. Although this problem is nominally subsumed by studies of fault tolerance in tree architectures, such as [1], the special structure and mode of use of LTMs opens avenues to fault tolerance that are exponentially more efficient than analogous techniques for general tree machines: we achieve the desired tolerance to faults merely by adding small-capacity queues to the edges of the LTM.

- \mathcal{H}_n is the physical LTM; its GOOD leaves are the leaf-PEs that are free of defects. The yield of the labeling is the yield of the leaf-PE fabrication process.
- \mathcal{G}_k is the logical, ideal LTM we want to “salvage” from \mathcal{H}_n .
- The S-embedding is the salvage process; its congestion is the capacity of the largest edge-queue; its progressiveness ensures that, as in the ideal LTM, messages follow an up-then-down path in the salvaged LTM.
- The *dilation* [8] of an S-embedding is not relevant here because of the assumed speed of the ideal LTM relative to the speed of each individual PE; see, e.g., [5].

It is important to note that our abstraction deals *only* with the problem of tolerating defects in the LTM. we do *not* deal with the problem of tolerating defects in any primary network that interconnects the “full-power” PEs of the LTM, should such exist (say, as in the scenario in [5, 6]). There is an abundant literature on techniques for salvaging the latter network, e.g., [2, 7, 9, 13].

1.4 A Fundamental Observation

The algorithms we present here depend in an essential way on a correspondence between adding binary representations of numbers, on the one hand, and devising S-embeddings of complete binary trees, on the other.

For each node x of the leaf-labeled tree \mathcal{H}_n , define the *yield at x* , denoted $\text{Yield}(x)$, to be the number of GOOD leaves in the subtree rooted at x . Transparently, if x is a leaf, then $\text{Yield}(x)$ is either 0 or 1, and if x is not a leaf, then

$$\text{Yield}(x) = \text{Yield}(x0) + \text{Yield}(x1).$$

The binary representation of the numbers $\text{Yield}(x)$ play a fundamental role in our study. The statement of the following lemma is depicted schematically in Figure 1.

Lemma 1.1 *Let x be a node of \mathcal{H}_n . The binary representation of $\text{Yield}(x)$ has 1s in positions $\{k_1, k_2, \dots, k_d\}$ if, and only if, there is an S-embedding of the forest $\{\mathcal{G}_{k_1}, \mathcal{G}_{k_2}, \dots, \mathcal{G}_{k_d}\}$ in \mathcal{H}_n , with congestion d on the edge leading from node x to its parent.*

Proof. The lemma is verified by an easy induction.

The lemma is obvious when x is a leaf of \mathcal{H}_n , for then $\text{Yield}(x)$ is either 0 because x is not GOOD, or is 1 because x is GOOD, hence admits a unit-congestion S-embedding of \mathcal{G}_0 .

The situation when x is not a leaf is depicted schematically in Figure 2. If both $\text{Yield}(x0)$ and $\text{Yield}(x1)$ have 1s in bit-position k , then there is a carry into bit-position $k + 1$ when these Yields are added to obtain $\text{Yield}(x)$. Within the tree, there are, by induction, copies of \mathcal{G}_k S-embedded in the subtrees of \mathcal{H}_n rooted at nodes $x0$ and $x1$. By embedding a new \mathcal{G} -root-node at node x , the S-embeddings of the copies of \mathcal{G}_k can be combined into an S-embedding of a copy of \mathcal{G}_{k+1} rooted at x . Continuing, if one of $\text{Yield}(x0)$ and $\text{Yield}(x1)$ has a 1 in bit-position $k + 1$ also (by induction at most one of these numbers can have such a 1), then the addition leads also to a carry into bit-position $k + 2$. In the tree, we can S-embed yet another \mathcal{G} -root-node at node x , thereby combining the S-embeddings of the old copy of \mathcal{G}_{k+1} coming from either $x0$ or $x1$ with the new copy just S-embedded at x , to obtain an S-embedding of a copy of \mathcal{G}_{k+2} rooted at x . The reader can easily continue the narrative if the chain of carries in the addition is even longer. \square

2 Optimizing Worst-Case Congestion

Given the leaf-labeled tree \mathcal{H}_n , the yield $0 < G \leq 1$, and a target harvest $0 < H \leq 1/2$, we wish to find an S-embedding of the tree $\mathcal{G}_{\lfloor \log HG2^n \rfloor}$ in the tree \mathcal{H}_n , that incurs as small congestion as possible (as a function of n , G , and H). This section is devoted to deriving both upper and lower bounds on the amount of congestion C_{\min} that we must suffer in order to accomplish this task no matter how the GOOD leaves are distributed among the leaves of \mathcal{H}_n .

2.1 An Upper Bound on Worst-Case Behavior

We formulate and analyze a “greedy” S-embedding algorithm which yields an upper bound on the quantity C_{\min} that is optimal to within constant factors.

A. The Algorithm

Overview. The algorithm proceeds from level 0 to level n in \mathcal{H}_n (i.e., from the leaves toward the root), processing each node at level ℓ before proceeding to level $\ell + 1$. As each level- ℓ node x is encountered, the algorithm assigns the node a label $\lambda(x)$ which is

the length- $(n + 1)$ binary representation of the level- ℓ interim assessment of how many GOOD leaves can be harvested from the subtree rooted at x ; we call this quantity the *level- ℓ potential of x* , denoted $\text{Pot}(x)$. Thus, if

$$\lambda(x) = \lambda_n(x)\lambda_{n-1}(x) \cdots \lambda_0(x),$$

then

$$\text{Pot}(x) = \sum_{i=0}^n \lambda_i(x)2^i.$$

The following features of the string $\lambda(x)$ are germane to our algorithm.

- $\text{Ones}(\lambda(x)) = \{i \mid \lambda_i(x) \neq 0\}$ = the set of nonzero bit-positions in $\lambda(x)$;
- $\text{Wgt}(\lambda(x)) = |\text{Ones}(\lambda(x))|$ = the *weight*, or, number of nonzero bit-positions in $\lambda(x)$.

By Lemma 1.1, if node x of \mathcal{H}_n receives label $\lambda(x)$, then there is an S-embedding of the forest $\{\mathcal{G}_k : k \in \text{Ones}(\lambda(x))\}$ in the subtree of \mathcal{H}_n rooted at x , with congestion $\text{Wgt}(\lambda(x))$. Note that the progressiveness of S-embeddings implies that $\lambda_k(x) = 0$ for all $k > \text{level}(x)$. Because of Lemma 1.1, our embedding algorithm is fundamentally a labeling algorithm.

The Labeling/Embedding Procedure. Say that C is the maximum congestion we are willing to allow in any S-embedding. We exploit the fact that all labels have the same length (namely, $n + 1$) by specifying each label $\lambda(x)$ implicitly, via the integer $\text{Pot}(\lambda(x))$. In overview, our labeling/embedding algorithm proceeds from level 0 to level n in \mathcal{H}_n , labeling each node at level ℓ before any node at level $\ell + 1$. A node's label is chosen as follows.

1. If node v is a GOOD *leaf*, then $\text{Label}(v) \leftarrow 1$.
2. If node v is a *non-GOOD leaf*, then $\text{Label}(v) \leftarrow 0$.
3. If node v is a *nonleaf*, then
 - (a) Add the labels of v 's children.
 - (b) "Prune" the sum: keep only the C highest-order 1s.
 - (c) Perform the embeddings dictated by the carries in the label-additions at the nodes.

The detailed version of our labeling/embedding algorithm is called Algorithm **Worst-Case**.

Algorithm Worst-Case:

Step 0. {Label nodes on level 0 of \mathcal{H}_n }

Scan the leaves of \mathcal{H}_n , assigning each leaf x a label $\lambda(x)$ as follows.

$$\text{Pot}(\lambda(x)) = \begin{cases} 1 & \text{if } x \text{ is GOOD} \\ 0 & \text{if } x \text{ is not GOOD} \end{cases}$$

Step $\ell > 0$. {Label nodes on level $\ell > 0$ of \mathcal{H}_n }

Scan the nodes at level ℓ of \mathcal{H}_n .

Substep $\ell.a$ {Assign the string label}

Assign each level- ℓ node x a label $\lambda(x)$ as follows.

$$\text{Pot}(\lambda(x)) = \text{Pot}(\lambda(x0)) + \text{Pot}(\lambda(x1))$$

Substep $\ell.b$ {Combine small embedded trees}

if there was a chain of carries from bit-positions $k - i, k - i + 1, \dots, k - 1$ of $\lambda(x0)$ and $\lambda(x1)$ into bit-position k of $\lambda(x)$

then embed the roots of copies of $\mathcal{G}_{k-i+1}, \dots, \mathcal{G}_k$ in node x , **and** route edges from those roots, along shortest paths, to the roots of two copies each of $\mathcal{G}_{k-i}, \dots, \mathcal{G}_{k-1}$ that are embedded in proper descendants of x **endif**

Substep $\ell.c$ {Honor the congestion bound C }

for $k = 0$ **to** $\lceil \log \text{Pot}(\lambda(x)) - C \rceil$

if $\text{Wgt}(\lambda(x)) > C$ **then** $\lambda_k(x) \leftarrow 0$ **endif**

endfor

□

B. Worst-Case Behavior of Algorithm Worst-Case

Theorem 2.1 *Let some GN leaves of \mathcal{H}_n be labeled GOOD, in any way, for some $0 < G \leq 1$. For any rational $0 < H \leq 1/2$, Algorithm **Worst-Case** finds an S -embedding of $\mathcal{G}_{\lfloor \log HGN \rfloor}$ in \mathcal{T}_n , with congestion*

$$C \leq \log^{(2)} N - \log((1 - H)G) + 1. \tag{1}$$

Proof. It is clear that, for each node x of \mathcal{H}_n , Algorithm **Worst-Case** S-embeds $\mathcal{G}_{\lfloor \log \text{Pot}(\lambda(x)) \rfloor}$ in the subtree of \mathcal{H}_n rooted at node x ; hence, overall, the Algorithm S-embeds the tree $\mathcal{G}_{\lfloor \log \text{Pot}(\lambda(r)) \rfloor}$ in \mathcal{H}_n , where r is the root of \mathcal{H}_n . We need only verify that the salvaged tree is big enough when C is as big as the bound in inequality (1).

Note first that Algorithm **Worst-Case** never requires us to abandon any GOOD leaves as we work up from level 0 through level $C - 1$ of \mathcal{H}_n , because the high-order bit of $\text{Pot}(\lambda(x))$ can be no greater than the level of x in \mathcal{H}_n . To see what happens above this level, focus on a specific (but arbitrary) node x at a specific (but arbitrary) level $\ell \geq C$ of \mathcal{H}_n . The congestion bound may require us, in Substep $\ell.c$ of the Algorithm, to abandon one bit in each position $k \leq \ell - C$ of $\lambda(x)$. This is equivalent to abandoning one GOOD-leafed copy of each tree \mathcal{G}_k with $k \leq \ell - C$; however, at most one tree of each size is abandoned, because any two trees of the same size would have been coalesced (by embedding a new root) at this step, if not earlier. It follows that, when the Algorithm processes node x , it abandons no more than

$$\sum_{i=0}^{\ell-C} 2^i < 2^{\ell-C+1}$$

GOOD leaves; hence, at the entire level ℓ , strictly fewer than

$$2^{\ell-C+1} 2^{-\ell} N = 2^{-C+1} N$$

previously unabandoned GOOD leaves are abandoned. Thus, the entire salvage procedure abandons fewer than

$$(n + 1 - C) 2^{-C+1} N$$

GOOD leaves due to congestion. Since there are GN GOOD leaves in all, we see that more than

$$(G - (n + 1 - C) 2^{1-C}) N$$

GOOD leaves are *not* abandoned due to congestion. Now, at the end of the Algorithm, we may have to abandon almost half of these unabandoned GOOD leaves — because the GOOD leaves we finally use in the S-embedding must be a power of 2 in number. The Algorithm will have succeeded in salvaging the desired fraction of GOOD leaves as long as the number of salvaged GOOD leaves, which we now see to be no fewer than

$$2^{\lfloor \log(G - (n+1-C)2^{1-C})N \rfloor},$$

is at least as large as the number of GOOD leaves we want to salvage from \mathcal{H}_n , which is

$$2^{\lfloor \log HGN \rfloor}.$$

Simple estimates show that, if we allow our S-embedding to have congestion C as large as

$$C = \log^{(2)} N - \log((1 - H)G) + 1,$$

then we shall have accomplished this task. \square

2.2 A Lower Bound on Worst-Case Behavior

Theorem 2.2 *Let G and H be rationals with $0 < G < 1$ and $0 < H \leq 1/2$. For each n , there exists a way of labeling some GN leaves of \mathcal{H}_n GOOD, such that every S -embedding of some \mathcal{G}_m in \mathcal{H}_n , where $2^m \geq HGN$, has congestion $C > \kappa(G, H) \log^{(2)} N$.*

Proof. Let us be given an algorithm, call it Algorithm **A**, that solves the Worst-Case Congestion-Harvest Tradeoff Problem. By Section 2.1, we know that the congestion incurred by Algorithm **A** for *any* labeling of the leaves of \mathcal{H}_n , in particular for the advertised malicious labeling, is $C \leq \kappa(G, H) \log^{(2)} N$.

The Bad Labeling. The labeling of \mathcal{H}_n that we claim defies efficient salvage is achieved via the following algorithm, wherein L is a parameter we choose later. Once we choose L , let us restrict attention to values of n that are multiples of L . (Clerical adjustments accommodate all other values of n .)

Algorithm Bad-Label

Step 1. {Mark leaves that will *not* be labeled GOOD}

for each level ℓ **from** 0 **to** n **in steps of** L

Proceed left to right along the level- ℓ nodes of \mathcal{H}_n , marking all of the leaves in the subtree rooted at every 2^L -th node encountered

endfor

Step 2. {Label the GOOD leaves}

Label GOOD all leaves not marked in Step 1.

□

This labeling scheme can be viewed as turning \mathcal{H}_n into a complete $(2^L - 1)$ -ary tree, all of whose leaves are labeled GOOD, providing that we look only at levels whose level-numbers are divisible by L . Therefore, our S -embedding problem now assumes some of the flavor of the problem of efficiently embedding a complete binary tree into a complete $(2^L - 1)$ -ary tree that is only slightly larger (by roughly the factor $1/H$). The results in [8] about a similar problem lead us to expect the large congestion that we now show is inevitable.

Bounds on L and C . Before considering our S -embedding problem, we must settle on a value for the parameter L . Our labeling of \mathcal{H}_n has left the tree with $(2^L - 1)^{n/L}$ GOOD leaves. The Worst-Case Congestion-Harvest Tradeoff Problem assumes that the number of GOOD leaves in \mathcal{H}_n is a positive fraction $G > 0$ of the total number of leaves, namely, 2^n .

Elementary estimates show that this assumption implies that $L \geq \log n - \log^{(2)} n - \kappa(G)$. We shall assume, therefore, that $L = \Theta(\log n)$.

In analyzing our putative Algorithm **A**, we shall assume that we are dealing with S-embeddings whose congestions satisfy $C < \frac{1}{3}L$ (or else, we have nothing to prove).⁴

The Analysis. For each $\ell \in \{0, 1, \dots, n/L\}$, let $M(\ell)$ denote the number of leaves in the largest GOOD-leafed tree that Algorithm **A** S-embeds at a level- ℓL node of \mathcal{H}_n . Even if there were no bound on congestion, the similarity of the labeled version of \mathcal{H}_n and a complete $(2^L - 1)$ -ary tree would guarantee that, for $0 \leq \ell < n/L$,

$$M(\ell + 1) \leq (2^L - 1)M(\ell).$$

In order to appreciate the effect of the bound C on congestion, note that, when the S-embedding of Algorithm **A** has congestion $\leq C$, each number $M(\ell)$ must be representable as the sum of no more than C powers of 2; in other words, the binary representation of $M(\ell)$ can have weight no greater than C .⁵ It follows in particular that

$$M(1) \leq 2^L - 2^{L-C}.$$

Starting from this upper bound on $M(1)$, we derive a sequence of upper bounds on the other numbers $M(\ell)$. It is clerically convenient to number the bit-positions in the shortest binary representation of $M(\ell)$ from left to right, i.e., high-order to low-order. Moreover, we need focus only on bit-positions $1, 2, \dots, L$ of $M(\ell)$, as will become clear in the course of the argument.

Focus on a specific $\ell \in \{0, 1, \dots, n/L - 1\}$ and on its associated $M(\ell)$. Note the effect of proceeding from $M(\ell)$ to $M(\ell + 1)$, thence to $M(\ell + 2)$, and so on. Each step in this progression, say proceeding from $M(\ell + i)$ to $M(\ell + i + 1)$, consists of multiplying the “current” number, $M(\ell + i)$, by $2^L - 1$ (thereby going up L levels in \mathcal{H}_n), followed by “pruning” all but the highest-order C 1s in the product. Note that the multiplication part of this step affects only the rightmost 1 in bit-positions $1, 2, \dots, L$ of $M(\ell + i)$; say that this rightmost 1 appears in bit-position k . The effect of the multiply-then-“prune” step is as follows. Our assumption that the rightmost 1 of $M(\ell + i)$ appears in bit-position k means that the high-order L bit-positions of $M(\ell + i)$ form a string

$$\xi_1 \xi_2 \cdots \xi_{k-1} 1 0 0 \cdots 0 \tag{2}$$

⁴This assumption about C simplifies the estimates in the upcoming argument.

⁵This is true because we focus on *progressive* S-embeddings. If we allowed arbitrary S-embeddings, then $M(\ell)$ would be the *algebraic* sum — i.e., the sum/difference — of at most C powers of 2. The added generality of arbitrary S-embeddings would influence only constant factors in our bounds.

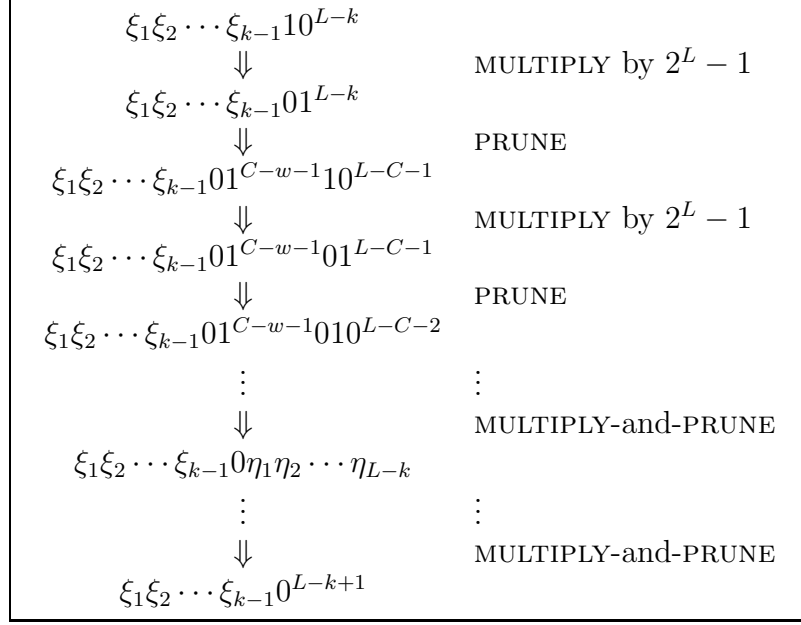


Figure 1: The “Migration” of the rightmost 1.

whose weight is

$$\text{Wgt}(\xi_1 \xi_2 \cdots \xi_{k-1} 1 0 0 \cdots 0) = w \leq C \quad (3)$$

and whose terminal string of 0s has length $\min(C - k + 1, L - k)$. The multiplication replaces this string with the like-length string

$$\xi_1 \xi_2 \cdots \xi_{k-1} 0 1 1 \cdots 1.$$

When this product string has weight exceeding C , the subsequent “pruning” replaces it by the like-length, weight- C string

$$\xi_1 \xi_2 \cdots \xi_{k-1} 0 1 1 \cdots 1 0 0 \cdots 0.$$

Note that the rightmost 1 in the resulting bit-string has moved to a bit-position $> k$. The reader can verify easily that in subsequent multiply-then-“prune” steps, the rightmost 1 continues to “migrate” rightward, in the sense illustrated in Figure x.

The important thing to note in Figure x is that eventually the 1 that started in bit-position k has been *annihilated*, in the sense that it, and all 1s “spawned” by it in the course of the successive multiply-then-“prune” steps, disappear from the high-order L bit-positions of the sequence of $M(\ell)$ s. When such an annihilation occurs, we have lost

at least the fraction 2^{-k+1} of the GOOD leaves we could conceivably have been salvaging at that point — to the detriment of our ultimate harvest.

The basis for our lower bound on C resides in our ability to bound how many multiply-then-“prune” steps have to take place before a 1 that began in bit-position k of the binary representation of some $M(\ell)$ is annihilated. Since each such annihilation loses us a significant fraction of the GOOD leaves, we wish to maximize the stretches of time between annihilations — by having $M(1)$ assume its maximum possible value, namely, $M(1) = 2^L - 2^{L-C}$, and by “pruning” as few leaves as possible after each multiplication. A corollary of this strategy is that we always strive to have the bit-string in positions $1, 2, \dots, L$ of our expression for the values $M(\ell)$ have maximum possible weight, namely C .

Let us focus on the 1 in bit-position k of $M(\ell + i)$; cf. equations (2, 3). Once this 1 begins to “migrate” in the multiply-then-“prune” game — which occurs when it becomes the rightmost 1 in the surviving representation — and until this 1 is annihilated, the configuration of bit-positions $1, 2, \dots, L$ of $M(\ell + i)$ has the form

$$\xi_1 \xi_2 \cdots \xi_{k-1} 0x.$$

where the bit-string x in bit-positions $k+1, k+2, \dots, L$ has weight at most $C - w + 1$ (by equation (3)). Now, the numerical value of x decreases monotonically during the multiply-then-“prune” game; therefore, the number of multiply-then-“prune” steps required to annihilate the 1 that began in bit-position k , once it starts to move, cannot exceed

$$T(w, k) =_{\text{def}} \sum_{i=0}^{C-w+1} \binom{L-k}{i}. \quad (4)$$

The notation $T(w, k)$ is appropriate for this quantity since the number of steps depends only on the weight and length of the bit-string $\xi_1 \xi_2 \cdots \xi_{k-1}$. Since $C < \frac{1}{3}L$, we can replace the summation in (4) by the more easily manipulated bound

$$T(w, k) < 2 \binom{L-k}{C-w+1}. \quad (5)$$

Inequality (5) on $T(w, k)$ allows us to bound easily the quantities that we are really interested in, namely the following sums, where the integer parameter r ranges over the set $\{0, 1, \dots, \lceil -\log H \rceil + 1\}$. (Recall that $H = \Theta(1)$.)

$$T^*(k; r) = \sum_{i=k}^C T(i-r, i) < 2 \sum_{i=k}^C \binom{L-i}{C+r+1-i} = 2 \binom{L-k+1}{C-k+r+1} - 2 \binom{L-C}{r}. \quad (6)$$

The importance of the quantities $T^*(k; r)$ resides in the following analysis.

If the number of multiply-then-“prune” steps, *from the very beginning of the multiply-then-“prune” game*, is at least

$$T^*(2; 0) = 2 \binom{L - k + 1}{C - k + 1} - 2,$$

then the 1 that began in bit-position 2 of $M(1) = 2^L - 2^{L-C}$ has been annihilated. If the game proceeds even one more step, then it is no longer possible to salvage the fraction $1/2$ of the original GOOD leaves of \mathcal{H}_n .

By similar reasoning, if the game proceeds an additional

$$T^*(3; 1) + 1 = 2 \binom{L - k + 1}{C - k + 2} - 2(L - C) + 1$$

steps, then it is no longer possible to salvage the fraction $1/4$ of the original GOOD leaves of \mathcal{H}_n .

Continuing with this reasoning, if the number of multiply-then-“prune” steps, *from the very beginning of the game*, is at least

$$\sum_{i=2}^{\lceil -\log H \rceil + 1} T^*(i; i - 2) = 2 \sum_{i=2}^{\lceil -\log H \rceil + 1} \left(\binom{L - i + 1}{C - 1} - \binom{L - C}{i - 2} \right) < 2 \binom{L}{C}, \quad (7)$$

then it is no longer possible to salvage the fraction

$$\frac{1}{2^{\lceil -\log H \rceil}} \leq \frac{1}{H}$$

of the original GOOD leaves of \mathcal{H}_n ; in other words, we shall have failed in our assigned task of salvaging at least the fraction H of these leaves.

The only way we can be certain that the multiply-then-“prune” game can not be played long enough to frustrate our attempts to salvage the fraction H of the original GOOD leaves is if the depth n of the LTM \mathcal{H}_n is small enough to preclude our playing the game for this many steps. On the basis of the estimates in equation (7), we must have

$$\frac{n}{L} < 2 \binom{L}{C} \leq 2 \left(\frac{eL}{C} \right)^C$$

(using standard estimates). Elementary manipulation demonstrates that this inequality implies $C \geq L = \Omega(\log n)$. \square

3 Optimizing Expected Congestion

This section is devoted to deriving an analog of the development in Section 2 that exposes the amount of congestion one must incur in order to survive “random” faults in \mathcal{H}_n . In order to discuss random faults and the expected behavior of a salvage algorithm, we must have a fault model in mind. We adopt the model that predominates in the literature by assuming that the leaves of our trees \mathcal{H}_n fail to be GOOD independently, with probability $1/2$.⁶ We turn now to the task of deriving an upper bound on C_{\min} for random faults. We have not yet settled whether or not our bound on C_{\min} can be lowered; this question presents an inviting challenge.

3.1 An Algorithm with Good Expected Behavior

Using the simple fault model just described, we find that a modified version of Algorithm **Worst-Case** of Section 2 produces S-embeddings which incur congestion that is only *triply* logarithmic in the size of the salvaged \mathcal{G}_m , with extremely high probability, providing that we lower our demands a bit. Specifically, we reduce our demand that our algorithm be able to harvest any fraction $H \leq 1/2$ of the GOOD leaves of \mathcal{H}_n to the demand that our algorithm be able to harvest any fraction $H \leq 1/8$ of the GOOD leaves of \mathcal{H}_n .

Theorem 3.1 *Let the leaves of \mathcal{H}_n be labeled GOOD or not, independently, with probability $1/2$. For any rational $0 < H \leq 1/8$, with probability $\geq 1 - 2^{-\Omega(n)}$, a modification of Algorithm **Worst-Case** will find an S-embedding having congestion*

$$C \leq \log^{(3)} N - \log \frac{1 - 4H}{4} + 1 \tag{8}$$

of some \mathcal{G}_m in \mathcal{H}_n , where $2^m \geq HN$.

Proof. The major insight leading to the desired modification of Algorithm **Worst-Case** resides in the following combinatorial fact.

Lemma 3.1 *Let the leaves of \mathcal{H}_n be labeled GOOD or not, independently, with probability $1/2$. If we partition the leaves of \mathcal{H}_n into blocks of size $10n$ in any way, then with probability $\geq 1 - 2^{-\Omega(n)}$, at least $2.5n$ leaves in each block are GOOD.*

⁶Changing the probability $1/2$ to any other fixed probability p merely changes the constants in our results.

Proof of Lemma. The proof proceeds by a series of transformations and estimates. Focus first on a single block of $10n$ leaves.

$$\begin{aligned} Pr(< 2.5n \text{ GOOD leaves}) &= Pr(> 7.5n \text{ not-GOOD leaves}) \\ &= 2^{-10n} \sum_{k=7.5n+1}^{10n} (\text{number of ways to choose } k \text{ not-GOOD leaves}) \end{aligned}$$

This last sum is readily transformed to

$$\sum_{k=0}^{2.5n-1} \binom{10n}{k} 2^{-10n} \leq 2 \binom{10n}{2.5n} 2^{-10n} \leq 2 \left(\frac{10e}{2.5}\right)^{2.5n} 2^{-10n} \leq 2 \left(\frac{\epsilon}{2}\right)^n$$

for some $\epsilon < 1$. It follows that

$$Pr(\geq 2.5n \text{ GOOD leaves}) \geq \left(1 - 2 \left(\frac{\epsilon}{2}\right)^n\right)^{2^n/10n} \geq \exp(-c_1 \epsilon^n/n) \geq 1 - \frac{1}{n2^{c_2n}}$$

for some constants $c_1, c_2 > 0$. \square -Lemma 3.1

Lemma 3.1 tells us that when we look at the labels assigned by our greedy algorithm to nodes at or above level $\lceil \log 10n \rceil$ of a randomly labeled instance of \mathcal{H}_n , then, with very high probability, we find every node having a label $\lambda(x)$ for which $\text{Pot}(\lambda(x)) \geq 2.5n$. This suggests that the following (informally stated) S-embedding algorithm achieves the goals of the Theorem with the indicated high probability. The reader should note that only Step 0 of the algorithm is not guaranteed to work as desired. To simplify exposition, we assume that $10n$ divides 2^n and that $2.5n$ is a power of 2; removing these assumptions is merely a clerical task.

Algorithm Expected-Case:

Step 0. {Select GOOD leaves of \mathcal{H}_n for salvage}

Scan the leaves of \mathcal{H}_n in blocks of $10n$ leaves; within each block, select $2.5n$ to be salvaged; remove the label GOOD from all unselected leaves.

Step 1. {Salvage small trees within each block}

Invoke Algorithm **Worst-Case** within each block, with harvest fraction $1/2$.

Step 2. {Hook up all the small salvaged trees}

Embed the “top” of a complete binary tree to hook together the $2^n/10n$ small, $2^{1.25n}$ -leaf, trees salvaged in Step 1.

\square

By Lemma 3.1, Step 0 of Algorithm **Expected-Case** succeeds, with high probability, to collect the desired number of GOOD leaves within each block. By Theorem 2.1, the congestion incurred by Algorithm **Worst-Case** when salvaging the small trees as mandated in Step 1 is no greater than the bound 8. Since Step 1 salvages just one tree from each block, the embedding of Step 2 incurs no further congestion. This completes the proof, modulo details that are left to the reader. \square

4 Optimizing Worst-Case Harvest

Algorithm **Worst-Case** of Section 2.1 is guaranteed to be efficient, both in running time — it operates in time $O(2^n)$, which is linear in the size of \mathcal{H}_n — and in harvest — it salvages the desired fraction H of the GOOD leaves. But, it is easy to find examples where a nongreedy strategy allows one to salvage a much larger fraction of the GOOD leaves. In particular, when the GOOD leaves are spread out sparsely, any greedy approach abandons many more GOOD leaves than it has to. One finds an analogous deficiency in a “lazy” salvage strategy — one that coalesces small trees as late as possible, rather than as early as possible; lazy strategies abandon too many GOOD leaves when the leaves are packed densely, in clumps. It might be of practical interest, therefore, to find a computationally efficient algorithm that salvages optimally many GOOD leaves, while honoring a prespecified limit on congestion. This section presents such an algorithm.

4.1 The Algorithm

Say that the allowable congestion is C .

Overview. Our salvage algorithm proceeds up \mathcal{H}_n , from level 0 to level n , labeling each node x at level ℓ with a set $\Lambda(x)$ of length- $(\ell + 1)$ integer vectors. Each vector in $\Lambda(x)$ will indicate one possible salvage decision available to x ; in particular, for each vector $\langle \nu_0, \nu_1, \dots, \nu_\ell \rangle$:

- there is an S-embedding in the subtree of \mathcal{H}_n rooted at x of a GOOD-leafed forest \mathcal{F} containing ν_k disjoint copies of \mathcal{G}_k , for $0 \leq k \leq \ell$;
- $\sum_{k=0}^{\ell} \nu_k \leq C$, so that the bound on congestion is always honored.

When we get to the root of \mathcal{H}_n (where $\ell = n$), we select the largest level k for which some vector in the set that labels the root has $\nu_k > 0$. Our harvest, then, is a GOOD-leafed copy of \mathcal{G}_k .

The Labeling/Embedding Procedure. We associate each level- ℓ node x of \mathcal{H}_n with a trie (i.e., a digital search tree [10]) of height $\ell + 1$. This trie will store the label-set $\Lambda(x)$ in the obvious way. We now present the details of Algorithm **Optimal-Harvest**.

Algorithm Optimal-Harvest:

Step 0. {Label nodes on level 0 of \mathcal{H}_n }

Assign each leaf x a label as follows.

$$\Lambda(x) = \begin{cases} \{\langle 1 \rangle\} & \text{if } x \text{ is GOOD} \\ \{\langle 0 \rangle\} & \text{if } x \text{ is red} \end{cases}$$

Step $\ell > 0$. {Label nodes on level ℓ of \mathcal{T}_n }

Substep $\ell.a$ {Assemble the vectors }

Assign each level- ℓ node x a label as follows.

for each pair of length- ℓ vectors $\xi \in \Lambda(x0)$ and $\eta \in \Lambda(x1)$, place the length- $(\ell + 1)$ vector ζ in $\Lambda(x)$, where

$$\zeta_k = \begin{cases} 0 & \text{if } k = \ell \\ \xi_k + \eta_k & \text{if } k \neq \ell \end{cases}$$

endfor

Substep $\ell.b$ {Combine small embedded trees}

for each vector ζ of $\Lambda(x)$, **for all** $0 \leq k < \ell$, **if** component ζ_k of ζ is the sum of a nonzero ξ_k (for some $\xi \in \Lambda(x0)$) and a nonzero η_k (for some $\eta \in \Lambda(x1)$), **then** add to $\Lambda(x)$ a vector ζ' that agrees with ζ except in positions $k, k + 1$; specifically:

$$\zeta'_i = \begin{cases} \zeta_{i+1} + 1 & \text{if } i = k + 1 \\ \zeta_i - 2 & \text{if } i = k \\ \zeta_i & \text{otherwise} \end{cases}$$

and embed the root of a copy of \mathcal{G}_{k+1} in x , routing edges from that root to the roots of two copies of \mathcal{G}_k that are embedded in proper descendants of x
endif

endfor endfor

Substep $\ell.c$ {Honor the congestion bound C }

for each vector $\xi \in \Lambda(x)$

if $\sum_{k=0}^{\ell} \xi_k > C$ **then** replace ξ in $\Lambda(x)$ by all possible vectors ξ' such that

- $\xi'_k \leq \xi_k$ for all $0 \leq k \leq \ell$

- $\sum_{k=0}^{\ell} \xi_k' \leq C$

endif
endfor

□

4.2 Timing Analysis

Theorem 4.1 *Let the leaves of \mathcal{H}_n be labeled GOOD or not, in any way, and let $1 < C \leq n$. Algorithm **Optimal-Harvest** finds, in time*

$$\text{TIME}(n) = O(n^{3C+2}2^n)$$

an S -embedding of some \mathcal{G}_m in \mathcal{H}_n , having congestion $\leq C$ and having optimal harvest among embeddings with congestion C .

Proof. The correctness and the quality of the output of Algorithm **Optimal-Harvest** being obvious, we concentrate on the timing analysis. The number of vectors in the set $\Lambda(x)$ for a level- ℓ node x of \mathcal{H}_n can be no greater than

$$\sum_{k=0}^C \binom{\ell+k}{k} = \binom{\ell+C+1}{C} < \ell^C.$$

This bound is verified by analogy with the problem of assigning $\leq C$ balls to $\ell+1$ urns. (We are selecting $\leq C$ trees, each having one of $\ell+1$ heights, to be carried along to the next step of the Algorithm.)

At each level- ℓ node x , we pair the length- ℓ vectors from the label-sets of its children x_0 and x_1 , in all possible ways. We then add each pair together componentwise, and we append a 0 (at the “high-order” end) of each sum-vector (to increase its length). The pairing operation leads to fewer than ℓ^{2C} pairs of vectors, so the addition step produces fewer than ℓ^{2C} sum-vectors. Producing a sum-vector takes $O(\ell)$ steps. Hence, this part of the processing of node x takes time $O(\ell^{2C+1})$.

Next, we adjust each sum-vector, in order to embed new tree roots. This involves selecting, in all possible ways, one level k such that we can combine two level- k trees into a level- $(k+1)$ tree. This level can be selected in at most ℓ ways, and the combination process requires no more than $O(1)$ operations. Since the set $\Lambda(x)$ initially contains fewer than ℓ^{2C} vectors, this part of the processing of node x takes time $O(\ell^{2C+1})$.

Finally, we “prune” the vectors in $\Lambda(x)$ in order to honor the bound on congestion. Since each vector $\langle \nu_0, \nu_1, \dots, \nu_\ell \rangle$ at a level- ℓ node is in the worst case (before pruning)

the sum of two vectors from level- $(\ell - 1)$ nodes, it is possible that $\sum_{k=0}^{\ell} \nu_k = 2C$. Hence, when we prune a vector in all possible ways, we may be adjusting it by adding as many as

$$\binom{\ell + C + 1}{C} \leq (\text{const})\ell^C$$

“correction vectors”. Each correction is a vector addition requiring $O(\ell)$ steps. Since $\Lambda(x)$ may have grown as large as $O(\ell^{2C+1})$ by this time (due to its expansion during the embedding of new roots), the time required for pruning $\Lambda(x)$ may be as much as (but can be no more than)

$$O(\ell) \cdot O(\ell^C) \cdot O(\ell^{2C+1}) = O(\ell^{3C+2}).$$

Since $\ell \leq n$ in this timing analysis, and since the processing we are analyzing takes place at every node of \mathcal{H}_n — although the processing at lower-level nodes is simpler because they require no pruning — it follows that the time required for this algorithm is

$$\text{TIME}(n) = O(n^{3C+2}2^n).$$

This is certainly within the realm of computational feasibility even when C is as big as, say,

$$C = \frac{1}{3} \left(\frac{n}{\log n} - 2 \right),$$

which makes $\text{TIME}(n)$ quadratic in the size of \mathcal{H}_n , and all the moreso when C is more modest in size. \square

ACKNOWLEDGMENTS: The authors thank Don Coppersmith for helpful suggestions.

The research of S. N. Bhatt was supported in part by NSF Grants MIP-86-01885 and CCR-88-07426, by NSF/DARPA Grant CCR-89-08285, and by Air Force Grant AFOSR-89-0382; the research of F. T. Leighton was supported in part by Air Force Contract OSR-86-0076, DARPA Contract N00014-80-C-0622, Army Contract DAAL-03-86-K-0171, and NSF Presidential Young Investigator Award with matching funds from ATT and IBM; the research of A. L. Rosenberg was supported in part by NSF Grants CCR-88-12567 and CCR-90-13184. A portion of this research was done while S. N. Bhatt, F. T. Leighton, and A. L. Rosenberg were visiting Bell Communications Research.

References

- [1] A. Agrawal (1990): Fault-tolerant computing on trees. Typescript, MIT.

- [2] F.S. Annexstein (1989): Fault tolerance in hypercube-derivative networks. *1st ACM Symp. on Parallel Algorithms and Architectures*, 179-188.
- [3] J.L. Bentley and H.T. Kung (1979): A tree machine for searching problems. *Intl. Conf. on Parallel Processing*, 257-266.
- [4] S. Browning (1980): *The Tree Machine: A Highly Concurrent Computing Environment*. Ph.D. Thesis, CalTech.
- [5] R.D. Chamberlain (1990): Multiprocessor synchronization network: design description. Tech. Rpt. WUCCRC-90-12, Washington Univ.
- [6] R.D. Chamberlain (1991): Matrix multiplication on a hypercube architecture augmented with a synchronization network. Typescript, Washington Univ.
- [7] J. Hastad, F.T. Leighton, M. Newman (1989): Fast computation using faulty hypercubes. *21st ACM Symp. on Theory of Computing*, 251-263.
- [8] J.-W. Hong, K. Mehlhorn, A.L. Rosenberg (1983): Cost tradeoffs in graph embeddings. *J. ACM* 30, 709-728.
- [9] C. Kaklamani, A.R. Karlin, F.T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, A. Tsantilas (1990): Asymptotically tight bounds for computing with faulty arrays of processors. *31st IEEE Symp. on Foundations of Computer Science*, 285-296.
- [10] D.E. Knuth (1973): *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Addison-Wesley, Reading, Mass.
- [11] C.E. Leiserson (1979): Systolic priority queues. *1979 CalTech Conf. on VLSI*, 199-214.
- [12] C. Mead and L. Conway (1980): *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass.
- [13] P. Raghavan (1989): Robust algorithms for packet routing in a mesh. *1st ACM Symp. on Parallel Algorithms and Architectures*, 344-350.