How to Pack Trees

JOSEPH GIL ALON ITAI [*]

Department of Computer Science

Technion – Israel Institute of Technology

Technion City; Haifa; Israel 3200

Email: yogi | itai@CS.Technion.AC.IL

FAX: +972-4-829-4353

February 21, 1999

Abstract

In a virtual memory system, the address space is partitioned into pages, and main memory serves as a cache to the disk. In this setting, we address the following problem: Given a tree, find an allocation of its nodes to pages, so called a packing, which optimizes the cache performance for some access pattern to the tree nodes. We investigate a model for tree access in which a node is accessed only via the path leading to it from the root. Two cost functions are considered: the total number of different pages visited in the search, and the number of page faults incurred. It is shown that both functions can be optimized simultaneously. An efficient dynamic programming algorithm to find an optimal packing is presented. The problem of finding an optimal packing which also uses the minimum number of pages is shown to be NP-complete. However, an efficient approximation algorithm is presented. This algorithm finds a packing that uses the minimum number of pages, and requires at most one extra page fault per search. Finally, we study this problem in the context of dynamic trees which allow insertions and deletions.

[*]The second author was supported by the fund for the promotion of research at the Technion.

1 Introduction

1.1 Motivation

The simplicity and the elegance of the Random Access Machine (RAM) model have made it the model of choice for many algorithm designers. However, it must be recognized that memory is hardly ever "random access". The rapid decrease in the cost-performance ratio of CPUs has not been matched by that of storage units. As a result, true random access machines are still economically infeasible. Most modern computer systems feature a memory hierarchy of several levels: the topmost level is small, fast, and expensive; each successive level is larger, slower and cheaper. The power of this structure is in that, due to caching, it may be able to offer an expected access time close to that of the fastest level while keeping the average cost per memory cell near the cost of the cheapest level.

A few words of reminder may be required here. On-chip-processor-cache and main memory are organized as an array of *pages* with identical and fixed page size. To access an item, the processor first checks whether the memory page holding the item is in its cache. If it is not there, a *cache miss* occurs and the entire memory page is loaded into the cache. A similar structure, but with larger page sizes (typically 4Kb to 32Kb) occurs in a virtual memory operating system where most of the memory resides on disk and is paged in on so called *page faults*. Due to the huge difference in access times between successive memory levels, a significant performance penalty is incurred by cache misses and page faults.

We should therefore also pay attention to questions pertaining to the cost of memory reference in a hierarchical system. One such work is that of Aggarwal, Alpernm, Chandra, and Snir [1] who suggested a computation model in which a charge of $\lg i$ is associated with the access to the ith memory cell.

Finding algorithms that enhance the effectiveness of caches is in a sense a problem dual to finding cache management policies that attempt to optimize the performance of algorithms. from the compiler optimization and numerical algorithms communities. A significant body of research (see [10] for a brief survey) demonstrated performance enhancement in many numerical algorithms operating on matrices. The key idea behind these results was the alignment of the algorithm and the data structures to achieve better cache performance.

The performance gains were obtained by non-trivial loop transformations and judicious layouts of the matrices with the aim of reducing the number of *cache-misses* and *page-faults*. Although a more precise definition is required, the basic principle is simple: since a "locality of reference" assumption underlies every hierarchical memory system, a program coerced to exhibit this kind of behavior, will perform better.

Improving the locality of memory references has also a positive impact on the performance of parallel performance by reducing communication overheads. Indeed, we see reports [11] of these optimizations leading to almost doubling the measured speedup on a multi-processor system, bringing it close to the theoretical maximum.

A natural question arising here is whether combinatorial computing can capitalize on similar techniques. In this paper, we try to give a partial answer to this problem by studying the issue of memory-level sensitive representation of trees in memory. Why trees? Most importantly, large trees are a ubiquitous data representation. They occur in a wide range of application domains including AI (Lisp functions and data), text processing (e.g., SGML [8] documents), and geometric modeling in CAD systems [5]. Also, trees are more interesting than matrices in that they have a greater variety of topologies and since the pointer structure allows for many different layouts in memory.

1.2 Definitions

To continue the discussion in a more rigorous manner we need the following notations and definitions. Let T be a given input tree of n nodes, whose root is r. We say that a node v belongs to a tree T, and by abuse of notation write $v \in T$. For a node $v \in T$, T_v denotes the sub-tree of T whose root is v. If $v \neq r$, then $\mathrm{parent}(v)$ denotes its parent. Also, if T is a binary tree then the left and right children of v are denoted by $\mathrm{left}(v)$ and $\mathrm{right}(v)$. A *descent* is a sequence $v_1, \ldots, v_k$ such that $v_i = \mathrm{parent}(v_{i+1})$ for $i = 1, \ldots, k-1$. We say that the descent leads from v_1 to v_k. The *path* to a node v is a descent leading from r to it; $\ell(v)$ denotes the number of edges in this path.

The nodes of T are to be represented in memory as records and its edges as pointers. We assume that every node requires one unit of space. Memory is partitioned into *pages*, each of which can contain p nodes. A *packing of* T is a function τ which maps its nodes to memory pages; formally $\tau : T \to Z^+$ such that $|\tau^{-1}(i)| \leq p$. A page is *full* if $|\tau^{-1}(i)| = p$. The *space* of a packing is the number of pages it uses, i.e., $|\tau(T)|$. We say that a packing is *compact* if its space consumption achieves the minimum, $N = \lceil n/p \rceil$; it is *k-compact* if its space usage is kN.

1.3 The problem

Operating systems' jargon includes the idiom "working set of a process"—a rather loosely defined term which refers to the set of pages a process will reference in the "near future". Virtual memory systems work only because in practice the working set is small, and most of it is stored in internal memory[1]. If a large tree is packed randomly in pages, then the working set of a program manipulating it is large and its processing may lead to extensive disk activity and even to thrashing. Our overall goal is therefore to find a packing that minimizes the working set.

The model we use for patterns of access to the tree is that of paths. A node is accessed only by traversing the path leading to it. With each node v we associate a positive weight $w(v)$ that can be thought of as the probability of accessing it (although weights are not necessarily normalized). We also use the notation

$$w(T_v) = \sum_{u \in T_v} w(u) \ .$$

The model is quite general and corresponds to a large family of tree algorithms ranging from ordinary search trees to ray tracing [4].

Consider a packing τ and the path to the node v: $r = v_0, v_1, \ldots, v_{\ell(v)} = v$. There are two extremal cases:

1. Suppose that the cache can accommodate only one page which is initially used for some other purpose. Then, the number of page faults that occur along a path is exactly the number of times *a page boundary is crossed*. More formally, we define for every node u

$$\Delta_\tau(u) = \begin{cases} 1 & \text{if } u = r \\ 0 & \text{if } \tau(\text{parent}(u)) = \tau(u) \\ 1 & \text{otherwise} \ . \end{cases}$$

   The *page fault cost* associated with the path to v, $PF_\tau(v)$, is defined as

$$PF_\tau(v) = \sum_{i=0}^{\ell(v)} \Delta_\tau(v_i) \ .$$

2. Conversely, suppose that the size of the cache is unlimited. The appropriate charge for a path in this case is the *working set cost*, $WS_\tau(v)$, the number of *distinct pages accessed* along it, defined formally as

$$WS_\tau(v) = |\{\tau(v_i) \mid 0 \le i \le \ell(v)\}| \ .$$

[1] For the sake of concreteness, we chose to relate to disk-caching in memory. However, the discussion equally applies to caching of main memory in a separate or on-chip CPU cache.

Two functions which should be used as targets for our global optimizations are therefore $PF_\tau(T)$ and $WS_\tau(T)$, defined as

$$\begin{aligned} PF_\tau(T) &= \sum_{v \in T} w(v) PF_\tau(v) \ , \\ WS_\tau(T) &= \sum_{v \in T} w(v)\, WS_\tau(v) \ . \end{aligned}$$

A packing τ is said to be *page-fault-optimal* if it minimizes $PF_\tau(T)$, and *working-set-optimal* if it minimizes $WS_\tau(T)$.

For a page P and a node v, $\tau(v) = P$, we say that v belongs to P, and by abuse of notation, we write $v \in P$.

We observe that a trivial instance of the problem occurs when $|T| \leq p$. This case is tacitly excluded from our consideration.

2 Results

In this section we present our main results. Proofs are postponed to later sections.

Intuition may also lead us to believe that in a page-fault-optimal packing a path never visits the same page twice. Indeed, this is the case. For an exact statement of this property we need the following definition.

Definition 1 *A packing is* convex *if for every descent $u_1, \ldots, u_k$ for which $\tau(u_1) = \tau(u_k)$ then for $i = 1, \ldots, k$, $\tau(u_i) = \tau(u_1)$.*

As it turns out, convexity is a property of not only page-fault optimal packing, but also of working-set optimal packings.

Theorem 1 (proof in Section 3) *Let τ be a packing. Then, if τ is page-fault-optimal or if τ is working-set-optimal then τ is convex.*

How about working-set-optimal packings? Perhaps surprisingly, we can show that both of our, seemingly different, complexity measures, *WS* and *PF*, can in fact be optimized simultaneously.

Theorem 2 (proof in Section 3) *A mapping is page-fault optimal if and only if it is working set optimal.*

We therefore may use the term *optimal* to refer to both *page-fault-optimal* and *working-set-optimal*. and denote the cost of an optimal packing τ^{opt} as $C_{\tau^{\text{opt}}}(T) = PF_{\tau^{\text{opt}}}(T) =$

$WS_{\tau^{\mathrm{opt}}}(T)$. (We omit T when it is understood by the context.) As it turns out, an optimal packing can be computed efficiently by using dynamic programming in a bottom-up traversal of the tree.

Theorem 3 (proof in Section 4) *Let T be a tree of n nodes and degree d. Then, an optimal packing can be computed in time $O(np^2 \log d)$ while using $O(p \log n)$ space.*

The packing obtained by this algorithm is 2-compact. Is it also possible to simultaneously optimize space and cache performance? Unfortunately, the answer is (most likely) negative.

Theorem 4 (proof in Section 5) *The problem of computing an optimal compact packing is NP-complete.*

On the other hand, it is possible to achieve a compact packing in which the average cost is not much worse than that of the optimal.

Theorem 5 (proof in Section 6) *Let τ^{opt} be an optimal packing. Then, in $O(n \log n)$ time and using $O(p)$ space, it is possible to compute from it a compact packing τ^{appr} such that*

$$\frac{C_{\tau^{\mathrm{appr}}}}{n} \leq \frac{C_{\tau^{\mathrm{opt}}}}{n} + \frac{1}{n}\sum_{v \in T} w(v) \ .$$

Furthermore, if all nodes v, $w(v) = 1$ then

$$C_{\tau^{\mathrm{appr}}}/n \leq C_{\tau^{\mathrm{opt}}}/n + 0.5 \ .$$

Outline Most of this paper is devoted to the question of packing fixed trees. The dynamic version of this problem, that is, efficiently updating the packing as the tree changes due to, say, insertions and deletions, seems to be much more difficult. In Section 7 we discuss this further and give some partial results that show how to maintain a reasonably good packing for two specific schemes of tree updates: B-trees and a variation of weight-balanced trees.

3 Equivalence of the two Complexity Measures

In order to show that the page-fault and the working-set complexity measures are equivalent, we show that the optimal packings for either measures are convex.

Lemma 3.1 *All working-set-optimal packings are convex.*

Proof. Suppose to the contrary that there exists a non-convex working-set-optimal packing τ. Each non-convex descent, $u_1, \ldots, u_k$ in τ can be extended into a path by prefixing it with the path that leads from r to parent(v_1). Let $v_1, \ldots, v_\ell, u_1, \ldots u_k$, $u_1 = r$, be a shortest path that can be obtained in this manner, that is, one achieving the minimum of $\ell + k$. Then, the path $v_1, \ldots, v_\ell$ is convex, and it never revisits a page after leaving it. In other words, for $i = 1, \ldots \ell - 1$, if $\tau(v_{i+1}) \neq \tau(v_i)$, then for all $j = 1, \ldots, i-1$, $\tau(v_{i+1}) \neq \tau(v_j)$.

We may assume that $v_1, \ldots, v_\ell$ is the longest convex prefix of our path. Then, $\tau(u_1) = \tau(u_k)$ and $\tau(u_{k-1}) \neq \tau(u_k)$. Let j be the minimum index for which $\tau(u_j) = \tau(u_{k-1})$. The reader is advised to consult Figure 1 as a reference at this stage. Define the mapping τ':

$$\tau'(v) = \begin{cases} \tau(u_k) & \text{if } v = u_j \\ \tau(u_j) & \text{if } v = u_k \\ \tau(v) & \text{otherwise .} \end{cases}$$

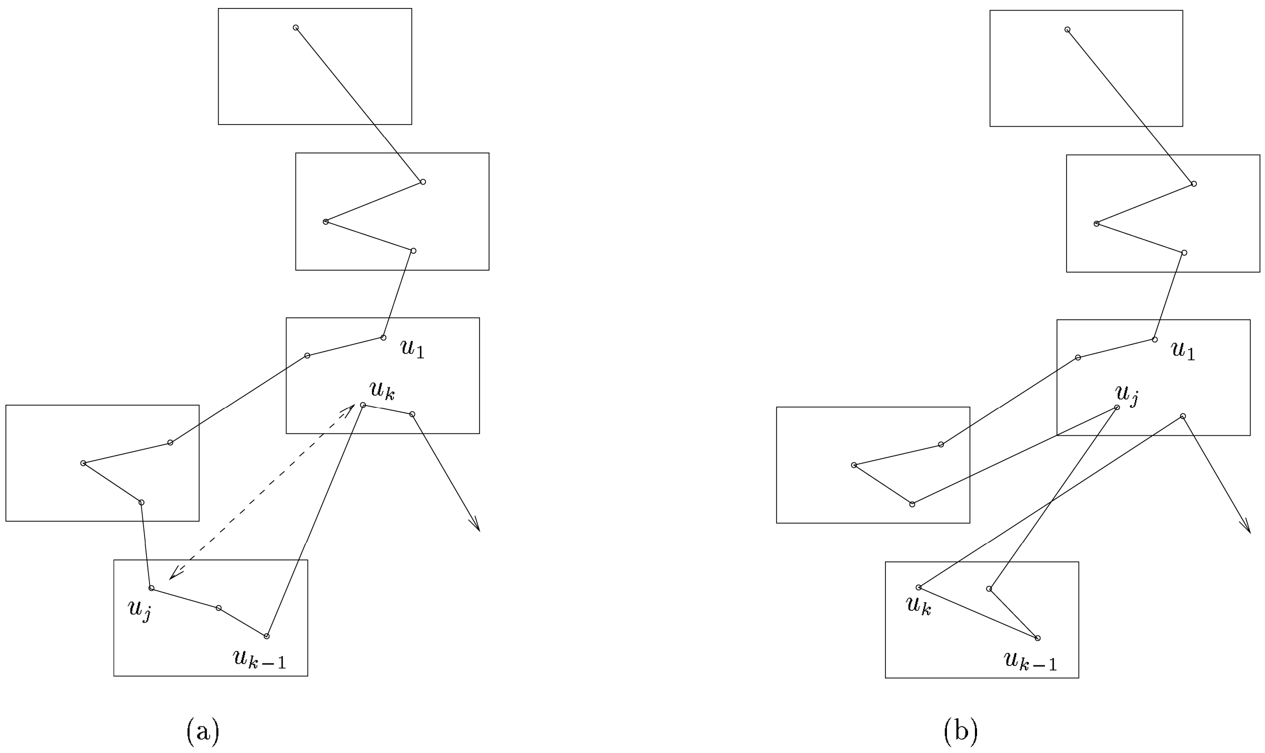

Figure 1: A non-convex descent in τ and in τ' (proof of Lemma 3.1)

For vertices w which are not descendents of u_j, the working set of w with respect to τ is equal to the working set with respect to τ'. Since for every descendent v of u_j the working set of v with respect to τ already contains the page $\tau(u_1)$, $WS_{\tau'}(v) \leq WS_\tau(v)$.

However, the working set of u_j with respect to τ' does not contain the page $\tau(u_j)$, thus $WS_{\tau'}(u_j) < WS_{\tau}(u_j)$. Hence, $WS_{\tau'} < WS_{\tau}$. Contrary to the optimality of τ. ∎

This proves the working set half of Theorem 1.

Since for every vertex v and every mapping τ, $WS_{\tau}(v) \leq PF_{\tau}(v)$, we get that this property also holds globally. Thus, for every mapping τ, $WS_{\tau} \leq PF_{\tau}$. In particular, for convex mappings, we can assert:

Claim 3.1 *For every convex mapping τ,*

$$\mathrm{WS}_{\tau} = \mathrm{PF}_{\tau} \ .$$

It follows that:

Corollary 3.2 *Let τ be a page-fault optimal mapping. Then*

$$\mathrm{WS}_{\tau} = \mathrm{PF}_{\tau} \ .$$

Proof. If the claim does not hold then there exists a page-fault optimal mapping τ that is not working-set optimal. Let τ' be a working-set optimal mapping. Then

$$WS_{\tau'} < WS_{\tau} \ . \tag{1}$$

Since τ' is working-set optimal then by Lemma 3.1 it is convex. By Claim 3.1

$$WS_{\tau'} = PF_{\tau'} \ . \tag{2}$$

By combining (1) and (2) we obtain

$$PF_{\tau'} < PF_{\tau} \ ,$$

contradicting the assumption that τ is page-fault optimal. ∎

Theorem 2 follows immediately. Further, this also completes the proof of the page-fault-optimal half of Theorem 1.

From now on, the term *optimal mapping* will denote a mapping which is both working-set-optimal and page-fault-optimal.

4 Finding an Optimal Packing

In this section we give the dynamic programming algorithm that lies behind Theorem 3. We begin by deriving the additional properties of optimal packings on which the algorithm is based. Next, the basic algorithm for binary trees is presented. This algorithm uses relatively large space for its computation. We then show how space consumption can be reduced. Finally, the extension to trees of arbitrary degree is described.

4.1 Properties of optimal packings

It is natural to conjecture that the page that contains the root must be full.

Lemma 4.1 *Let τ be a page-fault-optimal packing, and $P = \tau(r)$. Then, P is full.*

Proof. Suppose to the contrary that P is not full. Then, since we assumed $|T| > p$, there must exist a node v, $v \notin P$, parent$(v) \in P$. Moving v to P decreases $PF(v)$ without increasing $PF(u)$ for any other node u. ■

More generally, any "internal page" must be full, in an optimal packing, as stated formally in the following lemma.

Lemma 4.2 *Let τ be an optimal packing of T. Let P be a non-full page of τ. Then, P must contain a leaf of T, and furthermore, if $v \in P$, then $T_v \subseteq P$.*

Proof. Suppose that P does not contain a leaf, or that it has a node u such that not all of T_u is in P. Then, P must contain a vertex v that has a child $w \notin P$. Since P is not full, w can be moved to P, while reducing $PF(w)$ and not effecting the PF of any other vertex. ■

Lemma 4.3 *There exists an optimal packing τ such that for every page P the subgraph of T induced by the set $\tau^{-1}(P)$ is also a tree.*

Proof. Let τ be an optimal mapping for which the lemma does not hold. The following procedure constructs from τ an alternative optimal mapping τ' which satisfies the lemma. Replace each page P that contains a forest of $k_P > 1$ trees by k_P pages, each containing a single tree. For each vertex v, $PF_\tau(v) = PF_{\tau'}(v)$ and therefore $PF_\tau = PF_{\tau'}$. ■

Let $\mathcal{T}_\tau$ be the graph whose nodes are the pages of τ and (P, P') is an edge of $\mathcal{T}_\tau$ if and only if there exists $v \in \tau^{-1}(P')$ such that parent$(v) \in \tau^{-1}(P)$.

Lemma 4.4 *There exists an optimal packing τ such that $\mathcal{T}_\tau$ is a tree.*

Proof. Let τ be an optimal mapping which satisfies Lemma 4.3. The connectivity of $\mathcal{T}_\tau$ follows from the connectivity of T. If $\mathcal{T}_\tau$ does not contain a cycle, we are done. Otherwise, let $P_1, \ldots, P_q$ be a cycle in $\mathcal{T}_\tau$. Since (P_i, P_{i+1}) is an edge of $\mathcal{T}_\tau$, there exists $v_i \in \tau^{-1}(P_i)$ and $u_{i+1} \in \tau^{-1}(P_{i+1})$ such that $(v_i, u_{i+1}) \in T$. There also exists $v_q \in \tau^{-1}(P_q)$ and $u_1 \in \tau^{-1}(P_1)$ such that $(v_q, u_1) \in T$. Since by Lemma 4.3 each $\tau^{-1}(P_i)$ is a tree, $\tau^{-1}(P_i)$ contains a path, $\pi_i[u_i, v_i]$, from u_i to v_i. Connecting these paths with the edges (v_i, u_{i+1}) yields the cycle

$$\pi_1[u_1, v_1](v_1, u_2)\pi_2[u_2, v_2](v_2, u_3) \cdots (v_{q-1}, u_q)\pi_q[u_q, v_q](v_q, u_1) \in T \ .$$

Thus T contains a cycle, contradicting the assumption that it is a tree. ■

Corollary 4.5 *Let τ be an optimal packing and $v \in T \setminus \{r\}$ a node for which $\tau(v) \neq \tau(\text{parent}(v))$. Then, the pages assigned by τ to the nodes of T_v are disjoint from those assigned to the remainder of the tree. Further, the restriction of τ to T_v is an optimal packing of T_v.*

4.2 Algorithm for Binary trees

In this section we assume that T is a binary tree, i.e., each node has at most two children. Since initially all pages are isomorphic, we may assume that the root r is always mapped to a fixed page R. Consider the set

$$V = \{v \in T \mid \tau(v) \notin R, \tau(\text{parent}(v)) = R\} \ .$$

By Corollary 4.5, τ is an optimal packing of T_v for all $v \in V$. Thus, in order to find an optimal packing, it may be possible to first determine which other nodes reside in R and then continue recursively with all trees T_v, $v \in V$.

Dynamic programming provides a more efficient implementation of this idea. However, in order to implement this technique, we consider the following slightly more general problem: For $i = 1, \ldots, p$ an *i-confined* packing of a tree T is a packing of T in which R contains i nodes.

Suppose that in an optimal packing, exactly i nodes of $T_{\text{left}(r)}$ are mapped to R. Then τ, when restricted appropriately, is both an optimal i-confined packing of $T_{\text{left}(r)}$ and an optimal $(p-i-1)$-confined packing of $T_{\text{right}(r)}$. This property is the basis of our bottom-up dynamic programming algorithm. To find an optimal packing of T, we first find optimal i-confined packing of $T_{\text{left}(r)}$ and $T_{\text{right}(r)}$ for all values of i.

For node $v \in T$ let $A[v, i]$ be the cost of an optimal i-confined packing of T_v. If exactly i $(1 \leq i \leq p-2)$ of the nodes of $T_{\text{left}(r)}$ are mapped to R, then exactly $p-1-i \geq 1$ nodes of $T_{\mathrm{right}(r)}$) are mapped to R. Accounting for the cost of accessing r itself we have

$$A[r,p] \quad = \quad w(r) + A[\mathrm{left}(r), i] + A[\mathrm{right}(r), p-1-i] \ . \tag{3}$$

What about the extremal cases, $i = 0$ and $i = p-1$? If $T_{\mathrm{left}(r)} \cap R = \emptyset$, then all the nodes of $T_{\mathrm{left}(r)}$ incur a page fault when going from r to left(r). Hence, the cost of T in this case is

$$A[r,p] \quad = \quad w(r) + w(T_{\mathrm{left}(r)}) + A[\mathrm{left}(r), p] + A[\mathrm{right}(r), p-1] \ . \tag{4}$$

Similarly, in the case $T_{\mathrm{right}(r)} \cap R = \emptyset$, we have

$$A[r,p] \quad = \quad w(r) + A[\mathrm{left}(r), p-1] + w(T_{\mathrm{right}(r)}) + A[\mathrm{right}(r), p] \ . \tag{5}$$

Generalizing over equations (3), (4), and (5) for every node v and for any number $i = 2, \ldots, p$ of nodes in the page of v, we obtain:

$$A[v,i] = w(v) + \min \left(\begin{array}{l} A[\mathrm{left}(v), i-1] + w(T_{\mathrm{right}(v)}) + A[\mathrm{right}(v), p], \\ w(T_{\mathrm{left}(v)}) + A[\mathrm{left}(v), p] + A[\mathrm{right}(v), i-1], \\ \min_{1 \le j < i-1} \left(A[\mathrm{left}(v), j] + A[\mathrm{right}(v), i-j-1] \right) \end{array} \right) , \tag{6}$$

where $A[\mathrm{left}(v), i]$ is defined, for notational convenience, as 0 whenever left(v) does not exist (and similarly for right(v)). The case $i = 1$ is special and is given by

$$A[v,1] = w(T_v) + A[\mathrm{left}(v), p] + A[\mathrm{right}(v), p] \ . \tag{7}$$

This gives us the desired dynamic programming algorithm. The algorithm is detailed in Figure 2. It is easy to check that the algorithm runs in $O(np^2)$ time.

4.3 Reducing the space requirement of the algorithm

The algorithm computes the np values of $A[v,i]$. Therefore a naive implementation would use $O(np)$ space. In this section we show how to save space without compromising the time complexity.

In examining the algorithm we find that in order to calculate $A[v,\cdot]$, we need only the values of $A[\mathrm{left}(v),\cdot]$ and $A[\mathrm{right}(v),\cdot]$. After $A[v,\cdot]$ has been calculated, the values of $A[\mathrm{left}(v),\cdot]$ and $A[\mathrm{right}(v),\cdot]$ can be discarded. Let $\sigma(t)$ denote the number of nodes v for which at time t the value of $A[v,\cdot]$ has been calculated, but $A[\mathrm{parent}(v),\cdot]$ has not yet been calculated. Then, at time t we need $p\sigma(t)$ space. The total space requirement is $p \max_t\{\sigma(t)\}$. We wish to organize the computation so as to minimize

$$\max_t\{\sigma(t)\} \ .$$

```
For all leaves $v$ of $T$ do
    begin
        mark $v$;
        for $i := 1$ to $p$ do
            $A[v,i] := w(v)$;
    end;
While there exists an unmarked node $v$ do
    begin
        mark $v$;
        for $i := 2$ to $p$ do
            update $A[v,i]$ according to equation (6);
        update $A[v,1]$ according to equation (7);
    end;
```

Figure 2: Dynamic programming algorithm for computing an optimal packing of a binary tree T .

This problem has been studied before in a different context: finding the smallest number of registers required to calculate an arithmetic expression [2]. If T denotes a binary expression tree where the leaves are data elements and the internal nodes are operators, then $\sigma(v)$, the minimum number of registers required to calculate the value of a node v, can be found by the following recursive formula:

$$\sigma(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf,} \\ \max\{\sigma(\text{left}(v)), \sigma(\text{right}(v))\} & \text{if } \sigma(\text{left}(v)) \neq \sigma(\text{right}(v)) \\ 1 + \text{left}(v) & \text{otherwise .} \end{cases}$$

To calculate the value of the arithmetic expression using $\sigma(r)$ registers, we first calculate the left or right subtree of r with the larger value of σ , then free all the registers except the one holding the root of that subtree. We then calculate the other subtree using $\sigma(r) - 1$ registers. Again we free all the registers except those holding the roots of the children of r. Since $\sigma(r) \geq 3$ we have a remaining register to store the value of r. The maximum value of $\sigma(r)$ over all n-node trees occurs for the complete binary tree. In this case, $\sigma(T) = 2+\lg(n)$.

By using $\sigma(r)$ registers, each of size $O(p)$, to store the values of $A[v,\cdot]$ of "active" nodes, we can compute $A[r,\cdot]$ and hence also the minimal value of $PF_\tau(T)$. However, since all the temporary values are discarded, this space saving version of the algorithm yields only the values $A[v,p]$ and not the mapping $\tau(v)$ for all $v \in T$. It is possible to retrace the evaluation in order to compute the actual packing: Since we know the number of nodes in $T_{\text{left}(r)}$ that were mapped to R we may calculate $\tau(r)$, $\tau(\text{left}(r))$ and $\tau(\text{right}(r))$. However, to calculate the optimal mapping of the grandchildren of r we must rerun the algorithm on $T_{\text{left}(r)}$ and $T_{\text{right}(r)}$. The running time of this modified version is $O(p\sum_{vT}|T_v|)$. For a full tree this is $O(pn\text{log}n)$. For trees of depth $\Omega(n)$, the running time is in the order of n^2.

We now show a better tradeoff between space and time. For all $k > 0$ we compute the optimal packing τ of any binary tree in $O(p^2kn)$ time and $O(pkn^{1/k})$ space. In particular, for $k = \text{log}n$ we achieve $O(p^2n\text{log}n)$ time and $O(p^2\text{log}n)$ space.

We first consider $k = 2$. Recall that every n-node tree T has a vertex v such that both T'_v and $T \setminus T'_v$ have at least $n/3$ vertices. A simple generalization of this argument shows that T can be partitioned into subtrees $T'_1, \ldots, T'_{\sqrt{n}}$, each having between $\sqrt{n}/2$ and $2\sqrt{n}$ nodes.

Let $r_1, \ldots, r_m$ $(m \leq \sqrt{n})$ be the roots of these trees. We may assume that the root of the subtree T'_1 is also the root of T, i.e., $r_1 = r$. To save space, we compute $A[r,p]$ using our space saving scheme, but retain the values of $A[r_j, \cdot]$ $(j = 2, \ldots, m)$.

Consider the tree T'_1. Its leaves are either one of the r_j's or a leaf of T. Thus, we may recompute the values of $A[v, \cdot]$ for all $v \in T'_1$ using $O(p\sqrt{n})$ space. We now compute the optimal packing for $v \in T'_1$. Also, for every leaf of T'_1 which belongs to $\{r_2, \ldots, r_m\}$ we know the number of nodes of its subtree that belong to the same page as that leaf. We store this information, and free all the space used to compute $A[v, \cdot]$, $v \in T'_1$.

In the general step, let $\Gamma \subseteq \{T'_2, \ldots, T'_m\}$ be the set of trees for which we have not yet calculated the packing, but whose root is a leaf of a subtree for which the packing has been calculated. Pick $T'_j \in \Gamma$, recompute $A[v, \cdot]$ for all $v \in T'_j$, and update its leaves as we did for T'_1.

At any given time we need the values of $A[v, \cdot]$ only for $v \in \{r_1, \ldots, r_m\}$ and for one of the subtrees T'_j. The working space requirements are hence at most $p(m + \sqrt{n}) = 2p\sqrt{n}$.

The values of $A[v, \cdot]$ are calculated once for $v \in \{r_1, \ldots, r_{\sqrt{n}}\}$ and twice for the remaining nodes. Thus, the running time is multiplied by a factor of two.

To reduce the working space requirements even further to $O(pkn^{1/k})$, partition T into $n^{1/k}$ subtrees each of size between $n^{(k-1)/k}$ and $2n^{(k-1)/k}$. Then compute $A[r, \cdot]$, and store the values of $A[r_j, \cdot]$ for the roots of these subtrees. Finally, apply the procedure recursively on all these subtrees going from the subtree containing the root down.

The depth of the recursion is k. For each level of recursion we store $\leq 2n^{1/k}$ tables, making the total storage requirements $O(pkn^{1/k})$. The value of $A[v, \cdot]$ is calculated at most once at each level of the recursion. Thus, the running time is multiplied by a factor of k.

To prove Theorem 3 we set $k = \lg n$.

4.4 Arbitrary degree trees

Our results can be extended to trees of arbitrary degrees: Let v be a vertex with d children. Construct a full binary tree $\tilde{T}^v$ with d leaves. Then, $\tilde{T}^v$ has $2d-1$ nodes and is of height $\lg_2 d$. We identify v with the root of $\tilde{T}^v$, and the children of v with its leaves, and replace the edges from v to its children by $\tilde{T}^v$. Repeating this procedure for all internal vertices v yields a binary tree $\tilde{T}$, which has at most twice as many vertices as T.

To find an optimal packing of T, we run a modified version of the binary tree algorithm which accounts for the fact that the new vertices have zero weight and do not occupy any space.

5 Optimal Compact Packing is NP-Complete

In the previous section, we looked for a packing that minimized the number of page misses, and disregarded the number of pages required for the packing. In this section we show that if we insist on a compact packing, i.e., one that uses the minimum number of pages, then finding the time-optimal packing becomes NP-complete. The next section will show that slightly relaxing the minimality of page misses requirement allows us to make the solution of Section 4 compact. The penalty of the increase in the expected number of page misses is a small additive factor.

Consider the following decision problem:

TREE-PACKING

Instance: An integer p, a binary tree T of $n = mp$ nodes, and a positive integer C.

Question: Does there exist a compact packing of cost C, i.e., a packing τ such that $WS_\tau(T) \leq C$ and $|\tau(T)| = m$?

We can now turn to the proof of Theorem 4, namely showing that TREE-PACKING is NP-complete. The problem is in NP since given a packing τ it takes polynomial time to determine its cost and to check that it satisfies the compactness condition. To see that it is complete in NP, we present a reduction from the following 3-PARTITION problem, which is strongly NP-complete.

3-PARTITION

Instance: Set A of $3m$ elements, a bound $q \in Z^+$, and a size function $s(a) \in Z^+$ such that $q/4 < s(a) < q/2$ and

$$\sum_{a \in A} s(a) = mq \ .$$

Question: Can A be partitioned into m disjoint sets $A_1, \ldots, A_m$ such that

$$\sum_{a \in A_i} s(a) = q$$

for all $i = 1, \ldots, m$?

Note that it follows from the definition of the problem that $|A_i| = 3$ for all i.

The 3-PARTITION problem was shown by Garey and Johnson [7, pp. 96–105] to be NP-complete in the strong sense, i.e., it remains NP-complete even when all the numbers $s(a)$, $a \in A$, are written in unary.

We first show that the problem remains NP-Complete even when we restrict the bound q to be of the form $4^b - 1$, integer b. To do so, we replace the bound q by p which is of the desired form and is at least three times greater than q (but no greater than $12q$). Define

$$\begin{aligned} b &= \lceil \log_4 3q \rceil \ , \\ p &= 4^b - 1 \ , \\ \Delta p &= p - 3q \ , \\ s'(a) &= 3s(a) + \Delta p/3 \ . \end{aligned}$$

To see that $s'(a)$ attains only integer values, note that 2^b is not divisible by 3, hence,

$$p = \left(2^b\right)^2 - 1 = (2^b - 1)(2^b + 1)$$

must be divisible by 3, and therefore, Δp is also divisible by 3. Moreover, since $p + 1 = 4^b = 4^{\lceil \log_4 3q \rceil} \geq 3q$, and p is divisible by 3, we get

$$p \geq 3q \ . \tag{8}$$

The following lemma is required to show that the new definitions form an instance of 3-Partition.

Lemma 5.1 *For all a, $p/4 < s'(a) < p/2$.*

Proof. From the definition of $s'(a)$ and Δp,

$$s'(a) = 3s(a) + \Delta p/3 \ .$$

Since $s(a) < q/2$,

$$s'(a) < 3\frac{q}{2} + \frac{p}{3} - q = \frac{q}{2} + \frac{p}{3} \ .$$

By (8),

$$s'(a) \leq \frac{p}{6} + \frac{p}{3} = \frac{p}{2} \ .$$

For the lower bound side of the lemma, we have

$$s'(a) = 3s(a) + \Delta p/3 = 3s(a) + \frac{(p-3q)}{3} \ .$$

Now, since $s(a) > q/4$,

$$s'(a) > 3 \cdot \frac{q}{4} + \frac{(p-3q)}{3} = \frac{p}{3} - \frac{q}{4}$$

By (8)

$$s'(a) > \frac{p}{3} - \frac{p/3}{4} = \frac{p}{4} \ .$$

■

It remains to show that $A_1, \ldots, A_m$ is a solution of 3-PARTITION(s,q) if and only if it is a solution of 3-PARTITION(s',p). Assume therefore that $A_1, \ldots, A_m$ is a solution of 3-PARTITION(s,q), then

$$\sum_{a \in A_i} s'(a) = 3 \sum_{a \in A_i} s(a) + \sum_{a \in A_i} \Delta p/3 \ .$$

Since A is a solution to 3-PARTITION(s,q) and since $|A_i| = 3$,

$$\sum_{a \in A_i} s'(a) = 3q + \Delta p = p \ .$$

This variation on 3-PARTITION is also strongly NP-complete because there is only a polynomial increase in the input size: Since $p < 4^{\lceil \log_4 3q \rceil} < 4^{1+\log_4 3q} = 4 \cdot (3q) = 12q$. Also,

$$s(a) < s'(a) < p \leq 12q < 35s(a) \ .$$

Therefore $s'(a) = \theta\left(s(a)\right)$.

We now show that TREE-PACKING is NP-Complete by reducing an instance of 3-PARTITION(s',p) with $p = 4^b - 1$ to an instance of TREE-PACKING with page size p. The tree T is constructed from the following components:

1. *The full binary tree* $\mathbf{P}$ *of height* $h_{\mathbf{P}} = 2b - 1$. The tree $\mathbf{P}$ has

$$2^{h_{\mathbf{P}}+1} - 1 = 4^b - 1 = p$$

nodes and therefore fits exactly in a page.

2. *The full binary tree* $\mathbf{M}$ *of height* $h_{\mathbf{M}} + 1$ *and* M *leaves.* The tree $\mathbf{M}$ is chosen to be the minimum tree which satisfies

$$M \geq 3m$$
$$h_{\mathbf{P}} + 1 \text{ divides } h_{\mathbf{M}} + 1 \ .$$

An optimal packing of $\mathbf{M}$ is obtained by naturally partitioning it into copies of the full binary tree $\mathbf{P}$: Start at the root of $\mathbf{M}$, and put the subtree isomorphic to $\mathbf{P}$ rooted at the root of $\mathbf{M}$ in one page. Then, continue recursively with each of the $(p+1)/2$ remaining sub-trees. An example of this packing is given in Figure 3. Note that all the pages thus generated are completely full. Let $C_{\mathbf{M}}$ denote the cost of this packing.

3. *Trees* $\mathbf{S}(a)$ *for all* $a \in A$*, where* $\mathbf{S}(a)$ *is an arbitrary tree with* $s'(a)$ *nodes.*

The tree T is created from $\mathbf{M}$ by hanging subtrees on its leaves as follows. For all M leaves of $\mathbf{M}$ a copy of $\mathbf{P}$ is added as a right child. In the first $3m$ leaves of $\mathbf{M}$, we add the trees $\mathbf{S}(a)$ in sequence as left children. A copy of $\mathbf{P}$ is added as a left child of the remaining $M - 3m$ leaves.

The tree $\mathbf{M}$ has $2M - 1$ nodes. Also, $T \setminus \mathbf{M}$ contains $2M - 3m$ copies of $\mathbf{P}$. The total number of nodes added by the trees $\mathbf{S}(a)$ is

$$\sum_{a \in A} |\mathbf{S}(a)| = \sum_{a \in A} s'(a) = mp \ .$$

Thus, the total number of nodes of T is

$$(2M - 1) + (2M - 3m)p + mp = (2M - 1) + 2(M - m)p \ .$$

Denote

$$h = \frac{h_{\mathbf{M}} + 1}{h_{\mathbf{P}} + 1} \ .$$

Lemma 5.2 *There exists a compact packing of* T *into pages of size* p *which has cost*

$$C_T = C_{\mathbf{M}} + 2(M - m)(h + 1)p$$

iff there exists a solution to the 3-PARTITION problem.

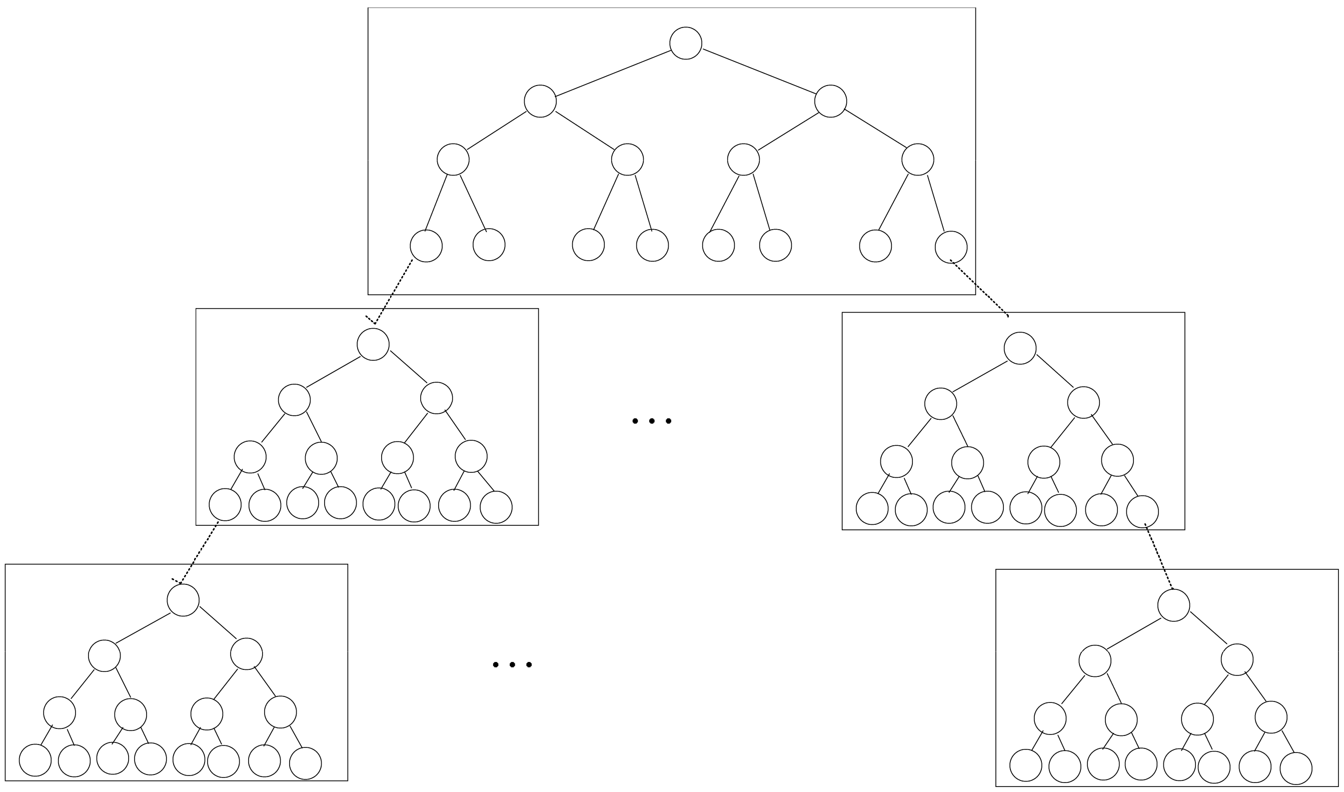

Figure 3: The tree $\mathbf{M}$ for $b = 2$, $p = 15$, $h_{\mathbf{M}} = 3$, $M = 8^3 = 512$.

Proof. For a mapping τ the term *τ-distance* of a node $v \in T$ denotes $PF_\tau(v)$.

Given a solution $A_1, \ldots, A_m$, to the 3-PARTITION problem, construct an optimal compact packing of T as follows. First pack $\mathbf{M}$ optimally with cost $C_\mathbf{M}$. The τ-distance of the leaves $\mathbf{M}$ is thus h. Then, put each copy of $\mathbf{P}$ hung on a leaf of $\mathbf{M}$ in a separate page. This adds to the cost $(2M-3m)p(h+1)$. Finally, for every $A_i = \{a_{i_1}, a_{i_2}, a_{i_3}\}$ assign the three subtrees $\mathbf{S}(a_{i_1})$, $\mathbf{S}(a_{i_2})$ and $\mathbf{S}(a_{i_3})$ that were hanged on the leaves of $\mathbf{M}$ to the same page. The packing thus obtained is compact. Its cost is C_T since there are altogether $2(M-m)$ nodes in the $3m$ auxiliary trees $\mathbf{S}(a)$ and the $2M-3m$ copies of $\mathbf{P}$. The τ-distance of all nodes in these trees is $h+1$.

For the other direction, note that in any packing, there are at most p nodes of T at τ-distance 1, at most $(p+1)p$ nodes at τ-distance 2, and at most $(p+1)^{i-1}p$ nodes at τ-distance i. In total, at most $2M-1$ nodes of T are at τ-distance h or less. The remaining $2(M-m)p$ nodes must be at a greater τ-distance. It follows from this observation that C_T is the minimum cost of any packing, and that in an optimal packing there will be no nodes at τ-distance greater than $h+1$.

Consider an optimal packing τ. Had any leaf v of $\mathbf{M}$ been at a τ-distance greater than h, then, since v's right subtree contains p nodes, this subtree contains a node v' which is at τ-distance $h+2$ or more. Thus, in an optimal packing, all leaves of $\mathbf{M}$ are at τ-distance at most h. (From the above counting argument, they are all at τ-distance h.)

Since the packing of $\mathbf{M}$ into pages at τ-distance $\leq h$ is compact, the children of all leaves of $\mathbf{M}$ cannot belong to the same page as their parent. Consequently, all the nodes of $\mathbf{S}(a)$ and copies of $\mathbf{P}$ are at τ-distance at least $h+1$.

In an optimal compact packing, if any $\mathbf{S}(a)$ is split between two pages, there would be a node in $\mathbf{S}(a)$ at τ-distance greater than $h+1$. Thus, in an optimal compact packing the $\mathbf{S}(a)$'s are grouped into pages—three per page. This grouping is a solution to the 3-PARTITION problem. ∎

This completes the proof of Theorem 4.

6 Approximating an Optimal Compact Packing

The NP-Completeness result is due to our requirement that the packing be compact and that the cost be optimal. Here we show that if the cost is allowed to increase by an additive factor, then a compact packing can be found efficiently.

We start with the Dynamic Programming solution. Let $P_1, \ldots, P_q$ be its non-full pages. Because of the optimality of the solution, each P_i is a leaf-page (the children of all the nodes of P_i are also mapped to the same page). Since the Dynamic Programming solution did not attempt to save space, P_i consists of a single tree (call it T_i).

We now apply an iterative process to repack these pages: In the i'th step, let P_i be the first page that is neither full nor empty and P_j the next such page. (We are done if no such P_j exists.)

If $|P_j| + |P_i| \leq p$, we move all of P_j to P_i and free P_j. This change does not affect the cost of the packing. Otherwise, we split T_j (the tree residing in P_j) into two pieces: one of size $p - |P_i|$ and the other consisting of the remaining nodes. The split is done so that the piece moved to P_i is a connected graph containing T_j's root. (This can be accomplished by conducting a breadth-first search or a depth-first search from T_j's root.) We move the first piece to P_i, thus filling it, and P_j remains with $|P_j| - (p - |P_i|)$ nodes. In this case, after the move, P_i is full and P_j is the next page to be filled. In this manner, no tree of P_j will be split again. The important point to notice is that each tree is split at most once and hence each path is split at most once. Therefore, the number of page faults of each path from the root to a node increased by at most 1. Hence, the increase in cost is less than $\sum_{v \in \bigcup T_j} w(v) < w(T)$. If the $w(v)$ are probabilities, then $w(T) = 1$ and the cost of our packing is at most 1 greater than that of the optimal tight solution.

If all the nodes have the same weight, say $1/n$, we can improve on this by observing that there are several ways to split s nodes from the tree T_j. If $s > |T_j|/2$, we split the tree so that the larger piece contains the root, and we move the root to P_i. Otherwise, we keep the root in P_j and move the other piece to P_i. Hence, at least half the vertices of T_j are mapped to the same page as the root of T_j, and therefore their cost did not increase. The cost increased for at most half the nodes. Since every path was split at most once, the increase is by 1 for these. This leads to a bound that exceeds the optimal cost by an additive factor of at most 1/2.

This improvement cannot be guaranteed when the weights are not equal. Let T_j consist of two nodes: u_0—the root, and its child u_1 (see Figure 4). Suppose exactly one node of T_j should be moved to P_i. Regardless whether u_0 or u_1 is moved, a page fault is incurred between u_0 and u_1. Therefore, the number of page faults on the path to u_1 increases by 1. Thus the expected path length has increased by $w(u_1) \cdot 1$. When $w(u_1) > 1/2$, the added cost is greater than 1/2.

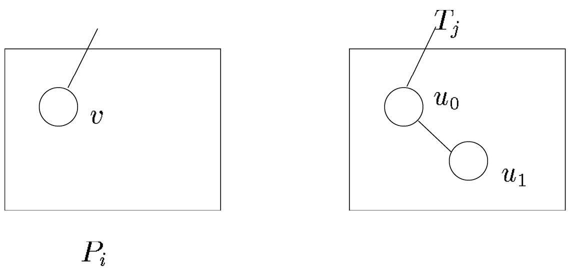

Figure 4: Any repacking of T_j increases the cost by at $w(u_1)$.

7 Packing Dynamically Changing Trees

The schemes we described so far for packing trees assumed that the tree T was given and does not change. Now we consider dynamic trees, i.e., trees in which nodes can be inserted and deleted. Often algorithms maintain "well-balanced" trees—trees whose height is $O(\log n)$. We assume that the reader has some familiarity with B-trees and weight balanced trees (see [3] for a textbook presentation). These schemes perform local operations on trees such as rotations and node splitting (replacing a node v by two siblings v_1, v_2, and attaching each of v's children to either v_1 or v_2). We will examine how an optimal packing is affected by such operations, and show how to maintain a near optimal packing for B-trees and a variant of weight balanced trees.

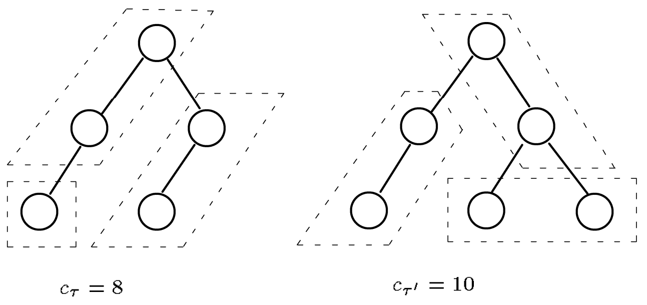

Figure 5: Adding a node completely changes the packing ($w(v) = 1$).

As can be seen from Figure 5, the insertion of a single node may affect the contents of all the pages of an optimal mapping. Therefore, if an efficient update algorithm is desired we can only hope to approximate the optimal packings. More formally, let $o_1, \ldots, o_m$ be a series of inserts, deletions and rotations, T_0 the initial tree, and T_i the tree obtained after applying the operations $o_1, \ldots, o_i$. Our aim is a *dynamic packing scheme* that, given T_i, the tree obtained after applying the operations $o_1, \ldots, o_i$, the packing τ_i which was produced for it, and the next update operation o_{i+1}, will efficiently compute τ_{i+1}, the packing for T_{i+1}. It is required that

$$c_{\tau_i}(T_i) = O\left(c_{\tau_{\text{opt}}}(T_i)\right) \qquad i = 0, \ldots, m \ , \tag{9}$$

where $\tau_{\text{opt}}(T_i)$ is the optimal packing of T_i.

Requirement (9) will follow from a stronger condition

$$c_{\tau_i}(v) = O\left(c_{\tau_{\text{opt}}}(v)\right) \qquad v \in T_i \text{ and } i = 0, \ldots, m. \tag{10}$$

We also wish that the size of $\tau_i(T_i)$ does not exceed that of a compact packing by more than a small factor.

A general solution to this problem still remains to be found. However, we can provide one for two families of trees. The main ideas behind our constructions are sketched below.

B-trees: B-trees are the most common data structure used to store databases. It is a tree for which the degree of every node is between $d/2$ and d (for some predetermined integer d), and all leaves are at the depth (distance from the root in terms of number of edges). If the node size of the B-tree is equal to the page size, then every packing maps each node to a unique page and thus is optimal. Suppose therefore that several nodes can be stored in a page. Let h be the height of the B-tree. Let η be such that a full B-tree of degree d and of height η can be stored in a page. Specifically, we set

$$\eta = \lfloor \log_d(1 + p(d-1)) \rfloor \ .$$

To get a packing we first cut the edges emanating from the nodes at height (distance from the leaves) $\eta, 2\eta, \ldots, \lfloor h/\eta \rfloor \eta$, whereby producing subtrees, each with at most p nodes. We then map each such subtree to a separate page. This is illustrated in Figure 6.

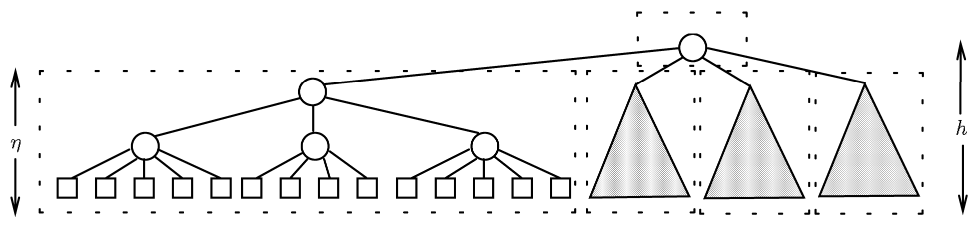

Figure 6: Packing a B-tree of degree 5, $\eta = 3$.

For a leaf v the cost of the packing is $c_\tau(v) = \lceil h/\eta \rceil$. Thus the cost of the packing is at most $n \lceil h/\eta \rceil$, while the cost of an optimal packing is at least $\frac{n}{2} \lceil h/\eta' \rceil$, where

$$\eta' = \left\lfloor \log_{d/2}(1 + p(d/2 - 1)) \right\rfloor \ .$$

Since $\eta' = \theta(\eta)$, $c_\tau(T) = \theta(c_{\text{opt}}(T))$.

Insertions to a B-tree may add nodes only to leaf pages. With our selection of η, the page is large enough to accommodate all insertions, unless the root of the subtree stored in it splits. If this happens, we store the two resultant subtrees in a separate page. If the root of the entire tree is split and η divides h, then the root is moved to a separate page. It follows that the updating of τ requires changing at most two pages. It can be shown that the amortized time for updating the packing is constant. Deletions are carried out similarly within the same complexity.

This dynamic packing scheme might require an excessive number of pages. The space consumption may be reduced by storing several trees in a single memory page. Using a rather standard memory management policy we can guarantee that at least a constant fraction of each page is used. Although more page overflows may occur with this policy, we can show that incurred overhead per operation is, in an amortized sense, constant.

Weight Balanced Binary Trees We follow the scheme suggested by G. Varghese [3, Problem 18-3, p. 376]: Let $1/2 < \alpha < 1$. We say that a binary tree T is α-balanced if for every node $v \in T$

$$|T_{\text{left}(v)}| \leq \alpha |T_v| \text{ and } |T_{\text{right}(v)}| \leq \alpha |T_v| \ . \tag{11}$$

Whenever, as a result of insertions and deletions, condition (11) is violated for a node v but not for any of its ancestors, the entire subtree T_v is reorganized: it is replaced by a full binary tree with $|T_v|$ nodes. This reorganization requires $O(|T_v|)$ time. One can show that the amortized time for insertion/deletion is $O(\log n)$.

To maintain a near optimal packing, every page P_i consists of a subtree of T. Since only leaves are added, if there is no room for a new node in its parent's page, it is allocated to a new page. Whenever T_v is reorganized we also repack T_v using our Dynamic Programming algorithm.

8 Further Research

We have shown how to efficiently pack static search trees, and have given schemes for handling two types of dynamically changing trees. The challenge still lies in finding a general scheme for dynamic trees, which will account for more elaborate balancing operations, such as rotations of AVL trees. Yet, another interesting direction would be to explore the problem of tree packing under a different model of access patterns, e.g., starting at any given node, and moving on the path to another given node.

Trees are not the only data structure that can benefit from efficient packing. It would be interesting to find efficient packing schemes for general graphs, sparse matrices, and data structures for multi-dimensional keys (Oct-trees, grid graphs) under a suitable model for access patterns.

In the caching model we presented, every disk-page can be mapped to any main memory page. Such mappings are called *distributed mapping* which capture the behavior of paging between the main memory and the disk . As indicated in the introduction, a similar hierarchy occurs within the processor between the fast cache memory resident on the CPU chip and the slower main memory. However, the mapping between main memory and memory cache is more restricted: each memory page can be mapped only to a single predetermined cache page. This restriction, is called *direct mapping* [9]. The packing problem in this context becomes more difficult, since the pages are no longer isomorphic. In particular, in a worst case situation, were all external pages are mapped to the same internal page, then the number of page faults would be exactly the same as the number of pages accessed. On the other hand, it might be beneficial to map the root page to a cache page, such that no other page shares the same cache page. Thus, direct mapping raises an additional question: how to map the nodes of a tree to main memory pages so as to minimize the number of page faults in a batch of several searches. What is required is a solution for the simultaneous packing and mapping problems.

References

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual Symposium on Computing*, pages 305–314, New York, 1987.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw Hill and The MIT Press, 1990.

[4] B. Farizon. Local and global memory for the implementation of ray tracing on a parallel machine. Master's thesis, Computer Science Dept., Technion – Israel Institute of Technology, 1995.

[5] J. D. Foley and A. V. Dam. *Fundamental of Interactive Computer Graphics.* The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1984.

[6] S. Gal, Y. Hollander, and A. Itai. Optimal mappings in a direct mapped cache environment. *Math. Programming B*, 63:371–387, 1994.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[8] C. F. Goldfarb. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[9] Y. Hollander. *Data structures for cache memory.* PhD thesis, Technion, 1996.

[10] M. S. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS IV)*, 1991.

[11] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.