David Clark. Compact Pat Trees.
PhD thesis, U. Waterloo, 1996

4.1 Partitioned Compact PAT Trees

In order to control the number of accesses to secondary storage required during CPT operations, we *partition* the tree into connected components each of which fits in a disk block. We call each component a *page* because of the similarity of this problem to the problem of efficiently laying out a tree or other data structure in a paged virtual memory system[21]. If the disk block size is such that it can hold two internal nodes then the PAT tree of Figure 1.6 could be partitioned as shown in Figure 4.1. In this case we need to perform three accesses to secondary

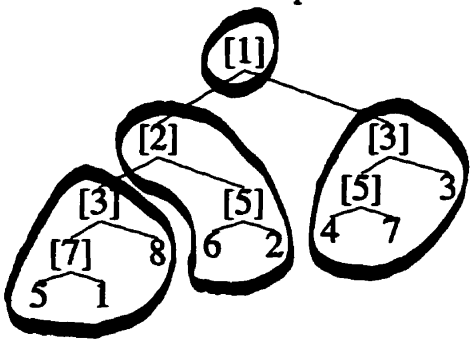

Figure 4.1: Tree Partition

storage to reach leaf 1, 5 or 8 from the root. The alternative partitioning in Figure 4.2 can reach any leaf in two accesses and so might be preferred.

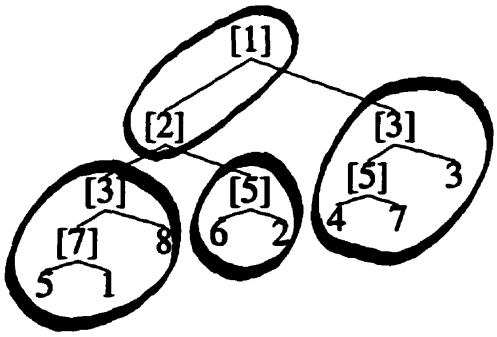

Figure 4.2: Alternative Tree Partition

Two possible criteria for choosing one partitioning over others are:

- the number of pages accessed when traversing from the root to a leaf, averaged over all the leaves, and
- the maximum number of pages accessed when traversing from the root to any leaf.

We call page partitionings that minimize these measures *average case optimal* and *worst case optimal* respectively. Let c_i be the number of pages accessed to reach the i'th leaf (under some ordering of the leaves). Then these partitionings minimize $\sum_i c_i$ and $\max c_i$ respectively. Implicit in these measures is the assumption that we consider all leaves equally important. Lukes[32] and Gil and Itai[21] consider more general cases where nodes and edges can have weights associated with them.

The partitionings considered here are restricted such that each page holds a connected portion of the tree. Gil and Itai use the term *convex* to describe such partitionings and show that loosening this restriction does not allow for better average case partitioning[21]. Because of this restriction, each page will itself be a tree and can be stored using the CPT structure from Chapter 2. The only change required to the CPT structure for storing the pages is that the leaf data may now point to either a suffix in the text or a sub-tree page so an extra bit is required to distinguish these two cases. We let the value p denote the number of internal nodes in the largest sub-tree we can place in a block. Using the representation from Chapter 3, $p \approx \frac{P - \lg n}{\lg n + \lg\lg\lg n + 4}$. The restriction to connected sub-trees allows us to refer to the root of the sub-tree in a page as the root of the page. In addition we will refer to the page containing the sibling node of a page's root as the page's sibling. Note that in some cases a page's root and its sibling may be the same page (consider the rightmost internal node of Figure 4.2).

Lukes[32] presents a dynamic programming method for finding an average case optimal partitioning in $O(np^2)$ time. A related method for finding a worst case optimal partitioning in $O(np)$ time is reported in Carlisle *et al.*[9]. Unfortunately both of these methods require np words of storage to compute the partitioning and so are not practical for trees of the size we are considering. Gil and Itai[21] develop a similar dynamic programming method for the average case that operates in much less memory. However, their algorithm performs multiple passes over the tree and so is unlikely to be efficient enough for our purposes. Additionally, these dynamic programming methods do not adapt well to the dynamic trees needed in the next chapter. Carlisle *et al.*[9] also discuss a top down greedy heuristic that is conceptually simple and works well on some classes of trees but can require $\Theta(\log n)$ extra page accesses on average to reach any leaf.

In the remainder of this chapter we present a new bottom up greedy algorithm for constructing a worst case optimal partitioning of a binary tree and demonstrate its use on the CPT.

4.2 Partitioning Algorithm

Define the *page height* of a node in a partitioned tree as the maximum number of pages that need to be read when traversing from the node to any leaf in its sub-tree and the page height of a page as the page height of its root. In each case we include the current page in the page height count. For any given assignment of nodes to pages, also define the *local page size* of a node as the number of descendents of that node that are on the same page as the node, plus one for the original node. The page height and local page size of a node may be defined for a partial partitioning provided the node and all of its descendents have been placed on pages.

We present a partitioning algorithm that starts by assigning each leaf its own page and a page height of one. Working upward, we apply the rule in Figure 4.3 at each node.

```
if both children have the same page height
    if the sum of the local page sizes of the children is less than p,
        merge the pages of the children and add the node
        set the page height of the node to that of the children
    else
        close off the pages of the children
        create a new page for the current node
        set the page height of the node to that of the children plus one
else
    close off the page of the child with the lesser height
    if the local page size of the remaining child is less than p,
        add the node to the child's page
        set the page height of the node to match the child
    else
        close off the page of the remaining child
        create a new page for the node
        set the page height of the node to that of the child plus one
```

Figure 4.3: Tree Partitioning Rules

Theorem 4.1 *A worst case optimal convex partitioning of a binary tree can be computed in linear time, irrespective of the page size.*

Proof: Using induction on the tree height, we show that the rule in Figure 4.3 produces a worst case optimal partitioning of the tree such that no other optimal partitioning has a smaller root page and moreover that this holds for each sub-tree. The basis case, $k = 1$, consists of a tree with a single node and so is

trivial. Assume the statement for $1..k-1$ and then consider the root of a tree of height k. There are several possible cases:

1. The root has only one child. In which case either the root fits on the topmost page of the child or it does not.

   - Root fits (the local page size of the child is less than p): Place the root in the topmost page. Any partitioning of smaller page height or top most page must contain a partitioning for the child of larger page height that violates the induction hypothesis for $k-1$.
   - Root does not fit (the local page size of the child is p): Create a new page for the root. Clearly there cannot be a partitioning with fewer than one vertex in the topmost page so any violation must be on the page height constraint. The existence of partitioning of lesser page height would imply a partitioning at height $k-1$ with room for the new root but the partitioning of the height $k-1$ sub-tree was completely full and also had smallest topmost page amongst all optimal partitioning so this situation cannot occur.

2. Next consider the case where the root has two children that differ in page height. By the rules above, the child of lesser page height is closed off. The root is placed in the topmost page of the other child if at all possible, and on a new page if not. There are two cases that are argued exactly as case 1 above. Case 1 is actually a specialization of case 2 so this is not surprising.

3. Finally assume the root has two children each of equal page height. Under the rules above the new partitioning is formed by merging the topmost pages of the two children and adding the root if the combined page is not too large. If the combined page is too large, the topmost pages of both children are closed and a new page is started for the root.

- Root fits (sum of children's local page sizes is less than p): the page height of the new partitioning is the same as that of the children so the existence of a partitioning of lesser page height would violate the induction hypothesis for $k-1$. If there is a partitioning of the same page height but smaller topmost page then it must contain a height $k-1$ partitioning for one of the two children that violates the induction hypothesis.
- Root does not fit (sum of children's local page sizes is at least p): Again the topmost page has size one so no other partitioning of the same page height can have a smaller topmost page. If there is a partitioning of smaller page height then as before, it must contain a partitioning for one of the two children that violates the induction hypothesis.

The "moreover" part holds because we never go back and invalidate the optimality of the partitioning of sub-trees.

The rule in Figure 4.3 performs a constant amount of work at each node and so can be applied in linear time. *QED*

Based on Theorem 4.1 we will refer to a partitioning resulting from the rules in Figure 4.3 as the "optimal bottom up partitioning" of a tree. The optimal bottom up partitioning is optimal in the sense that it minimizes the number of pages accessed in the worst case root-leaf traversal. However, it can produce a large number of very small pages. This problem results from the automatic closing off of a page if its sibling has a greater page height. Because we do not worry about aligning pages on block boundaries in the static text case, these small pages do not cause serious problems. However, it is still worthwhile running a post-processing pass that merges small pages into their parent whenever possible because each page has some small amount of storage overhead. The results reported later in this chapter include the use of such a pass. We will have to return to this problem

in the next chapter where small pages can cause storage management problems.

In order to judge the overall performance of data structures using the optimal bottom up partitioning, we want to bound the page height in terms of the number of nodes and the tree height, H. Before proving the bound, two simpler results are needed.

Lemma 4.1 *In an optimal bottom up partitioning, each sub-tree in a tree encoded in a page of page height $k > 1$ contains at least one node having children with page height $k - 1$.*

Proof: If all its children have page height $k - 2$ or lower, split off the sub-tree into its own page and obtain a partitioning with a smaller root node. The difference in page heights allows us to make this change without increasing the page height of the root. *QED*

Lemma 4.1 allows the simple observation that, under an optimal bottom up partitioning, all nodes in a page have the same page height.

Lemma 4.2 *While on a root-leaf path of pages in an optimal bottom up partitioning, the leaves within a page height k page where $k > 1$ either have one child page with page height $k - 1$ containing p nodes or two child pages with page height $k - 1$ containing a total of at least p nodes.*

Proof: Each leaf node is a sub-tree so by Lemma 4.1, it contains at least one page height $k - 1$ child. If neither of the conditions are met, then the parent would have been moved in with either or both of the children and a partitioning with a smaller root node obtained for that sub-tree. *QED*

Theorem 4.2 *Let $0 \leq t < 1$ be an arbitrary constant. The page height of the worst case optimal partitioning of a tree is bounded above by*

$$1 + \left\lceil \frac{H}{p^t} \right\rceil + \left\lceil \frac{1}{1-t} \log_p n \right\rceil \tag{4.1}$$

where H is the height of the tree and n is the number of nodes in the tree.

Proof: Our proof is based on the optimal bottom up partitioning. Given such a partitioning, we construct a sequence of pages on a deepest path, in the page sense, such that at each stage we either divide the number of nodes in the current sub-tree by $\lceil p^{1-t} \rceil$ or reduce the height (in the node sense) of the sub-tree by $\lceil p^t \rceil$. At each point in the construction we consider either a single page or a pair of sibling pages. Start the construction at the root page and select any node in the page that has children at page height $k-1$ and consider its page height $k-1$ children. By Lemma 4.2, we know that there are at least p nodes in these child pages. Because there are p nodes, one of the following two conditions must be met:

1. there are at least $\lceil p^{1-t} \rceil$ children pointing to child pages with page height $k-2$, in which case we select the child whose page height $k-2$ children have the smallest portion of the entire sub-tree, or
2. there is at least one node pointing to pages at page height $k-2$ such that the length of the path from the root of the page to the node has length at least $\lceil p^t \rceil$. Select that node's page height $k-2$ children for the next step.

If neither of these conditions are met, then we could not be dealing with p nodes. Case one can only occur $\left\lceil \log_{\lceil p^{1-t} \rceil} n \right\rceil$ times and case two can only occur $\left\lceil \frac{H}{\lceil p^t \rceil} \right\rceil$ times. Add one for the root page, remove the inner ceilings, and simplify the log to obtain an upper bound on the length of the path constructed. Because this path is a deepest path in the page sense, the bound also applies to the page height of the tree. *QED*

Two corollaries can be obtained by selecting specific values of t. Choosing $t = \frac{1}{2}$, we obtain a bound of the form $1 + \left\lceil \frac{H}{\sqrt{p}} \right\rceil + \lceil 2 \log_p n \rceil$ which is interesting for its simplicity. [redacted]