

A Linear Time Algorithm for Optimal Tree Sibling Partitioning and Approximation Algorithms in Natix

Carl-Christian Kanne

Guido Moerkotte

Department of Mathematics and Computer Science
University of Mannheim
ccjmoer@db.informatik.uni-mannheim.de

ABSTRACT

Document insertion into a native XML Data Store (XDS) requires to partition the document tree into a number of storage units with limited capacity, such as records on disk pages. As intra partition navigation is much faster than navigation between partitions, minimizing the number of partitions has a beneficial effect on query performance.

We present a linear time algorithm to optimally partition an ordered, labeled, weighted tree such that each partition does not exceed a fixed weight limit. Whereas traditionally tree partitioning algorithms only allow child nodes to share a partition with their parent node (i.e. a partition corresponds to a subtree), our algorithm also considers partitions containing several subtrees as long as their roots are adjacent siblings. We call this *sibling partitioning*.

Based on our study of the optimal algorithm, we further introduce two novel, near-optimal heuristics. They are easier to implement, do not need to hold the whole document instance in memory, and require much less runtime than the optimal algorithm.

Finally, we provide an experimental study comparing our novel and existing algorithms. One important finding is that compared to partitioning that exclusively considers parent-child partitions, including sibling partitioning as well can decrease the total number of partitions by more than 90%, and improve query performance by more than a factor of two.

1. INTRODUCTION

We consider the problem of tree partitioning from the perspective of native XML data stores (XDSs). In particular, we are concerned with the quality of the storage representation of XML documents in systems that natively store the ordered, labeled tree representation of the XML documents, and use navigational primitives to access this representation during query processing. Any storage engine designed to store trees that require more space than a single unit of secondary storage must have a tree partitioning algorithm. Tree partitioning decomposes the logical document tree into partitions

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

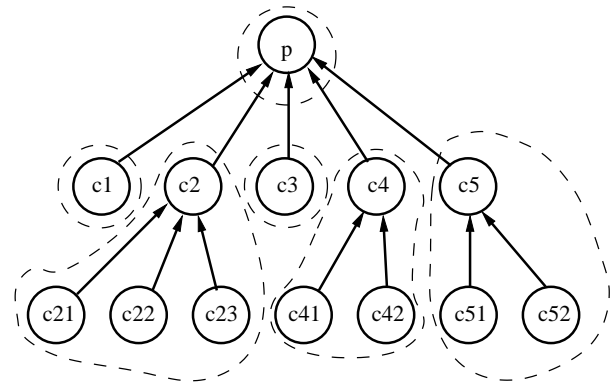


Figure 1: Partitioning with parent-child edges only

smaller than a weight limit, which corresponds to the storage unit's capacity, e.g. the disk page size. The tree partitioning algorithms may be ad-hoc in some systems which arbitrarily place nodes wherever there is sufficient space. In general, however, it is a good idea to carefully design partitioning algorithms for XDSs because (1) the number and structure of partitions is an important determinant of query performance, since crossing storage units during query processing is expensive, and (2) performance of the partitioning algorithm itself affects overall system performance because document insertion is a frequent operation.

An important feature of the XML data model is order, and this must be taken into account when designing partitioning algorithms. The storage engine of an XDS not only has to store parent-child edges of a tree, but must also capture the sibling order. Storage engines for native XDSs such as IBM's System RX/DB2 Viper [2] and the Natix system [6] provide such ordered tree storage. They go even further and provide optimized storage for consecutive siblings that share a storage unit, even if their parent is located on a separate storage unit. Without such an optimization, access to nodes with a large number of children would suffer from bad performance. Consider the tree shown in Fig. 1. Assume that the root node p that does not fit on a storage unit together with any of its children's subtrees. If the storage format does not allow to put consecutive siblings into a storage unit that does not contain their parent, the resulting partitioning looks as indicated by the dashed lines in Fig. 1. In this case, each child is stored separately, and every partition corresponds to a *single* subtree. Query evaluation with an XML query language such as XPath [1] and

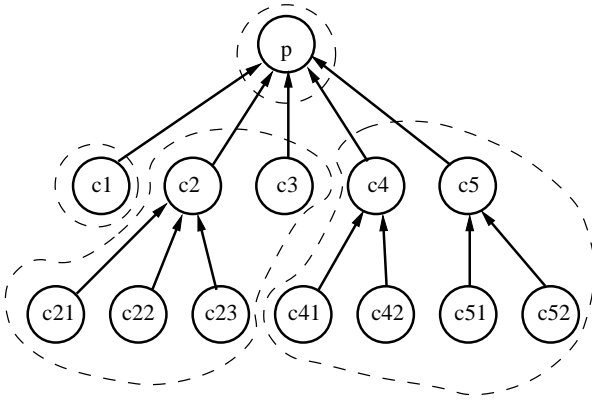


Figure 2: Partitioning with parent-child and sibling edges

XQuery [3] is expensive here. In case of an in-order traversal of all children or descendants of p , such as the evaluation of the `child` or `descendant` axis starting for context node p would access a different storage unit for every child of p , i.e. 5 storage units in total. If siblings can share a storage unit even if their parent is in a different storage unit, then we have a situation as shown in Fig. 2. Here, several subtrees may share a partition, as long as their roots are siblings. We call this partitioning style *sibling partitioning*. It results in fewer expensive crossings of storage unit borders (in our example, there are three), which in turn improves the query performance. To keep the number of such crossings as low as possible, a tree partitioning algorithm for XDS should create sibling partitionings and minimize the total number of partitions.

Our primary motivation for studying the tree sibling partitioning problem is our experience with the storage engine of our native XML data store Natix [6]. Natix uses a storage format where the storage units are physical records, each of which contains a fragment of the document tree whose nodes are connected by parent-child or sibling edges. Natix has two algorithms to determine which nodes share a physical record [9, 10]. The node-at-a-time algorithm [9] maintains the clustered XML storage format on incremental updates. Insertions of whole documents are handled by the bulkload component, whose design and implementation is described in [10]. Its standard partitioning algorithm for document import is a simple heuristics.

In practice, for several cases we observed peculiar partitioning decisions by this simple algorithm that lead to unacceptable query performance. Ad-hoc attempts to refine the heuristics were not very robust, i.e. always vulnerable to new pathological cases (some of them are presented throughout this paper). To be able to judge the quality of the various algorithms and to get an insight how to construct a more robust one, we wanted to know the theoretical optimum, i.e. a partitioning with a minimal number of partitions. However, determining the minimal number of partitions for a typical document is not an easy task: The number of potential sibling partitionings is exponential with respect to the number of nodes, so a brute force algorithm for determining the optimum is not feasible.

Over the last decades, a number of algorithms for tree partitioning has been developed, including [4, 5, 12, 13, 15, 17]. Several of them were specifically designed for the then-current storage engines. Tree partitioning algorithms have been studied in the context of hierarchical DBMS [13, 15], object-oriented DBMSs [17] and, recently, XDSs [4, 5]. Unfortunately, none of the algorithms considers sibling order or allows sibling subtrees to share a partition if their parent is in a different partition.

The three main contributions of this paper are:

1. We present a linear time algorithm for optimal tree sibling partitioning.
2. We present two novel, near-optimal heuristics that have much better runtime than the optimal algorithm.
3. We provide experimental results, comparing our algorithms and several existing heuristics with respect to the number of generated partitions and the query performance on the produced partitioning.

The paper is structured as follows. Sec. 2 formalizes the problem. Sec. 3 develops a sequence of algorithms for tree partitioning problems which culminate in a complete optimal algorithm for tree sibling partitioning. We discuss the problem substructure in detail, supported by formal proofs where necessary. The first of these algorithms is limited to flat trees and uses dynamic programming to partition the sequence of children. We proceed with an algorithm that applies the flat tree algorithm in a bottom-up manner to deep trees, using optimal solutions for subtrees to obtain a global solution. Unfortunately, this does not always yield an optimal solution. In some situations, a locally suboptimal tree partitioning is required for the global optimum. We present a method to generate the required local solutions. In a final step, we show how the proper local solutions can be chosen to achieve the global optimum. Although the algorithms get progressively more complex, all of them have a runtime proportional to the number of document nodes in the worst case. Sec. 4 explains why the optimal algorithm is not always a wise choice for document import into real XDSs, and presents a number of both existing and novel heuristics that are better suited for real systems. Sec. 5 assesses other existing algorithms for tree partitioning and XML document clustering. Sec. 6 evaluates our three novel and four existing sibling partitioning algorithms. Sec. 7 concludes the paper.

2. PROBLEM STATEMENT

2.1 Terms and Definitions

Let $T = (V, t, p, \triangleleft, w)$ be a rooted, ordered, and weighted tree with nodes V , a root t , a parent function p , a transitive sibling ordering \triangleleft , and a weight function w . p maps each nonroot node to its parent and the root to NIL, and w maps each node to a positive integer weight. In the following, the term tree always denotes a rooted, ordered and weighted tree.

Fig. 3 shows an example tree $T = (\{a, b, c, d, e, f, g, h\}, a, p, \triangleleft, w)$, which we will use to illustrate our definitions below. In the figure, the nodes are represented as ovals with identifiers, the parent function p is represented using solid child-parent arrows, the sibling ordering is represented by the \triangleleft symbols (with the transitive relationships such as $b \triangleleft g$ omitted), and the node weights w are the numbers in the ovals.

Given a tree $T = (V, t, p, \triangleleft, w)$, we denote the subtree induced by a node $v \in V$ with T_v . The *subtree weight* $W_T(v)$ is the sum of the weights of all nodes in T_v . In our example, the tree T_c consists of the nodes c , d , and e . c 's subtree weight $W_T(c)$ is 5.

A *sibling interval* $(l, r)_T$ of T is a set of consecutive siblings determined by a first sibling l and a last sibling r with $l \triangleleft r$, such that $(l, r)_T := \{x \mid x = r \vee x = l \vee l \triangleleft x \triangleleft r\}$. A *tree sibling partitioning* P of T is a set of disjoint sibling intervals. The subtree weight of a sibling interval is $W_T(l, r) := \sum_{x \in (l, r)_T} W_T(x)$. The weight of a set S of sibling intervals $W_T(S)$ is the sum of the weights of the

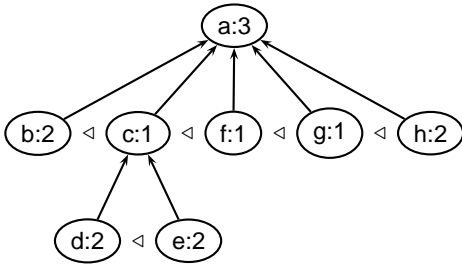


Figure 3: Example tree

contained intervals. In our example, the interval $(b,f)_T$ consists of the nodes b , c , and f , and has a subtree weight of 8.

Given a tree T and a tree sibling partitioning P as above, the *partition forest* F_T^P of T with respect to P is the set of trees that results from T when cutting the parent edges from those nodes that belong to a sibling interval in P . This is equivalent to having a parent function p^P such that for all $(l,r)_T \in P$, $\forall v \in (l,r)_T p^P(v) := \text{NIL}$. Hence, in F_T^P , each node that is contained in an interval in P becomes the root of a tree. The *partition* defined by an interval $(l,r)_T$ is the set of all trees from F_T^P whose root is in $(l,r)_T$. In our example, the partition defined by $(b,f)_T$ is $\{T_b, T_c, T_f\}$.

The *partition weight* $W_T^P(v)$ of a node v is its subtree weight in F_T^P . Analogously, the partition weight of a sibling interval $W_T^P(l,r)$ is the sum of all the partition weights of its nodes, and the partition weight of a set of sibling intervals is the sum of the partition weights of its intervals. The *root weight* of a partitioning is the partition weight of the root node, $W_T^P(t)$. In our example tree, consider the partitioning $P := \{(b,f)_T\}$. The root weight of P is 6, because only the nodes a , g , and h remain in the tree of the root a after the parent edges of b , c , and f have been removed.

Given T and a positive integer K , a tree sibling partitioning P of T is called *feasible* iff $(t,t)_T \in P$ and $\forall (l,r)_T \in P W_T^P(l,r) \leq K$. A feasible partitioning of our example tree and $K = 5$ is $P := \{(a,a)_T, (b,b)_T, (c,c)_T, (f,g)_T\}$. Here, h is in the same partition as the root, and the root weight is 5.

A tree sibling partitioning is called *minimal* iff it is feasible and has the smallest possible cardinality of all feasible partitionings. A tree sibling partitioning P is called *lean* iff its root weight is minimal among all partitionings with the same cardinality. A tree sibling partitioning is called *optimal* iff it is both minimal and lean. In our example, $R := \{(a,a)_T, (c,c)_T, (f,h)_T\}$ is a minimal partitioning ($K = 5$) with cardinality of 3. b is in the same partition as the root, so R has a root weight of 5. However, R is not lean. There is a partitioning with the same cardinality and a smaller root weight: In $\mathcal{P} := \{(a,a)_T, (c,h)_T, (d,e)_T\}$, the root weight is 3. \mathcal{P} is optimal. We will often denote optimal tree sibling partitionings with calligraphic letters such as \mathcal{P} or \mathcal{D} .

2.2 The Tree Sibling Partitioning Problem

Given these terms, the problem we want to solve is formally stated as follows:

Tree Sibling Partitioning: Given a tree T and a weight limit K , determine a minimal tree sibling partitioning.

To solve this problem, we develop algorithms that find partitionings with a stronger property, namely optimality. According to our definition, this means that the partitionings must have minimal root weight among all minimal partitionings. We will see below that the reason for this lies in our recursive, bottom-up approach: While minimality is all we need for the overall solution, the subproblems

we solve must also be lean to guarantee minimality on higher levels.

3. OPTIMAL TREE SIBLING PARTITIONING

The number of feasible tree sibling partitionings for a given tree with n nodes is very large, even if a fixed weight limit K is provided. For every parent node, we have to decide which subset of children to place in the same partition as the parent. For the remaining children, we must decide how to combine the siblings into partitions. It is not at all obvious how to find a minimal partitioning in time proportional to n , given a fixed partition weight limit K . In fact, we shall see that even simplified versions of the problem are not obviously solvable in linear time.

We pursue an incremental strategy. We approach tree sibling partitioning formally, proving a sequence of properties that enable us to develop progressively more advanced algorithms.

We start out by showing that a bottom-up approach is viable because we can combine partitionings for subtrees to obtain a global solution. As a second step, we present a dynamic programming algorithm that can partition flat trees (i.e. trees where all nodes but the root node are leaves) in $O(nK^2)$ time.

Unfortunately, we will see that the bottom-up application of this algorithm to a deep tree does not necessarily yield an optimal solution: Sometimes we have to choose a suboptimal solution in the lower levels of the tree to avoid extra partitions on the next higher level. However, we can show that at each step, we only need to choose between an optimal and a nearly optimal solution, for a rather simple definition of "nearly optimal". We also show how to incorporate this choice into our dynamic programming algorithm, finally arriving at an $O(nK^3)$ algorithm for optimal tree sibling partitioning.

3.1 Bottom-Up Tree Partitioning

Our algorithms are based on the assumption that in order to determine a globally optimal partitioning, we can select a node v from the tree and determine a partitioning for the subtree induced by that node. Then we can recursively determine a global partitioning for the remainder of the tree and combine the two solutions to obtain the global solution. We will now formalize this basic assumption.

Recall that we consider a solution optimal if it is not only minimal, but also lean. The reason for this is explained below.

In the following lemma, we do *not* assume that the local subtree partitioning for a subtree T_v , called S , is locally optimal. We just assume that we know for some reason that S is part of some global solution, and show how to get a global solution based on S . We do this by collapsing T_v from the original tree into a single node v with a weight that represents the whole collapsed subtree. We recursively determine an optimal solution $\tilde{\mathcal{P}}$ for this new tree \tilde{T} , and merge this result with S to obtain an optimal solution \mathcal{P}' .

LEMMA 1. *Let $T = (V, t, p, \triangleleft, w)$ be a tree. Let $v \in V$ be a node from T . Let V_v be the nodes of T_v . Let S be a feasible tree sibling partitioning of T_v such that there exists some optimal tree sibling partitioning \mathcal{P} of T that contains S and has no other intervals among the descendants of v , i.e. with $S - \{(v, v)_T\} = \{(l, r)_T \in \mathcal{P} \mid (l, r)_T \subseteq T_v\}$.*

Further, let $\tilde{T} = (V - V_v \cup \{v\}, t, \tilde{p}, \tilde{\triangleleft}, \tilde{w})$ be the tree T with the descendants of v removed, such that \tilde{p} , $\tilde{\triangleleft}$ and \tilde{w} are the functions from T restricted to \tilde{T} , with the exception of a new weight for v : $\tilde{w}(v) := W_T^S(v)$. Let $\tilde{\mathcal{P}}$ be an optimal tree sibling partitioning of \tilde{T} .

Then $\mathcal{P}' := \widetilde{\mathcal{P}} \cup (S - \{(v, v)_T\})$ is an optimal tree sibling partitioning of T .

We omit the proofs of our lemmas due to space constraints. They are contained in the extended version of this paper [11].

Lemma 1 suggests an algorithm that traverses the tree in a bottom-up manner. For each non-leaf node v , determine a partitioning S for T_v that is part of a global solution (we will explain how to do this in the remaining section). We then remove the sibling intervals in S (except $(v, v)_T$), and replace the whole subtree T_v by a single node whose weight is equal to the total weight of all the nodes in T_v that are not part of an interval in S . Then we proceed with the next node.

This approach reduces our original problem of finding partitionings for arbitrary trees to the simpler problem of finding partitionings only for flat trees. Flat trees are trees in which all nodes but the root node are leaves. Our bottom-up approach guarantees that, once we reach an inner node, all deeply nested subtrees below that node have been pruned and only a flat tree remains.

The bottom-up traversal is also the reason why we require the local solutions to be lean in addition to be minimal: By cutting away as much of the tree weight as possible, we generate a simpler subproblem in the next higher level of the tree. Of course, we only do so if it does not introduce additional sibling intervals, because the ultimate goal is to find a minimal partitioning.

However, we have not yet specified the subroutine to compute a suitable partitioning for flat trees. We turn to this problem in the next section.

3.2 Flat Tree Partitioning

Before looking at an optimized way to determine an optimal tree sibling partitioning for flat trees, we want to verify that a simple brute-force algorithm is not a viable solution. Let us look at the number of feasible tree sibling partitionings in a flat tree. Assume a flat tree with n leaf nodes in which all the nodes have weight 1. In this case, we can put up to $K - 1$ leaves of the n leaves in the same partition as the root. There are $\binom{n}{K-1}$ possible ways to do this. Hence, a lower bound for the number of feasible root partitions is $\Omega(n^{K-1})$. This estimate does not yet include the different possibilities to group the remaining sibling nodes into intervals. Hence, it is reasonable to conclude that a brute force algorithm is impractical for typical values¹ of K .

We approach the problem by showing that an optimal solution for a tree with n leaf nodes contains an optimal solution for a tree with less than n leaves. Then, we use this knowledge to develop a dynamic programming algorithm that finds the optimal solution in $O(nK^2)$ time. Finally, we discuss the optimization potential of the algorithm.

3.2.1 Optimal Substructure for Flat Trees

Consider the options we have for a leaf in the solution: either the child is put into the same partition with the root, or it belongs to an interval of the result partitioning. Together with the fact that there is only a limited number of feasible intervals to which the child can belong, this forms a useful problem substructure for dynamic programming.

The following lemma states that we can find an optimal tree sibling partitioning for a flat tree with n leaf nodes by choosing the best candidate solution from one of the following two subproblems: Either (1) the solution is the same as an optimal tree sibling partitioning for a similar tree with $n - 1$ nodes, which represents the

¹Keep in mind that K is the size limit for a storage unit, and typical disk page sizes are thousands of bytes.

original tree with the last child put into the root partition, or (2) the solution can be constructed by adding a single interval to an optimal partitioning for a smaller tree.

LEMMA 2. Let $T = (\{t, c_1, \dots, c_n\}, t, p, \triangleleft, w)$ be a tree in which all nodes but the root node are leaves, i.e. $p(c_i) = t$. \triangleleft orders the c_i according to their index value i . The tree $T_j^s = (\{t, c_1, \dots, c_j\}, t, p, \triangleleft, w_j^s)$ is the tree T with all children $\{c_i | i > j\}$ removed and a different weight s for the root node t , with p and \triangleleft regarded as restricted to $\{c_1, \dots, c_j\}$, and $w_j^s(t) := s$, and for $v \in \{c_1, \dots, c_j\}$, $w_j^s(c_i) := w(c_i)$. Let D_j^s be the set of optimal tree sibling partitionings for T_j^s .

Then, for $j = 0$ and any $s \leq K$ holds $D_0^s = \{(t, t)_T\}$.

For each j with $1 \leq j \leq n$, at least one of the following statements holds:

1. $D_{j-1}^{s'} \subseteq D_j^s$ with $s' := s + w(c_j)$.
2. For some m with $0 \leq m < j \wedge m < K$:
 $\forall M \in D_{j-m-1}^s (M \cup \{(c_{j-m}, c_j)_T\}) \in D_j^s$.

Given the notation used above, the problem that has to be solved by our algorithm can be stated like this: Find an arbitrary element of $D_n^{w(t)}$.

Since such an arbitrary element of $D_n^{w(t)}$ can be computed from optimal sibling partitionings D_j^s for smaller trees ($j < n$), the problem is susceptible to a dynamic programming approach, as we show below.

3.2.2 Algorithm FDW for Flat Trees

Our algorithm FDW (Flat trees, Dynamic programming for tree Width) employs dynamic programming by starting out with a tree that only contains the root node. We then successively add all leaf nodes in left-to-right order and iterate over all potential weights of the root node. For each such intermediate tree, we compute an optimal partitioning. For each node, we have to decide whether it is going to be part of a sibling interval in the solution, or whether it will be part of the root partition. This decision is based on optimal solutions for already processed intermediate trees.

More formally, for each $j < n$ and each $s \in \{w(t), \dots, K\}$, we determine a single element $\mathcal{P} \in D_j^s$, and store it in a table indexed by j and s .

For $j = 0$, we have $D_0^s = \{(t, t)_T\}$ for all s , hence $\mathcal{P} = \{(t, t)_T\}$. For $j > 0$, Lemma 2 states that we only have to consider a limited number of candidates: Either our desired partitioning \mathcal{P} is an arbitrary element of $D_{j-1}^{s'}$, or, for some m with $0 \leq m < K$, an arbitrary element of D_{j-m-1}^s together with $(c_{j-m}, c_j)_T$.

Hence, we have at most $K + 1$ candidates in each step. We process the steps in increasing order of j . This makes sure we already know a partitioning from every D_i^s with $i < j$. To determine \mathcal{P} in each step, we just check all candidates for feasibility and store a candidate with minimum cardinality that also has minimum root weight.

The algorithm in Fig. 4 implements this strategy. It uses a table $D(s, j)$ to store a partitioning from D_j^s . It is assumed that out-of-bounds accesses to D (i.e. where $s > K$) always return a dummy interval with $\text{card} = \infty$. This makes exposition of our algorithm simpler (we do not need to check for out-of-bounds conditions in the pseudocode).

Lemma 2 tells us that each partitioning $D(s, j)$ is either the same as an existing partitioning, or it extends an existing partitioning by at most one interval. Hence, it is sufficient to store as entries in D either a copy of another partitioning, or only the added interval and a pointer to the remaining chain of intervals. This next pointer is implemented as a pair of indices of another partitioning in the

input $T = (\{t, c_1, \dots, c_n\}, t, p, \triangleleft, w)$ flat tree
output D dynamic programming table with final result in $D(w(t), n)$

```

for  $s := w(t)$  to  $k$ 
   $D(s, 0).begin := t$ 
   $D(s, 0).end := t$ 
   $D(s, 0).card := 1$ 
   $D(s, 0).rootweight := w(t)$ 
   $D(s, 0).next := (0, 0)$ 
for  $j := 1$  to  $n$ 
  for  $s := w(t)$  to  $K$ 
     $s' := s + w(c_j)$ 
     $P := D(s', j - 1)$ 
     $w := 0$ 
     $m := 0$ 
    while  $m < j \wedge m < K \wedge w < K$ 
       $w := w + w(c_{j-m})$ 
      if  $w \leq K$ 
         $crd := D(s, j - m - 1).card + 1$ 
         $rw := D(s, j - m - 1).rootweight$ 
        if  $crd < P.card \vee (crd = P.card \wedge rw < P.rootweight)$ 
           $P.begin := c_{j-m}$ 
           $P.end := c_j$ 
           $P.card := crd$ 
           $P.rootweight := rw$ 
           $P.next := (s, j - m - 1)$ 
         $m := m + 1$ 
       $D(s, j) := P$ 

```

Entries in the dynamic programming table $D(i, j)$

begin	first node of interval
end	last node of interval
card	cardinality of best partitioning so far (length of next chain)
rootweight	rootweight of best partitioning so far
next	index of next interval

Figure 4: Algorithm FDW for flat tree partitioning

table. The new or copied interval is represented by the two bounding nodes (`begin` and `end`). In addition, each entry in D (Fig. 4) has two fields that contain the cardinality and the weight of the root partition to avoid recomputing them during comparisons.

Having run the algorithm, an optimal tree sibling partitioning can be obtained by starting at the interval in $D(w(t), n)$ and traversing the list of next pointers until we reach a next pointer with value $(0, 0)$.

It is easy to specify the runtime of this algorithm: There are three nested loops, the outermost loop is processed at most n times, and the two inner loops are processed at most K times. Hence, the algorithm has a worst-case runtime of $O(nK^2)$.

3.2.3 Optimizations

We do not need to determine $D(s, j)$ for every value of s . For each j , we only need to consider those values of s that are needed by higher values. These are always the sum of node weights. Hence, we only need s values that are sums of the weight of the root node and the weights of nodes to the right of c_j .

A simple way to achieve this is to use the memoization technique: We do not precompute all entries in the D table as shown in our pseudocode. Instead, we only compute the entries on demand and remember them in D . Subsequent requests for the same entry can then be satisfied in $O(1)$ time. This does not affect the asymptotical complexity, but significantly reduces real-world runtime: Only a fraction of the whole D table is really needed for real trees (for an example, see Sec. 3.3.6).

input T tree
output D dynamic programming table

```

for all nodes  $v$  of  $T$  in postorder
  for  $s := w_T(v)$  to  $K$ 
     $D(v, s, 0).begin := \text{NIL}$ 
     $D(v, s, 0).end := \text{NIL}$ 
     $D(v, s, 0).card := 0$ 
     $D(v, s, 0).rootweight := s$ 
     $D(v, s, 0).next := (0, 0)$ 
  for  $j := 1$  to  $childcount(v)$ 
    for  $s := w_T(v)$  to  $K$ 
       $s' := s + D(v).rootweight$ 
       $P := D(v, s', j - 1)$ 
       $w := 0$ 
       $m := 0$ 
      while  $m < j \wedge m < K \wedge w < K$ 
         $w := w + D(c_{j-m}(v)).rootweight$ 
        if  $w \leq K$ 
           $crd := D(v, s, j - m - 1).card + 1$ 
           $rw := D(v, s, j - m - 1).rootweight$ 
          if  $crd < P.card$ 
             $\vee (crd = P.card \wedge rw < P.rootweight)$ 
               $P.begin := c_{j-m}(v)$ 
               $P.end := c_j(v)$ 
               $P.card := crd$ 
               $P.rootweight := rw$ 
               $P.next := (s, j - m - 1)$ 
             $m := m + 1$ 
           $D(v, s, j) := P$ 

```

Figure 5: Algorithm GHDW for deep tree partitioning

3.3 Optimal Deep Tree Partitioning

In this section, we extend the FDW algorithm for optimal flat tree partitioning to deep trees. We first propose a straightforward extension of FDW to deep trees, called GHDW (for Greedy–Height/Dynamic–Width). GHDW uses locally optimal partitionings to construct a global partitioning. However, there are cases in which GHDW yields suboptimal results, and we show that a globally optimal solution for our partitioning problem sometimes requires locally suboptimal solutions. We then characterize the kind of locally suboptimal partitionings that are needed and show how to generate them. We incorporate this into our dynamic programming algorithm. Finally, we arrive at a linear time algorithm for optimal tree sibling partitioning.

3.3.1 The GHDW Algorithm

It is tempting to use the result of the FDW algorithm as S in Lemma 1. In a bottom-up manner, we can collapse flat subtrees into single nodes, producing a flat tree at the next higher level. When reaching the root, we have a feasible tree sibling partitioning that is constructed from locally optimal partitionings. This approach uses our dynamic programming algorithm for each inner node, but with respect to the "height" of the tree, it proceeds in a greedy manner, always choosing an optimal partitioning for each subtree. We call this approach GHDW (employing a Greedy strategy for the Height of the tree, and Dynamic programming for the Width).

The pseudo-code is shown in Fig. 5. The code looks like FDW with an additional outer loop over all nodes. However, we now deal with deep trees and use appropriate primitives to access the tree structure: We use $c_j(v)$ to specify the j th child of v , and $childcount(v)$ to denote the number of children of v .

For each node, flat tree partitioning is performed once and the results are stored in the dynamic programming table D . In GHDW, D has an extra dimension compared to FDW because we have one

