Erschienen in: Journal of Discrete Algorithms ; 5 (2007), 3. - S. 546-552
https://dx.doi.org/10.1016/j.jda.2006.09.003

Optimal leaf ordering of complete binary trees

Ulrik Brandes

Department of Computer and Information Science, University of Konstanz, Germany

Abstract

Ordering a set of items so as to minimize the sum of distances between consecutive elements is a fundamental optimization problem occurring in many settings. While it is $\mathcal{NP}$-hard in general, it becomes polynomially solvable if the set of feasible permutations is restricted to be compatible with a tree of bounded degree. We present a new algorithm for the elementary case of ordering the n leaves of a binary tree with height $\log n + \mathcal{O}(1)$. Our algorithm requires $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space. While the running time is a log-factor away from being asymptotically optimal, the algorithm is conceptually simple, easy to implement, and highly practical. Its implementation requires little more than a few bit-manipulations.

Keywords: Optimal leaf ordering; Bit-manipulation algorithms; Permutations

1. Introduction

Given a set of elements and a pairwise distance function, it is a fundamental problem to determine an ordering which minimizes the sum of distances between consecutive elements. For example, the well-known Traveling Salesman Problem (TSP) is an instance of this category, showing that the general problem is $\mathcal{NP}$-hard.

If the class of permutations is restricted to be compatible with a tree in which the elements to be ordered form the leaves, the problem is polynomially solvable if the maximum degree of a tree node is bounded by a constant [1]. Instances of this kind occur, e.g., in dendrogram seriation [2] or pixel ordering for image compression [3]. The best previous algorithms to solve this problem exactly required $\mathcal{O}(2^d n^3)$ time and $\mathcal{O}(2^d n^2)$ space [1] or $\mathcal{O}(4^d n^3)$ time and $\mathcal{O}(dn^2)$ space [2], where n is the number of leaves and d is the maximum degree of any node in the tree. Recently an optimal, though somewhat involved, $\mathcal{O}(4^d n^2)$ time and $\mathcal{O}(4^d n)$ space algorithm has been proposed by Deĭneko and Tiskin [4].

Here we consider the special case of complete binary trees, which naturally occurs in applications such as the above-mentioned pixel ordering where the permutation tree can be introduced at will. It is used to linearly order the pixels so that they form longer intervals with equal color and thus to improve image compression ratios. For this particular application, it is illustrated in [3] that quadratic space requirement is prohibitive, since for an image of 512×512 pixels, even a single byte per pixel pair results in a total of 64 Gigabytes. We here present a practical $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space algorithm, which reduces the memory requirement in this example to about a Megabyte.

E-mail address: ulrik.brandes@uni-konstanz.de.

The remainder is organized as follows. In Section 2, we formally define the leaf ordering problem and recapitulate a dynamic programming approach used in previous algorithms. Our new algorithm is introduced and analyzed in Section 3. We first show how to compute only the value of an optimal ordering, and then extend the algorithm to determine the optimal ordering as well. We conclude with a brief discussion in Section 4.

2. Preliminaries

Let $V = \{0, \dots, n-1\}$ denote a set of elements and $d : V \times V \to \mathbb{R}$ an arbitrary dissimilarity measure. We do not require any special properties of d such as symmetry or the triangle inequality. An *optimal ordering* of V is a bijective mapping $\pi : \{0, \dots, n-1\} \to V$ such that

$$D(\pi) = \sum_{i=0}^{n-2} d\big(\pi(i), \pi(i+1)\big)$$

is minimum. Several equivalent variants of the problem exist. In the Traveling Salesman Problem, for instance, the cyclic sum $D(\pi) + d(\pi(n-1), \pi(0))$ is to be minimized. If d is a measure of similarity, the objective is to be maximized.

Given a tree T with leaf set V, an ordering π of V is called *consistent* with T, if the children of the inner nodes of T can be ordered such that the leaves appear in the order given by π. An *optimal leaf ordering* of V with respect to T is an ordering π such that $D(\pi)$ is minimum among all orderings consistent with T.

2.1. Previous dynamic programming solution

For binary trees, the algorithms of Burkhard et al. [1] and Bar-Joseph et al. [2] agree (except for some heuristic improvements in the latter). Let T be a binary tree, and denote by $T(v)$ the subtree rooted at node v. An optimal leaf ordering consistent with T is determined by a bottom-up computation of subintervals. For a node v, denote by $opt(v, i, j)$ the value of an optimal leaf ordering of $T(v)$ that starts and ends with leaves $i, j \in T(v)$. If v is a leaf, then $opt(v, v, v) = 0$. Otherwise, let u and w be the children of v such that $i \in T(u)$ and $j \in T(w)$. Then the following optimality criterion holds

$$opt(v, i, j) = \min_{x \in T(u), y \in T(w)} opt(u, i, x) + d(x, y) + opt(w, y, j).$$

It can be shown that this dynamic program needs $\Theta(n^3)$ time in the worst case. Because solutions of subproblems for all combinations of left and right border leaves need to be computed, the space requirement is $\Theta(n^2)$ in the worst case.

3. The algorithm

In this section, we consider the optimal leaf ordering problem for complete binary trees B_n with $n = 2^k$ leaves. For an ordered binary tree, a standard labeling starts with the empty string at the root and appends a zero or one whenever we descend to the left or right. Note that the k-bit string assigned to a leaf represents the position of that leaf in the leaf order, and that the bit string assigned to an inner node is the common prefix of all its descendants. See Fig. 1 for illustration.

The following symbols are used to denote operations on the bit-string representation of positions and element indices.

$\odot$ bitwise AND

$\oplus$ bitwise OR (inclusive OR)

$\otimes$ bitwise XOR (eXclusive OR)

For a k-bit string $b_{k-1} \cdots b_0$ that is not all zeros, let $rob(b_{k-1} \cdots b_0) = \min\{i : 0 \leqslant i < k, b_i = 1\}$ denote the position of the rightmost 1-bit.

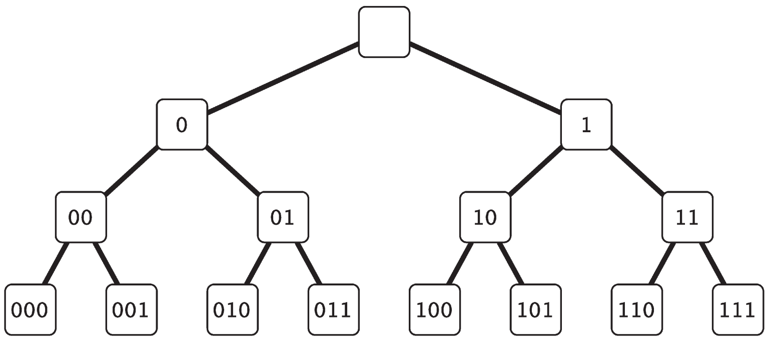

Fig. 1. Complete binary tree B_n with $n = 2^3$ leaves and canonically labeled nodes.

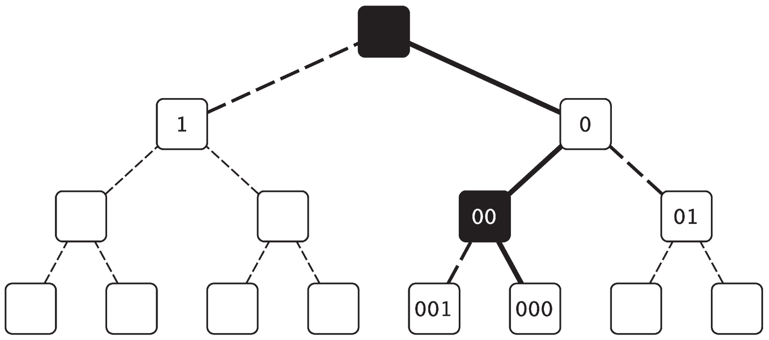

Fig. 2. Black nodes are flipped to bring leaf $i = 000_2$ into position $p = 101_2$. Since $000_2 \otimes 101_2 = 101_2$, the flipped nodes are the ancestors of i at levels 0 and 2. All leaves to the left of i are descendants of left siblings of nodes on the path to the root (i.e., of 1 and 001).

An inner node of the tree is said to be *flipped* by a permutation of the leaves, if the order of its children is reversed. There are exactly 2^{n-1} permutations consistent with B_n, since there is a one-to-one correspondence between feasible permutations and the subset of inner nodes that are flipped. A permutation can thus be encoded by a sequence of $n-1$ bits, where the ith bit indicates whether the corresponding inner node is to be flipped.

3.1. Optimal value

We first show how to determine the value of an optimal ordering with the desired time and space complexity. Previous algorithms are based on a dynamic programming approach, in which optimal solutions for subtrees with given boundary elements are determined bottom-up.

The crucial observation for reducing the large number of partial solutions is that, in a complete binary tree, fixing any leaf to a given position uniquely determines a partition into preceding and succeeding leaves. We first characterize those permutations that move a leaf into a given position. See also Fig. 2.

Lemma 1. *A permutation consistent with B_n moves leaf i, $0 \leqslant i < n$, into position p, $0 \leqslant p < n$, if and only if it flips exactly those ancestors of i that have a label of length h for all $0 \leqslant h < k-1$ with $b_{k-h-1} = 1$ in the bit-representation of $i \otimes p = b_{k-1} \cdots b_0$.*

Proof. Let $0 \leqslant i, p < n = 2^k$ and consider an ancestor v of i in level h, $0 \leqslant h < k-1$, with label $b_{k-1} \cdots b_{k-h}$. Flipping v corresponds to flipping bit b_{k-h-1} in the label of all its descendants.

To move leaf i into position p, we need to flip those bits in i that differ from the corresponding bits in p, which in turn corresponds to flipping the ancestors of i whose labels end just before those bits. □

It is important to note that the proof relies on the assumption that the tree is complete. In general, the following statement does not hold for incomplete trees.

Lemma 2. *Fixing leaf i, $0 \leqslant i < n$, at position p, $0 \leqslant p < n$ in B_n yields a unique partition of the remaining leaves into those to the left and right of i.*

Proof. To reach position p in B_n from the root, we read the bit-representation of p from left to right and descend to a right child if and only if the next bit equals 1. Clearly, we can reach a position to the left of p if and only if the first time we deviate from this procedure is by going to the left even though the next bit equals 1. From that node on we may continue the descent arbitrarily.

Since we assume that some leaf i is fixed to position p, it follows from Lemma 1 that the set of flipped nodes on the path from the root to i in p is uniquely determined. So the set of leaves reachable by descending to a position left of p is uniquely determined as well. □

The leaves, say, to the left of a fixed leaf can be determined explicitly by going through all 1-bits of p and enumerating all suffixes behind each of them (note that this gives exactly p leaves). See again Fig. 2. However, we are interested only in the subset of those leaves that are not only to the left, but also potential predecessors.

Lemma 3. *If leaf i, $0 \leqslant i < n$, is fixed at position p, $0 < p < n$, in B_n then there are exactly $r = 2^{rob(p)}$ leaves that can precede i in any permutation consistent with B_n. These leaves are numbered $i \otimes (r \oplus j)$ for $j = 0, \dots, r-1$, or, equivalently, $i \otimes s$ for $s = r, \dots, 2r-1$.*

Proof. Let $p = b_{k-1} \cdots b_0$, $0 \leqslant p \leqslant n-2$, and note that the prefix $b_{k-1} \cdots b_{rob(p)}$ belongs to the first ancestor on the path to the root that has a left sibling v. The potential predecessors of i have the same prefix as i to before the rightmost 1-bit of p, and a different value of that bit. All following bits may be altered arbitrarily. □

With the above observations, the value of an optimal leaf ordering can be determined by the dynamic programming approach shown in Algorithm 1. Note that the transposed procedure of iterating over all leaves in the first position and appending one leaf at a time is not feasible, since there may be several optimal extensions with the same value, and there is no way to tell with which one to continue.

Theorem 4. *For a complete binary tree, the value of an optimal leaf ordering can be determined in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space.*

Proof. Consider Algorithm 1, which solves the optimal leaf ordering value problem for complete binary trees because of the following invariant: $opt[i, p \odot 1]$ is the value of an optimal subordering ending with leaf i in position p. Lemma 2 states that fixing a leaf at some position uniquely determines the leaves that appear in the prefix up to that position, so that the optimal value of a prefix is completely determined by its last entry. The invariant clearly holds

```
Input : array $V$ with $n = 2^k$ elements (leaves)
        dissimilarity function $d : V \times V \to \mathbb{R}$
Output: optimal leaf ordering value
if $n < 2$ then return 0
for $i = 0, \dots, n-1$ do $opt[i, 0] \leftarrow 0$
// for each position $p$...
for $p = 1, \dots, n-1$ do
1.1     $r \leftarrow 2^{rob(p)}$; $odd \leftarrow p \odot 1$
        // ...determine value of optimal prefix ending with $i$ in $p$
        for $i = 0, \dots, n-1$ do
1.2         $opt[i, odd] \leftarrow \min_{s=r,\dots 2r-1} opt[i \otimes s, odd \otimes 1] + d(i \otimes s, i)$
return $\min_{i=0,\dots,n-1} opt[i, 1]$
```

Algorithm 1. Optimal leaf ordering value.

for the first position, and by Lemma 3, the minimum in line 1.2 is taken over all feasible predecessors of leaf i in position p.

To achieve the stated running time, the rightmost 1-bit of each position is determined by shifting 1 to the left until bitwise AND with the position produces a non-zero result. Note that this corresponds to starting from leaf p in the tree and walking up to an ancestor that is a right child (or the root), so that each node of the tree is touched only once. and the total time required is linear.

Let $n = 2^k$. For a fixed leaf i, the total number of predecessors considered in minimum computations is

$$\sum_{p=1}^{n-1} 2^{rob(p)} = \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \cdots + \frac{n}{2^k} \cdot 2^{k-1} < nk.$$

Hence the overall running time is in $\mathcal{O}(n^2 \log n)$. Clearly, the two *opt* arrays require only linear space. □

If the two's-complement representation of integers is used, the computation of the rightmost 1-bit in line 1.1 can be simplified, since then $2^{rob(p)} = p \odot (-p)$.

3.2. Optimal ordering

An optimal leaf ordering can be determined by using Algorithm 1 with an *opt*-array for each position and reconstructing optimal predecessors from right to left when the algorithm terminates. However, this approach requires quadratic space. We next show how to determine within the same asymptotic time bound and only linear space an ordering, for which the optimal value is attained.

The idea is to use linear additional space to remember the leaf in the middle of an optimal interval, and use this boundary condition to recursively repeat the computation in the first and second half of the interval. So we need to know which leaves are to be sorted in these sub-interval.

Lemma 5. *If a leaf i, $0 \leqslant i < n$ is fixed at position $p = (1+l) \cdot 2^m - 1$ with $0 \leqslant m \leqslant k$ and $0 \leqslant l \leqslant k - m$, then the set of leaves in positions $l \cdot 2^m, \ldots, (1+l) \cdot 2^m - 1$ is exactly $\{i \otimes j\colon 0 \leqslant j < 2^m\}$.*

Proof. The 2^m leaves in the interval ending with i in position p are exactly the descendants of the $(1+l)$th node v at level $k-m$, i.e. their label consist of the label of v as a prefix followed by any bit string of length m. Since i is fixed at p, the label of v is the prefix of length $k-m$ of the label of i. The exclusive-or operation thus enumerates all m leaves, though in non-canonical order. □

Theorem 6. *For a complete binary tree, an optimal leaf ordering can be determined in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ additional space.*

Proof. Consider Algorithm 2. The first call to method *subtreeorder* essentially results in an execution of Algorithm 1, except that for each optimal prefix of length at least $\frac{n}{2}$ we know its *pivot* element in position $\frac{n}{2} - 1$. When the method is executed again on the first and second half of the position interval, π already contains the last leaf in each of the two subintervals in an optimal ordering. So we can initialize the optimal prefix value of the right half with the distance to the known predecessor (the pivot), and pick the ordering that ends with the correct leaf. The number of entries in the optimal leaf ordering π thus doubles in each level of the recursion.

The running time for an execution of *subtreeorder* is that of Algorithm 1 on a tree with $p_{right} - p_{left} + 1$ leaves plus twice the same complexity for two problems of half the size, i.e. it obeys the recursion

$$\mathcal{T}(n) = 2 \cdot \mathcal{T}(n/2) + \mathcal{O}(n^2 \log n).$$

This recurrence yields $\mathcal{T}(n) \in \mathcal{O}(n^2 \log n)$ as is easily verified by induction. Only four arrays of linear size are used. □

For practical purposes it might be useful to increase the number of stored pivots and thus avoid some levels of recursion.

```
Input  : array $V$ with $n = 2^k$ elements (leaves)
         dissimilarity function $d : V \times V \to \mathbb{R}$
Output: optimal leaf ordering $\pi$

begin
    $\pi[n-1] \leftarrow 0$
    subtreeorder$(0, n-1)$
end

subtreeorder$(p_{left}, p_{right})$ begin
    $m \leftarrow p_{right} - p_{left}$
    if $m = 0$ then return
    if $m = 1$ then $\pi[p_{left}] \leftarrow \pi[p_{right}] \otimes 1$; return
    // initialization
    $i_{right} = \pi[p_{right}]$
    for $j = 0, \ldots, m$ do
        if $p_{left} = 0$ then
            $opt[i_{right} \otimes j, 0] \leftarrow 0$
        else
            // left boundary condition (predecessor already known)
            $opt[i_{right} \otimes j, 0] \leftarrow d(\pi[p_{left} - 1], i_{right} \otimes j)$
    $p_{mid} \leftarrow p_{left} + \frac{m-1}{2}$
    // optimal prefixes (pass on leaf at end of first half)
    for $p = p_{left} + 1, \ldots, p_{right}$ do
        $r \leftarrow 2^{rob(p)}$;  $odd \leftarrow p \odot 1$
        for $j = 0, \ldots, m$ do
            $i \leftarrow i_{right} \otimes j$
            $s^\star \leftarrow \underset{s=r,\ldots 2r-1}{\arg\min}\, opt[i \otimes s, odd \otimes 1] + d(i \otimes s, i)$
            $opt[i, odd] \leftarrow opt[i \otimes s^\star, odd \otimes 1] + d(i \otimes s^\star, i)$
            if $p = p_{mid}$ then
                $pivot[i, 1] \leftarrow i$
            else
                $pivot[i, odd] \leftarrow pivot[i \otimes s^\star, odd \otimes 1]$
    if $m = n-1$ then $\pi[n-1] \leftarrow \underset{i=0,\ldots,n-1}{\arg\min}\, opt[i, 1]$
    $\pi[p_{mid}] \leftarrow pivot[\pi[p_{right}], 1]$
    subtreeorder$(p_{left}, p_{mid})$
    subtreeorder$(p_{mid} + 1, p_{right})$
end
```

Algorithm 2. Optimal leaf ordering.

3.3. Checking and coding

The prefix-numbering of inner nodes in the order-restricting tree turned out to be a useful tool in the algorithms above. It also yields a simple algorithm to test the output of an implementation, or any given permutation, for consistency with the tree.

Theorem 7. *A permutation $\pi : V \to V$ of $V = \{0, \ldots, n-1\}$ is consistent with the complete binary tree with leaves V, if and only if*

$$\pi^{-1}(p-1) \otimes \pi^{-1}(p) \otimes 2^{rob(p)} < 2^{rob(p)} \tag{1}$$

for all $p = 1, \ldots, n-1$.

Proof. Clearly, any $i \in V$ can be mapped to 0. However, for π to be compatible with the complete binary tree B_n, the element mapped to 1 must differ from i exactly in the least significant bit. More generally, the labels of two consecutive leaves of B_n share the prefix of their lowest common ancestor, and differ in the following bit. This is exactly what is tested in (1), since the lowest common ancestor of positions $p-1$ and p has a label of length $\log n - rob(p)$.

The reverse implication follows from a simple recursive argument since each inner node is the lowest common ancestor of some pair of consecutive leaves, and (1) guarantees that all leaves in its subtree share the same prefix but are divided into those that have a zero or one in the next position. □

Corollary 8. *It can be checked in linear time whether a given permutation is consistent with the complete binary tree defined over its argument sequence.*

Another consequence is that we can encode and decode a permutation consistent with the complete binary tree in linear time using the $n-1$ bits that indicate for each inner node whether it is flipped or not. Inequality (1) implies that the inner node of the tree that is the lowest common ancestor of $p-1$ and p is flipped, if and only if

$$\pi^{-1}(p) \odot 2^{rob(p)} = 0.$$

To ease reconstruction, we additionally use the fact that $\pi(0)$ gives the flipping bits on the path to the leftmost leaf, and list the flipping bits in preorder.

4. Discussion

We have presented a highly practical algorithm for determining optimal leaf orderings of complete binary trees with respect to a dissimilarity function $d : V \times V \to \mathbb{R}$. It runs in near-optimal $\mathcal{O}(n^2 \log n)$ time, requires only $\mathcal{O}(n)$ extra space, and can be implemented with just a few bit-operations on position indices. In particular, an input array is ordered without constructing the order-restricting binary tree.

The algorithm works without modification for position-dependent dissimilarities (defined on $V \times V \times \{0, \ldots, |V|-1\}$). It is easily modified to maximize sums of similarities or optimize cyclic sums (tours instead of orderings). Furthermore it generalizes to binary trees with height $\log n + \mathcal{O}(1)$ by completing the input, i.e. by adding dummy subtrees. Note that completing a binary tree of height $\log n + h$ results in a tree with $\mathcal{O}(hn)$ leaves. Note also that completion of trees with larger height yields a superlinear number of leaves.

Unfortunately, the algorithm cannot be generalized to arbitrary bounded-degree trees in the same way as previous approaches [1,2] without allowing more than linear space.

Acknowledgements

I would like to thank an anonymous referee for helpful comments, in particular for pointing out the optimal algorithm in [4].

References

[1] R. Burkhard, V.G. Deĭneko, G. Woeginger, The travelling salesman and the PQ-tree, Mathematics of Operations Research 24 (1) (1999) 262–272.

[2] Z. Bar-Joseph, E.D. Demaine, D.K. Gifford, N. Srebro, A.M. Hamel, T.S. Jaakkola, K-ary clustering with optimal leaf ordering for gene expression data, Bioinformatics 19 (9) (2003) 1070–1078.

[3] Z. Bar-Joseph, D. Cohen-Or, Hierarchical context-based pixel ordering, in: Proceedings of EUROGRAPHICS, Computer Graphics Forum 22 (3) (2003) 349–358.

[4] V.G. Deĭneko, A. Tiskin, Double-tree approximations for metric TSP: Is the best one good enough? Mathematics of Operations Research, submitted for publication.