Linear-time Algorithms for Tree Partitions with Small Cutsize

Leonidas Palios
Department of Computer Science
University of Ioannina, Ioannina, Greece

Abstract: In this paper, we consider the problem of partitioning the node set of a tree T of n nodes and degree d ($d \geq 2$) into two sets of prespecified cardinalities. The goal is to keep the *cutsize* (i.e., the number of T's edges with one endpoint in either set) as small as possible. The problem finds applications in VLSI and networking. We describe two linear time algorithms which produce partitions of the tree nodes into two sets V_1 and V_2 of desired sizes achieving cutsizes that do not exceed $(d+1)/2 \ \log_{(d+3)/2} m+1$ and $(d+2)/2 \ \log_{d+1}(2\,m)+1/2$ respectively, where $m = \min\{|V_1|, |V_2|\}$. We also compute a lower bound of $\lfloor (d-1)/2 \rfloor \ (\log_d m - \log_d \log_d m) - d/2$ for the cutsize of such a partition when $d \geq 3$ and for all $m = \lfloor (d^i - 1)/(2d-2) \rfloor$ $(i = 2, \ldots, \lfloor \log_d n \rfloor)$; this proves that the algorithms we describe are asymptotically worst-case optimal and the ratio to the optimal solution is a small constant factor. Both algorithms are similar to the Tree Coloring algorithm of MacGregor for tree partitions [8] but improve on both its time complexity and upper bound on the partition cutsize. Brief experimentation with the two algorithms suggests that the first algorithm performs better than the second, despite the fact that the upper bound on the cutsize of the produced partition is worse than the corresponding upper bound for the second algorithm.

1. Introduction.

The work in this paper is motivated by a common *graph partitioning* problem, which can be stated as follows: given a graph G of n nodes and a positive integer $m < n$, find a partition of G's node set into two sets of sizes m and $n - m$ that minimizes the number of G's edges with one endpoint in either set. The problem is very interesting, as it both incorporates the minimization of a resource and a load balancing mechanism through appropriate values of m, and finds applications in VLSI (placement of components) and networking (minimization of communication cost). The most popular among all partitions of a graph is its *bisection*, where the cardinalities of the two partition sets differ by no more than 1. Due to the need for good graph bisections, especially in graph algorithms based on the divide-and-conquer paradigm, fast algorithms that yield optimal or near-optimal bisections are very important. Unfortunately, the graph bisection problem on general graphs is proven to be NP-complete [5]; this motivated research towards finding heuristics with good average case behavior ([1], [2], [6], [7]). It is yet unknown whether the graph bisection problem remains NP-complete when restricted to the class of planar graphs, whereas optimal bisections of trees can be computed in $O(n^3)$ time using dynamic programming. Returning to the problem of partitioning a graph into two sets of prespecified cardinalities, few results are known. In a recent paper [3], Bui and Peck gave an algorithm to compute an optimal partition of a planar graph of n vertices into sets of sizes m and $n-m$, where $m \in [0, n]$. The algorithm runs in $O(b^2 n^3 2^{4.5b})$ time, where b is the number of edges with one endpoint in either set of the computed optimal partition; since b can be as large as $\sqrt{n}$, the algorithm is not necessarily polynomial.

In this paper, we are interested in the graph partitioning problem applied to the case of trees. Although rather restricted, the problem is still important as many graphs in practical applications often are trees. For instance, graphs that illustrate the interdependence of the procedures of a program (in terms of procedure calls) frequently exhibit a tree structure; then, the problem of assigning procedures to two remote sites (each being able to handle a certain number of them without being unacceptably slow) while at the same time minimizing the communication cost reduces to optimally partitioning a tree.

The problem of partitioning trees such that the *cutsize* (that is, the number of tree edges with one endpoint in either partition set) is small has received considerable attention. Most of the work has been centered around bisections. MacGregor [8] describes the Tree Coloring algorithm which bisects the node set of a d-tree achieving an upper bound $\tau(d, n)$ on the cutsize, where

$$\tau(d,n) = \begin{cases} \frac{1}{2} + 2\log_3 n, & \text{if } d = 2, 3; \\ \\ \frac{1}{2} + \frac{d+2}{2}\log_{d+1} n, & \text{if } d \geq 4. \end{cases}$$

He extends the algorithm to handle k-partitions guaranteeing an upper bound of $(k-1)(\frac{1}{2} + (d-1)\log_{2d-3}\frac{2n}{k})$ in the cutsize. In either case, the algorithms take $O(n \log n)$ time. Diks et al. [4] have recently described a linear time algorithm for tree bisections which ensures a $d \log_d n$ cutsize. In the process of computing a tree bisection, their algorithm computes general $(m, n-m)$-partitions, but Diks et al. do not analyze its performance in the general case. We note that an optimal partition (i.e., one that achieves the minimum cutsize) can be computed by an algorithm based on the dynamic programming paradigm: we recursively compute the corresponding optimal cutsizes of each of the subtrees rooted at the children of the root, and then combining the results to cover all possible cases depending on whether and which among the edges incident upon the root belong to the cutset of a partition or not. The algorithm runs in $O(n^3)$ time and requires $O(n)$ space. Note however that if the cutsets of the optimal partitions need be computed too, then the required space increases to $O(n\, d \log_d n)$, as the cutsets of all intermediate optimal partitions need be stored. Finally, if the tree has a symmetric structure, then the algorithm may be made to run faster by taking advantage of the symmetry. For instance, the time complexity of the algorithm when applied to a complete binary tree reduces to $O(n^2)$.

In this work, we describe and analyze two algorithms for $(m, n-m)$-partitions of trees. The first algorithm works by collecting subtrees of the tree, eventually forming the desired partition; for an n-node rooted d-tree, it produces a partition of the node set of the tree into two sets of m and $n-m$ nodes, whose cutsize is no more than $(d+1)/2\ \log_{(d+3)/2}\min\{m, n-m\}+1$. The second algorithm relies in the same basic strategy and achieves partitions whose cutsize does not exceed $(d+2)/2\ \log_{d+1}(2\min\{m, n-m\})+1/2$ for $d \geq 4$, and $2\ \log_3(2\min\{m, n-m\})+1/2$ for $d = 2, 3$. This algorithm is an extension of the Tree Coloring algorithm of MacGregor; this is why, for bisections it matches the latter algorithm's bound on the cutsize.

Finally, we note that the performance of our algorithms is a small constant factor away from the corresponding lower bounds for $d \geq 3$. In [9], we show that the optimal bisection of appropriate families of trees exhibits cutsizes no less than $\lfloor (d-1)/2 \rfloor\ (\log_d n - \log_d \log_d n) - d/2$ for $d \geq 3$. This result can be used to prove lower bounds of $\lfloor (d-1)/2 \rfloor\ (\log_d m - \log_d \log_d m) - d/2$ for the cutsize of an $(m, n-m)$-partition when $d \geq 3$ and for all $m = \lfloor (d^i - 1)/(2d-2) \rfloor$ $(i = 2, \ldots, \log_d n)$.

The paper is structured as follows. In Section 2, we present the necessary terminology that we use throughout the paper. In Section 3, we present the two algorithms to partition the nodes of a tree into two sets of desired cardinalities, analyze their time complexity and derive upper bounds on the cutsize of the partition in terms of the sizes of the partition sets and parameters of the tree. In Section 4, we prove the corresponding lower bounds. Section 5 concludes the paper with the final remarks and some open questions.

2. Terminology.

The notion of a tree is one of the most fundamental notions in computer science; its definition and properties can be found in nearly every introductory book on graph theory. Below, we quickly review the basic terminology. A *tree* is a collection of *nodes* connected by *edges* such that for every pair of nodes u and v, there exists a unique path connecting them. In case a specific node of the tree has been selected as the *root*, the tree is called *rooted*; otherwise it is called *unrooted*. Since any unrooted tree can become rooted if one of its nodes is designated as its root, we will restrict our attention to rooted trees.

A rooted tree can be displayed in a layered fashion along the vertical direction as follows (Figure 1). The highest or *0-th level* is occupied by the root only. The *first level* contains all nodes of the tree that are connected to the root by an edge. In general, the *i-th level* contains all nodes that are connected to a node in the $(i-1)$-st level, and do not belong to the $(i-2)$-nd level. (The latter requirement ensures that each node will be displayed once, and that the layering process will eventually terminate.) If a node belongs to the i-th level, we say that its *depth* is i. The maximum among the depths of all the tree node defines the *height* of the tree (for instance, the height of the tree in Figure 1 is 4). Additionally, the nodes u_i of the j-th level that are connected to a node v in the $(j-1)$-st level by means of an edge are called v's *children*; v is called the *parent* of the u_is. If a node has no children, it is called a *leaf* of the tree. A non-leaf node of the tree is called *internal*. The tree in Figure 1 has 6 leaves and 6 internal nodes. The *subtree* rooted at a node v is the part of the tree that contains v when the edge between v and its parent is removed.

In case no node of the tree has more than d children, we say that the tree is a *d-tree*. For minimality reasons, we assume that there exists a node in a d-tree that has exactly d children. Finally, we say that a d-tree is *complete* if each of its internal nodes has exactly

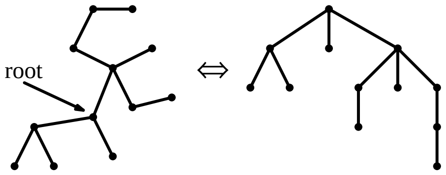

Figure 1

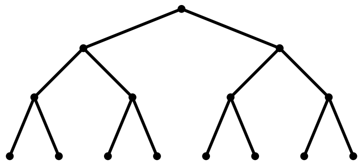

Figure 2

d children, and all the leaves belong to the same (lowest) level (Figure 2 shows a complete 2-tree of height 3). It is an easy exercise to prove that the i-th level of a complete d-tree contains exactly d^i nodes; if its height is h, the entire tree contains a total of $(d^{h+1}-1)/(d-1)$ nodes.

We now introduce the terminology and our notation regarding partitions. For $1 \leq m \leq n-1$, an *$(m, n-m)$-partition* of a graph G (or a tree, in particular) of n nodes is a partition of the node set of G into two sets with cardinalities m and $n-m$; therefore, there exist exactly $\lfloor n/2 \rfloor$ *different* partitions, namely, partitions into sets of 1 and $n-1$, 2 and $n-2$, ..., $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ nodes. A given partition of the node set of G uniquely defines the set of G's edges with one endpoint in either partition set, which is called the *cutset* of the partition; the size of the cutset is the *cutsize* of the partition. Conversely, each subset E of tree edges defines a unique partition of the nodes of the tree (note that the removal of any edge disconnects the tree), the underlying assumption being that the nodes incident upon each edge in E do not belong to the same partition set; the cutsize of this partition equals the cardinality of the set E. The above observations exhibit a special correspondence between partitions and sets of tree edges. For a specific m, there are $\binom{n}{m}$ ways to form the partition sets of an $(m, n-m)$-partition, and the corresponding cutsizes may vary substantially; the partition sets that exhibit the minimum cutsize define the *optimal* $(m, n-m)$-partition.

We close this section by noting that in the following we ignore the trivial case of a tree of 1 node, as well as 1-trees (chains), where all possible node set partitions can be achieved at cutsize 1. We therefore deal with d-trees ($d \geq 2$) of $n \geq 2$ nodes.

3. The Algorithms.

In this section, we present two linear-time algorithm that compute an $(m, n-m)$-partition of an n-node d-tree for any n and any $m = 1, \ldots, n-1$. The first algorithm is based on the result given by Lemma 3.1 below, whereas the second one in a similar result.

Lemma 3.1. *Let m_i $(1 \leq i \leq j)$ be j positive integers in non-increasing order such that $m_i < m$ for all $i = 1, \ldots, j$ and $\sum_{i=1}^{j} m_i > m$, and let k be the integer for which $\sum_{i=1}^{k} m_i \leq m < \sum_{i=1}^{k+1} m_i$. Then, $m - \sum_{i=1}^{k} m_i < \frac{m}{k+1}$.*

Proof: Clearly, $1 \leq k < j$. Since $\sum_{i=1}^{k+1} m_i > m$, we have that $m_{k+1} > m - \sum_1^k m_i$. Moreover, $m_{k+1} \leq (\sum_1^k m_i)/k$, because the m_is are in non-increasing order. Combining these two inequalities for m_{k+1}, we get $m - \sum_1^k m_i < (\sum_1^k m_i)/k \iff \sum_1^k m_i > \frac{k}{k+1} m \iff m - \sum_1^k m_i < \frac{1}{k+1} m$. ∎

Lemma 3.1 implies the following procedure for the problem of having to collect m items from batches of $m_1, \ldots, m_j$ items (where the m_is are in non-increasing order): we collect all the items from the first k batches which results into having to collect fewer than a fraction $1/(k+1)$ of m additional items. Since $k \geq 1$, the total number of times that this procedure will have to be applied until we collect all m items will be logarithmic in m.

The second algorithm is based on the same principle, albeit in an improved version of Lemma 3.1, which achieves an asymptotically better upper bound on the cutsize of the partition produced, but requires slightly more complicated handling.

The Algorithm A.

The first algorithm works as follows:

Algorithm A

traverse the tree and associate with each node t its weight $w(t)$ which is equal to the number of tree nodes of the subtree rooted at t;
root $\leftarrow$ the root of the tree; $\quad m \leftarrow \min\{m, n-m\}$;
while $(m > 0)$ **do**
    visit the nodes in the subtree rooted at *root* as long as their associated weights are at least equal to m; let s be the node whose weight $w(s)$ is the least among all these weights (which are at least equal to m);
    if $(w(s) = m)$
    then collect the nodes of the subtree rooted at s;
        add the edge (s,parent(s)) in the cutset;
        exit while-loop;
    else if $(w(s) = m + 1)$
    then /* collect the subtree rooted at s after removing one of its leaves */
        collect all the nodes of the subtree rooted at s except for one leaf, say, x;
        add the edge (x,parent(x)) in the cutset;
        if (the edge (s,parent(s)) does not belong to the cutset)
        then add this edge in the cutset;
        else remove this edge from the cutset;
        exit while-loop;
    else /* $w(s) > m + 1$ */
        $d_s \leftarrow$ degree of s;
        find k such that $\sum_{i=1}^{k} w_i \leq m < \sum_{i=1}^{k+1} w_i$ where w_i $(1 \leq i \leq d_s)$ are the weights of the children of s in non-increasing order; let the children of s with weights $w_1, \ldots, w_{d_s}$ be $t_1, \ldots, t_{d_s}$ respectively;
        sum $\leftarrow \sum_{i=1}^{k} w_i$;
        collect the nodes of the subtrees rooted at $t_1, \ldots, t_k$;
        if $(k \leq (d_s + 1)/2)$
        then add the edges $(s, t_1), \ldots, (s, t_k)$ in the cutset;

24. **else** /* $k > (d_s+1)/2$ */
25. collect s;
26. **if** (the edge (s,parent(s)) does not belong to the cutset)
27. **then** add this edge in the cutset;
28. **else** remove this edge from the cutset;
29. add the edges $(s, t_{k+1}), \ldots, (s, t_{d_s})$ in the cutset;
30. **if** ($sum = m$)
31. **then** remove from the collected set a leaf y of the subtree of t_1;
32. add the edge (y,parent(y)) in the cutset;
33. **else** $sum \leftarrow sum + 1$; /* include tree node s */
34. /* prepare next iteration of while-loop */
35. $root \leftarrow t_{k+1}$; $m \leftarrow m - sum$;

The correctness of the algorithm follows from the fact that in each iteration we reduce m by the number of collected nodes and that the weight of the new root t_{k+1} indeed exceeds the number of remaining nodes to be collected. Moreover, in lines 18-33 $w(s) > m + 1$, which implies that the total sum of the weights of the children of s is greater than m; thus, Lemma 3.1 indeed applies.

Most of the steps of algorithm A are straightforward. The steps in lines 1, 4, 6, 11, 21 and 31 can be carried out by means of a depth-first search traversal in time linear in the size of the corresponding subtree. In particular, in line 4, we do not proceed to subtrees with weight less than m while visiting the nodes of the subtree rooted at *root*; in this way, we guarantee that the time spent in the execution of line 4 in all the iterations of the while loop is $O(n)$. The location of k (line 19) is achieved in $O(d_s)$ time by means of the linear-time median finding algorithm. We determine the median of the set $\{w(t_1), \ldots, w(t_{d_s})\}$ and partition the set into two subsets S_1 and S_2 of elements less than or equal to the median and greater than the median respectively. If the sum of the elements of S_1 is larger than m, we repeat the process in S_1, otherwise, we collect the elements of S_1, we reduce m by the sum of the elements in S_1 and repeat the process in S_2 for the new m; eventually, we end up with a set of cardinality 2 or 3, which can be handled in a brute-force fashion in constant time. Since, in each of the median finding and partitioning rounds we deal with a set of half the size of the set in the previous round, the total amount of time required is linear in the degree d_s of s. Given that the new root t_{k+1} in each iteration of the while loop is different from that in the preceding iteration, the total time of the execution of line 19 is $O(n)$. The above discussion implies that the total running time of algorithm A is linear in the size of the tree.

The effect of lines 22-33 is to try to keep the cutsize small. Suppose that we want to collect the nodes of the subtrees rooted at the children $t_1, \ldots, t_k$ of s. If k does not exceed (approximately) half the degree of s, we simply collect the nodes of these subtrees, increasing the cutsize by no more than (approximately) $d/2$. If, however, k exceeds half the degree of s, we can include s in the collected set, increasing the cutsize by 1 plus the number of non-collected children of s, which does not exceed (approximately) $d/2$. (We note that in case the total number of nodes of the k subtrees is equal to the desired number, we may include s in the collected set by removing at the same time a leaf of t_1.)

We finally note that, in every execution of the lines 18-33, m is at least equal to the corresponding k of line 19; this follows from the fact that each of the collected subtrees $t_1, \ldots, t_k$ contains at least one node. In fact, $m \geq k+1$ unless the current iteration is the final iteration of the while loop.

An Upper Bound on the Cutsize The examination of all possible cases, in conjunction with Lemma 3.1, yields the following expression for the cutsize $C(m)$ of an $(m, n-m)$-partition produced by Algorithm A:

$$C(m) \leq \begin{cases} 1, & & (a) \\ 2, & \text{where } m \geq 2; & (b) \\ k + C(\lfloor m/(k+1) \rfloor), & \text{where } 1 \leq k \leq (d+1)/2; & (c) \\ k, & \text{where } 1 \leq k \leq (d+1)/2; & (d) \\ d-k+1+C(\lfloor m/(k+1) \rfloor), & \text{where } (d+1)/2 < k \leq d-1; & (e) \\ d-k+2, & \text{where } (d+1)/2 < k \leq d-1 \text{ and } m \geq k; & (f) \end{cases}$$

Case (a) results if $w(s) = m$ and may occur for any value of m (lines 6-8). Case (b) results if $w(s) = m+1$ (lines 10-16); it comes up only when $m \geq 2$, as there is always a leaf in a tree. Cases (c) and (d) correspond to the line 23 of the algorithm, the latter case occurring whenever $sum = m$. Similarly, cases (e) and (f) correspond to the lines 24-33, with case (f) resulting if $sum = m$; these cases occur only if $d \geq 3$, because, for $d = 2$, k is equal to 1 and the test $k \leq (d+1)/2$ (line 22) is true leading to cases (c) and (d).

Now, we are ready to prove our upper bound.

Theorem 3.1. *Let T be a d-tree ($d \geq 2$) of size $n \geq 2$. Then, for any positive integer $m < n$, Algorithm A finds an $(m, n-m)$-partition of the node set of T whose cutsize is at most*

$$\left\lfloor \frac{d+1}{2} \log_{(d+3)/2}\Big(\min\{m, n-m\}\Big) \right\rfloor + 1.$$

Proof: Let us assume without loss of generality that $m = \min\{m, n-m\}$. We will show that $C(m) \leq \frac{d+1}{2} \log_{(d+3)/2} m + 1$ for all $m \geq 1$, which will prove the theorem. The proof proceeds inductively on m. If $m = 1$ then our algorithm locates a leaf s with weight $w(s)$ equal to 1, collects it, and the cutsize of the resulting partition is 1, which does not exceed the value $\frac{d+1}{2} \log_{(d+3)/2} 1 + 1 = 1$ for all $d \geq 2$. Suppose now that the upper bound holds for $C(i)$ for all $i = 1, \ldots, r-1$. We want to prove that this also holds for $C(r)$, where $r \geq 2$. We examine all the cases in the expression of $C(m)$ verifying that they do not exceed the desired bound. We note that $\lfloor \frac{r}{k+1} \rfloor \leq \frac{r}{k+1} \leq r/2 < r$; thus, the inductive step implies that

$$\begin{aligned} C\Big(\Big\lfloor \frac{r}{k+1} \Big\rfloor\Big) &\leq \tfrac{d+1}{2} \log_{(d+3)/2} \Big\lfloor \tfrac{r}{k+1} \Big\rfloor + 1 \\ &\leq \tfrac{d+1}{2} \log_{(d+3)/2} \tfrac{r}{k+1} + 1 \\ &\leq \tfrac{d+1}{2} \log_{(d+3)/2} r + 1 - \tfrac{d+1}{2} \log_{(d+3)/2}(k+1) \end{aligned} \qquad (1)$$

since the function $\frac{d+1}{2} \log_a x$ is increasing for all $x > 0$.

(a) Trivially true.

(b) We need to show that $2 \leq \frac{d+1}{2} \log_{(d+3)/2} r + 1$ for all $r \geq 2$. It suffices to show that $1 \leq \frac{d+1}{2} \log_{(d+3)/2} 2$ or equivalently $\log_2 \frac{d+3}{2} - \frac{d+1}{2} \leq 0$. The last inequality holds for all $d \geq 2$; the function $f_1(x) = \log_2(x+1) - x$ is strictly decreasing for all $x > 0.45$ and $f_1(1) = 0$.

(c) In light of the inequality (1), we need only show that for $1 \leq k \leq \frac{d+1}{2}$ it is true that

$$k - \frac{d+1}{2} \log_{(d+3)/2}(k+1) \leq 0 \iff \frac{k}{\ln(k+1)} - \frac{\frac{d+1}{2}}{\ln(\frac{d+1}{2}+1)} \leq 0.$$

The last inequality holds for $k = 1$ and all $d \geq 2$; see case (b). It also holds for $k \geq 2$, since the function $f_1(x) = x/\ln(x+1)$ is strictly increasing for $x \geq 2$ and $k \leq \frac{d+1}{2}$.

(d) This case follows from case (c) as $C(.)$ is a non-negative-valued function.

(e) If we use $\kappa = d + 1 - k$, then the inequality that we want to prove becomes $\kappa + C(\frac{r}{d+2-\kappa}) \leq \frac{d+1}{2} \log_{(d+3)/2} r\ + 1$, where $2 \leq \kappa < \frac{d+1}{2}$; this follows from case (c), given that $\frac{d+1}{2} \log_a x$ is an increasing function and $d + 2 - \kappa > \kappa + 1$.

(f) Since $r \geq k$ in this case and the function $\frac{d+1}{2} \log_a x$ is increasing for all $x > 0$, it suffices to show that $d - k + 1 \leq \frac{d+1}{2} \log_{(d+3)/2} k$ where $\frac{d+1}{2} < k \leq d - 1$. If $k \geq \frac{d+3}{2}$ then $d - k + 1 \leq \frac{d-1}{2} \leq \frac{d+1}{2} \leq \frac{d+1}{2} \log_{(d+3)/2} k$, as desired. The only other case is if $k = \frac{d+2}{2}$, which implies that d is even and at least equal to 4 (recall that $d \geq 3$ in case (f)). Then, the inequality we want to prove is written as

$$d - \frac{d+2}{2} + 1 = \frac{d}{2} \leq \frac{d+1}{2} \log_{(d+3)/2} \frac{d+2}{2} \iff \frac{\frac{d}{2}}{\ln(\frac{d}{2}+1)} \leq \frac{\frac{d+1}{2}}{\ln(\frac{d+1}{2}+1)}$$

which holds, since the function $f_2(x) = x/\ln(x+1)$ is increasing for $x \geq 2$ and $\frac{d+1}{2} > \frac{d}{2} \geq 2$. ∎

In particular, for the case of a bisection: we substitute $n/2$ for m and we have

Corollary 3.1: *For any d-tree ($d \geq 2$) of $n \geq 2$ nodes, we can always find a bisection of the tree's node set whose cutsize does not exceed* $\left\lfloor \frac{d+1}{2} \log_{(d+3)/2} \frac{n}{2} \right\rfloor + 1$.

This upper bound on the cutsize of an optimal bisection of a d-trees implies an $O(\log_2 n)$ upper bound if d is a constant, and an $O(n)$ upper bound if d is $\Theta(n)$.

The Algorithm O.

Algorithm O is based on the following Lemma which is similar to Lemma 3.1.

Lemma 3.2. *Let m_i ($1 \leq i \leq j$) be j positive integers in non-increasing order such that $m_i < m$ for all $i = 1, \ldots, j$ and $\sum_{i=1}^{j} m_i > m$, and let k be the integer for which $\sum_1^k m_i \leq m < \sum_1^{k+1} m_i$. Then,* $\min\{m - \sum_{i=1}^{k} m_i, \sum_{i=1}^{k+1} m_i - m\} \leq m/(2k+1)$.

Proof: Let $a = m - \sum_{i=1}^{k} m_i$ and $b = \sum_{i=1}^{k+1} m_i - m$. Then $a + b = m_{k+1}$. Because the m_is are in non-increasing order, $m_{k+1} \leq (\sum_1^k m_i)/k$. Therefore $a + b \leq (\sum_1^k m_i)/k$. The combination of the last inequality with the equality $\sum_{i=1}^{k} m_i = m - a$, which results from rewriting the definition of a, implies that

$$a + b \leq \frac{m-a}{k} \implies (k+1)\,a + k\,b \leq m \implies \min\{a, b\} \leq \frac{m}{2k+1}. \quad ∎$$

Lemma 3.2 implies that if we allow ourselves to collect more nodes at a step and then remove the excessive ones, we may end up reducing the number of nodes that we seek to collect by at least a fraction $1/(2k+1)$ at each step. This is better than the $1/(k+1)$ fraction of Algorithm A and suggests fewer iterations and probably a smaller upper bound in the cutsize.

The algorithm works as described below. We use the variables X, Y and *swap* in order to ensure that the collected nodes are directed to their correct destination. Upon completion of the algorithm, the set A contains $\min\{m, n-m\}$ nodes, and the set B the remaining ones.

Algorithm O

1. traverse the tree and associate with each node t its weight $w(t)$ which is equal to the number of tree nodes of the subtree rooted at t;
2. *root* $\leftarrow$ the root of the tree; $m \leftarrow \min\{m, n-m\}$;
3. mark all the nodes of the tree as if they belong to the set B;
4. $X \leftarrow A$; $Y \leftarrow B$; $swap \leftarrow$ **false**;
5. **while** $(m > 0)$ **do**
6.   visit the nodes in the subtree rooted at *root* as long as their associated weights are at least equal to m; let s be the node whose weight $w(s)$ is the least among all these weights (which are at least equal to m);
7.   **if** $(w(s) \leq m+1)$ /* $w(s) = m$ or $w(s) = m+1$ */
8.   **then** put all the nodes of the subtree rooted at s in the set X;
9.   add the edge (s,parent(s)) in the cutset;
10.   **if** $(w(s) = m+1)$
11.   **then** /* remove a leaf */
12.   put one leaf, say, x, of the subtree rooted at s in the set Y;
13.   add the edge (x,parent(x)) in the cutset;
14.   exit while-loop;
15.   **else** /* $w(s) > m+1$ */
16. $d_s \leftarrow$ degree of s;
17.   find k such that $\sum_{i=1}^{k} w_i \leq m < \sum_{i=1}^{k+1} w_i$ where w_i $(1 \leq i \leq d_s)$ are the weights of the children of s in non-increasing order; let the children of s with weights $w_1, \ldots, w_{d_s}$ be $t_1, \ldots, t_{d_s}$ respectively;
18. $sum \leftarrow \sum_{i=1}^{k} w_i$;

        /* collect the subtrees rooted at $t_1, \ldots, t_k$ and possibly t_{k+1} */

19.   put all the nodes of the subtrees rooted at $t_1, \ldots, t_k$ in the set X and all the nodes of the subtrees rooted at $t_{k+2}, \ldots, t_{d_s}$ in the set Y;
20.   **if** $(w_{k+1} \geq 2 * (m - sum))$
21.   **then** put the nodes of the subtree rooted at t_{k+1} in the set Y;
22.   **if** $(k \leq d_s/2)$
23.   **then** add the edges $(s, t_1), \ldots, (s, t_k)$ in the cutset;
24.   **else** /* $k > d_s/2$ */
25.   put node s in the set X;
26.   add the edge (s,parent(s)) in the cutset;
27.   add the edges $(s, t_{k+1}), \ldots, (s, t_{d_s})$ in the cutset;

```
28.                     if  (sum = m)
29.                     then  put a leaf y of the subtree rooted at t_1 in the set Y;
30.                           add the edge (y,parent(y)) in the cutset;
31.                     else   sum ← sum + 1;       /* include tree node s */
32.                 m ← m − sum;
33.         else        /* w_{k+1} < 2 * (m − sum) */
34.                 put the nodes of the subtree rooted at t_{k+1} in the set X;
35.                 if  (k ≤ d_s/2)
36.                 then  add the edges (s, t_1), ..., (s, t_{k+1}) in the cutset;
37.                 else        /* k > d_s/2 */
38.                     put node s in the set X;
39.                     add the edge (s,parent(s)) in the cutset;
40.                     add the edges (s, t_{k+2}), ..., (s, t_{d_s}) in the cutset;
41.                     put one leaf, say, z, of the subtree rooted at t_1 in the set Y;
42.                     add the edge (z,parent(z)) in the cutset;
43.                 swap ← true;
44.                 m ← sum + w_{k+1} − m;
        /* prepare next iteration of while-loop */
45.     root ← t_{k+1};
46.     if  (swap)
47.     then  swap X and Y;
48.         swap ← false;
```

The correctness of the algorithm follows from the fact that m and *swap* are updated appropriately ensuring that the set A contains $\min\{m, n-m\}$ nodes eventually. Steps similar to those of the Algorithm A are carried out in the same way yielding $O(n)$ time complexity.

Observation 3.1. At every iteration of the while loop, $m \leq \text{weight}(root)/2$.
Trivially true in the first iteration. In subsequent iterations, we repeat the procedure in a tree of size $a + b$ trying to collect $\min\{a, b\}$ nodes.

An Upper Bound on the Cutsize The analysis of all possible cases in Algorithm O, in conjunction with Lemma 3.2, yields the following inequality for the cutsize $C(m)$ of the $(m, n-m)$-partition that the algorithm produces:

$$C(m) \leq \begin{cases} 1, & & (a) \\ 2, & \text{where } m \geq 2; & (b) \\ k + C(\lfloor m/(2k+1) \rfloor), & \text{where } 1 \leq k \leq d/2; & (c) \\ k, & \text{where } 1 \leq k \leq d/2; & (d) \\ d - k + 1 + C(\lfloor m/(2k+1) \rfloor), & \text{where } d/2 < k \leq d-1; & (e) \\ d - k + 2, & \text{where } d/2 < k \leq d-1 \text{ and } m \geq k; & (f) \\ k + 1 + C(\lfloor m/(2k+1) \rfloor), & \text{where } 1 \leq k \leq d/2; & (g) \\ d - k + 1 + C(\lfloor m/(2k+1) \rfloor), & \text{where } d/2 < k \leq d-1. & (h) \end{cases}$$

Case (a) results if $w(s) = m$ and may occur for any value of m (lines 8-9,14). Case (b) results if $w(s) = m + 1$ (lines 8-14); it comes up only when $m \geq 2$, as there is always a leaf in a tree. Cases (c) and (d) are due to line 23 of the algorithm, the latter corresponding to the case where $sum = m$. Similarly, for the cases (e) and (f), which correspond to the lines 25-31. Case (g) corresponds to the line 36 and case (h) to the lines 38-42 (note that in this case $sum < m$). It is important to observe that cases (e), (f) and (h) occur only if $d \geq 3$; if $d = 2$, then $k = 1$ and the tests $k \leq d/2$ of the lines 22 and 25 are true, thus leading to cases (c), (d) and (g).

Now we are ready to prove an upper bound on the cutsize.

Lemma 3.3. *Let T be a d-tree ($d \geq 4$) of size $n \geq 2$. Then, for any positive integer $m < n$, Algorithm O always finds an $(m, n-m)$-partition of the node set of T whose cutsize is at most*

$$\frac{d+2}{2} \ \log_{d+1}\Big(2 \min\{m, n-m\}\Big) \ + \frac{1}{2}.$$

Proof: We use induction on m, in a fashion similar to the one used in the proof of Theorem 3.1. If $m = 1$, then Algorithm O produces a partition with cutsize equal to 1; we therefore need to show that $1 \leq \frac{d+2}{2} \ (\log_{d+1} 2) + \frac{1}{2}$. This is equivalent to proving that $1 \leq (d+2) \ \log_{d+1} 2$, which holds given that the function $g_1(d) = (d+2) \ \log_{d+1} 2$ is strictly increasing for $d \geq 3$ and $g_1(3) = 5 \log_4 2 = 2.5 > 1$. For the inductive step, we assume that the upper bound of $C(i)$ holds for all $i = 1, \ldots, r-1$, and we will prove that it also holds for $C(r)$ where $r \geq 2$.

(a) Follows from the basis of the induction given that the function $\frac{d+2}{2} \log_{d+1} x$ is strictly increasing for $x > 0$ and $d > 0$.

(b) In this case,

$$2 \leq \frac{d+2}{2} \ \log_{d+1}(2r) + \frac{1}{2} \iff 3 \leq (d+2) \log_{d+1}(2r) \iff 1 \leq \frac{(d+2)\ln(2r)}{3\ln(d+1)}$$

which holds for all $r \geq 2$ and all $d \geq 2$; note that $(d+2)\ln(2r) \geq (d+1)\ln 3$ and the function $g_2(x) = x/\ln(x)$ is strictly increasing for all $x \geq 3$.

(c) Weaker than case (g).

(d) Follows from case (c), since $C(.)$ is a non-negative-valued function.

(e) We need to show that $d - k + 1 + C(r/(2k+1)) \leq \frac{d+2}{2} \log_{d+1}(2r) + \frac{1}{2}$, where $d/2 < k \leq d-1$. If we set $\kappa = d - k$, the inequality that needs to be proved is written as $\kappa+1+C(r/(2(d-\kappa)+1)) \leq \frac{d+2}{2} \log_{d+1}(2r)+\frac{1}{2}$, where $1 \leq \kappa < d/2$. Since $\kappa < d/2$, then $d-\kappa > r$ and therefore $r/(2(d-\kappa)+1) < r/(2\kappa+1)$; given that the function $\frac{d+2}{2} \log_{d+1} x$ is strictly increasing for $x > 0$, we have that $C(r/(2(d-\kappa)+1)) < C(r/(2\kappa+1))$. This implies that case (e) follows from case (g).

(f) We need to show that $d-k+2 \leq \frac{d+2}{2} \ \log_{d+1}(2r)+\frac{1}{2}$ where $d/2 < k \leq d-1$, $r \geq k$ and $d \geq 4$. Since the function $(d+2) \ \log_{d+1} x$ is strictly increasing for $x > 0$, it suffices to show that $d-k+2 \leq \frac{d+2}{2} \ \log_{d+1}(2k)+\frac{1}{2}$. Again, we set $\kappa = d-k$. Then, this inequality becomes $\kappa + 2 \leq \frac{d+2}{2} \ \log_{d+1}(2(d-\kappa)) + 1/2$, where $1 \leq \kappa < d/2$. Given that $\kappa < d/2$, we have $\kappa \leq (d-1)/2$; this implies firstly that $2d - 2\kappa \geq d+1 \Longrightarrow \log_{d+1}(2(d-\kappa)) \geq 1$ and secondly that $\kappa + 3/2 \leq (d+2)/2$. The combination of the last two inequalities yields the desired bound.

(g) We need to show that $k+1+C(r/(2k+1)) \leq (d+2)/2 \ \log_{d+1}(2r)+1/2$, where $1 \leq k \leq d/2$. Since $k+1+C(r/(2k+1)) \leq k+1+\frac{d+2}{2}\log_{d+1}(2r/(2k+1))+\frac{1}{2} = k+1-\frac{d+2}{2}\log_{d+1}(2k+1)+\frac{d+2}{2}\log_{d+1}(2r)+\frac{1}{2}$, we need only prove that $k+1-\frac{d+2}{2}\log_{d+1}(2k+1) \leq 0 \iff \frac{(2k+2)/\ln(2k+1)}{(d+2)\ln(d+1)} \leq 1$. The last inequality holds for $d \geq 4$ and any k such that $1 \leq k \leq d/2$, since the function $g_3(x)=(x+1)/\ln(x)$ is strictly increasing for $x \geq 4$ and $6/\ln(5) > 4/\ln(3)$.

(h) Same as case (e). ∎

Lemma 3.4. *Let T be a d-tree ($d=2,3$) of size $n \geq 2$. Then, for any positive integer $m<n$, the Algorithm O always finds an $(m,n-m)$-partition of the node set of T whose cutsize is at most*

$$2\log_3\Big(2\ \min\{m,n-m\}\Big)\ +\frac{1}{2}.$$

Proof: Let us assume again that $m=\min\{m,n-m\}$. For the specific values of d, the inequality for $C(m)$ becomes:

$$C_2(m) \leq \begin{cases} 1 & \\ 2 & (m \geq 2) \\ 1+C(\frac{m}{3}) & \\ 1 & \\ - & \\ - & \\ 2+C(\frac{m}{3}) & \\ - & \end{cases} \qquad C_3(m) \leq \begin{cases} 1 & \\ 2 & (m \geq 2) \\ 1+C(\frac{m}{3}) & \\ 1 & \\ 2+C(\frac{m}{5}) & \\ 3 & (m \geq 2) \\ 2+C(\frac{m}{3}) & \\ 2+C(\frac{m}{5}) & \end{cases}$$

It is an easy exercise to prove that both $C_2(m)$ and $C_3(m)$ do not exceed $2\log_3 2m\ +1/2$ for all values of $m \geq 1$. ∎

The combination of Lemmata 3.3 and 3.4 yields the following result regarding the cutsize of the $(m,n-m)$-partition that Algorithm O produces.

Theorem 3.2. *Let T be a d-tree ($d \geq 2$) of size $n \geq 2$. Then, for any positive integer $m<n$, Algorithm O always finds an $(m,n-m)$-partition of the node set of T whose cutsize does not exceed*

$$\begin{array}{ll} 2\ \log_3\Big(2\ \min\{m,n-m\}\Big)\ +\dfrac{1}{2} & \text{if } d=2,3 \\ \dfrac{d+2}{2}\ \log_{d+1}\Big(2\ \min\{m,n-m\}\Big)\ +\dfrac{1}{2} & \text{if } d \geq 4. \end{array}$$

4. The Lower Bound.

In [9], we proved that for $d \geq 3$ the bisection of an n-node complete d-tree admits cutsizes at least equal to $\lfloor (d-1)/2 \rfloor\ (\log_d n - \log_d \log_d n) - d/2$. We can extend this result for arbitrary partitions and we can show that:

Lemma 4.1. *Let T be a complete d-tree with $n=(d^h-1)/(d-1)$ nodes. Then, for $m_i = \lfloor (d^i-1)/(2d-2) \rfloor$ ($i=2,\ldots,\lfloor \log_d n \rfloor$), the optimal $(m_i,n-m_i)$-partition of T' nodes has cutsize that is at least equal to $\lfloor (d-1)/2 \rfloor\ (\log_d(2m_i) - \log_d \log_d(2m_i)) - d/2$.*

Proof: It can be shown that the cutsize will not increase if the m_i nodes are collected from the same complete d-subtree of height i. This subtree contains $(d^i-1)/(d-1) \simeq 2m_i$ nodes and therefore the problem reduces to optimally bisect the nodes of this subtree. The lemma follows from the above stated result for tree bisections. ∎

5. Concluding Remarks.

In this paper, we consider the problem of partitioning the node set of a tree T of n nodes and degree d ($d \geq 2$) into two sets of prespecified cardinalities. We described two linear time algorithms which produce partitions of the tree nodes into two sets V_1 and V_2 of desired sizes achieving cutsizes that do not exceed $(d+1)/2\ \log_{(d+3)/2} m + 1$ and $(d+2)/2\ \log_{d+1}(2m) + 1/2$ respectively, where $m = \min\{|V_1|, |V_2|\}$. These algorithms are worst-case optimal (in fact, yielding partitions whose cutsize is a small constant factor times the optimal solution in the worst case), as implied by the lower bound of $\lfloor(d-1)/2\rfloor\ (\log_d m - \log_d \log_d m) - d/2$ which we prove for the cutsize of such a partition when $d \geq 3$ and for all $m = \lfloor(d^i-1)/(2d-2)\rfloor$ $(i = 2, \ldots, \log_d n)$. Brief experimentation with the two algorithms suggests that the first algorithm performs better than the second, despite the fact that the upper bound on the cutsize of the partition that it produces is worse than the corresponding bound for the second algorithm.

We close this section by describing some open questions. Firstly, it is very interesting to come up with competitive algorithms for the case of partitions. The algorithms we described are worst-case optimal, i.e., in the worst case the produced cutsize is not more than a small constant factor times the optimal cutsize. They are not necessarily competitive, however, since we cannot guarantee that *at all times* the produced cutsize does not exceed a constant times the corresponding optimal cutsize. Secondly, it is of interest as well to further reduce the gap between the lower and the upper bounds for the cutsize. Thirdly, more extensive experimentation with the algorithms and variants of them could result into better understanding of the problem and could yield algorithms with even better theoretical or practical performance. Additionally, it might be interesting to work out tight lower bounds for the case of binary trees. The cutsize is of course $O(\log n)$, but it would be worth looking into the constant in front of the logarithmic term. (Brief experimentation with complete binary trees of up to 2^{21} nodes suggests that the maximum cutsize is roughly $\Theta(\log_2 n/2)$.) Last but not least, upper and lower bounds in the cutsize of optimal partitions (and bisections in particular) of planar or more general graphs are yet to be established. Such bounds will shed new light to partitioning algorithms, such as the one in [3], whose performance depends on the cutsize of the corresponding optimal partition.

5. References.

1. R. Boppana, "Eigenvalues and graph bisection: an average-case analysis," *Proc. 28th Annual IEEE Symposium on Foundation of Computer Science*, 280–285 (1987).

2. T. Bui, S. Chaudhuri, T. Leighton, and M. Sipser, "Graph bisection algorithms with good average case behavior," *Combinatorica* 7, 171–191 (1987).

3. T.N. Bui and A. Peck, "Partitioning planar graphs," *SIAM Journal on Computing* 21, 203–215 (1992).

4. K. Diks, H.N. Djidjev, O. Sykora, and I. Vrto, "Edge Separators of Planar and Outerplanar Graphs with Applications," *Journal of Algorithms* 14, 258–279 (1993).

5. M.R. Garey, D.S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical Computer Science* 1, 237–267 (1976).

6. B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Tech. Journal* 49, 291–307 (1970).

7. S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by simulated annealing," *Science* 220, 671–680 (1983).

8. R.M. MacGregor, *"On partitioning a graph: A theoretical and empirical study,"* Ph.D. Thesis, University of California, Berkeley (1978).

9. L. Palios, *"Upper and Lower Bounds for Optimal Tree Partitions,"* Research Report GCG47, The Geometry Center, University of Minnesota.