Clustering in Trees: Optimizing Cluster Sizes and Number of Subtrees

Susanne E. Hambrusch *Chuan-Ming Liu*

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
http://www.cs.purdue.edu
seh@cs.purdue.edu liucm@cs.purdue.edu

Hyeong-Seok Lim

Chonnam National University
Kwangju, 500-757, Korea
hslim@chonnam.chonnam.ac.kr

Abstract

This paper considers partitioning the vertices of an n-vertex tree into p disjoint sets $C_1, C_2, \ldots, C_p$, called clusters so that the number of vertices in a cluster and the number of subtrees in a cluster are minimized. For this NP-hard problem we present greedy heuristics which differ in (i) how subtrees are identified (using either a best-fit, good-fit, or first-fit selection criteria), (ii) whether clusters are filled one at a time or simultaneously, and (iii) how much cluster sizes can differ from the ideal size of c vertices per cluster, $n = cp$. The last criteria is controlled by a constant α, $0 \leq \alpha < 1$, such that cluster C_i satisfies $(1 - \frac{\alpha}{2})c \leq |C_i| \leq c(1+\alpha)$, $1 \leq i \leq p$. For algorithms resulting from combinations of these criteria we develop worst-case bounds on the number of subtrees in a cluster in terms of c, α, and the maximum degree of a vertex. We present experimental results which give insight into how parameters c, α, and the maximum degree of a vertex impact the number of subtrees and the cluster sizes.

---

1. Hambrusch's research supported in part by the National Science Foundation under Grant 9988339-CCR.

2. Lim's research supported in part by Korea Science and Engineering Foundation under Contract No. 98-0102-07-01-3.

1 Introduction

Tree clustering partitions the vertices of a given tree into disjoint sets, called clusters, subject to optimizing one or more objective functions. Tree clustering arises in parallel and distributed computing environments and external memory systems. For a tree representing an external search structure, the created clusters correspond to the blocks. Clusters should minimize the number of blocks as well as the access to external storage devices [1, 4, 7, 12]. For a tree representing data flow and communication requirements in a parallel and distributed environment, partitioning the vertices corresponds to assigning tasks to processors. The goal is to balance processor loads and to minimize communication between processors [6, 10, 11]. Not surprisingly, the combinatorial nature of clustering problems makes finding optimal solutions computationally intractable for most realistic situations [4, 5, 7, 14].

Let T be a tree with $n = cp$ vertices, $c \geq 2$. We assume that edges and vertices have no associated weights. A clustering of T partitions the vertices into p sets, $C_1, C_2, \ldots, C_p$. We consider generating clusters when the number of vertices assigned to different clusters should be as equal as possible and the number of subtrees assigned to every cluster should be minimized. While minimizing these two cost measures simultaneously captures desirable features for the above applications, it is an NP-hard problem.

An ideal load is achieved when every cluster contains c vertices. This corresponds to every block containing c data items and every processor assigned c tasks, respectively. Achieving an ideal load is straightforward in the absence of weights[1]. Our second cost measure is the number of subtrees in a cluster. For parallel and distributed applications, minimizing the number of subtrees enhances locality and decreases communication. When generating blocks for external tree structures, load and blocknumber are often optimized [4, 8, 12, 13]. The blocknumber measures the number of blocks needed during a search from the root to a leaf in the tree. Minimizing the blocknumber and achieving ideal load is NP-hard [7]. Existing heuristics first assign to every block a single subtree and then achieve a better load by partitioning selected subtrees [7, 8, 13]. This approach can assign many subtrees to a block and result in high I/O. Our approach is to minimize the number of subtrees and the load simultaneously. We refer to [9] for a more detailed discussion on the relationship between the blocknumber and the number of subtrees.

Achieving an ideal load and minimizing the maximum number of subtrees in the clusters is NP-hard [9]. We note that deciding whether there exists a clustering having an ideal load and every cluster containing one subtree can be done in linear time. However, deciding whether there exist clusters of size c with every cluster containing at most 3 subtrees is already NP-complete. An ideal load is desirable, but generating clusters of size of c is not always necessary.

In this paper we introduce the concept of α-clustering to capture such a tolerated slackness in cluster sizes. Given a tree T with $n = cp$ vertices and

[1] The existence of weights on the vertices results in an NP-hard problem, as clustering becomes a bin-packing like problem.

a parameter α, $0 \leq \alpha < 1$, an α-clustering generates p clusters so that every cluster C_i satisfies $(1-\frac{\alpha}{2})c \leq |C_i| \leq c(1+\alpha)$, $1 \leq i \leq p$. For $\alpha = 0$, we generate an exact clustering; i.e., $|C_i| = c$. The clustering algorithms presented are greedy heuristics. They differ in (i) the identification of subtrees (i.e., whether a best-fit, good-fit, and first-fit selection criteria is used), (ii) the order in which clusters are filled (i.e., whether clusters are filled one at a time or simultaneously), and (iii) different values of α which control how much cluster sizes are allowed to differ from the ideal size of c vertices per cluster. Our work provides insight into how cluster sizes and number of subtrees in a cluster are impacted by the value of α, the maximum degree d in the tree, the relationship between c and d, the subtree selection method, as well as the order in which clusters are filled. We develop worst-case upper bounds on the number of subtrees and the cluster sizes and provide experimental results supporting our claims.

The paper is organized as follows. In Section 2 we describe the ingredients of our clustering algorithms and prove that the cluster forming approaches generate cluster sizes in the required range. Section 3 presents the two single fill clustering algorithms along with asymptotic bounds on the number of subtrees in a cluster. Section 4 discusses the simultaneous fill algorithms. The experimental performance of the algorithms is discussed in Section 5.

2 Overview of the Clustering Algorithms

In this section we discuss the framework underlying our α-clustering algorithms. Figure 1 gives time and number of subtrees bounds for four α-clustering algorithms presented in this paper. Throughout, d is the maximum degree of a vertex in T.

The quantities $\log_{\frac{d-2}{d-1}} \frac{\alpha}{2}$ and $\log_{\frac{d-1}{d}} \frac{\alpha}{4}$ should be read as $\min\{c, \log_{\frac{d-2}{d-1}} \frac{\alpha}{2}\}$ and $\min\{c, \log_{\frac{d-1}{d}} \frac{\alpha}{4}\}$, respectively. Note that when $\alpha = 0$, the stated minima generate c. Figure 2 shows these two quantities (independent of c) for the range of degrees considered in this paper. Observe that the upper bounds can exceed the trivial bound of at most c vertices in a cluster.

2.1 Single versus Simultaneous Cluster Forming

Our algorithms assign subtrees to clusters in either a single fill or a simultaneous fill mode. Algorithms based on the *single fill mode* determine the subtrees for cluster C_i before generating cluster C_{i+1}. Algorithms based on a *simultaneous fill mode* assign subtrees to clusters without this restriction. Symultaneous fill algorithms may assign one subtree to each cluster in one iteration or use current cluster sizes to decide which cluster receives the next subtree. When $\alpha > 0$, single fill as well as simultaneous fill need to ensure that cluster sizes are within the required bounds. For example, if too many clusters are underfull (i.e., have $|C_i| < c$), the remaining vertices of T may force a cluster to exceed the upper bound. Figure 3 gives the outline of a generic single fill algorithm. The quantity $remain_i$ represents the total number of vertices to be made up due to underfull

Algorithm	Time	Maximum number of subtrees
SingFill-BF	$\Theta(np)$	$\lceil \log_{\frac{d-2}{d-1}} \frac{\alpha}{2} \rceil$
SingFill-FF	$\Theta(n)$	$\min\{c, d * \lceil \frac{\log c}{\log d} \rceil\}$
SimulFill-BF	$O(np \log_{\frac{d-1}{d}} \frac{\alpha}{4})$	$\lceil \log_{\frac{d-1}{d}} \frac{\alpha}{4} \rceil$
SimulFill-GF	$O(n \log_{\frac{d-1}{d}} \frac{\alpha}{4})$	$\lceil \log_{\frac{d-1}{d}} \frac{\alpha}{4} \rceil$

Figure 1: Bounds achieved by our clustering algorithms.

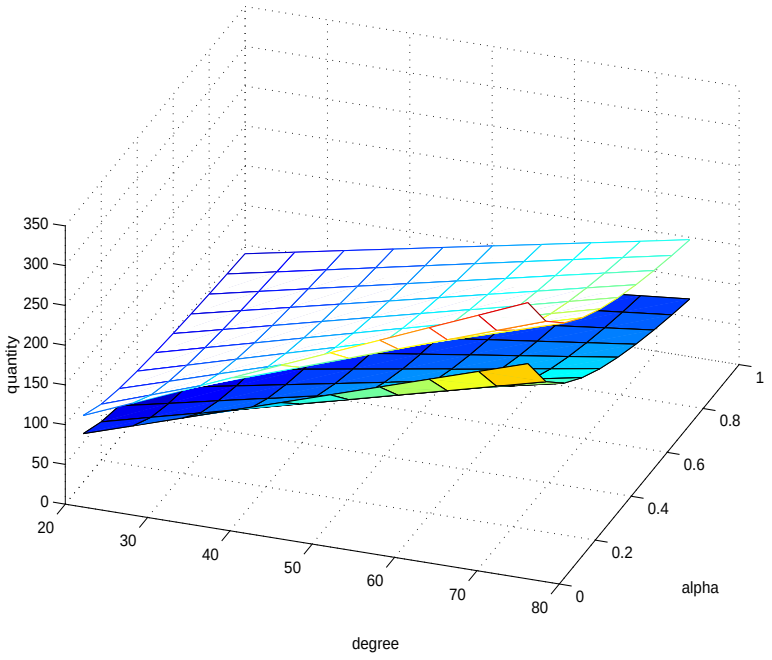

Figure 2: Comparing the quantities of $\log_{\frac{d-2}{d-1}} \frac{\alpha}{2}$ (filled grid) and $\log_{\frac{d-1}{d}} \frac{\alpha}{4}$ (non-filled grid) for different degrees.

clusters. Lemma 1 shows that $c + remain_i$ never exceeds the upper bound on the cluster size.

Algorithm Generic-SingFill
Input: tree $T = (V, E)$, $n = cp$, and parameter α
Output: $C_1, C_2, \ldots, C_p$ representing the p clusters of an α-clustering

1. Initialize each cluster as an empty set.
2. $remain_0 = 0$
3. **for** $i = 1$ **to** $p - 1$ **do**
   $target_i = c + remain_{i-1}$
   $remain_i = target_i$
   while $(|C_i| < (1 - \frac{\alpha}{2}) \times target_i)$ **do**
   (a) Determine a subtree $T' = (V', E')$ with $|V'| \leq remain_i$ using one of the subtree finding methods
   (b) Update: $T = T - T'$; $C_i = C_i \cup V'$
   $remain_i = remain_i - |V'|$
   endwhile
   endfor
4. $C_p = V$

Figure 3: Description of Algorithm Generic-SingFill.

The different ways of determining subtrees are described in Section 2.2. The following lemma shows that Algorithm Generic-SingFill generates cluster sizes which fall within the range needed for the α-clustering. The number of subtrees in a cluster depends on how subtrees are selected and bounds will be given when individual algorithms are described.

Lemma 1 *Cluster C_i generated by Algorithm Generic-SingFill satisfies* $(1 - \frac{\alpha}{2})c \leq |C_i| \leq c(1 + \alpha)$, $1 \leq i \leq p$.

Proof: Consider first the $p - 1$ clusters generated within the while-loop. Since $target_i \geq c$ and the algorithm terminates with $|C_i| \geq (1 - \frac{\alpha}{2}) \times target_i$, $1 \leq i \leq p - 1$, the lower bound on the cluster size is satisfied for the first $p - 1$ clusters. The upper bound of $|C_i| \leq c(1 + \alpha)$ is shown as follows. At the end of the first iteration we have $remain_1 \leq \frac{\alpha}{2}c$. Hence, $target_2 \leq c + \frac{\alpha}{2}c$ and $remain_2 \leq \frac{\alpha}{2}c + (\frac{\alpha}{2})^2 c$ at the end of the second iteration. In general,

$$target_i \leq c + remain_{i-1} \text{ and}$$

$$remain_i \leq \frac{\alpha}{2} \times target_i.$$

Hence, $target_i \leq c + \frac{\alpha}{2} \times target_{i-1}$ and

$$target_i \leq c \sum_{k=0}^{i-1} (\frac{\alpha}{2})^k < \frac{2}{2-\alpha} \times c.$$

For $0 < \alpha < 1$, we have $\frac{2}{2-\alpha} < 1 + \alpha$. Thus, $target_i < c(1+\alpha)$ and the upper bound on the cluster size holds for the first $p-1$ clusters.

Cluster C_p is assigned the remaining vertices of tree T. Since $\sum_{i=1}^{p-1} |C_i| + remain_{p-1} = (p-1)c$, we have $|C_p| = c + remain_{p-1}$. Since $remain_{p-1} \leq \frac{\alpha}{2} \times target_{p-1}$ and $target_{p-1} < \frac{2c}{2-\alpha}$, we have $remain_{p-1} \leq \frac{\alpha}{2-\alpha} \times c$. Hence, $c \leq |C_p| \leq c + \frac{\alpha}{2-\alpha} \times c \leq c(1+\alpha)$. □

Algorithm Generic SimulFill
Input: tree $T = (V, E)$, $n = cp$, and parameter α
Output: $C_1, C_2, \ldots, C_p$ representing the p clusters of an α-clustering

Initialize $C_i = \emptyset$ and $remain_i = c$, $1 \leq i \leq p$.
PHASE 1: Generate p safe clusters.
while there exists a cluster which is not safe **do**
for $i = 1$ **to** p **do**
if cluster C_i is not safe **then**

1. Determine the next subtree $T' = (V', E')$ with $|V'| \leq remain_i$ using one of the subtree finding methods.
2. Update: $T = T - T'$; $C_i = C_i \cup V'$
   $remain_i = remain_i - |V'|$

endfor
endwhile

PHASE 2: Assign the remaining vertices of T.
Update $remain$-entries: $remain_i = \alpha c + remain_i$, $1 \leq i \leq p$.
while tree T is not empty **do**
for $i = 1$ **to** p **do**
if tree T not empty and cluster C_i not full **then**

1. Determine the next subtree $T' = (V', E')$ with $|V'| \leq remain_i$ using one of the subtree finding methods.
2. Update: $T = T - T'$; $C_i = C_i \cup V'$
   $remain_i = remain_i - |V'|$

endfor
endwhile

Figure 4: Description of Algorithm Generic-SimulFill.

We now turn to the simultaneous filling of clusters. As for single fill, we need to ensure that deficits in cluster sizes can be made up by other clusters without exceeding the upper bound of $(1+\alpha)c$. Our clustering algorithms based on the simultaneous fill mode create the clusters in two phases, as evident from the outline given in Figure 4. We say cluster C_i is *safe* if $(1-\frac{\alpha}{2})c \leq |C_i| \leq c$. In Phase 1, we generate p safe clusters. The number of iterations executed in Phase 1 equals the maximum number of subtrees assigned to a safe cluster. After Phase 1, every cluster size lies within the required range. However, not all vertices of the tree may have been assigned to clusters yet.

Phase 2 assigns the remaining vertices of tree T to the safe clusters. We say cluster C_i is *full* if $|C_i| \geq (1+\frac{\alpha}{2})c$. Once a cluster becomes full, no more assignments are made to it. The while-loop is executed until all vertices of T have been assigned to a cluster. A cluster may thus not receive any additional vertices in Phase 2. In particular, when $\alpha = 0$, all vertices of T are assigned to clusters in Phase 1.

From the way Algorithm Generic-SimulFill forms clusters it is clear that the number of vertices assigned to a cluster lies in the required range determined by α. The number of subtrees assigned to a cluster depends on how subtrees are identified and bounds on the number of subtrees are developed in Section 4.

We conclude this section with a brief comparison of the two cluster filling modes. The advantage of the single-fill mode is that at the time cluster C_i is filled, the final sizes of the first $i-1$ clusters are known. A single-fill algorithm fills cluster C_i using α and information on how underfull previous clusters are. A single-fill algorithm tries to make up an earlier created deficit as soon as possible. The advantage of the simultaneous-fill mode is that during its first few iterations, every cluster has a chance to find subtrees in a large tree. This can lead to Phase 1 generating safe clusters consisting of few trees in each cluster. As will be discussed in Section 5.2, these characteristics show up in the experimental results. At the same time, corresponding disadvantages show up as well. For example, the final clusters created by a single-fill algorithm select subtrees from a relatively small tree. Since the number of subtree choices is now limited, these final clusters can end up being assigned a large number of subtrees.

2.2 Identifying Subtrees

In this section we sketch the three methods used by the clustering algorithms for identifying subtrees. Assume we are to determine the next subtree for cluster C_i. Let $remain_i$ be the maximum number of vertices that can still be assigned to C_i (without exceeding the upper bound on the cluster size of C_i).

Suppose we remove an edge $e = (u, v)$ in T. Then, T is divided into two subtrees. Let $T_{e,u} = (V_{e,u}, E_{e,u})$ (resp. $T_{e,v} = (V_{e,v}, E_{e,v})$) be the subtree containing vertex u (resp. v), but not edge e. Recall that d is the maximum degree of a vertex. The subtree $T' = (V', E')$ of T is found using one of the following:

- *Best-Fit:* Determine an edge $e = (u, v)$ and vertex u such that $|V_{e,u}| \leq remain_i$ and $|V_{e,u}|$ is a maximum. Set $T' = T_{e,u}$.
- *Good-Fit*: Choose the first tree T' encountered in the traversal of T with $\lceil remain_i/d \rceil \leq |V'| \leq remain_i$.
- *First-Fit:* Choose the first tree T' encountered in the traversal of T with $|V'| \leq remain_i$.

The different tree selection methods result in algorithms with different running times. Clustering algorithms using best-fit selection traverse, in the worst case, the entire tree T to find one subtree T'. For clustering algorithms based on good-fit and best-fit the running time depends on whether single-fill or simultaneous-fill is used. For single-fill, our implementations perform one tree traversal when forming one cluster. For simultaneous-fill, one traversal of the tree identifies p subtrees, one for every cluster. We refer to Figure 1 for running times and upper bounds on the number of subtrees in a cluster. A major focus of our experimental work is whether the use of the best-fit subtree selection results in significantly better clusters and thus justifies the increase in time.

3 Single Fill Clustering

We now present two single clustering algorithms, Algorithm SingFill-BF based on best-fit and Algorithm SingFill-FF based on first-fit subtree selection. Algorithm SingFill-BF creates one cluster by performing one traversal of the tree, and thus achieves a $\Theta(np)$ running time. Algorithm SingFill-FF determines all clusters during a single traversal of the tree, and thus has an $\Theta(n)$ running time. We do not consider good-fit subtree selection for single fill clusterings. Good-fit subtree selection can be implemented to achieve $O(np)$ time, as does best-fit (which determines better fitting subtrees). The good-fit strategy is used in the simultaneous fill algorithms described in Section 4.

3.1 Algorithm SingFill-BF

Algorithm SingFill-BF corresponds to the generic single fill algorithm described in Figure 3 with the best-fit subtree selection. We describe an $O(np)$ time implementation and then show that the number of subtrees in a cluster is bounded by $\min\{c, \lceil \log_{\frac{d-2}{d-1}} \frac{\alpha}{2} \rceil\}$.

A straightforward $O(np \log_{\frac{d-2}{d-1}} \frac{\alpha}{2})$ time bound is obtained by searching the current tree for the next subtree giving the best fit. The implementation described below determines the subtrees for one cluster in $O(n)$ time by using a queue to efficiently locate the subtrees giving the best fit.

Consider the beginning of the i-th iteration. Tree T now corresponds to the original tree from which the vertices assigned to clusters $C_1, \ldots, C_{i-1}$ have been removed. Before entering the while-loop of iteration i, we determine for all edges $e = (u, v)$ in tree T the quantities $|V_{e,u}|$ and $|V_{e,v}|$. A priority queue

Q in the form of an array of size $target_i$ is used to represent selected subtree entries. Subtree $T_{e,u} = (V_{e,u}, E_{e,u})$ is an entry in queue Q at index $|V_{e,u}|$ if the following two conditions hold:

1. $|V_{e,u}| \leq remain_i$ and
2. for every edge $e' = (u', v)$ with $u' \neq u$ we have $|V_{e',v}| > remain_i$.

Condition (1) selects for queue Q only those subtrees that "fit" (i.e., they do not exceed the remaining capacity). Condition (2) selects, among all subtrees that fit, the ones that are as large as possible. Using standard tree computations and traversals, queue Q can be set up in $O(n)$ time.

Step 3(a) of SingFill-BF determines the next best fitting subtree by scanning array Q starting at position $remain_i$. The subtree is found by scanning left, looking for the first non-empty entry in Q. Let $T' = T_{e,u}$ be the subtree chosen. Before $remain_i$ is decreased in Step 3(b), we update array Q. The entry representing subtree $T_{e,u}$ is deleted. Before the next subtree is selected, we "break up" subtrees which are now too large while satisfying conditions (1) and (2). Entries corresponding to subtrees larger than $remain_i - |V_{e,u}|$ are no longer needed. To record appropriate subtrees of these trees, we proceed as follows. Scan array Q from the position which contained $T_{e,u}$ to the left to position $remain_i - |V_{e,u}|$. Let $T_{b,x}$ be a subtree encountered during this scan, $b = (x, y)$. The entry corresponding to $T_{b,x}$ is deleted and every vertex adjacent to x (excluding y) is considered. Let w be such an adjacent neighbor. If $|V_{(w,x),w}| \leq remain_i - |V_{e,u}|$, condition (1) is satisfied. Observe that we do not need to check whether condition 2 is satisfied: since it was satisfied for tree $T_{e,u}$, it is also satisfied for $T_{(w,x),w}$. We thus insert $T_{(w,x),w}$ into Q. On the other hand, if condition (1) does not hold for subtree $T_{(w,x),w}$ (i.e., $|V_{(w,x),w}| > remain_i - |V_{e,u}|$), the vertices adjacent to w (excluding x) are considered for insertion. This process continues until subtrees of small enough size are found. During the entire while-loop of Step 3, an edge is considered at most a constant number of times. Thus the maintenance of array Q costs $O(n)$ time. The $O(np)$ overall time follows.

The correctness of the above approach relies on the subtrees represented in queue Q being disjoint. The existence of disjoint subtrees when creating clusters $C_1, \ldots, C_{p-2}$ is guaranteed since we have $n - |V_{e,u}| > 2c$ for every subtree in Q. For iteration $p-1$, subtrees represented in Q may not be disjoint. In our implementation, iteration $p-1$ does thus not use the queue, but it explicitly traverses the remaining tree for finding best fitting, disjoint subtrees. This does not impact the $O(np)$ overall time.

We now turn to bounding the number of subtrees in a cluster. The first lemma relates the size of subtree T' to $remain_i$.

Lemma 2 *Assume edge $e = (u, v)$ and vertex u are selected in Step 3(a) of the i-th iteration of Algorithm SingFill. Then, $|V_{e,u}| \geq \frac{remain_i}{d-1}$.*

Proof: Assume this is not true and let $T_{e,u}$ be a best fitting subtree satisfying $|V_{e,u}| < \frac{remain_i}{d-1}$. For any edge $e' = (u', v)$ incident to vertex v, we either have

- $|V_{e',u'}| \leq |V_{e,u}| < \frac{remain_i}{d-1}$ (i.e., subtree $T_{e',u'}$ could be chosen, but does not give a better fit), or
- $|V_{e',u'}| > remain_i$ (i.e., subtree $T_{e',u'}$ is too large).

There must exist at least one vertex u' with $|V_{e',u'}| > remain_i$. (To be precise, there must exist at least two such vertices.) Otherwise $|V_{e',u'}| < \frac{remain_i}{d-1}$ would hold for every vertex u' adjacent to v and thus for subtree $T_{e',v}$ we would have $|V_{e',v}| < \frac{remain_i}{d-1} \times (d-1) < remain_i$. This would contradict that $T_{e,u}$ is a best fitting subtree.

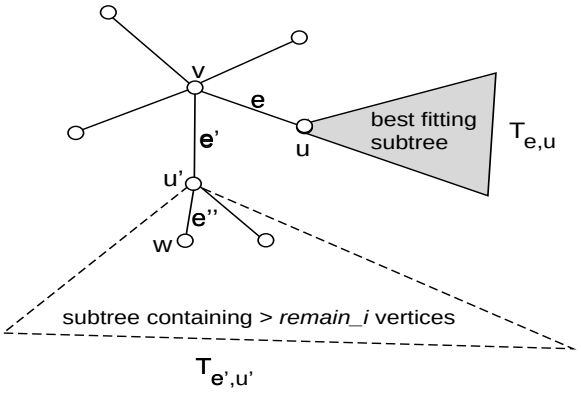

Figure 5: Illustrating the position of edges e, e', and e''.

We arrive at a contradiction for the assumption $|V_{e,u}| < \frac{remain_i}{d-1}$ by considering a subtree in $T_{e',u'}$ with $|V_{e',u'}| > remain_i$. Vertex u' is incident to at least one edge $e'' = (u', w)$ with $|V_{e'',w}| \geq \frac{remain_i}{d-1}$. This situation is illustrated in Figure 5. The case $|V_{e'',w}| \leq remain_i$ would imply that the subtree rooted at w is a better fit than $T_{e,u}$ and give a contradiction. If $|V_{e'',w}| \geq remain_i$, we apply the same argument using edge e'' in the role of e'. A subsequent step leads to a contradiction. Hence, $|V_{e,u}| \geq \frac{remain_i}{d-1}$. □

Lemma 3 *The number of subtrees assigned to a cluster by Algorithm SingFill-BF is at most* $\min\{c, \lceil \log_{\frac{d-2}{d-1}} \frac{\alpha}{2} \rceil\})$.

Proof: Let $t(i,j)$ be the minimum size of the subtree selected at the j-th step of the i-th iteration of the while-loop. We set $t(i,0) = target_i$. From Lemma 2 it follows that $t(i,1) = \frac{t(i,0)}{d-1}$ and $t(i,2) = \frac{t(i,0)-t(i,1)}{d-1} = t(i,0)\frac{d-2}{(d-1)^2}$. The j-th step of the while loop removes a subtree of size $t(i,j) = t(i,0)\frac{(d-2)^{j-1}}{(d-1)^j}$. The total number of vertices in cluster C_i after m steps of the while loop is thus

$$t(i,0)\sum_{j=1}^{m}\frac{(d-2)^{(j-1)}}{(d-1)^j} = \left(1 - \left(\frac{d-2}{d-1}\right)^{m-1}\right) * target_i.$$

The while loop terminates when $(1 - (\frac{d-2}{d-1})^m) \times target_i > (1 - \frac{\alpha}{2}) \times target_i$. This implies that the number of subtrees assigned to cluster C_i is bounded by $\log_{\frac{d-2}{d-1}} \frac{\alpha}{2} = \frac{1-\log \alpha}{\log(d-1)-\log(d-2)}$. Since no cluster contains more than c vertices, the claimed bound follows. □

The following theorem summarizes our discussion:

Theorem 4 *Algorithm SingFill-BF determines an α-clustering for an n-vertex tree T in time $\Theta(np)$, $n = cp$. The number of subtrees assigned to a cluster is bounded by* $\min\{c, \lceil \log_{\frac{d-2}{d-1}} \frac{\alpha}{2} \rceil\})$.

3.2 Algorithm SingFill-FF

In this section we describe Algorithm SingFill-FF, a single fill clustering algorithm using first-fit subtree selection. We describe the algorithm for the case $\alpha = 0$. Its generalization to arbitrary values of α's uses *target* and *remain*-entries as described in Algorithm Generic-SingFill in Figure 3.

Algorithm SingFill-FF uses the results of a weighted postorder numbering on a rooted version of tree T to form the clusters. Let r be an arbitrary vertex of T chosen as the root. With T rooted towards r, we determine the *weighted postorder number* of every vertex as follows. Let u be a vertex with children $v_1, v_2, \ldots, v_k$. The children are arranged by non-increasing sizes of subtrees; i.e., $|V_{(v_i,u),v_i}| \geq |V_{(v_{i+1},u),v_{i+1}}|$ for every i, $1 \leq i < k$. With the children ordered this way, perform a postorder traversal of T. Let $post(u)$ be the postorder number assigned to vertex u. Then, vertex u belongs to cluster $C_{\lceil post(u)/c \rceil}$. Figure 6 shows clusters C_1 and C_2 for the sketched tree. Ordering the children of all vertices by size can be done in $O(n)$ time. One implementation uses the fact that subtree sizes are bounded by n and thus all sizes can be indexed into an array of size n, allowing an $O(n)$ time rearranging. The assignment of vertices to clusters based on the weighted postorder traversal number can thus be done in $O(n)$ time. In the remainder of this section we show that the number of subtrees in a cluster is bounded by $\min\{c, d * \lceil \frac{\log c}{\log d} \rceil\}$.

W.l.o.g. assume the formation of cluster C_i starts at vertex u and only vertices in the subtree rooted at u are in cluster C_i. If this is not the case, the vertices in C_i having smaller postorder numbers form one subtree. For illustration, consider vertex a in Figure 6. Cluster C_2 contains vertices in the subtree rooted at a and the vertices not in this subtree form one tree as indicated. We ignore this one subtree when counting subtrees. Let $v_1, v_2, \ldots, v_k$, $k \leq d$, be the children of u. Assume cluster C_i receives the subtrees rooted at $v_1, \ldots, v_{l_1-1}$ and some of the vertices in the subtree rooted at v_{l_1}, $l_1 \geq 2$. The number of vertices needed from the subtree rooted at v_{l_1} is at most c/l_1. If more vertices were needed, the use of the weighted postorder numbering (i.e., $|V_{(u,v_j),v_j}| \geq |V_{(u,v_{j+1}),v_{j+1}}|$ and $|V_{(u,v_j),v_j}| > c/l_1$, $1 \leq j \leq l_1 - 1$) would imply that C_i contains more than c vertices.

To show the claimed bound on the number of subtrees in C_i we first show that after the inclusion of $d-1$ subtrees into cluster C_i, the cluster misses at most c/d vertices. In other words, the first $c - c/d$ vertices selected by

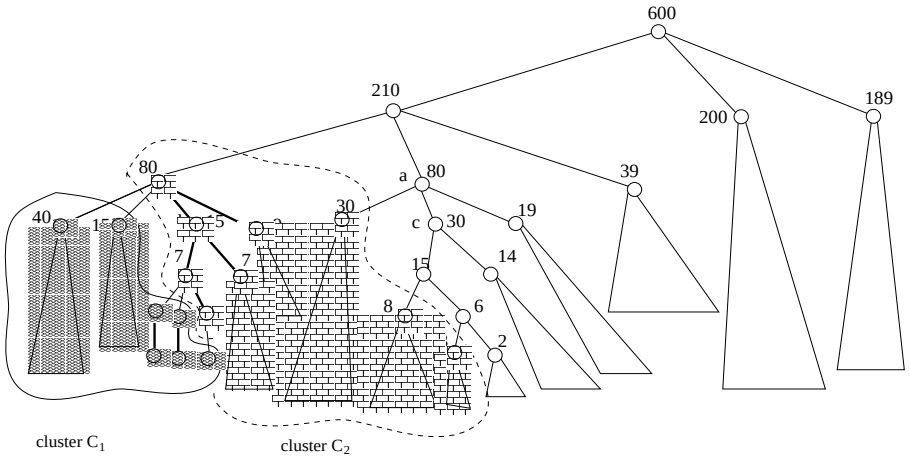

Figure 6: Forming exact clusters using weighted postorder numbers. The tree has $n = 600$, $c = 60$, $d = 10$; integers next to vertices represent the number of vertices in the subtree.

the algorithm induce at most $d-1$ subtrees. Observe that "the first $c - c/d$ vertices" refers to the $c - c/d$ vertices in C_i and in the subtree rooted at u with the smallest postorder numbers. We then apply the same argument to the at most c/d remaining vertices. This results in at most $\min\{c, \lceil \frac{\log c}{\log d} \rceil\}$ iterations, each iteration contributing at most $d-1$ subtrees.

The subtrees rooted at $v_1, \ldots, v_{l_1-1}$ represent $l_1 - 1$ subtrees in C_i. To avoid conflict in notation, rename $v_{l_1} = u_{l_1}$. The algorithm then continues including vertices from the subtree rooted at u_{l_1}. At vertex $u_{l_{j-1}}$, we include subtrees rooted at children of $u_{l_{j-1}}$ and identify at most one subtree rooted at child u_{l_j} which contains more vertices than needed. More specifically,

- u_{l_j}'s left siblings are roots of subtrees included into C_i and
- not all vertices in the subtree rooted u_{l_j} are needed for C_i.

Assume the process of including subtrees and identifying subtrees of size larger than needed considers vertices $u_{l_1}, u_{l_2}, \ldots, u_{l_t}$. See Figure 7 for an illustration. Observe that we assume $l_j \geq 2$. If for a vertex $u_{l_{j-1}}$ the subtree rooted at its leftmost child contains more vertices than needed, vertex $u_{l_{j-1}}$ does not appear in this enumeration. For example, for the tree shown in Figure 6, vertex a would appear in the enumeration, but vertex c would not.

As already stated, the maximum number of vertices needed for cluster C_i from the subtree rooted at u_{l_1} is $\frac{c}{l_1}$. Using the same argument, the number of vertices needed for cluster C_i from the subtree rooted at u_{l_j} is at most $\frac{c}{l_1 l_2 \ldots l_j}$. We stop the process of including subtrees into cluster C_i at vertex u_{l_j} when the actual number of vertices needed from the subtree rooted at u_{l_j} is smaller than

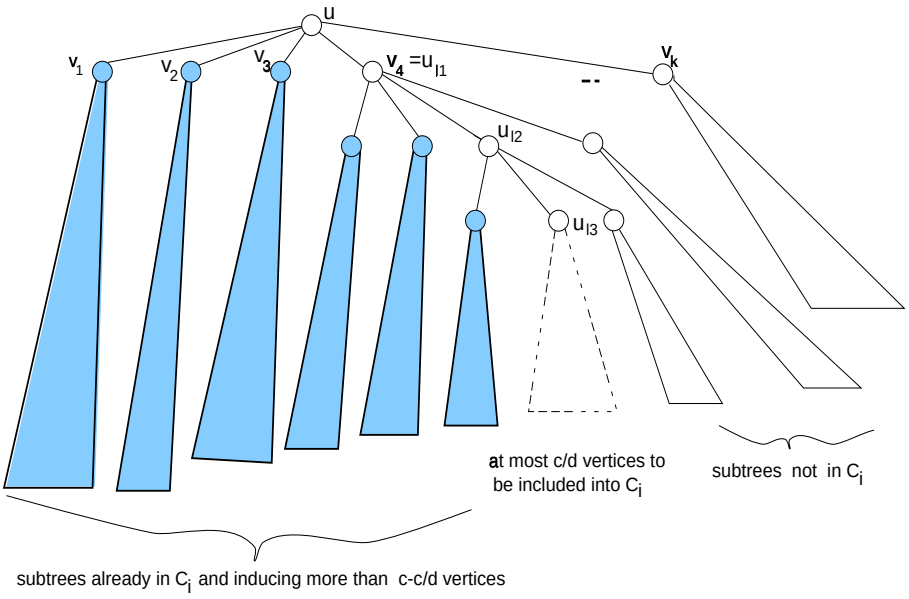

Figure 7: Illustrating vertices $u_{l_1}, u_{l_2}, \ldots, u_{l_t}$ and the subtrees in cluster C_i for $l_1 = 4$, $l_2 = 3$, $l_3 = 2$, and $t = 3$.

c/d for the first time. For cluster C_1 in the tree shown in Figure 6, the first iteration of this process stops at vertex b when C_1 already contains 55 vertices. Only 5 more vertices are needed and $5 < 6 = c/d$. It follows that

$$\frac{c}{l_1 l_2 \ldots l_t} \geq \frac{c}{d}$$

and $l_1 l_2 \ldots l_t \leq d$. Cluster C_i contains already $l_1 + l_2 + \ldots + l_t - t$ subtrees and we have $l_j \geq 2$, $1 \leq j \leq t$. The number of subtrees already in C_i (i.e., $\sum_{j=1}^{t}(l_j - 1)$) is maximized and $l_1 l_2 \ldots l_t \leq d$ is satisfied for $t = 1$ and $l_1 = d$. Hence, the first $c - c/d$ vertices in cluster C_i induce at most $d - 1$ subtrees.

This above argument is repeated for the subtree with root u_{l_t}. The goal is to include the remaining (i.e., at most c/d) vertices into cluster C_i. The next $c/d - c/d^2$ vertices assigned to cluster C_i induce at most $d - 1$ subtrees. After δ applications of the argument, $\frac{c}{d^\delta}$ vertices remain to be assigned to cluster C_i. This implies that $c \geq d^\delta$ and $\delta \leq \frac{\log c}{\log d}$.

The total number of subtrees assigned to cluster C_i is thus at most $\min\{c, d * \lceil \frac{\log c}{\log d} \rceil\}$. This bound on the number of subtrees also holds for $\alpha > 0$. We conclude this section with the following theorem.

Theorem 5 *Algorithm SingFill-FF determines an α-clustering for a given n-vertex tree T in time $\Theta(n)$. The number of subtrees assigned to a cluster is bounded by* $\min\{c, d * \lceil \frac{\log c}{\log d} \rceil\}$.

4 Simultaneous Fill Clustering

In this section we describe our clustering algorithms based on simultaneous cluster filling. To turn Algorithm Generic-SimulFill described in Figure 4 into a complete algorithm, we need to specify the subtree selection and the order in which clusters are considered. Algorithm SimulFill-GF uses the good-fit subtree selection and has $O(n \log_{\frac{d-1}{d}} \frac{\alpha}{4})$ running time. Algorithm SimulFill-BF uses best-fit subtree selection and achieves $O(np \log_{\frac{d-1}{d}} \frac{\alpha}{4})$ time. First-fit subtree selection can be implemnetd to achieve the same performance as SimulFill-GF. Since good-fit determines better fitting subtrees, we do not consider first-fit for simultaneous fill algorithms.

We first present Algorithm SimulFill-GF which considers clusters by non-decreasing *remain*-entries. This order is crucial for achieving the claimed time bound. Since the *remain* entries are between 0 and c, sorting the *remain*-entries costs $O(n)$ time per iteration. Recall that SimulFill-GF selects subtrees T' which satisfy $\lceil remain_i/d \rceil \leq |V'| \leq remain_i$. Let r be an arbitrary vertex of T chosen as the root. The algorithm first roots tree T at r. Next, it determines for every vertex v the number of vertices in the subtree rooted at v. Let $s(v)$ be this quantity. Rooting the tree and the computation of the $s(v)$-entries can be done in $O(n)$ time [3].

One **for**-loop in Phases 1 or 2 makes one traversal of the current tree and assigns one subtree to every cluster (if the cluster still qualifies for receiving vertices). Phase 1 executes iterations until every cluster is safe and the number of iterations equals the maximum number of subtrees assigned to a safe cluster.

Assume the last step determined a subtree for cluster C_i. Assume vertex v has children $u_1, u_2, \ldots, u_k$ and cluster C_i is assigned the subtree rooted at vertex u_l, $1 \leq l \leq k$. Hence, $remain_i/d \leq s(u_l) \leq remain_i$ and for $1 \leq j < l$ we have $s(u_j) < remain_i/d$ (i.e., the subtree rooted at u_j is too small for C_i). After the subtree rooted at u_l has been assigned to cluster C_i, a subtree for cluster C_{i+1} is determined. Note that we have $remain_i \leq remain_{i+1}$. The next paragraph sketches how the next subtree for C_{i+1} is found. The $O(n)$ time bound for one iteration follows from the way the tree traversal identifies subtrees.

In order to determine the subtree to be assigned to cluster C_{i+1}, the traversal first considers the remaining children of vertex v, namely vertices $u_{l+1}, \ldots, u_k$. Observe that since the subtrees rooted at $u_1, u_2, \ldots, u_{l-1}$ were not large enough for cluster C_i, they are also not large enough for C_{i+1} (since clusters are considered by non-decreasing *remain*-entries). If there exists a vertex u_j with $s(u_j) \geq remain_{i+1}/d$, $l+1 \leq j \leq k$, a subtree for C_{i+1} is found in a tree rooted at one of the siblings of u_l. The traversal considers thus vertices not yet traversed in the current iteration. Assume that for all vertices u_j, $l+1 \leq j \leq k$, we have $s(u_j) < remain_{i+1}/d$. The traversal now backs up the tree and considers vertex v next. For vertex v we maintain the number of vertices in its subtree which have already been assigned to clusters in the current iteration. If the current subtree rooted at vertex v does satisfy the size requirements for C_{i+1}, an assignment is made. Observe that the subtree rooted at vertex v cannot be

too large (since all children have subtrees which are too small). If the subtree rooted at v is too small, we continue with the parent of v, say v'. We repeat the same process of first considering the children of v' not previously considered and, if no suitable subtree is found, we consider the subtree rooted at v'. Again, we know that the children of v' considered earlier are not the roots of a big enough subtree and thus do not need to be checked. It follows that each cluster receives a subtree while executing one traversal of the tree and thus one iteration takes $O(n)$ time.

Phase 2 proceeds with the subtree selection and the ordering of the clusters as Phase 1. The while-loop is executed until all vertices of T have been assigned to a cluster. A cluster may thus not receive any additional vertices in Phase 2. The total time spent in Phases 1 and 2 is $O(n)$ times the maximum number of subtrees assigned to a cluster. The bound on the number of subtrees is given in the proof of the following theorem.

Theorem 6 *Algorithm SimulFill-GF determines an α-clustering for a tree T of n vertices in time $O(n \log_{\frac{d-1}{d}} \frac{\alpha}{4})$. The number of subtrees assigned to a cluster is bounded by* $\min\{c, \lceil \log_{\frac{d-1}{d}} \frac{\alpha}{4} \rceil\}$.

Proof: From the conditions Phase 1 and 2 impose on the cluster sizes it follows that $(1-\frac{\alpha}{2})c \leq |C_i| \leq (1+\alpha)c$, $1 \leq i \leq p$. The number of iterations within each phase gives an upper bound on the number of subtrees assigned to a cluster. Using an argument similar to that used in Lemma 2, it follows that the algorithm can always find a subtree T' such that $|V_{T'}| \geq remain_i/d$. (Since the tree is rooted and not all subtrees are considered in a rooted tree, the bound is $|V_{T'}| \geq remain_i/d$ instead of $|V_{T'}| \geq remain_i/(d-1)$.) Assume cluster C_i is safe after m iterations of Phase 1. We have $target_i = c$ and, using the argument in the proof of Lemma 3, we have

$$(1 - (\frac{d-1}{d})^m) \times c > (1 - \frac{\alpha}{2}) \times c.$$

This implies that the number of iterations in Phase 1 bounded by $\lceil \log_{\frac{d-1}{d}} \frac{\alpha}{2} \rceil$. In Phase 2, we have $target_i = (1+\alpha)c - |C_i|$ with $\alpha c \leq target_i \leq \frac{3}{2}\alpha c$. Assume cluster C_i is full after m iterations of Phase 2. Then,

$$(1 - (\frac{d-1}{d})^m) \times \alpha c > \frac{\alpha}{2} \times c.$$

Hence, the number of iterations of Phase 2 is bounded by $\lceil \log_{\frac{d-1}{d}} \frac{1}{2} \rceil$ and the claimed bound on the total number of iterations follows. □

Algorithm SimulFill-BF uses best-fit selection for determining the subtrees. Determining a subtree may result in a complete traversal of the current tree. Our implementation considers the clusters by non-increasing *remain*-entries. Even though this ordering does not impact the worst-case bounds, the approach of looking for large subtrees in large trees tends to produce better experimental results.

Theorem 7 *Algorithm SimulFill-BF determines an α-clustering for a given n-vertex tree T in time $O(np\log_{\frac{d-1}{d}}\frac{\alpha}{4})$. The number of subtrees is a cluster in bounded by* $\min\{c, \lceil\log_{\frac{d-1}{d}}\frac{\alpha}{4}\rceil\}$.

Proof: Using best-fit for the subtree selection results in a new traversal whenever a subtree is assigned to a cluster. This increases time to the stated bound. The bound on the number of subtrees is as in SimulFill-GF. □

5 Experimental Results

In this section we discuss the performance of the different clustering algorithms and show how parameters α, c, and d impact cluster sizes and number of subtrees in the clusters. We considered synthetically generated trees with n ranging from $1,000$ to $6,000$. Ideal cluster sizes considered varied from $c = 10$ to $c = 500$ and the maximum degree varied from $d = 20$ to $d = 74$. We used four classes of synthetic trees. All trees were created level by level and the classes differ on how the degree of a vertex is determined. Class 1 assumes that every degree between 1 and d is equally likely for every vertex. Class 2 assumes that the probability of a vertex being a leaf is significantly higher (we used 0.5 instead $1/d$) and that, once a vertex is identified as a non-leaf, every degree is equally likely. Class 3 generates degrees using a normal distribution. Trees in classes 1 to 3 are generated level-by-level starting with the root. Class 4 generates B-trees [2]. Trees in class 4 are created by specifying number of leaves and the value of B (which corresponds to the maximum degree). Trees are then generated from the leaves towards the root and for a non-root, interior vertex, every degree between $B/2$ and B is equally likely.

For all trees, we report the mean, median, and the maximum of number of subtrees in a cluster and the cluster sizes. When we report, for example, the median number of subtrees in a cluster for a tree, we report the mean of the medians of the p clusters over 10 different trees within the same class. The different classes of trees exhibit the same performance trend for trees with the same n, c, d, and α values. As is discussed in the next two sections, we observed that the choice of α and the relationship between c and d has significant impact on the performance. The plots shown in this paper are for trees in classes 1 and 4.

Given the NP-completeness of the problem and the considered tree sizes, we did not generate optimal results. Comparing the algorithms gives interesting and relevant insight into the different strategies as well as the parameter choices. The implementation of the clustering algorithms was done in Java. The implementations have no hidden constants and are based on the same data structures. We do not report actual running times and expect the running times to follow the asymptotic worst-case bounds established.

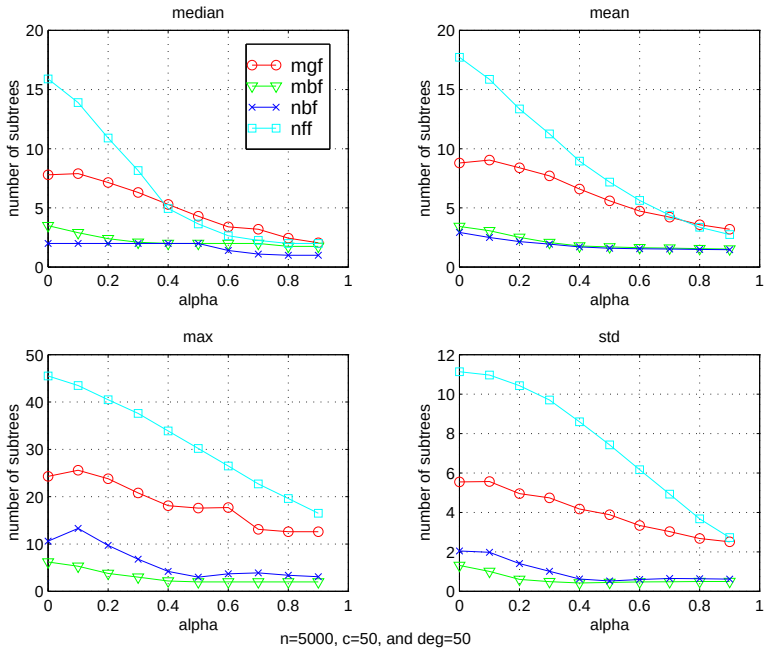

Figure 8: Comparing the number of subtrees for SimulFill-GF $\circ$, SingFill-FF $\square$, SimulFill-BF $\triangleleft$, SingFill-BF $\times$ for trees in class 1.

5.1 Comparing clustering algorithms

In this section we discuss the performance of Algorithms SingFill-FF, SimulFill-GF, SingFill-BF, and SimulFill-BF with respect to the number of subtrees and cluster sizes for synthetically generated trees belonging to classes 1 and 4. The graphs show results for ten α-values, $\alpha = j/10$, $j \in \{0, 1, 2, \ldots, 9\}$. Graphs for trees in class 1 (i.e., trees in which every degree is equally likely) have $n = 5,000$, $c = 50$, and a maximum degree of 50. Graphs for trees in class 4 (i.e., B-trees) have $5,000$ leaves.

Figures 8 and 9 show results for the number of subtrees. The graphs show trends which were observed for all trees classes and cluster sizes considered. Algorithm SingFill-FF generates clusters containing the largest number of subtrees. This holds when we consider the median, mean, and maximum number of subtrees (over all clusters and over 10 trees of the same type). This is not surprising since SingFill-FF simply arranges subtrees by size and proceeds greedily without further optimizations. Algorithms SimulFill-BF and SingFill-BF consistently outperform the two algorithms based on first-fit and good-fit with respect to minimizing the number of subtrees in the clusters. The relationship between the two best-fit approaches is examined in detail in the next section.

For all four clustering algorithms Figures 8 and 9 show a "leveling off" in the number of subtrees as α increases. Overall, our experimental work suggests that

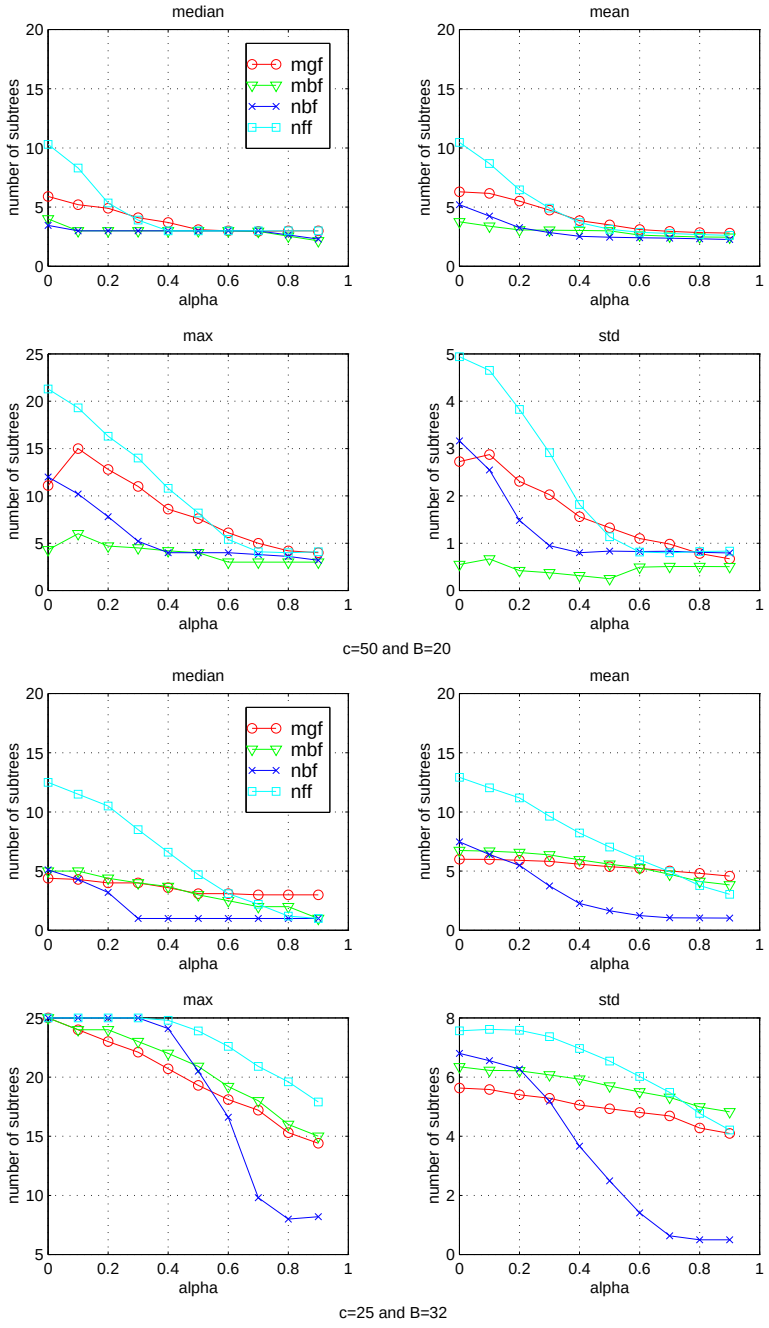

Figure 9: Comparing the number of subtrees for SimulFill-GF ∘, SingFill-FF □, SimulFill-BF ◁, SingFill-BF × for B-trees with 5,000 leaves; upper four graphs have $B = 20$ and $c = 50$; lower four have $B = 32$ and $c = 25$.

using α greater than 0.4 has little impact on reducing the number of subtrees. We next comment on a relationship between c and d we observed throughout. As the maximum degree exceeds c and a tree contains a significant number of nodes with high degree, it becomes harder - and at times impossible - to keep the number of subtrees in a cluster small. For trees in class 1 with $c = 50$ and values of d smaller than 50, we observed a decrease in the number of subtrees for all algorithms. For degrees larger than 50, we observed an increase. However, the relationship between the four algorithms remains stable. Figure 9 shows this for two types of B-trees. The upper four graphs show a typical behavior for the case when the maximum degree is smaller than c. In the lower four graphs the maximum degree exceeds c (32 versus 25). All algorithms produce clusters with more subtrees. In particular, the maximum number of subtrees is equal (or close to) to the possible worst case of c vertices in a cluster for small values of α. Plots in Section 5.2 which vary c or d will also reflect this characteristic.

We use trees in class 1 to illustrate observed behavior on the cluster sizes as α increases. Figure 10 shows the median, mean, and maximum difference between achieved and ideal cluster size (i.e., the quantities $|C_i - c|$). Clearly, as α increases, the differences in the cluster sizes continue to increase. Algorithm SimulFill-GF fills the clusters closer to the limits set by α than any other algorithm. In Figure 10, the maximum cluster sizes generated by the two simultaneous fill algorithms are consistently higher compared to those of the single fill algorithms. This is a characteristic of the simultaneous filling of clusters. Recall that the simultaneous filling of clusters proceeds in two phases: the first phase generates safe clusters and it does not allow a cluster size to exceed c. Clusters exceeding size c are generated in the second phase. This approach ensures correct cluster sizes, but it also makes it more likely that there exist clusters which are close to the extremes of the required range.

5.2 Comparing SingFill-BF and SimulFill-BF

We now turn to comparing the two best-fit clustering algorithms, SimulFill-BF and SingFill-BF. All graphs shown in this section were obtained using trees in class 1, but are representative for all types of trees we considered. Figure 11 shows typical results for the number of subtrees when α ranges from 0 to 0.9 and c ranges from 10 to 500. In the trees used, the maximum degree is 44. For small c values (up to around $c = d$), we observe a large number of subtrees for both best-fit algorithms. Note the significant increase in the maximum number of subtrees (and the different scale used). For larger c values, we see the number of subtrees decrease as α increases and we observe a leveling off around $\alpha = 0.4$ with respect the mean, median, and maximum number of subtrees in the clusters. Figure 12 also illustrates the observed leveling off for the mean number of subtrees and Algorithm SingFill-BF.

From our experimental results we can conclude that SingFill-BF outperforms SimulFill-BF with respect to the mean and the median number of subtrees. When considering the maximum number of subtrees, we see that SimulFill-BF outperforms SingFill-BF. This behavior showed up in all trees we considered

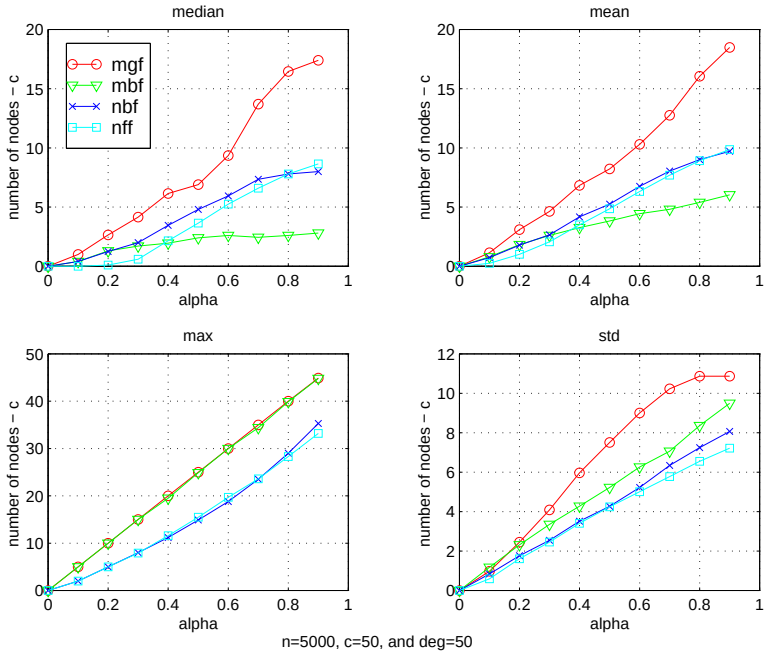

Figure 10: Comparing cluster sizes for SimulFill-GF ∘, SingFill-FF □, SimulFill-BF ◁, SingFill-BF × for trees in class 1.

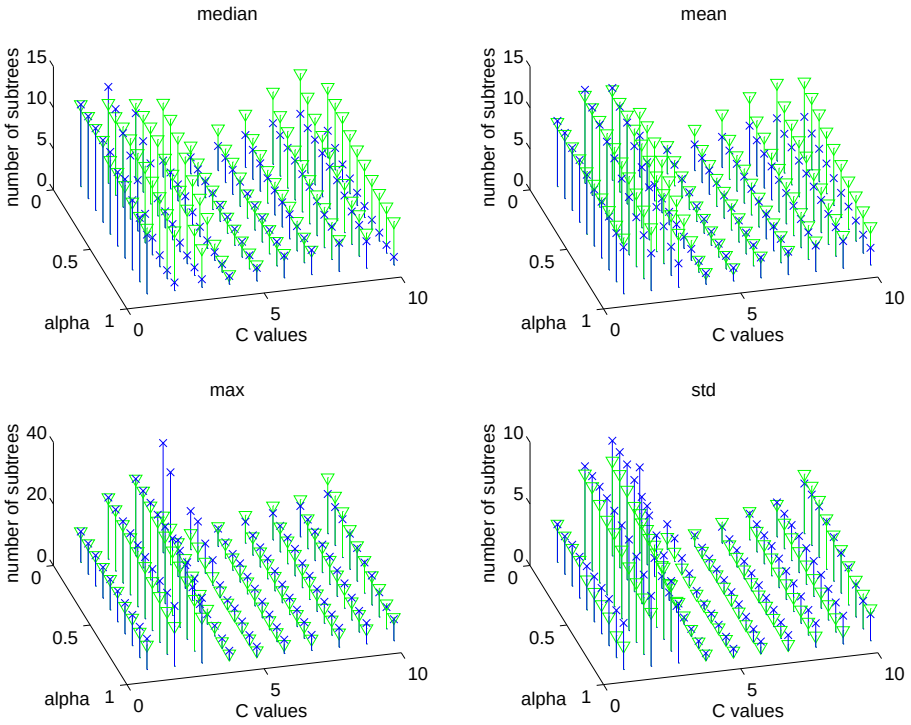

Figure 11: Comparing the number of subtrees for SimulFill-BF $\bigtriangledown$ and SingFill-BF $\times$ when c and α change; $n = 5,000$ and $d = 44$; c values are [10, 20, 25, 40, 50, 100, 125, 200, 250, 500].

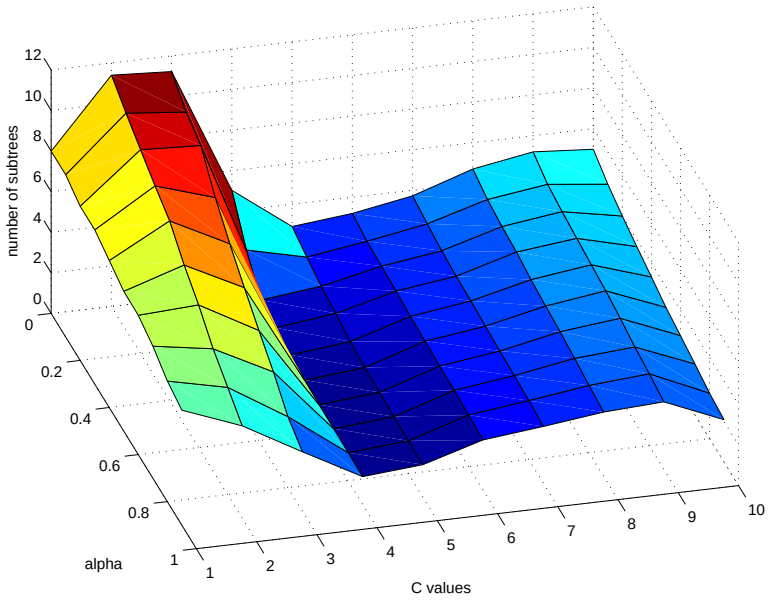

Figure 12: Mean number of subtrees in clusters for Algorithm SingFill-BF as α and c change for $n = 5,000$ and $d = 44$; c values are [10, 20, 25, 40, 50, 100, 125, 200, 250, 500].

and reflects a characteristic between the two approaches. Algorithm SingFill-BF is able to delay creating clusters containing a large number of subtrees until the remaining tree is small. The final iterations of SingFill-BF generate clusters with a larger number of subtrees compared to what SimulFill-BF generates. This happens since the small tree remaining allows fewer choices, creating thus a large maximum for SingFill-BF.

The final set of experimental results examines the impact of the maximum degree on the performance of the two best-fit algorithms. We show data obtained for $n = 5,000$, $c = 50$, and d in the range from 20 to 74. In Figure 13 we again see that Algorithm SimulFill-BF outperforms SingFill-BF with respect to the maximum number of subtrees placed in a cluster, but that SingFill-BF gives better results for the mean and median values. For the large degrees ($d = 62, 68$, and 74 in Figure 13), we observed a significantly larger number of subtrees for the mean, median, as well as the maximum. This confirms the relationship of c and d discussed earlier.

Figure 14 illustrates cluster sizes for $c = 50$ as maximum degrees and α increase. As to be expected, increasing α generates for both algorithms clusters whose sizes vary more and more from the ideal size of c. SimulFill-BF generates clusters much closer to their upper and lower limits, as was already mentioned in the discussion in single versus simultaneous fill for Figure 10. Using Figures 13 and 14 and $d = 26$ for SimulFill-BF, we see a maximum of 5 subtrees in the

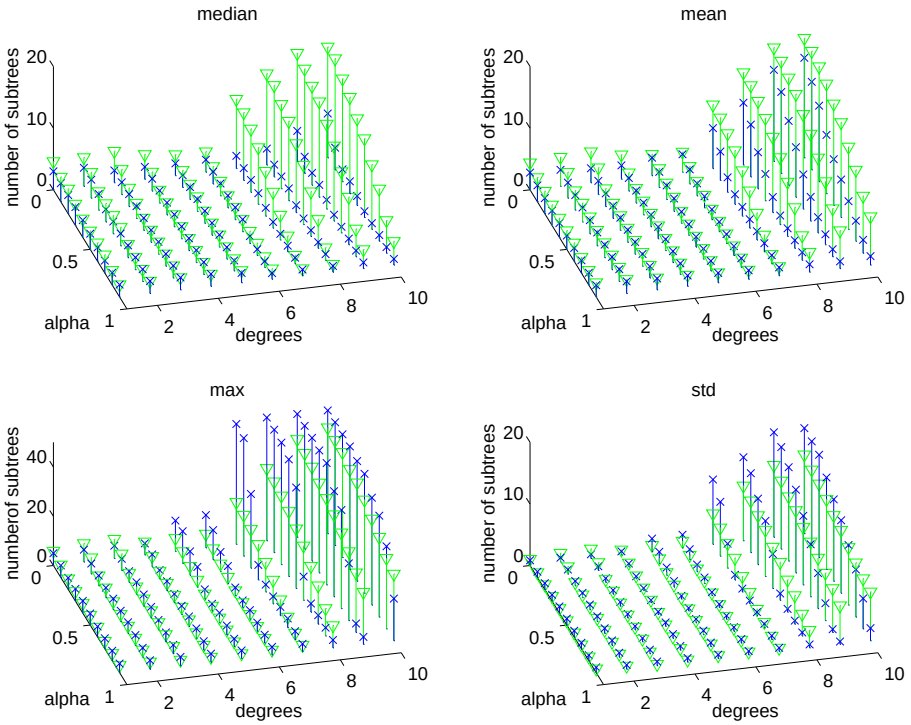

Figure 13: Comparing the number of subtrees for SimulFill-BF $\bigtriangledown$ and SingFill-BF $\times$ when the maximum degree d and α change; $n = 5,000$ and $c = 50$; d values are $[20, 26, 32, 38, 44, 50, 56, 62, 68, 74]$.

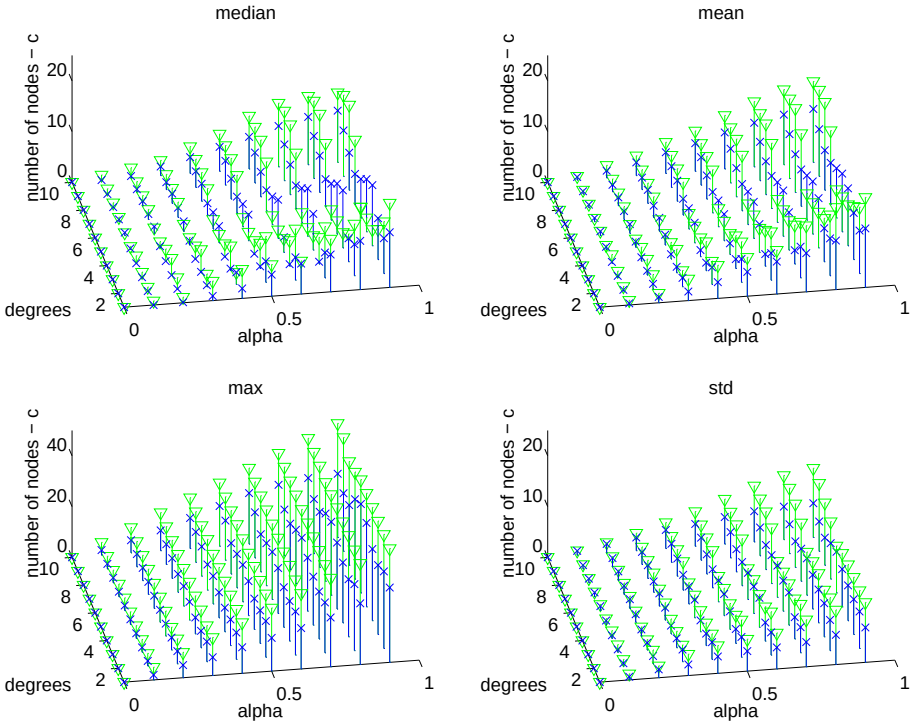

Figure 14: Comparing cluster sizes for SimulFill-BF $\bigtriangledown$ and SingFill-BF $\times$ when d and α change; $n = 5,000$ and $c = 50$; d values are $[20, 26, 32, 38, 44, 50, 56, 62, 68, 74]$.

clusters for $\alpha = 0.5$ and a maximum of 5 subtrees for $\alpha = 0.8$. In both cases, there exists clusters which are filled to the upper limit of 75 and 90 vertices in a cluster, respectively. While increasing α beyond 0.4 tends not to reduce the number of subtrees, it does generate clusters sizes lying in larger ranges.

6 Conclusion

We presented algorithms for α-clustering the vertices of a tree when cluster sizes need to lie in a range defined by α and the number of subtrees assigned to a cluster should be minimized. In addition to input parameter α, the algorithms differ in the identification of subtrees and the order in which clusters are filled. We described efficient implementation of the clustering algorithms and established upper bounds on the number of subtrees in a cluster. Our experimental results provided insight into how the maximum degree d, the relationship between c and d, the value of α, the subtree selection method, and the order in which clusters are filled impact the number of subtrees and the cluster sizes. In particular, our experimental result show that as α increases, the reduction in the

number of subtrees slows down considerably, but the differences between cluster sizes continues to increase. Overall, we observed that the best-fit clustering algorithm filling one cluster at a time generates consistently good results.

7 Acknowledgments

We thank the referees for their helpful and constructive comments.

References

[1] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Transactions on Software Engineering (SE)*, 14(11), Nov. 1988.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

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

[4] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of the 22-nd International Conference on Very Large Data Bases*, pages 342–353, 1996.

[5] A. Farley, S. Hedetniemi, and A. Proskurowski. Partitioning trees: matching, domination, and maximum diameter. *International Journal of Computer and Information Sciences*, 10(1):55–61, Feb. 1981.

[6] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[7] J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32, 1999.

[8] S. Hambrusch and C.-M. Liu. Data replication for external searching in static tree structures. In *Proceedings of the Ninth ACM International Conference on Information and Knowledge Management*, Nov. 2000.

[9] C.-M. Liu. Searching in static, external memory data structures. Ph.d. thesis, Purdue University, Department of Computer Sciences, in progress.

[10] P. Maheshwari and H. Shen. An efficient clustering algorithm for partitioning parallel program. *Parallel Computing*, 24(5-6):893–909, June 1998.

[11] D. M. Nicol and D. R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, Mar. 1991.

[12] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, Aug. 1996.

[13] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proceedings of the Thirteenth ACM Symposium on Principles of Database Systems*, volume 13, pages 25–35, 1994.

[14] R. Schrader. Approximations to clustering and subgraph problems on trees. *Discrete Applied Mathematics*, 6:301–309, 1983.