K-ary Clustering with Optimal Leaf Ordering for Gene Expression Data

Ziv Bar-Joseph,[*] Erik D. Demaine, David K. Gifford, Nathan Srebro
Laboratory for Computer Science, MIT, 200 Technology Square, Cambridge, MA 02139, USA
{zivbj,edemaine,gifford,nati}@mit.edu

Angèle M. Hamel
Wilfrid Laurier University, Dept. of Physics and Computing, Waterloo, On, N2L 3C5, Canada
ahamel@wlu.ca

Tommi S. Jaakkola
MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA 02139, USA
tommi@ai.mit.edu

Abstract

Motivation: A major challenge in gene expression analysis is effective data organization and visualization. One of the most popular tools for this task is hierarchical clustering. Hierarchical clustering allows a user to view relationships in scales ranging from single genes to large sets of genes, while at the same time providing a global view of the expression data. However, hierarchical clustering is very sensitive to noise, it usually lacks of a method to actually identify distinct clusters, and produces a large number of possible leaf orderings of the hierarchical clustering tree. In this paper we propose a new hierarchical clustering algorithm which reduces susceptibility to noise, permits up to k siblings to be directly related, and provides a single optimal order for the resulting tree.

Results: We present an algorithm that efficiently constructs a k-ary tree, where each node can have up to k children, and then optimally orders the leaves of that tree. By combining k clusters at each step our algorithm becomes more robust against noise and missing values. By optimally ordering the leaves of the resulting tree we maintain the pairwise relationships that appear in the original method, without sacrificing the robustness.

Our k-ary construction algorithm runs in $O(n^3)$ regardless of k and our ordering algorithm runs in $O(4^k n^3)$. We present several examples that show that our k-ary clustering algorithm achieves results that are superior to the binary tree results in both global presentation and cluster identification.

Availability: We have implemented the above algorithms in C++ on the Linux operating system. Source code is available upon request from zivbj@mit.edu.

[*] To whom correspondence should be addressed

1 Introduction

Hierarchical clustering is one of the most popular methods for clustering gene expression data. Hierarchical clustering assembles input elements into a single tree, and subtrees represent different clusters. Thus, using hierarchical clustering one can analyze and visualize relationships in scales that range from large groups (clusters) to single genes. However, hierarchical clustering is very sensitive to noise, since in a typical implementation if two clusters (or genes) are combined they cannot be separated even if farther evidence suggests otherwise [12]. In addition, hierarchical clustering does not specify the clusters, making it hard to distinguish between internal nodes that are roots of a cluster and nodes which only hold subsets of a cluster. Finally, the ordering of the leaves, which plays an important role in analyzing and visualizing hierarchical clustering results, is not defined by the algorithm. Thus, for a binary tree, any one of the 2^{n-1} orderings is a possible outcome.

In this paper we propose a new hierarchical clustering algorithm which reduces susceptibility to noise, permits up to k siblings to be directly related, and provides a single optimal order for the resulting tree, without sacrificing the tree structure, or the pairwise relationships between neighboring genes and clusters in the result. Our solution replaces the binary tree of the hierarchical clustering algorithm with a k-ary tree. A k-ary tree is a tree in which each internal node has at most k children. When grouping k clusters (or genes) together, we require that all the clusters that are grouped together will be similar to one another. It has been shown (e.g. in CLICK [12]) that relying on similarities among large groups of genes helps reduce the noise effects that are inherent in expression data. Our algorithm utilizes this idea for the hierarchical case. In our case, we are interested in groups of genes that are similar, where the notion of similarity depends on the scale we are looking at.

The number of children of each internal node is not fixed to k. Rather, k is an upper bound on this number, and if the data suggests otherwise this number can be reduced. An advantage of such a method is that it allows us to highlight some of the actual clusters since nodes with less than k children represent a set of genes that are similar, yet significantly different from the rest of the genes. Such a distinction is not available when using a binary tree.

Finally, our algorithm re-orders the resulting tree so that an optimally ordered tree is presented. This ordering maximizes the sum of the similarity of adjacent leaves in the tree, allowing us to obtain the best pairwise relationships between genes and clusters, even when $k > 2$.

The running time of our algorithm (for small values of k) is $O(n^3)$, which is similar to the running time of currently used hierarchical clustering algorithms. Our ordering algorithm runs in $O(n^3)$ for binary trees, while for $k > 2$ our algorithm runs in $O(4^k n^3)$ time and $O(kn^2)$ space which is feasible even for a large n (when k is small).

The rest of the paper is organized as follows. In Section 2 we present an algorithm for constructing k-ary trees from gene expression data. In Section 3 we present the new $O(n^3)$ optimal leaf ordering algorithm for binary trees, and its extension to ordering k-ary trees. In Section 4 we present our experimental results, and Section 5 summarizes the paper and suggests directions for future work.

1.1 Related work

The application of hierarchical clustering to gene expression data was first discussed by Eisen [7]. Hierarchical clustering has become the tool of choice for many biologists, and it has been used to both analyze and present gene expression data [7, 14, 10]. A number of different clustering algorithms, which are more global in nature, where suggested and applied to gene expression data. Examples of such algorithms are K-means, Self organizing maps [15] and the graph based algorithms Click [13] and CAST [3]. These algorithms generate clusters which are all assumed to be on the same level, thus they lack the ability to represent the relationships between genes and sub clusters on different scales as hierarchical clustering does. In addition, they are usually less suitable for large scale visualization tasks, since they do not generate a global ordering of the input data. In this work we try to combine the robustness of these clustering algorithms with the presentation and flexible groupings capabilities of hierarchical clustering.

Recently, Segal and Koller [11] suggested a probabilistic hierarchical clustering algorithm, to address the robustness problem. Their algorithm assumes a specific model for gene expression data. In contrast, our algorithm does not assume any model for its input data, and works with any similarity/distance measure. In addition, in this paper we present a method that allows not only to generate the clusters but also to view the relationships between different clusters, by optimally ordering the resulting tree.

The problem of ordering the leaves of a binary hierarchical clustering tree dates back to 1972 [9]. Due to the large number of applications that construct trees for analyzing datasets, over the years, many different heuristics have been suggested for solving this problem (cf. [9, 5, 8, 7]). These heuristics either use a 'local' method, where decisions are made based on local observations, or a 'global' method, where an external criteria is used to order the leaves. For the local methods, decisions made in an early stage of the ordering are irreversible, and thus can lead to less than optimal order in the following steps. For the global methods, fitting an external criterion that was generated in a different way is infeasible in most cases, since there are $n!$ possible global orderings, while only 2^{n-1} of them are consistent with the tree.

In order to overcome these drawbacks, Bar-Joseph *et al* [2] presented an $O(n^4)$ algorithm that maximizes the sum of the similarities of adjacent elements in the orderings for binary trees. This algorithm maximizes a global function with respect to the tree ordering, thus achieving both good local ordering and global ordering. In this paper we present an improved algorithm for this task, that has a running time of $O(n^3)$. Such an improvement reduces the running time from hours to just a few minutes on large datasets, making the algorithm much more attractive since most software packages (including Cluster) perform hierarchical clustering in $O(n^3)$. A second extension is performed in order to use the algorithm for k-ary trees instead of binary trees.

The problem of ordering the leaves of a binary hierarchical clustering tree dates back to 1972 [9]. Due to the large number of applications that construct trees for analyzing datasets, over the years, many different heuristics have been suggested for solving this problem (c.f. [9, 8, 7]). These heuristics either use a 'local' method, where decisions are made based on local observations, or

a 'global' method, where an external criteria is used to order the leaves. In [2] Bar-Joseph $et\ al$ presented an $O(n^4)$ algorithm that maximizes the sum of the similarities of adjacent elements in the orderings for binary trees. This algorithm maximizes a global function with respect to the tree ordering, thus achieving both good local ordering and global ordering. In this paper we extend and improve this algorithm by constructing a time and space efficient algorithm for ordering k-ary trees.

Recently it has come to our attention that the optimal leaf ordering problem was also addressed by Burkard $et\ al$ [4]. In that paper the authors present an $O(2^k n^3)$ time, $O(2^k n^2)$ space algorithm for optimal leaf ordering of PQ-trees. For binary trees, their algorithm is essentially identical to the basic algorithm we present in section 3.2, except that we propose a number of heuristic improvements. Although these do not affect the asymptotic running time, we experimentally observed that they reduce the running time by 50-90%. For k-trees, the algorithms differ in their search strategies over the children of a node. Burkard *et al.* suggest a dynamic programming approach which is more computationally efficient ($O(2^k n^3)$ vs. $O(4^k n^3)$), while we propose a divide and conquer approach which is more space-efficient ($O(kn^2)$ vs. $O(2^k n^2)$). The number of genes (n) in an expression data set is typically very large, making the memory requirements very important. In our experience, the lower space requirement, despite the price in running time, enables using larger ks.

2 Constructing K-ary Trees

In this section we present an algorithm for constructing k-ary trees. We first formalize the k-ary tree problem, and show that finding an optimal solution is hard (under standard complexity assumptions). We present a heuristic algorithm for constructing k-ary trees for a fixed k, and extend this algorithm to allow for nodes with at most k children.

2.1 Problem statement

As is the case in hierarchical clustering, we assume that we are given a gene similarity matrix S, which is initially of dimensions n by n. Unlike binary tree clustering, we are interested in joining together groups of size k, where $k > 2$. In this paper we focus on the average linkage method, for which the problem can be formalized as follows. Given n clusters denote by C the set of all subsets of n of size k. Our goal is to find a subset $b \in C$ s.t. $V(b) = max\{V(b')|b' \in C\}$ where V is defined in the following way:

$$V(b) = \sum_{i,j \in b, i<j} S(i,j) \qquad (1)$$

That is, $V(b)$ is the sum of the pairwise similarities in b. After finding b, we merge all the clusters in b to one cluster. The revised similarity matrix is computed in the following way. Denote by i a cluster which is not a part of b, and let the cluster formed by the merging the clusters of b be denoted by j. For a cluster m, let $|m|$ denote the number of genes in m, then:

$$S(i,j) = \frac{\sum_{m \in b} |m| S(m,i)}{\sum_{m \in b} |m|}$$

which is similar to the way the similarity matrix is updated in the binary case. This process is repeated $(n-1)/(k-1)$ times until we arrive at a single root cluster, and the tree is obtained.

Finding b in each step is the most expensive part of the above problem, as we show in the next lemma. In this lemma we use the notion of $W[1]$ hardness. Under reasonable assumptions, a $W[1]$ hard problem is assumed to be $fixed\ parameter$ intractable, i.e. the dependence on k cannot be separated from the dependence on n (see [6] for more details).

Lemma 2.1 *Denote by $MaxSim(k)$ the problem of finding the first b set for a given k. Then $MaxSim$ is NP-hard for arbitrary k, and $W[1]$ hard in terms of k.*

Proof outline: We reduce MAX-CLIQUE to $MaxSim(k)$ by constructing a similarity matrix S_G and setting $S_G(i,j) = 1$ iff there is an edge between i and j in G. Since MAX-CLIQUE is NP and $W[1]$ complete, $MaxSim(k)$ is NP and $W[1]$ hard.

2.2 A heuristic algorithm for constructing k-ary trees

As shown in the previous section, any optimal solution for the k-ary tree construction problem might be prohibitive even for small values of k, since n is very large. In this section we present a heuristic algorithm, which has a running time of $O(n^3)$ for any k, and reduces to the standard average linkage clustering algorithm when $k = 2$. The algorithm is presented in Figure 1 and works in the following way. Starting with a set of n clusters (initially each gene is assigned to a different cluster), we generate L_i which is a linked list of clusters for each cluster i. The clusters on L_i are ordered by their similarity to i in descending order. For each cluster i we compute $V(b_i)$ where b_i consists of i and the clusters that appear in the first $k-1$ places on L_i, and V is the function described in equation 1. Next, we find $b = argmax_i\{V(b_i)\}$, the set of clusters that have the highest similarity among all the b_i sets that are implied by the L_i list. We merge all the clusters in b to a single cluster denoted by p, and recompute the similarity matrix S. After finding b and recomputing

S we go over the L_i lists. For each such list L_i, we delete all the clusters that belong to b from L_i, insert p and recompute b_i. In addition, we generate a new list L_p for the new cluster p, and compute b_p.

Note that using this algorithm, it could be that even though j and i are the most similar clusters, j and i will not end up in the same k group. If there is a cluster t s.t. b_t includes i but does not include j, and if $V(b_t) > V(b_j)$, it could be that j and i will not be in the same k cluster. This allows us to use this algorithm to overcome noise and missing values since, even when using this heuristic we still need a strong evidence from other clusters in order to combine two clusters together.

The running time of this algorithm is $O(n^3)$. Generating the L_js lists can be done in $O(n^2 \log n)$, and finding b_j for all genes j can be done in kn time. Thus the preprocessing step takes $O(n^2 \log n)$.
For each iteration of the main loop, it takes $O(n)$ to find b, and $O(nk)$ to recompute S. It takes $O(kn^2 + n^2 + kn)$ to delete all the members of b from all the L_js, insert p into all the L_js and recompute b_j. We need another $O(n \log n + k)$ time to generate L_p and compute b_p. Thus, the total running time of each iteration is $O(kn^2)$. Since the main loop is iterated $(n-1)/(k-1)$ time, the total running time of the main loop is $O(k(n-1)n^2/(k-1)) = O(n^3)$ which is also the running time of the algorithm.

The running time of the above algorithm can be improved to $O(n^2 \log n)$ by using a more sophisticated data structure instead of the linked list. For example, using Heaps, the preprocessing step has the same complexity as we currently have, and it can be shown that the (amortized) cost of every step in the main loop iteration becomes $O(n \log n)$. However, since our ordering algorithm operates in $O(n^3)$, this will not reduce the asymptotic running time of our algorithm, and since the analysis is somewhat more complicated in the Heap case we left the details out.

2.3 Reducing the number of children

Using a fixed k can lead to clusters which do not have a single node associated with them. Consider for example a dataset in which we are left with four internal nodes after some main loop iterations. Assume $k = 4$ and that the input data is composed of two real clusters, A and B such that three of the subtrees belong to cluster A, while the fourth subtree belongs to cluster B (see Figure 2). If k was fixed, we would have grouped all the subtrees together, which results in a cluster (A) that is not associated with any internal node. However, if we allow a smaller number of children than k we could have first grouped the three subtrees of A and later combine them with B at the root. This can also highlight the fact that A and B are two different clusters, since nodes with less than k children

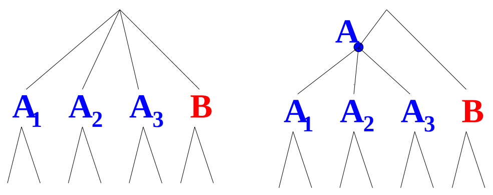


Figure 2: Fixed vs. non fixed k. In the left hand tree k is fixed at 4. This results in a cluster (A) which does not have any internal node associated with it. On the right hand side k is at most 4. Thus, the three subtrees that form cluster A can be grouped together and then combined with cluster B at the root. This results in an internal node that is associated with A.

represent a set of genes that are similar, yet significantly different than the rest of the genes.

We now present a permutation based test for deciding how many clusters to combine in each iteration of the main loop. There are two possible approaches one can take in order to perform this task. The first is to join as many clusters as possible (up to k) unless the data clearly suggests otherwise. The second is the opposite, i.e. to combine as few clusters as possible unless the data clearly suggests otherwise. Since we believe that in most cases more than 2 genes are co-expressed, in this paper we use the first approach, and combine all k clusters unless the data clearly suggests otherwise.

Let $k = 3$ and assume $b = b_c$, i.e. $b_c = argmax_i\{V(b_i)\}$ where i goes over all the clusters we have in this step. Let d be the first cluster on L_c and let e be the second cluster. Since d is the first on L_c, it is the most similar to c. We now wish to decide whether to combine the first two clusters (c and d) or combine all three clusters. Let $max_e = max\{S(c,e), S(d,e)\}$, that is S_e is the maximal similarity between e and one of the two clusters we will combine in any case. In order to test the relationship between max_e and $S(c,d)$, we perform the following test. In our case, each cluster c is associated with a profile (the average expression values of the genes in c). Assume our dataset contains m experiments, and let p_c, p_d and p_e be the three clusters profiles. Let p be the 3 by m matrix, where every row of p is a profile of one of the clusters. We permute each column of p uniformly and independently at random, and for the permuted p we compute the best (s_1) and second best (s_2) similarities among its rows. We repeat this procedure r times, and in each case test if s_2 is bigger than max_e or smaller. If $s_2 > max_e$ at least αr times (where α is a user defined value between 0 and 1) we combine c and d without e, otherwise we combine all three clusters. Note that if c and d are significantly different from e then it is unlikely that any permutation will yield an s_2 that is lower than max_e, and thus the above test will cause us to sep-

```
$KTree(n, S)$ {
    $C = \{1 \dots n\}$
    for all $j \in C$ // preprocessing step
        $L_j$ = ordered linked list of genes based on similarity to $j$
        $b_j = j \bigcup$ first $k-1$ genes of $L_j$
    for $i = 1 : (n-1)/(k-1)$ { // main loop
        $b = argmax_{j \in C}\{V(b_j)\}$
        $C = C \setminus b$
        Let $p = min\{m \in b\}$
        for all clusters $j \in C$
            $S(p,j) = \frac{\sum_{m \in b} |m| S(m,j)}{\sum_{m \in b} |m|}$
            remove all clusters in $b$ from $L_j$
            insert $p$ into $L_j$
            $b_j = j \bigcup$ first $k-1$ cluster of $L_j$
        $C = C \bigcup p$
        generate $L_p$ from all the clusters in $C$ and find $b_p$
    }
    return $C$ // $C$ is a singleton which is the root of the tree
}
```

Figure 1: Constructing k-trees from expression data

arate c and d from e. If c and d are identical to e, then all permutations will yield an s_2 that is equal to max_e, causing us to merge all three clusters. As for the values of α, if we set α to be close to 1 then unless e is very different from c and d we will combine all three clusters. Thus, the closer α is to 1, the more likely our algorithm is to combine all three clusters.

For $k > 3$ we repeat the above test for each $k' = 3 \dots k$. That is, we first test if we should separate the first two clusters from the third cluster, as described above. If the answer is yes, we combine the first two clusters and move to the next iteration. If the answer is no we apply the same procedure, to test weather we should separate the first three clusters from the fourth and so on. The complexity of these steps is rk^2 for each k' (since we need to compute the pairwise similarities in each permutations), and at most rk^3 for the entire iteration. For a fixed r, and $k << n$ this permutation test does not increase the asymptotic complexity of our algorithm. Note that if we combine $m < k$ clusters, the number of main loop iteration increases. However, since in this case each the iteration takes $O(n^2 m)$ the total running time remains $O(n^3)$.

3 Optimal leaf ordering

In this section we discuss how we preserve the pairwise similarity property of the binary tree clustering in our k-ary tree algorithm. This is done by performing optimal leaf ordering on the resulting tree. After formally defining the optimal leaf ordering problem, we present an algorithm that optimally orders the leaves of a binary tree in $O(n^3)$. We discuss a few improvements to this algorithm which further reduces its running time. Next, we extend this algorithm and show how it can be applied to order k-ary trees.

3.1 Problem statement

First, we formalize the optimal leaf ordering problem, using the following notations. For a tree T with n leaves, denote by $z_1, \cdots, z_n$ the leaves of T and by $v_1 \cdots v_{n-1}$ the $n-1$ internal nodes of T. A **linear ordering consistent with** T is defined to be an ordering of the leaves of T generated by flipping internal nodes in T (that is, changing the order between the two subtrees rooted at v_i, for any $v_i \in T$). See Figure 3 for an example of node flipping.

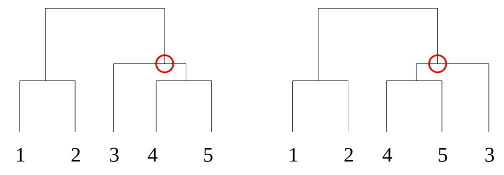


Figure 3: When flipping the two subtrees rooted at the red circled node we obtain different orderings while maintaining the same tree structure. Since there are $n-1$ internal nodes there are 2^{n-1} possible orderings of the tree leaves.

Since there are $n-1$ internal nodes, there are 2^{n-1} possible linear orderings of the leaves of a binary tree. Our goal is to find an ordering of the tree leaves that maximizes the sum of the similarities of adjacent leaves in the ordering. This could be stated mathematically in the following way. Denote by Φ the space of the 2^{n-1} possible orderings of the tree leaves. For $\phi \in \Phi$ we define $D^{\phi}(T)$ to be:

$$D^{\phi}(T) = \sum_{i=1}^{n-1} S(z_{\phi_i}, z_{\phi_{i+1}})$$

where $S(u, w)$ is the similarity between two leaves of the tree. Thus, our goal is to find an ordering ϕ that maximize $D^{\phi}(T)$. For such an ordering ϕ, we say that $D(T) = D^{\phi}(T)$.

3.2 An $O(n^3)$ algorithm for binary trees

Assume that a hierarchical clustering in form of a tree T has been fixed. The basic idea is to create a table M with the following meaning. For any node v of T, and any two genes i and j that are at leaves in the subtree defined by v (denoted $T(v)$), define $M(v, i, j)$ to be the cost of the best linear order of the leaves in $T(v)$ that begins with i and ends with j. $M(v, i, j)$ is defined only if node v is the least common ancestor of leaves i and j; otherwise no such ordering is possible. If v is a leaf, then $M(v, v, v) = 0$. Otherwise, $M(v, i, j)$ can be computed as follows, where w is the left child and x is the right child of v (see Figure 4 (a)):

$$M(v, i, j) = \max_{\substack{h \in T(w),\\ l \in T(x)}} M(w, i, h) + S(h, l) + M(x, l, j) \tag{2}$$

Let $F(n)$ be the time needed to compute all defined entries in table $(M(v, i, j))$ for a tree with n leaves. We analyze the time to compute Equation 2 as follows: Assume that there are r leaves in $T(w)$ and p leaves in $T(x)$, $r + p = n$. We must first compute recursively all values in the table for $T(w)$ and $T(x)$; this takes $F(r) + F(p)$ time.

To compute the maximum, we compute a temporary table $\mathit{Temp}(i, l)$ for all $i \in T(w)$ and $l \in T(x)$ with the formula

$$\mathit{Temp}(i, l) = \max_{h \in T(w)} M(w, i, h) + S(h, l); \tag{3}$$

this takes $O(r^2 p)$ time since there are rp entries, and we need $O(r)$ time to compute the maximum. Then we can compute $M(v, i, j)$ as

$$M(v, i, j) = \max_{l \in T(x)} \mathit{Temp}(i, l) + M(x, l, j). \tag{4}$$

This takes $O(rp^2)$ time, since there are rp entries, and we need $O(p)$ time to compute the maximum.

Thus the total running time obeys the recursion $F(n) = F(r) + F(p) + O(r^2 p) + O(rp^2)$ which can be shown easily (by induction) to be $O(n^3)$, since $r^3 + p^3 + r^2 p + rp^2 \leq (r + p)^3 = n^3$.

The required memory is $O(n^2)$, since we only need to store $M(v, i, j)$ once per pair of leaves i and j.

For a balanced binary tree with n leaves we need $\Omega(n^3)$ time to compute Equation 2; hence the algorithm has running time $\Theta(n^3)$.

We can further improve the running time of the algorithm in practice by using the following techniques:

Early termination of the search. We can improve the computation time for Equation 3 (and similarly Equation 4) by pruning the search for maximum whenever no further improvement is possible. To this end, set $s_{\max}(l) = \max_{h \in T(w)} S(h, l)$. Sort the leaves of $T(w)$ by decreasing value of $M(w, i, h)$, and compute the maximum for Equation 3 processing leaves in this order. Note that if we find a leaf h for which $M(w, i, h) + s_{\max}(l)$ is bigger than the maximum that we have found so far, then we can terminate the computation of the maximum, since all other leaves cannot increase it.

Top-level improvement. The second improvement concerns the computation of Equations 3 and 4 when v is the root of the tree. Let w and x be the left and right children of v. Unlike in the other cases, we do not need to compute $M(v, i, j)$ for all combinations of $i \in T(w)$ and $j \in T(x)$. Rather, we just need to find the maximum of all these values $M_{\max} = \max_{i,j} M(v, i, j)$.

Define $\mathrm{Max}(v, i) = \max_{h \in T(v)} M(v, i, h)$. From Equation 2 we have

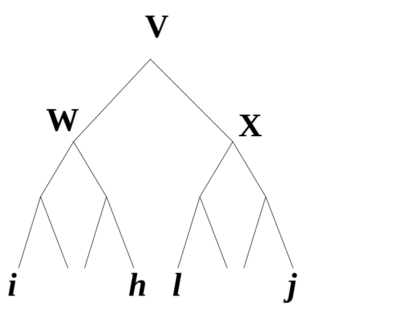
$$\begin{aligned} & M_{\max} \\ = \; & \max_{i \in T(w), j \in T(x)} \; \max_{h \in T(w), l \in T(x)} M(w, i, h) + \\ & S(h, l) + M(x, l, j) \\ = \; & \max_{h \in T(w), l \in T(x)} \mathrm{Max}(w, h) + S(h, l) + \mathrm{Max}(x, l) \end{aligned}$$

Therefore we can first precompute values $\mathrm{Max}(w, h)$ for all $h \in T(w)$ and $\mathrm{Max}(x, l)$ for all $l \in T(x)$ in $O(n^2)$ time, and then find $M_{\max}$ in $O(n^2)$ time. This is in contrast to the $O(n^3)$ time needed for this computation normally.

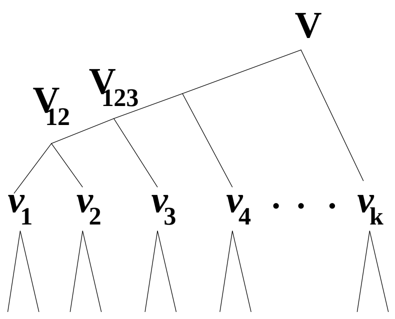
While the above two improvements do not improve the theoretical running time (and in fact, the first one increases it because of the sorting step), we found in experiments that on real-life data this variant of the algorithm is on average 50–90% faster.

3.3 Ordering k-ary trees

For a k-ary tree, denote by $v_1 \dots v_k$ the k subtrees of v. Assume $i \in v_1$ and $j \in v_k$, then any ordering of $v_2 \dots v_{k-2}$ is a possibility we should examine. For a specified ordering of the subtrees of v, $M(v, i, j)$ can be computed in the same way we computed M for binary trees



(a) Computing $M(v,i,j)$ for a binary tree



(b) K-ary tree

Figure 4: (a) A binary tree rooted at V. (b) Computing $M(v,i,j)$ for the subtrees order $1 \dots k$. For each possible ordering of $1 \dots k$ we can compute this quantity by adding internal nodes and using the binary tree algorithm.

by inserting $k-2$ internal nodes that agree with this order (see figure 4 (b)).

Thus, we first compute $M(v_{1,2},h,l)$ for all h and l leaves of v_1 and v_2. Next we compute $M(v_{1,2,3},*,*)$ and so on until we compute $M(v,i,j)$ for this order. This results in the optimal ordering of the leaves when the subtrees order is $v_1 \dots v_k$. Since there are $k!$ possible ordering of the subtrees, going over all $k!$ orderings of the subtrees in the manner described above gives rise to a simple algorithm for finding the optimal leaf ordering of a k-ary tree. Denote by $p_1 \dots p_k$ the number of leaves in $v_1 \dots v_k$ respectfully. Denote by Θ the set of $k!$ possible orderings of $1 \dots k$. The running time of this algorithm is $O(k!n^3)$ as can be seen using induction from the following recursion:

$$\begin{aligned} F(n) &= \sum_i F(p_i) + \\ &\sum_{\theta \in \Theta} \sum_{i=1}^{k-1} \left((\sum_{j=1}^{i} p_{\theta(j)})^2 p_{\theta(i+1)} + (\sum_{j=1}^{i} p_{\theta(j)}) p_{\theta(i+1)}^2 \right) \\ &\le k! \sum_i p_i^3 + \sum_{\theta \in \Theta} ((\sum_i p_i)^3 - \sum_i p_i^3) \\ &= k!(p_1 + p_2 + \dots p_k)^3 = k!n^3 \end{aligned}$$

Where the inequality uses the induction hypothesis. As for space complexity, for two leaves, $i \in v_1$ and $j \in v_k$ we need to store $M(v,i,j)$. In addition, it might be that for two other leaves $m \in v_2$ and $l \in v_{k-1}$ i and j are two boundary leaves in the internal ordering of the subtrees of v, and thus we need to store the distance between them for this case as well. The total number of subdistances we need to store for each pair is at most $k-2$, since there are only $k-2$ subtrees between the two leftmost and rightmost nodes, and thus by deleting all subpaths which we do not use we only need $O(kn^2)$ memory for this algorithm.

Though $O(k!n^3)$ is a feasible running time for small ks, we can improve upon this algorithm using the following observation. If we partition the subtrees of v into two groups, v' and v'', then we can compute $M(v)$ for this partition (i.e. when the subtrees of v' are on the right side and the subtrees of v'' on the left) by first computing the optimal ordering on v' and v'' separately, and then combining the result in the same way discussed in Section 3.2. This gives rise to the following divide and conquer algorithm. Assume $k = 2^m$, recursively compute the optimal ordering for all the $\binom{k}{k/2}$ possible partitions of the subtrees of v to two groups of equal size, and merge them to find the optimal ordering of v. In the Appendix, we formally prove that the running time of this algorithm is $O(4^k n^3)$. Thus, we can optimally order the leaves of a k-ary tree in $O(4^k n^3)$ time and $O(kn^2)$ space, which are both feasible for small ks.

4 Experimental results

First, we looked at how our heuristic k-ary clustering algorithm (described in Section 2.2) compares with the naive k-ary clustering which finds and merges the k most similar clusters in each iteration. As discussed in Section 2.1, the naive algorithm works in time $O(n^{k+1})$, and thus even for $k = 3$ it is more time consuming than our heuristic approach. We have compared both algorithms on a 1.4 GHz Pentium machine using an input dataset of 1000 genes and setting $k = 3$. While the our k-ary clustering algorithm generated the 3-ary tree in in 35 seconds, the naive algorithm required almost an hour (57 minutes) for this task. Since a dataset with 1000 genes is relatively small, it is clear that the naive algorithm does not scale well, and cannot be of practical use, especially for values of k that are greater than 3. In addition, the results of the naive algorithm where not significantly better when compared with the results generated by our heuristic algorithm. The average node similarity in the naive algorithm tree was only 0.8$\%$ higher than the average similarity in the tree generated by our algorithm (0.6371 vs. 0.6366).

Next we compared the binary and k-ary clustering using synthetic and real datasets, and show that in all cases we looked at we only gain from using the k-ary clustering algorithm.

Choosing the right value for k is a non trivial task. The major purpose of the k-ary algorithm is to reduce the influence of noise and missing values by relying on more than that most similar cluster. Thus, the value of k depends on the amount of noise and missing values in the input dataset. For the datasets we have experienced with, we have found that the results do not change much when using values of k that are higher than 4 (tough there is a difference between 2 and 4 as we discuss below). Due to the fact that the running time increases as a function of k, we concentrate on $k = 4$.

For computing the hierarchical clustering results we used the correlation coefficients as the similarity matrix (S). The clustering was performed using the average linkage method [7].

Generated data: To test the effect of k-ary clustering and ordering on the presentation of the data, we generated a structured input data set. This set represents 30 temporally related genes, each one with 30 time points. In order to reduce the effect of pairwise relationships, we chose 6 of these genes, and manually removed for each of them 6 time points, making these time points missing values. Next we permuted the genes, and clustered the resulting dataset with the three methods discussed above. The results are presented in Figure 5. As can be seen, using optimal leaf ordering (o.l.o.) with binary hierarchical clustering improved the presentation of the dataset, however o.l.o. was unable to overcome missing values, and combined pairs of genes which were similar due to the missing values, but were not otherwise similar to a larger set of genes. Using the more robust k-ary tree algorithm, we where able to overcome the missing values problem. This resulted in the correct structure as can be seen in Figure 5.

Visualizing Biological datasets: For the biological results we used two datasets. The first was a dataset from [10] which looked at the chicken immune system during normal embryonic B-cell development and in response to the overexpression of the myc gene. This dataset consists of 13 samples of transformed bursal follicle (TF) and metastatic tumors (MT). These samples where organized in decreasing order based on the myc overexpression in each of them. In that paper the authors focused on approximately 800 genes showing 3 fold change in 6 of 13 samples. These genes where clustered using Eisen's Cluster [7], and based on manual inspection, 5 different clusters where identified. In Figure 6 we present the results of the three clustering algorithms on this dataset. The left hand figure is taken from [10], and contains the labels for the 5 clusters identified. In [10] the authors discuss the five different classes, and separate them into two groups. The first is the group of genes in clusters A,B,E and D which (in this order) contain genes that are decreasingly sensitive to myc overexpression. The second is the cluster C which contains genes that are not correlated with myc overexpression. As can be seen, when using the k-ary clustering algorithm (right hand side of Figure 6) these clusters are displayed in their correct order. Furthermore, each cluster is organized (from bottom to top) based on the required level of myc overexpression. This allows for an easier inspection and analysis of the data. As can be seen, using binary tree clustering with o.l.o. does not yield the same result since the A and E clusters are broken into two parts.

The resulting 4-ary tree for the myc dataset is presented in the left hand side of Figure 6. Each node is represented by a vertical line. We highlighted (in red) some of the nodes that contain less than 4 children. Note that some of these nodes correspond to clusters that where identified in the original paper (for example, the top red node corresponds to cluster C). Had we used a fixed $k = 4$, these clusters might not have had a single node associated with them.

Clustering Biological datasets: The second biological dataset we looked at is a collection of 79 expression experiments that where performed under different conditions, from [7]. In order to compare our k-ary clustering to the binary clustering we used the MIPS complexes categories [1]. We focused on the 979 genes that appeared in both the dataset and the MIPS database. In order to compare the binary and 4-ary results, we have selected a similarity threshold (0.3, though similar results were obtained for other thresholds), and used this threshold to determine the clusters indicated by each of the trees. Starting at the root, we went down the tree and in each internal node looked at the average similarity of the leaves (genes) that belong to the subtree rooted at that node. If the average similarity was above the selected threshold the subtree rooted at that node was determined to be a cluster. Otherwise, we continued the same process using the sons of this internal node.

This process resulted in 36 distinct clusters for the binary tree and 35 for the 4-ary tree. Next, we used the MIPS complexes to compute a p-value (using the hyper geometric test) for each of the clusters with each of the different complexes, and to chose the complex for which the cluster obtained the lowest p-value. Finally, we looked at all the clusters (in both trees) that had both more than 5 genes from their selected complex and a p-value below 10^{-3}.

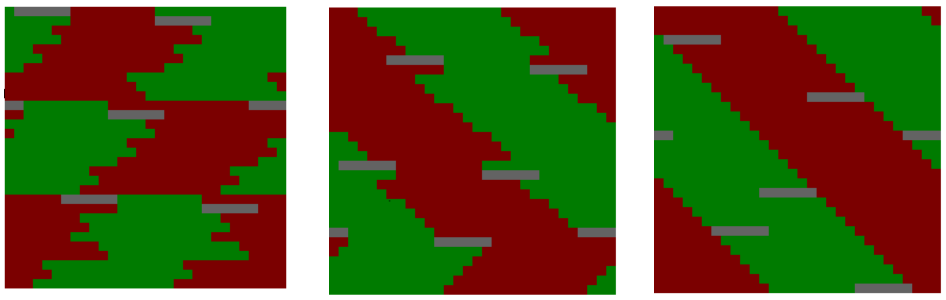


Figure 5: Color comparison between the three different clustering methods on a manually generated dataset. Green corresponds to decrease in value (-1) and red to increase (1). Gray represents missing values.

The results seem to support our assumption that the 4-ary tree can generate more meaningful biological results. The above process resulted in 10 significant clusters in the binary tree, while the 4-ary tree contained 11 significant clusters. Further, as shown in Table 4, the clusters generated by the 4-ary algorithm where, on average, more specific than the binary clusters. Out of the 7 complexes that where represented in both trees, the 4-ary tree contained 4 clusters for which more than 50% of their genes belonged to a certain complex, while the binary tree contained only two such clusters. In particular, for the Proteasome complex, the 4-ary tree contained a cluster in which almost all of its genes (28 out of 29) belonged to this complex, while for the corresponding cluster in the binary tree was much less specific (28 out of 39 genes). These results indicate that our k-ary clustering algorithm is helpful when compared to the binary hierarchical clustering algorithm. Note that many of the clusters in both the binary and 4-ary algorithms do not overlap significantly with any of the complexes. This is not surprising since we have only used the top level categorization of MIPS, and thus some of the complexes should not cluster together. However, those that do cluster better when using the 4-ary algorithm.

5 Summary and future work

We have presented an algorithm for generating k-ary clusters, and ordering the leaves of these clusters so that the pairwise relationships are preserved despite the increase in the number of children. Our k-ary clustering algorithm runs in $O(n^3)$. As for ordering, we presented an algorithm which optimally orders the leaves of a binary tree in $O(n^3)$ improving upon previous published algorithm for this task. We extended this algorithm to order k-ary trees in $O(2^k n^3)$, which is on the order of $O(n^3)$ for a constant k.

We presented several examples in which the results of our algorithm are superior to the results obtained using binary tree clustering, both with and without ordering. For synthetic data our algorithm was able to overcome missing values and correctly retrieve a generated structure. For biological data our algorithm is better in both identifying the different clusters and ordering the dataset.

There are several ways in which one can extend the results presented in this paper. One interesting open problem is if we can further improve the asymptotic running time of the optimal leaf ordering algorithm. A trivial lower bound for this problem is n^2 which leaves a gap of order n between the lower bound and the algorithm presented in this paper. Note that hierarchical clustering can be performed in $O(n^2)$, thus reducing the running time of the o.l.o. solution would be useful in practice as well. Another interesting (though more theoretical) problem is the extension of the o.l.o. algorithm for a two dimensional quadtree, where the goal is to order both the rows and the columns of the resulting outcome simultaneously.

Acknowledgements

We thank Therese Biedl, Brona Brejová, and Tomáš Vinař who made important contributions to Section 3.2. We thank the anonymous referees for useful suggestions and comments. Z.B.J was supported by the Program in Mathematics and Molecular Biology at the Florida State University with funding from the Burroughs Wellcome Fund. A.M.H. was supported by the Leverhulme foundation, UK and NSERC.

References

[1] *Saccharomyces cerevisiae - Protein Complexes*. URL: http://www.mips.biochem.mpg.de/proj/yeast/catalogues/complexes/.

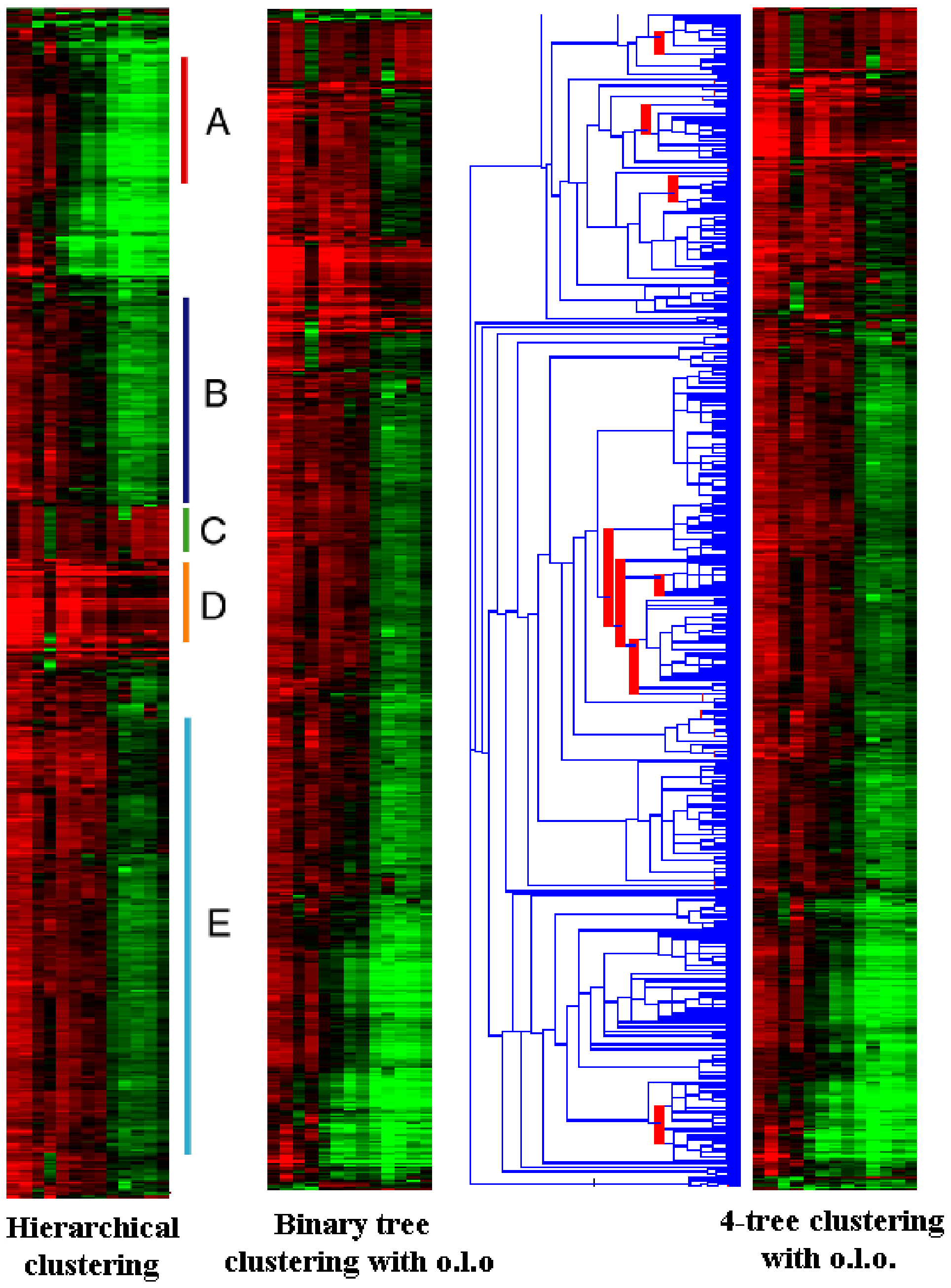


Figure 6: Color comparison of the three different clustering algorithms using the myc overexpression dataset. The left hand side is the figure that appears in the original paper containing the five different clusters that where identified. Note that by using the k-ary clustering algorithm we where able to both identify and order the clusters correctly (based on the required myc gene expression). The tree in the figure is the 4-ary tree where some of the nodes that contain less than 4 children are highlighted. See text for more details

Complex	**# genes**	**binary tree**			**4-ary tree**		
		#∈ complex	**cluster size**	**p-value**	**#∈ complex**	**cluster size**	**p-value**
Nucleosomal	8	8	48	$2 * 10^{-11}$	8	15	$3 * 10^{-16}$
Respiration chain	31	11	67	$2 * 10^{-6}$	14	27	$5 * 10^{-16}$
Proteasome	35	28	39	$8 * 10^{-39}$	28	29	0
Replication	48	27	88	$5 * 10^{-18}$	30	106	$3 * 10^{-19}$
Cytoskeleton	48	15	53	$3 * 10^{-9}$	9	28	$2 * 10^{-6}$
RNA processing	107	12	26	$4 * 10^{-6}$	13	46	$7 * 10^{-4}$
Translation	208	152	231	0	144	195	0

Table 1: Comparison between the binary tree and 4-ary clustering algorithms for the complexes that where identified in both trees. In most cases (5 out of 7) the 4-ary results where more significant than the binary results. See text for complete details and analysis.

[2] Z. Bar-Joseph, D. Gifford, and T. Jaakkola. Fast optimal leaf ordering for hierarchical clustering. In *ISMB01*, 2001.

[3] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6:281–297, 1999.

[4] R. E. Burkard, Deineko V. G., and G. J. Woeginger. The travelling salesman and the pq-tree. *Mathematics of Operations Research*, 24:262–272, 1999.

[5] R. Degerman. Ordered binary trees constructed through an application of kendall's tau. *Psychometrica*, 47:523–527, 1982.

[6] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, New-York, NY, 1999.

[7] M.B. Eisen, P.T. Spellman, P.O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *PNAS*, 95:14863–14868, 1998.

[8] N. Gale, W. C. Halperin, and C.M. Costanzo. Unclassed matrix shading and optimal ordering in hierarchical cluster analysis. *Journal of Classification*, 1:75–92, 1984.

[9] G. Gruvaeus and H. Wainer. Two additions to hierarchical cluster analysis. *British Journal of Mathematical and Statistical Psychology*, 25:200–206, 1972.

[10] P.E. Neiman and et al. Analysis of gene expression during myc oncogene-induced lymphomagenesis in the bursa of fabricius. *PNAS*, 98:6378–6383, 2001.

[11] E. Segal and D. Koller. Probabilistic hierarchical clustering for biological data. In *Recomb02*, 2002.

[12] R. Sharan, R. Elkon, and R. Shamir. Cluster analysis and its applications to gene expression data. *Ernst Schering workshop on Bioinformatics and Genome Analysis*, 2001.

[13] R. Sharan and R. Shamir. Click: A clustering algorithm for gene expression analysis. In *ISMB00*, 2000.

[14] T. S. Spellman and et al. Comprehensive identification of cell cycle-regulated genes of the yeast $saccharomyces\ cerevisia$ by microarray hybridization. *Molecular Biology of the Cell*, 9:3273–3297, 1998.

[15] P. Tamayo and et al. Interpreting patterns of gene expression with self organizing maps: Methods and applications to hematopoietic differentiation. *PNAS*, 96:2907–2912, 1999.

6 Appendix

In this appendix we prove that the running time of the divide and conquer algorithm for ordering k-ary trees (which is presented in Section 3.3) is $O(4^k n^3)$. In order to compute the running time of this algorithm we introduce the following notations: We denote by $\Gamma(v)$ the set of all the possible partitions of the subtrees of v to two subsets of equal size. For $\gamma \in \Gamma(v)$ let $v_{\gamma(1)}$ and $v_{\gamma(2)}$ be the two subsets and let $p_{\gamma(1)}$ and $p_{\gamma(2)}$ be the number of leaves in each of these subsets. The running time of this algorithm can be computed by solving the following recursion:

$$F(n) = F(p_1 + p_2 \ldots + p_k) = \sum_{i=1}^{k} F(p_i) + D(v)$$

Where

$$\begin{aligned} D(v) \quad &= \sum_{\gamma \in \Gamma(v)} p^2_{\gamma(1)} p_{\gamma(2)} + p_{\gamma(1)} p^2_{\gamma(2)} + \\ &\quad + D(v_{\gamma(1)}) + D(v_{\gamma(2)}) \end{aligned}$$

and $D(i) = 0$ if i is a singleton containing only one subtree. The following lemma discusses the maximal running time of the divide and conquer approach.

Lemma 6.1 *Assume $k = 2^m$ then*

$$D(v) \leq \frac{(2^m)!}{\prod_{i=0}^{m-1}(2^i)!}(\sum_{j=1}^{k} p_j)^3 - \sum_{j=1}^{k} p_j^3$$

Proof: By induction on m.
For $m = 1$ we have $k = 2$ and we have already shown in Section 3.2 how to construct an algorithm that achieves this running time for binary trees. So $D(v) = p_1^2 p_2 + p_1 p_2^2 < 2(p_1 + p_2)^3 - p_1^3 - p_2^3$.

Assume correctness for $m - 1$. Let $k = 2^m$. Then for every $\gamma \in \Gamma(v)$ we have

$$\begin{aligned}
&p^2_{\gamma(1)}p_{\gamma(2)} + p_{\gamma(1)}p^2_{\gamma(2)} + D(v_{\gamma(1)}) + D(v_{\gamma(2)}) \leq \\
&p^2_{\gamma(1)}p_{\gamma(2)} + p_{\gamma(1)}p^2_{\gamma(2)} + \frac{(2^{m-1})!}{\prod_{i=0}^{m-2}(2^i)!}(\sum_{j \in \gamma(1)} p_j)^3 \\
&- \sum_{j \in \gamma(1)} p_j^3 + + \frac{(2^{m-1})!}{\prod_{i=0}^{m-2}(2^i)!}(\sum_{j \in \gamma(2)} p_j)^3 - \sum_{j \in \gamma(2)} p_j^3 \\
&\leq \frac{(2^{m-1})!}{\prod_{i=0}^{m-2}(2^i!)}(\sum_{j=1}^{k} p_j)^3 - \sum_{j=1}^{k} p_j^3
\end{aligned}$$

The first inequality comes from the induction hypothesis, and the second inequality arises from the fact that $\gamma(1)$ does not intersect $\gamma(2)$ and $\gamma(1) \bigcup \gamma(2) = \{1 \ldots k\}$. Since $|\Gamma(v)| = \binom{k}{k/2} = \frac{(2^m)!}{(2^{m-1})!(2^{m-1})!}$, summing up on all $\gamma \in \Gamma$ proves the lemma. ■

It is now easy to see (by using a simple induction proof, as we did for the binary case) that the total running time of this algorithm is: $O(\frac{k!}{\prod_{i=0}^{m-1}(2^i)!}n^3)$ which is faster than the direct extension discussed above. If $2^m < k < 2^{m+1}$ then the same algorithm can be used by dividing the subtrees of v to two groups of size 2^m and $k - 2^m$. A similar analysis shows that in this case the running time is $O(\frac{k!}{\prod_{i=0}^{\lfloor \log k \rfloor - 1}(2^i)!}n^3)$, which (using the Sterling approximation) is $O(4^{k+o(k)}n^3)$.