Hierarchical clustering

Farid Arthaud

May 31, 2019

Abstract Clustering is a general problem where, given a certain number of data points along with a way of measuring their similarity or dissimilarity, one has to group them up according to that measure. Hierarchical clustering is a particular case of clustering, providing a recursive partitioning of a data set into successively smaller clusters. A hierarchical clustering can be represented by a tree whose leaves correspond to the data points, and each internal node corresponds to a cluster. It has grown to become a popular data analysis method, with various applications in data mining, phylogeny, and even finance. Hierarchical clustering is often applied in unsupervised machine learning contexts, where it provides a flexible way of categorizing data points into a variable number of clusters, as opposed to other clustering methods.

1 Introduction

The clearest and most illustrative application of hierarchical clustering is phylogenetics: given the similarity or dissimilarity of a set of species, one is asked to categorize them into recursively smaller clusters and thus a tree, from *kingdom*, *phylum* to *genus* and *species*. Such a tree is known as **dendrogram**, and can be used to form any number of clusters. Figure 1 is an example of a dendrogram applied to phylogenetics, which could be considered a hierarchical clustering if a measure of similarity or dissimilarity over these organisms is provided.

Indeed, unlike other approaches to clustering, hierarchical clustering does not provide a clear-cut number of clusters – once a hierarchy is obtained, one can recover a k-clustering by performing $k-1$ cuts in the dendrogram obtained from the hierarchical clustering. This is true at all levels of granularity, meaning the size and number of clusters can be chosen freely depending on where the cuts are made in the dendrogram. However, note that the choice of where the cuts are made can be a non-trivial question by and of itself. For instance leaving a subtree child of the root as a cluster and cutting the rest of the tree into $(k-1)$ clusters might be suboptimal. Most algorithms in their most popular form yield binary trees (for reasons explored in Lemma 1), meaning they naturally provide clusterings into $2, 4, 8, \ldots$ clusters.

As with most unsupervised machine learning problems, hierarchical clustering has focused on algorithms and suffered from a lack of a precise method for measuring performance – such as objective functions to evaluate the quality of a clustering and compare algorithms' performances. To remedy this, Dasgupta [Das16] introduced and studied an interesting objective function for hierarchical clustering for similarity measures, which we shall explore in Section 2.

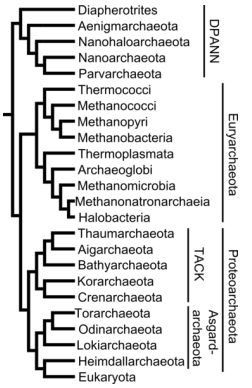

Figure 1: Dendrogram of the phylogenetic tree of single-celled microbes **archaea**.

Traditionally, there are two classes of methods to derive hierarchical clusterings. The most popular algorithms consist of '*bottom-up*' agglomerative methods: single, complete and average linkage. These start from 1-clusters and progressively merge them into a tree structure. The other class of methods are '*top-down*' divisive algorithms, of which we shall explore bisection 2-center. These algorithms start with a single cluster of all points, and divide it until obtaining a tree structure based on the cuts.

We will explore these algorithms in Section 3, where we will first calculate their complexities, before proving bounds on worst-case performance compared to an optimal clustering. We shall also explore these techniques in the context of one-dimensional data on the real line, improving on best-known upper bounds for general input.

All of the aforementioned algorithms have existed for much longer than Dasgupta's cost function, and can thus only be seen as approximation algorithms – they don't provide optimal trees in general. As a matter of fact, finding an optimal hierarchical clustering for a given set of data points is an NP-complete problem, as we shall see in Section 4.

2 Cost functions

2.1 Similarity and dissimilarity measures, generating trees

Definition 1. An **ultrametric space** is a metric space (X, d) such that all triangles are isosceles and have their two equal sides longer than the third:

$$d(x, y) \leq \max\left(d(x, z), d(y, z)\right)$$

Ultrametrics allow us to define a first type of input, graphs generated by ultrametrics. They are not the general framework in which similarity graphs are defined, but they lead to the important notion of *ground-truth inputs*.

Definition 2. A **similarity graph** is a weighted unoriented graph, whose weights represent the similarity between the nodes, which represent the data points.

A similarity graph is said to be **generated by an ultrametric** (X, d) if its vertices V are a subset of X and the weights are defined by the relation: $w(x, y) = f(d(x, y))$, where f is a non-negative decreasing function.

Similarly, a **dissimilarity graph** is said to be generated by an ultrametric if the same condition is verified with f a non-negative increasing function.

For most of the following work, we shall suppose that we are in the similarity case, unless stated otherwise. Definitions and proofs are somewhat similar in the dissimilarity case, only with inequalities occasionally flipped.

These graphs – whether of similarity or dissimilarity – are the input to our algorithm, giving us the structure of the data to be clustered. One could also imagine another way of generating such a graph, from a weighted tree structure over its nodes.

Definition 3. A tree T is said to be a **generating tree** for the similarity graph G if its leaves are labeled by the nodes of the graph, and there exists a *weight function* W mapping internal nodes of the tree to $\mathbb{R}_+$ such that:

- If N is a (possibly indirect) descendant of M then $W(M) \leq W(N)$.
- For any two vertices x, y in the graph, the weight of the edge connecting them in the graph is the weight according to W of their least common ancestor in T.

Theorem 1. *A graph is generated by an ultrametric if and only if it is generated by a tree. Such a graph is called a* ***ground-truth input****.*

Proof. Let G be a graph generated by the tree T. Take two vertices (u, v) which are of minimal distance $d(u, v)$ in G, assemble them into a tree of two leaves and a new root. The root should be weighted by $f(d(u, v)) = w(u, v)$ – which is maximal amongst the images of distances by f. We then insert the other points one by one into this tree structure to make it into a '*comb tree structure*' as in Figure 2, according to the following algorithm.

For any given node x, the triangle (x, u, v) is isosceles in x by definition. We insert the nodes in order of increasing distance to u – which is the same as the distance to v by the previous remark – along with a new root according to the comb structure. The weight of the new root is chosen as $w(u, x)$ at each step.

All nodes of equal distance to u are inserted in a same subtree to the right of the comb, which is generated recursively with the same algorithm. By induction, we can suppose the weights of the inner nodes in this subtree to be correct. The algorithm clearly places the weights in a decreasing fashion from the root along the comb (weights illustrated in Figure 2), and within the right subtrees by induction, we however have to prove that all the nodes at a given distance of u – a right subtree – are closer to one another than to u.

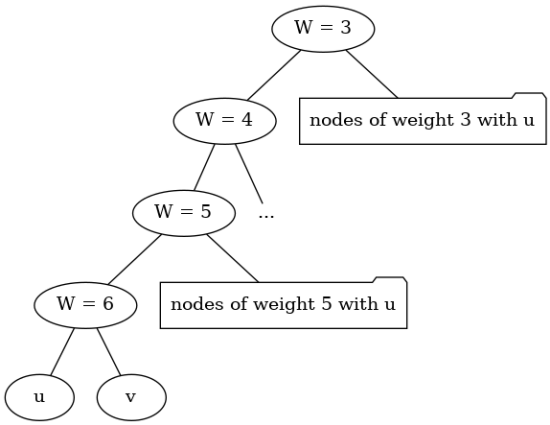

Figure 2: Comb tree structure.

Take x, y such that $d(x, u) = d(y, u)$. Then the triangle (x, y, u) is isosceles in u, and so $d(x, y) \leq d(x, u)$, so in turn $w(x, y) \geq w(x, u)$ and thus the weights are correctly ordered from the root into the right subtree.

It is only left to prove that the inner node weights yield the appropriate distances between two nodes inserted after u and v. If x is being inserted and y was previously inserted, then (x, u, y) forms an isosceles triangle, and we know that $d(u, x) > d(u, y)$ thus the triangle is isosceles in x. This tells us that $d(x, y) = d(u, x) = d(v, x)$, which allows us to conclude the fact the weight function $f(d(x, y)) = f(d(x, u))$ is correct.

Conversely, suppose our graph to be generated by a tree T, let us show there exists an ultrametric over the graph's vertices generating the same graph. For this we consider the distance to be an injective decreasing function of the weight yielded by the tree: $d(x, y) = 1/(1 + W(LCA_T(x, y)))$, where LCA_T is the least common ancestor – except $d(x, x) = 0$ of course, for the space to be metric.

We now have to verify the isosceles triangles condition: let (x, y, z) be three nodes in the tree. Let a be their least common ancestor, then there is at least one node of (x, y, z) in each of its subtrees, and therefore either its left or right subtree has exactly one node of (x, y, z).

We can suppose without loss of generality x is that node. We thus have $LCA_T(x, y) = LCA_T(x, y)$ so $w(x, y) = w(x, u)$ and therefore $d(x, y) = d(x, u)$. This allows us to conclude that the isosceles triangles condition is verified – which is in turn stronger than the triangle inequality. □

As stated above, the ultrametric case – or *ground-truth case* – is not the framework in which hierarchical clustering is done, since one could wish to use an Euclidean norm for points in $\mathbb{R}^n$ for instance. However, the *ground-truth* case allows for an intuition of what kind of problems hierarchical clustering can solve: these are the cases where we could input a graph generated by a tree to an algorithm and wish for it to return the tree generating it. We shall now

formalize this notion thanks to *admissible cost functions*.

2.2 Admissible cost functions

In the case of hierarchical clustering, a clustering for a graph is a tree whose leaves are labeled by the vertices of the graph. A cost function for a given graph is therefore a function which maps trees into $\mathbb{R}_+$, with lower costs being desirable in the similarity case. In his paper [Das16], Dasgupta considers cost functions of a particular form, later generalized in Cohen-Addad et al.'s paper [CKMM18], which also characterizes admissible cost functions amongst them.

Definition 4. For a given internal node N of T, let N_l, N_r be respectively its left and right subtrees, and $s(N)$ the number of leaves of the subtree rooted in N. The Γ cost function is defined in the following manner:

$$\begin{cases} \Gamma(T) & = \sum_{N \in T} \gamma(N) \\ \gamma(N) & = s(N) \sum_{x \in N_l, y \in N_r} w(x, y) \end{cases}$$

In words, the cost at a given internal node is the sum of the dissimilarities between pairs of points separated between left and right subtree, weighted by the amount of leaves under that node.

Definition 5. A desirable property for cost functions is that the generating tree performs well when compared to other tree structures. A cost function is said to be **admissible** if for any ground-truth input, it is minimal on generating trees for this input.

Remark 1. A slightly more general version for cost functions replaces $s(N)$ in γ by an arbitrary function $g(s(N_l), s(N_r))$. We will only look at the case where g is the sum of its arguments, and we rely on the characterization of admissible cost functions given in [CKMM18] to prove that Γ is indeed admissible in Corollary 1.

Remark 2. In the dissimilarity case, we can consider the same cost function Γ, but with objective of maximizing it. It is often called **objective** function, and the cost often called **score** of a clustering.

Definition 6. We define the cost function Ψ as follows,

$$\Psi(T) = \sum_{N \in T} \sum_{x \in N_l, y \in N_r} (s(T) - s(N)) w(x, y)$$

This cost function is sometimes called **dual** to Dasgupta's cost function, and is due to Moseley and Wang [MW17].

Proposition 1. The sum of Γ and Ψ is constant for a given graph.

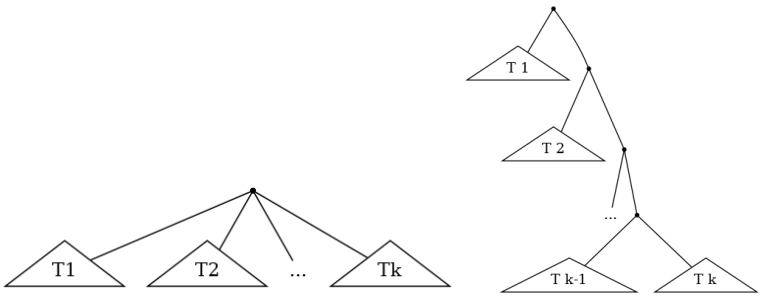

Figure 3: The transformation to make a non-binary tree binary.

Proof.

$$\Gamma(T) + \Psi(T) = \sum_{N \in T} \sum_{x \in N_l, y \in N_r} s(G) w(x, y) = s(G) \sum_{e \in E} w(e)$$

□

In particular, Property 1 shows that minimizing Γ is the same as maximizing Ψ and vice-versa. We shall be focused on Dasgupta's cost function until Subsection 3.4, though thanks to this duality most properties of Dasgupta's cost function Γ are closely tied to those of Moseley and Wang's cost function Ψ.

2.3 Elementary properties

There is a natural way to extend the Γ cost function to general non-binary trees, by summing over nodes in different subtrees of each node. One could try to imagine minimizing the cost in such a manner, by considering this vaster class of trees. However, this effort proves fruitless.

Lemma 1. *There always exists a binary optimal tree.*

Proof. Given an optimal tree T whose children of the root are $(T_1, \ldots, T_k)$, we recursively apply the following transformation to make it into a binary tree: take a new root, make its left child T_1 and its right child a node with children $(T_2, \ldots, T_k)$.

We then apply the same transformation to the children of this new root – that is T_1 and the new node. This clearly makes the tree into a binary tree, we only have to prove that it decreases cost. The final result is illustrated in Figure 3.

After a single step of the transformation, the cost at the root will be decreased,

$$\sum_{\substack{x_i \in T_i, x_j \in T_j \\ 1 \le i < j \le k}} w(x_i, x_j) \ge \sum_{\substack{x_1 \in T_1, x_j \in T_j \\ 1 < j \le k}} w(x_1, x_j)$$

since the second sum has its indexes strictly included in the first's.

Finally, a single step of the transformation will have, on top of the cost at the root, the cost of T_1 and the cost of the subtree containing $(T_1, \dots, T_k)$. The other costs will also be greater by induction over the transformation steps. □

From now on, clusterings shall be by definition binary trees only.

One first type of graph one might like to look at is one where all points are similarly similar: all of the weights are equal to 1 and every vertex is connected to every other vertex. Such a graph is known as a **clique**.

Lemma 2. *All clusterings on a clique have the same cost.*

Proof. If the result is true, then the cost of any clustering of the clique of size n can be written as a function A of n. We shall prove this by induction over n the size of the clique. Note the result is trivially true for $n = 2$ where only one tree structure exists.

Let T be a clustering over a clique of size $n > 2$. The cost at the root will be $ns(T_l)s(T_r)$. By induction, the cost of the subtrees will respectively be $A(s(T_l)), A(s(T_r))$. Let us denote $k = s(T_l)$ the size of the left subtree, we have to prove

$$\forall k \in [|1, n-1|], A(n) = nk(n-k) + A(k) + A(n-k)$$

that is, we have to prove the right hand term coincides for all values of k.

To do so, we prove by induction that

$$A(n) = 2\binom{n+1}{3} = \frac{n(n+1)(n-1)}{3}$$

This is true for $n = 2$, and then substituting it into the previous formula does indeed give the same result $A(n)$ for all k by developing the three products. □

In [CKMM18], it is proven that a cost function derived from g (as described in the previous subsection) is admissible if and only if it yields the same cost for all cliques, it is symmetrical and an increasing function of each of its arguments individually. The previous lemma thus allows us to deduce,

Corollary 1. *The Γ cost function is admissible.*

3 Classical algorithms

3.1 Agglomerative approaches

The most popular approach to hierarchical clustering is linkage algorithms: these start off with clusters of single vertices, and sequentially merge them up into a single tree. The difference in the single, complete and average linkage algorithms resides in the choice for the *similarity function*. This function is

defined on pairs of trees – not clusterings, simply trees labeled by a subset of the graph's nodes.

$$\mathrm{sim}(T_1, T_2) = \begin{cases} \dfrac{1}{s(T_1)s(T_2)} \displaystyle\sum_{x \in T_1, y \in T_2} w(x,y) & \text{average linkage} \\ \displaystyle\max_{x \in T_1, y \in T_2} w(x,y) & \text{single linkage} \\ \displaystyle\min_{x \in T_1, y \in T_2} w(x,y) & \text{complete linkage} \end{cases}$$

Algorithm 1 Linkage algorithms (similarity setting)

Input: A weighted graph $G = (V, E, w)$
  Create $s(G)$ singleton trees for each node in G
  Define the similarity function as described above
  while there are at least two trees **do**
    Take two trees T_1, T_2 maximizing the similarity function
    Replace these two trees by a new tree having these two trees as children
  end while
  return The single remaining tree

Proposition 2. The complexity of all three linkage algorithms is $O(s(G)^3)$.

Proof. Let $n = s(G)$, note that a naive attempt at a proof for the algorithm as described above yields a complexity $O(n^5)$: we will have to optimize it.

The while loop is performed exactly $n-1$ times, since it reduces the amount of trees by exactly one at each iteration. The algorithm can cache the similarities between trees in a table, thus only having to recalculate the similarities for the newly merged tree. The amount of trees at the end of step k is $n-k$, and there are thus $n-k-1$ similarity calculations to be done at each loop, and a table of size $(n-k)^2$ whose minimum is to be found.

Moreover, calculating the similarity of a merged tree with another tree can be done in constant time based on the two merged trees' similarities with the other tree:

$$\begin{aligned} \mathrm{sim}_{\mathrm{avg}}(T_1 \cup T_2, T_3) &= \frac{s(T_1)s(T_3)\mathrm{sim}_{\mathrm{avg}}(T_1, T_3) + s(T_2)s(T_3)\mathrm{sim}_{\mathrm{avg}}(T_2, T_3)}{(s(T_1) + s(T_2))s(T_3)} \\ \mathrm{sim}_{\mathrm{sng}}(T_1 \cup T_2, T_3) &= \max(\mathrm{sim}_{\mathrm{sng}}(T_1, T_3), \mathrm{sim}_{\mathrm{sng}}(T_2, T_3)) \\ \mathrm{sim}_{\mathrm{comp}}(T_1 \cup T_2, T_3) &= \min(\mathrm{sim}_{\mathrm{comp}}(T_1, T_3), \mathrm{sim}_{\mathrm{comp}}(T_2, T_3)) \end{aligned}$$

Initializing the similarity table is a $O(n^2)$ operation. Merging trees can be made into a constant time operation, and thus the overall complexity is bounded by a constant times, $n^2 + \sum_{k=1}^{n-1}(n-k)^2 + (n-k-1) = O(n^3)$ □

When the input to the linkage algorithms is a ground-truth input – a graph generated by a tree or an ultrametric – then the result will be optimal. We shall not prove this result here, but we do for the next algorithm in Subsection 3.2.

Remark 3. Being efficient in the ground-truth case does not mean being efficient in general. This can for instance be seen in Charikar et. al's paper [CCN19]: average linkage can yield no better approximations than $\frac{2}{3}$ for Γ and $\frac{1}{3}$ for Ψ in the worst case.

3.2 Divisive methods

The next algorithm, *bisection 2-center*, can be seen as an adaptation of a class of more general algorithms *bisection k-means* to the case of hierarchical clustering.

Algorithm 2 Bisection 2-center (similarity setting)

Input: A weighted graph $G = (V, E, w)$
Let (u, v) be an edge that maximizes $\min_x \max(w(x, u), w(x, v))$
Let A be the set of nodes x such that $w(x, u) \geq w(x, v)$
Let $B = V \setminus A$
Let T_A and T_B be the recursively obtained results on subgraphs A, B respectively
return A new root with subtrees T_A, T_B

The choice of (u, v) might seem a little cryptic at first. Recalling that the greater $w(x, y)$, the more similar x and y are, the interpretation for the centers (u, v) would be *the pair of points such that the point furthest from the closest of the centers is as close as possible.*

This might still sound cryptic. For each vertex x, we look at the closest of the centers from it; we look at the point for which this measure is the greatest, and we want to minimize it. In sum, we want all points to be close to at least one of the centers, which is intuitive for a clustering.

Proposition 3. The complexity of bisection 2-center is a $O(s(G)^4)$.

Proof. At each internal node N of the tree, the complexity is $S(N)^3$ in order to find the edge minimizing the *minmax*. The produced tree is full – there are no internal nodes that have only one child – and so the tree has $s(G) - 1$ internal nodes. Since we can bound $s(N)^3 \leq s(G)^3$, the overall complexity is $S(G)^4$. □

Unlike linkage algorithms, bisect 2-center does not always yield an optimal tree on ground-truth inputs. It however does when the optimal tree is unique – up to an automorphism – which can be defined in terms of generating trees.

Definition 7. A tree is said to be **strictly generating** for a graph G if the weight inequality in Definition 3 is strict: $W(M) < W(N)$ for any descendant M of N.

Proposition 4. If G is a strict ground-truth input, then the output of bisection 2-center is optimal.

Proof. We shall proceed by induction over the size of the graph. Let T be a strictly generating tree for the input and W the associated weight function.

For all $x \in T_l$ and $y \in T_r$, let us denote $w(x,y) = W(T) = \alpha$, which is consistent by the definition of a generating tree. Let (u,v) be the edge the algorithm chooses, and suppose both these vertices are in T_l. Then,

$$min_x \left(max(w(x,u), w(x,v))\right) \leq min_{x \in T_r} \left(max(w(x,u), w(x,v))\right) = \alpha$$

Now let $u' \in T_l, v' \in T_r$ be a new edge but separated between both subtrees of T. Then for $x \in T$, we have

$$\max\left(w(u',x), w(v',x)\right) \geq \min\left(W(T_l), W(T_r)\right) > W(T) = \alpha$$

where the first inequality comes from the definition of a generating tree and the second the definition of a strictly generating tree.

Taking the min over $x \in T$ thus yields the fact that (u',v') has a strictly greater '*minmax*' than (u,v), which is a contradiction with the fact that the algorithm chose the latter.

Therefore, if (u,v) is picked by the algorithm, they are in different subtrees of T. We can suppose without loss of generality $u \in T_l$, let us inspect the partition (A,B) performed by Algorithm 2:

- For $x \in T_r$, we have $w(x,u) = \alpha < W(T_r) \leq w(x,v)$ so the algorithm places x in the set B.
- For $x \in T_l$, we have $w(x,u) \geq W(T_l) > \alpha = w(x,v)$ so the algorithm places x in the set A.

Up to exchanging the sets A and B, the algorithm correctly separates the two subtrees, and by induction it yields T up to an isomorphism – which will always be a ground-truth tree, and thus optimal by Corollary 1. □

Remark 4. Note that we have avoided mentioning $w(u,u)$ and $w(v,v)$ in this last proof. In the similarity case, a node is more similar with itself than any other: one can recover this from the ultrametric case in Subsection 2.1, since $f(d(u,u)) = f(0) \geq f(x)$ for all x.

This thus means that the partitioning does indeed yield $u \in A$ and $v \in B$, since the reasoning extends to these two nodes.

3.3 Experimental performance

To assess the efficiency of these algorithms, I ran a simulation for randomly chosen integers in $[|-500; 500|]$ for the distance $w(x,y) = |x-y|$ – which falls in the dissimilarity case, recall Remark 2.

Definition 8. A clustering of points in $\mathbb{R}$ is said to be **non-interlaced** if the leaves are ordered along an in-order traversal.

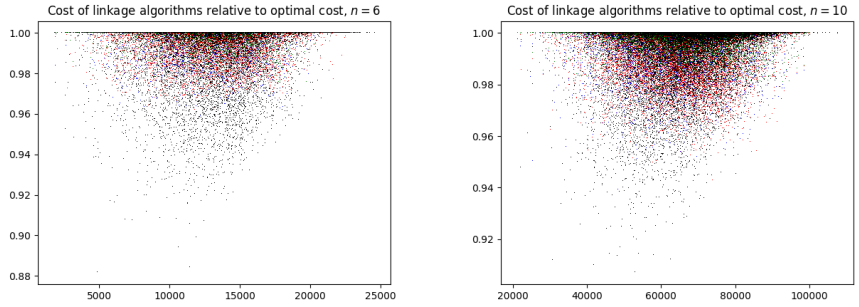

Figure 4: Experimental results for linkage and bisect 2-center algorithms

The hypothesis from my internship is that any input graph of points over $\mathbb{R}$ has an optimal non-interlaced clustering. To test this hypothesis in a few practical cases, I performed dynamic exhaustive searches for non-interlaced clusterings and general clusterings, and plotted the score obtained by the linkage algorithms and bisect 2-center compared to the optimal clustering.

The code differs from Algorithms 1 and 2 in two ways:

- Algorithm 1 is given in the similarity case, and had to be adapted here to the dissimilarity case – this means it selects two trees of smallest similarity to merge.
- The algorithms I coded try to make the final tree as non-interlaced as possible. For linkage, this means the merging procedure compares the smallest element of the trees to merge and puts the smallest one to the left. For bisect, this means the list is first sorted and kept sorted when partitioning, and both partitions' smallest elements are compared when assembling them into a tree.

I performed the experiment for $n = 6$ and $n = 10$, under the assumption that if a counterexample existed, it would exist for small values of n. The plots can be seen in Figure 4: the bottom axis is the optimal score for all clusterings, in **green** the score for single linkage, in **red** for average linkage and **blue** for complete linkage. All scores are relative to the optimal score.

I found no counterexamples, there always existed a non-interlaced optimal clustering. The algorithms all performed very well, often finding a non-interlaced optimal clustering.

The average performances of the algorithms can be seen in Figure 5. One should also note that the medians are generally slightly higher than the averages for all algorithms (1.000 median for all algorithms in the $n = 6$ case). This means the algorithms very often return an optimal clustering, with only a few cases returning suboptimal results.

	Single linkage	Complete linkage	Average linkage	Bisect 2-center
$n = 6$	0.998	0.997	0.996	0.994
$n = 10$	0.997	0.995	0.999	0.993

Figure 5: The average score of the four algorithms relative to the optimal score

3.4 Case of the real line

It is clear that the cases $n \geq 4$ are not ground-truth cases: a triangle on the real line is not isosceles unless one point is halfway between the two others. In this particular case, the many strong geometrical properties which the general ultrametric case does not verify allows us to prove interesting bounds on the results of the clustering algorithms. The results in this subsection are due to Charikar et. al [CCNY19].

A first useful property of average linkage on the line is the following lemma, whose intuitive - yet technical - proof we shall leave out for conciseness.

Lemma 3. *The result of average-linkage can be made to be non-interlaced up to choice of the way similarity ties are resolved.*

It is intuitive in the sense that the similarity of two adjacent clusters is always greater than that of two non-adjacent clusters – and thus we can merge adjacent clusters only, creating a non-interlaced clustering.

Recall the Ψ cost function from Definition 6 and Property 1: minimizing Γ is the same as maximizing Ψ.

Theorem 2. *The average linkage algorithm yields a tree of score for Ψ at least $\frac{1}{2}\Psi(T^*)$, where T^* is an optimal tree.*

Proof. For this proof, we use a potential function defined over partitions of V,

$$\Phi\left(S_1, \ldots, S_m\right) = \sum_{\substack{x<y<z\in V \\ x,y,z \text{ are separated}}} w(x,y) + w(y,z)$$

where "separated" is defined as: "are in pairwise distinct classes". If $V = \{x_i\}_i$, we define the quantity,

$$\alpha = \Phi\left(\{x_1\}, \ldots, \{x_n\}\right) = \sum_{x<y<z} w(x,y) + w(y,z)$$

Observe now that,

$$\Psi(T^*) = \sum_{\{x,y,z\}\subseteq V} S^x_{y,z}(T^*)w(y,z) + S^y_{x,z}(T^*)w(x,z) + S^z_{x,y}(T^*)w(x,y)$$

where $S^x_{y,z}$ equals 1 if and only if x is separated from y, z earlier than the others in the tree. Indeed, in the right hand sum, each weight will appear with an

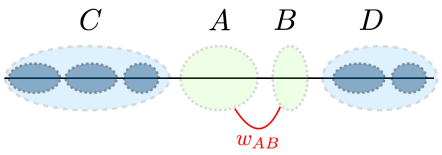

Figure 6: Naming of sets when merging in average linkage – image taken from [CCNY19] with permission.

indicator of 1 exactly as many times as there are leaves outside of its least common ancestor. We then have by definition of a similarity measure,

$$\begin{aligned}\Psi(T^*) &\leq \sum_{x<y<z} \max\left(w(x,y), w(y,z), w(x,z)\right) \\ &\leq \sum_{x<y<z} \max\left(w(x,y), w(y,z)\right)\end{aligned}$$

We can implicitly define the potential function over forests, by considering each tree as a class of the partition. Over the course of an execution of average linkage, the potential function will go from α down to 0 – initially all points are separated, and no points are separated in a one-tree forest.

On every merge in average linkage, the new root of the merged tree will be a node in the newly merged tree. We can thus consider that the merge will increase the score of the final tree by the score of that node. If we denote ΔS this variation in final score and $\Delta\Phi$ the variation in potential induced by the merge, we will show $\Delta S + \frac{\Delta\Phi}{2} \geq 0$ at every step, which will conclude the proof.

By Lemma 3, the clusters A and B being merged at a given step are adjacent, and we can denote C and D the sets of points at the left of cluster A and the right of B respectively – see Figure 6 for an illustration.

By definition of the score function we have $\Delta S = (|C|+|D|)\sum_{x\in A, y\in B} w(x,y)$. To condense notations, we shall denote $w(A,B) = \sum_{x\in A, y\in B} w(x,y)$. Consider now a previously separated triplet $x < y < z$ being merged in this step. We either have $(x,y,z) \in C\times A\times B$ or $(x,y,z)\in A\times B\times D$. For such a triplet, the potential will drop by $w(x,y)+w(y,z)$ and so,

$$-\Delta\Phi = (w(A,B)|C| + w(A,C)|B|) + (w(A,B)|D| + w(B,D)|A|)$$

Now we note that average linkage has picked (A,B) to merge rather than (C,A) and (B,D) – we consider all points in C to be a single cluster knowing that the similarity with the right-most cluster of C is greater than the similarity with the whole of C. This in particular means that,

$$\begin{cases} \dfrac{w(A,B)}{|A|\cdot|B|} & \geq \dfrac{w(A,C)}{|A|\cdot|C|} \\ \dfrac{w(A,B)}{|A|\cdot|B|} & \geq \dfrac{w(B,D)}{|B|\cdot|D|} \end{cases} \iff \begin{cases} w(A,B)|C| & \geq w(A,C)|B| \\ w(A,B)|D| & \geq w(B,D)|A| \end{cases}$$

By summing these final two lines, we get:

$$\Delta S = (|A| + |C|)w(A, B) \geq -\Delta\Phi - \Delta S$$

which yields the wanted result. □

Remark 5. If we write the result of average linkage T_{avg}, we have proven that $\Psi(T_{\text{avg}}) \geq \frac{1}{2}\Psi(T^*)$. We can transcribe this result in terms of Γ using Proposition 1:

$$\frac{1}{2}\left(s(G)\sum_{e\in E} w(e) + \Gamma(T^*)\right) \geq \Gamma(T_{\text{avg}})$$

I was unable to find an interesting interpretation of this inequality, since the constant term cannot really be controlled. Moreover, the constant term is much larger than $\Gamma(T^*)$.

Remark 6. For general input, Charika et. al [CCN19] have proven that an optimal bound is $\frac{2}{3}\Psi(T^*)$.

Note that there are examples on the real line where average linkage can yield results as low as $\frac{3}{4}\Psi(T^*)$, yet the gap between $\frac{1}{2}$ and $\frac{3}{4}$ has not yet been filled. This is a goal of my work during the internship – either find counterexamples closer to $\frac{1}{2}$ or prove a bound closer to $\frac{3}{4}$.

4 NP-completeness

Another type of algorithm is enumeration, or exhaustive search. This can be programmed dynamically, with complexity the amount of tree structures over the input. This is of the order e^n, which is clearly suboptimal. We can actually prove that any algorithm consistently returning an optimal clustering is of such complexity – under the assumption P $\neq$ NP. The results in this section are due to Dasgupta [Das16], and can be generalized to the Ψ cost function as well.

4.1 Maximizing is equivalent to minimizing

Lemma 4. *Finding the worst clustering is equivalent to finding the best clustering.*

Proof. Given a weighted graph $G = (V, E, w)$, we shall define its complementary weighted graph $G^c = (V, E, w^c)$ where $w^c = \max_{(x,y)\in E}(w(x, y)) - w$. In words, $w + w^c$ is constant over all edges, and $w^c \geq 0$.

Notice G and G^c have the same vertices, so clusterings over G can also be considered as clusterings for G^c. We shall prove that for a clustering T, the sum of the costs of T for G and G^c is constant.

Let us calculate the sum of costs at a node N of the tree:

$$\gamma_G(N)+\gamma_{G^c}(N) = s(N)\sum_{x\in N_l, y\in N_r} w(x, y)+w^c(x, y) = s(N)s(N_l)s(N_r)\max(w)$$

This is also the expression of the cost at a node for a clique with edge weights $\max(w)$. We thus conclude that by summing these terms over all nodes $N \in T$, we obtain the cost of a clustering for the clique, which by Lemma 2 equals $\max(w)A(s(G))$.

Finally, calculating the complementary weighted graph can be done in polynomial time, and is an involutive operation. Since we have proven $\Gamma_G(T) = C - \Gamma_{G^c}(T)$ where C is a constant, maximizing and minimizing Γ are equivalent problems. □

4.2 NAESAT*

To prove NP-completeness, we shall reduce instances of NAESAT* into instances of our problem, a variant of *not-all-equal satisfiability*.

Definition 9. NAESAT is the problem of satisfiability of a CNF formula with the follow constraint: in the truth affectation, each clause must have at least one literal evaluating to true and at least one evaluation to false. Schematically, the truth evaluation must turn each clause into

$$\textbf{true} \vee \textbf{false} \vee \textbf{false} \qquad \text{or} \qquad \textbf{true} \vee \textbf{true} \vee \textbf{false}$$

A formula satisfying NAESAT is said to be **not-all-equal satisfiable**.

NAESAT* is the problem of not-all-equal satisfiability of a CNF formula with clauses containing two to three variables, verifying the following: any literal appears exactly thrice, once in a three-clause and twice in two-clauses with opposing polarities.

Remark 7. Note that not-all-equal satisfiability in the case of two-clauses means the truth affectation turns them into **true** $\vee$ **false**.

We shall here accept that NAESAT* is equivalent to NAESAT, and that NAESAT is an NP-complete problem – since these proofs are of little relevance to the main proof.

4.3 NP-completeness of maximizing the cost

Theorem 3. *Maximizing* Γ *for a given input* G *is an NP-complete problem.*

Proof. To this end, we shall prove that for a given instance Φ of NAESAT*, we can construct a weighted graph G and an integer M such that

- $\max(w), s(G)$ and M are polynomial in the length of Φ
- Φ is not-all-equal satisfiable if and only if there exists a tree of cost greater than M for G.

We first remove the following types of redundancies from the formula:

- If a clause is included in another – either two equal three-clauses or a two-clause included in a three-clause – it is removed

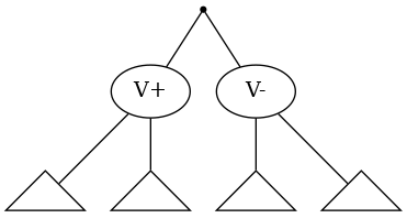

Figure 7: Tree structure constructed in the proof of NP-completeness.

- If a clause with its polarities inverted is included in another, it is also removed
- Trivially true clauses can be removed, i.e. $x \vee \bar{x}$

Let n be the number of variables in this simplified formula Φ, m the amount of three-clauses and m' the amount of two-clauses.

We create a graph G of size $2n$, whose vertices correspond to the literals, i.e. x_i and $\bar{x_i}$ for each variable x_i. For each three-clause, add three edges between the literals, and three edges between the negations of the literals in the clause. For each two-clause, add one edge between the literals, and one edge between the negations of the literals in the clause. These edges all have unit weight.

Finally, add edges between x_i and $\bar{x_i}$ of weight $\alpha = 2nm + 1$. Thanks to the previous simplifications, all of the previously added edges are pairwise distinct, and there is a total of $6m + 2m' + n$ edges.

Suppose now that Φ is not-all-equal satisfiable, we construct a tree as in Figure 7.

The root separates literals with true valuation V^+ and false valuation V^-. This will break exactly two edges in each triangle – by definition of not-all-equal satisfaction. It will also break the edges created for two-clauses for the same reason, as well as the edges formed between a literal and its negation.

The cost at the root will thus be $2n(4m + 2m' + n\alpha)$. The remaining edges are thus exactly one per created triangle, so exactly $2m$. Moreover, exactly half of these edges will be in V^+ and the other in V^- by definition of the triangles.

The second level of the tree is created by splitting the m edges. This is possible because a literal cannot appear in two triangles by the simplification rules made earlier. The sum of the costs at nodes V^+, V^- will be $2nm$. In total the total cost of this tree will be $2n(m + 4m + 2m' + n\alpha)$, which is how we now define M.

Conversely, suppose we have a tree T of cost at least M. The first cut of this tree must split all edges $(x_i, \bar{x_i})$: if it does not, its cost is at most the cost of all edges except for one of weight α,

$$2n(6m + 2m' + n\alpha) - \alpha < 2n(5m + 2m' + n\alpha) = M$$

Thus the algorithm divides the vertices in such a way to break all of these edges, and leaves at least one edge per triangle untouched. By a similar com-

parison, the algorithm must break all of these edges on the next step in order to have cost at least M.

By picking this split as a valuation, the first cut tells us the valuation is coherent (a literal is not both true and false) and the second tells us it satisfies Φ not-all-equally. □

5 Bibliography

[CCN19] Moses Charikar, Vaggos Chatziafratis, and Rad Niazadeh. Hierarchical clustering better than average-linkage. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2291–2304, 2019.

[CCNY19] Moses Charikar, Vaggos Chatziafratis, Rad Niazadeh, and Grigory Yaroslavtsev. Hierarchical clustering for euclidean data. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019*, 2019.

[CKMM18] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 378–397, 2018.

[Das16] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 118–127, 2016.

[MW17] Benjamin Moseley and Joshua Wang. Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 3097–3106, 2017.