Twister Tries: Approximate Hierarchical Agglomerative Clustering for Average Distance in Linear Time

Michael Cochez[*]
University of Jyvaskyla, Department of
Mathematical Information Technology
P.O. Box 35, FI-40014 University of Jyvaskyla,
Finland
michael.cochez@jyu.fi

Hao Mou[†]
University of Jyvaskyla, Department of
Mathematical Information Technology
P.O. Box 35, FI-40014 University of Jyvaskyla,
Finland
muhaocd@gmail.com

ABSTRACT

Many commonly used data-mining techniques utilized across research fields perform poorly when used for large data sets. Sequential agglomerative hierarchical non-overlapping clustering is one technique for which the algorithms' scaling properties prohibit clustering of a large amount of items. Besides the unfavorable time complexity of $O(n^2)$, these algorithms have a space complexity of $O(n^2)$, which can be reduced to $O(n)$ if the time complexity is allowed to rise to $O(n^2 \log^2 n)$. In this paper, we propose the use of locality-sensitive hashing combined with a novel data structure called *twister tries* to provide an approximate clustering for average linkage. Our approach requires only linear space. Furthermore, its time complexity is linear in the number of items to be clustered, making it feasible to apply it on a larger scale. We evaluate the approach both analytically and by applying it to several data sets.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining; H.3.3 [**Information Search and Retrieval**]: Clustering

Keywords

Hierarchical Clustering; Locality-Sensitive Hashing; Average Linkage; Linear Complexity

1. INTRODUCTION

Hierarchical clustering is used in many domains to analyze data and has a relatively long history. There are several benefits of hierarchical clustering over normal partitioning. The first one is that the hierarchy can be cut at any level to create a different partitioning of the collection. Further, the hierarchy can be used to navigate the data or displayed in a hierarchical form by using dendrograms.

One possible approach for hierarchical clustering is bottom-up. Initially, each item is put into its own cluster, and on each iteration two clusters are selected and merged into a larger one. This approach is often called agglomerative, but the algorithm is known by many names, such as Globally Closest Pair (GCP) clustering [9], Sequential Agglomerative Hierarchical Non-overlapping (SAHN) clustering [12, 16] or Agglomerative Hierarchical Clustering (AHC) [13, 8].

The application domains in which the algorithms are used today deal with ever-larger datasets, and hence the scalability of algorithms has become more important. However, the traditional algorithms for AHC do not scale well for large data sizes.

In this article, we introduce twister tries, which perform an approximation of the agglomerative hierarchical clustering. Both the time and space complexity of the algorithm are dependent on the desired precision (i.e. the probability that the algorithm chooses the correct clusters to be merged during one step of clustering). Once these parameters are fixed, the time and space complexity of the algorithm are linear with respect to the number of items clustered.

The main contributions of this article are the twister tries data structure and algorithm. Their most profound benefits are:

- The production of clusterings that are comparable to standard AHC algorithms (see section 7 and in particular figs. 6 and 7).
- Both linear time and space complexity. Therefore, the algorithm can scale to over a million data points (see section 5 and figs. 9 and 11).

As a minor contribution, we also introduce a locality-sensitive family of functions for the average Jaccard, cosine, and Hamming distance between sets of points (see section 4.1). This function can be used in any place were a locality-sensitive hashing is used (for instance, in a k-nearest neighbor search).

2. AGGLOMERATIVE HIERARCHICAL CLUSTERING

When creating a hierarchical clustering, two major strategies are commonly used, namely agglomerative and divisive. Agglomerative hierarchical clustering (AHC) is a "bottom-up" approach, which means that each node starts out as

[*] Member of the Industrial Ontologies Group (IOG)

[†] Student of the Web Intelligence and Service Engineering (WISE) master program.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2751521 .

a single cluster. Then pairs of clusters are combined into larger ones as the process continues, until only one cluster is left. The divisive strategy, on the other hand, initially treats all of the items as part of one large cluster. At each iteration, one cluster is split in two sub-clusters until each item belongs to its own cluster. In this section, we will briefly discuss the primitive AHC algorithm and some possible optimizations in the next one.

Algorithm 1 is the pseudocode for a primitive AHC algorithm. The algorithm takes the node labels and a distance matrix as input. The label of a node is a unique identifier and the distance matrix contains the pairwise distances between the nodes, with respect to a predetermined distance metric. As shown in the algorithm, the procedure merges two clusters I and J into a new cluster L and updates the distance between L and every other cluster by using an updating formula.

Algorithm 1 Primitive AHC algorithm (adapted from [15])

```
procedure PRIMITIVE AHC(s,d)
    $S_{origin} \leftarrow S$
    $n \leftarrow |S|$
    $den \leftarrow []$
    $size[x] \leftarrow 1$, for all $x \in S$
    for $i \leftarrow 0, \ldots, n-2$ do
        $(I, J) = argmin_{(SxS) \setminus \Delta} d$
        append (I,J) to den
        $S \leftarrow S \setminus \{I, J\}$
        Create a new label $L$, $L \notin S \cup S_{origin}$
        Update the matrix containing the distances
            $d[L,x] = d[x,L] = \mathrm{FORMULA}(d[I,x], d[b,J],$
            $d[I,J], size[I], size[J])$, for all $x \in S$
        $size[L] \leftarrow size[I] + size[J]$
        $S \leftarrow S \cup \{L\}$
    end for
    return den
end procedure
```

The FORMULA is the updating formula used for the chosen linkage, while d is the distance metric. Note that our notation somewhat freely uses I and J to mean either the label of the cluster or the cluster itself.

The distance-updating formula depends on the chosen linkage strategy or, in other words, the way in which the distance between two clusters of items is defined. Commonly found in the literature are single, complete, average, weighted, Ward, centroid and median linkage (see also [15]). In this article, we use the average distance between all pairs of elements as the measurement for cluster similarity. More formally:

Definition 1 (Average distance)**.** *Given a distance measure d, the average distance, d_A, between two clusters I and J is defined as*

$$d_A(I,J) = \frac{\sum_{i=1}^{|I|} \sum_{j=1}^{|J|} d(I_i, J_j)}{|I| * |J|}$$

For this linkage strategy, the updating formula is as follows:

Definition 2 (Average Updating Formula)**.** *Given the fact that the algorithm joins clusters I and J together, the distance between any other cluster K and the newly formed cluster is the weighted average distance of I, K and J, K with size $n_{I,K}$ and $n_{J,k}$*

$$\textit{Average Updating Formula} = \frac{n_{I,K}\, d(I,K) + n_{J,K}\, d(J,K)}{n_{I,K} + n_{J,K}}$$

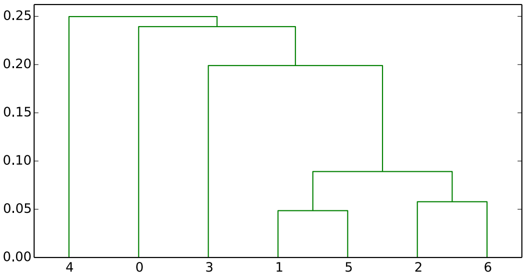

Figure 1: An example of a dendrogram.

The result of executing the AHC algorithm is called a dendrogram. Formally, one could define a dendrogram as

Definition 3 (Dendrogram)**.** *Given a finite set S_0 of size n, a stepwise dendrogram is a list of $n-1$ pairs $(I_i, J_i)(i = 0, ..., n-2)$ such that $I_i, J_i \in S_i$, where S_{i+1} is recursively defined as $(S_i \setminus \{I_i, J_i\}) \cup L_i$ and L_i is a label for a new node.*

This definition was adapted from Müllner [15], who extended the pair used in our definition to a triple which also contains the distance between the clusters. However, since this distance is unambiguously defined by the sequence of pairs, we left it out of this definition. The dendrogram can be visualized for illustrating the structure of a dataset. Figure 1 shows one such visualization.

One of the problems for AHC is the time complexity. The primitive algorithm, shown in algorithm 1, has a time complexity of $O(n^3)$, and requires $O(n^2)$ memory. Several improvements for this bound have been found. Some of these will be briefly discussed in the next section.

3. RELATED WORK

Hierarchical clustering has been used for decades, but mainly for small datasets. For these datasets, the scalability of the algorithm is not of major importance. Recently, however, more demanding applications have created demand for more scalable approaches. In the literature, two approaches to handle this scaling can be found. At first, research work has focused on finding faster, exact algorithms for hierarchical clustering, meaning that the devised methods produce exactly the same hierarchy as the original algorithm, but with a lower theoretical, or practical, time or space complexity. In later work, algorithms for approximate hierarchical clustering can also be found. The target here is to find a hierarchy which closely resembles the hierarchy of the exact algorithms, but in a faster or more memory-efficient fashion. Our research falls into this domain.

From the recent research about exact algorithms, we first highlight the work by Eppstein [6], which has a slightly more ambitious scope. The work introduces data structures that can maintain the closest pair in the dataset. $O(n \log^2 n)$ time is needed for each insertion or deletion when limited to $O(n)$ space. This time bound is improved to $O(n)$ when the algorithm is allowed $O(n^2)$ space. Note that if one is able to maintain a linear insertion and removal time, it is easy to see that one can implement hierarchical clustering in $O(n^2)$ time by removing the closest pair and adding their union at each step. Subsequently, Gronau and Moran [9] provided insight into optimal implementations of the Unweighted Pair Grouping Method with Arithmetic-mean (UPGMA), which is equivalent to exact hierarchical clustering with average linkage. In a reaction to inferior algorithms used in practical implementations of AHC, Müllner [16] provided an efficient C++ implementation for exact hierarchical clustering. We used the Python bindings of that work in our evaluation. The SparceHC algorithm described in the work by Nguyen et al. [17] is more focused on saving memory while performing an online clustering. They achieve this by only partially computing the distances. Their approach shows an empirical linear memory complexity, but requires quadratic time.

For approximate approaches, the authors in [11] worked on an idea similar to the one in this paper. However, their focus was on AHC, using the single linkage method, whereas we show results for average linkage in this work. Rasheed et al. [21] also use locality-sensitive hashing for clustering, but the clustering is non-hierarchical. Kull and Vilo [13] proposed the HappieClust algorithm, which uses similarity heuristics to perform the clustering, using either single, average or complete linking.

Patra et al. [18] have also proposed a method for AHC of large datasets by using average linking. Their proposed method, named leader Average-link *l-AL*, first derives a set of leaders and subsequently applies the standard average link method on those. The method works for any distance metric and has the benefit that the whole dataset does not have to be stored in memory, since only the leaders are retained. The drawback, however, is that only those leaders get clustered, meaning that an incomplete dendrogram gets created. Furthermore, the method has two parameters, $\tau :$ the maximum distance between any point and its leader, and $h :$ the inter-cluster distance; if clusters are further than h apart, they are not merged. In other words, the algorithm does not continue the clustering until only one cluster is left. These parameters will have to be adapted to the dataset and no recommendation has been given on how to determine good parameters. The work by Kriege et al. [12] also attempts to find the hierarchical clustering of data faster than the theoretical bound by providing a heuristic nearest-neighbor search, which can be used for all metric spaces. The crux of their algorithm is the use of pivot points, which lowers the need for pairwise distance calculations. Their proposal has a best-case running time of $O(n * \log n)$, but rises up to $O(n^2 \log n)$ in the worst case. Their experimental evaluation shows, however, that the running time is sub-quadratic in practice.

An algorithm with a similar aim as ours can be found from [8]. The authors proposed a linear time and space complexity algorithm for hierarchical clustering, based on quantization. The main difference with our work is that their approach can handle single, complete, and average linkage while our approach only supports average linkage. However, because of this quantization, this algorithm also allows the leaves of the dendrogram to contain multiple points and the method can only work for the cosine distance between positive data points. Our work does not have these limitations and, besides the more generalized cosine distance, it is also possible to use it for Jaccard and Hamming distance or, as we will show below, any similarity measure for which a proportionally sensitive family of functions (see definition 5) can be derived.

4. LOCALITY-SENSITIVE HASHING

Indyk and Motwani [10] presented the initial work about Locality-sensitive hashing (LSH) as a method for finding approximate nearest neighbors. In order to find nearest neighbors, one first needs a family of independent hash functions which are likely to hash similar objects together and dissimilar ones apart. Once all objects have been hashed using these functions, creating a database, it is possible to query for nearest neighbors of a given query point. In order to do this, one hashes the query point with the same hash functions. The result returned is a subset of objects from the database that were hashed into the same buckets as the query point. [1]

Mathematically speaking, to apply LSH we construct a family $\mathcal{H}$ of hash functions which map from a space $\mathcal{D}$ to a finite universe $\mathcal{U}$.

Definition 4 (Locality-sensitive family)**.** *(adapted from [1]) Let $\mathcal{H}$ be a family of hash functions mapping from a domain $\mathcal{D}$ to some universe $\mathcal{U}$ and d be a distance metric defined on $\mathcal{D}$. Then, given $d_1 < d_2$, $\mathcal{H}$ is called (d_1, d_2, p_1, p_2)-sensitive if for every two $p, q \in \mathcal{D}$ and every $h \in \mathcal{H}$*

$$\textit{if } d(p,q) \leq d_1 \textit{ then } \Pr\left[h\left(p\right) = h\left(q\right)\right] \geq p_1$$
$$\textit{if } d(p,q) \geq d_2 \textit{ then } \Pr\left[h\left(p\right) = h\left(q\right)\right] \leq p_2$$

where $p1 > p2$

In words, this means that we are working in a certain domain of interest D in which the distance between points is given by a metric d. A suitable family of functions $\mathcal{H}$ has the property that if we pick a function h uniformly at random, then the result of applying the function on points which are close to each other $(d(p,q) \leq d_1)$ is likely to produce the same results $(\Pr\left[h\left(p\right) = h\left(q\right)\right] \geq p_1)$. Moreover, applying this function to points that are far away from each other is unlikely to produce the same results.

Since the probabilities p_1 and p_2 might be close to each other, using only one function from $\mathcal{H}$ to decide whether points may be similar might not be sufficient. The solution to this problem is the use of amplification, which is achieved by creating b functions g_j, each consisting of r hash functions chosen uniformly at random from $\mathcal{H}$. In other words, the function g_j is the concatenation of r independent hash functions $g_{j,k}$.

The terms b and r stand for *bands* and *rows.* If one collects all outcomes of $g_{j,k}, 0 < j \leq b$ and $0 < k \leq r$ in a two-dimensional table, it can be regarded as consisting of b bands containing r rows each. A function g_j maps points p and q into the same bucket if all hash functions it is built from hash the points into the same buckets. If for any j, the function g_j maps p and q into the same bucket, p and q are considered close. The amplification creates a new locality-sensitive family which is $\left(d_1, d_2, 1-(1-{p_1}^r)^b, 1-(1-{p_2}^r)^b\right)$ sensitive.

A very accessible introduction to LSH can be found in the textbook by Rajaraman and Ullman [19]. Some notations used in this article are borrowed from that work.

4.1 A locality-sensitive hash function for average distance between clusters

In this section, we develop a family of hash functions that is locality-sensitive with regards to the average distance between clusters. The average distance (see definition 1) is defined in the function of the distances between all pairs of points in the clusters. In general, any distance metric can be used for calculating this average. The LSH family that we develop here, however, places stricter requirements. Namely, the distance metric must have a LSH family defined that is *proportionally sensitive.*

Definition 5 (Proportionally sensitive family)**.** *Let $\mathcal{H}$ be a family of hash functions mapping from $\mathcal{D}$ to some universe $\mathcal{U}$ and d be a distance metric defined on $\mathcal{D}$. Then, given $k > 0$, $\mathcal{H}$ is called k-proportionally sensitive with respect to the distance metric d if for every two $p, q \in \mathcal{D}$ and every $h \in \mathcal{H}$*

$$\Pr\left[h\left(p\right) = h\left(q\right)\right] = 1 - kd(p, q)$$

Note that any k-proportionally sensitive family is also $(d_1, d_2, 1 - kd_1, 1 - kd_2)$-sensitive.

A proportionally sensitive family can, for example, be found for the Jaccard distance, which is defined as follows:

Definition 6 (Jaccard distance)**.** *The Jaccard distance between two (finite, non-empty) sets I and J is $d_{\mathcal{J}} = 1 - \frac{|I| \cap |J|}{|I| \cup |J|}$*

The standard LSH family used for Jaccard distance is *Min-hash* [4].

Definition 7 (Min-hash)**.** *Min-hash is a family of functions $h_\pi\left(K\right) = \min\left\{\pi\left(k\right) | k \in K\right\}$, where π is a random permutation of the universe.*

Now, it can be shown [19] that

Fact 1. *For sets I and J,*

$$\Pr\left[h_\pi\left(I\right) = h_\pi\left(J\right)\right] = 1 - d_{\mathcal{J}}(I, J)$$

and hence, min-hash is 1-proportionally sensitive.

Note that a similar condition can be found for Hamming and cosine distance and their respective, traditionally used locality-sensitive families. For the Hamming distance d_H (i.e. the number of dimensions in which two binary vectors of length l differ), Indyk and Motwani [10] proposed a family of functions $\mathcal{H} = \{h_i\left(p\right) = p_i | i \in [1, l]\}$. This family is $(d_1, d_2, 1 - \frac{d_1}{l}, 1 - \frac{d_2}{l})$-sensitive. But also, $\Pr\left[h_i(x) = h_i(y)\right] = 1 - \frac{d_H(x,y)}{l}$. Hence, this family is also $\frac{1}{l}$-proportionally sensitive. Again, similarly, for the cosine distance one can find a $(d_1, d_2, 1 - \frac{d_1}{180}, 1 - \frac{d_2}{180})$-sensitive $\mathcal{H}_A$ family, derived from the random hyperplane hashing family [5]. This family is $\frac{1}{180}$-proportionally sensitive.

Now, given a k-proportionally sensitive family, we can define a new family of functions, which acts upon clusters and is locality-sensitive with respect to the average distance as follows:

Definition 8 ($\mathcal{H}_A$)**.** *Given the k-proportionally function family $\mathcal{H} = \{h_1, \ldots h_n : S \to U\}$ for the distance d, define a function family $\mathcal{H}_A = \{h_{A_1} \ldots h_{A_n} : 2^S \to U\}$ such that $h_{A_i}\left(s\right) = h_i$ (an element of s, selected uniformly at random).*

Note that without loss of generality, h_{A_i} is not a function in the strict mathematical sense since two invocations of h_{A_i} with the same argument could result in a different result (i.e. it is not necessarily true that $h_{A_i}(s) = h_{A_i}(s)$).

Theorem 1. *The family of functions $\mathcal{H}_A$, constructed according to definition 8, is a $(d_1, d_2, 1 - kd_1, 1 - kd_2)$-sensitive family of functions for the average distance d_A with respect to d.*

Proof. The theorem is true if for every two $I, J \in 2^S$ and every $h_A \in \mathcal{H}_A$

$$\text{if } d_A(I, J) \leq d_1 \text{ then } \Pr\left[h_A\left(I\right) = h_A\left(J\right)\right] \geq 1 - kd_1 \quad (1)$$
$$\text{if } d_A(I, J) \geq d_2 \text{ then } \Pr\left[h_A\left(I\right) = h_A\left(J\right)\right] \leq 1 - kd_2 \quad (2)$$

Now we will prove the stronger statement

$$\Pr\left[h_A\left(I\right) = h_A\left(J\right)\right] = 1 - kd_A(I, J),$$

which implies (1) and (2).

$$\begin{aligned}
\Pr\left[h_A\left(I\right) = h_A\left(J\right)\right] &= \frac{\sum_{i=1}^{|I|} \sum_{j=1}^{|J|} \Pr\left[h\left(I_i\right) = h\left(J_j\right)\right]}{|I| * |J|} \\
&= \frac{\sum_{i=1}^{|I|} \sum_{j=1}^{|J|} \left(1 - kd\left(I_i, J_j\right)\right)}{|I| * |J|} \\
&= 1 - k\frac{\sum_{i=1}^{|I|} \sum_{j=1}^{|J|} d\left(I_i, J_j\right)}{|I| * |J|} \\
&= 1 - kd_A\left(I, J\right)
\end{aligned}$$

□

Hence, the family of functions created according to definition 8 from a proportionally sensitive family is itself a locality-sensitive, and even a k-proportionally sensitive, family. Furthermore, we can create such a family of functions for the Jaccard, Hamming, and cosine distance, among others.

4.2 LSH Forest

LSH Forest was introduced in [2] and solves particular problems with the standard LSH algorithm. The first improvement over LSH is that in LSH Forest, the points do not get a fixed-length band. Instead, the length of the band is decided for each point individually. The length of the band r of a specific point and hash function g_j is such that there is no other point which is hashed to the same bucket by all $g_{j,k}$ with $k < r$. Put another way, in standard LSH the function g_j maps two points to same bucket if all functions it is composed of do so as well. LSH Forest, on the other hand, saves on the evaluation of hash functions and only evaluates that much of g_j as needed to distinguish between the different data points.

The second improvement of LSH forest is that it saves storage space. First and foremost, the use of dynamic labels eliminates the need for the construction of multiple indexes. Further, in the LSH Forest data structure, these labels are placed in a prefix tree (also called a *trie*) where the label on the edge is the value of the sub-hash function of g_j. In this way, duplicate prefixes only get stored once.

The height of the tries and multiple tries show similarity with the rows and bands of the original LSH algorithm. The height of the tries and also the number of rows ensure that items without much similarity get separated from each other while multiple tries and bands ensure that similar items do not, by coincidence, get separated from each other.

One issue with both the conventional LSH and LSH Forest is that when using hashing, it is not possible to distinguish between arbitrary close points. Therefore, LSH assumes a minimum distance between any two points and LSH Forest defines a maximum label length. This maximum label length is equal to the maximum height of the tree and is indicated as k_m.

The LSH Forest is used to answer k-nearest neighbor queries. In the next section, we will introduce twister tries, which can perform a hierarchical clustering with linear time and space complexity. A central part of the data structure are prefix tries which are very similar to the ones used in the LSH Forest. As shown below, the twister tries inherit issues with arbitrarily close points in the dataset.

5. TWISTER TRIES

Using the locality-sensitive family proposed in definition 8 and the LSH forest described in the previous section, one can easily devise a hierarchical clustering algorithm as follows. First, create a cluster for each point and insert these in the LSH Forest. Then, on each clustering step, get the nearest neighbor for each cluster and retain only the pair with the closest distance. Merge these clusters by removing both and reinsert their union. The algorithm ends when only one cluster is left.

This straightforward algorithm does have a time complexity of $O(n^2)$, since it involves $n-1$ clustering steps consisting of $O(n)$ operations. A more optimal approach was already proposed in the article which introduced the notion of locality-sensitive hashing [10], based on an earlier version of the work by Eppstein [6]. The idea is to use the LSH data structure to repeatedly find the closest pair, which can be done, in sub-linear time, by checking whether buckets contain two different points. This makes the final algorithm sub-quadratic.

In this section, we improve upon this result by using a more specialized data structure which we will call *twister tries*. This data structure allows finding a likely closest pair, removing it, and inserting it back in constant time. Being able to do this results in a linear running time of the overall clustering algorithm.

5.1 Data structure

The twister tries data structure consists of a collection of prefix trees, each with an accompanying splitmap (see fig. 2). Furthermore, clusters are represented by what we will call an *element*. Next, we will provide a description of each of these constituents, together with their respective invariants which are maintained between each insertion and removal step.

- The prefix trees or tries are like the ones used in LSH Forest. The outcome of the hash function evaluations are the labels encountered on the arcs when following the path from the root node to the leaf. In addition, each node has a pointer to its parent. Furthermore, each leaf of the trie maintains a list of pointers to the elements (see below) that this leaf corresponds to.

  Definition 9 (splitpoint). *We will use the term splitpoint to refer to a node in the trie which has more than one child or a leaf which refers to more than one element.*

  Definition 10 (depth). *The depth $\hat{d}(n)$ of a node n in the trie is the number of edges from the root node to n.*

  Hence, the root node is at depth 0 and the leaves are at depth k_m.

- For each trie, there is also a *splitmap*. This map is essentially a list fixed to the maximum height of the trie. Each item in the splitmap is itself a linked list of pointers to all splitpoints of the trie at the same level. The splitpoints also keep a pointer to the node in the linked list which points back to them.

- Finally, the elements represent the clusters which are currently in the system. Each element contains pointers to the leaf in each trie which represents this cluster and to the node in the linked list of the leaf which points back to the element.

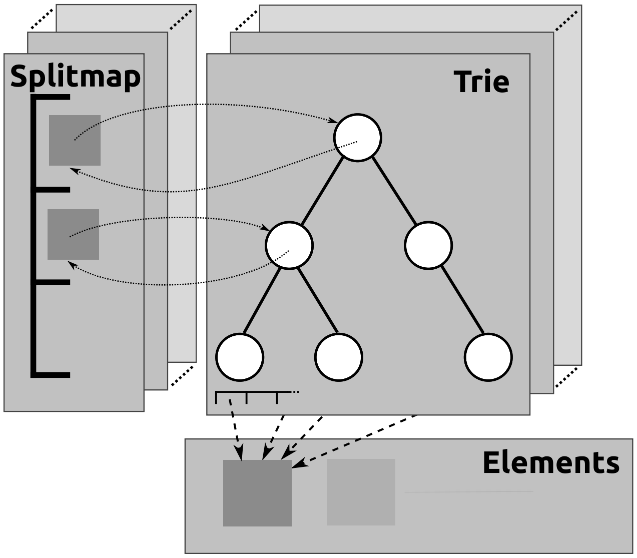

Figure 2: Parts of the twister tries data structure

For our further discussion, we need to introduce the notion of a joinpoint.

Definition 11 (joinpoint). *Given two subsets I and J, any lowest splitpoint (in any of the tries) having the records representing I and J connected to itself or its descending leaves is called a* joinpoint *(jp) of I and J.*

Note that this implies that a leaf is a joinpoint if both records are connected to it.

5.2 Operations

The clustering algorithm works by first inserting all items, represented by one-point clusters, into the tries. Then, at each iteration a pair of clusters is chosen. This pair is removed from the tries and merged into a new cluster. Finally, this merged cluster is re-inserted into the tries. We sometimes refer to the repeated part as the twisting phase. The detailed operations are as follows:

Insertion: Inserting or adding a cluster (containing one item or more) happens in a similar way to how a point is inserted in LSH Forest, with the addition that the splitmaps and the elements must be maintained. For each trie, the insertion is handled independently. At each depth in each trie, the associated hash function chosen from the k-proportional family is evaluated and the edge corresponding to the outcome is followed. In the event that the edge does not yet exist, it is created. Furthermore, if the node already had exactly one child before, the node is added as a splitpoint to the splitmap. Note that this can happen once at the most. When the leaf depth is reached, the element representing the cluster is added to the list that the leaf maintains and a pointer to the leaf is added to the element.

Selection: The algorithm needs to select two clusters for merging. First, the depth of the deepest entry in any of the splitmaps (or, in other words, the depth of the lowest splitpoint in any trie) is determined. Then, one of the splitpoints at this height, which is also necessarily a joinpoint, is considered. If this is a leaf node, the selected clusters are the first two clusters connected to this node. If this is an internal node, then the selected clusters are the ones connected to the descendant leaves of two of the children of this node selected at random.

Removal: The removal of a cluster from the trie is only possible if its corresponding element has already been obtained (for instance, from the selection procedure). Furthermore, it is basically the reverse operation of an insert. The element contains pointers to one leaf in each trie that represents the cluster. For each trie, first the element is removed from the leaf and the leaf from the element. Then unused arcs and nodes are disposed of while traversing up through the trie. During the process, potentially one splitpoint may lose a child. If that node had exactly two children, it is no longer a splitpoint and it needs to be removed from the splitmap.

Merge: After the removal of the clusters, they have to be merged. This is done by taking the union of the clusters. Then a new element representing this union is again inserted into the tries, using the standard insertion procedure.

5.3 Complexity

Twister tries perform the clustering in linear time and have a linear space complexity with regard to the number of items n. To see why this holds, we first have to note that each of the operations defined in the previous section are performed in constant time.

Insertion is a constant time operation since the size and number of the tries is fixed and the evaluation of the hash function is performed in constant time. Adding a node as a splitpoint to a splitmap can only happen once and essentially involves appending a pointer to a linked list. Adding the element to the leaf is also done in $O(1)$. Selection, removal, and merging are done in constant time for largely the same reasons.

With all these operations in constant time, it is easily seen that the total time complexity becomes linear. There are n insert operations, followed by $n-1$ times a selection, two removals, and a merge. Hence, the time complexity is $O(n)$.

To show the linear space complexity, we have to show that (1) the size of a trie is linear in function of the number of items and (2) the size occupied by the splitmap is limited by the number of items.

The first fact can be easily seen by observing the worst case scenario. If all n elements have a different hash value assigned during the first hashing upon insert, then there are $bk_m n$ arcs and $bk_m n + 1$ nodes. Furthermore, the element lists at the leaves occupy space. However this is limited to the number of elements.

The second fact is also easily shown. A leaf of the trie can refer to multiple elements, and each element is only connected with one leaf. Hence, there are no more leaves as elements. By induction on the number of leaves, one observes that their number is greater than the number of splitpoints in the trie. So, in conclusion, there are always less splitpoints than elements. And hence the size of the splitmap is limited by the number of elements.

One final remark is related to the number of children per node. In theory, one could define the hash functions such that they have only two outcomes. However, in practice, a hash function will produce a larger, but finite number of results. Hence, one might choose to let the number of children for each node be equal to this amount. Since this amount is fixed, it does not affect the theoretical time and space complexity.

5.4 Correctness

When the twister tries algorithm is used for clustering, it makes at each iteration a decision about which clusters to join into a bigger cluster. This decision is based on an approximation, and hence one could wonder how good this approximation is. When selecting the next two clusters to join from two pairs of clusters (I, J) and (K, L), the standard AHC algorithm will always choose the clusters with the smallest distance between them according to the used linkage (see section 2). The proposed algorithm will, however, base its choice on the position of the lowest joinpoints of the cluster pairs in any of the tries (see section 5.2).

In this section, we will develop an expression for the probability that the twister tries algorithm will decide to choose a pair of clusters (I, J) instead of the pair (K, L), given their distances.

For the definitions and theorems in this section, we assume a twister trie consisting of b tries with height k_m and four clusters $I, J, K,$ and L, inserted using the k-proportionally sensitive family $\mathcal{H}_A = \{h_{A_1} \dots h_{A_n}\}$, created according to definition 8 with respect to distance metric d.

First, we find an expression stating the probability that the depth of a joinpoint in a single trie is x or smaller than x.

Given the clusters I and J, with $p = \Pr[h_{A_i}(I) = h_{A_i}(J)] = 1 - kd(I, J)$, for $x \in \mathbb{N}_{\leq k_m}$

$$\Pr[\hat{d}\,(\text{jp of } (I, J)) = x] = \begin{cases} p^x(1-p) & \text{if } x < k_m \\ p^x & \text{if } x = k_m \end{cases}$$

$$\Pr[\hat{d}\,(\text{jp of } (I, J)) < x] = \sum_{g=0}^{x-1} \Pr[\hat{d}\,(\text{jp of } (I, J)) = g]$$

Now, we denote the lowest joinpoint of two clusters I and J in any of the tries as $m(I, J)$. The probability that $m(I, J)$

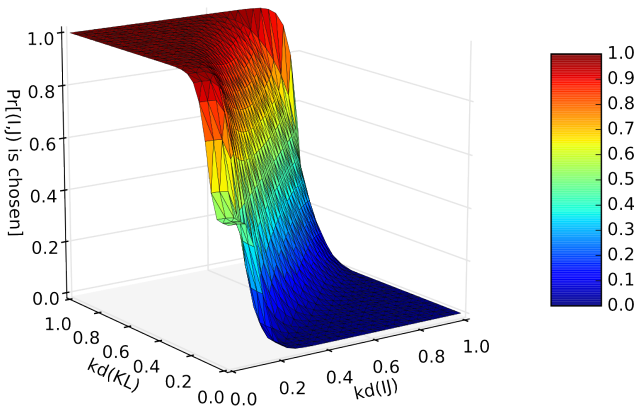

Figure 3: The probability that (I, J) is chosen over (K, L) for twister tries with $b = 25$ and $k_m = 25$

is equal to d can be expressed as

$$\Pr[m(I,J) = d] = \sum_{i=1}^{b} \binom{b}{i} \Pr\left[\hat{d}\,(\text{jp of } (I,J)) < d\right]^{b-i} * \Pr\left[\hat{d}\,(\text{jp of } (I,J)) = d\right]^{i}$$

and for inequality we find

$$\Pr[m(I,J) < d] = \left(\sum_{i=0}^{d-1} \Pr\left[\hat{d}\,(\text{jp of } (I,J)) = i\right]\right)^{b}$$

Using the above, we can now find an expression for the probability that (I, J) is chosen. The pair (I, J) gets chosen if it has a joinpoint deeper than (K, L) or in half of the cases when their joinpoint is at the same hight (because of random selection).

$$\begin{aligned}
&\Pr[(I,J) \text{ is chosen}]\\
&\quad= \Pr[m(I,J) > m(K,L)] + \frac{1}{2}\Pr[m(I,J) = m(K,L)]\\
&\quad= \sum_{d=0}^{k_m}(\Pr[m(I,J) = d > m(K,L)]\\
&\qquad\quad+ \frac{1}{2}\Pr[m(I,J) = d = m(K,L)])\\
&\quad= \sum_{d=0}^{k_m}(\Pr[m(I,J) = d]\Pr[m(K,L) < d]\\
&\qquad\quad+ \frac{1}{2}\Pr[m(I,J) = d]\Pr[m(K,L) = d])
\end{aligned}$$

This formula is hard to comprehend, so we illustrated its meaning by graphing the probability for $k_m = 25$ and $b = 25$ (see fig. 3). What we can see from this formula (and the graph) is that if $0 < d(I,J) < d(K,L)$, then $\Pr[(I,J) \text{ is chosen}] > \Pr[(K,L) \text{ is chosen}]$. Furthermore, the graph looks more or less like a 3-dimensional S-curve. From the graph, one can also see that if $d(I,J)$ and $d(K,L)$ are close to 0, then twister tries will be unable to distinguish between them, which is observable by the small plateau in the front of the figure. Graphs for other settings are similar, but the slope is steeper and the plateau smaller when higher tries are used. The plateau gets larger when the number of tries is increased.

We also observe that independent of the choice of parameters,

$$\lim_{|d(I,J)-d(K,L)|\to 0} Pr[(I,J) \text{ is chosen}] = 0.5$$

This means that no matter what setting is used, if the difference between the distances is arbitrarily small, the twister tries will be unable to distinguish between them This phenomenon is also observable in LSH in general and is caused by the fact that it is impossible to tell very close points apart by using hashing. However, we expect that the practical effect of this phenomenon will be small since a) in half of the cases the algorithm will, by chance, still choose the right option and b) if the wrong option is taken, it is still a pair of nearly correct clusters that gets joined.

This phenomenon can also be observed in other approximate methods for AHC. In the work by Gilpin et al. [8], points that fall within the same cone after quantization cannot be distinguished. Similarly, the work by Patra et al. [18] altogether avoids clustering close points by putting them in one cluster connected to a 'leader'.

5.5 Implementation considerations

The data structure and operations as detailed in the previous sections can be implemented as they are presented. One can, however, also implement certain optimizations to get a faster execution speed or save memory. We applied some of these in the implementation used for the evaluation, but left others out since they would be data-dependent and hence distort the evaluation results. The first important optimization that we applied is related to the merging of two clusters and re-insertion. If we start from the subsets $I = (i_1, \ldots, i_{|I|})$ and $J = (j_1, \ldots, j_{|J|})$ and merge them together, we obtain the set $I \cup J = (i_1, \ldots, i_{|I|}, j_1, \ldots, j_{|J|})$. Now, according to the definition of h_A, the values of $h_{A_n}(I)$ and $h_{A_n}(J)$ are possible answers for $h_{A_n}(I \cup J)$. So we could shortcut the computation of $h_{A_n}(I \cup J)$ by selecting either $h_{A_n}(I)$ or $h_{A_n}(J)$ as its outcome. To decide which one to choose, we observe that given $h_{A_n}(I \cup J) = h_{A_n}(I)$ or $h_{A_n}(J)$, then $\Pr[h_{A_n}(I \cup J) = h_{A_n}(I)] = \frac{|I|}{|I \cup J|}$. Hence, we shortcut the evaluation of $h_{A_n}(I \cup J)$ by randomly choosing between $h_{A_n}(I)$ and $h_{A_n}(J)$ with probability $\frac{|I|}{|I \cup J|}$ and $\frac{|J|}{|I \cup J|}$, respectively. What this means is that we do not evaluate $h_{A_n}(I \cup J)$ at all, but instead re-use one of the previously calculated hash values. This also means that there is no need to keep the actual data in memory. The previous observation also leads to the following. In each trie, above the splitpoint of (I, J), $h_{A_n}(I)$ and $h_{A_n}(J)$ are evaluated to the same value, and hence it is not necessary to choose between them. Therefore, instead of removing the two points completely, they can be removed up to the splitpoint and then re-inserted, starting from the splitpoint.

As suggested in section 5.3, a hash function can have many results. A possible naive implementation would allocate storage space for each of these in every node. Instead, we used a dictionary that only stores the ones that are in use.

Furthermore, it would be possible to save hash evaluations by not evaluating if there is no need for it. In particular, when a chain of nodes in the trie is only created for one cluster, the results of its hash evaluations will not be used and hence their calculation is not needed. However, when this cluster gets merged with another cluster and re-inserted into the trie, it may be necessary to evaluate them at that point. We decided to not apply this optimization, since its gain would be highly dependent on the characteristics of the dataset, which would make our evaluation difficult to comprehend. Furthermore, it would make it impossible to separate the overhead caused by the different parts of the algorithm, and it becomes difficult to guarantee the correctness of the implementation.

A further related saving would be to use Patricia trees [14], as was proposed for LSH Forest [2]. The idea is to compress chains of internal nodes with only one child, such that they are represented by only one node. This was not implemented, because it could skew the results of the evaluation.

5.5.1 Parallel and distributed algorithm

The algorithm spends the largest amount of its running time on computing the hash functions for the clusters when they are inserted into the trie. Hence, we can accelerate the whole process by computing these hash functions in parallel. As their evaluation completely independent, data parallelism can be exploited. We applied this technique since it only leads to a linear acceleration and does not skew the results.

Another way that parallelism or distribution can be exploited is as follows. Each peer or processor manages one trie as well as a map from each elementID to the leaf nodes of the trie (replacing the elements at the bottom). A peer with an entry at the lowest location in the splitmap now becomes the leader. The leader merges the corresponding clusters in the trie and reports the change to the other peers, which modify their tries accordingly. Then the process repeats until the whole clustering is done. Note that the merge steps of the algorithm have to be taken in sequential order. One could implement some form of speculative execution where several possible branches are followed and only the one taken is retained. However, the performance of this approach would be very dependent on the dataset.

6. DENDROGRAM COMPARISON

In the previous section, we described the twister tries algorithm and some of its theoretical properties. The result of the algorithm is a dendrogram (i.e. a tree representing the order in which the items are clustered together (see also definition 3)). The aim of our algorithm is to approximate the outcome of the standard AHC method (see section 2). Hence, in order to evaluate its performance, we would need to compare the dendrogram of the twister tries algorithm with the one that the standard algorithm produces. It should be noted that the aim of this work is not to evaluate the semantic correctness af the dendrogram. In other words, we do not evaluate whether the obtained dendrogram corresponds to the logical structure which might exist in the dataset.

There are several ways to compare dendrograms with each other. When comparing small dendrograms, we used the joining distance ratio (section 6.1) by Kull and Vilo [13], and the B_k metric (section 6.2) proposed by Fowlkes and Mallows [7]. For large dendrograms, on the other hand, we could not find any suitable metric in the literature. When attempting to define a metric independent of an exact dendrogram, we run into the issue that other proposed methods for large-scale hierarchical clustering do not produce a complete dendrogram, rendering comparison meaningless.

6.1 Joining Distance Ratio

Considering standard AHC as an optimization problem, the purpose of an algorithm is to minimize the distance of the joining partitions for every step, meaning that for each step, the algorithm selects the cluster pairs with the smallest distance. To test the performance of the approximate algorithm, a measurement of how it minimizes the joining distance could be considered.

The joining distance ratio is defined as follows:

Definition 12 (Joining distance ratio (JDR))**.** *The joining distance ratio (JDR) is the proportion of the sum of the joining distance of each step of the standard AHC dendrogram and the sum of joining distances of the approximate algorithm.*

$$JDR = \frac{\sum\limits_{I,J \in AHC\ dendrogram} d_A(I,J)}{\sum\limits_{I,J \in approximate\ dendrogram} d_A(I,J)}$$

Since the standard AHC algorithm is designed to choose the pair with minimal distance at every iteration and the approximate algorithm may select a non-minimal one, the joining distance ratio is expected to be below 1.

Note, however, that there is a possibility for the standard AHC algorithm to result in different values for the joining distance. This can happen when two or more pairs of clusters are at exactly the same distance and the algorithm chooses one of these at random. We ignore this possibility since it won't have a large effect in practice. Note that the JDR is insensitive to insignificant re-orderings of the dendrogram, which is not true for many other metrics, including the measurement by Fowlkes and Mallows introduced in the next subsection.

6.2 Fowlkes and Mallows measurement

Fowlkes and Mallows devised the B_k measurement based on the idea that similar dendrograms should contain similar subtrees. Their measurement was not specifically introduced for measuring the quality of approximate dendrograms, but is still applicable. An essential part of the B_k measurement is based on the proportion of each subtree from the standard AHC dendrogram that is preserved by the approximate version.

To compute B_k, suppose that there are two hierarchical clustering trees, A_1 and A_2 both with the same n items, and a cut $k = 2, \ldots, n-1$, which is the height at which the dendrograms are cut. Cutting at height k produces k subtrees, which can be seen as a partition of the items in k subsets. Then, for each $k \leq n-1$, compute the match matrix $M = [m_{ij}](i = 1, ..., k; j = 1, ..., k)$, where m_{ij} is the number of items that are common for the i-th partition of $A1$ and j-th partition of $A2$. Then, for each k , B_k is calculated as follows:

Definition 13 (Fowlkes and Mallows – B_k)**.**

$$B_k = T_k / \sqrt{P_k Q_k}, \qquad T_k = \sum_{i=1}^{k} \sum_{j=1}^{k} m_{ij}^2 - n,$$

$$P_k = \sum_{i=1}^{k} \left(\sum_{j=1}^{k} m_{ij} \right)^2 - n, \quad Q_k = \sum_{j=1}^{k} \left(\sum_{j=i}^{k} m_{ij} \right)^2 - n.$$

Next, B_k is plotted in function of k. Then, we have $0 \leq B_k \leq 1$ and only when all k clusters in each tree are exactly the same, $B_k = 1$. B_k can be interpreted as "the sum of all pairs of objects of those pairs that have matching cluster assignments"[7].

When an algorithm produces random clusterings, the items are randomly assigned to the clusters. The B_k value which is obtained in this case is called the expected B_k, and denoted $E(B_k)$. Besides this mean value, one can also calculate the variance of the random assignment $var(B_k)$. The details about their derivation can be found in the appendix of the paper by Fowlkes and Mallows [7].

To interpret the value, one compares the obtained B_k with $E(B_k)$. If B_k is systematically not within $E(B_k) \pm 2\sqrt{var(B_k)}$, then the two trees are similar, and hence the approximate algorithm showed a good performance.

6.3 Quality of large dendrograms

The dendrogram comparison methods described in the above subsections can be used to assess the quality of a dendrogram produced by an approximate method by comparing it with the outcome of the standard AHC algorithm. However, for large datasets, we were not able to find a meaningful way to compare our work with the closely related work of Gilpin et al. [8] and Patra et al. [18]. There are multiple issues when using the metrics proposed above and other metrics found in the literature, for large datasets. The issues include:

- *An exact clustering must be available.* The computation of the dendrogram using an exact algorithm takes an unacceptable amount of time for a large dataset. The problem is that the computation of an exact dendrogram is inherently serial since each clustering is dependent on the ones that came before. Hence, the applicability of methods that require an exact dendrogram is limited to relatively small dendrograms. This issue affects both the JDR and B_k measure described above. It is also a show-stopper for many other related metrics found in the literature, such as the relatively popular (but inferior, see [7]) Rand index, which measures "the number of similar assignments of point-pairs normalized by the total number of point-pairs" between two partitions [20].

  Based on the JDR, one could devise a way to compare two approximations with each other. For dendrograms A and B, both approximate dendrograms, one could calculate the joining distance. Because of the dependency of the joining distances on the particular dataset, it would still be difficult, if not impossible, to interpret these sums. Yet, we can argue that the dendrogram with the lowest joining distance has the highest quality.

- *The computation of the metric is infeasible.* Even if one were able to compute the exact dendrogram, there is still another problem for measurements based on sub-tree conservation, like the B_k measurement from the previous section. These measurements have a cubic computational complexity, rendering them infeasible to compute for large datasets.

- *Generation of incomplete dendrograms.* The methods that we seek to compare with do not produce complete dendrograms. In particular, Gilpin et al. [8] allow the leaves of the dendrogram to contain multiple points. Additionally, Patra et al. [18] do not continue the clustering until only one cluster is left. In the worst case, both methods could generate a single cluster containing all the points.

  This immediately leads to the fact that metrics like the JDR, the simplified method to compare dendrograms presented above, and B_k are no longer usable, since complete dendrograms are assumed to use these methods. It might seem a solution to cut the dendrograms at some height and use measurements developed for partitions. For instance, the B_k metric and the Rand index can be computed for a single level of the dendrogram. However, this would not give a complete view of the quality of the clustering. In particular, the B_k should be observed at each level of the hierarchy in order to interpret its quality, and the same goes for the Rand index. [7] The general problem is that a low-quality leaf cluster (e.g., one containing many points that are not very close) may produce a good average distance at a higher level, leading to good results when evaluated. However, the incomplete clustering cannot be cut at a lower level, and hence a comparison cannot be made. In effect, the approaches become incomparable.

As mentioned above, the works that we seek to compare with create large, incomplete clusterings. In order to evaluate their results, the authors worked as follows: Gilpin et al. cluster real-world datasets and measure the semantic quality of the clustering. This approach has a major drawback. What is measured is whether the algorithm is suitable for the clustering of the specific dataset for a given purpose, and not whether it performs a clustering similar to the standard AHC. This means that one cannot claim that the algorithm works where AHC does. Or, as stated by the authors: "the accuracy of our results is typically as good [as regular agglomerative clustering] and can sometimes be substantially better". Patra et al. evaluate their work by comparing the obtained dendrogram with the results of an earlier developed approximate method, using the Rand index at a given height. This approach displays several of the issues illustrated above. Therefore, we do not approve of this way of comparing the outcomes.

We do strongly agree that a comparison between our proposed approach and other methods would be highly beneficial. Because we could not determine a meaningful way to perform them, however, we did not include such types of comparisons.

7. EMPIRICAL EVALUATION

Using metrics for the dendrogram comparison from the previous section, we now empirically evaluate how the algorithm performs. These are the questions which we wish to answer:

- How many tries does the algorithm need and how high should these be in order to get a clustering comparable to the traditional AHC algorithm, using a typical dataset?

- The theoretical time and space complexity are linear. Does this promise hold when using the algorithm in practice, or will constant factors render the algorithm useless? And how does the algorithm scale for a very large number of items?

In order to answer these questions, we implemented the twister tries algorithm as suggested in section 5.5. Then, we selected two datasets. For these datasets we defined a distance metric and a proportionally sensitive family of hash functions. Then we ran the twister tries algorithm with varying settings and portions of data.

The experiments were executed on a OpenJDK 8 64-bit Server VM. The Java VM ran on hardware with two Intel Xeon E5-2670 processors (totaling 16 multi-threaded cores) and was limited to use a maximum of 120 GB RAM. When interpreting running time and memory results, one should keep in mind that we used the Java virtual machine, which implements garbage collection and scheduling. Hence, despite our attempts to minimize measurement error, small variations are possible due to the runtime behavior.

7.1 Datasets and proportionally sensitive hash functions

In order to evaluate our algorithm, we chose datasets with different characteristics. The first dataset was the Thomson Reuters Text Research Collection (TRC2).[1] This corpus contains roughly 1.8 million news stories and is about 2.7 GB large. We preprocessed the data such that each item in our dataset was a set of words representing the article. During the preprocessing, we split the article text on whitespace. Then, for each word we removed punctuation marks, converted to lowercase, and applied Porter stemming.[2] From the resulting set, we removed stop words, single characters and numbers. Articles that resulted in an empty set were ignored, which finally resulted in 1.68 million sets, each representing one article. As a metric for the distance between two articles, we used the Jaccard distance between their representing sets. The min-hash family was chosen for the locality-sensitive hashing. The permutation step of the min-hash was implemented using Rabin hashing [3].

Secondly, we chose the $cifar$-10 dataset[3], comprising 60,000 small images. We made this choice because it contains a reasonable amount of data and we could apply a different distance metric. The images were 32x32 pixels large and encoded using the RGB color model. Hence, each image was represented by a vector of 3,072 integers. Furthermore, the images were categorized in terms of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. There were 6,000 images belonging to each of these classes in the dataset. We measured the distance between two images as the cosine distance (i.e. the normalized angle) between their respective vectors. The locality-sensitive hashing was implemented using random hyperplane hashing [5].

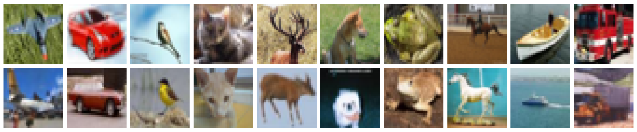

Figure 4: 20 sample images from the $cifar$-10 dataset.

7.2 Experiments and Results

Using the datasets described above, we devised several experiments aimed at answering the questions at the beginning of this section. There were more experiments using the TRC2 dataset, because it contains more data points.

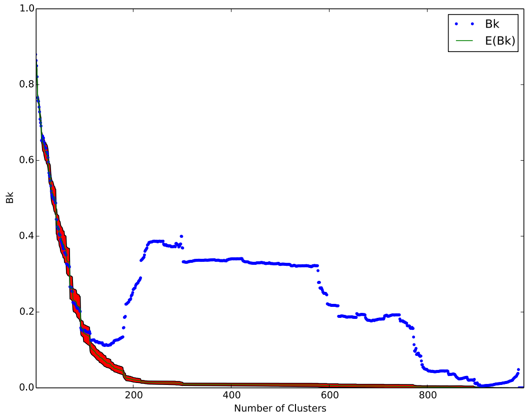

Figure 5: Fowlkes and Mallows – B_k measurement for 1000 news stories of the TRC2 dataset inserted into twister tries with $b = 1$ and $k_m = 1$.

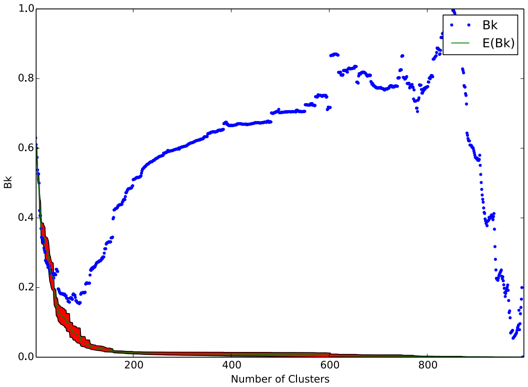

Figure 6: Fowlkes and Mallows – B_k measurement for 1000 news stories of the TRC2 dataset inserted into twister tries with $b = 120$ and $k_m = 120$.

7.2.1 Determining b and k_m

We started with four experiments to determine what the settings for the parameters b and k_m should be in order to obtain a reasonable clustering. For these experiments, we took a set of 1000 items and inserted them into twister tries with between 5 and 125 tries with a height between 5 and 125. For each of these settings, we obtained an approximate dendrogram. Next, the 1000 items were clustered using an exact algorithm (the fastcluster algorithm mentioned in section 3).

For the first two experiments, we applied the Fowlkes and Mallows B_k measurement and used the TRC2 dataset. We selected two settings ($b = 1$, $k_m = 1$ and $b = 120$, $k_m = 120$) and plotted the results (See figs. 5 and 6). The plots also contain the expected B_k, and the region surrounding it indicates two standard deviations. For the third and fourth experiments, we calculated the JDR in function of the height and number of tries for both datasets (see figs. 7 and 8).

The first observation seen from all of these figures is that an increase in the number of tries or their height led to a higher quality of the dendrogram. However, the relative

[1] http://trec.nist.gov/data/reuters/reuters.html

[2] http://tartarus.org/martin/PorterStemmer/

[3] http://www.cs.toronto.edu/~kriz/cifar.html

gain in quality became smaller the more tries there were and the larger they were. In particular, from the B_k diagrams it is clearly apparent that the quality increased when more and higher tries were used. The plot in fig. 5 shows that the B_k value remained close to $E(B_k)$, meaning that the clustering was not much better than random. In the other B_k figure (fig. 6), we see that the B_k only stayed close to the expected value when very few clusters were made. The chart is in accordance with the observations of Fowlkes and Mallows [7] regarding very similar dendrograms.

Secondly, from the JDR plots we notice that we obtained a better quality for the Reuters than for the $cifar$-10 dataset, which means that the characteristics of the data and the hash function used influenced the shape of the JDR diagram. Finally, we note that the increase in quality, in terms of JDR, became relatively small when crossing a certain threshold. For the TRC2 dataset, this happened as soon as the height and number of trees became higher than 20, while for the $cifar$-10 dataset this point was around (20, 60). From these observations, we won't make any general recommendations. However, we will use the aforementioned sizes and heights to evaluate the runtime and memory use when clustering more items.

7.2.2 *Time and Space Complexity*

To answer the second question, we set up experiments in which we fixed the parameters of the twister tries and clustered larger and larger portions of data. For the $cifar$-10 dataset, we inserted up to 60,000 images into the data structure with 20 tries of height 60. We stopped at 60,000 because there are no more items in the dataset. For the TRC2 data, we inserted multiples of 50,000 news stories into the twister tries until the whole dataset was fed into the data structure. For these experiments we measured the time needed to complete the insertion of data (the time needed to compute all the hashes and add it into the prefix trees) and the twisting phase (iteratively selecting, removing, merging, and re-inserting clusters). The amount of memory used was measured at the point when all data had been inserted into the tries. To minimize measurement error, these experiments were executed on the server while there were no other significant processes going on. Further minimization of measuring error was achieved by executing the running time and memory usage measurements separately to minimize measurement error due to the measuring instruments. We noticed that there was not much variation in the measurements and hence did not attempt to measure deviations; the reported figures are averages of three runs.

The outcome of the timing experiments can be found in figs. 9 and 10 for the TRC2 and $cifar$-10 dataset, respectively. For the $cifar$-10 dataset, we note that the plot indicates a linear relation between the number of images clustered and the time needed. Furthermore, we note that the time needed for clustering the whole dataset was about 160 seconds. Lastly, it can be seen that most of the time was spent on inserting the data into the tries. Also, the TRC2 plot is more or less linear. We note that the measurements between 20,000 and 50,000 points are clustered a bit faster compared to the other measurements. We could not find a clear explanation for this and it was also observed after repeated measurements. Most likely, there is a optimization that the JVM does not start using with the smaller amount of data. The fact that the measurements for larger datasets do not reflect this is likely caused by the amount of memory used. We observe that the algorithm was slightly slower when more than 16 GB of memory was used, which coincides with the size of a DIMM module on the server. From the plot, it can also be seen that for the TRC2 dataset, relatively more time was spent in the twisting phase when compared to the results for the $cifar$-10 dataset.

The memory consumption was plotted in figs. 11 and 12. What we observe from the figures is that the twister tries scaled well for large datasets. Moreover, both time and space complexity rose in a linear fashion with respect to the input size.

8. CONCLUSIONS

Our experiments showed that the twister tries algorithm is able to create clusterings of a reasonable quality. Furthermore, we showed both analytically and by using practical experiments that time and space complexity scale linearly in relation to the number of items clustered. The largest dataset that we clustered consisted of about 1.7 million news articles. Using the Jaccard dissimilarity of representing sets as a distance metric, the time needed for clustering this amount of data was about 14 minutes.

There are still several open questions to be researched. For one, it would be interesting to see whether it is possible to develop a scalable metric for dendrogram comparison. It would also be interesting to determine whether k-proportionality is a necessary condition for using twister tries; we only showed that it is a sufficient one. If this condition could be relaxed, it could be researched whether it is possible to broaden the applicability to other metrics and LSH functions. Further research could also investigate how this algorithm could be parallelized and distributed, while still maintaining reasonable performance, to further increase the size of the datasets that can be clustered. Another important open question is to find a good measurement that can determine the quality of a large, perhaps incomplete dendrogram within a reasonable time.

Acknowledgments

The authors would like to thank the Department of Mathematical Information Technology and the Industrial Ontologies Group of the University of Jyväskylä for making this research possible. This research was also financed in part by the TEKES N4S SHOK in collaboration with Steeri Oy. We would also like to thank the reviewers and copy editor for their thorough comments and suggestions, which helped us to improve the paper.

Furthermore, it has to be mentioned that the implementation of the software was greatly simplified by Google's Guava library, the Apache Commons Math$^{\mathrm{TM}}$ library, and the Rabin hash library by Bill Dwyer and Ian Brandt. In the evaluation code we used the fastcluster, NumPy, SciPy, and Matplotlib libraries. For the experiments, we used the "Thomson Reuters Text Research Collection (TRC2)".

The contributions to the article are as follows: Michael Cochez is the main author of the paper. He invented and implemented the twister tries data structure and algorithm. In addition, he worked on the optimizations, theoretical analysis and experimental evaluation. Mou Hao contributed the correctness analysis and did the majority of the work for the dendrogram comparisons.

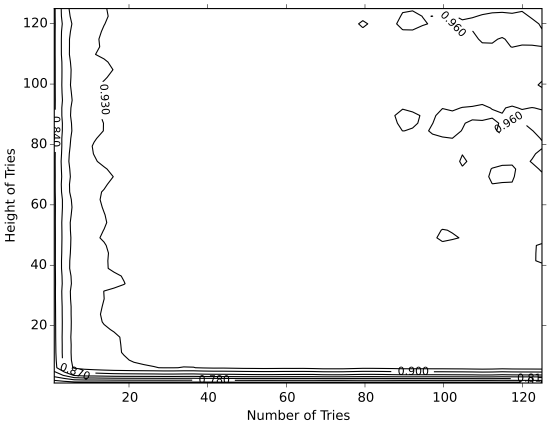

Figure 7: Influence of the number of tries and their height on the Joint Distance Measure of the dendrogram for 1000 news stories of the TRC2 dataset.

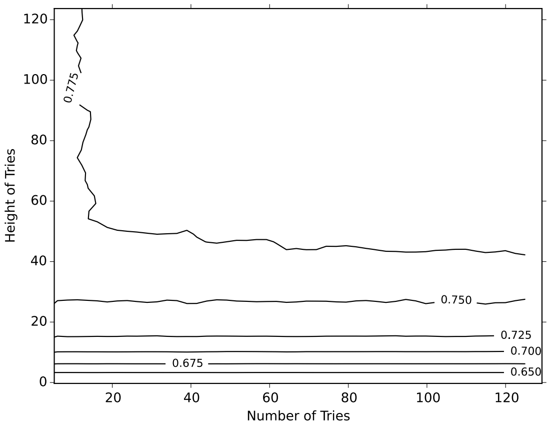

Figure 8: Influence of the number of tries and their height on the Joint Distance Measure of the dendrogram for 1000 images of the $cifar$-10 dataset.

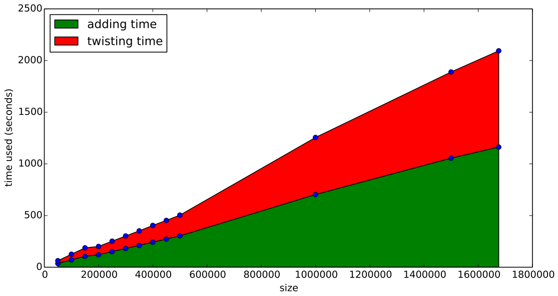

Figure 9: Runtime for adding and twisting TRC2 stories using twister tries with $b = 20$ and $k_m = 20$

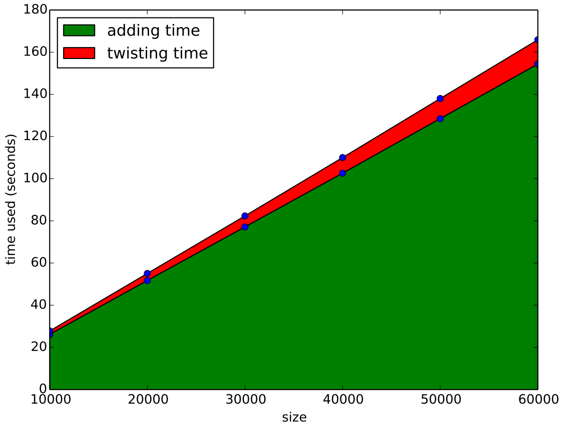

Figure 10: Runtime for adding and twisting $cifar$-10 images using twister tries with $b = 20$ and $k_m = 60$

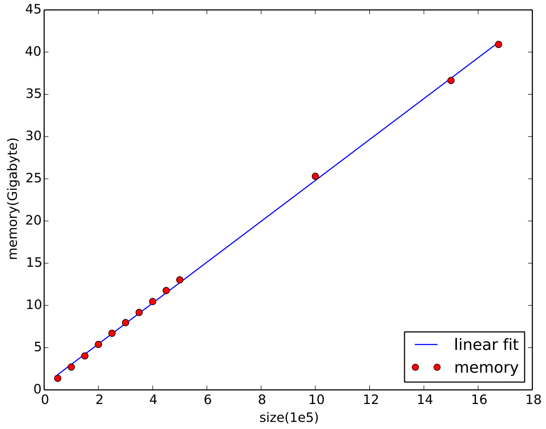

Figure 11: Memory usage for adding TRC2 stories using twister tries with $b = 20$ and $k_m = 20$

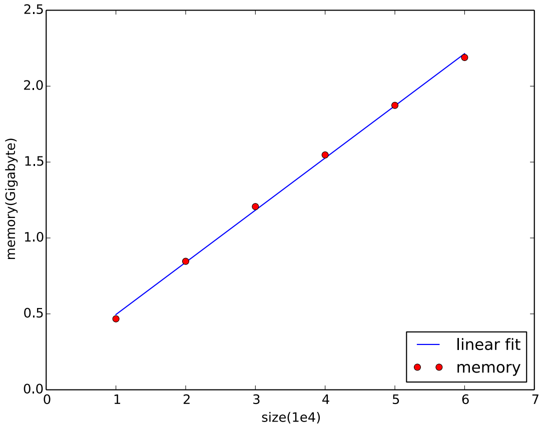

Figure 12: Memory usage for adding $cifar$-10 images using twister tries with $b = 20$ and $k_m = 60$

9. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.

[2] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660. ACM, 2005.

[3] A. Broder. Some applications of rabin's fingerprinting method. In R. Capocelli, A. Santis, and U. Vaccaro, editors, *Sequences II*, pages 143–152. Springer New York, 1993.

[4] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[5] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[6] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Exp. Algorithmics*, 5, Dec. 2000.

[7] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78(383):553–569, 1983.

[8] S. Gilpin, B. Qian, and I. Davidson. Efficient hierarchical clustering of large high dimensional datasets. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, CIKM '13, pages 1371–1380, New York, NY, USA, 2013. ACM.

[9] I. Gronau and S. Moran. Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters*, 104(6):205–210, 2007.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[11] H. Koga, T. Ishibashi, and T. Watanabe. Fast hierarchical clustering algorithm using locality-sensitive hashing. In E. Suzuki and S. Arikawa, editors, *Discovery Science*, volume 3245 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, 2004.

[12] N. Kriege, P. Mutzel, and T. Schäfer. SAHN clustering in arbitrary metric spaces using heuristic nearest neighbor search. In S. Pal and K. Sadakane, editors, *Algorithms and Computation*, volume 8344 of *Lecture Notes in Computer Science*, pages 90–101. Springer International Publishing, 2014.

[13] M. Kull and J. Vilo. Fast approximate hierarchical clustering using similarity heuristics. *BioData mining*, 1(1):9, 2008.

[14] D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[15] D. Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378*, 2011.

[16] D. Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software*, 53(9):1–18, 5 2013.

[17] T.-D. Nguyen, B. Schmidt, and C.-K. Kwoh. SparseHC: A memory-efficient online hierarchical clustering algorithm. *Procedia Computer Science*, 29(0):8 – 19, 2014. 2014 International Conference on Computational Science.

[18] B. Patra, N. Hubballi, S. Biswas, and S. Nandi. Distance based fast hierarchical clustering method for large datasets. In M. Szczuka, M. Kryszkiewicz, S. Ramanna, R. Jensen, and Q. Hu, editors, *Rough Sets and Current Trends in Computing*, volume 6086 of *Lecture Notes in Computer Science*, pages 50–59. Springer Berlin Heidelberg, 2010.

[19] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*, chapter 3. Finding Similar Items, pages 71–128. Cambridge University Press, 2012.

[20] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[21] Z. Rasheed, H. Rangwala, and D. Barbará. Efficient clustering of metagenomic sequences using locality sensitive hashing. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, SDM '12, pages 1023–1034, Philadelphia, PA, USA, 2012. SIAM.