Karl-Franzens-Universität Graz & Technische Universität Graz

SPEZIALFORSCHUNGSBEREICH F 003

OPTIMIERUNG

und

KONTROLLE

Projektbereich

DISKRETE OPTIMIERUNG

Oswin Aichholzer

Clustering the Hypercube

Bericht Nr. 93 – Dezember 1996

Clustering the Hypercube

Oswin Aichholzer*
Institute for Theoretical Computer Science
Graz University of Technology
Klosterwiesgasse 32/2, A-8010 Graz, Austria
e-mail: oaich@igi.tu-graz.ac.at

Abstract

In this paper we consider various clustering methods for objects represented as binary strings of fixed length d. The dissimilarity of two given objects is the number of disagreeing bits, that is, their Hamming distance. Clustering these objects can be seen as clustering a subset of the vertices of a d-dimensional hypercube, and thus is a geometric problem in d dimensions. We give algorithms for various agglomerative hierarchical methods (including single linkage and complete linkage) as well as for two-clusterings and divisive methods.

We only present linear space algorithms since for most practical applications the number of objects to be clustered is usually to large for non-linear space solutions to be practicable. All algorithms are easy to implement and the constants in their asymptotic runtime are small. We give experimental results for all cluster methods considered, and for uniformly distributed hypercube vertices as well as for specially chosen sets. These experiments indicate that our algorithms work well in practice.

1 Introduction and Notation

In this paper we consider various clustering methods for objects represented as binary strings of fixed length d. The dissimilarity of two given objects is the number of disagreeing bits, that is, their Hamming distance. Clustering these objects can be seen as clustering a subset of the vertices of a d-dimensional hypercube, and thus is a geometric problem in d dimensions. We give algorithms for various agglomerative hierarchical methods (including single linkage and complete linkage) as well as for two-clusterings and divisive methods.

With $\mathcal{C}^d = [0,1]^d$ we denote the d-dimensional hypercube, or *d-cube* for short. The set $V^d = \{0,1\}^d$ is the set of vertices of the d-cube, and can be seen as the set of all binary strings of length d. The vertices of the d-cube, (elements of V^d) represent the objects to be clustered, and we will use the terms 'object' and 'vertex of $\mathcal{C}^d$' synonymously. The goal of this paper is to give various methods to cluster a subset $P \subseteq V^d$ of objects.

The *Hamming distance* of two vertices $x, y \in P$ is the number of coordinates where x differs from y. For brevity, we will write $d_H(x,y)$. In the language of clustering, the distance function d_H is called symmetric and invariant. Invariance means that d_H is not affected by permuting coordinates (each bit in a string is equally important), and symmetry means that the probability of being '1' is 1/2 for all bits in a string randomly chosen from P. See e.g. Chapter 4 in [4] for a more detailed classification of binary distance functions.

We will use the term 'distance' to describe the dissimilarity of two objects, whereas the term dissimilarity itself is taken to measure the (non-)quality of a cluster. Thus a cluster containing very

*Research supported by the Spezialforschungsbereich F 003, *Optimierung und Kontrolle*.

close objects has high quality and thus a low dissimilarity. For a single object its dissimilarity is zero, for two objects it coincides with their distance. The dissimilarity of a cluster may be its diameter (e.g. for C-link) or the longest edge in its minimum spanning tree (S-link) or the mean distance between points (a version of A-link) and so on.

By symmetry of $\mathcal{C}^d$, for each vertex $x = (x_1, x_2, \ldots, x_d) \in V^d$ there is a vertex $\overline{x} = (1 - x_1, 1 - x_2, \ldots, 1 - x_d) \in V^d$ which we call the *antipode* of x in $\mathcal{C}^d$. Note that x and $\overline{x}$ represent two of the most disagreeing objects of a set. We will use antipodal pairs when constructing worst case configurations.

To make the description of the algorithms simpler we assume that the objects of P are pairwise different, i.e. their Hamming distance is greater or equal one. If multiple copies of an object are required in an application, we just weight an object by the number of its appearances. Where necessary the computation of the cluster quality has to be modified accordingly.

All algorithms we present in this paper use only linear space. In particular, we do not store any kind of a distance matrix. Thus, unlike for most other clustering methods, we do *not* assume that the distance of two objects can be computed in constant time. We rather have to spend $O(d)$ time for any such computation, and we will take this into account in the time analysis. The time requirements thus will be given in terms of the number of objects *and* the dimension.

For $P \subset V^d$ we denote with $n = |P|$ the number of objects to be clustered (i.e., the problem size), which may range from a few thousand to tens or hundreds of thousands. Typical values for the number d of bits (the problem dimension) are in the range from 32 (a 4 byte long integer) up to several hundreds (e.g. 256 or 512). We have $n > d$ and since a d-cube has 2^d vertices we get $d > \log_2 n$. Moreover we assume $\log_2 d \leq 31$ and $\log_2 n \leq 31$ so that we can handle ordinary variables and indices of objects in constant time (e.g. as long integers on current computers). Clearly this is not a serious limitation to applications since it allows about 10^9 objects to be represented.

For typical applications the space needed to store the input can range from some kilobytes up to several megabytes. A large example may have $n = 100000$ objects with $d = 200$ bits each. Thus we need 25 byte to store an object and about 2.5 MB to store the whole set. Therefore to be practicable the overhead in space required by the clustering algorithms must be at most linear in the input size, i.e. $O(dn)$ with some small constant. All algorithms given in this paper fulfill this requirement. Thus we suppose that all data can be kept in the main memory, and that an object and its augmenting information can be accessed in constant time.

2 Agglomerative Hierarchical Clustering

Agglomerative hierarchical clustering methods build a full cluster tree in the "bottom up" way, which means that leaves (and fruits?) grow first, then branches and the trunk come up, and finally the root appears. (Well, somehow that's top down, but we are all used to see trees up side down, so it's bottom up again.)

More formally an agglomerative hierarchical clustering method starts with n single objects, all seen as their own clusters. Step by step two specific clusters (objects at the beginning) are merged to a new cluster until, after $n-1$ steps, all objects form in a single final cluster, the root of the tree. Thus a hierarchical cluster tree is a family of clusters such that any two clusters are disjoint iff they lie on different paths (seen from the root), or one includes the other iff they are on the same path.

These cluster trees are sometimes visualized in the form of *dendrograms*: On the y-axis single objects and the agglomeration steps are shown, while the x-axis gives the dissimilarity of the resulting clusters, increasing from left to right. For an example of a dendrogram see Figure 9.

The two clusters to be merged in the next step are chosen in a way that the dissimilarity of the resulting cluster is minimized among all possible pairs. The criterion how to compute the dissimilarity of a (resulting) cluster determines the different methods. This criterion should fulfill the property that merging two clusters results in a new cluster with a dissimilarity at least the maximum of the two dissimilarities of the merged clusters. Thus the maximum dissimilarity of all clusters in a certain subtree is realized by the cluster last built (the subtree root). It is non-decreasing and can be seen as the measure of this subtree. Criteria fulfilling this property and cluster trees resulting from them are called *monotonous*. The closer a cluster lies to the root the more objects it will contain and the larger its dissimilarity will be. Therefore a monotonous tree gives us n levels of increasing maximum dissimilarity and decreasing number of clusters (seen from leaves to root).

The advantage of hierarchical clustering is now this trade-off between the number of clusters of a certain partition and their maximum dissimilarity. For applications where it is a priori not known how many, i.e. few, clusters are required, the best suitable combination of number and dissimilarity can easily be chosen once the cluster tree has been computed. Though needing only linear storage for the whole cluster tree, hierarchical cluster methods are more flexible than other clustering methods.

For a more detailed description of hierarchical clustering see e.g. Chapter 9 in [4] or Chapter 8 in [13]. Various methods for the dissimilarity computation of combinatorial agglomerative hierarchical clustering methods are given in Table 2 in [7]. Note however, that most of the dissimilarities there can only be computed in constant time if $O(n^2)$ space is available to store the distance matrix of the clusters. In the following sections we will use dissimilarity functions which turn out to be suitable for the d-cube case, needing only $O(dn)$ space.

An application of the hierarchical clustering of hypercubes given in this paper, can be found in [1]. There the structure of the cluster tree is utilized to give a new approach to the $\{0,1\}$-stringsearch problem: Given a set $P \subseteq V^d$ of $\{0,1\}$-strings of fixed length d, find for a query string $x \in V^d$ the "most similar" element $p \in P$. Most similar here means to minimize the Hamming distance between x and p. The cluster hierarchy now enables them to find the optimum vertex p without considering the whole set P by skipping irrelevant subclusters.

The widely used *single linkage* method (S-link) takes the two clusters which realize the smallest inter-cluster distance (i.e., the min-pair of elements in different clusters). The advantage of this method is its fast computability, but the quality of the resulting clusters may be poor due to chaining-effects; see e.g. [12]. To overcome this deficiency, *complete linkage* (C-link) uses a different criterion to chose the next two clusters: it minimizes the diameter (maximum distance of objects in one cluster) of the resulting cluster. Thus the resulting clusters are more "sphere-like" and produce better results for practical problems like document retrieval [12]. The clear disadvantage of C-link is the more difficult and thus time consuming computation.

Another variant is *average linkage* (A-link) where the mean (or sum) of the distances between all pairs of objects of a cluster is taken as the cluster dissimilarity. There are several variants of this methods, where e.g. a "central" representative of the cluster is chosen (either the center of gravity or a "typical" object of the set) and only the distances to this object are taken into account.

In the following subsections we will give algorithms for all mentioned methods.

2.1 Single linkage Clustering

The single linkage clustering (S-link, also called *nearest-neighbour clustering*) is one of the oldest methods of cluster analysis; it was first suggested by Florek et al. [10] and later on by various other authors. The first efficient general implementation of S-link was given by Sibson [22].

When building up the cluster tree S-link takes the two clusters which realize the smallest inter-cluster distance (min-pair with elements of different clusters). Intuitively spoken this approach tries to merge clusters having objects very close to each other. Thus the dissimilarity of the resulting clusters is defined as the longest edge in their minimum spanning tree (MST, cf. the next subsection for a definition).

The advantage of this method is its fast computability, because of its close relation to the well known MST of the objects. Thus, especially in low dimensions, it can be computed very fast, e.g. in $O(n \log n)$ time for general point sets for $d = 2$, with the help of the Delaunay triangulation (see e.g. [3]).

By definition of S-link, sorting the edges of a MST by length in non-decreasing order reflects the sequence of merging steps of S-link. Thus, once given the MST of n points, we can compute the S-link tree in $O(n \log n)$ time by sorting and inserting the $n-1$ edges of the MST. For the d-cube case we have to take care that comparing two elements for sorting can be done in constant time. To this end the MST algorithm has to store the lengths of the edges of the MST. Of course this does not affect the time analysis of the MST algorithms and needs only $O(n)$ additional space.

The MST algorithm for $\mathcal{C}^d$ we present in the following subsection runs in $O(dn^2)$ time. Together with the above remarks we have the following:

Theorem 1 *For a set $P \subseteq V^d$ with n elements, the S-link tree of P can be computed in $O(dn^2)$ time and $O(dn)$ space.*

2.1.1 The Minimum Spanning Tree in $\mathcal{C}^d$

Let $P \subseteq V^d$ be a set of vertices of $\mathcal{C}^d$. Then its *minimum spanning tree*, $MST(P)$, is a tree of P which minimizes the sum of its edge length. The length of an edge is defined as the distance of its two endpoints.

In the plane the Euclidean MST of a point set can be computed in time $O(n \log n)$ by using the fact that it is a subgraph of the Delaunay triangulation. Although in higher dimensions the MST is still a subset of the edges of the Delaunay triangulation this does not lead to an efficient algorithm since there the time to compute the Delaunay triangulation is $\Omega(n^{\lceil d/2 \rceil})$ [20]. Vertices of $\mathcal{C}^d$ are highly degenerate in the sense that there are many linear dependences between them. This would cause a lot of numerical troubles in computing the Delaunay triangulation for $\mathcal{C}^d$, and in fact most of the existing software packages fail on this task (compare e.g. [5]). In addition, vertices of P may also be cocircular (consider the hyper sphere with the same center as the cube and going through all vertices of $\mathcal{C}^d$) and thus the Delaunay triangulation may not even be unique.

Another way to compute the MST is the greedy approach of the graph algorithms of Prim and Kruskal. They insert the edges into an initially empty graph in a certain length order, adding only edges which do not close a circle. While the algorithm of Kruskal first sorts the edges in non-decreasing length to get a "global" order, the method of Prim always "locally" takes the shortest edge between two (not yet linked) components. Its not hard to see that both approaches indeed produce a MST of the given point set. But since Kruskals approach needs to sort the edges it would require $O(n^2)$ space for a complete graph and is thus impractical in the d-cube case.

For both methods the time to compute the MST is $O(m \log n)$ where m is the number of edges in the given graph, i.e. $m = O(n^2)$ in our case. Moreover they assume that the distance between two points can be computed in $O(1)$ time. In our case, we have to spend $O(d)$ time because, due to space limitations, we cannot simply store the distance matrix. On a first glance this gives a time of $O(dn^2 \log n)$.

A more careful look on Prims algorithm shows that it can be made to run in $O(dn^2)$ time and $O(dn)$ space. The algorithm is given in Figure 1. For the correctness of the algorithm see any basic text book an algorithms, e.g. [6].

The basic data structure we use is a simple priority queue Q containing all vertices not yet in the partially built MST. For any vertex in Q we store the minimum distance (=priority) to a vertex in the MST and a pointer to this vertex. Whenever a point is moved from Q to the MST this information is stored in the MST. The additional space needed for the algorithm is $O(n)$. As mentioned above S-link can then use this information to efficiently complete the cluster tree.

1. Let $MST(p)$ contain one arbitrary vertex $p \in P$. Let Q be an empty priority queue.
2. For all remaining vertices of P compute the distance to p, i.e. their priority, and insert them into Q.
3. **WHILE** Q is not empty **DO**
   (a) Remove the first element q of Q and add it to MST.
   (b) Restructure Q.
   (c) **FOR** all remaining elements b in Q **DO**
      i. Compute the distance from q to b.
      IF this distance is smaller than the priority of b **THEN**
      ii. update the priority and nearest neighbour pointer of b
      iii. update the position of b in Q

Figure 1: Prims MST algorithm for $\mathcal{C}^d$, running in $O(dn^2)$ time and $O(dn)$ space.

For the time analysis recall that we have $d > \log_2 n$. The initialization of steps 1 and 2 can be done in $O(dn)$ time. The while loop of step 3 is repeated $n-1$ times where step 3a and 3b can be done in constant and $O(\log n)$ time, respectively. Again the for loop of step 3c repeats $O(n)$ times and requires $O(d+1+\log n) = O(d)$ time per iteration. This gives a running time for the algorithm of $O(dn + n\log n + dn^2)$ which simplifies to $O(dn^2)$.

Lemma 1 *Given a set $P \subseteq V^d$ with n elements the MST of P can be computed in $O(dn^2)$ time and $O(dn)$ space.*

Note that since we have only d different distances we could also use, instead of the priority queue Q, d linked lists to organize the priorities. Thus the updates in step 3b and 3ciii could be done in constant time instead of $O(\log n)$. But since $d > \log_2 n$ this would not affect the asymptotic running time of the algorithm. It might simplify the implementation, however.

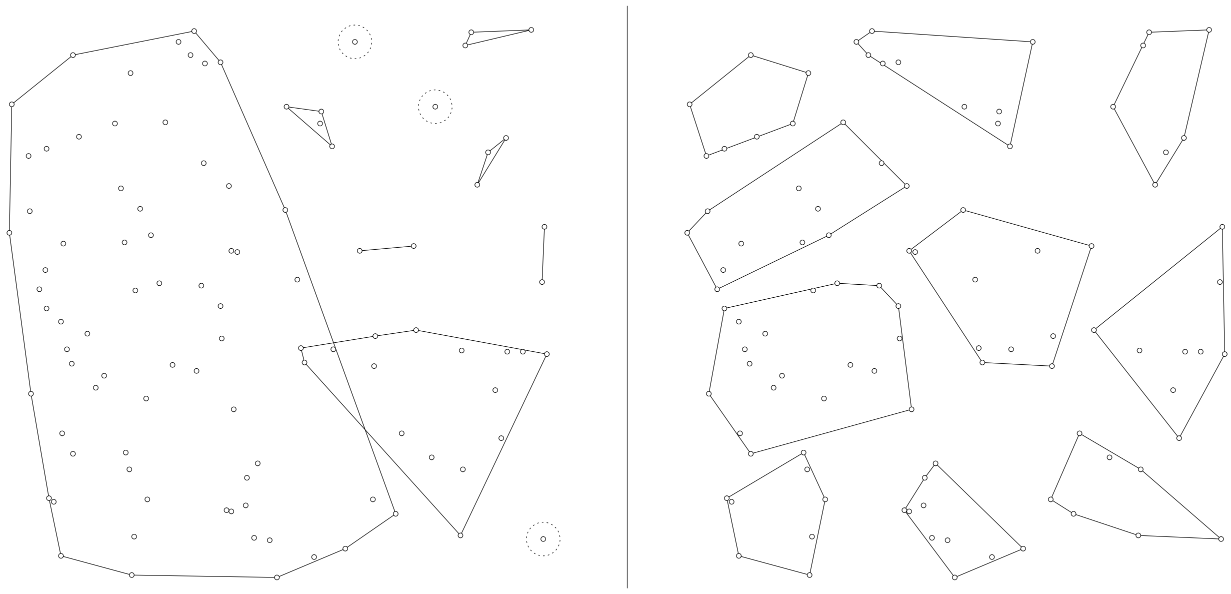
2.2 Generic Agglomerative Hierarchical Clustering Algorithms

Unfortunately the quality of the resulting clusters of S-link is often poor due to the chaining-effect of S-link [4]. It tends to build very uneven clusters, i.e. very long and tiny chains may contain nearly the whole set, while there are still non-clustered single objects. For an example of a planar point set see Figure 2(a).

We therefore give two generic hierarchical algorithms that run in $O(d^2n^2)$ and $O(dn^3)$ time, respectively, and require $O(dn)$ space. With the help of these algorithms we then can simply implement various criteria to choose the next pairs of clusters to be merged. We give some examples of criteria in the next subsections.

For a cluster C let $d(C)$ be its dissimilarity function or *δ-function* for short. Moreover let C be the agglomeration of the two subclusters C_i and C_j. To fit into the general framework, the dissimilarity function has to fulfill the following properties to which we will refer as *δ-function properties* in the sequel:

1. For a single-object cluster C, $d(C) = 0$.



(a) (b)

Figure 2: Planar point set of size $n = 100$ clustered by (a) S-link and (b) C-link. Both cluster hierarchies are shown at the level of 10 clusters. For S-link single-point clusters are marked by a dotted circle.

2. $d(C) \geq d(C')$ for any $C' \subseteq C$ in the cluster tree.

3. $d(C_i \cup C_j)$ can be computed with $O(|C_i| \cdot |C_j|)$ distance computations of single objects, where $|C|$ denotes the cardinality of C.

4. $d(C) \in \{0, 1, \ldots, d\}$.

Property 1) is not really necessary for the correctness of the algorithms but seems to be quite natural and is implicitly fulfilled by all reasonable δ-functions known to the author. Property 2) ensures that the dissimilarity of the clusters does not decrease with increasing level (i.e. towards the root) of the cluster tree. Note that, by transitivity, property 2) is equivalent to $d(C) = d(C_i \cup C_j) \geq \max\{d(C_i), d(C_j)\}$, i.e. the levels of the cluster tree are monotonous.

If property 2) is violated then the relation between the number of clusters of a level and their maximal dissimilarity is not longer monotonous. Thus reducing the number of clusters may give an even smaller maximum dissimilarity which is called an *inversion*. Of course this hardly makes sense in interpreting the given data and we thus only study δ-functions fulfilling property 2).

Properties 3) and 4) will be used to give good time bounds for the algorithms. Some of the presented δ-functions will violate property 4) and thus result in slower algorithms.

Related to property 3) we give a rather trivial but useful observation in the next lemma.

Lemma 2 *Given k integers $n_i \geq 1$, $i = 1, \ldots, k$ with $n_1 + \ldots + n_k = n$. Then*

$$\sum_{i=1}^{k} \sum_{j=i+1}^{k} n_i n_j \leq n^2$$

1. Put every object of P into a single-element cluster C_i, $1 \leq i \leq n$
2. **FOR** $diss = 1$ **TO** d **DO**
   (a) Set $i = 1$
   (b) **WHILE** i is less than the actual number of cluster **DO**
      i. Set $j = i + 1$
      ii. **WHILE** j is less than or equal to the actual number of clusters **DO**
         A. **IF** the dissimilarity of the union of C_i and C_j equals $diss$ **THEN**
            - Let C_i be the union of C_i and C_j
            - Replace C_j by the cluster with the highest index, changing its index to j
            - Reduce the number of clusters by one
            - (Set j = i+1) *; see text*
         B. **ELSE**
            - Set $j = j + 1$
   (c) Set $i = i + 1$

Figure 3: Generic hierarchical clustering Algorithm GHC1 for $\mathcal{C}^d$ with running time $O(d^2n^2)$.

Proof. $\sum_{i=1}^{k}\sum_{j=i+1}^{k} n_i n_j = \sum_{i=1}^{k}\left(n_i \sum_{j=i+1}^{k} n_j\right) = \sum_{i=1}^{k}\left(n_i\left(n - \sum_{j=1}^{i} n_j\right)\right) =$
$= n\sum_{i=1}^{k} n_i - \sum_{i=1}^{k}\sum_{j=1}^{i} n_i n_j = n^2 - \sum_{i=1}^{k} n_i^2 - \sum_{i=1}^{k}\sum_{j=1}^{i-1} n_i n_j =$
$= n^2 - \sum_{i=1}^{k} n_i^2 - \sum_{j=1}^{k}\sum_{i=j+1}^{k} n_i n_j = n^2 - \sum_{i=1}^{k} n_i^2 - \sum_{i=1}^{k}\sum_{j=i+1}^{k} n_i n_j$
Rearranging the equality gives $\sum_{i=1}^{k}\sum_{j=i+1}^{k} n_i n_j = \frac{1}{2}\left(n^2 - \sum_{i=1}^{k} n_i^2\right) \leq n^2$. □

We will use this lemma in the time analysis when the dissimilarity of every pair C_i, C_j of clusters has to be computed.

The scheme of the first generic hierarchical clustering algorithm *Algorithm GHC1* is given in Figure 3. In Step 1 every object is made to a single-object cluster. In the for loop of step 2 the dissimilarity to be considered is preset, starting from 1 and going up to d. In the body of this loop now all pairs of clusters are merged which produce a cluster with the given dissimilarity. Clearly from the definition of a δ-function the dissimilarity can not decrease during the iterations. But we have to be careful if there exists a triple C_i, C_j, C_k of clusters, such that merging any two out of these three gives the same dissimilarity of the resulting cluster. In this case e.g. C_i may be involved into two merge steps within one iteration of the for loop. This is the reason why we give the handling of the indices explicitly.

The outer while loop of step 2b controls the first cluster C_i of the pair to be considered. Note that the actual number of clusters may decrease during the iteration. The inner while loop (2bii) controls the second cluster C_j such that any possible pair of clusters is considered exactly once. If two clusters are merged then the resulting cluster gets the index i. Cluster C_j is replaced by the cluster with the highest index and the number of clusters is reduced.

To handle the above mentioned special case, that one cluster may be involved in several agglomeration steps during one iteration of the FOR loop, we have to distinguish between two different methods to compute the dissimilarity of a cluster. For the first method we assume that adding points (i.e. another cluster) to a cluster C can never decrease the dissimilarity of C. This is e.g.

the case for C-link, see Section 2.3. In this case no cluster with an index smaller than j can have a dissimilarity with C_i smaller or equal to *diss* after merging C_i and C_j. Thus it is sufficient to not increase the index j (which already points to a new cluster) after an agglomeration step.

In the second case adding points to a cluster C may reduce its dissimilarity. This is e.g. the case for centroid clustering if the centroid point is restricted to the set of cluster points, see Section 2.5.1. Note that this is no contradiction to δ-function property 2), since there the monotonicity is only required between clusters within the same path of the cluster tree. For example we may have three clusters C_i, C_j and C_k, $i < j < k$, where C_k contains a vertex which is the minimizing centroid point for C_i and C_j (cf. Section 2.5.1). Thus we first compute the dissimilarity of $C_i \cup C_j$ which may be too large. In the next step we merge C_i and C_k to C_i', since C_k provides a centroid point which gives a dissimilarity small enough for $C_i' = C_i \cup C_k$. But after this step the dissimilarity of $C_i' \cup C_j$ may be also small enough to be merged. Note that by δ-function property 2) it can not be smaller than the dissimilarity of C_i', but might be equal, which means that we might have to merge the two clusters within the same iteration of the FOR loop. Therefore it is necessary to reconsider all clusters C_j, $j > i$, after an agglomeration step, i.e. to set $j = i + 1$ as the last step in 2biiA.

In this way we guarantee that within one iteration of the FOR loop all possible pairs are merged and the correctness of the algorithm follows.

From the above discussion it also follows that in steps 2a up to 2c every possible pair of clusters is considered exactly once during an iteration of *diss* as long as there is no agglomeration step. If the objects of a cluster are stored in a linked list then by δ-function property 3) and Lemma 2 a single iteration of step 2 takes $O(dn^2)$ time, resulting in time $O(d^2n^2)$ for the whole Algorithm GHC1. If, as mentioned above, we have to reconsider all clusters C_j, $j > i$, after every agglomeration step, we add $n - 1$ times $O(dn)$ time, thus the overall time of the algorithm remains $O(d^2n^2)$.

Instead of using a FOR-loop in step 2 we can increase *diss* (after step 2c) to the smallest dissimilarity larger than *diss* which occurred during the actual iteration. Thus we may skip several steps, and experimental results show that this significantly speeds up the algorithm (cf. Section 4).

Essential for this first approach is that the dissimilarity of a cluster can only be an integer in the range $\{0, \ldots, d\}$ (δ-function property 4). For δ-functions which do not fulfill this property we give another approach, *Algorithm GHC2*, as a generic hierarchical clustering algorithm. GHC2 has been observed in several experiments to be faster than GHC1 despite its larger worst case running time. The main feature of GHC2 is its generality.

When merging a cluster C_i with another cluster we define a *best merge cluster of* C_i, $BMC(C_i)$, as a cluster which minimizes the resulting dissimilarity among all clusters C_k, $k \neq i$. For single points this usually coincides with the well known nearest neighbour of a point. If the best merge cluster of C_i has itself C_i as its best merge cluster, i.e. $BMC(BMC(C_i)) = C_i$, we call this a *merge pair*.

Note the special case if $BMC(C_i)$ is not unique, i.e. where C_k and C_l, $k \neq l$, both give the same dissimilarity when being merged with C_i. This may cause cyclic relations; for an example with a 4-cycle in $\mathcal{C}^3$ see Figure 5(a). Due to the symmetry of the δ-function all dissimilarities in a non-trivial cycle (a trivial cycle is just a merge pair) must of course be equal. In this case for any cluster C_i the cluster with the lowest index of all elements in $BMC(C_i)$ is taken to avoid non-trivial cycles where no merge pair may exist. Thus the graph of best merge cluster relations can be seen as a forest, where the roots of the trees consist of the merge pairs (cf. Figure 5(b)). Moreover, since the graph is free of non-trivial cycles, at least one merge pair always exists.

From the fact that the δ-function is monotonous (property 2) it follows that all existing merge pairs can be merged in parallel. After merging two clusters the BMC-information of a third cluster C_k can change if and only if the old $BMC(C_k)$ was one of the merged clusters or the new $BMC(C_k)$ is the new cluster. Thus after merging all merge pairs we have to update the information about

1. Put every object of P into a single-element cluster C_i, $1 \leq i \leq n$
2. For all clusters C_i compute the best merge cluster $BMC(C_i)$.
3. **WHILE** there is more than one cluster **DO**
   (a) **FOR** all clusters C_i **DO**
      i. **IF** $BMC(BMC(C_i)) = C_i$ **THEN**
         A. Let C_i be the union of C_i and $BMC(C_i)$ and mark the resulting cluster as new.
         B. Remove $BMC(C_i)$
   (b) Compute the best merge cluster $BMC(C_i)$ for all clusters C_i marked as new or with $BMC(C_i)$ marked as new or having been removed. *; see also text*
   (c) Delete all new markings

Figure 4: Generic hierarchical clustering Algorithm GHC2 for $\mathcal{C}^d$ with worst case running time $O(dn^3)$.

best merge clusters accordingly. This means that we recompute the BMC-information of every new cluster symmetrically, i.e. while searching for the BMC of the new cluster we may also update the information for all considered clusters (for implementation details see below).

The new approach now follows these lines. It first computes all best merge clusters. Then it merges all merge pairs, updates the information about best merge clusters only where necessary and repeats this until only one final cluster is left. The detailed algorithm is given in Figure 4. From the above discussion it is straight forward to see that Algorithm GHC2 indeed produces the required cluster tree.

Note that the construction sequence of the cluster tree is not monotonous for Algorithm GHC2, i.e. if necessary for an application we have to sort the levels of the tree by non-decreasing dissimilarity. Since the tree has only linear size this can clearly be done in $O(n \log n)$ time.

The worst case running time of Algorithm 2 is $O(dn^3)$: The initialization in step 2 requires to compute the distance function between all vertices, which takes $O(dn^2)$ time. The for-loop of step 3a repeats $O(n)$ times and the body of the loop can be done in constant time. Although in most cases in step 3b only a small number of clusters is concerned, in the worst case we have to update the information about the best merge clusters for a linear number of clusters. Thus step 3b needs $O(dn^2)$ time (cf. the analysis of the previous algorithm).

If the dissimilarity of a cluster has to be from $\{0, \ldots, d\}$ (e.g. for the diameter or radius of a cluster) and if $BMC(C_i)$ is unique, then every iteration of the while loop in step 3 increases the minimum dissimilarity by at least one. Since the maximum dissimilarity is d, we have at most d iterations and thus an overall running time of $O(d^2n^2)$. If $BMC(C_i)$ is not unique we may have only one merge pair per iteration, i.e. the while loop must repeat O(n) times. The same is true if the δ-function does not fulfill δ-function property 4). This gives the worst case time of $O(dn^3)$.

Let us point out some implementation details of step 3b of Algorithm GHC2. When computing the new $BMC(C)$ for a new cluster C, we have to distinguish between two kinds of dissimilarity functions. If for any agglomeration step $C = C_i \cup C_j$ we have

$$\min\{d_H(C_k, C_i), d_H(C_k, C_j)\} < d_H(C_k, C) \tag{1}$$

for any third cluster C_k, $k \neq i, j$, we clearly need not to update $BMC(C_k)$ as long as $BMC(C_k) \neq$

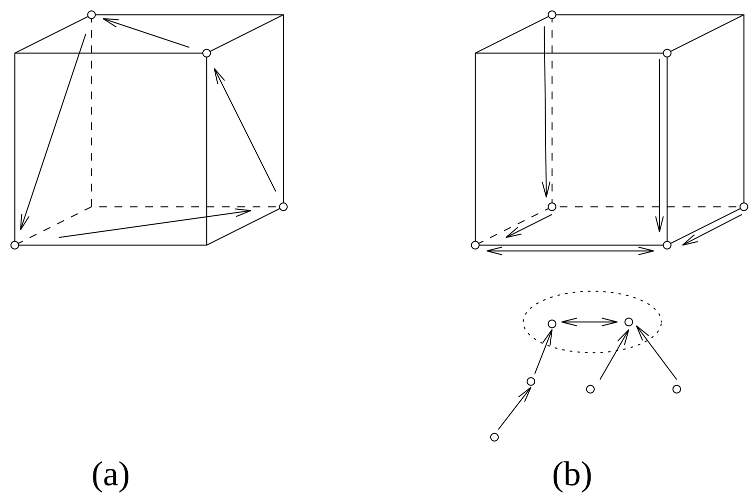


Figure 5: Cycle of best merge clusters (a) and best merge cluster tree (b) with merge pair as root (dotted ellipse).

C_i, C_j. Since $BMC(C_k)$ achieved a smaller dissimilarity with C_k than with the "better" one of C_i and C_j, C can never be the new $BMC(C_k)$.

But if inequality 1 is violated then $d_H(C_k, C)$ might be less than $d_H(C_k, BMC(C_k))$ even when $BMC(C_k) \neq C_i, C_j$. Another possibility is that $d_H(C_k, C) = \min\{d_H(C_k, C_i), d_H(C_k, C_j)\}$. If in this case the index of C is now less than the index of the old $BMC(C_k)$ we have to update the $BMC(C_k)$ to avoid non-trivial cycles, though the value of $BMC(C_k)$ remains unchanged (see above for a discussion on cycles).

In both cases we therefore have to recompute the BMC information in step 3b in a symmetric way, i.e. whenever we compute the dissimilarity between a cluster C marked as new and an arbitrary cluster C_k, we not only have to check whether this provides a new $BMC(C)$ but also if we get a new $BMC(C_k)$. Clearly this does not affect the running time of the algorithm, since it requires only a simple constant time comparison and there are no additional distance computations necessary. We thus implemented the additional symmetric test for all methods based on Algorithm GHC2.

Note that inequality 1 does not follow from the monotonicity of a dissimilarity function, since δ-function property 2 only has to hold for the relations given by the "edges" of the cluster tree. For example A_{mean}-link (see Section 2.4) is shown to be monotonous (cf. Lemma 6) but obviously violates inequality 1. In fact A_{mean}-link and centroidal clustering minimizing the mean distance (see Section 2.5.2) are the only methods we will present for GHC2, where this is the case and thus the symmetric test is required.

Experiments show that in step 3b only the information of a small number of clusters has to be updated and the while loop of step 3 repeats much less than $O(n)$ times. Thus Algorithm GHC2 is expected to be very efficient for applications. For more details see Section 4.

2.3 Complete linkage Clustering

In this subsection we consider the Complete-Linkage Clustering method (C-link) which was first mentioned by Sørensen [24] and also very early used in [14, 16, 18] (for a historical survey see e.g. Chapter 1.6 in [9]). To overcome the chaining effect of S-link, C-link uses a different criterion to choose the next two clusters: it minimizes the diameter (maximum distance of objects in one cluster) of the resulting cluster. Therefore the δ-function of C-link for a cluster C is $d(C) = \max_{x,y \in C} d_H(x, y)$. The resulting clusters are more 'sphere-like' and expected to produce better results for practical problems [12, 13]. Experiments on this method show that it produces rather even, i.e. equal sized, clusters, which may be suitable for many applications. For an example of a planar point set see Figure 2(b).

The clear disadvantage of C-link is the more difficult and thus time consuming computation. Another reason which caused C-link to be unpopular was given by the paper of Defays [8]. He gave an $O(n^2)$ implementation of C-link. Unfortunately this implementation does not really compute C-link but only a variant of it (although this was not pointed out explicitly in the paper). It is shown in [12] that this variant has not longer the nice properties of C-link and thus produces worse results. In fact, while C-link is reported to be one of the best methods there (beside A-link, see Section 2.4), the results obtained by Defays' implementation are the worst. Unfortunately the fast running time proposed by Defays prevented other researcher for considering new implementations of C-link and therefore only straightforward and thus slow codes exist.

The best linear space implementation for general objects given in the literature so far had a worst case time bound of $O(n^4)$, the fastest known implementation was the version used by Voorhees [25] (see also [12]), running in time $O(n^3)$, but using space $O(n^2)$. In a recent paper [2] practical implementations, with running time $O(n^2 \log n)$ for quadratic space, and $O(n^3)$ time for linear space, are given: The $O(n^2 \log n)$ time solution is based on n heaps, one for every subcluster. Each heap manages the dissimilarities of its cluster to all other clusters. Thus we can find the next two clusters to merge in linear or even logarithmic time, if an additional heap is used to store the minima. An update step for the dissimilarity information can be done in time $O(n \log n)$, which gives the promised bound.

For linear space there are two algorithms. The first one is very similar to the previously known $O(n^4)$ version, except that from the new implementation it is obvious to see that one merge step needs at most $O(n^2)$ distance computations. Thus the $O(n^3)$ time bound follows immediately. The second approach uses the concept of nearest neighbour pairs, very similar to the best merge clusters of Algorithm GHC2 (cf. Section 2.2). It thus has a worst case running time of $O(n^3)$, but experimental results show that it most likely runs in $O(n^2 \log n)$ time for practical problems (see [2], cf. also Section 4).

Most recently Krznaric and Levcopoulos gave an $O(n \log^2 n)$ time C-link algorithm for the special case of planar point sets, see [15].

Note that all mentioned implementations assume that the distance between two objects can be computed in constant time. To this end either the distance matrix has to be given explicitly, where then already the input has quadratic size, and we thus do not need to consider linear space solutions. Or, if the objects (points) lie e.g. in the real space $\mathbb{R}^d$, we can compute the distance from their coordinates. But then the dimension d must be of constant size. Considering arbitrary high dimensions we clearly have to multiply the given time bounds by a factor $O(d)$. If we may use quadratic space we can pre compute the distance matrix in $O(dn^2)$ time, and get thus an $O(n^2(\log n + d))$ algorithm.

For the special case of binary presentable objects we can now use the generic algorithms developed in the last section.

The diameter of a cluster can simply be computed by taking the maximum of the distances between all pairs of points. Thus it is straight forward to see that the diameter fulfills δ-function properties 1), 2) and 4) of our framework. The next lemma shows that it fulfills also property 3).

Lemma 3 *If a cluster C_k is build by a merge step in the C-link tree from two clusters C_i and C_j then there exists a pair of vertices $v_i \in C_i$, $v_j \in C_j$ which realize the diameter of C_k. That is the diameter of C_k can be computed with $O(|C_i| \cdot |C_j|)$ distance computations.*

Proof. Suppose that no such pair exists. Let r be the diameter of C_k. Then w.l.o.g. r is realized by vertices of C_i which means that the diameter of C_i is r. Looking at the history of C_i in the C-link tree there was a merge step where C_i or a subset of it achieved the diameter r the first time. At this stage the two subclusters merged had both a diameter of less than r. Thus merging one of them with C_j (or any at this time existing subcluster of C_j with diameter less than r) would have given a cluster with diameter strictly less than r. But then C-link would have merged these two subcluster instead, which is a contradiction to the current existence of C_i and C_j. □

From the above results we conclude that we can use Algorithm GHC1. Note that, as mentioned in Section 2.2, we do not need to include the line "j=i+1" in step 2biiA.

Theorem 2 *Given a set $P \subseteq V^d$ with n elements. The C-link tree of P can be computed in $O(d^2n^2)$ time and $O(dn)$ space.*

2.4 Average linkage Clustering

In this subsection we consider hierarchical clustering methods where the dissimilarity of a cluster depends on the sum or mean of distances between vertices. For general implementations and applications of this method see e.g. Voorhees [25]. We first show that these dissimilarities can be computed very efficiently in the d-cube case.

Lemma 4 *For all clusters $C, C_i, C_j \subseteq V^d$ arising during the hierarchical clustering of n objects from V^d the sum (mean) of distances between points within C or between points of C_i and C_j can be computed in $O(d)$ time, using $O(dn)$ additional space for the whole cluster tree.*

Proof. For all clusters C_i existing during the hierarchical clustering process we store for every bit position b, $b = 1, \dots, d$, the number $n_0(C_i, b)$ of elements from C_i with a '0' on position b and the number $n_1(C_i, b)$ of elements from C_i with a '1' on position b. To do this we need $O(d)$ additional space per cluster and thus $O(dn)$ additional space for the whole cluster tree. Clearly an update for a merge step takes only $O(d)$ additional time. Moreover we also know the number of elements of a cluster, thus once we have the sum of distances we can compute the mean of them in constant time. (In fact any reasonable implementation will only store the number n_i of elements in a cluster C_i and $n_0(C_i, b)$ since $n_1(C_i, b) = n_i - n_0(C_i, b)$.)

To compute the sum of distances between the elements of two clusters C_i and C_j one usually has to compare all pairs of elements and sum up the number of differing bits per pair. We now do this separately for every bit position b and then sum over all d positions. Clearly a pair of elements contributes one to the sum iff they have a different bit on position b. This means that any element of C_i with '0' on position b adds one to the sum whenever compared with an element of C_j with '1' on position b. Of course the same is true if we exchange '0' and '1' and thus summing over all positions b we get $\sum_{b=1}^{d} (n_0(C_i, b)n_1(C_j, b) + n_1(C_i, b)n_0(C_j, b))$ for the sum of all distances between C_i and C_j. This sum can easily be computed in $O(d)$ time using the information stored in the cluster tree. For the sum of distances between the elements of a single cluster C we similarly get $\sum_{b=1}^{d} n_0(C, b)n_1(C, b)$. □

The Average-linkage method (A-link, sometimes also called Group Average Linkage, [12,17]) uses the mean distance between the points of two clusters as their cluster distance, i.e. as the dissimilarity of the resulting cluster after a merge step. Note that this δ-function does not only depend on a cluster but also on the way it was built. Thus we can show properties of this δ-function only when used for hierarchical clustering. For δ-function property 1) we just have to define the mean distance of a single point as zero. Property 3) follows from Lemma 4 and property 4) is obviously violated, thus we may only use Algorithm GHC2 of the last section. It remains to prove property 2), i.e. that A-link builds a monotonous cluster tree. It is well known (see e.g. [4]) that A-link is monotonous, not only in the geometric or hypercube case. We therefore give the proof of the following lemma for the sake of completeness and because some of the subsequent proofs rely on the same cluster hierarchy.

Lemma 5 *For all merge steps $C = C_i \cup C_j$ of A-link we have $d(C) \geq \max\{d(C_i), d(C_j)\}$.*

Proof. W.l.o.g. assume that $d(C_i) \geq d(C_j)$ and suppose that $d(C) < d(C_i)$. We prove the lemma by induction over the merge steps. Let the merge step $C = C_i \cup C_j$ be the first step where an inversion occurs. Then by induction the last merge operation in this subtree was the agglomeration of $C_i = C_i' \cup C_i''$. Since $d(C) < d(C_i)$ the mean distance between points of C_i and C_j is less than the mean distance between points of C_i' and C_i''. But then either the mean distance between points of C_i' and C_j or C_i'' and C_j is also less than $d(C_i)$. Thus A-link would merge either C_i' and C_j or C_i'' and C_j instead of C_i' and C_i'', a contradiction to the existence of C_i. □

Using Algorithm GHC2 we thus get

Theorem 3 *Given a set $P \subseteq V^d$ with n elements. The A-link tree of P can be computed in $O(dn^3)$ time and $O(dn)$ space.*

It is quite unsatisfying that the δ-function of A-link depends on the history of a cluster. One reason for the way A-link is defined comes from the attempt to use the generic SAHN clustering methods (SAHN stands for sequential, agglomerative, hierarchical, nonoverlapping, see e.g. [7]). We now give two modified versions of A-link which do not fit into the scheme of SAHN but seem to be more natural at least to the author.

For *A_{sum}-link* (*A_{mean}-link*) we define the dissimilarity of a cluster C as the sum (mean) of all distances between points from C. Clearly the δ-functions do not longer depend on the history of a cluster. From the above discussion it follows that we need only to consider the δ-function property 2). The dissimilarity of A_{sum}-link obviously fulfills property 2) since adding points to a cluster always increases the sum of Hamming distances. For the mean distance this is in general not true as can be seen by the following example. Consider the two clusters $C_1 = ((0,0,0,0),(1,1,1,0))$ and $C_2 = ((0,0,0,1),(1,1,1,1))$ of $\mathcal{C}^4$ with mean distance 3 each. Merging them to $C = ((0,0,0,0),(0,0,0,1),(1,1,1,0)),(1,1,1,1))$ gives a cluster C with mean distance of $2\frac{2}{3}$. Clearly clusters C_1 and C_2 would not occur during a hierarchical clustering process of these four objects. Still A_{mean}-link defines a monotonous cluster tree, i.e., forces property 2.

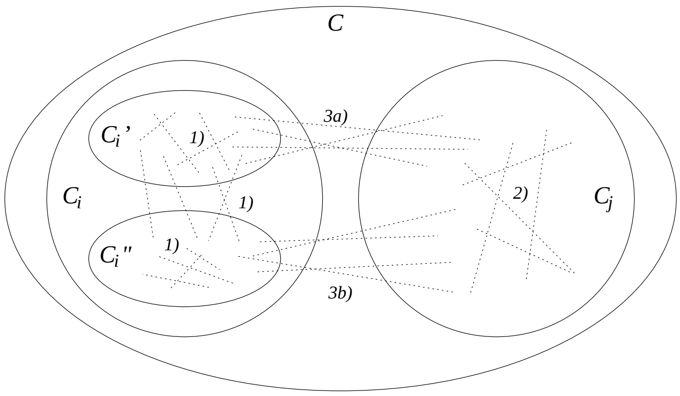


Figure 6: Cluster hierarchy for the proof of Lemma 6

Lemma 6 *Let the dissimilarity $d(C)$ be the mean distance between the vertices of the cluster C. Then for the hierarchical clustering method A_{mean}-link the dissimilarity is monotonous, i.e. for any cluster $C = C_i \cup C_j$ we have $d(C) \geq \max\{d(C_i), d(C_j)\}$.*

Proof. We prove the lemma by induction over the merge steps. It is clear that merging single points can not violate the monotonicity. So let the merge step $C = C_i \cup C_j$ be the first step where an inversion occurs, i.e. $d(C) < d(C_i)$ where we assume w.l.o.g that $d(C_i) \geq d(C_j)$. Then by induction the last merge operation in this subtree was the agglomeration of $C_i = C_i' \cup C_i''$, refer to Figure 6. Note that this step was monotonous.

The distances in C consist of three types: 1) the distances within C_i, 2) the distances within C_j and 3) the distances between vertices from C_i and vertices from C_j. Since the mean of all distances of C is less than $d(C_i)$ the mean of the distances of type 2 and 3 must be less than $d(C_i)$, too. Now split the distances of type 3 into two parts: 3a) distances between C_i' and C_j, 3b) distances between C_i'' and C_j. W.l.o.g we assume that the mean of the distances of type 3a is less or equal the mean of the distances of type 3b. Then clearly the mean of the distances of type 2 and type 3a is less than $d(C_i)$ (because $d(C_j) \leq d(C_i)$).

On the other hand we have $d(C_i') \leq d(C_i)$ since by induction this agglomeration step was monotonous. Therefore the mean of all distances within C_j (type 2), between C_j and C_i' (type 3a) and within C_i' is strictly less than $d(C_i)$. But thus $d(C_j \cup C_i') < d(C_i)$, i.e. merging C_j and C_i' results in a smaller dissimilarity than merging C_i' and C_i'', a contradiction to the current existence of C_i in the cluster tree. Thus the mean distance function is indeed monotonous for hierarchical clustering. □

It therefore follows that we can use Algorithm GHC2 to compute the A_{sum}-link and A_{mean}-link trees.

Theorem 4 *Given a set $P \subseteq V^d$ with n elements. The A_{sum}-link and A_{mean}-link tree of P can be computed in $O(dn^3)$ time and $O(dn)$ space.*

Note that for all presented versions of hierarchical average linkage clustering the dissimilarity of a cluster can be computed very efficiently. Taking only $O(d)$ time it is much faster than required by δ-function property 3). Unfortunately the true bottleneck of Algorithm GHC2 is the number of updates of the best merge cluster information. Thus in the worst case analysis the fast dissimilarity computation has no effect on the running time. But the experiments given in Section 4 show that for applications the number of necessary updates is far below the worst case. Therefore Algorithm GHC2 together with the fast dissimilarity computation of Lemma 4 gives a quite efficient solution for hierarchical average linkage clustering.

2.5 Hierarchical Centroidal Clustering

Centroid clustering methods for objects characterized by k-tuples of real numbers are based on the representation of clusters by single points in k-dimensional real space. Clusters (i.e. their centroid points) being merged are replaced by a single centroid point. For the d-cube case we can either allow arbitrary centroid points in $\mathbb{R}^d$ or we restrict the set of centroid points to elements of V^d. Moreover for a cluster $C \subseteq V^d$ we may only allow an element of C as its centroid point, which means that every cluster is represented by its "most typical" object. In this last case we may call the centroid point the *kernel* of C.

To measure the dissimilarity of a cluster C we either take the maximum distance of a point from C to the centroid point (*radius* of C), the sum of these distances or their mean (*mean radius*). Since the number of vertices is constant for a given cluster C, minimizing the mean distance coincides with minimizing the sum of distances from the centroid point. Thus the one-centroid problem can be solved for both optimization criteria similarly. Note however that this holds only for single clusters and is in general not true for the entire cluster tree. Minimizing the mean distance on one hand and the sum of distances on the other may result in different cluster trees.

Thus the centroid point of C has always to be chosen such that it minimizes the maximum (sum, mean) distance to points in C. We use the Hamming distance if the centroid point is in V^d and the

L_1-metric for arbitrary centroid points from $\mathbb{R}^d$. We therefore distinguish between three different domains for centroid points and three different methods to determine the centroid point of a cluster and to compute its dissimilarity.

2.5.1 Minimizing the Radius of a Cluster

We first consider hierarchical centroid clustering minimizing the radius of the cluster. To this end let for a cluster $C \subseteq V^d$ the dissimilarity $d(C)$ be the maximum distance from points in C to its centroid point. For an arbitrary point p we denote with $d(C,p)$ the maximum distance from points in C to p. Note that for any set P of points containing the centroid of C we have $d(C) \leq d(C,p)$ for all $p \in P$. In the following lemma we now show that this δ-function is indeed monotonous, i.e. fulfills δ-function property 2).

Lemma 7 *For $C \subseteq V^d$ let the dissimilarity $d(C)$ be the maximum distance between vertices of C and the centroid p of C, where p minimizes this maximum distance among all possible points in either $\mathbb{R}^d$, V^d or C. Then for the hierarchical centroid clustering the dissimilarity $d(C)$ is monotonous.*

Proof. If centroid points are from either V^d or R^d the lemma is obviously correct. Adding vertices to a cluster C can never decrease the radius of C since we choose the centroid point of C always such that it minimizes the maximum distance to points in C.

When we restrict the centroid point of a cluster C_i to be a vertex of the cluster itself, another cluster C_j may contain a vertex with a smaller maximum distance to vertices in C_i and thus the above argumentation does not hold. We therefore utilize the cluster hierarchy given in the proof of Lemma 6 and show Lemma 7 by induction over the merge steps.

Clearly merging single points can not violate the monotonicity. So let the merge step $C = C_i \cup C_j$ be the first step where an inversion occurs, i.e. $d(C) < d(C_i)$ where we assume w.l.o.g that $d(C_i) \geq d(C_j)$. Then by induction the last (monotonous) merge operation in this subtree was the agglomeration of $C_i = C_i' \cup C_i''$.

The kernel p of C can not lie in C_i because then $d(C,p)$ has to be at least $d(C_i,p) \geq d(C_i)$, a contradiction to the inversion. But then $p \in C_j$ and since $C_i' \cup C_j \subset C$ we have $d(C_i' \cup C_j) \leq d(C_i' \cup C_j, p) \leq d(C,p) = d(C) < d(C_i)$, i.e. merging C_j and C_i' results in a smaller dissimilarity than merging C_i' and C_i'', a contradiction to the current existence of C_i in the cluster tree. Thus the maximum distance from the centroid point is monotonous for all mentioned domains. □

If centroid points are allowed to be from $\mathbb{R}^d$ we can reduce the optimization problem of finding the centroid point minimizing the maximum distance to a point in C to linear programming by doing a binary search for the minimum distance r over the possible range $\{0, \ldots, \frac{d}{2}\}$. (Note that the point $(\frac{1}{2}, \ldots, \frac{1}{2})$ always achieves the upper bound.) We then ask whether for a given distance r there exists a centroid point $p \in \mathbb{R}^d$ for C such that the L_1-distance to every point $c_i \in C$ does not exceed r. We can formulate this as a linear inequality $x_i \cdot p \leq r - n_1(c_i)$, for every $c_i \in C$. Thereby $n_1(c_i)$ is the number of ones in c_i and the j^{th} component x_i^j of x_i is set to 1 if the corresponding bit c_i^j of c_i is zero, and to -1 else. (The dot-product $x_i \cdot p$ is given by $\sum_{j=1}^{d} x_i^j p^j$.) Finding a feasible solution for this linear system (i.e. a centroid point with the required radius r) can be solved in polynomial time (e.g. by Karmarkar's interior point methods [21, 23]).

For the case that centroid points are restricted to V^d we have to do the binary search only for integer distances from $\{1, \ldots, d\}$. However, in the context of linear programming the problem if for a given distance r there exists a centroid point in V^d, is now an instance of integer programming (see e.g. [21, 23]). In general integer programming asks whether a given system of linear inequalities, in d variables and with integer coefficients, has an integer solution. This problem is well known to be NP-complete [11, 21], especially the variant in which the solution variables are required to belong to $\{0,1\}$ (zero-one integer programming, shown to be NP-complete by a simple transformation from 3SAT). For an overview and a discussion on the complexity of integer programming see e.g. Chapter 9.5 in [19] and references given therein.

Note however, that this does clearly **not** imply that minimizing the radius to a centroid point in V^d is also NP-complete. To our knowledge the complexity class of this problem is still unknown. Moreover note that we consider the running time as function of n and d, i.e. we require that it is polynomial in both parameters. Obviously a straight forward approach runs in time $O(dn2^d)$ and thus we get a linear time solution for constant d. If n is a small constant, the problem can also be solved in polynomial time, e.g. in $O(d^3)$ for $n = 3$.

For complexity reasons we therefore restrict our investigations to the simpler case where for a cluster C its centroid point has to be contained in C.

Theorem 5 *Given a set $P \subseteq V^d$ with n elements. The centroid point of a cluster has to be contained in the cluster and to minimize the maximum distance of points from the cluster to it. Then the hierarchical centroid clustering tree of P can be computed in $O(d^2n^2)$ time and $O(dn)$ space.*

Proof. Properties 1) and 4) for δ-functions are obviously fulfilled and Lemma 7 shows that also property 2) holds. To apply Algorithm GHC1 it thus remains to show that for an agglomeration step in the cluster tree we can compute the kernel of two clusters C_i and C_j in $O(d|C_i| \cdot |C_j|)$ time. To this end for every point p_i of a cluster we store the maximum distance to another point within the same cluster. This requires only $O(n)$ additional space for the whole cluster tree and the centroid point of a cluster C can thus be obtained in $O(|C|)$ time by taking the point with the minimum value. The kernel of the union of two clusters C_i and C_j (i.e. an agglomeration step) now can be computed in $O(d|C_i| \cdot |C_j|)$ time. For any point of C_i (and symmetrically for C_j) we have to compute the maximum distance to a point in C_j (C_i, respectively) and store the maximum of this distance and the previous maximum of the point. From this information we can compute the kernel for the union of two clusters in the promised time bound and thus the whole cluster tree with Algorithm GHC1 in $O(d^2n^2)$ time and $O(dn)$ space. □

As mentioned in Section 2.2 we have to use the version of GHC1 where after an agglomeration step we reconsider all clusters C_j, $j > i$. Thus we have to include the line "j=i+1" in step 2biiA.

2.5.2 Minimizing the Mean Radius and Sum of Distances

Next we consider hierarchical centroid clustering minimizing the mean radius of the cluster, which is equivalent to minimize the sum of distances to the centroid point, i.e. both criteria generate the same centroid point. For a cluster $C \subseteq V^d$ let now the dissimilarity $d(C)$ be the mean distance (sum of distances) from points of C to its centroid point and similarly $d(C,p)$ the mean distance (sum of distances) from points of C to p. We first consider whether the δ-functions are monotonous, i.e. fulfill δ-function property 2).

Lemma 8 *For $C \subseteq V^d$ let the dissimilarity $d(C)$ be the mean distance (sum of distances) between vertices of C and the centroid p of C, where p minimizes this mean distance (sum of distances) among all possible points in either $\mathbb{R}^d$ or V^d. Then for the hierarchical centroid clustering the dissimilarity $d(C)$ is monotonous.*

Proof. The monotonicity of the sum of distances is obvious since by adding vertices (and thus distances) to a cluster we can only increase the sum. (We can not decrease the sum of distances by changing the centroid point, since the "old" centroid point already achieved the optimum.)

For the mean radius this is less obvious: To prove the lemma we basically reuse the idea of the proof of Lemma 6 (cf. Fig. 6), doing it by induction over the merge steps. It is clear that merging single points can not violate the monotonicity. So let the merge step $C = C_i \cup C_j$ be the first step where an inversion occurs, i.e. $d(C) < d(C_i)$ where we assume w.l.o.g that $d(C_i) \geq d(C_j)$. Then by induction the last (monotonous) merge operation in this subtree was the agglomeration of $C_i = C_i' \cup C_i''$.

Let p be the centroid point of C. From the inversion we have $d(C,p) = d(C) < d(C_i) \leq d(C_i,p)$. W.l.o.g. we assume that $d(C_i',p) \leq d(C_i'',p)$ and thus $d(C_i'',p) > d(C,p)$ since $d(C_i,p) = d(C_i' \cup C_i'',p)$. But then $d(C_i' \cup C_j) \leq (C_i' \cup C_j,p) = d(C \setminus C_i'',p) < d(C,p) = d(C) < d(C_i)$ i.e. merging C_j and C_i' results in a smaller dissimilarity than merging C_i' and C_i'', a contradiction to the current existence of C_i in the cluster tree. Thus $d(C)$ is indeed monotonous if we allow centroid points to be from $\mathbb{R}^d$ or V^d. □

Thus if centroid points are allowed to be in V^d or $\mathbb{R}^d$ then both δ-functions are monotonous. It is now interesting to observe what happens if we restrict the domain for centroid points to the set of vertices contained in cluster. Surprisingly in this case the two δ-functions have quite different monotonicity properties.

Lemma 9 *For $C \subseteq V^d$ let the dissimilarity $d(C)$ be the sum of distances between vertices of C and the centroid p of C, where p minimizes this sum of distances among all possible points in C. Then for the hierarchical centroid clustering the dissimilarity $d(C)$ is monotonous.*

Proof. The first part to prove the monotonicity is identical with the second paragraph of the proof of Lemma 8. Now let p be the kernel of C and observe that p can not be contained in C_i since then it immediately follows that $d(C) > d(C_i)$. Thus $p \in C_j$ and we have $d(C_i) > d(C) = d(C_i,p) + d(C_j,p) > d(C_i',p) + d(C_j,p) = d(C_i' \cup C_j,p) \geq d(C_i' \cup C_j)$. i.e. merging C_j and C_i' results in a smaller dissimilarity than merging C_i' and C_i'', a contradiction to the current existence of C_i in the cluster tree. □

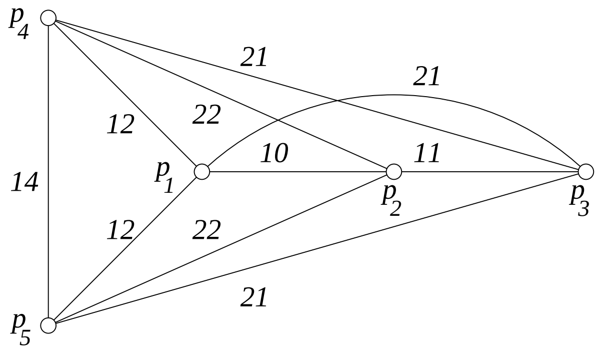


Figure 7: A five point example showing that hierarchical centroid clustering is not monotonous when centroid points are restricted to cluster points.

We now consider the case where for the kernel $p \in C$ $d(C)$ is the mean distance from the remaining points of C to p. The problem here is that in the proof of Lemma 8 either $d(C_i) \leq d(C_i,p)$ or $d(C_i' \cup C_j) \leq d(C_i' \cup C_j,p)$ need not to be true as was already pointed out in the proof of Lemma 7. In fact Figure 7 gives a five point example where this is the case and thus an inversion occurs. The given distances not only fulfill the triangle inequality but can easily be embedded into the hypercube of dimension 29 (cf. Table 1).

Let us give a brief explanation how the example works. In the first two steps of the hierarchy obviously points p_1, p_2 and p_3 are merged to a cluster C_A with kernel p_2 and mean radius $d(C_A) = 10.5$. For the next agglomeration step the two possibilities are either adding p_4 (or symmetrically

$p_1 =$	(0000000000 0000000000 000000000)
$p_2 =$	(1111111111 0000000000 000000000)
$p_3 =$	(1111111111 1111111111 100000000)
$p_4 =$	(0000000000 0000011111 100111111)
$p_5 =$	(0000000000 1111100000 111001111)

Table 1: The five points of Fig. 7 embedded into $\mathcal{C}^{29}$.

p_5) to C_A, resulting in a cluster C_B with kernel p_1 or p_2 and mean radius $d(C_B) = 14\frac{1}{3}$, or merging p_4 and p_5 to C_B with $d(C_B) = 14$. Thus the third step merges p_4 and p_5 to C_B. After the last step all points are in the final cluster C and p_1 minimizes the mean radius, i.e. is the kernel of C. But the mean radius of C is 13.75 which is less than $d(C_B) = 14$, the promised inversion.

One key fact for this example is that $d(C_B) = 14 > d(C_B, p_1) = 12$ and thus the argumentation used to prove Lemma 8 can not be applied.

Let us now take the algorithmic point of view if the centroid points are from either $\mathbb{R}^d$ or V^d. As for average linkage (cf. Section 2.4) we denote by $n_0(C,b)$ the number of elements from C with a '0' on position b and with $n_1(C,b)$ the number of elements from C with a '1' on position b, $b = 1, \ldots, d$. We first show that the centroid point minimizing the mean distance (sum of distances) over all points in either V^d or $\mathbb{R}^d$ can be computed very efficiently in the d-cube case.

Lemma 10 *The centroid point $p \in V^d$ of a cluster $C \subseteq V^d$ minimizing the mean distance (sum of distances) of points of C to p can be computed in $O(d|C|)$ time. Moreover using the hierarchical clustering method for n objects from V^d the centroid point of the union of two clusters $C_i, C_j \subseteq V^d$ can be computed in $O(d)$ time, using $O(dn)$ additional space for the whole cluster tree.*

Proof. Let $C_i = \{p\}$ and $C_j = C$ and consider the special case of Lemma 4 for these two clusters. From the prove given there it follows that we can minimize the sum of the distances between C_i and C_j, and thus the mean of the distances between p and C, if we choose p to be the "average element" of C: every bit position b of p is set to '1' if $n_1(C,b) > n_0(C,b)$ and to '0' else (thus ties are broken by favoring the assignment of '0'). Clearly $n_1(C,b)$ and $n_0(C,b)$ can be computed in $O(|C|)$ time and we can thus obtain the centroid point for C in $O(d|C|)$ time.

If we compute the centroid points during the hierarchical clustering we can store $n_0(C_i, b)$ and $n_1(C_i, b)$ for all clusters C_i of the tree, using only $O(dn)$ additional space for the whole cluster tree. Then the centroid point of a cluster C_i or the union of two clusters C_i, C_j can be computed in $O(d)$ time. □

With the above observation we can also handle the dynamic version of one-centroid problems for a given set C. Adding a vertex to C or removing it can be done in $O(d)$ time and space if we store and update $n_0(C,b)$ and $n_1(C,b)$ appropriately.

Lemma 11 *Let $p \in \mathbb{R}^d$ be the centroid point of a cluster $C \subseteq V^d$ minimizing the mean distance (sum of distances) of points of C to p, using the L_1-metric. Then either $p \in V^d$ or there exists another centroid point $p' \in V^d$ with the same mean distance (sum of distances) to points in C.*

Proof. For a fixed index $b \in \{1, \ldots, d\}$ consider a single coordinate value p^b of the centroid point p. Clearly p^b must be in the range $[0 \ldots 1]$. The sum of the distances of all elements from C to p for the direction b is $n_o(C,b)p^b + n_1(C,b)(1-p^b) = p^b(n_o(C,b) - n_1(C,b)) + n_1(C,b)$. We can minimize this sum by setting $p^b = 0$ if $n_o(C,b) \geq n_1(C,b)$ and $p^b = 1$ if $n_o(C,b) < n_1(C,b)$. Since this holds independently for all coordinates b we thus can minimize the sum and mean of distances by taking a centroid point from V^d. If $n_o(C,b) \neq n_1(C,b)$ for all directions $b = 1, \ldots, d$ then p is unique and in V^d. Else we have found one centroid point $p' \in V^d$ from an infinite set of possible centroid points with the same mean distance as p. □

Lemma 11 shows that we can handle the cases for V^d and $\mathbb{R}^d$ in the same way. Together with Lemma 10 it follows that properties 1) to 3) for δ-functions are fulfilled and therefore we can apply Algorithm GHC2.

Theorem 6 *Given a set $P \subseteq V^d$ with n elements. The centroid point of a cluster is from either V^d or $\mathbb{R}^d$ and minimizes the mean distance (sum of distances) of points from the cluster to the centroid point. Then the hierarchical centroid clustering tree of P can be computed in $O(dn^3)$ time and $O(dn)$ space.*

We use the same algorithm to minimize the mean radius and to minimize the sum of the distances. Moreover for a given cluster C the optimal centroid point is equivalent for both cases. Nevertheless the resulting cluster trees will be different in general, since the value of the dissimilarity is different for the two criteria. Therefore the choice which clusters are merged in the next step can be different and thus the methods result in different cluster trees.

As for average linkage the dissimilarity of a cluster can be computed very efficiently in $O(d)$ time and thus much faster than required by δ-function property 3). But again the bottleneck of Algorithm GHC2 is the number of updates of the best merge cluster information in the worst case. For experiments on centroid clustering see Section 4, showing that the average number of necessary update steps is small and thus Algorithm GHC2 is a practicable solution.

Finally let us consider the case where a centroid point has to be an element of the cluster itself. It is interesting to note that the one-kernel problem for minimizing the mean distance (sum of distances) can be solved in $O(dn^2)$ time with the same algorithm as for minimizing the maximum distance (cf. Theorem 5). Thus while minimizing the radius of a cluster gets easier when restricted to cluster points, the problem of minimizing the mean radius looses the monotonicity property and seems to be harder to compute in this case. For the problem of minimizing the sum of the distances to the kernel, we can again reuse the idea of Theorem 5 to perform the agglomeration step of two clusters C_i and C_j in $O(|C_i| \cdot |C_j|)$ time. Together with Lemma 9 we can thus apply Algorithm GHC2 and get

Theorem 7 *Given a set $P \subseteq V^d$ with n elements. The centroid point of a cluster has to be contained in the cluster and to minimize the sum of distances from points in the cluster to it. Then the hierarchical centroid clustering tree of P can be computed in $O(dn^3)$ time and $O(dn)$ space.*

3 Divisive Hierarchical and k-Clustering

Similar to agglomerative hierarchical methods divisive hierarchical clustering builds a hierarchical cluster tree out of a set $S \subseteq V^d$ of n objects. Whereas agglomerative methods start with the smallest clusters, i.e. the single objects, and build the hierarchy by gradually merging them, divisive algorithms work the other way round and start with the cluster C containing all objects. Step by step they divide the "largest" existing cluster into two smaller subsets, according to a certain optimization criterion which determines the different methods, until after $n-1$ steps all remaining clusters are single objects. As for agglomerative clustering a hierarchical cluster tree is again a family of clusters such that any two clusters are disjoint iff they lie on different paths (seen from the root), or one includes the other iff they are on the same path. Thus the properties of the cluster trees are the same as described in see Section 2. It is easy to see that all divisive methods we present are monotonous, thus we do not mention this explicitly in the sequel.

3.1 Two-Clustering

The crucial step for divisive clustering is the iterated "optimal" splitting of a cluster C into two disjoint subsets C_1 and C_2. The choice of an optimal two-clustering determines the hierarchical clustering and thus the cluster tree. Therefore in the following two subsections we consider the optimal division of a set $C \subseteq V^d$ of hypercube vertices into two disjoint subsets C_1 and C_2. The first subsection gives two algorithms to minimize the maximum diameter of the resulting subclusters, and to maximize the minimum inter-cluster distance between them, respectively. In the second subsection we consider two-center problems.

Related to two-clustering is the k-clustering problem which is well known to be NP-complete for many cases. Since k-clustering turns up to be a subproblem of divisive hierarchical clustering for several criteria, we will give some brief remarks on k-clustering in Section 3.2.

3.1.1 Diameter and MST Two-Clustering

A first optimization criterion for two-clustering might be to separate objects of different clusters as good as possible, i.e. to maximize the minimum distance between them. Obviously this task is related to the minimum spanning tree (MST) problem, cf. Section 2.1.1. Removing the longest edge of the MST results in two connected components with the required property, and thus from Lemma 1 we immediately get the following:

Theorem 8 *Given a set $C \subseteq V^d$ with n elements. A two-clustering of C maximizing the minimum distance between objects of different clusters can be computed in $O(dn^2)$ time and $O(dn)$ space.*

Another evident optimization criterion for two-clustering is the minimization of the maximum diameter of the resulting clusters. In contrast to the previous criterion this means that we minimize the maximum dissimilarity of objects within the same cluster. It can thus be seen as the divisive counterpart of C-link. An $O(d^2n^2)$ time algorithm is given in Figure 8.

The basic idea of the algorithm is to consider the interpoint distances in non-increasing order and then to mark the respective points as belonging to different clusters as long as possible. To this end we use a disjoint-set data structure (also called union-find data structure, [6]) where we put all points into the same set iff their pairwise classification is already fixed. That is, a set contains all vertices where the membership of a vertex to either cluster 'A' or 'B' is fixed with respect to all other vertices of the same set. Thus a single set might be seen as a bipartite graph of objects, where one side is labeled 'A' and the other 'B'. The "edges" between the vertices of the graph relate to the distances already concerned, i.e. all distances above the actual length.

Whenever two points achieve a given distance we have to check whether they already belong to the same set. If this is the case we also consider their labels and if the labels are different nothing remains to do, since we already know that they have to belong to opposite clusters. If the points have the same label then we have found an odd cycle of pairwise relations, where the current "edge" represents the smallest distance within the cycle. In this case we know that at least one subcluster has the actual distance as lower bound for its diameter. We thus can label all still unlabeled vertices arbitrarily and assign all elements labeled 'A' to a cluster 'A' and equally for 'B'.

If the two points are belonging to different sets (e.g. they are still single points) we mark them as 'A' and 'B' accordingly and unify the two sets. If both points were already marked with the same label we exchange the labels of all elements of the smaller set, containing one of the points, before we unify them; see steps 2b i.-iii. of the algorithm given in Figure 8 for details. We thus maintain that a set contains only vertices with fixed pairwise relations and that different sets are independent from each other.

1. Label every element $c_i \in C$ as 'blank' and put it into union i
2. **FOR** $diss = d$ **DOWN TO** 1 **DO**

   FOR $i = 1$ **TO** n **DO**

   FOR $j = i + 1$ **TO** n **DO**

   IF $d_H(c_i, c_j) = diss$ **THEN**

   (a) **IF** find(c_i)=find(c_j) **THEN**
      i. **IF** label(c_i)=label(c_j) **THEN**
         A. Label all elements still labeled 'blank' by either 'A' or 'B' arbitrarily
         B. According to their labeling separate the elements into the two clusters 'A' and 'B', minimizing the maximum diameter $diss$, achieved by c_i and c_j
         C. Output clusters 'A' and 'B' and **STOP**

   (b) **ELSE** {c_i and c_j belong to different sets}
      i. **IF** c_i and c_j are labeled 'blank' **THEN**
         A. Set label(c_i)='A' and label(c_j)='B'
         B. Union c_i and c_j

      ii. **ELSE IF** either c_i or c_j is labeled 'blank' **THEN**
         A. Label the 'blank' element conversely to the labeled element
         B. Union c_i and c_j

      iii. **ELSE** {both elements are labeled}
         A. **IF** label(c_i)=label(c_j) **THEN**
            Exchange all labels of the smaller set
         B. Union the sets containing c_i and c_j

Figure 8: Two-clustering algorithm minimizing the maximum diameter for $C \subseteq V^d$ with running time $O(d^2n^2)$.

We repeate the above steps (in order of non-increasing distance) until an odd cycle of "AB-relations" appears. Since in this way we avoid an odd cycle of long distances as long as possible the algorithm clearly minimizes the maximum diameter of the subclusters.

We now consider the running time of the algorithm. For the union-find operations we use the linked-list representation of disjoint sets and the weighted-union heuristic of [6] where we always append the smaller list onto the longer. Thus a sequence of n union and m find operations takes $O(m + n \log n)$ time. Generating all point distances in non-increasing order is done by the three FOR loops and the first IF statement of step 2 and takes thus $O(d^2n^2)$ time. (Note that we can not use any other kind of sorting the $O(n^2)$ distances since we are limited to linear space.) In 2a we have $O(n^2)$ find operations and in 2b $O(n)$ unions. Thus all disjoint set operations can be done in $O(n^2)$ time. The steps in 2ai. can be done in $O(n)$ and all label operations in 2b i.-iii. can be done within the same time as the related union operations. We thus get an overall running time of $O(d^2n^2)$.

Theorem 9 *Given a set $C \subseteq V^d$ with n elements. A two-clustering of C minimizing the maximum diameter of the resulting clusters can be computed in $O(d^2n^2)$ time and $O(dn)$ space.*

For practical implementations we can speed up the iteration of *diss* as mentioned for Algorithm GHC1: We decrease *diss* not only by one but reduce it to the next relevant value, i.e. for the next iteration we consider the largest distance which appeares during the current iteration and is strictly less than the actual value of *diss*. Clearly this does not effect the worst case running time, but it may significantely reduce the number of iterations of the outer loop. See Section 4 for experimental results.

3.1.2 Centroidal Two-Clustering

For centroidal two-clustering we ask in general for two centroids $c_1, c_2 \in V^d$ and a division of the given set $S \subseteq V^d$ into two subsets S_1 and S_2 such that the maximum of a given function of these subsets is minimized. For example if we want to minimize the maximum radius we have to optimize the term $\min_{c_1,c_2 \in V^d} \max_{p_i \in S} \min\{d_H(p_i, c_1), d_H(p_i, c_2)\}$. Note that in this example the division of S into S_1 and S_2 follows implicitly from the assignment of p_i to either c_1 or c_2.

As in Section 2.5 we distinguish between to different kinds of centers of a cluster: We either allow arbitrary centroid points from V^d as in the example above, where we will talk of two-centroid problems in the sequel. Or for a cluster $C \subseteq V^d$ we are restricted to an element of C as its centroid point, the so called kernel of C. These problems are then called two-kernel problems.

In the first case we know already that even the one-centroid problem is open if we want to minimize the radius. Clearly if we could solve the corresponding two-centroid problem efficiently, we could also solve the one-centroid problem: Let $S \subseteq V^d$ be the vertex set of the the one-centroid problem. Let S' be S embedded into a $(2d+1)$-dimensional hypercube by adding $d+1$ leading zeros to every vertex, and add the vertex $p = (1, \ldots, 1) \in V^{2d+1}$ to S'. The optimal solution of the two-centroid problem for S' now contains p and the optimal centroid of S as its two centroids. Therefore the complexity status of the two-centroid problem minimizing the maximum radius seems to be open, too.

In Section 2.5.2 we have seen that for the one-centroid problem the mean radius and the sum of distances can be minimized in $O(dn)$ time. Surprisingly the corresponding two-centroid problems seem to be much more difficult and in fact we are not aware of any polynomial time solution. One reason that the approach for the one-centroid problem can not be generalized is the fact that the bits of the centroid points are not longer independent. While for a single centroid point we can optimize every bit position separately, we now have a completely different situation. The bits of the two centroid points and the division of the vertex set S are highly nested. Changing a single bit of a centroid point c_1 may cause a different split of S and thus a different subset for which we want to optimize c_1. Note that once the division of S is known we can easily compute the two centroid points in $O(dn)$ time.

Let us now briefly consider two-kernel problems. A straight forward solution is to compute all $\binom{n}{2}$ pairs of vertices from S as kernels and for each of these pairs to check the optimal assignment of the points in S. For minimizing the maximum radius we therefore just have to assign each vertex to its nearest kernel which can thus be done in $O(dn^3)$ time. But for e.g. minimizing the maximum mean distance it is not clear to the author whether this can be done as efficiently. Using the $O(dn^3)$ time two-clustering for a divisive hierarchical clustering leads to an $O(dn^4)$ time algorithm. Although some provisional tests showed that this bound is rarely achieved for uniformly distributed point sets, the algorithm is to time consuming compared to other methods presented in this work. Thus we will not use centroidal two-clustering to develop divisive hierarchical clustering methods.

3.2 k-Clustering

A classical result on k-clustering is the following: Given a finite set X, a distance $d(x,y) \in Z_0^+$ for each pair $x, y \in X$, and two positive integers k and b. The question whether there is a partition of X into k disjoint sets $X_1, \ldots, X_k$ such that $d(x,y) \leq b$ for all pairs $x, y \in X_i$, $1 \leq i \leq k$, is NP-complete [11]. It remains even NP-complete for fixed $k = 3$ and all distances in $\{0, 1\}$. Variants in which the sum of $\max_{x,y \in X_i}\{d(x,y)\}$ or $\sum_{x,y \in X_i} d(x,y)$ over all X_i is bounded by b are similarly NP-complete [11].

A simple example for polynomial time k-clustering is the task to maximize the minimum inter cluster distance. From the discussion of the last section it follows that for a given point set S we just have to remove the $k-1$ longest edges of the MST(S) to obtain an optimal k-clustering. Any other division into k clusters would mean to remove another set of at least $k-1$ edges of the MST(S) and thus cause a shorter minimum inter cluster distance. Removing the edges from the MST(S) can be done in $O(dn)$ time and space and thus by Lemma 1 it immediately follows that we can compute a k-clustering of S maximizing the minimum distance between objects of different clusters in $O(dn^2)$ time and $O(dn)$ space.

3.3 Divisive Hierarchical Clustering

As we have seen for two-clustering the problem of maximizing the minimum distance between objects of different subclusters of a set S is related to the $MST(S)$. Clearly for this optimization criterion the optimal k-clustering of S and the resulting cluster tree after $k-1$ steps of a divisive hierarchical clustering are identical, since in both cases we just have to remove the $k-1$ longest edges of the corresponding MST. We thus get

Theorem 10 *Given a set $S \subseteq V^d$ with n elements. A divisive hierarchical cluster tree of S maximizing the minimum distance between objects of different clusters can be computed in $O(dn^2)$ time and $O(dn)$ space.*

Let us now consider the more challenging case of divisive hierarchical clustering minimizing the maximum diameter of the clusters. From Section 3.1.1 we know that one division step of this hierarchical clustering can be done in $O(d^2n^2)$ time if we only “locally” minimize the cluster diameter. If we also want to take into account that the resulting clusters are iteratively divided and that assigning unlabeled vertices or independent sets arbitrarily to cluster 'A' or 'B' affects the optimal division of the next level (or even more levels further down) we immediately get at least a three clustering problem. (Note that in steps 2ai A. and B. of the algorithm given in Figure 8 the assignment of the vertices is in fact arbitrary in the sense that we might exchange the labels 'A' and 'B' without effecting the minimization of the maximum diameter.)

From the last section we know that three clustering to minimize the maximum diameter is already NP-complete and thus the best we can hope for is a heuristic, locally trying to avoid all long distances. E.g. all not yet marked vertices p in step 2aiA might be assigned to the cluster with the smaller maximum distance to p and the labels of a set of step 2aiB might be exchanged in order to minimize the maximum distance between vertices of the set and vertices of the corresponding (partially built) clusters. Clearly all distances we additionally have to check for this heuristic were not considered in the algorithm before, since else the concerned vertices would have to belong to the same set already.

In other words we may continue the steps of the two-clustering algorithm of Figure 8, even after determining the diameter of the resulting subclusters, until we have considered all inter point distances. To this end we just have to skip steps 2ai A to C in Figure 8; if required we instead have only to store the largest value of $diss$, which occurs in this section, as the resulting diameter. Thus one dividing step can still be done in $O(d^2n^2)$ time, including the heuristic post processing. This

algorithm	based on	cf. Section	worst case	see Appendix
SLINK	MST	2.1	$O(dn^2)$	Tab. 3, 17; Fig. 11, 16
CLINK1	GHC1	2.3	$O(d^2n^2)$	Tab. 4, 18; Fig. 12, 21
CLINK3	GHC1	2.3	$O(d^2n^2)$	Tab. 5
MINRADI	GHC1	2.5.1	$O(d^2n^2)$	Tab. 6, 19
CLINK2	GHC2	2.3	$O(dn^3)$	Tab. 7, 20; Fig. 13, 17 22
ALINK	GHC2	2.4	$O(dn^3)$	Tab. 8
ASUMLNK	GHC2	2.4	$O(dn^3)$	Tab. 9
AMEANLNK	GHC2	2.4	$O(dn^3)$	Tab. 10
MINRAD2	GHC2	2.5.1	$O(dn^3)$	Tab. 11, Fig. 14
CTRDMEAN	GHC2	2.5.2	$O(dn^3)$	Tab. 12, Fig. 19
CTRDSUM	GHC2	2.5.2	$O(dn^3)$	Tab. 13
KRNLSUM	GHC2	2.5.2	$O(dn^3)$	Tab. 14, 21; Fig. 20, 23; 25
DIVIDIAM	divisive	3.3	$O(d^2n^2)$	Tab. 15, 22; Fig. 15, 18, 24
DIVIDIA2	divisive	3.3	$O(d^2n^3)$	Tab. 16

Table 2: Implemented and examined clustering algorithms.

yields a divisive hierarchical cluster algorithm minimizing the maximum diameter with $O(d^2n^3)$ time and $O(dn)$ space. Note however that still the order of assigning the sets and thus the given order of the vertices might influence the result of this heuristic.

If we do not add the heuristic to the division step we can improve the running time by a linear factor: Whenever we perform one division step we do not need to consider all previously considered inter point distances in the sequel. Therefore the generation of all point pairs in order of non-increasing distance needs only to be done once for the whole hierarchical clustering and takes $O(d^2n^2)$ time. Moreover all disjoint set operations can be done in $O(n^2 \log n) = O(dn^2)$ time since we have $O(n^2)$ find and not more than $O(n^2)$ union operations. Together with the discussion of Section 3.1.1 we thus get an overall running time of $O(d^2n^2)$.

Theorem 11 *Given a set $S \subseteq V^d$ with n elements. A divisive hierarchical cluster tree of S (locally) minimizing the maximum diameter of the clusters can be computed in $O(d^2n^2)$ time and $O(dn)$ space.*

4 Experimental Results

In this section we give numerical results on the number of necessary vertex comparisons, the running time and the number of rounds for the presented algorithms. We do not consider the quality of the resulting clusters, since this mainly depends on the application in mind and can thus not be made without specific requirements. We are therefore only interested in the efficiency of the implementations.

Additional to all mentioned clustering algorithms we have implemented some more variants to compare the behavior of the different basic approaches. Overall we thus have 14 different clustering algorithms. Table 2 gives an overview of the algorithms, their name, the basic method used, the worst case running time, the section they are described and where to find corresponding tables and figures.

Let us first give a brief overview of the examined algorithms. SLINK is the the S-link implementation of the MST-algorithm. CLINK1 is the implementation of C-link using GHC1 and skipping unnecessary steps in the FOR-iteration as mentioned in Section 2.2, while CLINK3 uses the simple FOR-loop and goes through all iterations. CLINK2 implements the C-link method using GHC2. MINRADI minimizes the radius of the cluster to its kernel and MINRAD2 is a variant of it based on

GHC2. We will uses the last four mentioned algorithms to compare the two basic approaches GHC1 and GHC2.

ALINK, ASUMLNK and AMEANLNK are the three average linkage methods A-link, A_{sum}-link and A_{mean}-link of Section 2.4. CTRDMEAN minimizes the mean distance of the vertices to the centroid of a cluster while CTRDSUM minimizes the sum of distance to the centroid and KRNLSUM minimizes the sum of distance to the kernel of a cluster, respectively.

Finally DIVIDIAM is a divisive clustering algorithm minimizing the maximum diameter of the subclusters. While DIVIDIAM stops for one division step after determining the minimum diameter of the subclusters, DIVIDIA2 uses the heuristic mentioned in Section 3.3 and has thus to go through all iterations for any divisive step.

In a first phase we have generated 42 test sets with uniformly distributed random vertices. For dimensions $d = 16, 32, 64, 128, 256, 512$ we used set sizes of $n = 100, 200, 500, 1000, 2000, 5000$ and 10000 elements. All 14 implemented algorithms were used to compute their cluster trees on these sets.

In a second phase we considered special point sets to get additional information. To this end we generated the vertices around a fixed number k of centers, such that the resulting sets are structured sphere-like around each center. Note, however, that the centers need not to belong to the set. We used $k = 10$ centers and required the Hamming distance between two centers to be $\geq \frac{d}{2}$. The points were uniformly distributed around these centers with the limitation that the Hamming distance to the corresponding center must not exceed $\frac{d}{8}$. Thus it is easy to see that no reasonable clustering method (especially non of the implemented algorithms) will combine vertices from different centers, as long as not every center has already "collected" all of its vertices. This forces of course a very special behavior of the algorithms and we will see that in fact the experimental results are sometimes quite different compared to that for random sets.

For these special sets we used the same dimensions and number of vertices as for random sets, except that for $d = 16$ we skipped all sets of size $n \geq 2000$. This was necessary because for $d = 16$ it is not possible to compute a set with more than 1370 vertices fulfilling the above requirements. Moreover we examined only the most interesting algorithms for these sets, since implementations based on the same approach behaved very similar.

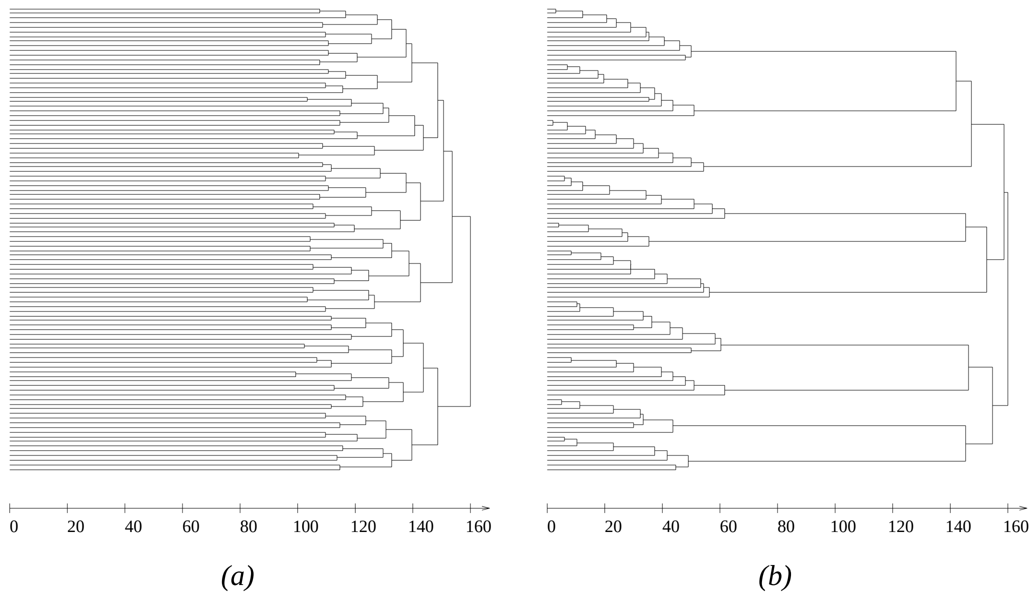


Figure 9: Dendrograms for CLINK1 with n=100 objects of dimension d=256; (a) for a random vertex set and (b) for a special vertex set with 10 cluster centers.

Figure 9 gives two typical dendrograms (cf. Section 2) for CLINK1 with $n = 100$ elements of dimension d=256. On the left side (a) shows a uniformly distributed random set. The difference between the distance of the minpair (approx. 100 bit) and the diameter of the final cluster (approx. 160 bit) is rather small. For the special set given in (b) one directly can see the 10 "cluster balls". Note the significant distinction of the diameter of these subclusters compared to the final cluster.

All experiments were done on customary personal computers (Pentium© processor with 90 MHz) and clearly the running time may differ on other systems. Therefore we also give the number of necessary vertex comparisons, which is the most basic operation of all mentioned methods and thus strongly related to the running time. Note, however, that we distinguish between two different ways of vertex comparisons. The first one is the usual computation of the Hamming distance between two vertices, as e.g. used for S-link or C-link. But for another class of algorithms, which use the idea of storing the number of ones and zeros for every bit position explicitly (e.g. A-link), we have to compare integer vectors of size d, refer to Sections 2.4 and 2.5.2 for details.

Although both variants run asymptotically in $\Theta(d)$ time, the first one has a much lower constant. This reflects the fact that on usual computers the Hamming distance of a 16 or 32 bit number can be computed within one step. Thus the constants of the second class of algorithms is expected to be significantly larger. In fact we can observe the following from the tables. While for algorithms using the binary distance computation we can perform up to about 1.4 10^5 vertex comparisons per second on the mentioned hardware, for ALINK, ASUMLNK, AMEANLNK, CTRDMEAN and CTRDSUM we have a typical number of 1000 to 10000 (with a maximum of 30000 for $d = 16$) vector computations. Clearly the actual number of comparisons per second linearly depends on the size of the vertices (vectors), i.e. the dimension. Thus the running time to compute the entire cluster hierarchy includes an additional factor d, which can be seen from the given curves. See Figures 16 to 18 and compare them to the number of vertex comparisons, given in Figures 11, 13 and 15, respectively.

Since it is clear from the algorithms which comparisons are involved, we will just speak of number of 'comparisons' in the sequel. But one should have this in mind when comparing the number of comparisons between different algorithms.

All tables and figures of experimental results are given in Appendices A to D

The tables containing the running time, the number of necessary comparisons and, for GHC2-based algorithms, the number of iterations, for random vertex sets are given in Appendix A.

Appendix B contains a selection of curves showing the number of comparisons, the running time and, for GHC2, the number of iterations for several implementations.

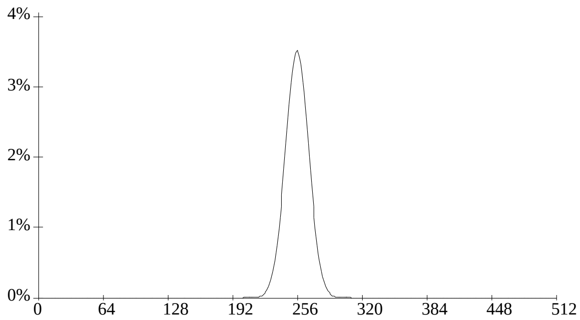
Similarly Appendices C and D contain tables and figures, respectively, for the above mentioned special vertex sets. For these sets we included only the more significant results.

Let us now point out some interesting observations made out of the experiments.

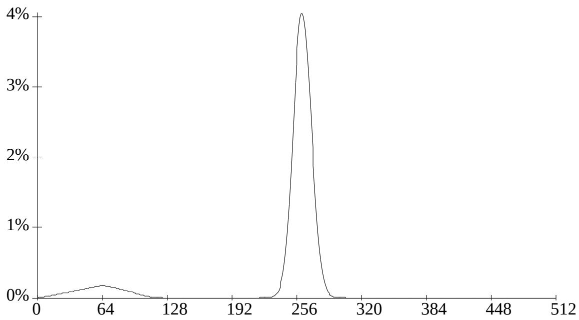
- The number of expected iterations for GHC2-based algorithms depends logarithmically on the number of vertices, see e.g. Figures 19, 20 and 25. Thus the running time of the algorithms was observed to be $O(dn^2 \log n)$ rather than $O(dn^3)$. For random sets the dependency of the number of iterations on the dimension d is rather weak for all algorithms, cf. the typical curves in Fig. 20, except for AMEANLNK and CMEANLNK (see Fig. 19).

- For GHC1-based algorithms it is, as expected, more efficient to skip unnecessary iterations in the FOR-loop (cf. Section 2.2), as can be seen from the Tables for CLINK1 (Tab. 4) and CLINK3 (Tab. 5).

- Considering the dependency of the number of vertex comparisons from the dimension d for random sets the algorithms behave quite differently. For example SLINK is entirely independent from d (c.f. Fig. 11), while e.g. for CLINK1 (c.f. Fig. 12) and DIVIDIAM (c.f. Fig. 15) there is a very distinctive relation. This is typical for all implementations based on GHC1 and for both divisive algorithms. In contrast the dependency for GHC2 based algorithms is rather weak, see e.g. Figures13 and Figures14.



(a)



(b)

Figure 10: Percentage of pairwise Hamming distances for (a) uniformly distributed vertices and (b) a specially structured set (see text) with $n = 10000$ vertices and dimension $d = 512$.

- For special point sets most implemented algorithms used approximately the same or more time to compute the cluster hierarchy than for the corresponding random sets. Especially algorithms based on GHC2 were running up to 10 times longer (e.g. MINRAD2), while SLINK, algorithms based on GHC1 and divisive algorithms showed not to much alteration. Sometimes they used even less comparisons than for random sets (e.g. DIVIDIAM, see Table 22). Compared to random sets GHC1-based algorithms are typically faster for small dimensions and large sets, e.g. CLINK1 or MINRADI for d=32,64 and n=10000 (Tables 18 and 19). For high dimensions (d=256 and more) special point sets may need significantly more comparisons (Tab. 19).

  So what caused the break down of the GHC2 method? Considering the Hamming distance between vertices of a random point set it is easy to show that with high expectation all these distances are arranged around $\frac{d}{2}$. A typical range for $d = 512$ and $n = 10000$ is about $\{200, \ldots, 300\}$, see Figure 10(a). Since these distances are "not too small" there typically is a large number of BMC-pairs which can be agglomerated in parallel. This may be the main reason why the number of iterations for GHC2-based algorithms is much less than the worst case $n - 1$.

  Unfortunately the specially structured sets now force the vertices to build smaller "balls" and thus within these balls a lot of small distances appear. In our examples with 10 cluster centers, for a minimum Hamming distance of $\frac{d}{2}$ between two centers, and for a maximum radius of $\frac{d}{8}$, these intra-ball distances are in the range of $\{0, \ldots, \frac{d}{4}\}$. In addition the inter-ball distances can be in the range $\{\frac{d}{4}, \ldots, d\}$, i.e. are still around $\frac{d}{2}$ (cf. Fig. 10(b)). Within one ball the dependency of the BMC-relations is now much higher due to the small and thus often multiple distances. Thus during one iteration of GHC2 only a small number of pairs can be merged. Furthermore, as long as not all pairs within a ball are merged, no BMC outside the ball can be used. Experiments showed that sometimes only one or two agglomeration steps per ball were possible. This increases the number of necessary iterations and thus the running time.

- DIVIDIAM is the only algorithm which runs faster, and has a smaller number of vertex comparisons, for all tested special sets compared to the corresponding random sets. One intuitive explanation is that during the initial iterations DIVIDIAM determines clusters related to the

10 balls around the given centers. Thus for the special sets the advantage of divisive methods, that after the first few iterations only distances within subclusters have to be considered, fully takes effect. Since for our special sets such smaller subclusters are more homogeneous and can be built faster than for random sets, the overall number of necessary vertex comparisons, and thus the running time is smaller.

- For special point sets the dependency for GHC2 based algorithms from the dimension d is stronger. Compare for example the number of comparisons for CLINK2 (Fig. 13 vs. Fig. 22) or the number of iterations for KRNLSUM (Fig. 20 vs. Fig. 25). For DIVIDIAM the situation is contrary, see Fig. 15 vs. Fig. 24. This again reveals the fact that divisive methods benefit from the special structure of these sets, as was already pointed out in the last paragraph.

- All algorithms based on GHC1 may alternatively be implemented using GHC2. Comparing the tables for CLINK1 and CLINK2 or MINRADI and MINRAD2, respectively, we observe that for random point sets they provide faster running times if the dimension is high enough (e.g. d=256 or 512). In this case they should be considered as a true alternative solution, although their worst case running time is larger.

  But note that for special point sets this is not necessary the case. Here CLINK2 is only faster than CLINK1 if the number of vertices is small enough, for example $d = 512$ and $n = 200$. The same is true when comparing MEANRADI and MEANRAD2. Thus for specially structured sets GHC1-based algorithms may be the better choice, as was already pointed out above.

- What are the fasted algorithms? As to be expected from the worst case running time, S-link is by far the fastest method. The second best implementation was the divisive method minimizing the diameter of the subclusters, DIVIDIAM. But also all other algorithms run efficiently for appropriately sized inputs. GHC2-based algorithms run fast for random vertex files, like KRNLSUM which was among the best. Especially for large vertex sets these algorithms may beat GHC1-based implementations. But for specially structured sets they may break down and should not be used.

5 Open Problems

An obvious question concerns the metric to compute the distance of two objects. The measure we used, the Hamming distance, is just the L_1-metric of $\mathbb{R}^d$. For objects from V^d clearly L_∞ makes no sense and for $2 \leq p < \infty$ we get $L_p = (L_1)^{\frac{1}{p}}$, i.e. the dependence between L_1 and L_p is monotone. Since most presented methods (except those using sum or mean of distances) only rely on the relative distance order of pairs of objects, a different choice of p would not change the resulting clusters, except the value of their weight.

Some open questions mentioned in previous sections concern the time complexity of the following problems (as functions of n and d):

- Centroidal clustering, for minimizing the radius to a centroid point in V^d (cf. Section 2.5.1)
  INSTANCE: A vertex set $S \subseteq V^d$, $|S| = n$.
  QUESTION: Compute the vertex $p \in V^d$ which minimizes the maximum distance of a point $q \in S$ to p.

- Two-centroid clustering, for minimizing the maximum sum of distances in subclusters (cf. Section 3.1.2)
  INSTANCE: A vertex set $S \subseteq V^d$, $|S| = n$.

QUESTION: Compute two centroids $c_1, c_2 \in V^d$ and a division of S into S_1 and S_2, such that $\max\left\{\sum_{p_i \in S_1} d_H(p_i, c_1), \sum_{p_j \in S_2} d_H(p_j, c_2)\right\}$ is minimized.

Related to this problem is the following question, which is also of interest in its own.

- Two-centroid clustering, for minimizing the maximum sum of distances to two given centroids (see Section 3.1.2)
  INSTANCE: A vertex set $S \subseteq V^d$, $|S| = n$, and two vertices $c_1, c_2 \in V^d$.
  QUESTION: Split S into S_1 and S_2, such that $\max\left\{\sum_{p_i \in S_1} d_H(p_i, c_1), \sum_{p_j \in S_2} d_H(p_j, c_2)\right\}$ is minimized.

A variant of this problem asks to minimize the mean distance to the centroids, i.e. to minimize the term $\max\left\{\frac{1}{|S_1|}\sum_{p_i \in S_1} d_H(p_i, c_1), \frac{1}{|S_2|}\sum_{p_j \in S_2} d_H(p_j, c_2)\right\}$.
For alternative divisive hierarchical clustering methods the following question is of interest.

- Are there more efficient methods for the two-kernel problems mentioned in Section 3.1.2 ?

Acknowledgements We would like to thank Franz Aurenhammer for several discussions and helpful comments on the presented topic and for carefully reading drafts of this paper.

References

[1] O. Aichholzer, Efficient $\{0,1\}$-String Searching Based on Pre-clustering, SFB Report Series 94, TU-Graz, December 1996,

[2] O.Aichholzer, Practical Implementations of the Complete Linkage Clustering Method, Manuscript, 1996

[3] F. Aurenhammer, R. Klein, Voronoi Diagrams, to appear as Chapter 18 in Textbook on Computational Geometry, J. Sack and G. Urrutia (eds), in preparation

[4] H.H. Bock, Automatische Klassifikation, Studia Mathematica, Vandenhoeck & Ruprecht, Göttingen, 1974

[5] B. Büeler, A. Enge, K. Fukuda, H.-J. Lüthi, Exact Volume Computation for Polytopes: A Practical Study, Abstracts of the 12th European Workshop on Computational Geometry, Münster 1996, pp 57-64

[6] T.H. Cormen, Ch.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press Cambridge and McGraw-Hill Book Company NY, 1991

[7] W.H.E. Day, H. Edelsbrunner, Efficient Algorithms for Agglomerative Hierarchical Clustering Methods, Journal of Classification Vol 1, pp 7-24, 1994

[8] D. Defays, An efficient algorithm for a complete link method, The Computer Journal, Vol. 20, No 4, 1977, pp 364-366

[9] B.S. Duran, P.L. Odell, Cluster Analysis - A Survey, Lecture Notes in Economics and Mathematical Systems 100, Springer Verlag, 1974

[10] K. Florek, J. Łukaszewicz, J. Perkal, H. Steinhaus, S. Zubrzycki, Sur la liaison et la division des points d'un ensemble fini, Colloquium Math., Vol 2, 1951, pp. 282-285

[11] Michael R. Garey, David S. Johnson, Computers and Intractability - A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, New York, 1979

[12] A.El-Hamdouchi, P. Willett, Comparison of Hierarchical Agglomerative Clustering Methods for document Retrieval, The Computer Journal, Vol. 32, No.3, 1989

[13] J. A. Hartigan, Clustering Algorithms, Wiley Series in Probability and Mathematical Statistics, John Wiley & Suns, NY, 1975

[14] S.C. Johnson, Hierarchical clustering schemes, Psychometrika 32, 1967, pp 241-254

[15] D. Krznaric, Ch. Levcopoulos, The first Subquadratic Algorithm for Complete Linkage Clustering, Proceedings of ISAAC'95, Ed. J. Staples et.al., LNCS 1004 Springer, Dec 1995, pp. 392-401

[16] P. MacNaughton-Smith, Some statistical and other numerical techniques for classifying individuals, Home Office Research Unit Report No. 6, H.M.S.O., London, 1965

[17] F. Murtagh, A Survey of Recent Advances in Hierarchical Clustering Algorithms, The Computer Journal, Vol 26, No.4, 1983, pp 354-359

[18] L.L. McQuitty, Hierarchical linkage analysis for the isolation of types, Educational Psych. Measurement 20, 1960, pp 55-67

[19] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994

[20] R. Seidel, Exact Upper Bounds for the Number of Faces in d-Dimensional Voronoi Diagrams. in Applied Geometry and Discrete Mathematics, Vol 4., P. Gritzmann and B. Sturmfels (ed.), pp. 517–530, 1991.

[21] A. Schrijver, Theory of Linear and Integer Programming, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, Chichester, 1986.

[22] R. Sibson, SLINK: An optimally efficient algorithm for the single-link cluster method, The Computer Journal, Vol 16, No 1, 1973, pp30-45

[23] G. Sierksma, Linear and Integer Programming - Theory and Practice, Marcel Dekker, Inc, New York, 1996

[24] T. Sørenson, A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons, Biol. Skrifter 5, 1948, pp 1-34

[25] E.M. Voorhees, The Effectiveness and Efficiency of Agglomerative Hierarchic Clustering in Document Retrieval, Ph.D. thesis, Cornell University, 1985

A Tables for Uniformly Distributed Random Vertex Sets

Appendix A summarizes the results of our experimental section for uniformly distributed random vertex sets in 14 tables. A discussion of the results is given in Section 4.

Tables 3 to 16 show the running time of the algorithms in seconds, and the number of comparisons, respectively, in dependency of the dimension d and the number of elements n. Tables 7 to 14 show the results for methods based on Algorithm GHC2 and give in addition the number of iterations of the WHILE loop (cf. Figure 4). Finally Tables 15 and 16 give the results for divisive methods.

$n \setminus d$	16	32	64
100	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$
200	$1.70\ 10^{-1}$ / $1.99\ 10^{4}$	$1.10\ 10^{-1}$ / $1.99\ 10^{4}$	$1.60\ 10^{-1}$ / $1.99\ 10^{4}$
500	$1.05\ 10^{0}$ / $1.24\ 10^{5}$	$1.10\ 10^{0}$ / $1.24\ 10^{5}$	$1.15\ 10^{0}$ / $1.24\ 10^{5}$
1000	$4.62\ 10^{0}$ / $4.99\ 10^{5}$	$4.50\ 10^{0}$ / $4.99\ 10^{5}$	$5.11\ 10^{0}$ / $4.99\ 10^{5}$
2000	$1.81\ 10^{1}$ / $1.99\ 10^{6}$	$1.89\ 10^{1}$ / $1.99\ 10^{6}$	$2.06\ 10^{1}$ / $1.99\ 10^{6}$
5000	$1.23\ 10^{2}$ / $1.25\ 10^{7}$	$1.32\ 10^{2}$ / $1.25\ 10^{7}$	$1.43\ 10^{2}$ / $1.25\ 10^{7}$
10000	$5.34\ 10^{2}$ / $5.00\ 10^{7}$	$5.64\ 10^{2}$ / $5.00\ 10^{7}$	$6.07\ 10^{2}$ / $5.00\ 10^{7}$
$n \setminus d$	128	256	512
100	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$6.00\ 10^{-2}$ / $4.95\ 10^{3}$	$1.10\ 10^{-1}$ / $4.95\ 10^{3}$
200	$2.70\ 10^{-1}$ / $1.99\ 10^{4}$	$2.20\ 10^{-1}$ / $1.99\ 10^{4}$	$3.90\ 10^{-1}$ / $1.99\ 10^{4}$
500	$1.59\ 10^{0}$ / $1.24\ 10^{5}$	$1.43\ 10^{0}$ / $1.24\ 10^{5}$	$2.25\ 10^{0}$ / $1.24\ 10^{5}$
1000	$6.21\ 10^{0}$ / $4.99\ 10^{5}$	$5.49\ 10^{0}$ / $4.99\ 10^{5}$	$9.01\ 10^{0}$ / $4.99\ 10^{5}$
2000	$2.41\ 10^{1}$ / $1.99\ 10^{6}$	$2.20\ 10^{1}$ / $1.99\ 10^{6}$	$3.58\ 10^{1}$ / $1.99\ 10^{6}$
5000	$1.64\ 10^{2}$ / $1.25\ 10^{7}$	$1.44\ 10^{2}$ / $1.25\ 10^{7}$	$2.30\ 10^{2}$ / $1.25\ 10^{7}$
10000	$6.93\ 10^{2}$ / $5.00\ 10^{7}$	$6.00\ 10^{2}$ / $5.00\ 10^{7}$	$9.42\ 10^{2}$ / $5.00\ 10^{7}$

Table 3: Running time in seconds and No. of comparisons for SLINK.

$n \setminus d$	16	32	64
100	$5.00\ 10^{-1}$ / $6.22\ 10^{4}$	$8.20\ 10^{-1}$ / $9.29\ 10^{4}$	$1.16\ 10^{0}$ / $1.18\ 10^{5}$
200	$2.25\ 10^{0}$ / $2.58\ 10^{5}$	$3.40\ 10^{0}$ / $3.90\ 10^{5}$	$5.50\ 10^{0}$ / $5.62\ 10^{5}$
500	$1.42\ 10^{1}$ / $1.68\ 10^{6}$	$2.66\ 10^{1}$ / $2.93\ 10^{6}$	$3.79\ 10^{1}$ / $3.80\ 10^{6}$
1000	$5.80\ 10^{1}$ / $6.79\ 10^{6}$	$1.11\ 10^{2}$ / $1.20\ 10^{7}$	$1.76\ 10^{2}$ / $1.73\ 10^{7}$
2000	$2.32\ 10^{2}$ / $2.74\ 10^{7}$	$4.48\ 10^{2}$ / $4.87\ 10^{7}$	$7.57\ 10^{2}$ / $7.44\ 10^{7}$
5000	$1.54\ 10^{3}$ / $1.75\ 10^{8}$	$3.18\ 10^{3}$ / $3.30\ 10^{8}$	$5.08\ 10^{3}$ / $4.79\ 10^{8}$
10000	$6.41\ 10^{3}$ / $7.08\ 10^{8}$	$1.34\ 10^{4}$ / $1.33\ 10^{9}$	$2.33\ 10^{4}$ / $2.09\ 10^{9}$
$n \setminus d$	128	256	512
100	$1.10\ 10^{0}$ / $1.58\ 10^{5}$	$2.25\ 10^{0}$ / $2.11\ 10^{5}$	$4.89\ 10^{0}$ / $2.81\ 10^{5}$
200	$5.33\ 10^{0}$ / $7.39\ 10^{5}$	$1.12\ 10^{1}$ / $1.05\ 10^{6}$	$2.39\ 10^{1}$ / $1.37\ 10^{6}$
500	$3.98\ 10^{1}$ / $5.47\ 10^{6}$	$8.03\ 10^{1}$ / $7.48\ 10^{6}$	$1.85\ 10^{2}$ / $1.06\ 10^{7}$
1000	$1.70\ 10^{2}$ / $2.31\ 10^{7}$	$3.74\ 10^{2}$ / $3.45\ 10^{7}$	$8.13\ 10^{2}$ / $4.62\ 10^{7}$
2000	$7.54\ 10^{2}$ / $1.01\ 10^{8}$	$1.58\ 10^{3}$ / $1.45\ 10^{8}$	$3.57\ 10^{3}$ / $2.02\ 10^{8}$
5000	$5.04\ 10^{3}$ / $6.71\ 10^{8}$	$1.06\ 10^{4}$ / $9.69\ 10^{8}$	$2.48\ 10^{4}$ / $1.39\ 10^{9}$
10000	$2.21\ 10^{4}$ / $2.85\ 10^{9}$	$4.49\ 10^{4}$ / $4.07\ 10^{9}$	$1.04\ 10^{5}$ / $5.84\ 10^{9}$

Table 4: Running time in seconds and No. of comparisons for CLINK1.

$n \setminus d$	16	32	64
100	$5.00\ 10^{-1}$ / $6.22\ 10^{4}$	$1.05\ 10^{0}$ / $1.12\ 10^{5}$	$2.14\ 10^{0}$ / $2.05\ 10^{5}$
200	$1.98\ 10^{0}$ / $2.58\ 10^{5}$	$4.06\ 10^{0}$ / $4.79\ 10^{5}$	$8.29\ 10^{0}$ / $8.61\ 10^{5}$
500	$1.31\ 10^{1}$ / $1.68\ 10^{6}$	$2.67\ 10^{1}$ / $3.05\ 10^{6}$	$5.56\ 10^{1}$ / $5.54\ 10^{6}$
1000	$5.73\ 10^{1}$ / $6.79\ 10^{6}$	$1.15\ 10^{2}$ / $1.25\ 10^{7}$	$2.39\ 10^{2}$ / $2.30\ 10^{7}$
2000	$2.28\ 10^{2}$ / $2.74\ 10^{7}$	$4.64\ 10^{2}$ / $5.07\ 10^{7}$	$9.78\ 10^{2}$ / $9.44\ 10^{7}$
5000	$1.65\ 10^{3}$ / $1.75\ 10^{8}$	$3.44\ 10^{3}$ / $3.30\ 10^{8}$	$7.02\ 10^{3}$ / $6.04\ 10^{8}$
10000	$6.23\ 10^{3}$ / $7.08\ 10^{8}$	$1.29\ 10^{4}$ / $1.33\ 10^{9}$	$2.69\ 10^{4}$ / $2.44\ 10^{9}$
$n \setminus d$	128	256	512
100	$2.86\ 10^{0}$ / $3.84\ 10^{5}$	$8.08\ 10^{0}$ / $7.44\ 10^{5}$	$2.53\ 10^{1}$ / $1.42\ 10^{6}$
200	$1.85\ 10^{1}$ / $1.58\ 10^{6}$	$3.38\ 10^{1}$ / $3.03\ 10^{6}$	$1.04\ 10^{2}$ / $5.78\ 10^{6}$
500	$1.25\ 10^{2}$ / $1.04\ 10^{7}$	$2.17\ 10^{2}$ / $1.93\ 10^{7}$	$6.72\ 10^{2}$ / $3.70\ 10^{7}$
1000	$3.23\ 10^{2}$ / $4.21\ 10^{7}$	$8.88\ 10^{2}$ / $7.91\ 10^{7}$	$2.71\ 10^{3}$ / $1.49\ 10^{8}$
2000	$2.08\ 10^{3}$ / $1.70\ 10^{8}$	$3.65\ 10^{3}$ / $3.24\ 10^{8}$	$1.09\ 10^{4}$ / $6.03\ 10^{8}$
5000	$1.50\ 10^{4}$ / $1.10\ 10^{9}$	$2.33\ 10^{4}$ / $2.04\ 10^{9}$	$7.03\ 10^{4}$ / $3.83\ 10^{9}$
10000	$3.49\ 10^{4}$ / $4.43\ 10^{9}$	$9.47\ 10^{4}$ / $8.25\ 10^{9}$	$2.88\ 10^{5}$ / $1.55\ 10^{10}$

Table 5: Running time in seconds and No. of comparisons for CLINK3.

$n \setminus d$	16	32	64
100	$5.50\ 10^{-1}$ / $5.60\ 10^{4}$	$8.20\ 10^{-1}$ / $8.66\ 10^{4}$	$1.10\ 10^{0}$ / $1.04\ 10^{5}$
200	$2.30\ 10^{0}$ / $2.57\ 10^{5}$	$3.62\ 10^{0}$ / $3.72\ 10^{5}$	$5.27\ 10^{0}$ / $4.90\ 10^{5}$
500	$1.60\ 10^{1}$ / $1.77\ 10^{6}$	$2.85\ 10^{1}$ / $2.84\ 10^{6}$	$3.67\ 10^{1}$ / $3.41\ 10^{6}$
1000	$6.65\ 10^{1}$ / $7.32\ 10^{6}$	$1.21\ 10^{2}$ / $1.21\ 10^{7}$	$1.81\ 10^{2}$ / $1.66\ 10^{7}$
2000	$2.77\ 10^{2}$ / $3.04\ 10^{7}$	$5.04\ 10^{2}$ / $5.05\ 10^{7}$	$7.57\ 10^{2}$ / $6.91\ 10^{7}$
5000	$2.29\ 10^{3}$ / $2.05\ 10^{8}$	$4.19\ 10^{3}$ / $3.45\ 10^{8}$	$6.14\ 10^{3}$ / $4.69\ 10^{8}$
10000	$9.84\ 10^{3}$ / $8.57\ 10^{8}$	$1.78\ 10^{4}$ / $1.42\ 10^{9}$	$2.78\ 10^{4}$ / $2.05\ 10^{9}$
$n \setminus d$	128	256	512
100	$1.76\ 10^{0}$ / $1.42\ 10^{5}$	$2.25\ 10^{0}$ / $1.95\ 10^{5}$	$3.85\ 10^{0}$ / $2.08\ 10^{5}$
200	$8.02\ 10^{0}$ / $6.44\ 10^{5}$	$9.99\ 10^{0}$ / $8.66\ 10^{5}$	$1.94\ 10^{1}$ / $1.06\ 10^{6}$
500	$5.98\ 10^{1}$ / $4.80\ 10^{6}$	$7.42\ 10^{1}$ / $6.47\ 10^{6}$	$1.52\ 10^{2}$ / $8.34\ 10^{6}$
1000	$2.57\ 10^{2}$ / $2.05\ 10^{7}$	$3.46\ 10^{2}$ / $2.98\ 10^{7}$	$6.92\ 10^{2}$ / $3.78\ 10^{7}$
2000	$1.19\ 10^{3}$ / $9.44\ 10^{7}$	$1.41\ 10^{3}$ / $1.22\ 10^{8}$	$3.15\ 10^{3}$ / $1.71\ 10^{8}$
5000	$9.09\ 10^{3}$ / $6.19\ 10^{8}$	$1.00\ 10^{4}$ / $8.60\ 10^{8}$	$2.23\ 10^{4}$ / $1.20\ 10^{9}$
10000	$4.18\ 10^{4}$ / $2.73\ 10^{9}$	$4.36\ 10^{4}$ / $3.66\ 10^{9}$	$9.64\ 10^{4}$ / $5.14\ 10^{9}$

Table 6: Running time in seconds and No. of comparisons for MINRADI.

$n \setminus d$	16	32	64
100	$1.04\ 10^{0}$ / $1.15\ 10^{5}$ / 17	$1.10\ 10^{0}$ / $1.19\ 10^{5}$ / 18	$1.21\ 10^{0}$ / $1.11\ 10^{5}$ / 16
200	$5.49\ 10^{0}$ / $5.74\ 10^{5}$ / 22	$5.55\ 10^{0}$ / $5.89\ 10^{5}$ / 23	$6.26\ 10^{0}$ / $6.19\ 10^{5}$ / 24
500	$4.28\ 10^{1}$ / $4.99\ 10^{6}$ / 33	$4.30\ 10^{1}$ / $4.65\ 10^{6}$ / 32	$4.70\ 10^{1}$ / $4.66\ 10^{6}$ / 29
1000	$1.78\ 10^{2}$ / $1.99\ 10^{7}$ / 32	$2.08\ 10^{2}$ / $2.20\ 10^{7}$ / 37	$2.28\ 10^{2}$ / $2.21\ 10^{7}$ / 37
2000	$9.98\ 10^{2}$ / $9.77\ 10^{7}$ / 40	$1.07\ 10^{3}$ / $1.01\ 10^{8}$ / 42	$1.17\ 10^{3}$ / $1.01\ 10^{8}$ / 41
5000	$7.60\ 10^{3}$ / $7.16\ 10^{8}$ / 51	$7.94\ 10^{3}$ / $7.15\ 10^{8}$ / 49	$9.32\ 10^{3}$ / $7.74\ 10^{8}$ / 52
10000	$3.16\ 10^{4}$ / $3.26\ 10^{9}$ / 62	$3.78\ 10^{4}$ / $3.33\ 10^{9}$ / 56	$4.37\ 10^{4}$ / $3.55\ 10^{9}$ / 62
$n \setminus d$	128	256	512
100	$1.54\ 10^{0}$ / $1.26\ 10^{5}$ / 20	$1.26\ 10^{0}$ / $1.12\ 10^{5}$ / 17	$2.20\ 10^{0}$ / $1.21\ 10^{5}$ / 21
200	$4.72\ 10^{0}$ / $6.13\ 10^{5}$ / 23	$6.98\ 10^{0}$ / $6.16\ 10^{5}$ / 24	$1.11\ 10^{1}$ / $6.08\ 10^{5}$ / 24
500	$3.56\ 10^{1}$ / $4.70\ 10^{6}$ / 29	$5.82\ 10^{1}$ / $5.20\ 10^{6}$ / 32	$9.05\ 10^{1}$ / $4.92\ 10^{6}$ / 32
1000	$2.75\ 10^{2}$ / $2.27\ 10^{7}$ / 35	$2.83\ 10^{2}$ / $2.52\ 10^{7}$ / 41	$4.45\ 10^{2}$ / $2.41\ 10^{7}$ / 42
2000	$8.27\ 10^{2}$ / $1.08\ 10^{8}$ / 44	$1.25\ 10^{3}$ / $1.11\ 10^{8}$ / 47	$2.13\ 10^{3}$ / $1.15\ 10^{8}$ / 51
5000	$6.17\ 10^{3}$ / $7.83\ 10^{8}$ / 53	$9.48\ 10^{3}$ / $8.23\ 10^{8}$ / 54	$1.62\ 10^{4}$ / $8.66\ 10^{8}$ / 62
10000	$3.04\ 10^{4}$ / $3.74\ 10^{9}$ / 66	$4.53\ 10^{4}$ / $3.85\ 10^{9}$ / 67	$7.77\ 10^{4}$ / $4.10\ 10^{9}$ / 79

Table 7: Running time in seconds, No. of comparisons and No. of iterations for CLINK2.

$n \setminus d$	16	32	64
100	$7.70\ 10^{-1}$ / $2.17\ 10^{4}$ / 17	$1.37\ 10^{0}$ / $2.13\ 10^{4}$ / 18	$2.91\ 10^{0}$ / $2.26\ 10^{4}$ / 16
200	$3.07\ 10^{0}$ / $8.96\ 10^{4}$ / 20	$5.71\ 10^{0}$ / $8.87\ 10^{4}$ / 19	$1.10\ 10^{1}$ / $8.76\ 10^{4}$ / 20
500	$1.93\ 10^{1}$ / $5.61\ 10^{5}$ / 27	$3.67\ 10^{1}$ / $5.65\ 10^{5}$ / 23	$7.12\ 10^{1}$ / $5.65\ 10^{5}$ / 26
1000	$7.86\ 10^{1}$ / $2.23\ 10^{6}$ / 29	$1.49\ 10^{2}$ / $2.25\ 10^{6}$ / 33	$2.92\ 10^{2}$ / $2.27\ 10^{6}$ / 28
2000	$3.19\ 10^{2}$ / $9.21\ 10^{6}$ / 35	$6.06\ 10^{2}$ / $9.29\ 10^{6}$ / 37	$1.19\ 10^{3}$ / $9.47\ 10^{6}$ / 36
5000	$2.05\ 10^{3}$ / $5.89\ 10^{7}$ / 39	$3.86\ 10^{3}$ / $5.88\ 10^{7}$ / 45	$7.73\ 10^{3}$ / $6.09\ 10^{7}$ / 42
10000	$8.85\ 10^{3}$ / $2.52\ 10^{8}$ / 47	$1.57\ 10^{4}$ / $2.40\ 10^{8}$ / 47	$3.18\ 10^{4}$ / $2.50\ 10^{8}$ / 52
$n \setminus d$	128	256	512
100	$5.60\ 10^{0}$ / $2.22\ 10^{4}$ / 16	$1.07\ 10^{1}$ / $2.15\ 10^{4}$ / 16	$2.10\ 10^{1}$ / $2.12\ 10^{4}$ / 17
200	$2.20\ 10^{1}$ / $8.88\ 10^{4}$ / 20	$4.51\ 10^{1}$ / $9.17\ 10^{4}$ / 20	$8.51\ 10^{1}$ / $8.69\ 10^{4}$ / 16
500	$1.41\ 10^{2}$ / $5.69\ 10^{5}$ / 25	$2.89\ 10^{2}$ / $5.86\ 10^{5}$ / 26	$5.78\ 10^{2}$ / $5.88\ 10^{5}$ / 24
1000	$5.91\ 10^{2}$ / $2.35\ 10^{6}$ / 29	$1.22\ 10^{3}$ / $2.46\ 10^{6}$ / 32	$2.48\ 10^{3}$ / $2.51\ 10^{6}$ / 36
2000	$2.39\ 10^{3}$ / $9.61\ 10^{6}$ / 38	$4.92\ 10^{3}$ / $9.96\ 10^{6}$ / 35	$9.88\ 10^{3}$ / $1.00\ 10^{7}$ / 40
5000	$1.57\ 10^{4}$ / $6.31\ 10^{7}$ / 47	$3.20\ 10^{4}$ / $6.49\ 10^{7}$ / 46	$6.51\ 10^{4}$ / $6.61\ 10^{7}$ / 47
10000	$6.51\ 10^{4}$ / $2.61\ 10^{8}$ / 55	$1.32\ 10^{5}$ / $2.67\ 10^{8}$ / 54	$2.73\ 10^{5}$ / $2.75\ 10^{8}$ / 57

Table 8: Running time in seconds, No. of comparisons and No. of iterations for ALINK.

$n \setminus d$	16	32	64
100	$6.00\ 10^{-1}$ / $3.19\ 10^{4}$ / 28	$1.15\ 10^{0}$ / $3.06\ 10^{4}$ / 31	$2.31\ 10^{0}$ / $3.14\ 10^{4}$ / 31
200	$2.81\ 10^{0}$ / $1.39\ 10^{5}$ / 37	$5.06\ 10^{0}$ / $1.33\ 10^{5}$ / 39	$9.66\ 10^{0}$ / $1.31\ 10^{5}$ / 37
500	$1.77\ 10^{1}$ / $8.67\ 10^{5}$ / 58	$3.47\ 10^{1}$ / $9.07\ 10^{5}$ / 57	$6.52\ 10^{1}$ / $8.85\ 10^{5}$ / 59
1000	$8.07\ 10^{1}$ / $3.81\ 10^{6}$ / 84	$1.48\ 10^{2}$ / $3.79\ 10^{6}$ / 74	$2.77\ 10^{2}$ / $3.68\ 10^{6}$ / 80
2000	$3.31\ 10^{2}$ / $1.59\ 10^{7}$ / 92	$6.13\ 10^{2}$ / $1.59\ 10^{7}$ / 97	$1.18\ 10^{3}$ / $1.60\ 10^{7}$ / 99
5000	$2.25\ 10^{3}$ / $1.06\ 10^{8}$ / 129	$4.34\ 10^{3}$ / $1.11\ 10^{8}$ / 132	$8.22\ 10^{3}$ / $1.10\ 10^{8}$ / 126
10000	$9.20\ 10^{3}$ / $4.37\ 10^{8}$ / 159	$1.81\ 10^{4}$ / $4.65\ 10^{8}$ / 162	$3.35\ 10^{4}$ / $4.47\ 10^{8}$ / 155
$n \setminus d$	128	256	512
100	$4.67\ 10^{0}$ / $3.21\ 10^{4}$ / 32	$8.63\ 10^{0}$ / $3.01\ 10^{4}$ / 29	$1.67\ 10^{1}$ / $2.91\ 10^{4}$ / 28
200	$1.86\ 10^{1}$ / $1.30\ 10^{5}$ / 39	$3.85\ 10^{1}$ / $1.35\ 10^{5}$ / 40	$7.41\ 10^{1}$ / $1.30\ 10^{5}$ / 37
500	$1.32\ 10^{2}$ / $9.20\ 10^{5}$ / 61	$2.54\ 10^{2}$ / $8.86\ 10^{5}$ / 62	$5.29\ 10^{2}$ / $9.27\ 10^{5}$ / 60
1000	$5.74\ 10^{2}$ / $3.89\ 10^{6}$ / 77	$1.13\ 10^{3}$ / $3.90\ 10^{6}$ / 81	$2.21\ 10^{3}$ / $3.82\ 10^{6}$ / 81
2000	$2.30\ 10^{3}$ / $1.58\ 10^{7}$ / 101	$4.60\ 10^{3}$ / $1.60\ 10^{7}$ / 103	$9.30\ 10^{3}$ / $1.62\ 10^{7}$ / 101
5000	$1.57\ 10^{4}$ / $1.07\ 10^{8}$ / 138	$3.13\ 10^{4}$ / $1.08\ 10^{8}$ / 136	$6.08\ 10^{4}$ / $1.06\ 10^{8}$ / 130
10000	$6.66\ 10^{4}$ / $4.53\ 10^{8}$ / 158	$1.32\ 10^{5}$ / $4.54\ 10^{8}$ / 170	$2.58\ 10^{5}$ / $4.47\ 10^{8}$ / 169

Table 9: Running time in seconds, No. of comparisons and No. of iterations for ASUMLNK.

$n \setminus d$	16	32	64
100	$4.90\ 10^{-1}$ / $2.33\ 10^{4}$ / 17	$8.20\ 10^{-1}$ / $2.22\ 10^{4}$ / 16	$1.65\ 10^{0}$ / $2.28\ 10^{4}$ / 20
200	$1.97\ 10^{0}$ / $9.25\ 10^{4}$ / 23	$3.57\ 10^{0}$ / $9.21\ 10^{4}$ / 23	$7.03\ 10^{0}$ / $9.40\ 10^{4}$ / 27
500	$1.29\ 10^{1}$ / $5.93\ 10^{5}$ / 32	$2.33\ 10^{1}$ / $5.91\ 10^{5}$ / 33	$4.39\ 10^{1}$ / $5.85\ 10^{5}$ / 34
1000	$5.30\ 10^{1}$ / $2.42\ 10^{6}$ / 42	$9.32\ 10^{1}$ / $2.34\ 10^{6}$ / 41	$1.78\ 10^{2}$ / $2.36\ 10^{6}$ / 49
2000	$2.21\ 10^{2}$ / $1.00\ 10^{7}$ / 49	$3.87\ 10^{2}$ / $9.71\ 10^{6}$ / 51	$7.34\ 10^{2}$ / $9.73\ 10^{6}$ / 57
5000	$1.41\ 10^{3}$ / $6.34\ 10^{7}$ / 66	$2.47\ 10^{3}$ / $6.16\ 10^{7}$ / 62	$4.75\ 10^{3}$ / $6.26\ 10^{7}$ / 69
10000	$5.78\ 10^{3}$ / $2.58\ 10^{8}$ / 67	$1.00\ 10^{4}$ / $2.49\ 10^{8}$ / 73	$1.94\ 10^{4}$ / $2.55\ 10^{8}$ / 80
$n \setminus d$	128	256	512
100	$3.29\ 10^{0}$ / $2.26\ 10^{4}$ / 21	$6.32\ 10^{0}$ / $2.21\ 10^{4}$ / 21	$1.21\ 10^{1}$ / $2.14\ 10^{4}$ / 22
200	$1.27\ 10^{1}$ / $8.80\ 10^{4}$ / 24	$2.63\ 10^{1}$ / $9.21\ 10^{4}$ / 28	$4.88\ 10^{1}$ / $8.58\ 10^{4}$ / 25
500	$8.37\ 10^{1}$ / $5.73\ 10^{5}$ / 35	$1.72\ 10^{2}$ / $5.99\ 10^{5}$ / 37	$3.51\ 10^{2}$ / $6.12\ 10^{5}$ / 40
1000	$3.47\ 10^{2}$ / $2.37\ 10^{6}$ / 51	$7.21\ 10^{2}$ / $2.49\ 10^{6}$ / 49	$1.44\ 10^{3}$ / $2.51\ 10^{6}$ / 49
2000	$1.44\ 10^{3}$ / $9.86\ 10^{6}$ / 60	$2.93\ 10^{3}$ / $1.01\ 10^{7}$ / 66	$5.92\ 10^{3}$ / $1.03\ 10^{7}$ / 72
5000	$9.38\ 10^{3}$ / $6.37\ 10^{7}$ / 82	$1.92\ 10^{4}$ / $6.62\ 10^{7}$ / 91	$4.09\ 10^{4}$ / $6.77\ 10^{7}$ / 93
10000	$3.89\ 10^{4}$ / $2.62\ 10^{8}$ / 97	$7.98\ 10^{4}$ / $2.73\ 10^{8}$ / 111	$1.63\ 10^{5}$ / $2.82\ 10^{8}$ / 119

Table 10: Running time in seconds, No. of comparisons and No. of iterations for AMEANLNK.

$n \setminus d$	16	32	64
100	$1.10\ 10^{0}$ / $1.16\ 10^{5}$ / 17	$6.60\ 10^{-1}$ / $1.26\ 10^{5}$ / 18	$7.70\ 10^{-1}$ / $1.21\ 10^{5}$ / 17
200	$5.65\ 10^{0}$ / $6.39\ 10^{5}$ / 24	$3.08\ 10^{0}$ / $6.02\ 10^{5}$ / 21	$3.52\ 10^{0}$ / $5.83\ 10^{5}$ / 21
500	$4.44\ 10^{1}$ / $4.94\ 10^{6}$ / 32	$2.31\ 10^{1}$ / $4.61\ 10^{6}$ / 27	$2.85\ 10^{1}$ / $4.87\ 10^{6}$ / 29
1000	$1.99\ 10^{2}$ / $2.14\ 10^{7}$ / 32	$1.12\ 10^{2}$ / $2.18\ 10^{7}$ / 35	$1.38\ 10^{2}$ / $2.30\ 10^{7}$ / 35
2000	$1.14\ 10^{3}$ / $1.00\ 10^{8}$ / 40	$5.17\ 10^{2}$ / $1.03\ 10^{8}$ / 43	$6.50\ 10^{2}$ / $1.04\ 10^{8}$ / 37
5000	$9.17\ 10^{3}$ / $7.66\ 10^{8}$ / 64	$9.57\ 10^{3}$ / $7.72\ 10^{8}$ / 51	$4.97\ 10^{3}$ / $7.80\ 10^{8}$ / 50
10000	$2.81\ 10^{4}$ / $3.44\ 10^{9}$ / 67	$2.92\ 10^{4}$ / $3.40\ 10^{9}$ / 57	$2.20\ 10^{4}$ / $3.59\ 10^{9}$ / 54
$n \setminus d$	128	256	512
100	$8.80\ 10^{-1}$ / $1.14\ 10^{5}$ / 16	$1.54\ 10^{0}$ / $1.39\ 10^{5}$ / 19	$2.25\ 10^{0}$ / $1.25\ 10^{5}$ / 17
200	$4.50\ 10^{0}$ / $5.87\ 10^{5}$ / 22	$6.70\ 10^{0}$ / $6.02\ 10^{5}$ / 20	$1.04\ 10^{1}$ / $5.88\ 10^{5}$ / 20
500	$3.65\ 10^{1}$ / $4.83\ 10^{6}$ / 29	$5.35\ 10^{1}$ / $4.87\ 10^{6}$ / 28	$8.92\ 10^{1}$ / $5.02\ 10^{6}$ / 32
1000	$1.74\ 10^{2}$ / $2.26\ 10^{7}$ / 35	$2.56\ 10^{2}$ / $2.30\ 10^{7}$ / 35	$4.11\ 10^{2}$ / $2.29\ 10^{7}$ / 35
2000	$8.77\ 10^{2}$ / $1.09\ 10^{8}$ / 41	$1.20\ 10^{3}$ / $1.03\ 10^{8}$ / 41	$1.97\ 10^{3}$ / $1.05\ 10^{8}$ / 39
5000	$6.45\ 10^{3}$ / $7.99\ 10^{8}$ / 51	$9.46\ 10^{3}$ / $8.11\ 10^{8}$ / 51	$1.55\ 10^{4}$ / $8.27\ 10^{8}$ / 53
10000	$2.77\ 10^{4}$ / $3.50\ 10^{9}$ / 53	$4.25\ 10^{4}$ / $3.59\ 10^{9}$ / 57	$6.84\ 10^{4}$ / $3.64\ 10^{9}$ / 56

Table 11: Running time in seconds, No. of comparisons and No. of iterations for MINRAD2.

$n \setminus d$	16	32	64
100	$4.90\ 10^{-1}$ / $2.38\ 10^{4}$ / 19	$8.70\ 10^{-1}$ / $2.38\ 10^{4}$ / 23	$1.87\ 10^{0}$ / $2.57\ 10^{4}$ / 27
200	$1.98\ 10^{0}$ / $9.74\ 10^{4}$ / 25	$3.73\ 10^{0}$ / $9.86\ 10^{4}$ / 28	$7.25\ 10^{0}$ / $1.01\ 10^{5}$ / 29
500	$1.24\ 10^{1}$ / $6.00\ 10^{5}$ / 28	$2.38\ 10^{1}$ / $6.30\ 10^{5}$ / 37	$4.66\ 10^{1}$ / $6.48\ 10^{5}$ / 36
1000	$5.23\ 10^{1}$ / $2.50\ 10^{6}$ / 38	$9.52\ 10^{1}$ / $2.51\ 10^{6}$ / 42	$1.93\ 10^{2}$ / $2.67\ 10^{6}$ / 56
2000	$2.20\ 10^{2}$ / $1.04\ 10^{7}$ / 49	$3.91\ 10^{2}$ / $1.04\ 10^{7}$ / 49	$7.76\ 10^{2}$ / $1.09\ 10^{7}$ / 60
5000	$1.46\ 10^{3}$ / $6.95\ 10^{7}$ / 65	$2.55\ 10^{3}$ / $6.75\ 10^{7}$ / 66	$5.16\ 10^{3}$ / $7.24\ 10^{7}$ / 78
10000	$5.84\ 10^{3}$ / $2.77\ 10^{8}$ / 70	$1.03\ 10^{4}$ / $2.72\ 10^{8}$ / 74	$2.09\ 10^{4}$ / $2.92\ 10^{8}$ / 92
$n \setminus d$	128	256	512
100	$3.62\ 10^{0}$ / $2.60\ 10^{4}$ / 26	$6.81\ 10^{0}$ / $2.46\ 10^{4}$ / 26	$1.36\ 10^{1}$ / $2.49\ 10^{4}$ / 27
200	$1.37\ 10^{1}$ / $9.90\ 10^{4}$ / 32	$2.93\ 10^{1}$ / $1.06\ 10^{5}$ / 42	$5.49\ 10^{1}$ / $1.00\ 10^{5}$ / 32
500	$9.48\ 10^{1}$ / $6.77\ 10^{5}$ / 53	$1.99\ 10^{2}$ / $7.20\ 10^{5}$ / 53	$3.93\ 10^{2}$ / $7.16\ 10^{5}$ / 57
1000	$3.95\ 10^{2}$ / $2.81\ 10^{6}$ / 62	$8.05\ 10^{2}$ / $2.90\ 10^{6}$ / 72	$1.71\ 10^{3}$ / $3.12\ 10^{6}$ / 83
2000	$1.62\ 10^{3}$ / $1.17\ 10^{7}$ / 71	$3.35\ 10^{3}$ / $1.23\ 10^{7}$ / 89	$6.95\ 10^{3}$ / $1.28\ 10^{7}$ / 108
5000	$1.06\ 10^{4}$ / $7.67\ 10^{7}$ / 98	$2.18\ 10^{4}$ / $7.98\ 10^{7}$ / 117	$4.64\ 10^{4}$ / $8.59\ 10^{7}$ / 140
10000	$4.45\ 10^{4}$ / $3.20\ 10^{8}$ / 116	$9.35\ 10^{4}$ / $3.41\ 10^{8}$ / 148	$1.91\ 10^{5}$ / $3.52\ 10^{8}$ / 173

Table 12: Running time in seconds, No. of comparisons and No. of iterations for CTRDMEAN.

$n \setminus d$	16	32	64
100	$5.50\ 10^{-1}$ / $2.84\ 10^{4}$ / 28	$1.15\ 10^{0}$ / $3.06\ 10^{4}$ / 29	$2.30\ 10^{0}$ / $3.17\ 10^{4}$ / 29
200	$2.52\ 10^{0}$ / $1.25\ 10^{5}$ / 36	$4.89\ 10^{0}$ / $1.32\ 10^{5}$ / 39	$9.61\ 10^{0}$ / $1.35\ 10^{5}$ / 37
500	$1.49\ 10^{1}$ / $7.46\ 10^{5}$ / 53	$3.32\ 10^{1}$ / $8.97\ 10^{5}$ / 57	$6.45\ 10^{1}$ / $9.07\ 10^{5}$ / 55
1000	$6.47\ 10^{1}$ / $3.21\ 10^{6}$ / 73	$1.37\ 10^{2}$ / $3.70\ 10^{6}$ / 77	$2.55\ 10^{2}$ / $3.58\ 10^{6}$ / 68
2000	$2.59\ 10^{2}$ / $1.27\ 10^{7}$ / 87	$5.46\ 10^{2}$ / $1.48\ 10^{7}$ / 92	$1.15\ 10^{3}$ / $1.63\ 10^{7}$ / 95
5000	$1.73\ 10^{3}$ / $8.56\ 10^{7}$ / 126	$3.75\ 10^{3}$ / $1.01\ 10^{8}$ / 136	$7.54\ 10^{3}$ / $1.06\ 10^{8}$ / 122
10000	$6.97\ 10^{3}$ / $3.44\ 10^{8}$ / 138	$1.54\ 10^{4}$ / $4.17\ 10^{8}$ / 162	$3.10\ 10^{4}$ / $4.38\ 10^{8}$ / 153
$n \setminus d$	128	256	512
100	$4.39\ 10^{0}$ / $3.18\ 10^{4}$ / 29	$8.12\ 10^{0}$ / $2.94\ 10^{4}$ / 27	$1.62\ 10^{1}$ / $2.96\ 10^{4}$ / 28
200	$1.73\ 10^{1}$ / $1.24\ 10^{5}$ / 32	$3.59\ 10^{1}$ / $1.30\ 10^{5}$ / 35	$7.04\ 10^{1}$ / $1.28\ 10^{5}$ / 36
500	$1.23\ 10^{2}$ / $8.82\ 10^{5}$ / 56	$2.42\ 10^{2}$ / $8.78\ 10^{5}$ / 57	$4.98\ 10^{2}$ / $9.08\ 10^{5}$ / 58
1000	$5.38\ 10^{2}$ / $3.86\ 10^{6}$ / 73	$1.05\ 10^{3}$ / $3.82\ 10^{6}$ / 81	$2.06\ 10^{3}$ / $3.74\ 10^{6}$ / 68
2000	$2.12\ 10^{3}$ / $1.55\ 10^{7}$ / 91	$4.23\ 10^{3}$ / $1.56\ 10^{7}$ / 88	$8.68\ 10^{3}$ / $1.61\ 10^{7}$ / 97
5000	$1.53\ 10^{4}$ / $1.11\ 10^{8}$ / 124	$2.91\ 10^{4}$ / $1.07\ 10^{8}$ / 122	$5.66\ 10^{4}$ / $1.04\ 10^{8}$ / 127
10000	$6.12\ 10^{4}$ / $4.43\ 10^{8}$ / 150	$1.21\ 10^{5}$ / $4.44\ 10^{8}$ / 161	$2.38\ 10^{5}$ / $4.39\ 10^{8}$ / 151

Table 13: Running time in seconds, No. of comparisons and No. of iterations for CTRDSUM.

$n \setminus d$	16	32	64
100	$1.98\ 10^{0}$ / $1.86\ 10^{5}$ / 28	$1.98\ 10^{0}$ / $1.88\ 10^{5}$ / 29	$2.47\ 10^{0}$ / $2.12\ 10^{5}$ / 30
200	$1.14\ 10^{1}$ / $1.05\ 10^{6}$ / 40	$1.05\ 10^{1}$ / $1.00\ 10^{6}$ / 39	$1.10\ 10^{1}$ / $9.73\ 10^{5}$ / 35
500	$8.44\ 10^{1}$ / $8.51\ 10^{6}$ / 51	$9.71\ 10^{1}$ / $9.34\ 10^{6}$ / 57	$9.71\ 10^{1}$ / $8.55\ 10^{6}$ / 52
1000	$4.33\ 10^{2}$ / $4.34\ 10^{7}$ / 75	$4.78\ 10^{2}$ / $4.55\ 10^{7}$ / 75	$4.80\ 10^{2}$ / $4.21\ 10^{7}$ / 70
2000	$2.42\ 10^{3}$ / $2.15\ 10^{8}$ / 93	$2.41\ 10^{3}$ / $2.04\ 10^{8}$ / 89	$2.47\ 10^{3}$ / $1.94\ 10^{8}$ / 81
5000	$1.91\ 10^{4}$ / $1.66\ 10^{9}$ / 119	$2.01\ 10^{4}$ / $1.66\ 10^{9}$ / 119	$2.08\ 10^{4}$ / $1.59\ 10^{9}$ / 109
10000	$8.90\ 10^{4}$ / $7.62\ 10^{9}$ / 140	$9.85\ 10^{4}$ / $8.05\ 10^{9}$ / 142	$9.33\ 10^{4}$ / $7.01\ 10^{9}$ / 126
$n \setminus d$	128	256	512
100	$2.64\ 10^{0}$ / $1.91\ 10^{5}$ / 28	$2.47\ 10^{0}$ / $2.01\ 10^{5}$ / 30	$3.30\ 10^{0}$ / $1.73\ 10^{5}$ / 24
200	$1.24\ 10^{1}$ / $9.48\ 10^{5}$ / 36	$1.13\ 10^{1}$ / $9.26\ 10^{5}$ / 36	$1.79\ 10^{1}$ / $9.43\ 10^{5}$ / 36
500	$1.11\ 10^{2}$ / $8.52\ 10^{6}$ / 55	$1.01\ 10^{2}$ / $8.45\ 10^{6}$ / 51	$1.64\ 10^{2}$ / $8.77\ 10^{6}$ / 54
1000	$5.37\ 10^{2}$ / $4.08\ 10^{7}$ / 67	$4.88\ 10^{2}$ / $4.05\ 10^{7}$ / 67	$7.67\ 10^{2}$ / $4.08\ 10^{7}$ / 66
2000	$2.74\ 10^{3}$ / $1.90\ 10^{8}$ / 79	$2.35\ 10^{3}$ / $1.95\ 10^{8}$ / 78	$3.60\ 10^{3}$ / $1.91\ 10^{8}$ / 80
5000	$2.29\ 10^{4}$ / $1.55\ 10^{9}$ / 106	$1.87\ 10^{4}$ / $1.51\ 10^{9}$ / 107	$2.90\ 10^{4}$ / $1.51\ 10^{9}$ / 101
10000	$1.07\ 10^{5}$ / $7.13\ 10^{9}$ / 128	$8.62\ 10^{4}$ / $6.85\ 10^{9}$ / 123	$1.31\ 10^{5}$ / $6.76\ 10^{9}$ / 121

Table 14: Running time in seconds, No. of comparisons and No. of iterations for KRNLSUM.

$n \setminus d$	16	32	64
100	$2.20\ 10^{-1}$ / $2.60\ 10^{4}$	$3.30\ 10^{-1}$ / $3.57\ 10^{4}$	$3.80\ 10^{-1}$ / $3.93\ 10^{4}$
200	$7.70\ 10^{-1}$ / $9.21\ 10^{4}$	$1.27\ 10^{0}$ / $1.43\ 10^{5}$	$1.60\ 10^{0}$ / $1.69\ 10^{5}$
500	$4.17\ 10^{0}$ / $4.81\ 10^{5}$	$7.47\ 10^{0}$ / $8.27\ 10^{5}$	$9.45\ 10^{0}$ / $9.81\ 10^{5}$
1000	$1.50\ 10^{1}$ / $1.73\ 10^{6}$	$2.93\ 10^{1}$ / $3.24\ 10^{6}$	$4.93\ 10^{1}$ / $5.05\ 10^{6}$
2000	$5.07\ 10^{1}$ / $5.83\ 10^{6}$	$9.68\ 10^{1}$ / $1.06\ 10^{7}$	$1.79\ 10^{2}$ / $1.84\ 10^{7}$
5000	$3.69\ 10^{2}$ / $3.29\ 10^{7}$	$1.00\ 10^{3}$ / $8.55\ 10^{7}$	$1.56\ 10^{3}$ / $1.26\ 10^{8}$
10000	$1.35\ 10^{3}$ / $1.12\ 10^{8}$	$3.79\ 10^{3}$ / $3.02\ 10^{8}$	$6.08\ 10^{3}$ / $4.64\ 10^{8}$
$n \setminus d$	128	256	512
100	$3.90\ 10^{-1}$ / $4.75\ 10^{4}$	$7.70\ 10^{-1}$ / $6.81\ 10^{4}$	$1.87\ 10^{0}$ / $1.01\ 10^{5}$
200	$1.48\ 10^{0}$ / $1.88\ 10^{5}$	$3.57\ 10^{0}$ / $3.21\ 10^{5}$	$6.98\ 10^{0}$ / $3.90\ 10^{5}$
500	$1.22\ 10^{1}$ / $1.61\ 10^{6}$	$2.08\ 10^{1}$ / $1.89\ 10^{6}$	$4.49\ 10^{1}$ / $2.52\ 10^{6}$
1000	$4.29\ 10^{1}$ / $5.71\ 10^{6}$	$8.83\ 10^{1}$ / $8.07\ 10^{6}$	$1.96\ 10^{2}$ / $1.10\ 10^{7}$
2000	$1.66\ 10^{2}$ / $2.22\ 10^{7}$	$4.04\ 10^{2}$ / $3.71\ 10^{7}$	$7.36\ 10^{2}$ / $4.14\ 10^{7}$
5000	$1.27\ 10^{3}$ / $1.66\ 10^{8}$	$2.34\ 10^{3}$ / $2.11\ 10^{8}$	$5.02\ 10^{3}$ / $2.77\ 10^{8}$
10000	$4.85\ 10^{3}$ / $6.09\ 10^{8}$	$9.82\ 10^{3}$ / $8.63\ 10^{8}$	$2.08\ 10^{4}$ / $1.14\ 10^{9}$

Table 15: Running time in seconds and No. of comparisons for DIVIDIAM.

$n \setminus d$	16	32	64
100	$9.40\ 10^{-1}$ / $1.25\ 10^{5}$	$1.37\ 10^{0}$ / $1.73\ 10^{5}$	$1.93\ 10^{0}$ / $2.27\ 10^{5}$
200	$3.96\ 10^{0}$ / $5.42\ 10^{5}$	$6.10\ 10^{0}$ / $7.60\ 10^{5}$	$9.17\ 10^{0}$ / $1.06\ 10^{6}$
500	$2.87\ 10^{1}$ / $3.49\ 10^{6}$	$4.82\ 10^{1}$ / $5.61\ 10^{6}$	$6.60\ 10^{1}$ / $7.33\ 10^{6}$
1000	$1.24\ 10^{2}$ / $1.45\ 10^{7}$	$2.07\ 10^{2}$ / $2.33\ 10^{7}$	$3.12\ 10^{2}$ / $3.37\ 10^{7}$
2000	$5.12\ 10^{2}$ / $5.89\ 10^{7}$	$8.72\ 10^{2}$ / $9.58\ 10^{7}$	$1.31\ 10^{3}$ / $1.41\ 10^{8}$
5000	$4.17\ 10^{3}$ / $3.77\ 10^{8}$	$7.44\ 10^{3}$ / $6.62\ 10^{8}$	$1.05\ 10^{4}$ / $9.40\ 10^{8}$
10000	$1.67\ 10^{4}$ / $1.52\ 10^{9}$	$3.06\ 10^{4}$ / $2.68\ 10^{9}$	$4.56\ 10^{4}$ / $3.99\ 10^{9}$
$n \setminus d$	128	256	512
100	$2.42\ 10^{0}$ / $3.16\ 10^{5}$	$4.89\ 10^{0}$ / $4.50\ 10^{5}$	$1.09\ 10^{1}$ / $6.17\ 10^{5}$
200	$1.06\ 10^{1}$ / $1.42\ 10^{6}$	$2.27\ 10^{1}$ / $2.07\ 10^{6}$	$4.89\ 10^{1}$ / $2.74\ 10^{6}$
500	$7.92\ 10^{1}$ / $1.05\ 10^{7}$	$1.58\ 10^{2}$ / $1.44\ 10^{7}$	$3.61\ 10^{2}$ / $2.02\ 10^{7}$
1000	$3.35\ 10^{2}$ / $4.44\ 10^{7}$	$7.07\ 10^{2}$ / $6.42\ 10^{7}$	$1.56\ 10^{3}$ / $8.74\ 10^{7}$
2000	$1.47\ 10^{3}$ / $1.94\ 10^{8}$	$3.05\ 10^{3}$ / $2.76\ 10^{8}$	$6.76\ 10^{3}$ / $3.77\ 10^{8}$
5000	$1.01\ 10^{4}$ / $1.31\ 10^{9}$	$2.05\ 10^{4}$ / $1.83\ 10^{9}$	$4.74\ 10^{4}$ / $2.60\ 10^{9}$
10000	$4.47\ 10^{4}$ / $5.54\ 10^{9}$	$8.94\ 10^{4}$ / $7.75\ 10^{9}$	$2.03\ 10^{5}$ / $1.10\ 10^{10}$

Table 16: Running time in seconds and No. of comparisons for DIVIDIA2.

B Figures for Uniformly Distributed Random Vertex Sets

Appendix B shows some of the experimental data as curves. The first group (Figures 11 to 15) shows the number of vertex comparisons over the number n of elements. The second group (Figures 16 to 18) gives the running time needed to compute the entire cluster hierarchy in seconds. In the last part (Figures 19 and 20) the number of iterations of the WHILE loop for algorithms based on GHC2 is shown.

For all figures there are six curves drawn for dimension $d = 16, 32, 64, 128, 256, 512$. For a discussion of these curves refer to Section 4.

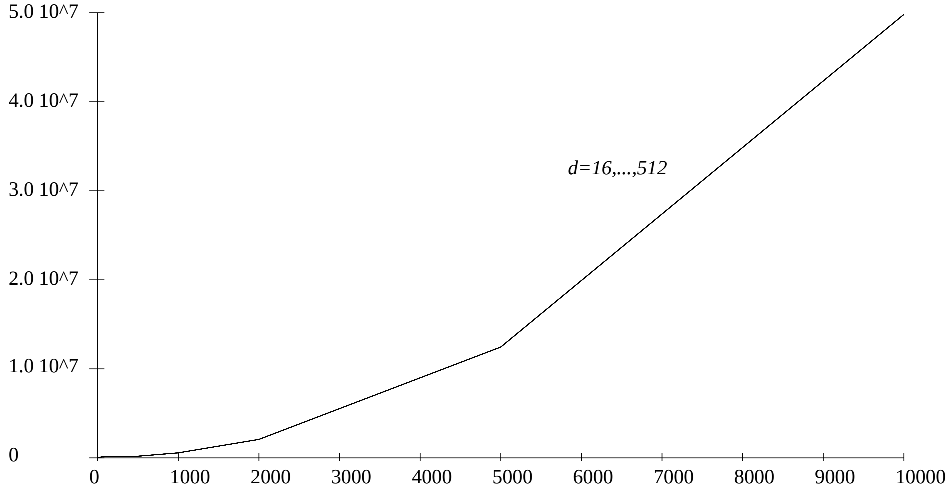


Figure 11: Number of vertex comparisons as function of the number of elements for SLINK.

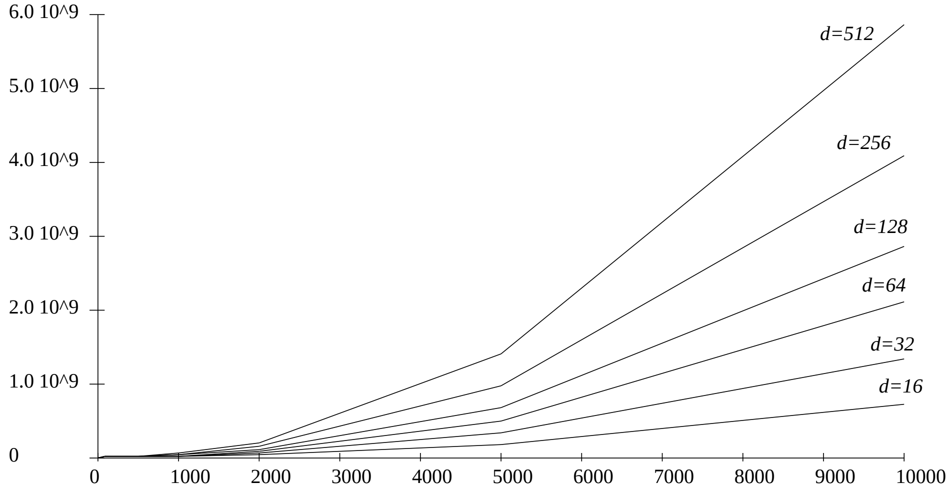


Figure 12: Number of vertex comparisons as function of the number of elements for CLINK1.

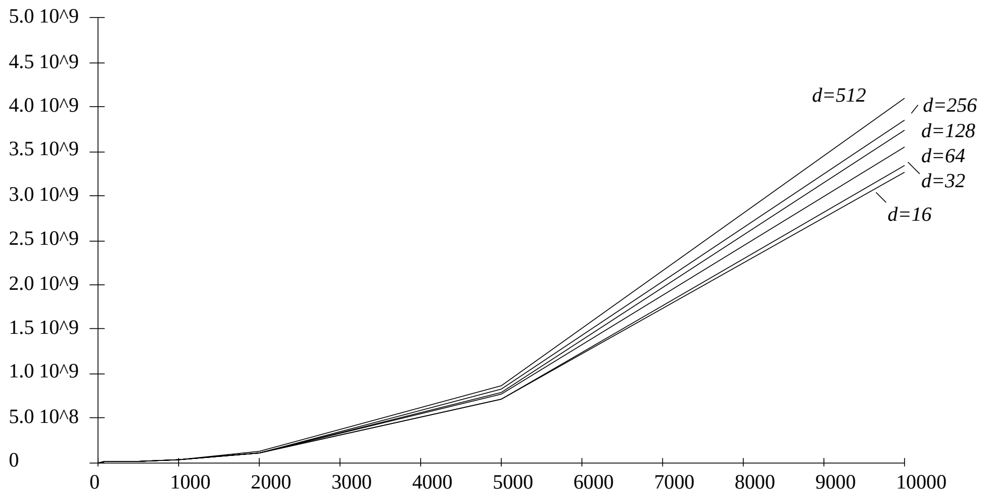


Figure 13: Number of vertex comparisons as function of the number of elements for CLINK2.

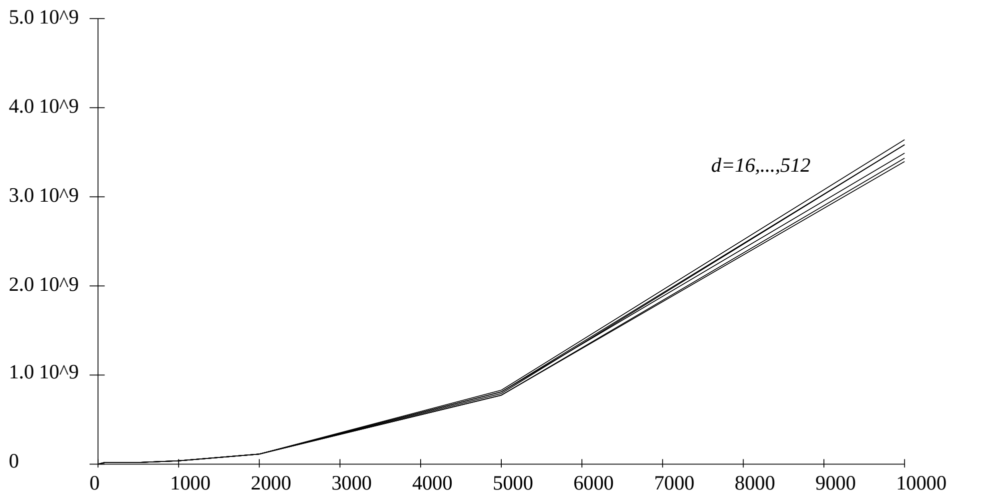


Figure 14: Number of vertex comparisons as function of the number of elements for MINRAD2.

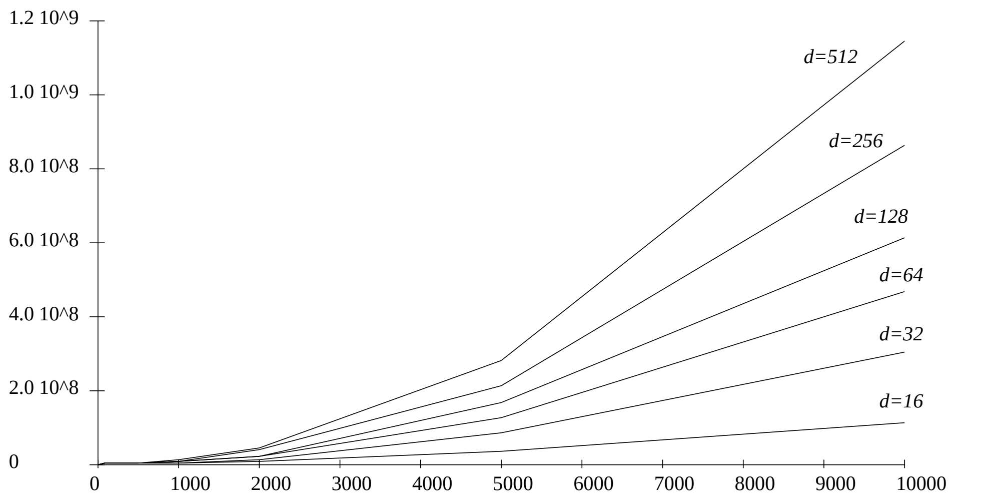


Figure 15: Number of vertex comparisons as function of the number of elements for DIVIDIAM.

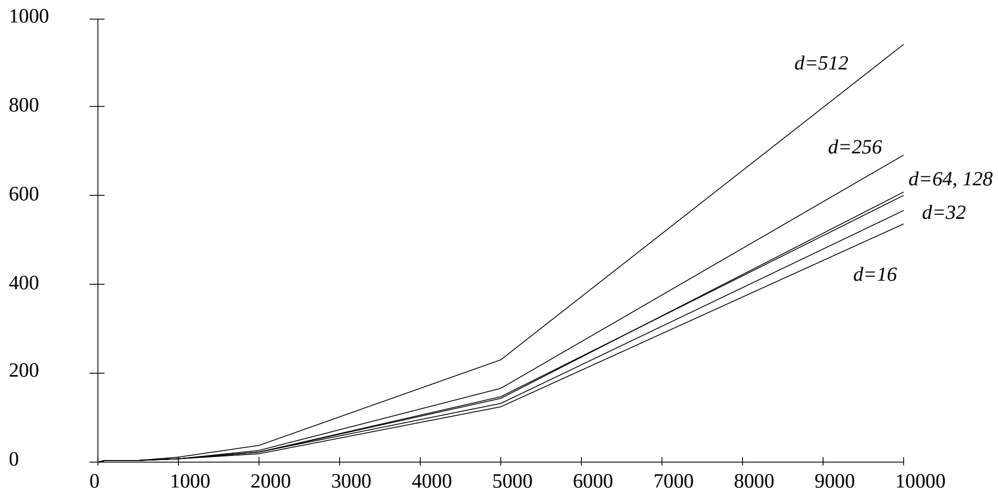


Figure 16: Running time in seconds as function of the number of elements for SLINK.

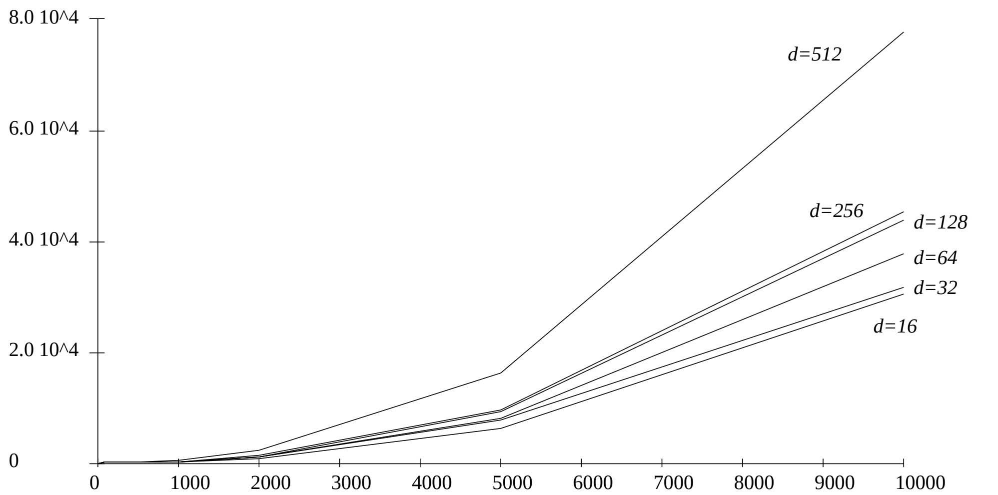


Figure 17: Running time in seconds as function of the number of elements for CLINK2.

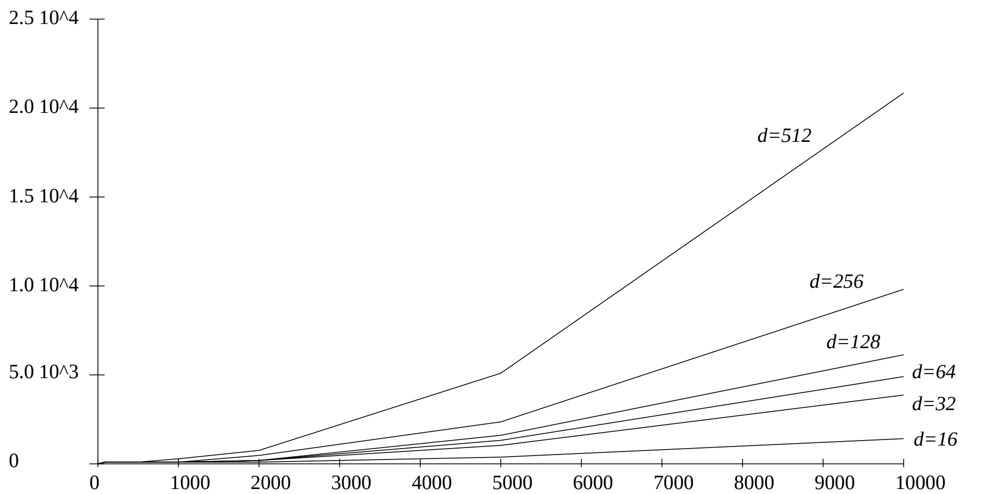


Figure 18: Running time in seconds as function of the number of elements for DIVIDIAM.

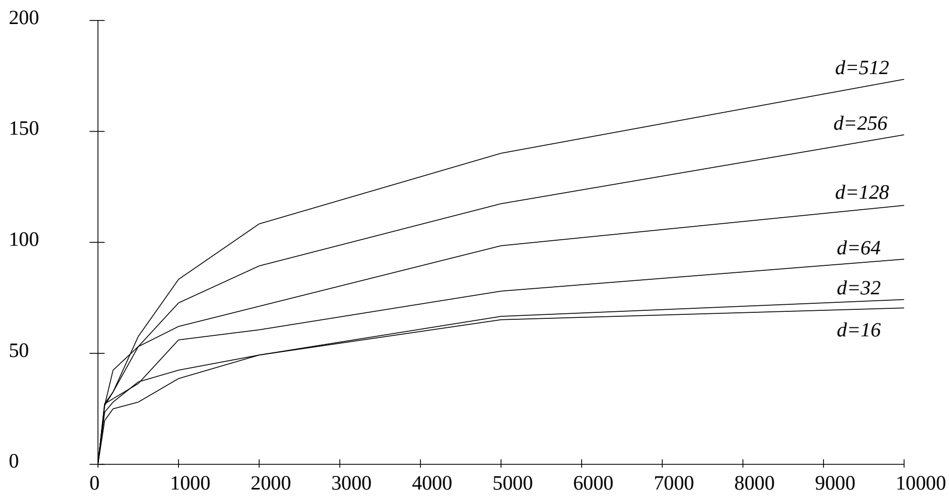


Figure 19: Number of iterations of the WHILE loop as function of the number of elements for CTRDMEAN.

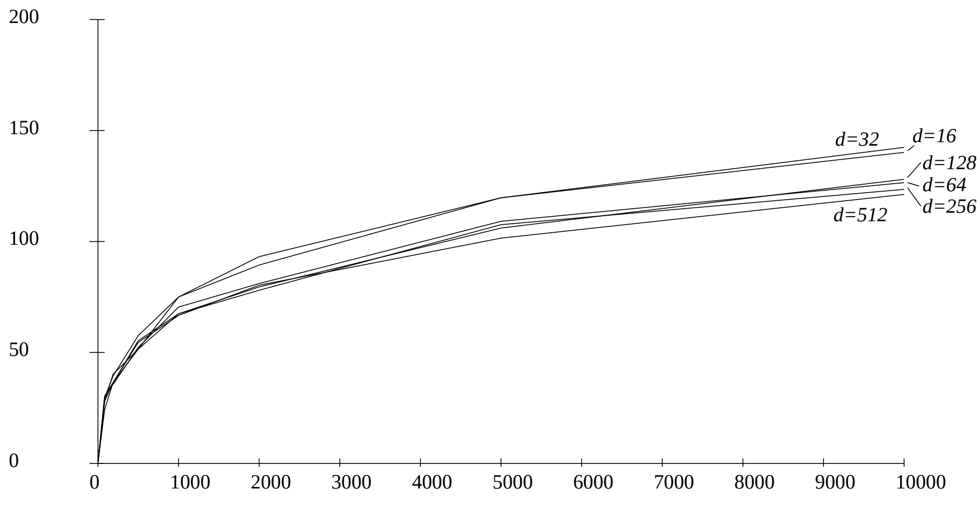


Figure 20: Number of iterations of the WHILE loop as function of the number of elements for KRNLSUM.

C Tables for Special Vertex Sets with 10 Cluster centers

Appendix C summarizes the experimental results for 39 special point sets described in Section 4. A discussion of the results can also be found there.

Tables 17 to 19 show the running time of the algorithms in seconds, and the number of comparisons, respectively, in dependency of the dimension d and the number of elements n. Tables 20 and 21 show these results for some methods based on Algorithm GHC2 and give in addition the number of iterations of the WHILE loop (cf. Figure 4). Finally Table 22 gives the results for the divisive method DIVIDIAM.

$n \setminus d$	16	32	64
100	$6.00\ 10^{-2}$ / $4.95\ 10^{3}$	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$
200	$1.60\ 10^{-1}$ / $1.99\ 10^{4}$	$1.80\ 10^{-1}$ / $1.99\ 10^{4}$	$1.20\ 10^{-1}$ / $1.99\ 10^{4}$
500	$1.04\ 10^{0}$ / $1.24\ 10^{5}$	$9.20\ 10^{-1}$ / $1.24\ 10^{5}$	$6.60\ 10^{-1}$ / $1.24\ 10^{5}$
1000	$4.45\ 10^{0}$ / $4.99\ 10^{5}$	$3.60\ 10^{0}$ / $4.99\ 10^{5}$	$2.80\ 10^{0}$ / $4.99\ 10^{5}$
2000	-	$1.42\ 10^{1}$ / $1.99\ 10^{6}$	$1.12\ 10^{1}$ / $1.99\ 10^{6}$
5000	-	$1.04\ 10^{2}$ / $1.25\ 10^{7}$	$7.44\ 10^{1}$ / $1.25\ 10^{7}$
10000	-	$4.45\ 10^{2}$ / $5.00\ 10^{7}$	$3.17\ 10^{2}$ / $5.00\ 10^{7}$
$n \setminus d$	128	256	512
100	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$5.00\ 10^{-2}$ / $4.95\ 10^{3}$	$1.20\ 10^{-1}$ / $4.95\ 10^{3}$
200	$1.80\ 10^{-1}$ / $1.99\ 10^{4}$	$2.40\ 10^{-1}$ / $1.99\ 10^{4}$	$3.60\ 10^{-1}$ / $1.99\ 10^{4}$
500	$9.20\ 10^{-1}$ / $1.24\ 10^{5}$	$1.34\ 10^{0}$ / $1.24\ 10^{5}$	$2.20\ 10^{0}$ / $1.24\ 10^{5}$
1000	$3.66\ 10^{0}$ / $4.99\ 10^{5}$	$5.36\ 10^{0}$ / $4.99\ 10^{5}$	$8.91\ 10^{0}$ / $4.99\ 10^{5}$
2000	$1.47\ 10^{1}$ / $1.99\ 10^{6}$	$2.16\ 10^{1}$ / $1.99\ 10^{6}$	$3.54\ 10^{1}$ / $1.99\ 10^{6}$
5000	$9.58\ 10^{1}$ / $1.25\ 10^{7}$	$1.39\ 10^{2}$ / $1.25\ 10^{7}$	$2.25\ 10^{2}$ / $1.25\ 10^{7}$
10000	$4.02\ 10^{2}$ / $5.00\ 10^{7}$	$5.77\ 10^{2}$ / $5.00\ 10^{7}$	$9.17\ 10^{2}$ / $5.00\ 10^{7}$

Table 17: Running time in seconds and No. of comparisons for SLINK.

$n \setminus d$	16	32	64
100	$2.80\ 10^{-1}$ / $3.90\ 10^{4}$	$3.60\ 10^{-1}$ / $6.18\ 10^{4}$	$7.80\ 10^{-1}$ / $1.14\ 10^{5}$
200	$8.80\ 10^{-1}$ / $1.24\ 10^{5}$	$1.46\ 10^{0}$ / $2.33\ 10^{5}$	$2.98\ 10^{0}$ / $4.14\ 10^{5}$
500	$6.42\ 10^{0}$ / $8.36\ 10^{5}$	$1.01\ 10^{1}$ / $1.57\ 10^{6}$	$1.90\ 10^{1}$ / $2.52\ 10^{6}$
1000	$2.70\ 10^{1}$ / $3.35\ 10^{6}$	$4.00\ 10^{1}$ / $6.05\ 10^{6}$	$8.18\ 10^{1}$ / $1.06\ 10^{7}$
2000	-	$1.48\ 10^{2}$ / $2.20\ 10^{7}$	$3.25\ 10^{2}$ / $4.14\ 10^{7}$
5000	-	$1.10\ 10^{3}$ / $1.46\ 10^{8}$	$2.24\ 10^{3}$ / $2.61\ 10^{8}$
10000	-	$4.18\ 10^{3}$ / $5.36\ 10^{8}$	$8.84\ 10^{3}$ / $9.90\ 10^{8}$
$n \setminus d$	128	256	512
100	$1.65\ 10^{0}$ / $1.83\ 10^{5}$	$3.11\ 10^{0}$ / $2.95\ 10^{5}$	$6.52\ 10^{0}$ / $3.74\ 10^{5}$
200	$6.90\ 10^{0}$ / $7.55\ 10^{5}$	$1.36\ 10^{1}$ / $1.28\ 10^{6}$	$3.65\ 10^{1}$ / $2.08\ 10^{6}$
500	$4.38\ 10^{1}$ / $4.66\ 10^{6}$	$9.05\ 10^{1}$ / $8.53\ 10^{6}$	$2.59\ 10^{2}$ / $1.48\ 10^{7}$
1000	$1.73\ 10^{2}$ / $1.80\ 10^{7}$	$3.66\ 10^{2}$ / $3.43\ 10^{7}$	$1.05\ 10^{3}$ / $6.02\ 10^{7}$
2000	$7.14\ 10^{2}$ / $7.35\ 10^{7}$	$1.48\ 10^{3}$ / $1.38\ 10^{8}$	$4.41\ 10^{3}$ / $2.50\ 10^{8}$
5000	$4.72\ 10^{3}$ / $4.55\ 10^{8}$	$9.40\ 10^{3}$ / $8.62\ 10^{8}$	$2.84\ 10^{4}$ / $1.60\ 10^{9}$
10000	$2.00\ 10^{4}$ / $1.85\ 10^{9}$	$3.76\ 10^{4}$ / $3.41\ 10^{9}$	$1.17\ 10^{5}$ / $6.54\ 10^{9}$

Table 18: Running time in seconds and No. of comparisons for CLINK1.

$n \setminus d$	16	32	64
100	$3.80\ 10^{-1}$ / $4.58\ 10^{4}$	$4.40\ 10^{-1}$ / $6.59\ 10^{4}$	$5.40\ 10^{-1}$ / $9.24\ 10^{4}$
200	$1.27\ 10^{0}$ / $1.63\ 10^{5}$	$2.20\ 10^{0}$ / $2.64\ 10^{5}$	$2.36\ 10^{0}$ / $3.88\ 10^{5}$
500	$8.35\ 10^{0}$ / $1.05\ 10^{6}$	$1.67\ 10^{1}$ / $1.87\ 10^{6}$	$2.03\ 10^{1}$ / $3.40\ 10^{6}$
1000	$3.92\ 10^{1}$ / $4.72\ 10^{6}$	$6.69\ 10^{1}$ / $7.51\ 10^{6}$	$1.20\ 10^{2}$ / $2.01\ 10^{7}$
2000	-	$2.42\ 10^{2}$ / $2.75\ 10^{7}$	$5.73\ 10^{2}$ / $9.46\ 10^{7}$
5000	-	$2.11\ 10^{3}$ / $2.03\ 10^{8}$	$3.12\ 10^{3}$ / $5.16\ 10^{8}$
10000	-	$9.92\ 10^{3}$ / $9.05\ 10^{8}$	$1.06\ 10^{4}$ / $1.72\ 10^{9}$
$n \setminus d$	128	256	512
100	$1.21\ 10^{0}$ / $1.48\ 10^{5}$	$2.52\ 10^{0}$ / $2.13\ 10^{5}$	$5.21\ 10^{0}$ / $2.82\ 10^{5}$
200	$4.83\ 10^{0}$ / $5.93\ 10^{5}$	$1.06\ 10^{1}$ / $9.15\ 10^{5}$	$2.85\ 10^{1}$ / $1.54\ 10^{6}$
500	$3.70\ 10^{1}$ / $4.70\ 10^{6}$	$7.81\ 10^{1}$ / $6.82\ 10^{6}$	$2.00\ 10^{2}$ / $1.08\ 10^{7}$
1000	$2.07\ 10^{2}$ / $2.65\ 10^{7}$	$3.82\ 10^{2}$ / $3.33\ 10^{7}$	$9.46\ 10^{2}$ / $5.14\ 10^{7}$
2000	$1.46\ 10^{3}$ / $1.86\ 10^{8}$	$2.29\ 10^{3}$ / $1.99\ 10^{8}$	$4.88\ 10^{3}$ / $2.65\ 10^{8}$
5000	$1.78\ 10^{4}$ / $2.24\ 10^{9}$	$2.94\ 10^{4}$ / $2.56\ 10^{9}$	$5.40\ 10^{4}$ / $2.93\ 10^{9}$
10000	$1.04\ 10^{5}$ / $1.30\ 10^{10}$	$2.16\ 10^{5}$ / $1.88\ 10^{10}$	$3.84\ 10^{5}$ / $2.06\ 10^{10}$

Table 19: Running time in seconds and No. of comparisons for MINRADI.

$n \setminus d$	16	32	64
100	$9.90\ 10^{-1}$ / $1.11\ 10^{5}$ / 14	$8.00\ 10^{-1}$ / $1.15\ 10^{5}$ / 16	$9.60\ 10^{-1}$ / $1.27\ 10^{5}$ / 17
200	$4.61\ 10^{0}$ / $5.29\ 10^{5}$ / 17	$4.33\ 10^{0}$ / $6.09\ 10^{5}$ / 20	$5.80\ 10^{0}$ / $7.28\ 10^{5}$ / 26
500	$4.53\ 10^{1}$ / $5.08\ 10^{6}$ / 28	$3.65\ 10^{1}$ / $5.15\ 10^{6}$ / 29	$4.94\ 10^{1}$ / $6.14\ 10^{6}$ / 35
1000	$2.44\ 10^{2}$ / $2.70\ 10^{7}$ / 37	$1.67\ 10^{2}$ / $2.34\ 10^{7}$ / 31	$2.30\ 10^{2}$ / $2.84\ 10^{7}$ / 45
2000	-	$9.00\ 10^{2}$ / $1.13\ 10^{8}$ / 37	$1.23\ 10^{3}$ / $1.38\ 10^{8}$ / 57
5000	-	$7.96\ 10^{3}$ / $9.77\ 10^{8}$ / 55	$9.83\ 10^{3}$ / $1.08\ 10^{9}$ / 65
10000	-	$3.78\ 10^{4}$ / $4.59\ 10^{9}$ / 65	$4.66\ 10^{4}$ / $5.07\ 10^{9}$ / 76
$n \setminus d$	128	256	512
100	$9.70\ 10^{-1}$ / $1.31\ 10^{5}$ / 18	$1.46\ 10^{0}$ / $1.35\ 10^{5}$ / 17	$2.62\ 10^{0}$ / $1.47\ 10^{5}$ / 22
200	$5.74\ 10^{0}$ / $7.77\ 10^{5}$ / 26	$8.66\ 10^{0}$ / $7.93\ 10^{5}$ / 26	$1.59\ 10^{1}$ / $8.99\ 10^{5}$ / 31
500	$4.87\ 10^{1}$ / $6.53\ 10^{6}$ / 37	$8.56\ 10^{1}$ / $7.82\ 10^{6}$ / 42	$1.66\ 10^{2}$ / $9.36\ 10^{6}$ / 53
1000	$2.49\ 10^{2}$ / $3.33\ 10^{7}$ / 53	$4.56\ 10^{2}$ / $4.15\ 10^{7}$ / 65	$9.11\ 10^{2}$ / $5.11\ 10^{7}$ / 80
2000	$1.27\ 10^{3}$ / $1.69\ 10^{8}$ / 76	$2.25\ 10^{3}$ / $2.04\ 10^{8}$ / 87	$4.34\ 10^{3}$ / $2.41\ 10^{8}$ / 100
5000	$1.02\ 10^{4}$ / $1.32\ 10^{9}$ / 107	$1.85\ 10^{4}$ / $1.64\ 10^{9}$ / 137	$3.59\ 10^{4}$ / $1.97\ 10^{9}$ / 159
10000	$4.86\ 10^{4}$ / $6.31\ 10^{9}$ / 128	$8.68\ 10^{4}$ / $7.68\ 10^{9}$ / 183	$1.71\ 10^{5}$ / $9.42\ 10^{9}$ / 218

Table 20: Running time in seconds, No. of comparisons and No. of iterations for CLINK2.

$n \setminus d$	16	32	64
100	$1.43\ 10^0$ / $1.47\ 10^5$ / 17	$1.76\ 10^0$ / $1.64\ 10^5$ / 20	$2.03\ 10^0$ / $1.73\ 10^5$ / 22
200	$7.36\ 10^0$ / $7.37\ 10^5$ / 24	$8.46\ 10^0$ / $8.09\ 10^5$ / 24	$1.08\ 10^1$ / $9.43\ 10^5$ / 29
500	$5.99\ 10^1$ / $5.91\ 10^6$ / 31	$7.00\ 10^1$ / $6.66\ 10^6$ / 34	$9.17\ 10^1$ / $8.00\ 10^6$ / 44
1000	$3.45\ 10^2$ / $3.33\ 10^7$ / 50	$3.47\ 10^2$ / $3.31\ 10^7$ / 43	$4.56\ 10^2$ / $3.94\ 10^7$ / 55
2000	-	$1.84\ 10^3$ / $1.56\ 10^8$ / 52	$2.56\ 10^3$ / $2.00\ 10^8$ / 71
5000	-	$1.43\ 10^4$ / $1.20\ 10^9$ / 70	$2.05\ 10^4$ / $1.60\ 10^9$ / 93
10000	-	$6.57\ 10^4$ / $5.49\ 10^9$ / 80	$9.85\ 10^4$ / $7.67\ 10^9$ / 117
$n \setminus d$	128	256	512
100	$2.09\ 10^0$ / $1.55\ 10^5$ / 19	$2.15\ 10^0$ / $1.72\ 10^5$ / 20	$3.73\ 10^0$ / $1.94\ 10^5$ / 26
200	$1.25\ 10^1$ / $9.50\ 10^5$ / 30	$1.25\ 10^1$ / $1.00\ 10^6$ / 31	$2.03\ 10^1$ / $1.05\ 10^6$ / 35
500	$1.13\ 10^2$ / $8.46\ 10^6$ / 44	$1.20\ 10^2$ / $9.66\ 10^6$ / 51	$2.05\ 10^2$ / $1.05\ 10^7$ / 56
1000	$5.86\ 10^2$ / $4.34\ 10^7$ / 62	$6.47\ 10^2$ / $5.16\ 10^7$ / 73	$1.15\ 10^3$ / $5.93\ 10^7$ / 88
2000	$3.70\ 10^3$ / $2.52\ 10^8$ / 94	$3.58\ 10^3$ / $2.85\ 10^8$ / 106	$6.32\ 10^3$ / $3.23\ 10^8$ / 122
5000	$3.20\ 10^4$ / $2.18\ 10^9$ / 136	$3.34\ 10^4$ / $2.69\ 10^9$ / 173	$6.24\ 10^4$ / $3.21\ 10^9$ / 209
10000	$1.61\ 10^5$ / $1.10\ 10^{10}$ / 177	$1.80\ 10^5$ / $1.42\ 10^{10}$ / 237	$3.42\ 10^5$ / $1.74\ 10^{10}$ / 300

Table 21: Running time in seconds, No. of comparisons and No. of iterations for KRNLSUM.

$n \setminus d$	16	32	64
100	$2.20\ 10^{-1}$ / $2.26\ 10^4$	$2.80\ 10^{-1}$ / $2.80\ 10^4$	$4.40\ 10^{-1}$ / $4.43\ 10^4$
200	$7.20\ 10^{-1}$ / $7.96\ 10^4$	$9.90\ 10^{-1}$ / $1.04\ 10^5$	$1.59\ 10^0$ / $1.64\ 10^5$
500	$4.78\ 10^0$ / $5.57\ 10^5$	$7.19\ 10^0$ / $8.04\ 10^5$	$1.04\ 10^1$ / $1.07\ 10^6$
1000	$1.71\ 10^1$ / $1.98\ 10^6$	$2.55\ 10^1$ / $2.80\ 10^6$	$4.49\ 10^1$ / $4.58\ 10^6$
2000	-	$9.39\ 10^1$ / $1.02\ 10^7$	$1.67\ 10^2$ / $1.69\ 10^7$
5000	-	$8.76\ 10^2$ / $7.97\ 10^7$	$1.07\ 10^3$ / $9.38\ 10^7$
10000	-	$3.26\ 10^3$ / $2.83\ 10^8$	$4.44\ 10^3$ / $3.67\ 10^8$
$n \setminus d$	128	256	512
100	$4.40\ 10^{-1}$ / $4.85\ 10^4$	$6.60\ 10^{-1}$ / $5.82\ 10^4$	$1.43\ 10^0$ / $7.80\ 10^4$
200	$1.26\ 10^0$ / $1.61\ 10^5$	$2.69\ 10^0$ / $2.38\ 10^5$	$5.82\ 10^0$ / $3.22\ 10^5$
500	$9.01\ 10^0$ / $1.17\ 10^6$	$1.49\ 10^1$ / $1.34\ 10^6$	$3.16\ 10^1$ / $1.76\ 10^6$
1000	$3.64\ 10^1$ / $4.80\ 10^6$	$6.28\ 10^1$ / $5.67\ 10^6$	$1.48\ 10^2$ / $8.27\ 10^6$
2000	$1.52\ 10^2$ / $2.02\ 10^7$	$2.96\ 10^2$ / $2.69\ 10^7$	$6.47\ 10^2$ / $3.62\ 10^7$
5000	$8.81\ 10^2$ / $1.14\ 10^8$	$1.77\ 10^3$ / $1.59\ 10^8$	$4.45\ 10^3$ / $2.47\ 10^8$
10000	$3.68\ 10^3$ / $4.62\ 10^8$	$6.27\ 10^3$ / $5.48\ 10^8$	$1.60\ 10^4$ / $8.78\ 10^8$

Table 22: Running time in seconds and No. of comparisons for DIVIDIAM.

D Figures for Special Vertex Sets with 10 Cluster centers

Appendix D shows some of the results of the tables in Appendix C as curves. The first group (Figures 21 to 24) shows the number of vertex comparisons over the number n of elements. The last Figure (No. 25) gives the number of iterations of the WHILE loop for the algorithm KRNLSUM.

For all figures we give five curves for dimension $d = 32, 64, 128, 256, 512$. For a discussion of these curves refer to Section 4.

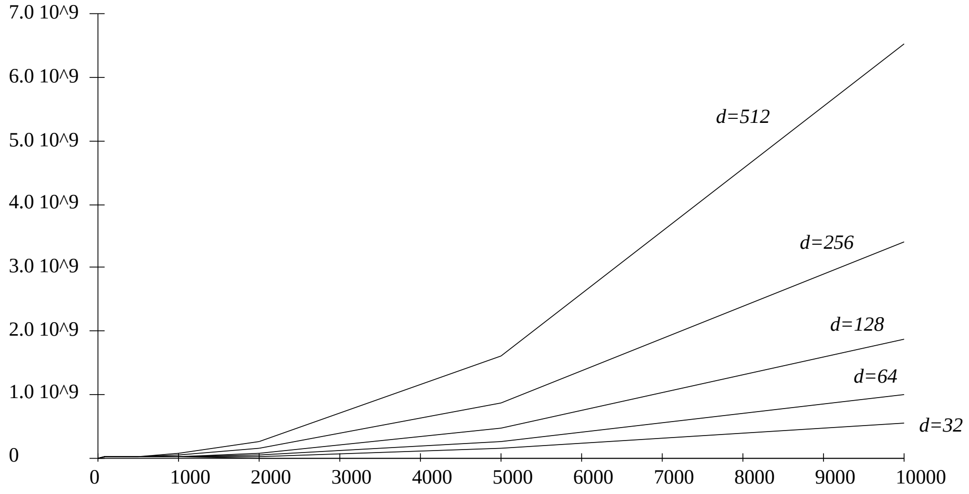


Figure 21: Number of vertex comparisons as function of the number of elements for CLINK1.

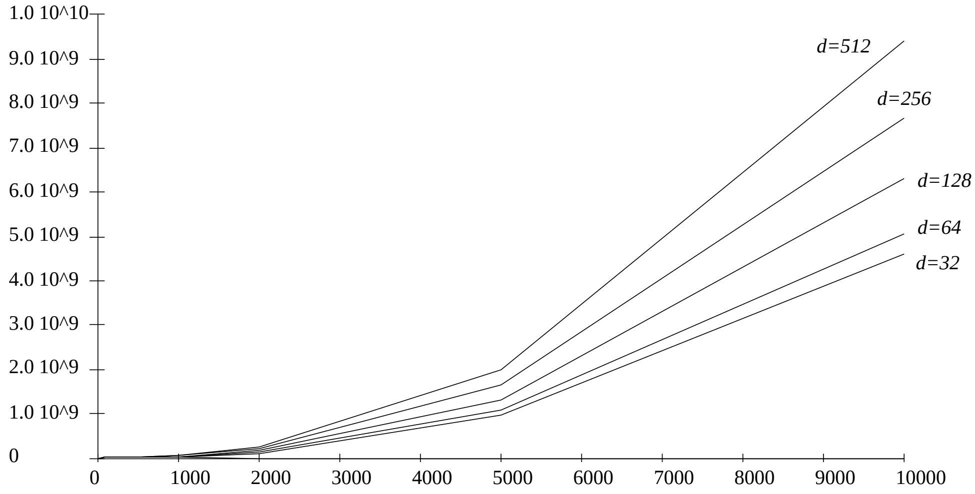


Figure 22: Number of vertex comparisons as function of the number of elements for CLINK2.

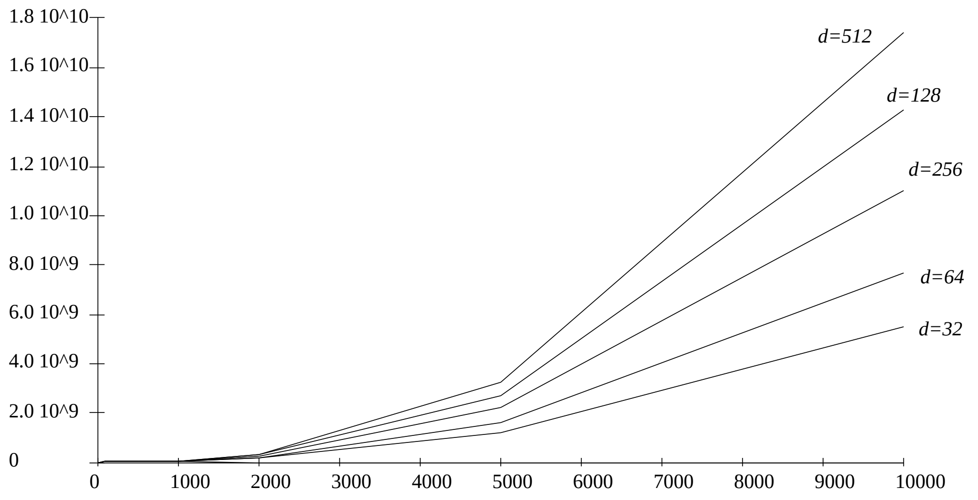


Figure 23: Number of vertex comparisons as function of the number of elements for KRNLSUM.

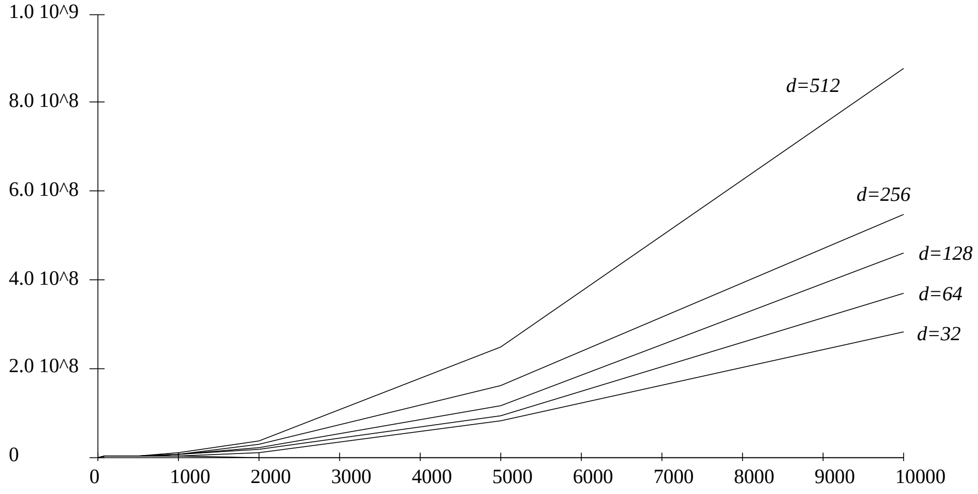


Figure 24: Number of vertex comparisons as function of the number of elements for DIVIDIAM.

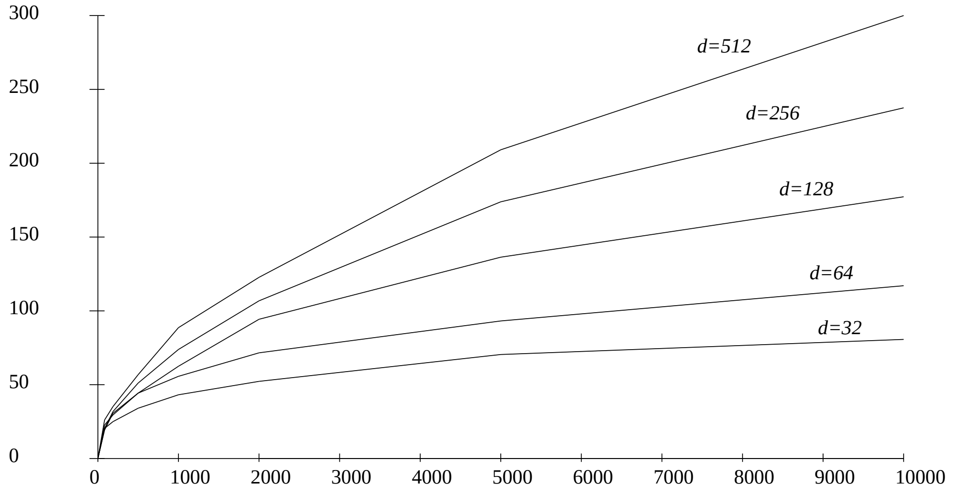


Figure 25: Number of iterations of the WHILE loop as function of the number of elements for KRNLSUM.