Karl-Franzens-Universität Graz & Technische Universität Graz

SPEZIALFORSCHUNGSBEREICH F 003

OPTIMIERUNG

und

KONTROLLE

Projektbereich

DISKRETE OPTIMIERUNG

Oswin Aichholzer

Efficient $\{0,1\}$-String Searching Based on Pre-clustering

Bericht Nr. 94 – Dezember 1996

Efficient $\{0,1\}$-String Searching Based on Pre-clustering

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 the {0,1}-string searching problem. For a given set S of binary strings of fixed length d and a query string q one asks for the most similar string in S. Thereby the dissimilarity of two given strings is the number of disagreeing bits, that is, their Hamming distance. We present an efficient {0,1}-string searching algorithm based on hierarchical pre-clustering. To this end we give several useful observations on the inter- and intra-cluster distances.

The presented algorithms are easy to implement and we give exhaustive experimental results for uniformly distributed sets as well as for specially chosen strings. These experiments indicate that our algorithms work well in practice.

1 Introduction

1.1 Notation and Problem Definition

In this paper we consider the {0,1}-string search problem. For a given set S of binary strings of fixed length d and a query string q we ask for the most similar string in S. Thereby the dissimilarity of two given strings is the number of disagreeing bits, that is, their Hamming distance. A set of such strings can be seen as a subset of the vertices of a d-dimensional hypercube, and thus the {0,1}-string search problem is a geometric problem in d dimensions.

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 also be seen as the set of all binary strings of length d. The *Hamming distance* of two vertices $x, y \in V^d$ is the number of coordinates where x differs from y. For brevity, we will write $d_H(x,y)$.

A formal definition of the string search problem is given below.

{0,1}-String Search Problem:
INSTANCE: Given a set $C \subset V^d$ with $|C| = n$.
QUESTION: For a query string $x \in V^d$ report an element c which attains $\min_{c \in C}\{d_H(x,c)\}$.

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

In other words, for a given set C of n strings of length d and a query string x we search for the best matching in C, i.e. an element of C with the minimum Hamming distance to x. Typical applications may have string length of, say, 64 to 512 bits or more, and several thousands to tens of thousands strings. Usually we can expect a huge number of queries for the same set, so that a preprocessing step for the problem is appropriate.

Trivially this problem can be solved in $O(dn)$ time by exhaustive comparison. Note that the constant in this notation is rather small since typical computers can compare 16 or 32 bits within one operations. Including some overhead for storage managing, a typical constant in the big O notation may therefore be $\frac{1}{10}$. Thus if looking for faster practical algorithms we not only have to look for asymptotic running times of $o(dn)$ but also have to ensure small constants.

Clearly the Hamming distance is symmetric and invariant where 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 a given set. Suppose this would not be the case, e.g. we can weight the bits according to their position. Clearly any such weighting which fulfills some obvious requirements leads to a fast sorting and searching solution similar to the binary bisection searching. Even if this requirements are only partially fulfilled we might use sorting to find a near optimal solution and then to do some local search according to the sorting. Although this might lead to interesting related problems we will not go into more detail for these special cases.

For a uniformly distributed random vertex set C and a random query point q it is very likely to have a large optimal Hamming distance, typically in the range $\{\frac{d}{3}, \dots, \frac{d}{2}\}$. But for practical applications this distance is much to large since we can not speak of a "very similar" object if about $\frac{1}{3}$ of the bits differ. Therefore we will not only consider random queries but also query vertices which are close to some vertex of the set C, i.e. which guarantee to have a small Hamming distance. We believe that this meets the needs of applications, since for applications there are most likely several groups of very similar "objects". The object, for which the search is done, might belong to one of these groups, that is, it might have a very small Hamming distance to its nearest neighbour. We will therefore study the number of necessary distance computations in comparison to the Hamming distance between the query vertex and its optimal solution. In fact it will turn out that this is a very distinctive relation.

Surprisingly it seems that that up to now there is no simple efficient solution even if we allow to use a wasteful preprocessing. Moreover M. Minsky conjectured in [7] that "...for large data sets with long word length there are no practical alternatives to large searches that inspect large parts of the memory.".

We will give a brief overview of previous approaches in Section 1.2. In Section 2 we then present an efficient {0,1}-string search algorithm based on hierarchical pre-clustering. To this end we also give several useful observations on the inter- and intra-cluster distances within a cluster tree. Section 3 gives the results of an exhaustive number of experiments, indicating that our algorithms work well in practice.

1.2 Related Work

There are several possible attempts to the $\{0, 1\}$-string search problem which can be found in the literature. A possible approach, which works well in two space, is the use of the well known Voronoi-diagram, see [3] for a survey. Unfortunately the storage needs grow exponentially with the dimension (according to the maximal size of a polytope in $d+1$ dimensions, $O(n^{\lceil d/2 \rceil})$), and is therefore not suitable for the d-cube case.

1.2.1 Approximate Elimination Search Algorithm (AESA) and Variants

In his paper Vidal [12] introduced a new technique for finding nearest neighbours in (approximately) constant time, the so called Approximate Elimination Search Algorithm (AESA). This approach is known to work quiet well for low dimensional point sets in $\mathbb{R}^d$, where typical values of d are 3 to 8.

There are several improvements reported on the AESA, e.g. a version using only linear storage by Ramasubramanian and Paliwal [8]. This approach is based on a new spherical distance coordinate formulation, the so called (Incremental) Fixed Anchor Point-AESA ((I)FAP-AESA). Moreover in [13] and [14] new formulations and slight extensions of [12] are given, e.g. the Reduced Overhead AESA (ROAESA).

In all these papers the application to the hypercube with the Hamming distance metric is reported, but the examples given there are only for dimension 10 or less. This means that we have at most 1024 possible locations for vertices. Thus a full table for any possible query point with the location of its best matching string in the given set C can easily be maintained. Storing the 10 bit information of the optimal solution for every potential query point needs only 2048 byte (exactly 2KB) and can be initialized with the trivial $O(dn)$ procedure within some milliseconds. Any query now can clearly be answered in constant time with only one index operation and beats thus all AESA based methods by far. Moreover even the overall storage for data and program is in this case much smaller than the storage needed for an implementation of AESA or its variants.

Thus the examples for hypercubes given in this papers are to small to be significant. We therefore considered the mentioned nearest neighbour methods also for higher dimensions. Our experiments showed that AESA and its variants are not suitable for the high dimensional hypercube case, which was also reported by other researchers [2].

One intuitive reason why these nearest neighbour algorithms fail to be efficient is the phenomena of the interplay of hypersphere and hypercube. Measuring the distance to a given point p is a "unit sphere operation": reporting all points within a distance of at most, say, k to a given point p is the same as reporting all points within a sphere of radius k and centered at p, where the type of sphere depends on the underlying metric. For low dimensions now the d-cube and d-sphere are similar enough, so that algorithms based on spherical distances work well. But for higher dimensions the unit cube and unit sphere behave totally different. While the d-cube still has unit volume, its diameter increases arbitrarily with the dimension. On the other hand the diameter of the unit sphere remains unchanged, whereas its volume vanishes with increasing dimension. This different behavior in "distance" and "volume" makes it impossible to use distances related to the hypersphere to cover vertices and thus fractions of the volume of the hypercube.

Note that this is not only true for the Euclidean distance, but for any L_p metric, especially the L_1-metric and thus the Hamming distance. E.g., the distance of an antipodal pair increases even linearly in the dimension for the L_1-metric, whereas it depends only with the square root in the Euclidean case.

To illustrate the situation consider the center point of the cube. If we generalize the Hamming distance to the L_1-metric, then the center point of the d-cube is at distance $\frac{d}{2}$ to the origin (which is quiet natural, since it lies half way to the antipodal of the origin). If in 2-space we place the unit circle (with radius 1) at the origin it clearly contains the center point $(\frac{1}{2}, \frac{1}{2})$ of the unit square. This remains true for the unit ball and unit cube in 3D, but already in 4D the center point of the 4-cube lies on the surface of the 4-ball. Starting with dimension 5 the unit ball centered at the origin does not longer cover the center point of the unit cube. (Cf. this with the well known paradoxon of the decreasing volume of the unit ball for dimensions greater than 5.)

1.2.2 Bucketing Algorithms

A very simple algorithm that works well for uniformly distributed points in low dimensions is the bucketing algorithm, also called Elias's algorithm [15]. The space is divided into identical cells, storing all vertices inside in a linear list. For a query point the cells are examined in increasing distance, until the minimum distance to a cell exceeds the distance to the closest point found.

An analysis and some experiments of Elias's algorithm can be found in [9]. In [4] this approach was rediscovered independently and also used for the hypercube case. But the experiments of the mentioned papers give only results for dimension 30 and less, and the approach seems to be inefficient for higher dimensions. Also the restriction to uniformly distributed points seems not to be appropriate for our problem, as will be shown in the next sections.

1.2.3 Local Search

Another attempt to the problem would be to use the well known "local search" technique which is used to solve many hard optimization problems. On a first glance our problem seems to have some nice properties for this approach. First we have no problem finding a starting point for the local search, since any vertex of our set C is a "feasible" solution. And second, since any step of the local search goes towards the minimum, it must decrease the Hamming distance to the query point by at least one. Therefore we know beforehand that the process of local search terminates after at most $O(d)$ iterations.

For the correctness of the local search approach we have to guarantee that, if we have not yet obtained the optimal solution, for any vertex $c \in C$ there is at least one local neighbour which decreases the Hamming distance to the query point. If this condition is fulfilled we clearly can not get stuck in a local optimum.

Obviously the edges of the convex hull of a point set fulfill these requirements. It is not hard to see that any point of C has to lie on the convex hull of C: consider the hyper sphere S^d with center $(\frac{1}{2}, \dots, \frac{1}{2})$ and radius $\frac{\sqrt{d}}{2}$. All vertices of C^d lie on S^d and thus any subset of V^d entirely consists of extreme vertices. Therefore computing the edge skeleton of the convex hull of C immediately gives us the required local neighbourhood relations for local search.

Unfortunately it turned out that the convex hull of a subset of hypercube vertices has a "complete-graph" like behavior. Figure 1 shows the average degree of the convex hull of a subset of vertices of the 10-cube opposed to the size of this set. Note that especially for small subsets, which are likely for higher dimensions, the degree increases linearly with the size. Thus for the interesting cases where $|V^d| = 2^d$ is much larger than the number n of vertices of C it can be seen that the convex hull of C is likely to be a 2-neighbourly polytope (cf. [16]) or a close approximation of it. That is, nearly every pair of vertices in C is connected by an edge

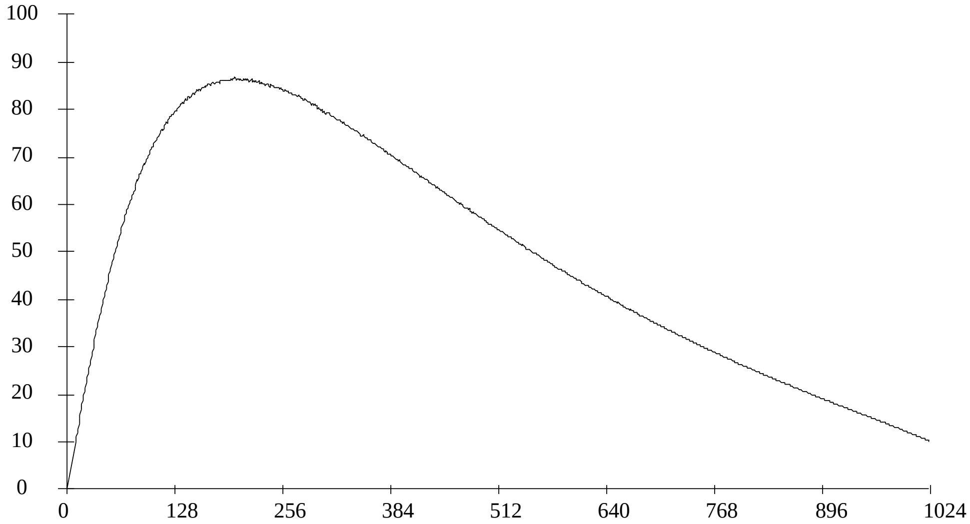


Figure 1: Average degree of the convex hull of a subset of n vertices of the 10-cube.

of the convex hull of C. For example there exists a 2-neighbourly polytope with 1035 edges and 46 vertices of $\mathcal{C}^{12}$.

Thus starting with some arbitrary vertex it is not only adjacent already with the optimal solution, but also to any other vertex of C. In this case only the first iteration of local search will already need $O(dn)$ time.

This observation is strongly related to a result of Füredi [6]. He showed that for a subset of $\mathcal{C}^d$ and for an arbitrary $\epsilon > 0$ the convex hull of more than $(2+\epsilon)d$ vertices almost always contains the center of the cube, while the convex hull of less than $(2+\epsilon)d$ points almost always avoids it. From linear programming it is easy to see that this coincides with the question whether the diagonal of a pair of vertices with Hamming distance k is part of the convex hull. If the k-dimensional subcube induced by these two vertices contains less than $2k$ vertices from the given set, it is likely that their convex hull does not contain the center of the subcube. The considered diagonal will then be part of the convex hull. Therefore it is reasonable that if the number of vertices is small, the edges of the convex hull will approximate the complete graph.

2 $\{0,1\}$-String Searching using Hierarchical Clustering

In this section we use a new approach to investigate the $\{0,1\}$-string search problem for a given set $S \subseteq V^d$, which is based on the structure of hierarchical cluster trees. 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.

In order to utilize the cluster structure for our string searching we compute for any cluster C_i of the cluster tree of S a kernel c_i, which has to be an element of C_i. Depending on the method used to build the cluster tree these kernels may already be computed during the clustering process. There are several methods to determine the kernel of a given point set and

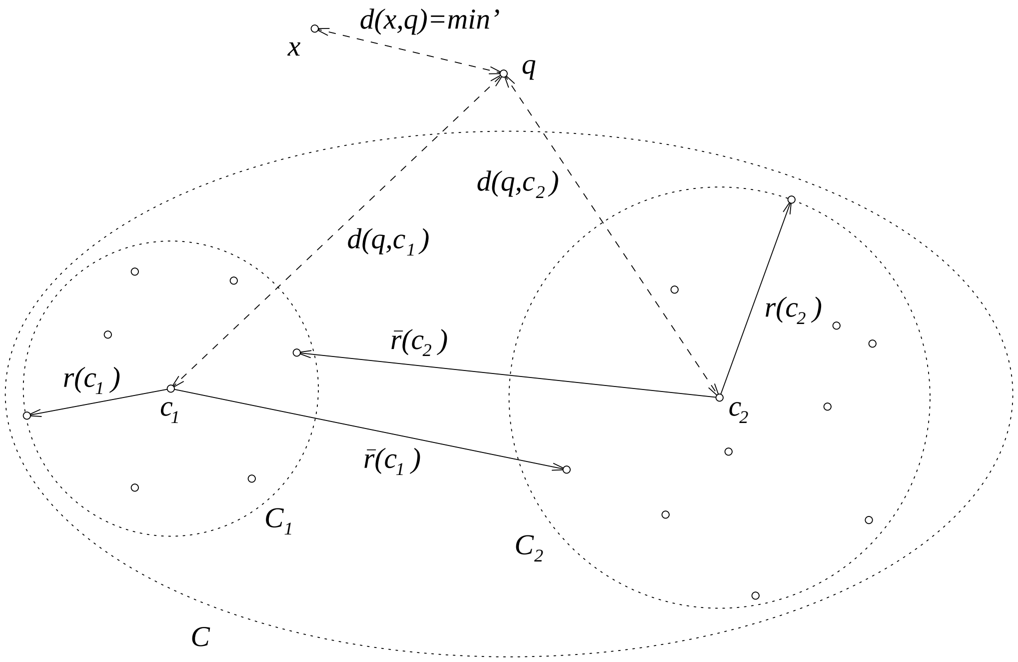


Figure 2: Relation of inter point distances for two subclusters C_1 and C_2 of C.

this clearly will effect the efficiency of the algorithms; we will give more details about that later on in this section.

Let us now give a description of our approach. For any query point $q \in V^d$ we start at the kernel of the whole set and then traverse the cluster tree in a depth-first-search like manner. For any branch of the cluster tree we consider the two subclusters and decide locally whether we have to search into them or not. That is whether or not they may contain vertices which are closer to the query point than the best solution found so far. Thus, by cutting off many of the branches, the algorithm may only consider a (hopefully small) subset of the given vertices of S in order to find the global optimum.

If a vertex $p \in S$ is the kernel of several clusters it may be considered more than once during the search task. Thus in addition to the cluster tree we store in a linear list whether the distance from the query point to a vertex $p \in S$ has already been computed, and, if yes, we store this distance. Note, however, that the overall number of kernels is $|S| - 1$ and that every vertex $p \in S$ has to be a kernel at least once, namely for the cluster only containing p.

In the worst case, if we have to search through all subclusters, we therefore have to consider every kernel, and thus every vertex of S, at most once. Thus the upper bound for the number of d-dimensional comparisons is $|S|$, the same bound as for the trivial approach.

We now give the notation used for our new approach, refer to Figure 2. Let $q \in V^d$ be the query point, $x \in S$ the best solution found so far, and let $min' = d_H(x, q)$ denote their Hamming distance. Initially we set x to the kernel of the whole set S. For any branch step during the search in S let C be the cluster under consideration and C_1 and C_2 the two subclusters of C, with kernels c_1 and c_2, respectively. With $r(c_i) = \max_{p \in C_i} d_H(c_i, p)$ we denote the radius of a cluster, and $\overline{r}(c_i) = \min_{p \in C \setminus C_i} d_H(c_i, p)$ is the minimum distance from c_i to a point from C outside the cluster C_i.

The decision whether or not to branch into a subcluster is now based on the following trivial, but useful observation.

Observation 1 *Let C be a cluster in the cluster tree of S, and let C_1 and C_2 the two subclusters of C, with kernels c_1 and c_2, respectively. Let $r(c_i)$ and $\overline{r}(c_i)$, $i = \{1, 2\}$, be defined as above. For all vertices $p_1 \in C_1$ we have*

(1a) $d_H(q, p_1) \geq d_H(q, c_1) - r(c_1)$ *and*
(1b) $d_H(q, p_1) \geq \overline{r}(c_2) - d_H(q, c_2)$.

Analogously for all vertices $p_2 \in C_2$ we have

(2a) $d_H(q, p_2) \geq d_H(q, c_2) - r(c_2)$ *and*
(2b) $d_H(q, p_2) \geq \overline{r}(c_1) - d_H(q, c_1)$.

Note that the observation does not require any special property of the hypercube, but only use the triangle inequality. Thus the method given in this section may be generalized for any high dimensional geometric closest point search problem.

Inequality (1a) of Observation 1 for subcluster C_1, and analogously (2a) for C_2, states that if the kernel c_1 of the cluster C_1 is too far away from the query point q, then no point in C_1 can achieve a better solution. Clearly this inequality will only provide good bounds if the radius $r(c_1)$ of the cluster is small enough.

The second inequality (1b) shows that if the kernel c_2 of C_2 is a rather good approximation to q and if the points of C_1 are far away from c_2, as we may expect for a good two clustering of C, then again no point of C_1 can achieve a better solution. This again holds analogously for the subcluster C_2, shown by inequality (2b).

Now let

$$m(C_1) = max\,\{d_H(q, c_1) - r(c_1), \overline{r}(c_2) - d_H(q, c_2), 0\}$$
$$m(C_2) = max\,\{d_H(q, c_2) - r(c_2), \overline{r}(c_1) - d_H(q, c_1), 0\}$$

From Observation 1 it follows that $m(C_i)$ is a lower bound for the Hamming distance $d_H(p, q)$ for any point $p \in C_i$ to the query point q. We thus get

Corollary 1 *Let C, C_1, C_2, c_1, c_2, $r(c_1)$, $r(c_2)$, $\overline{r}(c_1)$, $\overline{r}(c_2)$, $m(C_i)$, and min' as defined above. Then C_i, $i = \{1, 2\}$, may contain a new optimal vertex only if $m(C_i) < min'$.*

Corollary 1 provides a simple, local criterion whether or not to branch into a subcluster of the cluster tree while searching for an optimum vertex. The use of the relations of distances in our approach is in some sense similar to what AESA [12] and IFAP-AESA [8] do. But because of using the cluster hierarchy we are not limited to a fixed or randomly chosen subset of “anchor points”. Instead, we use a set of kernels which reflects the structure of the given data. We thus expect our approach to be adaptive to specially structured vertex sets and thus to be efficient in practice.

2.1 Preprocessing

We are now ready to consider algorithmic issues. To generate the cluster tree T of a vertex set $S \subseteq V^d$, $n = |S|$, we utilize the hierarchical clustering methods presented in [1]. The output of the algorithms given there is the full cluster tree T: a linear list gives the indices of the $n-1$ agglomeration steps in a bottom up like manner, from single objects (vertices) to the final cluster, the root of the tree. In addition, for clustering methods based on the kernels of subcluster, these kernels are also given for each cluster (e.g. for KRNLSUM, see Section 2.5.2 in [1]). Thus we might also use this kernel information for precomputation. But since most hierarchical clustering methods are not based on kernels, we give here a general scheme for precomputing all necessary informations from a given cluster tree T.

From Corollary 1 we know that we have to precompute c_i, $r(c_i)$, and $\overline{r}(c_i)$ for every cluster C_i which is part of T. We will compute the kernel c_i in a way such that it minimizes the radius of C_i, i.e. $r(c_i) = \min_{c \in C_i} \{\max_{p \in C_i} d_H(c,p)\}$. Considering inequalities (1a) and (2a) of Observation 1 this clearly makes sense, since it locally maximizes the lower bound. We thus use a similar approach as was given in [1] for hierarchical clustering minimizing the radius of a cluster.

We now show that for an agglomeration step $C = C_1 \cup C_2$ in the cluster tree T we can compute the kernel c of C in $O(d|C_1| \cdot |C_2|)$ time. To this end for every vertex p of S we dynamically store the maximum distance to any other point contained in the same cluster as p. This requires only $O(n)$ additional space for the whole cluster tree. Initially, for single vertices, these maximum distances are set to zero. 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 C_1 and C_2 now can be computed in $O(d|C_1| \cdot |C_2|)$ time: For any point of C_1 (and symmetrically for C_2) we have to compute the maximum distance to a point in C_2 (C_1, respectively) and store the maximum of this distance and the previous maximum of the point. The vertex c obtaining the minimum of these maxima is the kernel of C and also determines $r(c)$. Moreover for c_1 (c_2) we compute at the same time the minimum distance to a point in C_2 (C_1), which gives $\overline{r}(c_1)$ and $\overline{r}(c_2)$, respectively.

To get all required information we reiterate the above agglomeration step bottom up for every cluster of T. Since thereby any pair of vertices is considered at most once, we have $O(n^2)$ distance computations and get:

Lemma 1 *Given the cluster tree T of a vertex set $S \subseteq V^d$, $n = |S|$. The computation of c_i, $r(c_i)$, and $\overline{r}(c_i)$ for every cluster C_i of T can be done in $O(dn^2)$ time and linear space.*

2.2 The Algorithms

From the last two subsections we now immediately get the first $\{0,1\}$-string search algorithm SEARCH1 shown in Figure 3. Thereby $S \subseteq V^d$ is the vertex set, $q \in V^d$ the query point, and T is the cluster tree of S including the additional information computed in the preprocessing phase.

The correctness of SEARCH1 follows from Corollary 1 and the explanations given above. Note that within the procedure DEPTH-FIRST-SEARCH we do not need to consider whether a cluster has subclusters or not. A cluster C without subclusters consists of a single vertex c (its kernel) and has therefore a radius $r(C) = 0$. Thus $m(C) \geq d_H(c,q) \geq min'$ which means that for such a cluster there will not be a recursive call.

Algorithm SEARCH1(S,T,q)

1. Let C_r be the final cluster of S (root of T) and c_r its kernel
2. Set $min' = d_H(c_r, q)$ and $best = c_r$
3. Call DEPTH-FIRST-SEARCH(C_r)
4. Output $best$

Procedure DEPTH-FIRST-SEARCH(C)

1. Let C_1 and C_2 be the two subclusters of C in T, with kernels c_1 and c_2, respectively
2. **FOR** i = 1 to 2 **DO**
   (a) **IF** $d_H(c_i, q) < min'$ **THEN**
      - $min' = d_H(c_i, q)$
      - $best = c_i$
3. Compute $m(C_1)$ and $m(C_2)$
4. **IF** $m(C_1) < min'$ **THEN** call DEPTH-FIRST-SEARCH(C_1)
5. **IF** $m(C_2) < min'$ **THEN** call DEPTH-FIRST-SEARCH(C_2)
6. Return

Figure 3: SEARCH1: A first approach to $\{0,1\}$-string searching using hierarchical clustering.

Using the linear list mentioned in the above discussion it is straight forward to see that SEARCH 1 runs in time linear in the number k of visited kernels, that is, in time $O(dk)$. Thus, to improve the running time, we have to minimize the number of necessary subcluster branches. Observe that after the first recursive call in step 4 of DEPTH-FIRST-SEARCH min' might have changed and thus the call of step 5 might be unnecessary even if $m(C_2) < min'$ was true before step 4. In other words, the sequence of recursive calls might effect the number of vertices we have to consider. Thus a heuristic should try to consider 'promising candidates' first. Instead of doing an inflexible depth first search we can sort the subclusters by a certain criterion and then branch into the "best" one. The hope is to rapidly find a good approximation of the optimum and thus to minimize min', such that we can avoid to consider most of the remaining subclusters.

This observations lead to Algorithm SEARCH2 given in Figure 4. For the heuristic we use the Hamming distance from the kernel c_i of a cluster C_i to the query point q. Clearly if the kernel is close to q then we might expect other vertices of C also to be close to q. On the other hand if the kernel is far away, there is little hope to find a"good" vertex within C.

To organize the subclusters we use a simple heap sorted by the distances $d_H(c_i, q)$ of the kernel points c_i to the query point q. Clearly only subclusters C with $m(C) < min'$ are put onto the heap and the algorithm iterates until the heap is empty. Thus all kernels of appropriate clusters are considered and the correctness of this approach follows.

In fact we can also use d linked lists, since $d_H(c_i, q) \in \{0, \ldots, d\}$. Inserting an element into the heap (put) or removing the minimum from it (get) can therefore be done in time

Algorithm SEARCH2(S,T,q)

1. Let C_r be the final cluster of S (root of T) and c_r its kernel
2. Set $min' = d_H(c_r, q)$ and $best = c_r$
3. PUT(C_r,$d_H(c_r, q)$)
4. **WHILE** heap not empty **DO**
   (a) GET(C,d)
   (b) Let C_1 and C_2 be the two subclusters of C in T, with kernels c_1 and c_2, respectively
   (c) **FOR** i = 1 to 2 **DO**
      i. **IF** $d_H(c_i, q) < min'$ **THEN**
         - $min' = d_H(c_i, q)$
         - $best = c_i$
      ii. **IF** $m(C_i) < min'$ **THEN** PUT($C_i, d_H(c_i, q)$)
5. Output *best*

Figure 4: SEARCH2: Using a heap heuristic for subclusters.

$O(d)$. Thus, if k is again the number of kernels to be considered, the whole algorithm runs in space and time $O(dk)$. Note that, by the same argumentation as given for the procedure DEPTH-FIRST-SEARCH, clusters without subclusters will not be put onto the heap.

Alternatively we might use the lower distance bounds $m(C_i)$ to sort the heap. This again reflects the idea behind the computation of these bounds. It is natural to expect a cluster with a small lower bound to be a promising candidate which should be considered first. We thus give a second heap based version, Algorithm SEARCH3, in Figure 5. Again we have $m(C_i) \in \{0, \ldots, d\}$ and thus the above given space and time bounds still hold.

SEARCH3 has the additionally advantage that after getting a cluster C from the heap we can check in constant time whether $m(C) < min'$ still holds. If this it not the case then we are done, since all remaining clusters on the heap have an even larger lower bound (step 4b in Figure 5). Thus we expect SEARCH3 to be the fastest string search algorithm given in this section, which will be confirmed by the experimental results of Sec. 3.

For the running time of all three presented algorithms we have

Lemma 2 *Algorithms SEARCH1, SEARCH2, and SEARCH3 run in $O(dk)$ space and time, where k is the number of vertices to be considered.*

3 Experimental Results

In this section we give experimental results on the number of necessary distance computations. We compare the three different search algorithms (Subsection 3.1) and various hierarchical clustering methods from [1] (Subsection 3.2). Finally we give results for different kinds of special input sets (Subsection 3.3).

Algorithm SEARCH3(S,T,q)

1. Let C_r be the final cluster of S (root of T) and c_r its kernel
2. Set $min' = d_H(c_r, q)$ and $best = c_r$
3. PUT(C_r,0)
4. **WHILE** heap not empty **DO**
   (a) GET(C,$m(C)$)
   (b) **IF** $m(C) \geq min'$ **THEN** goto 5
   (c) Let C_1 and C_2 be the two subclusters of C in T, with kernels c_1 and c_2, respectively
   (d) **FOR** i = 1 to 2 **DO**
      i. **IF** $d_H(c_i, q) < min'$ **THEN**
         - $min' = d_H(c_i, q)$
         - $best = c_i$
      ii. **IF** $m(C_i) < min'$ **THEN** PUT($C_i, m(C_i)$)
5. Output *best*

Figure 5: SEARCH3: Using a heap controlled by $m(C_i)$.

The diagram of Figure 6 shows the relation between the different programs and the data. From the vertex set, which comes from an application or is generated for test purposes, we get the cluster tree T by one of the various hierarchical clustering methods given in [1]. In the preprocessing phase we then compute c, $r(c)$ and $\overline{r}(c)$ for every cluster $C \in T$. Now we can apply one of our search algorithms and treat the query vertices, given either by the application, or specially generated from the input set or taken at random from V^d.

The generation of vertex sets as well as the generation of an appropriate set of query vertices is only needed for test purposes and thus drawn with dashed lines in Fig. 6. That we indeed need to generate query vertices instead of performing random queries can be seen from the fact that for higher dimensions (e.g. $d \geq 100$) the expected Hamming distance between a random query vertex and its optimal approximation typically lies in the range $\{\frac{d}{3}, \dots, \frac{d}{2}\}$ if the number of vertices is small compared to 2^d. Thus the optimal solution would not be significantly better than a random answer, which is expected to have Hamming distance $\frac{d}{2}$. We therefore always generated a query vertex set at random, but with a fixed Hamming distance in the range $\{0, \dots, \frac{d}{2}\}$ to some (again randomly chosen) vertex of the input set.

For every search we give the number of considered vertices opposed to the Hamming distance between the query vertex and its optimal solution. In fact it turns out that this is a very distinctive relation. Note that for a query vertex chosen at Hamming distance approx. $\frac{d}{2}$ from "its" input vertex, there might be a "better" vertex in this set, which we will get as the optimal output. Thus the given curves will most likely end before achieving $\frac{d}{2}$.

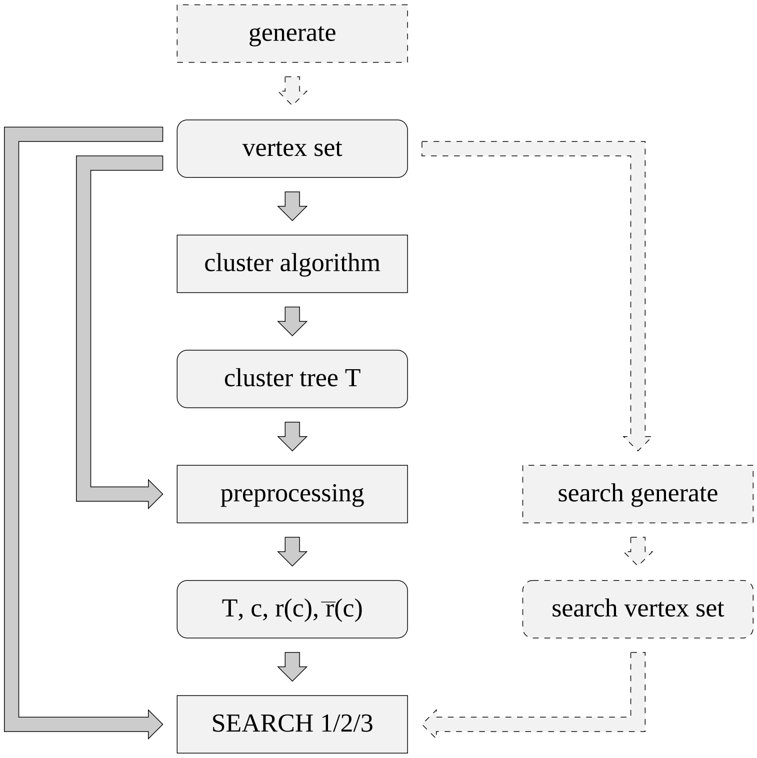


Figure 6: Relation between algorithms (rectangular boxes) and data (boxes with rounded corners). Dashed arrows and boxes are only for test purposes.

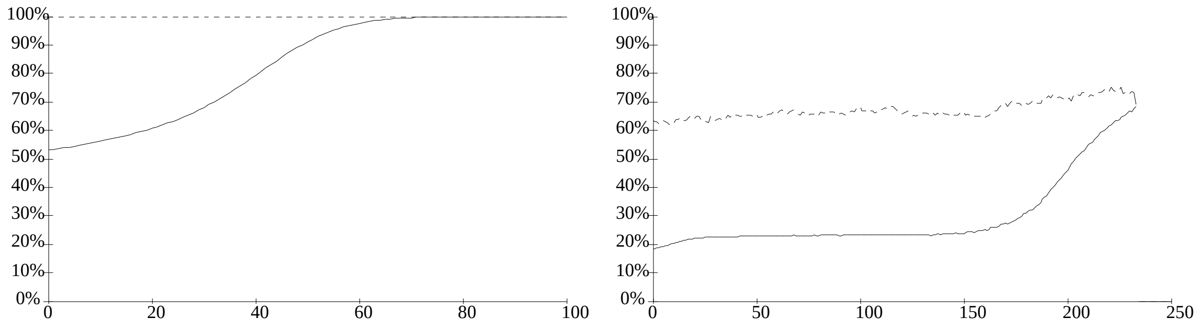
3.1 Comparison of Search Algorithms

In this subsection we compare the three search algorithms of Section 2.2 for two different input sets. Both sets consist of $n = 10000$ vertices of dimension $d = 512$. For the first set the vertices are chosen uniformly at random from V^d. Then these sets are clustered by Algorithm MINRADI (cf. [1]) in order to minimize the radius of the clusters. (For a short description of the cluster methods see Sec. 3.2.) We thus expect especially inequalities (1a) and (2a) of Observation 1 to take effect.

For the second set we generated the vertices around a fixed number k of centers, such that the resulting sets are structured sphere-like around each center, similar to the special sets used in [1]. 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 $\frac{d}{8}$. This forces of course a very special structure of the resulting cluster trees and we will see that in fact the experimental results are quite different compared to those for random sets.

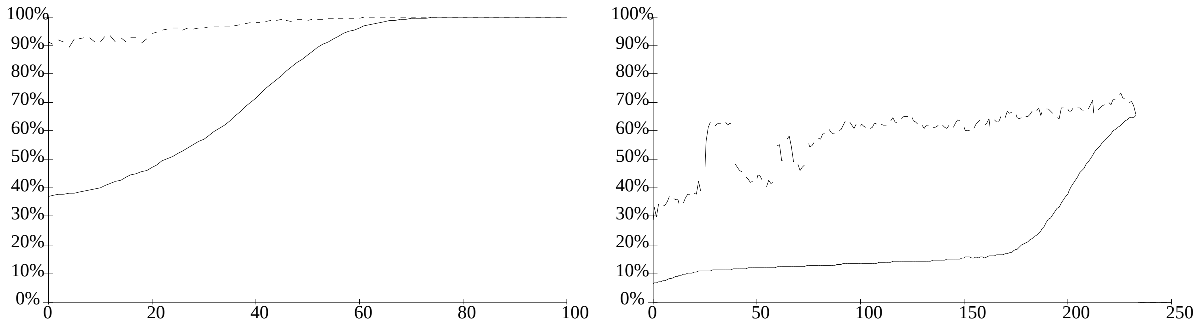
We believe that this kind of special vertex sets give a more realistic representation of real word problems than random sets do. For applications there are most likely several groups of very similar "objects", and the object for which the search is done might belong to one of these groups. In fact, not the optimal object, but the group the query object belongs to, might be the required answer.

To cluster these special sets we used a complete linkage clustering algorithm (CLINK1 in [1]).



(a) (b)

Figure 7: SEARCH1: Percentage of considered vertices against the Hamming distance between query vertex and optimal solution. Solid lines give the average case, dashed lines the worst case for over 100000 queries. (a) uniformly distributed random vertices; (b) special set with 10 cluster centers; Both sets with $n = 10000$ vertices of dimension $d = 512$.



(a) (b)

Figure 8: Analogous curves as in Fig. 7 for Algorithm SEARCH2.

Figures 7(a) to 9(b) give the results for over 100000 queries each, for Algorithms SEARCH1, SEARCH2, and SEARCH3, respectively.

For the random set the curves show that only for a small Hamming distance from the optimum we can avoid a reasonable number of comparisons. If the distance is to large (typically greater than $\frac{d}{8}$) we have to consider nearly 100% of the vertex set. Despite the fact that for the average case all three algorithms behave quite similar, SEARCH3 is obviously the most efficient one. If we consider the worst case behavior this distinction is very conspicuous. While for SEARCH1 there is always an example which requires at least 99% of the distance computations and also SEARCH2 needs 90% or more, the worst case of SEARCH3 lies only about $2-3\%$ above the average case. This clearly shows that for random vertex sets SEARCH3 is the recommendable algorithm.

For the mentioned special sets the situation is more pleasant for all three algorithms. This is especially true if the Hamming distance to the optimum vertex is small enough to quickly find the "cluster ball" where the query vertex belongs to (in our case below approx. $\frac{d}{3}$).

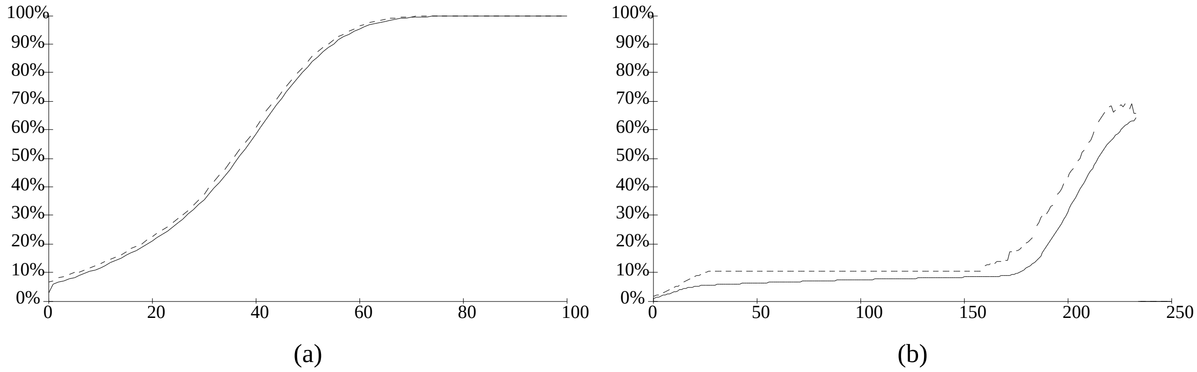


Figure 9: Analogous curves as in Fig. 7 for Algorithm SEARCH3.

Again SEARCH3 is clearly the best algorithm, having the average and worst case both below the average case of the other algorithms. For query points which have a very close optimal solution ("near miss") we have to consider only about $2-3\%$ of all vertices. Thus the running time of the search algorithm is output sensitive in this sense. Note that the worst case of SEARCH3 is again very close to the average case, and lies around 10% for Hamming distances between 20 and 150. This reflects the size of the "cluster balls" we have generated for the special sets. If we would use more and thus smaller groups this would improve the worst case. For more details on these effects see Section 3.3.

We conclude that SEARCH3 is the most promising search algorithm and in fact we will perform all experiments of the remaining subsections with this algorithm only.

3.2 Impact of different Clustering Methods

In this section we examine the impact of the different hierarchical clustering methods of [1] for the Algorithm SEARCH3. To this end we consider the same vertex sets, random and special, as in Section 3.1. All hierarchical clustering methods mentioned in [1] were examined and we will use the same notation given there.

Let us first give a brief explanation of the algorithms we will mention explicitly. SLINK computes the well known single linkage clustering (S-link) which was first suggested by Florek et al. [5] and later on by various other authors. The first efficient general implementation of S-link was given by Sibson [10]. CLINK1 computes the complete linkage clustering (C-link) which minimizes the diameter of the subclusters, and was first mentioned by Sørensen [11]. MINRADI is a centroidal clustering method and minimizes the radius of the cluster. The centroid of the cluster has to be contained in the cluster and is thus called its kernel. All these methods are agglomerative, i.e. build the cluster tree bottom up. In contrast, the two algorithms DIVIDIAM and DIVIDIA2 are divisive hierarchical cluster methods which locally minimize the cluster diameters in a top down like manner. While DIVIDIAM stops for one division step after determining the minimum diameter of the subclusters, DIVIDIA2 uses a post-heuristic to improve the quality of the resulting clusters. For our use of the cluster trees all other hierarchical methods of [1] turned out be very similar to some of the above methods and thus we will not mention them explicitly.

For random point sets all clustering methods except SLINK gave very similar results. Figure 9(a) gives the typical curves for MINRADI which was the best of all methods. The

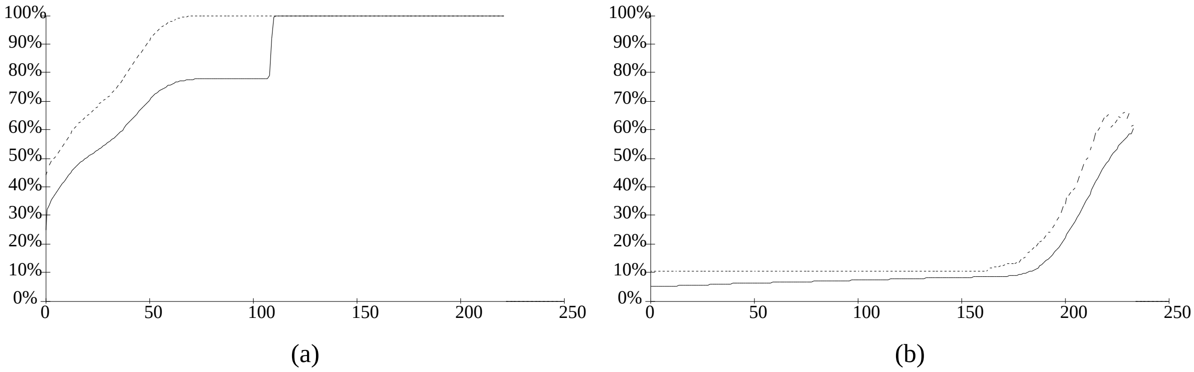


Figure 10: Percentage of considered vertices (analogous Fig. 7) for Algorithm SEARCH3 and hierarchical cluster method S-link.

curves of the other methods, except SLINK, all have the same structure, but are slightly varying in absolute values. For example the two divisive methods DIVIDIAM and DIVIDIA2 are the worst.

Figure 10(a) shows the curve for S-link which is quite different. Despite the fact that for small Hamming distances there are much more vertices considered, it has the best average behavior for a Hamming distance between 50 and 100 bit. An intuitive explanation of this characteristic and the sharp bend for a Hamming distance of around 110 bit may be the fact that S-link is a subgraph of the minimum spanning tree. Thus for small clusters the chaining effect of S-link prevents $r(c_i)$ to be sufficiently small. For larger clusters S-link maximizes the inter cluster distance and thus probably also $\overline{r}(c_i)$. Therefore inequalities (1b) and (2b) of Observation 1 may take effect. But when the Hamming distance exceeds half the length of the longest edges of the minimum spanning tree, the query vertex lies somewhere in between two kernels and thus no subclusters can be skipped. Clearly this explanations only hold for the average case and thus there is a big gap between average and worst case, which has not been observed for any other cluster method.

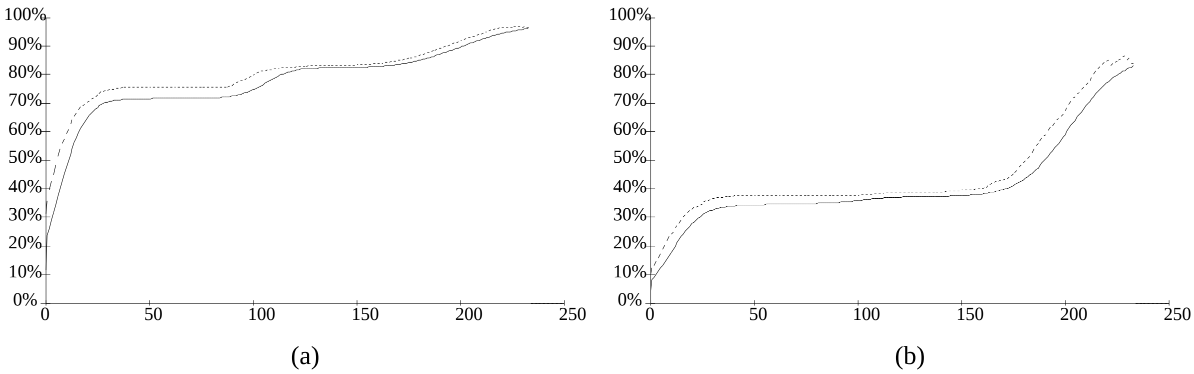


Figure 11: Percentage of considered vertices using SEARCH3 for special vertex sets. The cluster tree was build by the two divisive algorithms (a) DIVIDIAM, and (b) DIVIDIA2.

For the mentioned special sets there is a clear distinction between agglomerative and divisive hierarchical cluster methods. While agglomerative methods all showed the very pleasant behavior discussed in the last section, the two divisive methods DIVIDIAM (cf. Fig. 11(a)) and DIVIDIA2 (cf. Fig. 11(b)) poorly support our search algorithms.

All remaining methods have curves similar to the curve of S-link given in Figure 10(b). Only if the Hamming distance is rather small, i.e. less than 5% of the dimension, then C-link showed an interesting variation, see Figure 9(b). For query vertices which have a very similar optimal vertex, SEARCH3 needs only to consider $2-3\%$ of the whole vertex set. If the query even gives a "perfect hit" (Hamming distance 0) then only approx. 0.8% of the vertices have to be considered in the average. For the first 5% of the curve this relation seems to increase nearly linear for C-link, see also the next section.

We conclude that there are three especially suitable hierarchical cluster methods. The first is MINRADI which is the best for random vertex sets and behaves well for special sets, too. The second is C-link which is the best for special sets and "near-hit" queries, but may also be used for random sets. S-link is suitable for special sets and also for random vertices, if the queries are expected to be "not to close" to some vertex of the set. The additional advantage of S-link is its simple and thus efficient computation [1].

3.3 Further Investigations

In addition to the experiments of the last to subsections we investigated a large number of varying vertex sets. In this subsection we summarize the most interesting observations.

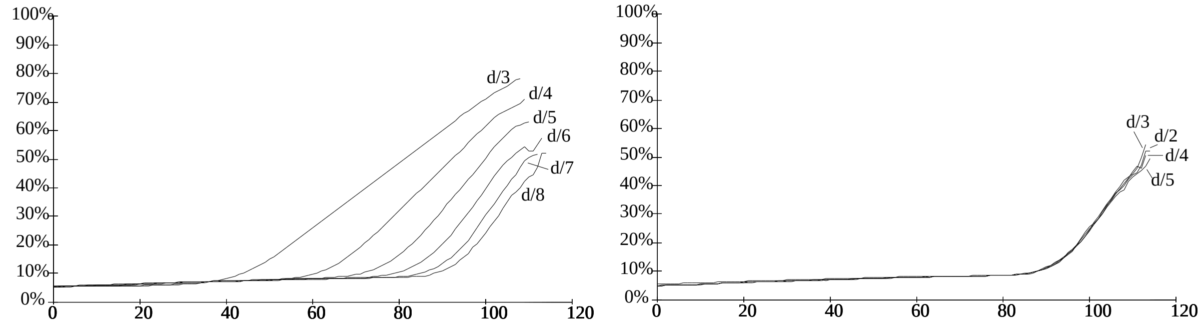
To strengthen the results for our special vertex sets with 10 cluster centers we varied the radius of the "cluster balls" and the minimum distance between the cluster centers. Figure 12 shows the results for a set of $n = 5000$ vertices of dimension $d = 256$ clustered by S-link. For part (a) the distance between the cluster centers is at least $\frac{d}{2}$ and the radius of the cluster balls around each center varies from $\frac{d}{8}$ to $\frac{d}{3}$. Clearly the average number of vertex comparisons increases with increasing radius, but even for a radius of $\frac{d}{3}$ the results are still satisfactory. Note that in this case there are already vertices which might belong to different cluster centers, i.e. the intersection of the cluster balls is not longer empty.

For Figure 12(b) we fixed the radius of the cluster balls with $\frac{d}{8}$, and the minimum distance between the cluster centers varied from $\frac{d}{5}$ to $\frac{d}{2}$. The resulting four curves show only minor differences. This reflects the fact that by choosing the cluster centers randomly the pairwise distance is rarely much less than $\frac{d}{2}$.

Next we considered a very large set with $n = 100000$ vertices of dimension $d = 512$ and $k = 16$ cluster centers. We used S-link to cluster this set and it can be seen from Figure 13(a) that Algorithm SEARCH3 still performs excellently for such large sets. Again the worst case roughly reflects the cardinality of the cluster balls.

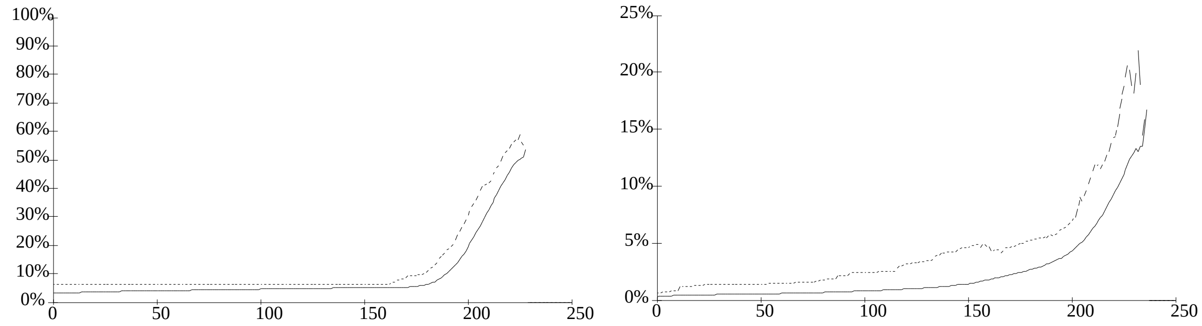
This immediately leads to an idea for another special vertex set. We randomly chose 10 cluster centers with minimum pairwise distance $\frac{d}{2}$. Within a radius of $\frac{d}{8}$ around each of these centers we again chose 10 subcluster centers. And finally we put a third layer of sub-subcluster centers within a radius of $\frac{d}{16}$ around each subcluster center. Note, however, that all these centers do not yet belong to our vertex set. To generate this set we took 10000 vertices uniformly distributed within a radius of $\frac{d}{32}$ around the sub-subcluster centers. To cluster this set we used C-link. Clearly the special structure of these points is very well suited for clustering and thus we expect it to be a very useful set for Algorithm SEARCH3.

Figure 13(b) shows that this is indeed the case. Note the different scaling of the y-axis. In the average case we have to consider less than 0.8% of the vertex set, if the Hamming distance



(a) (b)

Figure 12: Average percentage of considered vertices using SEARCH3 and S-link for special vertex sets with 10 "cluster balls", $d = 256$ and $n = 5000$. (a) Distance between cluster centers $\geq \frac{d}{2}$ and radius of cluster balls varying from $\frac{d}{8}$ to $\frac{d}{3}$. (b) Minimum distance between cluster centers varying from $\frac{d}{5}$ to $\frac{d}{2}$ with fixed radius $\frac{d}{8}$.



(a) (b)

Figure 13: Percentage of considered vertices for SEARCH3. (a) Large set with $n = 100000$ vertices and $k = 16$ "cluster balls", $d = 512$, clustered with S-link. (b) Special set with three layers of 10 kernels each, $n = 10000$, $d = 512$, clustered with C-link.

between query vertex and best vertex is less than 100 (which seems to be quite large). If the query is a "near hit" (say at most 4 bit difference) there are about $\frac{1}{3}\%$ ($\frac{2}{3}\%$) of the vertices to be considered in the average (worst) case.

Clearly we do not expect this set to appear in real world problems, but it points out the capability of our approach to adapt to specially structured vertex sets.

For the vertex sets of Section 3.1 Figure 14 gives the results for "near hit" queries of up to 25 bit difference (about 5% of the dimension). As already mentioned in the last sections this shows the good performance of SEARCH3 for this kind of queries. The random set was clustered by MINRADI and the special set was clustered by CLINK1. In the last section these two methods turned out to be most suitable for the given sets.

For completeness we also considered lower dimensional vertex sets. Figure 15 gives examples for two random sets with $d = 16$, $n = 200$, 5000, 30000, and $d = 32$, $n = 200$, 5000, 20000, respectively. From these curves we see that if the number of vertices is rather small then the percentage of considered vertices is higher, but the absolute number of vertices is still small.

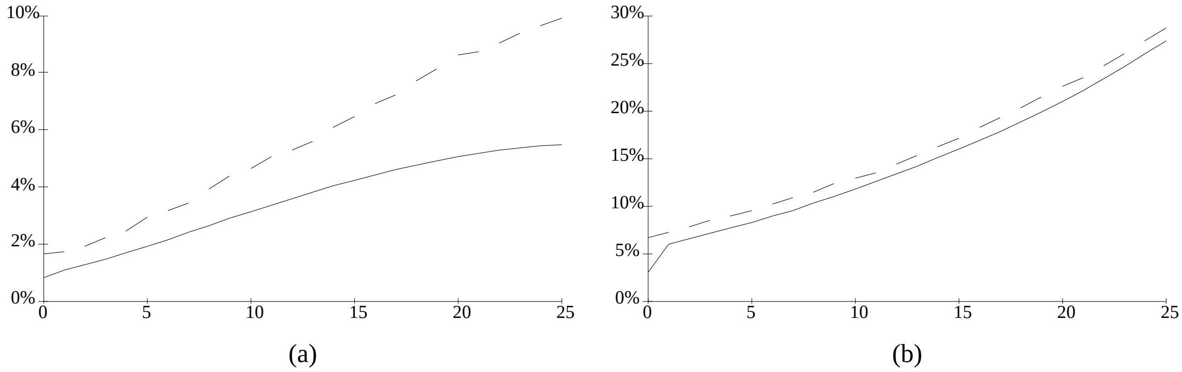


Figure 14: Percentage of considered vertices for SEARCH3 for "near hit" queries, $n = 10000$, $d = 512$. (a) Special set with 10 cluster balls, clustered with CLINK1. (b) Uniformly distributed random vertices, clustered with MINRADI.

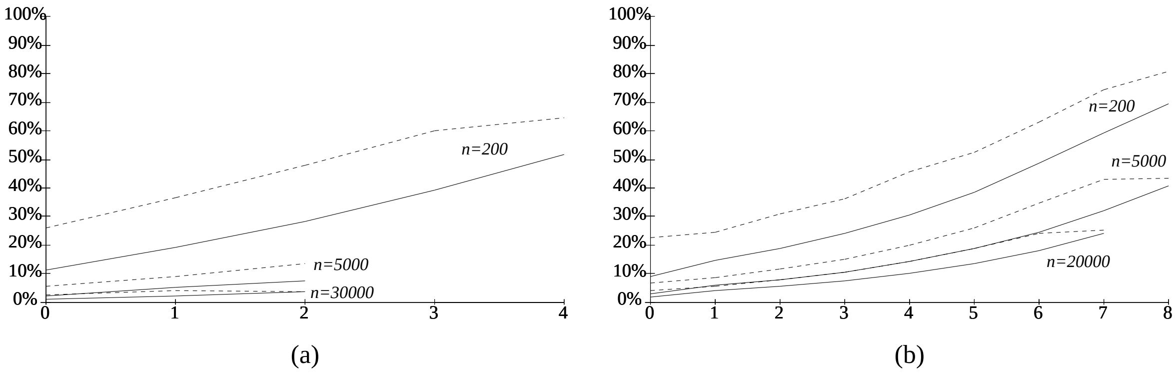


Figure 15: Percentage of considered vertices for SEARCH3 for low dimensional random vertex sets. (a) $d = 16$, $n = 200$, 5000, 30000 clustered with CLINK1, and (b) $d = 32$, $n = 200$, 5000, 20000 clustered with MINRADI.

Contrarily if the number of vertices is large then SEARCH3 terminates very quickly. For example consider the set with $d = 16$ and $n = 30000$, i.e. nearly half of the vertices of $\mathcal{C}^{16}$ are in the set. Then the optimum solution most likely has a Hamming distance of 0 or 1 (only 3 out of 100000 queries had a Hamming distance of 2). The experiments showed that in the average case less than 1.45% of the vertices of this set had to be considered.

Considering the hierarchical structure of the cluster trees, a question which suggests itself is whether we can utilize this for very huge vertex sets which do not entirely fit into the main memory. The obvious approach is to keep only the "upper" part of the cluster tree in memory. To this end we indicate every kernel by its level in the tree, where, as usual, the level of a vertex is its distance to the root. We then keep all vertices of level less than or equal to some fixed k in memory. If we start to search for a vertex then the first comparisons can obviously be done in memory. The hope is now that the search can skip most subtrees and thus avoids to branch into deeper levels.

Unfortunately, experiments showed that this approach does not really work. Figure 16 gives the percentage of internal comparisons (solid line) and of vertices to keep in memory (dashed line) against the number k of levels stored in the main memory. We used a set with $n = 10000$

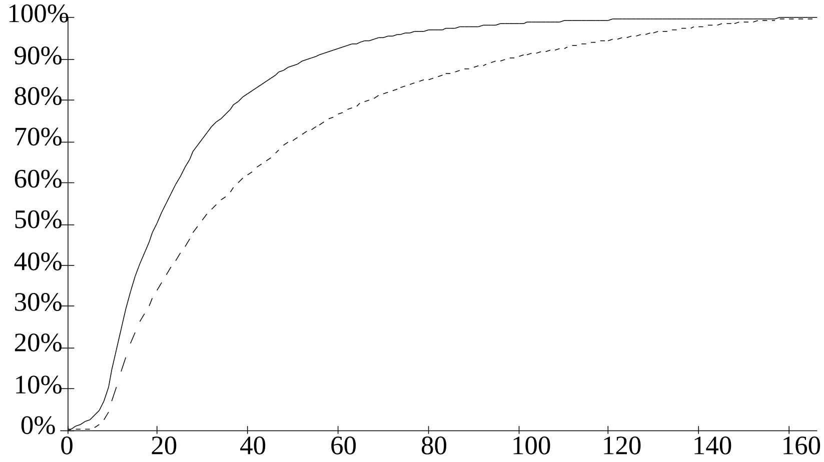


Figure 16: Percentage of internal comparisons (solid line) and of internal vertices (dashed line) over the number of stored levels.

vertices of dimension $d = 512$ and 10 kernels, clustered by C-link. The resulting cluster tree had 166 levels. The queries where chosen to guarantee an optimal Hamming distance in the range $\{0, \ldots, 25\}$. The curves show that the number of internal comparisons for the mentioned approach is at best only about 20% higher than the share of vertices in memory. For arbitrary query vertices the situation would be even worse, and for random point sets the two curves for internal comparisons and the number of vertices to keep in memory are almost identical.

4 Conclusion and Open Problems

In this paper we gave a new approach to the $\{0, 1\}$-string search problem, based on hierarchical clustering. We gave simple observations on the relation of intra- and inter-cluster distances and showed how to utilize them for string searching. In Section 2.2 we presented three different search algorithms which were exhaustively examined in Section 3. The experiments showed that Algorithm SEARCH3 is the most promising method and Section 3.2 gives an overview of suitable hierarchical clustering methods. Moreover the curves of Section 3 show that SEARCH3 performs quite well if either the distance between the query vertex and the best solution is not too large or the vertex set consists of "well grouped" vertices. In both cases we only need to consider a small percentage of the vertex set.

There are two open questions related to the presented approach:

- Is there a search algorithm which performs similar to SEARCH3 but needs less preprocessing? If there is a large number of queries then the preprocessing for SEARCH3 is acceptable. But if there is only a limited number of queries then especially the hierarchical clustering methods might be to time consuming.
- Does there exists an algorithm for the $\{0, 1\}$-string search problem which is guaranteed to run in $o(dn)$ time and $O(dn)$ space for any vertex distribution and any query vertex, similar to what binary bisection search does in the one dimensional case?

Acknowledgements We would like to thank Franz Aurenhammer for bringing this problem to our attention as well as for several discussions and helpful comments on the presented topic.

References

[1] O. Aichholzer, *Clustering the Hypercube*, SFB Report Series 93, TU-Graz, 1996

[2] H. Alt, personal communication, 1995

[3] F. Aurenhammer, *Voronoi diagrams - a survey of a fundamental geometric data structure*, ACM Computing Surveys 23 (1991), pp 345–405

[4] W.A. Burkhard, R.M. Keller, *Some approaches to best-match file searching*, Comm. Ass. Comput. Mach. 16, 1973, pp 230-236

[5] 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

[6] Z. Füredi, *Random Polytopes in the d-Dimensional Cube*, Discrete & Computational Geometry 1, 1986, pp 315-319

[7] M. Minsky, S. Papert, *Perceptrons: an introduction to computational geometry*, The MIT Press, Cambridge, 1969

[8] V. Ramasubramanian, K.K. Paliwal, *An efficient approximation-elimination algorithm for fast nearest-neighbour search based on a spherical distance coordinate formulation*, Pattern Recognition Letters 13, 1992, pp 471-480

[9] R.L. Rivest, *On the optimality of Elias's algorithm for performing best-match searches*, Information Processing, North Holland Publishing Company, 1974, pp 678-681

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

[11] 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

[12] E. Vidal, *An algorithm for finding nearest neighbours in (approximately) constant average time*, Pattern Recognition Letters 4, 1986, pp 145-157

[13] E. Vidal, *New formulation and improvements of the Nearest-Neighbour Approximating and Eliminating Search Algorithm (AESA)*, Pattern Recognition Letters 15, 1994, pp 1-7

[14] J.M. Vilar, *Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm*, Information Processing Letters 56, 1995, pp 265-271

[15] T. Welch, *Bounds on the information retrieval efficiency of static file structures*, Technical Report 88, MIT, 1971

[16] G.M. Ziegler, *Lectures on Polytopes*, Graduate Texts in Mathematics 152, Springer-Verlag, 1994