Int. Jnl. Multimedia Information Retrieval, IJMIR, vol. 2, no. 4, 2013.
*The final publication is available at Springer via http://dx.doi.org/*10.1007/s13735-013-0046-4

Very Large Scale Nearest Neighbor Search: Ideas, Strategies and Challenges

Erik Gast, Ard Oerlemans, Michael S. Lew

Leiden University

Niels Bohrweg 1

2333CA Leiden

The Netherlands

{gast, aoerlema, mlew} @liacs.nl

Abstract

Web-scale databases and Big Data collections are computationally challenging to analyze and search. Similarity or more precisely nearest neighbor searches are thus crucial in the analysis, indexing and utilization of these massive multimedia databases. In this work, we begin by reviewing the top approaches from the research literature in the past decade. Furthermore, we evaluate the scalability and computational complexity as the feature complexity and database size vary. For the experiments we used two different data sets with different dimensionalities. The results reveal interesting insights regarding the index structures and their behavior when the data set size is increased. We also summarized the ideas, strategies and challenges for the future.

1. Introduction

Very large scale multimedia databases are becoming common and thus searching within them has become more important. Clearly, one of the most frequently used searching paradigms is k-nearest neighbor(k-NN), where the k objects that are most similar to the query are retrieved. Unfortunately, this k-NN search is also a very expensive operation. In order to do a k-NN search efficiently, it is important to have an index structure that can efficiently handle k-NN searches on large databases. From a theoretical and technical point of view, finding the k nearest neighbors is less than linear time is challenging and largely

unsolved. Also implementing the structure can be a challenge because of memory, CPU and disk access time restrictions. Various high-dimensional index structures have been proposed trying to solve these challenges. Because of the number of different indexing structures and the big differences in databases, it is hard to determine how well different index structures perform on real-life databases, especially when doing a k-nearest neighbor search.

Similarity searches on low-dimensional features have been reported from the research literature to work very well, but it is still unclear under what conditions they give superior performance. This phenomenon is called the 'curse of dimensionality' and is caused by the fact that volume increases exponentially when a dimension is added. Intuitively, increasing a hypersphere just slightly in high-dimensional space, the volume of the sphere will increase significantly. For nearest neighbor searching, this is a problem, because with high dimensional data, it will look like the distance between the points in this high-dimensional space and the query point all have the same distance. This will result in a search space (sphere) around the query which is so large that it will capture all the points in space.

In this paper we investigate the performance of important index structures when doing *k-nearest neighbor search*. For the experiments we use the MIRFLICKR [1] database, which consists of one million images that were extracted from the Flicker website (www.flickr.com). Two different MPEG7 image descriptors were extracted and used for testing.

The paper is organized as follows: In section 2 we discuss a number of important index structures and we give a formal description of *k-nearest neighbor search* and we describe the 'curse of dimensionality'. In section 3 we give a more detailed description of the methods we have tested. In section 4, the experiments are described and the results are given. The results are discussed in section 5 and we conclude with challenges for the future.

2. Related Work

There are two main types of indexing structures; data-partitioning and space-partitioning methods. Data-partitioning methods divide the data space according to their distributions. Many of the data-partitioning indexing structures are derivatives of the R-tree [2]. Space-partitioning methods divide the data space according to their location in space. Index structures that use a space-partitioning method are often similar to KD-trees.

The R^*-tree [3] and X-tree [4] are both variants of the R-tree and are designed to handle multi-dimensional data. They do work well on low-dimensional data but their performance deteriorates quickly when

dimension increases. This is due to the 'curse of dimensionality' (section 2.1). The SS-tree [5] is a R-tree like structure that uses *minimum bounding spheres* instead of *bounding rectangles*. The SS-tree outperforms the R^*-tree, but high-dimensional data is still a problem. To overcome the problem that *bounding spheres* occupy more volume with high-dimensional data than the bounding rectangles do (the problem of which the SS-tree suffer from), the SR-tree [6] integrates *bounding spheres* and *bounding rectangles* into the structure. This increased performance in high-dimensional space. Another option which has been explored is to use Voronoi clusters for the partitioning [10].

In [7] *Henrich et al.* proposed the LSD-tree and it was later improved to become the LSD^h-tree [8]. The LSD-tree and the LSD^h-tree are both space-partitioning structures similar to that of the KD-tree. With the LSD^h-tree they combined the KD-tree with a R-tree to reduce the empty spaces and keep low fan-out. Their results showed that the LSD^h-tree reduces the fan-out and that the tree is independent of the number of dimensions but only as long as the distribution characteristics of the data allows for efficient query processing.

K. Chakrabarti & S. Mehrotra introduced the Hybrid-tree [9] which combines the advantages of space-partitioning and data-partitioning structures. The Hybrid-tree guaranties that the tree is dimension independent so that it is scalable to high-dimensions. The SH-tree (Super Hybrid-tree) [11] is also a combination of a space and data-partitioning structure. The SH-tree combines a SR-tree and a kd-based structure. They were unable to compare the performance of the SH-tree to other indexing structures.

In [12] the Pyramid Technique is introduced and is based on a mapping of high-dimension space to 1-dimension keys. A B^+-tree is used to index the 1-dimensional keys. The basic idea is to divide the data space such that the resulting partitions are shaped like peels of an onion. The *d*-dimensional space is divided in 2*d* pyramids with the center point of the space as their top. Then the pyramids are cut into slices which form the data pages. The Pyramid Technique outperformed both the X-tree and the Hilbert R-tree [13]. The NB-tree [14] also maps the high-dimensional data to a 1-dimensional key and uses the B^+-tree to index them. The index key is the Euclidian distance of a *d*-dimensional point to the center. Their results showed that their NB-tree outperformed the Pyramid Technique and the SR-tree, it did also scale better with growing dimensionality and data set size. *J. Cui et al.* introduced pcDistance [15] which is also a 1-dimensional mapping method. First, data partitions are found in the data set and Principal Component Analysis [16] is applied. The distance of a point to the center of its partition is used as key and is indexed using a B^+-tree. To improve the query performance, the principal component is used to filter the *nearest neighbor* candidates. Their results show that pcDistance outperforms iDistance [17] and the NB-Tree.

R. Weber et al. proposed the VA-file [18]. The VA-file tries to overcome the 'curse of dimensionality' by applying a filter-based approach instead of the conventional indexing methods. The VA-file keeps two files; one with approximations of the data points and another with the exact representations. You can see the approximation file as a (lossy) compressed file of data points. The approximation file is sequentially scanned to filter out possible *nearest neighbor* candidates and the exact representation file is used to find the exact *nearest neighbors*. The VA-file outperforms both the R^*-tree and the X-tree. Researchers have also looked at combining VA-file with partial linear scan [22] which has the advantage of a linear scan but avoids scanning the entire database by using 1D mapping values. An improvement of the VA-file is the VA^+-file [19]. The VA^+- improves the performance on non-uniformly distributed data sets by using *principal component analysis* and using a non-uniform bit allocation. They also integrated approximate *k*-NN searches.

2.1. Curse of Dimensionality

Similarity searches on low-dimensional generally work very well, but when the dimensionality increases the performance can degrade badly. This phenomenon is called the 'curse of dimensionality' [20] and is caused by the fact that volume increases exponentially when a dimension is added. This also that when even increasing a hypersphere just slightly in high-dimensional space, the volume of the sphere will increase enormously. For *nearest neighbor searching*, this is a problem, because with high dimensional data, it will look like that the distance between the points in this high-dimensional space and the query point all have the same distance. This will result in search space(sphere) around the query which is so large that it will capture all the points in space. Also when the radius of the sphere increases, the volume of the sphere will grow resulting in a lot of empty space inside the sphere.

2.2. k-Nearest Neighbor Search

k-Nearest neighbor (k-NN) searches for the *k* objects that are closest to the query. A more formal description would be: Given a *d*-dimensional dataset DB in the data space D and a query $q \in D$, to find the set $S = \{s_1, s_2, \dots, s_k\}$ with the *k* elements most similar to q the following must hold.

$$\forall s_i \in S \; \forall b_j \in DB - S, dist(q, s_i) \leq dist(q, b_j) \tag{1}$$

3. Indexing Structures

In this section, we give a more detailed description of the index structures we have used. Please note that we have not used recent approximate nearest neighbor methods such as [10, 23] because in this evaluation we are examining exact nearest neighbors only. We have tested five different structures which use a data-partitioning or space-partitioning method or use both.

3.1. NB-Tree

The NB-Tree (Normalized B^+-Tree) [14] is an index structure where the *d*-dimensional data points are mapped to a 1-dimensional value. To map the *d*-dimensional data the Euclidian norm (2) is used. The values resulting from the dimension reduction can be ordered and be used for searching. The *d*-dimensional data points are inserted into a B^+-Tree with their Euclidian norm as index key. After the insertion, the data points inside the leaves of the B^+-Tree will be ordered according their Euclidian norm values. An example of this dimension reduction for 2D points is shown in Figure 1.

$$\|P\| = \sqrt{{p_1}^2 + {p_2}^2 + \cdots + {p_d}^2} \tag{2}$$

The basic idea of the k-NN search algorithm of the NB-Tree is to find an initial starting location and then gradually increasing the radius *r* of a search sphere until the k nearest neighbors are found. Increasing the radius in the case of the NB-Tree, is searching along the leaves of the B^+-Tree. This is possible because the leaves are ordered. The first initial starting location is acquired by locating the leaf with the key that is near or equal to the Euclidian norm of the query. Now, because the leaves are linked sequentially, the leaves can be navigated freely. Increasing the search sphere is nothing more than navigating the leaves to the right and left. Increasing the sphere with *r* is the same as finding the leaves (points) to the right that have a key index $KeyIndex \leq \|q\| + r$ and finding the leaves on the left that have $KeyIndex \geq \|q\| - r$. During the visiting of the leave nodes, the distance between the query and the point at the leave node is computed. If the distance is smaller than the farthest current near neighbor then the point is stored in a list. The radius of the sphere is increased until there are k points in the list and when the distance between the *k*th nearest point and the query is less than the radius of the sphere. The pseudo-code of the algorithm is shown in Algorithm 1.

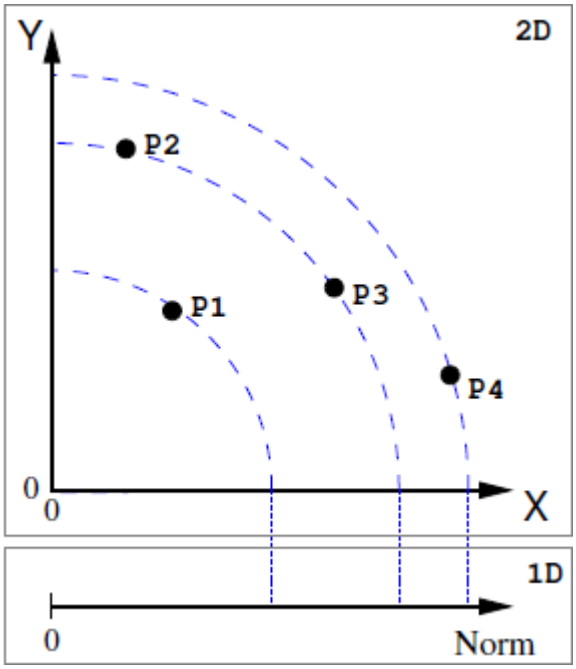


Figure 1: NB-Tree dimension reduction.

Algorithm 1: k-NN search algorithm for the NB-Tree.

```
1.  leaf = searchB+Tree(dist(q))
2.  Do
3.     leaf = leaf.PreviousSearchToTheRight
4.     upperLimit = upperLimit + delta
5.     While(leaf.key <= upperLimit)
6.        If(dist(leaf.point, q) < list.LastElement Or list.size < k)
7.           list.Insert(leaf.point)
8.        End If
9.        leaf = leaf.right
10.    End While
11.    leaf = leaf.PreviousSearchToTheLeft
12.    lowerLimit = lowerLimit + delta
13.    While(leaf.key >= lowerLimit)
14.       If(dist(leaf.point, q) < list.LastElement Or list.size < k)
15.          list.Insert(leaf.point)
16.       End If
17.       leaf = leaf.left
18.    End While
19. While(dist(list.LastElement(), q) > radius)
20. kNN = list[0, k-1]
```

3.2. pcDistance

pcDistance [15] is a method that is similar to the iDistance [17] method. It also maps, just like the iDistance and the NB-Tree, *d*-dimensional points to a 1-dimesional value. In iDistance, the dataset is partitioned and a reference point for each partition is defined. The reference points are used to calculate the index key for the B^+-Tree. The index key is calculated as follows:

$$IndexKey = i * c + dist(O_i, p) \tag{3}$$

Where i is the partition number of the partition that is closest to point p, c is a constant which is used to stretch the index range so that the indexes inside de B^+-Tree are also ordered based on the partition numbers and $dist(O_i, p)$ is the distance between p and its closest reference point. The main difference between pcDistance and iDistance is that pcDistance uses Principal Component Analysis (PCA) [16] to transform points to PCA space and it uses a filtering algorithm based on the first principal component. The filtering algorithm is used to prune more data points. The idea behind this is that the probability to satisfy both $dist(q_i, p_1) \leq r$ and $dist(O_i, q) - r \leq dist(O_i, p) \leq dist(O, q) + r$ simultaneously is much lower than satisfying $dist(q_i, p_1) \leq r$ or $dist(O_i, q) - r \leq dist(O_i, p) \leq dist(O, q) + r$. Where q is a query point and r is the search radius.

pcDistance uses the same mapping value (3) as index key for the B^+-Tree as iDistance. The only structural difference is that not only the points are stored inside the leaves but also the first principal component of each point. To acquire the partitions, the K-means clustering algorithm [21] is used. The reference points O of the different partitions are the centers of each partition. Furthermore, all the points that are stored inside the tree or used as query are first transformed to PCA space.

The *k*-NN search algorithm in pcDistance and iDistance is similar. The essence of the algorithm (just like the NB-Tree) is to first find an initial location and from there you start searching with a small sphere and incrementally enlarging the sphere until all the *k* nearest neighbors are found. Because (3) mapping is used, the leaf nodes cannot be traversed like in the B^+-Tree algorithm. You have to distinguish searching towards the reference point (inward) and away from the reference point (outward). Because it is possible that the search sphere intersects with multiple partitions, you have to inspect which partition contains the query or intersects with its search sphere. This is done by searching inwards and outwards when the partition contains the query point and search inward when the partition and the search sphere intersect.

When searching inward and outward the points inside the leaves are compared to the query and if the distance is smaller than the *k*th point in the *k*NN list, it is added to this list. To reduce the distance calculation between the query and the points, pcDistance filters the points using the first principal component. Only when (4) and (5) hold, then the distance between the point and the query must be calculated.

$$\max(w', pcMin) \leq p_1 \leq \min(v', pcMax) \tag{4}$$

$$\max(q_1 - r, pcMin) \leq p_1 \leq \min(q_1 + r, pcMax) \tag{5}$$

Where p_1 is the first principal component of point p, $pcMin$ and $pcMax$ are respectively the minimum and maximum values of the first principal component of the whole partition and w' and v' are the projections of the intersection between the partition and the search sphere onto the first principal component. An example intersection is shown in Figure 2.

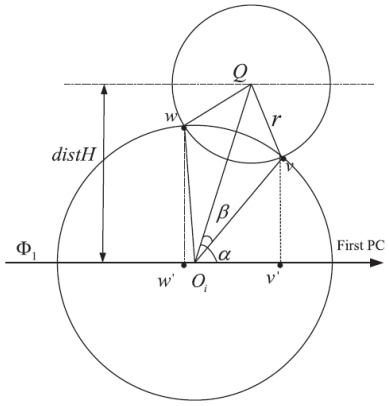


Figure 2: The spatial relationship between the query and the partition.

One thing I should note, something not mentioned in the original article, is that you should be careful when applying the pcDistance filter if the search radius is larger than the partition radius. It could happen that the partition is completely inside the search sphere (Figure 3, left) and therefore there won't be any sphere intersection points to calculate w' and v'. It is also possible that the partition is partially 'swallowed' by the search sphere (Figure 3, right) which results in intersection points that cannot be used to filter using the first principal component. This larger search sphere compared to the partition size is almost always due to a bad *number of object/number of partitions* ratio.

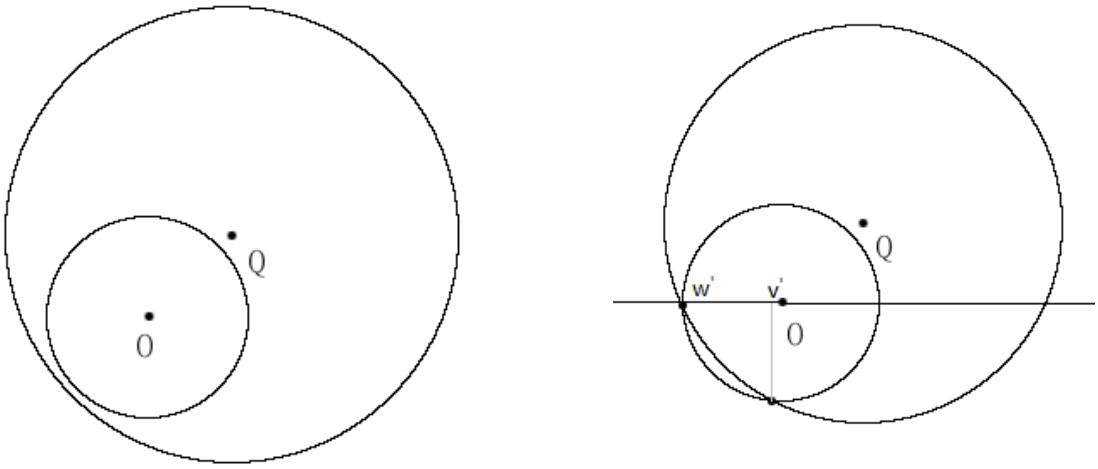


Figure 3: Examples when the pcDistance filter fails.

3.3. LSD^h-Tree

The LSD^h-tree [8] is an extension of the LSD-tree [7] and both are kd-tree-based access structures. Kd-trees are space partitioning data structures where the whole space is divided in subspaces. Kd-tree-based access structures, unlike most other access structures ([2],[3] ,[4]), have the nice property that there is no overlap between different nodes (space regions). Also the LSD^h-Tree divides the data space into pair wise disjoint data cells. With every data cell, a bucket of fixed size is associated where all the objects that the cell contains are stored. When a bucket has reached its capacity and another objected has to be inserted into this bucket, then the bucket is split and the objects in the bucket are distributed equally among the bucket and its bucket sibling (if present). Before the bucket can be split, a split dimension has to be computed. The LSD^h-Tree uses a *data dependent split strategy* i.e. a split dimension and value is chosen based on only the objects stored in the bucket which has to be split. If there are different feature values for dimension $(d_{old} + i)\ mod\ t$ in this bucket, use dimension $(d_{old} + i)\ mod\ t$ as split dimension. Otherwise increase i by 1 until there are different feature values for the dimension. d_{old} is the split dimension used in the node referencing the bucket to be split. The new split value is computed by taking the average of the values in the new dimension of the objects. To avoid bucket splits, objects in an full bucket can also be redistributed. If there is a sibling bucket that is not yet full, one object is shifted to that sibling.

The LSD^h-Tree also stores so-called *coded actual data region* (cadr) in the nodes. This is an approximation of the *actual data region* which is the minimal bounding rectangle containing all points stored in the bucket. Using the *coded actual data region* the amount of storage space for coding the region can be reduced. Using the *coded actual data region* instead of the *actual data region* reduces the size of

the tree and therefore the fan-out. In our implementation we don't use *coded actual data region* but the *actual data region* because we load the whole tree in the memory and the memory size is big enough to hold the structure.

The *k*-NN search algorithm for the LSD^h-Tree works as follows: First, the bucket that contains (or could contain) the query point is searched for. During the search, when a left child is followed the right child is inserted in a priority queue NPQ with the distance of the query to the minimal bounding rectangle of the right child as priority. And when the right child is followed then the left child is inserted in the priority queue. When the bucket is found, all the objects inside the bucket are inserted in another priority queue OPQ with their distance to the query as priority. All objects in OPQ that have a smaller or equal distance then the first element of NPQ are the nearest neighbors. Until *k* nearest neighbors are found, the directory or bucket is taken from NPQ and a new search is started as described above. The algorithm stops when *k* nearest neighbors are found or OPQ and NPQ are empty. Pseudo code of the *k*-NN algorithm is shown in Algorithm 2.

One important thing to note is that the order of insertion into the LSD^h-Tree matters and you should never insert an ordered list of objects, because that will result in a very unbalanced tree. Also, because the *k*-NN search algorithm makes use of priority queues, you are bound to the performance of the priority queues. If the LSD^h-Tree is unable to efficiently prune candidates, then a lot of nodes and objects will be inserted into the priority queue which makes the performance deteriorate fast.

Algorithm 2: The *k*-NN algorithm for the LSD^h-Tree.

```
1.  leaf = searchB+Tree(dist(q))
2.  Do
3.      bucket = LSDhTree.Search(q)
4.      For(each e in bucket)
5.          OPQ.Push(e, dist(e, q))
6.      End If
7.      While(OPQ.Size > 0)
8.          e = OPO.Top()
9.          If(dist(e, q) < dist(NPQ.Top(), q))
10.             kNN.Insert(e)
11.             OPO.Pop()
12.     End While
13. While(kNN.Size() < k And (OPQ.Size() > 0 Or NPQ.Size() > 0))
```

3.4. SH-Tree

The SH-Tree [11] is a very complex tree and is a mixture of a SR-Tree [6] and a KD-tree-like structure. The KD-tree-like structure is used to overcome the fan out problem and the SR-Tree is used to keep data clusters together. Because the structure is very complex we will only discus its structure quickly. If you need a more detailed description we refer to [11]. The SH-Tree has three kinds of nodes: Internal, balanced and leaf nodes. The internal nodes are organized as in the Hybrid tree. It is a KD-tree like structure but overlap is possible. Inside the internal nodes a lower and higher boundary is stored to code the overlap of its children. The balanced nodes are similar to the nodes of the SR-Tree. They contain the *bounding sphere* (BS), the *minimum bounding rectangle* (MBR) and pointers to the leaf nodes. Inside the leaf nodes, data objects are stored. The balanced nodes have a minimum and maximum number of entries (leaf nodes) which can be stored inside the node. Also the leaf nodes have a minimum and maximum number of objects that can be stored. A possible SH-Tree structure can be found in Figure 4. When a leaf node is full and an object needs to be stored inside this full leaf, then there are three possibilities. First, if the leaf has not yet been reinserted into the tree, then a part of the leaf is reinserted into the tree. Second, if reinsertion is not possible, try to redistribute one data object to a leaf sibling. Third, if the reinsertion and redistribution are not possible then the leaf node is split. A split position is chosen and the BS and the MBR of the balanced node is adjusted. If a balanced node is full (e.g. because of a leaf split) then, if possible, an entry of the full balanced node is shifted to a balanced node sibling. If this is not possible, the balanced node is split and a similar split algorithm to that of the R*-Tree is employed [3].

The writer proposed two different *k*-NN search algorithms for the SH-Tree. The first one implements a depth-first search method and the second algorithm makes use of a priority queue. We use the first algorithm (depth-first search) because it does not have the performance overhead of the priority queue. The algorithm is fairly simple: you do a depth-first search and visit only nodes that have a *bounding rectangle* where a possible nearest neighbor can be found. When a leaf node is reached, all the possible object are inserted into the *k*-NN list, but only if their distance to the query is smaller than the distance of the farthest current neighbor. Pseudo code for both the algorithms can be found in [11].

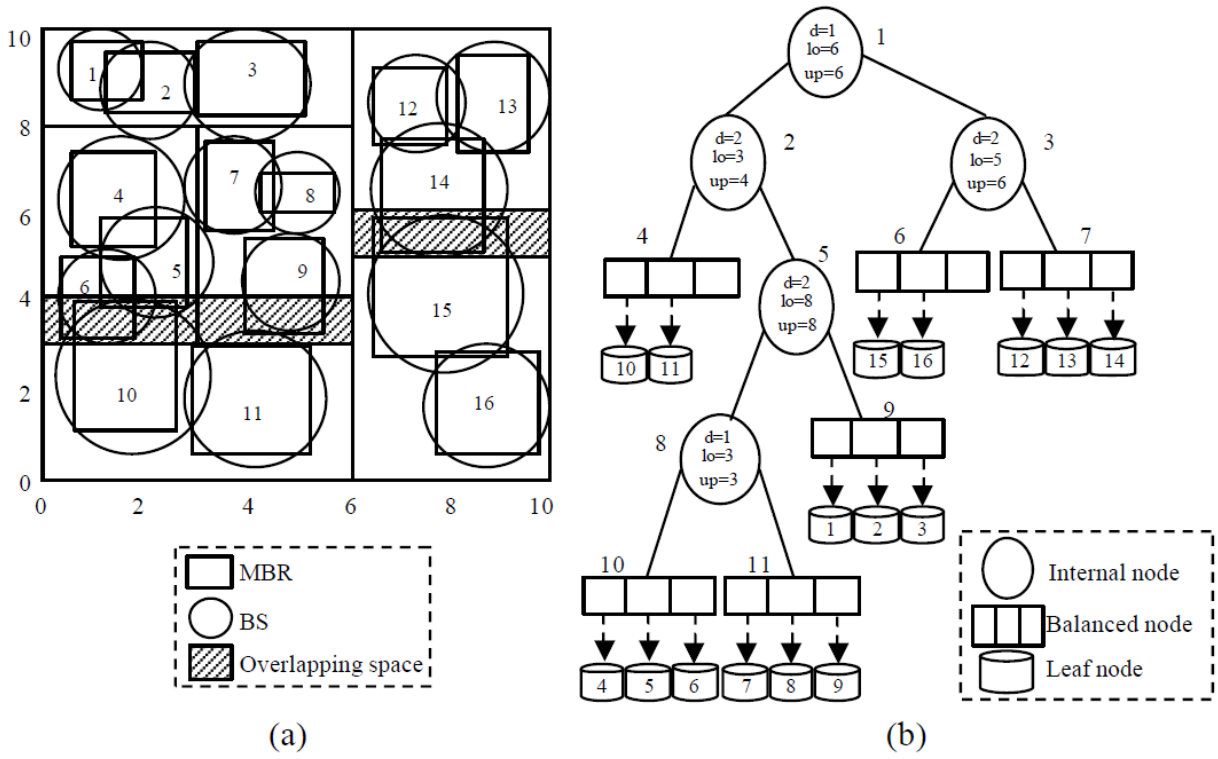


Figure 4: A possible partition of an example data space and the corresponding mapping to the SH-Tree.

3.5. VA-File

The approach of the *vector-approximation file* (VA-file) [18] is different from the other approaches. It does not use a data-partitioning approach, but rather uses a filter-based approach. Vector space is partitioned into cells and these cells are used to generate bit-encoded approximations for each data object. Each encoded object is put into the VA-file which itself is just a single array. Actually, the VA-file is just a (lossy) compressed array of vector objects.

For the construction of the file the number of bits b that is used for encoding one dimension has to be determined. The VA-file will use an even distribution of bits per dimension. Also the partition points of the cell have to be calculated in order to encode the object. We analyze the entire data set to compute the partition points. The approximation a_i of vector v_i is generated as follows. A point that falls into a region numbered $r_{i,j}$ is defined as:

$$p_j[r_{i,j}], = v_{i,j} < p_j[r_{i,j} + 1] \tag{6}$$

Where $r_{i,j}$ is the region number, $v_{i,j}$ is the dimension j of vector i and $p_j[e]$ is the partition point e of dimension j. The generated approximation can be used to derive higher and lower bounds between a query and a object (vector). You can find how this is done in [18].

The *k*-NN search algorithm we use is the *'Near-Optimal' search algorithm* (VA-NOA). This algorithm has two phases. The first phase tries to eliminate as much objects as possible before phase two starts. In the first phase, the whole VA-file is sequentially scanned and the lower and upper bounds of each object is computed. If the lower bound is greater than the farthest nearest neighbor, then it can be eliminated. Otherwise, the lower bound and the approximation are inserted into a priority queue with the lower bound as priority. In the second phase all objects in the priority queue are examined and the real distance between the object and the query is computed. If this distance is smaller than the farthest *k*-nearest neighbor then it is inserted in the nearest neighbor list. Algorithm 3 shows the pseudo code of the search algorithm.

The benefit of the VA-File is that is can effectively eliminate a lot of objects, so that only a few objects have to be retrieved. The drawback of the VA-File is that decoding every approximation and calculating both its lower and upper bounds is computationally expensive.

Algorithm 3: VA-NOA *k*-NN search algorithm for the VA-File.

Phase 1	*Phase 2*
1. ***For each*** *a* ***in*** *vaFile* 2. *lower = LowerBound(a, q)* 3. *upper = UpperBound(a, q)* 4. *d = pq.LastElement().priority* 5. ***If(****lower <= d)* 6. ***If(****upper < d)* ***Or*** *pq.Empty())* 7. *pq.push(a, upper)* 8. ***End If*** 9. *approxPq.Push(a, lower)* 10. ***End If*** 11. ***End For***	1. ***While(****approxPq.Size() > 0)* 2. *v = RealObject(approxPq.Top())* 3. ***If(****dist(v, q) < kNN.FarthersDist()* 4. *kNN.Insert(v)* 5. ***End If*** 6. *approxPq.Pop()* 7. ***End While***

4. Experiments

With our experiments we try to measure how well the different indexing structures perform when the database size increases. As a ground truth we use the results of a linear sequential search method which is a naïve method that is known to degrade gracefully. With all the experiments we measure the averages using two thousand *10-nearest neighbor search*es. The two thousand queries are randomly selected from the database.

We use the Euclidian distance as distance measurement and is defined as follows:

$$dist(Q,P) = \sqrt{\sum_{i=1}^{N}(Q_i - P_i)^2} \tag{7}$$

We measure two different things; the computation time and the feature vector access ratio. The feature vector access ratio is defined as follows:

$$ar = \frac{n_{fv}}{N} \tag{8}$$

Where n_{fv} is the number of feature vectors that has to be accessed during the search and N is the total number of objects stored in the index structure. The access ratio is an important measurement because often the feature vectors are not stored (or only partially) inside the memory but on a hard disk. And because hard disk access is slow, this could have a big influence in on the performance of the index structures.

4.1. Implementation Details

All the structures and algorithms are implemented in C++ and are all implemented from scratch. To improve memory management the Boost C++ library is used and OpenCV is used to calculate PCA and do K-Means clustering. When possible, different components are reused for the different index structures e.g. the exact same B^+-tree is used for the NB-tree and the pcDistance implementation. To more accurately measure CPU performance, the whole structure is loaded into the memory to eliminate influence of the disk IO. The configuration details of the index structures is shown in Table 1.

The experiments are carried out on an Intel Core 2 Quad Q9550 2,83 GHz with 4GB of DDR2 RAM memory. The computer runs Windows 7 64-bit. One thing we should note is that, because our implementations are not optimized for multi-core systems, the program will only run on one core (out of four). This means that it does not use all the computation power of the CPU and running the program on a single-core system might give a better performance.

Index structure	**Properties**	**Values**
NB-tree	*Max. number of childs per node*	*60*
	Delta	*0.01*
pcDistance	*Number partitions*	*64*
	Number of samples for K-Means and PCA	*500000*
	Max. number of childs per node	*60*
LSD^h-tree	*Max. bucket capacity*	*12*
SH-tree	*Max. balanced node capacity*	*4*
	Max. leaf node capacity	*12*
VA-file	*Bits per partition*	*8*

Table 1: Configurations of the different index structures.

4.2. Dataset and Feature Descriptors

The real-life dataset we used in the experiments is the MIRFLICKR dataset. This dataset consists of one million images that are obtained from the Flickr[Error! Bookmark not defined.] website. In our experiments we use three different feature descriptors; *MPEG-7 Edge Histogram* (eh), *MPEG-7 Homogeneous Texture* (ht) descriptors and a set of random feature vectors. Extracting the *edge histogram* from the images results in a 150-dimensional feature vector, the *homogeneous texture* descriptor results in a 43-dimensional feature vector and the random set of feature vectors was also created with 43 dimensions. So, for the experiments three collections of one million feature vectors are used; one with 150-dimensional *edge histogram* features, one with 43-dimensional *homogenous texture* features and one with 43-dimensional *random* feature vectors. The whole dataset along with the extracted *edge histogram* and *homogeneous texture* descriptors are publicly available[1].

[1] http://press.liacs.nl/mirflickr/

4.3. Results

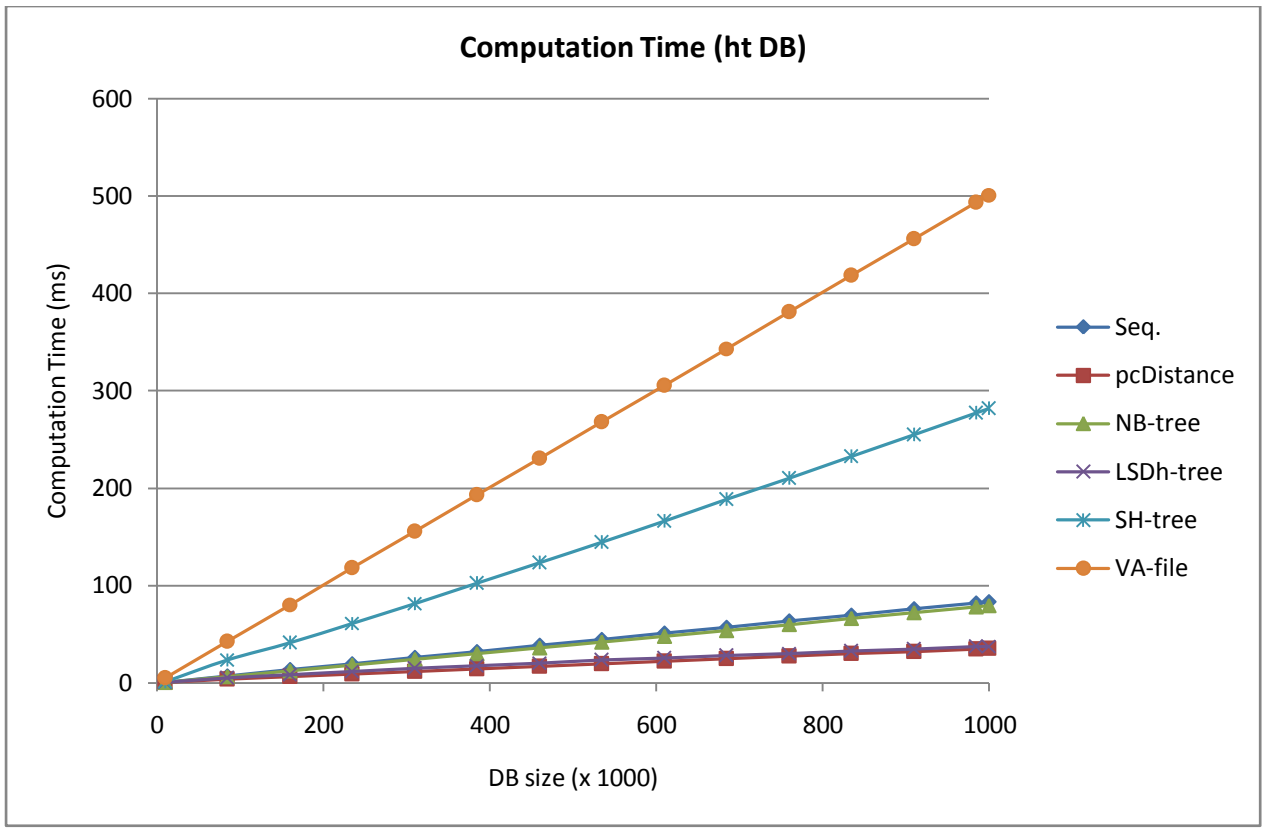


Figure 5: The average computation time of the different index structures with different data set sizes. These are the results of the *homogeneous texture* DB with a dimensionality of 43.

Figure 5 shows the performance comparison between the different index structures when using the *homogeneous texture* feature descriptor database. The figure shows the average computation time of the different index structures to do a *10-nearest neighbor search*. A thing that you will notice is that both pcDistance and the LSD^h-tree outperform the sequential search. They perform both more than 55% better then the sequential search. The NB-tree performs slightly better and the SH-tree and the VA-file perform far worse. The bad performance of the VA-file is due to the fact that is has to 'decode' every approximation vector. But when comparing the access ratio of the structures, the VA-file performs by far the best. When searching in the 1 million sized database it only has to access about 200 feature vectors. The access ratio of the different index structures is shown in Figure 6. Also the pcDistance and the LSD^h-tree perform here realy well compared to the other structures. When the database size is increased, the

access ratio of the pcDistance will converge to about 0.06 and about 0.08 for the LSD^h-tree.

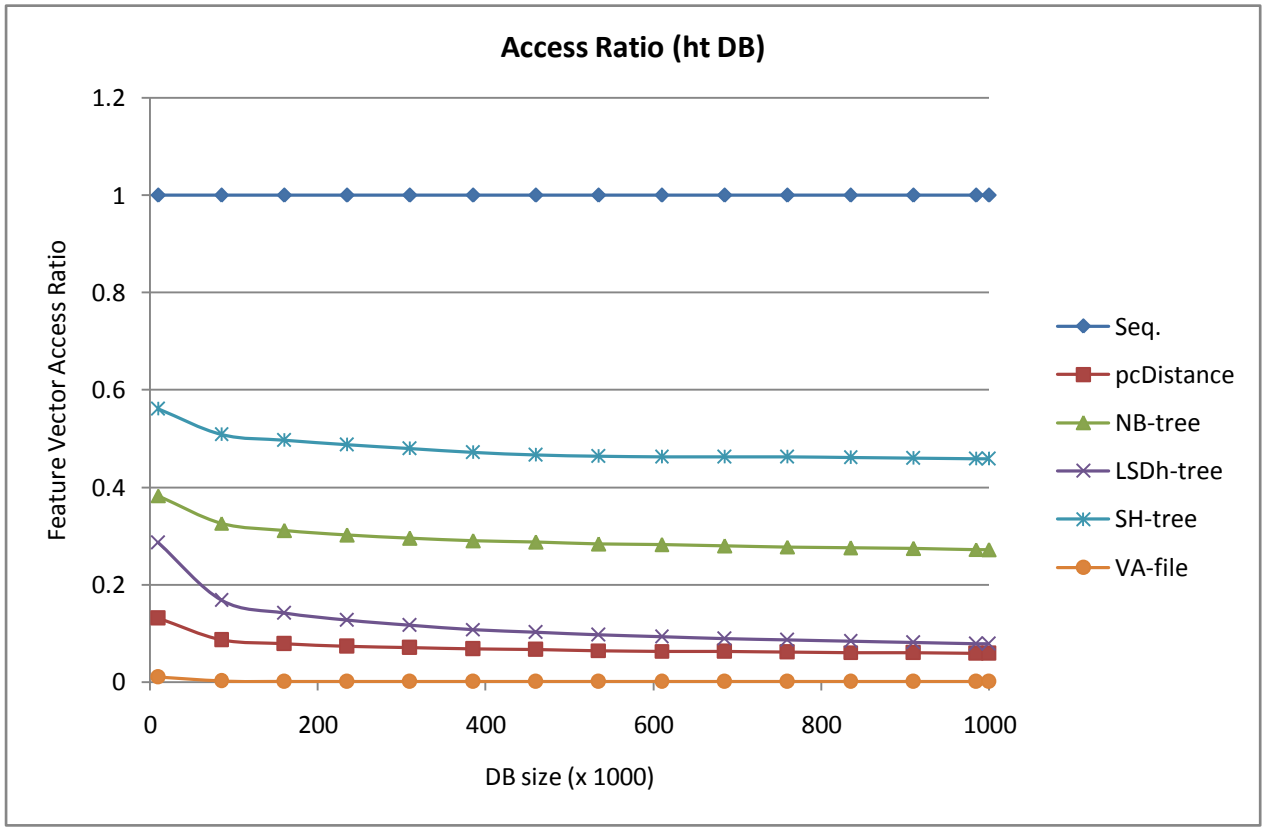


Figure 6: The average access ratio of the different index structures. These are the results of the *homogeneous texture* DB with a dimensionality of 43.

In Figure 7 all the averages and its standard deviations of the computation time and the number feature vector access is shown. Also here you can see that pcDistance and the LSD^h-tree perform similar in computation time and feature vector access and that pcDistance only performs slightly better on average and has a slightly lower standard deviation.

Testing the performance of the index structures on the *150-dimensional* database with *edge histogram* feature vectors yields different results. In Figure 8 the average computation time of the different index structures for the *edge histogram* database is visualized. You notice that compared to the results of the *ht* database, there is no structure that outperforms the sequential search (computational wise). Only the NB-tree comes close to the performance of the sequential search and is actually almost the same. The pcDistance that performed best on the *ht* database even performs worse (about 18%) than the NB-tree. The NB-tree now even performs about 100% better than the LSD^h-tree.

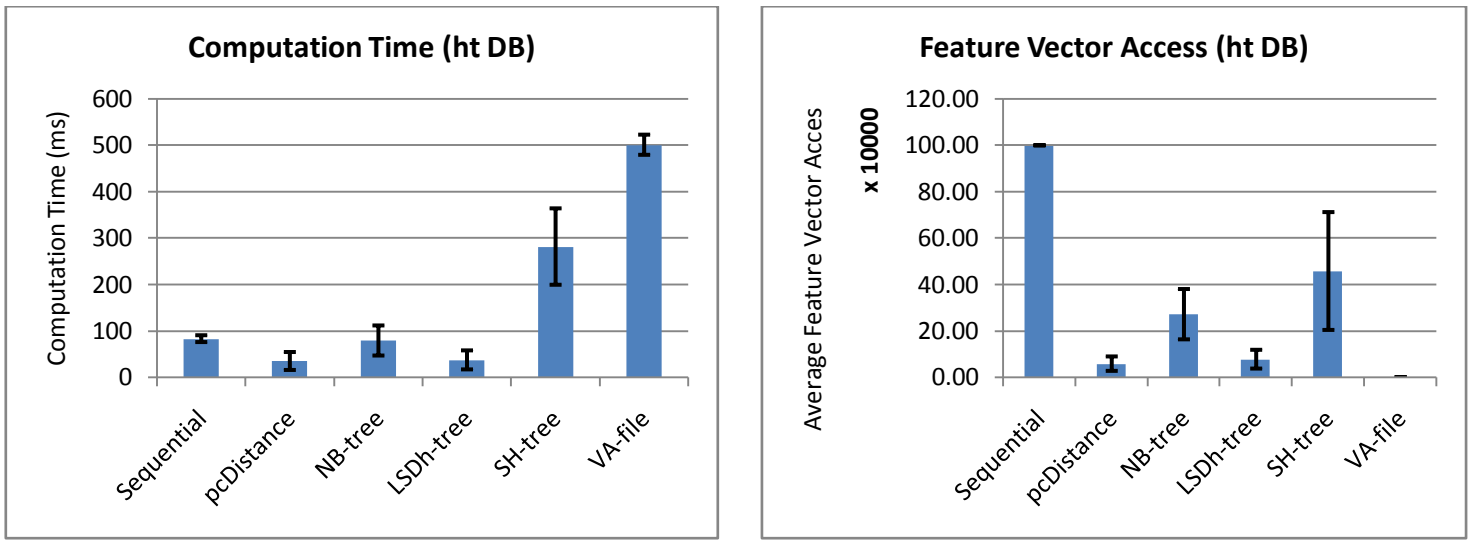


Figure 7: The left figure shows the average computation time and its standard deviation. On the right the average number of feature vector accesses and its standard deviation is shown. The *homogeneous texture* DB with 1 million feature vectors is used.

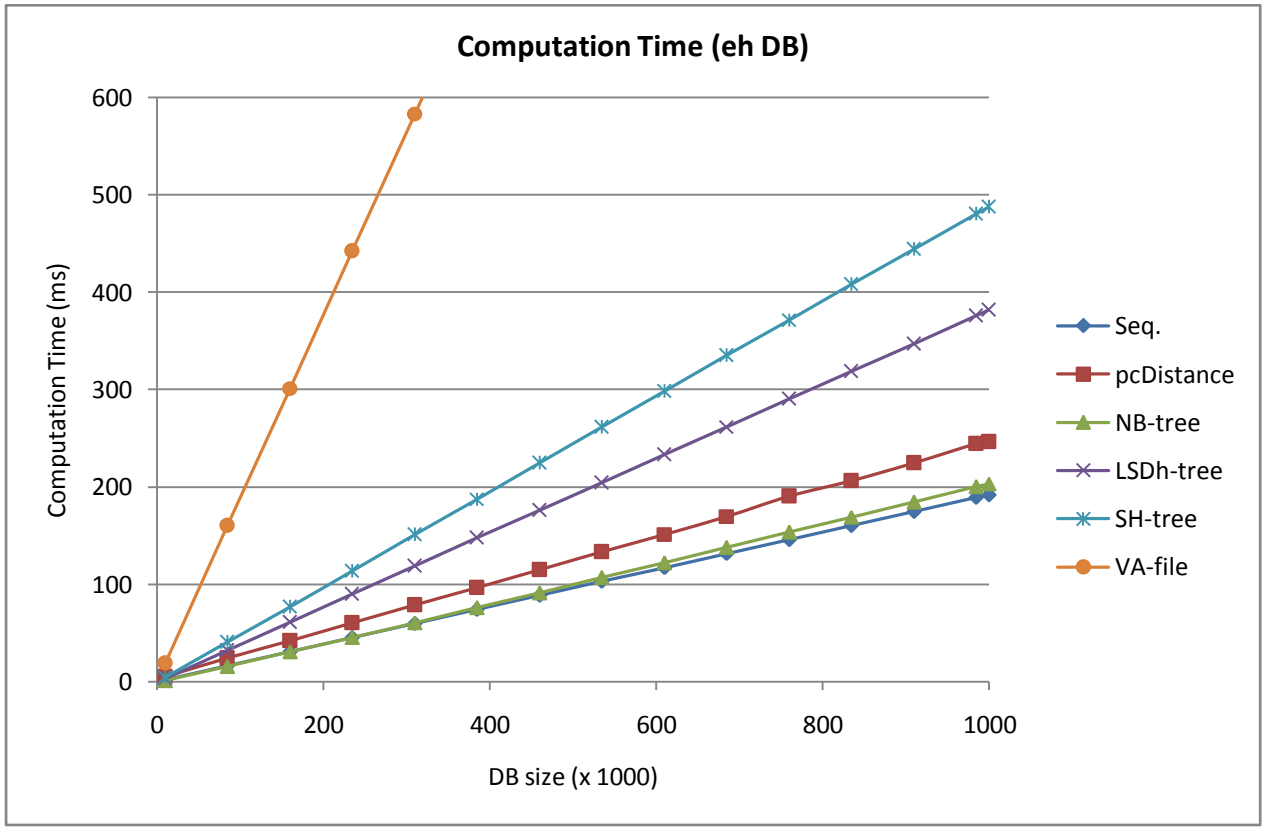


Figure 8: The average computation time of the different index structures with different data set sizes. These are the results of the *edge histogram* DB with a dimensionality of 150.

In Figure 9 the access ratio of the *edge histogram* database is shown. Also here, the VA-file outperforms all the other index structures where it comes to access ratio. pcDistance still performs second, but there are

some interesting differences between the access ratio results of the *homogeneous texture* and the *edge histogram* database. In Figure 6 the access ratio of the LSD^h-tree is very close to the access ratio of pcDistance but in Figure 9 there is a big difference between them and even the NB-tree outperforms the LSD^h-tree. Figure 10 also shows that the NB-tree is less influenced by the larger dimensionality than the other methods.

As expected, the performance of the index structures on a *43-dimensional* database with *random* feature vector is much worse than a sequential search. The index structures fail to find structure, which makes sense for random data. Figure 11 shows the average computation time of the index structures on the *random* database. Sequential search outperforms all other methods.

The access ratio of the *random* database is visualized in Figure 12. Again, the results are as expected for random data: each method except VA-file needs to access all feature vectors to find the 10 nearest neighbors.

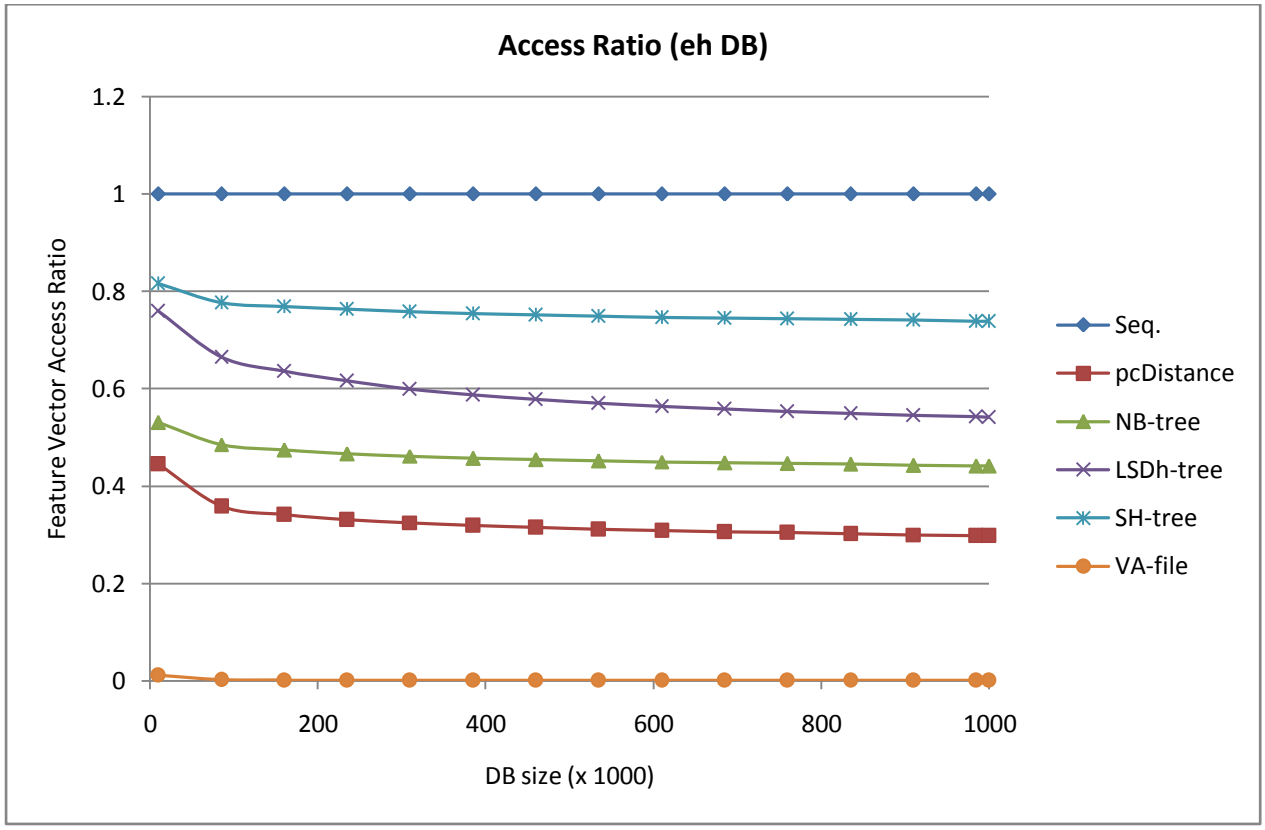


Figure 9: The average access ratio of the different index structures. These are the results of the *edge histogram* DB with a dimensionality of 150.

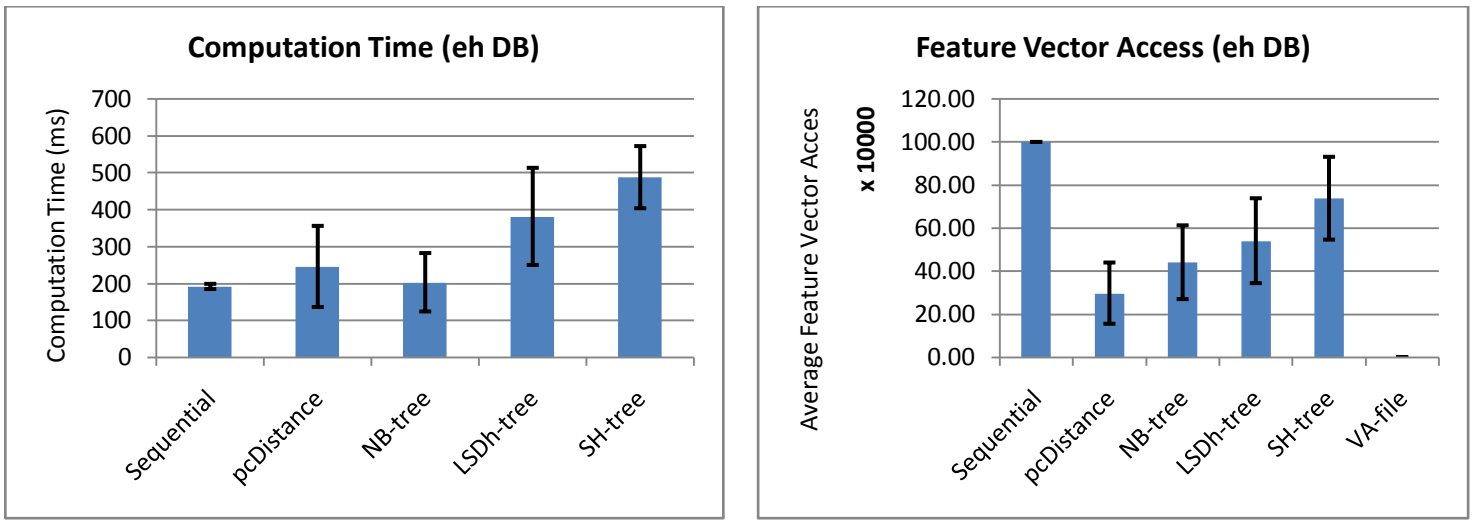


Figure 10: The left figure shows the average computation time and its standard deviation. Because of the long computation time of the VA-file, it is not shown. On the right the average number of feature vector accesses and its standard deviation is shown. The *edge histogram* DB with 1 million feature vectors is used.

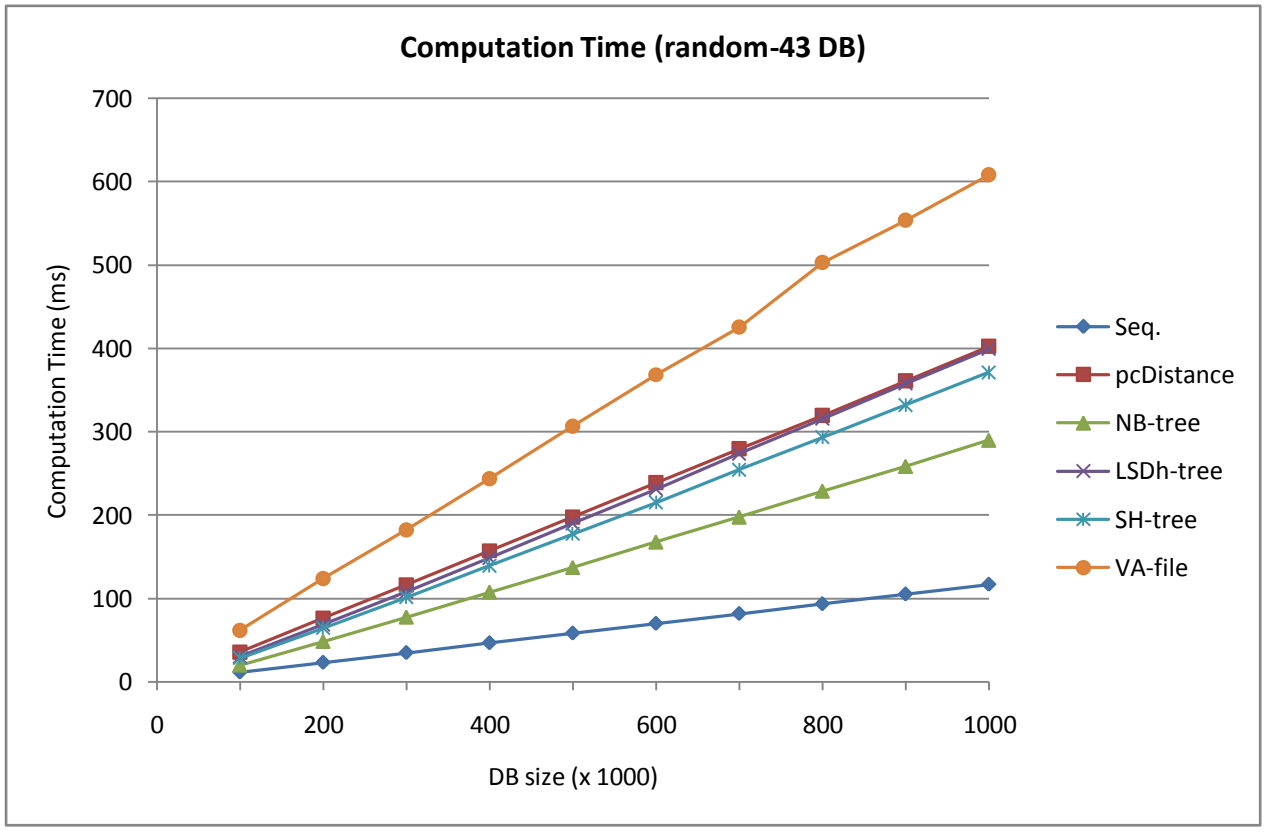


Figure 11: The average computation time of the different index structures with different data set sizes. These are the results of the *random* DB with a dimensionality of 43.

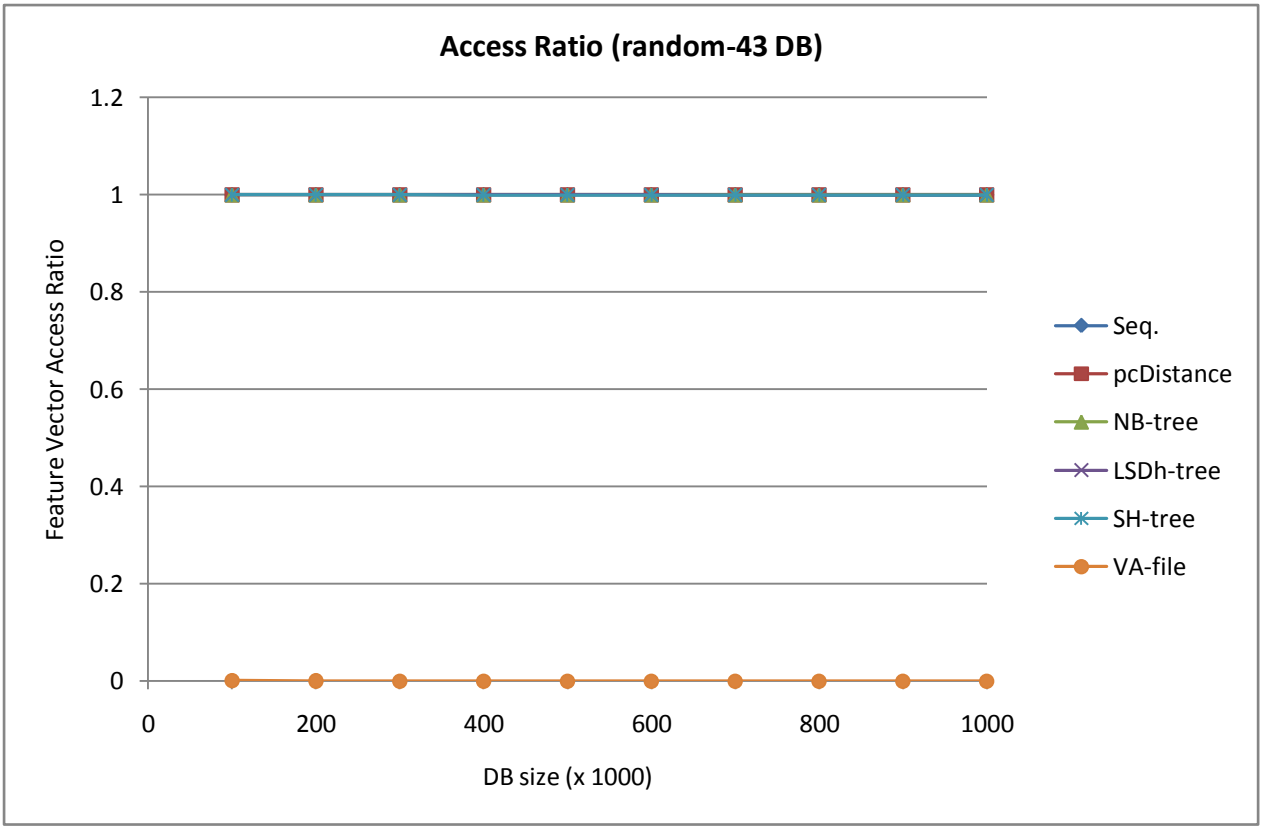
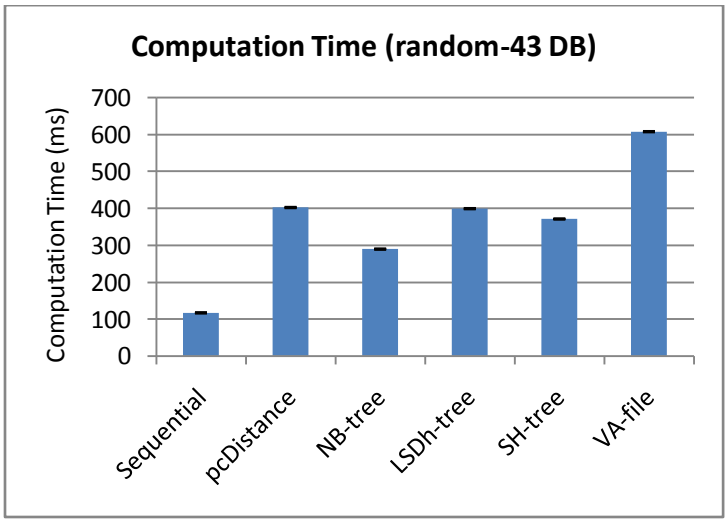


Figure 12: The average access ratio of the different index structures. These are the results of the *random* DB with a dimensionality of 43.



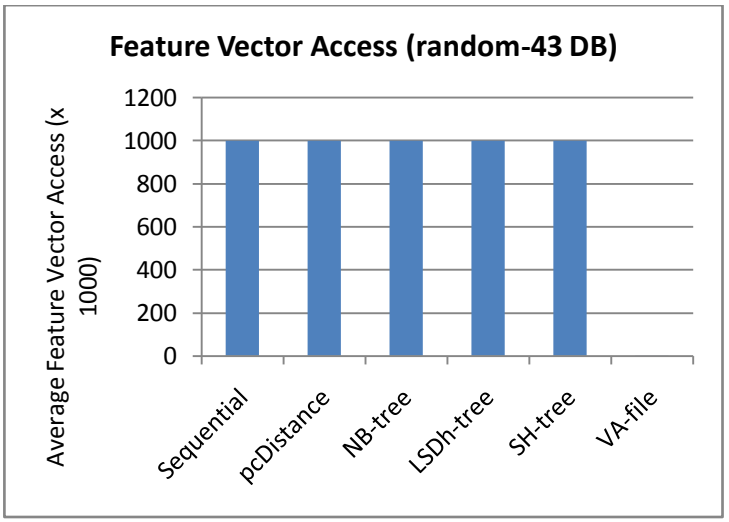


Figure 13: The left figure shows the average computation time and its standard deviation. On the right the average number of feature vector accesses and its standard deviation is shown. The *random* DB with 1 million feature vectors is used.

5. Discussion and Challenges

In this paper we have investigated the performance of a diverse high performance indexing methods in the context of very large scale search. The results show significant differences in performance between the index structures especially regarding the dimensionality of the search space. Recent prior research also had noted that naïve approaches tend to degrade more gracefully in high dimensional spaces [10]. From our experiments, we noted that there was significant disparity in the performance of the algorithms depending on which evaluation measure was used. This also gives an explanation of why there is a controversy in the perception of the high performance search algorithms. If one views them from the standpoint of the access ratio, then the high performance methods are usually greatly outperforming linear sequential search, however, one can see a different interpretation from the standpoint of computation time. Each evaluation measure gives a unique view on the situation and is informative in different ways.

The SH-tree performs poorly in computation time and access ratio for both the homogeneous texture and the edge histogram database. This is probably due to the complex structure and the inability to effectively prune the tree. During the search the SH-tree has to calculate a lot of distances to feature vectors and to minimum bonding rectangles and bounding spheres which increases computation time. The reason why the SH-tree is not capable of pruning a lot of branches is probably caused by the fact that there is a lot of overlap in the tree. The LSDh-tree performed worse on the edge histogram data set, this is also caused by the inability to prune the tree effectively. This resulted in a higher access ratio and computation time. Because the k-nearest neighbor search algorithm of the LSDh-tree uses priority queues, the performance of the search algorithm will degenerate more quickly when the algorithm is unable to prune effectively. When too many objects and nodes are pushed to and popped from the priority queue, the performance of the algorithm will be bound to the performance of the priority queue. The NB-tree had good middle-ground performance, It was shown to be capable of maintaining a good performance even when the dimensionality increases. The pcDistance is certainly promising because of its good computation time and access ratio. The tree can be effectively pruned which results in lower computation time. Because parts of the tree can be pruned without accessing the actual feature vector the access ratio is also reduced. The VA-file is interesting when it is important to access as few feature vectors as possible. The VA-file will also be smaller than a sequential method.

We found that for every index structure both computation time and feature vector access grows roughly linearly when the data set size is increased. This is the case for the 43-dimensional homogeneous texture and for the 150-dimensional edge histogram data set. There are differences between the performance of

the index structures. Some have a good access ratio like the VA-file and others have low computation time like pcDistance.

We also found significant differences in performance between the structures when using the 43-dimensional versus the 150-dimensional data set as described below:

In the panel sessions at several recent ACM conferences and as mentioned in the research literature, a controversy exists on the effectiveness of high performance nearest neighbor search algorithms. Do they outperform sequential linear search and if so, by what margin and how do they perform in very large scale similarity search? We have discovered some insights to these questions in this work.

Specifically, we have found that for the lower dimensional feature (43 dimensions), some of the high performance indexing algorithms such as pcDistance, LSDh-tree, and NB-tree outperform linear sequential search in all of the evaluation measures we used. This is rather significant because there are numerous important current societal applications and scientific areas ranging from satellite imagery to photographic to cellular microscopy to retina databases which can be directly improved in performance by using these approaches.

However, all approaches have weaknesses and the high performance indexing methods are no exception. We also found that the nature of the data is important to the performance. Specifically, random or high dimensional features may lead to poor performance in all of the high performance algorithms.

Based on our results, we conclude with the following major challenges:

The first major challenge is to develop methods which give better performance than linear sequential search for high dimensional search problems. In both big data analysis and in web search engines, it is more typical than not to have high dimensional feature vectors. While approximate methods appear to be moderately capable of delivering good results and high performance, it remains to be seen how the degradation in nearest neighbor similarity is perceived by the user.

In some situations, the feature data may appear to be nearly random. Furthermore, some systems preprocess the data so that it becomes evenly spread out over the feature axes which may lead to randomization of the data. Because none of the methods in our review performed well on random data, the second challenge is to develop methods which do perform better than linear search on random data (at least 50 dimensional, floating point).

The third challenge is to examine how users perceive the search results when approximate instead of exact nearest neighbor search methods are used. Currently, there is minimal research on this matter even though it appears that many researchers are integrating approximate search algorithms into their systems.

6. References

[1] M. J. Huiskes, B. Thomee and M. S. Lew, " New trends and ideas in visual concept detection: the MIR flickr retrieval evaluation initiative," in *MIR '10: Proceedings of the 2010 ACM International Conference on Multimedia Information Retrieval*, ACM Press, New York, pp. 527-536, 2010.

[2] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, Boston, pp. 47-57, 1984.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, pp. 322-331, 1990.

[4] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," in *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, San Francisco, pp. 28-39, 1996.

[5] D. A. White and R. Jain, "Similarity Indexing with the SS-tree," in *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, Washington, pp. 516-523, 1996.

[6] N. Katayama and S. Satoh, "The SR-tree: an index structure for high-dimensional nearest neighbor queries," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, Tucson, pp. 369-380, 1997.

[7] A. Henrich, H. .-W. Six, and P. Widmayer, "The LSD tree: spatial access to multidimensional and non-point objects," in *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, Amsterdam, pp. 45-53, 1989.

[8] A. Henrich, "The LSDh-Tree: An Access Structure for Feature Vectors," in *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, Washington, pp. 362-369, 1998.

[9] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," in *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*,

Washington, pp. 440-447, 1999.

[10] S. Ramaswamy and K. Rose, "Adaptive cluster distance bounding for high-dimensional indexing," IEEE Transactions. on Knowledge and Data Engineering, vol.. 23, no. 6, pp. 815–830, 2011.

[11] T. K. Dang, J. Küng, and R. Wagner, "The SH-tree: A Super Hybrid Index Structure for Multidimensional Data," in *DEXA '01: Proceedings of the 12th International Conference on Database and Expert Systems Applications*, London, pp. 340-349, 2001.

[12] S. Berchtold, C. Böhm, and H. Kriegal, "The pyramid-technique: towards breaking the curse of dimensionality," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, Seattle, pp. 142-153, 1998.

[13] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, pp. 500-509, 1994.

[14] M. J. Fonseca and J. A. Jorge, "Indexing High-Dimensional Data for Content-Based Retrieval in Large Databases," in *DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, Washington, p. 267, 2003.

[15] J. Cu, Z. An, Y. Guo, and S. Zhou, "Efficient nearest neighbor query based on extended B+-tree in high-dimensional space," in *Pattern Recognition Letters*, 2010.

[16] P. C. Analysis, *Principal Component Analysis*. New York, USA: Springer, 1986.

[17] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish, "Indexing the Distance: An Efficient Method to KNN Processing," in *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, San Francisco, pp. 421-430, 2001.

[18] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," in *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, San Francisco, pp. 194--205, 1998.

[19] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi, "High dimensional nearest neighbor searching," *Information Systems Journal*, vol. 31, no. 6, pp. 512-540, 2006.

[20] R. Bellman, *Adaptive control processes - A guided tour*. Princeton, USA: Princeton University Press, 1961.

[21] J. B. Macqueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281-297, 1967.

[22] J. Cui, Z. Huang, B. Wang, and Y. Liu, " Near-Optimal Partial Linear Scan for Nearest Neighbor Search in High-Dimensional Space," Lecture Notes in Computer Science Volume 7825, pp 101-115, 2013.

[23] M. Muja and D. Lowe: "Fast Matching of Binary Features". Conference on Computer and Robot Vision (CRV), 2012.