Improving the LSD^h-tree for fast approximate nearest neighbor search

Floris Kleyn
Leiden University, The Netherlands
Technical Report
kleyn@liacs.nl

1. Abstract

Finding most similar items in large datasets is a popular task. Examples are searching for similar images, scientific articles, songs and movies. Similarity search is also called nearest neighbor search and it is computationally expensive, especially when data is high dimensional. Many algorithms exists to perform nearest neighbor search, but the performance decreases enormously when the dimensionality of the data increases: the curse of dimensionality. To overcome this, it is also possible to perform an approximate nearest neighbor search which, as opposed to exact nearest neighbor search, does not always yield in the best neighbor(s) but gives good results quicker than currently possible with exact nearest neighbor search algorithms.

In this paper we propose a new algorithm for approximate nearest neighbor search and compare it with the current state of the art.

2. Related Work

In the field of multimedia retrieval[30, 26, 23, 22, 25, 16] one of the core tasks is searching for similar items by content. A lot of research has been done in the field of nearest neighbor search and many different algorithms exist. These algorithms can be divided into different categories. The most basic algorithm is linear search, where the query item is compared with all other items to obtain its nearest neighbors. More advanced algorithms limit the amount of comparisons needed to get the nearest neighbors, which can speedup the search significantly. Popular algorithms use for example trees, hashes or graphs. In the following subsections we will describe these categories in more detail.

2.1. Tree-based methods

There are many tree-based methods. Probably the most well-known is the *k*-d-tree[6]. The *k*-d-tree is a binary tree. Internal nodes split the data space by hyperplanes into two halves, based on a split dimension and split point. Points will fall in either of these two half-spaces, based on their value in the split dimension, and are therefore associated with either the left or right subtree. Splitting continues until a node only holds a single point.

Searching in a tree is done by traversing the tree to a leaf, determining in every internal node if the left or right subtree should be taken. This is based on the value in the split dimension of the query point. When a leaf node is reached it does not necessarily contain the nearest neighbor. Therefore, after reaching a leaf some backtracking is required to search for better neighbors.

As long as the dimensionality of the data is low the performance is very good. When the dimensionality increases, the performance quickly drops towards that of linear search since a lot more backtracking is required.

To reduce the time spent in backtracking methods such as best-bin-first search [5] and priority search [3] were proposed. These are approximate nearest neighbor search algorithms. Randomised *k*-d-trees were proposed in [24]. Multiple trees are created and then searched simultaneously. Every tree contains a random rotation of the data making the searches independent and speeding up the search. The LSD-tree[15] uses priority search and has buckets in its leaves that can hold several points.

Another group of trees do not use hyperplanes to decompose the space but use clustering algorithms. The hierarchical *k*-means tree was proposed in [9]. In [20] a modified version of the *k*-means tree is described that uses a best-bin-first strategy.

Other trees are for example the R-tree [10], used for spatial data, the B-tree [4], which keeps data sorted and internal nodes can have more than two children and the

Quadtree [8], which is used for two-dimensional data but can be extended toward more dimensions. The number of children of each internal node is 2^d, creating an Octree for 3-dimensional data. This rapid increase of children make it unsuitable for high dimensional data.

2.2. Hashing-based methods

Hashing-based methods are primarily about Locality Sensitive Hashing (LSH) [1]. It reduces the dimensionality of high dimensional data by projecting the data onto random chosen vectors. Items are hashed in such a way that similar items have a high probability to map to the same bucket. To achieve this a large number of hash functions is used.

Different hash families have been applied, such as Euclidean distance, Jaccard similarity and cosine similarity. A downside of LSH is the large memory requirement for hash tables. Multi-Probe LSH [19] greatly reduces the number of required hash tables while only losing a little search performance. Many methods exist that try to improve the quality of the hashes, such as kernelized LSH [17] and spectral hashing [28].

2.3. Graph-based methods

Graphs for nearest neighbor search are also used. In these graphs every data point is a vertex and edges connect data points. Nearest neighbor search may be performed on different types of graphs. In [2] a variant of a Relative Neighborhood Graph [27] is used called RNG*. A visibility graph is used in [18]. Another possibility is a nearest neighbor graph (NNG) as underlying graph[21].

A k-NNG is a directed graph where point x is connected to point y if y is one of x k-NNs. An approximate nearest neighbor search algorithm is proposed in [11], which uses hill-climbing on a k-NNG to improve search speed. It starts at a random vertex and moves to the vertex that is closest to the query point by calculating the distances of its neighbors to the query point. After a predetermined number of moves the algorithm stops. To prevent the algorithm from getting stuck in a local optimum nodes are marked when visited and are not visited again during search.

Constructing nearest neighbor graphs is an expensive operation, but [7] describes the nearest neighbor descent (NN-Descent) algorithm that constructs an approximate nearest neighbor graph, which can be built much faster than an exact graph. Since current state of the art approximate nearest neighbor search algorithms outperform exact algorithms it does not matter that the NNG is not exact.

3. Algorithm

The algorithm we developed is based on the Local Split Decision hierarchical (LSD^h) tree[14]. Therefore, we will first explain the LSD-tree[15] on which the LSD^h-tree is based. Then we will discuss the LSD^h-tree itself and finally our algorithm with the improvements we made.

3.1. LSD-tree

The LSD-tree is very similar to a k-d-tree. It is an exact nearest neighbor search algorithm. Internal nodes have a split dimension and a split point. The internal nodes partition the data space into disjoint regions based on some splitting rule. Every internal node has two children and leaf nodes contain the data points. Unlike the standard k-d-tree where a node holds a single data point, the LSD-tree has leaves that contain buckets of fixed size. Each bucket can be filled with data items until the bucket is full. When a new point is inserted into a full bucket, the bucket will overflow. New buckets are then created. The overflowing bucket is replaced by a new internal node and two buckets are attached to this node. Data points in the old bucket are distributed over the new buckets. To determine the distribution the split dimension and split point have to be determined first. This can be done solely based on the data space of the bucket (determined by all split dimensions and split points of its ancestors), either data dependent or distribution dependent. The split is thus determined locally, hence the name Local Split Decision tree.

In [13] different methods to improve the performance of k-d-tree based structures are described. They use the LSD-tree as an example. The methods all focus on higher bucket utilisation. The simplest method they proposed is the following: in case a bucket split is required, its sibling is checked first. If a sibling is also a bucket and its capacity is not yet reached, the split point is adjusted such that one point is shifted into the sibling and thus no bucket split is needed. Two other methods try to also shift a point into a bucket when the sibling is a subtree. The results of those approaches were disappointing. It resulted in a more unbalanced tree since a point is stored in a subtree, meaning it is stored at a higher level (e.g. further from

the root) than the overflowing bucket. To overcome this another method only shifts a point in another bucket if the level of the bucket is lower or equal to the overflowing bucket. This improves the bucket utilisation by 10-15% and and the same improvement was achieved with large range queries.

An algorithm to search for nearest neighbors in an LSD-tree is described in [12]. It uses two priority queues to determine which nodes should be visited: one queue for nodes (NPQ) and one queue for data points (OPQ). The priority is determined by the distance from the node or data point to the query point. Every time we descend in the tree its sibling node is stored in the node priority queue. Points are added to the queue for data points when a bucket is processed. Our search algorithm is based on this and since they are still quite similar we will only describe our algorithm in detail and mention differences.

3.2. LSD^h-tree

A new split strategy is introduced in [14] as well as coded actual data regions. These improvements made the LSD-tree more suitable for high dimensional data and make up the LSD^h tree. Coded actual data regions overestimate the bounding boxes of subtrees and buckets slightly, but reduces the amount of required memory.

Two split strategies were proposed. The first one splits in dimension $(d_{old} + i) \bmod k$, where i is 1 and k the number of dimensions of the data. If no variance within the points that are in the overflowing bucket is present in this dimension, i is increased by 1 until a dimension is reached where variance does occur. For high dimensions they mention that this method might not be suitable and alternatively the dimension with the highest variance (based on items in the overflowing bucket) can be used as the splitting dimension. This is the same strategy as used by the VAMSplit R-tree [29]. The split point is determined by calculating the average of the values in the split dimension of the data points present in the bucket.

The LSD^h-tree only uses the simplest described strategy to prevent bucket splits: it only shifts a point to another bucket in case its sibling is also a bucket (and still has space left).

3.3. Improving the LSD^h-tree

To improve the search speed we adjust the LSD^h-tree to an approximate nearest neighbor search algorithm. To achieve this our algorithm terminates after a certain number of leaves (e.g. buckets) have been visited. This is controlled by a *cost* parameter. By changing the number of buckets that we visit we can control the quality of the nearest neighbors that the algorithm returns.

After some nodes have been visited during search it is still possible to find good neighbors by processing nodes that are already in NPQ. The chance however that new nodes that are added to the queue have a high enough priority to actually be processed diminishes. Therefore, after the cost drops below a certain threshold no new nodes are added to NPQ, reducing the amount of distance calculations. Nodes already in the queue are still processed until *cost* is zero. The idea of a threshold is also present in the FLANN library [20].

Searching in more than a single k-d-tree simultaneously was proposed by [24]. Every tree has a different structure but holds the same data. Visiting many nodes in the same tree only results in visiting nodes that are far away from the node that contained the query point (the node does not really have to hold the query point but the query point would be in that bucket if it was present in the tree, based on the split dimensions and split points in the tree). This reduces the chance that it holds a nearest neighbor. Creating more than one tree and visiting them simultaneously makes sure that more buckets that contained the actual query point are searched (but have different content since other split decisions were made) and also that nodes nearby the query point are searched. To prevent that the same data item is processed (and potentially reported as a nearest neighbor) more than once all data items are marked when processed for the first time during a search.

A small improvement to speed up the algorithm is instead of using a priority queue (where only the top item in the queue is accessible) is using a bounded priority queue for OPQ where also the bottom item is available. The bounded priority queue can hold at most n items. In our case n is equal to the number of nearest neighbors the algorithm should return. This allows for incremental calculation of the distance to the query point: in a standard priority queue the worst element is not accessible so when calculating the distance between a query point and another point we have no idea of the quality of this point. It might be a very bad candidate but we do not know since we have only knowledge of the best neighbor found so far. Now, when the queue holds n elements and the distance to a new item is calculated, we can stop the distance calculation as soon as it is higher than the

distance of the worst item in the queue. If the distance stays smaller that the worst item in the queue the entire distance is calculated and the point is added to the queue. The worst item is then removed from the queue to keep n items in the queue.

We do not use the coded actual data regions since the extra memory requirement is not a problem.

3.3.1. Split approach

We have tested three different split methods: local, semi-local and global based splits. Local splitting is as described in Section 3.2 where the dimension with highest variance is used as split dimension. Split dimension and split point are only based on points in a bucket. For semi-global splits the split dimension is determined globally by calculating the variance based on a sample of all data that falls in the data region contained by the bucket (e.g. delimited by split points and split dimensions of all its ancestors). The split point is determined locally.

For global splitting the variance and the average are based on samples of the entire data set. As a consequence the tree is constructed in a different way. When construction starts all data points lie in the same data space region. If the number of points in the data space region is higher than the bucket capacity the data space is split. The split is determined by calculating the variance of a sample of points that lie in that region. To ensure that every tree in a forest is different we do not select the dimension with highest variance but pick one at random, between top-k dimensions with highest variance. After the dimension is picked the split point is determined by calculating the average of the sample in the split dimension. Then, for all points in the data space it is determined in which of the, now two, regions it falls. Regions are split as long as they contain more points than fit in a single bucket. Every time the region is split an internal node is added to the tree with the split dimension and split point. When the points that lie in a region fit in a bucket a leaf node is created.

Figure 1 shows the three split approaches. First, a parameter sweep was performed to select the best bucket size for every split approach. We tried global splitting approach for different top-k high variance dimensions. The best approach is to make the split decision based on global information, if a suitable k is chosen. The worst strategy is to split semi-local. We found it surprising that local split decisions perform so well since it depends entirely on the points in a bucket. There are no guarantees that

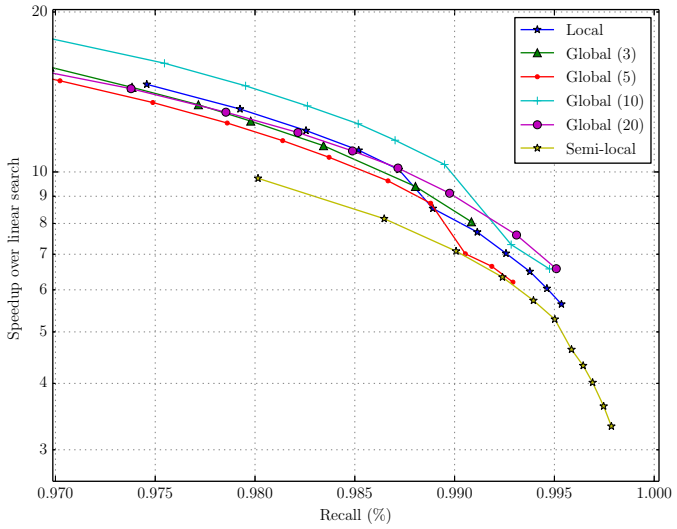

Figure 1: *Comparison between the three different split approaches on the SIFT-dataset. Between parentheses the number of top-*k *high variance dimensions is stated.*

the content of a bucket contains a representative sample of the dataset. This approach probably works well since the data is inserted in a random permutation per tree, creating more different trees than the global split approach does.

Global based splits profit from the fact that the statistics on which the split is based reflect the entire dataset. As a consequence the bucket utilisation is much higher so we achieve a smaller tree with a more even spread of points per bucket. In our experiments (Section 4) we use local split to keep one of the fundamental ideas behind the LSD-tree intact. This method is also more suitable for incrementally updating the tree and prevents further parameter tuning. Global splits need a second insert algorithm for incremental updating the tree. Of course, it is possible to rebuild the tree every time when new points need to be added. It is also not unlikely that after the tree is built it is used for a long time for running queries. In that case global splits are fine as long as a good k is picked.

3.4. Early stopping

The original search algorithm has a check that compares the top elements in NPQ and OPQ. If the top element in OPQ is closer to the query point than the top element in NPQ then the element is removed from OPQ, since no nearer points are present in the tree, and stored in a

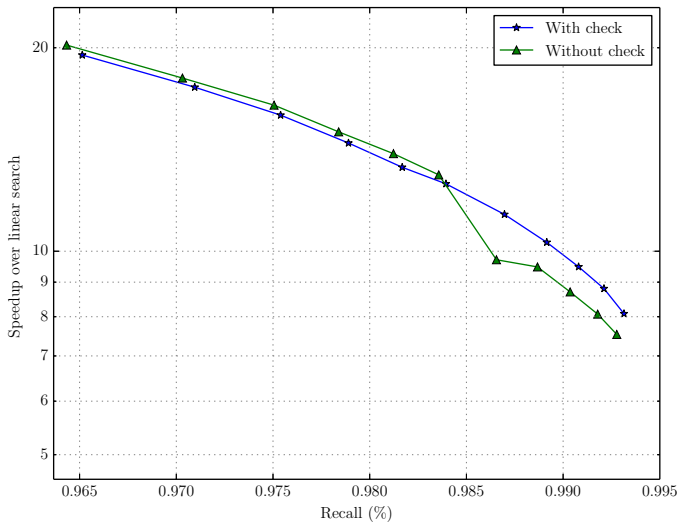

Figure 2: *Comparison between checking if best item in NPQ is worse than worst item in OPQ, and without this check on the SIFT-dataset.*

FIFO-queue holding the nearest neighbors in ascending order.

We perform a similar check, but since we not only know the closest element (best) in OPQ but also the furthest away element (worst) we perform a different check. If the worst element in OPQ is better than the best element in NPQ then we do not need to search any further and we have found the top-k exact nearest neighbors.

We ran an experiment where this check is available in the algorithm and where it is not. Since our algorithm is approximate we also stop the search when *cost* has reached zero. If *cost* always reaches zero before the check is true then it will only result in overhead. In Figure 2 the result of this experiment is shown. At very high recall levels it gives an improvement in the performance while only a very low decrease in performance exists at slightly lower recall. Therefore we keep this check in our algorithm.

3.5. Pruning

We tried two methods to prune the tree during search. We first tried to prune the tree while descending to a leaf by calculating the distance between every visited internal node and the query point. If the distance would be larger than the worst distance in OPQ we could stop the search and proceed with the next node in NPQ. However, the added cost of calculating the distance did not outweigh the time saved with pruning the tree. Then we tried to only prune buckets by calculating the distance between a bucket (determined by the bounding box of its content) and the query point. At the cost of one extra distance calculation comes the possibility to save calculating the distance for every item in the bucket. But this method decreased performance also. Adding distance calculations to potentially save more distance calculations did not seem to work.

3.6. Explaining the algorithm

In Algorithm 1 the pseudo code of our algorithm is listed. Our input consists of the roots of all trees in the forest (*trees*), the number of required nearest neighbors (*nn*), two empty queues OPQ (Object Priority Queue) and NPQ (Node Priority Queue) and our query point q. NPQ holds nodes as where OPQ holds data points. The priority is determined by the distance to q. The entire forest has just two queues so we search in the tree that contains the node with highest priority (smallest distance) to q. *Cost* determines the maximum number of leaves that will be visited. The output is result set R that will contain the *nn* approximate nearest neighbors to q.

At the start of our algorithm we put some dummy data in OPQ such that it always holds *nn*. Otherwise every time an item is added to OPQ the size of the queue should be checked to determine if the worst item from OPQ should be removed. All roots are stored in NPQ. We use a distance of 0 since the distance from the root to the query point will be very small (in most cases 0): The root contains the entire search space, of which its boundary is defined by the bounding box of all points, and it is very likely our query point falls within this bounding box. By choosing 0 we limit the amount of distance calculations. Finally, we also set the threshold. We set it to 83% of *cost*.

In our main loop the trees are searched as long as less than the maximum number of leaves have been visited ($cost > 0$) and the best item in NPQ is better than the worst item in OPQ. Every iteration the node closest (based on the distance between its bounding box and the query point) to the query point is removed from NPQ and used to traverse the tree. As long as this node has a left child (and thus a right child) the algorithm has to determine if the left or right subtree should be taken. This is determined on the value of the query in the split dimension of the node. If the cost is above the threshold the sibling node of the root of the subtree that will be taken is added to NPQ.

input : Empty priority queues NPQ and OPQ, cost $cost$, number of nearest neighbors nn, roots of all trees in $trees$ and query point q.
output: Result set R filled with nn approximate nearest neighbors.

```
for 0 to nn do                    // Fill queue to prevent checks
    OPQ.add(NULL, ∞);
foreach w ∈ trees do              // Push all roots in queue
    NPQ.add(w, 0);
threshold = cost * 0.83;
while cost > 0 and dist(NPQ_min, q) < dist(OPQ_max, q)
and NPQ.size() > 0 do
    w = NPQ_min;
    NPQ.deleteMin();
    while w_l do                  // Node is not a bucket
        s_dim = split dimension stored in w;
        s_point = split point stored in w;
        if q[s_dim] ≤ s_point then
            if cost > threshold then
                NPQ.add(w_r, dist(w_r, q));
            w = w_l;              // Take left branch
        else
            if cost > threshold then
                NPQ.add(w_l, dist(w_l, q));
            w = w_r;              // Take right branch
    // We reached a bucket
    foreach o ∈ bucket do
        if not isProcessed(o) then
            if dist(o, q) < dist(OPQ_max, q) then
                OPQ.deleteMax();
                OPQ.add(o, dist(o, q));
    cost = cost − 1;
for 0 to nn do
    R.add (OPQ_min);
    OPQ.deleteMin();
```

Algorithm 1: Approximate nearest neighbor search algorithm for LSD^h-tree.

A bucket is reached when a node does not have a left child (and therefore no right child). The distance between the items in the bucket and the query point is then calculated, if at least that item is not already processed in another tree. As soon as the, so far calculated, distance is larger than the maximum in OPQ, the distance calculation is stopped. If it is smaller it is added to OPQ and the worst item in OPQ is deleted.

When the main loop terminates the items in the queue are added to our result R. The main loop will terminate either because the $cost$ reached zero or because the exact nn nearest neighbors have been found.

4. Experiments

In this Section we describe the steps we took to compare the performance of our algorithm to some state of the art algorithms.

4.1. Algorithms

We picked randomised k-d-trees, hierarchical k-means, Multi-Probe LSH and hill-climbing on a k-NN graph. We calculated the speedup every algorithm achieved compared to linear search.

FLANN (Fast Library for Approximate Nearest Neighbors)[1] is a library written in C++ by the authors of [20]. It has implementations of randomised k-d-trees and hierarchical k-means. It currently sets the standard for nearest neighbor search. Auto-tuning always gave k-means as the most suitable algorithm which is in line with their research that said that k-means is more suitable when high precision is required. Therefore, we only ran experiments with k-means and dropped the k-d-tree. We also used their implementation of linear search.

We used LSHKIT[2] for Multi-Probe LSH. It is a Locality Sensitive Hashing library also written in C++. It contains several LSH algorithms and comes with the possibility of auto-tuning the parameters for Multi-Probe LSH, which we used.

We implemented hill-climbing on a k-NN graph ourselves. However, we used NNDES[3] for graph construction. It implements the NN-Descent algorithm described in [7]. We performed a parameter sweep to select the best parameters.

We also performed a parameter sweep on our improved LSD^h algorithm, selecting different values for the number of trees, the bucket size and the number of buckets to be visited.

[1] http://www.cs.ubc.ca/research/flann/
[2] http://lshkit.sourceforge.net/
[3] http://code.google.com/p/nndes/

4.2. Datasets

We selected a couple of datasets. First the SIFT1M[4] dataset with one million SIFT-feature vectors of 128 dimensions. It comes with a separate query dataset with 10,000 vectors. Another dataset we used is a subset of the Million Song Dataset[5] containing 515,345 feature vectors of 90 dimensions. We used 10,000 of these vectors as queries. The last dataset had one million vectors of 20 dimensions containing random data, which was generated by ourselves.

4.3. Recall

For every algorithm we measured the running time at different recall levels and compared that with the running time of linear search. We defined recall as the percentage of overlap between the groundtruth and the approximate nearest neighbors returned by an algorithm. All algorithms return neighbors in ascending distance to the query point so there was no need to take the position of a neighbor in the output into account.

We also experimented with two other recall measures. We gave a higher weight to nearer neighbors by awarding k points to the nearest neighbor up to 1 point to the k-NN. At the same percentage of correct nearest neighbors the algorithm that has more points returns on average nearer neighbors. The other measure applied a similar technique but now a lower score is better: 1 point for 1-NN up to n points for n-NN (where $n \gg k$). If a returned neighbor was worse than n-NN a fixed penalty was given. A higher score would mean that the incorrect neighbors are further away from the query than at a lower score (given that the same percentage of correct neighbors is returned). These methods are quite similar to each other but the second method allow us to punish returned neighbors that are far away better. We only report results based on the first recall measure since differences where low and had no impact on the ranking of algorithms.

4.4. Results

In Figures 3, 4 and 5 the results of our experiments are listed. In all our experiments with the LSD^h-tree we use 75 trees as more trees resulted in better performance. On the SIFT-dataset we use a bucket size of 30. For the Million Song Dataset and random dataset we used a bucket size of 90 and 20 respectively. The results show the performance on different datasets with a recall between 0.8 and 1.0. The most notable thing is that our algorithm performs really poor on the Million Song dataset (Figure 4) and does not achieve high recall here. A higher speedup can be achieved with another split approach but it will not improve recall. Another thing to notice is the difference in speedup achieved on the random data dataset. The nearest neighbor graph method really stands out here. The performance of the other methods, except Multi-Probe LSH, are all very similar to each other on the SIFT-dataset, although on the performance of the nearest neighbor graph method drops quickly at high recall.

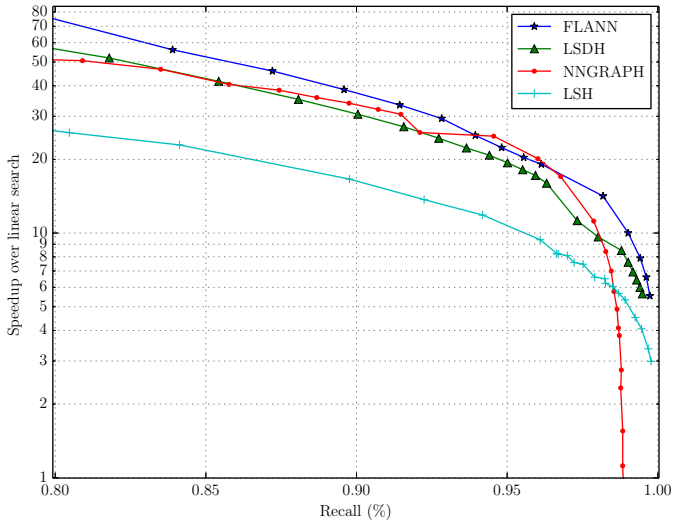

Figure 3: *Comparison of different approximate nearest neighbor search algorithms on the SIFT1M dataset at 10 nearest neighbors.*

To better highlight the differences in performance between methods we also show the performance with a recall between 0.97 and 1.0. These results are listed in Figures 6, 7 and 8.

5. Conclusion

There is no algorithm that performs best on all datasets. When high performance is required some experiments have to be done to determine which algorithm is the best algorithm for the particular dataset. FLANN performs really well, but it can still be outperformed by other algorithms. Although FLANN is currently the standard it is not in all cases the best choice.

Our improved LSD^h-tree did not perform as well as we hoped. A more extensive parameter sweep might improve

[4] `http://corpus-texmex.irisa.fr/`
[5] `http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD`

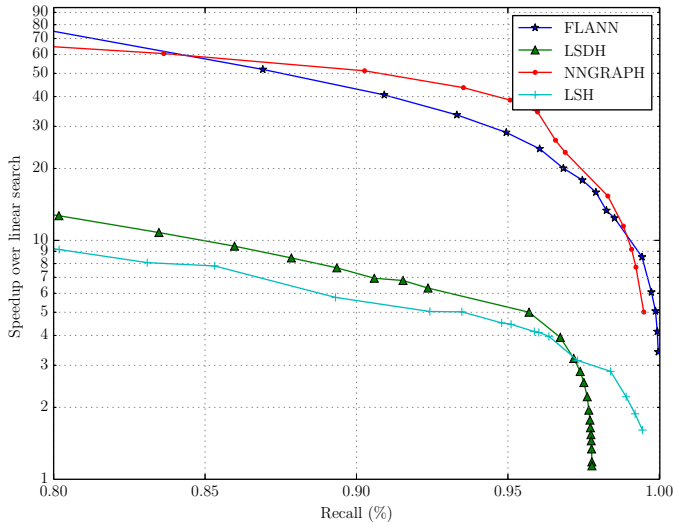

Figure 4: *Comparison of different approximate nearest neighbor search algorithms on the Million Song dataset at 10 nearest neighbors.*

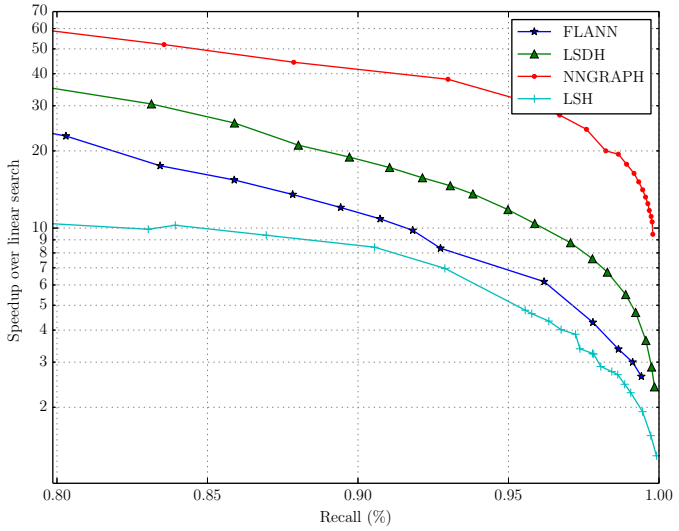

Figure 5: *Comparison of different approximate nearest neighbor search algorithms on the random data dataset at 10 nearest neighbors.*

the performance and switching from local splits to global splits does improve results, although it would not be Local Split Decision anymore. Further testing should also be done at different number of returned nearest neighbors.

If pruning would be cheaper performance might increase but calculating the Euclidean distance of a bounding box to the query point simply takes too much time. It might be possible to estimate the distance instead to speed up the pruning. Another problem is that the true distance from the query point to the bounding box is always too optimistic. Points inside the bounding box will (nearly) always be further away from the query point than the bounding box itself. Currently it is possible that a subtree is not pruned since its distance is close enough to the query to contain good neighbors although all of the data points in the subtree are actually too far away. If this distance underestimation can be tackled, pruning will be more feasible and search performance might increase.

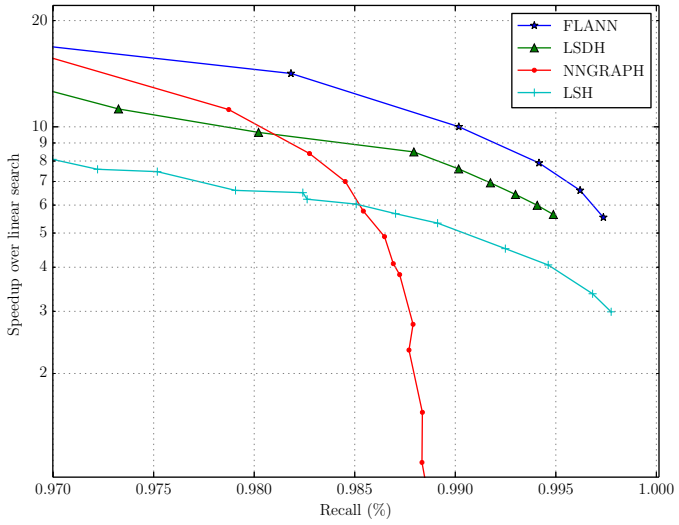

Figure 6: *Comparison of different approximate nearest neighbor search algorithms on the SIFT1M dataset at 10 nearest neighbors at high recall levels.*

References

[1] Alexandr Andoni and Piotr Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In: *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE. 2006, pp. 459–468.

[2] Sunil Arya and David M Mount. "Approximate Nearest Neighbor Queries in Fixed Dimensions." In: *SODA*. Vol. 93. 1993, pp. 271–280.

[3] Sunil Arya et al. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions". *Journal of the ACM (JACM)* 45.6 (1998), pp. 891–923.

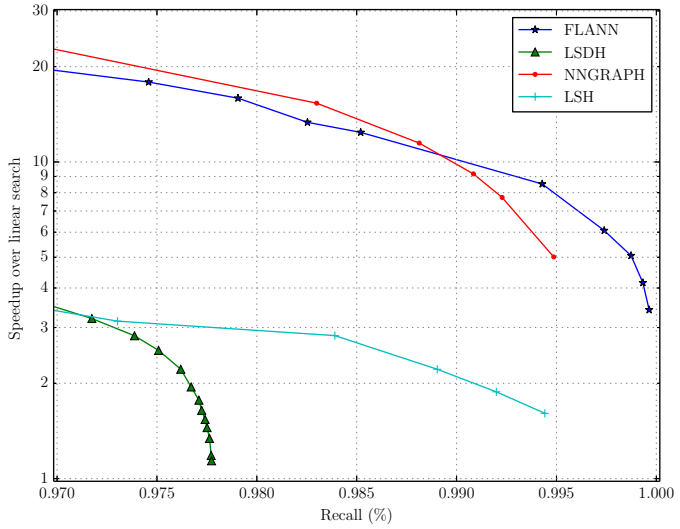

Figure 7: *Comparison of different approximate nearest neighbor search algorithms on the Million Song dataset at 10 nearest neighbors at high recall levels.*

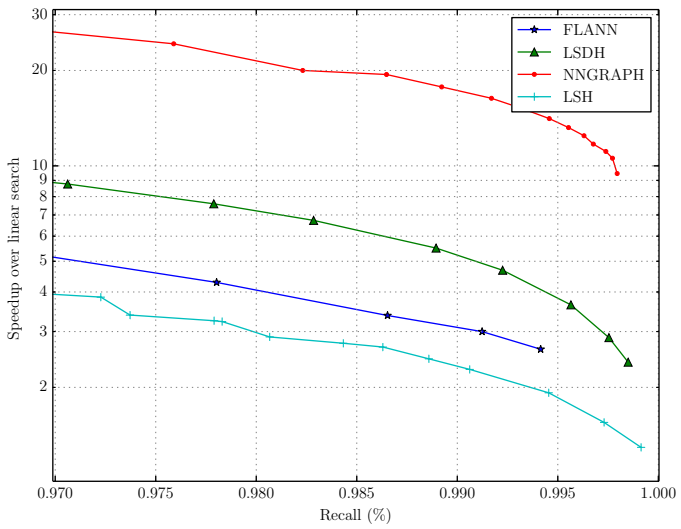

Figure 8: *Comparison of different approximate nearest neighbor search algorithms on the random data dataset at 10 nearest neighbors at high recall levels.*

[4] R BAYER. "Organization and maintenance of large ordered indexes". *Acta Informatica* 1.3 (1972), pp. 173–189.

[5] Jeffrey S Beis and David G Lowe. "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces". In: *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE. 1997, pp. 1000–1006.

[6] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". *Communications of the ACM* 18.9 (1975), pp. 509–517.

[7] Wei Dong, Charikar Moses, and Kai Li. "Efficient k-nearest neighbor graph construction for generic similarity measures". In: *Proceedings of the 20th international conference on World wide web*. ACM. 2011, pp. 577–586.

[8] Raphael A. Finkel and Jon Louis Bentley. "Quad trees a data structure for retrieval on composite keys". *Acta informatica* 4.1 (1974), pp. 1–9.

[9] Keinosuke Fukunaga and Patrenahalli M Narendra. "A branch and bound algorithm for computing k-nearest neighbors". *Computers, IEEE Transactions on* 100.7 (1975), pp. 750–753.

[10] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. 2. ACM, 1984.

[11] Kiana Hajebi et al. "Fast approximate nearest-neighbor search with k-nearest neighbor graph". In: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22. 1. 2011, p. 1312.

[12] Andreas Henrich. "A Distance Scan Algorithm for Spatial Access Structures." In: *ACM-GIS*. Citeseer. 1994, pp. 136–143.

[13] Andreas Henrich. "Improving the performance of multi-dimensional access structures based on kd-trees". In: *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*. IEEE. 1996, pp. 68–75.

[14] Andreas Henrich. "The LSD h-tree: An access structure for feature vectors". In: *Data Engineering, 1998. Proceedings., 14th International Conference on*. IEEE. 1998, pp. 362–369.

[15] Andreas Henrich et al. "The LSD tree: spatial access to multidimensional point and non-saint objects" (1989).

[16] M. Huiskes and M. Lew. "Performance evaluation of relevance feedback methods". In: *Proceedings of the ACM International Conference on Image and Video Retrieval*. 2008.

[17] Brian Kulis and Kristen Grauman. "Kernelized locality-sensitive hashing for scalable image search". In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 2130–2137.

[18] Yury Lifshits and Shengyu Zhang. "Combinatorial algorithms for nearest neighbors, near-duplicates and small-world design". In: *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2009, pp. 318–326.

[19] Qin Lv et al. "Multi-probe LSH: efficient indexing for high-dimensional similarity search". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 950–961.

[20] Marius Muja and David Lowe. "Scalable nearest neighbour algorithms for high dimensional data" (2014).

[21] Rodrigo Paredes and Edgar Chávez. "Using the k-nearest neighbor graph for proximity searching in metric spaces". In: *String Processing and Information Retrieval*. Springer. 2005, pp. 127–138.

[22] Y. Rui. "Big Data and Image Search". *IEEE Multimedia* (2014).

[23] N. Sebe, M. Lew, and A. Smeulders. "Video retrieval and summarization". *Computer Vision and Image Understanding* 92 (2003), pp. 141–146.

[24] Chanop Silpa-Anan and Richard Hartley. "Optimised KD-trees for fast image descriptor matching". In: *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE. 2008, pp. 1–8.

[25] Y. Sun et al. "Authentic emotion detection in real-time video". In: *LNCS 3058: International Workshop on Human-Computer Interaction*. 2004, pp. 92–101.

[26] B. Thomee and M. Lew. "Interactive search in image retrieval: a survey". *International Journal of Multimedia Information Retrieval* 1.2 (2012), pp. 71–86.

[27] Godfried T Toussaint. "The relative neighbourhood graph of a finite planar set". *Pattern recognition* 12.4 (1980), pp. 261–268.

[28] Yair Weiss, Antonio Torralba, and Rob Fergus. "Spectral hashing". In: *Advances in neural information processing systems*. 2009, pp. 1753–1760.

[29] David A White and Ramesh Jain. "Similarity Indexing: Algorithms and Performance." In: *storage and retrieval for image and video databases (SPIE)*. 1996, pp. 62–73.

[30] L. Zhang and Y. Rui. "Image search-from thousands to billions in 20 years". *ACM Transactions on Multimedia* 9.1 (2013).