Scalable Nearest Neighbour Algorithms for High Dimensional Data

Hussein Houdrouge
supervised by
Prof. Maks Ovsjanikov

Ecole Polytechnique - INF 513 - M1 Project

March 2018

Contents

1 Introduction

The main aim of this project is to study several data structures that solve efficiently the problem of Nearest Neighbour Search in Higher Dimensions. This problem is important for many application in different fields such as Pattern Recognition, Computer Vision, Data Bases, and Coding Theory. For instance, [1] presents a pattern matching method based on fast Nearest Neighbour Search algorithms. The use of fast Nearest Neighbour Search algorithms in [1] leads to speed improvement of several orders of magnitude.

1.1 Nearest Neighbour Search Problem

Consider a set of points $P = p_1, p_2, ..., p_n$ in a metric space M and a set of query points Q where $Q \subset M$. The goal is to find for each point $q \in Q$ the nearest point $N(q) \in P$. That is,

$$N(q) = \arg\min_{x \in P} d(q, x)$$

where d is a metric distance $d : M \times M \longrightarrow R$.
For some applications the user is not interested to find one closest point. He/She wants to find k closest points. This version of the problem is called *k-nearest neighbour*. This problem is defined as follow.

$$KNN(q, P, K) = A$$

where $\|A\| = K$, $A \subseteq P$ and $\forall x \in A, \forall y \in P - A, d(q, x) \leq d(q, y)$.

The naive way to solve the Nearest Neighbour Search problem is to use Linear Search. That is, scan all the data-set and report the point with the minimum distance. The complexity of such algorithm is $O(mn)$ where n is the number of points in P and m is the dimension of the data. The d factor is the complexity of computing the distance. For the K-Nearest Neighbour Search, the naive algorithm consists in sorting the points according to their distance from the query point. The complexity of this algorithm is mainly the complexity of the sorting algorithm $O(n \log n)$.

1.2 Objectives and Goals

The main goals of this project are implementing and analysing the data structures introduced in [2]. These data structures are Randomised k-d Tree and Priority Search K-Means Tree. In addition, the project includes the implementation of the standard K-d Tree because it is an essential part of the Randomised K-d Tree algorithm. Furthermore, the project studies the performance of these data structures over data sets of different nature such as SIFT (The scale-invariant feature transform) computed on classical painting and real world photos, which is 128-dimensional vector describes the local features of an image, and data set sampled uniformly at random. Moreover, the project studies the effect of the parameters of each data structure on their performance and precision. After these experiments and observations, the project suggests guidelines to effectively use each of the data structures.

2 Algorithms' Description

This section describes the construction and the search algorithm of each data structure. In addition, it analyses the computational complexity of each algorithm. The result of this analysis is illustrated in the following table.

Data Structure	Construction Algorithm	Search Algorithm
K-d Tree	$O(n \log n)$	$O(\log n)$
Randomised K-d Tree	$O(Nn \log n)$	$O(L \log n)$
Priority K-Means Tree	$O(ndKI(\log n / \log k))$	$O(Ld(\log n / \log k))$

where n is the number of points in the data-set. L and N for the Randomised K-d Tree Stand for the maximum number of visited leafs and the number of trees respectively. d, K, and I in the Priority K-Means Tree respectively stand for the dimensionality, branching factor, and the maximum number of iteration in the K-Means algorithm.

2.1 K-d Tree

The k-d Tree is one of the main data structures used to solve the Nearest Neighbour Search problem. It is a binary tree that splits the space recursively into two halves. The usage of K-d Tree consists of two stages preprocessing (construction) stage and the query stage. The following subsection describes the construction and the search algorithms as they are presented in [3].

2.1.1 Construction Algorithm

The construction of the K-d Tree starts over the set of n points P in d-dimensional space. Given a dimension i the construction algorithm computes the median of the coordinates of the i^{th} dimension. That is, the coordinate that splits the set of n points into two equal sets L and R where $\|L\| = \|R\| = \frac{n}{2}$. The algorithm is called recursively over the set L and R until $\|R\| = \|L\| = 1$. And at each step, the splitting dimension changes as follow. $i := i + 1 \ mod \ d$.
In this construction algorithm, the most expensive operation is the computation of the median. This operation is done by sorting the set of points which costs $O(n \log n)$. However, one can optimise this operation and reduce its cost to $O(n)$ using the median of medians algorithm as it is described in [4]. Therefore, the running time on can be expressed as

$$T(n) = \begin{cases} O(1), & if n = 1, \\ O(n) + 2T(\lceil \frac{n}{2} \rceil), & if n > 1, \end{cases}$$

which solves to $O(n \log n)$. Furthermore, the space complexity required to construct the K-d Tree is $O(n)$ [5].

2.1.2 Search Algorithm

The search algorithm starts from the root and recursively walks through the tree according to the value of the coordinate at the splitting dimension. If the coordinate at the splitting dimension less than the median, the search-algorithm explores the left sub-tree, otherwise it explores the right sub-tree. In addition, the algorithm maintains the distance to the closest encountered point. This distance is initialised to infinity ($R = \infty$). However, after

hitting a leaf node, R changes as follows $R = min(R, distance(q, p))$ where q is the query point and p is the point at the leaf node. Then, the algorithm checks if the circle centred at q with radius equals to R intersects any of the half-planes. If the test is positive, the algorithm continues its search in the corresponding half-planes. Otherwise, it terminates immediately. The complexity of this algorithm in the best case is $O(\log n)$ and in the worst case $O(n)$ [3].

2.2 Randomised K-d Tree

The Randomised k-d Tree algorithm [6] is an approximate nearest neighbour algorithm [2]. It consists of several Randomised K-d Trees searched in parallel. Having different space partitions (K-d Trees) increases the chance of finding a good initial point near the query point which leads to a good approximation.

2.2.1 Construction Algorithm

The construction of the Randomised K-d Tree is the construction of several K-d Trees with the difference that the splitting dimension is chosen at random from the top N_D dimensions with the highest variance. In our implementation and the implementation of [2], N_D is fixed to five. In this approach, choosing the splitting dimension adds an additional overhead on the construction time. The cost of this overhead is at least $O(dn)$ where d is the dimension of the space and n is the number of points. The computational complexity of this algorithm is still $O(n \log n)$ but up to a different constant since it builds multiple K-d Trees instead of one.

2.2.2 The Search Algorithm

The search algorithm maintains a shared priority queue across all trees. This priority queue is ordered by increasing distance to the decision boundary (half-plane) [2]. In addition, the algorithm keeps track of the visited data points to avoid later exploration. The algorithm starts by exploring the closest point from all the trees i.e the point on the top of the priority queue. The algorithm explores the tree in the same way as the K-d Tree search algorithm described in the previous section. However, the search algorithm is parameterized by the maximum number of leafs to be visited. This parameter is an important factor to determine the degree of approximation. Increasing this parameter leads to a better approximation. The computational complexity of this algorithm is the cost of searching the K-d Tree multiple times. The number of the search is bounded by L the maximum number of visited leafs which leads to $O(L \log n)$ time complexity.

2.3 The Priority Search K-Means Tree

This tree tries to take the advantage of the natural structure of the data by clustering it in a hierarchical fashion. At each level, the tree clusters the data into K clusters. Solving Nearest Neighbour search by clustering the data point has been used before by [7], [8], and [9].

2.3.1 Construction Algorithm

The construction algorithm takes the parameter K, the data set P of dimension equals to d and size equals to n, the maximum number of iterations I_{max}, and centre selection algorithm C_{alg}. The parameter K determines the maximum size of the cluster at the leaf node. In addition, it determines the number of the children at each level i.e the number of sub-clusters.
The algorithm starts by applying the K-Means algorithm to cluster the data into K distinct clusters. The K-Means algorithm makes use of C_{alg} to pick the initial centroid for the K cluster. In addition, the maximum number of iterations forces the K-means algorithm to halt if the centroids of the clusters do not converge to the clusters' mean. After clustering the current data, each cluster forms a node. Then, the algorithm is applied recursively on each cluster (node) until the size of the cluster becomes less than K. According to [2], the computational complexity of the construction algorithm is $O(ndKI_{max}(\log n/\log k))$. $O(ndKI_{max})$ is the complexity of clustering a level in a tree. And there is $(\log n/\log K)$ level. Combining the two results yields $O(ndKI_{max}(\log n/\log k))$ time complexity.

2.3.2 Search Algorithm

The search algorithm maintains two priority queues PQ and R. The first one is ordered in increasing order from the query point to the center of the unexplored clusters. The second one sorts the retrieved data point in increasing order of the distance to the query point. The search algorithm starts exploring the tree from the root to reach the closest leaf. At the leaf node, the algorithm adds all the stored points in the leaf to R. And At each node, it adds the unexplored clusters to PQ. While PQ is not empty and while the collected points in R are less than a given limit. The algorithm pulls nodes from PQ and explores them. At the end, it returns the top K points in R.
According to [2], the complexity of the search algorithm is $O(Ld(\log n/\log K))$. During each top-down traversal, the algorithm needs to check $O(\log n/\log k)$ inner nodes. The algorithm requires L/k top-down traversal where L is the maximum number of points collected by R. In addition, while traversing the tree the algorithm computes at each level $O(Kd)$ distance where d is the dimension of the space. Combining all these results yields to $O(Ld(\log n/\log k))$ complexity.

3 Implementation Description

The project is implemented using C++14. It does not rely on any external library other than the standard C++ library. The implementation makes use of the object-oriented paradigm (OOP). In addition, the implementation is pointer based implementation. However, it can be done without pointers which save space. Moreover, the implementation does not replicate the actual data in nodes but it uses the indices in order to save space. Furthermore, the priority queues are implemented as Red-Black Tree according to C++ specification. However, a different choice of these data structures and implementation techniques may lead to a better result concerning the performance and the storage space.

Concerning the structure of the implementation, there is one main component that captures the base structure (the Tree structure) for the three main algorithms. In addition, each algorithm consists of one component that contains its construction and search algorithms.

Considering the implementation of the algorithms, it follows the description of [2] and [3] (already mentioned in the previous section). The algorithms are implemented recursively as they are described.

4 Experiments and Results

In order to study the mentioned data structures, we performed many experiments. These experiments report the effects of changing the parameters on the performance and the precision of the search and the construction algorithms. The performance is measured in time(seconds). The precision is measured according to the following formula.

$$d(p, q) \leq (1 + \epsilon)d(P_k, q)$$

where q is the query point, d is the distance function. For the Priority k-Means Tree testing, P_k is the k^{th} furthest point from the exact answer returned by the linear search. For the Randomised K-d Trees testing, P_k is the correct answer returned by the linear search. Therefore, p is a correct solution if it satisfies the above inequality. For the K-d Tree, ϵ is fixed to zero since K-d Tree is an exact algorithm in contrast to the Randomised K-d Tree and the Priority K-Means Tree where epsilon is chosen between zero and one. It is important to notice that there is no single "ideal" algorithm, each algorithm performs differently in different scenarios. Thus, one of our goal is to evaluate the performance with respect to both changing the data set parameters (dimension, size, and structure) and the algorithms' parameters.

4.1 K-d Tree

To assess the performance of the K-d Tree, the performance of the K-d Tree is compared to the performance of the naive algorithm (Linear search). First, the dimension is fixed to two and the size of the data is variable. The dataset in the following experiment is sampled at random from a uniform distribution (i.e we sampled the coordinates of the vector uniformly at random).

The result illustrated in Figure 1 shows that the K-d Tree is unbeaten independently from the size of the data (with fixed dimension equals to two). However, fixing the data size and changing the dimensionality of the dataset shows a significant change in the K-d Tree's behaviour. The result illustrated in Figure 2 shows how the K-d Tree becomes similar (even worse) than the linear search after the dimension of the data exceeds ten. This result makes the developments of new data structure necessary because most of the nearest search problems are performed over data in high dimension.

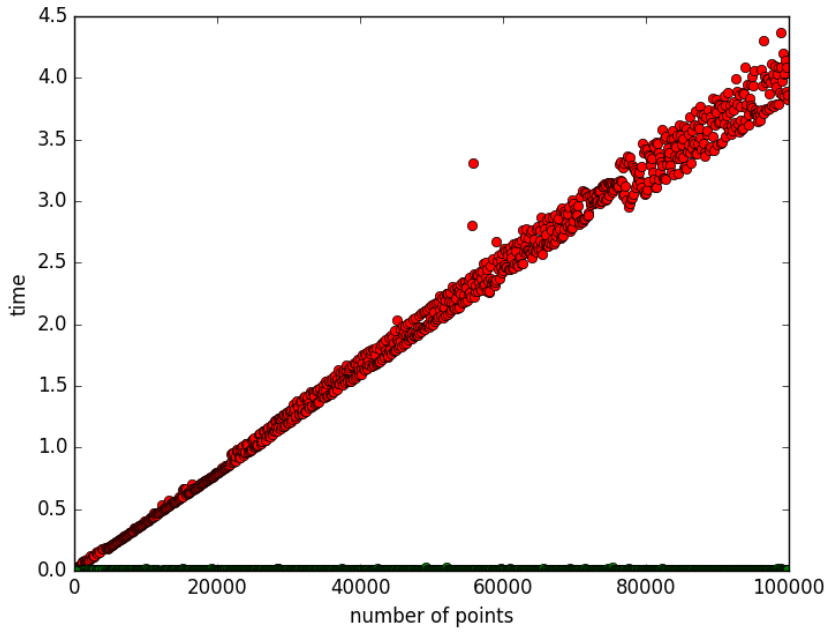


Figure 1: The change of the performance in seconds with respect to the size of the dataset (The K-d Tree is represented in green and the naive algorithm in red).

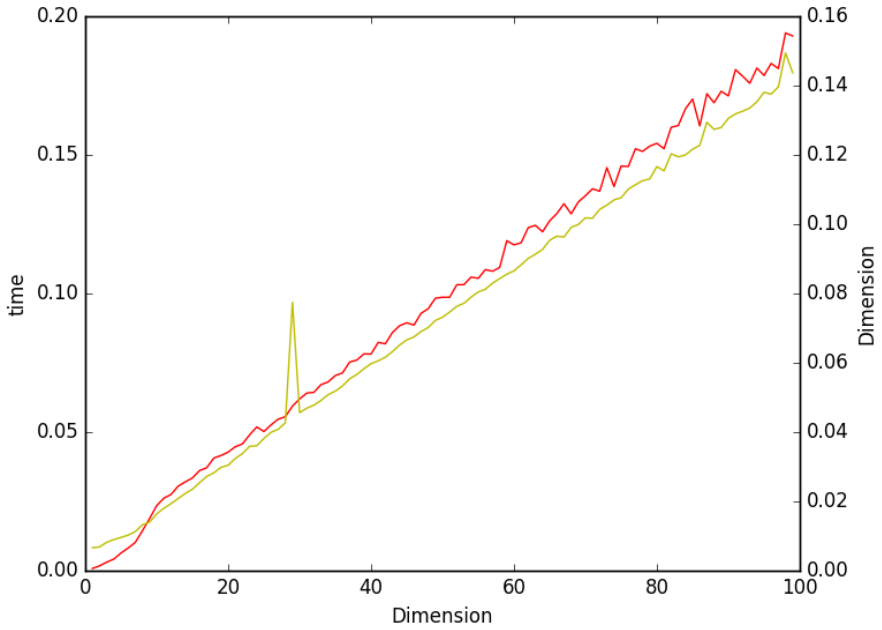


Figure 2: The change of the performance in seconds with respect to the dimensionality of the data. The k-d Tree performance is illustrated in red, and the linear search performance is illustrated in yellow.

4.2 Randomised K-d Tree

To have a better understanding of the Randomised K-d Tree, the following experiment measures the effects of changing the parameters on its performance and precision. Since the Randomised K-d Tree algorithms consist of multiples trees, changing the number of trees is an important factor to understand the behaviour of this data structure. In addition, this experiment is done over SIFT (The scale-invariant feature transform) with fixed dimension equals to 128 and fixed data set of size. From Figure 3, one can infer that increasing the number of trees improves the performance of the search algorithm. However, after certain value increasing this number becomes ineffective. In addition, a separate experiment yielded that the number of trees does not have a significant effect on the precision of the search. Furthermore, another experiment measured the effect of the

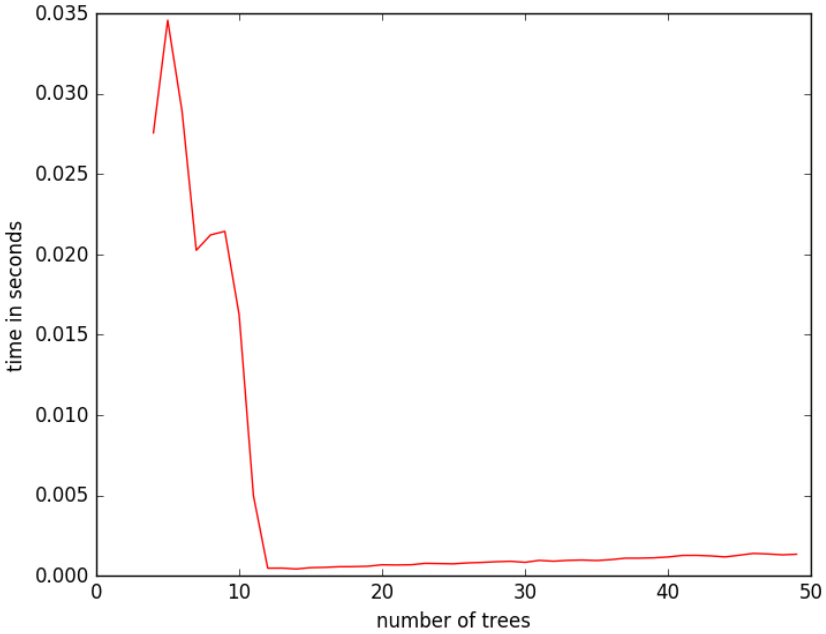
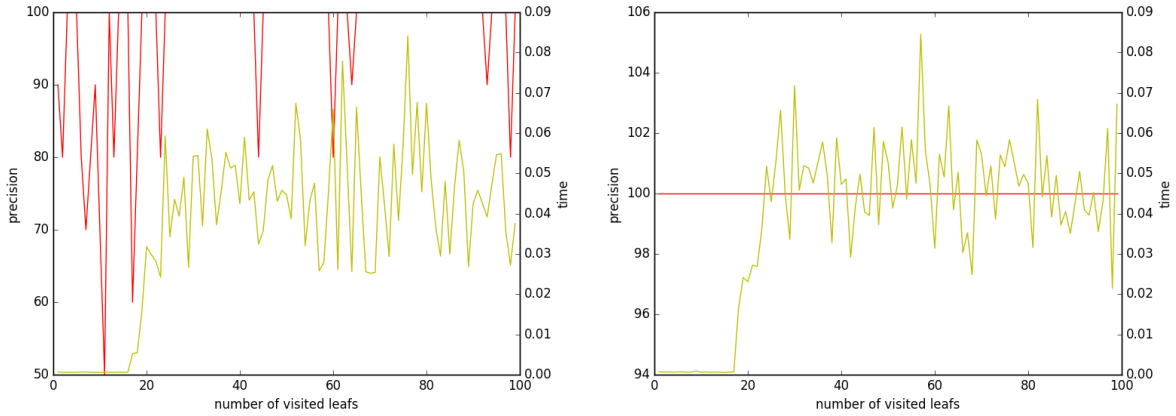


Figure 3: The change of the performance in seconds of Randomised K-d Tree with respect to the number of tree.

number of visited leaves on the performance and on the precision of the search results. we did this experiment for epsilon equals to 0.3 and 0.7. The following graphs (Figure 4) illustrate the results where the precision is measured in percentage (the red plot) and the performance is measured in seconds (the yellow plot). One can notice, that increasing the number of visited leaves decreases the performance. But, it improves the precision as it is shown in figure 4(a) (the fluctuation of the precision becomes steadier). This observation (results) confirms the claim in [2], the number of visited leaves across all the tree determines the degree of the precision. Moreover, increasing epsilon from 0.3 to 0.7 yields to a significant improvement in the precision without affecting the performance.



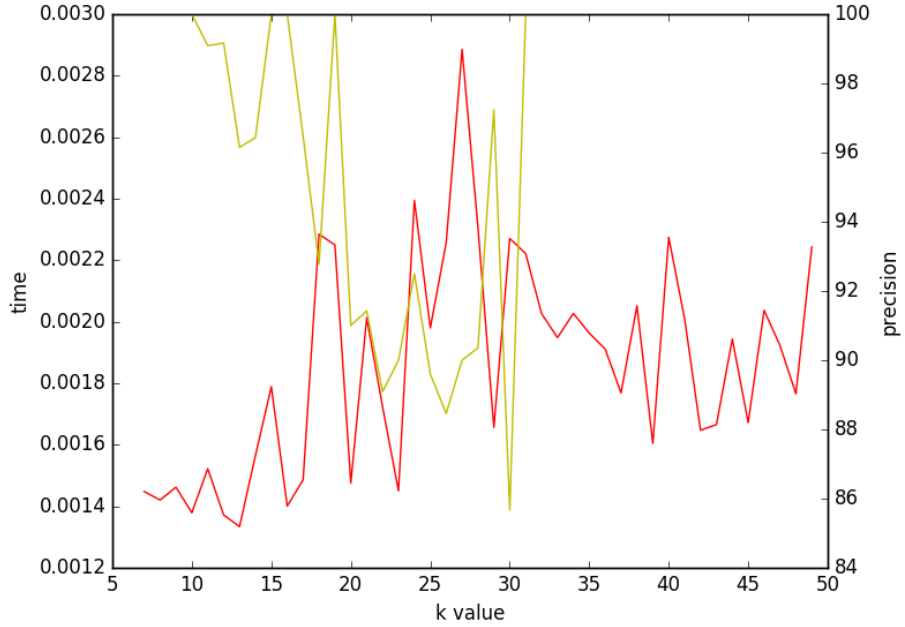
(a) Epsilon = 0.3

(b) Epsilon = 0.7

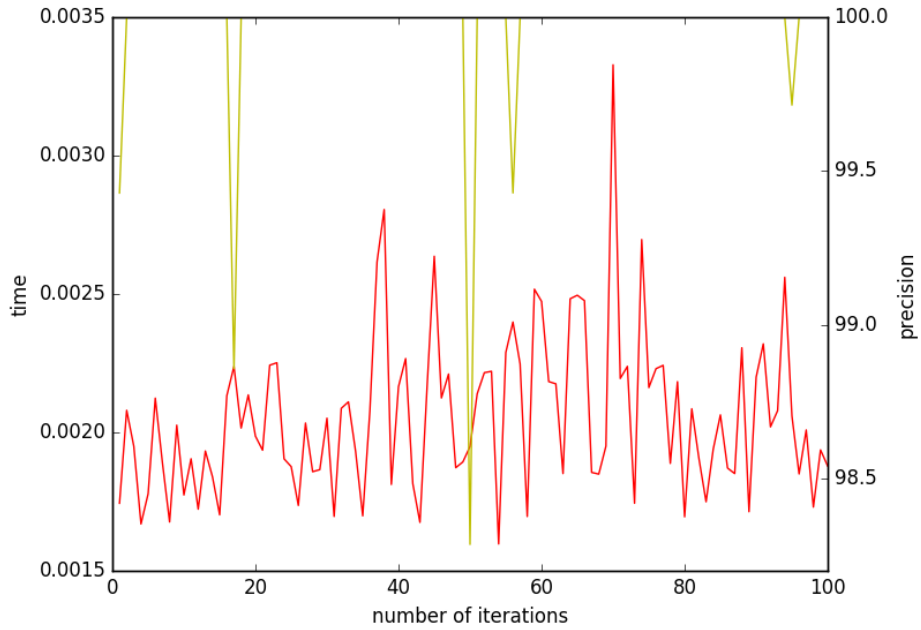
Figure 4

4.3 Priority K-Means Tree

Similarly to the previous experiments, the following one measures the effect changing the parameters on the precision and the performance of Priority K-Means Tree. This data structure has three important parameters K, I_{max}, and L. First, we measured the impact of changing the value of K. Figure five (a) illustrates the results. It is clear that the performance decreases slowly as K increases. However, the precision does not have a regular pattern, but for a certain K, it takes a maximal value and remains constant. Concerning the number of iterations, One can notice that this parameter does not have



(a) The change of the precision (in yellow) and the performance (in red) with respect of the change of K



(b) The change of the precision (in yellow) and the performance (in red) with respect of the change I_{max}

Figure 5

a significant impact on the performance and the precision. Figure five (b) shows that the patterns in the performance and in the precision do not change in a regular manner with the change in I_{max}.
Moreover, the experiment tests the effect of changing L, the maximum number of collect

points. As Figure 6 shows, increasing L has a significant impact on the precision. And in the same time increasing L has no significant impact on the performance.

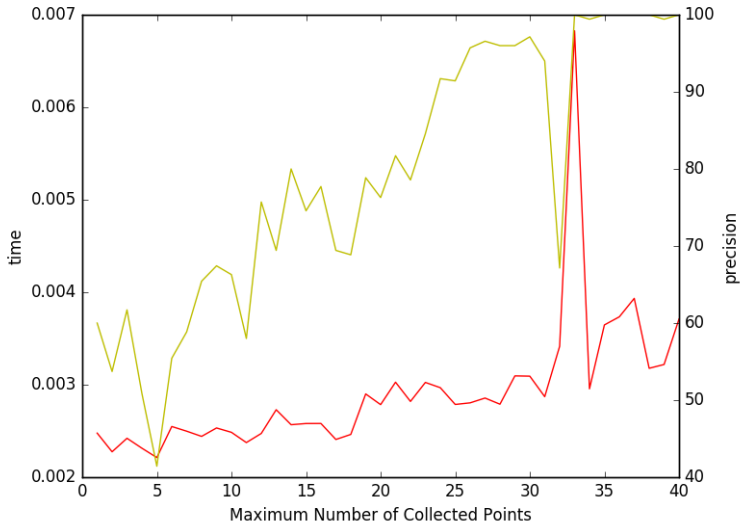


Figure 6: The change of the precision and performance with respect to the maximum number of collected points. (Precision in Yellow, Performance in Red)

4.4 General Comparison of the Algorithm

This section will present a general comparison for the main three algorithms and linear search. The comparison is between of the search and construction algorithms for each data structure. The experiment is done over SIFT (Vector with 128 dimension each) as in [2]. The comparison of the construction algorithms is presented in Figure 7. The K-d construction of the k-d Tree is obviously the fastest. However, the performance of construction of the Randomised K-d Tree depends on the number of trees, in this case it is ten. Increasing this number will affect negatively the construction time performance but it will improve the search performance. Concerning the search algorithms' performance, the K-d Tree and the linear search have almost the same performance (Figure 8) because of the dimensionality of the data. The performance of the K-d Tree decreases with the increase of the dimensionality of the data as it is illustrated in Figure 2. However, the Randomised K-d Tree has the best performance in term of accuracy and speed, then it comes the Priority K-Means Tree as it shown in Figure 8. Furthermore, increasing epsilon from 0.3 to 0.7 increases the precision from 87% to 99% for the Randomised K-tree, and increases the precision from 70 to 80% for the Priority K-Means Tree. In addition, increasing the K parameter for the Priority K-Means tree from 4 to 16 with epsilon fixed to 0.7 increases the precision from 80 % to 99%.

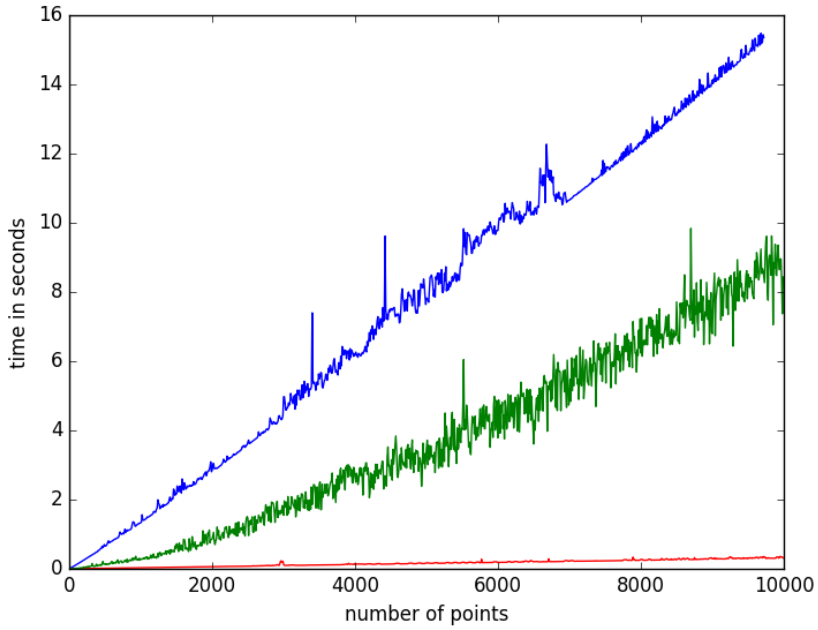


Figure 7: The change in the performance of the construction algorithms with respect to the change in the number of points. The performance of the k-d tree is plotted in red, the Randomised k-d tree (10 trees) is plotted in blue, and the Priority k-Means Tree is plotted in green.

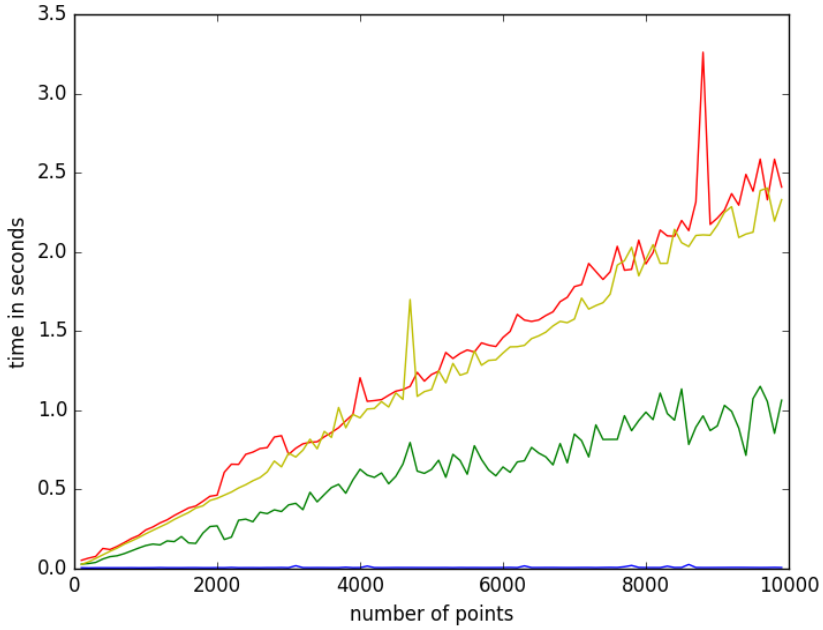


Figure 8: The performance of the search algorithms with respect to the number of points. The performance of the K-d Tree is plotted in res, the linear search in yellow, Randomised K-d Tree in blue, and Priority k-Means Tree in yellow.

5 Conclusion

In conclusion, Nearest Neighbour Search is one of the most important problems in many fields. Solving this problem in low dimensions can be done efficiently using K-d Tree. However, increasing the dimensionality of this data renders this data structure ineffective. Then the use of different data structure such as Randomised K-d Tree and Priority K-Means Tree becomes essential. But these data structures are affected by many parameters, therefore this project aims to discover the effects of these parameters and to provide the following guidelines to a better use of the data structures. A common observation for the two data structures is increasing epsilon always yields better results this means that we are almost very close to the correct answer by a factor of two in the worst case.

For the Randomised K-d Tree,

- Increasing the number of trees will lead to a better performance. However, after a certain threshold, increasing the number of trees becomes invective this is due to the increase in the use of memory [2].
- Increasing the number of visited leaves will lead to a better precision without a serious effect on the performance.

For the Priority K-Means Tree,

- Increasing the branching factor K affects slightly the performance.
- Increasing the Branching factor improves the precision.
- Increasing the number of collecting points L leads to a significant improvement in the precision with no significant effects on the precision.
- The number of iteration for the K-Means algorithm has no notable effect.

References

[1] D.G. Lowe, "Distinctive image features from scale-invariant key-point," Int. J. Comput. Vis., vol 60, no. 2, pp. 91-110, 2004.

[2] Marius Muja, David G. Lowe. "Scalable Nearest Neighbor Algorithms for High Dimensional Data", IEEE PAMI, Vol. 36, No. 11. (2014), pp. 2227-2240.

[3] Shakhnarovich, G., Darrell, T. & Indyk, P. (2006) Nearest-Neighbor Methods In Learning And Vision Theory and Practice. MIT Press

[4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

[5] De Berg, M. Cheong, O., Kreveld, M., Overmars, M. (2008). Computational Geometry Algorithms and Applications (3rd ed.). Springer.

[6] C. Silpa-Anan and R. Hartley, "Optimized KD-Trees for fast image descriptor matching," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2008, pp. 1-8.

[7] K. Fukunaga and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Trans. Comput., vol. C-24, no.7, pp.750-753*, Jul. 1975.

[8] S. Brin, "Nearest neighbor search in large metric spaces," in *Proc. 21th Int. Conf. Very Large Data Bases*, 1995, pp. 574-584.

[9] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," *in Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, 2006, pp. 2161-2168.