Query-based improvement procedure and self-adaptive graph construction algorithm for approximate nearest neighbor search

Alexander Ponomarenko

National Research University Higher School of Economics
Nizhny Novgorod, Russia, aponomarenko@hse.ru

Abstract. The nearest neighbor search problem is well known since 60s. Many approaches have been proposed. One is to build a graph over the set of objects from a given database and use a greedy walk as a basis for a search algorithm. If the greedy walk has an ability to find the nearest neighbor in the graph starting from any vertex with a small number of steps, such a graph is called a navigable small world. In this paper we propose a new algorithm for building graphs with navigable small world properties. The main advantage of the proposed algorithm is that it is free from input parameters and has an ability to adapt on the fly to any changes in the distribution of data. The algorithm is based on the idea of removing local minimums by adding new edges. We realize this idea to improve search properties of the structure by using the set of queries in the execution stage. An empirical study of the proposed algorithm and comparison with previous works are reported in the paper.

Keywords: Nearest neighbor search; non-metric search; approximate search;

1 Introduction

The nearest neighbor search problem has naturally appeared in many fields of science. The problem is formulated as follows. Let D be a domain, $X \subset D$ be a finite set of objects (database), $d : D \times D \to R^{[0;+\infty)}$ be a distance function. We need to preprocess X in such a way that for a given query $q \in D$ the k-closest objects from X can be found as fast as possible. Many methods have been proposed for searching an exact nearest neighbor as well as an approximate. Some recent methods are [6], [7]. Also a good overview of methods for the exact nearest neighbor search can be found in [1] and an empirical comparison of several approximate state of the art methods can be found in [3]. One of the recent promising approaches for the approximate nearest neighbor search problem is to build a graph $G(X, E)$ over the set of objects X and use the greedy walk algorithm as a base for the search algorithm. Thus, the search of k-closest objects takes the form of the search of vertices in the graph G, where each object from the set X uniquely corresponds to a different vertex of the graph. Several works based on this approach have been proposed [2,4,5]. Methods [2, 5] are based on the idea of connecting a set of approximate Voronoi regions into a network where one region corresponds to one data point. Voronoi regions are approximated by k-closest points together with a set of points for which the current point is one of the k-closest. The main drawback of this method is that it requires initial setting of the parameter k which in turn requires a priori knowledge about the properties of the input data set (a small value of the parameter leads to a small accuracy of search; too large value causes excessive search complexity and needs more memory to store the graph). In this paper we explore an idea how to form a graph without any a priori knowledge of input data. Instead of finding k-closest neighbors we explicitly try to build a graph in such a way that the greedy walk algorithm is able to find a new data point starting from any other random vertex of the current graph G. Moreover we propose a way to improve search properties of the structure with a similar idea by using the set of incoming queries.

2 Greedy Walk Algorithm

As mentioned above we use an approach where each object from the set X is uniquely mapped to a different vertices of the graph G. Thus, the search of k-closest objects takes the form of the search of vertices in the graph. As a search algorithm we suggest to use a simple greedy walk algorithm. The pseudo code of the greedy walk algorithm is presented below. The greedy walk algorithm is quite simple. It starts from some vertex v_{start}; calculates the distance between the query and each neighbor of v_{start}; goes to the vertex v_{curr} for which the distance is minimal; calculates the distance d between the query q and each object from neighbors $N(v_{curr})$ of vertex v_{curr} and so on, until the algorithm cannot improve the distance to the query. The vertices in which the greedy walk stops we call “local minimums”. Note that the `Greedy_Walk` algorithm with local minimum also returns the set P. This set P contains all vertices that were contacted during a search (lines 2 and 5).

```
Greedy_Walk($q \in D$, $G(V,E)$, $v_{start} \in V$)
1  $v_{curr} \leftarrow v_{start}$
2  $P \leftarrow P \cup N(v_{curr})$
3  while $\min_{x \in N(v_{curr})}(d(q,x)) < d(q,v_{curr})$ do
4      $v_{curr} \leftarrow \arg\min_{x \in N(v_{curr})}(d(q,x))$
5      $P \leftarrow P \cup N(v_{curr})$
6  end while
7  return $v_{curr}$, $P$
```

3 Insertion Algorithm

The insertion algorithm is based on the idea, that each time when we insert a new vertex to the graph G, we can explicitly check the possibility of finding the new vertex starting from other vertices by the greedy algorithm. We also exploit a trick of keeping alive old links. Since time this links starting to be used as long links by the greedy search algorithm. So, this trick allows us to produce the navigable small world property for our graph.
At first we will describe the `Get_Local_Minimums` function. It returns the set of local minimums in which randomly started greedy walks stop. The `Get_Local_Minimums` algorithm has parameter τ which means how many times we allow a randomly started greedy walk not to bring a new local minimum. Also the `Get_Local_Minimums` function returns the set P which we consider as global view of the all greedy walks.

```
Get_Local_Minimums( $q \in D$ , $G(V,E)$ , $\tau \in \mathrm{N}$ )
01 $L \leftarrow \{\}$ ; $\tau' \leftarrow 0$
02  while $\tau' < \tau$ do
03     $v_{start} \leftarrow$ Random( $V$ )//put to $v_{start}$ random vertex from $V$
04     $v, P \leftarrow$ Greedy_Walk( $q$ , $G$ , $v_{start}$ )
05     if $v \notin L$ then $\tau' \leftarrow 0$ ; $L \leftarrow L \cup v$
08     else $\tau' \leftarrow \tau' + 1$
09  end while
11  return $L$ , $P$
```

```
Insert_By_Repairing( $x \in X$ , $G(V,E)$ , $\tau \in \mathrm{N}$ )
01  $V \leftarrow V \cup x$ ; $P \leftarrow \{\}$
02  $L, P \leftarrow$ Get_Local_Minimums( $x$ , $G$ , $\tau$ )
03  $P \leftarrow P \cup x$
04  for each $z \in L$ do
05     $P' \leftarrow \{y \in P : d(y,x) < d(z,x)\}$
06     $x' \leftarrow \arg\min_{y \in P'}(d(z,y))$
07     $E \leftarrow E \cup (x', z) \cup (z, x')$
08  end for
```

Now we ready to describe `Insert_By_Repairing` procedure. At first, we collect in the set L all local minimums (line 02) which we found by a number of randomly started greedy walks. After that, we remove these local minimums in the following way. For every local minimum z in the set of all viewed vertices (points) P we select the vertex x' such that the distance from x' to x is less than the distance from z to x, and the distance from z to x' is minimal. So, we make the local minimum z

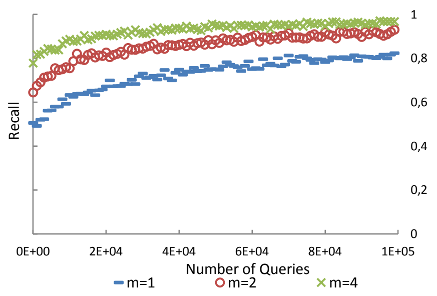

Fig. 1. Recall after Improving by queries applying `Repair_By_Query` procedure

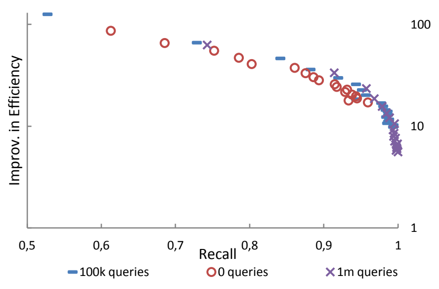

Fig. 2. The overall efficiency of the structure after applying `Repair_By_Query` procedure

to be able to route the greedy walk to the vertex x' which is closer to the destination point x by adding edge between z and x'.
Finally, we present procedure `Add_All` which builds a data structure over the set X. It sequentially selects a random object from the set X and inserts it to the graph by running the `Insert_By_Repairing` procedure.

```
Add_All($X \subset D$, $\tau \in \mathrm{N}$)
01   $G \leftarrow (\{\},\{\})$
02   while $X \neq \{\}$ do
03      $x \leftarrow$Random($X$); $X \leftarrow X \setminus x$
04      $G \leftarrow$Insert_By_Repairing($x$, $G$, $\tau$)
05   end while
```

4 Improvement Based on Queries

The idea of repairing by removing local minimums can be extended to improving the search properties of the structure by queries at the query execution stage. In the procedure `Repair_By_Query` we do the same as in `Insert_By_Repairing` procedure with only one exception. We search the local minimums relative to the query object q instead of the inserted one.

```
Repair_By_Query($q \in D$, $G(V,E)$, $\tau \in \mathrm{N}$)
01   $L, P \leftarrow$Get_Local_Minimums($q$, $G$, $\tau$)
02   $x \leftarrow \arg\min_{y \in L}(d(y,q))$
03   for each $z \in L \setminus x$ do
04      $P' \leftarrow \{y \in P : d(y,q) < d(z,q)\}$
05      $x' \leftarrow \arg\min_{y \in P'}(d(z,y))$
06      $E \leftarrow E \cup (x',z) \cup (z,x')$
07   end for
```

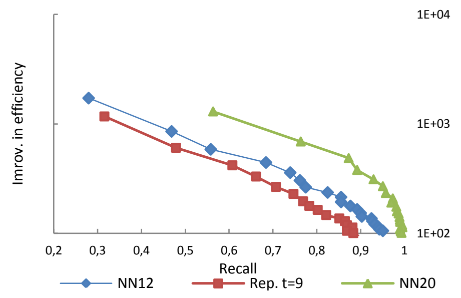

Fig. 3. Comparison with NN [5]

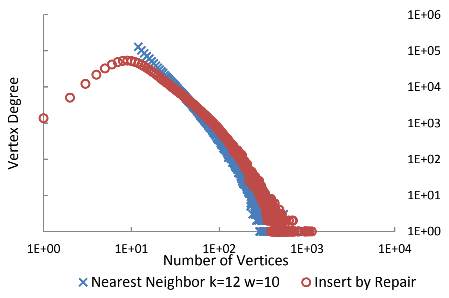

Fig. 4. Vertex Degree Distribution

5 Simulations

5.1 Improvement based on queries

We have performed simulations in order to verify the idea of improving the data structure using the stream of incoming queries. We have built a graph by `Add_All` algorithm with parameter $\tau = 9$ on the set of 10,000 points uniformly distributed in the 30-dimensional unit hyper cube. After that, we generate 100 times 1000 queries with the same distribution. Each time after applying 1000 times the `Re-pair_By_Query` procedure we have measured the value of recall for searching 5 nearest neighbors. As a search algorithm we have used `Multi_Search` algorithm proposed in [5]. This algorithm uses the sequences of greedy based searches started from a random vertex and selects the best results. The parameter “m” is the number of searches. We have used m = 1, 2, and 4 correspondingly. As can been seen from Fig. 1 the accuracy of each search has increased after applying the `Repair_By_Query` procedure.
Also we have measured how the overall efficiency of the structure has changed when applying `Repair_By_Query` procedure. We have measured the values of improvement in efficiency (how many times less the algorithm needs to calculate distances than the exhaustive search) and values of recall after applying `Re-pair_By_Query` procedure 100,000 and 1,000,000 times to the structure built over the set of 100,000 30-dimensional points by `Insert_By_Repairing` algorithm (Fig. 2). Unfortunately, it has not made a significant contribution to the overall efficiency of the structure.

5.2 Insertion by Repairing

We have made the comparison of the search properties of the graphs produced by `Insert_By_Repairing` (parameter $\tau = 9$) algorithm with the algorithm of connecting with k-nearest neighbors [5]. The parameters were w=10; k=12 and 20 accordingly. The parameter “w” is the number of searches used by `Multi_Search` procedure at the construction stage As a data set we have used one million random points uniformly distributed in the unit 30-dimensional hyper cube. As a distance function we use L_2. The result of comparison is presented in Fig. 3 and a valuation of the distribution of the vertex degrees is presented in Fig. 4.

6 Conclusion

In this paper we have explored the idea of using information about local minimums during the insertion and during the search. We have proposed an algorithm which uses this information to improve the accuracy of the search procedure. Also based on this idea we have proposed a new graph construction algorithm. Despite that the proposed algorithm cannot outperform the algorithm of connecting with k-nearest neighbors, it demonstrates the idea that the information about local minimums can be used for data insertion. Moreover we suppose that this idea can be used for tuning parameters of other data insertion algorithms, for example for tuning the parameter k in the algorithm of connection with k-nearest neighbors [5].

7 Acknowledgements

The work was conducted at National Research University Higher School of Economics and supported by RSF grant 14-41-00039.

8 References

1. Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. "Searching in metric spaces." ACM computing surveys (CSUR) 33, no. 3 (2001): 273-321.
2. Malkov, Yury, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces." In Similarity Search and Applications, pp. 132-147. Springer Berlin Heidelberg, 2012.
3. Alexander Ponomarenko, Nikita Avrelin, Bilegsaikhan Naidan, and Leonid Boytsov. "Comparative Analysis of Data Structures for Approximate Nearest Neighbor Search." In DATA ANALYTICS 2014, The Third International Conference on Data Analytics, pp. 125-130. 2014.
4. Lifshits, Yury, 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, pp. 318-326. Society for Industrial and Applied Mathematics, 2009.
5. Malkov, Yury, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. "Approximate nearest neighbor algorithm based on navigable small world graphs." Information Systems 45 (2014): 61-68.
6. Chávez, E., M. Graff, G. Navarro, and E. S. Téllez. "Near neighbor searching with K nearest references." Information Systems 51 (2015): 43-61.
7. Skopal, Tomáš. "Unified framework for fast exact and approximate search in dissimilarity spaces." ACM Transactions on Database Systems (TODS) 32, no. 4 (2007): 29.