# Fast Approximate Nearest-Neighbor Search with $k$-Nearest Neighbor Graph

**Kiana Hajebi and Yasin Abbasi-Yadkori and Hossein Shahbazi and Hong Zhang**
Department of Computing Science
University of Alberta
{hajebi, abbasiya, shahbazi, hzhang}@ualberta.ca

## Abstract

We introduce a new nearest neighbor search algorithm. The algorithm builds a nearest neighbor graph in an offline phase and when queried with a new point, performs hill-climbing starting from a randomly sampled node of the graph. We provide theoretical guarantees for the accuracy and the computational complexity and empirically show the effectiveness of this algorithm.

## 1 Introduction

Nearest neighbor (NN) search is a fundamental problem in computer vision, image retrieval, data mining, etc. The problem can be formulated as

$$X_* = \operatorname*{argmin}_{X \in \mathcal{D}} \rho(X, Q),$$

where $\mathcal{D} = \{X_1, \ldots, X_n\} \subset \mathbb{R}^d$ is a dataset, $Q$ is a query, and $\rho$ is a distance measure.

A naive solution to the NN search problem is to compute the distance from the query to every single point in the dataset and return the closest one. This approach is called the linear search method and is guaranteed to find the exact nearest neighbor. The computational complexity of the linear search method is $O(nd)$, where $n$ is the size of the dataset and $d$ is the dimensionality. This complexity can be expensive for large datasets. The difficulty of finding the exact nearest neighbor has led to the development of the approximate nearest neighbor search algorithms [Beis and Lowe, 1997; Indyk and Motwani, 1998].

In this paper, we propose a graph-based approach for the approximate NN search problem. We build a $k$-nearest neighbor ($k$-NN) graph and perform a greedy search on the graph to find the closest node to the query.

The rest of the paper is organized as follows. Section 2 briefly reviews the prominent NN search methods and those that use a $k$-NN graph or greedy search to perform the NN search. In Section 3.2, we introduce the Graph Nearest Neighbor Search algorithm (GNNS) and analyze its performance. In Section 4, we experimentally compare the GNNS algorithm with the KD-tree and LSH methods on a real-world dataset as well as a synthetically generated dataset.

## 2 Related works

There are a number of papers that use hill-climbing or $k$-NN graphs for nearest neighbor search, but to the best of our knowledge, using hill-climbing on $k$-NN graphs is a new idea.

Papadias [2000] assumes that each point (e.g., an image) is specified as a collection of components (e.g., objects). Each point has the form of $X_i = (V_1, \ldots, V_m)$, where each $V_j$ is an object and can take values from a finite set (e.g., a set of squares of different sizes). The objective is to find the point in the dataset that has the closest configuration to the query $Q$. Papadias [2000] says $X_i$ and $X_j$ are neighbors if one can be converted to the other by changing the value of one of its variables. Then several heuristics to perform hill-climbing on such a graph are proposed [Papadias, 2000].

Paredes and Chvez [2005] aim at minimizing the number of distance computations during the nearest neighbor search. A $k$-NN graph is built from dataset points and when queried with a new point, the graph is used to estimate the distance of all points to the query, using the fact that the shortest path between two nodes is an upper bound on the distance between them. Using the upper and lower bound estimates, Paredes and Chvez [2005] eliminate points that are far away from the query point and exhaustively search in the remaining dataset.

Lifshits and Zhang [2009] define a *visibility graph* and then perform nearest neighbor search by a greedy routing over the graph. This is a similar approach to our method, with two differences. First, Lifshits and Zhang [2009] search over the visibility graph, while we search on the $k$-NN graph. $k$-NN graphs are popular data structures that are used in outlier detection, VLSI design, pattern recognition and many other applications [Paredes and Chvez, 2005]. The second difference is that Lifshits and Zhang [2009] make the following strong assumption about the dataset.

**Assumption A1** Sort the points in the dataset according to their closeness to a point $U$. Let $r_U(V)$ be the rank of $V$ in this sorted list. Define $R(X, Y) = \max\{r_X(Y), r_Y(X)\}$. Then it holds that

$$R(X, Z) \leq C(R(X, Y) + R(Y, Z)),$$

where $C$ is a constant.

Under Assumption A1, Lifshits and Zhang [2009] prove that the computational complexity of the construction of the visibility graph and the nearest neighbor search are $O(poly(C) n log^2 n)$ and $O(C^4 \log^2 n)$, respectively.

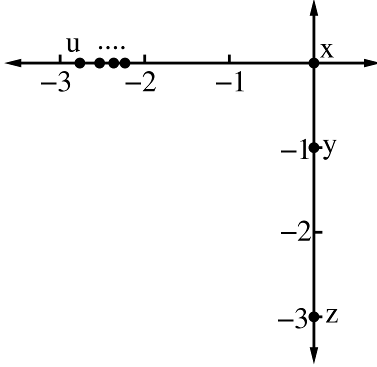


Figure 1: A counterexample to Assumption A1.

Assumption A1 does not hold in general. For instance, consider the simple 2-dimensional example shown in Figure 1, where we use the Euclidean distance as the metric. In this example, we have $R(x, y) = 1$, $R(y, z) = 2$, and depending on the number of points on the line between $(-2, 0)$ and $(-3, 0)$, $R(x, z)$ can be arbitrarily large. Thus, there is no constant $C$ that satisfies the inequality in Assumption A1, or it can be arbitrarily large.

In the following two subsections, we briefly explain two popular methods for approximate nearest neighbor search: KD-trees and Locality Sensitive Hashing (LSH). Our proposed approximate $k$-NN method will be compared against these two methods in the evaluation section.

### 2.1 Locality Sensitive Hashing (LSH)

LSH [Indyk and Motwani, 1998] uses several hash functions of the same type to create a hash value for each point of the dataset. Each function reduces the dimensionality of the data by projection onto random vectors. The data is then partitioned into bins by a uniform grid. Since the number of bins is still too high, a second hashing step is performed to obtain a smaller hash value. At query time, the query point is mapped using the hash functions and all the datapoints that are in the same bin as the query point are returned as candidates. The final nearest neighbors are selected by a linear search through candidate datapoints.

### 2.2 KD-tree

A KD-tree [Bentley, 1980; Friedman *et al.*, 1977] partitions the space by hyperplanes that are perpendicular to the coordinate axes. At the root of the tree a hyperplane orthogonal to one of the dimensions splits the data into two halves according to some splitting value. Each half is recursively partitioned into two halves with a hyperplane through a different dimension. Partitioning stops after $logn$ levels so that the bottom of the tree each leaf node corresponds to one of the datapoints. The splitting values at each level are stored in the nodes. The query point is then compared to the splitting value at each node while traversing the tree from root to leaf to find the nearest neighbor. Since the leaf point is not necessarily the nearest neighbor, to find approximate nearest neighbors, a backtrack step from the leaf node is performed and the points that are closer to the query point in the tree are examined. In our experiments, instead of simple backtracking, we use Best Bin First (BBF) heuristic [Beis and Lowe, 1997] to perform the search faster. In BBF one maintains a sorted queue of nodes that have been visited and expands the bins that are closer to query point first.

Further, we use the randomized KD-tree [Muja and Lowe, 2009], where a set of KD-trees are created and queried instead of a single tree. In each random KD-tree, the datapoints are rotated randomly, so that the initial choice of axes does not affect the points that are retrieved.. At query time, the same rotation is applied to the query point before searching each tree. The union of the points returned by all KD-trees is the candidate list. Similar to LSH, the best nearest neighbors are selected using linear search in the candidate list.

## 3 The Graph Nearest Neighbor Search Algorithm (GNNS)

We build a $k$-NN graph in an offline phase and when queried with a new point, we perform hill-climbing starting from a randomly sampled node of the graph. We explain the construction of $k$-NN graphs in Section 3.1. The GNNS Algorithm is explained in Section 3.2.

### 3.1 $k$-NN Graph Construction

A $k$-NN graph is a directed graph $\mathcal{G} = (\mathcal{D}, \mathcal{E})$, where $\mathcal{D}$ is the set of nodes (i.e. datapoints) and $\mathcal{E}$ is the set of links. Node $X_i$ is connected to node $X_j$ if $X_j$ is one of the $k$-NNs of $X_i$. The computational complexity of the naive construction of this graph is $O(dn^2)$, but more efficient methods exist [Chen *et al.*, 2009; Vaidya, 1989; Connor and Kumar, 2010].

The choice of $k$ is crucial to have a good performance. A small $k$ makes the graph too sparse or disconnected so that the hill-climbing method frequently gets stuck in local minimum. Choosing a big $k$ gives more flexibility during the runtime, but consumes more memory and makes the offline graph construction more expensive.

### 3.2 Approximate $K$-Nearest Neighbor Search

The GNNS Algorithm, which is basically a best-first search method to solve the $K$-nearest neighbor search problem, is shown in Table 1. Throughout this paper, we use capital $K$ to indicate the number of queried neighbors, and small $k$ to indicate the number of neigbors to each point in the $k$-nearest neighbor graph. Starting from a randomly chosen node from the $k$-NN graph, the algorithm replaces the current node $Y_{t-1}$ by the neighbor that is closest to the query:

$$Y_t = \underset{Y \in N(Y_{t-1}, E, \mathcal{G})}{\operatorname{argmin}} \rho(Y, Q),$$

where $N(Y, E, \mathcal{G})$ returns the first $E \leq k$ neighbors of $Y$ in $\mathcal{G}$, and $\rho$ is a distance measure (we use Euclidean distance in our experiments). The algorithm terminates after a fixed number of greedy moves $T$. If $K = 1$, we can alternatively terminate when the algorithm reaches a node that is closer to the query than its best neighbor. At termination, the current best $K$ nodes are returned as the $K$-nearest neighbors to the

**Input**: a $k$-NN graph $\mathcal{G} = (\mathcal{D}, \mathcal{E})$, a query point $Q$, the number of required nearest neighbors $K$, the number of random restarts $R$, the number of greedy steps $T$, and the number of expansions $E$.
$\rho$ is a distance function. $N(Y, E, \mathcal{G})$ returns the first $E$ neighbors of node $Y$ in $\mathcal{G}$.
$\mathcal{S} = \{\}$.
$\mathcal{U} = \{\}$.
$Z = X_1$.
**for** $r = 1, \dots, R$ **do**
  $Y_0$: a point drawn randomly from a uniform distribution over $\mathcal{D}$.
  **for** $t = 1, \dots, T$ **do**
    $Y_t = \mathrm{argmin}_{Y \in N(Y_{t-1}, E, \mathcal{G})}\, \rho(Y, Q)$.
    $\mathcal{S} = \mathcal{S} \bigcup N(Y_{t-1}, E, \mathcal{G})$.
    $\mathcal{U} = \mathcal{U} \bigcup \{\rho(Y, Q) : \ Y \in N(Y_{t-1}, E, \mathcal{G})\}$.
  **end for**
**end for**
Sort $\mathcal{U}$, pick the first $K$ elements, and return the corresponding elements in $\mathcal{S}$.

Table 1: The Graph Nearest Neighbor Search (GNNS) algorithm for $K$-NN Search Problems.

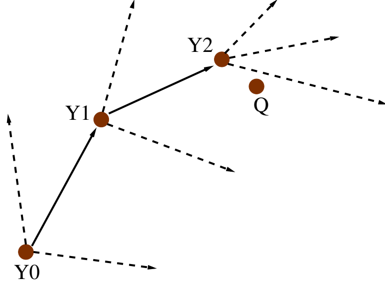



Figure 2: The GNNS Algorithm on a simple nearest neighbor graph.

query. Figure 2 illustrates the algorithm on a simple nearest neighbor graph with query $Q$, $K = 1$ and $E = 3$.

Parameters $R$, $T$, and $E$ specify the computational budget of the algorithm. By increasing each of them, the algorithm spends more time in search and returns a more accurate result. The difference between $E$ and $k$ and $K$ should be noted. $E$ and $K$ are two input parameters to the search algorithm (online), while $k$ is a parameter of the $k$-NN tree construction algorithm (offline). Given a query point $Q$, the search algorithm has to find the $K$nearest neighbors of $Q$. The algorithm, in each greedy step, examines only $E$ out of $k$ neighbors (of the current node) to choose the next node. Hence, it effectively works on an $E$-NN graph.

Next, we analyze the performance of the GNNS algorithm for the nearest neighbor search problem ($K = 1$).

**Theorem 1.** *Consider the version of the GNNS algorithm that uses $L_1$ norm as the metric and terminates when the greedy procedure gets stuck in a local minimum. Assume that the datapoints are drawn uniformly randomly from a d-dimensional hypercube of volume 1. Let $0 < \delta < 1$. Choose $M$ such that*

$$(M+1)^d(d\log(M+1)+\log 1/\delta) \geq n \geq M^d(d\log M+\log 1/\delta). \tag{1}$$

*Construct graph $\mathcal{G}$ by connecting each $X_i$ to the members of the set $V_i = \{X_j : \ \|X_i - X_j\|_1 \leq r\}$, where $r = 3/M$. Then with probability at least $1-\delta$, for any query point $Q$ and any starting point $Y_0$, the GNNS algorithm returns the true nearest neighbor to $Q$, and its computational cost is bounded by*

$$\min\{nd,\ 2^d dM^2(d\log(M+1) + \log 1/\delta)\}.$$

*Proof.* Discretize each hypercube edge into $M$ equal intervals. So the unit cube is partitioned into $M^d$ cubes of volume $(1/M)^d$. Denote the set of cubes by $\{A_1, \dots, A_{M^d}\}$. We compute the probability that there exists at least one point in each cube.

$$\begin{aligned}\mathbb{P}(\forall j,\, \exists i,\, X_i \in A_j) &= 1 - \mathbb{P}(\exists j,\, \forall i,\, X_i \notin A_j) \\ &= 1 - \mathbb{P}\left(\bigcup_{j=1}^{M^d} \forall i,\, X_i \notin A_j\right) \\ &\geq 1 - \sum_{j=1}^{M^d} \mathbb{P}(\forall i,\, X_i \notin A_j) \\ &= 1 - M^d\left(1 - \frac{1}{M^d}\right)^n.\end{aligned}$$

Let $M^d\left(1 - \frac{1}{M^d}\right)^n \leq \delta$. After reordering, we get

$$n \geq \frac{d\log M + \log 1/\delta}{\log\left(1 + \frac{1}{M^d - 1}\right)}.$$

By using the linear approximation of $\log(1 + x) \approx x$ for $x \approx 0$, we get

$$n \geq M^d(d\log M + \log 1/\delta).$$

In summary, we have shown that for any $0 < \delta < 1$, if Inequality (1) holds, then $\mathbb{P}(\forall j,\, \exists i,\, X_i \in A_j) \geq 1 - \delta$. Thus, with probability $1-\delta$ all cubes contain at least one data-point.

Now let $X_i$ be an arbitrary point in $\mathcal{D}$, and $Q$ be a query point that is not necessarily in $\mathcal{D}$. There are at least $2^d$ cubes in $V_i$. Under the condition that all cubes contain at least one data-point, there is at least one cube in $V_i$ that contains a point $X_k$ such that $\rho(X_k, Q) < \rho(X_i, Q)$, which is is easy to see because we use $L_1$ norm. Thus, the greedy approach makes progress. Further, recall that each axis is partitioned into $M$ intervals. Hence, the algorithm takes at most $Md$ steps. Because of (1), there are at most $M(d\log(M + 1) + \log 1/\delta)$ points in each cube. Thus, the computational complexity is $\min\{nd,\ 2^d dM^2(d\log(M + 1) + \log 1/\delta)\}$. □

*Remark* 2. Because $M = O(n^{1/d})$, the computational cost is $\min\{nd,\ 2^d n^{2/d}\}$. The theorem can be proven for other distributions. The crucial assumption is the assumption on the independence of the datapoints. The uniformity assumption is made to simplify the presentation.

# 4 Experimental Results

In this section, we compare the performance of our algorithm with state-of-the-art nearest neighbor search techniques (explained in Sections 2.2 and 2.1): randomized KD-trees with best-bin-first search heuristic and LSH[1]. The experiments are carried out on a real-world publicly available image dataset [Howard and Roy, 2003] as well as a synthetically generated dataset. We compare the methods in terms of both the speedup over the linear search and the number of Euclidean distance computations.

First, we explain the experiments with the real-world dataset. We extracted 5 datasets of 17000, 50000, 118000 and 204000 (128-dimensional) SIFT descriptors[2] [Lowe, 2004]. For each dataset, the query set containing 500 SIFT descriptors is sampled from different images than the ones used to create the dataset. The experiments are performed for $K = 1$ and 30 . The accuracy is measured by first computing the percentage of the $K$ nearest neighbors reported correctly, and then averaging over 500 queries. For each dataset, instead of building a new graph for each value of $E$, we constructed a single large graph ($k = 1000$) and reused it in all experiments.

We exhaustively tried LSH and KD-tree with different values of parameters and chose combinations that result in better speedup and precision (parameter sweep).

Figures 3 (a), (b), (c), and (d) show the results for $K = 1$ and datasets of different sizes. These figures are produced by varying the number of node expansions $E$; The other parameter $R$ is fixed and set to 1 and $T$ is not used as we alternatively terminated the search when it reached the node which is better than its neighbors. Figures 4 (a), (b), (c), and (d) compare the methods in terms of the number of distance computations that they perform (normalized by dividing over the number of distance computations that the linear search performs). As we can see from these figures, the GNNS method outperforms both the KD-tree and LSH algorithms. The figures also show how the performance improves with the size of dataset.

Figure 5 shows the results for 17k and 204k datasets and $K = 30$. In order to produce these figures, we performed parameter sweeps on $E$ and $T$ and chose combinations that result in better speedup and precision. The parameter $R$ is set to 1. The full experimental results with $K = 30$ as well as with more real-world datasets can be found in `https://webdocs.cs.ualberta.ca/~hajebi`.

The second set of experiments were performed on synthetically generated datasets of different dimensions to show how the performances of different methods degrade as dimensionality increases. To construct a dataset of dimension $d$, we sampled 50000 vectors from the uniform distribution over

[1]We used the publicly available implementations of KD-tree [`http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN`] and LSH [`http://ttic.uchicago.edu/~gregory/download.html`]

[2]We used SIFT descriptors due to their popularity in feature matching applications.

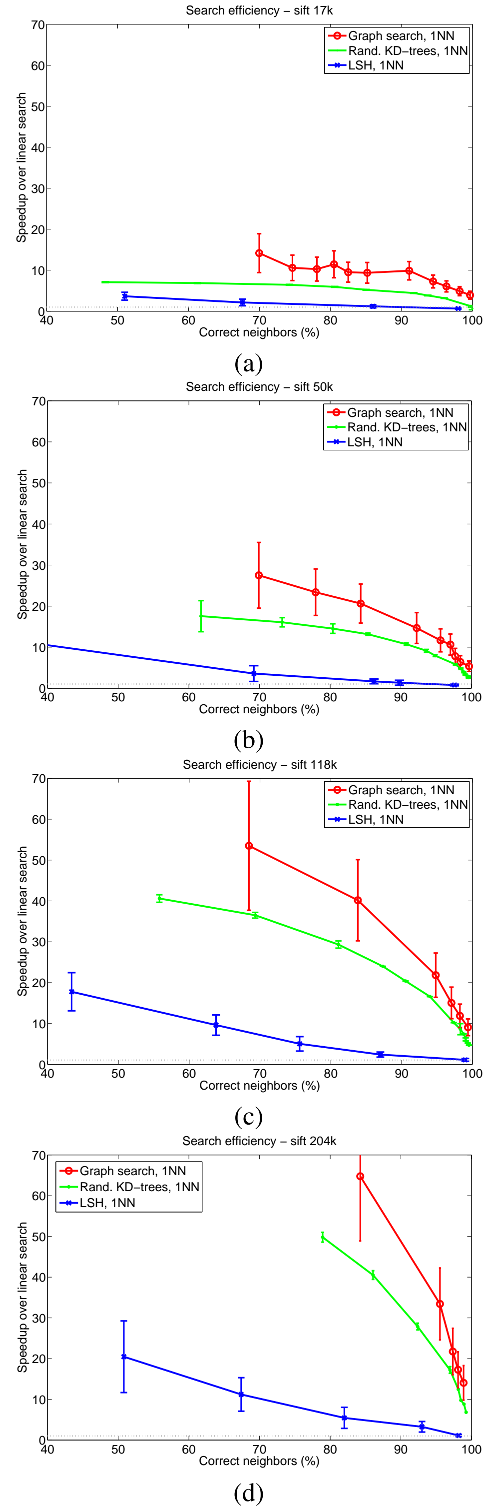
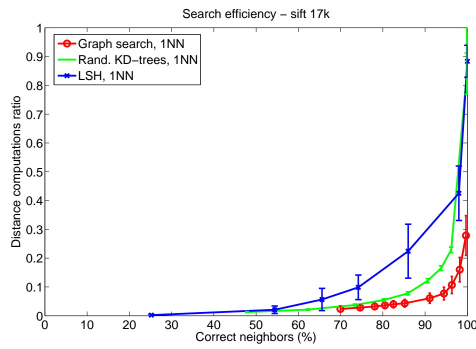
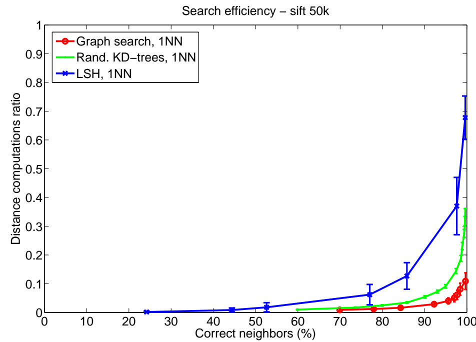


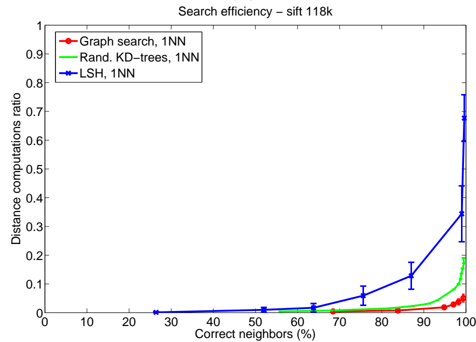

Figure 3: The results for 1-NN search problem ($K = 1$). Speedup vs. accuracy for different algorithms (GNNS, randomized KD-tree, and LSH) on datasets of (a) 17k (b) 50k (c) 118k (d) 204k points. The GNNS algorithm outperforms the two other methods. The gray dashed line indicates the speedup of 1. The error bars are standard deviations over 500 queries.



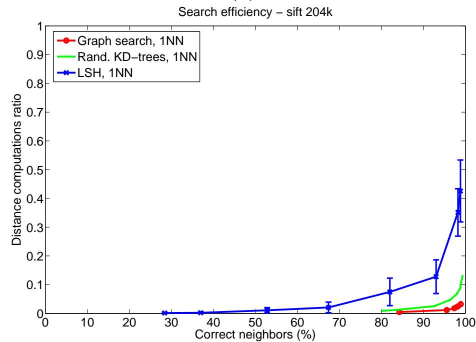

(a)




(b)

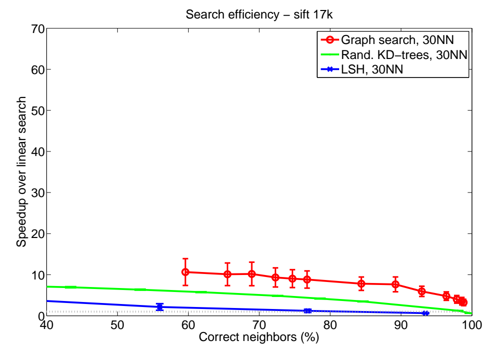



(c)

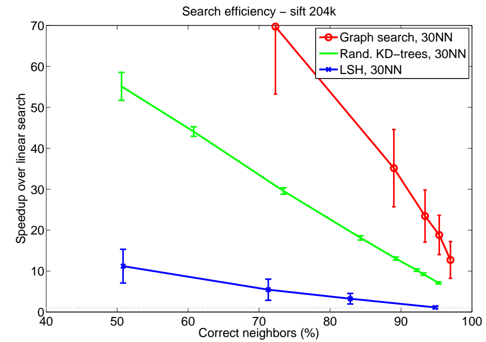



(d)

Figure 4: The results for 1-NN search problem ($K = 1$). The x-axis is accuracy and the y-axis is the number of distance computations that different algorithms perform (normalized by dividing over the number of distance computations that the linear search performs). The datasets have (a) 17k (b) 50k (c) 118k (d) 204k points. The GNNS algorithm outperforms the two other methods. The error bars are standard deviations over 500 queries.




(a)




(b)

Figure 5: The results for 30-NN search problem ($K = 30$). The results for (a) 17k (b) 204k datasets. Speedup vs. accuracy for different algorithms (GNNS, randomized KD-tree, and LSH). The GNNS algorithm outperforms the two other methods. The gray dashed line indicates the speedup of 1. The error bars are standard deviations over 500 queries.

$[0, 1]^d$. We also sampled 500 query vectors from the same distribution. Figures 6 and 7 show the results for the randomly generated datasets. Figure 6 compares the GNNS and the KD-tree methods. The GNNS method outperforms the KD-tree method. Figure 7 shows the results for the LSH method, which is much inferior to the two other methods. The figures also show how the speedup of different algorithms with respect to the linear search degrades as we increase the dimensionality and the precision.

## 5 Conclusions and Future Work

We have introduced a new algorithm that performs hill-climbing on a $k$-NN graph to solve the nearest neighbor search problem. The drawback of this method is the expensive offline construction of the $k$-NN graph. We experimentally show the effectiveness of the GNNS method on a high-dimensional real world problem as well as synthetically generated datasets.

In many cases, high-dimensional data lie on a low-dimensional manifold. Dasgupta and Freund [2008] show that a version of KD-tree exploits the low-dimensional structure of data to improve its NN search performance. We hypothesize that the GNNS algorithm has a similar property.

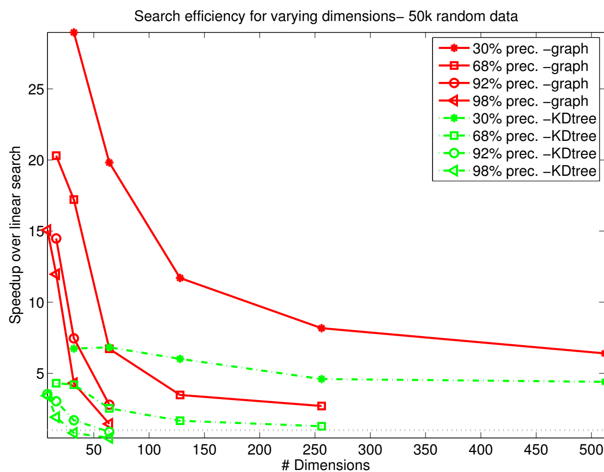



Figure 6: The results for 1-NN search problem ($K = 1$). Speedup vs. dimension for different precisions and algorithms (GNNS and randomized KD-tree). Datasets have 50k points. The GNNS algorithm outperforms the KD-tree method. The gray dashed line indicates the speedup of 1.

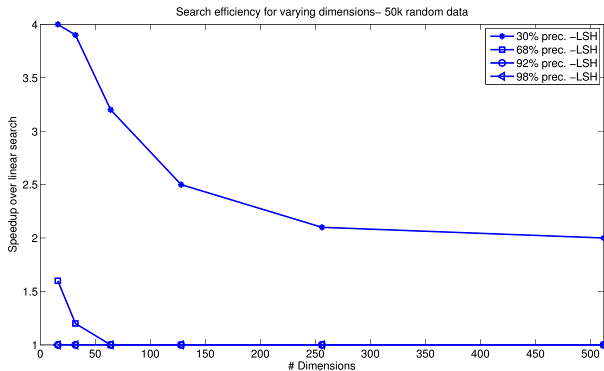



Figure 7: The results for 1-NN search problem ($K = 1$). Speedup vs. dimension for different precisions for LSH algorithm. Datasets have 50k points.

This remains as a future work. Another future work is to remove the exponential dependence on dimensionality in the average-case analysis, as is shown to be possible for a number of NN search methods [Goodman *et al.*, 2004].

## References


[Beis and Lowe, 1997] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pages 1000–1006, 1997.

[Bentley, 1980] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23:214–229, April 1980.

[Chen *et al.*, 2009] Jie Chen, Haw-ren Fang, and Yousef Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *J. Mach. Learn. Res.*, 10:1989–2012, December 2009.

[Connor and Kumar, 2010] M. Connor and P. Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *Visualization and Computer Graphics, IEEE Transactions on*, 16(4):599 –608, 2010.

[Dasgupta and Freund, 2008] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 537–546, 2008.

[Friedman *et al.*, 1977] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, September 1977.

[Goodman *et al.*, 2004] Jacob E. Goodman, Joseph O'Rourke, and Piotr Indyk. *Handbook of Discrete and Computational Geometry (2nd ed.)*. CRC Press, 2004.

[Howard and Roy, 2003] A. Howard and N. Roy. The robotics data set repository (radish), http://cres.usc.edu/radishrepository/view-all.php [ualberta-csc-flr3-vision], 2003.

[Indyk and Motwani, 1998] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[Lifshits and Zhang, 2009] Yury Lifshits and Shengyu Zhang. Combinatorial algorithms for nearest neighbors, near-duplicates and small-world design. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'09, pages 318–326, 2009.

[Lowe, 2004] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, pages 91–110, 2004.

[Muja and Lowe, 2009] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[Papadias, 2000] Dimitris Papadias. Hill climbing algorithms for content-based retrieval of similar configurations. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, pages 240–247, 2000.

[Paredes and Chvez, 2005] Rodrigo Paredes and Edgar Chvez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *In Proc. SPIRE'05, LNCS 3772*, pages 127–138, 2005.

[Vaidya, 1989] P. M. Vaidya. An o(n log n) algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, January 1989.