Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph

Cong Fu, Chao Xiang, Changxu Wang, Deng Cai*
The State Key Lab of CAD&CG, College of Computer Science, Zhejiang University, China
Alibaba-Zhejiang University Joint Institute of Frontier Technologies
Fabu Inc., Hangzhou, China
{fc731097343, changxu.mail, dengcai}@gmail.com; chaoxiang@zju.edu.cn

ABSTRACT

Approximate nearest neighbor search (ANNS) is a fundamental problem in databases and data mining. A scalable ANNS algorithm should be both memory-efficient and fast. Some early graph-based approaches have shown attractive theoretical guarantees on search time complexity, but they all suffer from the problem of high indexing time complexity. Recently, some graph-based methods have been proposed to reduce indexing complexity by approximating the traditional graphs; these methods have achieved revolutionary performance on million-scale datasets. Yet, they still can not scale to billion-node databases. In this paper, to further improve the search-efficiency and scalability of graph-based methods, we start by introducing four aspects: (1) ensuring the connectivity of the graph; (2) lowering the average out-degree of the graph for fast traversal; (3) shortening the search path; and (4) reducing the index size. Then, we propose a novel graph structure called Monotonic Relative Neighborhood Graph (MRNG) which guarantees very low search complexity (close to logarithmic time). To further lower the indexing complexity and make it practical for billion-node ANNS problems, we propose a novel graph structure named Navigating Spreading-out Graph (NSG) by approximating the MRNG. The NSG takes the four aspects into account simultaneously. Extensive experiments show that NSG outperforms all the existing algorithms significantly. In addition, NSG shows superior performance in the E-commercial scenario of Taobao (Alibaba Group) and has been integrated into their billion-scale search engine.

PVLDB Reference Format:
Cong Fu, Chao Xiang, Changxu Wang, Deng Cai. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB*, 12(5): 461-474, 2019.
DOI: https://doi.org/10.14778/3303753.3303754

1. INTRODUCTION

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303754

Approximate nearest neighbor search (ANNS) has been a hot topic over decades and provides fundamental support for many applications in data mining, databases, and information retrieval [2, 10, 12, 23, 37, 42]. For sparse discrete data (like documents), the nearest neighbor search can be carried out efficiently on advanced index structures (*e.g.*, inverted index [35]). For dense continuous vectors, various solutions have been proposed such as tree-structure based approaches [2, 6, 8, 17, 24, 36], hashing-based approaches [18, 20, 23, 32, 40], quantization-based approaches [1, 19, 26, 39], and graph-based approaches [3, 21, 33, 41]. Among them, graph-based methods have shown great potential recently. There are some experimental results showing that the graph-based methods perform much better than some popular algorithms from other types in the commonly used Euclidean Space [2, 7, 15, 27, 33, 34]. The reason may be that these methods cannot express the neighbor relationship as well as the graph-based methods and they tend to check much more points in neighbor-subspaces than the graph-based methods to reach the same accuracy [39]. Thus, their search time complexity involves large factors exponential in the dimension and leads to inferior performance [22].

Nearest neighbor search via graphs has been studied for decades [3,13,25]. Given a set of points S in the d-dimensional Euclidean space E^d, a graph G is defined as a set of edges connecting these points (nodes). The edge pq defines a neighbor-relationship between node p and q. Various constraints are proposed on the edges to make the graphs suitable for ANNS problem. These graphs are now referred to as the *Proximity Graphs* [25]. Some proximity graphs like Delaunay Graphs (or Delaunay Triangulation) [4] and Monotonic Search Networks (MSNET) [13] ensure that from any node p to another node q, there exists a path on which the intermediate nodes are closer and closer to q [13]. However, the computational complexity needed to find such a path is not given. Other works like Randomized Neighborhood Graphs [3] guarantee polylogarithmic search time complexity. Empirically, the average length of greedy-routing paths grows polylogarithmically with the data size on the Navigable Small-World Networks (NSWN) [9, 29]. However, the time complexity of building these graphs is very high (at least $O(n^2)$), which is impractical for massive problems.

Some recent graph-based methods try to address this problem by designing approximations for the graphs. For example, GNNS [21], IEH [27], and Efanna [15] are based on the kNN graph, which is an approximation of the Delaunay Graph. NSW [33] approximates the NSWN, FANNG [7] approximates the Relative Neighborhood Graphs (RNG)

Algorithm 1 Search-on-Graph(G, $\mathbf{p}$, $\mathbf{q}$, l)

```
Require: graph $G$, start node $\mathbf{p}$, query point $\mathbf{q}$, candidate pool size $l$
Ensure: $k$ nearest neighbors of $\mathbf{q}$
 1: $i$=0, candidate pool $S = \emptyset$
 2: $S$.add($\mathbf{p}$)
 3: while $i < l$ do
 4:     $i$ =the index of the first unchecked node in $S$
 5:     mark $\mathbf{p_i}$ as checked
 6:     for all neighbor $\mathbf{n}$ of $\mathbf{p_i}$ in $G$ do
 7:         $S$.add($\mathbf{n}$)
 8:     end for
 9:     sort $S$ in ascending order of the distance to $\mathbf{q}$
10:     If $S$.size() $> l$, $S$.resize($l$)
11: end while
12: return the first $k$ nodes in $S$
```

[38], and Hierarchical NSW (HNSW) [34] is proposed to take advantage of properties of the Delaunay Graph, the NSWN, and the RNG. Moreover, the hierarchical structure of HNSW enables multi-scale hopping on different layers.

These approximations are mainly based on intuition and generally lack rigorous theoretical support. In our experimental study, we find that they are still not powerful enough for billion-node applications, which are in great demand today. To further improve the search-efficiency and scalability of graph-based methods, we start with how ANNS is performed on a graph. Despite the diversity of graph indices, almost all graph-based methods [3, 7, 13, 21, 27, 33] share the same greedy best-first search algorithm (given in Algorithm 1), we refer to it as the *search-on-graph* algorithm below.

Algorithm 1 tries to reach the query point with the following greedy process. For a given query q, we are required to retrieve its nearest neighbors from the dataset. Given a starting node p, we follow the out-edges to reach p's neighbors, and compare them with q to choose one to proceed. The choosing principle is to minimize the distance to q, and the new iteration starts from the chosen node. We can see that the key to improve graph-based search is to shorten the search path formed by the algorithm and reduce the out-degree of the graph (*i.e.*, reduce the number of choices of each node). Intuitively, to improve graph-based search we need to: (1) Ensure the connectivity of the graph to make sure the query (or the nearest neighbors of the query) is (are) reachable; (2) Lower the average out-degree of the graph and (3) shorten the search path to lower the search time complexity; (4) Reduce the index size (memory usage) to improve scalability. Methods such as IEH [27], Efanna [15], and HNSW [34], use hashing, randomized KD-trees and multi-layer graphs to accelerate the search. However, these may result in huge memory usage for massive databases. We aim to reduce the index size and preserve the search-efficiency at the same time.

In this paper, we propose a new graph, named as Monotonic Relative Neighborhood Graph (MRNG), which guarantees a low average search time (very close to logarithmic complexity). To further reduce the indexing complexity, we propose the Navigating Spreading-out Graph (NSG), which is a good approximation of MRNG, inherits low search complexity and takes the four aspects into account. It is worthwhile to highlight our contributions as follows.

1. We first present comprehensive theoretical analysis on the attractive ANNS properties of a graph family called MSNET. Based on this, we propose a novel graph, MRNG, which ensures a close-logarithmic search complexity in expectation.

2. To further improve the efficiency and scalability of graph-based ANNS methods, we consider four aspects of the graph: ensuring connectivity, lowering the average out-degree, shortening the search path, and reducing the index size. Motivated by these, we design a close approximation of the MRNG, called Navigating Spreading-out Graph (NSG), to address the four aspects simultaneously. The indexing complexity is reduced significantly compared to the MRNG and is practical for massive problems. Extensive experiments show that our approach outperforms the state-of-the-art methods in search performance with the smallest memory usage among graph-based methods.

3. The NSG algorithm is also tested on the E-commercial search scenario of Taobao (Alibaba Group). The algorithm has been integrated into their search engine for billion-node search.

2. PRELIMINARIES

We use E^d to denote the Euclidean space under the l_2 norm. The closeness of any two points p, q is defined as the l_2 distance, $\delta(p, q)$, between them.

2.1 Problem Setting

Various applications in information retrieval and database management of high-dimensional data can be abstracted as the nearest neighbor search problem in high-dimensional space. The Nearest Neighbor Search (NNS) problem is defined as follows [20]:

DEFINITION 1 (**Nearest Neighbor Search**). *Given a finite point set S of n points in space E^d, preprocess S to efficiently return a point $p \in S$ which is closest to a given query point q.*

This naturally generalizes to the K **Nearest Neighbor Search** when we require the algorithm to return K points ($K > 1$) which are the closest to the query point. The approximate version of the nearest neighbor search problem (ANNS) can be defined as follows [20]:

DEFINITION 2 (**ϵ-Nearest Neighbor Search**). *Given a finite point set S of n points in space E^d, preprocess S to efficiently answer queries that return a point p in S such that $\delta(p, q) \leq (1 + \epsilon)\delta(r, q)$, where r is the nearest neighbor of q in S.*

Similarly, this problem can generalize to the **Approximate** K **Nearest Neighbor Search (AKNNS)** when we require the algorithm to return K points ($K > 1$) such that $\forall i = 1, ..., K, \delta(p_i, q) \leq (1 + \epsilon)\delta(r, q)$. Due to the intrinsic difficulty of exact nearest neighbor search, most researchers turn to AKNNS. The main motivation is to trade a little loss in accuracy for much shorter search time.

For the convenience of modeling and evaluation, we usually do not calculate the exact value of ϵ. Instead we use another indicator to show the degree of the approximation: *precision*. Suppose the point set returned by an AKNNS

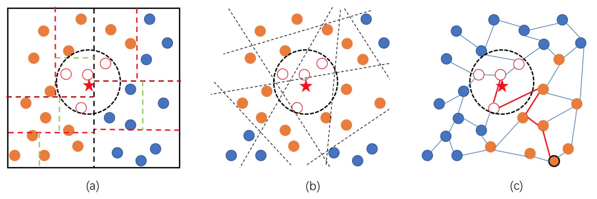

Figure 1: (a) is the tree index, (b) is the hashing index, and (c) is the graph index. The red star is the query (not included in the base data). The four red rings are its nearest neighbors. The tree and hashing index partition the space into several cells. Let each cell contain no more than three points. The out-degree of each node in the graph index is also no more than three. To retrieve the nearest neighbors of the query, we need to backtrack and check many leaf nodes for the tree index. We need to check nearby buckets with hamming radius 2 for the hashing index. As for the graph index, Algorithm 1 forms a search path as the red lines show. The orange circles are checked points during their search. The graph-based algorithm needs the least times of distance calculation.

algorithm of a given query q is R' and the correct k nearest neighbor set of q is R, then the *precision* (accuracy) is defined as below [15].

$$precision(R') = \frac{|R' \cap R|}{|R'|} = \frac{|R' \cap R|}{K}. \quad (1)$$

A higher *precision* corresponds to a smaller ϵ, thus, a higher degree of approximation. In this paper, we use the *precision* as the evaluation metric.

2.2 Non-Graph-Based ANNS Methods

Non-graph-based ANNS methods include tree-based methods, hashing-based methods, and quantization-based methods roughly. Some well-known and widely-used algorithms like the KD-tree [36], R^* tree [6], VA-file [39], Locality Sensitive Hashing (LSH) [20], and Product Quantization (PQ) [26] all belong to the above categories. Some works focus on improving the algorithms (*e.g.*, [2, 18, 19, 23, 32]), while others focus on optimizing the existing methods according to different platforms and scenarios (*e.g.*, [10, 12, 37, 42]).

In the experimental study of some recent works [2, 7, 15, 27, 33, 34], graph-based methods have outperformed some well-known non-graph-based methods (*e.g.*, KD-tree, LSH, PQ) significantly. This may be because the non-graph-based methods all try to solve the ANNS problem by partitioning the space and indexing the resulting subspaces for fast retrieval. Unfortunately, it is not easy to index the subspaces so that neighbor areas can be scanned efficiently to locate the nearest neighbors of a given query. See Figure 1 as an example (the figure does not include the PQ algorithm because it can be regarded as a hashing method from some perspective). The non-graph-based methods need to check many nearby cells to achieve high accuracy. A large number of distant points are checked and this problem becomes much worse as the dimension increases (known as the curse of the dimensionality). Graph-based methods may start from a distant position to the query, but they approach the query quickly because they are all based on proximity graphs which typically express the "neighbor" relationship better.

In summary, non-graph-based methods tend to check much more points than the graph-based methods to achieve the same accuracy. This will be shown in our later experiments.

2.3 Graph-Based ANNS Methods

Given a finite point set S in E^d, a graph is a structure composed of a set of nodes (representing the points) and edges which link some pairs of the nodes. A node p is called a neighbor of q if and only if there is an edge between p and q. Graph-based ANNS solves the ANNS problem defined above via a graph index. Algorithm 1 is commonly used in most graph-based methods. In past decades, many graphs are designed for efficient ANNS. Here we will introduce several graph structures with appealing theoretical properties.

Delaunay Graphs (or Delaunay Triangulations) are defined as the dual graph of the Voronoi diagram [4]. It is shown to be a monotonic search network [30], but the time complexity of high-dimensional ANNS on a Delaunay Graph is high. According to Harwood *et al.* [7], Delaunay Graphs quickly become almost fully connected at high dimensionality. Thus the efficiency of the search reduces dramatically. GNNS [21] is based on the (approximate) kNN graph, which is an approximation of Delaunay Graphs. IEH [27] and Efanna [15] are also based on the (approximate) kNN graph. They use hashing and Randomized KD-trees to provide better starting positions for Algorithm 1 on the kNN graph. Although they improve the performance, they suffer from large and complex indices.

Wen *et al.* [31] propose a graph structure called *DPG*, which is built upon an approximate kNN graph. They propose an edge selection strategy to cut off half of the edges from the prebuilt kNN graph and maximize the average angle among the remaining edges. Finally, they will make compensation on the graph to produce an undirected one. Their intuition is to make the angles among edges to distribute evenly around each node, but it lacks theoretical support. According to our empirical study, the DPG suffers from a large index and inferior search performance.

Relative Neighborhood Graphs (RNG) [38] are not designed for the ANNS problem in the first place. However, RNG has shown great potential in ANNS. The RNG adopts an interesting edge selection strategy to eliminate the longest edge in all the possible triangles on S. With this strategy, the RNG reduces its average out-degree to a constant $C_d + o(1)$, which is only related to the dimension d and usually very small [25]. However, according to Dearholt *et al.*'s study [13], the RNG does not have sufficient edges to be a monotonic search network due to the strict edge selection strategy. Therefore there is no theoretical guarantee on the length of the path. Dearholt *et al.* proposed a method to add edges to the RNG and turn it into a Monotonic Search Network (MSNET) with the minimal amount of edges, named as the minimal MSNET [13]. The algorithm is based on a prebuilt RNG, the indexing complexity of which is $O(n^{2-\frac{2}{1+d}+\epsilon})$, under the general position assumption [25]. The preprocessing of building the minimal MSNET is of $O(n^2 \log n + n^3)$ complexity. The total indexing complexity of the minimal MSNET is huge for high-dimensional and massive databases. Recent practical graph-based methods like FANNG [7] and HNSW [34] adopt the RNG's edge selection strategy to reduce the out-degree of their graphs and improve the search performance. However, they did not provide a theoretical analysis.

Navigable Small-World Networks [9, 29] are suitable for the ANNS problem by their nature. The degree of the nodes and the neighbors of each node are all assigned according to a specific probability distribution. The length of the search path on this graph grows polylogarithmically with the network size, $O(A[logN]^{\nu})$, where A and ν are some constants. This is an empirical estimation, which hasn't been proved. Thus the total empirical search complexity is $O(AD[logN]^{\nu})$, D is the average degree of the graph. The degree of the graph needs to be carefully chosen, which has a great influence on the search efficiency. Like the other traditional graphs, the time complexity of building such a graph is about $O(n^2)$ in a naive way, which is impractical for massive problems. Yury *et al.* [33] proposed NSW graphs to approximate the Navigable Small-World Networks and the Delaunay Graphs simultaneously. But soon they found that the degree of the graph was too high and there also existed connectivity problems in their method. They later proposed HNSW [34] to address this problem. Specifically, they stacked multiple NSWs into a hierarchical structure to solve the connectivity problem. The nodes in the upper layers are sampled through a probability distribution, and the size of the NSWs shrinks from bottom to top layer by layer. Their intuition is that the upper layers enable long-range short-cuts for fast locating of the destination neighborhood. Then they use the RNG's edge selection strategy to reduce the degree of their graphs. HNSW is the most efficient ANNS algorithm so far, according to some open source benchmarks on GitHub[1].

Randomized Neighborhood Graphs [3] are designed for ANNS problem in high-dimensional space. It is constructed in a randomized way. They first partition the space around each node with a set of convex cones, then they select $O(\log n)$ closest nodes in each cone as its neighbors. They prove that the search time complexity on this graph is $O((\log n)^3)$, which is very attractive. However, its indexing complexity is too high. To reduce the indexing complexity, they propose a variant, called RNG*. The RNG* also adopts the edge selection strategy of RNG and uses additional structures (KD-trees) to improve the search performance. However, the time complexity of its indexing is still as high as $O(n^2)$ [3].

3. ALGORITHMS AND ANALYSIS

3.1 Motivation

The heuristic search algorithm, Algorithm 1, has been widely used on various graph indices in previous decades. The algorithm walks over the graph and tries to reach the query greedily. Thus, two most crucial factors influencing the search efficiency are the number of greedy hops between the starting node and the destination and the computational cost to choose the next node at each step. In other words, the search time complexity on a graph can be written as $O(ol)$, where o is the average out-degree of the graph and l is the length of the search path.

In recent graph-based algorithms [7, 15, 21, 27, 31, 33, 34], the out-degree of the graph is a tunable parameter. In our experimental study, given a dataset and an expected search accuracy, we find there exist optimal degrees that result in optimal search performance. A possible explanation is that, given an expected accuracy, ol is a convex function of o, and the minima of ol determines the search performance of a given graph. In the high accuracy range, the optimal degrees of some algorithms (*e.g.*, GNNS [21], NSW [33], DPG [31]) are very large, which leads to very large graph size. The minima of their ol are also very large, leading to inferior performance. Other algorithms [15, 27, 34] use extra index structures to improve their start position in Algorithm 1 in order to minimize l directly, but this leads to large indices.

From our perspective, we can improve the ANNS performance of graph-based methods by minimizing o and l simultaneously. Moreover, we need to make the index as small as possible to handle large-scale data. What is always ignored is that one should first ensure the existence of a path from the starting node to the query. Otherwise, the targets will never be reached. In summary, we aim to design a graph index with high ANNS performance from the following four aspects. **(1) ensuring the connectivity of the graph, (2) lowering the average out-degree of the graph, (3) shortening the search path, and (4) reducing index size.** Point (1) is easy to achieve. If the starting node varies with the query, one should ensure that the graph is strongly connected. If the starting node is fixed, one should ensure all other nodes are reachable by a DFS from the starting node. As for point (2)-(4), we address these points simultaneously by designing a better sparse graph for ANNS problem. Below we will propose a new graph structure called Monotonic Relative Neighborhood Graph (MRNG) and a theoretical analysis of its important properties, which leads to better ANNS performance.

3.2 Graph Monotonicity And Path Length

The speed of ANNS on graphs is mainly determined by two factors, the length of the search path and the average out-degree of the graph. Our goal is to find a graph with both low out-degrees and short search paths. We will begin our discussion with how to design a graph with very short search paths. Before we introduce our proposal, we will first provide a detailed analysis of a category of graphs called *Monotonic Search Networks (MSNET)*, which are first discussed in [13] and have shown great potential in ANNS. Here we will present the definition of the MSNETs.

3.2.1 Definition And Notation

Given a point set S in E^d space, p, q are any two points in S. Let $B(p,r)$ denote an **open sphere** such that $B(p,r) = \{x|\delta(x,p) < r\}$. Let $\overrightarrow{pq}$ denote a directed edge from p to q.

First we define a monotonic path in a graph as follows:

Definition 3 (**Monotonic Path**). *Given a finite point set S of n points in space E^d, p, q are any two points in S and G denotes a graph defined on S. Let $v_1, v_2, ..., v_k, (v_1 = p, v_k = q)$ denote a path from p to q in G,* i.e., *$\forall i = 1, ..., k-1$, edge $\overrightarrow{v_i v_{i+1}} \in G$. This path is a monotonic path if and only if $\forall i = 1, ..., k-1, \delta(v_i, q) > \delta(v_{i+1}, q)$.*

Then the monotonic search network is defined as follows:

Definition 4 (**Monotonic Search Network**). *Given a finite point set S of n points in space E^d, a graph defined*

[1]https://github.com/erikbern/ann-benchmarks

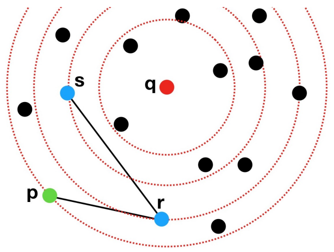

Figure 2: An illustration of the search in an MSNET. The query point is q and the search starts with point p. At each step, Algorithm 1 will select a node that is the closest to q among the neighbors of the current nodes. Suppose p, r, s is on a monotonic path selected by Algorithm 1. The search region shrinks from sphere $B(q, \delta(p, q))$ to $B(q, \delta(r, q))$, then to $B(q, \delta(s, q))$. The number of nodes in each sphere (may be checked) decreases by some ratio at each step until only q is left in the final sphere.

on S is a monotonic search network if and only if there exists at least one monotonic path from p to q for any two nodes $p, q \in S$.

3.2.2 Analysis On Monotonic Search Networks

The Monotonic Search Networks (MSNET) [13] are a category of graphs which can guarantee a monotonic path between any two nodes in the graph. MSNETs are strongly connected graphs by nature, which ensures the connectivity. When traveling on a monotonic path, we always make progress to the destination at each step. In an MSNET, Dearholt *et al.* hypothesized that one may be able to use Algorithm 1 (commonly used in graph-based search) to detect the monotonic path to the destination node, *i.e.*, no backtracking is needed [13], which is a very attractive property. Backtracking means, when the algorithm cannot find a closer neighbor to the query (*i.e.*, a local optimal), we need to go back to the visited nodes and find an alternative direction to move on. The monotonicity of the MSNETs makes the search behavior of Algorithm 1 on the graph almost definite and analyzable. However, Dearholt *et al.* [13] failed to provide a proof of this property. In this section, we will give a concrete proof of this property.

Theorem 1. *Given a finite point set S of n points, randomly distributed in space E^d, and an MSNET G constructed on S, a monotonic path between any two nodes p, q in G can be found by Algorithm 1 without backtracking.*

Proof. Due to the space limitation, please see the detailed proof in our technical report [16]. □

From Theorem 1, we know that we can reach the query $q \in S$ on a given MSNET with Algorithm 1 without backtracking, Therefore, the iteration expectation is the same as the length expectation of a monotonic path in the MSNET. Before we discuss the length expectation of a monotonic path in a given MSNET, we first define the MSNETs from a different perspective, which will help with the analysis.

Lemma 1. *Given a graph G on a set S of n points in E^d, G is an MSNET if and only if for any two nodes p, q, there is at least one edge $\overrightarrow{pr}$ such that $r \in B(q, \delta(p, q))$.*

Proof. Due to the space limitation, please see the detailed proof in our technical report [16]. □

From Lemma 1 we can calculate the length expectation of the monotonic path in the MSNETs as follows.

Theorem 2. *Let S be a finite point set of n points uniformly distributed in a finite subspace in E^d. Suppose the volume of the minimal convex hull containing S is V_S. The maximal distance between any two points in S is R. We impose a constraint on V_S such that when d is fixed, $\exists\kappa, \kappa V_S \geq V_B(R)$, where κ is a constant independent of n, and $V_B(R)$ is the volume of the sphere with radius R. We define $\triangle r$ as $\triangle r = min\{|\delta(a, b) - \delta(a, c)|, |\delta(a, b) - \delta(b, c)|, |\delta(a, c) - \delta(b, c)|\}$, for all possible non-isosceles triangles abc on S. $\triangle r$ is a decreasing function of n.*

For a given MSNET defined on such S, the length expectation of a monotonic path from p to q, for any $p, q \in S$, is $O(n^{1/d} log(n^{1/d})/\triangle r)$.

Proof. Due to the space limitation, please see the detailed proof in our technical report [16]. □

Theorem 2 is a general property for all kinds of MSNETs. The function $\triangle r$ has no definite expression about n because it involves randomness. We have observed that, in practice, $\triangle r$ decreases very slowly as n increases. In experiments, we estimate the function of $\triangle r$ on different public datasets, based on the proposed graph in this paper. We find that $\triangle r$ is mainly influenced by the data distribution and data density. Results are shown in the experiment section.

Because $O(n^{\frac{1}{d}})$ increases very slowly when n increases in high dimensional space, the length expectation of the monotonic paths in an MSNET, $O(n^{1/d} log(n^{1/d})/\triangle r)$, will have a growth rate very close to $O(\log n)$. This is also verified in our experiments. Likewise, we can see that the growth rate of the length expectation of the monotonic paths is lower when d is higher.

In Theorem 2, the assumption on the volume of the minimal convex hull containing the data points is actually a constraint on the data distribution. We try to avoid the special shape of the data distribution (*e.g.*, all points form a straight line), which may influence the conclusion. For example, if the data points are all distributed uniformly on a straight line, the length expectation of the monotonic paths on such a dataset will grow almost linearly with n.

In addition, though we assume a uniform distribution of the data points, the property still holds to some extent on other various distributions in practice. Except for some extremely special shape of the data distribution, we can usually expect that, as the search sphere shrinks at each step of the search path, the amount of the nodes remaining in the sphere decreases by some ratio. The ratio is mainly determined by the data distribution, as shown in Figure 2.

In addition to the length expectation of the search path, another important factor that influences the search complexity is the average out-degree of the graph. The degree of some MSNETs, like the Delaunay Graphs, grows when n increases [4]. There is no unified geometrical description of the MSNETs, therefore there is no unified conclusion about how the out-degree of the MSNETs scales.

Dearholt *et al.* [13] claim that they have found a way to construct an MSNET with a minimal out-degree. However, there are two problems with their method. Firstly, they did not provide analysis on how the degree of their MSNET

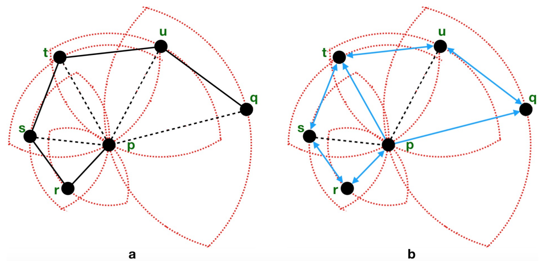

Figure 3: A comparison between the edge selection strategy of the RNG (a) and the MRNG (b). An RNG is an undirected graph, while an MRNG is a directed one. In (a), p and r are linked because there is no node in $lune_{pr}$. Because $r \in lune_{ps}$, $s \in lune_{pt}$, $t \in lune_{pu}$, and $u \in lune_{pq}$, there are no edges between p and s, t, u, q. In (b), p and r are linked because there is no node in $lune_{pr}$. p and s are not linked because $r \in lune_{ps}$ and $pr, sr \in$ *MRNG*. Directed edge $\overrightarrow{pt} \in$ *MRNG* because $\overrightarrow{ps} \notin$ *MRNG*. However, $\overrightarrow{tp} \notin$ *MRNG* because $\overrightarrow{ts} \in$ *MRNG*. We can see that the MRNG is defined in a recursive way, and the edge selection strategy of the RNG is more strict than MRNG's. In the RNG(a), there is a monotonic path from q to p, but no monotonic path from p to q. In the MRNG(b), there is at least one monotonic path from any node to another node.

scales with n. This is mainly because the MSNET they proposed is built by adding edges to an RNG and lacks a geometrical description. Secondly, the proposed MSNET construction method has a very high time complexity (at least $O(n^{2-\frac{2}{1+d}+\epsilon} + n^2 \log n + n^3)$) and is not practical in real large-scale scenarios. Below we will propose a new type of MSNET with lower indexing complexity and constant out-degree expectation (independent of n). Simply put, the search complexity on this graph scales with n in the same speed as the length expectation of the monotonic paths.

3.3 Monotonic Relative Neighborhood Graph

In this section, we describe a new graph structure for ANNS called as MRNG, which belongs to the MSNET family. To make the graph sparse, HNSW and FANNG turn to RNG [38], but it was proved that the RNG does not have sufficient edges to be an MSNET [13]. Therefore there is no theoretical guarantee of the search path length in an RNG, and the search on an RNG may suffer from long detours.

Consider the following example. Let $lune_{pq}$ denote a region such that $lune_{pq} = B(p, \delta(p,q)) \cap B(q, \delta(p,q))$ [25]. Given a finite point set S of n points in space E^d, for any two nodes $p, q \in S$, edge $pq \in$ RNG if and only if $lune_{pq} \cap S = \emptyset$. In Figure 3, (a) is an illustration of a non-monotonic path in an RNG. Node s is in $lune_{pr}$, so p, s are not connected. Similarly, t, u, q are not connected to p. When the search goes from p to q, the path is non-monotonic (*e.g.*, $rq < pq$).

We find that this problem is mainly due to RNG's edge selection strategy. Dearholt *et al.* tried to add edges to the RNG [13] to produce an MSNET with the fewest edges, but this method is very time-consuming. Instead, inspired by the RNG, we propose a new edge selection strategy to con-

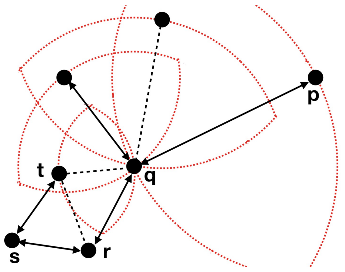

Figure 4: An illustration of the necessity that *NNG* $\subset$ *MRNG*. If not, the graph cannot be an MSNET. Path p, q, r, s, t is an example of non-monotonic path from p to t. In this graph, t is the nearest neighbor of q but not linked to q. We apply the MRNG's edge selection strategy on this graph. According to the definition of the strategy, t and r can never be linked. When the search goes from p to t, it must detour with at least one more step through s. This problem will be worse in practice.

struct monotonic graphs. The resulting graph may not be the minimal MSNET but it is very sparse. Based on the new strategy, we propose a novel graph structure called Monotonic Relative Neighborhood Graph (MRNG). Formally, an MRNG can be defined as follows:

Definition 5 (MRNG). *Given a finite point set S of n points in space E^d, an MRNG is a directed graph with the set of edges satisfying the following property: for any edge $\overrightarrow{pq}$, $\overrightarrow{pq}$ ∈MRNG if and only if $lune_{pq} \cap S = \emptyset$ or $\forall r \in (lune_{pq} \cap S)$, $\overrightarrow{pr} \notin$ MRNG.*

We avoid ambiguity in the following way when isosceles triangles appear. If $\delta(p,q) = \delta(p,r)$ and qr is the shortest edge in triangle pqr, we select the edge according to a predefined index, *i.e.*, we select $\overrightarrow{pq}$ if *index(q) < index(r)*. We can see that the MRNG is defined in a recursive way. In other words, Definition 5 implies that for any node p, we should select its neighbors from the closest to the farthest. The difference between MRNG's edge selection strategy and RNG's is that, for any edge $pq \in MRNG$, $lune_{pq} \cap S$ is not necessarily $\emptyset$. The difference can be seen in Figure 3 clearly. Here we show that the MRNG is an MSNET.

Theorem 3. *Given a finite point set S of n points. An MRNG defined on S is an MSNET.*

Proof. Due to the space limitation, please see the detailed proof in our technical report [16]. □

Though different in the structure, MRNG and RNG share some common edges. We will start with defining the Nearest Neighbor Graph (NNG) as follows:

Definition 6 (NNG). *Given a finite point set S of n points in space E^d, an NNG is the set of edges such that, for any edge $\overrightarrow{pq}$, $\overrightarrow{pq} \in$ NNG if and only if q is the closest neighbor of p in S.*

Similarly, we can remove the ambiguity in the NNG by assigning a unique index for each node and linking the node to its nearest neighbor with the smallest index. Obviously,

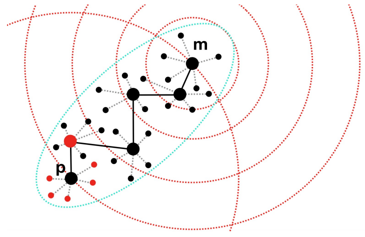

Figure 5: An illustration of the candidates of edge selection in NSG. Node p is the node to be processed, and m is the Navigating Node. The red nodes are the k nearest neighbors of node p. The big black nodes and the solid lines form a possible monotonic path from m to p, generated by the search-and-collect routine. The small black nodes are the nodes visited by the search-and-collect routine. All the nodes in the figure will be added to the candidate set of p.

we have $MRNG \cap RNG \supset NNG$ (if a node q is the nearest neighbor of p, we have $lune_{pq} \cap S = \emptyset$). This is necessary for MRNG's monotonicity. Figure 4 shows an example of the non-monotonic path if we apply MRNG's edge selection strategy on some graph G but do not guarantee $NNG \subset G$. The edges in Figure 4 satisfy the selection strategy of the MRNG except that q is forced not to be linked to its nearest neighbor t. Because t is the nearest neighbor of q, we have $\delta(q,r) > \delta(q,t)$. Because qt is the shortest edge in triangle qtr and q,r is linked, then rt must be the longest edge in triangle qtr according to the edge selection strategy of MRNG. Thus, r,t can not be linked and we can only reach t through other nodes (like s). Similarly, only when rt is the longest edge in triangle rst, edge rs and st can coexist in this graph. Therefore when we go from p to t, we need a detour at least via nodes r,s. Because $\delta(q,r) > \delta(q,t)$, it is a non-monotonic path from p to t. If we don't guarantee $NNG \subset MRNG$, detours are unavoidable, which may be worse in practice. It's easy to verify that a similar detour problem will also appear if we perform RNG's edge selection strategy on G but do not guarantee that $NNG \subset G$.

Here we will discuss the average out-degree of the MRNG. The MRNG has more edges than the RNG, but it's still very sparse because the angle between any two edges sharing the same node is at least 60° (by the definition of MRNG, for any two edges $pq, pr \in MRNG$, qr must be the longest edge in triangle pqr and $qr \notin MRNG$).

LEMMA 2. *Given an MRNG in E^d, the max degree of the MRNG is a constant and independent of n.*

PROOF. Due to the space limitation, please see the detailed proof in our technical report [16]. □

Now according to Lemma 2, Theorem 1, Theorem 2, and Theorem 3, we have that the search complexity in expectation on an MRNG is $O(cn^{\frac{1}{d}} \log n^{\frac{1}{d}} / \triangle r)$, where c is the average degree of the MRNG and independent of n, $\triangle r$ is a function of n, which decreases very slowly as n increases.

3.4 MRNG Construction

The MRNG can be constructed simply by applying our edge selection strategy on each node. Specifically, for each node p, we denote the set of rest nodes in S as $R = S \backslash \{p\}$. We calculate the distance between each node in R and p, then rank them in ascending order according to the distance. We denote the selected node set as L. We add the closest node in R to L to ensure $NNG \subset MRNG$. Next, we fetch a node q from R and a node r from L in order to check whether pq is the longest edge in triangle pqr. If pq is not the longest edge in triangle $pqr, \forall r \in L$, we add q to L. We repeat this process until all the nodes in R are checked. This naive construction runs in $O(n^2 \log n + n^2 c)$ time, where c is the average out-degree of MRNG, which is much smaller than that of the MSNET indexing method proposed in [13], which is at least $O(n^{2-\frac{2}{1+d}+\epsilon} + n^2 \log n + n^3)$ under the general position assumption.

Algorithm 2 NSGbuild(G, l, m)

```
Require: kNN Graph $G$, candidate pool size $l$ for greedy search, max-out-degree $m$.
Ensure: NSG with navigating node $\mathbf{n}$
1: calculate the centroid $\mathbf{c}$ of the dataset.
2: $\mathbf{r}$ =random node.
3: $\mathbf{n}$ =Search-on-Graph($G$,$\mathbf{r}$,$\mathbf{c}$,$l$)%navigating node
4: for all node $\mathbf{v}$ in G do
5:     Search-on-Graph($G$,$\mathbf{n}$,$\mathbf{v}$,$l$)
6:     $E$ =all the nodes checked along the search
7:     add $\mathbf{v}$'s nearest neighbors in $G$ to $E$
8:     sort $E$ in the ascending order of the distance to $\mathbf{v}$.
9:     result set $R = \emptyset$, $\mathbf{p}_0$ = the closest node to $\mathbf{v}$ in $E$
10:    $R$.add($\mathbf{p}_0$)
11:    while !$E$.empty() && $R$.size() $< m$ do
12:        $\mathbf{p} = E$.front()
13:        $E$.remove($E$.front())
14:        for all node $\mathbf{r}$ in $R$ do
15:            if edge $\mathbf{pv}$ conflicts with edge $\mathbf{pr}$ then
16:                break
17:            end if
18:        end for
19:        if no conflicts occurs then
20:            $R$.add($\mathbf{p}$)
21:        end if
22:    end while
23: end for
24: while True do
25:    build a tree with edges in NSG from root $\mathbf{n}$ with DFS.
26:    if not all nodes linked to the tree then
27:        add an edge between one of the out-of-tree nodes and
28:        its closest in-tree neighbor (by algorithm 1).
29:    else
30:        break.
31:    end if
32: end while
```

3.5 NSG:Practical Approximation For MRNG

Though MRNG can guarantee a fast search time, its high indexing time is still not practical for large-scale problems. In this section, we will present a practical approach by approximating our MRNG and starting from the four criteria to design a good graph for ANNS. We name it the **Navigating Spreading-out Graph** (NSG). We first present the NSG construction algorithm (Algorithm 2) as follows:

i We build an approximate kNN graph with the current state-of-the-art methods(*e.g.*, [14,28]).

ii We find the approximate medoid of the dataset. This can be achieved by the following steps. (1) Calculate the centroid of the dataset; (2) Treat the centroid as the query, search on the kNN graph with Algorithm 1, and take the returned nearest neighbor as the approximate medoid. This node is named as the Navigating Node because all the search will start with this fixed node.

iii For each node, we generate a candidate neighbor set and select neighbors for it from the candidate sets. This can be achieved by the following steps. For a given node p, (1) we treat it as a query and perform Algorithm 1 starting from the Navigating Node on the prebuilt kNN graph. (2) During the search, each visited node q (*i.e.*, the distance between p and q is calculated) will be added to the candidate set (the distance is also recorded). (3) Select at most m neighbors for p from the candidate set with the edge selection strategy of MRNG.

iv We span a Depth-First-Search tree on the graph produced in previous steps. We treat the Navigating Node as the root. When the DFS terminates, and there are nodes which are not linked to the tree, we link them to their approximate nearest neighbors (from Algorithm 1) and continue the DFS.

What follows is the motivation of the NSG construction algorithm. The ultimate goal is to build an approximation of MRNG with low indexing time complexity.

(i) MRNG ensures there exists at least one monotonic path between any two nodes, however, it is not an easy task. Instead, we just pick one node out and try to guarantee the existence of monotonic paths from this node to all the others. We name this node as the Navigating Node. When we perform the search, we always start from the Navigating Node, which makes the search on an NSG almost as efficient as on an MRNG.

(ii) The edge selection strategy of the MRNG treats all the other nodes as candidate neighbors of the current node, which causes a high time complexity. To speed up this process, we want to generate a small subset of candidates for each node. These candidates contain two parts: (1) As discussed above, the NNG is essential for monotonicity. Because it is very time-consuming to get the exact NNG, we turn to the approximate kNN graph. A high quality approximate kNN graph usually contains a high quality approximate NNG. It is acceptable when only a few nodes are not linked to their nearest neighbors. (2) Because the search on the NSG always starts from the Navigating Node p_n, for a given node p, we only need to consider those nodes which are on the search path from the p_n to p. Therefore we treat p as the query and perform Algorithm 1 on the prebuilt kNN graph. The nodes visited by the search and p's nearest neighbor in the approximate NNG are recorded as candidates. The nodes forming the monotonic path from the Navigating Node to p are very likely included in the candidates. When we perform MRNG's edge selection strategy on these candidates, it's very likely that the NSG inherits the monotonic path in the MRNG from the Navigating Node to p.

(iii) A possible problem in the above approach is the degree explosion problem for some nodes. Especially, the Navigating Node and nodes in dense areas will act as the "traffic hubs" and have high out-degrees. This problem is also discussed in HNSW [34]. They introduced a multi-layer graph structure to solve this problem, but their solution increased the memory usage significantly. Our solution is to limit the out-degrees of all the nodes to a small value $m \ll n$ by abandoning the longer edges. The consequence is the connectivity of the graph is no longer guaranteed due to the edge elimination.

To address the connectivity problem, we introduce a new method based on the DFS spanning tree as described above. After this process, all the nodes are guaranteed at least one path spreading out from the Navigating Node. Though the proposed method will sacrifice some performance in the worst case, the detours in the NSG will be minimized if we build a high-quality approximate kNN graph and choose a proper degree limitation m.

By approximating the MRNG, the NSG can inherit similar **low** search complexity as the MRNG. Meanwhile, the degree upper-bound makes the graph very **sparse**, and the tree-spanning operation guarantees the **connectivity** of the NSG. The NSG's index contains only a sparse graph and no auxiliary structures. Our method has made progress on all the four aspects compared with previous works. These improvements are also verified in our experiments. The detailed results will be presented in the later sections.

3.5.1 Indexing Complexity of NSG

The total indexing complexity of the NSG contains two parts, the complexity of the kNN graph construction and the preprocessing steps of NSG. In the million-scale experiments, we use the *nn*-descent algorithm [14] to build the approximate kNN graph on CPU. In the DEEP100M experiments, we use Faiss [28] to build it on GPU because the memory consumption of *nn*-descent explodes on large datasets. We focus on the complexity of Algorithm 2 in this section.

The preprocess steps of NSG include the search-collect-select operation and the tree spanning. Because the kNN graph is an approximation of the Delaunay Graph (an MSNET), the search complexity on it is approximately $O(kn^{\frac{1}{d}} \log n^{\frac{1}{d}} / \triangle r)$. We search for all the nodes, so the total complexity is about $O(kn^{\frac{1+d}{d}} \log n^{\frac{1}{d}} / \triangle r)$. The complexity of the edge selection is $O(nlc)$, where l is the number of the candidates generated by the search and c is the maximal degree we set for the graph. Because c and l are usually very small in practice (*i.e.*, $c \ll n, l \ll n$), this process is very fast. The final process is the tree spanning. This process is very fast because the number of the strongly connected components is usually much smaller than n. We only need to add a small number of edges to the graph. We can see that the most time-consuming part is the "search-collect" part. Therefore the total complexity of these processes is about $O(kn^{\frac{1+d}{d}} \log n^{\frac{1}{d}} / \triangle r)$, which is verified in our experimental evaluation in later sections. We also find that $\triangle r$ is almost a constant and does not influence the complexity in our experiments.

In the implementation of this paper, the overall empirical indexing complexity of the NSG is $O(kn^{\frac{1+d}{d}} \log n^{\frac{1}{d}} + f(n))$ ($f(n) = n^{1.16}$ with *nn*-descent and $f(n) = n \log n$ with Faiss), which is much lower than $O(n^2 \log n + cn^2)$ of the MRNG.

3.6 Search On NSG

We use Algorithm 1 for the search on the NSG, and we always start the search from the Navigating Node. Because the NSG is a carefully designed approximation of the

MRNG, the search complexity on the NSG is approximately $O(cn^{\frac{1}{d}} \log n^{\frac{1}{d}} / \triangle r)$ on average, where c is the maximal degree of the NSG, and d is the dimension. In our experiments, $\triangle r$ is about $O(n^{-\frac{\epsilon}{d}})$, $0 < \epsilon \ll d$. So the empirical average search complexity is $O(cn^{\frac{1+\epsilon}{d}} \log n^{\frac{1}{d}})$. Because $1 + \epsilon \ll d$, the complexity is very close to $O(\log n)$, which is verified in our experimental evaluation in later sections. Our code has been released on GitHub[2].

4. EXPERIMENTS

In this section, we provide a detailed analysis of extensive experiments on public and synthetic datasets to demonstrate the effectiveness of our approach.

4.1 Million-Scale ANNS

4.1.1 Datasets

Because not all the recent state-of-the-art algorithms can scale to billion-point datasets, this experiment is conducted on four million-scale datasets. SIFT1M and GIST1M are in the BIGANN datasets[3], which are widely used in related literature [7, 26]. RAND4M and GAUSS5M are two synthetic datasets. RAND4M and GAUSS5M are generated from the uniform distribution $U(0, 1)$ and Gaussian distribution $N(0, 3)$ respectively. Considering that the data may lie on a low dimensional manifold, we measure the local intrinsic dimension (LID) [11] to reflect the datasets' degree of difficulty better. See Table 1 for more details.

To prevent the indices from overfitting the query data, we repartition the datasets by randomly sampling one percent of the points out of each training set as a **validation set**. Since it's essential to be fast in the high-precision region (over 90%) in real scenarios, we focus on the performance of all algorithms in the high-precision area. We tune their indices on the validation set to get the best performance in the high-precision region.

4.1.2 Compared Algorithms

The algorithms we choose for comparison cover various types such as tree-based, hashing-based, quantization-based and graph-based approaches. The codes of most algorithms are available on GitHub and well optimized. For those who do not release their codes, we implement their algorithms according to their papers. They are implemented in C++, compiled by g++4.9 with "O3" option. The experiments of SIFT1M and GIST1M are carried out on a machine with i7-4790K CPU and 32GB memory. The experiments on RAND4M and GAUSS5M are carried out on a machine with Xeon E5-2630 CPU and 96GB memory. The indexing of NSG contains two steps, the kNN graph construction and Algorithm 2. We use the nn-descent algorithm [14] to build the kNN graphs.

Because not all algorithms support **inner-query parallelizing**, for all the search experiments, we only evaluate the algorithms with a single thread. Given that all the compared algorithms have the parallel versions for their index building algorithms, for time-saving, we construct all the indices with eight threads.

1. **Serial Scan** We perform serial scan on the base data to get the accurate nearest neighbors for the test points.

[2]https://github.com/ZJULearning/nsg
[3]http://corpus-texmex.irisa.fr/

Table 1: Information of experimental datasets. We list the dimension (D), local intrinsic dimension (LID) [11], the number of base vectors, and the number of query vectors.

dataset	D	LID	No. of base	No. of query
SIFT1M	128	12.9	1,000,000	10,000
GIST1M	960	29.1	1,000,000	1,000
RAND4M	128	49.5	4,000,000	10,000
GAUSS5M	128	48.1	5,000,000	10,000

2. **Tree-Based Methods. Flann**[4] is a well-known ANNS library based on randomized KD-tree, K-means trees, and composite tree algorithm. We use its randomized KD-tree algorithm for comparison. **Annoy**[5] is based on a binary search forest.

3. **Hashing-Based Methods.** FALCONN[6] is a well-known ANNS library based on multi-probe locality sensitive hashing.

4. **Quantization-Based Methods. Faiss**[7] is recently released by Facebook. It contains well-implemented codes for state-of-the-art product-quantization-based methods on both CPU and GPU. The CPU version is used here for a fair comparison.

5. **Graph-Based Methods. KGraph**[8] is based on a kNN Graph. **Efanna**[9] is based on a composite index of randomized KD-trees and a kNN graph. **FANNG** is based on a kind of graph structure proposed in [7]. They did not release their code. Thus, we implement their algorithm according to their paper. **HNSW**[10] is based on a hierarchical graph structure, which was proposed in [34]. **DPG**[11] is based on an undirected graph whose edges are selected from a kNN graph. According to an open source benchmark[12], **HNSW is the fastest ANNS algorithm on CPU so far.**

6. **NSG** is the method proposed in this paper. It contains only one graph with a navigating node where the search always starts.

7. **NSG-Naive** is a designed baseline to demonstrate the necessity of NSG's search-collect-select operation and the guarantee of the graph connectivity. We directly perform the edge selection strategy of MRNG on the edges of the approximate kNN graph to get NSG-Naive. There is no navigating node, thus, we use Algorithm 1 with random initialization on NSG-Naive.

4.1.3 Results

A. Non-Graph-Based v.s. Graph-Based. We record the numbers of distance calculations of Flann (Randomized KD-trees), FALCONN(LSH), Faiss(IVFPQ), and NSG on SIFT1M and GIST1M to reach certain search precision. In

[4]https://github.com/mariusmuja/flann
[5]https://github.com/spotify/annoy
[6]https://github.com/FALCONN-LIB/FALCONN
[7]https://github.com/facebookresearch/faiss
[8]https://github.com/aaalgo/kgraph
[9]https://github.com/ZJULearning/efanna
[10]https://github.com/searchivarius/nmslib
[11]https://github.com/DBWangGroupUNSW/nns_benchmark
[12]https://github.com/erikbern/ann-benchmarks

Table 2: Information of the graph-based indices involved in all of our experiments. AOD means the Average Out-Degree. MOD means the Maximum Out-Degree. The NN(%) means the percentage of the nodes which are linked to their nearest neighbor. Because HNSW contains multiple graphs, we only report the AOD, MOD, and NN(%) of its bottom-layer graph (HNSW$_0$) here.

dataset	algorithms	memory(MB)	AOD	MOD	NN(%)
SIFT1M	NSG	**153**	25.9	50	99.3
	HNSW$_0$	451	32.1	50	66.3
	FANNG	374	30.2	98	60.4
	Efanna	1403	300	300	99.4
	KGraph	1144	300	300	99.4
	DPG	632	165.1	1260	99.4
GIST1M	NSG	**267**	26.3	70	98.1
	HNSW$_0$	667	23.9	70	47.5
	FANNG	1526	29.2	400	39.9
	Efanna	2154	400	400	98.1
	KGraph	1526	400	400	98.1
	DPG	741	194.3	20899	98.1
RAND4M	NSG	$\mathbf{2.7 \times 10^3}$	174.0	220	96.4
	HNSW$_0$	6.7×10^3	161.0	220	76.5
	FANNG	5.0×10^3	181.2	327	66.7
	Efanna	6.3×10^3	400	400	96.6
	KGraph	6.1×10^3	400	400	96.6
	DPG	4.7×10^3	246.4	5309	96.6
GAUSS5M	NSG	$\mathbf{2.6 \times 10^3}$	146.2	220	94.3
	HNSW$_0$	6.7×10^3	131.9	220	57.6
	FANNG	5.2×10^3	152.2	433	53.4
	Efanna	7.8×10^3	400	400	94.3
	KGraph	7.6×10^3	400	400	94.3
	DPG	3.7×10^3	194.0	15504	94.3

our experiments, at the same precision, The other methods checks tens of times more points than NSG (see the figure in our technical report [16]). This is the main reason of the big performance gap between graph-based methods and non-graph based methods.

B. Check Motivation. In this paper, we aim to design a graph index with high ANNS performance from the following four aspects: (1) ensuring the connectivity of the graph, (2) lowering the average out-degree of the graph and (3) shortening the search path, and (4) reducing index size.

1. **Graph connectivity.** Among the graph-based methods, NSG and HNSW start their search with fixed node. To ensure the connectivity, NSG and HNSW should guarantee the other points are reachable from the fixed starting node. The others should guarantee their graph to be strongly connected because the search could start from any node. In our experiments, we find that, except NSG and HNSW, the rest methods have more than one strongly connected components on some datasets. Only NSG and HNSW guarantee the connectivity over different datasets (see the table in our technical report [16]).

2. **Lower the out-degree and Shorten the search path.** In **Table 2**, we can see that NSG is a very sparse graph compared to other graph-based methods. Though the bottom layer of HNSW is sparse, it's much denser than NSG if we take the other layers into consideration. It's impossible to count the search path lengths for each method because the query points are not in the base data. Considering that all the graph-based method use the same search algorithm and most

Table 3: The indexing time of all the graph-based methods. The indexing time of NSG is recorded in the form $t_1 + t_2$, where t_1 is the time to build the kNN graph, and t_2 is the time of Algorithm 2.

dataset	algorithms	time(s)	algorithm	time(s)
SIFT1M	NSG	**140+134**	HNSW	376
	FANNG	1860	DPG	1120
	KGraph	824	Efanna	355
GIST1M	NSG	**1982+2078**	HNSW	**4010**
	FANNG	34530	DPG	6700
	KGraph	4300	Efanna	4335
dataset	algorithm	time(h)	algorithm	time(h)
RAND4M	NSG	**2.1+2.5**	HNSW	5.6
	FANNG	38.3	DPG	6.0
	KGraph	4.9	Efanna	5.1
GAUSS5M	NSG	**2.3+2.5**	HNSW	6.7
	FANNG	46.1	DPG	6.4
	KGraph	5.1	Efanna	5.3

of the time is spent on distance calculations, the search performance can be a rough indicator of the term ol, where o is the average out-degree and l is the search path length. In **Figure 6**, NSG outperforms the other graph-based methods on the four datasets. NSG has a lower ol than other graph-based methods empirically.

3. **Reduce the index size.** In **Table 2**, NSG has the smallest indices on the four datasets. Especially, the index size of NSG is about 1/2-1/3 of the HNSW, which is the previous best performing algorithm[13]. It is important to note that, the memory occupations of NSG, HNSW, FANNG, Efanna's graph, and KGraph are all determined by the maximum out-degree. Although different nodes have different out-degrees, each node is allocated the same memory based on the maximum out-degree of the graphs to enable the continuous memory access (for better search performance). DPG cannot use this technique since its maximal out-degree is too large.

   The small index of the NSG owes to approximating our MRNG and limit the maximal out-degree to a small value. The MRNG provides superior search complexity upper-bound for NSG. We have tried different auxiliary structures to replace the Navigating Node or use random starting node. The performance is not improved but gets worse sometimes. This means NSG approximates the MRNG well, and it does not need auxiliary structures for higher performance.

C. Some Interesting Points:

1. It's usually harder to search on datasets with higher local intrinsic dimension due to the "curse of the dimensionality". In **Figure 6**, as the local intrinsic dimension increases, the performance gap between NSG and the other algorithms is widening.

2. When the required precision becomes high, the performance of many methods becomes even worse than that of the serial scan. NSG is tens of times faster than the serial scan at 99% precision on SIFT1M and GIST1M. On RAND4M and GAUSS5M, all the algorithms have lower speed-up over the serial scan. NSG is still faster than the serial scan at 99% precision.

[13]https://github.com/erikbern/ann-benchmarks

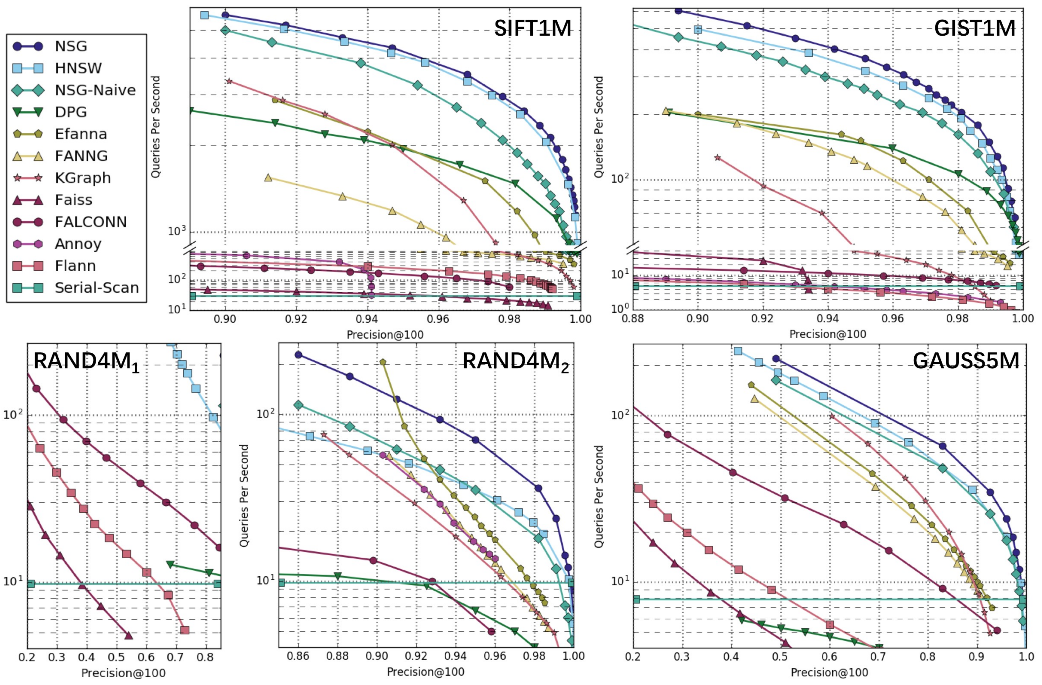

Figure 6: ANNS performance of graph-based algorithms with their optimal indices in high-precision region on the four datasets (top right is better). Some of the non-graph based methods have much worse performance than the graph-based ones. So we break the y-axis of SIFT1M and GIST1M figures, and we break the x-axis of the RAND4M figure (RAND4M$_1$ and RAND4M$_2$) to provide better view of the curves. The x-axis is not applicable for Serial-Scan because the results are accurate.

3. The indexing of NSG is almost the fastest among graph-based methods but is much slower than the non-graph-based methods. Due to the space limit, we only list the preprocessing time of all the graph-based methods in **Table 3**.

4. We count how many edges of the NNG are included in a given graph (**NN-percentage**) for all the compared graph-based methods, which are shown in **Table 2**. We can see that HNSW and FANNG suffer from the same problem: a large proportion of edges between nearest neighbors are missing (**Table 2**). It is because they initialize their graphs with random edges then refine the graphs iteratively. Ideally, they can link all the nearest neighbors when their indexing algorithms converge to optima, but they don't have any guarantee on its convergence, which will cause detour problems as we have discussed in Section 3.3. This is one of the reasons that the search performance of FANNG is much worse than the NSG. Another reason is that FANNG is based on RNG, which is not monotonic. HNSW is the second best-performing algorithm because HNSW enables fast short-cuts via multi-layer graphs. However, it results in very large index size.

5. The difference between NSG-Naive and NSG is that NSG-Naive does not select Navigating Node and does not ensure the connectivity of the graph. Moreover, the probability to reserve the monotonic paths is smaller because its candidates for pruning only cover a small neighborhood. Though NSG-Naive uses the same edge selection strategy, the degree of the approximation is inferior to NSG, which leads to inferior performance.

6. In the optimal index of KGraph and Efanna, the out-degrees are much larger than NSG. This is because the kNN graph used in KGraph and Efanna is an approximation of the Delaunay Graph. As discussed before, the Delaunay Graph is monotonic, which is almost fully connected on high dimensional datasets. When the k of the kNN graph is sufficiently large, the monotonicity may be best approximated. However, the high degree damages the performance of KGraph and Efanna significantly.

4.1.4 *Complexity And Parameters*

There are three parameters in the NSG indexing algorithm, k for the kNN graph; l and m for Algorithm 2. In our experiments, we find that the optimal parameters will not change with the data scale. Therefore we tune the parameters by sampling a small subset from the base data and performing grid search for the optimal parameters.

We estimate the search and indexing complexity of the NSG on SIFT1M and GIST1M. Due to the space limitation, please see our technical report [16] for the figures and the detailed analysis. The search complexity is about $O(n^{\frac{1}{d}} \log n^{\frac{1}{d}})$. The complexity of Algorithm 2 is about $O(n^{1+\frac{1}{d}} \log n^{\frac{1}{d}})$, where d is approximately equal to the intrinsic dimension. This agrees with our theoretical analysis. We estimate how the search complexity scales with K, the number of neighbors required. It's about $O(K^{0.46})$ or $O((logK)^{2.7})$.

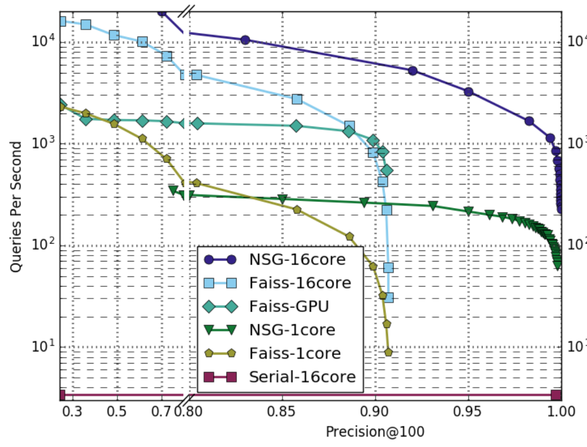

Figure 7: The ANNS performance of NSG and Faiss on the 100M subset of DEEP1B. Top right is better.

4.2 Search On DEEP100M

The DEEP1B is a dataset containing one billion float vectors of 96 dimension released by Artem *et al.* [5]. We sample 100 million vectors from it and perform the experiments on a machine with i9-7980 CPU and 96GB memory. The dataset occupies 37 GB memory, which is the largest dataset that the NSG can process on this machine. We build the kNN graph with Faiss [28] on four 1080Ti GPUs. The time of building the kNN graph is 6.75 hours and the time of Algorithm 2 is 9.7 hours. The peak memory usage of NSG at indexing stage is 92GB, and it is 55 GB at searching stage. We try to run HNSW, but it always triggers the Out-Of-Memory error no matter how we set the parameters. So we only compare NSG with Faiss. The result is in **Figure 7**.

NSG-1core means we build one NSG on the dataset and evaluate its performance with one CPU core. **NSG-16core** means we break the dataset into 16 subsets (6.25 million vectors each) randomly and build 16 NSG on these subsets respectively. In this way, we can enable inner-query parallel search for NSG by searching on 16 NSGs simultaneously and merging the results to get the final result. We build one Faiss index (IVFPQ) for the 100 million vectors and evaluate its performance with one CPU-core (**Faiss-1core**), 16 CPU-core (**Faiss-16 core**), and 1080Ti GPU (**Faiss-GPU**) respectively. Faiss supports inner-query parallel search. **Serial-16core** means we perform serial scan in parallel with 16 CPU cores.

NSG outperforms Faiss significantly in high-precision region. NSG-16core outperforms Faiss-GPU and is about 430 times faster than Serial-16core at 99% precision. Meanwhile, building NSG on 6.25 million vectors takes 794 seconds. The total time of building 16 NSGs sequentially only spends 3.53 hours, which is much faster than building one NSG on the whole DEEP100M. The reason may be as follows. The complexity of Algorithm 2 is about $O(n^{1+\frac{1}{d}} \log n^{\frac{1}{d}})$. Suppose we have a dataset D with n points. We can partition D into r subsets evenly. The time of building one NSG on D is t_1. The time of building an NSG on one subset is t_2. It is easy to verify that we can have $t_1 > rt_2$ if we select a proper r. Consequently, sequential indexing on subsets can be faster than on the complete set.

4.3 Search In E-commercial Scenario

We have collaborated with Taobao on the billion-scale high-dimensional ANNS problem in the E-commercial scenario. The billion-scale data, daily updating, and response time limit are the main challenges. We evaluate NSG on the E-commercial data (128-dimension vectors of users and commodities) with different scales to work out a solution.

We compare NSG with the baseline (a well-optimized implementation of IVFPQ [26]) on the e-commerce database. We use a 10M dataset to test the performance on a single thread, and a 45M dataset to test the **Distributed Search** performance in a simulation environment. The simulation environment is a online scenario stress testing system based on MPI. We split the dataset and place the subsets on different machines. At search stage, we search each subset in parallel and merge the results to return. In our experiments, we randomly partition the dataset evenly into 12 subsets and build 12 NSGs. NSG is 5-10 times faster than the baseline at the same precision (See our technical report [16] for details) and meet the response time requirement.

On the complete dataset (about 2 billion vectors), we find it impossible to build one NSG within one day. So we use the distributed search solution with 32 partitions. The average response time is about 5 ms at 98% precision, and the indexing time is about 12 hours for a partition. The baseline method (IVFPQ) cannot reach the response time requirement (responsing within 10 ms at 98% precision) on the complete dataset.

5. DISCUSSIONS

The NSG can achieve very high search performance at high precision, but it needs much more memory space and data-preprocessing time than many popular quantization-based and hashing-based methods (*e.g.*, IVFPQ and LSH). The NSG is very suitable for high precision and fast response scenarios, given enough memory. In frequent updating scenarios, the indexing time is also important. Building one NSG on the large dataset is impractical. The distributed search solution like our experiments is a good choice.

It's also possible for NSG to enable incremental indexing. We will leave this to future works.

6. CONCLUSIONS

In this paper, we propose a new monotonic search network, MRNG, which ensures approximately logarithmic search complexity. We propose four aspects (ensuring the connectivity, lowering the average out-degree, shortening the search paths, and reducing the index size) to design better graph structure for massive problems. Based on the four aspects, we propose NSG, which is a practical approximation of the MRNG and considers the four aspects simultaneously. Extensive experiments show the NSG outperforms the other state-of-the-art algorithms significantly in different aspects. Moreover, the NSG outperforms the baseline method of Taobao (Alibaba Group) and has been integrated into their search engine for billion-scale search.

7. ACKNOWLEDGEMENTS

The authors would like to thank Tim Weninger and Cami G Carballo for their invaluable input in this work.This work was supported by the National Nature Science Foundation of China (Grant Nos: 61751307).

8. REFERENCES

[1] F. André, A.-M. Kermarrec, and N. Le Scouarnec. Cache locality is not enough: high-performance nearest neighbor search with product quantization fast scan. *PVLDB*, 9(4):288–299, 2015.

[2] A. Arora, S. Sinha, P. Kumar, and A. Bhattacharya. Hd-index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. *PVLDB*, 11(8):906–919, 2018.

[3] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 271–280, 1993.

[4] F. Aurenhammer. Voronoi diagramsa survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[5] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *Acm*, 19(2):322–331, 1990.

[7] H. Ben and D. Tom. FANNG: Fast approximate nearest neighbour graphs. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] M. Boguna, D. Krioukov, and K. C. Claffy. Navigability of complex networks. *Nature Physics*, 5(1):74–80, 2009.

[10] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2005.

[11] J. A. Costa, A. Girotra, and A. Hero. Estimating local intrinsic dimension with k-nearest neighbor graphs. *Statistical Signal Processing, 2005 IEEE/SP 13th Workshop on*, pages 417–422, 2005.

[12] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient k-nn search on vertically decomposed data. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 322–333, 2002.

[13] D. Dearholt, N. Gonzales, and G. Kurup. Monotonic search networks for computer vision databases. *Signals, Systems and Computers, 1988. Twenty-Second Asilomar Conference on*, 2:548–553, 1988.

[14] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th international Conference on World Wide Web*, pages 577–586, 2011.

[15] C. Fu and D. Cai. Efanna : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv:1609.07228*, 2016.

[16] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2018.

[17] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, 100(7):750–753, 1975.

[18] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1127–1138, 2014.

[19] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

[20] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *PVLDB*, pages 518–529, 1999.

[21] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. *Proceedings of the International Joint Conference on Artificial Intelligence*, 22:1312–1317, 2011.

[22] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[23] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *PVLDB*, 9(1):1–12, 2015.

[24] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[25] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, 1992.

[26] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[27] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE transactions on cybernetics*, 44(11):2167–2177, 2014.

[28] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv:1702.08734*, 2017.

[29] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798):845–845, 2000.

[30] G. D. Kurup. Database organized on the basis of similarities with applications in computer vision. *Ph.D. thesis, New Mexico State University*, 1992.

[31] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement (v1. 0). *arXiv:1610.02455*, 2016.

[32] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. Sk-lsh: an efficient index structure for approximate nearest neighbor search. *PVLDB*, 7(9):745–756, 2014.

[33] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[34] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv:1603.09320*, 2016.

[35] C. D. Manning, P. Raghavan, H. Schütze, et al. Introduction to information retrieval. *Cambridge university press Cambridge*, 1(1), 2008.

[36] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[37] G. Teodoro, E. Valle, N. Mariano, R. Torres, W. Meira, and J. H. Saltz. Approximate similarity search for online multimedia services on distributed cpu–gpu platforms. *The VLDB Journal*, 23(3):427–448, 2014.

[38] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.

[39] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *PVLDB*, 98:194–205, 1998.

[40] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. *Advances in neural information processing systems*, pages 1753–1760, 2009.

[41] Y. Wu, R. Jin, and X. Zhang. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1139–1150, 2014.

[42] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. *Proceedings of the 2016 International Conference on Management of Data*, pages 2023–2037, 2016.