Fast Approximate Nearest Neighbor Search With Navigating Spreading-out Graphs

Cong Fu Changxu Wang Deng Cai
State Key Lab of CAD&CG, College of Computer Science, Zhejiang University, China
{fc731097343, changxu.mail, dengcai}@gmail.com

Abstract

Approximate nearest neighbor search (ANNS) is a fundamental problem in many tasks of machine learning and computer vision. An ANNS algorithm is required to be efficient on both memory and search performance. Recently, several graph based methods have achieved revolutionary performance on public data sets, but their indices are usually complicated and of large size. In this paper, we present a novel search algorithm based on a Navigating Spreading-out Graph (NSG). All the search begins with the navigating vertex and iteratively examines the neighbors’ neighbors for closer neighbors on NSG. An NSG contains only a navigating vertex and a spreading-out graph, in which the angles among any two connected edges are ensured to be no smaller than 60°. We propose an efficient algorithm to build the NSG, and the max degree of resulting NSG is very small, thus it’s quite memory-efficient. Extensive experiments on public datasets show that our approach outperforms the state-of-art algorithms significantly on both index size and search performance.

1. Introduction

Approximate nearest neighbor search (ANNS) has been a hot topic over decades and provides fundamental support for many applications in data mining, machine learning and computer vision[1, 12]. For sparse data (like documents), efficient nearest neighbor search can be carried out on advanced index structures (e.g. inverted index[14]). For dense data, various solutions have been proposed such as tree-structure based approaches, hashing based approaches, and graph based approaches.

Tree based approaches such as KD-tree [3], Randomized KD-tree[15], K-means tree[8] follow a similar routine. They build tree-like structure(s) and perform depth-first-search on them for given query point. These approaches perform well on data of low dimension, but dramatically worse when dimension grows[15].

Hashing based approaches, such as Locality Sensitive Hashing (LSH) [9] and Spectral Hashing [16], try to generate binary codes for multi-dimensional real vectors while preserving original similarity among them, and the hash tables are used as indices. In the ideal case, similar real vectors should have the same or the similar binary codes (measured by the hamming distance), thus fall into the same or nearby hash buckets. However, the methods proposed so far are far from ideal state. They need a large search radius in hamming space to ensure a high accuracy, which means a huge quantity of hash buckets to be checked and results in an inferior performance[4].

Recently, graph based ANNS methods received considerable interests and show promising results. These methods carry out a greedy search on graph(s) built from stored data. The basic idea is a neighbor of a neighbor is also likely to be a neighbor[10, 6]. By iteratively checking neighbors’ neighbors in the graph (can be named as NN-expansion), these methods can efficiently find the true neighbors of the query. Previous work [7, 11, 4] has proved experimentally that graph based approaches outperform the traditional tree based and hashing based methods significantly.

The graph based approaches have various types in terms of the graph structure they use. The first kind approaches are based on a traditional approximate \( k \) nearest neighbor graphs (\( k \)NN Graph), including GNNS [10], KGraph[5], IEH[11] and Efanna[7]. A \( k \)NN Graph reserves \( k \) nearest neighbors of each vertex according to a given metric. Kiana et al. [10] first proposed a algorithm called GNNS to perform greedy search on a \( k \)NN Graph, according to the NN-expansion heuristics. KGraph[5] follows the same idea but with different implementation. Both of the two algorithms start from random selected vertices in graph and perform NN-expansion until the algorithms converge. However, They are easy to get trapped in local optimal when going through low density area of the data set. Efanna[7] and IEH[11] use additional structures (truncated KD-tree or hashing) to get better initial vertices, from which they refine the results with NN-expansion on a \( k \)NN Graph. According to their experiments, Efanna and IEH outperforms GNNS.
and KGraph greatly on public data sets[7]. However, $k$NN Graph may not be the optimal graph structure for search. Recently, Harwood et al. [2] presented an occlusion rule, which ensures that the angles among any two connected edges are no smaller than 60°. Therefore, the edges stretch widely outwards from all the vertices intuitively. They proposed an algorithm to build their fast approximate nearest neighbor graph (FANNG) according to the occlusion rule, and experimentally proved that with the same search algorithm, the performance on a FANNG is significantly better than that on a $k$NN Graph (the average out-degree of vertices in the FANNG and the $k$NN Graph are the same). However, the algorithm[2] to build a FANNG is quite time-consuming.

Malkov et al. proposed Hierarchical Navigable Small World graphs (HNSW)[13] and achieves the best performance on search so far, according to an open source ANNS benchmark on GitHub1. They build HNSW in a hierarchical way from top to bottom. The graph scale shrinks layer by layer upwards, and each layer of HNSW also meets the occlusion rule (called heuristic neighbor selection in [13]). But there is additional memory cost for the upper layer structures, which is a common drawback for approaches requiring additional structures.

Inspired by the above observations, we propose a novel algorithm to build a Navigating Spreading-out Graph (NSG) and enable faster search on it. An NSG is a one-layer graph built out of an approximate $k$NN Graph, and the building process all starts from a pre-chosen navigating vertex (the medoid of the data set) and collect non-occluding edges for each vertices. Extensive experiments show that our approach outperforms the state-of-art algorithms significantly on both index size and search performance.

It is worthwhile to highlight the following points:

1. All the three kinds of graphs, FANNG, HNSW and NSG, satisfy the occlusion rule. Since the occlusion rule is a very weak constraint, there are numerous graphs satisfying the occlusion rule. Different graph construction algorithms will lead to totally different graph structures.

2. An NSG can be built significantly faster than a FANNG, given that both of them are single layer graphs. Meanwhile, search with an NSG is significantly better than with a FANNG.

3. An NSG is a one-layer graph, which is very memory-efficient in contrast with HNSW which is a hierarchical graph structure. While the search on NSG can still outperform that on HNSW.

4. The NSG is built out of an approximate $k$NN Graph, which ensures the NSG has the property that for each vertex in the NSG, the total length of its out-going edges is likely to be the shortest among all the possible non-occluding edge sets with same number of edges. Intuitively, the edges of the NSG are guaranteed to spread well into local neighborhood of each vertex from nearest area to farther.

2. Graph Based Approximate Nearest Neighbor Search

Approximate nearest neighbor search on all kinds of graphs actually follows the same heuristics, which was first proposed in Kiana et al.’s work[10]. For a given query, the algorithm starts from given vertices and iteratively checks the neighbors’ neighbors and keep the closest ones until it makes no progress on the graph. This Naive Heuristic Greedy search (NHGsearch) is given in Algorithm.1. The GNNS they proposed in [10] is to repeat Algorithm.1 $R$ times to get the best results., where $R$ is an additional parameter for GNNS.

However, Algorithm.1 gets trapped in local optimal very easily. One improved version of the algorithm is to maintain a fix-sized candidates pool and keep it sorted in ascending order of distance to query. The pool is updated frequently to ensure the vertices in it are the closest ones to query, and additional bookkeeping will be done to prevent repeated neighbor checking. This Improved Heuristic Greedy search algorithm (IHGsearch) is given more formally in Algorithm.2. This algorithm can be apply to all kinds of graph structures. KGraph[3], IEH[11] and Efanna[7] used it on a $k$NN Graph. Harwood et al. [2] used it on their FANNG, while Malkov et al. [13] used it on the bottom layer of HNSW and Algorithm.1 on the upper layers.

Recently, Harwood et al. [2] showed that the $k$NN Graph is not optimal for search. They believe that in the ideal graph, we can use Algorithm.1 to begin with each vertex to get to any other vertex, meanwhile, the total number of edges should be minimized. We may call this property greedy reachable property.

---

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

---

**Algorithm 1 NHGsearch($G, p, q, l$)**

**Require:** graph $G$, start vertex $p$, query point $q$, candidate pool size $l$

**Ensure:** $k$ nearest neighbors of query point $q$

1: candidate pool $S = \emptyset$
2: $S.add(p)$
3: while $S$ still changes do
4: add all neighbors of vertex $S[0]$ into $S$.
5: sort $S$ in ascending order of distance to $q$.
6: remove the distant vertices in $S$ to keep its size no larger than $l$
7: end while
8: return the first $k$ vertices in $S$. 
Algorithm 2 IHGsearch(G, p, q, l)

Require: graph G, start vertex p, query point q, candidate pool size l
Ensure: k nearest neighbors of q
1: i=0
2: candidate pool S = ∅
3: S.add(p)
4: while i < l do
5: p_i = first unchecked vertex in S
6: i = position of p_i in S
7: mark p_i as checked
8: for all neighbor n of p_i in G do
9: S.add(n)
10: end for
11: sort S in ascending order of distance to q
12: remove the distant vertices in S to keep its size no larger than l
13: end while
14: return the first k vertices in S

Harwood et al. [2] also presented the occlusion rule to remove the “redundant” edges, which actually needn’t be checked by Algorithm 1 in a kNN Graph. For example, for vertex p_1, edge p_1p_3 is the longest in triangle p_1p_2p_3 (Fig. 1). Because Algorithm 1 always checks the shortest edge in the first place, the search process will go from p_1 to p_2 and then check edges outgoing from p_2. Thus the edge p_1p_3 is of less importance. To Harwood et al.’s opinion, the edge p_1p_2 occludes the edge p_1p_3, thus the edge p_1p_3 will be removed. Intuitively, the occlusion rule tends to keep shorter edges and ensures the edges around each vertex stretching out well to its local neighborhood.

For a kNN Graph and a FANNG with the same average out-degree of the vertices, considering a given vertex, the FANNG ensures its outgoing edges covering a much larger range than the kNN Graph. Meanwhile, they proved experimentally in their work that carrying out the same search algorithm on a FANNG outperforms that on a graph satisfying the occlusion rule. The navigating vertex is the medoid of the given dataset. We first compute the mean of the dataset and use Algorithm 2 to search for the medoid on a pre-built approximate kNN Graph. The graph of NSG is also constructed out of the kNN Graph. And the construction algorithm of the graph is formally given in the following sections. As for the kNN Graph, we use Efanna[7] to build it because it’s the fastest kNN Graph building algorithm to the best of our knowledge.

3. Navigating Spreading-out Graphs

In this section, we will present our novel Navigating Spreading-out Graph (NSG) and how to search on it. An NSG contains a navigating vertex and a graph satisfying the occlusion rule. The navigating vertex is the medoid of the given dataset. We first compute the mean of the dataset and use Algorithm 2 to search for the medoid on a pre-built approximate kNN Graph. The graph of NSG is also constructed out of the kNN Graph. And the construction algorithm of the graph is formally given in the following sections. As for the kNN Graph, we use Efanna[7] to build it because it’s the fastest kNN Graph building algorithm to the best of our knowledge.

3.1. Navigating Spreading-out Graph Building

Due to that the graph structure of FANNG is superior to kNN Graph in terms of ANNS problem but is time costing to be built, we hope to build a graph according to the occlusion rule but with small time cost. Considering that if all the searches begin with the same vertex, then we needn’t guarantee all the vertex satisfying the greedy reachable property. FANNG builds the graph from an empty graph, randomly links the vertex initially and refines the graph with Algorithm 2. They hope to gather non-occluding edges for each vertex along with the converging process of the graph. However, during the graph converging process, whenever a new edge comes, the occlusion property among this edge and old ones will be corrupted. Too many edges are absorbed in but latter abandoned during the long converging process, which may be the main reason why it’s so slow.

Instead of letting the algorithm making blind walks among vertex on a coarse graph at early stage, we build our NSG from an approximate kNN Graph of high quality. Meanwhile, thinking about the greedy reachable property, if

---

2See open source ann-benchmarks: https://github.com/erikbern/ann-benchmarks
we carry out Algorithm 2 on a kNN Graph from vertex p to q, the computation is heavy because there are so many covered edges offering little or even no contribution. Among those covered edges, there must exist a path satisfying the greedy reachable property. We just need to cut them out with occlusion rule. And we only need to carry out this process from the navigating vertex to all the other vertices once. If all the vertices in NSG is greedy reachable from the navigating vertex, then all the nearest neighbors of the given query are greedy reachable from the navigating vertex.

Here comes to our NSG building algorithm quite straightforwardly. Firstly, we choose a navigating vertex N, in our implementation, the medoid. Next, we carry out heuristic greedy search from N to all the other vertices. For efficiency, we use our version but with a little modification from Algorithm 2, which is to add another candidate cutting pool to record all the covered vertices. Then we merge the pool with neighbors in the guidebook kNN Graph. Finally we cut the edges according to the occlusion rule and reserve k edges which is a given parameter. The details are given in Algorithm 3, Navigating Spread-out Graph Building Algorithm (NSGbuild).

### Algorithm 3 NSGbuild(G, l, k)

**Require:** kNN Graph G, candidate pool size l for greedy search, maximum out-degree k of NSG.

**Ensure:** NSG with navigating vertex n

```python
1. n = medoid of vertices in G.
2. for all vertex v in G do
3.   % heuristic greedy search with additional candidate cutting pool recording visited vertices
4.   cutting pool E = IHGsearch(G, n, v, l)
5.   merge E with v’s neighbors in G
6.   sort E in ascending order of distance to v
7.   result set R = ∅, p₀ = E[0]
8.   R.add(p₀)
9.   while R.size() < k && !E.empty() do
10.  p = E.front()
11.  E.remove(E.front())
12.  for all vertex r in R do
13.    if edge pv occluded by rv then
14.      break
15.    end if
16.  end for
17.  if no occlusion occurs then
18.    R.add(p)
19.  end if
20. end while
21. end for
```

### Algorithm 4 HNSGbuild(V, level, k, p)

**Require:** Vertex set V, maximum level level, maximum out-degree k of NSG, candidate pool size p for greedy search.

**Ensure:** HNSG with navigating vertex n

```python
1. V_temp = V
2. for l = 1 : level do
3.   build kNN Graph G_l on V_temp
4.   HNSG[l] = NSGbuild(G_l, p, k)
5.   V_temp = ∅
6.   V_temp.add(n)
7.   for all vertex v in HNSG[l] do
8.     nearest neighbors V_nn = NHGSearch(HNSG[l], n, v, p)
9.     V_temp.add(V_nn[0])
10. if v_k = V_nn[0] then
11.   V_temp.add(v)
12. end if
13. end for
14. end for
```

3.2. Hierarchical NSG

Although we try to meet the greedy reachable property for the navigating vertex as possible as we can, the resulting NSG still cannot read 100% greedy reachability with the limit of parameter configuration and the quality of kNN Graph. To further improve the search performance, a straightforward way is to build NSG hierarchically. But we are different from HNSW[13], which builds the index top-down and randomly choose vertices to put into each layer. We carry out Algorithm 1 on resulting NSG, and record the vertex pair including the vertex where the search stops and the object vertex we search for. Then all these recorded vertices and the navigating vertex will be chosen as vertices of upper layer. Intuitively, performing NSG building algorithm on this subset (smaller scale of vertices) will force those unreachable distant vertex to be linked together. See Algorithm 4 Hierarchical Navigating Spread-out Graph Building Algorithm (HNSGbuild) for more details. What’s more, for flexibility, the maximum out-degree of each layer can be different.

3.3. Search With NSG

Search with NSG is quite simple. For any query, we just apply Algorithm 2 in which we use the navigating vertex as the start vertex.

Our codes will be released at GitHub soon.
Table 1. Information on Experimental Data Sets

| data set   | dimension | base number | query number |
|------------|-----------|-------------|--------------|
| SIFT1M     | 128       | 1,000,000   | 10,000       |
| GIST1M     | 960       | 1,000,000   | 1,000        |

4. Experiments

In this section, we will give a detailed illustration and analysis of extensive experiments on public datasets to demonstrate the effectiveness of our approach, which includes experiments on both NSG based search and NSG construction.

4.1. Data Set and Experiment Setting

The experiments are conducted on two datasets, SIFT1M and GIST1M, in the well-known BIGANN public dataset collection for approximate nearest neighbor search\(^3\). See Table 1 for more details about the datasets. All of the codes are implemented in C++, compiled with g++4.9 and optimized with “O3” option. For fair comparison, we use SIMD instructions (AVX) for acceleration on distance calculation, which is used in all of compared algorithms. All the experiments are carried out on a machine with i7-4790 CPU and 32G memory. We test each parameter setting 10 times to get an average for each algorithm.

Because the parallelization in search is just inter-query parallelizing, which is trivial. For all the search experiments, we just need to evaluate the algorithms on 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 8 threads.

4.2. Evaluation Protocol

For ANNS problem, an algorithm is expected to return $K$ points. We need to verify how many of them are true $K$ nearest neighbors (ground truth). Therefore, we use the precision defined in \([7, 4]\) as our evaluation protocol. Suppose the returned point set of a given query $q$ is $R'$ and the true $k$ nearest neighbor set of $q$ is $R$, then the precision is defined as below.

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

We will report the performance with $K = 10$ and 100 throughout all the experiments.

4.3. Compared Algorithms

Given that previous work \([7, 4, 11]\) has experimentally proved that graph based methods are significantly better than traditional tree based and hashing based methods. And it’s also showed that Efanna\([7]\) outperforms many other $k$NN Graph based methods in their work. We just choose some state-of-art graph based methods as below for comparison.

1. **KGraph** KGraph has open source code on \([5]\), which is based on a $k$NN Graph. It uses Algorithm 2 as search method and uses random points as the initialization of the algorithm. The source code on GitHub is used for comparison.

2. **Efanna** Efanna is a $k$NN Graph based method proposed in \([7]\). It uses truncated Randomized KD-trees to get better initial points and uses Algorithm 2 to refine them. The source code on GitHub\(^4\) is used for comparison.

3. **FANNG** FANNG is a kind of graph structure proposed in \([2]\) which is quite different from a $k$NN Graph as described in previous sections. They proved experimentally that with same average-out-degree, search on a FANNG is much better than on a $k$NN Graph. They use Algorithm 2 as their search method. Because they didn’t release their code and they conducted their experiments on GPU, we implement their algorithms on our own in C++ with same optimization as all the other algorithms for fair.

4. **HNSW** HNSW is a hierarchical graph structure, which was propose in \([13]\). The graph structure of each layer satisfies the occlusion rule, thus is quite similar to FANNG. According to an open source benchmark\(^5\), they are the fastest algorithm on CPU by now. They use Algorithm 1 on the upper layer and Algorithm 2 on the bottom layer. The source code on GitHub\(^6\) is used for comparison.

5. **NSG** NSG is the index proposed in this paper. It contains a navigating point and a graph meeting the occlusion rule. We use Algorithm 2 as our search algorithm. The codes will be released on GitHub soon.

4.4. Overall Comparison

We run all the algorithms on SIFM1M and GIST1M, tune them to get the best performance and record the time and corresponding precision for each algorithm to get the curve on precision against queries per second. And the results are shown in Figure 2. The results of index building is given in Table 2.

---

\(^3\)http://corpus-texmex.irisa.fr/

\(^4\)https://github.com/fc731097343/efanna

\(^5\)https://github.com/erikbern/ann-benchmarks

\(^6\)https://github.com/searchivarius/nmslib
4.4.1 Parameters and Indices Configuration

The search performance of KGraph is affected largely by the out-degree ($k$) of the $k$NN Graph used. We try many graphs and choose the best performing 250NN Graph for KGraph on SIFT1M and 300NN Graph on GIST1M. There are two essential parameters, trials $T$ and expansion size $P$, controlling the accuracy. We tune it to get the curves.

Efanna’s index includes two parts, truncated Randomized KD-trees and a $k$NN Graph. The performance of Efanna is affected largely by the $k$NN Graph it uses. The best performing and meanwhile smallest configuration is 32 trees and 150NN Graph for SIFT1M and 32 trees and 300N Graph GIST1M. We tune the initial pool size $L$, expansion size $P$ and iterations $T$ to get the curves.

The best index we get for FANNG on SIFT1M is a FANNG with max trial $t = 25,000,000$ and second stage max iteration $l = 15$, the resulting graph is of average out-degree 25 and max out-degree 98. While on GIST1M, the best FANNG is built with max trial $t = 400,000,000$ and second stage max iteration $l = 15$, the resulting graph is of average out-degree 25 and max out-degree 400. For search, there is only one parameter, expansion size $l$, to tune to get the curves.

There are two main parameters controlling the quality of HNSW indices, max degree of each layer $M$ and search range $efConstruction$. $M$ controls both index size and quality, while $efConstruction$ mainly affects the quality. We search $M$ in the range of 4 to 50 and 200 to 700 for $efConstruction$. And the best index on SIFT1M is an HNSW with $M = 24$ and $efConstruction = 500$, while on GIST1M is one with $M = 35$ and $efConstruction = 600$. We tune the search parameter $efSearch$ to get the curves.

The best index we get for NSG on SIFT1M is an NSG built from a 40NN Graph, with candidate pool size $l = 80$ and max out-degree $k = 40$. On GIST1M, the best NSG is built from a 300NN Graph, with candidate pool size $l = 150$ and max out-degree $k = 70$. For search, we tune the parameter expansion size $l$ to get the curves.

See our supplementary material for more details on parameter choosing.

4.4.2 Results

On SIFT1M, our approach achieves comparable performance on both 10NN and 100NN cases with HNSW. While on GIST1M (much higher dimension), our approach achieves better performance than HNSW and is the best one. This may be because our graph structure is much better than any layer of HNSW, we can outperform HNSW with only one layer graph. The performance of Efanna and KGraph relies on large $k$NN Graphs, which results in large memory use and index building time. However, they still perform much worse than ours.

In terms of index size, NSG is the smallest among all the algorithms, and is only about one third of the indices of HNSW on both datasets. On SIFT1M, the NSG construc-
tion time is also the smallest. HNSW is fastest on index building for GIST1M. This is because the NSG should be built from an approximate $k$NN Graph. On SIFT1M, we just build the NSG from a 40NN Graph, which is fast because SIFT1M is a quite simple dataset. While on GIST1M, the NSG is built from a 300NN Graph, which is quite time consuming (about 4300 seconds). However, HNSW performs worse than ours on search on GIST1M and has much larger index size.

Our approach outperforms FANNG significantly with much smaller index size and much shorter index building time. This is because without a $k$NN Graph as a “template”, the building algorithm of FANNG takes lots of time on gathering non-occlusion edges, most of which are cut off latter. Moreover, their graph converges very slow and doesn’t guarantee the edges stretch well from the closest area to farther. And this is also the reason that they under-perform our approach on search a lot. The memory use of FANNG is much larger than ours and especially large on GIST1M. See the analysis in the next section for more details.

4.5. Comparison between NSG, KGraph and FANNG

In Harwood et al.’s work[2], they have shown that, with the same search algorithm and average-out-degree of the graphs, search performance on the FANNG is significantly better than on the $k$NN Graph. In this section, we compare the search performance on a NSG, a FANNG and $k$NN Graphs with the same search method, Algorithm.2. The results are shown in Figure.3.

It’s important to note that:

1. The NSG and FANNG we use are the best performing graphs we can build from respective construction algorithms.

2. Search with Algorithm.2 on a $k$NN Graph is actually how KGraph[5] works. So we just use their code for the search experiments on a $k$NN Graph.

In Harwood et al.’s work[2], they compare their approach with the search on $k$NN Graphs. The results in Figure.3 show that the performance of our implementation agrees with what they claimed. With same average out-degree, search on a FANNG is far more efficient than on a $k$NN Graph.

However, in software engineering perspective, the graph can be treated as a table and each entry recording the neighbors of the respective vertex. The graph structure can be designed as linked lists or array in memory. For better cache locality and fast access to the graph, which greatly affects search speed, the best way to place the graph in memory is to allocate continuous memory for it and align entries of the graph to same length. In this way, the max degree of the graph determines the graph size in memory.

On SIFT1M, the KGraph25 25 has the same average out-degree as the NSG, and the KGraph40 40 has the same size as the NSG in memory. ON GIST1M, the KGraph25 25 has the same average out-degree as the NSG, and the KGraph70 70 has the same size as the NSG in memory. From Figure.3, we can see that search on the NSG is much better than on a $k$NN Graph of same average out-degree and a $k$NN Graph of the same max out-degree (the same size in memory) on both datasets. This is because NSG has much better structure than the $k$NN Graph for search. It also illustrates that all the $k$NN Graph based method can be improved by employing NSG, however, it is beyond the scope of this paper.

Combined with Figure.3 and Table.2, we can also draw a conclusion that our approach performs significantly better than FANNG on search, index building time and index size in memory, which illustrates that the graph structure of the NSG is superior to the FANNG, and the index building algorithm of the NSG is much more efficient, too.

4.6. Comparison between NSG and HNSW

To the best of our knowledge, HNSW is the most efficient algorithm on ANNS. For large scale ANNS scenario, a good algorithm should be not only efficient on search, but
also efficient on memory use, in other words, it should have small index. Here we compare HNSW with NSG comprehensively on both search efficiency and index size.

We first carry out some experiments to get the parameters of best performing indices for HNSW on SIFT1M and GIST1M. We compare the search on best NSG with best HNSW in previous section and an HNSW with same index size on the two datasets. The size of best NSG is 153MB on SIFT1M and 267MB on GIST1M in Table 2. HNSW shrinks to the same index size as NSG when we set $M = 4$ on SIFT1M and $M = 10$ on GIST1M. Then we tune $efConstruction$ to get the best index under respective index size constraints. The best HNSW of 155MB is built with $M = 4$ and $efConstruction = 700$ for SIFT1M.

And for GIST1M, the best HNSW of 267MB is built with $M = 10$ and $efConstruction = 1000$. Finally we tune the $efSearch$ to get the curves in Figure 4.

In big data scenario, index size usually grows in proportion to data size. It’s vital for indices to be memory-efficient. In the Figure 4, our approach achieves comparable performance with best HNSW on SIFT1M, and better performance on GIST1M. However, our index size in memory is significantly smaller than their best indices. And when we add the same-index-size constraint to HNSW, it performs much worse than our approach. This is a clear demonstration that the graph structure of HNSW is quite inferior to the NSG. In other words, our approach uses the least edges to get a best performing graph.

### 4.7. Hierarchical NSG

We build a Hierarchical NSG (HNSG) based on the best NSG on SIFT1M, and compare it with the NSG on SIFT1M. The result is shown in Figure 5.

In this experiment, we just build a two-layer NSG. In our experiments, more layers doesn’t provide better performance. And From Figure 5, the two-layer HNSG outperforms the NSG and the best HNSW a little on SIF1M, with a additional memory cost of 76.3 MB (total 230 MB). This may be because the performance of a NSG is close to the up limits, the improvement is small. The size of HNSG in memory is only a half of HNSW, which is 451MB.

On GIST1M, the HNSG has just the same performance as NSG. This is because GIST1M is a much harder dataset than SIFT1M. We find that the greedy search process stops quickly at local optimal on upper layer. And for high accuracy, the distance calculations on bottom layer are hundreds times of those on upper layer. The contribution of the upper layer can be almost ignored. On the contrary, for SIFT1M, the distance calculations on bottom layer are only several times of those on upper layer for high accuracy.

### 5. Conclusions

In this paper, we present a new kind of graph index, NSG, to carry out efficient approximate nearest neighbor search. Our approach outperforms other state-of-the-art algorithms on public datasets. Particularly, NSG outperforms HNSW[13], which is the most efficient open source ANNS algorithm so far. In addition, The NSG is much more memory-efficient than other state-of-the-art approaches, which can best fit in the big data scenario nowadays.

### References

[1] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces.
In 1997 Conference on Computer Vision and Pattern Recognition (CVPR ’97), pages 1000–1006, 1997. 1

[2] 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. 2, 3, 5, 7

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. Communications of the Acm, 18(9):509–517, 1975. 1

[4] D. Cai. A revisit of hashing algorithms for approximate nearest neighbor search. arXiv:1612.07545, 2016. 1, 5

[5] W. Dong. Kgraph, an open source library for k-nn graph construction and nearest neighbor search. www.kgraph.org, 2014. 1, 2, 5, 7

[6] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In Proceedings of the 20th international Conference on World Wide Web, pages 577–586, 2011. 1

[7] C. Fu and D. Cai. Efanna : An extremely fast approximate nearest neighbor search algorithm based on knn graph. arXiv:1609.07228, 2016. 1, 2, 3, 5

[8] 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. 1

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In Proceedings of the 25th International Conference on Very Large Data Bases, pages 518–529, 1999. 1

[10] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In IJCAI 2011, Proceedings of the International Joint Conference on Artificial Intelligence, volume 22, pages 1312–1317, 2011. 1, 2

[11] 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. 1, 2, 5

[12] T. Liu, C. R. Rosenberg, and H. A. Rowley. Clustering billions of images with large scale nearest neighbor search. In 8th IEEE Workshop on Applications of Computer Vision (WACV 2007), page 28, 2007. 1

[13] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. arXiv:1603.09320, 2016. 2, 3, 4, 5, 8

[14] P. R. Manning, Christopher D. and H. Schtze. Introduction to information retrieval. Cambridge: Cambridge university press, 2008. 1

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

[16] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In Advances in neural information processing systems, pages 1753–1760, 2009. 1