Greedy Filtering: A Scalable Algorithm for K-Nearest Neighbor Graph Construction

Youngki Park[1], Sungchan Park[1], Sang-goo Lee[1], and Woosung Jung[2]

[1] School of Computer Science and Engineering, Seoul National University
`{ypark,baksalchan,sglee}@europa.snu.ac.kr`
[2] Department of Computer Engineering, Chungbuk National University
`wsjung@cbnu.ac.kr`

Abstract. Finding the k-nearest neighbors for every node is one of the most important data mining tasks as a primitive operation in the field of Information Retrieval and Recommender Systems. However, existing approaches to this problem do not perform as well when the number of nodes or dimensions is scaled up. In this paper, we present *greedy filtering*, an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a time complexity of $O(n)$, our algorithm chooses essentially a fixed number of node pairs as candidates for every node. We also present a faster version of greedy filtering based on the use of inverted indices for the node prefixes. We conduct extensive experiments in which we (i) compare our approaches to the state-of-the-art algorithms in seven different types of datasets, and (ii) adopt other algorithms in related fields (similarity join, top-k similarity join and similarity search fields) to solve this problem and evaluate them. The experimental results show that greedy filtering guarantees a high level of accuracy while also being much faster than other algorithms for large amounts of high-dimensional data.

Keywords: k-nearest neighbor graph, similarity join, similarity search.

1 Introduction

Constructing a k-Nearest Neighbor (k-NN) graph is an important data mining task which returns a list of the most similar k nodes for every node [1]. For example, assuming that we constructed a k-NN graph whose nodes represent users, we can quickly recommend items to user u by examining the purchase lists of u's nearest neighbors. Furthermore, if we implement an enterprise search system, we can easily provide an additional feature that finds k documents most similar to recently viewed documents.

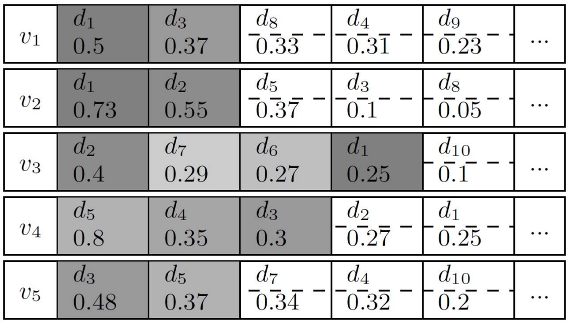
We can calculate the similarities of all possible pairs of k-NN graph nodes by a brute-force search, for a total of $n(n-1)/2$. However, because there are many nodes and dimensions (features) in the general datasets, not only does calculating the similarity between a node pair require a relatively long execution time, but the total execution time will be very large. The inverted index join algorithm [2] is much faster than a brute-force search in sparse datasets. It is one of the fastest algorithms among those producing exact k-NN graphs, but it also requires $O(n^2)$ asymptotic time complexity and its actual execution time grows exponentially.

Another way to construct a k-NN graph is to execute a k-nearest neighbor algorithm such as locality sensitive hashing (LSH) iteratively. LSH algorithms [3, 4, 5, 11] first generate a certain number of signatures for every node. When a query node is given, the LSH compares its signatures to those of the other nodes. Because we have to execute the algorithm for every node, the graph construction time will be long unless one query can be executed in a short time.

As far as we know, NN-Descent [6] is the most efficient approach for constructing k-NN graphs. It randomly selects k-NN lists first before exploiting the heuristic in which a neighbor of a neighbor of a node is also be a neighbor of the node. This dramatically reduces the number of comparisons while retaining a reasonably high level of accuracy. Although the performance is adequate as the number of nodes grows, it does not perform well when the number of dimensions is scaled up.

In this paper, we present greedy filtering, an efficient, scalable algorithm for k-NN graph construction. This finds an approximate k-NN graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a time complexity of $O(n)$, our algorithm selects essentially a fixed number of node pairs as candidates for every node. We also present a faster version of greedy filtering based on the use of inverted indices for the prefixes of nodes. We demonstrate the effectiveness of these algorithms through extensive experiments where we compare various types of algorithms and datasets. More specifically, our contributions are as follows:

- We propose a novel algorithm to construct a k-NN graph. Unlike existing algorithms, the proposed algorithm performs well as the number of nodes or dimensions is scaled up. We also present a faster version of the algorithm based on inverted indices (Section 3).
- We present several ways to construct a k-NN graph based on the top-k similarity join, similarity join, and similarity search algorithms (Section 4.1). Additionally, we show their weaknesses by analyzing their experimental results (Section 4.2).
- We conduct extensive experiments in which we compare our approaches to existing algorithms in seven different types of datasets. The experimental results show that greedy filtering guarantees a high level of accuracy while also being much faster than the other algorithms for large amounts of high dimensional data. We also analyze the properties of the algorithms with the TF-IDF weighting scheme (Section 4.2).



v_1	d_1 0.5	d_3 0.37	d_8 0.33	d_4 0.31	d_9 0.23	...
v_2	d_1 0.73	d_2 0.55	d_5 0.37	d_3 0.1	d_8 0.05	...
v_3	d_2 0.4	d_7 0.29	d_6 0.27	d_1 0.25	d_{10} 0.1	...
v_4	d_5 0.8	d_4 0.35	d_3 0.3	d_2 0.27	d_1 0.25	...
v_5	d_3 0.48	d_5 0.37	d_7 0.34	d_4 0.32	d_{10} 0.2	...

Fig. 1. Example of Greedy Filtering [1]: The prefixes of vectors are colored. We assume that the hidden elements (as described by the ellipse) have a value of 0 and $k = 2$.

2 Preliminaries

2.1 Problem Formulation

Let G be a graph with n nodes and no edges, V be the set of nodes of the graph, and D be the set of dimensions of the nodes. Each node $v \in V$ is represented by a vector, which is an ordered set of elements $e_1, e_2, ..., e_{|v|-1}, e_{|v|}$ such that each has a pair consisting of a dimension and a value, $\langle d_i, r_j \rangle$, where $d \in D$ and $0 \leq r_j \in \mathbb{R} \leq 1$. The values are normalized by L_2-norm such that the following equation holds:

$$\sum_{\langle d_i \in D, r_j \in \mathbb{R} \rangle \in V} r_j^2 = 1. \tag{1}$$

Definition 1 (k-NN Graph Construction): Given a set of vectors V, the k-NN graph construction returns for each vector $x \in V$, $argmax^k_{y \in V \wedge x \neq y}(sim(x, y))$, where $argmax^k$ returns the k arguments that give the highest values.

We use the cosine similarity as the similarity measure for k-NN graph construction. The cosine similarity is defined as follows:

$$sim(v_i \in V, v_j \in V) = \frac{v_i \cdot v_j}{\| v_i \| \| v_j \|} = v_i \cdot v_j. \tag{2}$$

Example 1: In Figure 1, if we assume that the hidden elements (as described by the ellipse) have a value of 0 and $k = 2$, the k-nearest neighbors of v_1 are v_2 and v_4, because $sim(v_1, v_2)$, $sim(v_1, v_3)$, $sim(v_1, v_4)$, and $sim(v_1, v_5)$ are 0.42, 0.13, 0.34, 0.28, respectively. The k-NN graph is obtained by finding k-nearest neighbors for every vector: $\{v_2, v_4\}, \{v_4, v_1\}, \{v_2, v_4\}, \{v_2, v_1\}, \{v_1, v_4\}$.

2.2 Related Work

The k-NN graph construction is closely related to other fields, such as the *similarity join*, *top-k similarity join*, and *similarity search* fields. First, we introduce the similarity join problem as follows:

Definition 2 (Similarity Join): Given a set of vectors V and a similarity threshold ϵ, a similarity join algorithm returns all possible pairs $\langle v_i \in V, v_j \in V \rangle$ such that $sim(v_i, v_j) \geq \epsilon$.

The inverted index join algorithm [2] for similarity join builds inverted indices for all dimensions and then exploits them to calculate the similarities. While it performs much faster than the brute-force search algorithm for sparse datasets, it still has to calculate all of the similarities between the vectors. On the other hand, prefix filtering techniques [2, 7, 8] effectively reduce the search space. They sort the elements of all vectors by their dimensions and set the prefixes such that the similarity between two vectors is below a threshold when their prefixes do not have a common dimension. As a result, we can easily prune many vector pairs by only looking at their prefixes.

The top-k similarity join is identical to the similarity join with regards to finding the most similar pairs. The difference is that it is based on a parameter k instead of ϵ. The top-k similarity join is defined as follows:

Definition 3 (Top-k Similarity Join): Given a set of vectors V and a parameter k, a top-k similarity join algorithm returns $argmax^k_{\langle x \in V, y \in V \rangle \wedge x \neq y} \left(sim(x, y) \right)$, where $argmax^k$ returns the k arguments that give the highest values.

The most common strategy is to calculate the similarities of the most *probable* vector pairs first and then to iterate this step until a stop condition occurs. For example, Kim *et al.* [9] estimates a similarity value ϵ corresponding to the parameter k, selects the most probable candidates, and continues to select candidates until it can be guaranteed that all vector pairs excluding those that were already selected as the candidates have similarity values of less than ϵ. Similarly, Xiao *et al.* [10] stops its iteration when it can be guaranteed that the similarity value of the next probable vector pair is not greater than that of any candidate that has been selected.

Lastly, we define the similarity search problem as follows. This is usually referred to as the approximate k-Nearest Neighbor Search problem (k-NNS) [3].

Definition 4 (Similarity Search): Given a set of vectors V, a parameter k and a query vector $x \in V$, the similarity search returns $argmax^k_{y \in V \wedge x \neq y} \left(sim(x, y) \right)$, where $argmax^k$ returns the k arguments that give the highest values.

The Locality-Sensitive Hashing (LSH) scheme is one of the most common approaches for similarity search. It initially generates for every node a certain number m of signatures, $s_1, s_2, ..., s_{m-1}, s_m$. When a query vector is given, the LSH compares its signatures to those of the other vectors. Because the degree of signature match between two vectors indicates the similarity between them, we can find the k-nearest neighbors based on the results of the matches. The method used for generating signatures mainly depends on the target similarity measure. For example, Broder *et al.* [11] represents a *shingle* vector-based approach for the jaccard similarity measure; while Charikar *et al.* [5] presents a random hyperplane-based approach for cosine similarity.

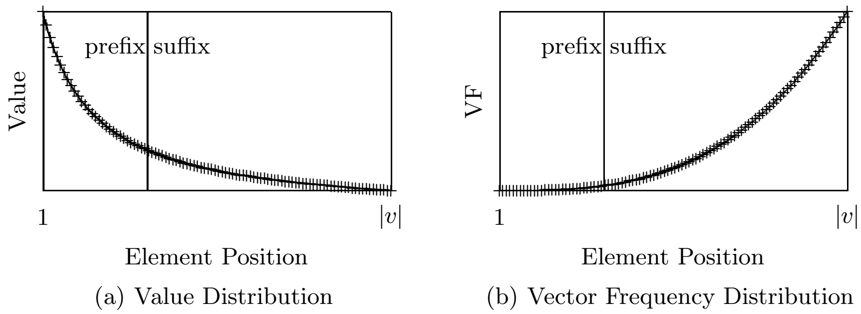


Fig. 2. Typical Distributions After Performing Our Pre-Processing Steps

Note that the problem definitions in related work are analogous to that of k-NN graph construction such that the abovementioned solutions can also be applied to constructing k-NN graphs. For example, if we know all of the similarities between vectors by the inverted index join algorithm, we can obtain the k-NN graph by taking the most similar k vectors for each vector and throwing the rest away. However, these types of approaches do not perform well as the number of nodes or dimensions is scaled up. In Section 4, we discuss these issues in detail, present several ways to construct a k-NN graph based on the algorithms of these fields, and analyze their performance results.

A few approaches have been presented for the purpose of the k-NN graph construction. Examples include kNN-overlap [12], kNN-glue [12], and NN-Descent [6]. The process of the first two algorithms is as follows: (i) all of the vectors are divided into subsets, (ii) k-NN graphs for the subsets are recursively computed, and (iii) the results are formed into a final k-NN graph. Then a heuristic in which a neighbor of a neighbor of a vector is also the neighbor of the vector is additionally applied to improve the accuracy further. On the other hand, the NN-Descent algorithm exploits only the above heuristic, but in a more sophisticated way by adopting four additional optimization techniques: the local join, incremental search, sampling, and early termination techniques. The experimental results show that although NN-Descent outperforms the other algorithms in terms of accuracy and execution time, it does not perform well as the number of dimensions scales up [6]. In the rest of this paper, we present novel approaches to cope with this problem and compare the approaches to existing algorithms.

3 Constructing a *k*-Nearest Neighbor Graph

3.1 Greedy Filtering

Before presenting our algorithms, we introduce several distributions of datasets, as follows: Figure 2(a) shows the value of each element of a vector $v \in V$, where the value of the i^{th} element is larger than that of $(i+1)^{th}$ element. Figure 2(b) shows the *vector frequency* of the i^{th} element of a vector $v \in V$, where the vector frequency of the i^{th} element is smaller than that of $(i+1)^{th}$ element. Let $dim(e)$ be the dimension of element e. Then the vector frequency of the element e is defined as the number of vectors that have the element of dimension $dim(e)$.

An interesting finding is that regardless of the dataset used, the dataset often follows distributions similar to those shown in Figure 2(a) and Figure 2(b) by performing some of the most common pre-processing steps. If we sort the elements of each vector in descending order according to their values, their value distributions will be similar to the distribution shown in Figure 2(a). Note that this pre-processing step does not change the similarity values between vectors. Furthermore, if we weigh the value of each element according to a scheme that adds weights to the values corresponding to sparse dimensions, such as IDF, TF-IDF, or BM25, then the vector frequency distributions will be similar to the distribution shown in Figure 2(b). These weighting schemes are widely used in Information Retrieval and Recommender Systems along with popular similarity measures [13].

Let v' and v'' be the prefix and suffix of vector $v \in V$, respectively. The prefix v' consists of the first n elements and the suffix v'' consists of the last m elements such that $|v| = n + m$. Then $sim(v_i \in V, v_j \in V) = sim(v_i', v_j') + sim(v_i', v_j'') + sim(v_i'', v_j') + sim(v_i'', v_j'')$. Our intuition is as follows: assuming that the elements of each vector follows the distributions shown in Figure 2 and that the prefix and suffix of each vector is determined beforehand, $sim(v_i, v_j)$ would not have a high value when $sim(v_i', v_j') = 0$ because, first, $sim(v_i', v_j'')$ would not have a high value; the vector frequencies of the elements in v_i' are so small that there is a low probability that v_i' and v_j'' have a common dimension. Although there are some common dimensions in their elements, the values of the elements in v_j'' are so small that they do not increase the similarity value significantly. For a similar reason, $sim(v_i'', v_j')$ and $sim(v_i'', v_j'')$ would not have a high value: The elements of high values have low vector frequencies and the elements of high vector frequencies have low values. When $sim(v_i', v_j') \neq 0$, on the other hand, $sim(v_i, v_j)$ would have a relatively high value because v_i' and v_j' have the highest values.

If we generalize this intuition, we can assert that two vectors are not one of the k-nearest neighbors of each other if their prefixes do not have a single common dimension. That is to say, we would obtain an approximate k-NN graph by calculating the similarities between vectors that have at least one common dimension in their prefixes. Note that the vector frequencies of the prefixes are so small that they usually do not have a common dimension. Thus we can prune many vector pairs without computing the actual similarities. Because this approach initially checks whether the dimensions of large values match, we call it *greedy filtering.*

Definition 5 (Match): Let v_i and v_j be the vectors in V, and let e_i and e_j be any of the elements of v_i and v_j, respectively. We hold that e_i and e_j match if $dim(e_i) = dim(e_j)$. We also say that v_i and v_j match if any $e_i \in v_i$ and $e_j \in v_j$ match.

Definition 6 (Greedy Filtering): Greedy filtering returns for each vector $x \in V$, $argmax^k_{y \in V \wedge x \neq y \wedge match(x,y)} (sim(x, y))$, where $argmax^k$ returns the k arguments that give the highest values, and $match(x, y)$ is true if and only if x and y match.

Example 2: Figure 1 shows an example of greedy filtering, where the prefixes are colored. If we assume that the hidden elements (described by ellipse) have a value of 0 and $k = 2$, greedy filtering calculates the similarities of $\langle v_1, v_2 \rangle$, $\langle v_1, v_3 \rangle$, $\langle v_1, v_4 \rangle$, $\langle v_1, v_5 \rangle$, $\langle v_2, v_3 \rangle$, $\langle v_4, v_5 \rangle$, and $\langle v_1, v_4 \rangle$, filters out $\langle v_2, v_4 \rangle$, $\langle v_2, v_5 \rangle$, $\langle v_3, v_4 \rangle$ and $\langle v_3, v_5 \rangle$, and returns k-nearest neighbors for every vector: $\{v_2, v_4\}$, $\{v_3, v_1\}$, $\{v_2, v_1\}$, $\{v_5, v_1\}$, $\{v_4, v_1\}$.

Note that the result of Example 2 is slightly different from that of Example 1, because greedy filtering is an approximate algorithm. If the dataset follows the distributions similar to those of Figure 2, the algorithm will be more accurate. In Section 4, we will justify our intuition in more detail.

3.2 Prefix Selection Scheme

If we set the prefix such that $|v'_i| = |v_i|, \forall v_i \in V$, then greedy filtering generates the exact k-NN graph though its execution time will be very long. On the other hand, if we set the prefix such that $|v'_i| = 0, \forall v_i \in V$, then greedy filtering returns a graph with no edges while the algorithm will terminate immediately. Note that the elapsed time of greedy filtering and the quality of the constructed graph depend on the prefix selection scheme. In general, there is a tradeoff between time and quality.

Assume that greedy filtering can find the approximate k-nearest neighbors for $v_i \in V$ if the number of matched vectors of v_i is equal to or greater than a small value μ. Then if for each vector v_i we find v'_i such that $|v'_i|$ is minimized and the number of matched vectors of v_i is at least μ, then we can expect a rapid execution of the algorithm and a graph of good quality.

Algorithm 1 describes our prefix selection scheme, where $e^j_{v_i}$ denotes the j^{th} element of vector v_i and $dim(e)$ denotes the dimension of element e. In line 2, we initially prepare an empty list for each dimension. Because one list $L[d_i]$ contains vectors that have the dimension of d_i in their prefixes, if any list has the two different vectors v_i and v_j, then greedy filtering will calculate the similarity between them. In lines 7-8, we insert the vectors in R into the lists, meaning that we increase the prefixes of the vectors in R by 1. In lines 10-13, we estimate the number of matched vectors, denoted by M, for each $v_i \in R$. In lines 14-16, we check the stop conditions for each vector and determine which vectors will increase their prefixes again.

Note that Algorithm 1 sacrifices two factors for the performance and ease of implementation. First, it allows the duplicate execution of the brute-force search (lines 19-20 of Algorithm 1 and lines 3-5 of Algorithm 2). If the two vectors v'_i and v'_j have the d number of dimensions that match, we will calculate the d number of calculations of $sim(v_i, v_j)$. Although we can avoid these redundant computations by exploiting a hash table, this is not good for scalability in general. Second, we overestimate the value μ for a similar reason: if the two vectors v'_i and v'_j have

Algorithm 1. Greedy-Filtering (V, μ)

```
Input: a set of vectors $V$, a parameter $\mu$
Output: $k$-NN queues $Q$
begin
    $L[d_i] \longleftarrow \phi, \forall d_i \in D$ /* candidates */
    $C \longleftarrow 1$ /* an iteration counter */
    $R \longleftarrow V$ /* vectors to be processed */
    repeat
        /* find candidates */
        for $v_i \in R$ do
            add $v_i$ to $L[dim(e^C_{v_i})]$
        /* check stop conditions */
        for $v_i \in R$ do
            $M \longleftarrow 0$
            for $j \leftarrow 1$ to $C$ do
                $M \longleftarrow M + |L[dim(e^j_{v_i})]|$
            if $M \geq \mu$ or $C \geq |v_i|$ then
                $P[v_i] \longleftarrow C$
                remove $v_i$ from $R$
        $C \longleftarrow C + 1$
    until $|R| > 0$
    if default algorithm then
        return Brute-force-search(L)
    else
        return Inverted-index-join(V, P)
```

d number of dimensions that match, then M increases by d instead of 1. Also, we calculate the value of M only once per iteration; this makes M slightly larger.

Example 3: Figure 1 shows the result of our prefix selection scheme when $\mu = 2$. Let $M(v)$ be the value M of the vector v. Initially, the prefix size of each vector is 1: $M(v_1) = M(v_2) = 1$ and $M(v_3) = M(v_4) = M(v_5) = 0$, because only $\langle v_1, v_2 \rangle$ match. As the next step, we increase the prefix sizes of all vectors by 1, as $M(v_i) < \mu, \forall v_i \in V$. Then $\langle v_1, v_5 \rangle$, $\langle v_2, v_3 \rangle$ and $\langle v_4, v_5 \rangle$ match. At this point, $M(v_1) = M(v_2) = M(v_5) = 2$ and $M(v_3) = M(v_4) = 1$. Thus we increase the prefix sizes of v_3 and v_4. As we continue until the stop condition is satisfied, our prefix selection scheme selects the colored elements shown in Figure 1.

Our prefix selection scheme has $O(|V||D|^2)$ time complexity, and the brute-force search has to compare each vector v to M number of other vectors. However, our preliminary results show that the prefix sizes are so small that we can regard D as a constant. Furthermore, we set the variable M close to μ; empirically, M is not twice as large as μ. Assuming that D is a constant and $M = 2\mu$, the total complexity of greedy filtering is $O(|V| + 2\mu|V|) = O(|V|)$.

Algorithm 2. Brute-force-search (L)

```
  Input: lists for dimensions $L$
  Output: $k$-NN queues $Q$
1 begin
2     $Q[v_i] \longleftarrow \phi, \forall v_i \in V$ /* empty queues */
3     for $d_i \in D$ do
4         compare all vector pairs $\langle v_x, v_y \rangle$ in $L[d_i]$
5         update the priority queues, $Q[v_x]$ and $Q[v_y]$
6     return $Q$
```

3.3 Optimization

Our algorithm uses a brute-force search a constant number of times for each vector. Because the execution times of the brute-force search highly dependent on the sizes of the vectors, it will take a relatively long time when a dataset contains very large vectors. For instance, experimental results show that the execution time of datasets whose vector sizes are relatively large, such as TREC 4-gram, is longer than that of other datasets.

We present one variation of greedy filtering, called *fast greedy filtering.* The main idea of this approach is that if $sim(v_i', v_j')$ is relatively high, then $sim(v_i, v_j)$ will also be relatively high. Then we can formulate an approximate k-NN graph by calculating the similarities between prefixes. Algorithm 1 and Algorithm 3 describe the process of this algorithm in detail: $e_{v_i}^j$ denotes the j^{th} element of vector v_i, and $dim(e)$ and $value(e)$ denote the dimension and value of element e, respectively. In Algorithm 1, we set the prefix of each vector according to the abovementioned prefix selection scheme and invoke Algorithm 3. Then in lines 6-12 of Algorithm 3, we calculate the similarities between the current vector $v_i \in V$ and the other vectors already indexed and update the k-nearest neighbors of v_i and the indexed vectors. In lines 14-15, we put the current vector v_i into the inverted indices. Unlike greedy filtering, the execution time of fast greedy filtering is highly dependent on the number of dimensions and the vector frequencies of the datasets rather than the vector sizes.

Definition 7 (Fast Greedy Filtering): For each vector $x \in V$, fast greedy filtering returns $argmax^k_{y \in V \wedge x \neq y \wedge match(x,y)} (sim(x', y'))$, where $argmax^k$ returns the k arguments that give the highest values, and where $match(x, y)$ is true if and only if x and y match.

Example 4: If we apply fast greedy filtering to the example in Figure 1, the algorithm returns slightly different results: $\{v_2, v_5\}$, $\{v_3, v_2\}$, $\{v_2, v_1\}$, $\{v_5, v_1\}$, $\{v_4, v_5\}$ when $k = 2$.

Algorithm 3. Inverted-index-join (L, P)

```
Input: a set of vectors $V$, prefix sizes $P$
Output: $k$-NN queues $Q$
1  begin
2      $Q[v_i] \longleftarrow \phi, \forall v_i \in V$ /* empty queues */
3      $I[d_i] \longleftarrow \phi, \forall d_i \in D$ /* empty indices */
4      for $v_i \in V$ do
5          /* verification phase */
6          $C[v_j] \longleftarrow 0, \forall v_j \in V$ /* $sim(v_i, v_j) = 0$ */
7          for $l \leftarrow 1$ to $P[v_i]$ do
8              for $\langle v_j, r_j \rangle \in I[dim(e^l_{v_i})]$ do
9                  $C[v_j] \longleftarrow C[v_j] + r_j * value(e^l_{v_i})$
10         for $v_j \in V$ do
11             if $C[v_j] > 0$ then
12                 update the queues, $Q[v_x]$ and $Q[v_y]$
13         /* indexing phase */
14         for $l \leftarrow 1$ to $P[v_i]$ do
15             add $\langle v_i, value(e^l_{v_i}) \rangle$ to $I[dim(e^l_{v_i})]$
16     return $Q$
```

4 Experiments

4.1 Experimental Setup

Algorithms. We considered eight types of algorithms for a comparison. Three algorithms among them adopt the similarity join (abbreviated by SIM) [2], the top-k similarity join (TOP) [10], and similarity search (LSH) [5] approaches. Two algorithms among them are NN-Descent (DE1) and Fast NN-Descent (DE2) [6], originally developed for the purpose of constructing k-NN graphs. The other two algorithms are greedy filtering (GF1) and fast greedy filtering (GF2) algorithms as proposed in this paper. Finally, we use the inverted index join (IDX) [2], which calculates all similarities with inverted indices, as a baseline algorithm. In all experiments, we set the number of neighbors to 10 (k=10).

We adopted the similarity join algorithm for k-NN graph construction. First, we implement the vector similarity join algorithm, *MM-join* [2], which outperforms the All-pairs algorithm [7] in various datasets. Then, we iterate the execution of the algorithm while decreasing the threshold ϵ by δ until either at least s% of vectors find k-nearest neighbors or until the elapsed time is higher than that of inverted index join. We used the following values in the experiments: $\epsilon = 1.00$ (the initial value), $\delta = 0.05$ and $s = 30$.

Adapting the top-k similarity join algorithm [10] for the k-NN graph construction process is along the same lines as that of the similarity join algorithm, except (1) we increase the parameter k at each iteration instead of decreasing δ,

Table 1. Datasets and Statistics

Dataset Statistics	$\|V\|$	$\|D\|$	Avg. Size	Avg. VF
DBLP	**250,000**	163,841	5.14	7.85
TREC	125,000	484,733	79.83	20.59
Last.fm	125,000	56,362	4.78	10.60
DBLP 4-gram	150,000	279,380	27.97	15.02
TREC 4-gram	50,000	**731,199**	**509.20**	34.82
Last.fm 4-gram	100,000	194,519	20.77	10.68
MovieLens	60,000	10,653	141.23	**795.44**

and (2) because the top-k similarity join algorithm uses *sets* as data structures, we need to transform the data structures into vectors and set new upper bounds for the suffixes of vectors using the prefix filtering and length filtering conditions. We set $s = 70$ for the top-k similarity join algorithm.

We also adopted the similarity search algorithm for k-NN graph construction by executing the algorithm N times. We used random hyperplane-based locality sensitive hashing for cosine similarity [5]. We cannot adopt other LSH algorithms, such as those in Broder et al. [11] or Gionis et al. [3], as they were originally developed for other similarity measures. We set the number of signatures for each vector to 100.

Datasets. We considered seven types of datasets for a comparison. There are two document datasets (DBLP[1] and TREC[2]), one text dataset that consists of music metadata (Last.fm[3]), three artificial text datasets (DBLP 4-gram, TREC 4-gram and Last.fm 4-gram), and one log dataset that consists of the movie ratings of users (MovieLens[4]). Note DBLP 4-gram, TREC 4-gram, and Last.fm 4-gram are derived from DBLP, TREC and Last.fm, respectively. We remove whitespace characters in the original vectors and extracted the 4-gram sequences from them. Table 1 shows their major statistics, where $|V|$ denotes the number of vectors and $|D|$ is the number of dimensions, *Avg. Size* denotes the average size of all vectors, and *Avg. VF* is defined as the average vector frequencies of all dimensions.

Evaluation Measures. We use the execution time and the scan rate as the measures of performance. The execution time is measured in seconds; it does not include the data preprocessing time, which accounts for only a minor portion. The scan rate is defined as follows:

$$\mathit{Scan\ Rate} = \frac{\#\ \mathit{similarity\ calculations}}{|V|\,(|V|-1)/2} \tag{3}$$

The *similarity calculation* expresses the exact calculation of the similarity between a pair. Thus, the brute-force search and the inverted index join always

[1] `http://dblp.uni-trier.de/xml/`
[2] `http://trec.nist.gov/data/t9_filtering.html/`
[3] `http://www.last.fm/`
[4] `http://grouplens.org/datasets/movielens/`

have a scan rate of 1, as they calculate all of the similarities between vectors. On the other hand, fast greedy filtering has a scan rate of 0 because this algorithm only estimates the degrees of similarity.

We use the level of accuracy as the measure of quality. Assuming that an algorithm returns k neighbors for each vector, the accuracy of the algorithm is defined as follows:

$$Accuracy = \frac{\#\ correct\ k\text{-}nearest\ neighbors}{k\,|V|} \tag{4}$$

Weighting Schemes. The value of each element can be weighted by the popular weighting scheme, such as *TF-IDF*. Let v be a vector in V and e_i be an element in v. Then, we define the TF-IDF as follows:

$$tf\text{-}idf(e_i, v) = \left(0.5 + \frac{0.5 * value(e_i)}{max\,\{value(e_j) : e_j \in v\}}\right) * \left(log\frac{|V|}{VF(e_i)}\right), \tag{5}$$

Here, $value(e)$ is the initial value of e. In the text datasets, the initial values are the term frequencies; in the MovieLens dataset, the values are the ratings.

4.2 Performance Comparison

Comparison of All Algorithms. Figure 3 and Table 2 show the execution time, accuracy, and scan rate of all algorithms with a small number of TREC nodes. We do not specify the accuracy and scan rate of inverted index join in Table 2, as its accuracy is always 1 and its scan rate is always 0. By the same token, the scan rates of LSH and GF2 are left blank. We set $\mu = 300$ for our greedy filtering algorithms.

The experimental results show that the greedy filtering approaches (GF1 and GF2) outperform all other approximate algorithms in terms of the execution time, accuracy and scan rate. The second best algorithms behind GF1 and GF2 are the NN-Descent algorithms (DE1 and DE2). However, as already descrbed in work by Dong et al., the accuracy of the algorithms significantly decreases as the number of dimensions scales up. The other algorithms require either a long execution time or return results that are not highly accurate. The top-k similarity join and similarity join algorithms require a considerable amount of time to construct k-NN graphs, and locality sensitive hashing based on random hyperplanes requires many signatures (more than 1,000 signatures in our experimental settings) to ensure a high level of accuracy.

Comparison of All Datasets. Table 3 shows the comparison results of the two outperformers, greedy filtering and NN-Descent, over the seven types of datasets with the TF-IDF weighting scheme. The results of their optimized versions are specified within the parentheses. In this table, we define a new measure, *time*, as the execution time divided by the execution time of inverted index join. We set the parameters μ such that the accuracy of GF1 is at least 90%. The experimental results show that GF1 outperforms the NN-Descent algorithms in all of the datasets except for DBLP and MovieLens. Although the execution time of GF1

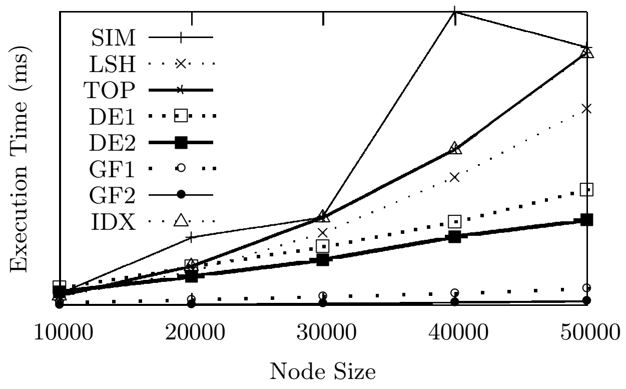


Fig. 3. Execution Time of All Algorithms (TREC)

Table 2. Accuracy and Scan Rate of All Algorithms

Node	Accuracy (TREC)							Scan Rate (TREC)						
	SIM	LSH	TOP	DE1	DE2	GF1	GF2	SIM	LSH	TOP	DE1	DE2	GF1	GF2
10K	0.00	0.01	0.68	0.54	0.43	**0.96**	0.65	0.05	-	0.06	0.27	0.19	**0.05**	-
20K	0.00	0.01	0.76	0.50	0.38	**0.95**	0.63	0.07	-	0.08	0.15	0.11	**0.03**	-
30K	0.00	0.01	0.76	0.48	0.36	**0.94**	0.62	0.05	-	0.08	0.10	0.08	**0.03**	-
40K	0.00	0.01	0.78	0.47	0.34	**0.93**	0.61	0.08	-	0.08	0.08	0.06	**0.02**	-
50K	0.00	0.01	0.79	0.46	0.33	**0.93**	0.59	0.05	-	0.08	0.07	0.05	**0.02**	-

is slower than the times required by the the NN-Descent algorithms for the two datasets, its accuracy is much higher.

Note that while fast greedy filtering exploits inverted index join instead of brute-force searches, it is not always faster than greedy filtering. Fast greedy filtering can be more effective in a dataset for which the vector sizes are relatively large and the number of dimensions and the vector frequencies are relatively small. For example, fast greedy filtering outperforms the other algorithms in the TREC 4-gram datasets, which have the largest average size.

Performance Analysis. Recall that before executing the greedy filtering algorithm, we utilize some of the most common pre-processing steps, as described in Section 3. First, we weigh the value of each element according to a weighting scheme, and then we sort the elements of each vector in descending order according to their values. Figure 4 shows the distributions of all of our datasets after performing these pre-processing steps. Note that the distributions after pre-processing are similar to those in Figure 2. Note also that Figure 4 and the experimental results are in accord with our intuition as presented in Section 3: For example, the distributions of DBLP 4-gram, Last.fm, and TREC in Figure 4 are very similar to those shown in Figure 2; moreover, the experimental results in Table 3 show that their execution time is better than those of the other

Table 3. Comparison of All Datasets

Datasets (TF-IDF)	DE1 (DE2)			GF1 (GF2)		
	Time	Accuracy	Scan Rate	Time	Accuracy	Scan Rate
DBLP	0.015 **(0.013)**	0.14 (0.11)	0.005 **(0.004)**	0.242 (0.076)	**0.98** (0.90)	0.102 (-)
TREC	0.190 (0.140)	0.43 (0.27)	0.030 (0.021)	0.030 **(0.009)**	**0.90** (0.56)	**0.007** (-)
Last.fm	0.322 (0.189)	0.69 (0.68)	0.014 (0.008)	**0.063** (0.149)	**0.98** (0.80)	**0.003** (-)
DBLP 4-gram	0.066 (0.046)	0.52 (0.34)	0.019 (0.011)	**0.004** (0.006)	**0.93** (0.59)	**0.001** (-)
TREC 4-gram	0.228 (0.163)	0.60 (0.42)	0.066 (0.047)	0.106 **(0.003)**	**0.90** (0.48)	**0.035** (-)
Last.fm 4-gram	1.207 (0.800)	0.65 (0.65)	0.013 (0.008)	**0.139** (0.204)	**0.90** (0.59)	**0.001** (-)
MovieLens	0.244 (0.161)	0.55 (0.38)	0.046 (0.028)	0.302 **(0.013)**	**0.90** (0.19)	**0.073** (-)

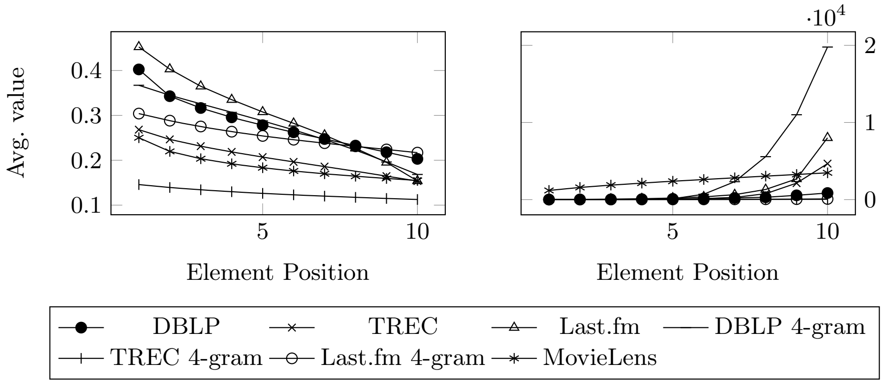


Fig. 4. Distributions of All Datasets

datasets. As another example, the distributions of MovieLens with TF-IDF are relatively less similar to those shown in Figure 2 in that the element position does not greatly affect the vector frequency. Thus, their performance is slightly worse than the performance levels of the other datasets.

5 Conclusion

In this paper, we present *greedy filtering*, an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a linear time complexity, our algorithm chooses essentially a fixed number of node pairs as candidates for every node. We also present *fast greedy filtering* based on the use of inverted indices for the node prefixes. We demonstrate the effectiveness of these algorithms through extensive experiments in which we compare various types of algorithms and datasets.

The limitation of our approaches is that they are specialized for high dimensional sparse datasets, weighting schemes that add weight to the values corresponding to sparse dimensions, and cosine similarity measure. In future work, we would like to extend our approaches to more generalized algorithms.

Acknowledgment. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 20110017480). This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2012R1A1A1043769).

References

[1] Park, Y., Park, S., Lee, S., Jung, W.: Scalable k-nearest neighbor graph construction based on Greedy Filtering. In: WWW 2013, pp. 227–228 (2013)
[2] Lee, D., Park, J., Shim, J., Lee, S.-g.: An efficient similarity join algorithm with cosine similarity predicate. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part II. LNCS, vol. 6262, pp. 422–436. Springer, Heidelberg (2010)
[3] Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB 1999, pp. 518–529 (1999)
[4] Durme, B., Lall, A.: Online generation of locality sensitive hash signatures. In: ACL 2010, pp. 231–235 (2010)
[5] Charikar, M.: Similarity estimation techniques from rounding algorithms. In: STOC 2002, pp. 380–388 (2002)
[6] Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: WWW 2011, pp. 577–586 (2011)
[7] Bayardo, R., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW 2007, pp. 131–140 (2007)
[8] Xiao, C., Wang, W., Lin, X., Yu, J., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Trans. on Database Systems 36(3), 15–41 (2011)
[9] Kim, Y., Shim, K.: Parallel top-k similarity join algorithms using MapReduce. In: ICDE 2012, pp. 510–521 (2012)
[10] Xiao, C., Wang, W.: X Lin, and H. Shang. Top-k set similarity joins. In: ICDE 2009, pp. 916–927 (2009)
[11] Broder, A., Glassman, S., Manasse, M., Zweig, G.: Syntactic clustering of the web. Computer Networks and ISDN Systems 29(8), 1157–1166 (1997)
[12] Chen, J., Fang, H., Saad, Y.: Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection. The Journal of Machine Learning Research 10, 1989–2012 (2009)
[13] Said, A., Jain, B., Albayrak, S.: Analyzing weighting schemes in collaborative filtering: Cold start, post cold start and power users. In: SAC 2012, pp. 2035–2040 (2012)