IEICE TRANS. INF. & SYST., VOL.E97–D, NO.12 DECEMBER 2014

PAPER

Efficient K-Nearest Neighbor Graph Construction Using MapReduce for Large-Scale Data Sets

Tomohiro WARASHINA[†a)], ***Nonmember***, **Kazuo AOYAMA**[††], **Hiroshi SAWADA**[†††], ***Members***, ***and*** **Takashi HATTORI**[††], ***Nonmember***

SUMMARY This paper presents an efficient method using Hadoop MapReduce for constructing a *K*-nearest neighbor graph (*K*-NNG) from a large-scale data set. *K*-NNG has been utilized as a data structure for data analysis techniques in various applications. If we are to apply the techniques to a large-scale data set, it is desirable that we develop an efficient *K*-NNG construction method. We focus on *NN-Descent*, which is a recently proposed method that efficiently constructs an approximate *K*-NNG. *NN-Descent* is implemented on a shared-memory system with OpenMP-based parallelization, and its extension for the Hadoop MapReduce framework is implied for a larger data set such that the shared-memory system is difficult to deal with. However, a simple extension for the Hadoop MapReduce framework is impractical since it requires extremely high system performance because of the high memory consumption and the low data transmission efficiency of MapReduce jobs. The proposed method relaxes the requirement by improving the MapReduce jobs, which employs an appropriate key-value pair format and an efficient sampling strategy. Experiments on large-scale data sets demonstrate that the proposed method both works efficiently and is scalable in terms of a data size, the number of machine nodes, and the graph structural parameter *K*.

key words: *k-nearest neighbor graph, Hadoop MapReduce, distributed computing*

1. Introduction

The goal of *K*-nearest neighbor graph (*K*-NNG) construction is the *efficient* realization of a list of *K* vertices closest to each vertex in a given vertex set based on a defined dissimilarity between a pair of vertices. A graph can be regarded as a general expression of a relationship between objects, where a vertex and an edge correspond to an object and a relationship, respectively. *K*-NNGs have been used in a wide variety of research fields including computer graphics [1], [2], data clustering [3], [4], dimensionality reduction [5], [6], recommender systems [7], and similarity search [8], [9]. In these fields, large-scale high-dimensional data sets are often used in practice. Hence a method that *efficiently* constructs a *K*-NNG from such a data set is necessary to make the developed techniques practical.

Methods that construct an *exact K*-NNG, such as those based on the *triangle inequality* [10], [11], are not suitable for high-dimensional data sets due to their high computational cost. Most approaches represented by divide-and-conquer methods [12]–[14] construct an *approximate K*-NNG from a high-dimensional data set. Recently, the heuristic method *NN-Descent* was proposed; it constructs an approximate *K*-NNG more quickly and accurately than other previously described approximate methods [15].

In [15], *NN-Descent* is implemented on a shared-memory system with OpenMP-based parallelization. The implementation may not be suitable for large-scale data sets due to its limited disk access speed and memory capacity since it serially reads a whole data set from a disk and loads it onto the memory. An easy and convenient way of handling a large-scale data set is to adopt a distributed system based on a shared-nothing architecture. Hadoop MapReduce is one of the most typical ways of implementing such distributed systems. It is both a programming model and a framework for processing large-scale data sets by exploiting the parallelism among computing nodes in the distributed system, and has gained popularity for its simplicity, flexibility and fault tolerance [16].

A MapReduce implementation of *NN-Descent*, which we call *simple extension*, is implied along with the OpenMP-based implementation in [15]. However, it is difficult to use *simple extension* directly because of its high memory consumption and low data transmission efficiency in MapReduce jobs. In this paper, we propose an efficient method to relax the burden of *simple extension*. The contributions of this paper are as follows.

Sophisticated MapReduce jobs
The burden imposed by *simple extension* is the result of the costly feature vectors needed to calculate the dissimilarity between vertices. We rearrange the data structure and the algorithm of *simple extension*, and employ sophisticated MapReduce jobs, which maintain the same output as *simple extension*. The MapReduce jobs use a novel *format of input-outputs (key-value pairs)*, which have none of the redundant feature vectors that are included in *simple extension*.

High performance and unique properties of scalability
In addition to the sophisticated MapReduce jobs, we employed an improved sampling strategy that enables the proposed method to reduce the elapsed time for the approximate *K*-NNG construction and increase the recall of the *K*-NNG to the corresponding exact *K*-NNG. Our experimental results for both real large-scale data

Manuscript received March 26, 2014.
Manuscript revised August 11, 2014.

[†]The author is with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
[††]The authors are with NTT Communication Science Laboratories, NTT Corporation, Kyoto-fu, 619–0237 Japan.
[†††]The author is with NTT Service Evolution Laboratories, NTT Corporation, Yokosuka-shi, 239–0847 Japan.
a) E-mail: t.warashina197@gmail.com

DOI: 10.1587/transinf.2014EDP7108

sets and synthetic data sets demonstrate that (1) the proposed method constructs a K-NNG from a data set of 1×10^7 vertices around five times faster than *simple extension*, and (2) the method is scalable in terms of the data size, the number of machine nodes, and the graph structural parameter K.

When constructing an approximate K-NNG, in particular, our method requires an elapsed time of $O(K^{0.45\sim0.55})$ experimentally while the original *NN-Descent* requires $O(K^{1.5\sim1.8})$. This result means that the proposed method has superiority over *NN-Descent* in accuracy of the obtained approximate K-NNG. This is because the proposed method can handle a larger K value than *NN-Descent*; we can always make an approximate L-NNG ($L > K$) first, and then extract the first K-nearest neighbors to make a more accurate approximate K-NNG than a directly made approximate K-NNG.

The remainder of this paper consists of five sections. Section 2 begins with a brief review of *NN-Descent* and Hadoop MapReduce, and then describes *simple extension* and related problems to clarify our motivation. Section 3 details the proposed method, which consists of four distinct MapReduce jobs with a newly introduced format of key-value pairs. Section 4 describes the experimental performance of the proposed method. Section 5 provides a survey of related work. The final section offers our conclusions.

2. Preliminaries

This section provides an overview of *NN-Descent* and introduces Hadoop MapReduce after describing the notations and definitions used in this paper.

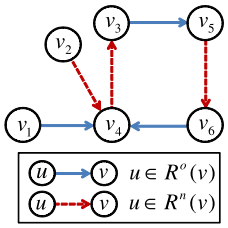
2.1 Notations and Definitions

Given a set of n vertices (objects) $V = \{v_1, v_2, \cdots, v_n\}$, the number of neighbors K and dissimilarity measure $\sigma : V \times V \to \mathbb{R}$, a problem of constructing a K-nearest neighbor graph (K-NNG) is equivalent to finding the K vertices closest to each v_i based on σ. Note that we use identical symbols for an object and a vertex, or a relationship and an edge.

Let $B(v)$ be a set of approximate K-NN vertices of vertex v ($|B(v)| = K$). A *reverse* vertex set and an adjacent vertex set of v are defined by $R(v) = \{u \in V \mid v \in B(u)\}$ and $A(v) = B(v) \cup R(v)$, respectively. A set of neighbor's neighbors of v is expressed by $T(v) = \bigcup_{u\in A(v)} A(u)\backslash\{v\}$. Subsets of $B(v)$, $R(v)$, and $A(v)$ are indicated by $B_s(v)$, $R_s(v)$, and $A_s(v)$, respectively.

Figure 1 helps us understand intuitively the meanings of the above symbols in the graph where each vertex in $v_1, v_2, \cdots, v_6$ is connected to another vertex with one directed edge. Note that the graph is not an exact 1-NN graph in the two-dimensional Euclidean space. When choosing vertex v_4, $B(v_4) = \{v_3\}$, $R(v_4) = \{v_1, v_2, v_6\}$, and $A(v_4) = B(v_4) \cup R(v_4) = \{v_1, v_2, v_3, v_6\}$. Focusing on v_3, $T(v_3) = \bigcup_{u\in A(v_3)} A(u)\backslash\{v_3\} = A(v_4)\backslash\{v_3\} \cup A(v_5)\backslash\{v_3\} = \{v_1, v_2, v_6\}$.

We summarize the notations and definitions we use in this paper in Table 1.



v	$B(v)$	$R(v)$	$R^o(v)$	$R^n(v)$	$T(v)$
v_1	$\{v_4\}$	ϕ	ϕ	ϕ	$\{v_2, v_3, v_6\}$
v_2	$\{v_4\}$	ϕ	ϕ	ϕ	$\{v_1, v_3, v_6\}$
v_3	$\{v_5\}$	$\{v_4\}$	ϕ	$\{v_4\}$	$\{v_1, v_2, v_6\}$
v_4	$\{v_3\}$	$\{v_1, v_2, v_6\}$	$\{v_1, v_6\}$	$\{v_2\}$	$\{v_5\}$
v_5	$\{v_6\}$	$\{v_3\}$	$\{v_3\}$	ϕ	$\{v_4\}$
v_6	$\{v_4\}$	$\{v_5\}$	ϕ	$\{v_5\}$	$\{v_1, v_2, v_3\}$

Fig. 1 Graph consisting of six vertices where each has one directed edge. $R^o(v)$ and $R^n(v)$ are described in Sect. 2.2.

Table 1 Notations and definitions.

Notation	Description and Definition
V	Vertex (object) set
n	The number of vertices in V; $n = \|V\|$
K	The number of neighbors; $K \in \mathbb{N}$, $K \leq n-1$
$\sigma(v,u)$	Dissimilarity from v to u, $v \in V$, $u \in V$ ($\sigma : V \times V \to \mathbb{R}$)
$B(v)$	Approximate K-NN set of v
$R(v)$	Reverse vertex set of v; $R(v) = \{u \in V \mid v \in B(u)\}$
$A(v)$	Adjacent vertex set of v; $A(v) = B(v) \cup R(v)$
$T(v)$	Neighbor's neighbor vertex set of v; $T(v) = \bigcup_{u\in A(v)} A(u)\backslash\{v\}$
$B_s(v)$	Approximate K-NN vertex subset of v; $B_s(v) \subseteq B(v)$
$R_s(v)$	Reverse vertex subset of v; $R_s(v) \subseteq R(v)$
$A_s(v)$	Adjacent vertex subset of v; $A_s(v) \subseteq A(v)$
$T_s(v)$	Neighbor's neighbor vertex subset of v; $T_s(v) \subseteq T(v)$
$B^o(v)$	Set of vertices with *false* flag in $B(v)$
$B^n(v)$	Set of vertices with *true* flag in $B(v)$
$B^n_s(v)$	Set of vertices sampled from $B^n(v)$
$R^o(v)$	Set of vertices with *false* flag in $R(v)$
$R^n(v)$	Set of vertices with *true* flag in $R(v)$
$A^o_s(v)$	Set of vertices with *false* flag in $A_s(v)$
$A^n_s(v)$	Set of vertices with *true* flag in $A_s(v)$
ρ	Sampling rate; $\rho \in (0,1]$
δ	Parameter for *early termination*

2.2 NN-Descent

NN-Descent [15], which constructs an approximate K-NNG from a given vertex set V, can be regarded as an iterative algorithm that minimizes objective function $F(V)$ expressed as

$$F(V) = \sum_{v\in V} \sum_{u\in B(v)} \sigma(v,u) ,$$

where $\sigma(v,u)$ denotes the dissimilarity from v to u.

NN-Descent employs as the initial graph a random graph where each vertex has K directed edges, i.e., the initial $B(v)$ is a set of K vertices randomly sampled from V. At each iteration, $B(v)$ is updated by replacing the vertex in $B(v)$ with one closer to v, which is obtained by calculating a dissimilarity from v to the vertices in a neighbor's neighbor set of v, i.e., $T(v)$. *NN-Descent* terminates just when the number of updates of $B(v)$ falls below a pre-determined value.

Next, four techniques in *NN-Descent* are reviewed, which are closely related to the method proposed in Sect. 3.

Local join in [15] is a procedure based on *only the local information*, where dissimilarities from vertex u to its neighbor's neighbors are calculated and $B(u)$ is updated. Given vertex $v \in V$ and $A_s(v)$, *local join* on $A_s(v)$ is to calculate a

dissimilarity between each pair of $u \in A_s(v)$ and $p \in A_s(v)$ ($u \neq v$), and to update $B(u)$ and $B(p)$. Figure 1 shows the operation of *local join*. Let us consider updating $B(v_3)$ under the condition of $A_s(v) = A(v)$. A naïve approach is to calculate dissimilarities between v_3 and $\forall u \in T(v_3) = \{v_1, v_2, v_6\}$ and update $B(v_3)$. On the other hand, by using *only the local information* $A_s(v_4)$ instead of $T(v_3)$, the *local join* on $A_s(v_4)$ calculates dissimilarities $\sigma(v_3, v_1)$, $\sigma(v_3, v_2)$ and $\sigma(v_3, v_6)$ and updates $B(v_3)$.

Incremental search is a technique that allows us to avoid redundant dissimilarity calculations. A boolean flag is attached to each vertex in $B(v)$ to indicate whether or not the vertex is a candidate for the *local join* on $A_s(v)$. The flag is set to *true* when a vertex is newly inserted into $B(v)$. In the *local join* on $A_s(v)$, only the dissimilarities between two vertices where at least one of their flags is set to *true* are calculated. After the *local join*, the flag of the remaining vertex in $B(v)$ is set to *false*.

To provide simple expressions in the following sections, new symbols are introduced as follows. Vertices with a *false* and a *true* flag are called an *old* and a *new* vertex, respectively. Let $B^o(v)$ and $B^n(v)$ denote subsets of *old* and *new* vertices in $B(v)$, respectively. Let $R^o(v)$ and $R^n(v)$ denote subsets of *old* and *new* vertices in $R(v)$, respectively. Let $A_s^o(v)$ and $A_s^n(v)$ denote subsets of *old* and *new* vertices in $A_s(v)$, respectively. We confirm the above symbols using the graph in Fig. 1. Suppose that the graph is an intermediate graph in *NN-Descent* and vertices in $B^o(v)$ and $B^n(v)$ are connected from v with blue line edges and red dotted edges, respectively. Then $B^o(v_2) = \emptyset$ and $B^n(v_2) = \{v_4\}$. $R^n(v_4) = \{u \in V \mid v_4 \in B^n(u)\} = \{v_2\}$.

Sampling enables *NN-Descent* to avoid a high computational cost in *local join*. In the *sampling*, ρK vertices are randomly sampled from $B^n(v)$ before *local join*, where $\rho \in (0, 1]$ is the sampling rate. In terms of reverse vertices $u \in R(v)$, ρK vertices each are sampled from $R^o(v)$ and the already sampled subset of $R^n(v)$. Then $A_s(v)$, $\forall v \in V$, is generated. The *local join* is executed only for $A_s(v)$.

Early termination counts the number of updates of $B(v)$, $\forall v \in V$ in each iteration, and stops the algorithm when the number of the updates becomes less than δKn, where δ is a given parameter.

2.3 Hadoop MapReduce

Hadoop is open source software, which contains Hadoop Distributed File System (HDFS) and Hadoop MapReduce. HDFS divides a given data set into subsets and stores them at the machine nodes of a cluster in a distributed file system. Hadoop MapReduce is not only a software framework but also a programming model; the model is designed for data-intensive parallel computation in shared-nothing clusters [16], [17].

A job in MapReduce is executed in three phases, a MAP, a SHUFFLE, and a REDUCE phase, in this order. In the MAP and REDUCE phases, respectively, MapReduce executes a MAP and a REDUCE function, which are defined by a user, in each computing node of a cluster. The functions limit their input-output (I/O) formats to key-value pairs expressed by <*key*, *value*>. In contrast, in a SHUFFLE phase, MapReduce automatically sorts key-value pairs that are output from the MAP functions by *key*, and merges the values for each *key*. The resultant key-value pairs are used as REDUCE function inputs.

Although MapReduce is simple and easy to use, its operations are not always optimized for I/O efficiency [16]. MapReduce often suffers from low data transmission efficiency because in a SHUFFLE phase all key-value pairs generated in a MAP phase are transmitted to computing nodes executing a REDUCE function. Therefore, designing MAP and REDUCE functions with high data transmission efficiency is important if we are to realize high performance for the entire MapReduce process.

2.4 Simple Extension of NN-Descent for MapReduce

Simple extension of *NN-Descent* for MapReduce, which is implied in [15], repeats two MapReduce jobs until the termination condition is satisfied. The first MapReduce job generates <v, $\{B(v), A_s(v)\}$> from <v, $B(v)$>, $\forall v \in V$. The MAP function first divides the input $B(v)$ into $B^o(v)$ and $B^n(v)$. Then, the function applies the *sampling* to $B^n(v)$ and generates $B_s^n(v)$, which is a set of *new* vertices sampled from $B^n(v)$. The function finally emits both <v, $B(v)$> and <u, v>, $\forall u \in B^o(v) \cup B_s^n(v)$. The REDUCE function receives both $B(v)$ and a subset of $R(v)$ whose vertices share the same vertex v in the *key*, generates $A_s(v)$, and emits key-value pairs of <v, $\{B(v), A_s(v)\}$>. The second MapReduce job updates $B(v)$ by *local join* on $A_s(v)$, $\forall v \in V$. The MAP function receives <v, $\{B(v), A_s(v)\}$> and emits key-value pairs of <v, $B(v)$>, <u, $A_s(v) \setminus \{u\}$> ($\forall u \in A_s^n$) and <u, $A_s^n(v)$> ($\forall u \in A_s^o(v)$). The vertex u in the *key* and one of the vertices in the *value* of <u, $A_s(v) \setminus \{u\}$> and <u, $A_s^n(v)$> are neighbor's neighbor vertices to each other. The vertices in the *value* of the key-value pairs (vertices in $A_s(v) \setminus \{u\}$ and $A_s^n(v)$) are neighbor's neighbors of the vertex in the *key* (vertex u) via the vertex v. The REDUCE function receives <v, $\{B(v), T_s(v)\}$>, updates $B(v)$, and emits the newly updated key-value pair <v, $B(v)$>. The pseudo-code of *simple extension* is available in Appendix.

Simple extension suffers from the disadvantages of high memory consumption and low data transmission efficiency. These disadvantages originate in the naïve formats of the key-value pairs in MapReduce jobs. A dissimilarity calculation in the *local join* requires an object entity, e.g., a feature vector if an object is represented as a point in a feature space. Hereafter, we use a feature vector instead of an object entity for ease of understanding. For this requirement and the naïve format, each vertex has to retain a feature vector throughout all the processes in the jobs. In particular, when an object is a point in a high-dimensional feature space and a vertex with a high degree, i.e., a hub that appears in a K-NNG, a serious problem occurs regarding memory consumption and data transmission efficiency. If hub v appears in a K-NNG, the REDUCE function in the first job has to deal with a lot of vertices in $R(v)$ to generate $A_s(v)$. This makes the memory consumption too high and

the processing speed lower. The MAP function in the second job emits a lot of key-value pairs, namely the pairs of the neighbor's neighbors $<u, A_s(v)\backslash\{u\}>$ $(\forall u \in A_s^n(v))$ and $<u, A_s^n(v)>$ $(\forall u \in A_s^o(v))$. The number of neighbor's neighbors is too large to transmit through the network, and much larger than that of the adjacent vertices that are emitted in the first job. Their transmission leads to increases in network loads in the SHUFFLE phase. In fact, the heavy network loads in the second job decrease the efficiency of *simple extension* as shown in Sect. 4.4. Thus, *simple extension* of *NN-Descent* for MapReduce is not scalable.

3. Proposed Method

We begin with an overview of the proposed method by providing its main ideas from an algorithmic perspective. Next, we explain how to design our key-value pairs that contribute to a reduction in both memory consumption and network loads. Finally, we detail each of the four MapReduce jobs.

Hereafter, we focus on a high-dimensional metric feature space. In the metric feature space, a dissimilarity is identical to a distance ("dist" for short).

3.1 Overview

Simple extension described in Sect. 2.4 is not scalable. This is because each vertex in key-value pairs keeps its feature vector, which causes high memory consumption and increased network loads. The main idea of the proposed method is to deal with as many vertices as possible without costly feature vectors by using *low cost* information, such as *distance* and *attributes* as described in Sect. 3.2. By introducing a new symbol '*', we distinguish vertex v^* accompanied by its feature vector from vertex v with no feature vector, as shown in Fig. 3. Then v^* has a costly feature vector, while v has no feature vector and can be loaded onto the memory and be transmitted at *low cost*. Similarly, a set in which each vertex has a feature vector is represented by the corresponding symbol accompanied by '*' such as $R(v)^*$. For instance, in the graph in Fig. 1, $R(v_4)^* = \{v_1^*, v_2^*, v_6^*\}$, while $R(v_4) = \{v_1, v_2, v_6\}$, in which no vertex has a feature vector. Since each vertex in $R(v)^*$ has a costly feature vector, $R(v)^*$ is loaded onto the memory and transmitted at much higher cost than $R(v)$. The proposed method uses as few vertices and sets accompanied by '*' such as v^* and $R(v)^*$ as possible by instead using vertices and sets accompanied by no symbol such as v and $R(v)$.

We divide the processing at each iteration in *NN-Descent* into two stages.

Stage 1: Generates $A_s(v)^*$ from $B(v)$, $\forall v \in V$

When creating $A_s(v)^*$, *simple extension* loads almost all the vertices in $R(v)^*$ onto the memory, resulting in high memory consumption. In contrast, we use only the necessary vertices in $R(v)^*$ for creating $A_s(v)^*$ by using a tag that is a symbol indicating whether a vertex is sampled or not, as detailed in Sect. 3.2. This stage is divided into two sub-steps: (1) we first apply a newly introduced sampling technique to $R(v)$ and attach a tag (*attribute* symbol) to the sampled vertices; (2) we then create $A_s(v)^*$ with only $B(v)^*$ and the tagged vertices in $R(v)^*$. Since we use vertices in $R(v)$ with no feature vectors and only the tagged vertices in $R(v)^*$, i.e., not use redundant vertices in $R(v)^*$, we can create $A_s(v)^*$ with a low memory consumption.

Stage 2: Updates $B(v)$ with $A_s(v)^*$, $\forall v \in V$

Simple extension simply emits many large-size pairs of neighbor's neighbor vertices and their feature vectors, $<u^*, A_s(v)^* \backslash \{u^*\}>$ $(\forall u^* \in A_s^n(v)^*)$ and $<u^*, A_s^n(v)^*>$ $(\forall u^* \in A_s^o(v)^*)$, which cause an increase in network load. In contrast, the proposed method emits pairs of neighbor's neighbor vertices, where each vertex of *value* in the key-value pair has *distance*, instead of its feature vector. Then, the proposed method updates $B(v)$ with only the *distance*. Since the proposed method emits neighbor's neighbor vertices without their costly feature vectors, it can update $B(v)$ with high transmission efficiency.

The proposed method consists of four MapReduce jobs that execute the processes of the two stages: *CreateRevVertices*, *CreateKVertices*, *CreateAdjVertices* and *UpdateKVertices*. Figure 2 shows an overview flowchart of the four MapReduce jobs, which are executed in the above order. The first three jobs correspond to Stage 1, i.e., these MapReduce jobs create $A_s(v)^*$, $\forall v \in V$, with a low memory consumption. *CreateRevVertices* and *CreateKVertices* execute sub-step (1) in Stage 1. To tag vertices in $R(v)$, *CreateRevVertices* first creates $R(v)$ from $B(v)$. Then, *CreateKVertices* creates $B(v)^*$. This job also applies our sampling technique to $R(v)$ and tags the sampled vertices. *CreateAdjVertices* executes sub-step (2) in Stage 1, i.e., creates $A_s(v)^*$ with $B(v)^*$ and only the tagged vertices in $R(v)^*$. Note that these three jobs deal with only adjacent vertices, which are much fewer than the neighbor's neighbors. Hence, each of them can be executed at much lower cost than the first

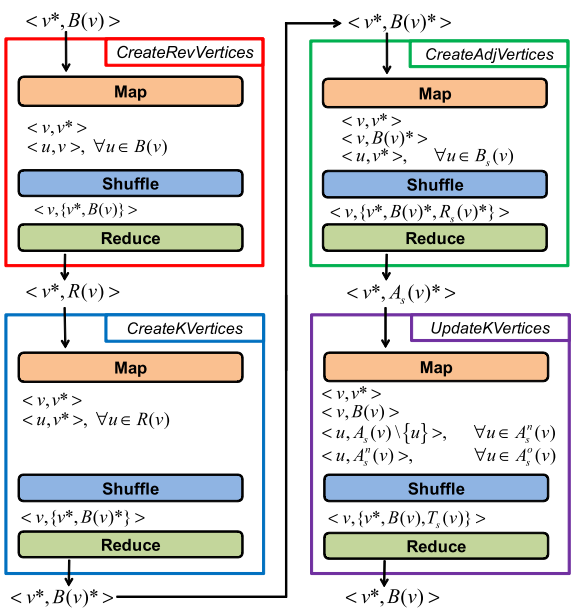


Fig. 2 Overview flowchart of each iteration in the proposed method.

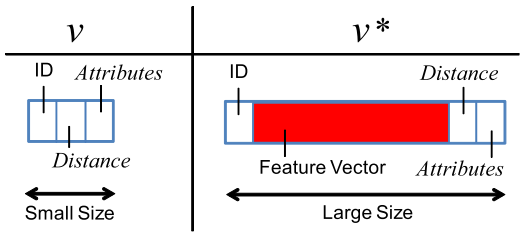


Fig. 3 Composition of v and v^* of *value*-vertex. While v has no feature vector and takes *low cost* for loading onto the memory and for transmission, v^* has feature vector and incurs much higher cost.

job in *simple extension*. *UpdateKvertices*, the fourth job, corresponds to Stage 2, i.e., the job updates $B(v)$, $\forall v \in V$, with only *distance* instead of a feature vector. Owing to the transmission of only the *distance*, the I/O cost is greatly improved. This results in a significant decrease in the elapsed time needed for updating $B(v)$, $\forall v \in V$, as shown in Sect. 4.4. We explain how to design a key-value pair in Sect. 3.2 and each MapReduce job using pseudo-codes in Sect. 3.3.

We change *sampling* in *NN-Descent* described in Sect. 2.2 to improve the elapsed time by reducing the number of iterations. The proposed method samples ρK vertices from each of $R^o(v)$ and $R^n(v)$ while *NN-Descent* first samples ρK vertices from $B^n(v)$ and next samples ρK vertices from each of $R^o(v)$ and $R^n(v)$. In other words, the proposed method can be regarded as a method using the sampling rate of 1.0 ($\rho = 1.0$) for $B(v)$. Our experimental results show that our sampling technique is useful in the Hadoop MapReduce framework in Sect. 4.

3.2 Design of Key-Value Pairs

We replace as many feature vectors as possible with the *distance* and the *attributes* as described later. Let us consider a key-value of <v, u>. We call vertex v and u in the key-value pair the *key*-vertex and the *value*-vertex, respectively. The *key*-vertex v consists of an identification number (ID) and may be accompanied by a feature vector. The *value*-vertex u is a vertex with additional information; it consists of an ID, the *distance* in between v and u, and three *attributes*, and may be accompanied by a feature vector, as shown in Fig. 3. The *distance* calculated in *UpdateKVertices* is kept at the *value*-vertex in the key-value pair for updating $B(v)$. Keeping only the *distance* without the feature vector leads to lower network loads. The *value*-vertex u is a vertex in any of $B(v)$, $R(v)$, and $T_s(v)$ in most settings. The three *attributes*, *a1*, *a2*, and *a3*, are as follows.

a1: holds one of the following three symbols, ‘b’, ‘r’ or ‘t’, and distinguishes the set that u is in: $u \in B(v)$, $u \in R(v)$ or $u \in T_s(v)$;

a2: holds one of the following two symbols, ‘o’ or ‘n’, and distinguishes whether u is an *old* or a *new* vertex on *incremental search*;

a3: holds one of the following two symbols, ‘s’ or ‘u’, and distinguishes whether u is sampled (‘s’) or unsampled (‘u’) in our sampling technique.

Note that the *attributes* are essential information when implementing *simple extension* although this is not mentioned in [15]. In particular, the introduction of the *attribute a3*, i.e., the tag is the key idea for decreasing the memory consumption, as described in Sect. 3.1. A *value*-vertex can have at most one symbol per attribute. The *distance* and the *attributes* are renewed every time MapReduce jobs are performed.

3.3 MapReduce Jobs

3.3.1 Job1: Create Reverse Vertex Sets $R(v)$

CreateRevVertices, namely the first MapReduce job, creates *reverse* vertex sets ($R(v), \forall v \in V$) as shown in Algorithm 1. None of the vertices in $R(v)$ have a feature vector and they are used for our sampling technique in the next job, namely *CreateKVertices*. The MAP function receives <v^*, $B(v)$> as its input and emits two types of key-value pair: (1) <v, v^*> where v^* keeps its feature vector, the maximum value represented by infinity for its distance, and no symbols for its *attributes*; (2) <u, v>, $\forall u \in B(v)$, for creating $R(v)$ in the SHUFFLE phase. The vertex v in the pair <u, v> is a reverse vertex of $R(u)$, since $R(u) = \{v \in V | \ u \in B(v)\}$. By merging all the pairs in which the *key*-vertex is u, we can generate <u, $R(u)$>. In the SHUFFLE phase, v^* and $\forall u \in R(v)$ for each *key*-vertex v are merged, and <v, $\{v^*, R(v)\}$> ($\forall v \in V$) are generated. The REDUCE function generates <v^*, $R(v)$> from the input pair <v, $\{v^*, R(v)\}$> and then emits it.

Algorithm 1 *CreateRevVertices*

```
1: function MAP(v*, B(v))
2:     EMIT(v, v*)
3:     Attach a1 symbol 'r' to v
4:     for all u ∈ B(v) do
5:         Attach u's distance to v
6:         Attach u's a2 symbol to v
7:         EMIT(u, v)
8:     end for
9: end function

10: function REDUCE(v, {v*, R(v)})
11:     EMIT(v*, R(v))
12: end function
```

Figure 4 shows the data flow in *CreateRevVertices* when $R(v_4)$ is created, with reference to Fig. 1. The MAP function receives the input key-value pair whose *key*-vertex is v_4 and emits <v_4, v_4^*>. Furthermore, the function also emits <v_4, v_1>, <v_4, v_2>, and <v_4, v_6>, i.e., <v_4, $R(v_4)$>, which are generated from the key-value pair whose *key*-vertex p ($= v_1, v_2, v_6$) satisfies $v_4 \in B(p)$. In the SHUFFLE phase, all the key-value pairs are merged and then <v_4, $\{v_4^*, v_1, v_2, v_6\}$> = <v_4, $\{v_4^*, R(v_4)\}$> is generated. The REDUCE function receives <v_4, $\{v_4^*, R(v_4)\}$> as its input and then emits <v_4^*, $R(v_4)$>.

3.3.2 Job2: Create Approximate K-NN Vertex Sets $B(v)^*$

CreateKVertices, namely the second MapReduce job, creates approximate K-NN vertex sets ($B(v)^*, \forall v \in V$) as shown in Algorithm 2. Furthermore, the job applies our sampling technique to $R(v)$ and attaches the *a3* symbol ‘s’ to the sampled vertices, i.e., tags them. By using the attribute a3, the next job can create $A_s(v)^*$ with a low memory consumption.

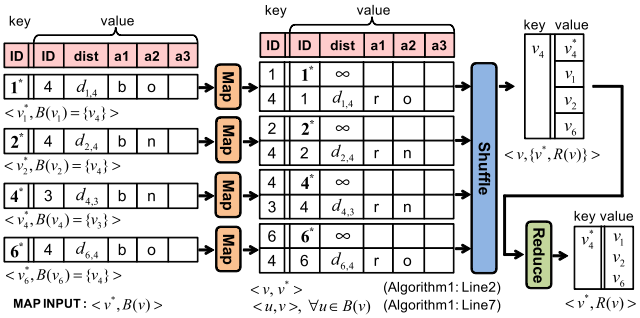


Fig. 4 Data flow in *CreateRevVertices* that creates $R(v_4)$.

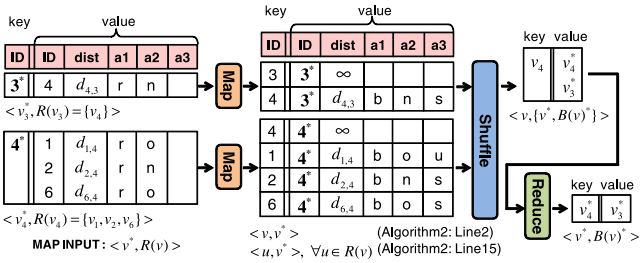


Fig. 5 Data flow in *CreateKVertices* that creates $B(v_4)^*$.

First, the MAP function receives <v^*, $R(v)$> as its input. Next, the function creates $R_s(v)$ by applying the sampling technique to $R(v)$ as the original *NN-Descent* employs the *sampling*. The sampled vertices ($\forall u \in R_s(v)$) are attached the *a3* symbol 's' to, as shown on Line 11 and inserted into $A_s(v)$ in the next job. Finally, the function emits two types of key-value pair: (1) <v, v^*>; (2) <u, v^*>, $\forall u \in R(v)$, to create $B(v)^*$ in the SHUFFLE phase. In the SHUFFLE phase, v^* and $\forall u \in B(v)^*$ for each *key*-vertex v are merged, and <v, {v^*, $B(v)^*$}> ($\forall v \in V$) are generated in the same way as *CreateRevVertices*. The REDUCE function generates <v^*, $B(v)^*$> from the input pair <v, {v^*, $B(v)^*$}> and then emits it.

Algorithm 2 *CreateKVertices*

```
 1: function MAP(v*, R(v))
 2:     EMIT(v, v*)
 3:     R^o(v) ← vertices with a2 symbol 'o' in R(v)
 4:     R^n(v) ← vertices with a2 symbol 'n' in R(v)
 5:     R_s(v) ← SAMPLE (R^o(v), ρK) ∪ SAMPLE (R^n(v), ρK)
 6:     Attach a1 symbol 'b' to v*
 7:     for all u∈R(v) do
 8:         Attach u's distance to v*
 9:         Attach u's a2 symbol to v*
10:         if u∈R_s(v) then
11:             Attach a3 symbol 's' to v*
12:         else
13:             Attach a3 symbol 'u' to v*
14:         end if
15:         EMIT(u, v*)
16:     end for
17: end function

18: function REDUCE(v, {v*, B(v)*})
19:     EMIT(v*, B(v)*)
20: end function
```

Figure 5 shows the data flow in *CreateKVertices* when $B(v_4)^*$ is created with sampling rate $\rho = 1.0$. First, the MAP function receives the input key-value pair whose *key*-vertex is v_4. Next, the function divides $R(v_4)$ into $R^o(v_4) = \{v_1, v_6\}$ and $R^n(v_4) = \{v_2\}$ by using *a2*, and then samples $\rho K (= 1.0 \times 1 = 1)$ vertices from $R^o(v_4)$ and $R^n(v_4)$, respectively. We assume that v_6 is sampled from $R^o(v_4)$ and v_2 from $R^n(v_4)$. Hence, $R_s(v_4) = \{v_2, v_6\}$ is created. Finally, the function emits the key-value pairs. In the SHUFFLE phase, all the key-value pairs are merged and then <v_4, {v_4^*, $B(v_4)^*$}> is generated. The REDUCE function receives <v_4, {v_4^*, $B(v_4)^*$}> as its input and then emits <v_4^*, $B(v_4)^*$>.

3.3.3 Job3: Create Adjacent Vertex sets $A_s(v)^*$

CreateAdjVertices, namely the third MapReduce job, creates adjacent vertex sets ($A_s(v)^*, \forall v \in V$) by loading only the tagged vertices onto the memory, as shown in Algorithm 3. The MAP function receives <v^*, $B(v)^*$> as its input, and then emits three types of a key-value pair: (1) <v, v^*>; (2) <v, $B(v)^*$> for creating $A_s(v)^*$; (3) <u, v^*>, $\forall u \in B_s(v)$, for transmitting only the sampled vertices in $R(v)^*$. The vertex v^* in the pair <u, v^*> is the vertex in $R(v)$ that is sampled in the previous job, namely *CreateKVertices*. In the SHUFFLE phase, v^* and all the vertices in $B(v)^*$ and $R_s(v)^*$ are collected every *key*-vertex v. The REDUCE function generates <v^*, $A_s(v)^*$> from the input pair <v, {v^*, $B(v)^*$, $R_s(v)^*$}> and then emits it.

Algorithm 3 *CreateAdjVertices*

```
 1: function MAP(v*, B(v)*)
 2:     B_s(v) ← vertices with a3 symbol 's' in B(v)*
 3:     EMIT(v, v*)
 4:     EMIT(v, B(v)*)
 5:     Attach a1 symbol 'r' to v*
 6:     for all u∈B_s(v) do
 7:         Attach u's distance to v*
 8:         Attach u's a2 symbol to v*
 9:         EMIT(u, v*)
10:     end for
11: end function

12: function REDUCE(v, {v*, B(v)*, R_s(v)*})
13:     A_s(v)* ← B(v)* ∪ R_s(v)*
14:     Remove a3 symbol 's' of vertices in A_s(v)*
15:     EMIT(v*, A_s(v)*)
16: end function
```

In *simple extension*, the first MAP function emits redundant vertices, and the first REDUCE function creates $A_s(v)^*$ by loading almost all the vertices in $R(v)^*$ onto the memory, which leads to a high memory consumption. In contrast, the MAP function in our method emits only the vertices that are needed for $A_s(v)^*$ by using the *attribute a3* ($B_s(v)$), as shown on Line 9, and the REDUCE function creates $A_s(v)^*$ by loading only the vertices in $A_s(v)^*$ onto the memory. Thus, we create $A_s(v)^*$ with a low memory consumption.

Figure 6 shows the data flow when $A_s(v_4)^*$ is created. The MAP function receives the input key-value pair whose *key*-vertex is v_4 and emits <v_4, v_4^*> and <v_4, v_3^*> = <v_4, $B(v_4)^*$>. Furthermore, the function also emits <v_4, v_2^*>

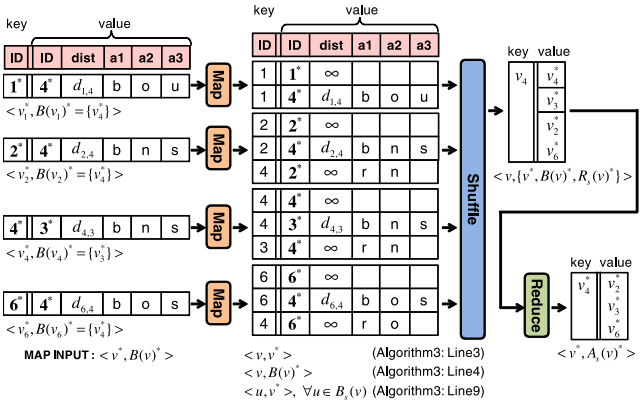


Fig. 6 Data flow in *CreateAdjVertices* that creates $A_s(v_4)^*$.

and <v_4, v_6^*>, i.e. <v_4, $R_s(v_4)^*$>, which are generated from the key-value pair whose *key*-vertex p (= v_2, v_6) satisfies $v_4 \in B_s(p)^*$. The vertex p is the sampled vertex in the MAP function of the previous job, namely *CreateKVertices* when the pair is processed whose *key*-vertex is v_4. In the SHUFFLE phase, all the key-value pairs are merged and then <v_4, $\{v_4^*, v_3^*, v_2^*, v_6^*\}$> = <$v_4$, $\{v_4^*, B(v_4)^*, R_s(v_4)^*\}$> is generated. Finally, the REDUCE function receives <v_4, $\{v_4^*, B(v_4)^*, R_s(v_4)^*\}$> as its input and then emits <v_4^*, $A_s(v_4)^* = \{v_3^*, v_2^*, v_6^*\}$>.

3.3.4 Job4: Update Approximate K-NN Vertex Sets $B(v)$

UpdateKVertices, namely the final MapReduce job, updates $B(v)$, $\forall v \in V$ by using *local join* with high data transmission efficiency. *Simple extension* described in Sect. 2.4 has an issue, namely that the MAP function in the second MapReduce job emits a lot of key-value pairs and each has a costly feature vector. We design *UpdateKVertices* so that the emitted key-value pair has no redundant feature vector by attaching only *distance* to a *value*-vertex in the key-value pair instead of its feature vector.

Algorithm 4 shows the pseudo-code of *UpdateKVertices*. The MAP function is carefully designed for high performance in terms of the number of distance calculations and emitted data size. First, the function uses the getDistance(u_1^*, u_2^*) function to obtain the distance between u_1 and u_2, where $u_1 \in A_s(v)$, $u_2 \in A_s(v) \backslash \{u_1\}$ and at least one of them is a *new* vertex. If the *distance* is already calculated and cached, the getDistance(u_1^*, u_2^*) function fetches it from the cache and returns it, otherwise the getDistance(u_1^*, u_2^*) function calculates the *distance* and caches it. Second, the MAP function emits the four types of key-value pair: <v, v^*>, <v, $B(v)$>, <u, $A_s(v) \backslash \{u\}$>, $\forall u \in A_s^n(v)$, and <u, $A_s^n(v)$>, $\forall u \in A_s^o(v)$. Vertex $u_i \in A_s(v)$ in <u, $A_s(v) \backslash \{u\}$> is a neighbor's neighbor of $u_j \in A_s(v)$, $u_i \neq u_j$, via vertex v. Before emitting <u, $A_s(v) \backslash \{u\}$> and <u, $A_s^n(v)$>, the function removes the feature vector of the *value*-vertices to reduce the *size* of the key-value pair. Note that the only key-value pair that contains its feature vector is <v, v^*>. In the SHUFFLE phase, v^* and all the vertices in both $B(v)$ and $T_s(v)$ for each *key*-vertex v are merged, and <v, $\{v^*, B(v), Ts(v)\}$> ($\forall v \in V$) are generated. The REDUCE function updates $B(v)$ by using the update(u, $B(v)$) function, $\forall u \in T_s(v)$. The update(u, $B(v)$) function replaces p ($p \in B(v)$) with u if $\sigma(v,u) < \sigma(v,p)$, where p has the largest *distance* value in $B(v)$. After the updates, the function emits the updated <v, $B(v)$>.

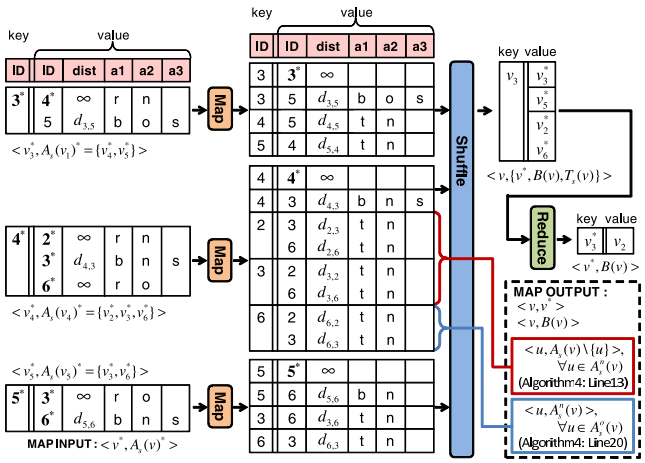


Fig. 7 Data flow in *UpdateKVertices* that updates $B(v_3)$.

Algorithm 4 *UpdateKVerteices*

```
1: function MAP(v*, A_s(v)*)
2:     B(v) ← vertices with a1 symbol 'b' in A_s(v)*
3:     A_s^o(v)* ← vertices with a2 symbol 'o' in A_s(v)*
4:     A_s^n(v)* ← vertices with a2 symbol 'n' in A_s(v)*
5:     Attach a1 symbol 't' to all vertices in A_s^o(v)* and A_s^n(v)*
6:     EMIT(v, v*)
7:     EMIT(v, B(v))
8:     for all u_1* ∈ A_s^n(v)* do
9:         for all u_2* ∈ A_s(v)*\{u_1*} do
10:            l ← getDistance(u_1*, u_2*)
11:            Attach l to u_2*
12:        end for
13:        EMIT(u_1, A_s(v)\{u_1})
14:    end for
15:    for all u_1* ∈ A_s^o(v)* do
16:        for all u_2* ∈ A_s^n(v)* do
17:            l ← getDistance(u_1*, u_2*)
18:            Attach l to u_2*
19:        end for
20:        EMIT(u_1, A_s^n(v))
21:    end for
22: end function

23: function REDUCE(v, {v*, B(v), T_s(v)})
24:     Attach a2 symbol 'o' to all vertices in B(v)
25:     Attach a1 symbol 'b' to all vertices in T_s(v)
26:     Attach a2 symbol 'n' to all vertices in T_s(v)
27:     for all u ∈ T_s(v) do
28:         update(u, B(v))
29:     end for
30:     EMIT(v*, B(v))
31: end function
```

Figure 7 shows the data flow in *UpdateKVertices* when $B(v_3)$ is updated. Suppose that $A_s(v_3) = \{v_4, v_5\}$, $A_s(v_5) = \{v_3, v_6\}$ ($A_s(v_3) = A(v_3)$, $A_s(v_5) = A(v_5)$). The MAP function receives the input pair whose *key*-vertex is v_3, and emits <v_3, v_3^*> and <v_3, v_5> = <v_3, $B(v_3)$>. The function emits <v_3, $\{v_2, v_6\}$> and <v_3, $\{v_6\}$>, i.e. <v_3, $T_s(v_3)$>, which are generated from the key-value pair whose vertex p (= v_4, v_5) satisfies $v_3^* \in A_s(p)^*$. In the SHUFFLE phase, all the key-value

pairs whose *key*-vertex is v_3 are merged, and the key-value pair $<v_3, \{v_3^*, \{v_5\}, \{v_2, v_6\}, \{v_6\}\}> = <v_3, \{v_3^*, B(v_3), T_s(v_3)\}>$ is generated. Since we suppose that $\sigma(v_3, v_2) < \sigma(v_3, v_5)$ and $\sigma(v_3, v_2) < \sigma(v_3, v_6)$, i.e., v_2 is the vertex closest to v_3 in $T_s(v_3)$, the REDUCE function updates $B(v_3)$ from $B(v_3) = \{v_5\}$ to $B(v_3) = \{v_2\}$, and then emits $<v_3, B(v_3) = \{v_2\}>$.

4. Experiments

This section provides the experimental setup in Sect. 4.1 and reports performance evaluations of the proposed method in comparison with other methods including *simple extension* for MapReduce described in Sect. 2.4.

We first confirm the scalability of the proposed method in Sect. 4.2, which is an important aspect of performance for evaluating efficiency in parallel processing [18]–[22]. Next, in Sect. 4.3, we investigate the performance of the proposed method when the graph structural parameter K is varied. *NN-Descent* suffers from a high computational cost [15] when a large K value is employed. Finally, we show that the proposed method relaxes the disadvantages of *simple extension* namely its high memory consumption and low network efficiency.

As the performance measures, we adopted the shuffling cost that mainly corresponds to the transmission cost of intermediates in the SHUFFLE phase, and utilized the elapsed time that depends heavily on an amount of memory consumption and a transmission cost; these alternatives are used as evaluation measures in other Hadoop MapReduce related papers such as [20], [22]–[24]. The approximate K-NNG construction method is more efficient if it is carried out with smaller shuffling cost and elapsed time.

4.1 Experimental Setup

We used a Hadoop cluster that contained one master node and 127 computing nodes. Each node had one Intel Xeon processor E3-1240v2 3.40 GHz with four cores, 16 GB of RAM, and one 4 TB hard disk. A CentOS 5.8 operating system, Java 1.6 with a 64-bit server, and Hadoop 0.20.2 were installed on each node. These nodes were connected to a network whose bandwidth was 1 Gb/s. For our tasks, we configured the Hadoop environment as follows. (1) The block size of the distributed file system (DFS) was fixed at 128 MB. (2) Each node allocated 1 GB of virtual memory (JVM heap size) to the Hadoop daemon. (3) Each computing node allocated 2.5 GB and 4 GB of virtual memory (JVM heap size) to a Map task and a REDUCE task, respectively. (4) Each computing node ran three MAP tasks and one REDUCE task. For the experiments in Sect. 4.3, we employed a server with an Intel Xeon processor E3-1290v2 3.70 GHz, 32 GB of RAM, and a 120 GB Solid State Drive.

We evaluated the proposed method by comparing it with three approaches, using two types of data sets: a real large-scale data set (Image) and a synthetic data set (Random). We first list the four approaches including the proposed method.

Proposed Method (PRO): The proposed method consists of the *four* MapReduce jobs described in Sect. 3. In the MapReduce job, the newly introduced sampling technique was employed to reduce the number of the iterations in the MapReduce jobs.

Sophisticated Approach for MapReduce (SOP): This approach uses the same *four* MapReduce jobs as the proposed method except for the sampling technique. The sampling technique in SOP was the same as that in the original *NN-Descent*. SOP was evaluated as a baseline for confirming the effect of the new sampling technique.

Simple Extension for MapReduce (SIE): A simple MapReduce implementation of *NN-Descent* is implied in [15], which uses *two* MapReduce jobs as described in Sect. 2.4. In Sect. 4.4, we compare SIE with the proposed method to demonstrate that the proposed method constructed an approximate K-NNG much faster than SIE.

NN-Descent on Single Thread (STH): *NN-Descent* was executed on the server by a single thread to measure its elapsed time when the graph structural parameter K was a variable. The original *NN-Descent* in [15] requires the number of the dissimilarity calculations to be proportional to $K^{1.5\text{-}1.8}$. The proposed method was compared with STH regarding the elapsed time for various K values.

We prepared two different types of data sets: Image and Random.

- **Image**: As a real large-scale data set, we used 80 million tiny images [25], which contains 79,302,017 images with a size of 32×32 pixels. The feature vector for each image is a 384-dimensional Gist vector [26]. We randomly sampled almost the half of the data set, 36,499,700 images, without duplication. To evaluate the scalability with respect to the data size in Sect. 4.2.1, we used subsets of 2, 10, 20, 60, 100 percent of the sampled data set.
- **Random**: We generated 1×10^7 vectors with a unit length in a 16-dimensional Euclidean space by randomly sampling from the uniform distribution and normalizing the Euclidean distance of the vector from the origin to 1. Then all the vectors are on the surface of the hyper-sphere in the 16-dimensional Euclidean space.

Moreover, we set three parameters in *NN-Descent*, namely the graph structural parameter K, sampling rate ρ, and parameter for *early termination* δ, as follows; $\rho = 0.5$, $\delta = 0.001$ and we set the default value of K at 100 ($K = 100$) for Image and 40 ($K = 40$) for Random. Note that K was dealt with as a variable in Sect. 4.3.

4.2 Scalability of Proposed Method

The performance of the proposed method was evaluated by measuring elapsed time, the number of iterations in *NN-Descent*, and the recall of the constructed K-NNG. The recall was calculated by

$$Recall = \sum_{v \in V'} Recall(v)/|V'|,$$

Table 2 Performance comparison of the proposed method and the sophisticated approach when they were applied to real data sets of various sizes. The proposed method successfully reduced the number of the iterations with higher recalls.

Data size n	Proposed method (PRO)			Sophisticated approach for MapReduce (SOP)		
	Elapsed time (min)	**# Iterations**	Recall (%)	Elapsed time (min)	**# Iterations**	Recall (%)
729,994	52.82	**7**	98.30	57.28	**9**	98.30
3,649,970	329.20	**9**	95.95	329.22	**11**	95.84
7,299,940	831.25	**10**	94.36	804.47	**12**	94.20
21,899,820	3189.85	**12**	91.20	3242.83	**15**	90.97
36,499,700	6526.72	**13**	89.26	6640.33	**16**	88.44

$$Recall\,(v) = (\text{ Number of true } K\text{-NNs in } B(v)\,)/K\ ,$$

where V' denotes the set of vertices whose true K-NNs were obtained by the brute-force method. In the experiments, $|V'| = |V| = n$ for $n \leq 7,299,940$ and $|V'| = 240,000$ for both $n = 21,899,820$ and $n = 36,499,700$. We applied the method to the real data sets, Image, of various sizes.

We evaluated the scalability from two viewpoints, that is the data size and the number of computing nodes. As regards the scalability of the data size, we evaluated the performance when the size of the data set was a variable and the number of the computing nodes was fixed. As regards the scalability of the number of nodes, the size of the data set was fixed and the number of the computing nodes was a variable.

4.2.1 Scalability as Regards Data Size

We used the data sets with five distinct sizes as shown in Table 2, which were generated from the real data set Image. The number of computing nodes and the graph structural parameter K were fixed at 127 and 100, respectively. Table 2 summarizes the performance of the proposed method compared with the sophisticated approach. The difference between the two methods is in their sampling techniques. As intended, the proposed method successfully reduced the number of iterations in *NN-Descent*, achieving the higher recalls. In particular, as the size of the data set increased, the difference in the numbers of iterations increased. This shows that the proposed method was more efficient for a large-scale data set than the sophisticated approach.

Figure 8 shows elapsed times of both methods versus the size of the data set on a log-log scale. We observed that the curves of the proposed method (PRO) and the sophisticated approach (SOP) were almost proportional to $n^{1.35}$ and $n^{1.36}$, respectively. These exponents 1.35 and 1.36 were close to that of the original *NN-Descent* 1.14, which was measured not in terms of the elapsed time but by the number of dissimilarity calculations. We think that the difference between these exponents arose from the difference of the measurement methods. The elapsed time included I/O costs in addition to distance calculations. Thus the proposed method was scalable with respect to the size of the data sets.

4.2.2 Scalability as Regards Number of Computing Nodes

We varied the number of computing nodes in the cluster from 20 to 100. The data size n and the graph structural parameter K were fixed at $n = 729,994$ and $K = 100$. We employed the measure [19] expressed by

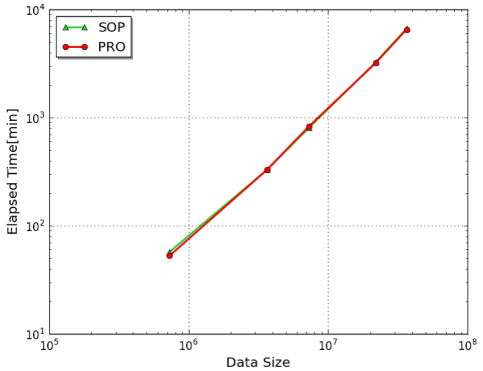


Fig. 8 Elapsed times of the proposed method (PRO) and the sophisticated approach (SOP) versus data sizes on a log-log scale. The curves of PRO and SOP are almost proportional to $n^{1.35}$ and $n^{1.36}$, respectively.

$$Speedup\,(m, m') = \frac{\text{Elapsed time on } m' \text{ nodes}}{\text{Elapsed time on } m \text{ nodes}}\ ,$$

where m denotes the current number of computing nodes, and m' denotes the standard number of computing nodes. In an ideal case, $Speedup\,(m, m') = m/m'$. In practice, it is difficult to achieve an ideal speedup because the data transmission efficiency decreases, i.e., the communication cost between computing nodes increases with increases in the number of computing nodes as described in [18], [21].

Figure 9 shows the $Speedup(m, 20)$ of the proposed method and the sophisticated approach for $m = 20, 30, \cdots, 100$, where m' was set at 20. We observed that the speedup of both methods increased roughly linearly with the number of nodes. The speedup properties are similar to those of the other MapReduce algorithms reported in [21]. Note that the number of computing nodes in our experiments is larger than those in [21] and [20]. In fact, the number range was from 20 to 100 in our experiments whereas they were 1 to 4 in [21] and 2 to 10 in [20]. Thus, our methods were scalable as regards the number of computing nodes for larger numbers than those described in [20], [21].

4.3 Performance with Respect to K

NN-Descent is not practical for constructing a K-NNG with a large K value because *NN-Descent* requires a computational cost proportional to $K^{1.5\sim1.8}$ experimentally [15]. We evaluated the proposed method regarding the performance with K. In the experiments, the data size was fixed at $7,299,940$ ($n = 7,299,940$), and the K value was changed from 20 to 120 in increments of 20. Table 3 summarizes

Table 3 Performance comparison of the proposed method and *NN-Descent* on a single thread when they constructed K-NNGs with various Ks. The proposed method suppressed the increase in the elapsed time with K and achieved the higher recalls.

K	Proposed method (PRO)			NN-Descent on Single Thread (STH)		
	Elapsed Time (min)	# Iterations	Recall (%)	Elapsed Time (min)	# Iterations	Recall (%)
20	373.75	31	49.64	302.08	36	48.91
40	481.32	19	75.66	708.16	22	75.06
60	585.97	14	86.25	1421.85	17	85.87
80	**756.07**	12	**91.48**	**2432.73**	14	**91.23**
100	**831.25**	10	**94.36**	**3737.91**	12	**94.20**
120	**915.48**	9	**96.11**	**5320.09**	11	**96.01**

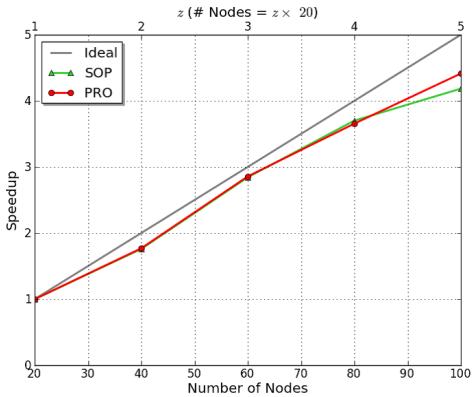


Fig. 9 Speedup of the proposed method (PRO) and the sophisticated approach (SOP) versus the number of computing nodes. The PRO and SOP curves increased approximately linearly.

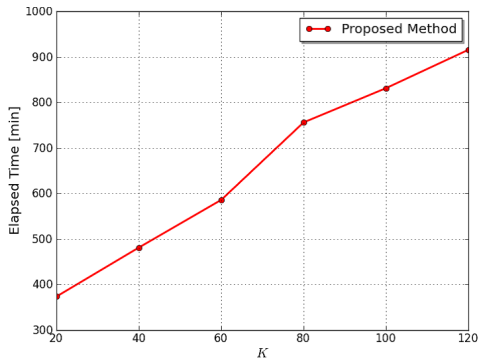


Fig. 10 Elapsed times of the proposed method (PRO) versus K. The elapsed time was nearly sublinear with respect to K.

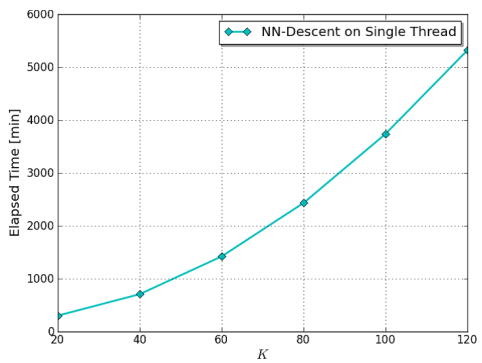


Fig. 11 Elapsed times of the NN-Descent on single thread (STH) versus K. The elapsed time was observed as an almost quadratic curve.

the performance, the elapsed time, the number of iterations in *NN-Descent*, and the recall, with the proposed method (PRO) and *NN-Descent* on a single thread (STH). Compared with STH, PRO suppressed the increase in the elapsed time for large K values, where the approximate K-NNGs were constructed with high recalls exceeding 90%.

Figure 10 shows the elapsed time of PRO versus K. The elapsed time was almost proportional to $K^{0.55}$, i.e., sublinear, compared with $K^{1.88}$ of STH as shown in Fig. 11. This shows that PRO was more scalable as regards K than STH. This property is very important since we often need a sufficiently large value of K to construct an approximate K-NNG with a high recall from a large-scale data set.

4.4 Comparison with Simple Extension

We compared the proposed method (PRO) with *simple extension* (SIE) regarding the transmission efficiency and the elapsed time. In particular, we focused on the transmission efficiency in the final MapReduce job of the two approaches. This is because the transmission efficiency of the final job in SIE is the main problem as described in Sect. 2.4, and we would like to know how many improvements *UpdateKVertices*, namely the final job in PRO, achieves. The transmission efficiency is measured by shuffling cost as is [22], [24], which is total amount of data transmitted in the SHUFFLE phase. The elapsed time of the two approaches was measured for only the final job (FINAL) and for all the MapReduce jobs (ALL), respectively. We used the data sets (Random), which were generated from the synthetic data set with a size of 1×10^7 described in Sect. 4.1. The K range was from 20 to 60. Since SIE often failed to construct a K-NNG with a larger data size or a larger K value in our system environment, we adopted the above settings, the full data size and the maximum K value, which may be small for a large-scale K-NNG.

Figure 12 shows the shuffling cost and the elapsed time of PRO and SIE with $K = 40$ versus the data size, and Fig. 13 shows the results obtained with a data set size of 1×10^7 versus K. Both sets of results show that the proposed method successfully reduced the shuffling cost and the elapsed time of the final job. This results in shorter elapsed times for all the jobs. In particular, PRO was around five times faster than SIE for the full-size data set (1×10^7), illustrated in Fig. 12(b), and around seven times faster for the maximum K ($K = 60$), illustrated in Fig. 13(b). Our novel

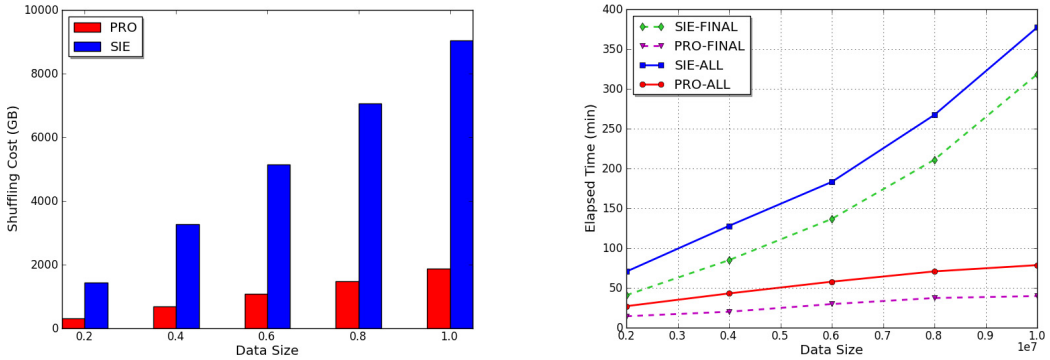


Fig. 12 Shuffling costs and elapsed times of the proposed method (PRO) and *simple extension* for MapReduce (SIE) versus data sizes. (a) PRO achieved around five times lower shuffling cost than SIE. (b) PRO constructed the K-NNG around five times faster than SIE for the full-size data set.

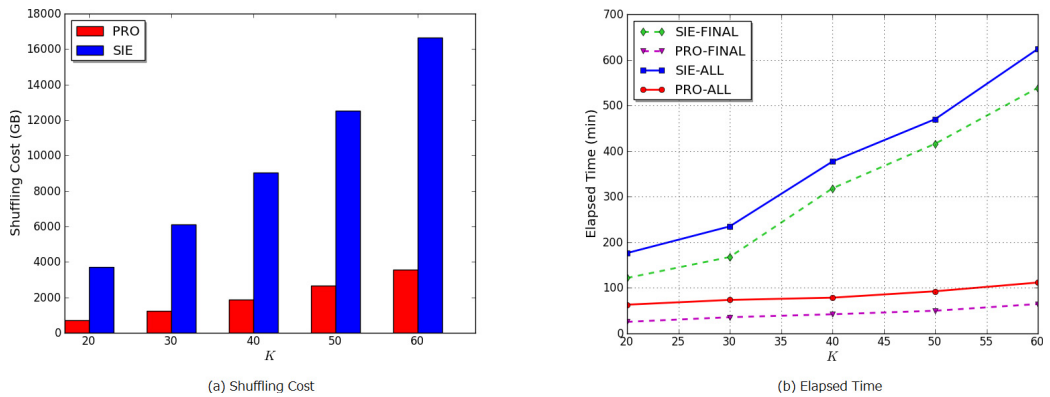


Fig. 13 Shuffling costs and elapsed times of the proposed method (PRO) and *simple extension* for MapReduce (SIE) versus K. (a) PRO achieved around five times lower shuffling costs than SIE. (b) PRO constructed the K-NNG around seven times faster than SIE for the maximum K ($K = 60$).

MapReduce functions and sophisticated key-value formats were more effective and efficient for constructing a K-NNG than SIE.

5. Related Work

This section briefly reviews several methods for K-NNG construction and K-nearest neighbor join that is a closely-related problem.

For a low-dimensional metric space where a data set is not so large, most methods [10], [11] construct an exact K-NNG by utilizing the *triangle inequality*, which is a metric axiom, to reduce the search space. The method described in [10] first builds an index whose structure is either a tree or a pivot table, and then finds the K-nearest neighbors of each object using the index. The method reported in [11] introduces a new pruning metric *NXNDIST*, which provides a tighter upper bound on the distance between an object and its nearest neighbor than traditional pruning metrics. Although these methods work well in a low-dimensional metric space, they become inefficient in a high-dimensional metric space.

A useful method for a high-dimensional metric space is to construct an approximate K-NNG instead of an exact one. The methods based on the divide-and-conquer strategy [12]–[14] are its representatives. The method described in [12] uses the Lanczos algorithm, which can be executed at low computational cost of empirically $O(n^{1.22\sim1.36})$. The methods reported in [13] and [14] first randomly divide a data set into subsets, then construct one K-NN subgraph from each subset, and finally construct one approximate K-NNG by connecting the subgraphs. Although the divide-and-conquer methods are available, they are costly for a large-scale data set. This is because the pruning based on the *triangle inequality* is no longer valid there.

NN-Descent [15] is a heuristic method that efficiently constructs an approximate K-NNG from a high-dimensional data set. *NN-Descent* is based on a simple principle "*a neighbor of a neighbor is also likely to be a neighbor*." From the perspective of the *small-world network* [27], [28], the principle can be interpreted in relation to the hypothesis that a constructed K-NNG has a high *clustering coefficient* [27], [29]. *NN-Descent* achieves high recall with a small number of dissimilarity calculations. In fact, *NN-Descent* outperformed other approximate K-NNG construction methods in reported experiments [15]. For a large-scale data set, a MapReduce implementation of *NN-Descent* is implied in

[15], and we implemented *NN-Descent* as *simple extension*. *NN-Descent* and *simple extension* are detailed in Sect. 2.2 and Sect. 2.4, respectively.

K-nearest neighbor join (*K*-NN join) is superordinate to the *K*-NNG construction and can be applied to it. In recent work on *K*-NN join, some approaches [22], [23] utilize the MapReduce framework for a large-scale data set. H-zkNN [23] maps a multi-dimensional data set into one dimension and performs *K*-NN join by conducting a sequence of one-dimensional range searches. PGBJ [22] is another method for *K*-NN join and exploits pruning rules based on the *triangle inequality* to improve shuffling and computational costs. Although these methods are efficient for a large-scale data set in a low-dimensional metric space, they do not work well for a high-dimensional metric space.

6. Conclusion

We presented an efficient method for constructing approximate *K*-nearest neighbor graphs (*K*-NNGs). The method consists of four MapReduce jobs that employ an appropriate key-value pair format and an efficient sampling strategy. We designed the format of the key-value pairs, where we replaced costly feature vectors with the distance and three attributes. This format led to low memory consumption and high data transmission efficiency measured in terms of the elapsed time. Furthermore, we improved the sampling strategy, which reduced the number of iterations in *NN-Descent*. This improvement was useful for reducing the elapsed time and increasing the recall of the constructed approximate *K*-NNG.

We confirmed that the proposed method is scalable in terms of a data size, the number of machine nodes, and the graph structural parameter *K*. We demonstrated that the proposed method constructed a *K*-NNG around five times faster than *simple extension* of *NN-Descent* for MapReduce.

An important problem to solve still remains although the proposed method successfully achieved efficient *K*-NNG construction from a large-scale data set in a high-dimensional metric feature space. The original *NN-Descent* can handle an arbitrary dissimilarity but the proposed method can deal only with a distance metric. In future work, we intend to extend the proposed method so that it can deal with any dissimilarity.

References

[1] J. Sankaranarayanan, H. Samet, and A. Varshney, "A fast all nearest neighbor algorithm for applications involving large point-clouds," Comput. Graph., vol.31, no.2, pp.157–174, April 2007.
[2] M. Connor and P. Kumar, "Fast construction of k-nearest neighbor graphs for point clouds," IEEE Trans. Vis. Comput. Graphics, vol.16, no.4, pp.599–608, 2010.
[3] A.K. Jain, "Data clustering: 50 years beyond K-means," Pattern Recognit. Lett., vol.31, no.8, pp.651–666, June 2010.
[4] C.T. Chang, J.Z. Lai, and M. Jeng, "Fast agglomerative clustering using information of k-nearest neighbors," Pattern Recognition, vol.43, no.12, pp.3958 – 3968, 2010.
[5] J.B. Tenenbaum, V. de Silva, and J.C. Langford, "A global geometric framework for nonlinear dimensionality reduction," Science, vol.290, no.5500, pp.2319–2323, Dec. 2000.
[6] P. Indyk and A. Naor, "Nearest-neighbor-preserving embeddings," ACM Trans. Algorithms, vol.3, no.3, article 31, Aug. 2007.
[7] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," IEEE Trans. Knowl. Data Eng., vol.17, no.6, pp.734–749, June 2005.
[8] K. Aoyama, K. Saito, H. Sawada, and N. Ueda, "Fast approximate similarity search based on degree-reduced neighborhood graphs," Proc. ACM SIGKDD, pp.1055–1063, 2011.
[9] J. Wang and S. Li, "Query-driven iterated neighborhood graph search for large scale indexing," Proc. ACM Multimedia, pp.179–188, 2012.
[10] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, "Practical construction of k-nearest neighbor graphs in metric spaces," Proc. Int. Workshop Experimental Algorithms, pp.85–97, 2006.
[11] Y. Chen and J. Patel, "Efficient evaluation of all-nearest-neighbor queries," Proc. IEEE ICDE, pp.1056–1065, 2007.
[12] J. Chen, H. Fang, and Y. Saad, "Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection," JMLR, vol.10, pp.1989–2012, Dec. 2009.
[13] P.W. Jones, A. Osipov, and V. Rokhlin, "Randomized approximate nearest neighbors algorithm," Proc. Natl. Acad. Sci., vol.108, no.38, pp.15679–15686, 2011.
[14] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-NN graph construction for visual descriptors," Proc. IEEE CVPR, pp.1106–1113, 2012.
[15] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," Proc. WWW Conf., pp.577–586, 2011.
[16] K.H. Lee, Y.J. Lee, H. Choi, Y.D. Chung, and B. Moon, "Parallel data processing with MapReduce: A survey," Proc. ACM SIGMOD, vol.40, no.4, pp.11–20, Jan. 2012.
[17] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.107–113, Jan. 2008.
[18] X. Xu, J. Jäger, and H.P. Kriegel, "A fast parallel clustering algorithm for large spatial databases," Data Min. Knowl. Discov., vol.3, no.3, pp.263–290, Sept. 1999.
[19] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," Commun. ACM, vol.35, no.6, pp.85–98, June 1992.
[20] R. Vernica, M.J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," Proc. ACM SIGMOD, pp.495–506, 2010.
[21] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on MapReduce," Proc. IEEE CloudCom, pp.674–679, 2009.
[22] W. Lu, Y. Shen, S. Chen, and B.C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," Proc. VLDB Endow., vol.5, no.10, pp.1016–1027, June 2012.
[23] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," Proc. ACM EDBT, pp.38–49, 2012.
[24] B. Bahmani, A. Goel, and R. Shinde, "Efficient distributed locality sensitive hashing," Proc. ACM CIKM, pp.2174–2178, 2012.
[25] A. Torralba, R. Fergus, and W.T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," IEEE Trans. Pattern Anal. Mach. Intell., vol.30, no.11, pp.1958–1970, Nov. 2008.
[26] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," Proc. IEEE CVPR, pp.1–8, 2008.
[27] D.J. Watts and S.H. Strogatz, "Collective dynamics of 'small-world' networks.," Nature, vol.393, no.6684, pp.409–410, 1998.
[28] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," Proc. ACM STOC, pp.163–170, 2000.
[29] M. Newman, "The structure and function of complex networks," SIAM Review, vol.45, no.2, pp.167–256, 2003.

Appendix: Simple Extension Algorithm

The algorithms for the first and second jobs in *simple extension* are shown as a pseudo-code in Algorithms 5 and 6, respectively. The symbols in the pseudo-code have the same meaning as in Sect. 2.4.

Algorithm 5 The first job in *simple extension*

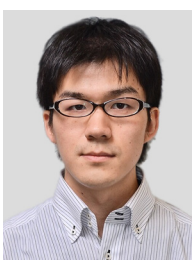
function MAP(v, $B(v)$)
  $B^o(v) \leftarrow$ *old* vertices in $B(v)$, $B^n(v) \leftarrow$ *new* vertices in $B(v)$
  $B^n_s(v) \leftarrow$ SAMPLE($B^n(v), \rho K$)
  EMIT(v, $B(v)$)
  for all $u \in B^o(v) \cup B^n_s(v)$ **do**
    EMIT(u, v)
  end for
end function

function REDUCE(v, $\{B(v), R^o(v), R^n_s(v)\}$)
  $B^o(v) \leftarrow$ *old* vertices in $B(v)$,
  $B^n_s(v) \leftarrow$ sampled *new* vertices in $B(v)$
  $A^o_s(v) \leftarrow B^o(v) \cup$ SAMPLE($R^o(v), \rho K$)
  $A^n_s(v) \leftarrow B^n_s(v) \cup$ SAMPLE($R^n_s(v), \rho K$)
  $A_s(v) \leftarrow A^n_s(v) \cup A^o_s(v)$
  EMIT(v, $\{B(v), A_s(v)\}$)
end function

Algorithm 6 The second job in *simple extension*

function MAP(v, $\{B(v), A_s(v)\}$)
  $A^o_s(v) \leftarrow$ old vertices in $A_s(v)$, $A^n_s(v) \leftarrow$ new vertices in $A_s(v)$
  EMIT(v, $B(v)$)
  for all $u \in A^n_s(v)$ **do**
    EMIT(u, $A_s(v) \backslash \{u\}$)
  end for
  for all $u \in A^o_s(v)$ **do**
    EMIT(u, $A^n_s(v)$)
  end for
end function

function REDUCE(v, $\{B(v), T_s(v)\}$)
  for all $u \in T_s(v)$ **do**
    $l \leftarrow$ calculate dissimilarity $\sigma(v, u)$
    update(u, l, $B(v)$)
  end for
  EMIT(v, $B(v)$)
end function



Tomohiro Warashina received a B.E. degree in Computer Science and Engineering from The University of Aizu, Fukushima, Japan, in 2012 and an M.E. degree in Information Science from Nara Institute of Science and Technology, Nara, Japan, in 2014. His research interests include similarity search and cloud database systems and applications. He is currently a research engineer at Yahoo! Japan Corporation, Tokyo, Japan.

Kazuo Aoyama received a B.E. degree in applied physics from Waseda University in 1986 and an M.E. degree from Tokyo Institute of Technology in 1988. In 1988, he joined NTT Laboratories, NTT Corporation, where he worked on the device modeling of various types of MOSFETs for circuit simulation and the studies of both reconfigurable computer architectures and circuit design methodology. He is engaged in the research of data structures and algorithms.

Hiroshi Sawada received B.E., M.E. and Ph.D. degrees in information science from Kyoto University, Kyoto, Japan, in 1991, 1993 and 2001, respectively. He joined NTT Corporation in 1993. From 2009 to 2013, he was the leader of the Learning and Intelligent Systems Research Group at NTT Communication Science Laboratories, Kyoto, Japan. He is now a senior research engineer, supervisor at NTT Service Evolution Laboratories, Yokosuka, Japan. His research interests include statistical signal processing, audio source separation, array signal processing, machine learning, latent variable model, graph-based data structure, and computer architecture.

Takashi Hattori received a B.E. degree in Mechanical Engineering and an M.S. degree in Informatics from Kyoto University in 2002 and 2004, respectively. In 2004, he joined NTT Communication Science Labs. He is currently a research scientist in the Learning and Intelligent Systems Research Group of the Innovative Communication Lab, and is engaged in research on fast approximate similarity search for various large-scale media.