Efficient MinHash-based Algorithms for Big Structured Data

Wei Wu

Faculty of Engineering and Information Technology

University of Technology Sydney

A thesis is submitted for the degree of

Doctor of Philosophy

July 2018

I would like to dedicate this thesis to my loving parents ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text. This dissertation contains less than 65,000 words including appendices, bibliography, footnotes, tables and equations and has less than 150 figures.

Wei Wu
July 2018

Acknowledgements

I would like to express my earnest thanks to my principal supervisor, Dr. Ling Chen, and co-supervisor, Dr. Bin Li, who have provided the tremendous support and guidance for my research. Dr. Ling Chen always controlled the general direction of my PhD life. She made the recommendation letter for me, and contacted the world leading researchers for collaboration with me. Also, she encouraged me to attend the research conferences, where my horizon was remarkably broaden and I knew more about the research frontiers by discussing with the worldwide researchers. Dr. Bin Li guided me to explore the unknown, how to think and how to write. He once said to me, "It is not that I am supervising you. Instead, this is a spiraling process and we make progress together." In the first two years of PhD, I was still groping in my research topic and did not obtain any results. During that time, I was very anxious and unconfident. However, my supervisors were patient enough and always encouraged me. No doubt that I cannot acquire my current research performance without their patient supervision.

I am grateful to my best friend, Dr. Chuan Luo, who shares three-year research life in Peking University. Although we are currently in different research communities, we often discuss the research trends. In my most depressing days, he enlightened me patiently, even though we were in different countries.

I am grateful to all members in CAI for their participation in and valuable comments on my research. Particularly, I thank Prof. Chengqi Zhang, Dr. Chunyang Liu, Dr. Sujuan Hou, Dr. Meng Fang, Dr. Qinzhe Zhang, Dr. Weiwei Liu, Mr. Yu Liu, Mr. Bo Han, Mr. Yuangang Pan, Mr. Jiangchao Yao, Mr. Adi Lin, Mr. Daokun Zhang, Mr. Dong Wen, Dr. Qian Zhang and Miss Jiamiao Wang, and other students for their help in my PhD study.

I am grateful to the financial support from UTS scholarship and FEIT Travel funds.

Finally, and above all, I would like to thank my parents for their unconditional support and encouragement from emotion and finance. They always believe in me, give me invaluable suggestions and fully support all my decisions. No words could possibly express my deepest gratitude for their endless love, self-sacrifice and unwavering help.

To them I dedicate the dissertation.

Abstract

Nowadays, data are growing explosively in the information society. Every day, an incredibly large number of data are generated from social networks, financial industries, medical device and digital equipment, etc. – Big data have been driving data mining research in both academia and industry. No matter how data mining and machine learning develop, in most tasks such as classification, clustering and retrieval, it is one of the most fundamental operations to compute the similarity of data. However, exact similarity computation has become daunting for big data due to the "3V" nature (volume, velocity and variety). Therefore, the thesis aims to develop a number of efficient randomized hashing algorithms for big structured data to approximate the similarity.

Taking into account the popularity and importance of data modelled as weighted sets, graphs/networks, the MinHash-based algorithms are proposed to transform weighted sets, graphs and network nodes into low-dimensional hash codes for efficient similarity estimation.

In Chapter 4, we propose two MinHash-based algorithms for weighted sets, which are two instances of the Consistent Weighted Sampling (CWS) scheme, for real-value weighted sets by uncovering the working mechanism of the state-of-the-art weighted MinHash algorithm, Improved Consistent Weighted Sampling (ICWS) algorithm. The two algorithms both represent a weighted set as a low-dimensional hash code. The first algorithm, the Canonical Consistent Weighted Sampling (CCWS) algorithm, reconstructs the hash functions in ICWS in order to overcome the flaw that ICWS breaks the uniformity of the MinHash scheme. Consequently, our CCWS algorithm not only shares the same theoretical computational complexity as ICWS but also strictly complies with the definition of the MinHash algorithm. The second algorithm, the Practical Consistent Weighted Sampling (PCWS) algorithm, simplifies ICWS by transforming the original form of ICWS into an equivalent expression. As a result, our PCWS algorithm is not only mathematically equivalent to ICWS and preserves the same theoretical properties, but also saves 20% memory footprint and substantial computational cost compared to ICWS.

In Chapter 5, we propose a MinHash-based algorithm for graphs, i.e., the K-Ary Tree Hashing (KATH) algorithm, for representing a graph as a low-dimensional feature vector.

The main idea of KATH is to construct a traversal table to quickly approximate the subtree patterns in Weisfeiler-Lehman (WL) graph kernel using K-ary trees. Based on the traversal table, our KATH algorithm employs a recursive indexing process that performs only r times of matrix indexing to generate all $(r-1)$-depth K-ary trees, where the leaf node labels of a tree can uniquely specify the pattern. After that, the MinHash scheme is used to fingerprint the acquired subtree patterns for a graph. The experimental results show that our KATH algorithm runs significantly faster than state-of-the-art methods while achieving competitive or better accuracy.

In Chapter 6, we propose a MinHash-based algorithm for network nodes, i.e., the NetHash algorithm. Consequently, each node in the network is mapped to a low-dimensional vector space. Our NetHash algorithm employs the randomized hashing technique to encode shallow trees, each of which is rooted at a node of the network. The main idea is to efficiently encode both attributes and structure information of each node by recursively sketching the corresponding rooted tree from bottom (i.e., highest-order neighboring nodes) to top (i.e., root node), and particularly, to preserve as much information closer to the root node as possible. The extensive experimental results show that the proposed algorithm, which does not need learning, runs significantly faster than the state-of-the-art learning-based network embedding methods while achieving competitive or even better performance in accuracy. Furthermore, the simplicity of the algorithm enables our NetHash algorithm to embed a millions-of-nodes network within 1,000 seconds on a single machine.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, data are growing explosively in the information society. Every day, an incredibly large number of structured or unstructured data are generated from social networks, financial industries, medical devices and digital equipment, etc. Consequently, we are currently in the era of big data.

Big data have been driving data mining research in both academia and industry [25, 97]. No matter how data mining and machine learning develop, in most tasks such as classification, clustering and retrieval, computing the similarity of data is one of the most fundamental operations. However, exact similarity computation has become daunting for big data due to the "3V" nature (volume, velocity and variety). For example, in the scenario of text mining, it is intractable to enumerate the complete feature set (e.g., over 10^8 elements in the case of 5-grams in the original data [97]). Therefore, it is urgent to develop efficient yet accurate similarity estimation techniques.

However, the traditional approaches of data analysis hit the bottleneck in the case of big data due to their limitations, e.g., some trivial problems are dramatically enlarged under the background of the curse of dimensionality so that the traditional methods cannot process effectively. Furthermore, users have different needs in different scenarios. For example, in the real-time systems, efficiency is the most important and users aim to acquire relatively accurate results within a limited time; in the case of limited storage space, users hope to compress data as effectively as possible; in the retrieval tasks, the digests for the original data play a significant role because they contribute to efficient query. In such cases, a powerful tool is the hashing techniques. By mapping the similar objects into the close points in a low-dimensional vector space, the hashing approaches are tactfully designed to solve the problem of curse of dimensionality and nearest neighbors search. Consequently, the

compact representation for the original data via hashing can effectively decrease storage space and accelerate process. So far most existing hashing techniques have been designed for sets or vectors. However, in the real-world, structured data are very common, e.g., networks, chemical compounds, etc. If we adopt sets or vectors for the structured data, a large amount of structure information, e.g., importance of each element and dependence between elements, will be simply ignored, because the simple data structure (i.e., sets or vectors) can only imply whether one element exists. To this end, it is very necessary to design new hashing algorithms for structured data to preserve both content and structure information for the sake of performance enhancement.

1.2 Problem Statement

The core goal of this research is to develop a number of randomized hashing algorithms for large-scale structured data. Specifically, we aim to apply the MinHash algorithm, which is a powerful similarity estimation tool in the bag-of-words model, to different structured data, e.g., weighted sets, graphs and network nodes, for acquiring low-dimensional hash codes and in turn computing similarity. Consequently, we propose some MinHash-based algorithms for various structured data.

1.2.1 The MinHash-based Algorithms for Weighted Sets

The binary set, as the bag-of-words model, is very common in many fields, e.g., natural language processing, computer vision and information retrieval. However, the model only implies whether a data object contains a certain element. In order to better describe data, we assign each element with a real-value weight, which shows the importance of the element in the data. Consequently, a weighted set is obtained.

Fig 1.1 shows a classical example about the tf-idf model in document analysis. Each word is assigned with a statistic numerical, i.e., tf-idf, which reflects how important a word is for a document in a corpus. Fig 1.1 (a) presents a document, and Fig 1.1 (b) gives a universal set for the document analysis, which contains all words in the corpus. Fig 1.1 (c) shows a binary set for the document, where 1 denotes that the document contains the corresponding word, 0 otherwise. By contrast, Fig 1.1 (d) represents the document via a weighted set, where each word is assigned with a tf-idf value. Our task is to represent the weighted sets as low-dimensional hash codes for the sake of performance enhancement in some problems. Our problem statement is as follows:

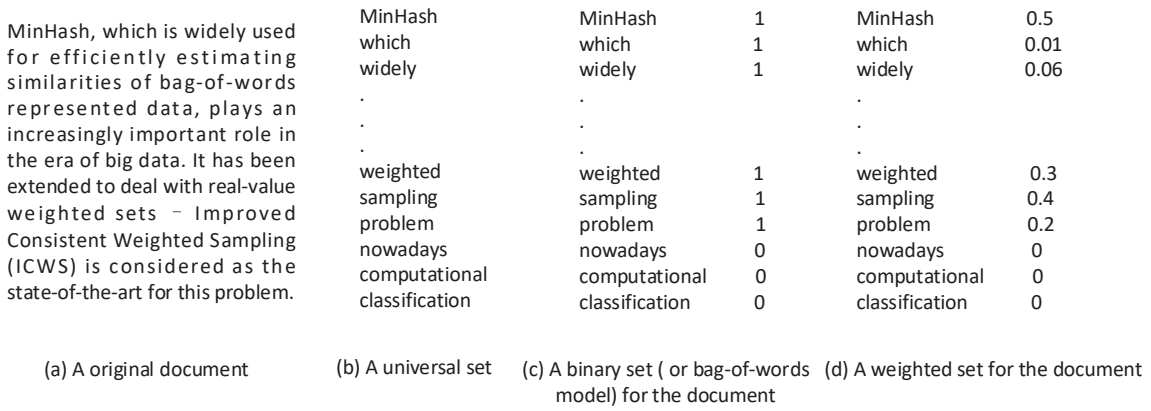


Fig. 1.1 Comparison between a binary set and a weighted set for a document.

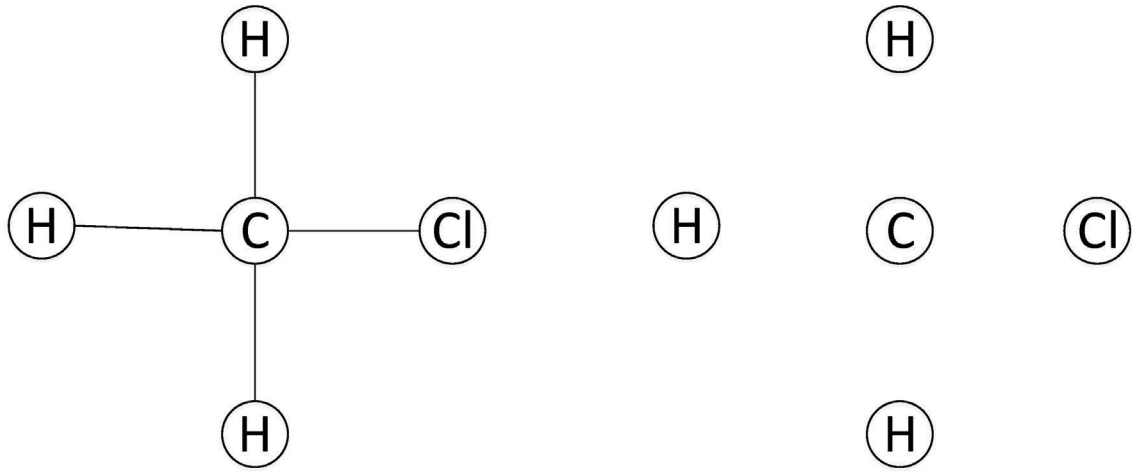
Given a universal set $\mathscr{U} = (U_1, U_2, \cdots, U_n)$ and two subsets $\mathscr{S}, \mathscr{T} \subseteq \mathscr{U}$, each element in the subsets is assigned with a positive real-value weight, i.e., $\mathscr{S} = \{(U_1, S_1), (U_2, S_2), \cdots, (U_n, S_n)\}$ and $\mathscr{T} = \{(U_1, T_1), (U_2, T_2), \cdots, (U_n, T_n)\}$, where $S_i, T_i > 0, i = \{1, 2, \cdots, n\}$. We aim to propose the MinHash-based algorithm (i.e., weighted MinHash) for representing the weighted sets as compact hash codes and in turn computing the similarity.

1.2.2 The MinHash-based Algorithm for Graphs

The weighted set contains more information than the binary set so that the former can describe data more accurately. However, in the real world, there exist a large number of dependence relationships between elements, for example, graphs. Obviously, the (weighted) MinHash algorithms cannot preserve the important structure information in the graph. In order to address the issue, we propose the MinHash-based algorithm for graphs, which can map each graph into a low-dimensional vector space.

Fig 1.2 shows a toy example with regarding to a chemical compound, CH_3Cl. Fig 1.2 (a) gives the corresponding graph which consists of 4 atoms as nodes and 4 edges as bonds. The names of the atoms denote the label of each node. Fig 1.2 (b) shows the corresponding set representation. We observe that the graph preserves the label information of the nodes and structure information between nodes, while the set discards all structure information. Our task is to sketch the graph and obtain the compact graph representation via the MinHash algorithm. Our problem statement is as follows:

Give a set of labeled graph $\mathscr{G} = \{g_i\}_{i=1}^N$, where N denotes the number of graph objects. A graph g_i is represented as a triplet $(\mathscr{V}, \mathscr{E}, \ell)$, where $\mathscr{V}$ denotes the node set, $\mathscr{E}$ denotes the undirected edge set, and $\ell : \mathscr{V} \to \mathscr{L}$ is a function that assigns a label from a label set $\mathscr{L}$ to each node. We aim to propose the MinHash-based algorithm, i.e., K-Ary Tree Hash-



(a) A chemical compound, CH_3Cl, modeled as a graph.

(b) A chemical compound, CH_3Cl, modeled as a set.

Fig. 1.2 Comparison between a graph and a set for a chemical compound, CH_3Cl.

ing (KATH), for representing the graph as a low-dimensional feature vector. Furthermore, we can compute the similarity between compact graph representation and conduct graph classification.

1.2.3 The MinHash-based Algorithm for Network Nodes

In the above tasks, each data object, i.e., a weighted set or a graph, is represented as a low-dimensional hash code, and furthermore, we can compute the similarity between two data objects. However, in some scenarios, we need to measure the similarity between components within a data object, but the components are dependent on each other. For example, in a network, we compute the similarity between any two nodes for the sake of the high-level applications, e.g., recommendation and query. We would like to note that any node in the network is not independent, that is, the node is heavily impacted by its own attributes and neighboring nodes.

Fig 1.3 shows a toy example about a citation network, where each node represents a paper and the edge between two nodes denotes the citation relationship. In addition, each node uses the abstract of the paper as attributes. Our goal is that each node in the network is embedded into a low-dimensional vector space. Our problem statement is as follows:

Given an attributed network $G = (\mathscr{V}, \mathscr{E}, f)$, where $\mathscr{V}$ denotes the node set of the network, $\mathscr{E}$ denotes the undirected edge set of the network, and $f : \mathscr{V} \mapsto \mathscr{A}$ is a function that assigns several attributes from an attribute set (or a global vocabulary) $\mathscr{A}$ to each node. We aim to

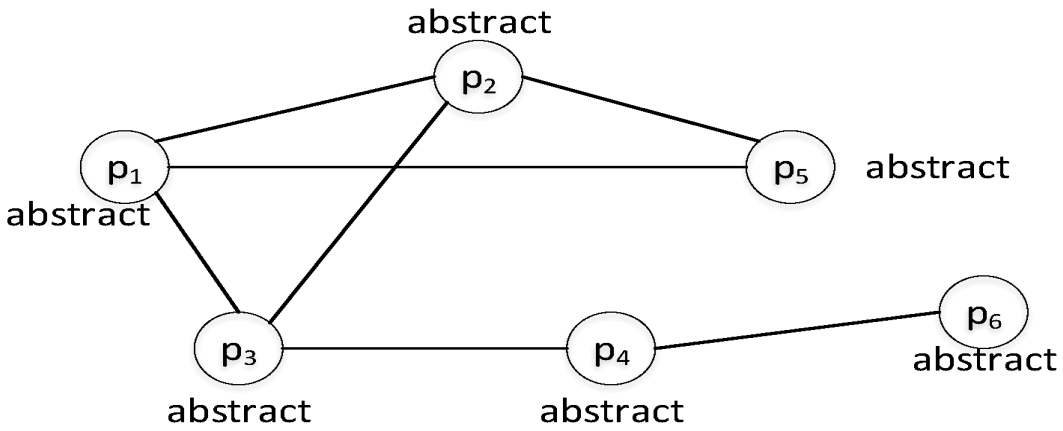


Fig. 1.3 A citation network.

propose the MinHash-based algorithm, i.e., NetHash, in order to represent network nodes as low-dimensional vector and compute the similarity between nodes.

1.3 Contributions

The thesis aims to propose the MinHash-based algorithms for efficiently approximating the similarity between structured data.

Our contributions are listed as below:

- **MinHash-based Algorithms for weighted sets:** The Consistent Weighted Sampling (CWS) scheme [84] was proposed as a variant of the MinHash algorithm on weighted sets, i.e., weighted MinHash, and can unbiasedly estimate the generalized Jaccard similarity on weighted sets. To the best of our knowledge, the Improved Consistent Weighted Sampling (ICWS) algorithm [49] is remarkable in both theory and practice, and considered as the state-of-the-art method for weighted MinHash algorithms. We further explore the approach from two perspectives, and propose two instances of the CWS scheme. Our contributions are as follows:

    1. We show that ICWS violates the uniformity of the MinHash scheme. In order to solve the flaw, we propose the Canonical Consistent Weighted Sampling (C-CWS) algorithm by reestablishing the function relationship. CCWS not only shares the same theoretical computational complexity as ICWS but also strictly complies with the definition of MinHash.
    2. By transforming the original form of ICWS into a simpler expression, we propose a mathematically equivalent approach, the Practical Consistent Weighted

Sampling (PCWS) algorithm, which has a lower complexity than ICWS but preserves the same theoretical properties.

- **MinHash-based Algorithm for Graphs:** Weighted sets show that each element has a different importance degree via weights, but they cannot illustrate the dependence relationship between elements. A graph, as a classic data structure, contains multiple link relationships, and thus is very common in the real-world, e.g., chemical compounds. By simulating the generation of the Weisfeiler-Lehman (WL) graph kernel, we propose a K-Ary Tree Hashing (KATH) algorithm to fast sketch graphs. Consequently, we efficiently represent the graph as a low-dimensional vector for computing the similarity between graphs. Our contributions are as follows:

  1. We build a *traversal table* to fast generate K-ary trees and in turn simulate the WL graph kernel.
  2. We propose the K-Ary Tree Hashing (KATH) algorithm to efficiently sketch graphs for compact graph representation. Compared with the WL graph kernel [108], the KATH algorithm achieves the competitive accuracy with a dramatically reduced time.

- **MinHash-based Algorithm for Network Nodes:** The MinHash-based algorithm for graphs aim at compact graph representation, which consists of the label and structure information of the graph. By contrast, in a network, each node contains not only their own attributes of the nodes, but also structure information from dependence relationship, which means that the nodes in the network are dependent of each other. Therefore, our task is to represent each node as a low-dimensional vector, i.e., network node embedding, and furthermore, we can efficiently compute the similarity between nodes based on the compact representation. To this end, we propose the MinHash-based algorithm for network node embedding. Our contributions are as follows:

  1. This work is the first endeavor to introduce randomized hashing techniques into attributed network node embedding.
  2. We propose the NetHash algorithm, which can achieve competitive or better performance than the state-of-the-art learning-based network node embedding algorithms in accuracy with significantly reduced computational cost, to embed each node in the attributed network into a low-dimensional vector space.

1.4 Structure

The framework of the whole thesis is in Figure 1.4:

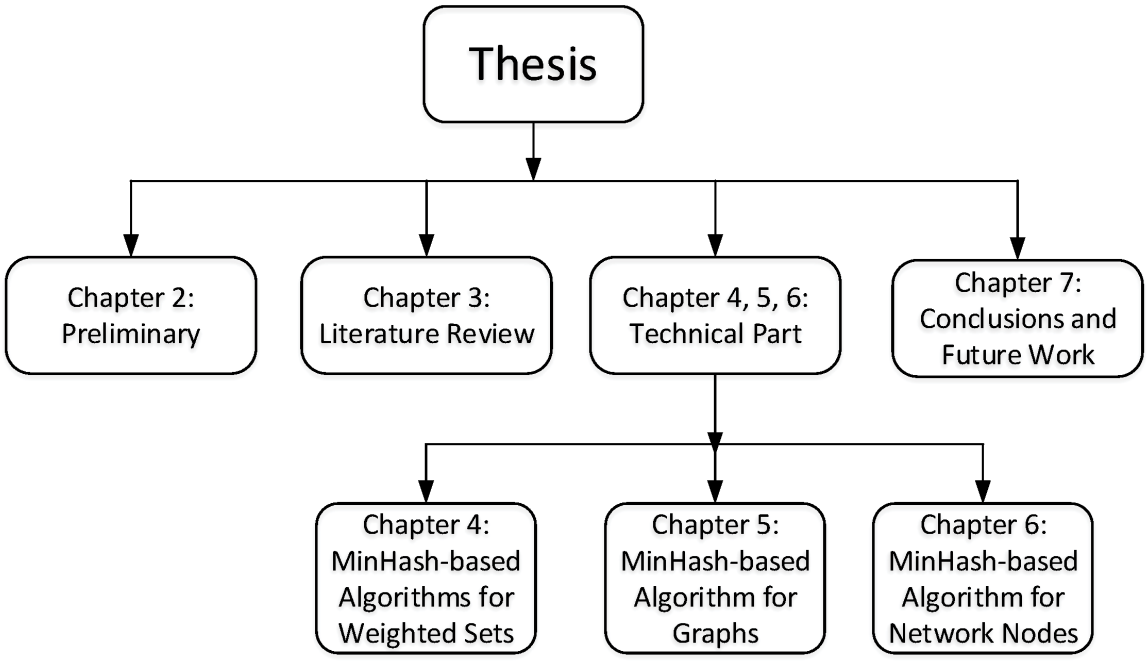


Fig. 1.4 Thesis structure.

The content of each chapter in the thesis is summarized as follows:

Chapter 2: This chapter provides preliminary knowledge and definitions for the Jaccard similarity and the MinHash algorithm. It also summarizes major notations in the thesis, and shows all data sets adopted in the thesis.

Chapter 3: This chapter is a literature review that surveys the existing works on the hashing techniques.

Chapter 4 presents two MinHash-based algorithms for weighted sets, which are two instances of the CWS scheme. By exploring the state-of-the-art weighted MinHash algorithm, ICWS, we improve it from two perspectives. The first one is the CCWS algorithm which overcomes the flaw that ICWS violates the uniformity of the MinHash scheme. The second one is the PCWS algorithm which improves efficiency of ICWS by uncovering the working mechanism and in turn simplifying the mathematical expression.

Chapter 5 proposes a MinHash-based algorithm for graphs, i.e., K-Ary Tree Hashing (KATH). By recursively sketching the K-ary tree, which is used to approximate the state-of-the-art graph kernel, the Weisfeiler-Lehman (WL) kernel, the KATH algorithm represents the graph as a low-dimensional feature vector and in turn efficiently computes the similarity between the graphs.

Chapter 6 proposes a MinHash-based algorithm for network nodes, i.e., NetHash. By expanding each node along with its neighboring nodes into a rooted tree with the node itself being the root node, we can preserve both attributes and structure information in the rooted tree. Then, by recursively sketching the rooted tree, the NetHash algorithm represents the node as a low-dimensional vector for efficiently computing the similarity between nodes.

Chapter 7 concludes this thesis and outlines the direction for future works.

1.5 Publications

Below it is a list of the papers associated with my PhD research that have been submitted, accepted, and published:

Journal Publications:

1. **Wei Wu**, Bin Li, Ling Chen, Xingquan Zhu, Chengqi Zhang. *K*-Ary Tree Hashing for Fast Graph Classification. ***IEEE Transactions on Knowledge and Data Engineering*** (TKDE), 30(5):936-949, 2018.

2. **Wei Wu**, Bin Li, Ling Chen, Chengqi Zhang, Philip S. Yu. Improved Consistent Weighted Sampling Revisited. ***IEEE Transactions on Knowledge and Data Engineering (major revision)***.

Conference Publications:

1. **Wei Wu**, Bin Li, Ling Chen, Chengqi Zhang. Efficient Attributed Network Embedding via Randomized Hashing. *Proceedings of the 27th International Joint Conference on Artificial Intelligence (**IJCAI 2018**)*, pp. 2861-2867 (2018).

2. **Wei Wu**, Bin Li, Ling Chen, Chengqi Zhang. Consistent Weighted Sampling Made More Practical. *Proceedings of the 26th International Conference on World Wide Web (**WWW 2017**)*, pp. 1035–1043 (2017).

3. **Wei Wu**, Bin Li, Ling Chen, Chengqi Zhang. Canonical Consistent Weighted Sampling for Real-Value Min-Hash. *Proceedings of the 16th International Conference on Data Mining (**ICDM 2016**)*, pp. 1287–1292 (2016).

4. **Wei Wu**, Bin Li, Ling Chen, Chengqi Zhang. Cross-View Feature Hashing for Image Retrieval. *Proceedings of the 20th Pacific Asia Conference on Knowledge Discovery and Data Mining (**PAKDD 2016**)*, pp. 203–214 (2016).

Chapter 2

Preliminary

In this chapter, we first give our common definitions and important notations which will be used throughout the thesis. Next, we will summarize all data sets which will be used in our experiments.

2.1 Definitions

Definition 1 (Jaccard Similarity) *Given a universal set $\mathscr{U} = (U_1, U_2, \cdots, U_n)$ and its subset $\mathscr{S} \subseteq \mathscr{U}$, if for any element $S_k \in \mathscr{S}$, $S_k = 1$ or $S_k = 0$, then we call $\mathscr{S}$ a binary set. The Jaccard Similarity is defined as*

$$J(\mathscr{S}, \mathscr{T}) = \frac{|\mathscr{S} \cap \mathscr{T}|}{|\mathscr{S} \cup \mathscr{T}|}.$$

The Jaccard similarity is simple and effective in many applications, especially for document analysis based on the bag-of-words representations [112].

Definition 2 (MinHash [11]) *Given a universal set $\mathscr{U}$ and a subset $\mathscr{S} \subseteq \mathscr{U}$, MinHash is generated as follows: Assuming a set of D hash functions (or D random permutations), $\{\pi_d\}_{d=1}^{D}$, where $\pi_d : \mathscr{U} \mapsto \mathscr{U}$, are applied to $\mathscr{U}$, the elements in $\mathscr{S}$ which have the minimum hash value in each hash function (or which are placed in the first position of each permutation), $\{\min(\pi_d(\mathscr{S}))\}_{d=1}^{D}$, would be the MinHashes of $\mathscr{S}$.*

MinHash [11] is an approximate algorithm for computing the Jaccard similarity of two sets. It is proved that the probability of two sets, $\mathscr{S}$ and $\mathscr{T}$, to generate the same MinHash

value (hash collision) is exactly equal to the Jaccard similarity of the two sets:

$$\Pr\left(\min(\pi_d(\mathscr{S})) = \min(\pi_d(\mathscr{T}))\right) = J(\mathscr{S},\mathscr{T}).$$

From the above MinHash scheme we can see that all the elements in $\mathscr{U}$ are considered equally because all the elements can be mapped to the minimum hash value with equal probability.

2.2 Notations

The important notations used in the thesis are summarized in table 2.2. Note that in each chapter, we may also use some additional notations for each chapter.

Table 2.1 Important notations used in the thesis

Symbols	Definition
$\mathscr{U}$	A universal set
U_i	An element in $\mathscr{U}$
$\mathscr{S},\mathscr{T}$	Subsets of $\mathscr{U}$
S_k, T_k	Weight of the k-th element in $\mathscr{S}$ and $\mathscr{T}$
$\min(S_k, T_k)$	Minimum between S_k and T_k
$\max(S_k, T_k)$	Maximum between S_k and T_k
π_d	A random permutation
D	The length of fingerprint or the number of dimension of embedding
$\min(\pi_d(\mathscr{S}))$	A MinHash of $\mathscr{S}$
$h : k \mapsto v_k$	A random hash function
$h : (k, y_k) \mapsto v_{k,y}$	A random hash function
$\mathbf{K}$	Similarity matrix
$\mathbf{x}_S, \mathbf{x}_T$	Fingerprints for $\mathscr{S},\mathscr{T}$
Pr	Probability value
pdf	Probability density function
Exp	Exponential distribution
Uniform	Uniform distribution
Gamma	Gamma distribution
Beta	Beta distribution
$\mathbb{E}$	Expectation value
$\hat{S}_k$	Estimator for S_k

$J(\mathscr{S},\mathscr{T})$	Jaccard similarity between S and T
$generalizedJ(\mathscr{S},\mathscr{T})$	Generalized Jaccard similarity between S and T
$[0,S_k]$	A closed interval between 0 and S_k
det	Determination
$\mathscr{G}$	A set of graphs
g_i	A graph
$\mathscr{V}$	A set of nodes
$\mathscr{E}$	A set of edges
$\mathscr{L}$	A set of node labels
$\ell : \mathscr{V} \mapsto \mathscr{L}$	A function that assigns a label in $\mathscr{L}$ to each node
y	A graph label
κ	A kernel between two graphs
G	A network
$\mathscr{A}$	An attribute set
$f : \mathscr{V} \mapsto \mathscr{A}$	A function that assigns attributes in $\mathscr{A}$ to each node
$\mathbf{v}_v$	A feature vector of Node v
$\mathbf{x}_v$	An embedding representation for Node v
$\{0,1\}^{\lvert\mathscr{A}\rvert}$	A 0/1 vector with the dimension being $\lvert\mathscr{A}\rvert$
$\mathbb{R}^D$	A real-value vector with the dimension being D
d_l	The number of MinHash functions at the l-th level
ν_v	The degree of Node v
S	Entropy

2.3 Benchmark Data Sets

We have collected a number of real-world data sets including texts, chemical compounds, social networks and scientific publication networks, and the data sets are modeled as weighted sets, graphs and networks.

2.3.1 Weighted Sets

Each data set is composed of instances, each of which represents a weighted set. We introduce a wide range of text benchmarks.

Table 2.2 Summary of the six weighted set data sets

Dataset	$\lvert\mathscr{D}\rvert$	$\lvert\mathscr{F}\rvert$	avg.ρ	avg.$\sigma(w)$
Real-sim	20,000	20,958	0.0025	0.0069
Rcv1	20,000	47,236	0.0016	0.0059
Webspam	350,000	680,715	0.0051	0.0015
DBLP-set	14,000	23,199	0.0032	0.0027
KDD	20,000	20,216,830	0.0004	0.0030
Url	2,396,130	3,231,961	0.000036	0.0000027

Real-sim[1]: The Real-sim data set is a collection of UseNet articles from four discussion groups about *simulated auto racing*, *simulated aviation*, *real autos* and *real aviation*, respectively. The original document dataset[2], with 72,309 samples and 20,958 features, has been arranged in two classes, *real* and *simulated*. Since the real class and the simulated class of the original data are unbalanced, data preprocessing has been performed in advance by randomly selecting 10,000 real samples and 10,000 simulated samples as positive instances and negative ones, respectively.

Rcv1 [64]: The Rcv1 data set is a large collection of newswire stories drawn from online databases. The formatted data set[3] has 20,242 training samples and 677,399 testing ones with 47,236 features. The data set has been categorized into two types: positive instances contain CCAT and ECAT, while negative ones have GCAT and MACT on the website. Similarly, we randomly select 10,000 positive instances and 10,000 negative ones to compose a balanced data set.

Webspam[4]: The Webspam data set, provided by a large-scale learning challenge, is a web text data set. The data set has 350,000 samples and 680,715 features, and each sample is classified into spam or non-spam. In the classification task, we randomly choose 10,000 positive samples and 10,000 negative ones to form a balanced data set; in the retrieval task, we randomly select 1,000 samples from the original data set as query examples and the rest as the database.

DBLP-set [120]: There are 1,572,277 papers in the downloaded raw data package. Following [19], we consider a binary classification task: Artificial Intelligence (AI) vs. Computer Vision (CV). We define the papers in {IJCAI, AAAI, NIPS, UAI, COLT, ACL, KR, ICML, ECML, IJCNN} as the AI category and {CVPR, ICCV, ECCV, ICIP, ICPR, ACM

[1] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#real-sim
[2] https://people.cs.umass.edu/~mccallum/data.html
[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#rcv1.binary
[4] http://largescale.ml.tu-berlin.de/instructions/

Multimedia, ICME} as the CV category. Since the AI and CV papers of the original data are unbalanced, data preprocessing was performed in advance by randomly selecting 7,000 AI abstracts and 7,000 CV ones.

KDD: This is a large educational data set from the KDD Cup 2010 competition. The formatted data set has 8,407,752 training samples with 20,216,830 features. We also randomly select 10,000 positive instances and 10,000 negative ones to form a balanced data set for classification.

Url [82]: The data set contains 2,396,130 URLs and 3,231,961 features. We randomly select 1,000 samples from the original data set as query examples and the rest as the database.

We summarize the five adopted data sets in Table 2.2. In this table, we present four characteristics for each data set, where $|\mathscr{D}|$ means the size of the data set, $|\mathscr{F}|$ means the size of the dictionary (or the universal set) of each data set, avg.ρ indicates the mean value of the density for each data instance (a small value represents that the data set is sparse), and avg.$\sigma(w)$ indicates the mean value of the standard deviation of each element in the dictionary (a large value represents that the data set contains weights with high variance).

2.3.2 Graphs

Each data set is composed of objects, each of which represents a graph. We introduce a wide range of graph data sets, including chemical compounds and social networks.

Mix [108]: The first group comprises 3 real-world graph data sets[5] used as the testbed in [108]: (1) MUTAG consists of 188 mutagenic aromatic and heteroaromatic nitro compounds, where the classification task is to predict whether they have a mutagenic effect on the Gram-negative bacterium Salmonella typhimurium. (2) ENZYMES consists of 600 enzymes represented in protein tertiary structures, which are required to be classified into one of the 6 EC top-level classes. (3) D&D consists of 1,178 protein structures, where the classification task is to predict whether a protein structure is an enzyme or non-enzyme.

NCI [65]: The second group comprises 9 NCI data sets[6] used as the testbed in [65]. Each data set has $28,000 \sim 42,000$ chemical compounds, each of which is represented as a graph whose nodes are atoms and edges are bonds. Each data set belongs to a bioassay test for predicting anti-cancer activity against a certain kind of cancer, where a graph (chemical compound) is labeled positive (active) if it is active against the cancer. Since the positive and negative examples of the original data are unbalanced (about 5% positive samples), following [65], data preprocessing is performed in advance by randomly selecting a negative

[5] http://mlcb.is.tuebingen.mpg.de/Mitarbeiter/Nino/WL/Data.zip
[6] https://sites.google.com/site/libin82cn/nshk-data&codes.zip

Table 2.3 Summary of the seventeen graph data sets.

Data Set	$\lvert\mathscr{C}\rvert$	$\lvert\mathscr{G}\rvert$	avg. $\lvert\mathscr{V}\rvert$	avg. $\lvert\mathscr{E}\rvert$	$\lvert\mathscr{L}\rvert$
MUTAG [108]	2	188	17.93	19.80	7
ENZYMES [108]	6	600	32.63	62.14	3
D&D [108]	2	1,178	284.32	715.66	82
NCI1 [65]	2	4,464	31.01	33.61	51
NCI33 [65]	2	3,612	31.14	33.73	48
NCI41 [65]	2	3,380	31.33	33.90	39
NCI47 [65]	2	4,362	31.06	33.65	52
NCI81 [65]	2	5,240	30.34	32.86	50
NCI83 [65]	2	4,874	30.39	32.90	40
NCI109 [65]	2	4,504	30.83	33.41	51
NCI123 [65]	2	6,776	29.66	32.12	57
NCI145 [65]	2	4,240	30.78	33.35	49
qHTS	2	20,000	26.24	28.34	18
NCI-Yeast	2	20,000	23.42	50.03	68
DBLP-graph [92]	2	19,456	10.48	19.64	41,325
Twitter [93]	2	20,000	4.03	5.00	1,307
Syn10K20L10E	2	10,000	8.74	11.77	20

subset from each data set with the same size as the positive one (e.g., 2,232 positive and 2,232 negative samples in NCI1).

qHTS[7]: This is a relatively large data set for graph classification. It comprises 362,359 chemical compounds, each of which is represented as a graph. They are tested in a bioassay test to predict the inflammasome activations. A graph is labeled positive (active) if it is active against inflammasome. As in NCI, for the sake of balance, data preprocessing is performed in advance by randomly choosing 10,000 positive and 10,000 negative instances.

NCI-Yeast[8]: This is another relatively large data set for graph classification. It comprises 86,130 chemical compounds, each of which is represented as a graph. They are tested in a bioassay test to measure their ability to inhibit the growth of yeast strains. A graph is labeled positive (active) if it can inhibit yeast growth. Again, data preprocessing is performed in advance by randomly choosing 10,000 positive and 10,000 negative instances.

DBLP-graph [92]: Each record in DBLP represents one paper, containing paper ID, title, abstract, authors, year of publication, venue, and references of the paper. Following [92],

[7] http://pubchem.ncbi.nlm.nih.gov/assay/assay.cgi?aid=743279
[8] http://pubchem.ncbi.nlm.nih.gov/assay/assay.cgi?aid=175

each paper named as a core paper is represented as a graph, whose nodes denote the paper ID or a keyword in the title. If there exists citation between the core paper and another, the paper ID and all keywords will be added to the graph as nodes. Edges are produced as follows: 1) citation between two paper IDs corresponds to one edge in the graph; 2) two nodes, one representing paper ID and the other representing a keyword in the paper title, corresponds to one edge in the graph; 3) all nodes representing keywords of a paper are fully connected with each other as edges in the graph. Core papers in computer vision (CV) are labeled as positive; while core papers in database/data mining (DBDM) and artificial intelligence/machine learning (AIML) are labeled as negative. Consequently, there are 9,530 positive instances and 9,926 negative instances. We adopt this unbalanced data set in the experiments.

Twitter [93]: Following [93], each tweet is represented as a graph with nodes being terms and/or smiley symbols (e.g, :-D and :-P) while edges being the co-occurrence relationship between two words or symbols in each tweet. Furthermore, each tweet is labeled as a positive feeling or a negative one. Consequently, we obtain 67,987 positive and 76,046 negative tweets. We randomly sample an unbalanced data set (9,425 positive instances and 10,575 negative instances) with the same ratio of positive and negative instances as the original data set.

Synthetic 10K: The last is a synthetic graph data set. We use a synthetic graph data generator[9] to generate 10,000 labeled, undirected and connected graphs, named Syn10K20L10E, where "20L" indicates that the number of unique node labels in the graph set is 20 while "10E" indicates that the average number of edges in each graph is 10. To simulate graph classification tasks, we perform the following steps: 1) we find a set of frequent subgraph patterns (support > 0.02) using gSpan[10] [146] from each graph set; 2) we represent each graph as a feature vector, with a dimension being 1 if it has the corresponding subgraph, otherwise it being 0; 3) we perform 2-Means clustering for each graph set and labeled the graphs in one cluster positive and the other negative. As a result, we obtain one relatively large graph classification task.

The statistics of the above graph data sets are summarized in Table 2.3, in which $|\mathscr{C}|$ denotes the number of classes in each data set, $|\mathscr{G}|$ denotes the number of graphs, $|\mathscr{V}|$ and $|\mathscr{E}|$ denote the number of nodes and edges of graphs, respectively, and $|\mathscr{L}|$ denotes the number of unique node labels in a graph data set.

[9] http://www.cais.ntu.edu.sg/~jamescheng/graphgen1.0.zip
[10] http://www.cs.ucsb.edu/~xyan/software/gSpan.htm

Table 2.4 Summary of the seven network data sets.

Data Set	$\lvert\mathscr{V}\rvert$	$\lvert\mathscr{E}\rvert$	$\overline{v}$	$\lvert\mathscr{A}\rvert$	$\lvert\overline{\mathscr{A}}\rvert$	S	$\lvert\mathscr{L}\rvert$
Cora	2,708	5,429	3.8	1,433	18.17	2.11	7
Wikipedia	2,405	17,981	9.08	4,973	673.31	3.06	19
Flickr	7,575	239,738	63.29	12,047	24.13	4.81	9
BlogCatalog	5,196	171,743	66.11	8,189	71.10	4.89	6
ACM	1,108,140	6,121,261	11.05	586,190	59.09	3.26	-

2.3.3 Networks

Each data set represents a network. We introduce a wide range of network benchmarks, including citation networks and social networks.

Cora [87]: This is a typical machine learning paper citation network. The formatted data set, with 2,708 papers, 5,429 links and 1,433 attributes, has been arranged into 7 classes.

Wikipedia [147]: This data set is a collection of articles in Wikipedia. The formatted data set contains 2,405 articles with 4,973 attributes from 19 classes and 17,981 hyperlinks among the articles.

Flickr [66]: It is an image and video hosting website where users interact with each other via photo sharing. We use the following relationships among users to form a network. Each user can specify a list of tags of interest which are considered as his/her attribute information. The formatted data set is comprised of 5,196 users with 8,189 tags of interest and 171,743 follower-followee relationship.

BlogCatalog [66]: It is an online community where people can post blogs. The bloggers follow each other and form a network. We use the keywords in the bloggers' blog descriptions as the attribute information. The formatted data set consists of 7,575 bloggers with 12,047 blog descriptions and 239,738 relationships.

ACM [120]: The original data is a collection of papers from ACM, which consists of 2,381,688 papers and 10,476,564 citation relationship. Before experiments, we clean up papers without abstracts or citations in the data. Then, we build a citation network with papers being nodes and citation being edges, where each node uses abstract as attributes.

The data sets are summarized in Table 2.4, where $\lvert\mathscr{V}\rvert$ denotes the number of node in the network, $\lvert\mathscr{E}\rvert$ denotes the number of edges in the network, $\overline{v}$ means the average node degrees, $\lvert\mathscr{A}\rvert$ is the size of attribute set, $\lvert\overline{\mathscr{A}}\rvert$ indicates the average number of node attributes, S is entropy of degrees of nodes and $\lvert\mathscr{L}\rvert$ is the number of node labels.

Chapter 3

Literature Review

This chapter presents a series of relevant works in connection with this thesis. This thesis aims to design efficient randomized hashing algorithms for various structured data, i.e., weighted sets, graphs and network nodes. Before introducing the proposed hashing algorithms for structured data in Chapters $4 \sim 6$, we investigate the related work. First, the hashing techniques are introduced in Section 3.1. Then, we review two categories of the hashing techniques, which are both used for similarity estimation, i.e., learning to hash and locality sensitive hashing in Section 3.2. Subsequently, we introduce the MinHash algorithm, on which the proposed algorithms all focus in details in Section 3.3. Finally, we introduce the current hashing algorithms for structured data in Section 3.4.

3.1 Hashing Techniques

Data query is a basic operation in the database. However, it is very time-consuming to scan the whole database record by record, especially in large-scale database. On the other hand, it is impossible to load all data in the database into the memory because in real-world scenarios, the former is much more larger than the latter. One feasible solution is to employ the hashing techniques (or hash functions) which can transform data of arbitrary size into data of shorter fixed-size or bucket addresses. Formally, given a data object $\mathbf{x}$ and a hash function $h(\cdot)$, the object is mapped to the address, $h(\mathbf{x})$ (also known as the hash value). Consequently, we can perform various operations, e.g., insertion, deletion and query, etc., on the compressed data very efficiently. However, the traditional hashing techniques encounter a serious problem of collision, that is, some different data might be mapped to the same hash value in the case that the original data are more than the number of hash values. To this end, in addition to adopting good hash functions which generate a few collisions, it is inevitable to pay the extra expenses for processing the collisions.

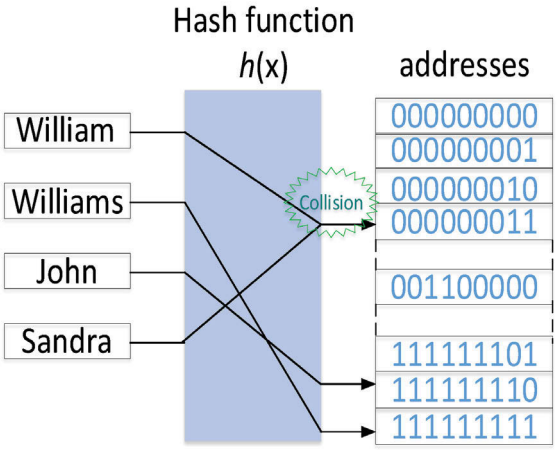


Fig. 3.1 A toy example for a traditional hash function and collision.

Fig 3.1 shows a toy example for a traditional hash technique, where each name is mapped to an address via a traditional hash function $h(\cdot)$. When querying the name "Williams", we will not compare the query with records one by one in the database and instead, we will be able to directly locate the address "111111111" by hashing the query. Consequently, we find the query in $O(1)$ time. On the other hand, we observe that two names, "William" and "Sandra", are mapped to the same address, "000000011" and thus collision emerges in the traditional hashing techniques.

3.2 Similarity Hashing

Although the traditional hashing techniques can save the storage space and accelerate lookup, it is impossible to preserve neighborhood structure information between data objects, which is very important in many machine learning tasks, e.g., classification, clustering and retrieval, etc. For example, if there exists difference of only one character between two data objects, the hash values for them will be probably totally different because a good hash function is able to decrease the collision probability. As shown in Figure 3.2(a), two very similar names, William and Williams, are mapped to the two addresses by a good hash function $h(\cdot)$, "000000000" and "111111111", respectively, which are far from each other. Given a query "Will", we aim to acquire all names which begin with "Will". In this case, we cannot return the two names quickly, i.e., "William" and "Williams", because the query and the two records are all mapped to different hash values via the traditional hashing techniques ("001100000" for the query). Consequently, we still need to exhaustively scan the original

database to find all names meeting the requirement of the query, which tends to be expensive in large-scale database.

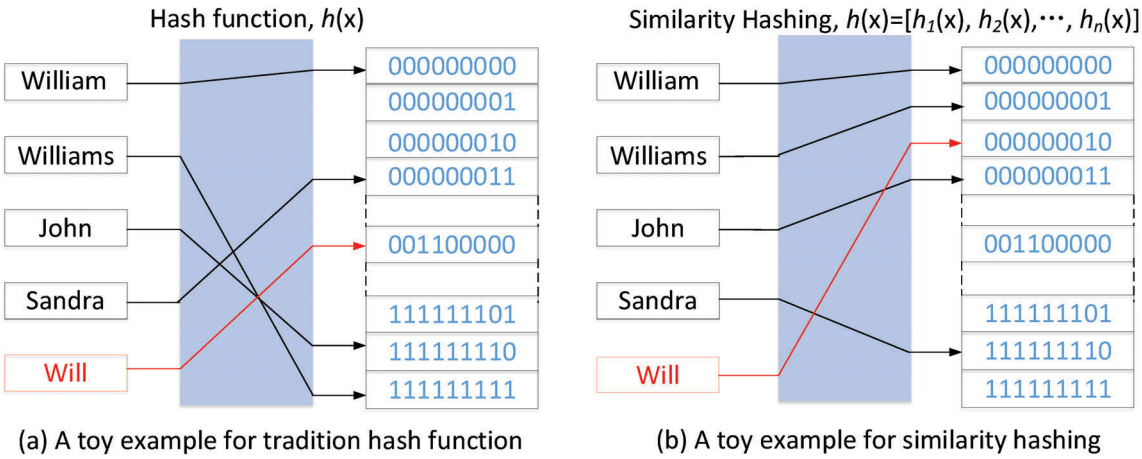


Fig. 3.2 Comparison between a traditional hash function and the similarity hashing scheme.

The aforementioned problem can be modelled as fundamental problems in the database, the nearest neighbor search problem (also known as similarity search) and the approximate nearest neighbor search problem.

Definition 3 (Nearest Neighbor Search (NNS)) *Given a query object* $\mathbf{q}$*, the goal is to find an object* $\mathrm{NN}(\mathbf{q})$*, called nearest neighbor, from a set of objects* $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$ *so that* $\mathrm{NN}(\mathbf{q}) = \arg\min_{\mathbf{x} \in X} \mathrm{dist}(\mathbf{q}, \mathbf{x})$*, where* $\mathrm{dist}(\mathbf{q}, \mathbf{x})$ *is a distance between* $\mathbf{q}$ *and* $\mathbf{x}$*.*

Note that the distance measure involved in NNS can employ the Jaccard distance, the Euclidean distance, the Cosine distance, etc. The nearest neighbor search problem is easily solved in the case of low-dimensional data. However, in high-dimensional data, various phenomena arise, which do not occur in low-dimensional settings. The common problem in high-dimensional spaces is that when the dimensionality increases, the volume of the space increases very quickly so that the available data become sparse. We call the phenomena in the high-dimensional space as the curse of dimensionality. Consequently, some efficient algorithms for NNS in low-dimensional spaces will spend high computational cost in the high-dimensional settings. To this end, a large amount of research has been devoted to approximating the nearest neighbor search problem.

Definition 4 (Approximate Nearest Neighbor Search (ANNS)) *Given a query object* $\mathbf{q}$*, the goal is to find an object an object* $\mathrm{ANN}(\mathbf{q})$ *so that* $\mathrm{dist}(\mathbf{q}, \mathbf{x}) \leq (1+\varepsilon)\mathrm{dist}(\mathbf{q}, \mathbf{x})^*$*, where* $\mathrm{dist}(\mathbf{q}, \mathbf{x})^*$ *denotes the true nearest neighbor.*

In order to solve the ANNS problem, similarity hashing is proposed [130]. The hashing technique aims to map the similar data objects to the close and even the same data points represented as the hash codes with a higher probability than the dissimilar ones. Consequently, ANNS can be efficiently and accurately performed in a small number of compact data representation. As shown in Figure 3.2(b), two names, "William" and "Williams", and the query "Will" are mapped to the three similar hash codes, "000000000", "000000001" and "000000010", respectively. Furthermore, the efficiency can be improved from two aspects: 1) ANNS can be limited to a small range, that is, the Hamming distance is smaller than or equal to 2, so that the searched candidates are a small subset of the whole database; 2) ANNS can be efficiently conducted based on the compact hash code space where the size of the hash code is much smaller than the size of the original data in the database.

Currently, according to whether the hash techniques depend on data distributions, they can be categorized into *Learning to Hash* (i.e., Data-Dependent Hashing) and *Locality Sensitive Hashing* (Data-Independent Hashing). We categorize the similarity hashing in Figure 3.3.

3.2.1 Learning to Hash

Learning to Hash, as the name implies, is to learn data-dependent and task-specific hash functions by employing the machine learning techniques. The goal of learning to hash is to achieve better performance in the tasks by closely fitting the data distribution in the feature space and preserving neighborhood structure information between data objects. Formally, given a data object $\mathbf{x}$, we aim to learn a compound hash function, $\mathbf{y} = \mathbf{h}(\mathbf{x})$, which maps the data object $\mathbf{x}$ to a compact hash code $\mathbf{y}$. Based on the hash code, we can conduct an effective and efficient approximation of the true nearest neighbor search result.

Currently, based on the level of supervision in the machine learning paradigms, learning to hash approaches can be categorized into three subgroups: unsupervised, supervised and semi-supervised, and they have been widely applied in computer vision [106, 131], machine learning [61, 75], and deep learning [77, 78], etc. The unsupervised learning to hash methods design the compact binary hash codes by exploiting just the unlabelled data, and attempt to preserve the data similarity, e.g, spectral hashing [137], graph hashing [79], manifold hashing [105], isotropic hashing [60], iterative quantization hashing [37], angular quantization hashing [36], etc. One of the classical methods is spectral hashing [137], which aims to obtain a good hash function such that similar data objects are mapped to as short and similar hash codes as possible based on the Hamming distance. The algorithm gives a hard criterion for a good hash code and points out that it is equivalent to a particular form of graph partitioning. Consequently, a spectral relaxation is adopted to obtain an eigenvector

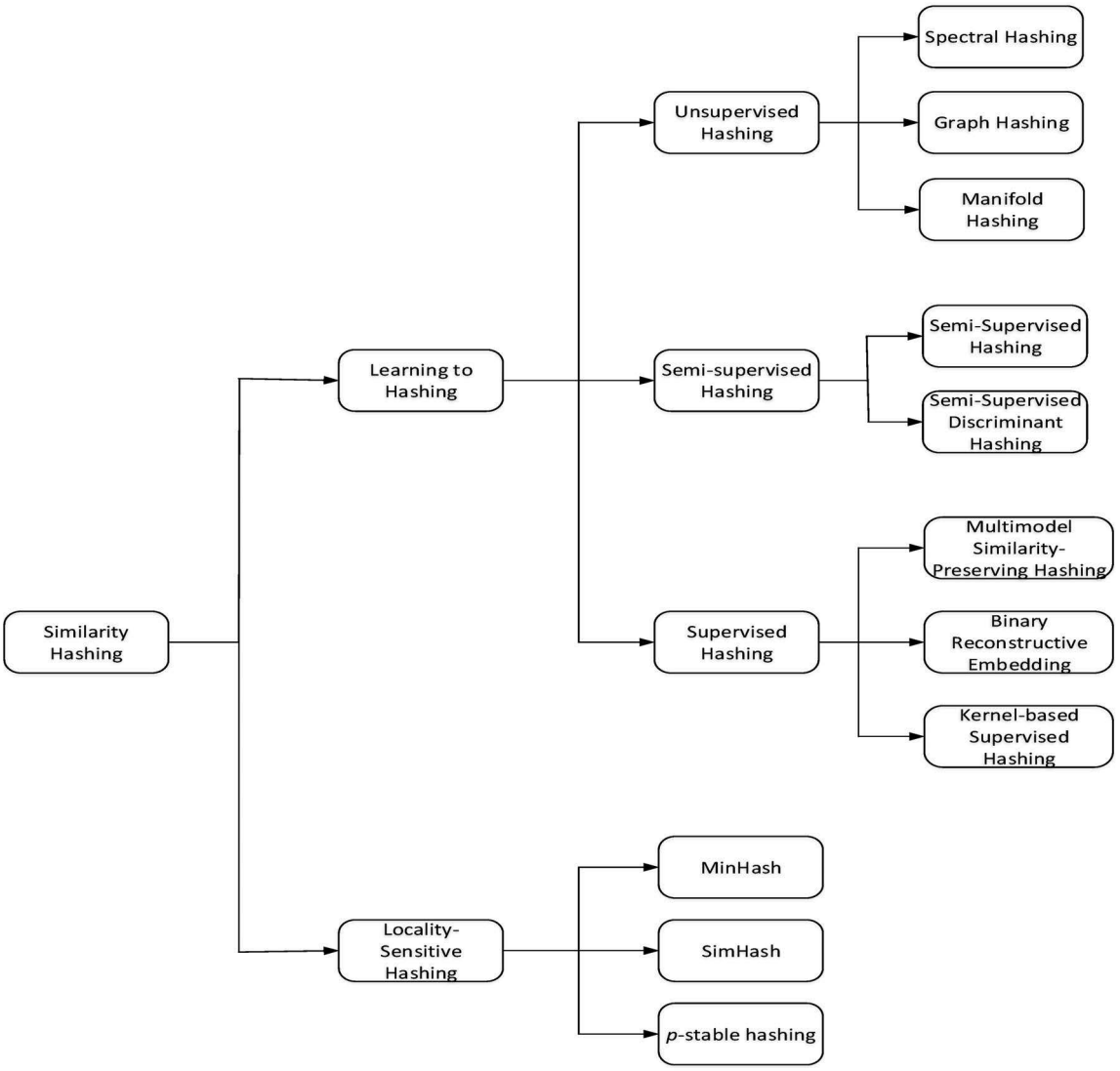


Fig. 3.3 Categorization for Similarity Hashing.

solution. Subsequently, some learning to hashing methods extend the spectral hashing. For example, kernelized spectral hashing [44] uses kernels to explicitly define the hash function; hypergraph spectral hashing [81, 157] extends the spectral hashing from an ordinary graph to a hypergraph, and further replaces the graph Laplacian in spectral hashing with hypergraph Laplacian; sparse spectral hashing [104] integrates sparse principal component analysis and boosting similarity sensitive hashing into spectral hashing.

By contrast, semi-supervised hashing methods employ both labelled data and unlabelled data to train the model, e.g., semi-supervised hashing [36], label-regularized max-margin partition algorithm [90], semi-supervised discriminant hashing [59], bootstrap sequential projection learning for semi-supervised nonlinear hashing [90] and semi-supervised topol-

ogy preserving hashing [154], etc. One of the most popular approaches is semi-supervised Hashing [36], which extends spectral hashing into the semi-supervised case. To this end, some pairs of data objects are labelled as the same semantic concept while some pairs are labelled as different ones.

Supervised hashing approaches only utilize labelled data to train the model, and the current methods exploit various supervised learning paradigms, e.g., kernel learning, metric learning and deep learning, to learn binary hash codes, for example, multimodal similarity-preserving hashing [86], semantic correlation maximization [153], boosting similarity sensitive coding [103], binary reconstructive embedding [61], kernel-based supervised hashing [80], linear discriminant analysis based hashing [115], etc. The supervised hashing methods can adopt more information to improve performance than the unsupervised hashing ones, but the former generally encounters the problem of the training inefficiency.

3.2.2 Locality Sensitive Hashing

Learning to hash aims to closely fit the data distribution in the feature space for preserving as much data similarity as possible, and thus the methods depend on the specific data. By contrast, the *Locality Sensitive Hashing* (LSH) [48] is a non-learning scheme for the ANNS problem. The scheme aims to map the similar objects to the similar and even the same hash values by employing a family of randomized hash functions. Formally, given a data object $\mathbf{x}$ and a set of hash functions $\mathbf{h} = [h_1(\cdot), h_2(\cdot), \cdots, h_n(\cdot)]$, we obtain a hash value as the hash code for $\mathbf{x}$, $\mathbf{h}(\mathbf{x}) = [h_1(\mathbf{x}), h_2(\mathbf{x}), \cdots, h_n(\mathbf{x})]$.

LSH is used to approximate similarity (or distance) measures. So far various LSH methods have been proposed based on l_p distance, angle-based distance, hamming distance and Jaccard similarity, which have been widely applied in statistics [26], computer vision [57, 156], multimedia [151], data mining [145, 150], machine learning [91, 101], natural language processing [99, 124], etc. Interestingly, the term "Locality Sensitive Hashing" (LSH) was introduced in 1998 [48], but the first LSH method, MinHash [10], which is used to approximate the Jaccard similarity between two binary sets, was proposed in 1997 for near-duplicate web detection and clustering. The thesis extends the well-known method to structured data, and we will introduce it in details in Section 3.3. LSH with p-stable distribution [24] is designed for l_p norm $||\mathbf{x}_i - \mathbf{x}_j||_p$, where $p \in (0, 2]$. This scheme employs the property of stable distributions, e.g., Cauchy distribution ($p = 1$) and Gaussian distribution ($p = 2$), to estimate the corresponding l_p norm. Andoni and Indyk [6] extend LSH with p-stable distribution in [24] into the multi-dimensional space by randomly projecting data points into $\mathbb{R}^t$. Dasgupta et al. [23] fast estimate l_2 distance between two vectors using randomized Hadamard transforms in a non-linear settting. The SimHash [16], as the

classical angle-based LSH, is designed to approximate cosine distance between vectors representing data points. In the approach, the vectors are projected into the normal vector of a random hyperplane, and the hash values (0 or 1) are either side of the hyperplane on which the vector lies. Manku et al. [85] practically implement SimHash and propose an algorithmic technique to judge whether a document represented as D-bit fingerprints is different from a given document represented as fingerprints with the same bit number in at most k bit-positions of the fingerprints where k is small. Ji. et al. [51] improve SimHash by partitioning the random projections into different groups and orthogonalizing them in each group. Consequently, their results in each group can be combined together. Kulis et al. [62, 63] extend the angle between vectors in [16] into the angle between kernel functions, while multiple kernel LSH approaches [132, 133] generate hash functions by adopting multiple kernels, each of which is assigned to the same number of bits. Xia et al. [144] present a boosted version of multi-kernel LSH. Instead of assigning the same number of bits in each kernel, the method in [144] automatically allocates various number of bits via the boosting scheme. The above angle-based LSH approaches are used to retrieve points (or vectors) that are close to a query point (or vectors), while hyperplane hashing aims to retrieve points that are close to a query hyperplane [50, 127]. For binary vectors, the LSH method for the Hamming distance is proposed in [48], where one LSH function randomly samples one index from the binary vector. Besides, Gorisse et al. [38] propose a LSH method for the χ^2 distance between two vectors.

3.3 MinHash

MinHash [10, 11] is one of the most popular algorithms in the LSH family. It is used to approximate the Jaccard similarity between two binary sets, and has been widely applied in the bag-of-words model, for example, duplicate webpage detection [29, 45], webspam detection [55, 125], text mining [19, 58], large-scale machine learning systems [73, 74], content matching [94], etc. In this section, we will introduce MinHash and its variants in details.

We illustrate the working process of the MinHash algorithm in Fig 3.4. In the three tables, the first column denotes the elements and the last two columns represent two sets, S_1 and S_2, respectively. In the last two columns, "1" denotes that the set contains the corresponding the element, and "0" otherwise. In the toy example, we show two sets, $S_1 = \{1,3,6,7\}$ and $S_2 = \{1,4,7\}$, and two independent random permutations (or hash functions), $h_1 = [2,5,1,6,3,4,7]$ and $h_2 = [3,1,7,2,5,4,6]$. Consequently, h_1 returns the minimum hash values, 1 for S_1 and 1 for S_2, and h_2 does the minimum hash values, 3 for S_1

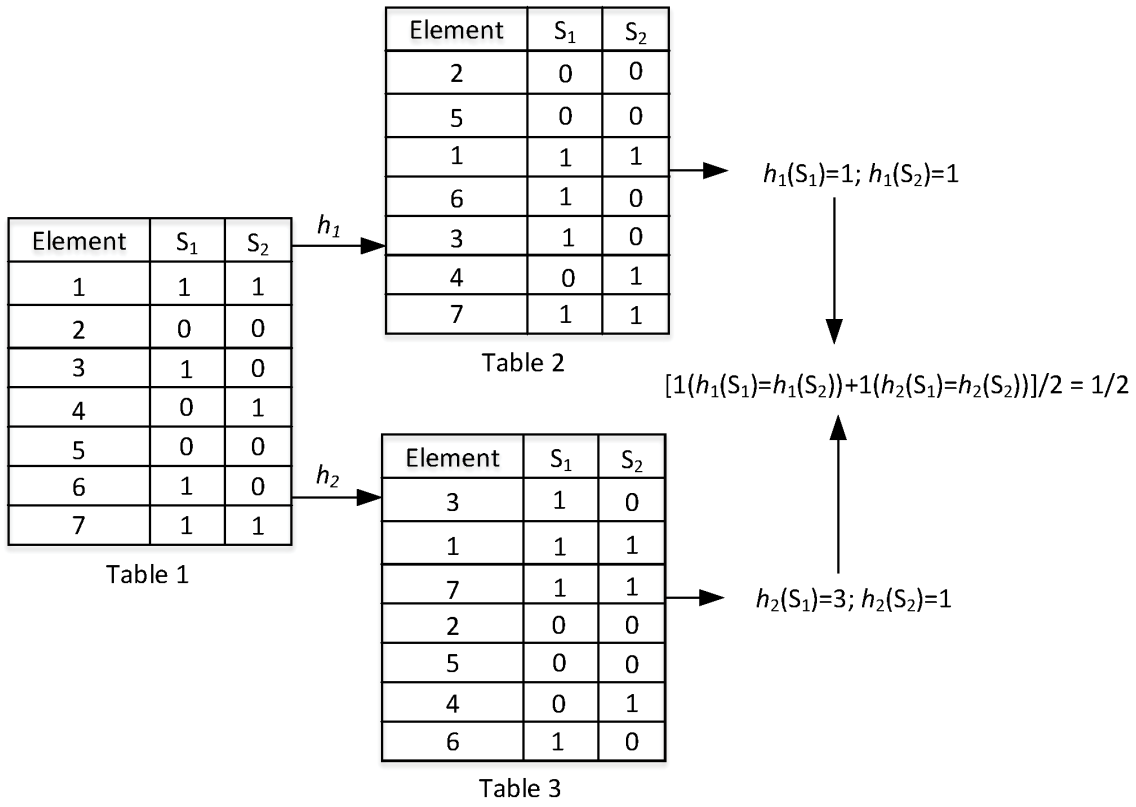


Fig. 3.4 An illustration for MinHash.

and 1 for S_2, that is, the MinHash values for S_1 and S_2 are $[1,3]$ and $[1,1]$, respectively. Finally, we obtain the estimated similarity between S_1 and S_2, $\hat{J}(S_1,S_2) = \frac{\sum_{k=1}^{2}\mathbf{1}(h_k(S_1)=h_k(S_2))}{2} = \frac{1}{2}$.

Furthermore, Shrivastava and Li [112] give theoretical and experimental evidence that the MinHash algorithm outperforms SimHash in document analysis where data are represented as binary sets. Also, many variations of the MinHash algorithm have been subsequently proposed to improve the efficiency because the MinHash algorithm needs K random permutations. To this end, Min-Max Hash [52] generates K hash values by employing only $\frac{K}{2}$ random permutations and taking the smallest as well as the largest values of each permutation, but it is still able to obtain an unbiased estimator. Moreover, some methods use only one permutation to speed up the MinHash algorithm. For example, Conditional Random Sampling [68, 70] achieves better estimators than random projections by permutating only once and taking the k smallest nonzeros. By contrast, One Permutation Hashing [74] permutates only once, breaks the permutation into K bins and then concatenates the smallest nonzero value in each bin as a fingerprint. Unfortunately, the method in [74] gives rise to the issue of empty bins, and subsequently, Shrivastava and Li solves the problem and supplies an unbiased estimator in [113]. On the other hand, b-bit MinHash [69, 71, 72]

not only remarkably improves the storage efficiency and provides an unbiased estimator by storing only the lowest b bits of each MinHash value (e.g., $b = 1$ or 2) in the case of K permutations, but also can be integrated with linear learning algorithms, e.g., linear SVM and logistic regression, to solve large-scale and high-dimensional learning tasks.

3.4 Structured Hashing

The hashing technique, which is a very powerful tool to sketch high-dimensional data, can efficiently map the high-dimensional data to a fixed number of dimensions as an (unbiased) estimator of the original high-dimensional feature vectors. Although most existing hashing algorithms are applied to vectors or sets, some works are designed for structured data.

Some works aim to hash tree-structured data. The hierarchical MinHash algorithm [35] proceeds from bottom to top of the tree, where the information of labels obtained by fingerprinting is propagated to an upper level of the tree. The approach simply considers lower-level MinHash values as an element of the current-level set for the MinHash algorithm, which completely ignores the similarity information between lower-level sets. By contrast, in [19], the authors improve the algorithm in [35] by reorganizing lower-level MinHash values into a number of sampling-sets for the MinHash algorithm. Consequently, the lower-level similarities can be propagated to higher-levels in probability. In [121], the authors proposed a subtree pattern called embedded pivot substructure, which is comprised of two nodes and their lowest common ancestor in the tree. Naturally, a tree can be represented as a bag of such embedded pivots, and subsequently, the MinHash algorithm sketches the bag. However, the approach can hardly be scaled up to deal with large-scale data sets because it spends a computational complexity of $O(n^2)$, where n is the number of nodes in the tree, in acquiring all embedded pivots in the tree.

On the other hand, some graph hashing algorithms have been proposed to characterize substructures in the graphs. In [32], the authors identified large dense subgraphs in massive graphs by a variant of the MinHash algorithm, where each hash function is generalized to return multiple elements for preserving more neighboring information, while in the standard MinHash scheme, each hash function returns only one element. In [7], the authors estimated the local number of triangles in large graphs by a slight modification of the MinHash algorithm, which accelerates the standard method. Some works employ two random hashing schemes: In [2], the first hashing scheme is used to reduce the size of edges, and the second one summarizes the frequent patterns of co-occurrence edges; in [18], the first hashing scheme compresses a graph into a small graph for clique detection, and the second one maps unlimitedly emerging cliques into a fixed-size clique set.

Additionally, some graph hashing approaches first extract substructures and then hash them for graph mining tasks. For example, [20, 65, 143] extract substree structures in the graph and approximately count substructures by the hashing technique. Consequently, the graph is represented as a feature vector for classification, where each dimension comprises a set of substructures and the value of each dimension is related to the estimated number of the substructures. By contrast, the MinHash Fingerprints algorithm [122] first enumerates all paths in the graph as a pattern set for representing each graph and then employ the MinHash algorithm to fingerprint.

Chapter 4

Two MinHash-based Algorithms for Weighted Sets

In this chapter, we propose two MinHash-based algorithms for weighted sets. Specifically, the algorithms transform each weighted set into a low-dimensional hash code, and subsequently, we can approximate the similarity between the weighted sets based on the compact hash codes.

4.1 Introduction

Nowadays, data are growing explosively on the Web. In 2008, Google processed 20PB data every day [102]; in 2012, Google received more than 2 million search queries per minute; while in 2014 this number had been more than doubled [40, 116]. Social networking services also have to face the data explosion challenge. More than 500TB of data need to be processed in Facebook every day [117] and 7 million check-in records are produced in Foursquare every day [148] – Big data have been driving data mining research in both academia and industry [25, 97]. A fundamental research that underpins many high-level applications is to efficiently compute similarities (or distances) of data. Based on the data similarity, one can further conduct information retrieval, classification, and many other data mining tasks. However, the standard similarity computation has been incompetent for big data due to the "3V" nature (volume, velocity and variety). For example, in text mining, it is intractable to enumerate the complete feature set (e.g., over 10^8 elements in the case of 5-grams in the original data [97]). Therefore, it is urgent to develop efficient yet accurate similarity estimation algorithms.

A typical solution to the aforementioned problem is to approximate data similarities using a family of Locality Sensitive Hashing (LSH) techniques [48]. By adopting a collection of hash functions to map similar data objects to the same hash code with higher probability than dissimilar ones, LSH is able to approximate certain similarity (or distance) measures. One can thus efficiently and, in many cases, unbiasedly calculate the similarity (or distances) between objects. Many LSH schemes have been successively proposed, e.g., MinHash for estimating the Jaccard similarity [11], SimHash for estimating angle-based distance [16, 85], and LSH with p-stable distribution for estimating l_p distance [24]. In particular, MinHash has been widely used for approximating the similarity of documents which are usually represented as sets or bags-of-words. Recently, some variations of MinHash have further improved its efficiency. For example, b-bit MinHash [69] and odd sketches [89] remarkably improve the storage efficiency by storing only b bits of each hash value; while one-permutation MinHash [74, 113] employs only one permutation to reduce the computational workload.

Although the set similarity can be efficiently estimated based on the standard MinHash scheme and its variations, the target data are restricted to binary sets. However, in most real-world scenarios, weighted sets are more common. For example, a *tf-idf* value is assigned to each word to represent its importance in a collection of documents. As MinHash treats all the elements in a set equally for computing the Jaccard similarity, it cannot handle weighted sets properly. To address the limitation, weighted MinHash algorithms have been explored to approximate the generalized Jaccard similarity [43], which is used for measuring the similarity of weighted sets. Roughly speaking, existing works on weighted MinHash algorithms can be classified into *quantization-based* and *sampling-based* approaches.

Quantization-based methods quantize each weighted element into a number of distinct and equal-sized subelements. The resulting subelements are treated equally just like independent elements in the universal set. One can simply apply the standard MinHash scheme to the collection of subelements. For integer weighted sets, it is natural to treat each "1" as a subelement; for real-value weighted sets, one has to choose a small value "Δ" to quantize the element. Despite the feasibility, the quantization process explicitly increases the size of the universal set which significantly increases the computational workload, because each subelement is independently permutated according to the definition of MinHash. In order to improve the efficiency of weighted MinHash (WMH), Gollapudi and Panigraphy [34] proposed an idea of "active index" to skip many invalid subelements and developed a more efficient integer weighted MinHash (IWMH) algorithm. However, the algorithm is still inefficient for dealing with real-value weighted sets because each weight must be transformed

into integer weight by multiplying a large constant, which dramatically expands the universal set with numerous quantized subelements.

To avoid computing hash values for all the subelements, researchers have resorted to sampling-based methods. In [110], a uniform sampling algorithm is proposed, which strictly complies with the definition of the generalized Jaccard similarity. However, this algorithm requires knowing the upper bound of each element in the universal set in advance, which makes it impractical in real-world applications. In [21], a sampling method is derived through the distribution of the minimum of a set of random variables for integer weighted sets, which results in a biased estimator. Later, Consistent Weighted Sampling (CWS) [41, 84] and Improved CWS (ICWS) [49] are proposed to remarkably improve the efficiency of weighted MinHash by introducing the notion of "active index" [34] into real-value weighted elements (the "active indices" on a weighted element are independently sampled as a sequence of subelements whose hash values monotonically decrease [138]). Thus far, ICWS [49], as an efficient and unbiased estimator of the generalized Jaccard similarity, is recognized as the state-of-the-art. Recently, [67] approximates ICWS by simply discarding one component of the MinHash values.

The mystery of ICWS [49] is that it implicitly constructs an exponential distribution for each real-value weighted element using only two "active indices". The algorithm is so simple that the working mechanism is hidden deeply. In this chapter, we parallel explore the theory of the ICWS algorithm from two perspectives: 1) Although ICWS is theoretically proved to comply with the uniformity of the MinHash scheme, we observe that the implicit quantization adopted by the algorithmic implementation of ICWS ([49] does not use quantization in the proof) is conducted on the logarithm of a weight instead of the original weight – *This violates the definition of the MinHash scheme because the quantization results in a collection of unevenly quantized subelements*. Therefore, ICWS does not strictly comply with the MinHash scheme. 2) In the establishment of the exponential distribution for each real-value weighted element, the minimum of the collection of exponential variables of all the elements yields one of the two components of the real-value weighted MinHash code (the other component can be easily obtained), meanwhile complying with the uniformity of the CWS scheme – *This enables ICWS to produce MinHash code for a real-value weight with computational complexity that is independent of the number of quantized subelements*.

In order to address the first problem, we propose the Canonical Consistent Weighted Sampling (CCWS) algorithm, which not only achieves the same theoretical computational complexity as ICWS but also strictly complies with the definition of the MinHash scheme. To this end, CCWS performs the implicit quantization on the original weight rather than any non-linearly transformed weight. Since the change of quantization breaks the underlying

mechanism between the “active indices” and the minimum hash values, we reestablish the mechanism by discovering a different set of random variables that can bridge the “active indices" and the minimum hash values. We theoretically prove the uniformity and consistency of the proposed CCWS algorithm for the real-value weighted MinHash algorithm.

For the second observation, we propose the Practical Consistent Weighted Sampling (PCWS) algorithm. By transforming the original form of ICWS into a different but mathematically equivalent expression, we find some interesting properties that inspire us to further make ICWS more practical. Consequently, our PCWS algorithm is simpler and more efficient in both space and time complexities compared to ICWS. Furthermore, we theoretically prove the uniformity and consistency of the PCWS algorithm for the real-value weighted MinHash algorithm.

We also conduct extensive empirical tests on a number of real-world text data sets to compare the two proposed algorithms, i.e., CCWS and PCWS, and the state-of-the-arts methods. First, we compare CCWS with IWMH and ICWS on document classification in terms of accuracy and efficiency. For accuracy, CCWS achieves almost the same classification performance as IWMH, which strictly complies with the definition of MinHash; for efficiency, CCWS runs much faster than IWMH while enjoys the similar efficiency as ICWS. Therefore, CCWS is a real MinHash algorithm with very high computational efficiency. Second, we compare the proposed PCWS algorithm and the state-of-the-arts for classification and retrieval. For accuracy, PCWS obtains the same (and even better) classification and retrieval performance as ICWS; for efficiency, PCWS performs more efficiently than ICWS.

In summary, our contributions are as follows:

1. We review the evolution of MinHash algorithms for weighted sets, from the integer weighted MinHash approach to the CWS scheme, and uncover inherent connections between them.

2. We show that ICWS [49], the state-of-the-art real-value weighted MinHash algorithm, is not a real MinHash algorithm because it violates the uniformity of the MinHash scheme.

3. We establish an alternative mechanism of using random variables to bridge the “active indices" and the minimum hash values. We propose a new algorithm named Canonical Consistent Weighted Sampling (CCWS), which strictly meets the definition of MinHash as IWMH and achieves the same efficiency as ICWS.

4. We transform the original form of ICWS [49] into a simpler and easy-to-understand expression, uncovering the working mechanism of ICWS.

5. We propose the PCWS algorithm, which is mathematically equivalent to ICWS and preserves the same theoretical properties as ICWS.

The remainder of the chapter is organized as follows: Section 4.2 briefly introduces the evolution of weighted MinHash algorithms. We review the state-of-the-art algorithm, ICWS [49] in Section 4.3. Then, we present our CCWS algorithm and our PCWS algorithm in Section 4.4 and Section 4.5, respectively. The experimental results are shown in Section 4.6. We review the related work of the weighted MinHash algorithm in Section 4.7 and the paper is concluded in Section 4.8.

4.2 Preliminaries

In this section, we first give some notations which will be used through the paper; then we introduce the related work of MinHash and weighted MinHash.

Given a universal set $\mathscr{U} = (U_1, U_2, \cdots, U_n)$ and its subset $\mathscr{S} \subseteq \mathscr{U}$, if for any element $S_k \in \mathscr{S}$, $S_k = 1$, then we call $\mathscr{S}$ a binary set; if for any element $S_k \in \mathscr{S}$, $S_k \geq 0$, then we call $\mathscr{S}$ a weighted set. A typical example of universal set is the dictionary of a collection of documents, where each element corresponds to a word.

For hashing a binary set $\mathscr{S}$, a MinHash scheme assigns a hash value to each element S_k, $h : k \mapsto v_k$. By contrast, for hashing a weighted set, there is a different form of hash function: $h : (k, y_k) \mapsto v_{k,y}$, where $y_k \in [0, S_k]$. A random permutation (or sampling) process returns the first (or uniformly selected) k from a binary set or (k, y_k) from a weighted set. If the set is sampled D times, we will obtain a fingerprint with D hash values.

4.2.1 MinHash

Definition 5 (MinHash [11]) *Given a universal set $\mathscr{U}$ and a subset $\mathscr{S} \subseteq \mathscr{U}$, MinHash is generated as follows: Assuming a set of D hash functions (or D random permutations), $\{\pi_d\}_{d=1}^{D}$, where $\pi_d : \mathscr{U} \mapsto \mathscr{U}$, are applied to $\mathscr{U}$, the elements in $\mathscr{S}$ which have the minimum hash value in each hash function (or which are placed in the first position of each permutation), $\{\min(\pi_d(\mathscr{S}))\}_{d=1}^{D}$, would be the MinHashes of $\mathscr{S}$.*

MinHash [10, 11] is an approximate algorithm for computing the Jaccard similarity of two sets. It has been proved that the probability of two sets, $\mathscr{S}$ and $\mathscr{T}$, to generate the same MinHash value (hash collision) is exactly equal to the Jaccard similarity of the two sets:

$$\Pr\big(\min(\pi_d(\mathscr{S})) = \min(\pi_d(\mathscr{T}))\big) = J(\mathscr{S}, \mathscr{T}) = \frac{|\mathscr{S} \cap \mathscr{T}|}{|\mathscr{S} \cup \mathscr{T}|}.$$

The Jaccard similarity is simple and effective in many applications, especially for document analysis based on the bag-of-words representations [112].

From the above MinHash scheme we can see that all the elements in $\mathscr{U}$ are considered equally because all the elements can be mapped to the minimum hash value with equal probability. To sample a weighted set based on the standard MinHash scheme, the weights, which indicates different importance of each element, will be simply replaced with 1 or 0, which in turn leads to a serious information loss.

4.2.2 Weighted MinHash

In most real-world scenarios, weighted sets are more commonly seen than binary sets. For example, a document is usually represented as a tf-idf set. In order to reasonably compute the similarity of two weighted sets, the generalized Jaccard similarity was introduced in [43]. Considering two weighted sets, $\mathscr{S}$ and $\mathscr{T}$, the generalized Jaccard similarity is defined as

$$generalizedJ(\mathscr{S},\mathscr{T}) = \frac{\sum_k \min(S_k, T_k)}{\sum_k \max(S_k, T_k)}.$$

Based on the generalized Jaccard similarity, the weighted MinHash (WMH) [34] adopts the idea of quantization for the weighted set. If $\mathscr{S}$ contains real-value weights, WMH first transforms the weights into integer ones by multiplying a constant C and taking their rounding values. Suppose a weighted set $\mathscr{S}$ has been quantized and take the k-th element of $\mathscr{S}$ for example: After it is explicitly quantized into S_k distinct subelements with all the weights being 1, we adopt a hash function $h : (k, y_k) \mapsto v_{k,y}$, where $y_k \in \{1, \ldots, S_k\}$, to assign a hash value to each subelement. Through quantization, WMH can unbiasedly estimate the generalized Jaccard similarity of weighted sets; however, it is costly in terms of both time and space, whose complexities are proportional to the total number of subelements $O(\sum_k S_k)$. This is the immediate consequence of the expansion of the universal set $\mathscr{U}$ with all the distinct subelements, which are required to be independently assigned a hash value.

Actually, WMH can be broken down into two steps: (1) For the k-th element, assign a hash value to each subelement (k, y_k) and finds (k, y_k^*) that has the minimum hash value; (2) Find (k^*, y_k^*) that has the minimum hash value in $\{(k, y_k^*)\}_{k=1}^n$, where n denotes the number of elements in the original universal set. This two-step treatment inspires the idea of "active index" in the following subsection.

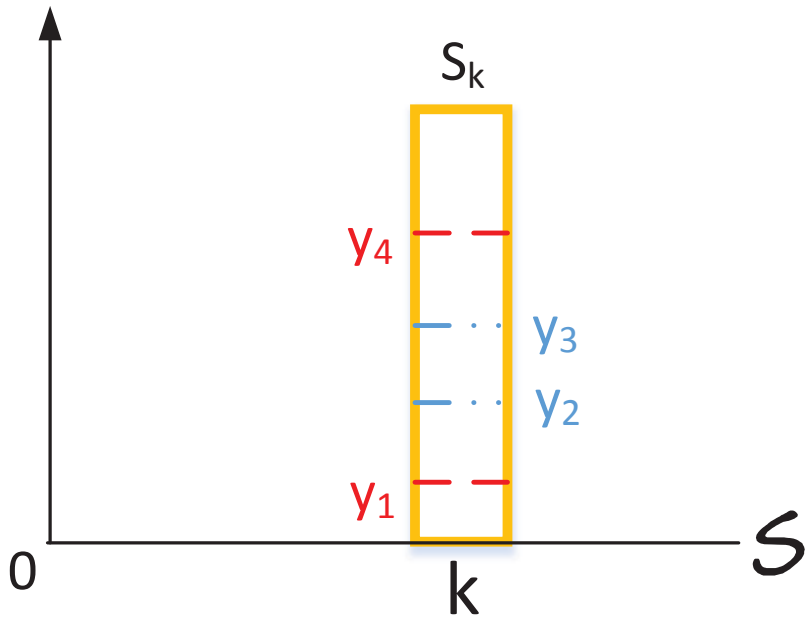


Fig. 4.1 An illustration of “active indices”. The red dashed lines represent “active indices” whose hash values are monotonically decreasing from bottom to top; while the blue dashed lines are non-active subelements, where $v_{k,y_4} < v_{k,y_1} < \{v_{k,y_2}, v_{k,y_3}\}$. IWMH proceeds traversing “active indices" from bottom to top by skipping many non-active subelements.

4.2.3 Integer Weighted MinHash

WMH based on quantization has been able to unbiasedly estimate the generalized Jaccard similarity, but its dramatically increased computational cost becomes a burden in practical use. To avoid computing the hash value for each subelement, [34] proposes an efficient integer weighted MinHash (IWMH) algorithm, which is able to accelerate computation by skipping much unnecessary hash value computation[1] by employing the idea of “active index”.

IWMH successively traverses all the subelements in the integer or quantized weighted set $\mathscr{S}$. Take S_k in Figure 4.1 as an illustration: Computation and comparison of hash values for each subelement proceeds from bottom to top. Assuming that for four subelements, $(k,y_1),(k,y_2),(k,y_3)$ and (k,y_4), we have $v_{k,y_4} < v_{k,y_1} < \{v_{k,y_2}, v_{k,y_3}\}$. When the algorithm starts from y_1, in the following steps, it may skip subelements y_2 and y_3 and directly reaches y_4, because the blue subelements with larger hash values than y_1 are useless when one only wants to keep the minimum hash value v_{k,y_k^*} for the k-th element. Naturally, “active indices”

[1] As WMH does, the real-value weights can also be transformed into integers by multiplying a constant and rounding the result as a preprocess.

can be defined as a sequence of subelements whose hash values are monotonically decreasing from bottom to top. Furthermore, due to the fact that the hash value of the following "active index" (e.g., y_4) after the previous one (e.g., y_1) is uniformly distributed on $[0, v_{k,y_1}]$, the number of subelements between two adjacent "active indices" (e.g., y_1 and y_4) conforms to a geometric distribution with the parameter being v_{k,y_1}. Based on the above observation, IWMH is allowed to directly skip the subelements between two adjacent "active indices" by sampling a skipping interval from the geometric distribution. In this way, IWMH can recursively sample all the "active indices" from bottom to top to obtain (k, y_k^*) with the minimum hash value. It is easy to see that y_k^* is the largest "active index" in $\{1, \ldots, S_k\}$.

As IWMH does not change the underlying uniform distribution of the minimum hash value among the subelements in each element of the weighted set, it is still an unbiased estimator of the generalized Jaccard similarity. By skipping non-active subelements, IWMH drastically reduces the computational complexity from $O(\sum_k S_k)$ to $O(\sum_k \log S_k)$. However, for real-value weighted sets, IWMH still needs an explicit quantization, which brings tremendous overhead in both time and space.

4.3 Review of Improved CWS

In order to remedy the drawback that IWMH cannot directly deal with real-value weighted sets without an explicit quantization, [84] proposes the Consistent Weighted Sampling (CWS) algorithm, which can be directly applied to the real-value weighted sets and avoid traversing all "active indices". To the best of our knowledge, Improved CWS (ICWS) [49] is remarkable in both theory and practice and considered as the state-of-the-art [67].

Definition 6 (Consistent Weighted Sampling [84]) *Given a weighted set $\mathscr{S} = \{S_1, \ldots, S_n\}$, where $S_k \geq 0$ for $k \in \{1, \ldots, n\}$, Consistent Weighted Sampling (CWS) produces a sample $(k, y_k) : 0 \leq y_k \leq S_k$, which is uniform and consistent.*

- **Uniformity:** The subelement (k, y_k) should be uniformly sampled from $\bigcup_k (\{k\} \times [0, S_k])$, i.e., the probability of selecting the k-th element is proportional to S_k, and y_k is uniformly distributed on $[0, S_k]$.
- **Consistency:** Given two non-empty weighted sets, $\mathscr{S}$ and $\mathscr{T}$, if $\forall k, T_k \leq S_k$, a subelement (k, y_k) is selected from $\mathscr{S}$ and satisfies $y_k \leq T_k$, then (k, y_k) will also be selected from $\mathscr{T}$.

CWS has the following property

$$\Pr[\mathrm{CWS}(\mathscr{S}) = \mathrm{CWS}(\mathscr{T})] = \mathit{generalizedJ}(\mathscr{S}, \mathscr{T}).$$

Recall that IWMH accelerates computation by sampling "active indices" from quantized subelements, which have a weight of $\Delta = 1$. CWS further extends this idea to the real-value weights by allowing $\Delta \to 0$. However, it is impossible to traverse all "active indices" in order to find the largest[2] y_k satisfying $y_k \leq S_k$ because the number of "active indices" tends to be infinite when $\Delta \to 0$. To conquer the problem, ICWS [49] introduces another "active index", z_k, which is the smallest active index greater than S_k (i.e., $S_k \in [y_k, z_k]$), where $S_k \leq z_k < +\infty$. By constructing relationships between the two bounding active indices, y_k and z_k, as well as the relationships among the two active indices and the minimum hash value a_k and the known weight S_k, ICWS is able to locate y_k in $[0, S_k]$ within a constant time.

In the following we briefly review how to deduce the underlying relationships among $\{y_k, z_k, S_k, a_k\}$ to meet the two conditions of the CWS scheme, uniformity and consistency. Firstly we note that the Exponential distribution has an important and interesting property about the distribution of the minimum of exponential random variables: Let $X_1, \ldots, X_n$ be independently exponentially distributed random variables, with the parameters being $\lambda_1, \ldots, \lambda_n$, respectively, then we have

$$\Pr(X_k = \min\{X_1, \ldots, X_n\}) \quad = \quad \frac{\lambda_k}{\lambda_1 + \ldots + \lambda_n}. \tag{4.1}$$

Therefore, one can employ this property to facilitate the uniformity for the CWS scheme – If each hash value $a_{k'}$ of the k'-th element conforms to an Exponential distribution parameterized with its corresponding weight, that is, $a_{k'} \sim \text{Exp}(S_{k'})$, the minimum hash value a_k will be proportional to S_k,

$$\Pr(a_k = \min\{a_1, \ldots, a_n\}) \quad = \quad \frac{S_k}{\sum_{k'} S_{k'}}. \tag{4.2}$$

In addition, note that k and y_k are mutually independent, which means that a_k and y_k are mutually independent as well. Formally, the condition of uniformity in terms of (a_k, y_k) can be expressed as

$$\text{pdf}(y_k, a_k) \quad = \quad \frac{1}{S_k}(S_k \mathrm{e}^{-S_k a_k}) = \text{pdf}(y_k)\text{pdf}(a_k), \tag{4.3}$$

where $y_k \sim \text{Uniform}(0, S_k)$ and $a_k \sim \text{Exp}(S_k)$.

In order to meet the requirement that y_k is uniformly distributed in $[0, S_k]$, ICWS introduces the following equation

$$\ln y_k \quad = \quad \ln S_k - r_k b_k, \tag{4.4}$$

[2] By an abuse of notation we let y_k denote the largest active index on the k-th element S_k in the remainder of the paper.

where $r_k \sim \text{Gamma}(2,1)$ and $b_k \sim \text{Uniform}(0,1)$. Eq. (4.4) is used for proving the uniformity in [49]. However, in its algorithmic implementation of ICWS, the above is replaced by the following equation

$$\ln y_k = r_k \left(\left\lfloor \frac{\ln S_k}{r_k} + \beta_k \right\rfloor - \beta_k \right), \tag{4.5}$$

where $\beta_k \sim \text{Uniform}(0,1)$. In Eq. (4.5) there exists an implicit quantization of $\ln S_k$ through the floor function, where $\ln S_k$ can thus be divided into several intervals, each of which has a length of r_k. [49] argues that through Eq. (4.5), $\ln y_k$ is sampled from the same uniform distribution on $[\ln S_k - r_k, \ln S_k]$ as that sampled through Eq. (4.4). The floor function and the uniform random variable β_k in Eq. (4.5) ensures that a fixed y_k is sampled in each interval, maintaining global samples. Obviously, Eq. (4.5) gives rise to consistency because small changes in S_k cannot affect the value of y_k.

In order to sample k in proportion to its corresponding weight S_k, in addition to an "active index", y_k, ICWS [49] introduces a second "active index", $z_k \in [S_k, +\infty)$, and builds the relationship among y_k, z_k and r_k:

$$r_k = \ln z_k - \ln y_k. \tag{4.6}$$

Furthermore, by using the two "active indices", ICWS implicitly constructs an exponential distribution parameterized with S_k, i.e., $a_k \sim \text{Exp}(S_k)$:

$$a_k = \frac{c_k}{z_k}, \tag{4.7}$$

where $c_k \sim \text{Gamma}(2,1)$.

On the other hand, considering the random variable, $z_k \in [S_k, +\infty)$, we observe that, by employing knowledge of calculus and marginal distribution, Eq. (4.3) can be expressed as

$$\text{pdf}(y_k, a_k) = \int_S^{+\infty} \text{pdf}(y_k, z_k, a_k) dz_k. \tag{4.8}$$

Because random variables y_k, z_k and a_k all depend on S_k, it is difficult, if not infeasible, to gain their values within a constant time. To solve this problem, we can transform the probability distribution with respect to the three random variables into another one which is independent of S_k. Specifically, we can construct a probability density function of another three random variables, r_k, b_k and c_k, which are all independent of S_k, as follows

$$\text{pdf}(y_k, z_k, a_k) = \text{pdf}(r_k, b_k, c_k) \left| \det \frac{\partial(r_k, b_k, c_k)}{\partial(y_k, z_k, a_k)} \right|, \tag{4.9}$$

where r_k and b_k are defined in Eq. (4.4).

The key trick is to discover the distributions of r_k, b_k and c_k and to build the relationships among the three variables. To this end, ICWS adopts Eqs. (4.6) and (4.7), and derives that

$$r_k, c_k \sim \text{Gamma}(2,1) \tag{4.10}$$

$$b_k \sim \text{Uniform}(0,1) \tag{4.11}$$

Combining Eqs. (4.5), (4.6) and (4.7) with the above distributions, we can see that y_k,z_k and a_k can be expressed as functions of the three random variables, r_k, b_k and c_k. As a result, we can obtain random samples of y_k and a_k through sampling the three random variables from their distributions.

Essentially, ICWS proceeds the sampling process as follows: It first samples y_k using Eq. (4.12), which is derived from Eq. (4.5). Then the sampled y_k, as an independent variable, is fed into Eq. (4.13), which is derived from Eqs. (4.6) and (4.7), and outputs a hash value conforming to the exponential distribution parameterized with the corresponding weight S_k.

$$y_k = \exp\left(r_k\left(\left\lfloor \frac{\ln S_k}{r_k} + \beta_k \right\rfloor - \beta_k\right)\right), \tag{4.12}$$

$$a_k = \frac{c_k}{y_k \exp(r_k)}. \tag{4.13}$$

In ICWS [49], a MinHash code (y_k, a_k) for a weighted element S_k is calculated using the hash functions, Eqs. (4.12) and (4.13), respectively.

4.4 Canonical Consistent Weighted Sampling

In this section, we show that although ICWS conforms to uniformity of the CWS scheme, it actually does not strictly comply with the MinHash scheme. To this end, we propose the Canonical Consistent Weighted Sampling (CCWS) algorithm.

4.4.1 Breach of Uniformity

Admittedly, the uniformity of ICWS has been proved in [49]. However, the proof is based on Eq. (4.4). We would like to challenge Eq. (4.5), which is in fact used for its algorithmic implementation, will lead ICWS to violate the uniformity. In Eq. (4.5), the implicit quantization process (i.e., the weight S_k divided by r_k) is conducted on the logarithm of the weight, $\ln S_k$, instead of the original weight, S_k, which results in unevenly quantized subelements. In essence, ICWS samples y_k uniformly in $[0, \ln S_k]$. Although the function $f^{-1} : \ln x \mapsto x$ is still

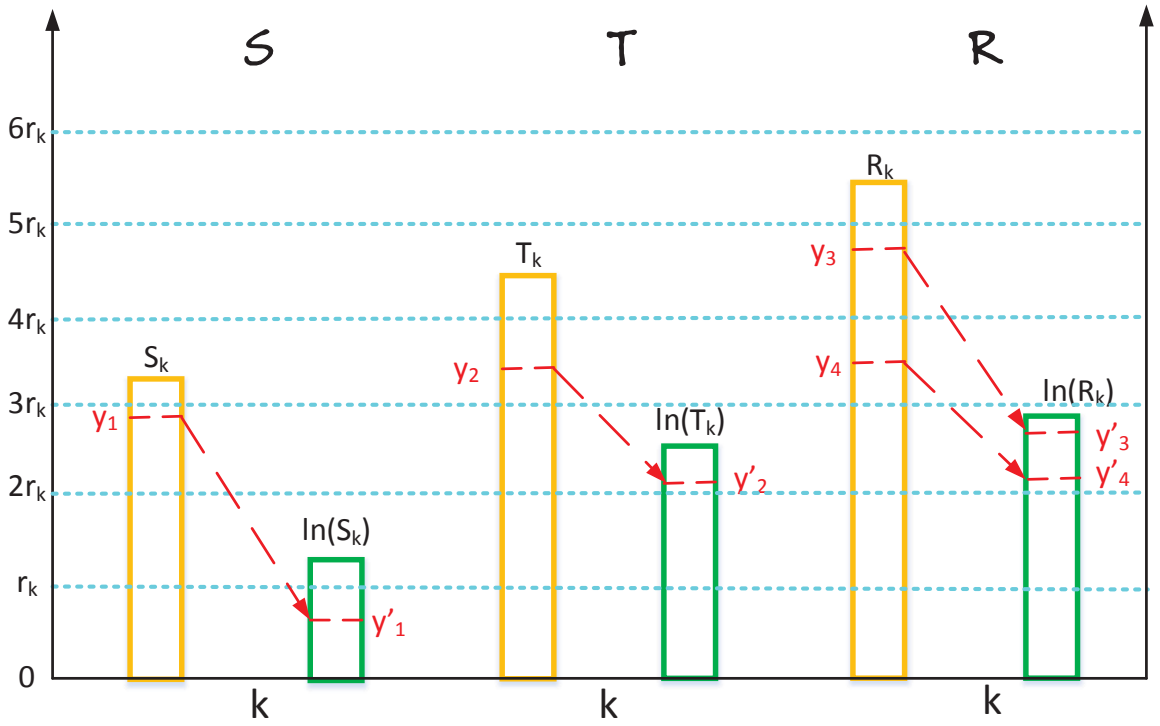


Fig. 4.2 Breach of uniformity. The left bars represent the original weights, while the right ones take the logarithm of the original weights. The red arrows show that the samples are mapped from the original weights onto the logarithmic ones. Because of the logarithmic transform, the samples in different subelements of $\mathscr{T}$ and $\mathscr{R}$ are mapped into the same subelement, which increases the collision probability of the "active indices".

monotonic and bijective, it is unable to preserve the uniformity of y_k in $[0, S_k]$. Therefore, ICWS indeed violates the definition of the MinHash scheme.

Take Figure 4.2 to illustrate the breach of uniformity by [49]: The three pairs of bars represent the k-th element in $\mathscr{S}$, $\mathscr{T}$ and $\mathscr{R}$, respectively. The left yellow bars represent the original weights (e.g., S_k), while the right green ones represent the logarithm of the weights (e.g., $\ln S_k$) used in ICWS [49]. Assume that the weights are evenly divided by the intervals with the length of r_k (indicated by blue dashes). On the left bars, the red dashes labeled by $\{y_1, y_2, y_3\}$ represent the largest "active indices" on the original S_k, T_k and R_k, respectively. By contrast, on the right bars, the red dashes correspond to the ones on the left bars by $f : x \mapsto \ln x$ (indicated by red arrows). The "active indices" y_3 and y_4, which do not belong to the same subelement in the original weight R_k, becomes to be in the same subelement after the nonlinear mapping of the logarithm operation; and y_4, which is not the largest "active index" in R_k, also becomes the largest "active index" in $\ln R_k$. Because there exists such a nonlinear mapping between R_k and $\ln R_k$, uniformly sampling in $[0, \ln S_k]$ as ICWS [49] will result in a non-uniform distribution of y_k in $[0, S_k]$.

Interestingly, taking logarithm on the weights [49] can increase the probability of collision of active indices. Apparently, in the case of Figure 4.2, it is impossible for $\{S_k, T_k, R_k\}$ to yield collision since $\{y_1, y_2, y_3\}$ are in different subelements and mutually unequal. However, the collision will occur between $\ln T_k$ and $\ln R_k$ because y_3' and y_4' belong to the same subelement such that y_4' also becomes the largest "active index", which will collide with y_2'. Despite the breach of uniformity, such a sublinear transform of the weights is more insensitive to large variances of weights (with a similar effect of inverse document frequency) and is sometimes beneficial to classification.

4.4.2 The CCWS Algorithm

Due to its efficiency, ICWS [49] has been considered as the state-of-the-art real-value weighted MinHash algorithm. Thus, if we could uniformly sample "active index" in the original weights to preserve uniformity of the MinHash scheme and, in the meantime, preserve the high efficiency of the ICWS algorithm, we can develop an ideal algorithm. However, we could not simply replace the implicit quantization on $\ln S_k$ by S_k because the underlying mechanism in ICWS will be broken. Therefore, we need to reconstruct the mechanism between "active indices" and the minimum hash values. In this section, we will propose the Canonical Consistent Weighted Sampling (CCWS) algorithm, which strictly meets the definition of the MinHash scheme and has the same efficiency as ICWS.

We follow ICWS [49] by adopting two significant random variables, y_k and z_k. In order to comply with uniformity in the MinHash scheme, Eq. (4.4) is replaced with the following equation

$$y_k = S_k - r_k b_k, \tag{4.14}$$

where $b_k \sim \text{Uniform}(0,1)$. The above equation effectively avoids scaling the weight sublinearly. Similarly, to implicitly quantize the weights in its algorithmic implementation, we adopt the following equation

$$y_k = r_k \left(\left\lfloor \frac{S_k}{r_k} + \beta_k \right\rfloor - \beta_k \right), \tag{4.15}$$

where $\beta_k \sim \text{Uniform}(0,1)$. Eq. (4.15) shows that the quantization is directly conducted on the original weight S_k. Likewise, through Eq. (4.15), y_k is sampled from the same uniform distribution on $[S_k - r_k, S_k]$ as that sampled through Eq. (4.14). In addition, y_k is uniformly distributed on $[0, S_k]$ as there exists linear relationship. Also, y_k is fixed in each interval through the floor function and the uniform random variable β_k in Eq. (4.15). In terms of

Algorithm 1 The Canonical Consistent Weighted Sampling Algorithm (CCWS)

Input: $\mathscr{S} = \{S_1, \cdots, S_n\}$

Output: (k^*, y_{k^*})

for $k = 1, \ldots, n$ **do**
  $\beta_k \sim \text{Uniform}(0,1)$
  $r_k \sim \text{Beta}(2,1)$ // i.e., $p(r_k) = 2r_k$
  $c_k \sim \text{Gamma}(2,1)$ // i.e., $p(c_k) = c_k \text{e}^{-c_k}$
end for
for all k such that $S_k > 0$ **do**
  $t_k = \left\lfloor \frac{S_k}{r_k} + \beta_k \right\rfloor$
  $y_k = r_k(t_k - \beta_k)$
  $a_k = \frac{c_k}{y_k} - 2r_k c_k$
end for
$k^* = \arg\min_k a_k$
return (k^*, y_{k^*})

consistency, the value of y_k is not affected by small changes in S_k due to the floor function in Eq. (4.15).

After sampling y_k using Eq. (4.15), we aim to construct a function with respect to y_k such that the hash value a_k of (k, y_k) is exponentially distributed with the parameter being its weight S_k. Summarizing Eq. (4.3), (4.8) and (4.14), we have

$$\int_S^{+\infty} \text{pdf}(y_k, z_k, a_k) dz_k = \frac{1}{S_k}(S_k \text{e}^{-S_k a_k}) = \text{e}^{-S_k a_k} \tag{4.16}$$

$$\text{pdf}(y_k, z_k, a_k) = a_k \text{e}^{-a_k z_k} \tag{4.17}$$

$$y_k = S_k - r_k b_k \tag{4.18}$$

where $b_k \sim \text{Uniform}(0,1)$.

Our goal is to discover the distributions of r_k and c_k then use these random variables to build the relationship among $\{y_k, z_k, a_k\}$. Combining Eq. (4.9) with the above equations, we are able to construct a partial differential equation. By solving the equation, we acquire the following expressions

$$r_k = \frac{1}{2}\left(\frac{1}{y_k} - \frac{1}{z_k}\right) \tag{4.19}$$

$$c_k = a_k z_k, \tag{4.20}$$

where $r_k \sim \text{Beta}(2,1)$ and $c_k \sim \text{Gamma}(2,1)$.

In summary, we have obtained the distributions of the three independent random variables: $\beta_k \sim \text{Uniform}(0,1)$, $r_k \sim \text{Beta}(2,1)$, and $c_k \sim \text{Gamma}(2,1)$. For each k, we just need to draw three global random variables from their distributions, which are all independent of the data (thus they can be sampled offline). Based on the sampled $\{\beta_k, r_k, c_k\}_{k=1}^n$, we proceed some basic arithmetic operations to acquire the random samples of y_k and a_k for a weighted set $\mathscr{S}$. Finally, we can easily obtain (k^*, y_{k^*}) which is assigned with the minimum hash value. Thus far, we have presented the CCWS algorithm for weighted sets, which strictly complies with the definition of the MinHash scheme – Because our CCWS algorithm directly quantizes the original weights. The CCWS algorithm is outlined in Algorithm 1.

4.4.3 Proof of Correctness

In this subsection, we will prove that our CCWS algorithm indeed generates samples which satisfy uniformity and consistency in the CWS scheme [49, 84] .

Uniformity

We drop the element index $k \in \{1, \cdots, n\}$ for conciseness. Note that the random variable $y = r\left(\lfloor \frac{S}{r} + \beta \rfloor - \beta\right)$, where β is uniformly distributed on $[0,1]$, shares the same distribution as $y = S - rb$, where b is uniformly distributed on $[0,1]$. In the two situations, y is both uniformly sampled from $[S-r, S]$. The distribution $P(y,z,a)$, defined for $y \leq S, z \geq S$ and $a > 0$, can be obtained by transforming the distributions of the random variables as

$$\text{pdf}(y,z,a) = \text{pdf}(r,b,c)\left|\det\frac{\partial(r,b,c)}{\partial(y,z,a)}\right|, \tag{4.21}$$

where $r = \frac{1}{2}(\frac{1}{y} - \frac{1}{z})$, $b = \frac{2yz(S-y)}{z-y}$, and $c = az$. Recall that r, b and c are mutually independent and have the following probability density functions: $p(r) = 2r$, $p(b) = 1$, and $p(c) = c\mathrm{e}^{-c}$. by computing the Jacobian determinant, we obtain

$$\begin{aligned}
\text{pdf}(y,z,a) &= \text{pdf}(r)\text{pdf}(b)\text{pdf}(c) \\
&\quad \times \left\| \begin{array}{ccc} -\frac{1}{2y^2} & \frac{1}{2z^2} & 0 \\ \frac{2Sz^2 - 4yz^2 + 2y^2z}{(z-y)^2} & \frac{-2Sy^2 + 2y^3}{(z-y)^2} & 0 \\ 0 & a & z \end{array} \right\| \\
&= a\mathrm{e}^{-az}
\end{aligned}$$

Simply, integrating the above equation over z gives

$$\begin{aligned}\mathrm{pdf}(y,a) &= \int_S^{+\infty} \mathrm{pdf}(y,z,a)dz \\ &= \frac{1}{S}(S\mathrm{e}^{-Sa}) \\ &= \mathrm{pdf}(y)\mathrm{pdf}(a).\end{aligned}$$

It is easy to see that y is uniformly distributed in $[0,S]$ and a complies with an exponential distribution parameterized with S, that is, $a \sim \mathrm{Exp}(S)$; meanwhile, y and a are independent. For all the weights $\{S_1,\ldots,S_n\}$ in the weighted set, there exist a set of exponential distributions parameterized with the corresponding weights. According to Eq. (4.2), the minimum hash value a_{k^*} is proportional to S_{k^*},

$$\Pr(a_{k^*} = \min_k a_k) = \frac{S_{k^*}}{\sum_k S_k}.$$

Therefore, (k^*, y_{k^*}) is uniformly sampled from $\bigcup_k (\{k\} \times [0, S_k])$.

Consistency

Following [49], we will demonstrate that, for two non-empty weighted sets $\mathscr{S}$ and $\mathscr{T}$, if $\forall k, T_k \leq S_k$, a subelement (k^*, y_{k^*}) is sampled from $\mathscr{S}$ and satisfies $y_{k^*} \leq T_{k^*}$, then (k^*, y_{k^*}) will be also sampled from $\mathscr{T}$. Considering an element k^*, we have $t_{k^*} = \left\lfloor \frac{S_{k^*}}{r_{k^*}} + \beta_{k^*} \right\rfloor$. According to the definition of the floor function, $\frac{S_{k^*}}{r_{k^*}} + \beta_{k^*} - 1 < t_{k^*} \leq \frac{S_{k^*}}{r_{k^*}} + \beta_{k^*}$ holds. By hypothesis, $y_{k^*} \leq T_{k^*} \leq S_{k^*}$, We can easily obtain

$$\begin{aligned}\frac{T_{k^*}}{r_{k^*}} + \beta_{k^*} - 1 &\leq \frac{S_{k^*}}{r_{k^*}} + \beta_{k^*} - 1 \\ &< t_{k^*} = \frac{y_{k^*}}{r_{k^*}} + \beta_{k^*} \\ &\leq \frac{T_{k^*}}{r_{k^*}} + \beta_{k^*}.\end{aligned}$$

Obviously, we have

$$t_{k^*} = \left\lfloor \frac{T_{k^*}}{r_{k^*}} + \beta_{k^*} \right\rfloor = t'_{k^*}, \tag{4.22}$$

which indicates $y_{k^*} = y'_{k^*}$. Thus y_{k^*} and y'_{k^*} will be sampled from the k^*-th elements of $\mathscr{S}$ and $\mathscr{T}$, respectively. Combining Eq. (4.15), (4.19) and (4.20), we can find that, for any k,

a_k is actually a function with respect to S_k,

$$a_k = \frac{c_k}{r_k\left(\left\lfloor \frac{S_k}{r_k}+\beta_k \right\rfloor - \beta_k\right)} - 2r_k c_k, \tag{4.23}$$

and it is monotonically non-increasing. Therefore, $\forall k, a'_k \geq a_k$ due to $T_k \leq S_k$, while $a'_{k^*} = a_{k^*} = \min_k a_k$ because of $y_{k^*} = y'_{k^*}$. As a result, $a'_{k^*} \leq a_k \leq a'_k$ and in turn $\arg\min_k a'_k = \arg\min_k a_k = k^*$, which demonstrates that (k^*, y_{k^*}) is sampled from $\mathscr{S}$ and $\mathscr{T}$ simultaneously. Thus consistency holds.

In summary, our CCWS algorithm strictly abides by the definition of the MinHash scheme, and also satisfies the two conditions, uniformity and consistency, of the CWS scheme.

4.4.4 Remarks

Our CCWS algorithm inherits the merit of ICWS [49]: For each element S_k, it acquires y_k in $O(1)$ by introducing another "active index" z_k and bridging the two "active indices" y_k and z_k as well as the minimum hash value a_k; thus it enjoys the the same theoretical computational complexity as ICWS. It is easy to see from Algorithm 1 that the time and space complexities of the CCWS algorithm are both $O(n)$, where n represents the size of the set. This merit enables the algorithm to avoid traversing all "active indices" of each element in IWMH [34], especially in the case that each weight is multiplied by a large constant ($C \gg 1$), which gives rise to a dramatic rise in computation.

In practical implementation of the CCWS algorithm, there is an option of scaling weights with a constant. During the process of quantization in Eq. (4.15), if S_k is too small or too large, the underlying random quantization might fail: In the case of very small weights, only one interval is able to cover the weights; while in the case of very large weights, numerous intervals are required to cover the weights so that the subelement which is close to 0 is barely sampled as y_k. In order to avoid these numerical problems, we can simply scale the weight S_k in Eq. (4.15) by a proper constant C as

$$y_k = r_k\left(\left\lfloor \frac{CS_k}{r_k}+\beta_k \right\rfloor - \beta_k\right), \tag{4.24}$$

Although scaling is applied to the weights, it has no effects on the correctness and efficiency of the CCWS algorithm because Eq. (4.24) still shares the same distribution as Eq. (4.14). The empirical analysis shows that the scaled weights CS_k works properly in the order of magnitude ranging from 0.01 to 100.

4.5 Practical Consistent Weighted Sampling

In this section, we further explore the ICWS algorithm and successfully uncovers the mathematical foundation. First, we revisit ICWS [49] and transform its original form Eq. (4.13) for sampling a_k into an equivalent version. Based on this equivalent form, we find some interesting properties which inspire us to make the ICWS algorithm simpler and more efficient in both space and time complexities. Consequently, we propose a more practical algorithm for consistent weighted sampling.

4.5.1 ICWS Revisited

Recall that in ICWS [49] one needs to sample two Gamma variables, $r_k \sim \text{Gamma}(2,1)$ and $c_k \sim \text{Gamma}(2,1)$, in Eqs. (4.12) and (4.13) to directly generate y_k and a_k. In fact, the simplest programming implementation[3] to generate a random variable x from $\text{Gamma}(2,1)$ is to first sample two uniform random variables, $u_1, u_2 \sim \text{Uniform}(0,1)$ and then conduct a transformation as $x = -\ln(u_1 u_2)$. Thus if we represent the two independent Gamma variables r_k and c_k in terms of uniform random variables, $r_k = -\ln(u_{k1}u_{k2})$ and $c_k = -\ln(v_{k1}v_{k2})$, where $u_{k1}, u_{k2}, v_{k1}, v_{k2} \sim \text{Uniform}(0,1)$, we can obtain an equivalent version of Eqs. (4.12) and (4.13):

$$y_k = \exp(-\ln(u_{k1}u_{k2})(t_k - \beta_k)), \tag{4.25}$$

$$a_k = \frac{-\ln(v_{k1}v_{k2})}{y_k(u_{k1}u_{k2})^{-1}}, \tag{4.26}$$

where

$$t_k = \left\lfloor \frac{\ln S_k}{-\ln(u_{k1}u_{k2})} + \beta_k \right\rfloor. \tag{4.27}$$

Now, the ICWS algorithm in [49] can be presented equivalently in Algorithm 2 for producing D MinHash codes.

Algorithm 2 is the standard implementation of ICWS. If we further slightly rearrange Eq. (4.26) as follows

$$a_k = \frac{-\ln(v_{k1}v_{k2})u_{k2}}{y_k u_{k1}^{-1}}, \tag{4.28}$$

[3]Some programming languages may provide built-in functions to generate random variable from $\text{Gamma}(\alpha,\beta)$, which however has a complexity no better than this method in the case of $\alpha = 2, \beta = 1$.

we can find two interesting properties: (1) the denominator, $y_k u_{k1}^{-1}$, is essentially an unbiased estimator of the weight S_k; and (2) the numerator $-\ln(v_{k1}v_{k2})u_{k2}$ is actually a standard exponential distribution.

The first property can be easily verified based on the uniformity $y_k = u_{k1}S_k$, where $u_{k1} \sim \text{Uniform}(0,1)$, because the derivation of the ICWS algorithm [49] is based on $y_k = u_{k1}S_k$. However, in its algorithm, ICWS [49] samples y_k through Eq. (4.25) instead of $y_k = u_{k1}S_k$. Thus $y_k u_{k1}^{-1}$ is not exactly equivalent to S_k but its unbiased estimator:

$$\mathbb{E}(y_k u_{k1}^{-1}) = \mathbb{E}(\hat{S}_k) = S_k. \tag{4.29}$$

To see the second property, let $m_k = -\ln(v_{k1}v_{k2})u_{k2} = c_k u_{k2}$ then we have

$$\begin{aligned} \text{pdf}_{M_k}(m_k) &= \int_{0^+}^{1} \frac{1}{u_{k2}} \text{pdf}_{U_{k2}}(u_{k2}) \text{pdf}_{C_k}\left(\frac{m_k}{u_{k2}}\right) du_{k2} \\ &= \int_{0^+}^{1} \frac{1}{u_{k2}} \cdot 1 \cdot \frac{m_k}{u_{k2}} \mathrm{e}^{-\frac{m_k}{u_{k2}}} du_{k2} \\ &= \mathrm{e}^{-m_k} \end{aligned} \tag{4.30}$$

which implies that

$$m_k = -\ln(v_{k1}v_{k2})u_{k2} \sim \text{Exp}(1). \tag{4.31}$$

Substituting Eqs. (4.29) and (4.31) into Eq. (4.28) we obtain $a_k = \frac{m_k}{\hat{S}_k}$. Through the Jacobian transformation, we have

$$\text{pdf}_{A_k}(a_k) = \text{pdf}_{M_k}(m_k)\left|\frac{dm_k}{da_k}\right| = \hat{S}_k \mathrm{e}^{-\hat{S}_k a_k}, \tag{4.32}$$

which further implies $a_k \sim \text{Exp}(\hat{S}_k)$. Thus far, *we have found that the mystery of ICWS is to implicitly employ an exponential distribution parameterized with $\hat{S}_k$ to sample a_k*, that is

$$a_k = \frac{m_k}{\hat{S}_k} \sim \text{Exp}(\hat{S}_k). \tag{4.33}$$

In this case, the probability of the k-th element being sampled is equal to the probability of the k-th element being assigned with the minimum exponential random variable a_k, where the probability is in proportion to its weight:

$$\Pr(a_k = \min\{a_1, \cdots, a_n\}) = \frac{\hat{S}_k}{\sum_{k'} \hat{S}_{k'}}. \tag{4.34}$$

Algorithm 2 The ICWS Algorithm (equivalent to [49])

Input: $\mathscr{S} = \{S_1, \cdots, S_n\}$; number of samples D

Output: $\{(k_*^{(d)}, y_{k_*^{(d)}}^{(d)})\}_{d=1}^{D}$

for $k = 1, \ldots, n$ **do**
  for $d = 1, \ldots, D$ **do**
    $u_{k1}^{(d)}, u_{k2}^{(d)} \sim \text{Uniform}(0,1)$
    $\beta_k^{(d)} \sim \text{Uniform}(0,1)$
    $v_{k1}^{(d)}, v_{k2}^{(d)} \sim \text{Uniform}(0,1)$
  end for
end for
for all k such that $S_k > 0$ **do**
  for $d = 1, \ldots, D$ **do**
    $t_k^{(d)} = \left\lfloor \dfrac{\ln S_k}{-\ln(u_{k1}^{(d)} u_{k2}^{(d)})} + \beta_k^{(d)} \right\rfloor$
    $y_k^{(d)} = \exp(-\ln(u_{k1}^{(d)} u_{k2}^{(d)})(t_k^{(d)} - \beta_k^{(d)}))$
    $a_k^{(d)} = \dfrac{-\ln(v_{k1}^{(d)} v_{k2}^{(d)})}{y_k^{(d)} (u_{k1}^{(d)} u_{k2}^{(d)})^{-1}}$
  end for
end for
for $d = 1, \ldots, D$ **do**
  $k_*^{(d)} = \arg\min_k a_k^{(d)}$
end for
return $\{(k_*^{(d)}, y_{k_*^{(d)}}^{(d)})\}_{d=1}^{D}$

4.5.2 The PCWS Algorithm

By transforming the original ICWS algorithm into an equivalent version in terms of five independent uniform random variables, we have revealed the mystery of ICWS Eq. (4.33). In addition to understanding the underlying mechanism of ICWS, one may ask what else we have been inspired to improve the ICWS algorithm. In the following, we will make use of the uncovered property Eq. (4.31) to reduce both time and space complexities of ICWS.

Recall in Eq. (4.28), the numerator employs three independent uniform random variables to produce a sample in the form of $-\ln(v_1 v_2) u_2$, which is proved to be a standard exponential distribution, $m_k \sim \text{Exp}(1)$ in Eq. (4.31). Obviously, it is costly in terms of both time and space to produce $-\ln(v_1 v_2) u_2$. Due to the fact that $-\ln x_k \sim \text{Exp}(1)$, $x_k \sim \text{Uniform}(0,1)$, we can adopt $-\ln x_k$ insted of $-\ln(v_1 v_2) u_2$ to achieve the same goal.

To this end, we can simply replace $a_k = \frac{-\ln(v_{k1}v_{k2})}{y_k(u_{k1}u_{k2})^{-1}}$ (Line 12 in Algorithm 2) with $a_k = \frac{-\ln x_k}{y_k u_{k1}^{-1}}$ and obtain the proposed PCWS algorithm (see Algorithm 3). The only two differences between ICWS and PCWS lie in

- ICWS has to generate one more uniform random variable than PCWS for each element k (Line 5).
- ICWS has a relatively more complicated expression than PCWS to calculate a_k (Line 12).

Complexity: In programming implementation, ICWS requires sampling five global uniform random variables for each element k (i.e., $u_{k1}, u_{k2}, \beta_k, v_{k1}, v_{k2}$); while PCWS only requires sampling four (i.e., $u_{k1}, u_{k2}, \beta_k, x_k$). From Algorithm 2 and Algorithm 3 it is easy to see that the space complexity of PCWS is $\mathcal{O}(4nD)$ while the space complexity of ICWS is $\mathcal{O}(5nD)$, where n denotes the size of the universal set and D the number of samples (number of MinHashes). Although the uniform random variables can be sampled off-line, it is worth noting that these variables have to be cached in the memory during the hashing process. In text mining applications, the size of the universal set (number of features) can easily reach 10^7; if we adopt 10^3 samples, an additional memory footprint of 10^{10} floats have to be allocated. Thus, using one less uniform random variable can save substantial amount of memory cost on large-scale data sets. The simpler expression of PCWS for calculating a_k also saves time complexity $\mathcal{O}(nD)$. On a large-scale data set, the aggregated time saving due to one less uniform random variable and simpler expression can be substantial (see empirical test in Section 6.4). Compared to ICWS, PCWS enjoys 20% lower memory footprint and saves $1/5 \sim 1/3$ empirical runtime.

4.5.3 Analysis

In this subsection, we will prove that our PCWS algorithm generates (y_k, a_k) which indeed satisfy uniformity and consistency of the CWS scheme [84] .

Uniformity

We drop the element index k for conciseness. Following ICWS [49], the random variable $\ln y = r\left(\left\lfloor \frac{\ln S}{r} + \beta \right\rfloor - \beta\right)$, where $r = -\ln(u_1 u_2) \sim \text{Gamma}(2,1)$, $\beta \sim \text{Uniform}(0,1)$, shares the same distribution as $\ln y = \ln S - rb$, where $b \sim \text{Uniform}(0,1)$. In the both situations, $\ln y$ is uniformly sampled from $[\ln S - r, \ln S]$. Since our PCWS algorithm still employs two "active indices", y and z, as ICWS does, we have $r = \ln z - \ln y$ for the sake of proof of

Algorithm 3 The PCWS Algorithm

Input: $\mathscr{S} = \{S_1, \cdots, S_n\}$; number of samples D

Output: $\{(k_*^{(d)}, y_{k_*^{(d)}}^{(d)})\}_{d=1}^{D}$

for $k = 1, \ldots, n$ **do**
  for $d = 1, \ldots, D$ **do**
    $u_{k1}^{(d)}, u_{k2}^{(d)} \sim \text{Uniform}(0,1)$
    $\beta_k^{(d)} \sim \text{Uniform}(0,1)$
    $x_k^{(d)} \sim \text{Uniform}(0,1)$ // different from Algorithm 2
  end for
end for
for all k such that $S_k > 0$ **do**
  for $d = 1, \ldots, D$ **do**
    $t_k^{(d)} = \left\lfloor \dfrac{\ln S_k}{-\ln(u_{k1}^{(d)} u_{k2}^{(d)})} + \beta_k^{(d)} \right\rfloor$
    $y_k^{(d)} = \exp(-\ln(u_{k1}^{(d)} u_{k2}^{(d)})(t_k^{(d)} - \beta_k^{(d)}))$
    $a_k^{(d)} = \dfrac{-\ln x_k^{(d)}}{y_k^{(d)} (u_{k1}^{(d)})^{-1}}$ // different from Algorithm 2
  end for
end for
for $d = 1, \ldots, D$ **do**
  $k_*^{(d)} = \arg\min_k a_k^{(d)}$
end for
return $\{(k_*^{(d)}, y_{k_*^{(d)}}^{(d)})\}_{d=1}^{D}$

uniformity. The distribution $\text{pdf}(y,z,a)$, defined for $y \leq S$, $z \geq S$ and $a > 0$, can be obtained by transforming the distributions of the random variables as

$$\text{pdf}(y,z,a) = \text{pdf}(r,b,x) \left| \det \frac{\partial(r,b,x)}{\partial(y,z,a)} \right|,$$

where $r = \ln z - \ln y$, $b = \frac{\ln S - \ln y}{\ln z - \ln y}$, and $x = \exp(-ayu_1^{-1})$. Recall that r, b, x are mutually independent and have the following probability density functions: $\text{pdf}(r) = re^{-r}$ and $\text{pdf}(b) = \text{pdf}(x) = 1$. By computing the Jacobian determinant, we obtain

$$\text{pdf}(y,z,a) = \frac{1}{z^2}(yu_1^{-1})e^{-(yu_1^{-1})a}.$$

Marginalizing out z in $\mathrm{pdf}(y,z,a)$ gives

$$\mathrm{pdf}(y,a) \quad = \quad \int_S^{+\infty} \mathrm{pdf}(y,z,a)dz = \frac{1}{S}(yu_1^{-1})\mathrm{e}^{-(yu_1^{-1})a}.$$

Actually $\mathrm{pdf}(y,a)$ is still conditioned on u_1. We can do an expectation over the distribution of yu_1^{-1} and, according to Eq. (4.29), obtain $\mathbb{E}(yu_1^{-1}) = S$. Therefore, we have

$$\mathrm{pdf}(y,a) = \frac{1}{S}(S\mathrm{e}^{-Sa}) = \mathrm{pdf}(y)\mathrm{pdf}(a).$$

It is easy to see that y is uniformly distributed in $[0,S]$ and a complies with an exponential distribution parameterized with S, that is, $a \sim \mathrm{Exp}(S)$; meanwhile, y and a are independent. For all the weights $\{S_1,\ldots,S_n\}$ in weighted set $\mathscr{S}$, there exist a set of exponential distributions parameterized with the corresponding weights. According to Eq. (4.2), a_{k_*} is the minimum hash value with a probability in proportion to S_{k_*}, $\Pr(a_{k_*} = \min_k a_k) = \frac{S_{k_*}}{\sum_k S_k}$. Therefore, (k_*, y_{k_*}) is uniformly sampled from $\bigcup_k (\{k\} \times [0, S_k])$.

Consistency

Following ICWS [49], we will demonstrate that, for two non-empty weighted sets $\mathscr{S}$ and $\mathscr{T}$, if $\forall k, T_k \leq S_k$, a subelement (k_*, y_{k_*}) is sampled from $\mathscr{S}$ and satisfies $y_{k_*} \leq T_{k_*}$, then (k_*, y_{k_*}) will be sampled from $\mathscr{T}$.

Considering an element k_*, $t_{k_*}^S = \left\lfloor \frac{\ln S_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*} \right\rfloor$, and thus $\frac{\ln S_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*} - 1 < t_{k_*}^S \leq \frac{\ln S_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*}$. By hypothesis, $y_{k_*}^S = y_{k_*} \leq T_{k_*} \leq S_{k_*}$, thus $\frac{\ln T_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*} - 1 < t_{k_*}^S = \frac{\ln y_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*} \leq \frac{\ln T_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*}$. Obviously,

$$t_{k_*}^S = \left\lfloor \frac{\ln T_{k_*}}{-\ln(u_{k_*1}u_{k_*2})} + \beta_{k_*} \right\rfloor = t_{k_*}^T,$$

which indicates $y_{k_*}^S = y_{k_*} = y_{k_*}^T$. Thus $y_{k_*}^S$ and $y_{k_*}^T$ will be sampled from the k_*-th elements of $\mathscr{S}$ and $\mathscr{T}$, respectively.

On the other hand, we note that, for any k, a_k is essentially a monotonically non-increasing function of S_k:

$$a_k = \frac{-\ln x_k}{u_{k1}^{-1} \exp\left(r_k\left(\left\lfloor \frac{\ln S_k}{r_k} + \beta_k \right\rfloor - \beta_k\right)\right)}, \tag{4.35}$$

where $r_k = -\ln(u_{k1}u_{k2})$. Therefore, $\forall k, a_k^T \geq a_k^S$ due to $T_k \leq S_k$, while $a_{k_*}^T = a_{k_*}^S = \min_k a_k^S$ because of $y_{k_*}^S = y_{k_*}^T$. As a result, $a_{k_*}^T \leq a_k^S \leq a_k^T$ and in turn $\arg\min_k a_k^T = \arg\min_k a_k^S = k_*$,

which demonstrates that (k_*, y_{k_*}) is sampled from $\mathscr{S}$ and $\mathscr{T}$ simultaneously. Thus consistency holds.

In summary, our PCWS algorithm satisfies the two properties, uniformity and consistency, of the CWS scheme. Thus, PCWS not only has an equivalent (but more efficient) algorithm to ICWS but also holds the same theoretical properties as ICWS.

4.6 Experimental Results

In this section, we will report the performance of the two proposed algorithms, i.e., CCWS and PCWS, and its competitors on a number of real-world text data sets. First, we investigate the effectiveness and efficiency of our CCWS algorithm and the competitors for document classification; second, we report the effectiveness and efficiency of our PCWS algorithm and the competitors for document classification and retrieval.

4.6.1 Experimental Preliminaries

All compared algorithms are as follows:

- **MinHash**: The standard MinHash scheme is applied by simply treating weighted sets as binary sets;
- **IWMH**: It firstly introduces the idea of "active index" to accelerate the integer weighted Min-Hash. To handle real-valued weights, it needs to transform the weights into integers by multiplying a constant C.
- **[Gollapudi et.al., 2006] [34]**: It transforms weighted sets into binary sets by thresholding real-value weights with random samples and then applies the standard MinHash scheme (another algorithm is introduced in the same paper which is however extremely inefficient for real-value weights and thus not reported);
- **[Chum et.al., 2008] [21]**: It approximates the generalized Jaccard similarity with a bias for real-value weighted sets despite that the derivation is based on integer weighted sets;
- **ICWS [49]**: It is introduced in Section 4.3, which is currently the state-of-the-art for weighted MinHash in terms of both effectiveness and efficiency;
- **[Li, 2015] [67]**: It approximates ICWS by simply discarding one of the two components, that is y_k in Eq. (4.12), of ICWS;

- **[Haeupler et.al., 2014] [41]**: It approximates the generalized Jaccard similarity by rounding the real-value weights with probability and then quantizing the integer weights.

All the compared algorithms are implemented in Matlab. For [Haeupler et.al., 2014], each weight is scaled up by a factor of 100, while in IWMH, we empirically keep four effective digits for real weights, so we set the parameter C to 10,000. We first apply all the algorithms to generate the fingerprints of the data. Suppose that each algorithm generates $\mathbf{x}_S$ and $\mathbf{x}_T$, which are the fingerprints with the length of D for the two real-value weighted sets, $\mathscr{S}$ and $\mathscr{T}$, respectively, the similarity between $\mathscr{S}$ and $\mathscr{T}$ is

$$\mathbf{Sim}_{\mathscr{S},\mathscr{T}} = \sum_{d=1}^{D} \frac{\mathbf{1}(x_{S,d} = x_{T,d})}{D}, \tag{4.36}$$

where $\mathbf{1}(state) = 1$ if $state$ is true, and $\mathbf{1}(state) = 0$ otherwise. The above equation calculates the ratio of the same MinHash values (i.e., collision) between $\mathbf{x}_S$ and $\mathbf{x}_T$, which is used to approximate the probability that $\mathscr{S}$ and $\mathscr{T}$ generate the same MinHash value. We set D, the parameter of the number of hash functions (or random samples), such that $D \in \{2,4,8,16,32,64,128,256,512,1024\}$. All the random variables are globally generated at random. That is, in one sampling process, the same elements in different sets use the same set of random variables. All the experiments are conducted on a node of a Linux Cluster with 8×3.1 GHz Intel Xeon CPU (64 bit) and 1TB RAM.

4.6.2 Experiments for CCWS

In the experiments, we compare our CCWS algorithm with the compared methods, i.e., IWMH and ICWS. We would like to note that since it strictly complies with the definition of the MinHash scheme, *it can be regarded as the state-of-the-art in terms of accuracy in the experiments of CCWS*; by contrast, ICWS improves the efficiency of IWMH by avoiding the explicit quantization and traversing all "active indices", and thus *it is currently the state-of-the-art weighted MinHash in terms of efficiency* (although it violates the uniformity of the MinHash scheme).

We evaluate the classification performance of the compared methods using LIBSVM [14] with 10-fold cross-validation on four real-world text data sets for document classification:

- **Real-sim** [4]: The data set is a collection of UseNet articles from four discussion groups about simulated auto racing, simulated aviation, real autos and real aviation, respectively. The formatted document data set, with 72,309 samples and 20,958 features,

[4] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#real-sim

has been arranged into two classes: real and simulated. Since the real class and the simulated class of the original data are unbalanced, data preprocessing is performed by randomly selecting 10,000 real samples and 10,000 simulated samples as positive and negative instances, respectively.

- **Rcv1** [64]: The data set is a large collection of newswire stories drawn from online databases. The formatted data set [5] has 20,242 training samples with 47,236 features. The data set has been categorized into two classes: positive instances contain CCAT and ECAT while negative ones contain GCAT and MACT on the website. Similarly, we randomly select 10,000 positive instances and 10,000 negative ones to compose a balanced data set.

- **Webspam** [6]: It is a web text data set provided by a large-scale learning challenge. The data set has 350,000 instances and 16,609,143 features. We randomly select 1,000 samples from the original data set as query examples and the rest as the database.

- **DBLP** (Paper Abstract) [120]: It contains 1,572,277 papers. Following [19], we consider a binary classification task: Artificial Intelligence (AI) vs. Computer Vision (CV). We randomly select 7,000 AI abstracts and 7,000 CV abstracts for balance.

Discussions on Real-sim: The first row in Figure 4.3 reports the experimental results on the Real-sim data set, where the upper subplot shows the accuracy and the lower one illustrates the runtime. It can be observed that our CCWS algorithm achieves almost the same accuracy as IWMH, which demonstrates that our CCWS satisfies the MinHash definition as IWMH does. Note that, although ICWS violates the uniformity of MinHash, it also achieves competitive accuracy, if not even better, on this data set. For example, CCWS performs slightly worse than ICWS within 1% with D ranging from 32 to 1024. However, as the length of fingerprints (the number of hash functions) D increases, the gap between our CCWS algorithm and ICWS gradually narrows. The reason may be the one discussed in Section 4.4.1, that is, ICWS has higher probability of collision due to its sublinear transformation of the weights. In the cases of smaller D, it is easier for ICWS to get collided hash values; as the number of hash values increases, CCWS also becomes to get collided in terms of hash values and the performance gap between ICWS and CCWS is closed.

Discussions on Rcv1: The subplots in the second row of Figure 4.3 illustrate the experimental results on the Rcv1 data set. Likewise, our CCWS algorithm still remains its advantage in terms of accuracy: it competes well with ICWS, and maintains the same level

[5] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#rcv1.binary
[6] http://largescale.ml.tu-berlin.de/instructions/

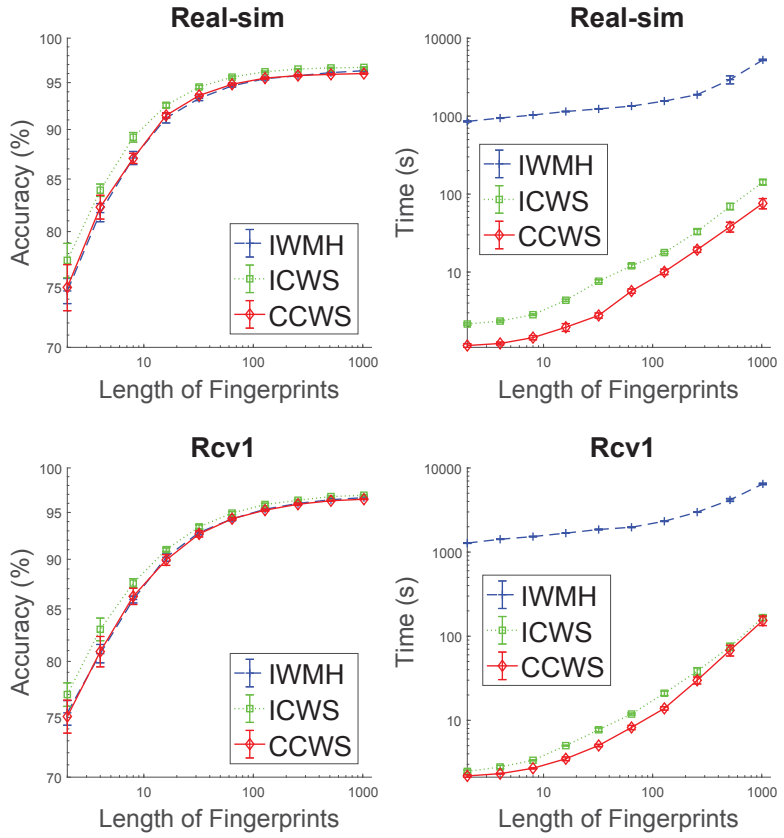


Fig. 4.3 Performance results, classification accuracy (left column) and runtime (right column) of the compared methods, on the four real-world text data sets. The x-axis denotes the length of fingerprints, D.

as IWMH. Again, the slight performance gap between our CCWS algorithm and ICWS becomes increasingly close when the number of hash values (the number of sampling times) increases. From the perspective of runtime, our CCWS is generally superior to ICWS; besides, it still overwhelmingly defeats IWMH (by $2 \sim 3$ orders of magnitude).

Discussions on Webspam: In order to evaluate the genuine ability on data with low variances of weights, we compare our CCWS algorithm with its competitors on the Webspam data set. The experimental results are reported in the first row of Figure 4.4, where each algorithm is given a cutoff time of 50,000 seconds because the data set contains much more features (as shown in Table 2.2) and is denser than the previous data sets so that it is time-consuming. As shown in Table 2.2, the average standard deviation of weights of the 20,000 documents is only 0.0015, which is much smaller than those of the other data sets. We observe that on the data set with low variances of weights, our CCWS algorithm even achieves nearly the same accuracy performance as ICWS, which is more insensitive to large variances of weights. Besides, CCWS even slightly outperforms IWMH in accuracy.

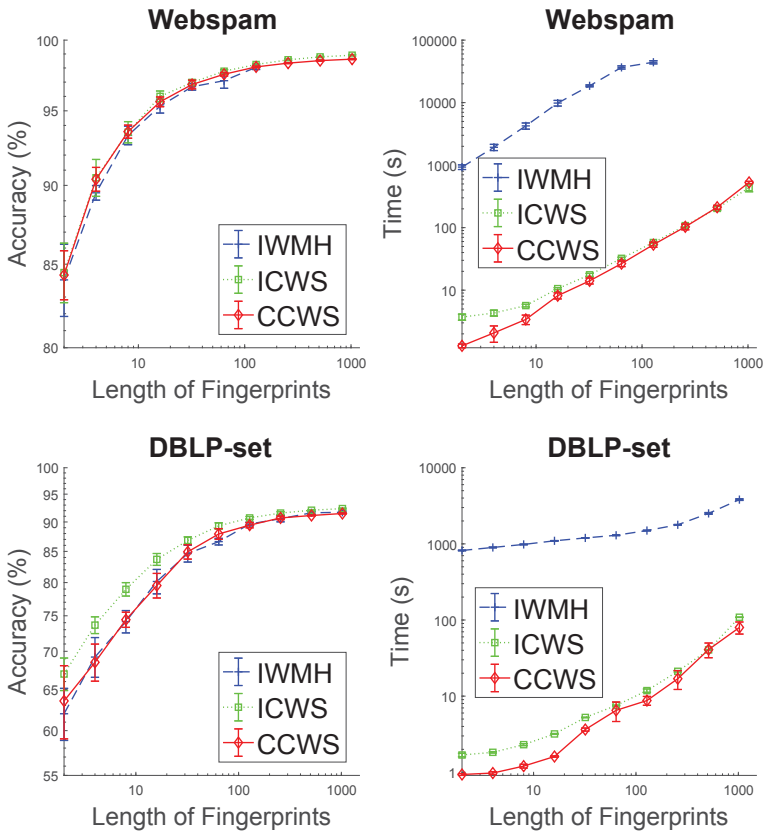


Fig. 4.4 Performance results, classification accuracy (left column) and runtime (right column) of the compared methods, on the four real-world text data sets. The x-axis denotes the length of fingerprints, D.

In terms of runtime, CCWS runs similarly with ICWS. Compared to IWMH, again, CCWS performs remarkably faster by $2 \sim 3$ orders of magnitude.

Discussions on DBLP-set: Subplots in the second row of Figure 4.4 show the experimental results on the DBLP-set data set, where the upper subplot reports the accuracy and the lower one illustrates the runtime. In terms of accuracy, although our CCWS algorithm performs worse than ICWS, it still achieves almost the same accuracy as IWMH does, which again demonstrates that even if our CCWS algorithm performs relatively worse than ICWS (actually, the gap between our CCWS algorithm and ICWS gradually narrows down to less than 1% when D increases to 256), the results still conform to our expectation that our CCWS algorithm keeps nearly the same accuracy as IWMH does and in turn strictly meets the definition of the Min-Hash scheme. In terms of computational time, our CCWS algorithm remarkably outperforms IWMH by $2 \sim 3$ orders of magnitude. In addition, our CCWS algorithm enjoys the similar efficiency as CCWS and even runs clearly faster with D varying from 2 to 32.

In summary, we compare the proposed CCWS algorithm with the state-of-the-arts, including IWMH and ICWS on four real-world data sets for document classification. The experimental results demonstrate that CCWS enjoys almost the same classification performance more efficiently (by $2 \sim 3$ orders of magnitude) as IWMH does, which shows that our CCWS algorithm strictly complies with the definition of the MinHash scheme. Compared to ICWS which violates the uniformity of the MinHash scheme, our CCWS algorithm achieves the competitive performance while keeping the similar and even higher efficiency. In particular, on the data set with large variances of weights, e.g., the Webspam data set, our CCWS algorithm achieves almost the same performance as ICWS. Therefore, our CCWS algorithm is a real MinHash algorithm with very practical efficiency for weighted sets.

4.6.3 Experiments for PCWS

Results on Classification

We investigate classification performance of the compared methods using LIBSVM [14] with 10-fold cross-validation on three binary classification benchmarks[7]:

1. **Real-sim**: The data set is a collection of UseNet articles from four discussion groups about simulated auto racing, simulated aviation, real autos and real aviation, respectively. The formatted document data set, with 72,309 samples and 20,958 features, has been arranged into two classes: real and simulated. Since the real class and the simulated class of the original data are unbalanced, data preprocessing is performed by randomly selecting 10,000 real samples and 10,000 simulated samples as positive and negative instances, respectively.

2. **Rcv1**: The data set is a large collection of newswire stories drawn from online databases. The formatted data set has 20,242 training samples with 47,236 features. The data set has been categorized into two classes: positive instances contain CCAT and ECAT while negative ones contain GCAT and MACT on the website. Similarly, we randomly select 10,000 positive instances and 10,000 negative ones to compose a balanced data set.

3. **KDD**: This is a large educational data set from the KDD Cup 2010 competition. The formatted data set has 8,407,752 training samples with 20,216,830 features. We also randomly select 10,000 positive instances and 10,000 negative ones to form a balanced data set for classification.

[7] Real-sim, Rcv1, Kdd and Webspam can be downloaded at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html

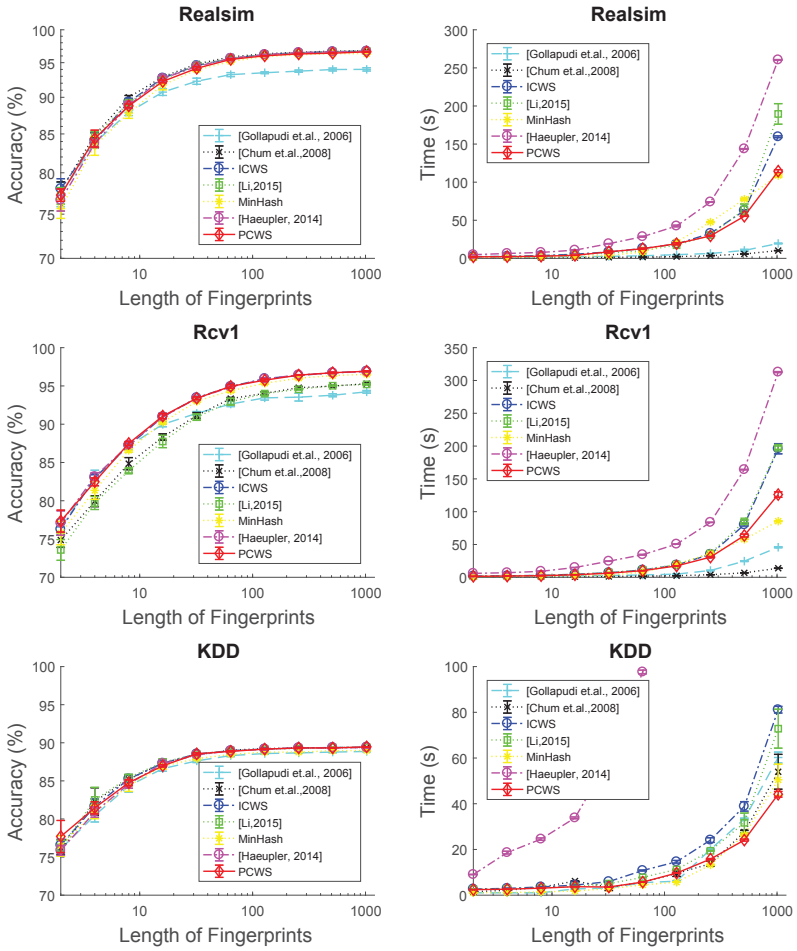


Fig. 4.5 Classification results in accuracy (left column) and runtime (right column) of the compared methods on Real-sim, Rcv1 and KDD. The x-axis denotes the length of fingerprints, D.

We repeat each experiment 5 times and compute the mean and the standard derivation of results.

Discussions on Real-sim: The subplots in the first row in Figure 4.5 show the comparison results on Real-sim. One can see that our PCWS algorithm achieves almost the same accuracy as ICWS, [Li, 2015] and [Chum et.al., 2008]. The comparison between PCWS and ICWS demonstrates that our PCWS algorithm indeed satisfies the CWS scheme. The reason that [Chum et.al., 2008] also performs similarly with [Li, 2015] may be the one discussed in [67], that is, one component of ICWS, y_k, is trivial to approximate the generalized Jaccard similarity for most data sets. In terms of runtime, our PCWS algorithm performs similarly with ICWS and [Li, 2015] when D ranges from 2 to 128, and subsequently, the gap

between PCWS and ICWS clearly widens. Particularly, PCWS takes around 3/4 of ICWS runtime and nearly 3/5 of [Li, 2015] runtime, when $D = 1024$.

Discussions on Rcv1: The subplots in the second row of Figure 4.5 show the comparison results on Rcv1. Our PCWS algorithm preserves almost the same accuracy as ICWS. Furthermore, the two CWS algorithms clearly perform better than the standard Min-Hash scheme and other algorithms which biasedly estimate the generalized Jaccard Similarity. In terms of runtime, our PCWS algorithm maintains the same level as ICWS and [Li, 2015] with D varying from 2 to 64. Our PCWS algorithm is remarkably superior to ICWS and [Li, 2015] when D varies from 128 to 1024. Again, the runtime of PCWS is reduced by a factor of 1/3 compared to ICWS and [Li, 2015] when $D = 1024$.

Discussions on KDD: In order to evaluate the classification ability on data with a large number of features (i.e., size of the universal set), we test the compared methods on the Kdd data set. The comparison results are reported in the subplots in the third row of Figure 4.5 (in this experiment, each algorithm is given a cutoff time of 100 seconds). Our PCWS algorithm still preserves the same accuracy as ICWS. In terms of runtime, PCWS runs much more efficiently than ICWS and [Li, 2015] this time on the data set with a large number of features. The performance gain of our PCWS algorithm in terms of runtime starts in the very beginning; as the length of fingerprint D increases, the gap becomes more significant: 1/3 faster than ICWS and [Li, 2015] when D approaches to 1024.

Results on Top-K Retrieval

In this experiment, we carry out top-K retrieval, for $K = \{1, 20, 50, 100, 500, 1000\}$. We adopt Precision@$K$ and Mean Average Precision (MAP)@K to measure the performance in terms of accuracy because precision is relatively more important than recall in large-scale retrieval; furthermore, MAP contains information of relative orders of the retrieved samples, which can reflect the retrieval quality more accurately. To this end, we select two large-scale public data sets:

1. **Webspam**: It is a web text data set provided by a large-scale learning challenge. The data set has 350,000 instances and 16,609,143 features. We randomly select 1,000 samples from the original data set as query examples and the rest as the database.

2. **Url** [82]: The data set contains 2,396,130 URLs and 3,231,961 features. We randomly select 1,000 samples from the original data set as query examples and the rest as the database.

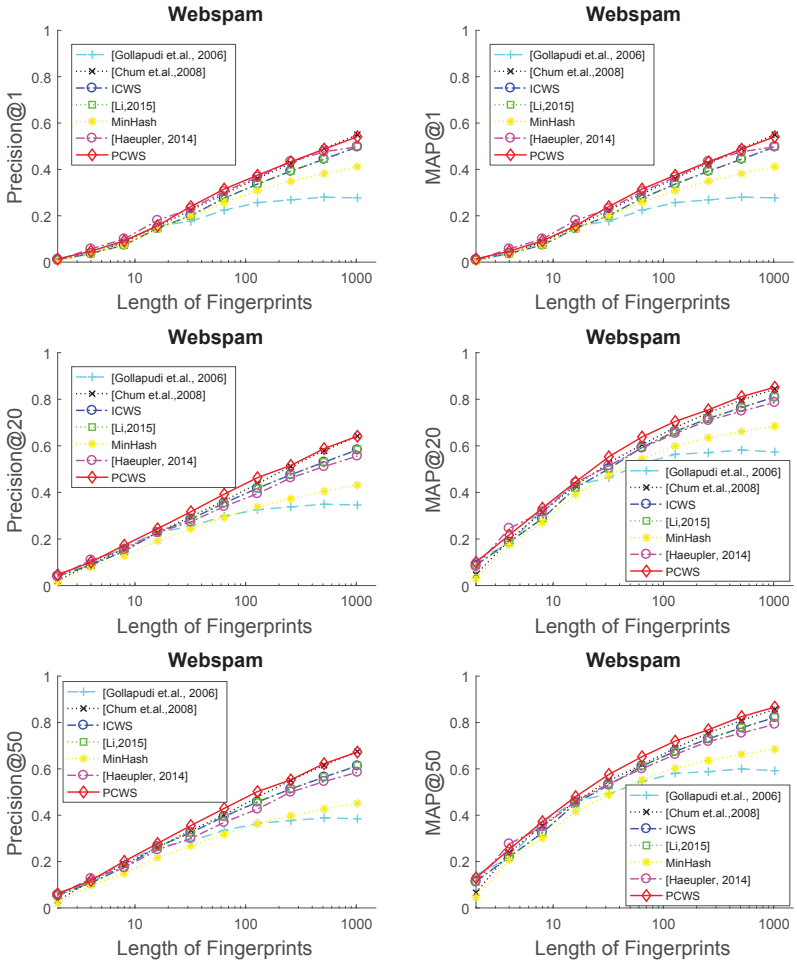


Fig. 4.6 Retrieval results in Precision@K (first columns) and MAP@K (second columns) of the compared methods on Webspam. The x-axis denotes the length of fingerprints, D.

Discussions on Webspam: Figures 4.6 and 4.7 report the comparison results on the Webspam data set which has more than 16 million features. We observe that our PCWS algorithm generally outperforms all the competitors under all configurations. Again, as the length of fingerprints D increases, the performance gain of PCWS compared to the other algorithms becomes clearer. PCWS outperforms ICWS and [Li, 2015] by about 5% when $D = 1024$; this improvement over its counterparts might be due to the positive effect of using one less random variable. PCWS performs much better than MinHash and [Gollapudi et.al., 2006] and it is superior to the two algorithms by around 25% and 40%, respectively, when $D = 1024$. PCWS also shows better performance than [Chum et.al., 2008] and [Haeupler et.al., 2014] in most cases. In the left subplot of Figure 4.10 for comparison of runtime, PCWS clearly runs faster than ICWS and [Li, 2015] as the length of fingerprints D

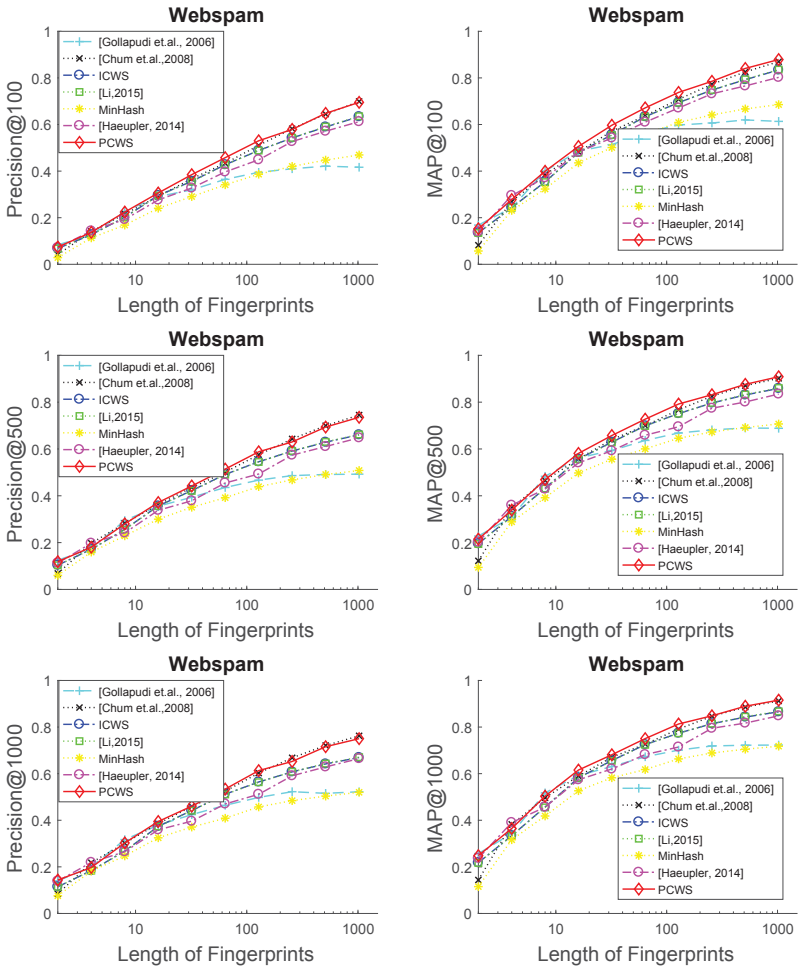


Fig. 4.7 Retrieval results in Precision@K (first columns) and MAP@K (second columns) of the compared methods on Webspam. The x-axis denotes the length of fingerprints, D.

increases. For example, PCWS performs approximately 30% faster than the two algorithms when $D = 1024$.

Discussions on Url: Figures 4.8 and 4.9 report the comparison results on the Url data set which has more than 2 million instances and over 3 million features (in this experiment, each algorithm is given a cutoff time of 40,000 seconds). Generally, our PCWS algorithm performs similarly with other algorithms, and even does better than ICWS with D ranging from 2 to 8. In the right subplot of Figure 4.10 for comparison of runtime, PCWS clearly outperforms ICWS and [Li, 2015]; in particular, PCWS runs nearly 30% faster than ICWS and around 1.25 times as fast as [Li, 2015] when $D = 1024$.

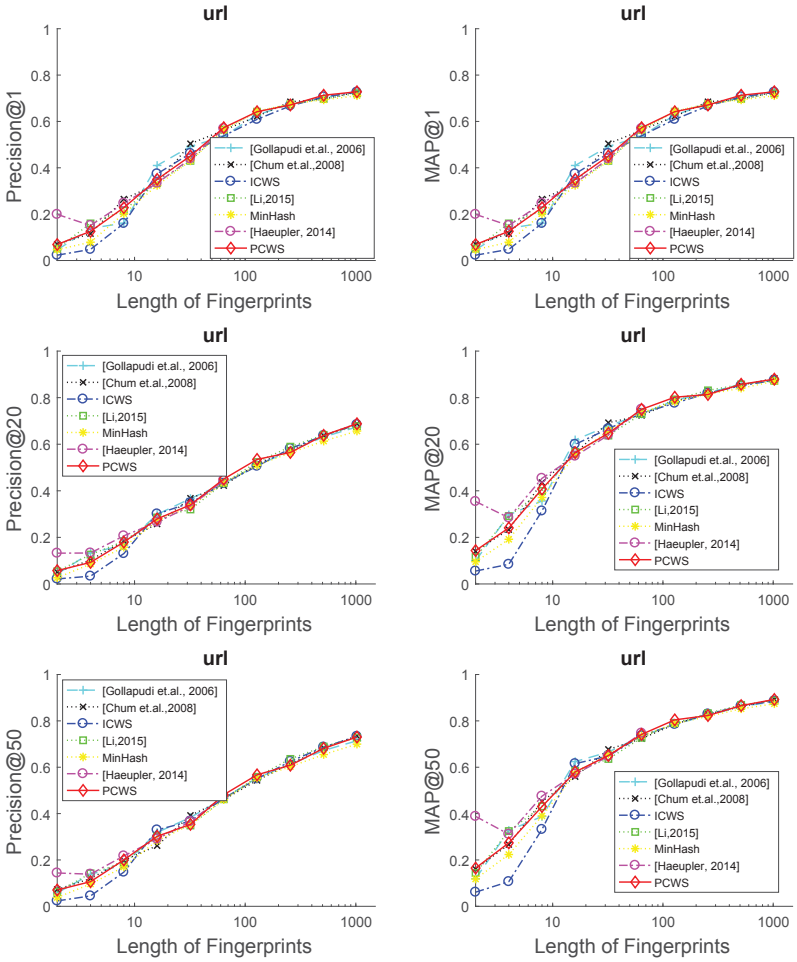


Fig. 4.8 Retrieval results in Precision@K (first columns) and MAP@K (second columns) of the compared methods on Url. The x-axis denotes the length of fingerprints, D.

Discussion on Space Efficiency

Finally, we analyze the advantage of our PCWS algorithm on space efficiency. Recall that our PCWS algorithm is able to save $\mathcal{O}(nD)$ of memory footprint (see Section 4.5.2), where n is the number of features and D is the length of fingerprints, because PCWS generates one less uniform random variable than ICWS does. This advantage in space complexity may not be remarkable when the size of the universal set n is small; however, when n is large in most cases of real-world data sets, the saved memory footprint can be substantial. For example, Kdd, Webspam, and Url have 20.2 million, 16.6 million and 3.2 million features, respectively, and our PCWS algorithm can enjoy 150 GB, 130 GB and 24 GB lower memory footprints in the case of $D = 1024$, compared to ICWS without accuracy degradation. If the number of features becomes even larger or more samples of MinHashes are used, the advantage of PCWS on space efficiency will be more remarkable. Obviously, our PCWS

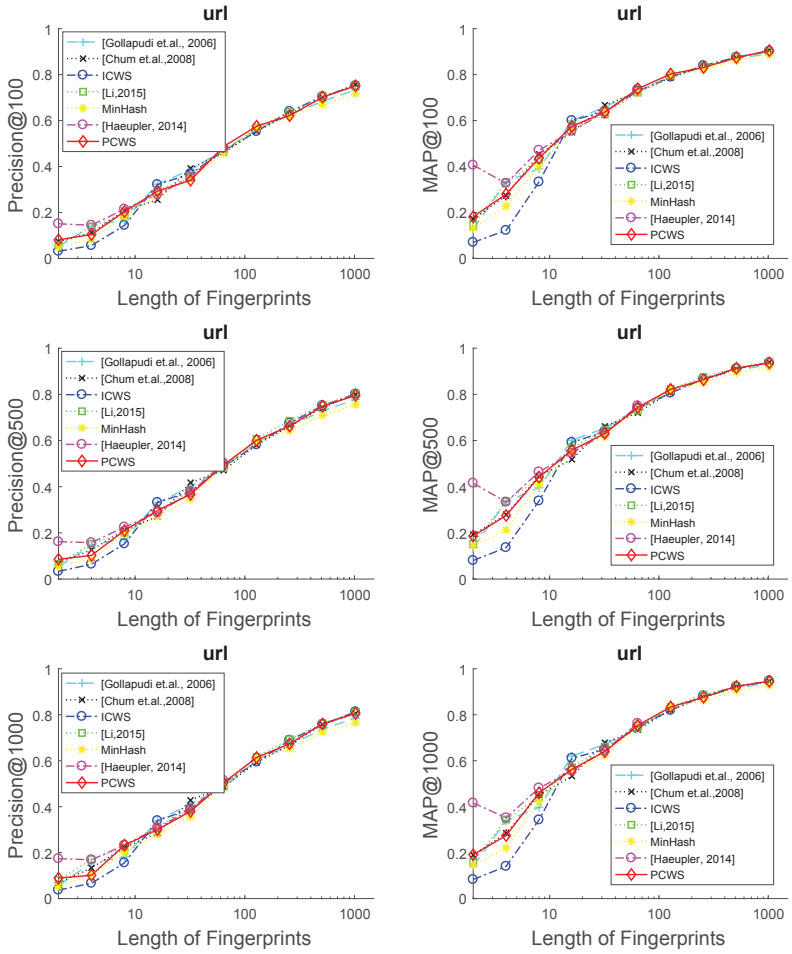


Fig. 4.9 Retrieval results in Precision@K (first columns) and MAP@K (second columns) of the compared methods on Url. The x-axis denotes the length of fingerprints, D.

algorithm significantly improves ICWS in terms of both time and space efficiency, which indicates that PCWS is more practical.

4.7 Related Work

The standard MinHash algorithm and its all variants are all designed to approximate the Jaccard similarity for binary sets. However, in the real-world applications, weighted sets are more common than binary sets. For example, each word in document analysis is assigend with a *tf-idf* value for its importance. To this end, the generalized Jaccard similarity was proposed for the weighted sets because the generalized Jaccard similarity is able to sketch sets more accurately than the Jaccard similarity [43], and subsequently, some weighted MinHash algorithms have been proposed to approximate the generalized Jaccard similarity.

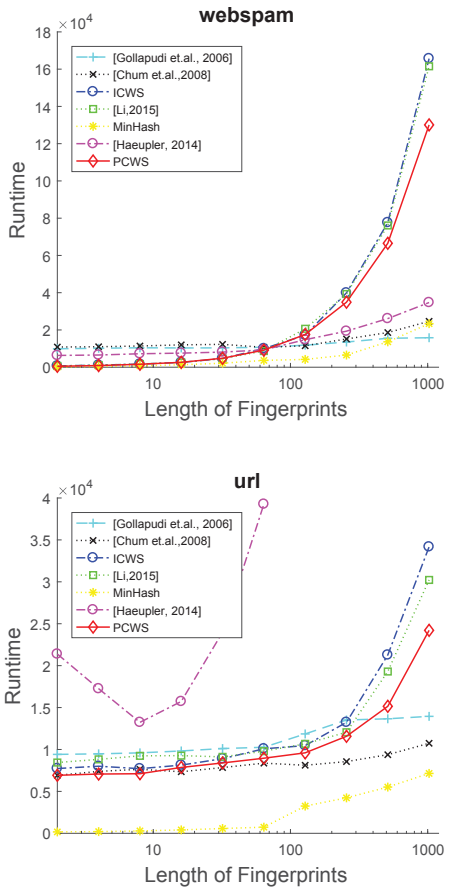


Fig. 4.10 Retrieval runtime of the compared methods on Webspam and Url. The x-axis denotes the length of fingerprints, D.

The naive idea of weighted MinHash algorithm is to quantize each weighted element into a number of distinct and equal-sized subelements, and then apply the standard MinHash to the collection of subelements. The remaining float part of each weighted element stemming from quantization can be operated by either rounding off or saving with probability [41]. Despite the feasibility, the quantization process explicitly increases the size of the universal set which significantly increases the computational workload because each subelement is independently permutated according to the definition of MinHash. In order to conquer the issue, the second method in [34] applies the standard MinHash to the binary sets which are transformed by thresholding real-value weights with random samples; by contrast, Chum et al. [21] compute the MinHash values for integer weighted sets by deducing the minimum of a set of random variables. Although the two approaches are efficient – the second method in [34] traverses the original set twice and permutates once for every MinHash value while

the method in [21] traverses only once for every MinHash value, there exist gaps between the expectation of the approximation and the true Jaccard similarity.

In order to further improve the effectiveness and efficiency, researchers have resorted to sampling-based methods. Shrivastava [111] proposes a weighted MinHash algorithm based on uniform sampling. Unfortunately, it requires to know the upper bound of each element in the universal set in advance, and thus it is not practical in real-world applications. As a milestone work, the first method in [34] proposes the idea of “active indices". The “active indices" are independently sampled on a weighted element as a sequence of subelements whose hash values are monotonically decreasing. Consequently, a large number of inactive subelements between two adjacent “active indices" are skipped, and the computational complexity is proportional to the logarithm of the weight. However, this method is still inefficient for real-value weighted sets because real-value weights must be transformed into integer weights by multiplying a large constant. Subsequently, the CWS scheme [84] is proposed to solve the efficiency problem for real-value weighted sets by considering only two special “active indices", one of which is the largest “active indices" smaller than the weight and the other of which is the smallest “active indices" greater than the weight. The algorithm in [84] still needs to traverse some “active indices" in order to find the two special ones, and thus it runs in expected constant time for weighted elements; while ICWS [49] runs in worst-case constant time by directly sampling the two special ones. Li [67] approximates ICWS by simply discarding one component of the MinHash values returned by ICWS, and empirically shows that the discarded component hardly affects performance.

4.8 Conclusion

In this chapter, we propose two real-value weighted MinHash algorithms, named Canonical Consistent Weighted Sampling (CCWS) and Practical Consistent Weighted Sampling (PCWS), both of which are based on the CWS scheme. By directly quantizing the original weights rather than the logarithm of the weights as ICWS, we guarantee that our CCWS algorithm is able to strictly comply with the definition of the MinHash scheme by uniformly sampling on the original weights. Based on such a canonical quantization strategy, we reestablish the mechanism between “active indices” and the minimum hash values. By contrast, our PCWS algorithm further improves the efficiency of ICWS by a simple replacement of the equivalent form. Our PCWS algorithm is mathematically equivalent to ICWS and preserves the same theoretical properties as ICWS, but with reduced theoretical complexity in both time and space.

We conduct extensive empirical tests of our CCWS algorithm, our PCWS algorithm and the state-of-the-art methods for document classification and information retrieval. We evaluate the effectiveness and efficiency on a number of real-world text datasets. The experimental results validate that our CCWS algorithm is able to not only achieve almost the same classification performance as IWMH (which implies that the CCWS algorithm strictly complies with the definition of the MinHash scheme as IWMH does), but also perform much faster than IWMH by $2 \sim 3$ orders of magnitude and keep the similar efficiency as ICWS; our PCWS algorithm is able to achieve the same (even better) performance than ICWS with $1/5 \sim 1/3$ reduced empirical runtime and 20% reduced memory footprint. In the cases of large number of features, PCWS can save hundreds of GB of memory footprint, which makes it more practical in dealing with real-world data sets in the era of big data.

Chapter 5

A MinHash-based Algorithm for Graphs

In Chapter 4, we generate weighted sets by assigning each element with a weight such that the weighted sets can contain more information. Consequently, we propose two MinHash-based algorithms in order to accurately compute the similarity between the weighted sets. Obviously, the data model definitely ignore the relationship between elements in the set. However, in the real-world, there exists another variety of structured data – graphs, which contain not only elements but also the dependence relationship between elements. In this chapter, we propose a MinHash-based algorithm for graph. By representing each graph as a low-dimensional feature vector, we can efficiently compute the similarity between the graphs.

5.1 Introduction

The surge of real-world graph data, such as chemical compounds, networks and social communities, has led to the rise of graph mining research [2, 4, 5, 20, 100, 155]. Graph classification is an important graph mining task that aims to train a classifier from training data to predict class labels of testing data, where both training and testing examples are graphs.

Due to the arbitrary structure of a graph, it is not easy to compute graph similarities because graphs are not in the same intrinsic space. The essential challenge for graph classification is to extract features from graphs and represent them as a vector in a common feature space for facilitating classifier training within a generic machine learning framework, such as support vector machines (SVM) [126] and logistic regressions.

In the past, there have been numerous works devoted to extracting substructures, e.g., walks [31, 56], paths [9], and subtrees [83, 107]. Graph comparison can be conducted in a common feature space spanned by the exhaustively enumerated substructures from the entire graph set. However, the substructure set expands exponentially with the increase of

the size of the graph set. Both the computational and spatial capabilities of lab computers will become insufficient for handling large-scale graph classification tasks engaging 10^5 or more graphs. So far, most studies on graph kernels and graph fingerprinting have been conducted on small benchmark graph sets with $10^2 \sim 10^3$ graphs. Even on such small graph sets, many classical graph kernel methods, such as the fast geometric random walk kernel [128], the *p*-random walk kernel [31, 56], and the shortest path kernel [9], may take hours or even days to construct a kernel matrix [108].

Recently, the Weisfeiler-Lehman (WL) graph kernel [108] achieves the best performance in terms of both accuracy and efficiency among the state-of-the-art graph kernel methods. It recursively relabels the subtree pattern composed of itself and its neighboring nodes over iterations as graph features, the number of which is only in linear in the number of edges. However, the WL graph kernel needs to maintain a global subtree pattern list in memory as a common feature space for representing all the graphs. As the size of the graph set increases and new subtree patterns appear, the efficiency of subtree insertion and search operations is dramatically slowed.

One solution to sketching high-dimensional (or even infinite-dimensional) data is to use hashing techniques [22, 53, 54, 67, 69, 73, 74, 109, 112–114, 134, 138–141, 148, 149], or similarly, random projections [1, 47] to map them into a fixed number of dimensions as an estimator of the original high-dimensional feature vectors. Nevertheless, most existing sketching algorithms are designed for vectors. Few studies on sketching graphs have been reported. To the best of our knowledge, only three works so far, [3, 18, 65], have exploited hashing techniques to accelerate graph feature extraction. In particular, [3] hashes the edge set of graphs, [65] hashes the set of node labels, and [18] hashes the set of cliques. Among the three methods, [3] and [18] cannot be applied to sketching labeled graphs as they are designed for sketching network flows. The Nested Subtree Hash (NSH) kernel, proposed in [65], employs a random hashing scheme to improve the efficiency of the WL graph kernel [108] in both time and space. However, it suffers from some performance loss compared to [108]. Thus, we aim to develop a technique which is able to achieve the same state-of-the-art classification performance as the WL graph kernel [108] with dramatically improved efficiency in both time and space.

To this end, we first observe an interesting connection between the WL relabeling process and *K*-ary trees, which motivates us to design an efficient data structure named *traversal table* for facilitating fast approximation of the subtree patterns in WL. Given a traversal table, which has $(K+1)$ columns, constructed from the node labels and edges of a graph, a subtree (i.e., *K*-ary tree) of depth *r* is efficiently acquired through *r* times of recursive matrix indexing. Moreover, a *K*-ary tree pattern can be uniquely specified by ordered leaf nodes.

We generate all K-ary tree patterns, rooted at individual nodes of a graph, of a specified depth and adopt the MinHash scheme to fingerprint the K-ary tree patterns. By virtue of recursive indexing, we obtain almost the same K-ary tree patterns as those found in the WL kernel, avoiding the expensive insertion and search operations.

We conduct extensive empirical tests of the proposed K-Ary Tree based Hashing (KATH) algorithm and the compared methods [65, 108] for fast graph feature extraction and in turn classification. We evaluate its effectiveness and efficiency on 17 real-world graph benchmarks including four large-scale real data sets (10^4) and a large-scale synthetic data set (10^4). Compared to the two state-of-the-art approaches, WL kernel [108] with the best classification performance and NSH [65] with the highest efficiency in both time and space, KATH not only achieves a similar classification performance to [108] but also performs much faster and used less space than [65]. More specifically, KATH

- achieves competitive accuracy with WL kernel, but with a dramatically reduced CPU time (around $1/3$ compared to WL on the real data set qHTS).

- outperforms NSH in accuracy and with significantly reduced CPU time (over $1/2$ of NSH on the real data set qHTS).

- can be scaled up to even larger graph sets due to its slowly and steadily increasing computational cost.

In summary, the proposed KATH algorithm is the most efficient graph feature extraction method for graph classification without observable performance loss, when compared to state-of-the-art methods.

The remainder of the chapter is organized as follows: Section 5.2 introduces the necessary preliminary knowledge. We introduce the KATH algorithm for fast graph feature extraction in Section 5.3. The experimental results are presented in Section 5.4. We review the related work in Section 5.5. Finally, we conclude the paper in Section 5.6.

5.2 Preliminaries

In this section, we first describe notations and the graph classification problem considered in the paper. Then, we briefly introduce two existing graph feature extraction methods, WL and NSH, which represent the state-of-the-art approaches to fast graph classification. Finally, we give a quick review of the MinHash scheme that will be used in our method.

5.2.1 Problem Description

Given a set of labeled graphs denoted by $\mathcal{G} = \{g_i\}_{i=1}^N$, where N denotes the number of graph objects. A graph g_i is represented as a triplet $(\mathcal{V}, \mathcal{E}, \ell)$, where $\mathcal{V}$ denotes the node set, $\mathcal{E}$ denotes the undirected edge set, and $\ell : \mathcal{V} \mapsto \mathcal{L}$ is a function that assigns labels from a label set $\mathcal{L}$ to the nodes[1] in g_i. The node label represents a certain attribute of the node. Each graph g_i in $\mathcal{G}$ is associated with a class label y_i, which represents a certain property of the graph determined by the structure of the graph and its node labels. Take chemical compounds for example: a molecule is represented as a graph, where its atoms form the node set and the bonds between atoms form the edge set. Each node in the molecule is assigned with a node label, that is, the name of atom (e.g., carbon and oxygen). The structure of the molecule and its node labels determine the chemical property of the molecule. Given the graphs in $\mathcal{G}$, a feature extraction (fingerprinting) process is performed to transform arbitrary graph structures into a set of feature vectors (fingerprints) $\{\mathbf{x}_i\}_{i=1}^N \in \mathcal{X}$, where each dimension of $\mathcal{X}$ usually corresponds to a substructure of graphs (e.g., subtrees [65, 107, 108]). The goal of graph classification is to learn a classifier from $\{\mathbf{x}_i, y_i\}_{i=1}^N$ to classify unseen graphs.

5.2.2 Weisfeiler-Lehman Graph Kernels

Weisfeiler-Lehman (WL) graph kernels [108] (or fast subtree kernels [107]) are known as the fastest graph kernel family, based on the Weisfeiler-Lehman (WL) isomorphism test [136]. WL kernels have a theoretical computational complexity that only scales linearly in the number of edges of a graph, $|\mathcal{E}|$, and the number of the WL isomorphism test iterations, R (considering subtree patterns of depth $R-1$). A WL graph kernel is written as follows

$$\kappa(g_i, g_j) \quad = \quad \sum_{r=1}^{R} \kappa(g_i^{(r)}, g_j^{(r)}) = \sum_{r=1}^{R} \langle \mathbf{x}_i^{(r)}, \mathbf{x}_j^{(r)} \rangle = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$$

where $g_i^{(1)} = (\mathcal{V}, \mathcal{E}, \ell^{(1)})$ and $\ell^{(1)} = \ell$. That is, $g_i^{(1)}$ is the same as the input graph g_i. $g_i^{(r)} = (\mathcal{V}, \mathcal{E}, \ell^{(r)})$ is the graph with a set of relabeled nodes in the r-th iteration. $\mathbf{x} = [\mathbf{x}^{(1)}; \ldots; \mathbf{x}^{(R)}]$, where $\mathbf{x}^{(r)}$ is the generated fingerprint in the r-th iteration, recording the number of subtree patterns of depth $(r-1)$.

To illustrate the relabeling process of WL to obtain the fingerprint $\mathbf{x}$, we use the 4-node labeled graph in Figure 5.1(a) as an example. Firstly, $g^{(1)}$ is the same as the given graph, so $\mathbf{x}^{(1)}$ in the first iteration records the number of subtree patterns of depth 0. As

[1] Edge labels can also be considered in a similar way. For simplicity, we only consider node labels in this paper.

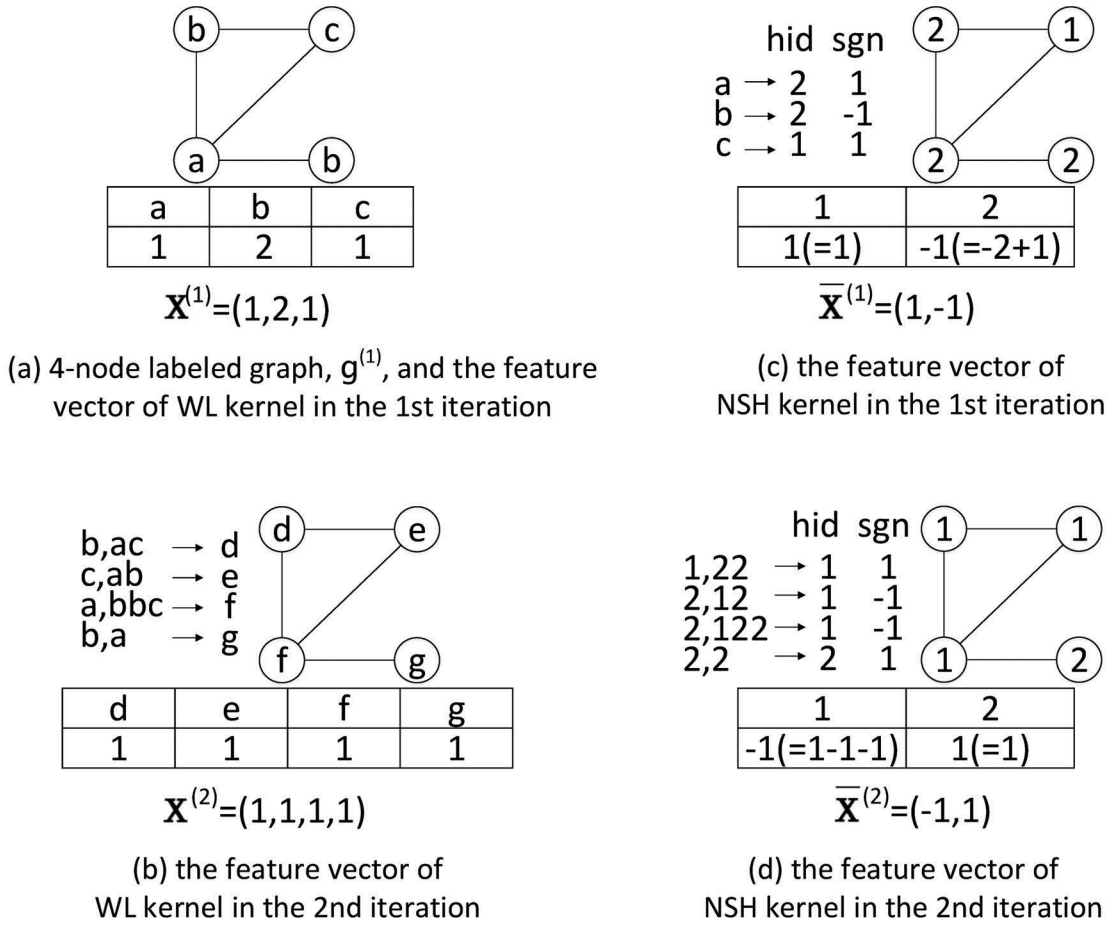


Fig. 5.1 Illustration of the WL kernel and the NSH kernel with $r = 2$.

shown in the table of Figure 5.1(a), there are three subtrees of depth 0: a, b and c, with the occurrence numbers as 1, 2 and 1 respectively. Therefore, we have $\mathbf{x}^{(1)} = (1,2,1)$. Figure 5.1(b) illustrates the relabeling process in the second iteration, which considers subtrees of depth 1. Thus, for each subtree in the first iteration, we consider its neighboring nodes. For instance, for the subtree rooted at node c in the first iteration, the corresponding subtree of depth 1 contains two leaf nodes a and b. We can encode the subtree as a string, which is initiated with the label of the root, followed by the ordered labels of the root's one-step neighboring nodes (e.g., c,ab). Each string representing a subtree pattern will acquire a new label for compression: if the subtree (or the string) has existed, it will directly get the label assigned to the subtree; otherwise, it will obtain a new label. As shown in Figure 5.1(b), since all subtree patterns of depth 1 are different, each node is assigned with a new label to denote the corresponding subtree. Similarly, a histogram $\mathbf{x}^{(2)}$ is used to count the numbers of the subtree patterns in $g^{(2)}$. In this case, $\mathbf{x}^{(2)} = (1,1,1,1)$.

The intuition of WL graph kernels is thus used to compare the "bags of subtrees" in a range of depths, that is, given two graphs, WL kernel computes the similarity of the graphs in terms of subtree patterns of different depths.

Although the theoretical computational complexity is only $O(NR|\mathscr{E}|+N^2R|\mathscr{V}|)$ for computing a WL graph kernel matrix, the WL isomorphism tests need to maintain a global subtree pattern list which spans the underlying graph feature space. In the cases of large-scale graph classification scenarios, the subtree pattern list will be dramatically expanded, and thus the search and insertion operations and subtree pattern storage for the list will be infeasible very soon because of both time- and spatial-inefficiency problems.

5.2.3 Nested Subtree Hash Kernels

A hash kernel based on the WL isomorphism test, say Nested Subtree Hash (NSH) kernels [65], for fast graph classification is proposed to address the time- and spatial-inefficiency of WL graph kernels. An NSH kernel is expressed in the following form,

$$\begin{aligned}\bar{\kappa}(g_i,g_j) &= \sum_{r=1}^{R}\bar{\kappa}(g_i^{(r)},g_j^{(r)}) = \sum_{r=1}^{R}\langle\phi(\mathbf{x}_i^{(r)}),\phi(\mathbf{x}_j^{(r)})\rangle \\ &= \sum_{r=1}^{R}\langle\bar{\mathbf{x}}_i^{(r)},\bar{\mathbf{x}}_j^{(r)}\rangle = \langle\bar{\mathbf{x}}_i,\bar{\mathbf{x}}_j\rangle\end{aligned}$$

where $\mathbf{x}_i^{(r)}$ and $\mathbf{x}_j^{(r)}$ denote the complete feature vectors in the r-th iteration (e.g., the same as the WL graph kernel), and ϕ is a linear projection, composed of two random functions, the hash *hid* and the bias-sign *sgn*, which maps $\{\mathbf{x}_i^{(r)}\}$ to $\{\bar{\mathbf{x}}_i^{(r)}\}$. In particular, *hid* is used to allocate a position for the subtree in the feature vector $\{\bar{\mathbf{x}}_i^{(r)}\}$ and *sgn* assigns signs to subtrees for subtree counting. Thanks to ϕ, the dimensionality of $\{\bar{\mathbf{x}}_i^{(r)}\}$ can be fixed without unlimited expansion; feature counting can be indexed by hashing without insertion and search.

For example, consider the 4-node graph in Figure 5.1(a) again. The first two iterations of NSH to find $\{\bar{\mathbf{x}}_i^{(1)}\}$ and $\{\bar{\mathbf{x}}_i^{(2)}\}$ are shown in Figures 5.1(c) $\sim$ (d). In Figure 5.1(c), two functions of ϕ, the hash *hid* and the bias-sign *sgn*, are imposed on each subtree. The purpose of *sgn* is to eliminate the bias of the estimator derived from the mapping from the complete feature spaces to the hashed low-dimensional feature spaces. For instance, given the three subtrees of depth 0, *a*, *b* and *c*, in Figure 5.1(a), the *hid* function in Figure 5.1(c) respectively hashes them to the positions 2, 2 and 1 in the feature vector $\bar{\mathbf{x}}^{(1)}$. However, since both subtrees *a* and *b* are hashed to 2, directly counting the number of subtrees mapped to the second position will produce significant errors. Since the *sgn* function assigns a negative

sign to the subtree b, the value of the second element in $\bar{\mathbf{x}}^{(1)}$ is then obtained as $(-2+1) = -1$. It has been proved theoretically that the *sgn* function can unbiasedly estimate the WL kernel from the perspective of statistics [65]. Afterwards, the original graph is relabelled with the hash values. Figure 5.1(d) shows the NSH kernel in the second iteration. Similarly, each subtree pattern of depth 1 (e.g., $1,22$) is relabelled via *hid*. New labels from *hid* are assigned to the corresponding nodes, and a feature vector $\bar{\mathbf{x}}^{(2)}$ is acquired.

As we can see from the illustrating example, by hashing subtree patterns of different depths, NSH avoids maintaining the list of subtrees and the expensive subtree insertion and search in WL, so that NSH improves the efficiency both in space and time. However, NSH suffers from some performance loss by mapping the complete feature space to the hashed low-dimensional feature space.

5.2.4 MinHash

We aim to design a graph feature extraction algorithm that is more efficient while maintains the same performance as the WL graph kernel. Before introducing our algorithm, we briefly review the MinHash scheme [12] which is used in our method. MinHash is an approximate method for measuring the similarity of two sets, say $\mathscr{S}_i$ and $\mathscr{S}_j$. A set of D hash functions (random permutations) $\{\pi_d\}_{d=1}^{D}$ are applied to the elements in $\mathscr{S}_*$ and we say $\min(\pi_d(\mathscr{S}_*))$ is a MinHash of $\mathscr{S}_*$. A nice property of MinHash is that the probability of $\mathscr{S}_i$ and $\mathscr{S}_j$ to generate the same MinHash value is exactly the Jaccard similarity of the two sets:

$$Pr\big(\min(\pi_d(\mathscr{S}_i)) = \min(\pi_d(\mathscr{S}_j))\big) = \frac{|\mathscr{S}_i \cap \mathscr{S}_j|}{|\mathscr{S}_i \cup \mathscr{S}_j|} = J(\mathscr{S}_i, \mathscr{S}_j)$$

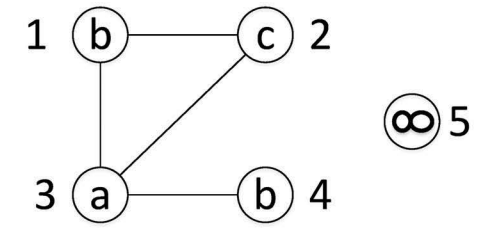
To approximate the expected probability, multiple independent random permutations are used to generate MinHashes. The similarity between two sets based on D MinHashes is calculated by

$$\hat{J}(\mathscr{S}_i, \mathscr{S}_j) = \frac{\sum_{d=1}^{D} \mathbf{1}\big(\min(\pi_d(\mathscr{S}_i)) = \min(\pi_d(\mathscr{S}_j))\big)}{D}$$

where $\mathbf{1}(state) = 1$, if $state$ is true, and $\mathbf{1}(state) = 0$, otherwise. As $D \to \infty$, $\hat{J}(\mathscr{S}_i, \mathscr{S}_j) \to J(\mathscr{S}_i, \mathscr{S}_j)$.

In practice, it is unnecessary to do an explicit random permutation on a large number. A permutation function as follows can be used to directly generate the permutated index

$$\pi_d(i) = \mathtt{mod}((a_d i + b_d), c_d) \tag{5.1}$$



(a) 4-node labeled graph with a dummy node as the 5th node

Algorithm 2

$\ell(3)<\ell(2)<\ell(5)$	1	3	2	5
$\ell(3)<\ell(1)<\ell(5)$	2	3	1	5
$\ell(1)=\ell(4)<\ell(2)$	3	1	4	2
$\ell(3)<\ell(5)=\ell(5)$	4	3	5	5
$\ell(5)=\ell(5)=\ell(5)$	5	5	5	5

(b) Naïve traversal table with $|\mathcal{V}| = 4$ and $\mathrm{K} = 3$

Algorithm 3

minhash(1) = [c a a]	1	2	3	5
minhash(2) = [b b a]	2	1	5	3
minhash(3) = [b c b]	3	1	2	4
minhash(4) = [a a a]	4	3	5	5
	5	5	5	5

(c) Minhash traversal table with $|\mathcal{V}| = 4$ and $\mathrm{K} = 3$

Fig. 5.2 Two examples of traversal table construction. The first column of the traversal tables are the five root nodes which are followed by their neighboring nodes sorted or min-hashed in terms of labels.

where i is the index of an item from the set $|\mathscr{S}|$, $0 < a_d, b_d < c_d$ are two random integers and c_d is a big prime number such that $c_d \geq |\mathscr{S}|$. Our method would benefit from this expression in cases with massive graph features.

5.3 K-ary Tree Hashing

In this section, we propose a K-Ary Tree based Hashing (KATH) algorithm to sketch the graph and extract graph features in terms of leaf node labels of a sequence of K-ary trees. The derived graph features have an equivalence to the subtree patterns used in the WL isomorphism test based algorithms [65, 107, 108]. Recall that in these algorithms, the relabeling process for node v in the r-th WL isomorphism test iteration only considers its direct neighboring node set $\mathscr{N}_v$ to assign a new label to the ordered label string $\ell^{(r)}(v, \mathtt{sort}(\mathscr{N}_v))$, which uniquely encodes a subtree pattern. In each WL isomorphism test iteration, the node

Algorithm 4 K-Ary Tree based Hashing

Input: $g = (\mathscr{V}, \mathscr{E}, \ell)$, K, $\{D^{(r)}\}_{r=1}^{R}$

Output: $\{x^{(r)}\}_{r=1}^{R}$

$V \leftarrow |\mathscr{V}|$

$\ell(V+1) \leftarrow \infty$

$\mathbf{T} \leftarrow (V+1) * \texttt{ones}(V+1, 1+K)$

for $v = 1 : V$ **do**

  $\mathscr{N}_v \leftarrow \texttt{neighbor}(v)$

  $\mathbf{T}(v) = \texttt{Selection\_Solution}(\ell, \mathscr{N}_v)$ // Use Algorithm 2 or 3

end for

$\mathbf{z}^{(1)} \leftarrow [1 : V]^\top$

$\mathbf{S}^{(1)} \leftarrow \ell(\mathbf{z}^{(1)})$

for $r = 1 : R$ **do**

  if $r > 1$ **then**

    $\mathbf{z}^{(r)} \leftarrow \texttt{reshape}(\mathbf{T}(\mathbf{z}^{(r-1)}, :), [1, *])$

    $\mathbf{S}^{(r)} \leftarrow \texttt{reshape}(\ell(\mathbf{z}^{(r)}), [V, *])$

  end if

  $\mathbf{f}^{(r)} \leftarrow \left[\hbar(\mathbf{S}^{(r)}(1,:)), \ldots, \hbar(\mathbf{S}^{(r)}(V,:))\right]^\top$

  $\mathbf{x}^{(r)} \leftarrow \left[\texttt{min}(\pi_1^{(r)}(\mathbf{f}^{(r)})), \ldots, \texttt{min}(\pi_{D^{(r)}}^{(r)}(\mathbf{f}^{(r)}))\right]^\top$

end for

label information is propagated at most one step on the graph and a node can receive the label information of those nodes $r-1$ steps away in the r-th iteration.

We observe an interesting connection between the relabeling process of the WL isomorphism test and K-ary trees: The relabeling process is similar to the breadth-first traversal of a virtual K-ary tree on a graph. Motivated by this observation, we wish to employ a mechanism similar to breadth-first traversal that can fast update node label information using matrix operations. The only difficulty is that the numbers of neighboring nodes of different nodes in different graphs are inconsistent, and this will make the traversal matrices of different graphs get inconsistent number of columns (i.e., inconsistent encoding rules which will be detailed later) so that the resulting graph fingerprints of different graphs will be incomparable. If we can find solutions to avoid the problem of inconsistent number of columns in traversal matrices, the graph feature (subtree pattern) encoding can be dramatically accelerated through a recursive indexing operation. To this end, we adopt two alternative solutions: a naive solution that fixes the number of neighboring nodes to be considered, and a MinHash based scheme which projects any numbers into a fixed size vector. We compare the

Algorithm 5 Naive Selection

Input: ℓ, $\mathcal{N}_v$

Output: $\mathbf{T}(v)$

1: $temp \leftarrow \texttt{sort}(\ell(\mathcal{N}_v))$

2: $k \leftarrow \texttt{min}(K, |\mathcal{N}_v|)$

3: $\mathbf{T}(v) \leftarrow \left[v, \texttt{index}(temp(1:k))\right]$

Algorithm 6 MinHash Selection

Input: ℓ, $\mathcal{N}_v$

Output: $\mathbf{T}(v)$

1: $temp \leftarrow \left[\texttt{min}(\pi_1(\ell(\mathcal{N}_v))), \ldots, \texttt{min}(\pi_K(\ell(\mathcal{N}_v)))\right]$

2: $\mathbf{T}(v) \leftarrow \left[v, \texttt{index}(temp)\right]$

performance of the two solutions empirically in Section 5.4. In the following section, we elaborate on our KATH algorithm which resembles WL isomorphism test based methods but is much more efficient.

5.3.1 The Algorithm

The KATH algorithm processes graphs one by one. The input of the algorithm includes a graph $g = \{\mathcal{V}, \mathcal{E}, \ell\}$ and some parameters: K denotes the size of the traversal table and $\{D^{(r)}\}_{r=1}^{R}$ are the MinHashes for R iterations (or depth $R-1$), where D is the number of MinHash functions, i.e., the length of fingerprints. The parameters of the permutation functions $\{\pi_d^{(r)}\}$ for MinHashes like Eq. (5.1) are randomly generated online [2]. All the aforementioned parameters are set to the fixed values for all processed graphs.

We outline the KATH algorithm in Algorithm 4. The algorithm mainly comprises three steps: (1) Traversal Table Construction (Lines 1–7), which constructs an indexing structure based on the node labels and edges in g using two alternative approaches; (2) Recursive Leaf Extension (Lines 8–9 & 12–13), which recursively extends the new leaves of all full K-ary trees based on the obtained indexing structure to approximate subtrees; (3) Leaf Sequence Fingerprinting (Lines 15–16), which uses the MinHash scheme to fingerprint the leaf sequences (subtree patterns) of g. In the following, we will introduce the KATH algorithm in details in three steps, with a running example based on the toy graph in Figure 5.2.

[2]Note that, the input MinHash parameters are for the MinHash that fingerprints leaf node labels of K-ary trees, instead of the MinHash that projects neighbors of a node into a fixed size vector.

Traversal Table Construction

After receiving a graph $g = \{\mathscr{V}, \mathscr{E}, \ell\}$, the algorithm first adds a dummy node as the $(V+1)$th node of g (for example, node 5 in Figure 5.2(a)), where $V = |\mathscr{V}|$ (Line 1), and assigns ∞ as its label (Line 2). The dummy node is necessary because, if the number of neighboring nodes of the root node v is smaller than the size of the traversal table, K, we can use the dummy node to fill the vacancy left in the traversal table, and a virtually full K-ary tree can still be completed. The reason why we use ∞ as its label is that the dummy node can be placed after all the real nodes in terms of sorted labels.

We allocate a compact $(V+1) \times (1+K)$ matrix (Line 3) for storing the indices of each nodes and their children nodes. We refer to this compact traversal matrix as the traversal table, denoted by $\mathbf{T}$. In $\mathbf{T}$, the first V rows correspond to the V nodes in g and the last row is for the dummy node; the first column represents all nodes, and the last K columns record the neighbors of corresponding nodes in the graph. Since we maintain only K columns of the traversal table, while the number of neighbors of a node may be more than K, we adopt two alternative solutions, Algorithm 5 and Algorithm 6, to address the problem.

Given the set of neighbors of node v as $\mathscr{N}_v$, Algorithm 5 is a naive solution that only considers the first K neighbors. Essentially, we first sort the nodes in $\mathscr{N}_v$ in terms of their labels in ascending alphabetical order. If two nodes have the same label, they will be sorted by their node indices, and this guarantees that all identical strings are mapped to the same number (Line 1). We then determine the number of nodes k for selection. $k = K$ if the number of neighboring nodes of v is larger than K; $k = |\mathscr{N}_v|$ otherwise (Line 2). Last, v and the indices of the first k nodes in the sorted array are stored in the v-th row of $\mathbf{T}$ (Line 3). If $(k+1) < (K+1)$, the vacancy entries are filled with the index of the dummy node. Certainly, the naive solution suffers from some information loss if the number of neighbors of a node in a graph is usually greater than K. However, with graph data from some particular domains such as chemical compounds, where the number of neighbors of a node is usually no greater than 4, this simple solution will achieve effective performance by setting K as 4, demonstrated by our experimental results in Section 6.4.

Alternatively, we may adopt the MinHash scheme to project any number of neighbors to a vector of size K. The idea of the solution is outlined in Algorithm 6. We apply K MinHashes to the set of labels of a node's neighbors (Line 1). Then, v and the indices of nodes carrying corresponding labels generated by K MinHashes are stored in the v-th row (Line 2). Note that, if the result of K MinHashes includes repeated labels, the corresponding node indices in ascending order are stored, or if all matching nodes are exhausted, the dummy node is used. This guarantees that no repetitive real node indices emerge in the same row.

Examples of traversal table construction using the two alternative solutions are illustrated in Figure 5.2(b) and 5.2(c). The first column of two traversal tables store the indices of the four nodes in the graph and one dummy node. The last three columns store the indices of the nodes generated by the two solutions from the corresponding neighbor sets, respectively. Figure 5.2(b) represents naive traversal table. Consider node 1, $\mathcal{N}_1 = \{2,3\}$ and the labels of the node's neighbors are sorted as $(\ell(3) = a) < (\ell(2) = c)$. Hence, node 3 is placed in the second entry while node 2 in the third entry. The last vacancy entry is filled with the index of the dummy node. Figure 5.2(c) shows MinHash traversal table. Consider node 1 with $\mathcal{N}_1 = \{2,3\}$ with labels of $\{\ell(2) = c, \ell(3) = a\}$. Suppose $\{c,a,a\}$ is generated by applying the $K = 3$ MinHash functions to the label set $\{c,a\}$. For the first MinHash value c, since there exists only one neighbor of node 1 that carries the label c, node 2 is placed in the second entry. Both the second and the third MinHash values are a. However, there is only one neighbor, node 3, which has the label a. Hence, 3 is stored in the third entry and the index of the dummy node is stored in the fourth entry.

Obviously, both the parameter of the traversal table size K and the method used to select neighbors will affect the effectiveness of KATH. We will discuss the influence of the neighbor selection methods and the value of K in Section 5.4.

Recursive Leaf Extension

For ease of explanation, in this subsection and the next subsection, we take the naive traversal table as an example. The traversal table $\mathbf{T}$ constructed in the first step is used for a very fast subtree pattern extraction process. In particular, the $(r-1)$-depth full K-ary tree rooted at v can be represented as the leaf node labels of a sequence of r iterations originated from v, where the leaf node labels can be collected through a recursive indexing operation on $\mathbf{T}$ without explicitly recording the traversals.

Let $\mathbf{z}^{(r)}$ store the indices of the leaf nodes of all r iterations originated from the V root nodes. It is initialized with the V root nodes on g as the root nodes of the full K-ary trees, say $\mathbf{z}^{(1)} = [1:V]^\top$ (Line 8). Next, the algorithm goes into the iterations of recursive indexing operation for $r = 2,\ldots,R$ which recursively generate the leaf node labels of all $(r-1)$-depth full K-ary trees originated from the V root nodes. We use the example graph in Figure 5.2 to illustrate this operation:

$$
\begin{aligned}
\mathbf{z}^{(1)} &= [\underline{1}234] \\
\mathbf{z}^{(2)} &= \big[\underline{[1\bar{3}25]}[2315][3142][4355]\big] \\
\mathbf{z}^{(3)} &= \Big[\underline{[[1325]\overline{[3142]}[2315][5555]]}\,[\ldots]\,[\ldots]\,[\ldots]\,[\ldots]\Big]
\end{aligned}
$$

The above equations illustrate a recursive indexing operation for generating all full 2-depth 3-ary trees on the example graph. From the underlined node $\underline{1}$ in $\mathbf{z}^{(1)}$, one 1-depth 3-ary tree with the leaf nodes in the four underlined nodes $\underline{[1325]}$ in $\mathbf{z}^{(2)}$ are generated. These 3 ordered leaf nodes uniquely specify the 1-depth subtree rooted at node 1. Similarly, from the overlined node $\overline{3}$ in $\mathbf{z}^{(2)}$, one 1-depth 3-ary trees with the leaf nodes in the four overlined nodes $\overline{[3142]}$ in $\mathbf{z}^{(3)}$ are generated. If we consider the two breadth-first iterations together, we can obtain one 2-depth 3-ary tree from each root node. For example, the 16 underlined nodes in $\mathbf{z}^{(3)}$ are the leaf nodes of one 2-depth 3-ary tree originated from node 1. Again, these 12 ordered leaf nodes uniquely specify the 2-depth subtree rooted at node 1. More iterations of 3-ary trees can be obtained through this recursive indexing operation. Since all leaf nodes will be used as the indices to select rows from the traversal table $\mathbf{T}$ and the selected rows are catenated into a single row vector (Line 12), we refer to this iteration as recursive leaf extension.

Although $\mathbf{z}^{(r)}$ can be used to specify the $(r-1)$-depth subtree pattern rooted at v, the leaf node information from all root nodes is connected together. We should retrieve the corresponding labels of the nodes indexed by $\mathbf{z}^{(r)}$, say $\ell(\mathbf{z}^{(r)})$, evenly split $\ell(\mathbf{z}^{(r)})$ into V parts, and rearrange them into V rows to form a $V \times (1+K)^r$ label matrix $\mathbf{S}^{(r)}$ (Line 13). Thus far, the label array in the v-th row of $\mathbf{S}^{(r)}$ directly specifies the $(r-1)$-depth subtree rooted at node v. The label matrix of $\mathbf{z}^{(2)}$ in the example graph is in the following form

$$\mathbf{S}^{(2)} = \texttt{reshape}(\ell(\mathbf{z}^{(2)}), [4, *]) = \begin{bmatrix} bac\infty \\ cab\infty \\ abbc \\ ba\infty\infty \end{bmatrix}$$

The advantages of the recursive indexing operation include: all $(r-1)$-depth K-ary trees can be generated very fast through several matrix indexing operations without search or matrix addition; the leaf node labels of all $(r-1)$-depth K-ary trees originated from node v uniquely specify the $(r-1)$-depth subtree rooted at v.

Leaf Sequence Fingerprinting

So far, we have extracted all $(r-1)$-depth subtree patterns from g, and stored the corresponding encoding information in $\mathbf{S}^{(r)}$. To compute the similarity between the graphs based on the extracted subtree patterns encoded as an array of $(1+K)^r$ node labels, we can do the following two hashing operations. First, we employ a random hashing function, $\hbar : str \mapsto \mathbb{N}$, to hash the label array in each row of $\mathbf{S}^{(r)}$ into an integer as an identity to represent an $(r-1)$-depth subtree pattern (Line 15), which helps to relabel the subtree patterns rapidly.

The obtained subtree pattern index vector $\mathbf{f}^{(r)}$ comprises the $(r-1)$-depth subtree patterns of the V nodes on g. Second, we can view $\mathbf{f}^{(r)}$ as a multi-set[3] of the identities of V $(r-1)$-depth subtrees. A natural approach to comparing the similarity of two subtree sets is to fingerprint them using the MinHash scheme, instead of the simple random hash functions in NSH because MinHash is able to store as much information as possible between graphs, and it is suitable for substructure comparison in graph classification. By virtue of the random permutation function without explicit permutations, we can directly obtain the permutated positions of the identities in $\mathbf{f}^{(r)}$, say $\pi_d^{(r)}(\mathbf{f}^{(r)})$, using Eq. (5.1) and find the smallest one as a MinHash $\texttt{min}(\pi_d^{(r)}(\mathbf{f}^{(r)}))$. In the r-th iteration, $D^{(r)}$ random permutations are performed on $\mathbf{f}^{(r)}$ and $D^{(r)}$ MinHashes can be obtained to form the fingerprint of g in the r-th iteration, denoted by $\mathbf{x}^{(r)}$ (Line 16). In practice, the two hashing operations (Lines 15 and 16) can be performed quickly in matrix operations.

We still take the example in Figure 5.2 for illustration. The node label arrays in the v-th row of the label matrix $\mathbf{S}^{(2)}$ encode a subtree pattern. To identify the underlying subtree pattern using an integer, we apply the random hash function $\hbar$ to each row of $\mathbf{S}^{(2)}$ as follows

$$\mathbf{f}^{(2)} = \begin{bmatrix} \hbar(bac\infty) \\ \hbar(cab\infty) \\ \hbar(abbc) \\ \hbar(ba\infty\infty) \end{bmatrix} = \begin{bmatrix} 7 \\ 3 \\ 5 \\ 2 \end{bmatrix}$$

where the first row of $\mathbf{S}^{(2)}$, corresponding to the 1-depth subtree pattern rooted at node 1, is hashed to 7 by $\hbar(bac\infty) = 7$, where 7 is the identity of the underlying subtree pattern. Now, $\mathbf{f}^{(2)}$ comprises a set of 4 subtree patterns and $D^{(2)}$ permutation functions can be applied to $\mathbf{f}^{(2)}$. For example, the d-th MinHash $x_{\cdot,d}^{(2)} = \texttt{min}(\pi_d^{(2)}(\mathbf{f}^{(2)}))$.

Finally, we construct the kernel matrix $\mathbf{K}$ on $\mathscr{G}$ based on the obtained fingerprints $\{\mathbf{x}_1^{(r)}, \ldots, \mathbf{x}_N^{(r)}\}_{r=1}^{R}$. The kernel between g_i and g_j is

$$\mathbf{K}_{i,j} = \kappa(g_i, g_j) = \sum_{r=1}^{R} \sum_{d=1}^{D^{(r)}} \frac{\mathbf{1}(x_{i,d}^{(r)} = x_{j,d}^{(r)})}{D^{(r)}} \tag{5.2}$$

where $\mathbf{1}(state) = 1$ if $state$ is true and $\mathbf{1}(state) = 0$ otherwise. Eq. (5.2) calculates the ratio of the same MinHashes between $\mathbf{x}_i^{(r)}$ and $\mathbf{x}_j^{(r)}$, which is exactly the Tanimoto kernel [98].

[3]It is a multi-set since the same subtree pattern may appear multiple times.

5.3.2 Complexity Analysis

The theoretical computational complexity of the KATH algorithm for fingerprinting a graph set is $O(NR|\mathscr{V}|)$, compared to $O(NR|\mathscr{E}|)$ for feature vector construction in WL and NSH. In particular, $O(NR|\mathscr{V}|)$ is for fingerprinting N graphs in R iterations (Lines 10–17 in Algorithm 4), where each graph requires $O(|\mathscr{V}|)$ for hashing V label arrays and computing their permutated positions. Another computational advantage of the KATH algorithm is that the length of fingerprints (10^2) is smaller than the dimensionality of feature vectors ($10^5 \sim 10^7$) in WL by several orders of magnitudes, which makes kernel construction more efficient.

Although the theoretical complexities of the three algorithms (WL, NSH, and KATH) are similar, the KATH algorithm is much more computationally efficient than the two other algorithms in practical implementation since it is able to directly generate the encoding information of subtree patterns through a recursive indexing process without search and matrix addition. In contrast, the NSH algorithm needs to update each dimension of a feature vector and the WL graph kernel needs additional cost for subtree pattern insertion and search (not counted in its computational complexity).

An advantage of the KATH kernel, in spatial complexity, is that no additional cost is required to store subtree patterns. This is similar to NSH, but the WL graph kernel requires a complexity of $O(NR|\mathscr{E}|)$ in the worst case to maintain a global subtree pattern list in the memory, which keeps increasing as new graphs are fed in. Moreover, KATH can further reduce spatial complexity in caching fingerprints that are significantly shorter than feature vectors used in two other algorithms before computing the kernel.

5.4 Experiments

In this section, we report the performance of KATH and its competitors on both real-world and synthetic data sets.

We use four groups of real-world data sets, two of which are used in [108] and [65], respectively. One group comes from PubChem[4]. The fourth group consists of two data sets from social networks: DBLP [92] and twitter [93]. We also generate a synthetic graph classification data set. Specifically, we adopt the following benchmarks (17 graph data sets in total):

[4] http://pubchem.ncbi.nlm.nih.gov/

5.4.1 Experimental Preliminaries

The classification performance of the compared methods is evaluated using `libsvm` [15] with 10-fold cross validation. We construct the kernel matrices generated by the compared methods and plug them into C-SVM in `libsvm` for classifier training. We repeat each experiment 5 times and average the results (accuracy and CPU time). All the experiments are conducted on a node of a Linux Cluster with 8×3.1GHz Intel Xeon CPU (64 bit) and 126G RAM.

We only compare KATH with existing graph sketching methods that are scalable to large graph classification problems, say 10^4 graphs with tens of labeled nodes. The comprehensive graph classification study in [108] shows that the WL graph kernel proposed therein (also known as fast subtree kernels [107]) easily outperform all the compared classical graph kernels, including the fast geometric random walk kernel [128], the p-random walk kernel (a special case of [31, 56]), and the shortest path kernel [9], in both accuracy and efficiency. By contrast, most of the compared classical graph kernels require hours or even days to process thousands of graphs. The NSH graph kernel is an estimator of the WL graph kernel using random hashes to improve efficiency in both time and space. NSH improves the computational and spatial efficiency of the WL isomorphism, but loses some classification accuracy. Therefore, we compare KATH with the two state-of-the-art graph kernels, i.e., WL and NSH.

We evaluate two variations of KATH. KATH-naive uses a naive method to construct the traversal table, and KATH-MinHash uses the MinHash scheme to build the traversal table.

Since all compared methods involve iterations which have underlying connections to the WL isomorphism test, we evaluate the algorithms by varying the number of iterations from 1 to 5. In other words, we let $Depth \in \{0, \ldots, 4\}$, where $Depth$ denotes the depth of subtrees. In the NSH algorithm, the dimensionality setting follows $[30, 500, 5000, 10000, 10000, 10000]$ used in [65] for the hashed feature spaces in 5 iterations. In the two variations of the KATH algorithm, we set the default size of the traversal table to 4 according to our empirical tests. We investigate the number of MinHashes (length of fingerprints) in $\{100, 200, 300\}$ and find that the two variations of KATH generally achieved the best performance when the number of MinHashes is set to 300. Besides, the subtrees of depth 0, i.e., $r = 1$, are individual nodes, which are considered to be non-discriminative subtree patterns. Therefore, we only report performance with the number of MinHashes as 300 and the depth from 1 to 4. The parameters of the hash functions for random permutations are randomly generated.

5.4.2 Data Sets

Mix [108]: The first group comprises 3 real-world graph data sets[5] used as the testbed in [108]: (1) MUTAG consists of 188 mutagenic aromatic and heteroaromatic nitro compounds, where the classification task is to predict whether they have a mutagenic effect on the Gram-negative bacterium Salmonella typhimurium. (2) ENZYMES consists of 600 enzymes represented in protein tertiary structures, which are required to be classified into one of the 6 EC top-level classes. (3) D&D consists of 1,178 protein structures, where the classification task is to predict whether a protein structure is an enzyme or non-enzyme.

NCI [65]: The second group comprises 9 NCI data sets[6] used as the testbed in [65]. Each data set has $28,000 \sim 42,000$ chemical compounds, each of which is represented as a graph whose nodes are atoms and edges are bonds. Each data set belongs to a bioassay test for predicting anti-cancer activity against a certain kind of cancer, where a graph (chemical compound) is labeled positive (active) if it is active against the cancer. Since the positive and negative examples of the original data are unbalanced (about 5% positive samples), following [65], data preprocessing is performed in advance by randomly selecting a negative subset from each data set with the same size as the positive one (e.g., 2,232 positive and 2,232 negative samples in NCI1).

qHTS[7]: This is a relatively large data set for graph classification. It comprises 362,359 chemical compounds, each of which is represented as a graph. They are tested in a bioassay test to predict the inflammasome activations. A graph is labeled positive (active) if it is active against inflammasome. As in NCI, for the sake of balance, data preprocessing is performed in advance by randomly choosing 10,000 positive and 10,000 negative instances.

NCI-Yeast[8]: This is another relatively large data set for graph classification. It comprises 86,130 chemical compounds, each of which is represented as a graph. They are tested in a bioassay test to measure their ability to inhibit the growth of yeast strains. A graph is labeled positive (active) if it can inhibit yeast growth. Again, data preprocessing is performed in advance by randomly choosing 10,000 positive and 10,000 negative instances.

DBLP [92]: Each record in DBLP represents one paper, containing paper ID, title, abstract, authors, year of publication, venue, and references of the paper. Following [92], each paper named as a core paper is represented as a graph, whose nodes denote the paper ID or a keyword in the title. If there exists citation between the core paper and another, the paper ID and all keywords will be added to the graph as nodes. Edges are produced as

[5] http://mlcb.is.tuebingen.mpg.de/Mitarbeiter/Nino/WL/Data.zip
[6] https://sites.google.com/site/libin82cn/nshk-data&codes.zip
[7] http://pubchem.ncbi.nlm.nih.gov/assay/assay.cgi?aid=743279
[8] http://pubchem.ncbi.nlm.nih.gov/assay/assay.cgi?aid=175

follows: 1) citation between two paper IDs corresponds to one edge in the graph; 2) two nodes, one representing paper ID and the other representing a keyword in the paper title, corresponds to one edge in the graph; 3) all nodes representing keywords of a paper are fully connected with each other as edges in the graph. Core papers in computer vision (CV) are labeled as positive; while core papers in database/data mining (DBDM) and artificial intelligence/machine learning (AIML) are labeled as negative. Consequently, there are 9,530 positive instances and 9,926 negative instances. We adopt this unbalanced data set in the experiments.

Twitter [93]: Following [93], each tweet is represented as a graph with nodes being terms and/or smiley symbols (e.g, :-D and :-P) while edges being the co-occurrence relationship between two words or symbols in each tweet. Furthermore, each tweet is labeled as a positive feeling or a negative one. Consequently, we obtain 67,987 positive and 76,046 negative tweets. We randomly sample an unbalanced data set (9,425 positive instances and 10,575 negative instances) with the same ratio of positive and negative instances as the original data set.

Synthetic 10K: The last is a synthetic graph data set. We use a synthetic graph data generator[9] to generate 10,000 labeled, undirected and connected graphs, named Syn10K20L10E, where "20L" indicates that the number of unique node labels in the graph set is 20 while "10E" indicates that the average number of edges in each graph is 10. To simulate graph classification tasks, we perform the following steps: 1) we find a set of frequent subgraph patterns (support > 0.02) using gSpan[10] [146] from each graph set; 2) we represent each graph as a feature vector, with a dimension being 1 if it has the corresponding subgraph, otherwise it being 0; 3) we perform 2-Means clustering for each graph set and labeled the graphs in one cluster positive and the other negative. As a result, we obtain one relatively large graph classification task.

The statistics of the above graph data sets are summarized in Table 2.3.

5.4.3 Mix

Figure 5.3 shows experimental results on the first group of real-world data, including MUTAG, ENZYMES and D&D. In MUTAG and ENZYMES, the accuracy performance of KATH-naive is better than that of NSH and WL from the depth of 2 to 4, while on D&D, KATH-naive performs worse than WL but better than NSH. In terms of runtime, KATH-naive significantly outperforms its competitors on all settings of depth and all three data sets. Specifically, it performs at least 2.5 times as fast as NSH and WL on MUTAG, and

[9] http://www.cais.ntu.edu.sg/~jamescheng/graphgen1.0.zip
[10] http://www.cs.ucsb.edu/~xyan/software/gSpan.htm

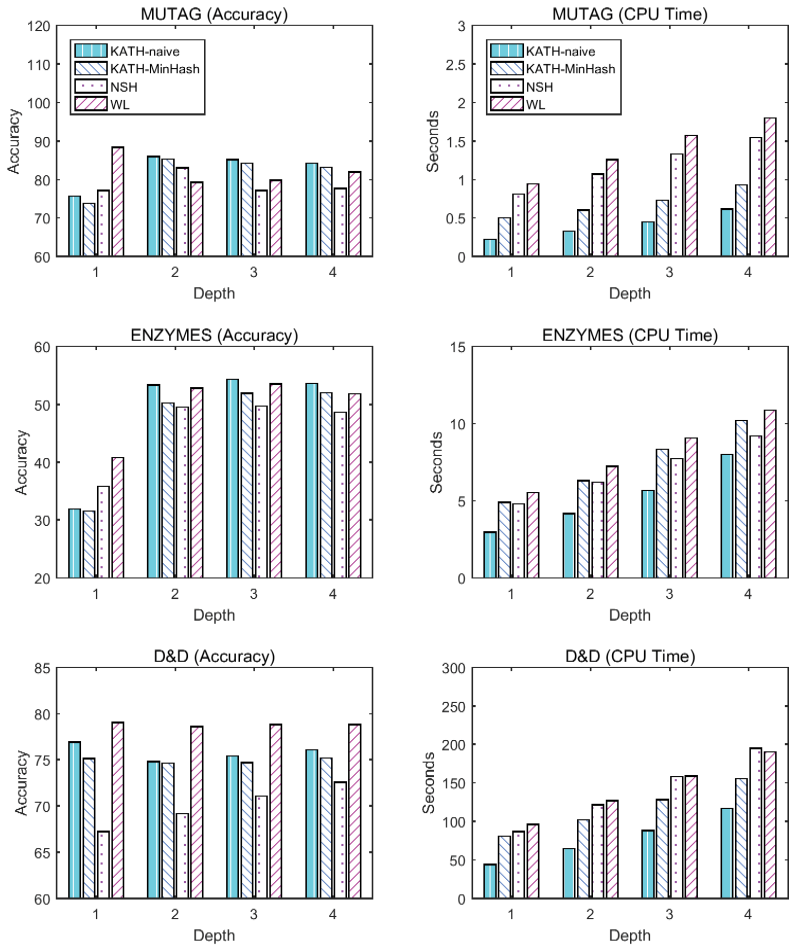


Fig. 5.3 Classification Accuracy (upper row) and CPU Time (bottom row) comparison results on the three graph data sets in the group of Real-World Mix. The x-axis denotes the depth of subtrees.

at least 40% faster than NSH and WL on D&D. KATH-MinHash generally outperforms NSH on the three data sets, but is inferior to KATH-naive. It also runs more slowly than KATH-naive, which indicates that the MinHash technique is not suitable for storing neighborhood information in the traversal table with small-scale graph data. Intuitively, this is because KATH-naive is able to save nearly all information about small-scale data without compression, while MinHash introduces losses from compression. Overall, KATH-naive achieves competitive accuracy compared to the state-of-the-art algorithm, WL, in a much shorter runtime.

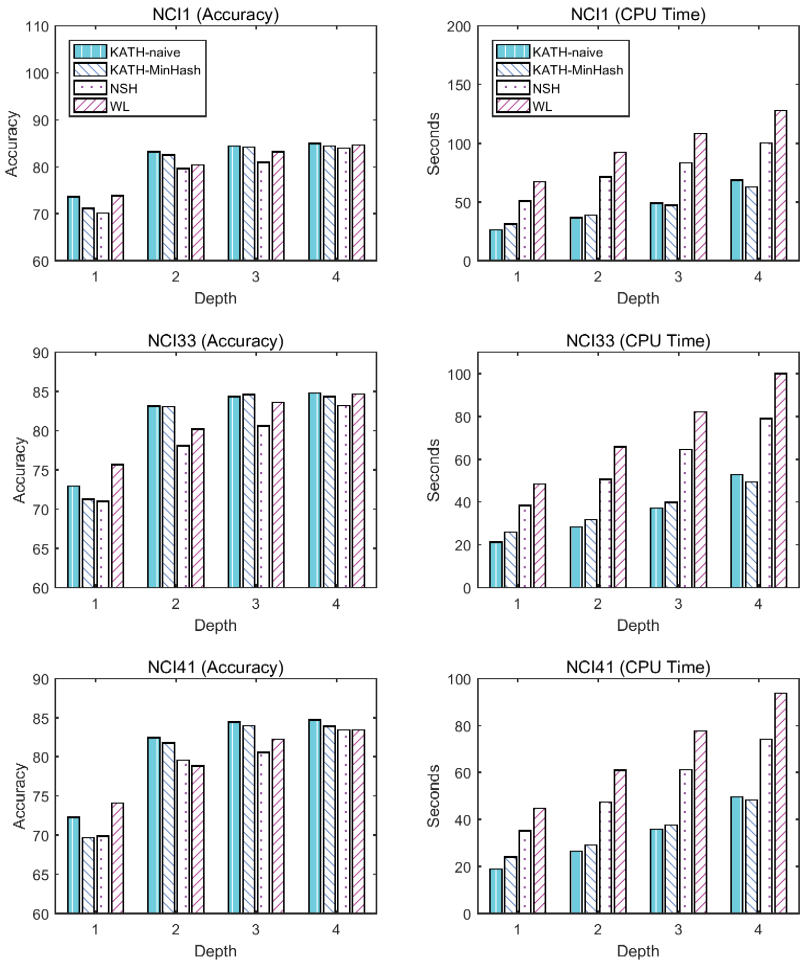


Fig. 5.4 Classification Accuracy (odd rows) and CPU Time (even rows) comparison results on the nine graph data sets in NCI. The x-axis denotes the depth of subtrees.

5.4.4 Results on NCI

Figures 5.4, 5.5 and 5.6 report the experimental results on nine NCI data sets. The two variations of KATH generally outperform their competitors from the depth of 2 to 4 in terms of both accuracy and time. Specifically, KATH-naive performs at least twice as fast as WL. Furthermore, we observe that KATH-naive and KATH-MinHash achieve similar accuracy. Compared to the results on the first group of real-world data from Section 5.4.3, KATH-MinHash becomes more effective when the scaling size of the data sets increases. This also conforms to the above intuitive explanation.

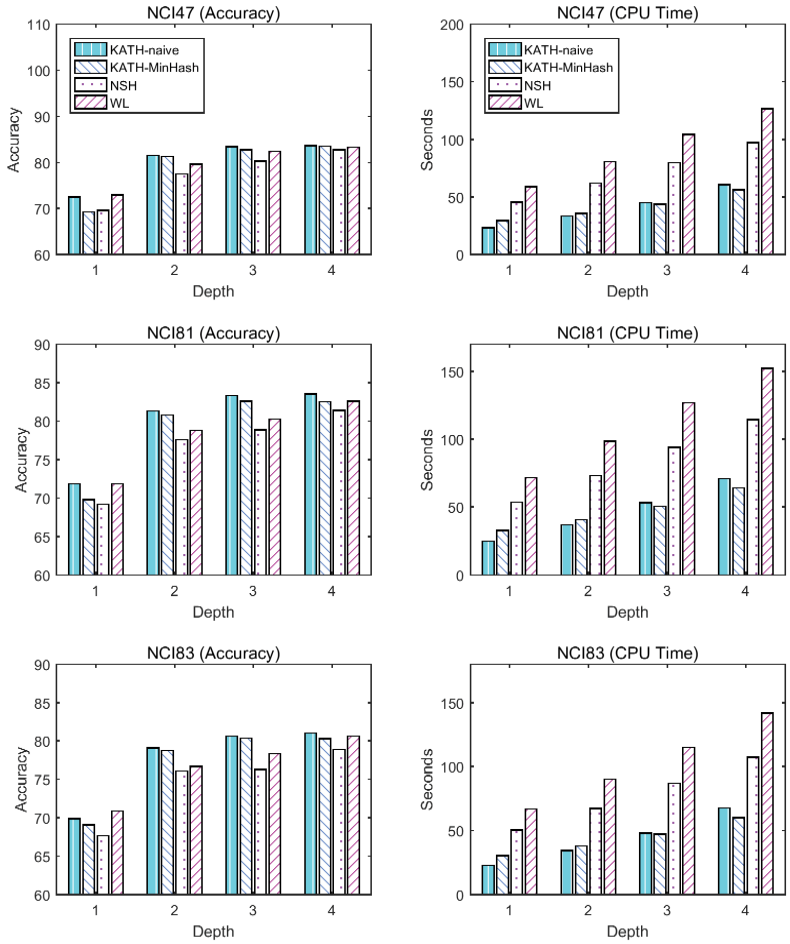


Fig. 5.5 Classification Accuracy (odd rows) and CPU Time (even rows) comparison results on the nine graph data sets in NCI. The x-axis denotes the depth of subtrees.

5.4.5 Results on qHTS

We further compare the two variations of KATH with its competitors on a large data set, qHTS. The experimental results are reported in Figure 5.7. Considering the fact that the subtrees with the depth of 1 and 2 are not discriminative in large-scale data and the traversal tables cause information loss, the performance of our algorithms is worse than WL. Thus we have only reported the results where the depth is 3 and 4 in this and the next subsections. KATH-MinHash achieves slightly better accuracy than KATH-naive from the depth of 3 to 4. As the scale of data increases, the number of substructure features becomes larger. Therefore, using the MinHash scheme in the traversal table would capture information more accurately. Furthermore, we note that the runtime of KATH-MinHash only increases slowly with respect to the increase of the subtree depth. For example, when the depth of subtrees increases, the gap between KATH-naive and KATH-MinHash narrows. Although the two

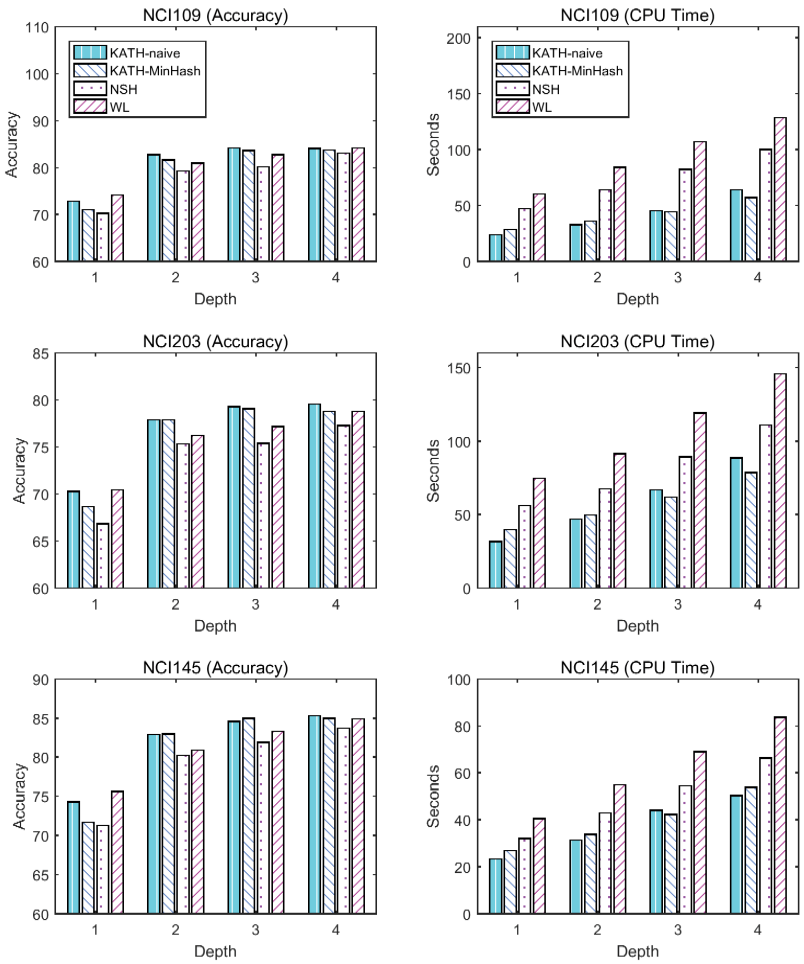


Fig. 5.6 Classification Accuracy (odd rows) and CPU Time (even rows) comparison results on the nine graph data sets in NCI. The x-axis denotes the depth of subtrees.

variations of KATH are inferior to WL (around 5%) in terms of accuracy, they perform much faster than WL. This is largely because KATH approximates the subtree patterns in WL and some performance is lost. Taking KATH-MinHash as an example, it runs approximately 1.5 times faster than NSH, and around 3 times faster than WL. Therefore, the results clearly show the suitability of KATH in classifying large graph data sets.

5.4.6 Results on NCI-Yeast

Figure 5.8 shows the experimental results on a second large dataset, NCI-Yeast. The two variations of KATH significantly outperform NSH when the depth is 3 and 4 in terms of both accuracy and time. Compared to results in Section 5.4.5, KATH-MinHash performs remarkably better than KATH-naive in terms of accuracy, which indicates that KATH-MinHash is

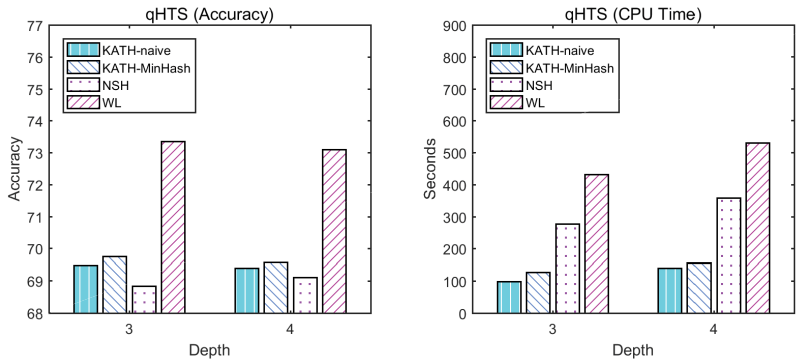


Fig. 5.7 Classification Accuracy (left) and CPU Time (right) comparison results on qHTS. The x-axis denotes the depth of subtrees.

more suitable for large graph data with more labels. This is because the MinHash scheme used to construct the traversal table is imposed on the node labels, and a small number of labels give rise to large compression loss. Although the two variations of KATH show slightly worse performance than WL (about 3%), in terms of accuracy, they run much faster. Taking KATH-MinHash as an example, it performs around 1.5 times as fast as NSH, and runs more than 2.5 times as fast as WL.

5.4.7 Results on DBLP

In addition to the graphs of chemical compounds, we have also investigated the graphs obtained from social networks. Figure 5.9 shows the experimental results on DBLP. Our KATH algorithms remarkably outperform NSH in terms of both accuracy and time. Compared to the results in Section 5.4.6, KATH-naive achieves almost the same performance as KATH-MinHash. The reason of this phenomenon might be that DBLP has a large number

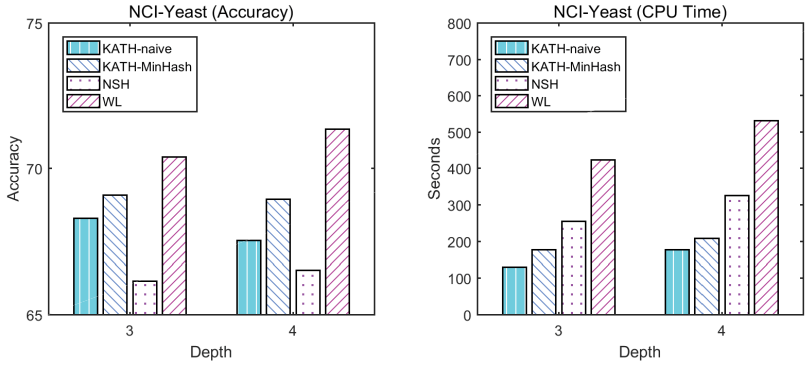


Fig. 5.8 Classification Accuracy (left) and CPU Time (right) comparison results on NCI-Yeast. The x-axis denotes the depth of subtrees.

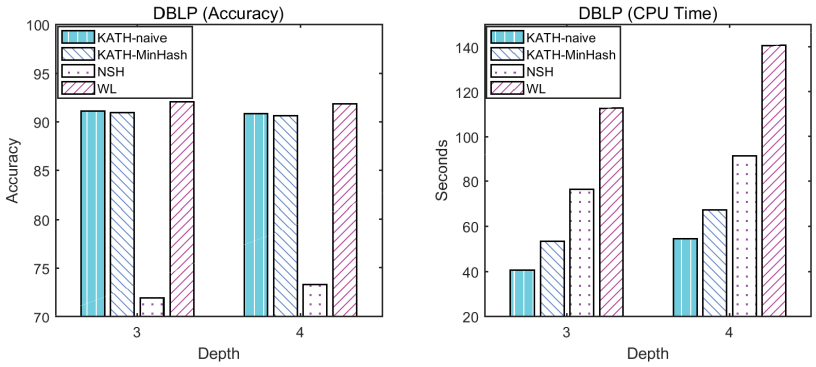


Fig. 5.9 Classification Accuracy (left) and CPU Time (right) comparison results on DBLP. The x-axis denotes the depth of subtrees.

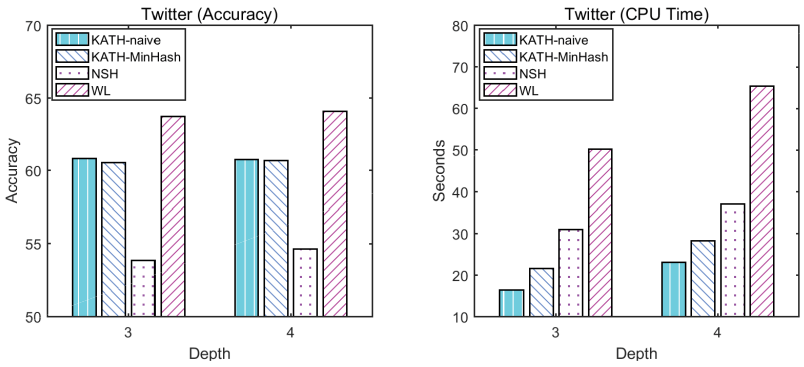


Fig. 5.10 Classification Accuracy (left) and CPU Time (right) comparison results on Twitter. The x-axis denotes the depth of subtrees.

of labels while the size of the traversal table is much smaller than the number of labels ($K = 4 \ll |\mathscr{L}| = 41,325$). Consequently, no matter what construction method of the traversal table is adopted, very little information is retained. In this experiment, WL performs slightly better (about 1%) than our KATH algorithms, however both KATH algorithms are superior to WL in terms of time. In particular, KATH-naive and KATH-MinHash only take around 1/3 and 1/2 of WL's runtime, respectively.

5.4.8 Results on Twitter

We also report the experimental results on the second graph data set of social network, Twitter, in Figure 5.10. Our KATH algorithms are significantly superior to NSH in terms of both accuracy and time. The reason that KATH-naive performs similarly with KATH-MinHash may be the same as that discussed in Section 5.4.7, that is, the size of the traversal table is much smaller than the number of labels ($K = 4 \ll |\mathscr{L}| = 1,307$). Likewise, in this experiment WL outperforms our KATH algorithms by around 4% in terms of accuracy;

however our KATH algorithms clearly outperform WL in terms of time. In particular, the runtime of KATH-naive and KATH-MinHash is reduced by a factor of around 2/3 and 1/2 compared to WL, respectively.

5.4.9 Results on 10K Synthetic Graph Sets

In our previous experiments, we find that the two variations of KATH generally perform best when the depth of subtrees is 4, so we only investigate KATH-naive, KATH-MinHash, and other algorithms by setting the subtree depth to 4 in the experiments on the synthetic data set. Figure 5.11 illustrates the experimental results on the synthetic data set of size $10K$. The two variations of KATH achieve much better accuracy performance with KATH-naive clearly outperforming NSH, KATH-MinHash significantly outperforming NSH and slightly beating WL. According to the difference in performance achieved by the two variations of KATH on the synthetic data set, we can conclude that KATH-MinHash, again, is more suitable for graph data with more labels. Also, we note that the two variations of KATH run much faster than their competitors: KATH-naive takes around $1/3$ of NSH runtime, and approximately $1/4$ of WL runtime; KATH-MinHash runs about $1/3$ faster than NSH, and about twice as fast as WL. The results again demonstrate the advantage of KATH on relatively large data in terms of runtime.

5.4.10 Impact of Traversal Table Size on KATH

Furthermore, we carry out experiments on the nine NCI datasets to analyze the influence of the traversal table size K in KATH graph kernel. In our previous experiments, we find that KATH achieves the best performance on NCI when the depth is set to 4 and the number of MinHashes is set to 300. Therefore, in this experiment, we set $Depth$ to 4, D to 300, and

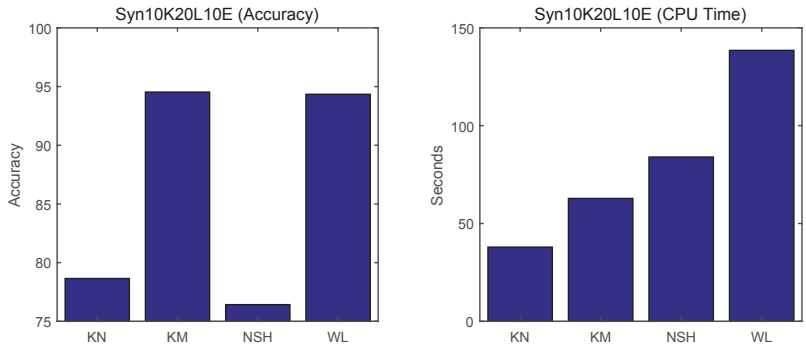


Fig. 5.11 Classification Accuracy and CPU Time comparison results on Synthetic 10K20L10E (KN short for KATH-naive, and KM short for KATH-MinHash).

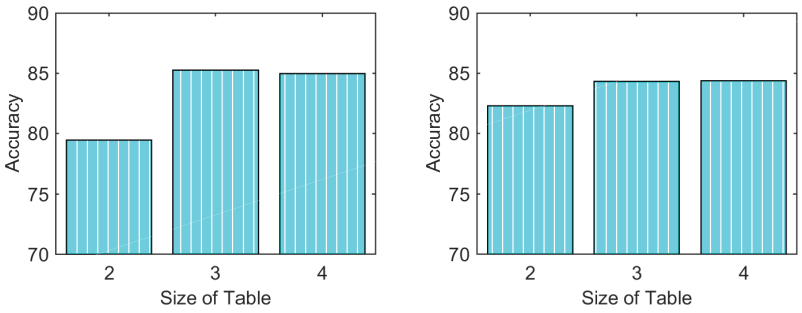


Fig. 5.12 Classification Accuracy comparison results on Real-World NCI1. The x-axis denotes the size of traversal table. The left figure represents KATH-naive, and the right figure represents KATH-MinHash.

Table 5.1 Summary of distribution of No. neighbors for NCI.

No. Neighbors	No. Nodes	Percentage
1	268,893	21.18%
2	550,193	43.33%
3	403,387	31.77%
4	47,254	3.72%

vary K from 2 to 4. Since the results on the nine NCI datasets show similar trends, we report only the experimental results on NCI1.

Figure 5.12 presents experimental results on NCI by varying the traversal table size. The accuracy performance of the two variations of KATH clearly gets improved when K varies from 2 to 3. Intuitively, when only two neighbors are considered, the traversal table will lose a large portion of graph information, which in turn results in performance decline. The accuracy performance of the two variations of KATH is nearly unchanged when K varies from 3 to 4. This can be explained by the distribution of the number of neighbors on the dataset. As shown in Table 5.1, there are only 3.72% of nodes owning four neighbors - much less than those having two and three neighbors. Therefore, a traversal table with a size of 3 only loses a tiny portion of the graph information compared with that of size 4.

5.5 Related Work

Graph feature extraction (and fingerprinting) is closely related to graph kernel design for measuring graph similarity. A large number of graph kernels have been proposed in the last decade, most of which are based on the similar idea of extracting substructures from graphs to compare their co-occurrences [42]. Typical substructures for graph representation

include walks [56, 128], paths [9], subtrees (neighboring nodes) [46, 83, 107], and subgraphs (usually based on a frequent subgraph mining technique, e.g., [28, 146]). The walk-based approach [56] constructs a kernel between graphs with node labels and edge labels within the framework of the marginalized kernel, where the walk comparison can be reduced to a system of linear simultaneous equations. A unified framework integrating random walk and marginalized kernels was proposed by [128]. The kernels for labeled graphs, can be computed efficiently by extending linear algebra. The path-based approach [9] constructs a kernel between graphs based on the shortest paths. Although the kernel can be computed in polynomial time, it is still not applicable to large-scale graphs due to expensive time cost. On the other hand, the kernel only takes the attributes of the starting and ending nodes into consideration, which means that this kernel loses some precious information in the graphs. The WL graph kernel can easily outperform the three kernels [108]. The subtree-based approach [83] constructs a family of graph kernels by detecting common subtree patterns as features to represent graphs. This kernel presents good performance on toxicity and anticancer activity prediction for small graph models, but it could not solve large graphs with high degrees in average. In [107], a fast subtree kernel was designed to deal with graphs with node labels based on the WL isomorphism test; however, the kernels only run on limited graph database. The neighborhood kernel in [46] adopts binary arrays to represent node labels and uses logical operations to process labels, but the complexity is proportional to the number of nodes times the average degree of nodes. By contrast, [28, 146] both focus on how to mine frequent substructure patterns in graph data instead of graph classification.

There have also been several studies on fast graph classification using hashing techniques. Actually, hashing has been one of the most effective solutions for approximately nearest neighbor search and similarity search, and in turn contributes to the kernel designs in [130]. A 2-dimensional hashing scheme is employed to construct an "in-memory" summary of the graph streams in [3]. The first random-hash scheme is used to reduce the size of the edge set, while the second MinHash scheme is used to dynamically update a number of hashes, which is able to summarize the frequent patterns of co-occurrence edges in the graph stream observed thus far. Recently, [18] proposed to detect clique patterns from a compressed network obtained via random edge hashing, and the detected clique patterns are further hashed to a fixed-size feature space [134]. Nevertheless, the two hashing techniques aim to deal with network flows and thus cannot be applied to arbitrary labeled graphs with node labels. To the best of our knowledge, there have been two studies on hashing techniques for graph sketching. One is NSH, which is extended from the fast subtree kernel [107] by employing a random hashing scheme for recursively hashing tree structures. The other is MinHash Fingerprints (MHF) [122], which first enumerates all the paths (between a cer-

tain range of orders) as a pattern set to represent each graph and then employs a standard MinHash scheme for fingerprinting. Compared to MHF, our algorithms adopt efficient matrix operations to extract subtree patterns, which are much more effective than paths for graph classification. Also, some studies have been conducted on hashing tree-structured data [35, 121]. The hierarchical MinHash algorithm [35] proceeds from bottom to top of the tree, where the information of labels obtained by fingerprinting is propagated to an upper level of the tree. In contrast, our KATH algorithms extend the tree from the root node. After new leaf nodes are generated via extension, the algorithm will fingerprint the new pattern to encode the information of labels on each node and the tree structure. On the other hand, [35] is only applicable to trees where only leaf nodes own labels such that it cannot be directly adopted in our problem setting, where each node of the graph has a label. In [121], a subtree pattern called embedded pivot substructure is proposed, which is composed of two nodes and their lowest common ancestor in the tree. A tree can thus be represented as a bag of such embedded pivots, and subsequently, the MinHash algorithm is applied to the bag. Unfortunately, in order to obtain all embedded pivots in the tree, the algorithm requires a computational complexity of $O(n^2)$, where n is the number of nodes in the tree. Obviously, compared to the KATH algorithm that is linear in the number of nodes, it can hardly be scaled up to deal with large-scale graph data sets.

5.6 Conclusion

In this chapter, we propose a graph feature extraction algorithm, called KATH, which makes use of K-ary trees to approximate the relabeling process of the WL kernel with dramatically reduced computation time and negligible space. In order to build K-ary trees, we design an efficient data structure named a traversal table. Our algorithm adopts two alternative solutions for selecting neighbors for the traversal table, and employs a recursive indexing process that only performs r times of matrix indexing to generate all $(r-1)$-depth K-ary trees, whose leaf nodes can be ordered to uniquely specify an $(r-1)$-depth K-ary tree pattern. By virtue of the recursive indexing process, we can quickly obtain almost the same K-ary tree patterns as those found in the WL kernel, without global insertion and search.

We conduct extensive empirical tests of KATH and compare it to state-of-the-art methods for fast graph classification, i.e., WL and NSH. We evaluate its effectiveness and efficiency on 16 real-world graph benchmarks and 1 large-scale synthetic graph set (10^4). The experimental results show that KATH not only achieves similar classification performance to WL but also performs much faster and uses less space than WL and NSH, and KATH-MinHash is more suitable for relatively large data with many node labels.

Chapter 6

A MinHash-based Algorithm for Network Nodes

In Chapter 5, we propose the MinHash-based algorithm for graphs, i.e., the K-Ary Tree Hashing (KATH) algorithm. Consequently, the graph can be represented as a low-dimensional feature vector, which contributes to fast graph classification. Also, we observe that there exists the dependence relationship between nodes in the graph. In order to study each node in the graph, especially a large network, recently, a large number of researchers are devoted to network node embedding, which aims to embed each node in the network into a low-dimensional vector. Essentially, the two problems share the idea of embedding, that is, each data object (i.e., graph or node in the network) is mapped to a low-dimensional vector space and furthermore, the similar data objects are embedded into the nearby points in the vector space. The KATH algorithm transforms the graph into a low-dimensional feature vector where each dimension consists of a set of substructure patterns. Therefore, in this chapter, we will investigate how to represent each node in the network as a low-dimensional vector via the MinHash algorithm.

6.1 Introduction

The surge of the real-world network data, e.g., social networks and citation networks, has led to the rise of network mining research. An important research that underpins many high-level applications is to effectively and efficiently represent the network by embedding each node in the network into a low-dimensional space. Based on the simplified network representation, one can further conduct node classification [8], link prediction [76], visualization [119], anomaly detection [13], content recommendation [30] and many other data

mining tasks. In the real-world scenarios, *attributed networks*, where nodes contain rich attributes, are more common than *plain networks*, which only have node identifiers and edges between nodes. For example, a citation network is composed of nodes representing papers with words as attributes, and edges corresponding to citation relationships; in a follower-followee social network, nodes denote users with user profiles as attributes. Such attributed networks comprise both structure information (i.e., dependency between nodes) and content information (i.e., attributes of nodes), which facilitates better network understanding. It is of great significance to develop attributed network node embedding algorithms by simultaneously preserving structure and content information of the nodes.

Recently, various network node embedding algorithms have been proposed, including plain network node embedding methods with only structure information preserved [39, 95, 118, 129], and attributed network node embedding approaches with both structure and content information captured [123, 147, 152]. However, most attributed network node embedding algorithms involve learning procedures which are inefficient because of massive matrix operations. Consequently, the problem is much more challenging in real-time systems with large-scale networks. For example, given an evolving large-scale social network, if we aim to retrieve the most similar users w.r.t. a given queried user (e.g., for friend recommendation or user search), the network node embedding should be performed very quickly to return precise results. Current attributed network node embedding algorithms are unsuitable in this case due to inefficiency.

In order to preserve both content and structure information and significantly reduce the computational cost in situations without powerful workhorses (e.g., query in mobile devices), one possible solution is to employ randomized hashing techniques [33, 69, 73, 135, 138, 140, 142], which can efficiently sketch high-dimensional data and map them into a fixed number of dimensions as an (unbiased) estimator of the original high-dimensional feature vectors. Some hashing algorithms have been used to characterize substructures in the graph, e.g., [32] discovers large dense subgraphs, and [7] estimates local triangle counting. By contrast, in [65, 143], the graph is represented as a feature vector, where each dimension comprises a set of substructures, for classification by employing the hashing techniques to approximately count substructures. To the best of our knowledge, no works are devoted to applying randomized hashing techniques to network node embedding. Therefore, we aim to develop a randomized hashing technique, which does not involve learning and can achieve competitive or even better accuracy than the state-of-the-art learning-based network node embedding methods with dramatically improved efficiency in time.

Although there have been some research using randomized hashing techniques to approximately count substructures in a graph as features, to our knowledge there is no report-

ed work using randomized hashing for graph node embedding (node representation). In this paper, we represent each node of a graph as a shallow rooted tree through expanding its neighboring nodes, and then adopt the Locality-Sensitive Hashing (LSH) algorithm to sketch level-wise content of the rooted tree from bottom (the predefined highest-order neighboring nodes) to top (i.e., the root node). In particular, to summarize multiple attributes at each level of the rooted tree, MinHash, which is a well-known LSH scheme in the bag-of-words model, is recursively employed. In the recursive operation of LSH along the rooted tree, we take into account the law of network information diffusion, i.e., exponential decay from the information source to the distance. This implies that the proposed recursive randomized hashing algorithm should be able to discount the content information level by level in the course of propagation. We name the resulting algorithm for attributed network node embedding NetHash, which can encode the attributes and structure information of the attributed network for each node using a low-dimensional discrete vector.

We provide theoretical analysis of the estimator of the similarity between two rooted trees, and also empirically evaluate effectiveness and efficiency of our NetHash algorithm and the state-of-the-art methods on a number of real-world citation network data sets and social network data sets. We evaluate their effectiveness and efficiency on four real-world network benchmarks with 10^3 nodes and one large-scale citation network with 10^6 nodes. In summary, our contributions are three-fold:

1. This work is the first endeavor to introduce randomized hashing techniques into attributed network embedding.

2. We propose a novel non-learning attributed network node embedding algorithm named NetHash, which can preserve content and structure information for each node by expanding the node and its neighboring nodes into a rooted tree and recursively sketching the rooted tree from bottom to top.

3. The experimental results show that our NetHash algorithm can achieve competitive or better performance in accuracy with significantly reduced computational cost, compared to the state-of-the-art network node embedding methods.

The remainder of the chapter is organized as follows: Section 6.2 introduces the network node embedding problem and preliminary knowledge. We describe our NetHash algorithm in Section 6.3. The experimental results are presented in Section 6.4. We review the related work in Section 6.5. Finally, we conclude this paper in Section 6.6.

6.2 Preliminaries

In this section, we first describe the attributed network node embedding problem and some notations. Then, we briefly introduce a popular randomized hashing algorithm, MinHash, which will be employed for recursively sketching a rooted tree.

6.2.1 Problem Definition

Given an attributed network $G = (\mathscr{V}, \mathscr{E}, f)$, where $\mathscr{V}$ denotes the nodes of the network, $\mathscr{E}$ denotes the undirected edge of the network, and $f : \mathscr{V} \mapsto \mathscr{A}$ is a function that assigns attributes from an attribute set (or a global vocabulary) $\mathscr{A}$ to the nodes. Consequently, a node $v \in \mathscr{V}$, which owns attributes $f(v)$, can be represented as a 0/1 feature vector $\mathbf{v}_v$ for convenience where each dimension corresponds to an attribute in $\mathscr{A}$. Figure 6.1(a) shows a toy example where an attributed network is composed of 6 nodes, 8 edges and 10 attributes (the global vocabulary). Each node is assigned with an attribute subset (bottom left), which can be represented as a 10-dimensional 0/1 feature vector (bottom right). For example, Node 1 can be represented as [1 1 1 1 1 1 1 0 0 0], which means that it has attributes of $\{a, b, c, d, e, f, g\}$. Take paper citation network for example: papers can form a network with papers being nodes and the citation relationship between papers being edges. Each node is assigned with multiple attributes, e.g., the words from abstracts. The attributes of the node describe content information of the node, and the edges reflect structure information. Given a network G, attributed network node embedding aims to embed each node $\mathbf{v}_v \in \{0,1\}^{|\mathscr{A}|}$ into a low-dimensional vector $\mathbf{x}_v \in \mathbb{R}^D$, where $D \ll |\mathscr{A}|$, by preserving as much content and structure information as possible[1].

6.2.2 MinHash

The Jaccard similarity is simple and effective in many applications, especially for document analysis based on the bag-of-words model [112].

Given a universal set $\mathscr{U}$ and a subset $\mathscr{S} \subseteq \mathscr{U}$, MinHash is generated as follows: Assuming a set of D random permutations, $\{\pi_d\}_{d=1}^{D}$, where $\pi_d : \mathscr{U} \mapsto \mathscr{U}$, are applied to $\mathscr{U}$, the elements in $\mathscr{S}$ which are placed in the first position of each permutation, $\{\min\left(\pi_d(\mathscr{S})\right)\}_{d=1}^{D}$, would be the MinHashes of $\mathscr{S}$.

MinHash [11] is an approximate algorithm for computing the Jaccard similarity of two sets. It is proved that the probability of two sets, $\mathscr{S}_1$ and $\mathscr{S}_2$, to generate the same MinHash

[1] Different from embedding into l_2 (Euclidean) space in most algorithms, we embed nodes into l_1 (Hamming) space as [33].

value is exactly equal to the Jaccard similarity of the two sets:

$$\Pr\Big(\min\big(\pi_d(\mathscr{S}_1)\big) = \min\big(\pi_d(\mathscr{S}_2)\big)\Big) = J(\mathscr{S}_1,\mathscr{S}_2) = \frac{|\mathscr{S}_1 \cap \mathscr{S}_2|}{|\mathscr{S}_1 \cup \mathscr{S}_2|}. \tag{6.1}$$

To approximate the expected probability, multiple independent random permutations (i.e., MinHash functions) are used to generate MinHashes. The similarity between two sets based on D MinHashes is calculated by

$$\hat{J}(\mathscr{S}_1,\mathscr{S}_2) = \frac{\sum_{d=1}^{D} \mathbf{1}\Big(\min\big(\pi_d(\mathscr{S}_1)\big) = \min\big(\pi_d(\mathscr{S}_2)\big)\Big)}{D},$$

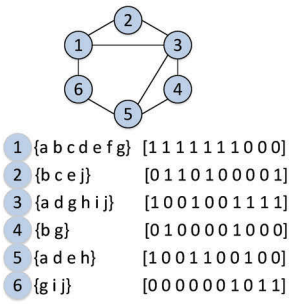
where $\mathbf{1}(state) = 1$, if $state$ is true, and $\mathbf{1}(state) = 0$, otherwise. As $D \to \infty$, $\hat{J}(\mathscr{S}_1,\mathscr{S}_2) \to J(\mathscr{S}_1,\mathscr{S}_2)$.

However, the explicit random permutations are expensive for a large universe set. Therefore, in practice, a hash function as follows can be used to generate the permutated index $\pi_d(i) = \texttt{mod}\big((a_d i + b_d), c_d\big)$, where i is the index of an element from the universal set $\mathscr{U}$, $0 < a_d, b_d < c_d$ are two random integers and c_d is a big prime number such that $c_d \geq |\mathscr{U}|$ [11].

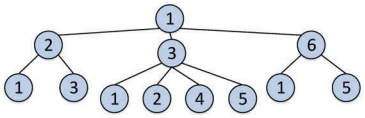
In the following, we will employ the MinHash scheme to generate the sketch of the attributes of a node (the non-zero elements in $\mathbf{v}_v$), say "digest", and diffuse such a digest to its parent node level by level in the rooted tree.

6.3 NetHash

In this section, we propose the NetHash algorithm for attributed network node embedding. In Section 6.2.2 we have introduced a randomized hashing scheme, MinHash, which can efficiently sketch the set of attributes of a node; but in its vanilla version MinHash is unable to further take into account structure information of the node. In order to take advantage of randomized hashing and further incorporate structure information at the same time, we represent each node as a shallow rooted tree through expanding its neighboring nodes. Then, we can encode attributes and structure information by recursively sketching the rooted tree from the bottom level (i.e., highest-order neighboring nodes) to the top (i.e., root node), and capture as much information closer to the root node as possible. Finally, each node embedding is obtained in a low-dimensional vector space which is much smaller than the size of the attribute set.



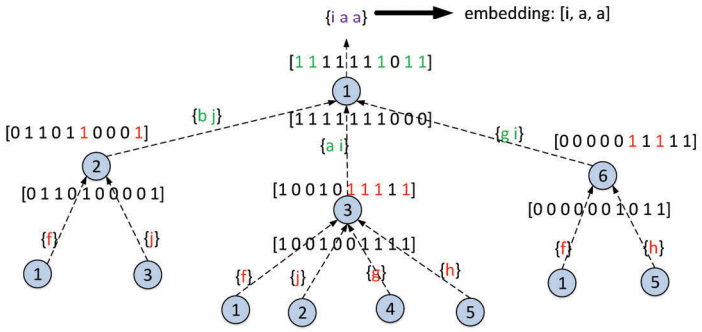
(a) 6-node network (top), attribute set (bottom left) and feature vector (bottom right) on each node.



(b) A tree rooted at Node 1 with the depth of 2.

node	parent	level
1	−1	0
2	1	1
3	1	1
6	1	1
1	2	2
3	2	2
1	3	2
2	3	2
4	3	2
5	3	2
1	4	2
5	4	2

(c) The structure of parent pointer tree (PPT) corresponding to the tree in Subfigure (b) , whose size is 12*3.



(d) NetHash with numbers of Min-Hash being 1, 2 and 3 from bottom to top. *Digest* (set) via sketching the node, *merger* (colorful feature vector) by merging *digest* and attributes (black feature vector), and *digest* diffusion (dash arrows). Root returns the embedding for the root.

Fig. 6.1 An illustration of embedding a node (recursively sketching a rooted tree) on a network.

6.3.1 The NetHash Algorithm

To simultaneously capture content and structure information in the network, we expand each node along with its neighboring nodes into a rooted tree. By recursively sketching the rooted tree from bottom to top, the NetHash algorithm diffuses content information level by level. The way of recursively sketching follows an inverse breadth-first traversal on the rooted tree. To this end, we employ the structure of *parent pointer tree* (PPT) to store the tree such that each node of the tree is visited only once. Figure 6.1(b) shows a tree rooted at Node 1 with depth 2. The 1st level has Nodes 2, 3 and 6, and the 2nd level has Nodes {1, 3}, {1, 2, 4, 5}, and {1, 5}. Figure 6.1(c) shows the corresponding PPT, which is a 12×3 array: The first column shows node identifiers on the network; the second column points to the parent node in the array, e.g., “-1" means that root Node 1 has no parent node and “1" in the 2nd row (i.e., $[2, 1, 1]$) shows that the parent node of Node 2 is in the 1st row (i.e., Node 1); the third column is the level where the node is located.

We outline NetHash in Algorithm 7. NetHash processes nodes one by one (Line 1). We first build the PPT to store the rooted tree whose root node is currently being processed (Line 2). To traverse the rooted tree from bottom to top, we introduce an auxiliary queue Q to store the currently visited node in the rooted tree and the corresponding diffused content information (Line 3). Subsequently, NetHash traverses the PPT from end to head for recursively sketching the rooted tree, where $\uparrow$ in Line 4 means inverse traversal in PPT (Lines 4-12). The recursive sketching process consists of two steps. Sketching a node produces a set of MinHashes called *"digest", which represents the content information diffused from the node to its parent node*, e.g., $\{b, j\}$ produced by sketching Node 2 in Figure 6.1(d). Actually, the step summarizes content information of each node. In the while loop, an *internal node* `merges` its own attributes with all digests, which are generated by sketching its all child nodes, to produce the *"merger", whose entries of 1 correspond to attributes or digests,* e.g., $[0\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1]$ on Node 2 in Figure 6.1(d). Consequently, the *internal node* carries both its own and diffused content information. Essentially, this step preserves structure information by diffusing and merging because the structure (i.e., edges) is reflected by digests. Note that no `merging` operation occurs on *leaf nodes*. Finally, when the root node is visited, Q becomes empty and the corresponding digest is `pushed` into Q. Consequently, the only element in Q is the embedding for the root node (Line 13). For example, the digest for the root node, i.e., $\{i, a, a\}$, is generated by `MinHash` at the top level, and `pushed` into Q. The only digest in Q is the embedding for the root node, that is $[i, a, a]$.

6.3.2 Exponentially Decayed Diffusion

We build a framework of capturing content and structure information by recursively sketching the rooted tree. Intuitively, the lower-order neighboring nodes contribute more to representation of the root node than the higher-order ones do, which implies that NetHash should discount content information level by level in the course of propagation. To this end, we combine the recursive operation of the MinHash algorithm with the law of network information diffusion, i.e., exponential decay from the information source to the distance. Natually, the number of MinHashes, d_l, which is the amount of digest produced from a node at the l-th level, conforms to the following equation:

$$d_l = \max\{1, \texttt{int}\big(D\exp(-l\lambda)\big)\}, \tag{6.2}$$

where D is the number of embedding dimensions, i.e., the number of MinHashes at the root node (its level is 0 and $d_0 = D$), λ is the exponential decay rate, and `int` is a rounding

[1] Note that $D = 0$ indicates that the tree has only one root node.

Algorithm 7 The NetHash Algorithm

Input: $G = (\mathscr{V}, \mathscr{E}, f)$; number of dimensions for embedding D; entropy of degrees of network S; depth of tree L; hash functions at the l-th level of tree $\{\pi_d^{(l)}\}_{l=0,d=1}^{L,D_l}$

Output: G's embedding $\mathbf{h}$

for $r = 1, \ldots, |\mathscr{V}|$ **do**
  Build a parent pointer tree $\mathbf{T}$ for node r;
  Initialize an empty auxiliary queue Q;
  for $v \in \mathbf{T} \uparrow$ **do**
    $l \leftarrow$ level of v in $\mathbf{T}$;
    $merger \leftarrow f(v)$; // initial merger from attributes on v
    while Q is not empty and
    v is the parent node of $Q[0]$ in $\mathbf{T}$ **do**
      $merger \leftarrow \texttt{merge}(Q.\texttt{pop()}.digest, merger)$;
    end while
    $digest \leftarrow \texttt{MinHash}(merger, \{\pi_d^{(l)}\}_{d=1}^{d_l})$;
    $Q.\ \texttt{push}(\{digest, v\})$;
  end for
  $\mathbf{h}(r) \leftarrow Q.\texttt{pop()}.digest$;
end for

function. If $d_l < 1$, one attribute of digest will be diffused. Note that nodes at the same level are sketched by the same MinHash functions for the sake of consistency.

Content information is diffused along edges, so information diffusion is closely related to node degrees. If treating node degrees in a network as a distribution, we can define the entropy of node degrees, S, from the perspective of information theory:

$$S = -\sum_{\nu \in \texttt{unique}(\{\nu_v\}_{v \in \mathscr{V}})} \Pr(\nu) \log \Pr(\nu), \tag{6.3}$$

where ν_v is degree of node v and $\texttt{unique}(\{\nu_v\}_{v \in \mathscr{V}})$ is the set of unique degrees observed in the network. A large entropy suggests that the network tends to have evenly distributed degrees, which implies that content information diffused in the network is unstable since the amount of diffused content (in our case the "digest") varies widely from node to node. To address the issue, such a network ought to diffuse little digest by accelerating the decay of diffusion. To this end, we use entropy as the decay rate and Eq. (6.2) is rewritten as

$$d_l = \max\{1, \texttt{int}\big(D \exp(-lS)\big)\}. \tag{6.4}$$

6.3.3 Theoretical Analysis

Similarity: According to Section 6.2.2, the similarity between two rooted tree with the depth being 1 can be defined as:

$$J(\mathscr{A},\mathscr{B}) = \mathbb{E}_{\pi_0,\pi_1}\Big[\mathbf{1}\Big(\pi_0\big(f(\mathscr{A}_0^{(0)})\cup\pi_1(f(\mathscr{A}_*^{(1)}))\big) = \pi_0\big(f(\mathscr{B}_0^{(0)})\cup\pi_1(f(\mathscr{B}_*^{(1)}))\big)\Big)\Big],$$

where $f(\mathscr{A}_0^{(0)})$ denotes the attributes on the root node, $\pi_1(f(\mathscr{A}_*^{(1)}))$ denotes the digest sets from the nodes at the 1st level by Min-Hash and $*$ denotes all nodes at the 1st level. Note that only expectation of $\pi_1(f(\mathscr{A}_*^{(1)}))$ can give rise to real similarity. Therefore, we can recursively extend the similarity to rooted trees with arbitrary depth.

Given two rooted trees, $\mathscr{A}^{(L)}$ and $\mathscr{B}^{(L)}$, where each node contains attributes and L denotes the depth of the rooted trees, the similarity between $\mathscr{A}^{(L)}$ and $\mathscr{B}^{(L)}$ is:

$$J(\mathscr{A}^{(L)},\mathscr{B}^{(L)}) = \mathbb{E}_{\pi_0,\cdots,\pi_L}\Big[\mathbf{1}\big(\rho(a)=\rho(b)\big)\Big],$$

$$\rho(a) = \pi_0\Big(f(\mathscr{A}_0^{(0)})\cup\pi_1\big(\cdots f(\mathscr{A}_*^{(L-1)})\cup\pi_L(f(\mathscr{A}_*^{(L)}))\big)\Big),$$

$$\rho(b) = \pi_0\Big(f(\mathscr{B}_0^{(0)})\cup\pi_1\big(\cdots f(\mathscr{B}_*^{(L-1)})\cup\pi_L(f(\mathscr{B}_*^{(L)}))\big)\Big),$$

where $\mathscr{A}_*^{(L)}$ and $\mathscr{B}_*^{(L)}$ represent leaf nodes, π_l is MinHash functions at the l-th level and particularly, the number of π_0 is L.

Bounds: We give theoretical bounds between the real similarity $J(\mathscr{A}^{(L)},\mathscr{B}^{(L)})$ and the estimated similarity $\hat{J}(\mathscr{A}^{(L)},\mathscr{B}^{(L)})$.

Theorem 1 *Given $0<\delta<1,\varepsilon>0$, and $K>2\delta^{-2}s^{-1}\log\varepsilon^{-1}$, for two rooted trees, $\mathscr{A}^{(L)}$ and $\mathscr{B}^{(L)}$, we have bounds:*

1. *If $J(\mathscr{A}^{(L)},\mathscr{B}^{(L)})\geq s$, then $\hat{J}(\mathscr{A}^{(L)},\mathscr{B}^{(L)})\geq(1-\delta)s$ with a probability at least $1-\varepsilon$.*
2. *If $J(\mathscr{A}^{(L)},\mathscr{B}^{(L)})\leq s$, then $\hat{J}(\mathscr{A}^{(L)},\mathscr{B}^{(L)})\leq(1+\delta)s$ with a probability at least $1-\varepsilon$.*

Proof 1 *Let $X_{\pi_0,\cdots,\pi_L}$ be a random variable. $X_{\pi_0,\cdots,\pi_L}=1$ if $\rho(a)=\rho(b)$, and 0 otherwise. Let $X=\sum_{d=1}^{D}X_{\pi_0,\cdots,\pi_L}^{(d)}$. We have $\mathbb{E}_{\pi_0,\cdots,\pi_L}[X_{\pi_0,\cdots,\pi_L}^{(d)}]=J(\mathscr{A}^{(L)},\mathscr{B}^{(L)})\geq s$; $\mathbb{E}[X]\geq Ds$ as only D MinHash functions sketch the root node whatever L is. Based on the Chernoff bound [17],*

we have

$$\begin{aligned}\Pr\big(X < (1-\delta)Ks\big) &\leq \Pr\big(X < (1-\delta)\mathbb{E}(X)\big) \\ &\leq \mathrm{e}^{-\frac{\delta^2\mathbb{E}(X)}{2}} \leq \mathrm{e}^{-\frac{\delta^2 Ds}{2}}\end{aligned} \tag{6.5}$$

Substituting Eq. (6.5) with $\hat{J}(\mathscr{A}^{(L)}, \mathscr{B}^{(L)}) = \frac{X}{D}$ *gives*

$$\Pr\big(\hat{J}(\mathscr{A}^{(L)}, \mathscr{B}^{(L)}) < (1-\delta)s\big) \leq \mathrm{e}^{-\frac{\delta^2 Ds}{2}} < \varepsilon \tag{6.6}$$

By rearranging Eq. (6.6), we prove the first proposition for $D > 2\delta^{-2}s^{-1}\log\varepsilon^{-1}$. *Similarly, the second one is proved.*

Furthermore, we observe that the bound is independent of the depth of the rooted trees, D, which implies that we can easily control quality of the estimator by only tuning K.

Complexity: Let $\overline{v}$ denote the mean value of degrees of the network, $|\overline{\mathscr{A}}|$ the mean number of attributes of each node, d_l the number of MinHash functions at the l-th level of the rooted tree. Consequently, the rooted tree can be averagely considered as a $\overline{v}$-ary tree, where each node contains $|\overline{\mathscr{A}}|$ attributes. In order to obtain the embedding for one node, NetHash traverses the tree rooted at the node only once. Specifically, each leaf node costs $O(|\overline{\mathscr{A}}|d_L)$ to generate digest and there are a total of $\overline{v}^L$ leaf nodes. Each internal node at the l-th level (including the root node) spends $O\big((|\overline{\mathscr{A}}| + d_{l+1}\overline{v})d_l\big)$ producing its digest, where $O(d_{l+1}\overline{v})$ denotes the total amount of digest diffused from $(l+1)$-th level. Also, there are a total of $\overline{v}^l$ nodes at the l-th level. Finally, the theoretical computational complexity of NetHash is $O(|\mathscr{V}|\big(\sum_{l=0}^{L-1}(|\overline{\mathscr{A}}| + d_{l+1}\overline{v})d_l\overline{v}^l + |\overline{\mathscr{A}}|d_L\overline{v}^L\big))$. Considering Eq. (6.4), we use $d_l = D\mathrm{e}^{-lS}$ for simplicity, and have $O(|\mathscr{V}|\overline{v}^L(D|\overline{\mathscr{A}}|\mathrm{e}^{-Ls} + D^2\mathrm{e}^{(-2L-1)s}))$. Additionally, NetHash has a space complexity of $O(\overline{v}^{L+1})$ because the rooted tree has a total of $O(\overline{v}^{L+1})$ nodes, which is equal to the size of the auxiliary queue in the worst case.

In comparison, TADW [147] and HSCA [152], first spend $O(|\mathscr{V}|^2)$ in the matrix M, where each entry is related to probability that a node randomly walks to another one in fixed steps; then in each iteration of the learning process, TADW costs $O\big(|\mathscr{V}|D^2 + |\mathscr{V}||\mathscr{A}|D + D\mathtt{nnz}(M)\big)$, and HSCA costs at least $O\big(|\mathscr{V}|D^2 + |\mathscr{V}||\mathscr{A}|D + |\mathscr{V}||\mathscr{A}|^2 + D\mathtt{nnz}(M)\big)$, where $\mathtt{nnz}$ gives the number of non-zero entries. In practice, we normally consider trees with $L \in \{1,2,3\}$ as the distant nodes can hardly diffuse effective attributes to the root node; while TADW and HSCA iterate 10 and 1000 times, respectively. Besides, in most networks, $\overline{v}, \overline{v}^L \ll |\mathscr{V}|$ due to sparsity, $|\overline{\mathscr{A}}|, L \ll |\mathscr{A}|$. Therefore, the time complexity of NetHash is practically efficient, especially in large-scale networks because it is linear with respect to $|\mathscr{V}|$.

6.4 Experimental Results

In this section, we report the performance of our NetHash algorithm and its competitors on the five real-world network data sets. For each network data, edge directions are not considered. In Section 6.4.1, we first explore the best parameters for our NetHash algorithm. Then, we validate the effectiveness and efficiency of the compared methods through the task of node classification in Section 6.4.2, the task of link prediction in Section 6.4.3, and a case study of query on a large-scale citation network in Section 6.4.4. Finally, we discuss the results of the above tasks in Section 6.4.5.

Data sets: We evaluate our NetHash algorithm on a wide range of benchmarks, including citation networks and social networks. (1) Cora [87]: A typical citation network of machine learning papers. (2) Wikipedia [147]: A collection of articles in Wikipedia. (3) Flickr [66]: This data set from an image and video sharing website consists of users as nodes and following relationship as edges in the network. The tags of interest specified by users act as attribute information. (4) BlogCatalog [66]: This data set consists of bloggers as nodes and following relationship as edges in the network. The keywords in blog descriptions are used as attribute information. (5) ACM [120]: The original data is a collection of papers from ACM, which consists of 2,381,688 papers and 10,476,564 citation relationship. Before experiments, we clean up papers without abstracts or citations in the data. Then, we build a citation network with papers being nodes and citation being edges, where each node uses abstract as attributes. The data sets are summarized in Table 2.4.

The state-of-the-art methods: We compare our NetHash algorithm with six state-of-the-art methods. (1) DeepWalk [95]: It captures contextual structure information based on random walks; (2) node2vec [39]: It performs a biased DeepWalk to explore diversity of the neighboring nodes; (3) LINE [118]: It preserves the first-order and second-order proximities via the breadth-first search instead of the depth-first search in DeepWalk; (4) NetMF [96]: It unifies LINE, PTE, DeepWalk, and node2vec in the framework of matrix factorization. (5) TADW [147]: It learns node representations by combining attributes with structure in matrix factorization; (6) HSCA [152]: It adds the first-order proximity to TADW for performance gain in attributed networks; (7) CANE [123]: In addition to attribute and structure information, it captures different contextual information which a node expresses for different neighboring nodes.

Experimental Settings: For all compared methods, the dimension of the embedding for each node, D, is set to 200 by default, as used in attributed network embedding [123, 147]. All implementations and parameters of the compared algorithms are provided by the authors. We would like to note that, different from embedding into l_2 (Euclidean) space by the

Table 6.1 Tuning depth for Cora.

Depth	Micro-F1(%)					Runtime(s)
	Training Ratio					
	50%	60%	70%	80%	90%	
1	85.41	85.99	86.55	86.98	87.28	0.1
2	**85.65**	**86.22**	**86.74**	**87.13**	**87.31**	**0.4**
3	85.12	85.93	86.55	86.75	87.00	0.8

Table 6.2 Tuning decay rate for Cora.

Decay rate	Micro-F1(%)					Runtime(s)
	Training Ratio					
	50%	60%	70%	80%	90%	
0.5	85.11	85.99	86.55	86.98	87.28	6
2.11	**85.65**	**86.22**	**86.74**	**87.13**	**87.31**	**0.4**
5	85.12	85.93	86.55	86.75	87.00	0.2

compared algorithms, our NetHash algorithm embeds each node into l_1 (Hamming) space[2] [33]. Suppose an embedding algorithm generates $\mathbf{x}_{v_1}$ and $\mathbf{x}_{v_2}$, which are the representations of length D for two nodes, v_1 and v_2, respectively. In our NetHash algorithm, the Jaccard similarity between $\mathbf{x}_{v_1}$ and $\mathbf{x}_{v_2}$ can then be computed as

$$\mathbf{Sim}_{\mathbf{x}_{v_1},\mathbf{x}_{v_2}} = \frac{\sum_{d=1}^{D} \mathbf{1}(x_{v_1,d} = x_{v_2,d})}{D}, \tag{6.7}$$

where $\mathbf{1}(state) = 1$ if $state$ is true, and $\mathbf{1}(state) = 0$ otherwise. All the random variables of hash functions are globally generated at random. The runtime of NetHash consists of generating hash functions, constructing and sketching the rooted trees. All experiments are conducted on a node of a Linux Cluster with 8×3.4 GHz Intel Xeon CPU (64 bit) and 32GB RAM.

Table 6.3 Node classification results on citation networks.

Data	Algorithms	Micro-F1(%) Training Ratio					Macro-F1(%) Training Ratio					Runtime(s)
		50%	60%	70%	80%	90%	50%	60%	70%	80%	90%	
Cora	DeepWalk	78.83	79.68	80.29	80.78	81.20	77.63	78.48	79.08	79.55	79.83	436.39
	LINE	76.04	77.77	76.65	76.13	77.20	74.92	76.59	75.78	75.15	76.41	5.00
	node2vec	81.22	81.98	82.68	83.09	83.52	80.14	80.91	81.61	81.99	82.28	24.01
	NetMF	83.76	84.18	85.11	85.39	87.20	82.72	83.05	84.08	84.26	86.10	11.26
	TADW	85.74	86.17	86.46	86.71	86.78	84.35	84.71	85.23	85.25	85.26	66.54
	HSCA	85.70	85.79	85.97	86.25	86.38	84.15	84.25	84.41	84.62	84.82	72.17
	CANE	**85.78**	86.13	86.70	**87.27**	**87.32**	84.41	84.73	85.26	85.57	85.86	5622.72
	NetHash	85.65	**86.22**	**86.74**	87.13	87.31	**84.80**	**85.42**	**85.93**	**86.34**	**86.43**	**0.4**
Wikipedia	DeepWalk	63.71	64.63	65.33	65.83	66.55	50.63	51.60	52.28	52.47	52.11	355.56
	LINE	59.35	59.62	59.12	61.41	61.42	42.89	44.03	44.15	44.95	44.51	5.00
	node2vec	63.70	64.61	65.31	65.91	66.36	50.78	51.53	52.22	52.49	52.04	24.81
	NetMF	66.21	67.57	68.38	70.20	70.04	55.54	56.59	57.05	57.63	56.53	8.66
	TADW	**75.58**	**76.06**	76.59	76.96	77.20	**59.39**	**60.23**	**61.10**	**61.57**	60.90	90.97
	HSCA	72.35	72.61	72.71	72.82	73.05	56.57	56.93	57.19	57.01	56.40	103.96
	CANE	72.62	73.27	73.91	74.36	74.71	57.30	58.23	58.97	59.02	58.28	10878.74
	NetHash	74.12	75.49	**76.64**	**77.51**	**78.12**	55.10	57.45	59.39	61.00	**61.28**	**2.00**

6.4.1 Parameter Settings

Our NetHash algorithm has two exclusive parameters, tree depth L and decay rate λ. Therefore, we tune the parameters to obtain the best settings. Due to similar trends, we report only the node classification results on Cora.

Table 6.1 shows accuracy of our NetHash algorithm w.r.t. the depth L. our NetHash algorithm achieves the best accuracy on Cora when $L = 2$. The reason that the accuracy on Cora improves first when L varies from 1 to 2 is probably because more useful attributes are captured from higher-order neighboring nodes. However, when L further increases to 3, more noise may be introduced to deteriorate the performance. Therefore, tuning depth L is a tradeoff between noise and useful attributes, and the optimal value of L depends on the

[2]It is worth noting that the l_1 embedding suffices for most network mining and analysis tasks such as node classification, link prediction, and similar node retrieval. If necessary, it is easy to transform the obtained l_1 embedding into l_2 embedding using a spectral method.

Table 6.4 Link prediction results on social networks.

Data	Algorithms	Training Ratio									
		50%		60%		70%		80%		90%	
		AUC	Runtime(s)	AUC	Runtime(s)	AUC	Runtime(s)	AUC	Runtime(s)	AUC	Runtime(s)
Flickr	DeepWalk	31.75	1144.68	31.98	1150.99	33.01	1156.30	33.41	1156.31	34.51	1151.60
	LINE	21.51	5.00	24.93	5.00	24.81	5.00	26.29	5.00	27.05	5.00
	node2vec	17.93	289.60	16.79	354.92	16.70	423.26	17.76	479.61	19.30	528.56
	NetMF	82.46	49.41	84.08	50.06	84.91	51.77	85.39	52.17	**85.96**	54.46
	TADW	64.61	365.34	65.44	392.96	66.59	417.07	67.30	443.16	67.25	477.08
	HSCA	55.35	387.11	54.54	412.93	54.03	456.80	54.03	498.20	53.25	497.93
	NetHash	**85.26**	**0.60**	**84.85**	**0.60**	**85.54**	**0.80**	**85.79**	**1.00**	85.40	**1.00**
BlogCatalog	DeepWalk	78.02	822.70	79.04	819.44	79.53	824.86	79.66	831.62	80.14	845.66
	LINE	49.71	5.00	53.78	5.00	56.92	5.00	58.92	5.00	60.44	5.00
	node2vec	58.24	92.51	63.29	109.65	68.21	125.15	72.10	141.07	75.29	159.97
	NetMF	**78.98**	21.80	**79.86**	22.27	**80.49**	23.87	**81.11**	24.65	**81.97**	25.46
	TADW	68.52	213.70	69.51	219.51	70.50	226.92	72.00	218.44	71.94	221.91
	HSCA	50.40	209.94	50.25	221.68	50.02	267.34	49.85	277.38	50.02	302.77
	NetHash	69.07	**0.60**	71.14	**0.80**	72.16	**1.00**	72.51	**1.00**	74.24	**1.00**

In the experiment, each algorithm is given a cutoff time of 100,000 seconds. Consequently, CANE is forced to stop within the cutoff time, and we are not able to show the result.

particular data. The runtime grows with L increasing. Finally, we set $L = 1$ for Wikipedia, Flickr and BlogCatalog, and $L = 2$ for Cora and ACM to achieve the best accuracy.

Table 6.2 shows accuracy of our NetHash algorithm w.r.t. the decay rate λ. The accuracy first increases and then declines. The best accuracy is achieved when the decay rate is set to the entropy of node degrees S. A smaller decay rate implies more diffused information, and thus, more noises; while a larger decay rate implies less diffused information, which might be insufficient. The runtime decreases remarkably with λ increasing. Hence, we adopt the entropy of node degrees S as the decay rate.

6.4.2 Node Classification on Citation Networks

We investigate classification performance of the compared methods on two labeled citation network benchmarks, i.e., Cora and Wikipedia. In the task, we adopt LIBSVM [14] for our NetHash algorithm which uses the Jaccard similarity matrix as the precomputed kernel, and LIBLINEAR [27] [3] for the other learning-based algorithms, as used in

[3]We test the compared methods on LIBSVM. DeepWalk, node2vec and CANE achieve similar accuracy while TADW and HSCA perform worse, so we just report results on LIBLINEAR.

Table 6.5 Top-5 Query results on a large-scale citation network.

Query: Genetic Algorithms in Search, Optimization and Machine Learning	Runtime (s)
DeepWalk	19,087
1. Dynamic Identification of Inelastic Shear Frames by Using Prandtl-Ishlinskii Model	
2. Application of Nontraditional Optimization Techniques for Airfoil Shape Optimization	
3. Genetic Algorithms and Neural Networks in Optimal Location of Piezoelectric Actuators and identification of Mechanical Properties	
4. Trajectory Controller Network and Its Design Automation Through Evolutionary Computing	
5. Flow Restrictor Design for Extrusion Slit Dies for a Range of Materials: Simulation and Comparison of Optimization Techniques	
LINE	2,087
1. No Silver Bullet: Extending SDN to the Data Plane	
2. A Leakage-Resilient Mode of Operation	
3. Program Improvement by Internal Specialization	
4. Using Schemas to Simplify Access Control for XML Documents	
5. A Visual Environment for the Creation of Intelligent Distributed Decision Support Systems	
NetHash	**723**
1. Neural Networks: A Comprehensive Foundation	
2. Machine Learning	
3. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II	
4. Introduction to Algorithms	
5. An Introduction to Support Vector Machines: and Other Kernel-Based Learning Methods	

In the experiment, each algorithm is given a cutoff time of 24 hours. Consequently, NetMF, TADW, HSCA and CANE are forced to stop because of overtime, while node2vec fails due to out of memory..

their papers. We vary the training ratio (i.e., percentage of nodes as the training set) in $\{50\%, 60\%, 70\%, 80\%, 90\%\}$, for each ratio of which we repeat the experiment 10 times and average the results.

Table 6.3 reports the experimental results in accuracy (Micro-F1 and Macro-F1) and runtime. Our NetHash algorithm defeats all plain network node embedding algorithms except NetMF in the case of the training ratio being 50% on Wikipedia, and achieves the competitive and even better accuracy than all attributed network node embedding ones. In runtime, our NetHash algorithm presents the strong advantage: it performs more efficiently than all compared methods. Generally, our NetHash algorithm outperforms TADW and HSCA by one to two orders of magnitudes, and CANE by four orders.

6.4.3 Link Prediction on Social Networks

We conduct link prediction task on two social network data sets, i.e., Flickr and BlogCatalog. Our NetHash algorithm adopts the Jaccard similarity in l_1 space while the compared methods uses the Euclidean distance in l_2 space. We obtain a training network by randomly preserving a training ratio of edges, while the removed edges will be treated as the unobserved (test) links. Based on the training network, we compute the similarity between each pair of nodes (including the preserved links, the unobserved links and the nonexistent links in the original network). Then, we randomly pick an unobserved link and a nonexistent link to compare their similarities. We calculate AUC values, which represent the probability that a randomly selected unobserved link is more similar than a randomly selected nonexistent link, by repeating the picking operation 10,000 times. We compute AUC values 10 times and average the results.

Table 6.4 shows the experimental results of AUC and runtime. Our NetHash algorithm significantly outperforms the compared methods in AUC and runtime on Flickr except NetMF in the case of the training ratio being 90%, while it defeats all attributed network node embedding algorithms on BlogCatalog. Our NetHash algorithm exhibits superiority in runtime – it performs much more efficiently than the compared methods by orders of magnitudes.

6.4.4 Case Study: Node Retrieval on a Large-scale Citation Network

In this case study, given 5 query papers on ACM that have top 5 highest citation numbers, we retrieve 5 most similar papers for each query based on their embeddings from all algorithms. Due to similar results, we only report the result of the paper with the highest citation number (i.e., 6,294), titled “Genetic Algorithms in Search, Optimization and Machine Learning", based on their embeddings. Note that our NetHash algorithm adopts the Jaccard similarity while the compared methods uses Euclidean distance.

Table 6.5 presents the query result. All results retrieved by our NetHash algorithm are closely related to algorithms and machine learning, and particularly, it accurately returns one paper regarding the genetic algorithm (i.e, the third result). Moreover, from the semantic perspective, we find that the results, except the third one, and the query all focus on introduction of foundation knowledge. By contrast, although the results retrieved by DeepWalk overlap with the query on some keywords, the retrieved papers tend to be applications that adopt some relevant methods. For LINE, there appears no correlation between the query and the results. In terms of runtime, our NetHash algorithm is still very efficient when running on such a large-scale network data, and it takes nearly 1/3 of LINE’s runtime.

6.4.5 Discussion on the Results

In the above three tasks, our NetHash algorithm generally achieves competitive and even better accuracy, because it preserves both content and structure information while DeepWalk, LINE, node2vec and NetMF only capture structure information. Exceptionally, DeepWalk and NetMF perform better than NetHash on BlogCatalog, largely because in the complicated network (e.g., large average node degree and entropy), the former captures sufficient structure information by long random walks and the latter exploits enough structure information by employing a large window size. In terms of attributed network node embedding algorithms, NetHash and TADW essentially preserve attributes and structure information, so they keep the same level in accuracy on Cora, Wikipedia and BlogCatalog. Furthermore, the reason why our NetHash algorithm defeats TADW in most cases is largely because the recursive operation of MinHash can more effectively capture structure information and additionally, MinHash is more powerful to capture attribute information than matrix factorization, that is, the former has a stronger ability of summarization for attributes from high-dimension to low-dimension, especially for data with a large vocabulary. For example, on Flickr which owns 12,000 attributes, NetHash significantly outperforms all attributed network embedding algorithms. Although HSCA and CANE additionally capture first-order proximity and contexts, respectively, our NetHash algorithm still outperforms them on Wikipedia, Flickr and BlogCatalog because the additional information negatively impacts representation and in turn deteriorates the accuracy.

In terms of runtime, our NetHash algorithm performs much more efficiently than DeepWalk and node2vec, because our NetHash algorithm expands nodes into the trees with the depth being around 2 while DeepWalk and node2vec perform long random walks. In addition, although LINE performs competitively fast as NetHash on small-scale network data, it is significantly slow on a large-scale network because it needs to first reconstruct the whole network to improve low-degree network embedding. NetMF runs more slowly than NetHash as NetMF considers more nodes by adopting a larger window size, although it employs an efficient approximation algorithm. We know from Section 6.3.3 that empirically, our NetHash algorithm has lower time complexity than TADW and HSCA. CANE experimentally executes a substantial number of matrix operations in deep learning, which is costly in runtime and demands powerful workhorses. In addition, our NetHash algorithm, as a randomized algorithm, avoids the iteration process in the learning-based algorithms, which is also an important factor of high efficiency.

In summary, MinHash, as a powerful tool in the bag-of-words model, can effectively represent data as the compact hash codes. Therefore, NetHash is very suitable for the network data where nodes can be represented as the bag-of-words model, for example, citation

networks where nodes are composed of texts. Furthermore, if the network has a large vocabulary on nodes, NetHash will show good performance due to the strong ability of MinHash to summarize attribute information. In terms of efficiency, NetHash can be successfully applied to the network data of low node degrees because the time complexity dramatically grows with the node degrees increasing. Also, in a large-scale network, NetHash performs efficiently due to the linear relationship with the number of nodes.

6.5 Related Work

In this section, we first introduce current works of network node embedding, and then review works of graph hashing.

6.5.1 Network Node Embedding

Current network node embedding approaches consist of plain network node embedding and attributed network node embedding. The former describes each node by considering only structure information in the network, while the latter preserves both content and structure information for performance enhancement. They both aim to learn the latent representation for each node in a low-dimensional space.

DeepWalk [95] is the pioneer work in network node embedding. The method simulates word sequences with a word representation learning model, Skip-Gram [88], by performing some random walks in the network so that the nodes in the random walks can capture contextual structure information. Node2vec [39] introduces bias into DeepWalk in order to explore diverse neighboring nodes of a node, while DeepWalk uniformly samples nodes from neighbors. The first-order proximity composed of edges between nodes preserves local structure information, and the second-order proximity shows that nodes with shared neighboring nodes are similar and thus captures global structure information. Based on the two proximities, LINE [118] explicitly defines an object function to capture local and global structure information separately, and then obtains the final representation by concatenating the two representations. The above methods aim to implicitly approximate and factorize the matrices. Furthermore, NetMF [96] unifies them in the framework of explicit matrix factorization. Recently, SDNE [129] adopts a semi-supervised deep model to capture the highly non-linear network structure. Specifically, the first-order proximity is used as the supervised information by the supervised component, and the second-order proximity is used by the unsupervised component. The above methods are all designed for plain network embedding without considering content information on the nodes.

To incorporate content information, researchers have successively proposed attributed network node embedding algorithms. TADW [147], as the first approach in attributed network node embedding, injects texts into matrix factorization by proving that DeepWalk is equivalent to matrix factorization. Based on TADW, HSCA [152] adds information of the first-order proximity to improve network representation. In other words, HSCA can be considered as a combination of attributes of a node with the first-order proximity and the higher-order proximity in matrix factorization. Essentially, TADW and HSCA are random walk based methods. In addition to content and structure information, one node may have different contexts when interacting with different neighboring nodes. To this end, CANE [123] adopts deep learning to capture such contexts for performance enhancement. All of the above approaches are learning based techniques. By contrast, our work aims to remarkably reduce the overall computational cost while maintaining the same level of accuracy as existing attributed network node embedding algorithms by exploiting a randomized hashing technique.

6.5.2 Graph Hashing

Although most randomized hashing algorithms are designed for vectors or sets, some works aim to capture structure information by characterizing substructures in the graph. In [32], large dense subgraphs in massive graphs are identified via a variant of MinHash, where each hash function is generalized to return multiple elements for capturing more neighboring information, while in the standard MinHash scheme, each hash function returns an element. In [7], a slight modification of MinHash, which improves the computation speed of the standard method, is used to estimate the local number of triangles in large graphs. Some works adopt two random hashing schemes: In [2], the first hashing scheme reduces the size of edges, and the second one summarizes the frequent patterns of co-occurrence edges; in [18], the first hashing scheme compresses a graph into a small graph, and the second one maps unlimitedly emerging cliques into a fixed-size clique set.

[20, 65, 143] all hash tree structures in the graph and approximately count substructures. Consequently, the graph is represented as a feature vector for classification, where each dimension comprises a set of substructures and the value of each dimension is related to the estimated number of the substructures. Therefore, the above methods cannot be used for our problem.

6.6 Conclusion

In this chapter, we propose an attributed network node embedding algorithm dubbed NetHash, which employs the randomized hashing technique to recursively sketch the shallow trees, where each tree is rooted at a node of the network, from bottom to top, and meanwhile preserves as much information closer to the root node as possible by simulating the exponential decay in information diffusion of the network.

We conduct extensive empirical tests of our NetHash algorithm and the state-of-the-art methods. We evaluate its effectiveness and efficiency on four citation and social network data sets and one large-scale citation network (10^6 nodes). The experimental results show that our NetHash algorithm not only achieves competitive or better performance but also performs much faster than the compared methods by orders of magnitudes. In the application scenarios involving large-scale network analysis, our NetHash algorithm can achieve state-of-the-art performance with a limited budget of computational capacity, which makes it more practical in the era of big data.

Chapter 7

Conclusions and Future Work

This chapter summarizes the whole thesis and provides some further research directions.

7.1 Summary of This Thesis

In this thesis, we focus on exploring new randomized hashing algorithms in order to efficiently compute similarities on structured data, e.g., texts, chemical compounds, social networks and scientific publication networks. We do a literature review regarding the hashing algorithms.

In Chapter 4, we explore the MinHash algorithm on weighted sets based on the Consistent Weighted Sampling (CWS) scheme, and successively propose two MinHash-based algorithms for weighted sets from two perspectives: 1) By analyzing the flaw that the Improved Consistent Weighted Sampling (ICWS) algorithm breaks the uniformity of the MinHash scheme, we reconstruct the hash functions and propose the Canonical Consistent Weighted Sampling (CCWS) algorithm which not only shares the same theoretical time complexity as ICWS but also strictly obeys the MinHash scheme. 2) By transforming the original form of ICWS into the equivalent mathematical expression, we observe the working mechanism of ICWS and furthermore propose the Practical Consistent Weighted Sampling (PCWS) algorithm, which is simpler and more efficient than ICWS in terms of both time and space complexities.

In Chapter 5, we propose a MinHash-based algorithm for graphs, i.e., K-Ary Tree Hashing (KATH). The KATH algorithm fast approximates the state-of-the-art method, the Weisfeiler-Lehman (WL) graph kernel, by generating K-ary trees in a traversal table. Consequently, the graph can be represented as a low-dimensional feature vector. Based on the compact graph representation, we can efficiently compute the similarity between graphs and in turn conduct the graph classification task.

In Chapter 6, we propose the MinHash-based algorithm for network nodes, i.e., NetHash, to efficiently capture attributes and structure information of the rooted trees, each of which is rooted at a node in the network. Consequently, we can embed each node in the network into a low-dimensional vector space for implementing attributed network node embedding. Furthermore, we can efficiently compute the similarity between nodes and conduct various network mining tasks, e.g., node classification and retrieval, and link prediction.

7.2 Future Work

In this thesis, we have studied randomized hashing algorithms for structured data, and verified their effectiveness and efficiency on various data mining tasks. However, in the big data era, there are still the following challenges: 1) how to represent high-dimensional data more effectively, i.e., less fingerprints can preserve more information; 2) how to improve efficiency of hashing approaches, for example, further decreasing the number of uniform random variables in the PCWS algorithm; 3) sequence data are also very common in the real world, so it is a good topic to apply randomized hashing techniques to sequence data. In our future work, we are devoted to developing more efficient randomized hashing algorithms that require only moderate computing capacity.

Additionally, existing similarity-preserving hashing techniques can only deal with nested binary sets [65] and tree-structured categorical data [19]. It will be interesting to extend the MinHash algorithm or the CWS scheme to hash nested weighted sets and graphs/network, which not only encode the importance of feature but also preserve the *multi-level exchangeability* [19] of features, in our future work.

References

[1] Dimitris Achlioptas. Database-friendly Random Projections: Johnson-Lindenstrauss with Binary Coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003.

[2] Charu C Aggarwal. On Classification of Graph Streams. In *SDM*, pages 652–663, 2011.

[3] Charu C. Aggarwal. On Classification of Graph Streams. In *SDM*, pages 652–663, 2011.

[4] Charu C Aggarwal and Haixun Wang. *Managing and Mining Graph Data*, volume 40. 2010.

[5] Charu C Aggarwal, Yuchen Zhao, and S Yu Philip. On Clustering Graph Streams. In *SDM*, pages 478–489, 2010.

[6] Alexandr Andoni and Piotr Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*, pages 459–468, 2006.

[7] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient Algorithms for Large-scale Local Triangle Counting. *ACM Transactions on Knowledge Discovery from Data*, 4(3):13, 2010.

[8] Smriti Bhagat, Graham Cormode, and S Muthukrishnan. Node Classification in Social Networks. In *Social Network Data Analytics*, pages 115–148. 2011.

[9] Karsten M. Borgwardt and Hans-Peter Kriegel. Shortest-Path Kernels on Graphs. In *ICDM*, pages 74–81, 2005.

[10] Andrei Z Broder. On the Resemblance and Containment of Documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29, 1997.

[11] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations. In *STOC*, pages 327–336, 1998.

[12] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-Wise Independent Permutations. In *STOC*, pages 327–336, 1998.

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[14] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):1–27, 2011.

[15] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Trans. on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[16] Moses S Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *STOC*, pages 380–388, 2002.

[17] Herman Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis based on the Sum of Observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

[18] Lianhua Chi, Bin Li, and Xingquan Zhu. Fast Graph Stream Classification Using Discriminative Clique Hashing. In *PAKDD*, pages 225–236, 2013.

[19] Lianhua Chi, Bin Li, and Xingquan Zhu. Context-preserving Hashing for Fast Text Classification. In *SDM*, pages 100–108, 2014.

[20] Lianhua Chi, Bin Li, Xingquan Zhu, Shirui Pan, and Ling Chen. Hashing for Adaptive Real-time Graph Stream Classification with Concept Drifts. *IEEE Transactions on Cybernetics*, Accepted(1):1–14, jun 2017. ISSN 2168-2267.

[21] Ondřej Chum, James Philbin, Andrew Zisserman, et al. Near Duplicate Image Detection: Min-Hash and Tf-idf Weighting. In *BMVC*, pages 1–10, 2008.

[22] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and its Applications. *Journal of Algorithm*, 55:58–75, 2005.

[23] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Fast Locality-Sensitive Hashing. In *KDD*, pages 1073–1081, 2011.

[24] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive Hashing Scheme Based on p-stable Sistributions. In *SOCG*, pages 253–262, 2004.

[25] Edd Dumbill. A Revolution That Will Transform How We Live, Work, and Think: An Interview with the Authors of Big Data. *Big Data*, 1(2):73–77, 2013.

[26] Kave Eshghi and Shyamsundar Rajaram. Locality Sensitive Hash Functions Based on Concomitant Rank Order Statistics. In *KDD*, pages 221–229. ACM, 2008.

[27] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9(Aug):1871–1874, 2008.

[28] Wei Fan, Kun Zhang, Hong Cheng, Jing Gao, Xifeng Yan, Jiawei Han, Philip Yu, and Olivier Verscheure. Direct Mining of Discriminative and Essential Frequent Patterns via Model-based Search Tree. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 230–238. ACM, 2008.

[29] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A Large-Scale Study of the Evolution of Web Pages. In *WWW*, pages 669–678, 2003.

[30] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, 2007.

[31] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On Graph Kernels: Hardness Results and Efficient Alternatives. In *COLT*, pages 129–143, 2003.

[32] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB*, pages 721–732, 2005.

[33] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity Search in High Dimensions via Hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[34] Sreenivas Gollapudi and Rina Panigrahy. Exploiting Asymmetry in Hierarchical Topic Extraction. In *CIKM*, pages 475–482, 2006.

[35] Sreenivas Gollapudi and Rina Panigrahy. The Power of Two Min-Hashes for Similarity Search among Heirarchical Data Objects. In *PODS*, pages 211–219, 2008.

[36] Yunchao Gong, Sanjiv Kumar, Vishal Verma, and Svetlana Lazebnik. Angular Quantization-based Binary Codes for Fast Similarity Search. In *NIPS*, pages 1196–1204, 2012.

[37] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-scale Image Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(12):2916–2929, 2013.

[38] David Gorisse, Matthieu Cord, and Frederic Precioso. Locality-Sensitive Hashing for Chi2 Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(2):402–409, 2012.

[39] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, pages 855–864, 2016.

[40] Susan Gunelius. *The Data Explosion in 2014 Minute by Minute Infographic*, Jul 2014. http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic.

[41] Bernhard Haeupler, Mark Manasse, and Kunal Talwar. Consistent Weighted Sampling Made Fast, Small, and Easy. *arXiv preprint arXiv:1410.4266*, 2014.

[42] David Haussler. Convolution Kernels on Discrete Structures. Technical Report UCSC-CRL-99-10, UC Santa Cruz, 1999.

[43] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable Techniques for Clustering the Web. In *WebDB*, pages 129–134, 2000.

[44] Junfeng He, Wei Liu, and Shih-Fu Chang. Scalable Similarity Search with Optimized Kernel Hashing. In *SIGKDD*, pages 1129–1138, 2010.

[45] Monika Henzinger. Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms. In *SIGIR*, pages 284–291, 2006.

[46] Shohei Hido and Hisashi Kashima. A Linear-Time Graph Kernel. In *ICDM*, pages 179–188, 2009.

[47] Piotr Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation. *Journal of the ACM*, 53(3):307–323, 2006.

[48] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: towards Removing the curse of Dimensionality. In *STOC*, pages 604–613, 1998.

[49] Sergey Ioffe. Improved Consistent Sampling, Weighted Minhash and L1 Sketching. In *ICDM*, pages 246–255, 2010.

[50] Prateek Jain, Sudheendra Vijayanarasimhan, and Kristen Grauman. Hashing Hyperplane Queries to Near Points with Applications to Large-Scale Active Learning. In *NIPS*, pages 928–936, 2010.

[51] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. Super-Bit Locality-Sensitive Hashing. In *NIPS*, pages 108–116, 2012.

[52] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Qi Tian, and Bo Zhang. Min-Max Hash for Jaccard Similarity. In *ICDM*, pages 301–309, 2013.

[53] Zhongming Jin, Cheng Li, Yue Lin, and Deng Cai. Density Sensitive Hashing. *Cybernetics, IEEE Transactions on*, 44(8):1362–1371, 2014.

[54] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. Fast and Accurate Hashing via Iterative Nearest Neighbors Expansion. *Cybernetics, IEEE Transactions on*, 44(11):2167–2177, 2014.

[55] Nitin Jindal and Bing Liu. Opinion Spam and Analysis. In *WSDM*, pages 219–230, 2008.

[56] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized Kernels Between Labeled Graphs. In *ICML*, pages 321–328, 2003.

[57] Yan Ke, Rahul Sukthankar, Larry Huston, Yan Ke, and Rahul Sukthankar. Efficient Near-Duplicate Detection and Sub-Image Retrieval. In *ACMMM*, volume 4, page 5, 2004.

[58] Chulyun Kim and Kyuseok Shim. Text: Automatic Template Extraction from Heterogeneous Web Pages. *IEEE Transactions on Knowledge and Data Engineering*, 23 (4):612–626, 2011.

[59] Saehoon Kim and Seungjin Choi. Semi-Supervised Discriminant Hashing. In *ICDM*, pages 1122–1127, 2011.

[60] Weihao Kong and Wu-Jun Li. Isotropic Hashing. In *NIPS*, pages 1646–1654, 2012.

[61] Brian Kulis and Trevor Darrell. Learning to Hash with Binary Reconstructive Embeddings. In *NIPS*, pages 1042–1050, 2009.

[62] Brian Kulis and Kristen Grauman. Kernelized Locality-Sensitive Hashing for Scalable Image Search. In *ICCV*, pages 2130–2137, 2009.

[63] Brian Kulis and Kristen Grauman. Kernelized Locality-Sensitive Hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1092–1104, 2012.

[64] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A New Benchmark Collection for Text Categorization research. *The Journal of Machine Learning Research*, 5:361–397, 2004.

[65] Bin Li, Xingquan Zhu, Lianhua Chi, and Chengqi Zhang. Nested Subtree Hash Kernels for Large-Scale Graph Classification over Streams. In *ICDM*, pages 399–408, 2012.

[66] Jundong Li, Xia Hu, Jiliang Tang, and Huan Liu. Unsupervised Streaming Feature Selection in Social Media. In *CIKM*, pages 1041–1050, 2015.

[67] Ping Li. 0-Bit Consistent Weighted Sampling. In *KDD*, pages 665–674, 2015.

[68] Ping Li and Kenneth W Church. A Sketch Algorithm for Estimating Two-Way and Multi-Way Associations. *Computational Linguistics*, 33(3):305–354, 2007.

[69] Ping Li and Christian König. *b*-Bit Minwise Hashing. In *WWW*, pages 671–680, 2010.

[70] Ping Li, Kenneth W Church, and Trevor Hastie. Conditional Random Sampling: A Sketch-based Sampling Technique for Sparse Data. In *NIPS*, pages 873–880, 2006.

[71] Ping Li, Arnd Konig, and Wenhao Gui. b-Bit Minwise Hashing for Estimating Three-Way Similarities. In *NIPS*, pages 1387–1395, 2010.

[72] Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. b-Bit Minwise Hashing for Large-Scale Learning. In *Big Learning 2011: NIPS 2011 Workshop on Algorithms, Systems, and Tools for Learning at Scale*, December 2011.

[73] Ping Li, Anshumali Shrivastava, Joshua L Moore, and Arnd C König. Hashing Algorithms for Large-Scale Learning. In *NIPS*, pages 2672–2680, 2011.

[74] Ping Li, Art Owen, and Cun-Hui Zhang. One Permutation Hashing. In *NIPS*, pages 3113–3121, 2012.

[75] Xi Li, Guosheng Lin, Chunhua Shen, Anton Hengel, and Anthony Dick. Learning Hash Functions Using Column Generation. In *ICML*, pages 142–150, 2013.

[76] David Liben-Nowell and Jon Kleinberg. The Link-prediction Problem for Social Networks. *Journal of the Association for Information Science and Technology*, 58(7): 1019–1031, 2007.

[77] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. Deep Learning of Binary Hash Codes for Fast Image Retrieval. In *CVPRW*, pages 27–35, 2015.

[78] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, Jie Zhou, et al. Deep Hashing for Compact Binary Codes Learning. In *CVPR*, volume 1, page 3, 2015.

[79] Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Hashing with Graphs. In *ICML*, pages 1–8, 2011.

[80] Wei Liu, Jun Wang, Rongrong Ji, Yu-Gang Jiang, and Shih-Fu Chang. Supervised Hashing with Kernels. In *CVPR*, pages 2074–2081, 2012.

[81] Yang Liu, Jian Shao, Jun Xiao, Fei Wu, and Yueting Zhuang. Hypergraph Spectral Hashing for Image Retrieval with Heterogeneous Social Contexts. *Neurocomputing*, 119:49–58, 2013.

[82] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying Suspicious URLs: an Application of Large-scale Online Learning. In *ICML*, pages 681–688, 2009.

[83] Pierre Mahé and Jean-Philippe Vert. Graph Kernels based on Tree Patterns for Molecules. *Machine Learning*, 75(1):3–35, 2009.

[84] Mark Manasse, Frank McSherry, and Kunal Talwar. Consistent Weighted Sampling. *Unpublished technical report*, 2010.

[85] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting Near-duplicates for Web Crawling. In *WWW*, pages 141–150, 2007.

[86] Jonathan Masci, Michael M Bronstein, Alexander M Bronstein, and Jürgen Schmidhuber. Multimodal Similarity-Preserving Hashing. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):824–830, 2014.

[87] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval*, 3(2):127–163, 2000.

[88] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *NIPS*, pages 3111–3119, 2013.

[89] Michael Mitzenmacher, Rasmus Pagh, and Ninh Pham. Efficient Estimation for High Similarities Using Odd Sketches. In *WWW*, pages 109–118, 2014.

[90] Yadong Mu, Jialie Shen, and Shuicheng Yan. Weakly-Supervised Hashing in Kernel Space. In *CVPR*, pages 3344–3351, 2010.

[91] Behnam Neyshabur and Nathan Srebro. On Symmetric and Asymmetric LSHs for Inner Product Search. In *ICML*, pages 1926–1934, 2015.

[92] Shirui Pan, Xingquan Zhu, Chengqi Zhang, and Philip S Yu. Graph Stream Classification Using Labeled and Unlabeled Graphs. In *ICDE*, pages 398–409, 2013.

[93] Shirui Pan, Jia Wu, Xingquan Zhu, and Chengqi Zhang. Graph Ensemble Boosting for Imbalanced Noisy Graph Stream Classification. *Cybernetics, IEEE Transactions on*, 45(5):940–954, 2015.

[94] Sandeep Pandey, Andrei Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. Nearest-Neighbor Caching for Content-Match Applications. In *WWW*, pages 441–450, 2009.

[95] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *KDD*, pages 701–710, 2014.

[96] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network Embedding as Matrix Factorization: Unifying DeePwalk, LINE, PTE, and node2vec. In *WSDM*, pages 459–467, 2018.

[97] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of Massive Datasets*, volume 1. Cambridge University Press Cambridge, 2012.

[98] Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. Graph Kernels for Chemical Informatics. *Neural Networks*, 18(8):1093–1110, 2005.

[99] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. Randomized Algorithms and NLP: Using Locality Sensitive Hash Function for High Speed Noun Clustering. In *ACL*, pages 622–629, 2005.

[100] Kaspar Riesen and Horst Bunke. Graph Classification by means of Lipschitz Embedding. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39 (6):1472–1483, 2009.

[101] Venu Satuluri and Srinivasan Parthasarathy. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *VLDB*, 5(5):430–441, 2012.

[102] Erick Schonfeld. *Google Processing 20,000 Terabytes A Day, And Growing*, Jan 2008. http://techcrunch.com/2008/01/09/google-processing-20000-terabytes-a-day-and-growing.

[103] Gregory Shakhnarovich, Paul Viola, and Trevor Darrell. Fast Pose Estimation with Parameter-Sensitive Hashing. In *ICCV*, page 750, 2003.

[104] Jian Shao, Fei Wu, Chuanfei Ouyang, and Xiao Zhang. Sparse Spectral Hashing. *Pattern recognition letters*, 33(3):271–277, 2012.

[105] Fumin Shen, Chunhua Shen, Qinfeng Shi, Anton Van Den Hengel, and Zhenmin Tang. Inductive Hashing on Manifolds. In *CVPR*, pages 1562–1569, 2013.

[106] Fumin Shen, Chunhua Shen, Wei Liu, and Heng Tao Shen. Supervised Discrete Hashing. In *CVPR*, volume 2, page 5, 2015.

[107] Nino Shervashidze and Karsten Borgwardt. Fast Subtree Kernels on Graphs. In *NIPS*, pages 1660–1668, 2009.

[108] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[109] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S.V.N. Vishwanathan. Hash Kernels for Structured Data. *Journal of Machine Learning Research*, 10:2615–2637, 2009.

[110] Anshumali Shrivastava. Exact Weighted Minwise Hashing in Constant Time. *arXiv preprint arXiv:1602.08393*, 2016.

[111] Anshumali Shrivastava. Simple and Efficient Weighted Minwise Hashing. In *NIPS*, pages 1498–1506, 2016.

[112] Anshumali Shrivastava and Ping Li. In Defense of Minhash Over SimHash. In *AISTATS*, pages 886–894, 2014.

[113] Anshumali Shrivastava and Ping Li. Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search. In *ICML*, pages 557–565, 2014.

[114] Jingkuan Song, Yi Yang, Xuelong Li, Zi Huang, and Yang Yang. Robust Hashing with Local Models for Approximate Similarity Search. *Cybernetics, IEEE Transactions on*, 44(7):1225–1236, 2014.

[115] Christoph Strecha, Alex Bronstein, Michael Bronstein, and Pascal Fua. LDAHash: Improved Matching with Smaller Descriptors. *IEEE transactions on pattern analysis and machine intelligence*, 34(1):66–78, 2012.

[116] Danny Sullivan. *Google Still Doing At Least 1 Trillion Searches Per Year*, Jan 2015. http://searchengineland.com/google-1-trillion-searches-per-year-212940.

[117] Donna Tam. *Facebook processes more than 500 TB of data daily*, Jul 2014. http://www.cnet.com/news/facebook-processes-more-than-500-tb-of-data-daily.

[118] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-scale Information Network Embedding. In *WWW*, pages 1067–1077, 2015.

[119] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. Visualizing Large-scale and High-dimensional Data. In *WWW*, pages 287–297, 2016.

[120] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD*, pages 990–998, 2008.

[121] Shirish Tatikonda and Srinivasan Parthasarathy. Hashing Tree-Structured Data: Methods and Applications. In *ICDE*, pages 429–440, 2010.

[122] Carlos H. C. Teixiera, Arlei Silva, and Wagner Meira Jr. Min-Hash Fingerprints for Graph Kernels: A Trade-off among Accuracy, Efficiency, and Compression. *Journal of Information and Data Management*, 3(3):227–242, 2012.

[123] Cunchao Tu, Han Liu, Zhiyuan Liu, and Maosong Sun. Cane: Context-Aware Network Embedding for Relation Modeling. In *ACL*, volume 1, pages 1722–1731, 2017.

[124] Ferhan Ture, Tamer Elsayed, and Jimmy Lin. No Free Lunch: Brute Force vs. Locality-Sensitive Hashing for Cross-Lingual Pairwise Similarity. In *SIGIR*, pages 943–952, 2011.

[125] Tanguy Urvoy, Emmanuel Chauveau, Pascal Filoche, and Thomas Lavergne. Tracking Web Spam with HTML Style Similarities. *ACM Transactions on the Web (TWEB)*, 2(1):3, 2008.

[126] V Vapnik. *Statistical Learning Theory*. John Wiley, 1998.

[127] S. Vijayanarasimhan, P. Jain, and K. Grauman. Hashing Hyperplane Queries to Near Points with Applications to Large-Scale Active Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(2):276–288, Feb 2014.

[128] S.V.N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph Kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.

[129] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural Deep Network Embedding. In *KDD*, pages 1225–1234, 2016.

[130] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for Similarity Search: A Survey. *arXiv preprint arXiv:1408.2927*, 2014.

[131] Jun Wang, Wei Liu, Andy X Sun, and Yu-Gang Jiang. Learning Hash Codes with Listwise Supervision. In *ICCV*, pages 3032–3039, 2013.

[132] S. Wang, Q. Huang, S. Jiang, and Q. Tian. S^3MKL: Scalable Semi-Supervised Multiple Kernel Learning for Real-World Image Applications. *IEEE Transactions on Multimedia*, 14(4):1259–1274, Aug 2012.

[133] Shuhui Wang, Shuqiang Jiang, Qingming Huang, and Qi Tian. S^3MKL: Scalable Semi-Supervised Multiple Kernel Learning for Image Data Mining. In *ACMMM*, pages 163–172, 2010.

[134] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature Hashing for Large Scale Multitask Learning. In *ICML*, pages 1113–1120, 2009.

[135] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature Hashing for Large Scale Multitask Learning. In *ICML*, pages 1113–1120, 2009.

[136] B. Ju. Weisfeiler and A. A. Leman. A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction. *Nauchno-Technicheskaya Informatsia*, 2 (9):12–16, 1968.

[137] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral Hashing. In *NIPS*, pages 1753–1760, 2009.

[138] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Canonical Consistent Weighted Sampling for Real-Value Weighted Min-Hash. In *ICDM*, pages 1287–1292, 2016.

[139] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Cross-View Feature Hashing for Image Retrieval. In *PAKDD*, pages 203–214, 2016.

[140] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Consistent Weighted Sampling Made More Practical. In *WWW*, pages 1035–1043, 2017.

[141] Wei Wu, Bin Li, Ling Chen, Chengqi Zhang, and Philip S Yu. Improved Consistent Weighted Sampling Revisited. *arXiv preprint arXiv:1706.01172*, 2017.

[142] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Efficient Attributed Network Embedding via Recursive Randomized Hashing. In *IJCAI-18*, pages 2861–2867, 2018.

[143] Wei Wu, Bin Li, Ling Chen, Xingquan Zhu, and Chengqi Zhang. K-Ary Tree Hashing for Fast Graph Classification. *IEEE Transactions on Knowledge and Data Engineering*, 30(5):936–949, 2018.

[144] Hao Xia, Pengcheng Wu, Steven CH Hoi, and Rong Jin. Boosting Multi-Kernel Locality-Sensitive Hashing for Scalable Image Retrieval. In *SIGIR*, pages 55–64, 2012.

[145] Yun Xiong, Yangyong Zhu, and S Yu Philip. Top-K Similarity Join in Heterogeneous Information Networks. *IEEE Transactions on Knowledge and Data Engineering*, 27 (6):1710–1723, 2015.

[146] Xifeng Yan and Jiawei Han. gSpan: Graph-based Substructure Pattern Mining. In *ICDM*, pages 721–724, 2002.

[147] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network Representation Learning with Rich Text Information. In *IJCAI*, pages 2111–2117, 2015.

[148] Dingqi Yang, Bin Li, and Philippe Cudré-Mauroux. POIsketch: Semantic Place Labeling over User Activity Streams. In *IJCAI*, pages 2697–2703, 2016.

[149] Dingqi Yang, Bin Li, Laura Rettig, and Philippe Cudré-Mauroux. HistoSketch: Fast Similarity-Preserving Sketching of Streaming Histograms with Concept Drift. In *ICDM*, 2017.

[150] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. A Generic Method for Accelerating LSH-Based Similarity Join Processing. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):712–726, 2017.

[151] Yi Yu, Michel Crucianu, Vincent Oria, and Ernesto Damiani. Combining Multi-Probe Histogram and Order-Statistics Based LSH for Scalable Audio Content Retrieval. In *ACMMM*, pages 381–390, 2010.

[152] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. Homophily, Structure, and Content Augmented Network Representation Learning. In *ICDM*, pages 609–618, 2016.

[153] Dongqing Zhang and Wu-Jun Li. Large-Scale Supervised Multimodal Hashing with Semantic Correlation Maximization. In *AAAI*, volume 1, page 7, 2014.

[154] Lei Zhang, Yongdong Zhang, Xiaoguang Gu, Jinhui Tang, and Qi Tian. Scalable Similarity Search with Topology Preserving Hashing. *IEEE Transactions on Image Processing*, 23(7):3025–3039, 2014.

[155] Yan-Ming Zhang, Kaizhu Huang, Xinwen Hou, and Cheng-Lin Liu. Learning Locality Preserving Graph from Data. *Cybernetics, IEEE Transactions on*, 44(11):2088–2098, 2014.

[156] Wan-Lei Zhao, Hervé Jégou, and Guillaume Gravier. Sim-Min-Hash: An Efficient Matching Technique for Linking Large Image Collections. In *ACM MM*, pages 577–580, 2013.

[157] Yueting Zhuang, Yang Liu, Fei Wu, Yin Zhang, and Jian Shao. Hypergraph Spectral Hashing for Similarity Search of Social Image. In *ACM MM*, pages 1457–1460, 2011.