K-Ary Tree Hashing for Fast Graph Classification

Wei Wu, Bin Li, Ling Chen, Xingquan Zhu, *Senior Member, IEEE*, and Chengqi Zhang, *Senior Member, IEEE*

Abstract—Existing graph classification usually relies on an exhaustive enumeration of substructure patterns, where the number of substructures expands exponentially w.r.t. with the size of the graph set. Recently, the Weisfeiler-Lehman (WL) graph kernel has achieved the best performance in terms of both accuracy and efficiency among state-of-the-art methods. However, it is still time-consuming, especially for large-scale graph classification tasks. In this paper, we present a K-Ary Tree based Hashing (KATH) algorithm, which is able to obtain competitive accuracy with a very fast runtime. The main idea of KATH is to construct a traversal table to quickly approximate the subtree patterns in WL using K-ary trees. Based on the traversal table, KATH 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. Our experimental results on both real world and synthetic data sets show that KATH runs significantly faster than state-of-the-art methods while achieving competitive or better accuracy.

Index Terms—Graph classification, labeled graphs, MinHash, tree hashing, randomized hashing

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 [1], [2], [3], [4], [5], [6], [7], [8], [9]. 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 to facilitate classifier training within a generic machine learning framework, such as support vector machines (SVM) [10] and logistic regressions.

In the past, there have been numerous works devoted to extracting substructures, e.g., walks [11], [12], paths [13], and subtrees [14], [15]. 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 [16], the p-random walk kernel [11], [12], and the shortest path kernel [13], may take hours or even days to construct a kernel matrix [17].

Recently, the Weisfeiler-Lehman (WL) graph kernel [17] 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 [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], or similarly, random projections [35], [36] 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

• W. Wu, L. Chen, and C. Zhang are with the Centre for Artificial Intelligence, FEIT, University of Technology Sydney, Ultimo, NSW 2007, Australia. E-mail: william.third.wu@gmail.com, {ling.chen, chengqi.zhang}@uts.edu.au.
• B. Li is with the School of Computer Science, Fudan University, Shanghai 200433, China. E-mail: libin@fudan.edu.cn.
• X. Zhu is with the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, and the School of Computer Science, Fudan University, Shanghai 200433, China. E-mail: xzhu3@fau.edu.

*Manuscript received 24 Apr. 2017; revised 16 Nov. 2017; accepted 27 Nov. 2017. Date of publication 11 Dec. 2017; date of current version 30 Mar. 2018. (Corresponding author: Wei Wu.)
Recommended for acceptance by H. Xiong.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2782278*

vectors. Few studies on sketching graphs have been reported. To the best of our knowledge, only three works so far, [37], [38], [39], have exploited hashing techniques to accelerate graph feature extraction. In particular, [37] hashes the edge set of graphs, [38] hashes the set of node labels, and [39] hashes the set of cliques. Among the three methods, [37] and [39] cannot be applied to sketching labeled graphs as they are designed for sketching network flows. The Nested Subtree Hash (NSH) kernel, proposed in [38], employs a random hashing scheme to improve the efficiency of the WL graph kernel [17] in both time and space. However, it suffers from some performance loss compared to [17]. 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 [17] 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* to facilitate fast approximation of the subtree patterns in WL. Given a traversal table constructed from the node labels and edges of a graph, a subtree of depth r is efficiently acquired through r times of recursive matrix indexing. Moreover, a subtree pattern can be uniquely specified by ordered leaf nodes. We generate all subtree patterns, rooted at individual nodes of a graph, of a specified depth and adopt the MinHash scheme to fingerprint subtree patterns. By virtue of recursive indexing, we obtain almost the same subtree 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 compare methods [17], [38] for fast graph 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 [17] with the best classification performance and NSH [38] with the highest efficiency in both time and space, KATH not only *achieves a similar classification performance to [17]* but also *performs much faster and used less space than [38]*. 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 paper is organized as follows: Section 2 introduces the necessary preliminary knowledge. We introduce the KATH algorithm for fast graph feature extraction in Section 3. The experimental results are presented in Section 4 and related work is discussed in Section 5. Finally, we conclude the paper in Section 6.

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.

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 $\ell : \mathcal{V} \rightarrow \mathcal{L}$ 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 [15], [17], [38]). The goal of graph classification is to learn a classifier from $\{\mathbf{x}_i, y_i\}_{i=1}^N$ to classify unseen graphs.

2.2 Weisfeiler-Lehman Graph Kernels

Weisfeiler-Lehman (WL) graph kernels [17] (or fast subtree kernels [15]) are known as the fastest graph kernel family, based on the Weisfeiler-Lehman (WL) isomorphism test [40]. 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:

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

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 rth iteration. $\mathbf{x} = [\mathbf{x}^{(1)}; \ldots; \mathbf{x}^{(R)}]$, where $\mathbf{x}^{(r)}$ is the generated fingerprint in the rth 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 Fig. 1a as an example. First, $g^{(1)}$ 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 shown in the table of Fig. 1a, there are three subtrees of depth 0: a, b and c, with the occurrence numbers

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

as 1, 2 and 1 respectively. Therefore, we have $\mathbf{x}^{(1)} = (1, 2, 1)$. Fig. 1b 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 Fig. 1b, 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 $O(NR|\mathcal{E}| + N^2R|\mathcal{V}|)$ 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.

2.3 Nested Subtree Hash Kernels

A hash kernel based on the WL isomorphism test, say Nested Subtree Hash (NSH) kernels [38], 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,

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

$$= \sum_{r=1}^{R} \langle \bar{\mathbf{x}}_i^{(r)}, \bar{\mathbf{x}}_j^{(r)} \rangle = \langle \bar{\mathbf{x}}_i, \bar{\mathbf{x}}_j \rangle$$

where $\mathbf{x}_i^{(r)}$ and $\mathbf{x}_j^{(r)}$ denote the complete feature vectors in the rth 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 Fig. 1a again. The first two iterations of NSH to find $\{\bar{\mathbf{x}}_i^{(1)}\}$ and $\{\bar{\mathbf{x}}_i^{(2)}\}$ are shown in Figs. 1c $\sim$ 1d. In Fig. 1c, 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,

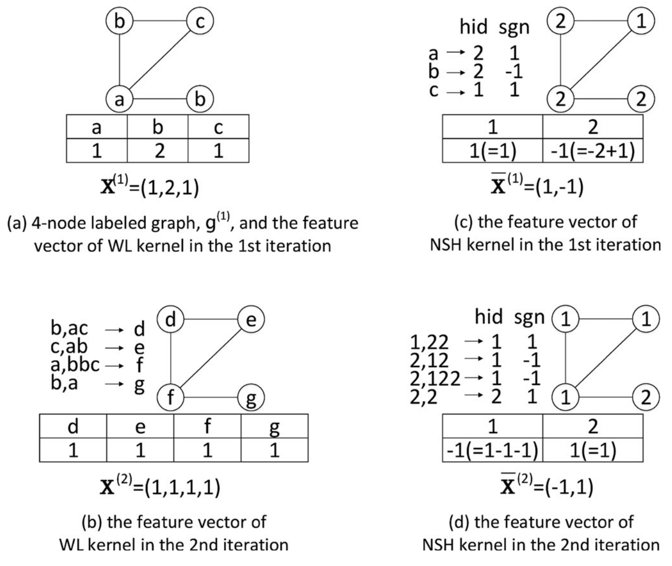

a
b
c
1
2
1
X(1)=(1,2,1)
(a) 4-node labeled graph, g(1), and the feature vector of WL kernel in the 1st iteration
hid sgn
a → 2 1
b → 2 -1
c → 1 1
1
2
1(=1)
-1(=-2+1)
X̄(1)=(1,-1)
(c) the feature vector of NSH kernel in the 1st iteration
b,ac → d
c,ab → e
a,bbc → f
b,a → g
d e f g
1 1 1 1
X(2)=(1,1,1,1)
(b) the feature vector of WL kernel in the 2nd iteration
hid sgn
1,22 → 1 1
2,12 → 1 -1
2,122 → 1 -1
2,2 → 2 1
1
2
-1(=1-1-1)
1(=1)
X̄(2)=(-1,1)
(d) the feature vector of NSH kernel in the 2nd iteration

Fig. 1. Illustration of the WL kernel and the NSH kernel with $r = 2$.

given the three subtrees of depth 0, a, b and c, in Fig. 1a, the hid function in Fig. 1c 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 [38]. Afterwards, the original graph is relabelled with the hash values. Fig. 1d 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.

2.4 MinHash Scheme

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 [41] which is used in our method. MinHash is an approximate method for measuring the similarity of two sets, say $\mathcal{S}_i$ and $\mathcal{S}_j$. A set of D hash functions (random permutations) $\{\pi_d\}_{d=1}^{D}$ are applied to the elements in $\mathcal{S}_*$ and we say $\min(\pi_d(\mathcal{S}_*))$ is a MinHash of $\mathcal{S}_*$. A nice property of MinHash is that the probability of $\mathcal{S}_i$ and $\mathcal{S}_j$ to generate the same MinHash value is exactly the Jaccard similarity of the two sets:

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

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

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

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

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

$$\pi_d(i) = \texttt{mod}((a_d i + b_d), c_d) \tag{1}$$

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

3 K-ARY TREE HASHING

In this section, we propose a K-Ary Tree based Hashing (KATH) algorithm to 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 [15], [17], [38]. Recall that in these algorithms, the relabeling process for node v in the rth WL isomorphism test iteration only considers its direct neighboring node set $\mathcal{N}_v$ to assign a new label to the ordered label string $\ell^{(r)}(v, \texttt{sort}(\mathcal{N}_v))$, which uniquely encodes a subtree pattern. In each WL isomorphism test iteration, the node 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 rth 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 performance of the two solutions empirically in Section 4. In the following section, we elaborate on our KATH algorithm which resembles WL isomorphism test based methods but is much more efficient.

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. (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 1. 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 Fig. 2.

Algorithm 1. K-Ary Tree Based Hashing

```
Require: $g = (\mathcal{V}, \mathcal{E}, \ell)$, $K$, $\{D^{(r)}\}_{r=1}^{R}$
Ensure: $\{x^{(r)}\}_{r=1}^{R}$
 1: $V \leftarrow |\mathcal{V}|$
 2: $\ell(V+1) \leftarrow \infty$
 3: $\mathbf{T} \leftarrow (V+1) * \texttt{ones}(V+1, 1+K)$
 4: for $v = 1 : V$ do
 5:     $\mathcal{N}_v \leftarrow \texttt{neighbor}(v)$
 6:     $\mathbf{T}(v) = \texttt{Selection\_Solution}(\ell, \mathcal{N}_v)$ // Use Algorithm 2 or 3
 7: end for
 8: $\mathbf{z}^{(1)} \leftarrow [1 : V]^\top$
 9: $\mathbf{S}^{(1)} \leftarrow \ell(\mathbf{z}^{(1)})$
10: for $r = 1 : R$ do
11:     if $r > 1$ then
12:         $\mathbf{z}^{(r)} \leftarrow \texttt{reshape}(\mathbf{T}(\mathbf{z}^{(r-1)}, :), [1, *])$
13:         $\mathbf{S}^{(r)} \leftarrow \texttt{reshape}(\ell(\mathbf{z}^{(r)}), [V, *])$
14:     end if
15:     $\mathbf{f}^{(r)} \leftarrow [\hbar(\mathbf{S}^{(r)}(1,:)), \ldots, \hbar(\mathbf{S}^{(r)}(V,:))]^\top$
16:     $\mathbf{x}^{(r)} \leftarrow [\texttt{min}(\pi_1^{(r)}(\mathbf{f}^{(r)})), \ldots, \texttt{min}(\pi_{D^{(r)}}^{(r)}(\mathbf{f}^{(r)}))]^\top$
17: end for
```

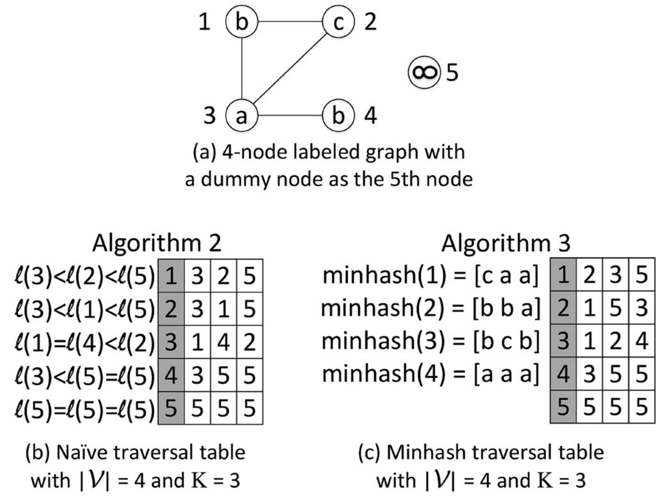
Algorithm 2. Naive Selection

```
Require: $\ell, \mathcal{N}_v$
Ensure: $\mathbf{T}(v)$
1: $temp \leftarrow \texttt{sort}(\ell(\mathcal{N}_v))$
2: $k \leftarrow \texttt{min}(K, |\mathcal{N}_v|)$
3: $\mathbf{T}(v) \leftarrow [v, \texttt{index}(temp(1:k))]$
```

Algorithm 3. MinHash Selection

```
Require: $\ell, \mathcal{N}_v$
Ensure: $\mathbf{T}(v)$
1: $temp \leftarrow [\texttt{min}(\pi_1(\ell(\mathcal{N}_v))), \ldots, \texttt{min}(\pi_K(\ell(\mathcal{N}_v)))]$
2: $\mathbf{T}(v) \leftarrow [v, \texttt{index}(temp)]$
```

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.


1 b
c 2
∞ 5
3 a
b 4
(a) 4-node labeled graph with a dummy node as the 5th node
Algorithm 2
Algorithm 3
(b) Naïve traversal table with |V| = 4 and K = 3
(c) Minhash traversal table with |V| = 4 and K = 3

ℓ(3)<ℓ(2)<ℓ(5)	1	3	2	5
ℓ(3)<ℓ(1)<ℓ(5)	2	3	1	5
ℓ(1)=ℓ(4)<ℓ(2)	3	1	4	2
ℓ(3)<ℓ(5)=ℓ(5)	4	3	5	5
ℓ(5)=ℓ(5)=ℓ(5)	5	5	5	5

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

Fig. 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.

3.1.1 Traversal Table Construction

After receiving a graph $g = \{\mathcal{V}, \mathcal{E}, \ell\}$, the algorithm first adds a dummy node as the $(V+1)$th node of g (for example, node 5 in Fig. 2a), where $V = |\mathcal{V}|$ (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, Algorithms 2 and 3, to address the problem.

Given the set of neighbors of node v as $\mathcal{N}_v$, Algorithm 2 is a naive solution that only considers the first K neighbors. Essentially, we first sort the nodes in $\mathcal{N}_v$ 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; $k = |\mathcal{N}_v|$ otherwise (Line 2). Last, v and the indices of the first k nodes in the sorted array are stored in the vth 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 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 3. 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 vth 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 Fig. 2b and 2c. 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. Fig. 2b 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. Fig. 2c 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 4.

3.1.2 Recursive Leaf Extension

For ease of explanation, in this section and the next section, 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 Fig. 2 to illustrate this operation:

$$\mathbf{z}^{(1)} = [\underline{1}234]$$

$$\mathbf{z}^{(2)} = \big[\underline{[1\overline{3}25]}[2315][3142][4355]\big]$$

$$\mathbf{z}^{(3)} = \Big[\underline{[[1325]\overline{[3142]}[2315][5555]]}\,[\ldots]\,[\ldots]\,[\ldots]\,[\ldots]\Big]$$

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 vth 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

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

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.

3.1.3 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. (1) and find the smallest one as a MinHash $\mathtt{min}(\pi_d^{(r)}(\mathbf{f}^{(r)}))$. In the rth 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 rth 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 Fig. 2 for illustration. The node label arrays in the vth 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 dth MinHash $x_{\cdot,d}^{(2)} = \mathtt{min}(\pi_d^{(2)}(\mathbf{f}^{(2)}))$.

Finally, we construct the kernel matrix $\mathbf{K}$ on $\mathcal{G}$ 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

$$\mathbf{K}_{i,j} = \kappa(g_i, g_j) = \sum_{r=1}^{R} \sum_{d=1}^{D^{(r)}} \frac{\mathbf{1}(x_{i,d}^{(r)} = x_{j,d}^{(r)})}{D^{(r)}} \quad (2)$$

where $\mathbf{1}(state) = 1$ if $state$ is true and $\mathbf{1}(state) = 0$ otherwise. Eq. (2) calculates the ratio of the same MinHashes between $\mathbf{x}_i^{(r)}$ and $\mathbf{x}_j^{(r)}$, which is exactly the Tanimoto kernel [42].

3.2 Complexity Analysis

The theoretical computational complexity of the KATH algorithm for fingerprinting a graph set is $O(NR|\mathcal{V}|)$, compared to $O(NR|\mathcal{E}|)$ for feature vector construction in WL and NSH. In particular, $O(NR|\mathcal{V}|)$ is for fingerprinting N graphs in R iterations (Lines 10–17 in Algorithm 1), where each graph requires $O(|\mathcal{V}|)$ 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).

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

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 $O(NR|\mathcal{E}|)$ 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.

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 [17] and [38], respectively. One group comes from PubChem.[4] The fourth group consists of two data sets from social networks: DBLP [43] and twitter [5], [8]. We also generate a synthetic graph classification data set. Specifically, we adopt the following benchmarks (17 graph data sets in total):

4.1 Data Sets

Real-World Mix [17]. The first group comprises 3 real-world graph data sets[5] used as the testbed in [17]: (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.

Real-World NCI [38]. The second group comprises 9 NCI data sets[6] used as the testbed in [38]. 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 percent positive samples), following [38], 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).

Real-World 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.

Real-World 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.

Real-World DBLP [43]. Each record in DBLP represents one paper, containing paper ID, title, abstract, authors, year of publication, venue, and references of the paper. Following [43], 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.

Real-World Twitter [5], [8]. Following [5], [8], 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] [44] 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 1, in which $|\mathcal{C}|$ denotes the number of classes in each data set, $|\mathcal{G}|$ denotes the number of graphs, $|\mathcal{V}|$ and $|\mathcal{E}|$ denote the number of nodes and edges of graphs, respectively, and $|\mathcal{L}|$ denotes the number of unique node labels in a graph data set.

4. http://pubchem.ncbi.nlm.nih.gov/
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
9. http://www.cais.ntu.edu.sg/~jamescheng/graphgen1.0.zip
10. http://www.cs.ucsb.edu/~xyan/software/gSpan.htm

TABLE 1
Summary of the Five Groups of Graph Data Sets

Data Set	$\|\mathcal{C}\|$	$\|\mathcal{G}\|$	avg. $\|\mathcal{V}\|$	avg. $\|\mathcal{E}\|$	$\|\mathcal{L}\|$
MUTAG [17]	2	188	17.93	19.80	7
ENZYMES [17]	6	600	32.63	62.14	3
D&D [17]	2	1,178	284.32	715.66	82
NCI1 [38]	2	4,464	31.01	33.61	51
NCI33 [38]	2	3,612	31.14	33.73	48
NCI41 [38]	2	3,380	31.33	33.90	39
NCI47 [38]	2	4,362	31.06	33.65	52
NCI81 [38]	2	5,240	30.34	32.86	50
NCI83 [38]	2	4,874	30.39	32.90	40
NCI109 [38]	2	4,504	30.83	33.41	51
NCI123 [38]	2	6,776	29.66	32.12	57
NCI145 [38]	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 [43]	2	19,456	10.48	19.64	41,325
Twitter [5], [8]	2	20,000	4.03	5.00	1,307
Syn10K20L10E	2	10,000	8.74	11.77	20

4.2 Experimental Preliminaries

The classification performance of the compared methods is evaluated using `libsvm` [45] 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 8×3.1 GHz Intel Xeon CPU (64 bit) and 126 G RAM.

We only compare KATH with existing graph feature extraction methods that are scalable to large graph classification problems, say 10^4 graphs with tens of labeled nodes. The comprehensive graph classification study in [17] shows that the WL graph kernel proposed therein (also known as fast subtree kernels [15]) easily outperform all the compared classical graph kernels, including the fast geometric random walk kernel [16], the p-random walk kernel (a special case of [11], [12]), and the shortest path kernel [13], 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 $Depth \in \{0, \dots, 4\}$, where $Depth$ denotes the depth of subtrees. In the NSH algorithm, the dimensionality setting follows $[30, 500, 5000, 10000, 10000, 10000]$ used in [38] 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.

4.3 Results on Real-World Mix

Fig. 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

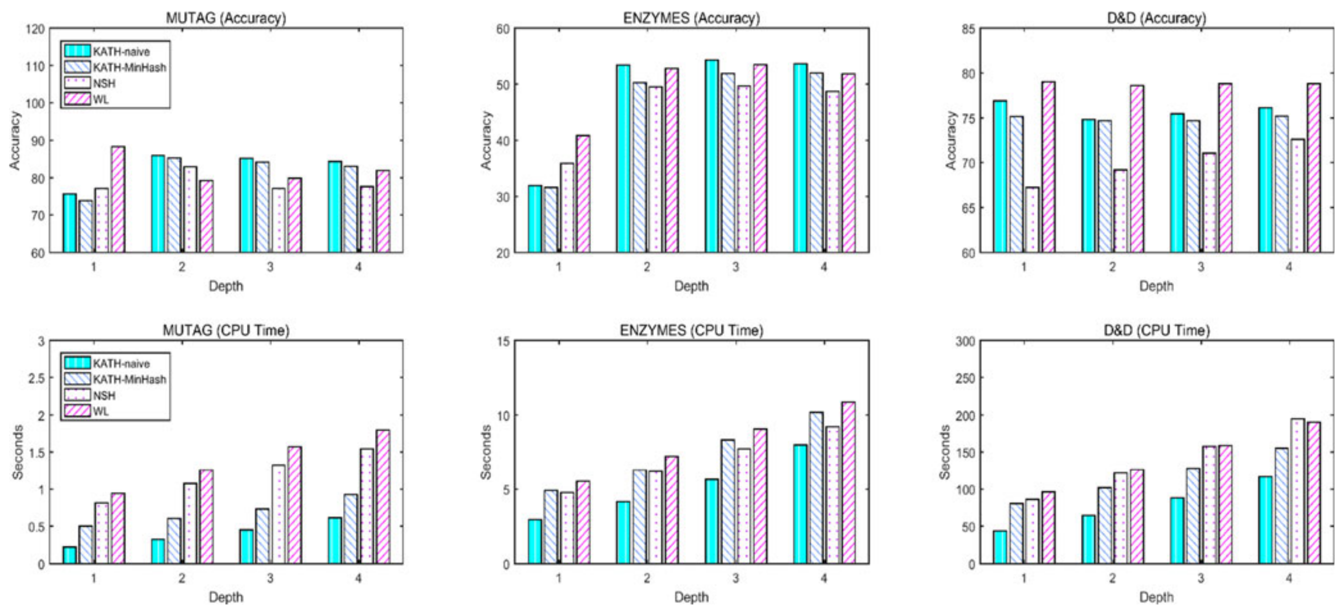


Fig. 3. Classification Accuracy (upper row) and CPU Time (bottom row) comparison results on the three graph data sets in the group of Real-World Mix. The x-axis denotes the depth of subtrees.

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 at least 40 percent 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.

4.4 Results on Real-World NCI

Fig. 4 reports 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 4.3, KATH-MinHash becomes more effective when the scaling size of the data sets increases. This also conforms to the above intuitive explanation.

4.5 Results on Real-World qHTS

We further compare the two variations of KATH with its competitors on a large data set, qHTS. The experimental results are reported in Fig. 5. 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 variations of KATH are inferior to WL (around 5 percent) 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.

4.6 Results on Real-World NCI-Yeast

Fig. 6 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 4.5, KATH-MinHash performs remarkably better than KATH-naive in terms of accuracy, which indicates that KATH-MinHash is 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 percent), 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.

4.7 Results on Real-World DBLP

In addition to the graphs of chemical compounds, we have also investigated the graphs obtained from social networks. Fig. 7 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 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 of labels while the size of the traversal table is much smaller than the number of labels ($K = 4 \ll |\mathcal{L}| = 41,325$). 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 percent) 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.

4.8 Results on Real-World Twitter

We also report the experimental results on the second graph data set of social network, Twitter, in Fig. 8. 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 4.7, that is, the size of the traversal table is much smaller than the number of labels ($K = 4 \ll |\mathcal{L}| = 1{,}307$). Likewise, in this experiment WL outperforms our KATH algorithms by around 4 percent 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.

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. Fig. 9 illustrates the experimental results on the synthetic data set of size 10 K. 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 $1/3$

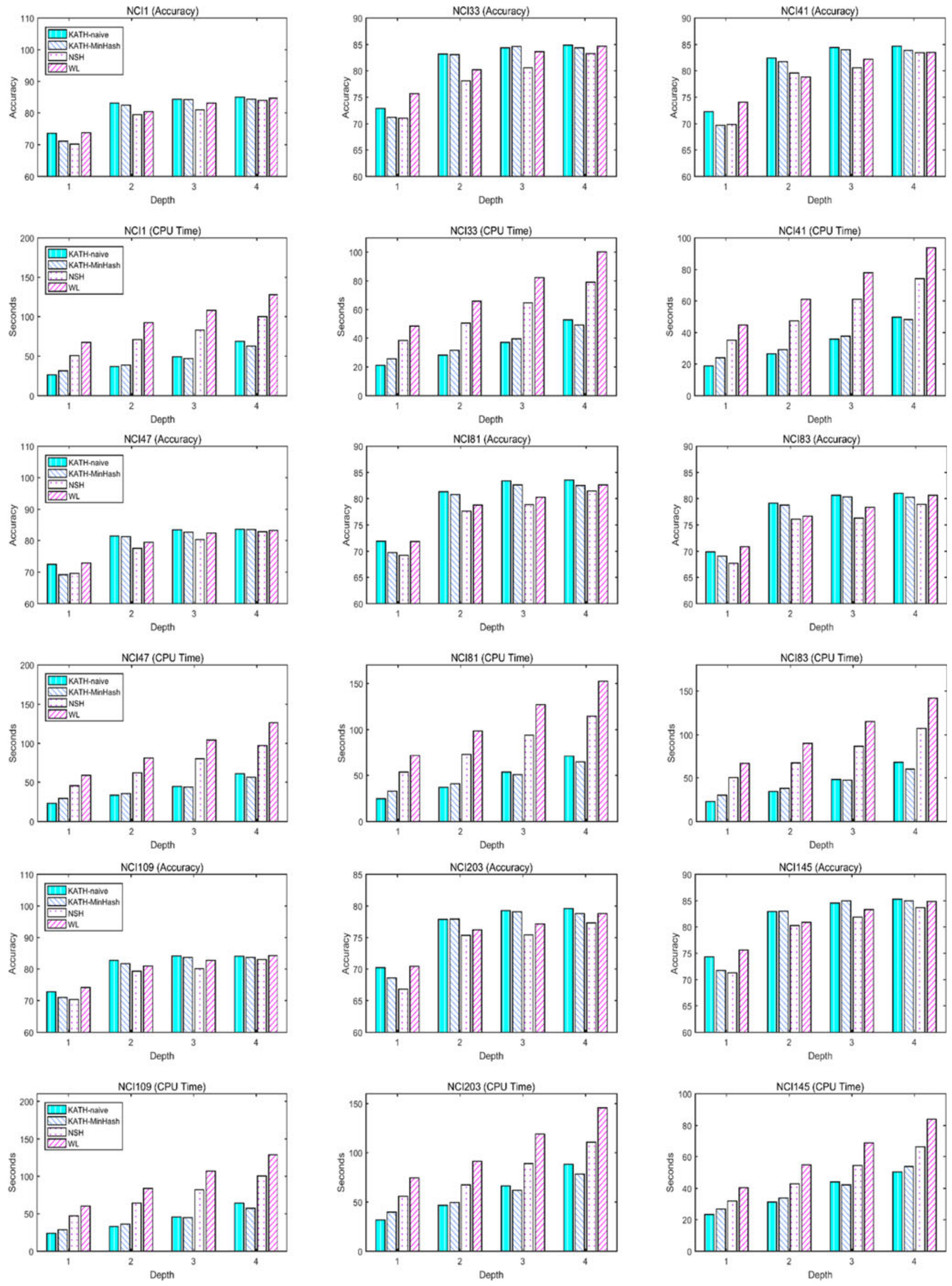


Fig. 4. Classification Accuracy (odd rows) and CPU Time (even rows) comparison results on the nine graph data sets in NCI. The x-axis denotes the depth of subtrees.

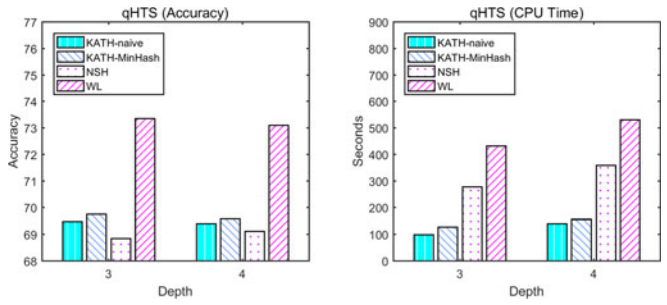


Fig. 5. Classification Accuracy (left) and CPU Time (right) comparison results on qHTS. The x-axis denotes the depth of subtrees.

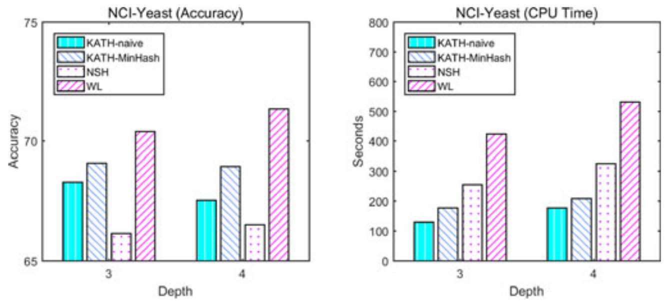


Fig. 6. Classification Accuracy (left) and CPU Time (right) comparison results on NCI-Yeast. The x-axis denotes the depth of subtrees.

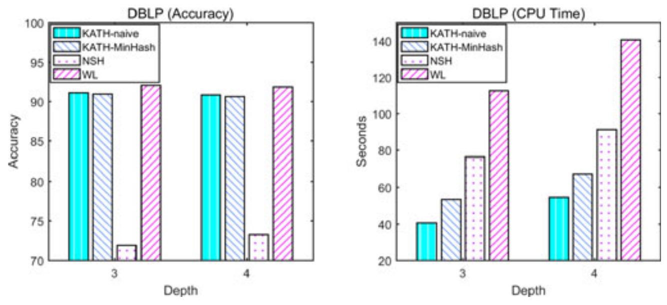


Fig. 7. Classification Accuracy (left) and CPU Time (right) comparison results on DBLP. The x-axis denotes the depth of subtrees.

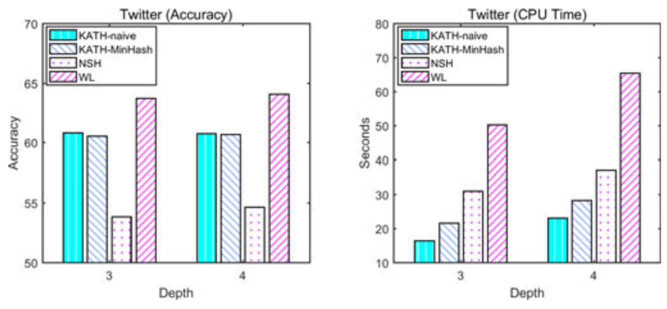


Fig. 8. Classification Accuracy (left) and CPU Time (right) comparison results on Twitter. The x-axis denotes the depth of subtrees.

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.

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

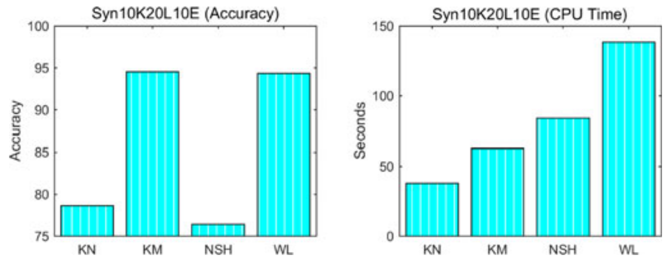


Fig. 9. Classification Accuracy and CPU Time comparison results on Synthetic 10K20L10E (KN short for KATH-naive, and KM short for KATH-MinHash).

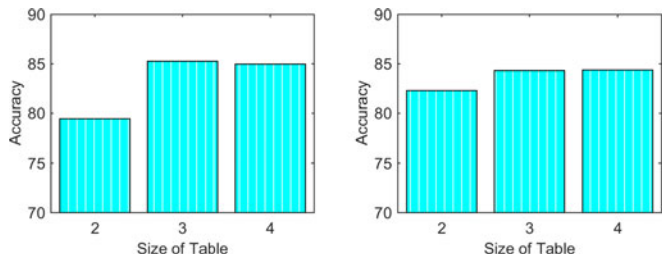


Fig. 10. Classification Accuracy comparison results on Real-World NCI1. The x-axis denotes the size of traversal table. The left figure represents KATH-naive, and the right figure represents KATH-MinHash.

TABLE 2
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%

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 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.

Fig. 10 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 2, there are only 3.72 percent 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 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 [46]. Typical substructures for graph representation include walks [11], [16], paths [13], subtrees (neighboring nodes) [14], [15], [47], and subgraphs (usually based on a frequent subgraph mining technique, e.g., [44], [48]). The walk-based approach [11] 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 [16]. The kernels for labeled graphs, can be computed efficiently by extending linear algebra. The path-based approach [13] 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 [17]. The subtree-based approach [14] constructs a family of graph kernels by detecting common subtree patterns as features to represent graphs. This kernel presents good performance on toxicity and anti-cancer activity prediction for small graph models, but it could not solve large graphs with high degrees in average. In [15], 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 [47] 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, [44], [48] 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 [49]. A 2-dimensional hashing scheme is employed to construct an "in-memory" summary of the graph streams in [37]. 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, [39] 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 [19]. 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 [15] by employing a random hashing scheme for recursively hashing tree structures. The other is MinHash Fingerprints (MHF) [50], which first enumerates all the paths (between a certain 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 [51], [52]. The hierarchical MinHash algorithm [51] 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, [51] 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 [52], 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.

6 Conclusion

In this paper, we propose a graph feature extraction (including fingerprinting) 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 subtree pattern. By virtue of the recursive indexing process, we can quickly obtain almost the same subtree 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.

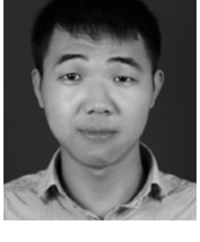
Acknowledgments

This work was partially supported by the Fudan University Startup Research Grant and the Program for Professor of Special Appointment (Eastern Scholar) at the Shanghai Institutions of Higher Learning.

References

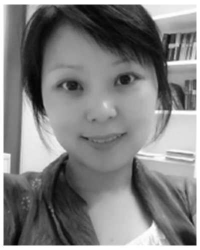
[1] C. C. Aggarwal and H. Wang, *Managing and Mining Graph Data*, vol. 40. Berlin, Germany: Springer, 2010.
[2] C. C. Aggarwal, Y. Zhao, and S. Y. Philip, "On clustering graph streams," in *Proc. SIAM Int. Conf. Data Mining*, 2010, pp. 478–489.

[3] C. C. Aggarwal, "On classification of graph streams," in *Proc. SIAM Int. Conf. Data Mining*, 2011, pp. 652–663.
[4] J. Wu, S. Pan, X. Zhu, and Z. Cai, "Boosting for multi-graph classification," *IEEE Trans. Cybern.*, vol. 45, no. 3, pp. 430–443, Mar. 2015.
[5] S. Pan, J. Wu, X. Zhu, and C. Zhang, "Graph ensemble boosting for imbalanced noisy graph stream classification," *IEEE Trans. Cybern.*, vol. 45, no. 5, pp. 940–954, May 2015.
[6] Y.-M. Zhang, K. Huang, X. Hou, and C.-L. Liu, "Learning locality preserving graph from data," *IEEE Trans. Cybern.*, vol. 44, no. 11, pp. 2088–2098, Nov. 2014.
[7] K. Riesen and H. Bunke, "Graph classification by means of Lipschitz embedding," *IEEE Trans. Syst. Man Cybern. Part B: Cybern.*, vol. 39, no. 6, pp. 1472–1483, Dec. 2009.
[8] S. Pan, J. Wu, and X. Zhu, "CogBoost: Boosting for fast cost-sensitive graph classification," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 11, pp. 2933–2946, Nov. 2015.
[9] S. Pan, J. Wu, X. Zhu, G. Long, and C. Zhang, "Task sensitive feature exploration and learning for multitask graph classification," *IEEE Trans. Cybern.*, vol. 47, no. 3, pp. 744–758, Mar. 2017.
[10] V. Vapnik, *Statistical Learning Theory*. Hoboken, NJ, USA: Wiley, 1998.
[11] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Proc. Int. Conf. Mach. Learn.*, 2003, pp. 321–328.
[12] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Proc. Ann. Conf. Learning Theory*, 2003, pp. 129–143.
[13] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Proc. IEEE Int. Conf. Data Mining*, 2005, pp. 74–81.
[14] P. Mahé and J.-P. Vert, "Graph Kernels based on Tree Patterns for Molecules," *Mach. Learn.*, vol. 75, no. 1, pp. 3–35, 2009.
[15] N. Shervashidze and K. Borgwardt, "Fast subtree kernels on graphs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2009, pp. 1660–1668.
[16] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph Kernels," *J. Mach. Learning Res.*, vol. 11, pp. 1201–1242, 2010.
[17] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learning Res.*, vol. 12, pp. 2539–2561, 2011.
[18] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan, "Hash kernels for structured data," *J. Mach. Learning Res.*, vol. 10, pp. 2615–2637, 2009.
[19] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proc. Int. Conf. Mach. Learn.*, 2009, pp. 1113–1120.
[20] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithm*, vol. 55, pp. 58–75, 2005.
[21] Z. Jin, C. Li, Y. Lin, and D. Cai, "Density sensitive hashing," *IEEE Trans. Cybern.*, vol. 44, no. 8, pp. 1362–1371, Aug. 2014.
[22] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He, "Fast and accurate hashing via iterative nearest neighbors expansion," *IEEE Trans. Cybern.*, vol. 44, no. 11, pp. 2167–2177, Nov. 2014.
[23] J. Song, Y. Yang, X. Li, Z. Huang, and Y. Yang, "Robust hashing with local models for approximate similarity search," *IEEE Trans. Cybern.*, vol. 44, no. 7, pp. 1225–1236, Jul. 2014.
[24] W. Wu, B. Li, L. Chen, and C. Zhang, "Cross-view feature hashing for image retrieval," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining*, 2016, pp. 203–214.
[25] W. Wu, B. Li, L. Chen, and C. Zhang, "Canonical consistent weighted sampling for real-value weighted min-hash," in *Proc. IEEE Int. Conf. Data Mining series*, 2016, pp. 1287–1292.
[26] W. Wu, B. Li, L. Chen, and C. Zhang, "Consistent weighted sampling made more practical," in *Proc. Int. World Wide Web Conf.*, 2017, pp. 1035–1043.
[27] P. Li, "0-bit consistent weighted sampling," in *Proc. Annu. ACM SIGKDD Conf.*, 2015, pp. 665–674.
[28] P. Li and C. König, "*b*-Bit minwise hashing," in *Proc. Int. World Wide Web Conf.*, 2010, pp. 671–680.
[29] A. Shrivastava and P. Li, "Densifying one permutation hashing via rotation for fast near neighbor search," in *Proc. Int. Conf. Mach. Learning*, 2014, pp. 557–565.
[30] P. Li, A. Owen, and C.-H. Zhang, "One permutation hashing," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2012, pp. 3113–3121.
[31] A. Shrivastava and P. Li, "In defense of minhash over SimHash," in *Proc. Int. Conf. Artif. Intell. Statistics*, 2014, pp. 886–894.
[32] P. Li, A. Shrivastava, J. L. Moore, and A. C. König, "Hashing algorithms for large-scale learning," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2011, pp. 2672–2680.
[33] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, "HistoSketch: Fast similarity-preserving sketching of streaming histograms with concept drift," in *Proc. IEEE Int. Conf. Data Mining*, 2017, to be published.
[34] D. Yang, B. Li, and P. Cudré-Mauroux, "POIsketch: Semantic place labeling over user activity streams," in *Proc. Int. Joint Conf. Artif. Intell.*, 2016, pp. 2697–2703.
[35] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 671–687, 2003.
[36] P. Indyk, "Stable distributions, pseudorandom generators, embeddings and data stream computation," *J. ACM*, vol. 53, no. 3, pp. 307–323, 2006.
[37] C. C. Aggarwal, "On classification of graph streams," in *Proc. SIAM Int. Conf. Data Mining*, 2011, pp. 652–663.
[38] B. Li, X. Zhu, L. Chi, and C. Zhang, "Nested subtree hash kernels for large-scale graph classification over streams," in *Proc. IEEE Int. Conf. Data Mining*, 2012, pp. 399–408.
[39] L. Chi, B. Li, and X. Zhu, "Fast graph stream classification using discriminative clique hashing," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining* , 2013, pp. 225–236.
[40] B. J. Weisfeiler and A. A. Leman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.
[41] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," in *Proc. Annu. ACM Symp. Theory Comput.*, 1998, pp. 327–336.
[42] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi, "Graph kernels for chemical informatics," *Neural Netw.*, vol. 18, no. 8, pp. 1093–1110, 2005.
[43] S. Pan, X. Zhu, C. Zhang, and S. Y. Philip, "Graph stream classification using labeled and unlabeled graphs," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 398–409.
[44] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 721–724.
[45] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, pp. 27:1–27:27, 2011.
[46] D. Haussler, "Convolution kernels on discrete structures," UC Santa Cruz, Santa Cruz, CA, USA, Tech. Rep. UCSC-CRL-99–10, 1999.
[47] S. Hido and H. Kashima, "A linear-time graph kernel," in *Proc. IEEE Int. Conf. Data Mining*, 2009, pp. 179–188.
[48] W. Fan, et al., "Direct mining of discriminative and essential frequent patterns via model-based search tree," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 230–238.
[49] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *CoRR*, abs/1408.2927, 2014.
[50] C. H. C. Teixiera, A. Silva, and W. M. Jr, "Min-hash fingerprints for graph kernels: A trade-off among accuracy, efficiency, and compression," *J. Inform. Data Manag.*, vol. 3, no. 3, pp. 227–242, 2012.
[51] S. Gollapudi and R. Panigrahy, "The power of two min-hashes for similarity search among heirarchical data objects," in *Proc. Symp. Principles Database Syst.*, 2008, pp. 211–219.
[52] S. Tatikonda and S. Parthasarathy, "Hashing tree-structured data: Methods and applications," in *Proc. IEEE Int. Conf. Data Eng.*, 2010, pp. 429–440.



Wei Wu received the MSc degree in computer science from Peking University, Beijing, China, in 2014. He is currently working toward the PhD degree in the Centre for Artificial Intelligence, University of Technology Sydney, Australia. His research interests are randomized hashing algorithms, data mining, and local search, and his papers appear in major conferences including WWW, ICDM, and PAKDD.

Bin Li received the PhD degree in computer science from Fudan University, Shanghai, China. He was a senior research scientist with Data61 (formerly NICTA), CSIRO, Eveleigh, NSW, Australia, and a lecturer with the University of Technology Sydney, Broadway, NSW, Australia. He is currently an associate professor in the School of Computer Science, Fudan University, Shanghai, China. His current research interests include machine learning and data analytics, particularly in complex data representation, modeling, and prediction.



Ling Chen received the PhD degree from Nanyang Technological University, Singapore. She is a senior lecturer in the Centre for Artificial Intelligence, University of Technology Sydney (UTS). Before joining UTS, she was a post-doctoral research fellow in the L3S Research Center, University of Hannover, Germany. Her research interests include data mining and machine learning, social network analysis, and recommender systems. Her papers appear in major conferences and journals including SIGKDD, ICDM, SDM, and ACM TOIS.

Xingquan Zhu (SM'12) received the PhD degree in computer science from Fudan University, Shanghai, China. He is an associate professor in the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, Florida, and a distinguished visiting professor (Eastern Scholar) with the Shanghai Institutions of Higher Learning. His current research interests include data mining, machine learning, and bioinformatics. Since 2000, he has published more than 220 refereed journal and conference papers in these areas, including two Best Paper Awards and one Best Student Paper Award. He is an associate editor of the *IEEE Transactions on Knowledge and Data Engineering* (2014-current), and an associate editor of the *ACM Transactions on Knowledge Discovery from Data* (2017 - current). He is serving on the editorial board of the *Journal of Big Data* (2014-current), the *International Journal of Social Network Analysis and Mining* (2010-current) and the *Network Modeling Analysis in Health Informatics and Bioinformatics Journal* (2014-current). He was the program committee cochair for the 14th IEEE International Conference on Bioinformatics and BioEngineering (BIBE-2014), IEEE International Conference on Granular Computing (GRC-2013), 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI-2011), and the 9th International Conference on Machine Learning and Applications (ICMLA-2010). He also served as a conference co-chair for ICMLA-2012. He also serves (or served) as a program vice chair, finance chair, publicity co-chair, and program committee member for many international conferences, including ACM-KDD, IEEE-ICDM, and ACM-CIKM. He is a senior member of the IEEE.

Chengqi Zhang (SM'95) received the PhD degree from the University of Queensland, Brisbane, Australia, in 1991 and the DSc degree (higher doctorate) from Deakin University, Geelong, Australia, in 2002. Since December 2001, he has been a professor of information technology with the University of Technology, Sydney, Australia, and has been the executive director of UTS Data Science since September 2016. Since November 2005, he has been the chairman of the Australian Computer Society National Committee for Artificial Intelligence. He has published more than 200 research papers, including several in first-class international journals, such as *Artificial Intelligence*, and *IEEE* and *ACM Transactions*. He has published six monographs and edited 16 books, and has attracted 11 Australian Research Council grants. His research interests mainly focus on data mining and its applications. He has served as an associate editor for three international journals, including the *IEEE Transactions on Knowledge and Data Engineering* (2005-2008); and as general chair, PC chair, or organizing chair for five international conferences including ICDM 2010 and WI/IAT 2008. He was/is general co-chair of KDD 2015 in Sydney, the local arrangements chair of IJCAI-2017 in Melbourne, a fellow of the Australian Computer Society, and a senior member of the IEEE.