# Scalable Classification for Large Dynamic Networks

Yibo Yao and Lawrence B. Holder
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164
Email: {yibo.yao, holder}@wsu.edu

***Abstract*—We examine the problem of node classification in large-scale and dynamically changing graphs. An entropy-based subgraph extraction method has been developed for extracting subgraphs surrounding the nodes to be classified. We introduce an online version of an existing graph kernel to incrementally compute the kernel matrix for a unbounded stream of these extracted subgraphs. After obtaining the kernel values, we adopt a kernel perceptron to learn a discriminative classifier and predict the class labels of the target nodes with their corresponding subgraphs. We demonstrate the advantages of our learning techniques by conducting empirical evaluations on two real-world graph datasets.**

## I. Introduction

Networks are often used as a general framework to describe the relations between entities. Many real-world applications produce networked data. Typical examples include traffic flows in IP networks, who-calls-who communication in telephone networks, and friendships in social networks. A research focus in the data mining community has been to extend traditional learning techniques to perform supervised classification tasks on these graph datasets. The problem of classification in a large-scale graph usually involves classifying nodes [1] or subgraphs [2] into suitable categories. Graph kernels [3] are a set of popular methods for classifying graph data. They quantify the similarity between graphs by implicitly mapping them into a high-dimensional space and then computing their inner products. Conventionally, graph kernels assume that the whole graph data fits in memory, which allows them to scan the data multiple times within reasonable time constraints. However, this assumption has been violated since the recent growth in the sizes of real-world graphs. These graph datasets are now measured in terabytes and heading toward petabytes. Furthermore, real-world graphs are usually generated in a streaming fashion with frequent updates (e.g., insertions/deletions of nodes/edges) to the underlying graph structures. For example, social networks (e.g., Facebook, Twitter) are continuously formed by the increasing volume of interactions among entities.

**Motivation:** Due to the dynamic nature of those networks, classical graph kernels are incapable of calculating similarities effectively for the following reasons.

- It is impossible to hold all information of the underlying network structure in memory with the increasing volume of graph data.
- The computational load of enumerating substructures (e.g., random walks, subtrees) for graph kernels and computing their kernel matrices will become extremely expensive or even impractical.
- Noisy structural information may deteriorate the classification performance or introduce unexpected structural complexity during the learning process.

**Contribution:** In order to address the above challenges, we report in this paper an online classification framework to investigate the problem of node classification in large-scale and dynamically changing graphs. The framework consists of three parts: a subgraph extraction method, an online kernel computation algorithm, and an online kernel perceptron.

1) We design an *entropy-based scheme* (called **SubExtract**) to extract a subgraph surrounding a target node, in which the informative neighbor nodes are included through discriminative links and the irrelevant ones are filtered out.

2) We propose an *online version* (known as **OWL**) of an existing fast graph kernel, namely Weisfeiler-Lehman (WL) [4], to enable the kernel computation to be performed in an online fashion rather than the traditional batch mode.

3) We propose to use a *kernel perceptron* (called **WLPer**) which takes the similarity values between graphs as inputs and predicts the class memberships. To the best of our knowledge, this is the first work to combine a graph kernel with a perceptron to classify nodes in graphs.

The rest of this paper is organized as follows: Section II reviews the related work. Some notations and the problem of interest are defined in Section III. In Section IV, the proposed framework including the detailed methods is described. Experimental results are shown in Section V. Some conclusions are drawn in Section VI.

## II. Related Work

The problem of graph classification has often been studied in the context of static graph data. There are a large number of effective graph kernels that have been developed for classifying graphs. Most of them follow the same principle of enumerating common substructures to quantify the similarity between graphs. Some typical substructures that have been used to build graph kernels include random walks [5], [6], shortest paths [7], subtrees [8], and graphlets [9]. In [5], [6], the authors develop random walk-based graph kernels, which count the number of common labeled walks existing in two graphs. A path-based kernel is introduced in [7], which computes the length of shortest-paths between all pairs of nodes and then counts pairs with similar attributes and lengths. A drawback of these walk- or path-based kernels is that the structures chosen to express the similarities among graphs are too simple. Another direction in developing an efficient graph kernel is to employ graphlets (small subgraphs with order $k \in \{3, 4, 5\}$) to characterize graphs when computing their similarities [9]. The recently proposed Weisfeiler-Lehman (WL) kernel [4], [8] is a fast subtree kernel whose runtime scales linearly in the number of edges of the graphs. It aims to count the matching multiset labels of the entire neighborhood of each node up to a given distance $h$. However, most of these methods are required to be applied in batch mode (i.e., all data need to be available for training), which limits their applicability on data streams.

Aggarwal proposed a hash-based probabilistic approach for finding discriminative subgraphs to facilitate the classification on massive graph streams [10]. A 2-dimensional hashing scheme has been designed for constructing an in-memory summary of the structural statistics of the underlying graph stream. The hashing techniques are used to compress the continuously incoming edge streams and explore the relation between edge pattern co-occurrence and class distributions. Hashing techniques have also been used in [11], [12] to classify graph streams by detecting discriminative cliques and mapping them onto a fixed-size common feature space. Chi et al. [11] designed a fast rule-based classification algorithm by compressing the unlimited expanding edge space onto a fixed-size one and detecting discriminative clique features from the compressed graphs. Similar ideas have been applied in [12]. The authors utilize graph factorization for assigning numerical weights to the discovered cliques, and each graph thus can be represented by a fine-grained clique feature vector. Li et al. [13] propose a Nested Subtree Hash (NSH) kernel algorithm to project different subtree patterns from graph streams onto a set of common low-dimensional feature spaces, and construct an ensemble of NSH kernels for large-scale graph classification over streams. Yao and Holder [14] use an incremental support vector machine (SVM) with the combination of the WL graph kernel to classify the extracted subgraphs from a single large-scale dynamic graph. The idea is to retain the support vectors (SVs) from the SVM trained based on graphs seen so far and incorporate them into the incoming batch to form a new training set. In case that the amount of SVs may grow unconstrained, they suggest to employ a sliding window on the graph streams to maintain a bounded number of graphs for training. The framework presented in this paper is similar to the work in [14], but we adopt a kernel perceptron to perform online classification instead of SVM. In addition, we propose an online version of the WL kernel to compute kernel values incrementally rather than compute them in a batch mode, and present new experimental results.

## III. Background

In this section, we formally describe the problem of interest and then present a brief review of the WL kernel and the kernel perceptron. We focus on the dynamic graphs which are subject to incremental changes, i.e., continuous insertions of new nodes and edges. Other types of changes such as deletions/modifications of nodes/edges are beyond the scope of this paper.

### A. Terminologies and Notations

**Definition 1.** *A **graph** is denoted by a 3-tuple $G = (V, E, l)$, where $V = \{v_1, \ldots, v_{|V|}\}$ is a set of nodes, $E \subseteq V \times V$ is a set of directed/undirected edges, and $l : V \cup E \to \mathcal{L}$ is a function assigning labels from an alphabet $\mathcal{L}$ to the nodes or edges in $G$.*

**Definition 2.** *Let $G = (V, E, l)$ and $G' = (V', E', l')$ denote two graphs. $G$ is said to be a **subgraph** of $G'$, i.e., $G \subseteq G'$, if and only if: (1)$V \subseteq V'$, (2)$\forall v \in V, l(v) = l'(v)$, (3)$E \subseteq E'$, (4)$\forall (u, v) \in E, l(u, v) = l'(u, v)$.*

We particularly focus on a specified kind of subgraphs, namely subtrees, which are the core parts for the WL kernel. The subtrees have successfully shown discriminative power when combined with kernel methods (e.g., SVM) in classifying graph data from various domains [4], [8], [13].

**Definition 3.** *A **subtree** is a subgraph of a graph, with a designated root node but no cycles. The **height** of a subtree is the maximum distance between the root and any leaf node in the subtree.*

**Definition 4.** *Let $G$ be a graph and $v \in V$ be a node. The **neighborhood** of $v$, denoted by $\mathcal{N}(v)$, is the set of nodes to which $v$ is connected by an edge in $E$, i.e., $\mathcal{N}(v) = \{u|(u, v) \in E\}$.*

**Definition 5.** *A **dynamic graph**, denoted by $\mathcal{G}$, is defined over a sequence of **updates** $\{\cdots, U_i, U_{i+1}, \cdots\}$, where each $U_i$ contains an operation of inserting a new node or a new edge into the graph in a streaming fashion.*

Generally, $\mathcal{G}^t = \{\bigcup_{i=1}^{t} U_i\}$ denotes the graph at time $t$, and $\mathcal{G}^0 = \emptyset$ is the initial graph without any nodes or edges. We assume that each $U_i$ can be denoted in the form of an edge, i.e., $e_i = \langle v_{i1}, v_{i2}, l_i \rangle$, where $v_{i1}, v_{i2}, l_i$ denote the two endpoints and the label of $e_i$. If the underlying dynamic graph is a directed graph, then $e_i$ has a direction pointing from $v_{i1}$ to $v_{i2}$. We assume that the updates are received in the form of batches, then a dynamic graph can be denoted as a collection of batches $\mathcal{G} = \{B_1, \cdots, B_t, \cdots\}$ where $B_t$ denote a certain number of updates to be applied on the object network at time $t$. See Fig. 1 for an example.

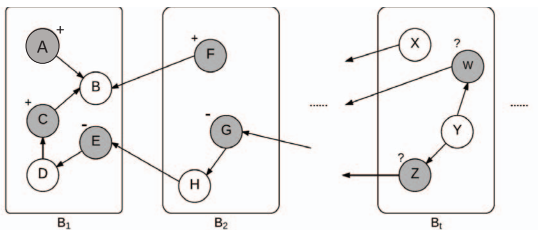


Fig. 1. Central node classification. The shaded nodes denote the central nodes, and the symbols $\pm 1$ indicate their class labels. $B_t$ denotes the batch of updates received at time $t$.

It is common that graphs from real-world applications consist of different types of nodes and relationships. Users are often interested in categorizing a certain type of node (e.g., paper nodes in an author-paper network) with the help of other types of nodes (e.g., author nodes in a paper-author network). In this paper, we focus on classifying a particular type of node in a large dynamic network which contains different types of nodes.

**Definition 6.** *A node to be classified is defined as a **central node** which denotes a central entity from the original data,* *and a node is defined as a* ***side node*** *if it is not a central one.*

Fig. 2 shows one instance of a citation network in which all papers $P_i$ are marked as central nodes while authors $A_i$ are side nodes. Such a network would support the classification of papers, e.g., papers that will receive a high number of citations versus papers that will not.

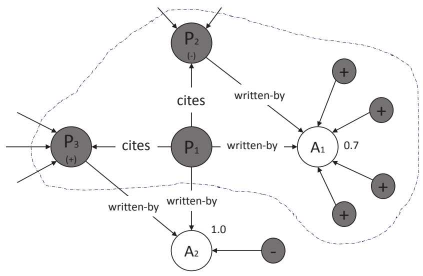


Fig. 2. An instance of a citation network. A shaded node $P_i$ represents a paper while a white node $A_j$ represents an author associated to a paper. An edge label represents the relationship between the two nodes it connects.

Given the aforementioned setting, we now formulate the problem that we aim to tackle in this paper as follows: Given a dynamic graph with central and side nodes indicated in its representation, and each central node $v_i$ is associated with a class membership $y_i \in \{+1, -1\}$, learn a classifier using the available information up until time $t$, and predict the class labels of any new central nodes arriving at time $t+1$. Fig. 1 illustrates a toy example.

### B. Weisfeiler-Lehman Kernel

The Weisfeiler-Lehman (WL) graph kernel [4] is a state-of-the-art graph kernel whose runtime scales linearly in the number of edges of the graphs. It is based on the one-dimensional variant of the Weisfeiler-Lehman isomorphism test [15]. In this algorithm, each node's label is augmented by a sorted set of node labels of the neighboring nodes, and the augmented labels are compressed into new labels using mapping functions. These steps are then repeated for $R$ iterations. Let $G^{(i)}$ denote the augmented graph at the $i$th iteration ($G^{(0)}$ is the initial graph with original node labels), then the WL kernel with $R$ iterations between two graphs can be calculated as

$$K^{(R)}(G, G') = \sum_{i=0}^{i=R} \kappa(G^{(i)}, G'^{(i)})$$

where $\kappa(G^{(i)}, G'^{(i)}) = \langle \phi(G^{(i)}), \phi(G'^{(i)}) \rangle$ and $\phi(G)$ is a numeric vector whose elements record the numbers of occurrences of node labels in $G$.

However, the WL kernel based on batch mode will suffer from memory and time issues when applied to dynamic graphs since multiple scans and holding all data in memory are not realistic in this situation. The kernel matrix has to be recomputed from scratch when a batch of new data is streamed in and needs to be classified. As a result, the running time grows dramatically. In order to enable the kernel computation for graph streams to proceed in a faster way, we propose an online version of the WL kernel based on a variant of the WL kernel developed in [13].

### C. Online Kernel Perceptron

The online perceptron and its variants have been increasingly applied to classifying data streams because of their outstanding performance on many real-world tasks [16]–[19]. One appealing advantage of the online perceptron is its ability to process open-ended data streams. Given a data stream $\mathcal{D} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_i, y_i), \cdots\}$, where $\mathbf{x}_i \in \mathcal{X}$ is a feature vector of the $i$th data point and $y_i \in \{\pm 1\}$ is the corresponding class label, the perceptron incrementally construct a discriminative classification function $f : \mathcal{X} \to \mathbb{R}$ by processing the data points in an online mode. The perceptron maintains a subset of instances which are misclassified data points. This subset is called a **budget** or **support set**, and the instances in this subset are called **support vectors**.

At the $i$th round, the perceptron receives $\mathbf{x}_i$ and predicts its class membership using $\hat{y}_i = \text{sgn}(f_{i-1}(\mathbf{x}_i))$, where $f_{i-1}$ is the hypothesis learned from the previous round and $f_0 = 0$. Once the true label $y_i$ is received, the perceptron checks it against the predicted label. If the prediction is incorrect, $\mathbf{x}_i$ is added to the budget and used to update the hypothesis $f_i$. Otherwise, the hypothesis and the budget remain unchanged. When the budget reaches its maximum size (usually defined by a user), old instances need to be deleted from the budget in order to maintain a bounded memory storage. Various discarding techniques [18]–[22] have been proposed to address this issue. In this paper, we have adopted a simple strategy which removes the oldest instance from the budget [19] to keep the number of support vectors bounded.

## IV. Framework

In this section, we introduce the three key components included in our framework.

1) **Subgraph extraction (SubExtract):** extracting a subgraph $G_i$ for a central node $v_i$ in a large dynamic network.
2) **Online kernel computation (OWL):** converting $G_i$ into a feature vector, and computing the similarity values between $G_i$ and all the retained old subgraphs.
3) **Class membership prediction (WLPer):** predicting the class label of $G_i$ using the kernel perceptron. $G_i$ is added into the budget to update the model if the predicted label differs from the true label of $G_i$.

### A. Subgraph Extraction

There are many approaches for extracting a subgraph surrounding a node, for example, 1-edge hops, random walks. However, a star-like subgraph extracted using 1-edge hops may not be discriminative enough since it contains less structural information. On the other hand, a subgraph extracted by random walks may include too much irrelevant information for classification, thus reducing the classification performance.

In order to classify a central node in a dynamic graph, we design an effective strategy to extract a subgraph for it by selecting the informative neighboring nodes and discarding those with less discriminative power. For a node $v_i \in \mathcal{G}$ ($v_i$ can be a central or side node), if it is connected to any central nodes, then we can define the entropy value for $v_i$. Let $n_{pos}$ and $n_{neg}$ denote the numbers of central nodes with positive class labels and negative class labels in $\mathcal{N}(v_i)$, respectively. The probabilities of positive and negative instances in $\mathcal{N}(v_i)$ can then be estimated by: $p_1 = \frac{n_{pos}}{n_{pos}+n_{neg}}$ and $p_2 = \frac{n_{neg}}{n_{pos}+n_{neg}}$. Thus the entropy for $v_i$ can be written as:

$$EN(v_i) = -p_1\log_2 p_1 - p_2\log_2 p_2$$

The entropy of $v_i$ expresses the discriminative power of $v_i$ with respect to the two classes (positive and negative) during the classification process. Lower values for $EN(v_i)$ means $v_i$ has more discriminative power.

To obtain a subgraph for a target central node to be classified, denoted as $v_c$, an entropy threshold parameter $\theta$ needs to be set for selecting the discriminative neighbor nodes. Moreover, we assume that each side node must be connected only to central nodes in our graph representation. In other words, we do not allow interconnections between any two side nodes in the representation. In most domains, this constraint does not impose undue limitations on the representation of information, because most side nodes represent attributes of the central node, and most relationships of interest in the graph are between central nodes. The main idea of our extraction method is that we start from $v_c$ and keep extracting neighbor nodes with entropy $\leq \theta$ until we meet other central nodes of the same type as $v_c$. We then extract the subgraph induced from the interconnections between the extracted nodes. Algorithm 1 shows the detailed procedure for extracting a subgraph surrounding a central node from a graph. Fig. 2 illustrates a paper subgraph extracted from a large citation network. Besides the explicitly labeled paper nodes (i.e., $P_i$s), the authors also have many other papers connected to them. The $\pm$ symbols imply the class memberships of the papers. The entropy values for the side nodes are indicated using real numbers. If we set the entropy threshold $\theta \leq 0.8$, then the subgraph surrounded by the dash line is the one that will be extracted for the paper node $P_1$.

**Algorithm 1** Subgraph Extraction (SubExtract)

```
Input:
    $\mathcal{G}$: A graph
    $v_c$: A target central node
    $\theta$: A threshold for selecting discriminative nodes
Output:
    $Subg_{v_c}$: A subgraph surrounding $v_c$

1:  $I(v_c) = \{v_c\}$
2:  $N_v = \mathcal{N}(v_c)$
3:  while $N_v \neq \emptyset$ do
4:      pop a node $v'$ from $N_v$
5:      if $v'$ is not visited then
6:          compute the entropy $EN(v')$
7:          if $EN(v') \leq \theta$ then
8:              $I(v_c) = I(v_c) \cup \{v'\}$
9:              mark $v'$ as visited in $\mathcal{G}$
10:             if $v'$ is not the same type as $v_c$ then
11:                 $N_v = N_v \cup \mathcal{N}(v')$
12:             end if
13:         end if
14:     end if
15: end while
16: induce a subgraph $Subg_{v_c}$ from $\mathcal{G}$ with the nodes in $I(v_c)$
17: return $Subg_{v_c}$
```

The proposed extraction scheme will only extract, at most, all the nodes connected to $v_c$ and all the central nodes connected to the side nodes of $v_c$. When computing entropy values for each node in $\mathcal{N}(v_c)$, the class label of $v_c$ should not be counted because $v_c$ is the target node to be classified. The subgraph extraction scheme aims to serve two purposes: (1) select informative neighbor nodes for classification, and (2) reduce the structural complexity of the subgraph to facilitate the efficient computation of the kernel values. When dealing with a dynamic graph, the subgraph extraction becomes complicated. In order to control the size of the dynamic graph stored in memory, we use a fixed-sized budget to retain the most recent data batches and discard the oldest ones (see Section IV-C). There is a possibility that a certain update in the batch $B_t$ may refer to older nodes which have been deleted from memory because they are outside the budget. During the extraction, we allow these older nodes to be re-inserted into the graph when new incoming edges refer to them. And they can be included in the extracted subgraphs if they are discriminative enough.

## *B. Online Computation of WL Kernel*

The WL kernel counts common subtree patterns in two graphs. The subtree discovery process follows the WL isomorphism test [15] and iterates for $R$ rounds. Assume that $s_i(v)$ denotes the augmented label of a node $v$ at the $i$th iteration. Initially, the multiset-label assigned to $v$ is its original label, i.e., $s_0(v) = l(v)$. At the $i$th iteration when $i > 0$, the multiset-label for $v$ is determined by the subtree rooted at $v$. Let $M_i(v) = \{s_{i-1}(u) | u \in \mathcal{N}(v)\}$ denote the list of node labels of $v$'s neighborhood. The elements in $M_i(v)$ are sorted in ascending order and concatenated to form a string representation $str(M_i(v))$. Then the multiset-label of $v$ is formed by adding $l_{i-1}(v)$ as a prefix of $str(M_i(v))$, that is, $s_i(v) = strcat(l_{i-1}(v), str(M_i(v)))$. For simplicity, at each iteration, after obtaining the string representations of the multiset-label sets for all nodes, those strings are mapped to unique integers, i.e., $\forall v \in V, l_i(v) = \lambda(s_i(v))$ where $\lambda : \mathcal{L} \to \mathbb{N}$ is a mapping function. Since $s_i(v)$ encodes the subtree pattern rooted at $v$ with height $i$, if two subtree patterns are identical, they will be mapped to the same integer value.

Let $G^i$ denote the augmented graph at the $i$th iteration ($G^0$ is the initial graph with original node labels), and $\mathcal{P}_i = \{p_{i_1}, \cdots, p_{i_n}\}$ denote the set of distinct subtree patterns discovered from the two graphs at the $i$th iteration. We use a function $\phi_i(\cdot)$ to map $G^i$ onto the subtree pattern vector $\mathcal{P}_i$. $\phi_i(G^i)$ actually counts the occurrences of every subtree pattern in $G^i$, i.e., $\phi_i(G^i) = [\#(p_{i_1}), \cdots, \#(p_{i_n})]$. Then the WL kernel value between $G_1$ and $G_2$ after $R$ iterations can be formulated as

$$K^{(R)}(G_1, G_2) = \sum_{i=1}^{R} \langle \phi_i(G_1^i), \phi_i(G_2^i) \rangle$$

where $\langle \cdot, \cdot \rangle$ denotes an inner product between two vectors.

Almost all graph kernel methods require the set of graphs to be available for pre-scan in order to find the subgraph patterns (subtrees, shortest-paths, random walks, etc.) to form a feature space. However, they fail to scale efficiently on data streams with unlimited number of graphs due to the following drawbacks [13]: 1) the feature space keeps expanding with emerging subgraph patterns when new graphs are continuously fed in; 2) the memory and runtime to find the desired subgraph patterns become prohibitive as the size of graph set grows; 3) multiple scans are required in order to calculate the global kernel matrix for all graphs seen so far.

The Nested Subtree Hashing (NSH) kernel, a variant of the WL kernel, was proposed by Li et al. [13] to address the above issues. At each iteration, they use a random hash function, $h : str \rightarrow \mathbb{N}$, to project the strings representing subtree patterns onto a set of common low-dimensional feature spaces. The function takes a string $str$ and an integer $d$ as inputs, and maps $str$ to a random integer which is no greater than $d$. They prove that their NSH kernel is an unbiased estimator of the original WL kernel. One parameter for the NSH kernel is the hashing dimension $D = \{d^0, \cdots, d^R\}$, where $d^i$ represents the size of the subtree feature space at the $i$th iteration. All subtree patterns encountered at the $i$th iteration will be mapped to integers no greater than $d^i$. The output of the NSH kernel is a multi-resolution feature vector set $\{\mathbf{x}^i\}_{i=0}^R$, where $\mathbf{x}^i$ is a $d^i$-length vector and records the occurrences of subtree patterns encountered at the $i$th WL test.

One merit of the NSH algorithm is that the size of the feature space at each iteration is bounded. In this way, we are able to manage the emerging subtree patterns at each iteration and map them onto a fixed-size space. We no longer need to find the global set of distinct subtree patterns from all the graphs at each iteration. Based on the NSH kernel, we propose an online version of the WL kernel, namely OWL, which can be applied for calculating the similarity between graphs from a data stream in an online mode. The approach is that: once we receive a graph from the data stream, we call the NSH method to convert that graph into a set of feature vectors in which every feature vector records the numbers of occurrences of subtree patterns at the corresponding iteration. The similarity between graph $G_i$ and all the previous graphs $\{G_1, \cdots, G_{i-1}\}$ can then be calculated by the inner products between $G_i$'s feature vectors and $G_j$'s ($j < i$) feature vectors. Since every graph is characterized by a set of feature vectors, we can store those vectors in memory and discard the graph itself. This is time-efficient since there is no need to fetch the original graph again and scan it for multiple iterations to obtain the subtree patterns. And it is also memory-efficient by limiting the bucket size for hashing, thus to avoid unboundedly emerging subtree patterns. Algorithm 2 shows the pseudocode of OWL.

To compute the similarity between the current graph $G_t$ and all the graphs streamed in before $t$, OWL needs to perform $t-1$ inner products between $\mathbf{x}_t$ and all $\mathbf{x}_i$ $(i < t)$ stored in $\mathcal{F}$ (see line 6-9). Therefore, the time complexity of OWL is quadratic in the number of graphs. This will become infeasible when applied to computing the kernel values for an open-ended graph stream. In order to address this issue, we fix the size of $\mathcal{F}$ to store a limited number of instances in memory when incorporating OWL into a kernel perceptron (see Section IV-C for details).

Since the elements of $\mathbf{x}_t$ are non-negative integers whose values indicate the numbers of occurrences of the subtree patterns, the resulting similarity measure between two graphs $G_t$ and $G_n$ is also a non-negative integer. Usually, the diagonal elements in the resulting kernel matrix are not equal to 1. It is common to normalize the matrix in order to set all the similarity values in the range between 0 and 1. We use the following formula to normalize a kernel matrix

**Algorithm 2** Online WL (OWL)

```
Input:
    $\mathcal{D} = \{G_1, \cdots, G_j, \cdots\}$: a collection of graphs
    $R$: number of iterations for the WL test
    $D = \{d^0, \cdots, d^R\}$: the dimensions for hashing
Output:
    $K$: a kernel matrix

 1: Initialize $\mathcal{F} = \emptyset$
 2: for $t = 1, \cdots, j, \cdots$ do
 3:     $\{x^i\}_{i=1}^R = \text{NSH}(G_t, R, D)$
 4:     $\mathbf{x}_t = vec(\{x^i\}_{i=1}^R)$ [1]
 5:     $\mathcal{F} = \mathcal{F} \cup \{\mathbf{x}_t\}$
 6:     for $n = 1$ to $t-1$ do
 7:         $\mathbf{x}_n = \mathcal{F}(n)$
 8:         $K(G_t, G_n) = \langle \mathbf{x}_t, \mathbf{x}_n \rangle$
 9:     end for
10: end for
```

$$K_{norm}(G_t, G_n) = \frac{K(G_t, G_n)}{\sqrt{K(G_t, G_t) \cdot K(G_n, G_n)}}$$

## C. Online Budget Perceptron

Despite its simplicity, the perceptron has produced good results in many real-world applications. And it becomes especially effective when combined with kernels due to the benefits generated by kernel trick. A kernel-based perceptron can be written as a kernel expansion

$$f(\mathbf{x}) = \sum_{i \in S_t} y_i K(\mathbf{x}_i, \mathbf{x})$$

where $K(\cdot, \cdot)$ is the kernel value between two instances and $S_t$ is the support set at time $t$. The prediction can be made by $\hat{y}_t = \text{sgn}(f(\mathbf{x}_t))$. If the predicted result $\hat{y}_t$ is not equal to the true label $y_t$, $\mathbf{x}_t$ is added to the budget, that is, $S_t = \{i | \hat{y}_i \neq y_i, i < t\}$.

Although the online perceptron will potentially utilize less memory by only retaining the misclassified instances, it is still possible that the algorithm will not scale up effectively when, firstly, a large number of instances tend to be retained. Generally, the amount of memory required for storing the support set may grow unboundedly if the learning task is a hard one when almost all training instances have been misclassified. An extremely large support set can lead to significant computational difficulties for this online learning algorithm. Secondly, the target concept of the data stream may change over time so that the old support vectors may not be a promising predictor for future data. In order to maintain a bounded memory usage, we adopt a simple but effective way to

[1]We define $vec(\cdot)$ as the vectorized operator which concatenates all the row vectors or column vectors together to form a larger row vector or column vector. $vec(\{x^i\}_{i=1}^R) = [x^1, \ldots, x^R]$ forms a new row vector with length $d_1 + \cdots + d_R$, where $x^i$ is also a row vector and $|x^i| = d_i$.

discard old instances from the support set. When the budget is full, we remove the oldest instance [19] so that we can always maintain a set of instances which are closely related to the current instance in the time line, which makes the algorithm tolerant to potential concept drifts. At time $t$ when the new batch $B_t$ arrives, the oldest instance in the budget will be discarded if the budget is full. As a result, the corresponding nodes and their associated edges will also be deleted from the graph if they are not included in any existing instances in the budget. However, in our approach, we allow the nodes which have been deleted to be re-inserted into the graph if new edges in $B_t$ refer to them. Algorithm 3 lists the pseudocode of the perceptron (WLPer) method in combination with the online version of WL kernel.

**Algorithm 3** Online Perceptron with OWL (WLPer)

```
Input:
    $\mathcal{D} = \{G_1, \cdots, G_j, \cdots\}$: a collection of graphs
    $R$: number of iterations for the WL test
    $D = \{d^0, \cdots, d^R\}$: the dimensions for hashing
    $B$: budget size
Output:
    $\{\hat{y}_1, \cdots, \hat{y}_j, \cdots\}$: predicted class labels

1:  Initialize $S_0 = \emptyset$, $f_0 = 0$
2:  for $t = 1, \cdots, j, \cdots$ do
3:      $\{x\}_{i=1}^{R} = \text{NSH}(G_t, R, D)$
4:      $\mathbf{x}_t = vec(\{x\}_{i=1}^{R})$
5:      $\hat{y}_t = \text{sgn}(f_{t-1}(\mathbf{x}_t))$
6:      if $\hat{y}_t \neq y_t$ then
7:          if $|S_{t-1}| \geq B$ then
8:              delete the oldest element from $S_{t-1}$ and $f_{t-1}$
9:          end if
10:         $S_t = S_{t-1} \cup \{t\}$
11:         $f_t = \sum_{i \in S_t} y_i K(\mathbf{x_i}, \mathbf{x_t})$
12:     else
13:         $f_t = f_{t-1}$
14:         $S_t = S_{t-1}$
15:     end if
16: end for
```

An important parameter for this method is the size of the budget. WLPer with a larger budget size will generally produce higher accuracy but need more learning time. In practice, when the extracted subgraphs are dense, we should avoid setting $B$ arbitrarily large, in order to keep the accessibility and runtime at a tractable level. When the subgraphs are sparse, we should avoid using a small $B$ since a learner trained on fewer examples will produce higher generalization error.

### D. Complexity Analysis

For any central node, we assume the maximum number of central nodes and side nodes connected to it are $n_c$ and $n_s$ respectively, and the maximum degree of any side node is $d_s$. Therefore the maximum degree of a central node is bounded by $O(n_c+n_s)$. SubExtract takes time $O(n_c)$ and $O(n_s)$ to extract the central nodes and side nodes from the neighbor of a central node of interest. For each extracted side node, it additionally needs $O(d_s)$ to extract the surrounding nodes in order to reach the same type of nodes as the central nodes. As a result, the overall time complexity for SubExtract is $O(n_c + n_s d_s)$. The time complexity for the WL process for the $t$th graph $G_t = (V_t, E_t)$ in the graph set is $O(R|E_t|)$ [15], where $R$ is the number of WL iterations. The value of $R$ is usually chosen by using cross-validation on the training data [4]. But for practical problems, it is recommended to set $R < 10$ [4], [8], [13]. In our experiments[2] (Section V), we have set $R = 4$.

Since $G_t$ is converted into a feature vector $\mathbf{x}_t$ recording the occurrences of subtree patterns, the space complexity for storing that feature vector is $O(|D|)$ where $|D| = \sum_{i=0}^{R} d^i$ and $d^i$ denotes the hash dimension at the $i$th iteration. However, it is recommended to choose $d^i \gg |V_t|$ in order to avoid information loss caused by hashing collision. A further observation is that $\mathbf{x}_t$ has no more than $R|V_t|$ nonzero elements, because $G_t$ has at most $|V_t|$ distinct rooted subtree patterns at every WL iteration. Therefore, the space needed for storing a feature vector can be further reduced by using sparse representation of the vector in programming. During the learning process of the perceptron algorithm, we only need to store at most $B$ feature vectors representing the recent $B$ graphs retained in the budget. The total memory consumption is then bounded by $O(B|D|)$.

## V. Experimental Study

In this part, we present preliminary experimental results which demonstrate the merits of the proposed learning framework. In particular, we evaluate the classification performance of our algorithm by comparing it with two baseline methods in terms of efficiency, effectiveness, and sensitivity to the subgraph extraction threshold $\theta$. Moreover, we report some results of our framework using different parameter settings (i.e., hashing dimensions, budget size).

### A. Benchmark Data

**DBLP stream**[3]: DBLP is a database containing millions of publications in computer science. Each paper is associated with abstract, authors, year, venue, title and references. Our classification task is to predict which of the following two topics a paper belongs to: DBDM (database and data mining: published in conferences VLDB, SIGMOD, PODS, ICDE, EDBT, SIGKDD, ICDM, DASFAA, SSDBM, CIKM, PAKDD, PKDD, SDM and DEXA) and CVPR (computer vision and pattern recognition: published in conferences CVPR, ICCV, ICIP, ICPR, ECCV, ICME and ACM-MM). We have identified 45,270 papers published between 2000 and 2009, and their references and authors. 19,680 of them are considered as positive examples (DBDM-related) while 25,590 of them as negative ones (CVPR-related). The dynamic DBLP network is then formed by chronological insertions of papers according to their publication dates, and their authors. In particular, we denote that (1) each paper ID is a central node while each author ID is a side node; (2) if a paper P1 cites another paper P2, there is a directed edge labeled with *cites* from P1 to P2; (3) if a paper P1's author is A1, there is a directed edge labeled with *written-by* from P1 to A1. After inserting a paper

[2]We conducted experiments with $R \in \{1, 2, \cdots, 9\}$, and found that the classification effectiveness of $R = 4$ was much better than that of $R = 1, 2, 3$. The effectiveness of $R = 5, \ldots, 9$ was slightly better than that of $R = 4$ with much lower efficiency.

[3]http://arnetminer.org/citation

and its associated entities and relations, we use SubExtract to extract a subgraph for this paper. As a result, we have 45,270 subgraphs extracted for all the papers. Fig. 2 shows an example of an extracted subgraph. To test the scalability of the proposed classification method, we duplicate every extracted subgraph for nine times to simulate that ten identical papers are published with the same authors and citations[4]. The final dataset contains 452,700 paper subgraphs with about $1.0 \times 10^7$ nodes and $1.3 \times 10^7$ edges in total.

**IBM SENSOR stream**[5]: This dataset records information about local traffic from a large sensor network which contains over 300 different intrusion types. The IP-addresses are denoted by nodes and the local traffic flows between IP-addresses are denoted by edges in the network. Each local traffic pattern corresponds to a subgraph in the large network and is associated with a particular intrusion type. For each pattern, we introduce an additional node showing the unique ID of the pattern and connect it to each IP node in this pattern. We have retrieved 65,463 positive traffic patterns with intrusion *'HTML NullChar Evasion'* and 50,875 negative traffic patterns with intrusion *'BWEST Login Successful'*. The dynamic Sensor network is formed by inserting the IP and ID nodes, and their relations in these traffic patterns chronologically. In this case, the ID nodes are central nodes while the IP nodes are side nodes. Our task is to classify the ID nodes into one of the two categories. We call SubExtract for extracting a subgraph for every ID node. Fig. 3 shows an example of an extracted subgraph. We duplicate every extracted subgraph for nine times. As a result, we have obtained $1,163,380$ subgraphs with $2.4 \times 10^6$ nodes and $3.5 \times 10^6$ edges in total.

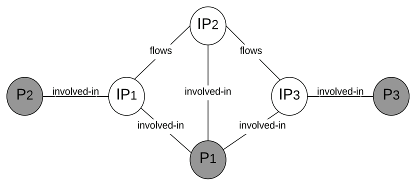


Fig. 3. A subgraph extracted from the SENSOR network. $P_i$ represents a traffic pattern ID while $IP_j$ represents an IP address. The edge between $P_i$ and $IP_j$ denotes that $IP_j$ is involved in the traffic pattern $P_i$. The edge between two IP nodes $IP_j$ and $IP_k$ indicates that there is a traffic flow from $IP_j$ to $IP_k$.

## B. Baseline Methods

**Window-based Incremental SVM (WinSVM)** [14]. This method incorporates the WL kernel into an incremental SVM learner. At each learning step when a SVM model is built, WinSVM will retain the set of support vectors of the model and discard all other instances from the training set. These support vectors are combined with the next available graph in order to form a new training set. In case that the number of support vectors grows indefinitely a sliding window strategy has been proposed to keep data stored in memory at a moderate size. In the experiments, we use LIBSVM [23] as the classifier.

[4]We realize that the duplication of data could decrease the diversity of a real-world dataset and affect the error rate since a significant part of the data has been seen by the algorithm at a prior time. However, the major concerns of this paper have focused on scaling the proposed techniques to big graphs.

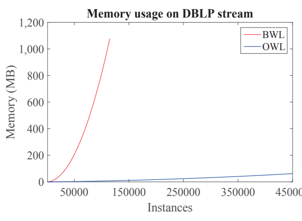
[5]http://charuaggarwal.net/sens1/gstream.txt

**DIscriminative Clique Hashing (DICH)** [11]. The algorithm uses a random hashing to compress the infinite edge space from a graph stream onto a fixed-size space, and employs a fast clique detection algorithm to detect frequent discriminative clique patterns. A rule-based classifier is constructed based on the set of discovered cliques.

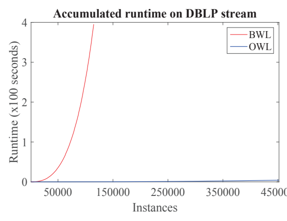
All experiments have been performed on a 3.40GHz Intel 64-bit machine with 8GB of memory. Since WLPer and the two baseline methods take a set of extracted subgraphs as their inputs, we test the variation in the quality of these methods by using different values for $\theta$. In addition, we also test the sensitivity of WLPer with the budget size $B$ and hashing dimensions $D$. Unless otherwise mentioned, the default values of the parameters are as follows: budget size $B = 1000$, hashing dimensions $D = \{500, 1000, 5000, 10000\}$, and subgraph extraction threshold $\theta = 1.0$. For fair comparison, in our experiments, we have set the window size of WinSVM equal to the budget size of WLPer.

## C. Evaluation of OWL

First, we empirically compare the runtime and memory usage of the OWL kernel to the traditional batch-mode WL (BWL) kernel. BWL requires that all graphs be available for computing a global kernel matrix. In a streaming scenario, each time a new graph is fed in, we compute a kernel matrix for all the available graphs using the batch-mode method. We set the maximum available memory as 1GB. If the memory usage reaches the maximum, then we stop the running process. The results of DBLP are shown in Fig. 4. We see that the memory usage and accumulated runtime for BWL grow exponentially as the number of graph instances increases. However, OWL scales linearly to the number of instances in terms of memory and runtime. We find that BWL can only process about $11,500$ graphs before running out of the maximum allowed memory. But OWL can process all the graphs using much less memory and time. Similar results can be observed from the SENSOR stream. Because of space limitations, we have not included the plots of memory usage and runtime for the SENSOR stream here, but the overall results are similar and demonstrate that OWL outperforms BWL in terms of time and memory usage, and scales well on large graph streams.



(a) Memory



(b) Runtime

Fig. 4. Memory and runtime on DBLP stream.

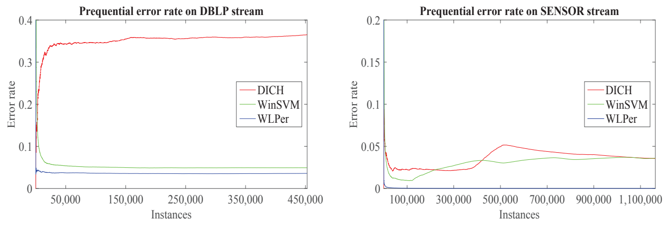
## D. Classification Performance Comparison

The two datasets we are handling are presented in a streaming fashion, which means that every data entity (e.g., nodes or edges) is received in a chronological order. Therefore, we adopt Test-Then-Train as our evaluation methodology: every subgraph for a target node is used for testing the learning model before it is used to train.

*1) Effectiveness Results:* We use error rate as a metric for measuring the effectiveness, which refers to the number of misclassified instances divided by the total number of instances. Prequential error rate [24] records the mean loss at every learning step and allows us to monitor the evolution of the performance over time. A loss function $L(y_i, \hat{y}_i)$ is defined to return 1 if the predicted label $\hat{y}_i$ is not consistent with the true label $y_i$, and 0 otherwise. The mean loss at the $t$th step can then be calculated as

$$e_t = \frac{1}{t}\sum_{i=1}^{t} L(y_i, \hat{y}_i)$$

In Fig. 5, we plot the prequential error rates of the three algorithms on these two datasets when the subgraph extraction threshold $\theta = 1.0$. At the very beginning, the error rates of these algorithms fluctuate due to the fact that they have not received enough training examples for constructing a stable learning model. After receiving a certain number of instances, these models tend to stabilize the error rates. We clearly see that WLPer constantly outperforms its peers after processing a small number of instances. Another observation is that WLPer converges more quickly to its stable error rate, which indicates that WLPer needs the smallest number of instances to train a classifier which produces stable error rates.



(a) DBLP (b) SENSOR

Fig. 5. Prequential error rate at each learning step.

Table I reports the final error rates of all the algorithms using different threshold values for extracting subgraphs. For DBLP, WLPer obtains the best result with the lowest error rates at all the given thresholds, followed by WinSVM and then DICH. The better performance of WLPer and WinSVM, compared to DICH, might be attributed to the fact that subtree patterns (used in the WL kernel) are more informative for classification in DBLP than clique patterns (used in DICH). On the other hand, since WinSVM and WLPer both use the WL kernel values as input, the slightly better performance of WLPer compared to WinSVM might be caused by the fact that the perceptron keeps all misclassified instances as support vectors, whereas WinSVM keeps a mixture of misclassified and correctly-classified instances (and probably tends toward more correctly-classified ones). We also find that, for DBLP, the performance of DICH and WLPer is affected by $\theta$, while the error rate of WinSVM remains relatively stable under different $\theta$s. When $\theta$ tends to be larger, both DICH and WLPer are likely to obtain lower error rates. For SENSOR stream, DICH and WinSVM have generated similar results w.r.t. different $\theta$s. However, WLPer has achieved much lower error rates compared to the other two algorithms. In our experiment, we observe that each subgraph extracted from SENSOR is composed of only a few nodes (most subgraphs contain less than four nodes). As a result, most of the clique patterns discovered in DICH are single nodes or edges. Similarly, most of the subtree patterns mined in the WL iterations have heights 0 (i.e., a single node) or 1 (i.e., an edge). This fact might cause the phenomenon that WinSVM and DICH have similar classification performance on SENSOR stream. However, WLPer has generated much lower error rates than WinSVM due to the fact that WinSVM fails to retain enough support vectors to form a more accurate classification boundary.

TABLE I. FINAL ERROR RATE W.R.T. $\theta$

| | DBLP | | | SENSOR | | |
|---|---|---|---|---|---|---|
| $\theta$ | DICH | WinSVM | WLPer | DICH | WinSVM | WLPer |
| 0.2 | 0.3756 | 0.0493 | **0.0375** | 0.0435 | 0.0359 | **0.000075** |
| 0.4 | 0.3671 | 0.0493 | **0.0375** | 0.0356 | 0.0359 | **0.000052** |
| 0.6 | 0.3684 | 0.0494 | **0.0365** | 0.0356 | 0.0359 | **0.000052** |
| 0.8 | 0.3682 | 0.0493 | **0.0363** | 0.0356 | 0.0359 | **0.000052** |
| 1.0 | 0.3650 | 0.0493 | **0.0359** | 0.0356 | 0.0359 | **0.000052** |

TABLE II. TOTAL RUNTIME W.R.T. $\theta$ (SECONDS)

| | DBLP | | | SENSOR | | |
|---|---|---|---|---|---|---|
| $\theta$ | DICH | WinSVM | WLPer | DICH | WinSVM | WLPer |
| 0.2 | 23894 | 2091 | **744** | 4561 | 1422 | **403** |
| 0.4 | 24972 | 2209 | **774** | 5530 | 1429 | **581** |
| 0.6 | 25507 | 2299 | **799** | 5514 | 1469 | **585** |
| 0.8 | 25666 | 2356 | **821** | 5526 | 1465 | **587** |
| 1.0 | 26025 | 2452 | **854** | 5523 | 1449 | **590** |

*2) Efficiency Results:* We compare the efficiency of the three algorithms on these two datasets in terms of time usage. Table II summarizes the total runtime for the three algorithms w.r.t. different $\theta$s. As expected, WLPer uses much less time than its peers at all $\theta$s. The computational cost for DICH comes from the procedure of detecting discriminative clique patterns and updating the in-memory summary table frequently. By contrast, the other two algorithms do not require any complicated subgraph detection process. WLPer only calculates the inner products between a new graph and all the retained old support vectors and then the linear expansion of these inner products, which is more efficient than the dual optimization process involved in WinSVM in terms of runtime. We further find that SubExtract can reduce the runtime for all the three methods by setting smaller values for $\theta$ on DBLP. However, for SENSOR, the runtime is more steady when $\theta$ is from 0.4 to 1.0. This is probably due to the fact that there are few edges have been filtered out during the extraction process using these $\theta$s.

We also record the average number of edges one algorithm can process within a second as the processing rate to show another aspect of scalability. For DBLP, WLPer is able to process about $16,488$ edges per second, which is **2.9x** faster than WinSVM (about $5,775$ edges/second) and **29x** faster than DICH (about $565$ edges/second). For SENSOR, WLPer can process about $6,499$ edges per second, which is **2.7x** faster than WinSVM (about $2,413$ edges/second) and **9.9x** faster than DICH (about $659$ edges/second).

## E. Sensitivity of Parameters

In the previous subsection, we demonstrate the advantages of our learning framework for different values of the subgraph extraction threshold by comparing with two state-of-the-art classification algorithms. In this part, we study the sensitivity of WLPer w.r.t. the two parameters: budget size and hashing dimensions.

*1) Effectiveness Results:* We investigate the classification error rates of WLPer on the two datasets by fixing one parameter and varying the other. In Fig. 6, we show the variation in effectiveness of WLPer with increasing budget size, while the hashing dimensions are set to the default values. We find that, for DBLP, WLPer with budget size 1000 has generated the lowest error rates across all the instances, followed by the one with budget size 500 and then with budget size 50. Initially when the numbers of retained support vectors are no larger than 50, the three learning curves are identical. After receiving 50 instances in its budget, the learning curve with $B = 50$ starts to deviate from the other two curves. Similarly, the learning curve with $B = 500$ starts to deviate from the curve with $B = 1000$ after it achieves its maximum budget size, say 500. For SENSOR, we find that the learning curves with $B = 500$ and $B = 1000$ overlap. Their numbers of retained support vectors are both 61, which is far lower than their budget sizes. That is the reason why these two curves are identical. The performance of the curve with $B = 50$ is slightly worse than the other two because it can store at most 50 instances in its budget while the other two can retain more support vectors. In both cases, we note that the classification error rate is sensitive to the budget size if the number of support vectors that WLPer tries to retain exceeds the budget size.

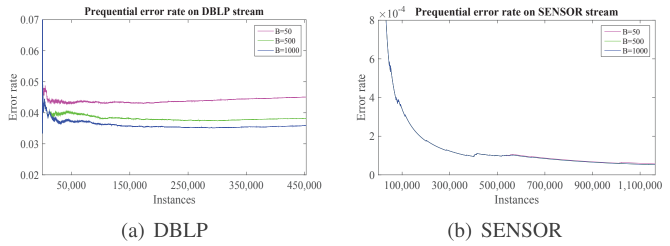


Fig. 6. Prequential error rate w.r.t. budget size.

In order to show how different hashing dimension settings affect classification effectiveness and efficiency, we have chosen three sets of dimensions: $D_1 = \{500, 1000, 5000, 10000\}$, $D_2 = \{10000, 20000, 30000, 40000\}$ and $D_3 = \{50000, 60000, 70000, 80000\}$. For each dimension setting, the dimension value increases, since more distinct subtree patterns will appear as the height of the subtrees increases during the WL iterations. Fig. 7 shows the prequential error rates for WLPer on the two datasets. An obvious finding is that the classification effectiveness with larger dimension values is superior to that with smaller dimension values on the two datasets. This is because large dimension settings will avoid hash collisions at a high probability while small dimension values bring more hash collisions. As a result, the low dimension setting will cause much information loss and therefore deteriorate the classification effectiveness.

*2) Efficiency Results:* We evaluate the efficiency of WLPer on the two datasets in terms of processing speed and memory

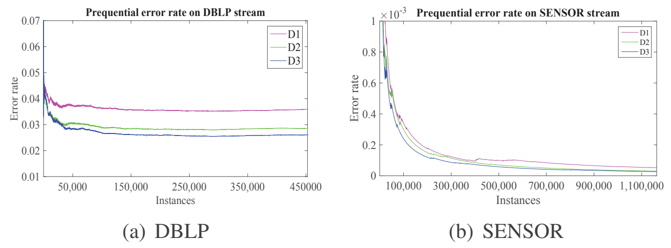


Fig. 7. Prequential error rate w.r.t. hashing dimensions.

consumption[6]. In Fig. 8 and Table III, we show the processing rates and memory usage respectively using different budget sizes. For DBLP, we observe that larger budget size causes longer training time and therefore lower processing rate. The larger $B$ is, the more instances WLPer tends to retain in its budget. Therefore, it takes more time for WLPer to compute similarity values between a new instance and old instances in the budget and more space to store those instances in memory. For SENSOR, since only 61 support vectors have been retained with $B = 500$ and 1000 in our experiment, the processing rates of these two cases are approximately equivalent and they use the same memory. WLPer has used slightly less time and memory with $B = 50$ on SENSOR than the other two, because it reaches its maximum budget size and retains fewer support vectors. We note that WLPer can process around $15,000$ edges per second for DBLP and around $5,900$ edges per second for SENSOR with the default budget size $B = 1000$. This is a very fast processing speed for performing classification tasks on most real-world applications. The last column of Table III shows the theoretical memory bound of the corresponding budget size (see Section IV-D for details). We can see that the actual memory consumption in all these cases is far lower than the theoretical value, which indicates the scalability of WLPer in terms of memory.

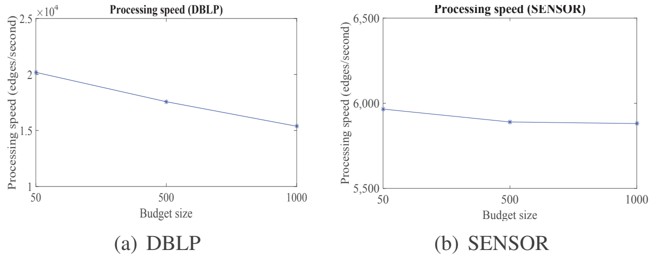


Fig. 8. Processing speed w.r.t. budget size.

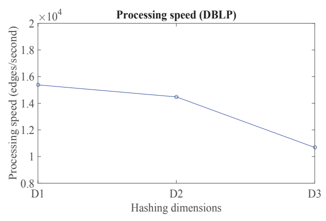
TABLE III. MEMORY USAGE W.R.T. $B$ (IN MB)

| $B$ | DBLP | SENSOR | BOUND |
|---|---|---|---|
| 50 | 0.169 | 0.010 | 6.294 |
| 500 | 0.980 | 0.012 | 62.94 |
| 1000 | 1.706 | 0.012 | 125.9 |

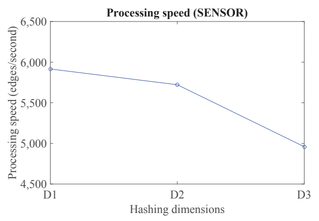
We also test the time and memory usage of WLPer with different sets of hashing dimensions. The results are shown in Fig. 9 and Table IV. We note that WLPer with larger dimensions needs more memory to store the support vectors

[6]The largest amount of memory ever used by an algorithm during its learning process is considered as the memory consumption.

and more time to compute the inner products between a new instance and all the support vectors. This is mainly because large hashing dimensions will generate large feature vectors for representing the subgraph instances. As a result, the memory consumption increases as the dimensions increase while the processing speed decreases as the dimensions increase. However, for each case, the actual memory usage is far less than the theoretical bound listed in the last column of Table IV, and $10,000$ and $5,000$ edges are processed per second for DBLP and SENSOR, respectively.



(a) DBLP



(b) SENSOR

Fig. 9. Processing speed w.r.t. hashing dimensions.

TABLE IV. MEMORY USAGE W.R.T. $D$ (IN MB)

| Hashing | DBLP | SENSOR | BOUND |
|---|---|---|---|
| $D_1$ | 1.706 | 0.012 | 125.9 |
| $D_2$ | 1.725 | 0.016 | 762.9 |
| $D_3$ | 1.754 | 0.017 | 1983.6 |

Overall, we may conclude that WLPer is not only effective but also efficient over a wide variety of practical scenarios. It can achieve impressive classification performance with a proper setting of hashing dimensions and budget size. Therefore, WLPer is a promising classification technique with superior properties in terms of effectiveness and efficiency.

## VI. CONCLUSION

In this paper, we present a framework for performing node classification in a dynamic graph with new nodes and edges streamed in continuously. Firstly, an entropy-based subgraph extraction method is developed to facilitate the usage of a graph kernel to classify nodes in a large graph. Secondly, we propose a kernel computation scheme which enables us to compute the WL graph kernel values in an online mode. As a result, this can be applied to calculating similarity between graphs in an open-ended data stream. Lastly, we incorporate the online kernel into an online perceptron to classify the extracted subgraphs from a large dynamic network. The experimental results on two real-world graph datasets validate our framework for classification effectiveness and efficiency.

Our future work will include experimenting on larger dynamic graphs with nodes and edges streamed in a high speed, to investigate the scalability of our framework. We would also like to conduct a comprehensive comparison between the proposed subgraph extraction method and the existing extraction methods (e.g., random walk-based method).

## ACKNOWLEDGMENT

This research is based upon work sponsored by the National Science Foundation under Grant No. 1318913.

## REFERENCES

[1] S. Macskassy and F. Provost, “Simple models and classification in networked data,” in *Information Systems Working Papers Series*. Stern School of Business, New York University, 2004.

[2] N. S. Ketkar, L. B. Holder, and D. J. Cook, “Mining in the proximity of subgraphs,” in *ACM KDD Workshop on Link Analysis: Dynamics and Statics of Large Networks*, 2006.

[3] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, “Graph kernels,” *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.

[4] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels,” *Journal of Machine Learning Research*, vol. 12, pp. 2539–2561, 2011.

[5] T. Gärtner, P. Flach, and S. Wrobel, “On graph kernels: Hardness results and efficient alternatives,” in *Learning Theory and Kernel Machines*. Springer, 2003, pp. 129–143.

[6] H. Kashima, K. Tsuda, and A. Inokuchi, “Marginalized kernels between labeled graphs,” in *ICML*, vol. 3, 2003, pp. 321–328.

[7] K. M. Borgwardt and H.-P. Kriegel, “Shortest-path kernels on graphs,” in *ICDM*. IEEE, 2005, pp. 74–81.

[8] N. Shervashidze and K. M. Borgwardt, “Fast subtree kernels on graphs,” in *NIPS*, 2009, pp. 1660–1668.

[9] N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S. Vishwanathan, “Efficient graphlet kernels for large graph comparison,” in *International Conference on Artificial Intelligence and Statistics*, 2009, pp. 488–495.

[10] C. C. Aggarwal, “On classification of graph streams.” in *SDM*. SIAM, 2011, pp. 652–663.

[11] L. Chi, B. Li, and X. Zhu, “Fast graph stream classification using discriminative clique hashing,” in *Advances in Knowledge Discovery and Data Mining*. Springer, 2013, pp. 225–236.

[12] T. Guo, L. Chi, and X. Zhu, “Graph hashing and factorization for fast graph stream classification,” in *CIKM*. ACM, 2013, pp. 1607–1612.

[13] B. Li, X. Zhu, L. Chi, and C. Zhang, “Nested subtree hash kernels for large-scale graph classification over streams.” in *ICDM*. IEEE, 2012, pp. 399–408.

[14] Y. Yao and L. Holder, “Scalable svm-based classification in dynamic graphs,” in *ICDM*. IEEE, 2014, pp. 650–659.

[15] B. Weisfeiler and A. Lehman, “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.

[16] J. Kivinen, A. J. Smola, and R. C. Williamson, “Online learning with kernels,” *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, 2004.

[17] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, “Online passive-aggressive algorithms,” *Journal of Machine Learning Research*, vol. 7, pp. 551–585, 2006.

[18] F. Orabona, J. Keshet, and B. Caputo, “The projectron: a bounded kernel-based perceptron,” in *ICML*, 2008, pp. 720–727.

[19] O. Dekel, S. Shalev-shwartz, and Y. Singer, “The forgetron: A kernel-based perceptron on a fixed budget,” in *NIPS*, 2005, pp. 259–266.

[20] K. Singer, “Online classification on a budget,” in *NIPS*, 2004, pp. 225–232.

[21] J. Weston, A. Bordes, and L. Bottou, “Online (and offline) on an even tighter budget,” in *International Workshop on Artificial Intelligence and Statistics*, 2005, pp. 413–420.

[22] G. Cavallanti, N. Cesa-Bianchi, and C. Gentile, “Tracking the best hyperplane with a simple budget perceptron,” *Machine Learning*, vol. 69, no. 2-3, pp. 143–167, 2007.

[23] C.-C. Chang and C.-J. Lin, “Libsvm: a library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, p. 27, 2011.

[24] J. Gama, R. Sebastião, and P. P. Rodrigues, “Issues in evaluation of stream learning algorithms,” in *KDD*. ACM, 2009, pp. 329–338.