Trustworthy anomaly detection for smart manufacturing

Li, Z.

Chapter 2

Graph Neural Networks based Log Anomaly Detection and Explanation

Authors: **Zhong Li**, Jiayang Shi, Matthijs van Leeuwen

A short version of this chapter was published in Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. The extended version (the version presented here) was submitted to Engineering Applications of Artificial Intelligence journal for possible publication.

Abstract

Event logs are widely used to record the status of high-tech systems, making log anomaly detection important for monitoring those systems. Most existing log anomaly detection methods take a log event count matrix or log event sequences as input, exploiting quantitative and/or sequential relationships between log events to detect anomalies. However, only considering quantitative or sequential relationships may result in low detection accuracy. To alleviate this problem, we propose a graph-based method for unsupervised log anomaly detection, dubbed *Logs2Graphs*, which first converts event logs into attributed, directed, and weighted graphs, and then leverages graph neural networks to perform graph-level anomaly detection. Specifically, we introduce One-Class Digraph Inception Convolutional Networks, abbreviated as OCDiGCN, a novel graph neural network model for detecting graph-level anomalies in a collection of attributed, directed, and weighted graphs. By integrating graph representation and anomaly detection, OCDiGCN learns a specialized representation that leads to high detection accuracy. Crucially, we furnish a concise set of nodes pivotal in OCDiGCN's prediction as explanations for each detected anomaly, offering valuable insights for subsequent root cause analysis. Experiments on five benchmark datasets show that *Logs2Graphs* exhibits comparable or superior performance when compared to state-of-the-art log anomaly detection methods. Moreover, we introduce an ongoing case study wherein we are deploying *Logs2Graphs* to lithography systems, aiming to enhance fault detection and analysis in the wafer transfer subsystem.

2.1 Introduction

Modern high-tech systems, such as cloud computers or lithography machines, typically consist of a large number of components. Over time these systems have become larger and more complex, making manual system operation and maintenance hard or even infeasible [56]. Therefore, automated system operation and maintenance is highly desirable. To achieve this, system logs are universally used to record system states and important events. By analyzing these logs, faults and potential risks can be identified, and remedial actions may be taken to prevent severe problems. System logs are usually semi-structured texts though, and identifying anomalies through log anomaly detection is often challenging.

Since both industry and academia show great interest in identifying anomalies from logs, a plethora of log anomaly detection methods have been proposed. Existing log anomaly detection methods can be roughly divided into three categories: quantitative-based, sequence-based, and graph-based methods. Specifically, quantitative-based methods, such as OCSVM [63] and PCA [64], utilize a log event count matrix to detect anomalies, and are therefore unable to capture semantic information of and sequential information between log events. Meanwhile, sequence-based methods, including DeepLog [65] and LogAnomaly [66], aim to detect anomalies by taking sequential (and sometimes semantic) information into account. They cannot consider the full structure among log events though. In contrast, graph-based methods, such as GLAD-PAW [67] and GLAD [68], convert logs to graphs and exploit semantic information as well as the structure among log events, exhibiting the following three advantages over the former two categories of methods [69]: 1) they are able to identify problems for which the structure among events is crucial, such as performance degradation; 2) they are capable of providing contextual log messages corresponding to the identified problems; and 3) they can provide the 'normal' operation process in the form of a graph, helping end-users find root-causes and take remedial actions. However, graph-based methods like GLAD transform log events into *undirected* graphs though, which may fail to capture important information on the order among log events. Moreover, most existing graph-based methods perform graph representation and anomaly detection separately, leading to suboptimal detection accuracy.

Moreover, as highlighted by Li et al. [70], with the growing adoption of anomaly detection algorithms in safety-critical domains, there is a rising demand for providing explanations for the decisions made within those domains. This requirement, driven by ethical considerations and regulatory mandates, underscores the importance of

accountability and transparency in such contexts. Moreover, in practical applications, the attainment of precise anomaly explanations contributes to the timely isolation and diagnosis of anomalies, which can mitigate the impact of anomalies by facilitating early intervention [71]. However, to our knowledge, most existing log anomaly detection methods focus exclusively on accurate detection without giving explanations.

To overcome these limitations, we propose *Logs2Graphs*, a graph-based unsupervised log anomaly detection approach for which we design a novel one-class graph neural network. Specifically, *Logs2Graphs* first utilizes off-the-shelf methods to learn a semantic embedding for each log event, and then assigns log messages to different groups. Second, *Logs2Graphs* converts each group of log messages into an attributed, directed, and weighted graph, with each node representing a log event, the node attributes containing its semantic embedding, directed edges representing how an event is followed by other events, and the corresponding edge weights indicating the number of times the events follow each other. Third, by coupling the graph representation learning and anomaly detection objectives, we introduce One-Class Digraph Inception Convolutional Networks (OCDiGCN) as a novel method to detect anomalous graphs from a set of graphs. As a result, *Logs2Graphs* leverages the rich and expressive power of attributed, directed, and edge-weighted graphs to represent logs, followed by using graph neural networks to effectively detect graph-level anomalies, taking into account both semantic information of log events and structure information (including sequential information as a special case) among log events. Importantly, by decomposing the anomaly score of a graph into individual nodes and visualizing these nodes based on their contributions, we provide understandable explanations for identified anomalies.

Overall, our contributions can be summarized as follows: (1) We introduce *Logs2Graphs*, which formalizes log anomaly detection as a graph-level anomaly detection problem and represents log sequences as directed graphs to capture more structure information than previous approaches; (2) We introduce OCDiGCN, a general end-to-end unsupervised graph-level anomaly detection method for attributed, directed and edge-weighted graphs. By coupling the graph representation and anomaly detection objectives, we improve the potential for accurate anomaly detection over existing approaches; (3) For each detected anomaly, we identify important nodes as explanations, offering cues for subsequent root cause diagnosis; (4) We empirically compare our approach to eight state-of-the-art log anomaly detection methods on five benchmark datasets, showing that *Logs2Graphs* performs at least on par with and often better than its competitors.

Comparison with our previous work. This paper has been published as a two-pages short paper [55], in which we only presented the problem addressed, a brief

summary of the algorithm, and the achieved main results. In contrast, in this paper we give the motivations of designing algorithms, present related work and the design details of algorithms, elucidate the experiment setting, and present full experiment results with detailed interpretation. Importantly, we also present an ongoing case study wherein we are applying our algorithm to lithography systems.

Organization of the paper. The remainder of this paper is organized as follows. Chapter 2.2 revisits related work, after which Chapter 2.3 formalizes the problem. Chapter 2.4 describes Digraph Inception Convolutional Networks [72], which are used for *Logs2Graphs* in Chapter 2.5. We then evaluate *Logs2Graphs* in Chapter 2.6 with publicly available datasets, and present an ongoing case-study in Chapter 2.7. We analyze the threats to validity in Chapter 2.8, and conclude in Chapter 2.9.

2.2 Related Work

Graph-based log anomaly detection methods usually comprise five steps: log parsing, log grouping, graph construction, graph representation learning, and anomaly detection. In this paper we focus on graph representation learning, log anomaly detection, and explanation, thus only revisiting related work in these fields.

2.2.1 Graph Representation Learning

Graph-level representation learning methods, such as GIN [73] and Graph2Vec [74], are able to learn a mapping from graphs to vectors. Further, graph kernel methods, including Weisfeiler-Lehman (WL) [75] and Propagation Kernels (PK) [76], can directly provide pairwise distances between graphs. Both types of methods can be combined with off-the-shelf anomaly detectors, such as OCSVM [63] and iForest [77], to perform graph-level anomaly detection.

To improve on these naïve approaches, efforts have been made to develop graph representation learning methods especially for anomaly detection. For instance, OCGIN [78] and GLAM [79] combine the GIN [73] representation learning objective with the SVDD objective [80] to perform graph-level representation learning and anomaly detection in an end-to-end manner. GLocalKD [81] performs random distillation of graph and node representations to learn 'normal' graph patterns. Further, OCGTL [82] combines neural transformation learning and one-class classification to learn graph representations for anomaly detection. Although these methods are unsupervised or semi-supervised, they can only deal with attributed, undirected, and unweighted

graphs.

iGAD [83] considers graph-level anomaly detection as a graph classification problem and combines attribute-aware graph convolution and substructure-aware deep random walks to learn graph representations. However, iGAD is a supervised method, and can only handle attributed, undirected, and unweighted graphs. CODEtect [84] takes a pattern-based modeling approach using the minimum description length principle and identifies anomalous graphs based on *motifs*. CODEtect can (only) deal with labeled, directed, and edge-weighted graphs, but is computationally very expensive. In contrast, we introduce a general unsupervised method for graph-level anomaly detection that can handle attributed, directed and edge-weighted graphs.

2.2.2 Log Anomaly Detection and Explanation

Log anomaly detection methods can be roughly divided into: 1) traditional, 'shallow' methods, such as principal component analysis (PCA) [64], one-class SVM (OCSVM) [63], isolation forest (iForest) [77], and histogram-based outlier score (HBOS) [85], which take a log event count matrix as input and analyze quantitative relationships; 2) deep learning based methods, such as DeepLog [65], LogAnomaly [66], and AutoEncoder [86], which employ sequences of log events (and sometimes their semantic embeddings) as input, analyzing sequential information and possibly semantic information of log events to identify anomalies; and 3) graph-based methods, such as TCFG [69] and GLAD-PAW [67], which first convert logs into graphs and then perform graph-level anomaly detection.

To our knowledge, only a few works [67, 87, 88, 68] have capitalized on the powerful learning capabilities of graph neural networks for log anomaly detection. GLAD [68] transforms logs into undirected, weighted, and attributed heterogeneous graph, and propose a temporal-attentive graph edge anomaly detection model to detect anomalous relations. However, converting logs into undirected graphs may result in loss of important sequential information. Further, DeepTraLog [87] combines traces and logs to generate a so-called Trace Event Graph, which is an attributed and directed graph. On this basis, they train a Gated Graph Neural Networks based Deep Support Vector Data Description model to identify anomalies. However, their approach requires the availability of both traces and logs, and is unable to handle edge weights. In contrast, like LogGD [88] and GLAD-PAW [67], our proposed *Logs2Graphs* approach is applicable to generic logs by converting logs into attributed, directed, and edge-weighted graphs. However, LogGD is a supervised method that requires fully labeled train-

ing data, which is usually impractical and even impossible. Moreover, LogGD and GLAD-PAW [67] are not capable of providing explanations for identified anomalies. In contrast, our proposed algorithm OCDiGCN is a general *unsupervised* graph-level anomaly detection method for attributed, directed, and edge-weighted graphs, able to provide explanations for identified anomalies.

Although anomaly explanation has received much attention in traditional anomaly detection [70, 89], only a few studies [90] considered log anomaly explanation. Specifically, PLELog [90] offers explanations by quantifying the significance of individual log events within an anomalous log sequence, thereby facilitating improved identification of relevant log events by operators. Similarly, our method provides explanations for anomalous log groups by identifying and visualizing a small subset of important nodes.

2.3 Problem Statement

Before we state the log anomaly detection problem, we first introduce notation and definitions regarding event logs and graphs.

Event logs. *Logs* are used to record system status and important events, and are usually collected and stored centrally as log files. A *log file* typically consists of many *log messages*. Each *log message* is composed of three components: a timestamp, an event type (*log event* or *log template*), and additional information (*log parameters*). *Log parsers* are used to extract log events from log messages.

Further, log messages can be grouped into *log groups* (a.k.a. *log sequences*) using certain criteria. Specifically, if a *log identifier* is available for each log message, one can group log messages based on such identifiers. Otherwise, one can use a *fixed* or *sliding window* to group log messages. Besides, counting the occurrences of each log event within a log group yields an *event count vector*. Consequently, for a log file consisting of many log groups, one can obtain an *event count matrix*. The process of generating an *event count matrix* (or other feature matrix) is known as *feature extraction*. Extracted features are often used as input to an anomaly detection algorithm to identify *log anomalies*, i.e., log messages or log groups that deviate from what is considered 'normal'.

Graphs. We consider an attributed, directed, and edge-weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X}, \mathbf{Y})$, where $\mathcal{V} = \{v_1, ..., v_{|\mathcal{V}|}\}$ denotes the set of *nodes* and $\mathcal{E} = \{e_1, ..., e_{|\mathcal{E}|}\} \subseteq \mathcal{V} \times \mathcal{V}$ represents the set of edges. If $(v_i, v_j) \in \mathcal{E}$, then there is an edge from node v_i to node v_j. Moreover, $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$ is the node attribute matrix, with the i-th row representing the

attributes of node v_i, and d is the number of attributes. Besides, $\mathbf{Y} \in \mathbb{N}^{|\mathcal{E}|\times|\mathcal{E}|}$ is the edge-weight matrix, where $\mathbf{Y}_{ij}$ represents the weight of the edge from node v_i to v_j.

Equivalently, $\mathcal{G}$ can be described as $(\mathbf{A}, \mathbf{X}, \mathbf{Y})$, with adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, where $\mathbf{A}_{ij} = \mathbb{I}[(v_i, v_j) \in \mathcal{E}]$ indicates whether there is an edge from node v_i to node v_j, for $i, j \in \{1, ..., |\mathcal{V}|\}$.

2.3.1 Graph-based Log Anomaly Detection

Given a set of log files, we let $\mathcal{L} = \{L_1, ..., L_{|\mathcal{L}|}\}$ denote the set of unique log events. We divide the log messages into M log groups $\mathbf{Q} = \{\mathbf{q}_1, ..., \mathbf{q}_m, ..., \mathbf{q}_M\}$, where $\mathbf{q}_m = \{\mathbf{q}_{m1}, ..., \mathbf{q}_{mn}, ..., \mathbf{q}_{mN}\}$ is a log group and $\mathbf{q}_{mn}$ a log message.

For each log group $\mathbf{q}_m$, we construct an attributed, directed, and edge-weighted graph $\mathcal{G}_m = (\mathcal{V}_m, \mathcal{E}_m, \mathbf{X}_m, \mathbf{Y}_m)$ to represent the log messages and their relationships. Specifically, each node $v_i \in \mathcal{V}_m$ corresponds to exactly one log event $L \in \mathcal{L}$ (and vice versa). Further, an edge $e_{ij} \in \mathcal{E}_m$ indicates that log event i is at least once immediately followed by log event j in $\mathbf{q}_m$. Attributes $\mathbf{x}_i \in \mathbf{X}_m$ represent the semantic embedding of log event i, and $y_{ij} \in \mathbf{Y}_m$ is the weight of edge e_{ij}, representing the number of times event i was immediately followed by event j. In this manner, we construct a set of log graphs $\{\mathcal{G}_1, ..., \mathcal{G}_m, ..., \mathcal{G}_M\}$.

We use these definitions to define graph-based log anomaly detection:

Problem 1 (Graph-based Log Anomaly Detection). *Given a set of attributed, directed, and weighted graphs that represent logs, find those graphs that are notably different from the majority of graphs.*

What we mean by 'notably different' will have to be made more specific when we define our method, but we can already discuss what types of anomalies can potentially be detected. Most methods aim to detect two types of anomalies:

- A log group (here a graph) is considered a *quantitative anomaly* if the occurrence frequencies of some events in the group are higher or lower than expected from what is commonly observed. For example, if a file is opened (event A) twice, it should normally also be closed (event B) twice. In other words, the number of event occurrences $\#A = \#B$ in a normal pattern and an anomaly is detected if $\#A \neq \#B$.

- A log group is considered to contain *sequential anomalies* if the order of certain events violates the normal order pattern. For instance, a file can be closed only

after it has been opened in a normal workflow. In other words, the order of event occurrences $A \rightarrow B$ is considered normal while $B \rightarrow A$ is considered anomalous.

An advantage of graph-based anomaly detection is that it can detect these two types of anomalies, but also anomalies reflected in the structures of the graphs. Moreover, no *unsupervised* log anomaly detection approaches represent event logs as attributed, directed, weighted graphs, which allow for even higher expressiveness than undirected graphs (and thus limiting the information loss resulting from the representation of the log files as graphs).

2.4 Preliminaries: Digraph Inception Convolutional Nets

To learn node representations for attributed, directed, and edge-weighted graphs, Tong et al. [72] proposed Digraph Inception Convolutional Networks (DiGCN).

Specifically, given a graph $\mathcal{G}$ described by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, a node attribute matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}|\times d}$, and an edge-weight matrix $\mathbf{Y} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, DiGCN defines the k-th order digraph convolution as

$$\mathbf{Z}^{(k)} = \begin{cases} \mathbf{X}\Theta^{(0)} & k = 0 \\ \Psi\mathbf{X}\Theta^{(1)} & k = 1 \\ \Phi\mathbf{X}\Theta^{(k)} & k \geq 2, \end{cases} \tag{2.1}$$

where $\Psi = \frac{1}{2}\left(\Pi^{(1)\frac{1}{2}}\mathbf{P}^{(1)}\Pi^{(1)\frac{-1}{2}} + \Pi^{(1)\frac{-1}{2}}\mathbf{P}^{(1)T}\Pi^{(1)\frac{1}{2}}\right)$ and $\Phi = \mathbf{W}^{(k)\frac{-1}{2}}\mathbf{P}^{(k)}\mathbf{W}^{(k)\frac{-1}{2}}$. Particularly, $\mathbf{Z}^{(k)} \in \mathcal{R}^{|\mathcal{V}|\times f}$ denotes the convolved output with f output dimension, and $\Theta^{(0)}, \Theta^{(1)}, \Theta^{(k)}$ represent the trainable parameter matrices.

Moreover, $\mathbf{P}^{(k)}$ is the k-th order proximity matrix defined as

$$\mathbf{P}^{(k)} = \begin{cases} \mathbf{I} & k = 0 \\ \tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}} & k = 1 \\ Ins\left((\mathbf{P}^{(1)})^{(k-1)}(\mathbf{P}^{(1)T})^{(k-1)}\right) & k \geq 2, \end{cases} \tag{2.2}$$

where $\mathbf{I} \in \mathcal{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is an identity matrix, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, and $\tilde{\mathbf{D}}$ denotes the diagonal degree matrix with $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. Besides, $Ins\left((\mathbf{P}^{(1)})^{(k-1)}(\mathbf{P}^{(1)T})^{(k-1)}\right)$ is defined

as

$$\frac{1}{2}Intersect\left((\mathbf{P}^{(1)})^{(k-1)}(\mathbf{P}^{(1)T})^{(k-1)},(\mathbf{P}^{(1)T})^{(k-1)}(\mathbf{P}^{(1)})^{(k-1)}\right)$$

, with $Intersect(\cdot)$ denoting the element-wise intersection of two matrices (see [72] for computation details). In addition, $\mathbf{W}^{(k)}$ is the diagonalized weight matrix of $\mathbf{P}^{(k)}$, and $\Pi^{(1)}$ is the approximate diagonalized eigenvector of $\mathbf{P}^{(1)}$. Particularly, the approximate diagonalized eigenvector is calculated based on personalized PageRank [91], with a parameter α to control the degree of conversion from a digraph to an undirected graph. We omit the details to conserve space, and refer to [72] for more details.

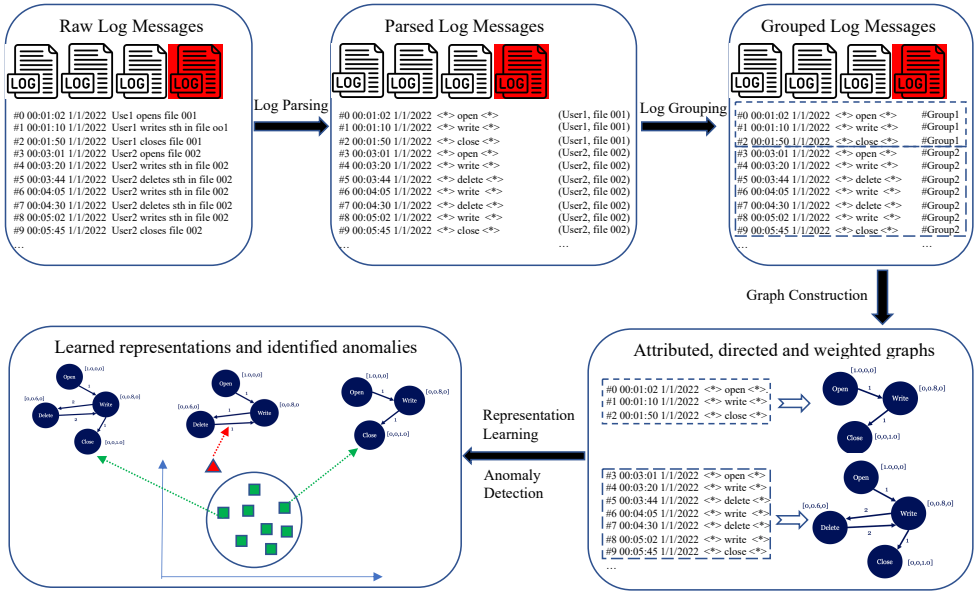

Figure 2.1: The Logs2Graphs pipeline. We use attributed, directed, and weighted graphs for representing the log files with high expressiveness, and integrate representation learning and anomaly detection for accurate anomaly detection. We use off-the-shelf methods for log parsing, log grouping, and graph construction.

After obtaining the multi-scale features $\{\mathbf{Z}^{(0)},\mathbf{Z}^{(1)},...,\mathbf{Z}^{(k)}\}$, DiGCN defines an Inception block as

$$\mathbf{Z}=\sigma\left(\Gamma\left(\mathbf{Z}^{(0)},\mathbf{Z}^{(1)},...,\mathbf{Z}^{(k)}\right)\right), \tag{2.3}$$

where σ represents an activation function, and $\Gamma(\cdot)$ denotes a fusion operation, which can be summation, normalization, and concatenation. In practice, we often adapt a fusion operation that keeps the output dimension unchanged, namely $\mathbf{Z}\in\mathcal{R}^{|\mathcal{V}|\times f}$. As

a result, the i-th row of $\mathbf{Z}$ (namely $\mathbf{Z}_i$) denotes the learned vector representation for node v_i in a certain layer.

2.5 Graph-Based Anomaly Detection for Event Logs

We propose *Logs2Graphs*, a graph-based log anomaly detection method tailored to event logs. The overall pipeline consists of the usual main steps, i.e., log parsing, log grouping, graph construction, graph representation learning, and anomaly detection, and is illustrated in Figure 2.1. Note that we couple the graph representation learning and anomaly detection steps to accomplish end-to-end learning once the graphs have constructed.

First, after collecting logs from a system, the *log parsing* step extracts log events and log parameters from raw log messages. Since log parsing is not the primary focus of this article, we use Drain [92] for this task. Drain is a log parsing technique with fixed depth tree, and has been shown to generally outperform its competitors [93]. We make the following assumptions on the log files:

- Logs files are written in English;
- Each log message contains at least the following information: date, time, operation detail, and log identifier;
- The logs contain enough events to make the mined relationships (quantitative, sequential, structural) statistically meaningful, i.e., it must be possible to learn from the logs what the 'normal' behavior of the system is.

Second, the *log grouping* step uses the log identifiers to divide the parsed log messages into log groups. Third, for each resulting group of log messages, the *graph construction* steps builds an attributed, directed, and edge-weighted graph, as described in more detail in Chapter 2.5.1. Fourth and last, in an integrated step for *graph representation learning and anomaly detection*, we learn a One-Class Digraph Inception Convolutional Network (OCDiGCN) based on the obtained set of log graphs. The resulting model can be used for graph-level anomaly detection. This model couples the graph representation learning objective and anomaly detection objective, and is thus trained in an end-to-end manner. The model, its training, and its use for graph-level anomaly detection are explained in detail in Chapter 2.5.2.

2.5.1 Graph Construction

We next explain how to construct an attributed, directed, and edge-weighted graph given a group of parsed log messages, and illustrate this in Figure 2.2.

First, we utilize nodes to represent different log events. As a result, the number of nodes depends on the number of unique log events that occur within the log group. Second, starting from the first line of log messages in chronological order, we add a directed edge from log event L_i to L_j and set its edge-weight to 1 if the next event after L_i is L_j. If the corresponding edge already exists, we increase its edge-weight by 1. In this manner, we obtain a labeled, directed, and edge-weighted graph.

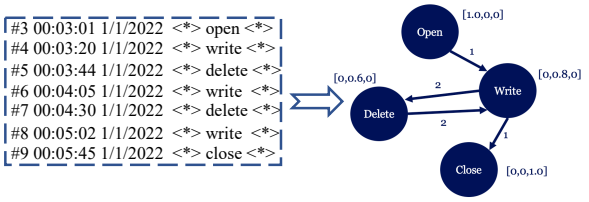

Figure 2.2: The construction of an attributed, directed, and edge-weighted graph from a group of log messages.

However, using only the labels (e.g., *open* or *write*) of log events for graph construction may lead to missing important information. That is, we can improve on this by explicitly taking the semantic information of log events into account, by which we mean that we should look at the text of the log event in entirety. Specifically, we generate a vector representation for each log event as follows:

1. *Preprocessing*: for each log event, we first remove non-character words and stop words, and split compound words into separate words;

2. *Word embedding*: we use Glove [94], a pre-trained word embedding model with 200 embedding dimensions to generate a vector representation for each word in a log event;

3. *Sentence embedding*: we generate a vector representation for each log event. Since the words in a sentence are usually not of equal importance, we use Term Frequency-Inverse Document frequency (TF-IDF) [95] to measure the importance of words. As a result, the weighted sum of word embedding vectors composes the vector representation of a log event.

By augmenting the nodes with the vector representations of the log events as attributes, we obtain an attributed, directed, and edge-weighted graph.

2.5.2 OCDiGCN: One-Class Digraph Inception Convolutional Nets

We next describe One-Class Digraph Inception Convolutional Networks, abbreviated as OCDiGCN, a novel method for end-to-end graph-level anomaly detection. We chose to build on Digraph Inception Convolutional Networks (DiGCN) [72] for their capability to handle directed graphs, which we argued previously is an advantage in graph-based log anomaly detection.

Given that DiGCN was designed for node representation learning, we repurpose it for graph representation learning as follows:

$$\mathbf{z} = \text{Readout}(\mathbf{Z}_i \mid i \in \{1, 2, ..., |\mathcal{V}|\}). \tag{2.4}$$

That is, at the final iteration layer, we utilize a so-called Readout$(\cdot)$ function to aggregate node vector representations to obtain a graph vector representation. Importantly, Readout$(\cdot)$ can be a simple permutation-invariant function such as maximum, sum or mean, or a more advanced graph-level pooling function [96].

Next, note that DiGCN work did not explicitly enable learning edge features (i.e., $\mathbf{Y}$). However, as DiGCN follows the Message Passing Neural Network (MPNN) framework [97], incorporating $\mathbf{Y}$ into Equation (1) and conducting computations in Equations (2-4) analogously enables learning edge features.

Now, given a set of graphs $\{\mathcal{G}_1, ..., \mathcal{G}_m, ..., \mathcal{G}_M\}$, we can use Equation (2.4) to obtain an explicit vector representation for each graph, respectively. We denote the vector presentation of $\mathcal{G}_m$ learned by the DiGCN model as DiGCN$(\mathcal{G}_\text{m}; \mathcal{H})$.

In graph anomaly detection, anomalies are typically identified based on a reconstruction or distance loss [98]. In particular, the One-Class Deep SVDD objective [99] is commonly used for two reasons: it can be easily combined with other neural networks, and more importantly, it generally achieves a state-of-the-art performance [36]. To detect anomalies, we thus train a one-class classifier by optimizing the following One-Class Deep SVDD objective:

$$\min_{\mathcal{H}} \frac{1}{M} \sum_{m=1}^{M} \|\text{DiGCN}(\mathcal{G}_\text{m}; \mathcal{H}) - \mathbf{o}\|_2^2 + \frac{\lambda}{2} \sum_{l=1}^{L} \|\mathbf{H}^{(l)}\|_F^2, \tag{2.5}$$

Table 2.1: Summary of datasets. #Events refers to the number of log event templates obtained using log parser Drain [92]. #Groups means the number of generated graphs. #Anomalies represents the number of anomalous graphs. #Nodes denotes the average number of nodes in generated graphs. #Edges indicates the average number of edges in the generated graphs.

Name	#Events	#Graphs	#Anomalies	#Nodes	#Edges
HDFS	48	575,061	16,838	7	20
Hadoop	683	978	811	34	120
BGL	1848	69,251	31,374	10	30
Spirit	834	10,155	4,432	6	24
Thunderbird	1013	52,160	6,814	16	52

where $\mathbf{H}^{(l)}$ represents the trainable parameters of DiGCN at the l-th layer, namely $(\Theta^{(0)(l)}, \Theta^{(1)(l)}, ..., \Theta^{(k)(l)})^T$, $\mathcal{H}$ denotes $\{\mathbf{H}^{(1)}, ..., \mathbf{H}^{(L)}\}$, $\lambda > 0$ represents the weight-decay hyperparameter, $\|\cdot\|_2$ is the Euclidean norm, and $\|\cdot\|_F$ denotes the Frobenius norm. Moreover, $\mathbf{o}$ is the center of the hypersphere in the learned representation space. Ruff et al. [99] empirically found that setting $\mathbf{o}$ to the average of the network representations (i.e., graph representations in our case) obtained by performing an initial forward pass is a good strategy.

Ruff et al. [99] also pointed out, however, that One-Class Deep SVDD classification may suffer from a hypersphere collapse, which will yield trivial solutions, namely mapping all graphs to a fixed center in the representation space. To avoid a hypersphere collapse, the hypersphere center $\mathbf{o}$ is set to the average of the network representations, the bias terms in the neural networks are removed, and unbounded activation functions such as ReLU are preferred.

After training the model on a set of non-anomalous graphs (or with a very low proportion of anomalies), given a test graph $\mathcal{G}_m$, we define its distance to the center in the representation space as its anomaly score, namely

$$score(\mathcal{G}_m) = \|\mathrm{DiGCN}(\mathcal{G}_\mathrm{m}; \mathcal{H}) - \mathbf{o}\|_2. \tag{2.6}$$

Training and hyperparameters: In summary, OCDiGCN is composed of an L-layer DiGCN architecture to learn node representations, plus a Readout$(\cdot)$ function to obtain the graph representation. It is trained in an end-to-end manner via optimizing the SVDD objective, which can be optimized using stochastic optimization techniques such as Adam [100]. Overall, OCDiGCN takes a collection of non-anomalous graphs

Table 2.2: Description of hyperparameters involved in OCDiGCN. **Range** indicates the values that we have tried on validation data, and boldfaced values are the values suggested to use in experiments. Particularly, for the embedding dimensions: 300 is suggested for BGL and 128 for others. For the batch sizes: 32 is suggested for HDFS and 128 for others. For the training epochs: 100 for BGL and Thunderbird, 200 for HDFS, 300 for Hadoop and 500 for Spirit are suggested.

Symbol	Meaning	Range
Bs	Batch size	{16, 32, 64, **128**, 256, 512, 1024, 1536, 2048, 2560}
Op	optimization method	Adam, **SGD**
L	number of layers	$\{\mathbf{1}, 2, 3, 4, 5\}$
λ	weight decay parameter	$\{\mathbf{0.0001}, 0.001, 0.01, 0.1\}$
η	learning rate	$\{0.0001, 0.001, \mathbf{0.01}\}$
k	proximity parameter	$\{\mathbf{1}, 2\}$
α	teleport probability	$\{0.05, \mathbf{0.1}, 0.2\}$
Γ	fusion operation if $k \geq 2$	sum, concatenation
Re	readout function	**mean**, sum, max
d	embedding dimension	$\{32, 64, \mathbf{128}, 256, 300\}$
Ep	Epochs for training	range(100,1000,50)

and a set of hyperparameters, which are outlined in Table 2.2, as inputs. The pseudo-code for Logs2Graphs is given in Algorithm 1.

Algorithm 1 Pseudo-code of Logs2Graphs

Input: Training dataset D_{tr}, testing dataset D_{ts}, model θ
Output: Predicted labels and explanations for D_{ts}

1: $\hat{D}_{tr}, \hat{D}_{ts} \leftarrow Drain\_parse(D_{tr}), Drain\_parse(D_{ts})$
2: Group $\hat{D}_{tr}$ and $\hat{D}_{ts}$ based on log identifier $\rightarrow$ Obtain grouped dataset $\tilde{D}_{tr}$ and $\tilde{D}_{ts}$
3: Construct graphs using $\tilde{D}_{tr}$ and $\tilde{D}_{ts}$ $\rightarrow$ Obtain graph sets $\mathbf{Q}_{tr}$ and $\mathbf{Q}_{ts}$
4: Train the OCDiGCN model using Equation (2.5) with $\mathbf{Q}_{tr}$ $\rightarrow$ Obtain trained model $\hat{\theta}$
5: Use $\hat{\theta}$ to predict anomalies in $\mathbf{Q}_{ts}$ $\rightarrow$ Obtain a set of anomalies $\{Q_1, ..., Q_n\}$
6: Generate explanations for $Q_i \in \{Q_1, ..., Q_n\}$

2.5.3 Anomaly Explanation

Our anomaly explanation method can be regarded as a decomposition method [101], that is, we build a score decomposition rule to distribute the prediction anomaly score to the input space. Concretely, a graph $\mathcal{G}_m$ is identified as anomalous if and only if its graph-level representation has a large distance to the hyper-sphere center (Equation

2.6). Further, the graph-level representation is obtained via a Readout(·) function applied on the node-level representations (Equation 2.4). Therefore, if the Readout(·) function is attributable (such as the sum or the mean), we can easily obtain the a small subset of important nodes (in the penultimate layer) whose node embeddings contribute the most to the distance. Specifically, the importance score of node v_j (in the penultimate layer) in a graph $\mathcal{G}_m$ is defined as

$$\frac{|score(\mathcal{G}_m) - score(\mathcal{G}_m \setminus \{\mathbf{Z}_j\})|}{score(\mathcal{G}_m)} \tag{2.7}$$

where $score(\mathcal{G}_m)$ is defined in Equation 2.6 and $score(\mathcal{G}_m \setminus \{\mathbf{Z}_j\})$ is the anomaly score by removing the embedding vector of v_j (namely $\mathbf{Z}_j$) when applying the Readout function to obtain the graph-level representation.

Next, for each important node (with a high importance score) in the penultimate layer, we extend the LRP (Layerwise Relevance Propagation) algorithm [102] to obtain a minor set of important nodes in the input layer (this is not the contribution of our paper and we simply follow the practice in [103, 104]). If certain of these nodes are connected by edges, the resulting subgraphs can provide more meaningful explanations. As the LRP method generates explanations utilizing the hidden features and model weights directly, its explanation outcomes are deemed reliable and trustworthy [70].

2.6 Experiments

We perform extensive experiments to answer the following questions:

1. **Detection accuracy:** How effective is *Logs2Graphs* at identifying log anomalies when compared to state-of-the-art methods?

2. **Directed vs. undirected graphs**: Is the directed log graph representation better than the undirected version for detecting log anomalies?

3. **Node Labels vs. Node Attributes**: How important is it to use semantic embeddings of log events as node attributes?

4. **Robustness analysis:** To what extent is Logs2Graphs robust to contamination of the training data?

5. **Ability to detect structural anomalies:** Can Logs2Graphs better capture

structural anomalies and identify structurally equivalent normal instances than its contenders?

6. **Explainability Analysis:** How understandable are the anomaly explanations given by Logs2Graphs?

7. **Sensitivity analysis:** How do the values of the hyperparameters influence the detection accuracy?

8. **Runtime analysis:** What are the runtimes for the different methods?

2.6.1 Experiment Setup

Datasets

The five datasets that we use, summarized in Table 2.1, were chosen for three reasons: 1) they are commonly used for the evaluation of log anomaly detection methods; 2) they contain ground truth labels that can be used to calculate evaluation metrics; and 3) they include log identifiers that can be used for partitioning log messages into groups. For each group of log messages in a dataset, we label the group as anomalous if it contains at least one anomaly. More details are given as follows:

- HDFS [64] consists of Hadoop Distributed File System logs obtained by running 200 Amazon EC2 nodes. These logs contain *block_id*, which can be used to group log events into different groups. Moreover, these logs are manually labeled by Hadoop experts.

- Hadoop [105] was collected from a Hadoop cluster consisting of 46 cores over 5 machines. The *ContainerID* variable is used to divide log messages into different groups.

- BGL, Spirit, and Thunderbird contain system logs collected from the Blue-Gene/L (BGL), Spirit, and Thunderbird supercomputing systems located at Sandia National Labs, respectively. For those datasets, each log message was manually inspected by engineers and labeled as normal or anomalous. For BGL, we use all log messages, and group log messages based on the *Node* variable. For Spirit and Thunderbird, we only use the first 1 million and first 5 million log messages for evaluation, respectively. Furthermore, for these two datasets, the *User* is used as log identifier to group log messages. However, considering that an ordinary user may generate hundreds of thousands of logs, we regard every

100 consecutive logs of each user as a group. If the number of logs is less than 100, we also consider it as a group.

Baselines

To investigate the performance of *Logs2Graphs*, we compare it with the following seven log anomaly detection methods: Principal Component Analysis (PCA) [64], One-Class SVM (OCSVM) [63], Isolation Forest (iForest) [77], HBOS [85], DeepLog [65], LogAnomaly [66], and AutoEncoder [86], and one state-of-the-art graph level anomaly detection method: GLAM [79].

We choose these methods as baselines because they are often regarded to be representatives of traditional machine learning-based (PCA, OCSVM, IForest, HBOS) and deep learning-based approaches (DeepLog, LogAnomaly and AutoEncoder), respectively. All methods are unsupervised or semi-supervised methods that do not require labeled anomalous samples for training the models.

Evaluation Metrics

The Area Under Receiver Operating Characteristics Curve (ROC AUC) and the Area Under the Precision-Recall Curve (PRC AUC) are widely used to quantify the detection accuracy of anomaly detection [106]. This is mainly because they can provide a single value that summarizes the overall performance of the anomaly detection model across various thresholds. In contrast, other metrics such as Precision, Recall and F1-score depend on choosing a threshold to determine whether an instance is anomalous or normal. Consequently, different thresholds can result in different values. Therefore, we employ ROC AUC and PRC AUC to evaluate and compare the different log anomaly detection methods. PRC AUC is also known as Average Precision (AP). For both ROC AUC and PRC AUC, values closer to 1 indicate better performance.

2.6.2 Model Implementation and Configuration

Traditional machine learning based approaches—such as PCA, OCSVM, iForest, and HBOS—usually first transform logs into log event count vectors, and then apply traditional anomaly detection techniques to identify anomalies. For these methods, we utilize their open-source implementations provided in PyOD [107]. Meanwhile, for deep learning methods DeepLog, LogAnomaly, and AutoEncoder, we use their open-source implementations in Deep-Loglizer [108]. For these methods, we use their default hyperparameter values.

Table 2.3: Anomaly detection accuracy on five benchmark datasets for *Logs2Graphs* and its eight competitors. AP and RC denote Average Precision and ROC AUC, respectively. HDFS, BGL, and Thunderbird have been downsampled to 10,000 graphs each while maintaining the original anomaly rates. For each method on each dataset, to mitigate potential biases arising from randomness, we conducted ten experimental runs with varying random seeds and report the average values along with standard deviations of AP and RC. Moreover, we highlight the best results with **bold** and the runner-up with underline.

Method	HDFS		Hadoop		BGL		Spirit		Thunderbird	
	AP	RC	AP	RC	AP	RC	AP	RC	AP	RC
PCA	0.91±0.03	**1.0**±0.00	0.84±0.00	0.52±0.00	0.73±0.01	0.82±0.00	0.31±0.00	0.19±0.00	0.11±0.00	0.34±0.01
OCSVM	0.18±0.01	0.88±0.01	0.83±0.00	0.45±0.00	0.47±0.00	0.47±0.01	0.34±0.00	0.29±0.00	0.12±0.00	0.45±0.01
IForest	0.73±0.04	0.97±0.01	0.85±0.01	0.55±0.01	0.79±0.01	0.83±0.01	0.32±0.03	0.23±0.02	0.11±0.01	0.24±0.10
HBOS	0.74±0.04	0.99±0.00	0.84±0.00	0.50±0.00	0.84±0.02	0.87±0.03	0.35±0.00	0.22±0.00	0.15±0.01	0.29±0.05
DeepLog	**0.92**±0.07	0.97±0.04	**0.96**±0.00	0.47±0.00	0.89±0.00	0.72±0.00	0.99±0.00	0.97±0.00	0.91±0.01	0.96±0.00
LogAnomaly	0.89±0.09	0.95±0.05	**0.96**±0.00	0.47±0.00	0.89±0.00	0.72±0.00	0.99±0.00	0.97±0.00	0.90±0.01	0.96±0.00
AutoEncoder	0.71±0.03	0.84±0.01	**0.96**±0.00	0.52±0.00	0.91±0.01	0.79±0.02	0.96±0.00	0.92±0.01	0.44±0.02	0.46±0.05
GLAM	0.78±0.08	0.89±0.04	0.95±0.00	**0.61**±0.00	0.94±0.02	0.90±0.03	0.93±0.00	0.91±0.00	0.75±0.02	0.85±0.01
Logs2Graphs	0.87±0.04	0.91±0.02	0.95±0.00	0.59±0.00	**0.96**±0.01	**0.93**±0.01	**1.0**±0.00	**1.0**±0.00	**0.99**±0.00	**1.0**±0.00

For all deep learning based methods, the experimental design adopted in this study follows a train/validation/test strategy with a distribution of 70% : 5% : 25% for normal instances. Specifically, the model was trained using 70% of normal instances, while 5% of normal instances and an equal number of abnormal instances were employed for validation (i.e., hyperparameter tuning). The remaining 25% of normal instances and the remaining abnormal instances were used for testing. Table 2.2 summarizes the hyperparameters involved in OCDiGCN as well as their recommended values.

We implemented and ran all algorithms in Python 3.8 (using PyTorch [109] and PyTorch Geometric [110] libraries when applicable), on a workstation equipped with an Intel i7-11700KF CPU and Nvidia RTX3070 GPU.

For reproducibility, all code and datasets are released on GitHub.[1]

2.6.3 Comparison to the State Of The Art

We first compare *Logs*2*Graphs* to the state of the art. We have the following main observations according to the results in Table 2.3:

- In terms of ROC AUC, *Logs2Graphs* achieves the best performance against its competitors on three out of five datasets. Particularly, *Logs2Graphs* outperforms the closet competitor on BGL with 9.6% and delivers remarkable results (i.e., an ROC AUC larger than 0.99) on Spirit and Thunderbird. Similar observations can be made for Average Precision.

[1] https://github.com/ZhongLIFR/Logs2Graph

- Deep learning based methods generally outperform the traditional machine learning based methods. One possible reason is that traditional machine learning based methods only leverage log event count vectors as input, which makes them unable to capture and exploit sequential relationships between log events and the semantics of the log templates.

- The performance of (not-graph-based) deep learning methods is often inferior to that of *Log2Graphs* on the more complex datasets, i.e., BGL, Spirit, and Thunderbird, which all contain hundreds or even thousands of log templates. This suggests that LSTM-based models may not be well suited for logs with a large number of log templates. One possible reason is that the test dataset contains many unprecedented log templates, namely log templates that are not present in the training dataset.

- In terms of ROC AUC score, all methods except for OCSVM and AutoEncoder achieve impressive results (with $RC > 0.91$) on HDFS. One possible reason is that HDFS is a relatively simple log dataset that contains only 48 log templates. Concerning AP, PCA and LSTM-based DeepLog achieve impressive results (with $AP > 0.89$) on HDFS. Meanwhile, *Logs2Graphs* obtains a competitive performance (with $AP = 0.87$) on HDFS.

2.6.4 Directed vs. Undirected Graphs

To investigate the practical added value of using *directed* log graphs as opposed to *undirected* log graphs, we convert the logs to attributed, undirected, and edge-weighted graphs, and apply GLAM [79], a graph-level anomaly detection method for undirected graphs. We use the same graph construction method as for *Logs2Graphs*, except that we use undirected edges. Similar to our method, GLAM also couples the graph representation learning and anomaly detection objectives by optimizing a single SVDD objective. The key difference with OCDiGCN is that GLAM leverages GIN [73], which can only tackle undirected graphs, while OCDiGCN utilizes DiGCN [72] that is especially designed for directed graphs.

The results in Table 2.3 indicate that GLAM's detection performance is comparable to that of most competitors. However, it consistently underperforms on all datasets, except for Hadoop, when compared to *Logs2Graphs*. Given that the directed vs undirected representation of the log graphs is the key difference between the methods, a plausible explanation is that directed graphs have the capability to retain the

temporal sequencing of log events, whereas undirected graphs lack this ability. Consequently, GLAM may encounter difficulties in detecting sequential anomalies and is outperformed by *Logs2Graphs*.

2.6.5 Node Labels vs. Node Attributes

To investigate the importance of using semantic embeddings of log events as node attributes, we replace the node semantic attributes with one-hot-encoding of node labels (i.e., using an integer to represent a log event). The performance comparisons in terms of ROC AUC for Logs2Graphs are depicted in Figure 2.3, which shows that using semantic embeddings is always superior to using node labels. Particularly, it can lead to a substantial performance improvement on the Hadoop, Spirit and HDFS datasets. The PRC AUC results show a similar behavior and thus are omitted.

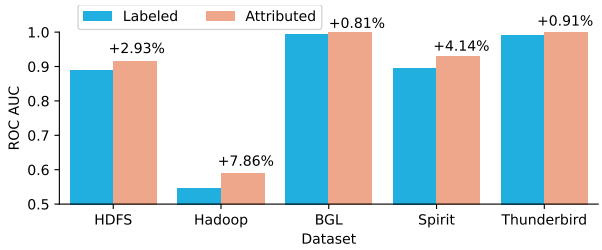

Figure 2.3: The comparative performance analysis of Logs2Graphs, measured by ROC AUC, demonstrating the distinction between utilizing node semantic attributes and node labels.

2.6.6 Robustness to Contamination

To investigate the robustness of Logs2Graphs when the training dataset is contaminated (namely integrating anomalous graphs in training data), we report its performance in terms of ROC AUC under a wide range of contamination levels. Figure 2.4 shows that the performance of Logs2Graphs decreases with an increase of contamination in the training data. The PRC AUC results show a similar behavior and thus are omitted. Hence, it is important to ensure that the training data contains only normal graphs (or with a very low proportion of anomalies).

2.6.7 Ability to Detect Structural Anomalies and Recognize Unseen Normal Instances

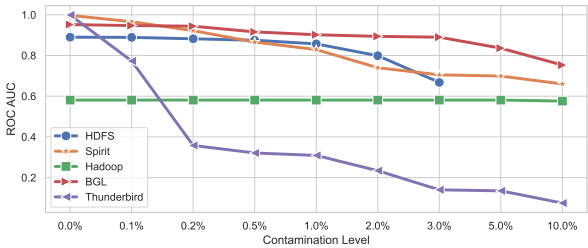

Figure 2.4: ROC AUC results of Logs2Graphs w.r.t. a wide range of contamination levels. Results are averaged over 10 runs. Particularly, HDFS contains only 3% anomalies and thus results at 5% and 10% are not available.

To showcase the effectiveness of different neural networks in detecting structural anomalies, we synthetically generate normal and anomalous directed graphs as shown in Figure 2.5. As Deeplog, LogAnomaly and AutoEncoder require log sequences as inputs, we convert directed graphs into sequences by sequentially presenting the endpoints pair of each edge. Moreover, for GLAM we convert directed graphs into undirected graphs by turning each directed edge into an undirected edge.

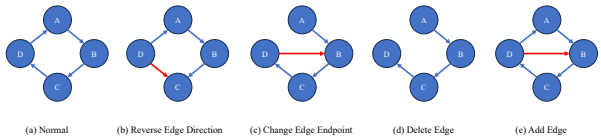

Figure 2.5: Synthetic generation of normal (10000) and structurally anomalous (200 each) graphs.

Moreover, to investigate their capability of recognising unseen but structurally equivalent normal instances, we generate the following normal log sequences based on the synthetic normal graph as training data: $A \to B \to C \to D \to A$ (1000), $B \to C \to D \to A \to B$ (1000) and $C \to D \to A \to B \to C$ (1000), and the following as test dataset: $D \to A \to B \to C \to D$ (1000).

The results in Table 2.4 indicate that Logs2Graphs, Deeplog and LogAnomaly can effectively detect structural anomalies, while AutoEncoder and GLAM fail in some cases. However, log sequences based methods, namely Deeplog, LogAnomaly and AutoEncoder, can lead to high false positive rates due to their inability of recognizing unseen but structurally equivalent normal instances.

Table 2.4: ROC AUC results (higher is better) of detecting structural anomalies and False Positive Rate (lower is better) of recognizing unseen normal instances. S1: Reverse Edge Direction; S2: Change Edge Endpoint; S3: Delete Edge; S4: Add Edge; N1: Unseen normal instances.

Case	Deeplog	LogAnomaly	AutoEncoder	GLAM	Ours
S1 (ROC)	1.0	1.0	0.0	0.0	1.0
S2 (ROC)	1.0	1.0	0.50	1.0	1.0
S3 (ROC)	1.0	1.0	1.0	1.0	1.0
S4 (ROC)	1.0	1.0	1.0	1.0	1.0
N1 (FPR)	100%	100%	100%	0%	0%

2.6.8 Anomaly Explanation

Figure 2.6 provides an example of log anomaly explanation with the HDFS dataset. For each detected anomalous log graph (namely a group of logs), we first quantify the importance of nodes according to the description in Chapter 2.5.3. Next, we visualize the anomalous graph by assigning darker shade of red to more important nodes. In this example, the node "WriteBlock(WithException)" contributes the most to the anomaly score of an anomalous log group and thus is highlighted in red.

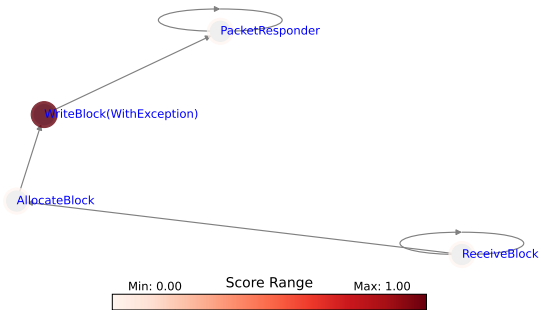

Figure 2.6: Example of anomaly explanation with HDFS (the log event templates are simplified for better visualization).

2.6.9 Sensitivity Analysis

We examine the effects of three hyperparameters in OCDiGCN on the detection performance.

The Number of Convolutional Layers: L is a potentially important parameter as it determines how many convolutional layers to use in OCDiGCN. Figure 2.7 (top

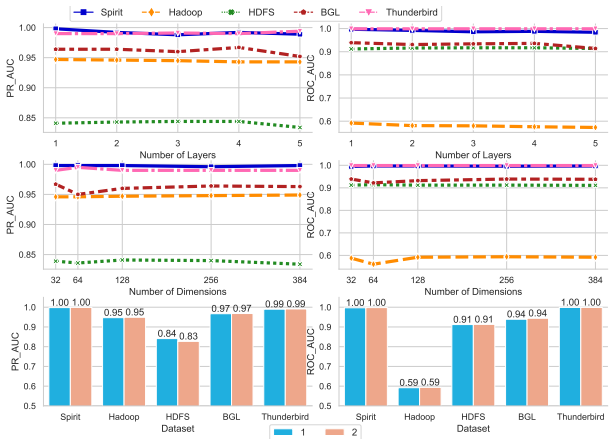

Figure 2.7: The effects of the number of layers (top row), the embedding dimensions (middle row) and the proximity parameter (bottom row) on AP (left column) and ROC AUC (right column).

row) depicts PRC AUC and ROC AUC for the five benchmark datasets when L is varied from 1 to 5. We found that $L = 1$ yields consistently good performance. As the value of L is increased, there is only a slight enhancement in the resulting performance or even degradation, while the associated computational burden increases substantially. We thus recommend to set $L = 1$.

The Embedding Dimension d: From Table 2.7 (middle row) , one can see that $d = 128$ yields good performance on Spirit, Hadoop, HDFS and Thunderbird, while further increasing d obtains negligible performance improvement or even degradation. However, an increase of d on BGL leads to substantially better performance. One possible reason is that BGL is a complex dataset wherein anomalies and normal instances are not easily separable on lower dimensions.

The Proximity Parameter k: As this parameter increases, a node can gain more information from its further neighbors. Figure 2.7 (bottom row) contrasts the detection performance when k is set to 1 and 2, respectively. Particularly, we construct one Inception Block when $k = 2$, using concatenation to fuse the results.

We observe that there is no large improvement in performance when using a value of $k = 2$ in comparison to $k = 1$. It is important to recognize that a node exhibits 0th-order proximity with itself and 1st-order proximity with its immediately connected neighbors. If $k = 2$, a node can directly aggregate information from its 2nd-order neighbors. As described in Table 1, graphs generated from logs usually contain a limited number of nodes, varying to 6 to 34. Therefore, there is no need to utilize the

Inception Block, which was originally designed to handle large graphs in [72].

2.6.10 Runtime Analysis

Traditional machine learning methods, including PCA, OCSVM, IForest and HBOS, usually perform log anomaly detection in a transductive way. In other words, they require the complete dataset beforehand and do not follow a train-and-test strategy. In contrast, neural network based methods, such as DeepLog, LogAnomaly, AutoEncoder, and Logs2Graphs, perform log anomaly detection in an inductive manner, namely following a train-and-test strategy.

Figure 2.8 shows that most computational time demanded by *Logs2Graphs* is allocated towards the graph generation phase. In contrast, the training and testing phases require a minimal time budget. The graph generation phase can be amenable to parallelization though, thereby potentially reducing the overall processing time. As a result, *Logs2Graphs* shows great promise in performing online log anomaly detection. Meanwhile, other neural networks based models—such as DeepLog, LogAnomaly, and AutoEncoder—demand considerably more time for the training and testing phases.

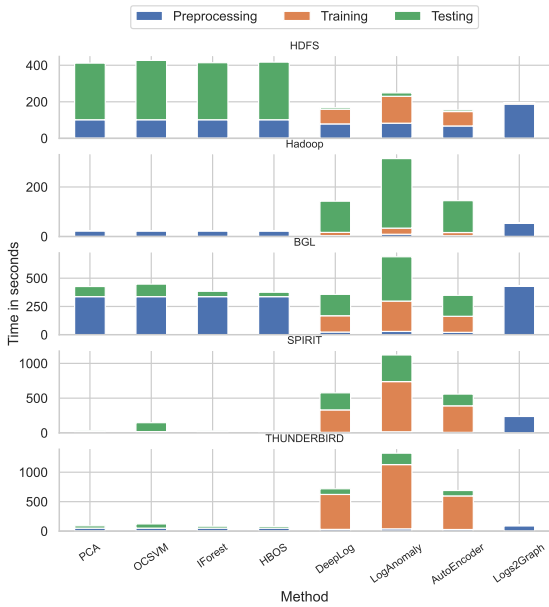

Figure 2.8: Runtime for all eight methods on all datasets, wherein HDFS, BGL, and Thunderbird have been downsampled to 10,000 graphs. Runetimes are averaged over 10 repetitions. We report the training time per epoch for neural network based methods.

Table 2.5: An example event log of lithography systems. All values are fictional. This table is taken from [56], and please refer to the original paper for more information.

M	Code	Level	Detail	DateTime
1	AA-BBBB	Low	descr	2020-01-01 00:00:01
1	CC-DDDD	Med	descr	2020-01-01 00:00:01
1	AA-BBBB	Low	descr	2020-01-01 00:01:00
1	AA-BBBB	Low	descr	2020-01-01 00:02:03
1	EE-FFFF	High	descr	2020-01-01 00:05:00
⋮	⋮	⋮	⋮	⋮

2.7 An Ongoing Case Study to Lithography Systems

In this section, we present an ongoing case study in which we are applying *Logs2Graphs* to lithography systems, aiming to enhance fault detection and analysis in the wafer transfer subsystem.

2.7.1 Background

A lithography system is an intricate apparatus designed for the fabrication of chips. Specifically, a lithography system usually consists of several main subsystems, including the objective lens subsystem, the table subsystem, the mask table subsystem, the mask transfer subsystem, the light source subsystem, the exposure subsystem, and the wafer transfer subsystem [111]. Particularly, the wafer transfer subsystem, which plays a crucial role in the precision of semiconductor fabrication, facilitates the movement of silicon wafers between the track and the wafer stage. This subsystem is essential for operational efficiency during lithography processes, and it typically comprises a load robot and an unload robot. However, these robots are susceptible to malfunctions during production, potentially impacting the fabrication precision.

To optimize productivity by reducing machine downtime, it is essential to detect and analyze the robot faults in an automated and intelligent manner. As shown in Table 2.5, a lithography machine is equipped with an information system that can record all critical events activated during its operation into logs. The event logs contain critical information for fault detection and analysis, motivating us to apply log-based anomaly detection methods to detect faults.

2.7.2 Deployment of *Logs2Graphs*

As shown in Figure 2.1, the deployment of *Logs2Graphs* consists of the following main steps:

1) Log Collection: we use 25 real-world datasets provided by our industrial partners to test our method. Specifically, the 25 datasets were generated by 20 different test machines, of which 16 machines each generated one dataset and the remaining 4 machines each generated at least 2 datasets. Please refer to Table 3 in [56] for more detail. Besides, more data will be collected by our industrial partners;

2) Log Parsing: we use Drain [92] for this task;

3) Log Grouping (Ongoing): we dedicate much efforts in this step as lithography system is very complicated in the following aspects: first, the information system records event logs from all subsystems while some events are irrelevant to the wafer transfer subsystem; second, rather than simply using time as log identifier, we must use domain knowledge to construct more advanced log identifiers using the information provided in the "Detail" column;

4) Graph Construction (Ongoing): we will construct directed, attribute and edge-weighted graphs;

5) Graph Representation Learning and Anomaly Detection (Ongoing): we will utilize OCDiGCN;

6) Anomaly Explanation and Analysis (Ongoing): we follow the methods given in Chapter 2.5.3.

At the request of our industrial partners, the datasets, code, and detailed results of this case-study cannot be published. Despite this, *Logs2Graphs* is a promising tool for intelligent fault detection and analysis when combined with domain expertise.

2.8 Threats to Validity

We have discerned several factors that may pose a threat to the validity of our findings.

Limited Datasets. Our experimental protocol entails utilizing five publicly available log datasets, which have been commonly employed in prior research on log-based anomaly detection. However, it is important to acknowledge that these datasets may not fully encapsulate the entirety of log data characteristics. To address this limitation,

our future work will conduct experiments on additional datasets, particularly those derived from industrial settings, in order to encompass a broader range of real-world scenarios.

Limited Competitors. This study focuses solely on the experimental evaluation of eight competing models, which are considered representative and possess publicly accessible source code. However, it is worth noting that certain models such as GLAD-PAW did not disclose their source code and it requires non-trivial efforts to re-implement these models. Moreover, certain models such as CODEtect require several months to conduct the experiments on our limited computing resources. For these reasons, we exclude them from our present evaluation. In subsequent endeavors, we intend to re-implement certain models and attain more computing resources to test more models.

Purity of Training Data. The purity of training data is usually hard to guarantee in practical scenarios. Although Logs2Graphs is shown to be robust to very small contamination in the training data, it is critical to improve the model robustness by using techniques such as adversarial training [112] in the future.

Graph Construction. The graph construction process, especially regarding the establishment of edges and assigning edge weights, adheres to a rule based on connecting consecutive log events. However, this rule may be considered overly simplistic in certain scenarios. Therefore, application-specific techniques will be explored to construct graphs in the future.

2.9 Conclusions

We introduced *Logs2Graphs*, a new approach for unsupervised log anomaly detection. It first converts log files to attributed, directed, and edge-weighted graphs, translating the problem to an instance of graph-level anomaly detection. Next, this problem is solved by OCDiGCN, a novel method based on graph neural networks that performs graph representation learning and graph-level anomaly detection in an end-to-end manner. Important properties of OCDiGCN include that it can deal with directed graphs and do unsupervised learning.

Extensive results on five benchmark datasets reveal that *Logs2Graphs* is at least comparable to and often outperforms state-of-the-art log anomaly detection methods such as DeepLog and LogAnomaly. Furthermore, a comparison to a similar method for graph-level anomaly detection on *undirected* graphs demonstrates that using directed log graphs leads to better detection accuracy in practice.