Substructure-aware Log Anomaly Detection

Yanni Tang
The University of Auckland
New Zealand
ytan370@aucklanduni.ac.nz

Zhuoxing Zhang
The University of Auckland
New Zealand
zzha969@aucklanduni.ac.nz

Kaiqi Zhao*
The University of Auckland
New Zealand
kaiqi.zhao@auckland.ac.nz

Lanting Fang*
Southeast University
China
lantingf@outlook.com

Zhenhua Li
Southwest University
China
lzh20231947@email.swu.edu.cn

Wu Chen*
Southwest University
China
chenwu@swu.edu.cn

ABSTRACT

System logs, recording critical information about system operations, serve as indispensable tools for system anomaly detection. Graph-based methods have demonstrated superior performance compared to other methods in capturing the interdependencies of log events. However, existing methods often neglect the complex substructure patterns of nodes within log graphs, making it challenging to capture the subtle alteration in event type, structure, and the location of exceptions that indicate node anomalies. To address this limitation, this paper proposes a novel framework called Substructure-aware Log Anomaly Detection at Code File Level (SLAD). It first introduces a Monte Carlo Tree Search strategy tailored specifically for log anomaly detection to discover representative substructures. Then, SLAD incorporates a substructure distillation way to enhance the efficiency of anomaly inference based on the representative substructures. After that, we introduce a soft pruning to obtain key substructure for nodes. Experimental results show SLAD outperforms all baselines. Particularly, SLAD demonstrates at least 15 times faster than substructure-based graph learning methods in anomaly inference.

PVLDB Reference Format:
Yanni Tang, Zhuoxing Zhang, Kaiqi Zhao, Lanting Fang, Zhenhua Li, and Wu Chen. Substructure-aware Log Anomaly Detection. PVLDB, 18(2): 213 - 225, 2024.
doi:10.14778/3705829.3705840

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZhuoxingZhang/SLAD.

1 INTRODUCTION

System logs are a valuable resource in system monitoring and troubleshooting, as they encompass the interdependencies among various components and operational processes within the system [43]. Log anomaly detection aims to identify anomalies that deviate from the expected patterns of normal behavior through log analysis [11]. Particularly, anomalies in system code files can lead to system crashes or other unpredictable behaviors, and these anomalies are typically documented in the logs. Detecting these anomalies through logs can provide crucial insights into faster debugging during the system development and maintenance phases. This capability is essential for maintaining system stability and security. Therefore, efficiently identifying anomalous code files from system logs is a research direction worthy of exploration.

Early research in log anomaly detection relies on either rules or performance metrics predefined by domain experts [17, 21, 35]. However, these manually designed metrics fail to capture the complex relationships within logs [10, 13, 14, 16, 32]. Sequence-based models have been proposed to capture the temporal dependencies of log events by formulating log events as sequences [6, 8, 9, 19, 34]. Despite exhibiting some effectiveness, these methods often struggle to capture the structural relationships inherent within log events. Recent advancements [29, 30, 39] have demonstrated that graph-based methods offer a promising solution to capture the interrelations among log events, where log events serve as nodes, and the temporal dependencies between events are regarded as edges. These methods employ graph representation learning techniques to automatically learn vector representations of log events that encode the log event features and their correlations. The understanding of the interactions between log events has enabled graph-based methods to achieve better anomaly detection accuracy.

However, on log anomaly detection, the majority of work focuses on detecting whether a log segment is anomalous, with relatively little attention paid to the *code file anomaly detection from system logs*. To fill this research gap, our work focuses on the detection of anomalous code files from system logs. Existing studies solely capture the relationships between log events instead of code files. As shown in Fig. 1, there are two segments of log messages, where Logs (A) is an anomalous log segment and Logs (B) is a normal log segment. Based on existing methods, these two log segments would form identical sequences of log events (Fig. 1a) or log event graphs (Fig. 1b). During system execution, an anomaly occurs in code file C_2, leading to an unsuccessful invocation of C_3 by C_2. As a result, C_1 invokes C_3 instead, thereby forming the anomalous log segments in Logs (A). In this case, such subtle variations in invocation relationships of code files cannot be distinguished by mining relationships between log events, leading to ineffective

*Corresponding authors.

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705840

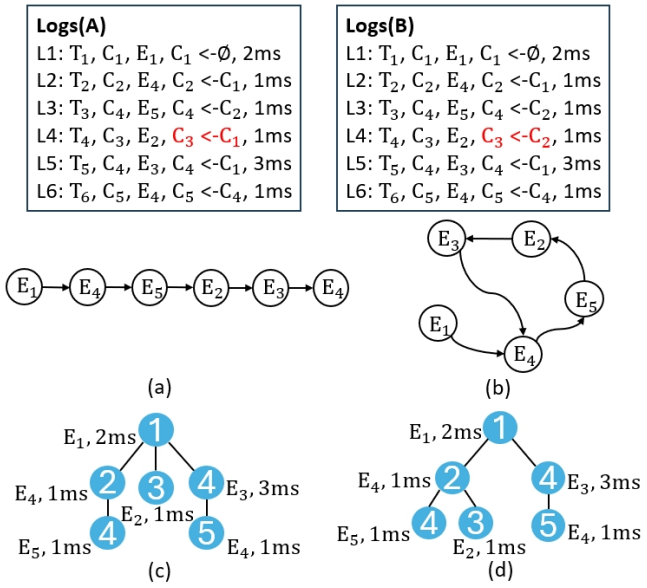


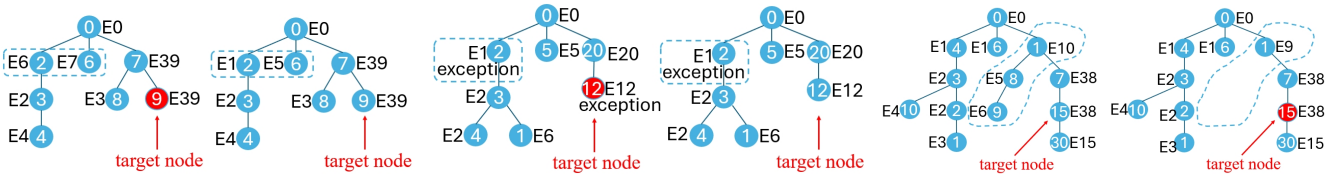
Figure 1: Different log representation ways for two log segments, anomalous Logs (A) and normal Logs (B). The two log segments only differ from one invocation as colored by red. T_i, C_i, E_i refer to time, code files, and event types, respectively.

detection of such anomalies. So, existing methods cannot perform code file-level log anomaly detection.

To address the problem of detecting anomalous code files from system logs, we propose a log graph, inspired by the fact that system logs contain essential information about the code files and their invocation relationships. Unlike the log event graph constructed in the existing work, each node in our log graph denotes a log message, which is associated with a code file. Each edge denotes the invocation (call) relationship between code files. To this end, we formulate the problem as node classification on a log graph. Following the example in Fig. 1, two log graphs Fig. 1c and Fig. 1d are constructed from Logs (A) and (B), respectively. In our log graphs, each node corresponds to a code file, associated with an event type, duration time, and exception information as the node's attributes. Edges are the invocation relationships between code files. For instance, the node with a label "3" in Fig. 1c refers to code file C_3, whose attributes include event type "E_2" and duration time "1ms" in the log message L4 of Logs (A). The edge between node 1 and node 3 indicates the invocation $C_3 \leftarrow C_1$. Our log graph encloses essential runtime information and the invocation relations between code files, thus facilitating the discovery of complex system behaviors for anomalous code file detection.

The main challenge of anomaly detection within the log graph lies in understanding the complex invocation relationships between code files and discerning how they correlate with normal or anomalous behaviors. Therefore, we investigated various log graphs and summarized three observations that are essential for the detection of code file anomalies from system logs: 1) event type alteration, 2) exception information location, and 3) structure alteration. Fig. 2 illustrates the three observations where the target nodes are normal (blue) and anomalous (red), respectively. The differences between the normal/abnormal graphs are highlighted with dashed lines.

- **O1: Event type alteration**. Fig. 2a shows the event type alterations on other nodes caused by the anomaly of the target node 9. Determining whether node 9 is anomalous only depends on the event types from nodes 2 and 6, which are in the different invocation paths with the target node. For example, in a forum system, node 9 is a code file responsible for accessing the database. Due to a bug with an SQL query in node 9, the system, which intended to query the user's liked posts records, instead retrieves the user's commented posts records. As a result, node 9 returns an abnormal return value, which is then received by node 2 and node 6. Node 2 is supposed to handle records of the user's liked posts (generating the event type E1), but it ends up processing records of the user's commented posts instead (generating the event type E6), thereby changing the log event type. Consequently, node 6, responsible for displaying the collected information, receives the incorrect data, causing its log event type also to change, from the event type E5 to E7. This situation stems from the unique characteristics of software systems. When a code file encounters an anomaly, it generates an abnormal return value, which may propagate backward along the invocation path to some code files, resulting in executing different code blocks to display different log event types. Therefore, the detection of node anomalies may rely on nodes with event type alteration.
- **O2: Exception information location** (Fig. 2b). Exception is often a signal of underlying issues occurring within the system. The location of the exception may indicate the root causes of anomalies. Fig. 2b demonstrates that in the same invocation environment, if only node 2 reports exception information, then no nodes in this graph are anomalous. However, when nodes 2 and 12 all report exception information, node 12 is the root cause node. For example, in the forum system, node 2 is responsible for accessing user information. When it reports an exception message like "zero value happens", it does not necessarily indicate an anomaly with the corresponding code file, such as the absence of queried user information. On the other hand, node 12 is responsible for calculating the average view count of user posts. If node 12 has a bug that prevents it from correctly handling cases where the user's post count is zero, it might encounter an error and report an exception message "divided by zero". This observation indicates that the location of the exception information is crucial for detecting anomalous nodes.
- **O3: Structure alteration** (Fig. 2c). A node anomaly could lead to the propagation of anomalous information along the invocation path, altering the invocation in other nodes. As illustrated in Fig. 2c, when node 15 experiences an anomaly, it results in an alteration to the event and local invocation in node 1 (nodes 8 and 9 disappear). For example, in the forum system, when a user posts a message, node 15 is responsible for validating the user's identity and permissions. If a bug occurs in node 15 that checks the user's permissions, it will propagate the user authentication failure message to node 1. As a result, node 1 will be unable to proceed with posting the message, and nodes 8 and 9 will not be invoked. These structural alterations in certain nodes may provide valuable clues for detecting anomalies.



(a) Event type alteration (b) Exception information location (c) Structure alteration

Figure 2: Observations on log graphs. Each node in the log graph corresponds to a log message associated with the code file that generates the message. Edges represent invocation relationships between code files of the log messages. Ei denotes the i-th log event type. Red nodes are the anomalous nodes while blue ones refer to the normal nodes.

From the above observations, node anomalies may not necessarily manifest through changes within the node itself but rather through changes occurring in other nodes within the graph. Consequently, anomalies of nodes in a log graph typically manifest within specific *substructures* (subgraphs) of the log graphs, including changes in the event types of certain nodes, the location of exception information, and structure. Further, the essential substructures of anomalous and normal nodes often exhibit different sizes and structural forms. For example, in Fig. 2a, determining whether target node 9 is anomalous depends on the substructures formed by nodes 0, 2, 6, 7, and 9. However, in Fig. 2b and Fig. 2c, the key substructures consist of nodes 0, 2, 5, 12, 20 and nodes 1, 7, 15, respectively. Thus, a natural question arises: how to effectively capture these important changes in substructures that the size and form of substructures may vary.

Recently, graph neural networks (GNNs) have been exploited to discover the local structural information of nodes. However, existing GNNs only capture fixed-hop neighborhood information of nodes, which is not capable of capturing changes in substructures of varying sizes and structural forms. Even though some random walk-based methods have been proposed to identify substructures of various sizes and structural forms [25, 38, 42], these methods 1) require online walks during the anomaly inference phase, leading to significant time consumption; 2) cannot identify specific substructures within software systems as illustrated in Fig. 2.

To this end, we introduce a novel framework to detect anomalous code files from system logs, namely Substructure-aware Log Anomaly Detection at Code File Level (SLAD). The core concept of SLAD is to **automatically discover representative substructures** for both anomalous and normal nodes. During the training phase, SLAD acquires the representative substructure candidates through a Monte Carlo Tree Search (MCTS) strategy designed according to domain knowledge in software systems and obtains their embeddings using graph neural networks. To avoid substructure exploration during anomaly inference, we design a knowledge distillation method to obtain a fixed set of distilled substructure representations from the representative substructure embeddings. Subsequently, based on distilled substructure representations, SLAD conducts soft pruning to discover key substructures from an input log graph, enabling accurate anomaly detection. Our framework has three salient features: 1) it can discover representative substructures of various sizes and structural forms that existing graph-based methods cannot capture; 2) it eliminates the necessity of exploring substructures during the inference process via effective knowledge distillation, thereby reducing inference overhead; 3) it enhances anomaly detection accuracy through soft pruning with the distilled substructure representations. Our contributions are as follows.

- We propose a framework SLAD to detect code file anomalies from system logs. It introduces a novel Monte Carlo Tree Search strategy to discover representative substructures with domain knowledge for anomaly detection.
- To avoid substructure exploration and reduce overhead during inference, SLAD introduces a knowledge distillation method to obtain compact substructure representations.
- SLAD further devises a soft pruning method that integrates distilled substructure representations to obtain key substructures of an input log graph, thereby enhancing log anomaly detection.
- Extensive experiments on our datasets show that SLAD outperforms the state-of-the-art methods, improving the F1 score by at least 4% while achieving a reasonable inference running time.

2 PRELIMINARIES

System logs are records generated by computer systems that document various events and activities occurring within the system. Each log message is generated by a code file.

Definition 2.1 (Raw Log Message). A log message is a 6-tuple:

$$(file, traceID, event, invoInfo, cost, exception),$$

where $file \in \mathcal{F}$ denotes the code file that generates this log message; $traceID$ is the unique identifier for a request; $event \in \mathcal{ET}$ is a log event type that describes system behaviors; $invoInfo \in \mathcal{F}$ represents the upstream code file that invokes the current code file; $cost$ is the duration time; $exception \in \mathcal{EI}$ is the exception type.

For instance, a raw log message L2 in Fig. 3 can be represented by a 6-tuple ($file$="forum.UserManager", $traceID$="f119a251bfc149", $event$="get user id:* information", $invoInfo$="forum.UserService", $cost$=4 milliseconds, $exception$="invalid token"),

It is noteworthy that in software systems, the set of event types is fixed due to the system's functionalities and operations. For example, in a forum system, basic operations like article comments and user logins may be recorded, constituting a finite set of event types. Despite varying event descriptions from different user interactions, these events belong to the same event type. For instance, the event "get user id:17965 information" in log message L2 of Fig. 3

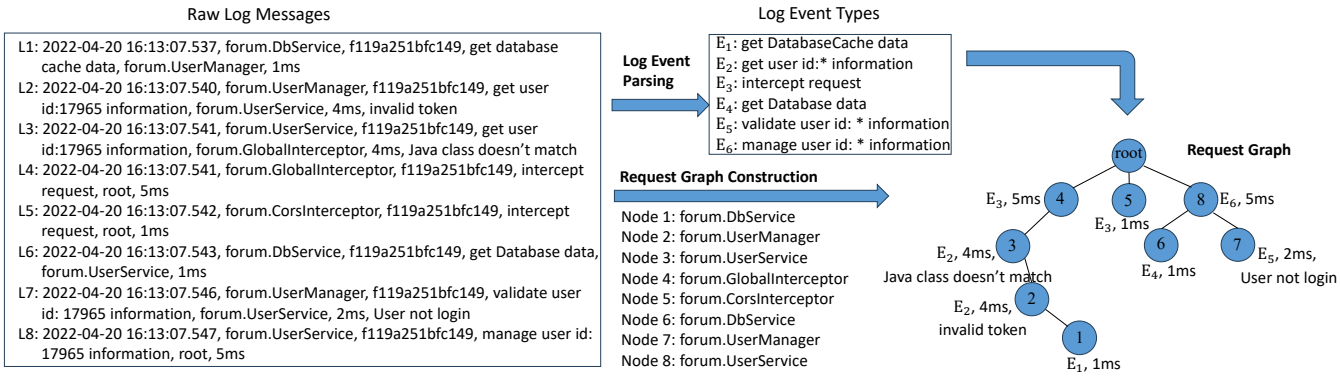


Figure 3: Request graph construction. It parses raw log messages to log event types and then constructs the request graph. In the request graph, each node corresponds to a log message associated with the code file that generates the message. Edges represent invocation relationships between code files of the log messages.

can be parsed as "get user id:* information" by discarding the specific ID information. In this way, all events related to retrieving user information can be categorized as the same type.

The interconnections between code files naturally create associations between the logs. Invocation information can effectively link discrete log events. As mentioned earlier, some anomalies can be manifested through the invocation relationships between code files. For example, anomalies may propagate between code files or appear in specific patterns, aiding issue identification. Therefore, we construct a request graph for each request's logs.

Definition 2.2 (Request Graph). A request graph is an attributed undirected graph $G = (V, E)$, where V and E are defined as follows:

- V corresponds to a set of log messages. Each $v \in V$ is a 4-tuple $(file, event, cost, exception)$ where $file \in \mathcal{F}$ represents the source code file that creates the current log message; $event \in \mathcal{ET}$ is a log event type produced by $file$; $cost \in \mathbb{R}$ denotes duration time of this log message and $exception \in \mathcal{EI}$ is the type of exception produced by $file$;
- E denotes a set of invocation relationships between the $file$ within the log messages ;

Fig. 3 illustrates the process of how a request graph is constructed from raw log messages. Request Graph G is a tree-like graph, that displays the process of program invocation. There exists only one root node $v_{root} \in V$ in G, and v_{root} is the starting point for the program execution. Let $Deg(v)$ be the degree of node v. If a node v has $Deg(v) = 1$, we refer to it as a leaf node v_{leaf}.

Definition 2.3 (Invocation Path). An invocation path is the shortest path from root node v_{root} to a leaf node v_{leaf}, denoted as

$$v_{root} \rightarrow ... \rightarrow v_i \rightarrow ... \rightarrow v_{leaf}.$$

For ease of presentation, we use $invo(v)$ to denote the set of invocation paths to which v belongs. For instance, in Fig. 3, the request graph contains 4 invocation paths: 1. root→4→3→2→1; 2.root→5; 3.root→8→6; 4.root→8→7. Node 8 belongs to 2 invocation paths.

Given the definition of request graph, we formulate the problem of anomalous code file detection from system logs as the binary classification of nodes in a request graph.

Definition 2.4 (Anomalous Code File Detection from System Logs). Given a request graph G, each node corresponds to a code file. The task of code file-level log anomaly detection is to predict the binary class labels $y_v \in \{0, 1\}$ for all nodes V in G, where $y_v = 0$ denotes a node is normal, otherwise anomalous.

3 METHODOLOGY

In this section, we introduce a novel substructure-aware log anomaly framework, SLAD. To the best of our knowledge, SLAD is the first to detect anomalous code files via mining representative substructures in log graphs.

3.1 Overall Framework

Fig. 4 depicts the overall framework of SLAD. The key design objective of SLAD is to capture representative substructures in the request graph for normal and anomalous nodes. Specifically, SLAD employs three novel components to achieve the design objective: (1) Substructure search via Monte Carlo Tree Search (MCTS), (2) Substructure distillation and (3) Soft pruning.

- Analysis of log request graphs indicates that *normal and anomalous nodes may be associated with specific substructure patterns*, including sets of nodes with event type alteration, exception information location, and structure alteration. However, identifying these substructures proves challenging due to their varied patterns and sizes. To address this, we design a Monte Carlo Tree Search strategy to discover representative substructures. This method involves pruning subgraphs to remove irrelevant nodes and preserve crucial substructure patterns to distinguish normal and anomalous nodes.
- To enhance anomaly detection and avoid the time consumption of substructure exploration during inference, we further propose a substructure distillation component. Considering different nodes may share certain similar substructures, this component aims to summarise the representative patterns of both normal and anomalous nodes. Specifically, it utilizes the representative substructure embeddings S to assist in the distillation process

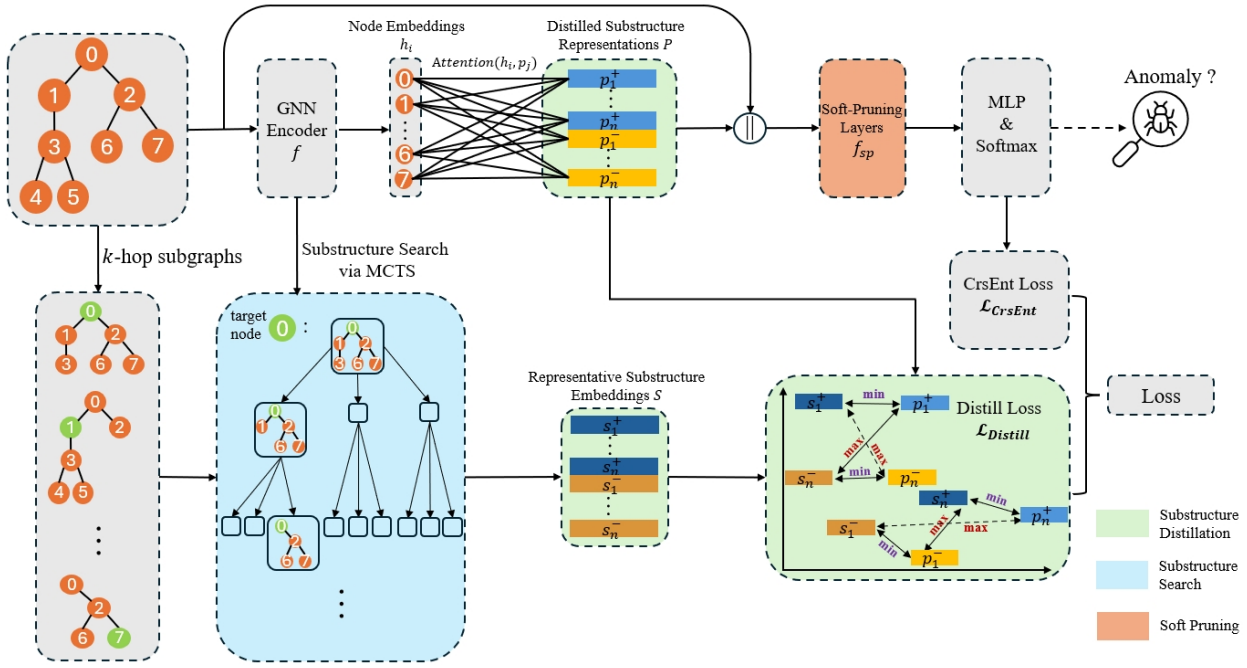


Figure 4: The framework of SLAD. The framework encompasses three novel components: substructure search via MCTS, substructure distillation, and substructure soft-pruning. "||" refers to concatenation operation.

for distilled substructure representations P, enabling them to acquire knowledge about representative substructures.

- To capture the key substructure of nodes, we introduce a soft pruning component. Based on the approximate substructure information of nodes, this component uses an attention mechanism to search important neighboring node information, thus achieving the goal of soft pruning.

In the training phase, with a training set of graphs $\{G_i\}_{i=1}^{N_G}$, each node $v_j \in G_i$ is associated with a class label $y_{v_j} \in \{0, 1\}$. SLAD first learns a graph encoder f, an initialization of the distilled substructure representations P and soft pruning layers f_{sp} through cross-entropy loss $\mathcal{L}_{CrsEnt}$ in the warm-up process. This step aims to learn a graph encoder f for obtaining substructure representation. The graph encoder f can adopt any GNN models as a backbone, e.g. graph transformer neural network (GTNN)[23]. Next, for each class $c \in \{0, 1\}$, it selects a specified number n of representative substructures via Monte Carlo Tree Search (Sec. 3.2) and uses GNN encoder f to obtain their embeddings S. After that, we design a distillation loss $\mathcal{L}_{Distill}$ to leverage S to further refine the distilled substructure representations P (Sec. 3.3). This allows P to obtain representative substructure information, which can be used directly during the inference phase to avoid substructure exploration on the fly, thus reducing the inference time.

During the inference phase, for a given node, SLAD obtains its embedding through the GNN encoder, then gets approximate substructure information with P through an attention mechanism. Following this, soft pruning layers (Sec. 3.4) are introduced to get key substructure information of the node. Finally, a multi-layer perception(MLP) with softmax is used to output the predicted label.

3.2 Substructure Search via MCTS

Due to inherent business rules and workflow, software systems often result in many graphs sharing similar substructures. Besides, our observations on the log request graphs show distinct substructures of normal and anomalous nodes. Therefore, this anomaly detection can be facilitated through mining representative substructures.

When capturing representative substructures, a straightforward method is to enumerate all substructures directly. However, this method is impractical due to the exponential number of possible substructures. An alternative is to use walk-based strategies to discover representative substructures, but these methods often struggle to capture the diverse substructure patterns inherent in software systems. The complex functional requirements of a system lead to a series of invocation substructures designed to accommodate specific business rules and workflow. These substructures vary in size and structure forms. Walk-based strategies are typically based on local neighborhoods, focusing on exploring the substructure of a graph through the adjacency relationships of nodes or edges. Moreover, their tendency to extensively traverse different parts of the graph introduces redundancy. This redundancy includes irrelevant information that is unrelated to the anomaly detection task, potentially complicating the analysis process.

Inspired by the remarkable achievements of Monte Carlo Tree Search (MCTS) in solving combinatorial problems [24], we propose an MCTS-based strategy for exploring representative substructures.

Definition 3.1 (Monte Carlo Tree). Given a decision problem with a finite state space $\mathcal{D}$ and corresponding actions that transit one state to the other $A : \mathcal{D} \to \mathcal{D}$, a Monte Carlo Tree is a tree structure $T = (\mathcal{N}, \mathcal{E})$:

- The set of nodes $\mathcal{N}$ contains all possible decision states, and each node $n \in \mathcal{N}$ corresponds to a state $D_n \in \mathcal{D}$.
- The set of edges $\mathcal{E}$ contains the connection relationships between nodes, with each edge $(n, n') \in \mathcal{E}$ representing taking an action from state D_n to reach state $D_{n'}$.

Monte Carlo Tree Search is to dynamically construct a Monte Carlo Tree by selecting the optimal actions based on the visit count and reward value of each tree node to balance the trade-off between exploration and exploitation [5]. Formally, the next state used for creating the $(i+1)$-th Monte Carlo Tree node is selected by:

$$D_{i+1} = \arg\max_{D \in A(D_i)} \left(\frac{Q(D)}{N(D)} + \alpha_1 R(D) \frac{\sqrt{\sum_m N(D_m)}}{1 + N(D)} \right), \tag{1}$$

where $N(D)$ is the number of occurrences of state D, $R(D)$ is the current reward of state D and $Q(D)$ is the accumulated reward of state D. The reward function $R(\cdot)$ is typically task-specific. The weight α_1 is a hyperparameter that plays a crucial role in balancing the trade-off between exploration and exploitation. Intuitively, the first term $\frac{Q(D)}{N(D)}$ is the average reward of state D. In the second term $\frac{\sqrt{\sum_m N(D_m)}}{1+N(D)}$, the more a state D is selected (quantified by $N(D)$), the less likely D will be selected.

In the anomaly detection problem, we regard a substructure (i.e., subgraph) originating from a target node v_t in the request graph G as a state in the Monte Carlo Tree, and the removal of a node in a substructure as an action. To perform MCTS, we first create the root node n_{root} of the Monte Carlo Tree with the k-hop subgraph of v_t in G. Subsequently, MCTS removes graph nodes from the k-hop subgraph iteratively, resulting in a sequence of substructures, i.e., tree nodes in the Monte Carlo Tree. The sequence of substructures forms a search path, which is also known as a rollout in MCTS. A rollout in MCTS terminates when the node number of the current substructure reaches a specific value. Fig. 5 illustrates a rollout in a Monte Carlo Tree, which terminates with a substructure of 4 nodes.

The effectiveness of an MCTS algorithm depends on the policy of selecting the optimal action (i.e., the graph node to remove) as illustrated in Eq. 1, to expand the Monte Carlo Tree. Motivated by the operational characteristics of the system, we re-design Eq. 1 for our task with the following considerations:

- During the MCTS process, we aim to find a substructure that distinguishes normal and anomalous nodes. Therefore, the embedding of a representative substructure for normal nodes is expected to be close to the embeddings of normal nodes and vice versa. Based on this, we design the reward function for a state D^{v_t} (i.e., a substructure) of a Monte Carlo Tree node as:

$$R(D^{v_t}) = \frac{1}{\mathcal{K}} \sum_{j=1}^{\mathcal{K}} w * log\left(\frac{\left\| h_{D^{v_t}} - h_j \right\|_2^2 + 1}{\left\| h_{D^{v_t}} - h_j \right\|_2^2 + \mu} \right), \tag{2}$$

where $w = \begin{cases} 1, y_{v_t} = y_{v_j} \\ -1, y_{v_t} \neq y_{v_j} \end{cases}$ refers to the weight that encourages label consistency between $h_{D^{v_t}}$ and its $\mathcal{K}$-nearest node embeddings in the training set, and penalizes cases where they are inconsistent. The embedding $h_j \in KNN(h_{D^{v_t}})$ denotes one of the $\mathcal{K}$-nearest node embeddings to $h_{D^{v_t}}$. We add 1 and a small constant $\mu << 1$ to the numerator and denominator, respectively, within the logarithm to ensure the reward is always positive.

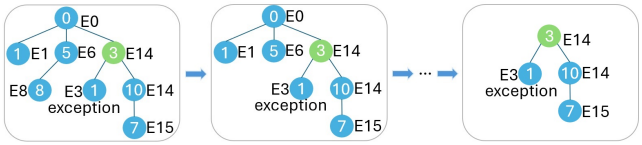


Figure 5: An illustration of rollout in MCT.

- Nodes with exception information are often internally connected to the anomalous nodes. Analyzing exception information facilitates a more straightforward identification of anomalous nodes. Motivated by this, we design a cost function to penalize the substructure that does not include graph nodes with exception information:

$$E(D^{v_t}) = \frac{|\{v | v.exception \neq null, v \in D^{v_t}\}|}{|\mathcal{N}_k(v_t)|}, \tag{3}$$

where $\mathcal{N}_k(v_t)$ denotes the set of graph nodes in the k-hop neighbors of the target graph node v_t.

- An invocation path reflects the program's execution path, with each node in the invocation path closely connected to others. If a node encounters an anomaly, the anomalous information might spread along its invocation path, potentially causing nodes along the path to exhibit anomalous behavior. This phenomenon offers valuable clues to pinpoint the node anomalies. Based on this, we design a cost function to preserve graph nodes in the invocation path of the target node v_t as much as possible:

$$C(D^{v_t}) = \frac{|\{v | invo(v) \cap invo(v_t) \neq \emptyset, v \in D^{v_t}\}|}{|\mathcal{N}_k(v_t)|} \tag{4}$$

By integrating Equations 2, 3 and 4, we obtain the objective function for finding the optimal pruning action given a state D^{v_t}:

$$D_{i+1}^{v_t} = \arg\max_{D^{v_t} \in A(D_i^{v_t})} \left(\frac{Q(D^{v_t})}{N(D^{v_t})} + \alpha_1 R(D^{v_t}) \frac{\sqrt{\sum_m N(D_m^{v_t})}}{1 + N(D^{v_t})} + \alpha_2 E(D^{v_t}) + \alpha_3 C(D^{v_t}) \right), \tag{5}$$

where D^{v_t} is a possible selection of the resulting substructures for the target node v_t by removing a certain node from the substructure $D_i^{v_t}$ at the i-th Monte Carlo Tree node in the current search path; $\alpha_1, \alpha_2, \alpha_3$ are hyperparameters. α_2 and α_3 control the importance of exception information and invocation path in the selection process. Initially, Eq. 5 tends to prioritize the discovery of unexplored substructures. Subsequently, it leans towards exploring substructures with larger average rewards. It is worth noting that the pruning actions $A(D_i^{v_t})$ are executed on the node with a degree of 1 in the substructure to maintain connectivity.

After a rollout in MCTS, we update the occurrence and the accumulated reward of the selected states D^{v_t} in the search path:

$$N(D^{v_t}) = N(D^{v_t}) + 1, \tag{6}$$

$$Q(D^{v_t}) = Q(D^{v_t}) + \frac{1}{L} \sum_{i=1}^{L} R(D_i^{v_t}), \tag{7}$$

where L is the length of the search path.

Upon the completion of MCT exploration for all nodes in the training set, numerous candidate substructures are obtained. In order to keep the diversity of the representative substructures, for each class c, we utilize k-means clustering to group the candidate substructures into n clusters for both normal and anomalous classes. Within the i-th cluster, the substructure embedding $h_{D^{v_t}}$ with the highest reward $R(D^{v_t})$ is chosen as the representative substructure embedding, denoted by $s_i^c \in S(1 \leq i \leq n, n \in \mathbb{N}, c \in \{0,1\})$, where $c = 0$ and $c = 1$ denotes normal and anomalous classes, respectively.

3.3 Substructure Distillation

After obtaining n representative substructure embeddings for both classes, the next goal of SLAD is to summarize key information about these representative substructures, making them directly usable in the inference process and avoiding excessive consumption in substructure exploration during inference. To achieve this, we propose a substructure distillation method to generalize essential structural patterns of both anomalous and normal classes in the distilled substructure representations.

The design of our distillation method follows two intuitions:

- **Intuition 1**: To preserve the representative patterns of substructures belonging to the same class (either normal or anomalous), each representative substructure embedding should exhibit proximity with its class's distilled substructure representations;
- **Intuition 2**: To ensure the substructures can distinguish the two classes, each representative substructure embedding should exhibit differences from the distilled substructure representations of the other class.

In response to Intuition 1, we design an approximation loss $\mathcal{L}_{Approx.}$ that minimizes the average distance of each distilled representation $p_j^c \in P(j \in \{1, ..., n\}, c \in \{0,1\})$ to the substructure representations s_i^c:

$$\mathcal{L}_{Approx.} = \frac{1}{2n}\Big(\sum_{c\in\{0,1\}}\sum_{i=1}^{n} \min_{1\leq j\leq n} \left\|s_i^c - p_j^c\right\|_2^2\Big). \tag{8}$$

In response to Intuition 2, we design a loss $\mathcal{L}_{Diff.}$ to maximize the distance of each distilled representation p_j^c to the substructure representations $s_i^{\bar{c}}$ of the other class $\bar{c} = 1 - c$:

$$\mathcal{L}_{Diff.} = -\frac{1}{2n}\Big(\sum_{c\in\{0,1\}}\sum_{i=1}^{n} \min_{1\leq j\leq n} \left\|s_i^c - p_j^{\bar{c}}\right\|_2^2\Big). \tag{9}$$

Then, the overall distillation loss is defined by:

$$\mathcal{L}_{Distill} = \beta_1 \mathcal{L}_{Approx.} + \beta_2 \mathcal{L}_{Diff.}, \tag{10}$$

where β_1 and β_2 are two hyperparameters that determine the importance of $\mathcal{L}_{Approx.}$ and $\mathcal{L}_{Diff.}$.

3.4 Soft Pruning with Attention Mechanism

SLAD aims to perform anomaly detection by leveraging the substructure information of nodes. Therefore, once distilled substructure representations P are learned, the next step is to effectively use P to capture the crucial substructure information of nodes for anomaly detection. Building on this, we introduce a method of soft pruning to discover crucial neighbors of nodes, thereby capturing the key substructure information.

The core idea is that a node on a graph should pay more attention to the neighbors with similar substructure information. The input of the soft pruning layers is the approximate substructure information $\mathcal{P}_i$ and the original feature x_{v_i} of each node v_i. $\mathcal{P}_i$ is obtained by performing attention computation between the node embedding $f(x_{v_i})$ and the distilled substructure representations P:

$$\mathcal{P}_i = Attn(f(x_{v_i}), P), \tag{11}$$

where $Attn(\cdot)$ is the attention function [27]. We use each node embedding $f(x_{v_i})$ to query the distilled substructure representations P to get approximate substructure information $\mathcal{P}_i$.

After that, we transform the original feature x_{v_i} of each node v_i and then concatenate it with corresponding approximate substructure information $\mathcal{P}_i$ as input node embeddings of the soft-pruning process, shown in Eq. 12.

$$h_i^0 = \sigma(W x_{v_i} + b) \parallel \mathcal{P}_i \tag{12}$$

The soft pruning layer employs the multi-head attention mechanism with GNNs to capture the key substructures for nodes. More specifically, soft pruning assigns attention weights to neighbors of the target node in the aggregation step of GNNs. The attention weights are determined by both the features and the approximate substructure information of nodes. Given the node embeddings $H^l = \{h_1^l, h_2^l, ..., h_n^l\}$ in layer l of soft pruning layers, node v_i's attention weight to node v_j is formulated as:

$$\begin{aligned} q_{e,i}^l &= W_{e,q}^l h_i^l + b_{e,q}^l \\ k_{e,j}^l &= W_{e,k}^l h_j^l + b_{e,k}^l \\ \omega_{e,ij}^l &= \frac{\langle q_{e,i}^l, k_{e,j}^l \rangle}{\Sigma_{u\in\mathcal{N}(i)} \langle q_{e,i}^l, k_{e,u}^l \rangle}, \end{aligned} \tag{13}$$

where $q_{e,i}^l \in \mathbb{R}^d$ and $k_{e,j}^l \in \mathbb{R}^d$ refer to query vector and key vector, respectively; $W_{e,q}^l$ and $W_{e,k}^l$ denote some trainable parameters in layer l for head e; $\langle q, k \rangle = exp(\frac{qk^\top}{\sqrt{d}})$ refers to dot product function with exponential scale; $\mathcal{N}(i)$ refers to the neighbors of node v_i.

Then we aggregate the messages from neighbors with attention weight and concatenate each head to get node embeddings:

$$\begin{aligned} v_{e,j}^l &= W_{e,v}^l h_j^l + b_{e,v}^l \\ h_i^{l+1} &= \|_{e=1}^{C} \Sigma_{j\in\mathcal{N}(i)} \omega_{e,ij}^l v_{e,j}^l \end{aligned} \tag{14}$$

where $v_{e,j}^l \in \mathbb{R}^d$ refers to value vector as messages from neighbors; $W_{e,v}^l$ denotes the trainable parameters; C is the number of heads; and $\|$ is the concatenation operation.

In this way, each node can identify more important neighbors and thus find a key substructure in a soft pruning manner. Then, we employ a multi-layer perception with a softmax activation function to output the final classification result for the node v_i.

The pseudo-code of the anomaly inference of SLAD is illustrated in Alg. 1. Initially, given a request graph G, line 1 utilizes a GNN encoder for the node embeddings. Subsequently, for each node embedding, at line 3 it gets the approximate substructure information $\mathcal{P}_i$. Following this, lines 4 - 5 make a soft pruning. Finally, line 6 gets a prediction label by an MLP with softmax for each node.

Algorithm 1 Overview of SLAD for anomaly inference phase

Require: (1) A request graph G with nodes $\{v_i \mid v_i \in G, 1 \leq i \leq |G|\}$; (2) the well-trained SLAD including GNN encoder f, distilled substructure representations P, soft-pruning layers f_{sp} and an MLP layer with Softmax

Ensure: The predicted label $\hat{y}_{v_i}$ for each node $v_i \in G$

1: Encode each node v_i via f to get node embedding $f(x_{v_i})$.
2: **for** Each node embedding $f(x_{v_i})$ in $\{f(x_{v_i})\}_1^{|G|}$ **do**
3: Acquire the approximate substructure information $\mathcal{P}_i$ via the attention mechanism $Attn(f(x_{v_i}), P)$.
4: Concatenate transformed original feature with $\mathcal{P}_i$ to get node embedding h_i^0, shown in Eq. 12.
5: Acquire the key substructure information $h_i := f_{sp}(h_i^0)$ with soft pruning in Eq. 13 and Eq. 14.
6: Apply $Softmax(MLP_W(h_i))$ to predict node v_i's label $\hat{y}_{v_i}$.
7: **end for**

3.5 Objective Function

The loss function of SLAD mainly consists of two components, distill loss $\mathcal{L}_{Distill}$ and cross-entropy loss $\mathcal{L}_{CrsEnt}$. Our objective is to: 1) enable representative substructures to assist SLAD in distilling important structural patterns of both anomalous and normal classes; and 2) maximize the classification accuracy, which is quantified by cross-entropy loss, on the training data. Overall, the loss function of SLAD is defined as:

$$\mathcal{L} = \mathcal{L}_{Distill} + \mathcal{L}_{CrsEnt}, \tag{15}$$

$$\mathcal{L}_{CrsEnt} = \frac{1}{N_v} \sum_{i=1}^{N_v} CrsEnt(\text{SLAD}(x_{v_i}), y_{v_i}), \tag{16}$$

where N_v is the number of nodes in the training set, and SLAD$(\cdot)$ denotes our framework as a function.

The pseudo-code of the training phase of SLAD is illustrated in Alg. 2. Within each request graph, line 4 represents the warm-up stage. Line 6 uses loss in Eq. 15 to train SLAD. Line 8 involves identifying and updating the current representative substructures via MCTS, followed by model training.

Algorithm 2 Overview of SLAD for training phase

Require: Training graph dataset $\{G_i\}_{i=1}^{N_G}$ with node dataset $\{(v_j, y_{v_j}) \mid v_j \in G_i\}$, Training epochs T, Warm-up epoch T_w, MCTS epoch set T_m

Ensure: The trained SLAD for anomaly detection

1: Initialize the SLAD model
2: **for** Training epoch from $t = 1, 2, 3, ..., T$ **do**
3: **if** $t < T_w$ **then**
4: Warm-up the model using the loss in Eq. 16.
5: **else if** $t \geq T_w$ & $t \notin T_m$ **then**
6: Train the model by the loss in Eq. 15.
7: **else**
8: Update representative substructures via MCTS, and train the model by loss in Eq. 15.
9: **end if**
10: **end for**

3.6 Complexity Analysis

Next, we analyze the complexity of SLAD. Given a set of request graphs with total n nodes as input, SLAD mainly consists of four components, i.e., substructure search via MCTS, GNN encoder, approximate substructure acquisition, and soft-pruning layers. For substructure search via MCTS, we get n k-hop subgraphs to look for representative substructures. For the MCTS of each k-hop subgraph, the complexity is $O(dmnr)$, where d is the dimension of GNN hidden layer, m is the node number of the k-hop subgraph, and r is the number of rollouts. So for the MCTS component, the complexity is $O(dmn^2r)$. For the GNN encoder component (GTNN [23] for SLAD), the complexity is $O(and^2)$, as well as the soft-pruning layers, in which a is the average neighbor number of nodes. For the approximate substructure acquisition, the complexity is $O((n+p)d^2)$, where p is the number of the distilled substructure representations.

Therefore, during the training phase, the complexity of SLAD is $O(dmn^2r + (an+p)d^2)$. While in the anomaly inference phase, the complexity will be reduced to $O((an+p)d^2)$.

4 EXPERIMENTS

4.1 Experimental Setup

4.1.1 Benckmark. We evaluate SLAD on three log datasets, Forum [15], Halo, and Novel, collected from three open-source systems, respectively. Forum contains 6,785,624 log messages with 13.19% anomalous, Halo includes 37,528,612 log messages with 10.58% anomalous, Novel has 14,492,161 log messages with 6.4% anomalous.

4.1.2 Data Preprocessing. It mainly includes log Parsing and log Grouping. Log parsing involves extracting event types that transform log messages with similar structures into a universal type. Log grouping is to divide logs into different groups, with each group representing a distinct set of log messages. In experiments, we use trace window for log grouping, assigning log messages with the same traceID to the same group and constructing a request graph.

After data preprocessing, we construct a request graph for each log group. The statistics of the three datasets are shown in Table 1.

4.1.3 Baselines. We compare four classes of baselines. 1. General GNNs: GAT [28], GIN [31], GTNN [23]; 2. Substructure-based methods: SAGNN [38], GNNAK [42]; 3. Log anomaly detection methods: DeepLog [6], LogGD [30], Glad-PAW [29], TP-GNN [15]; 4. Graph anomaly detection methods: BWGNN [26], GHRN [7].

4.1.4 Experimental Settings. All experiments are conducted on a Nvidia RTX 4090 GPU with 24GB GPU memory, an AMD Ryzen 7 7700X 8-Core CPU with 64GB CPU memory, and Ubuntu 22.04.3 LTS. The split ratio for train/validation/test is set to 8:1:1. In order to prevent information leakage, we place all graphs obtained from requests accessing the same functionality into either the training, validation, or testing set. For dataset Forum, Halo and Novel, the input feature dimension of each node is 247, 484 and 499, respectively, which includes the one-hot representation of the file name, the one-hot representation of the event type, the one-hot representation of the exception type and the cost time of the action instance. The number of hidden layers of GNNs is set to 3 (k of k-hop subgraphs is equal to the number of hidden layers). The

Table 1: The statistics of our benchmarks.

	|G|	|V|	|E|	# Anomalous Graphs	% Anomalous Graphs	# Anomalous Nodes	% Anomalous Nodes
Forum	214,233	6,455,450	6,241,217	97,825	45.66%	854,988	13.24%
Halo	100,000	1,563,462	1,687,186	45,181	45.18%	54,448	3.48%
Novel	149,499	1,926,609	1,777,110	68,466	45.80%	68,471	3.80%

dimension of the hidden size is set to 256. The number of distilled substructure representations is set to 110 for Forum, 90 for Halo and 70 for Novel. For other parameter settings, μ in Eq. 2 is set to 1e-4; α_1, α_2, α_3 in Eq. 5 are set to 10,1000,1000,respectively; β_1, β_2 in Eq. 10 are set to 0.25 and 0.0025,respectively; $\mathcal{K}$ in Eq. 2 is set to 200. The total training epochs are 20, containing 14 epochs for warm-up. For each MCTS, we make rollout for 5 times and each rollout ends when the resulting substructures are with 4 nodes. For each approach, experiments are repeated 5 times to get mean and standard derivation. In the end, we use Adam as the optimizer with a learning rate of 1e-3.

4.2 Overall Comparison

Table 2 compares F1 score, Recall, Precision, and PR-AUC for baselines and SLAD. SLAD consistently achieves competitive or superior results, demonstrating robustness and effectiveness. In comparison, TP-GNN, BWGNN, GHRN are relatively competitive approaches. TP-GNN combines temporal information and structural information to learn node representations, achieving good results. BWGNN and GHRN focus on the anomaly problem by analyzing the graph spectral energy distribution. However, these approaches primarily consider the overall information of the entire graph and overlook the subtle changes in substructure, which are essential for code file anomaly detection from system logs. Compared to GHRN, SLAD achieves additional performance gains of at least 4% on the F1 score. In contrast, substructure-based approaches, GNNAK and SAGNN, capture information from the perspective of substructures. However, they tend to explore substructures about the entire graph for graph classification. This emphasis on overall graph structure changes, rather than from the perspective of nodes, results in worse performance compared to SLAD. Sequence-based approaches like DeepLog, which are based on sequences, only focus on the temporal correlations of log events and fail to effectively capture anomalous behaviors in code files, as discussed in the introduction, hence exhibiting poor performance. In addition, for existing log anomaly detection methods, such as LogGD, although they have achieved significant results in detecting whether a segment of logs is anomalous, their performance is not ideal for our more fine-grained node classification task. This is because they typically consider the overall information of the graph or sequence and fail to effectively capture changes in the local environment of nodes.

4.3 Ablation Study

The ablation study aimed to analyze the effectiveness of different components in SLAD. The results are summarized in Table 3, which evaluates key metrics including F1 score, Recall, Precision, and PR-AUC on our datasets. Different component combinations are assessed, ranging from 1) *GNN Encoder only*, which only preserves

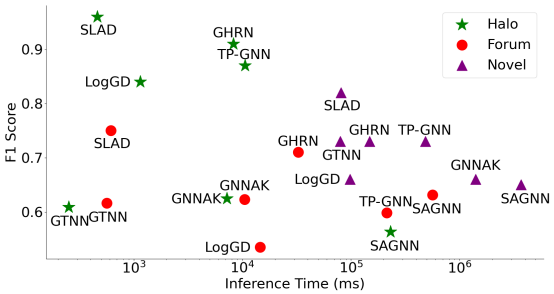


Figure 6: Inference Efficiency Comparison. For each dataset, the approach is positioned closer to the top-left if it achieves better performance in both efficiency and effectiveness.

GNN encoder and MLP with Softmax; 2) *SLAD w/o MCTS*, which retains GNN encoder, soft-pruning layers and MLP with Softmax; 3) *SLAD w/ random MCTS* which, based on our framework, uses a random strategy in substructure search via MCTS; 4) *SLAD w/o soft-pruning*, which refers to our full framework without soft-pruning layers; 5) *SLAD (full components)*. This suggests that each component of SLAD contributes significantly to the overall performance. MCTS improves performance owing to its capability to search essential substructures. The soft-pruning layers further refine the model by selectively focusing on key substructures, leading to noticeable gains.

4.4 Inference Efficiency and Efficacy Analysis

Then, we compare SLAD with some competitive baselines in terms of inference efficiency and F1 score, as shown in Fig. 6. The closer an approach is to the top-left corner of this figure, the better the trade-off between the F1 score and inference time. For each dataset, SLAD consistently occupies the top-left corner. Although SLAD slightly lags behind GTNN in inference efficiency, its accuracy far surpasses that of GTNN. Furthermore, compared to other methods, SLAD significantly outperforms them in both F1 score and inference efficiency. Particularly noteworthy is SLAD's inference time, which is at least 15 times faster than substructure-based methods.

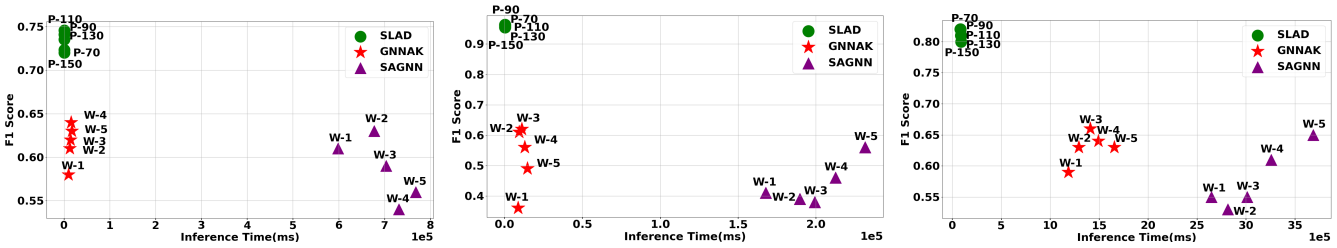
SLAD is a substructure exploration-based approach. To better evaluate the performance of SLAD, we further compared it with two substructure exploration-based approaches, GNNAK and SAGNN, in terms of the trade-offs in substructure exploration. Fig. 7 displays the F1 score of SLAD with distilled substructure representations set to 70, 90, 110, 130, and 150, along with two existing substructure exploration approaches with walk step lengths set to 1, 2, 3, 4, and 5. In summary, SLAD outperforms GNNAK and SAGNN significantly

Table 2: The experiment results with 11 baselines and our SLAD. The best performance on each metric is marked in bold font.

	Forum				Halo				Novel			
	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC
GAT	0.34±0.15	0.26±0.14	0.59±0.05	0.49±0.06	0.56±0.01	0.52±0.02	0.60±0.02	0.57±0.01	0.63±0.03	0.78±0.12	0.54±0.09	0.49±0.07
GIN	0.39±0.05	0.28±0.06	0.71±0.11	0.64±0.04	0.64±0.02	0.76±0.01	0.56±0.02	0.65±0.03	0.55±0.10	0.58±0.13	0.53±0.08	0.45±0.13
GTNN	0.62±0.08	0.66±0.15	0.61±0.09	0.68±0.09	0.74±0.01	0.91±0.01	0.62±0.01	0.84±0.01	0.73±0.06	0.80±0.09	0.69±0.13	0.86±0.08
GNNAK	0.62±0.06	0.64±0.08	0.63±0.11	0.66±0.14	0.62±0.09	0.60±0.18	0.71±0.11	0.71±0.12	0.66±0.09	0.77±0.11	0.61±0.13	0.71±0.14
SAGNN	0.63±0.07	0.62±0.14	0.68±0.10	0.70±0.06	0.56±0.15	0.52±0.18	0.72±0.04	0.68±0.10	0.65±0.09	0.78±0.08	0.57±0.13	0.78±0.09
DeepLog	0.34±0.09	0.69±0.01	0.23±0.06	0.44±0.04	0.43±0.03	0.45±0.01	0.41±0.02	0.48±0.02	0.51±0.09	0.51±0.11	0.52±0.10	0.46±0.11
LogGD	0.53±0.03	0.48±0.08	0.66±0.16	0.61±0.13	0.84±0.08	0.95±0.04	0.75±0.11	0.91±0.13	0.66±0.11	0.70±0.08	0.65±0.15	0.77±0.13
Glad-PAW	0.51±0.10	0.42±0.12	0.73±0.11	0.66±0.08	0.36±0.12	0.27±0.13	0.61±0.05	0.38±0.10	0.63±0.02	0.60±0.06	0.68±0.08	0.71±0.04
TP-GNN	0.60±0.02	0.49±0.04	**0.79±0.10**	0.71±0.13	0.87±0.01	0.81±0.02	0.95±0.04	0.95±0.01	0.73±0.08	0.72±0.14	0.74±0.04	0.78±0.09
BWGNN	0.68±0.08	0.83±0.03	0.61±0.10	0.73±0.12	0.90±0.02	0.86±0.02	0.94±0.03	0.95±0.03	0.75±0.06	0.88±0.07	0.65±0.09	0.80±0.07
GHRN	0.71±0.11	0.79±0.08	0.65±0.11	0.78±0.10	0.91±0.01	0.88±0.01	0.95±0.02	0.96±0.01	0.73±0.10	0.89±0.12	0.63±0.12	0.77±0.10
SLAD(ours)	**0.75±0.06**	**0.85±0.10**	0.69±0.09	**0.83±0.09**	**0.96±0.01**	**0.98±0.01**	**0.95±0.02**	**0.99±0.01**	**0.82±0.02**	**0.89±0.12**	**0.78±0.10**	**0.91±0.05**

Table 3: Ablation study. *GNN Encoder only* represents only using GNNs to detect anomalies; *SLAD w/o MCTS* denotes SLAD without MCTS; *SLAD w/ random MCTS* refers to SLAD with a random strategy in MCTS; *SLAD w/o soft-pruning* means that the soft-pruning module is removed from SLAD.

	Forum				Halo				Novel			
	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC
GNN Encoder only	0.62±0.08	0.66±0.15	0.61±0.09	0.68±0.09	0.74±0.01	0.91±0.01	0.62±0.01	0.84±0.01	0.73±0.06	0.80±0.09	0.69±0.13	0.86±0.08
SLAD w/o MCTS	0.61±0.07	0.89±0.11	0.48±0.12	0.68±0.09	0.81±0.01	0.94±0.01	0.71±0.01	0.94±0.01	0.75±0.07	0.80±0.12	0.71±0.11	0.85±0.09
SLAD w/ random MCTS	0.65±0.12	0.93±0.05	0.52±0.17	0.74±0.11	0.87±0.01	0.96±0.01	0.79±0.01	0.97±0.01	0.69±0.08	0.81±0.10	0.62±0.11	0.81±0.09
SLAD w/o soft-pruning	0.71±0.10	**0.94±0.07**	0.59±0.18	0.78±0.10	0.91±0.01	0.97±0.01	0.86±0.02	0.98±0.01	0.77±0.04	0.84±0.11	0.73±0.13	0.88±0.05
SLAD	**0.75±0.06**	0.85±0.10	**0.69±0.09**	**0.83±0.09**	**0.96±0.01**	**0.98±0.01**	**0.95±0.02**	**0.99±0.01**	**0.82±0.02**	**0.89±0.12**	**0.78±0.10**	**0.91±0.05**



(a) Forum (b) Halo (c) Novel

Figure 7: Substructure exploration trade-off for SLAD, GNNAK and SAGNN. "•" refers to SLAD(ours) in different numbers of distilled substructure representations P(70 to 150), "⋆" denotes GNNAK in walks with varying step lengths W(1 to 5), and "△" represents SAGNN in walks with varying step lengths W(1 to 5). The approach is positioned closer to the top-left if it achieves better performance in both efficiency and effectiveness.

in both time and accuracy. GNNAK and SAGNN exhibit some instability on the these datasets, and they consume lots of time with the increase in step length, especially for SAGNN.

We also conduct a statistical analysis of the computational resource usage of baselines and SLAD during the anomaly inference phase. The statistical results are shown in Table 4. The computational resources used by SLAD and other baselines during anomaly inference are within reasonable ranges.

Overall, SLAD achieves a good balance between inference efficiency and efficacy. It ensures the advantage of anomaly detection accuracy while maintaining good inference efficiency.

4.5 Impact of Hyperparameters

We conducted a series of experiments to investigate the impact of various hyperparameters on SLAD. To study the impact of each hyperparameter on the overall performance of the model, we vary one hyperparameter at a time while fixing all other hyperparameters (shown in 4.1.4). Each experiment is repeated 5 times. The results related to the scale of SLAD parameters are shown in Fig. 8.

We vary the size of the hidden layer in the GNN encoder from 32 to 512. The F1 score of SLAD peaks at the hidden layer size of 256 and then shows a slight decline. Therefore, we set the GNN hidden layer size at 256. For the number of hidden layers, we vary it from 1 to 5. The F1 score reaches its maximum value when 3 hidden layers are applied. As the number of hidden layers continues to increase, over-smoothing may occur, leading to a slight decline in the F1 score. Consequently, we set the number of GNN hidden layers as 3. For the number of distilled substructure representations, we set values from 30 to 110. Within this range, the F1 score on Halo, peaks at 90 and then stabilizes. On Forum, the F1 score shows an upward trend

Table 4: The statistics of computing resource usage for the anomaly inference phase on our datasets.

	Forum				Halo				Novel			
Baseline	CPU	Memory	GPU	GPU Memory	CPU	Memory	GPU	GPU Memory	CPU	Memory	GPU	GPU Memory
GAT	2.60%	5.50%	11.62%	1.00%	0.42%	7.98%	8.99%	0.68%	0.57%	9.88%	9.21%	1.03%
GIN	0.40%	5.30%	8.65%	1.54%	0.70%	7.84%	9.60%	0.86%	0.98%	8.99%	9.73%	1.33%
GTNN	0.60%	5.30%	12.32%	0.47%	0.60%	7.90%	11.63%	0.51%	0.97%	8.97%	11.87%	1.99%
GNNAK	0.69%	7.50%	12.50%	2.54%	0.85%	8.87%	10.56%	2.30%	1.35%	8.65%	10.34%	1.79%
SAGNN	0.65%	6.40%	10.65%	2.62%	0.60%	6.40%	9.55%	2.62%	0.91%	6.88%	9.39%	2.88%
DeepLog	0.64%	9.40%	11.20%	3.43%	0.80%	7.56%	10.10%	3.64%	1.08%	8.76%	10.55%	3.95%
LogGD	0.50%	5.50%	11.54%	2.62%	0.45%	7.95%	8.69%	2.61%	0.75%	8.33%	9.77%	2.91%
Glad-PAW	0.52%	5.40%	8.33%	2.64%	0.63%	7.80%	7.92%	2.35%	0.91%	8.11%	8.54%	2.81%
TP-GNN	0.62%	5.40%	10.57%	2.25%	0.55%	5.80%	8.10%	2.40%	0.97%	6.44%	8.99%	2.79%
BWGNN	0.71%	6.10%	9.47%	2.45%	0.64%	7.10%	8.83%	2.10%	0.75%	7.44%	8.19%	3.82%
GHRN	0.92%	9.23%	12.55%	3.16%	0.80%	9.22%	10.01%	3.12%	1.11%	9.57%	10.98%	3.45%
SLAD(ours)	0.53%	5.60%	10.23%	0.95%	0.71%	8.35%	9.50%	1.03%	1.11%	8.65%	10.01%	1.96%

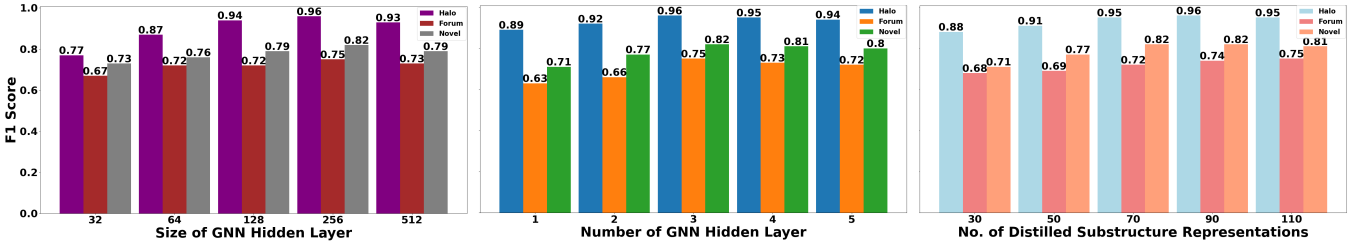


Figure 8: Impact of hyper-parameters for SLAD

as the number of distilled substructure representations increases. On Novel, it reaches its peak when the number is 70. Based on these results, we set the number of distilled substructure representations to 90 for Halo, 110 for Forum, and 70 for Novel.

4.6 Performance on Different Anomaly Types

Next, we display the performance on anomaly detection of different anomaly types for SLAD and some competitive baselines in Table 5. For different types of these anomalies, "chain change" represents the removal or addition of other source code files in the original invocation chain, "invocation change" denotes an anomaly resulting in invoking the wrong source code file, while "argument change" indicates an anomaly that causes anomalous parameters. Finally, "condition change" refers to an anomaly leading to anomalous conditional statements. For each type of anomaly, we compute the prediction accuracy, which represents the proportion of correctly predicted instances within a particular anomaly type.

Overall, SLAD demonstrates the best performance, particularly showing significant advantages on Forum and Halo datasets, while the other baselines exhibit unstable performance on different anomaly types. For Novel, where the log graphs are relatively small in scale, making the advantage of substructure information less pronounced in smaller graphs, SLAD's superiority is less prominent. This result highlights SLAD's stability and comprehensiveness in recognizing different types of anomalies.

4.7 Performance on Unseen Anomaly Types

System updates may lead to the emergence of new anomaly types, with anomaly patterns potentially differing from those previously observed. To demonstrate the performance of SLAD for unseen anomaly types, we respectively conduct experiments by removing one type of anomaly sample from the training set and then using the trained model to predict the unseen anomaly type. Furthermore, we fine-tune the trained model sequentially using 100, 300, 500, and 700 graphs of the corresponding anomaly type. Due to space constraints, only Halo dataset results are shown in Fig. 9, consistent across all datasets.

The results indicate that SLAD significantly outperforms other baselines in recognizing unseen anomaly types. With only a small portion of the unseen anomaly type data used for fine-tuning, SLAD can effectively identify these anomalies, clearly surpassing the performance of the baselines. This demonstrates the robust capabilities of SLAD in adapting to new anomaly patterns.

5 RELATED WORK

5.1 Log Anomaly Detection

Early studies on log anomaly detection often rely on predefined rules, such as explicit criteria and thresholds [17, 21, 35]. However, these approaches require strong domain expertise and thus cannot be generalized to different datasets. Subsequently, researchers begin to explore machine learning techniques for log anomaly detection, including Principal Component Analysis [32], Clustering [14] and Support Vector Machines [13]. However, they struggle to effectively capture complex patterns and dependencies within log data.

Recently, deep learning techniques have been exploited in this field. Some treat logs as event sequences and apply sequence-based neural networks to detect anomalies, such as the Long Short-Term

Table 5: The accuracy of anomaly detection w.r.t. different anomaly types on our datasets. The column "Normal" denotes the prediction accuracy of normal nodes. The other columns denote the detection accuracy of different anomaly types.

	Forum				Halo					Novel				
	Normal	Chain	Invocation	Argument	Normal	Chain	Invocation	Condition	Argument	Normal	Chain	Invocation	Condition	Argument
GTNN	0.80	0.80	1.00	0.85	0.98	0.88	0.91	0.94	0.26	0.98	0.93	0.37	0.56	0.87
SAGNN	0.85	0.75	0.92	0.74	0.99	0.49	0.27	0.28	0.18	0.97	0.89	0.51	0.30	0.90
GNNAK	0.80	0.63	0.77	0.77	0.78	0.55	0.53	0.46	0.30	0.97	0.95	0.38	0.73	0.79
TP-GNN	**0.98**	0.54	0.31	0.18	0.99	0.67	0.83	0.95	**0.91**	0.97	0.47	0.51	**0.83**	0.85
GHRN	0.88	0.79	0.85	0.81	0.99	0.85	0.92	0.95	0.83	0.97	0.98	**0.72**	0.51	0.89
SLAD(ours)	0.91	**0.81**	**1.00**	**0.88**	**0.99**	**0.92**	**0.99**	**0.99**	0.77	**0.99**	**1.00**	0.62	0.76	**0.90**

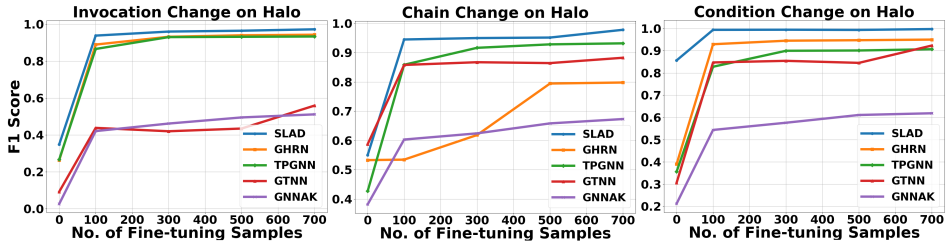


Figure 9: Performance for unseen anomaly types on Halo

Memory (LSTM) [3, 6, 12, 18, 40], Gated Recurrent Unit (GRU) [34]. Additionally, some studies also explore the application of recently popular neural networks such as Transformer and BERT [8, 9, 19].

Following this, several studies have shown that graph representation learning can significantly improve the performance of log anomaly detection. These studies concentrate on the task of graph classification. Xie et al. [30] propose to represent logs as graphs and employ a Graph Transformer Neural Network to detect anomalies within the graph. Wang et al. [29] integrate positional information of events into the graph representation, followed by the application of a position-aware weighted graph attention layer to acquire the graph representation for log anomaly detection. Zhang et al. [39] constructs graphs by utilizing log events occurring within the same trace. They employ a gated graph neural network to acquire graph representations and detect log anomalies through Deep Support Vector Data Description. Yan et al. [33] incorporated temporal information into graph learning models for log anomaly detection.

Existing graph-based approaches emphasize the analysis of the overall graph structure. However, they fail to adequately explore the substructures containing crucial information regarding node anomalies, thus impeding accurate anomaly detection.

5.2 Substructure-based Graph Representation Learning

Several research studies have been dedicated to extracting substructure to enhance the expressive capabilities of GNNs. Certain research focuses on prior encoding substructure [1, 2], such as cliques, Others concentrate on random walk-based methods [25, 38, 41], and some utilize the information related to the k-hop subgraph [4, 20, 22, 42]. Zeng et al. [38] combine the random walk-based method and k-hop subgraphs to capture substructure information. Nonetheless, these methods not only require significant time for walks during inference but also fail to discover specific substructures within software systems.

Additionally, some research focuses on learning a representation of query-induced subgraphs for making link predictions [36, 37]. However, these methods store the local structural context of nodes during training to save inference time. Because the induction of subgraphs relies on the local structural context of query points, they cannot represent unseen points, making it difficult to apply them to inductive node classification tasks.

Comparsion. Our research focuses on detecting anomalous code files through log analysis, integrating file invocation information into log graphs, and employing substructure patterns of nodes for anomaly detection. It distinguishes the existing works: 1) an MCTS-based method that automatically discovers representative substructures; 2) an effective knowledge distillation method is to avoid substructure exploration during inference, thus reducing the computational cost; 3) a novel soft pruning method is introduced to obtain the key substructure during inference.

6 CONCLUSION

This paper introduces a framework, SLAD, to identify anomalous code files from system logs. It employs a novel Monte Carlo Tree Search strategy aimed to automatically identify representative substructures. Additionally, SLAD integrates a substructure distillation method to summarize the general patterns from the representative substructures, avoiding the exploration of substructures during inference. Building upon this, a soft pruning method is designed to find the key substructure for each node. The experiments indicate that SLAD outperforms the state-of-the-art log anomaly detection approaches and node classification methods. Furthermore, SLAD excels in inference efficiency compared to current substructure exploration approaches.

ACKNOWLEDGMENTS

This research is supported by the Marsden Fund (MFP-UOA2123) administered by the Royal Society of New Zealand.

REFERENCES

[1] Cristian Bodnar, Fabrizio Frasca, Nina Otter, Yuguang Wang, Pietro Lio, Guido F Montufar, and Michael Bronstein. 2021. Weisfeiler and lehman go cellular: Cw networks. *Advances in Neural Information Processing Systems* 34 (2021), 2625–2640.

[2] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. 2022. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 1 (2022), 657–668.

[3] Rui Chen, Shenglin Zhang, Dongwen Li, Yuzhe Zhang, Fangrui Guo, Weibin Meng, Dan Pei, Yuzhi Zhang, Xu Chen, and Yuqing Liu. 2020. Logtransfer: Cross-system log anomaly detection for software systems with transfer learning. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 37–47.

[4] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can graph neural networks count substructures? *Advances in neural information processing systems* 33 (2020), 10383–10395.

[5] Pengfei Ding, Guanfeng Liu, Yan Wang, Kai Zheng, and Xiaofang Zhou. 2021. A-MCTS: adaptive monte carlo tree search for temporal path discovery. *IEEE Transactions on Knowledge and Data Engineering* 35, 3 (2021), 2243–2257.

[6] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[7] Yuan Gao, Xiang Wang, Xiangnan He, Zhenguang Liu, Huamin Feng, and Yongdong Zhang. 2023. Addressing heterophily in graph anomaly detection: A perspective of graph spectrum. In *Proceedings of the ACM Web Conference 2023*. 1528–1538.

[8] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. Logbert: Log anomaly detection via bert. In *2021 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[9] Shangbin Han, Qianhong Wu, Han Zhang, Bo Qin, Jiankun Hu, Xingang Shi, Linfeng Liu, and Xia Yin. 2021. Log-based anomaly detection with robust feature extraction and online learning. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2300–2311.

[10] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2017. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2017), 931–944.

[11] Dennis Hofmann, Peter VanNostrand, Huayi Zhang, Yizhou Yan, Lei Cao, Samuel Madden, and Elke Rundensteiner. 2022. A demonstration of AutoOD: a self-tuning anomaly detection system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3706–3709.

[12] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 92–103.

[13] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 583–588.

[14] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 102–111.

[15] Jie Liu, Yanni Tang, Kaiqi Zhao, Jiamou Liu, and Wu Chen. 2024. TP-GNN: Continuous Dynamic Graph Neural Network for Graph Classification. *IEEE 40th International Conference on Data Engineering (ICDE)* (2024).

[16] Zhaoli Liu, Tao Qin, Xiaohong Guan, Hezhi Jiang, and Chenxu Wang. 2018. An integrated method for anomaly detection from massive system logs. *IEEE Access* 6 (2018), 30602–30611.

[17] Jian-Guang Lou, Qiang Fu, Shenqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[18] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *IJCAI*, Vol. 19. 4739–4745.

[19] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-attentive classification-based anomaly detection in unstructured logs. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1196–1201.

[20] Giannis Nikolentzos, George Dasoulas, and Michalis Vazirgiannis. 2020. k-hop graph neural networks. *Neural Networks* 130 (2020), 195–205.

[21] John P Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC).. In *LISA*, Vol. 4. 133–150.

[22] Dylan Sandfelder, Priyesh Vijayan, and William L Hamilton. 2021. Ego-gnns: Exploiting ego structures in graph neural networks. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 8523–8527.

[23] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2021. Masked label prediction: Unified message passing model for semi-supervised classification. *International Joint Conference on Artificial Intelligence* (2021).

[24] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[25] Yifei Sun, Haoran Deng, Yang Yang, Chunping Wang, Jiarong Xu, Renhong Huang, Linfeng Cao, Yang Wang, and Lei Chen. 2022. Beyond homophily: structure-aware path aggregation graph neural network. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI*. 2233–2240.

[26] Jianheng Tang, Jiajin Li, Ziqi Gao, and Jia Li. 2022. Rethinking graph neural networks for anomaly detection. In *International Conference on Machine Learning*. PMLR, 21076–21089.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[29] Yi Wan, Yilin Liu, Dong Wang, and Yujin Wen. 2021. Glad-paw: Graph-based log anomaly detection by position aware weighted graph attention network. In *Pacific-asia conference on knowledge discovery and data mining*. Springer, 66–77.

[30] Yongzheng Xie, Hongyu Zhang, and Muhammad Ali Babar. 2022. LogGD: Detecting Anomalies from System Logs with Graph Neural Networks. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 299–310.

[31] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[32] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09* (2009).

[33] Lejing Yan, Chao Luo, and Rui Shao. 2023. Discrete log anomaly detection: A novel time-aware graph-based link prediction approach. *Information Sciences* 647 (2023), 119576.

[34] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1448–1460.

[35] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th annual computer security applications conference*. 199–208.

[36] Haoteng Yin, Muhan Zhang, Jianguo Wang, and Pan Li. 2023. SUREL+: Moving from Walks to Sets for Scalable Subgraph-based Graph Representation Learning. *arXiv preprint arXiv:2303.03379* (2023).

[37] Haoteng Yin, Muhan Zhang, Yanbang Wang, Jianguo Wang, and Pan Li. 2022. Algorithm and system co-design for efficient subgraph-based graph representation learning. *arXiv preprint arXiv:2202.13538* (2022).

[38] Dingyi Zeng, Wanlong Liu, Wenyu Chen, Li Zhou, Malu Zhang, and Hong Qu. 2023. Substructure aware graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 11129–11137.

[39] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 623–634.

[40] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[41] Zhen Zhang, Mianzhi Wang, Yijian Xiang, Yan Huang, and Arye Nehorai. 2018. Retgk: Graph kernels based on return probabilities of random walks. *Advances in Neural Information Processing Systems* 31 (2018).

[42] Lingxiao Zhao, Wei Jin, Leman Akoglu, and Neil Shah. 2021. From stars to subgraphs: Uplifting any GNN with local structure awareness. *arXiv preprint arXiv:2110.03753* (2021).

[43] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.