Brain: Log Parsing With Bidirectional Parallel Tree

Siyu Yu, Pinjia He, *Member, IEEE*, Ningjiang Chen, *Member, IEEE*, and Yifan Wu

***Abstract*—Automated log analysis can facilitate failure diagnosis for developers and operators using a large volume of logs. Log parsing is a prerequisite step for automated log analysis, which parses semi-structured logs into structured logs. However, existing parsers are difficult to apply to software-intensive systems, due to their unstable parsing accuracy on various software. Although neural network-based approaches are stable, their inefficiency makes it challenging to keep up with the speed of log production. In this work, we found that the longest common pattern among logs is likely to be part of the log template. Inspired by this key insight, we propose a new stable log parsing approach, called Brain, which creates initial groups according to the longest common pattern. Then a bidirectional tree is used to hierarchically complement the constant words to the longest common pattern to form the complete log template efficiently. Experimental results on 16 benchmark datasets show that our approach outperforms the state-of-the-art parsers on two widely-used parsing accuracy metrics, and it only takes around 46 seconds to process one million lines of logs.**

***Index Terms*—Bidirectional tree, log analysis, log data, log parsing.**

I. Introduction

LOGS are important for tracking software runtime information, which helps engineers diagnose failures. However, software-intensive systems such as cloud platforms and edge clouds, which combine many software for different services, have led to an increase in log volume and a wide range of software types. It is now impossible to manually analyze such a large amount of logs to ensure service quality following a failure [1]. To improve diagnosis efficiency and reduce human effort, automated log analysis (e.g., log-based anomaly detection [2], [3]) has been developed to expedite failure diagnosis for engineers.

Log parsing is the prerequisite for automated log analysis, converting a raw log into a log template [4]. A raw log consists of a log header and a log content. The log header usually records the timestamp, PID, etc., and the log content consists of constant and variable words. The variable word records diverse runtime information and the constant word is hard coded by engineers. For example, the raw log "`Dec 10 07:42:49 LabSZ sshd[24318]: {Invalid} user inspur from 183.136.162.51`" contains log header "`Dec 10 07:42:49 LabSZ sshd[24318]:`" and log content "`Invalid user inspur from 183.13-6.162.51`". Log parsing converts a raw log into a log template by preserving constant words and replacing variable words with wildcard "`<*>`". For example, the log content "`Invalid user inspur from 183.136.162.51`" will be parsed into log template "`Invalid user <*> from <*>`".

Traditional log parsing mainly involves setting up regular expressions [5], which is straightforward but labor-intensive. This is because software can generate logs with many templates (e.g., 76 K templates in Android dataset) [6]. To reduce manual work, automated log parsing has been proposed. This includes frequent item mining [7], [8], heuristics [9], clustering [10], and neural networks [11], [12], [13]. For example, Drain [9] is an online log parser that groups logs by the first few words. Spell [14] extracts log templates using the longest common substring algorithm.

However, existing approaches are still unsatisfactory in software-intensive systems. We have identified the following limitations of the existing approaches: *1) Poor stability,* most parsers can not work equally well across software and log types [6]. For example, the parsing accuracy of Drain on Apache (Apache server log) is 1.000, but the parsing accuracy on Proxifier (Proxifier software log) is 0.527 [6]. However, the accuracy of log parsers may significantly affect the follow-up analysis, e.g., the accuracy of log-based anomaly detection can decrease by more than 20% with just a few parsing errors [15]. *2) Unsatisfactory for different downstream tasks*, various downstream tasks of log parsing are necessary to ensure the quality of service in a software-intensive system. Different downstream tasks may require different abilities of the log parser, e.g., log key sequence anomaly detection requires accurate log grouping, and parameter anomaly detection requires precise variable extraction. Existing approaches are not effective in achieving both word-level parsing accuracy and grouping accuracy. *3) Inefficient runtime performance*, although neural network-based parsers can provide stable parsing accuracy, retraining for new software can be time-consuming. Furthermore, inference speed is slow [13], which cannot keep up with high log generation speeds. Log parsers should buy time for failure recovery to reduce economic loss.

Manuscript received 9 October 2022; revised 20 April 2023; accepted 23 April 2023. Date of publication 26 April 2023; date of current version 8 October 2023. The work was supported by the National Natural Science Foundation of China under Grants 62162003, 62102340, and 61762008. Recommended for acceptance by M. Hauswirth. *(Corresponding author: Ningjiang Chen.)*

Siyu Yu is with the School of Computer and Electronic Information, Guangxi University, Nanning, Guangxi 530004, China (e-mail: gaiusyu6@gmail.com).

Pinjia He is with the School of Data Science, Chinese University of Hong Kong, Shenzhen, Longgang 518172, China (e-mail: hepinjia@cuhk.edu.cn).

Ningjiang Chen is with the School of Computer and Electronic Information, Guangxi University, Nanning, Guangxi 530004, China, also with the Guangxi Intelligent Digital Services Research Center of Engineering Technology, Nanning 530004, China, and also with Key Laboratory of Parallel, Distributed and Intelligent Computing, Education Department of Guangxi Zhuang Autonomous Region, Guangxi University, Nanning, Guangxi 530004, China (e-mail: chnj@gxu.edu.cn).

Yifan Wu is with the Peking University, Beijing 100871, China (e-mail: yifanwu@pku.edu.cn).

Digital Object Identifier 10.1109/TSC.2023.3270566

The above three limitations motivate us to make new thinking on log parsing. *To design a log parser with good stability,* we delved into the common characteristics of logs and found the longest common patterns of logs are likely to be part of log templates. Thus, we may group logs more accurately by the longest common pattern. *To facilitate different downstream tasks*, both good group accuracy and word-level parsing accuracy are required. The longest common pattern needs to be complemented with constant words to form a complete log template. Once a constant word is added to the longest common pattern, the log group continues to divide until the grouping is correct. *To implement our approach efficiently,* such a hierarchical complement process could benefit from tree structure because it facilitates hierarchical algorithms.

Inspired by the above key insights, we proposed Brain, a stable and efficient log parser. It begins by splitting words and creating initial log groups based on the longest common pattern. Then, it adds words of the same column to a bidirectional parallel tree at the same depth, with the longest common pattern as the root. The number of branches determines the classification of each node. Finally, Brain combines the longest common pattern and constant nodes in the bidirectional tree to form the log template.

We evaluated Brain on the most commonly used benchmark datasets [6], which contain 16 different log datasets. Experimental results show that Brain achieves an average group accuracy of 0.981 and word-Level parsing accuracy of 0.929, outperforming state-of-the-art parsers by 11.6% and 10.4%, respectively. For efficiency, Brain only takes about 46 seconds to process one million logs, which is strongly competitive with state-of-the-art parsers. Moreover, Brain is also stable across different log datasets.

In summary, we make the following main contributions:

- We propose a novel bidirectional parallel tree for building a stable and efficient log parser that can work well across different log types.
- We evaluate Brain on 16 benchmark datasets and the results demonstrate that Brain outperforms the state-of-the-art parsers. The experimental results confirm the effectiveness and efficiency of Brain.
- We publish our code for better reproducibility and allow researchers to use it for further study.[1]

The rest of the paper is organized as follows. Section II introduces the preliminaries of this paper. Section III describes the details of Brain. Section IV presents our experimental results. We do some discussion in Section V. Finally, Section VI discusses the threats to the validity of our findings and Section VII concludes the paper.

II. Preliminaries

A. Log Parsing

As shown in Fig. 1, logs are produced by logging statements (e.g., `logInfo()`, `print()`) in the program code. The produced log consists of the log header (e.g., time "`17/06/09 20:10:48`", PID, log level "`INFO`") and the log content (e.g., "`Block broadcast_0_piece0 stored as bytes in memory(estimated size 93.0B, free 317.1 K-B)\}`", where the log content is composed of constant words (e.g., "`Block`") and variable words (e.g., "`93.0, 317.1`"). To get structured log, existing log parsers parse semi-structured logs into structured logs by keeping predicted constant words and replacing predicted variable words with wildcards: "`< * >`" like generated log template in Fig. 1: "`Block <*> stored as bytes in memory (estimated size <*> B, free <*> KB)`".

```
/*
A logging code snippet from spark/storage/memory/Memory Store scala
*/
logInfo("Block %s stored as bytes in memory (estimated size %s, free %s )").
format(
blockId,
Utils.bytesToString(size),
Utils.bytesToString(maxMemory – blocksMemory)))
```

⇩

```
17/06/09 20:18:48 INFO storage MemoryStore: Block broadcast_0_piece0
Stored as bytes in memory (estimated size 93.0B, free 317.1 KB)
```

⇩

TIMESTAMP	17/06/09 20:18:48
LEVEL	INFO
CLASS	Storage.MemoryStore
TEMPLATE	Block* stored as bytes in memory (estimated size * B, free * KB)
PARAMETERS	["broadcast_0_piece0", "93.0","317.1"]

Fig. 1. Example of log parsing.

B. Related Works

Existing log parsing approaches mainly include regular expression filters [5], frequent item mining [7], [8], heuristic [9], clustering [10] and neural network [11], [12]. However, regular expression filter requires a lot of manual operations, making it inadequate for the continuously growing log volume in software-intensive systems. Automated log parsing is thus the key to dealing with such a huge volume of logs.

Frequent Item Mining: Such approaches consider that items (e.g., tokens, n-gram) that occur more in the entire dataset are more likely to contain constant words. SLCT [8] is the first paper on automated log parsing, and it is also the first research about log parsing with frequent item mining. The latest frequent item mining-based research is Logram [7], which combines words in the entire log dataset into n-grams. Logram thinks that the words in the low-frequency n-gram are more likely to be variables. FT-tree (Frequency template tree) [16] determines the final template tree by cutting branches.

Clustering: The clustering-based parser believes that logs belonging to the same log template can be clustered together by a certain feature. LKE [17] adopts a hierarchical clustering algorithm with a custom-weighted edit distance metric. LenMa [10] focuses on the word length feature and converts log into a vector of the number of word letters, e.g., log "`token has expired on`" will be converted to a vector `[5, 3, 6, 2]`.

[1]https://github.com/gaiusyu/Brain

TABLE I
BEST AND WORST GROUP ACCURACY OF PARSERS ON 16 BENCHMARK DATASETS

	Drain	Logram	LenMa
Best	0.999	0.981	0.998
Worst	0.250	0.460	0.141

TABLE II
LOG SAMPLES FROM JENKINS

ID	Logs
0	multi-web #45 main build action completed: SUCCES
1	multi-web #46 main build action completed: FAILURE
2	security #46 main build action completed: FAILURE
3	multi-back #18 main build action completed: SUCCESS
4	multi-back #19 main build action completed: SUCESS

Heuristics: Drain [9] is an online parser, which employs a fixed-depth tree structure to assist in dividing logs into different groups. Spell [14] uses the longest common sub-sequence algorithm to extract log templates.

Neural Network: Transformer encoder is used by Nulog [12] to classify masked words one by one. Uniparser [11] trains BLSTM network to use contextual information to parse words. Semlog [13] trains a bert-based model to widen the semantic contribution difference between constants and variables.

Code Parser: Programming languages and logs have similar characteristics, both are semi-structured natural languages. For program language parsing, the abstract syntax tree (AST) is a hierarchical program representation that presents source code structure according to the grammar of a programming language, each AST node corresponds to an item of a source code. AST-based code parsing has become a prerequisite for many code analyses, e.g., software defection prediction [18], [19], [20], code2seq [21]. Due to the similarity between logs and program code, log parsing may benefit from AST-like hierarchical algorithms.

C. Limitations of Existing Work

In the software-intensive system, the speed of log generation and various types of logs require log parsers to be stable and efficient across different types of software logs. Even though significant efforts have been put into log parsing, existing parsers are still unsatisfactory in the software-intensive system. We summarized three major limitations as follows:

1) Poor Stability: Software-intensive systems such as cloud and edge platforms require log parsers to be effective across different log types. However, existing parsers do not perform equally well on various benchmark datasets [6], [22]. Table I shows a wide range of accuracy for log parsers on different datasets. For example, MoLFI [23] only achieves an accuracy of 0.213 on OpenStack software logs, making it difficult to apply to the real world. But on the Apache dataset, MoLFI achieves an accuracy of 1.000. In addition, we found that some assumptions of existing approaches are only applicable to a portion of the software, such as Drain not working well on logs that start with a variable. Table II shows log samples from Jenkins, a software providing continuous integration and deployment services ,[2] and these logs start with the project name (i.e., variable words). LenMa's theory [10], which groups logs with similar length vectors, is also flawed, as there are many logs with similar length vectors but belonging to different templates, such as "`VM started`" and "`VM stopped`".

2) Unsatisfactory for Different Downstream Tasks: In the operation and maintenance of the software-intensive system, in addition to log sequence anomaly detection [2], [24], parameter anomaly detection [25] is also helpful in failure diagnosis. This requires accurate extraction of variables from logs. However, most existing parsers perform well only on group accuracy, which only measures whether logs from the same template are grouped. Recent studies [7], [11] have evaluated the word-level parsing accuracy of existing parsers, showing that most are not accurate in extracting variable words. Logram is the most accurate parser in terms of word-level parsing accuracy, but its average grouping accuracy is below 50%, making it difficult to apply in the industry to group logs. To meet the needs of different downstream tasks in software-intensive systems, a log parser that performs well in both group accuracy and word-level parsing accuracy is needed.

3) Inefficient Runtime Performance: Software-intensive systems often experience frequent updates and new systems coming online [26], requiring the trained model to adjust parameters or even retrain. Additionally, neural network inference is relatively slow, which may not keep up with log generation speeds during peak periods, and some traditional parsing algorithms have relatively high time complexity. For instance, Alibaba Inc's cloud computing system generates around 120 to 200 million log lines per hour [1]. With such a high log volume, system failure is more likely to have catastrophic repercussions, making the parsing process essential to buy time for failure diagnosis and recovery.

D. Opportunities and New Idea

The above three limitations of existing parsers motivate us to design a new approach for log parsing in software-intensive systems. To design a stable log parser, we delved into the common characteristics of log generation and found a logging statement always generates the same constants (i.e., log template), and a random combination of variables, so that more constant words than variable words are consistent in frequency. Thus, we can group logs more accurately by longest word combinations with the same frequency instead of the first n words. *Specifically, we define that the frequency of a word refers to its occurrence number in all logs. Words with the same frequency in a log form the word combination.* As shown in Fig. 2, "`open through proxy`" is a word combination in log0, as these words occur the same time in all logs (i.e., frequency is 7). Logs with the same length and the same longest word combination are grouped as initial groups, and the longest word combination is the *longest common pattern* for the log group. For example, log0, log2, log3, log4, log8, log9, and log10 are grouped because they have the same longest word

[2]https://www.jenkins.io/

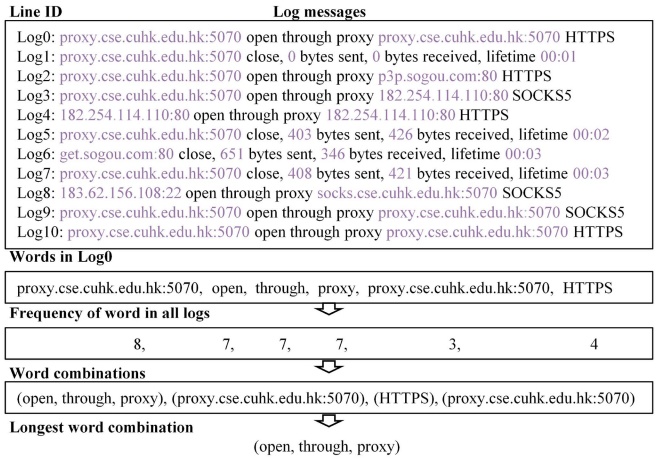

Fig. 2. Example of extracting the longest word combination.

TABLE III
TERMINOLOGY DEFINITIONS

Terminology	Definitions
Frequency of a word	Its occurrence number in all logs
Word combination	Words with the same frequency in a log
Longest word combination	Word combination with most words
Initial group	Logs with the same length and longest word combination
Longest common pattern	Common longest word combination in an initial group

combination: "`open through proxy`". The definitions of these terminology are summarized in Table III.

To design a log parser that performs well on both group accuracy and word-level parsing accuracy, we need to get a complete log template, not just a part (i.e., longest common pattern). This is because different logging statements may generate the same constant words, which interferes with the consistency of frequencies of constant words. In the benchmark datasets, on average, each log template has about 3.15 constant words that are the same as those in other log templates. We summarize two cases where the longest common pattern misses the constant word: *Case 1):* Words with a higher frequency than the longest common pattern could be constant words because these words may be constant words generated by different logging statements and accumulate more frequencies. For example, in Openstack dataset,[3] the longest common patterns of logs belonging to the template "`[instance: <*>] Claim successful`" and "`[instance: <*>] Creating image`" are "`Claim, successful`" and "`Creating, image`". The longest common patterns miss the constant word "`instance`" compared to these two log templates. In this case, we can check if these columns contain different words to identify constant words due to constant position generating the same words. *Case 2):* Words with a less frequency than the longest common pattern also could be constant words because words in the longest common pattern may be generated by different logging statements and accumulate more frequencies. As an example in Fig. 2, the longest common pattern among log0, log2, log3, log4, log8, log9 and log10 "`open through proxy`" misses constant words "`HTTPS`" and "`SOCKS5`". In this case, it is possible for the words in the column to be constant even though the columns contain different words. We use empirical knowledge to set a threshold for the number of words due to the variable position to generate more different words. Furthermore, once a new constant word is added to the longest common pattern, the log group will continue to be divided into new log groups. The complement process is then applied recursively until the grouping is accurate and all the constant words are found.

To efficiently implement the complement process, we use a tree structure to design our algorithm. This is because it can facilitate the implementation of hierarchical algorithms. We need two trees for each initial group. One tree checks for missed constants caused by case 1 (e.g., "`instance`"). The other tree checks for missed constants caused by case 2 (e.g., "`HTTPS`" and "`SOCKS5`"). Words in the same column are at the same depth, and the number of branches represents the number of different words in each column. Therefore, we can determine if a node is constant or variable by checking the number of branches. These two trees have different thresholds for the number of branches. In our implementation, we merge the two trees into a bidirectional tree rooted at the longest common pattern.

A more detailed description of our approach will be presented in the next section.

III. OUR APPROACH

In this section, we will introduce the detail of Brain, which is a **b**idirectional parallel t**r**ee-b**a**sed log pars**in**g approach.

A. Overview of Our Approach

The log parsing process of Brain has five main steps to finish parsing as shown in Fig. 3, including *1) preprocessing, 2) initial group creation*, *3) node update in the parent direction, 4) node update in the child direction and 5) template generation.*

A more detailed description of each step follows:

B. Step 1: Preprocessing

Preprocessing mainly includes word split and common variable filtering. For word split, we will use common delimiters (e.g., "`,=:`") to split the log into separate words. For example, if we choose equal sign "=" and space " " as delimiters, log "`Source file=idotransportmgr.cpp Source line=1043 Function=int`" will be split into 8 different words "`Source`", "`file=`", "`idotransportmgr.cpp`", "`Source`", "`line=`","`1043`", "`Function=`", "`int`". After the split, we will collect the frequency of each word in all logs. In our implementation, we form a tuple for each word. For example, for logs in Fig. 2, Brain will generate the tuple `(7, open, 1)`, which means the word "`open`" appears 7 times in column1. Then, tuples with the same frequency in each log will be combined into word combinations. Table IV shows all the word combinations generated by the

[3]https://github.com/logpai/loghub/tree/master/OpenStack

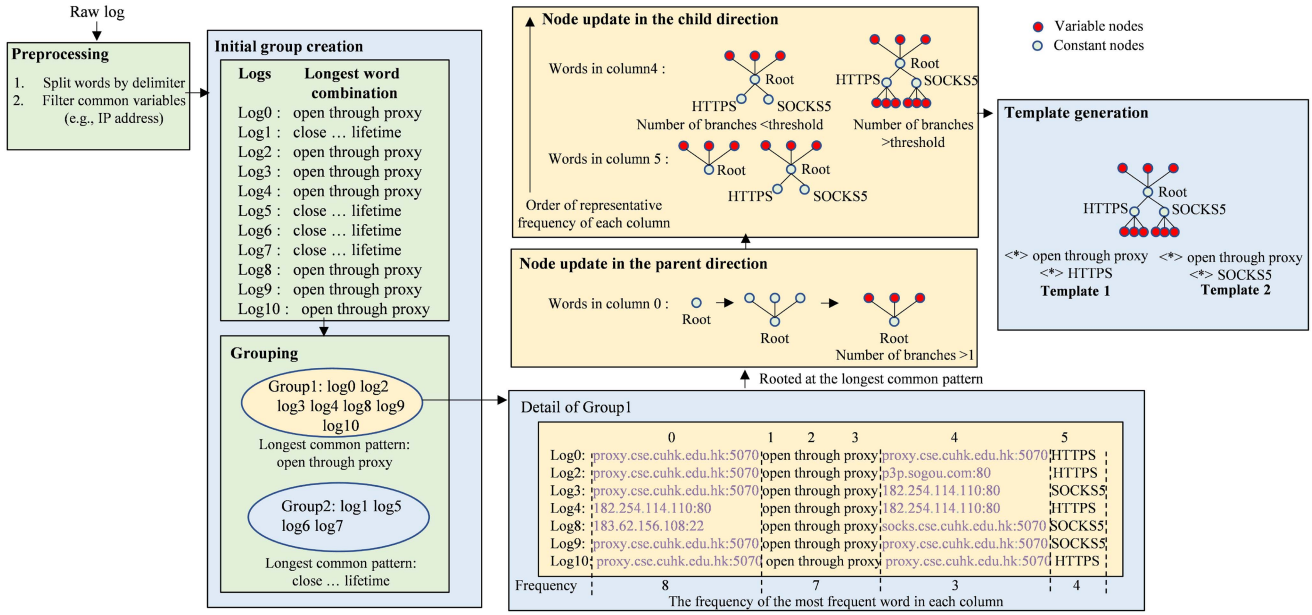

Fig. 3. The workflow of Brain parsing the log messages in Fig. 2.

TABLE IV
WORD COMBINATIONS OF LOG0 IN FIG. 2

Word combination	Length
(8, proxy.cse.cuhk.edu.hk:5070, 0)	1
(7, open, 1), (7, through, 2), (7, proxy, 3)	3
(3, proxy.cse.cuhk.edu.hk:5070, 4)	1
(4, HTTPS, 5)	1

log0 in Fig. 2. For common variable filtering, we use regular expressions set by the operator based on empirical knowledge to match common variables, e.g., IP address and block id. Matched words will be converted to wildcard "<*>". After preprocessing, we can get all word combinations of each log.

C. Step 2: Initial Group Creation

In this section, we will elaborate on how Brain creates initial groups. Since more constant words than variable words are consistent in frequency, the longest word combination in each log is most likely to be part of the log template. However, when some variables have low frequency, they may not interfere with each other in their frequency so that their frequency is consistent and combined into the longest word combination. For example, variables in these two logs "`en0: supported channels 36 40 44 48`" and "`en0: supported channels 35 52 56 60`" will have the same frequency and form word combinations. This situation results in only two word combinations in a log (i.e., variable and constant word combination), and variable word combination has more words than constant word combination. Operators need to find these logs with only two different frequencies and check whether the longest word combination is a variable combination based on empirical knowledge. If it is a variable combination, operators can set a frequency threshold to avoid selecting these variable words.

Therefore, Brain will create initial groups based on the longest word combination selected from word combinations with a frequency greater than a frequency threshold. The threshold is set by multiplying the highest frequency of words in the log by a hyperparameter "weight". For example, if we set the "weight" to 0.5, then the word combination "`(7, open, 1) (7, through, 2) (7, proxy, 3)`" will be selected as the longest word combination of log0 in Fig. 2, since it has the most members (3) and its frequency (7) is greater than the threshold ($0.5 * 8 = 4$), where 8 is the highest frequency of words in log0, i.e., column 0's "`proxy.cse.cuhk.edu.hk:5070`" in Fig. 2. Algorithm 1 shows the implementation details. First, we group logs with the same length and collect the word combinations generated by each log with a frequency greater than a threshold set by empirical knowledge (lines 2-11). Second, each log will select the word combination with the most words as the longest word combination (line 12). Finally, logs with the same longest word combination will be grouped (lines 12-14). After creating initial log groups, the common longest word combination is the longest common pattern in each initial group.

For each initial group, we will create a bidirectional tree rooted at the longest common pattern. Words in the same column will be located at the same depth because they have the same classification. The maximum frequency of a word in a column is considered to be the representative frequency of the column. The two directions of bidirectional tree are called the parent direction and the child direction, where the parent direction is used to check for missing constant words in columns with a representative frequency higher than the root node, and the child direction is used to check for missing constant words in columns with a representative frequency lower than the root node.

Algorithm 1: Initial Group Creation.

```
Input: logs with same length
Output: log groups.
1: log_groups ← {}
2: for log in logs do
3:   for word in log do
4:     tuple_set.add((word_frequency,word
       character,word_column))
5:   end for
6:   word_combination_set=generate (tuple_set) # group
     tuple with same frequency to word combination
7:   for word_combination in word_combination_set do
8:     if word_combination.frequency < threshold then
9:       word_combination_set.remove(word_combination)
10:    end if
11:  end for
12:  log_groups.setdefault
     (word_combination_set.mostwords(),[]). append(log)
13: end for
14: return log_groups
```

Algorithm 2: Tree Building in the Parent Direction.

```
Input: root_node, initial_group
Output: a tree with parent direction nodes.
1: tree ← bitree(root_node)
2: for column in initial_group do
3:   representative_frequency← max(word.frequency for
     word in column);
4:   if representive_frequency >root_node.frequency then
5:     parent_set.add(column)
6:   end if
7: end for
8: for column in parent_set do
9:   if different_words_num(colum) != 1 then
10:    root_node.add_variable_parent(words in column)
11:  else
12:    root_node.add_constant_parent (words in column)
13:  end if
14: end for
15: return tree
```

D. Step 3: Node Update in the Parent Direction

The detail of the node update in the parent direction is shown in Algorithm 2. All words in the column with a representative frequency higher than the root node will be added in the parent direction of the tree (lines 2 - 7). In these columns, if there are variable words, there will be a high probability that different words will appear because variable position always generates different words. As an example in Fig. 2, variable "`proxy.cse.cuhk.edu.hk:5070`" in column0 can generate by various logging statements and accumulate enough frequency to be greater than the root node. However, in an initial group rooted at "`open through proxy`", there are different words in column0 to prove that the words in it belong to variables.

Therefore, we will count the number of different words in each parent column (line 9). If there are different words, words in the column will be added to the tree as variable nodes (lines 8 - 10). If there are multiple columns that satisfy the parent node requirements, we will add and classify nodes in parallel.

E. Step 4: Node Update in the Child Direction

We use Algorithm 3 to introduce in detail the tree building in the child direction. All words in the column with a representative frequency lower than the root node will be added in the child direction of the tree (lines 2 - 7). As in case 2, words in a child column with different words may also be constant words, so that the rule in parent direction cannot be used in this direction. Since variable positions produce more different words than constant positions, columns will be sorted according to the number of different words in them (line 8), and a threshold for the number of different words in each column will be used to classify (line 10). After sorting, the node addition in the child direction can start following the order (line 9). If the number of different words in a column exceeds the preset threshold set based on operators' empirical knowledge, the words in this column will be classified as variable words (lines 10 - 11). If the number of different words in this column does not exceed the preset threshold, words in this column will be classified as constant (lines 12 - 13). Once a new constant word is added to the bidirectional tree, all logs containing words from the root node to this leaf node will generate a new log group, and subsequent child node additions will be based on this new log group (line 14). This hierarchical process will be performed until the grouping is exact and find all the constant words.

In order to give readers a clearer understanding of the tree construction process in the child direction, the detailed process is shown in Fig. 4. Based on logs in Fig. 2, there are two columns with representative frequency lower than the root node, i.e., column4 and column5. The words in column5 will be added to the tree first due to its higher representative frequency. The root node will add two new constant child nodes"`HTTPS`" and "`SOCKS5`" because the number of branches (2) is below the threshold. Then, the initial group will be divided into two subgroups according to the left and right paths, i.e., subgroup1 and subgroup2. Words in column4 of subgroup1 will be added as children of node "`HTTPS`", and words in column4 of subgroup2 will be added as children of node "`SOCKS5`". Since the number of branches (3) of these two constant nodes is greater than the threshold, these new nodes will be classified as variable nodes.

F. Step 5: Template Generation

For each initial group, after the node addition in two directions is complete, the bidirectional parallel tree will output log templates. Each word in the initial group can correspond to a node in the tree, which is labeled as either a constant or a variable.

As an example in Fig. 5, this is a bidirectional tree generated for group1 in Fig. 3, and the classification information of its

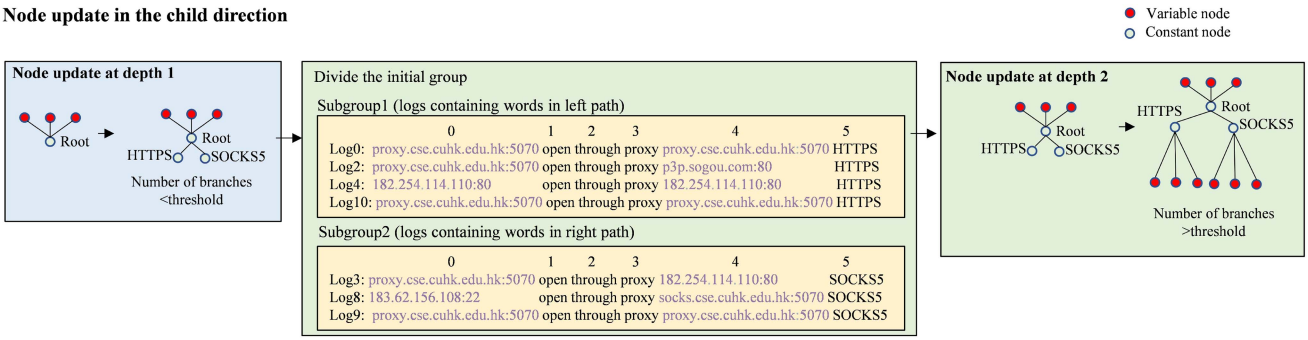

Fig. 4. Flowchart of adding nodes in the child direction, where the data sample is from Fig. 2.

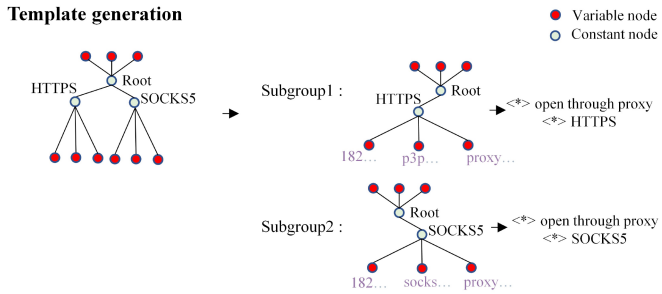

Fig. 5. An example of a template generation.

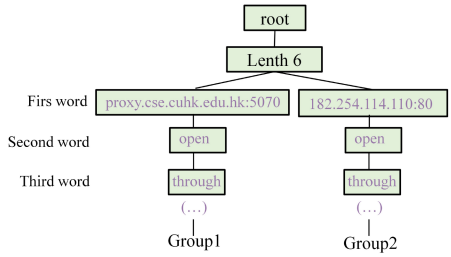

Fig. 6. The parsing process example of Drain.

Algorithm 3: Tree Building in the Child Direction.

Input: root_node, initial_group
Output: a tree with child direction nodes.

```
tree ← bitree(root_node)
for column in initial_group do
    representative_frequency← max(word.frequency for word in column);
    if representive_frequency ≤ root_node.frequency then
        child_set.add(column)
    end if
end for
sorted_candidate ← sort child_set with the number of different words in each column
for (column, num) in sorted_candiate do
    if num ≥ threshold then
        root_node.add_variable_child(words in column)
    else
        root_node.add_constant_child (words in column)
        initial_group will be split into num different sub-group, the algorithm will be called recursively
    end if
end for
return tree
```

nodes can generate into two log templates. The first log template is "`<*> open through proxy <*> HTTPS`", and all logs in subgroup1 belong to this log template. The second log template is "`<*> open through proxy <*> SOCKS5`", and all logs in subgroup2 belong to this log template.

For comparison, Fig. 6 shows the tree generated by Drain for logs in subgroup1. Since the word in the first position of logs in subgroup1 is variable, Drain will mistakenly create new branches for these logs, resulting in different log groups. However, these logs should belong to one group. This kind of situation is very common in the real world, such as log "`multi-web 45 main build action completed: SUCCESS`" from Jenkins, which is a very common software in cloud platforms. The first word of this type of log is changed according to the project name, that is, it is a variable word. In summary, the bidirectional parallel tree structure in Brain makes it possible to avoid these errors.

IV. Evaluation

In this section, experiment detail on the existing 16 benchmark datasets is introduced. How we set up our experiment is to answer the following three research questions:

- RQ1: How effective is Brain on two widely-used accuracy metrics compared with existing log parsing approaches?
- RQ2: How efficient is Brain compared with existing log parsing approaches?
- RQ3: How effective is each module of the Brain?

All our experiments are performed on the most widely-used benchmark datasets published in LogPai. For RQ1, the effectiveness of Brain was evaluated on two different metrics: group accuracy, and word-level parsing accuracy. For RQ2, we conduct a comparative experiment and record the time log parsers take to

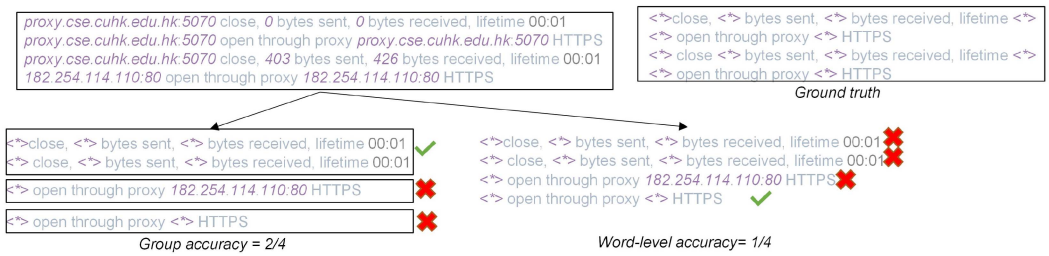

Fig. 7. Illustration of the group accuracy and word-level parsing accuracy.

TABLE V
DATA STATISTICS AND HYPERPARAMETER SETTING

Dataset	Messages	Delimiter	Threshold
HDFS	11,175,629	, :	2
Apache	56,481	,	4
Zookeeper	74,380	, : =	3
Mac	117,283	,	5
HealthApp	253,395	, : = —	4
Android	1,546,686	, : = ( )	5
OpenStack	207,820	,	5
BGL	4,747,963	.. ( ) ,	6
Proxifier	21,329	,	3
Linux	25,567	, : =	4
HPC	433,489	, : - =	5
Hadoop	394,308	: = - ( ) ,	6
Windows	114,608,388	= : [ ] ,	3
Thunderbird	211,212,192	: = ,	3
Spark	33,236,604	, :	4
OpenSSH	655,146	,	6

parse different volumes of logs. For RQ3, we do an ablation study for each modules of Brain and verify the rationality of the way we create initial groups by the longest common pattern. Further, we also performed a sensitivity analysis on the hyperparameters.

A. Experimental Setting

Dataset: Our experiments are conducted on the most widely-used benchmark datasets published in LogPai. The dataset includes distributed system logs (HDFS, Zookeeper, Spark, Hadoop, OpenStack), supercomputer logs (HPC, BGL, Thunderbird), operating system logs (Mac, Linux,Windows), mobile system logs (Android, HealthApp), server application logs (Apache, OpenSSH) and standalone software logs (Proxifier), more detail statistics about the benchmark dataset is available in Table V. And each benchmark dataset provides a 2 K sub-dataset where logs are labeled with the log template they belong to as ground truth. For better reproducibility, we follow the guidelines in [6], the effectiveness evaluation results are generated from 2 K sample logs, which have been labeled with the templates they belong to. Efficiency evaluation results are based on different volume of logs from the original dataset.

Evaluation Metric: The most widely-used parsing accuracy metric (group accuracy) is used in our experiment. Group accuracy (abbreviated as GA in the following) measures the ratio of correctly parsed logs over the total number of logs. A log is considered correctly parsed if its log template corresponds to the same group of logs as the ground truth does. For example, if we parse the log sequence [e1, e2, e2] as [e1, e4, e5], we get GA = 1/3 because the second and third messages are not grouped together. However, GA is not satisfactory in different downstream tasks of log parsing, such as parameter anomaly detection requiring accurate variable extraction. Thus, word-level parsing is also important [7], [11], [27], so we also calculated word-level parsing accuracy proposed in [7]. The word-level parsing accuracy indicates that a log can be considered correctly parsed if and only if both variables and constants in the log are correctly classified. A conceptual comparison of GA and word-level parsing accuracy is shown in Fig. 7. Good GA can benefit log sequence anomaly detection, and good word-level parsing accuracy can benefit parameter anomaly detection. For a more comprehensive evaluation, both metrics are adopted in our experiment.

Implementation and Configuration: We implement Brain based on Python 3.8, AMD Ryzen 7 5800H with Radeon Graphics, 3.20 GHz, 16 GB RAM, x64 Windows 11. The selection of thresholds (child direction branches threshold) and delimiters for each dataset are shown in Table V. Since we provide convenient interfaces for them, these hyperparameters are easy to tune. In our comparative experiments, the hyperparameters of the other parsers were tuned as much as possible to make them perform their best.

Log Parser Selection for Comparison: In our comparative experiment, we choose the most advanced log parser possible. However, there are some approaches that are not open source [11], [28], [29], and their technical details are not described in detail in their paper (e.g., the 96 characters they used to cover the most of tokens formed by their combinations in Uniparser), so it is difficult for us to reproduce their results. Therefore, we chose the latest open source log parser Logram [7] and the log parser that achieved the top 4 group accuracy among the log parsers reproduced in LogPai, i.e., Drain [9], Spell [14], AEL [30], Lenma [10]. (1) *Logram* generates n-gram dictionaries for all logs, finds high-frequency n-gram combinations, then determines log variables and constants. It achieves the best word-level parsing accuracy among open source log parsers. (2) *Drain* is an online log parser and employs a fixed-depth tree structure to assist in dividing logs into different groups. It achieves the best group accuracy among open source log parsers. (3) *Spell* uses the longest common sub-sequence algorithm to extract log templates. (4) *AEL* is an offline log parser which employs a list of specialized heuristic rules. (5) *Lenma* is a clustering-based log parser and focuses on the word length

TABLE VI
GROUP ACCURACY OF 6 PARSERS ON 16 BENCHMARK DATASETS

Dataset	AEL	LenMa	Spell	Drain	Logram	Brain
HDFS	0.998	0.998	**1.000**	0.998	0.940	0.998
BGL	0.758	0.69	0.787	0.963	0.645	**0.998**
HPC	0.903	0.830	0.654	0.887	0.906	**0.937**
Apache	**1.000**	**1.000**	**1.000**	**1.000**	0.314	**1.000**
HealthApp	0.568	0.174	0.639	0.780	0.279	**1.000**
Mac	0.764	0.698	0.757	0.787	0.520	**0.937**
Proxifier	0.518	0.508	0.527	0.527	0.027	**1.000**
Zookeeper	0.921	0.841	0.964	0.967	0.725	**0.989**
Thunderbird	0.941	0.943	0.844	0.955	0.189	**0.975**
Spark	0.905	0.884	0.905	0.920	0.382	**0.983**
Android	0.682	0.880	0.919	0.911	0.791	**0.961**
Linux	0.673	0.701	0.605	0.690	0.147	**0.937**
Hadoop	0.538	0.885	0.778	0.948	0.428	**0.981**
OpenStack	0.758	0.743	0.764	0.733	0.236	**1.000**
Windows	0.690	0.566	0.989	**0.997**	0.695	**0.997**
OpenSSH	0.538	0.925	0.554	0.788	0.430	**1.000**
Average	0.754	0.721	0.751	0.865	0.479	**0.981**

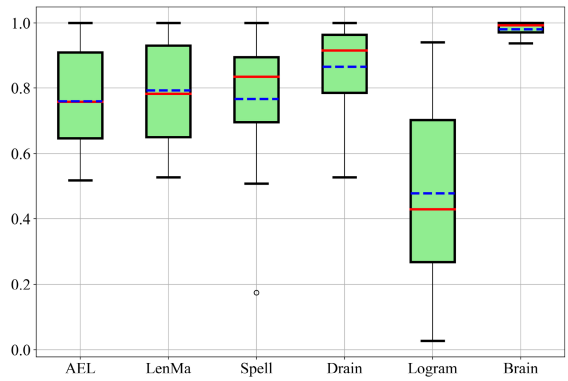

Fig. 8. Boxplot of group accuracy on 16 benchmark datasets.

feature and converts the log into a vector of the number of word letters.

B. *Effectiveness of Brain*

The GA of these 6 log parsers on 16 benchmark datasets is shown in Table VI. Brain achieves an average GA of 0.981, outperforming the state-of-the-art parser (Drain) by 11.6%. In addition, Brain achieves GA of 1.000 on 5 benchmark datasets, and over 0.95 GA on 12 benchmark datasets, both the most.

As shown in Fig. 8, we draw a box plot to show the stability of the parsers more intuitively. For each box, the horizontal solid lines from bottom to top correspond to the minimum, 25-percentile, median, 75-percentile, and maximum GA values, and the blue dotted line represents the average GA of each log parser. We can see that Brain achieves the smallest GA distribution range. For example, Logram only achieves a GA of 0.027 in Proxifier, compared with GA of 0.940 in HDFS, the span reaches 0.913. And no parsers perform well in Proxifier except ours. Further, we analyzed why some log parsers performed poorly on some datasets. Lenma achieves only 0.174 GA on HealthApp, this is because there are many different logs belonging to different log templates

TABLE VII
WORD-LEVEL PARSING ACCURACY OF 6 PARSERS ON 16 BENCHMARK DATASETS

Dataset	AEL	LenMa	Spell	Drain	Logram	Brain
HDFS	**0.999**	0.998	**0.999**	**0.999**	0.981	0.998
BGL	0.818	0.577	0.639	0.822	0.740	**0.921**
HPC	**0.990**	0.915	0.694	0.929	0.959	0.791
Apache	0.693	0.693	0.693	0.693	0.699	**0.994**
HealthApp	0.615	0.141	0.602	0.609	0.969	**0.988**
Mac	0.579	0.551	0.434	0.515	0.666	**0.686**
Proxifier	0.968	0.955	0.785	0.973	0.951	**1.000**
Zookeeper	0.922	0.842	0.955	0.962	0.955	**0.994**
Thunderbird	0.782	0.814	0.773	0.803	0.761	**0.987**
Spark	0.965	0.943	0.865	0.902	0.903	**0.994**
Android	0.867	0.976	0.933	0.933	0.848	**0.963**
Linux	0.241	0.251	0.131	0.250	0.460	**0.863**
Hadoop	0.539	0.535	0.192	0.545	**0.965**	0.859
OpenStack	0.718	0.759	0.592	0.538	0.545	**0.993**
Windows	0.983	0.277	0.978	0.983	0.957	**0.991**
OpenSSH	0.247	0.522	0.507	0.507	0.847	**0.873**
Average	0.748	0.672	0.669	0.748	0.825	**0.929**

TABLE VIII
LABELS THAT DO NOT AFFECT VARIABLE IDENTIFICATION

Ground truth	Our Label
<*>.<*>	<*>
@<*>	<*>
(<*>)	<*>
#<*>	<*>
[<*>]	<*>
<*>K/GB	<*>

but with the same length vector, e.g., "`calculateAltitudeWithCache totalAltitude=<*>`" and `calculateCaloriesWithCache totalCalories =<*>`". Logram does not work at all in Proxifier, this is because the frequency of variable words in this dataset is too high to distinguish them from constant n-grams. The lowest GA of Brain is 0.937 in Mac, which we think is a good performance compared to state-of-the-art parsers. In summary, Brain is able to achieve 14 best out of 16 benchmark datasets, we think it is an effective parser. Experimental results indicate that grouping logs by the longest common pattern is more robust than first n words (Drain), and other assumptions.

For word-level parsing accuracy, we obtained the word-level parsing accuracy of Brain on 16 benchmark datasets through manual labeling. In our manual labeling of word-level parsing accuracy, we consider the variable labeling as shown in Table VIII as correct as it does not affect the operator's perception of these variables. In addition, we consider all preprocessed common variables (e.g., IP address, block ID) to be parsed correctly, for example, the ground truth of "`proxy.cse.cuhk:5070`" is "`<*>:<*>`", our label is" `<*>`", and the ground truth of "`blk_841511351`" is "`blk_<*>`", our label is" `<*>`". In our manual check, we found that some templates in the ground truth seemed to be wrong and we used the correct version, e.g., the ground truth of" `at Fri Jul 22 09:27:24 2005`" should not be" `at <*>:<*>:<*>`". The word-level parsing accuracy of 6 log parsers is shown in Table VII, Brain is able to achieve 13 best out of 16 benchmark datasets and over 0.95 GA on 10 benchmark datasets. From the box plot shown in Fig.

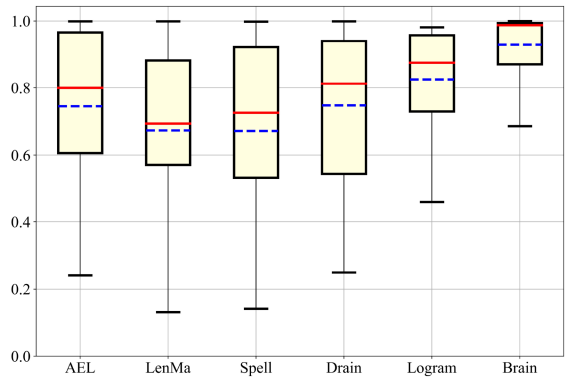

Fig. 9. Boxplot of word-level parsing accuracy on 16 benchmark datasets.

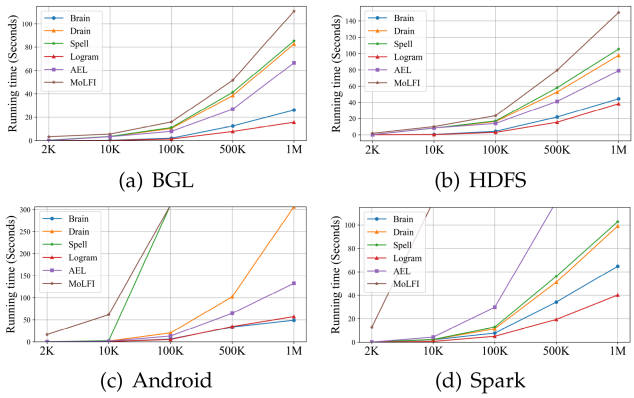

Fig. 10. Time taken by 6 parsers to parse different log volume on 4 benchmark datasets.

9, we can see that Brain also achieves the smallest distribution range, highest mean, and highest median. For the poor results on Mac, this is because there are hundreds of log templates and only one log belongs to them in 2 K benchmark. This poses a great challenge to Brain since it uses the number of different words in each column to determine the classification.

In summary, Brain outperforms state-of-the-art approaches for both traditional GA and word-level parsing accuracy. Therefore, we consider Brain to be an efficient log parser.

C. Efficiency of Brain

A log parser that can be applied to the software-intensive system must have good efficiency. The downstream tasks of log parsing are usually time-consuming because data-driven approaches usually use models with huge parameters to infer the occurrence of failures. The sooner the log parsing is completed, the more time will be bought for failure recovery. In addition, log production during peak periods is not something a slow log parser can handle. We perform an efficiency evaluation of 6 parsers on 4 public benchmark datasets (BGL, HDFS, Android, Spark). The log lines of 2 K, 10 K, 100 K, and 1 M were selected for each dataset, and then we recorded the time each log parser finished parsing. The experimental results are shown in Fig. 10. In these 4 datasets, Logram took the least average time, 37.91 s to process every million logs, Brain takes an average of 46.14 seconds, while Drain takes 146.48 seconds and Spell takes 224.81 seconds. This is mainly because the time complexity of the algorithms in Brain and Logram is almost O(n) and O(m), where n is the number of log lines and m is the number of words. Brain is able to take less time than Logram on the Android dataset, the Android collected in the benchmark dataset claims to contain 76 K templates, which is much more than other datasets, we found that Logram generates 140,000 2-grams and 3-grams for one million Android logs, and then only 900 2-grams and 3-grams in HDFS. We can infer that Brain can perform better on more complex log data. In summary, we think the efficiency of Brian is competitive with the most efficient parser, Logram.

D. Effectiveness of Each Module of Brain

Ablation Study: Brain has two main modules, one is to create initial groups according to the longest common pattern, and the other is to complement the longest common pattern with constant words through a bidirectional parallel tree. We conduct an ablation study on each module, the results on GA and word-level parsing accuracy are shown in Figs. 11 and 12. The GA and word-level parsing accuracy of Brain with only initial grouping on many datasets are not satisfactory in industry, e.g., GA of 0.215 on Spark, word-level parsing accuracy of 0.140 on OpenSSH. Bidirectional trees can lead to great improvements on both GA and word-level parsing accuracy, GA improves on average by 0.154 and word-level parsing accuracy improves on average by 0.201. This comparison shows that initial grouping by the longest common pattern can roughly group logs belonging to the same template together. In addition, the bidirectional tree is useful in complementing constant words, which greatly improves the word-level parsing accuracy, and can group the logs more accurately to obtain a higher GA.

Root Node Selection: As mentioned in Section III-C, if the quality of system available logs are not good enough, an adjustable "weight" can prevent variable words from being the longest common pattern. This situation often arises in online parsing of new systems. However, in our experiment, we evaluated Brain offline and all logs are available, so that "weight" is uniformly set to 0. The probability distribution that the words selected by the root node are all constant words in the 16 benchmark datasets is shown in Fig. 13, and the average accuracy is 0.997. This result indicates that it is a good choice to start parsing by creating initial groups by longest common pattern.

E. Sensitivity Analysis on Hyperparameters

Researchers in [31] study the stability of log parsers with different hyperparameters. The effectiveness of Brain is related to the threshold set by the operator's empirical knowledge. Thus, we conduct a sensitivity analysis on hyperparameters of Brain, including branch number threshold in Section III-E and an adjustable "weight" in Section III-C. For adjustable "weight", the frequency of the selected longest word combination should be greater than or equal to this threshold. Fig. 14 shows the average GA obtained in 16 benchmark datasets with different

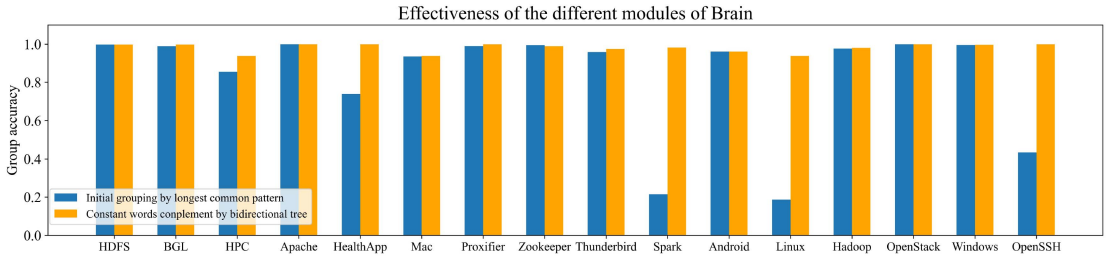

Fig. 11. Group accuracy achieved by different modules of Brain on 16 benchmark datasets.

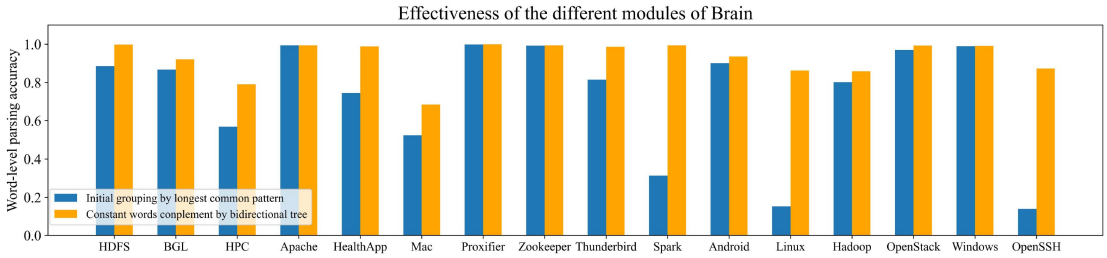

Fig. 12. Word-level parsing accuracy achieved by different modules of Brain on 16 benchmark datasets.

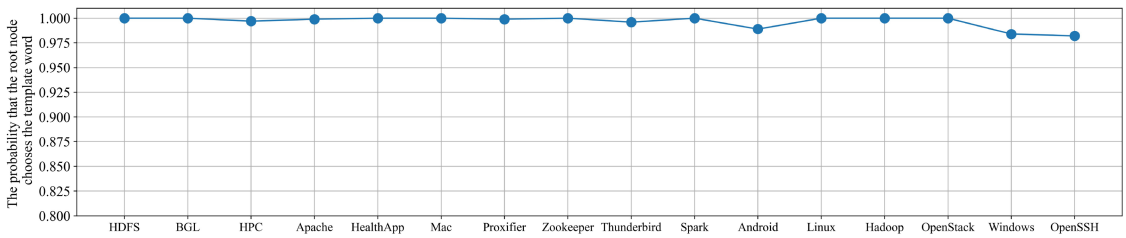

Fig. 13. Probability that all words in the longest common pattern are constant words on 16 benchmark datasets.

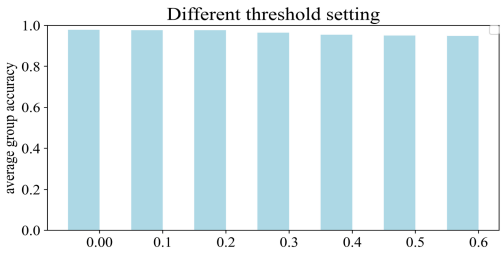

Fig. 14. Average group accuracy when different thresholds are set for the frequency of selecting root node.

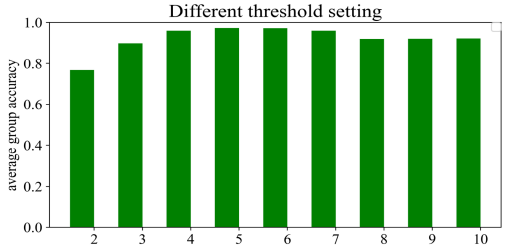

Fig. 15. Average group accuracy when setting different thresholds for child-direction pruning.

thresholds generated by different "weight" (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6). It can be seen that the setting of this threshold does not greatly affect the average accuracy.

For the branch number threshold of child-direction, the average GA achieved on 16 benchmark datasets using different thresholds (2, 3, 4, 5, 6, 7, 8, 9, 10) are shown in Fig. 15. Except when the threshold is set to 2, we can see that Brain's sensitivity to the threshold is not high, because there may be antonyms or synonyms in different log templates with similar structures, such as "`[instance: <*>] VM Started (Lifecycle Event)`" and "`[instance: <*>] VM Stopped (Lifecycle Event)`". Therefore, it is more reasonable to set the threshold value to be greater than or equal to three. As a result, when we gradually increase the threshold from 3 to 10, the average GA of Brain does not change much. We can conclude that Brain is not sensitive to its hyperparameters.

In summary, experiment results indicate that Brain outperforms the state-of-the-art open-source parsers on both GA and

TABLE IX
COMPARISON OF KEY CHARACTERISTICS OF EXISTING APPROACHES

Parsers	Technique	GA	Word-level parsing accuracy	Efficiency	Stability	Open source
Drain	Heuristic	Medium	Medium	Good	Medium	✓
Logram	Frequent item extraction	Low	Medium	Excellent	Low	✓
Lenma	Clustering	Medium	Medium	Low	Low	✓
Uniparser	Neural network	Excellent	Medium	Medium	Medium	✘
Brain	Frequent item extraction & Heuristic	Excellent	Good	Excellent	Good	✓

word-level parsing accuracy, and the time it takes to process a million logs is very comparative among existing log parsers.

V. DISCUSSION

Comparison of Key Characteristics of Log Parsers: Table IX makes a simple summary of some key characteristics of log parsers for application to software-intensive systems. Technique describes what kind of strategy these parsers use, e.g., Brain combines the techniques of frequent item extraction and heuristic. For GA and Word-level parsing accuracy, "Excellent" means that the accuracy rate is higher than 0.95, "Good" means 0.9 - 0.95, "Medium" means 0.6 - 0.9 and "low" means lower than 0.6. Efficiency relates to the time it takes for the log parser to parse one million BGL logs, "Excellent" means that the time is lower than 50 seconds, "Good" means 50 - 100 seconds, "Medium" means 100 - 200 seconds and "low" means higher than 200 seconds. The standard deviation of the two metrics over 16 benchmark datasets is used to measure the stability of the parser. "Excellent" means that the standard deviation value is lower than 0.05, "Good" means 0.1 - 0.05, "Medium" means 0.2 - 0.1 and "low" means higher than 0.2. The experimental data of Uniparser we used is as same as claimed in their paper. The average word-level parsing accuracy of Uniparser on 16 benchmark datasets is 0.785. And it claims to take about 140 seconds to parse one million logs. Statistically, Brain is competitive.

Table IX indicates that Brain is superior in efficiency, effectiveness and stability. However, it is difficult for Brain to be directly applied to new systems because its first component requires high-quality data to create accurate initial groups offline. Section III introduces Brain for offline parsing. Fig. 16 shows the workflow of online Brain. The log will be input in the form of a stream. The frequency of each word in the new-coming logs is calculated based on system available logs. Then, like Drain's fixed-depth tree, our bidirectional parallel tree will perform as a heuristic approach to improve each initial group. Thus, online Brain, like deep learning-based methods, is more useful for systems that have been running for a period of time. How to apply and modify online Brain to new systems effectively is our future work.

VI. THREATS TO VALIDITY

External Validity: Threats to external validity relate to the generalization of experimental results. In this work, we evaluate Brain on the 16 most widely-used benchmark datasets. Brain achieves an average group accuracy 0.981 and an average

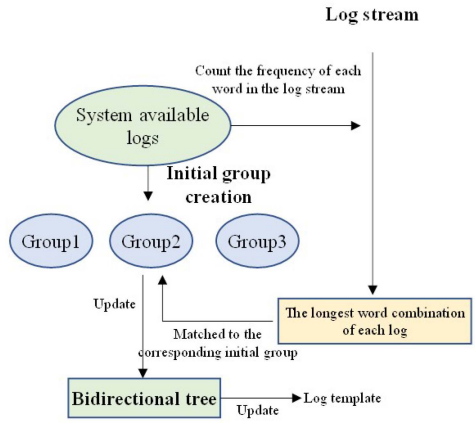

Fig. 16. Workflow of online Brain.

word-level parsing accuracy 0.929. Although the 16 benchmark datasets contain various types of logs, they do not represent that Brain's effectiveness generalizes to unseen industry log types. However, the main idea of Brain is based on the common characteristics of log generation, which will facilitate the adaptation of Brain to new log types. For the efficiency evaluation of Brain, we conducted experiments with different log volumes on 4 benchmark datasets because the log data volume in some existing public datasets is not enough. These four benchmark datasets cannot represent various log types in the industry.

Internal Validity: Threats to internal validity link to potential shortcomings in method design and experimental methodology. It is difficult for Brain to handle logging statements that generate multiple variable words in one variable position because Brain groups logs with the same length into an initial group. In the future, we will pay attention to adding some similarity matching steps to merge log templates with various numbers of variables in one position. In our experiment, the delimiter and regular expression used in the benchmark dataset have been adjusted based on empirical knowledge. However, setting up delimiters and regular expressions for the new system is laborious, and inappropriate delimiters and regular expressions can have an impact on Brain's effectiveness.

Construct Validity: In our evaluation, we evaluate the effectiveness of Brain using two widely-used metrics, group accuracy and word-level parsing accuracy. For rich downstream tasks, more metrics may be used to verify the effectiveness of the log parser, we will consider more metrics [27] in the future. In the effectiveness and efficiency experiments, the compared parsers

do not include all existing parsers, and some non-open source log parsers that are not described in detail in the article are very labor-intensive to reproduce, e.g., Uniparser [11], SPINE [28], Drain+[29].

VII. Conclusion

In this article, we propose a stable and efficient log parser based on the bidirectional parallel tree, called Brain. The first component of Brain is creating initial groups by the longest common pattern, which takes advantage of the common characteristic of log generation. The second component of Brain is to hierarchically complement the longest common pattern with constant words using a bidirectional tree to achieve both good group accuracy and word-level parsing accuracy. Finally, log templates of each initial log group are available on the bidirectional tree. Extensive experiments conducted on the most widely-used benchmark datasets verified the effectiveness and efficiency of Brain. Experimental result indicates that Brain outperforms state-of-the-art parser (Drain) by 11.6% in parsing accuracy, and outperforms state-of-the-art log parser (Logram) by 10.4% in word-level parsing accuracy. And the efficiency of Brain is competitive with the state-of-the-art log parser (Logram), it only takes about 46 seconds for Brain to process 1 million logs. In the future, various automated log analysis tasks in the software-intensive system can benefit from Brain.

References

[1] H. Mi, H. Wang, Y. Zhou, M. -T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1245–1255, Jun. 2013.

[2] X. Zhang et al., "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 807–817.

[3] T. Jia, Y. Wu, C. Hou, and Y. Li, "LogFlash: Real-time streaming anomaly detection and diagnosis from system logs for large-scale software systems," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 80–90.

[4] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–37, 2021.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 117–132.

[6] J. Zhu et al., "Tools and benchmarks for automated log parsing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2019, pp. 121–130.

[7] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 879–892, Mar. 2022.

[8] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proc. IEEE 7th Work. Conf. Mining Softw. Repositories*, 2010, pp. 114–117.

[9] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. IEEE Int. Conf. Web Serv.*, 2017, pp. 33–40.

[10] K. Shima, "Length matters: Clustering system log messages using length of words," 2016, *arXiv:1611.03213*.

[11] Y. Liu et al., "UniParser: A unified log parser for heterogeneous log data," in *Proc. ACM Web Conf.*, 2022, pp. 1893–1901.

[12] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-supervised log parsing," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2020, pp. 122–138.

[13] S. Yu, N. Chen, Y. Wu, and W. Dou, "Self-supervised log parsing using semantic contribution difference," *J. Syst. Softw.*, vol. 200, Art. no. 1116462023.

[14] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. IEEE 16th Int. Conf. Data Mining*, 2016, pp. 859–864.

[15] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *Proc. IEEE/ACM 36th Int. Conf. Automated Softw. Eng.*, 2021, pp. 492–504.

[16] S. Zhang et al., "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *Proc. IEEE/ACM 25th Int. Symp. Qual. Serv.*, 2017, pp. 1–10.

[17] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. IEEE 9th Int. Conf. Data Mining*, 2009, pp. 149–158.

[18] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[19] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.

[20] X. Huang et al., "UDA-DP: Unsupervised domain adaptation for software defect prediction," 2023.

[21] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2018, *arXiv: 1808.01400*.

[22] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *Proc. IEEE/IFIP 46th Annu. Int. Conf. Dependable Syst. Netw.*, 2016, pp. 654–661.

[23] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proc. IEEE/ACM 26th Int. Conf. Prog. Comprehension*, 2018, pp. 167–16710.

[24] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "LogSed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *Proc. IEEE 10th Int. Conf. Cloud Comput.*, 2017, pp. 447–455.

[25] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1285–1298.

[26] R. Chen et al., "LogTransfer: Cross-system log anomaly detection for software systems with transfer learning," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, 2020, pp. 37–47.

[27] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand, "Guidelines for assessing the accuracy of log message template identification techniques," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1095–1106.

[28] X. Wang et al., "Spine: A scalable log parser with feedback guidance," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1198–1208.

[29] Y. Fu et al., "Investigating and improving log parsing in practice," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1566–1577.

[30] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *Proc. IEEE 8th Int. Conf. Qual. Softw.*, 2008, pp. 181–186.

[31] H. Dai, Y. Tang, H. Li, and W. Shang, "PILAR: Studying and mitigating the influence of configurations on log parsing," 2023.

Siyu Yu received the BS degree from Guangxi University. He is currently working toward the master's degree in the School of Computer and Electronic Information with Guangxi University, Guangxi, China, supervised by professor Ningjiang Chen. His research lies within Software Engineering, with special interests in software log analysis.

Pinjia He (Member, IEEE) received the BEng degree in computer science from the South China University of Technology, Guangzhou, China, in 2013 and the PhD degree in computer science from the Chinese University of Hong Kong, Hong Kong, China, in 2018. He worked as a postdoctoral scholar with the Department of Computer Science with ETH Zurich from 2018 to 2021. He is currently an Assistant Professor in the School of Data Science, The Chinese University of Hong Kong, Shenzhen. His research interests include software reliability engineering, AIOps, software testing, and natural language processing. He received the most influential paper award from ISSRE. He received the first IEEE Open Software Services Award.

Ningjiang Chen (Member, IEEE), received the PhD degree from the Institute of Software, Chinese Academy of Sciences, in 2006. He is currently a professor with Guangxi University. His research interests include intelligent software engineering, Big Data, and cloud computing, etc.

Yifan Wu received the BS degree from the University of Electronic Science and Technology of China, in 2015. He is currently working toward the PhD degree in the School of Software & Microelectronics at Peking University, in China. His current research interests include AIOps, software engineering.