A Search-based Approach for Accurate Identification of Log Message Formats

Messaoudi, Salma ; Panichella, Annibale; Bianculli, Domenico; Briand, Lionel; Sasnauskas, Raimondas

DOI
10.1145/3196321.3196340

Publication date
2018

Document Version
Accepted author manuscript

Published in
Proceedings of the 26th International Conference on Program Comprehension

A Search-based Approach for Accurate Identification of Log Message Formats

Salma Messaoudi
University of Luxembourg
Luxembourg
salma.messaoudi@uni.lu

Annibale Panichella
University of Luxembourg
Luxembourg
a.panichella@tudelft.nl

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Lionel Briand
University of Luxembourg
Luxembourg
lionel.briand@uni.lu

Raimondas Sasnauskas
SES
Luxembourg
raimondas.sasnauskas@ses.com

ABSTRACT

Many software engineering activities process the events contained in log files. However, before performing any processing activity, it is necessary to parse the entries in a log file, to retrieve the actual events recorded in the log. Each event is denoted by a log message, which is composed of a fixed part—called *(event) template*—that is the same for all occurrences of the same event type, and a variable part, which may vary with each event occurrence. The formats of log messages, in complex and evolving systems, have numerous variations, are typically not entirely known, and change on a frequent basis; therefore, they need to be identified automatically.

The *log message format identification problem* deals with the identification of the different templates used in the messages of a log. Any solution to this problem has to generate templates that meet two main goals: generating templates that are not too general, so as to distinguish different events, but also not too specific, so as not to consider different occurrences of the same event as following different templates; however, these goals are conflicting.

In this paper, we present the MoLFI approach, which recasts the log message identification problem as a multi-objective problem. MoLFI uses an evolutionary approach to solve this problem, by tailoring the NSGA-II algorithm to search the space of solutions for a Pareto optimal set of message templates. We have implemented MoLFI in a tool, which we have evaluated on six real-world datasets, containing log files with a number of entries ranging from 2K to 300K. The experiments results show that MoLFI extracts by far the highest number of correct log message templates, significantly outperforming two state-of-the-art approaches on all datasets.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196340

KEYWORDS

log parsing, log analysis, log message format, NSGA-II

ACM Reference Format:
Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A Search-based Approach for Accurate Identification of Log Message Formats. In *Proceedings of ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension (ICPC '18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196340

1 INTRODUCTION

Logging is a programming practice that is used for gathering run-time information of a software system. Developers carry out logging by inserting into the source code of an application statements that specify which messages and which run-time information to print into the entries of log files.

Logging is a pervasive activity: recent studies [35, 36] show that between 1/30 and 1/58 of the lines of code in large software systems correspond to logging statements. Furthermore, the importance of logging is also recognized by developers: a recent survey reported that 96% of a group of experienced developers from a leading software company "strongly agree/agree that logging statements are important in system development and maintenance" [13]. Indeed, the information contained in log files can be used for a variety of purposes, such as process mining [15, 30], anomaly detection [4, 12, 14], behavioral differencing [14], fault localization [33], invariant inference [5], performance diagnosis [21], and offline trace checking [3].

All these activities carry out some sort of log analysis, which processes the events corresponding to the entries contained in the log files. Before performing any processing activity, it is necessary to parse the log entries, to retrieve the actual events recorded in the log. A log entry typically includes a timestamp (which records the time at which the logged event occurred) and the actual log message (containing run-time information associated with the logged event). An example of log entry is the following:

```
20050605-06.45.36 send RST CORE to addr 0x0000df30
```

The log message part of a log entry (e.g., the block "`send RST CORE to addr 0x0000df30`" in the above example) is a block of free-form text, which poses a challenge to parsing because it does not have a structured format. More specifically, a log message is

composed of two parts: 1) a fixed part, also called *(event) template*[1], which is the same for all occurrences of the same event type; 2) a variable part, which may vary with each event occurrence, containing tokens filled at run time with dynamic information. In the above example, the template contains the fixed words "`send`", "`to`", "`addr`", while "`RST`", "`CORE`" and "`0x0000df30`" are variable tokens. A template is represented as "`send * * to address *`", where the asterisks indicate placeholders for tokens of the variable part.

The lack of a structured format for log messages leads to the definition of the *log message format identification*[2] problem: given a log file, we want to identify the different templates used in the log messages contained in the log in order to enable automated data extraction and analysis on a large scale.

Solving this problem for complex and evolving systems requires to tackle several challenging issues. First, in these systems log message formats are numerous, changing on a frequent basis (e.g., in Google systems hundreds of new logging statements are added every month [34]), and are typically not entirely known by those who need to analyze log files.

Second, these systems can produce around 120–200 million log entries per hour [20]. Therefore, log message formats need to be identified *automatically* and in a *scalable* way.

Such requirements rule out the use of regular expressions for extracting the templates, since it would still require a manual effort, to create and update regular expressions based on the logging statements contained in the source code of the application. Manual creation and update of regular expressions would be a tedious and error-prone task, given the number of logging statements and the fast pace of their updates [34]. Another strategy would be to statically analyze the application source code, locate logging statements, and extract the templates from the print operations. However, the definition of the static analysis would be tedious and require an extensive knowledge of logging techniques, since logging statements can take different forms in different programming languages and logging frameworks. Furthermore, both strategies outlined above would require to access the source code, which is not always possible, especially in the case of complex software systems that rely on 3rd-party components.

To overcome these limitations, some approaches [11, 12, 17, 19] adopt a black-box strategy that relies on a combination of clustering and heuristic rules to group words into templates, based on their similarity and the frequency within log message blocks. However, our experience on real-world logs shows that these approaches yield low accuracy, as demonstrated by the empirical results reported in this paper. Furthermore, their parameters (e.g., text similarity) need to be fine-tuned for each log to analyze, usually following a trial-and-error process; these requirements make such approaches neither scalable nor effective.

Independently from the specific strategy adopted, any technique for extracting message templates from logs ought to meet two objectives: the generated templates should 1) match as many log messages as possible (i.e., achieve high *frequency* in matching log messages); 2) correspond to the largest extent possible to a particular type of event (i.e., achieve high *specificity*). However, these two objectives—high frequency and high specificity—are conflicting. A template achieving high frequency will contain many tokens in the variable part (to match many log messages), but will be too generic (i.e., it will match messages corresponding to different events); on the other hand, a template achieving high specificity will have a few or no tokens in the variable part (to be able to distinguish between different event types), but it will match only few messages.

Given the presence of conflicting objectives and the limitations of existing solutions [11, 12, 17, 19], in this paper we propose *to recast the log message format identification problem as a multi-objective optimization problem*, where *frequency* and *specificity* are explicitly considered as two competing objectives to optimize simultaneously. Our approach, named MoLFI (Multi-objective Log message Format Identification), leverages an evolutionary approach to solve this problem. MoLFI applies the Non-dominated Sorting Genetic Algorithm II (NSGA-II [10]) on a given log file to search the space of solutions for a Pareto optimal set of message templates. The two main strong points of MoLFI are: 1) it does not require access to the source code of the application producing the log(s) being analyzed, since it is a black-box technique that works only on the log files; 2) different from existing approaches, it does not require any parameter tuning before its execution.

We implemented MoLFI in a prototype tool. We evaluated the accuracy and efficiency of MoLFI on one proprietary and five publicly-available real-world datasets, containing log files with a number of entries ranging from 2K to 300K; we also compared our approach with IPLoM [19] and Drain [17], two state-of-the-art approaches. The results show that MoLFI achieves by far the highest precision and recall, outperforming the other approaches with substantial improvements in both precision (ranging between +14 *pp* and +86 *pp*, with *pp*=percentage points) and recall (ranging between +25 *pp* and +75 *pp*) on all datasets, while keeping a running time of less than 126 s when analyzing the largest dataset. A higher accuracy in the identification of log message formats usually has practical implications, in terms of effectiveness, in the log analysis tasks that rely on log message format identification. For example, in the context of anomaly detection—the original motivation for existing work [12, 17]—log analysis is effective only when the parsing accuracy is high enough [16].

To summarize, the main contributions of this paper are: 1) the formulation of the log message format identification problem as a multi-objective optimization problem; 2) the MoLFI approach for the solution of this problem, based on the NSGA-II algorithm; 3) a publicly-available implementation of MoLFI[3]; 4) the empirical evaluation, in terms of accuracy and efficiency, of the implementation of MoLFI and its comparison with two state-of-the-art approaches.

The rest of the paper is organized as follows. Section 2 gives an overview of multi-objective optimization and genetic algorithms. Section 3 illustrates the problem of log message format identification with an example. Section 4 describes how MoLFI tailors NSGA-II to solve the log message identification problem. Section 5 reports on the evaluation of MoLFI. Section 6 discusses practical

[1] Additional names used in the literature for denoting the fixed part of a log message are "line pattern", "log key", and "message signature".

[2] This problem is often called "log parsing" in the literature; we believe "log message format identification" is a more specific term, since "log parsing" includes also parsing more structured elements like timestamps and log verbosity levels.

[3] The evaluation artifacts are available from the following links:
- tool `https://github.com/SalmaMessaoudi/MoLFI.git`;
- log files `https://github.com/SalmaMessaoudi/ICPC-2018-Artifacts.git`.

implications, alternative solutions, and limitations of our approach. Section 7 examines related work. Section 8 concludes the paper and gives directions for future work.

2 BACKGROUND

This section summarizes basic concepts of multi-objective optimization and briefly describes NSGA-II [10].

2.1 Multi-objective optimization problems

A multi-objective problem is an optimization problem that involves *multiple* objective functions.

Let S be the space (set) of all feasible solutions and F be a vector-valued objective function $F\colon S \to \mathbb{R}^k$ composed of k real-valued objective functions $F = (f_1, \ldots, f_k)$, where $f_i\colon S \to \mathbb{R}$ for $j = 1, \ldots, k$; a multi-objective optimization problem is defined as $\max(f_1(x), \ldots, f_k(x))$ subject to $X \subseteq S$. In other words, the problem consists in finding a set of feasible solutions that maximize the objective functions in F.

The goodness of a solution in a multi-objective optimization problem is defined in terms of the *dominance relation* and *Pareto optimality*. More precisely, a solution X is said to *dominate* another solution Y, denoted as $X \prec Y$, if and only if for all indices $i \in \{1, \ldots, k\}$, $f_i(X) \geq f_i(Y)$ and $f_j(X) > f_j(Y)$ for at least one index $j \in \{1, \ldots, k\}$. A solution X is called *Pareto optimal* if there does not exist another solution in the search space that dominates it. The set of all Pareto optimal solutions of a given problem is called *Pareto front*. The Pareto front can be used to decide which solution to select, according to the preferences of a decision maker.

2.2 NSGA-II

The Non-dominated Sorting Genetic Algorithm II (NSGA-II) [10] is a well-known and efficient technique to solve multi-objective problems. NSGA-II is a multi-objective genetic algorithm (GA) that provides well-distributed Pareto fronts and good performance when dealing with up to three objectives [10, 18]; it has been widely used in software engineering to solve problems involving multiple objectives [32] and with chromosome representations that require complex data structures (as in our case, see Section 4.2.2).

In NSGA-II (and GAs in general), the candidate solutions to a problem are called *chromosomes*. The encoding of a chromosome depends on the type of problem to solve. GAs refine and evolve randomly-generated chromosomes through subsequent iterations (called *generations*), mimicking selection and reproduction mechanisms in nature.

NSGA-II starts with a pool of randomly generated chromosomes (i.e., *population*). In each generation, the algorithm evaluates the goodness of a chromosome in the current population based on the objectives to optimize. Chromosomes dominating other chromosomes are considered as *fitter* solutions and therefore have higher chances to be selected for reproduction (i.e., for generating new chromosomes). NSGA-II selects the best solutions (parents) within the current population by using *binary tournament selection* [10]. Reproduction is performed by combining pairs of parents to form new chromosomes (called *offsprings*) using two operators: *crossover* and *mutation*. The crossover operator generates two offsprings by exchanging some chromosome parts between the two parents. The mutation operator applies small changes to each offspring to get a more diverse solution. Notice that the implementation of mutation and crossover depends on the problem to solve. The new population for the next generation is formed by selecting the fittest individuals among parents and offsprings according to the dominance relation (non-dominated ranking) and *crowding distance* (to promote diversity) [10]. The process of selecting and recombining chromosomes is repeated multiple times, once for each generation. It terminates either when a given amount of generations is reached or when a time-out occurs. The non-dominated solutions contained in the population of the last iteration represent the final Pareto front.

```
1  20050605-06.45.36 INFO generating core 135
2  20050605-06.45.36 INFO generating core 198
3  20050605-06.45.36 INFO generating core 199
4  20050605-06.45.36 FATAL instruction address 0x0000df30
5  20050605-06.45.36 FATAL instruction address 0x0000f450
6  20050605-06.45.36 FATAL machine state register 0x00003000
7  20050605-06.45.36 FATAL wait state enable 0
8  20050605-06.45.36 FATAL critical input interrupt enable 0
9  20050605-06.45.36 FATAL external input interrupt enable 0
10 20050605-06.45.36 FATAL problem state (0=sup,1=usr)
11 20050605-06.45.36 FATAL floating point instr. enabled 1
12 20050605-06.45.36 FATAL machine check enable 1
13 20050605-06.45.37 FATAL rts internal error
14 20050605-06.45.37 FATAL rts panic - stopping execution
15 20050605-06.59.14 FATAL data TLB error interrupt
```

Figure 1: Excerpt (simplified) of real-world log entries

3 THE PROBLEM OF LOG MESSAGE FORMAT IDENTIFICATION

We illustrate the problem of log message format identification through the example in Figure 1, which provides a simplified excerpt of log entries extracted from an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs.

One can see that the log messages of the first three entries in the example log correspond to the same event type. This event type could be matched with the template ⟨`INFO generating core *`⟩, where the variable part contains one token (indicated with the placeholder `*`). Similarly, entries of log messages at lines 4–5 could be matched with the template ⟨`FATAL instruction address *`⟩.

However, one could define other templates for the log messages considered above. For example, another template that could match the messages at lines 4–5 would be ⟨`FATAL * * *`⟩, with three tokens in the variable part. Notice that this template is more general than the previous one, since it matches two different types of event (one type associated with messages at lines 4–5, and another type associated with the message at line 13). Another possible template would be ⟨`FATAL * address 0x0000df30`⟩, which is too specific because it matches only the log message at line 4 and misses the message at line 5, even if it is of the same event type.

These examples show that two distinct objectives must be met when identifying message templates:

- maximizing the number of log messages matched by each template, i.e., maximizing the *frequency* of message matches;
- maximizing the *specificity* of a template to a particular type of event.

These two goals are conflicting: to maximize frequency, templates should contain many tokens in the variable part (to match many log messages); however, such templates would have a low specificity (i.e., they would be too generic), matching messages corresponding to different events. On the other hand, to maximize specificity, templates should contain only a few or no tokens in the variable part (to be able to distinguish between different event types); however, they would match only few messages.

Any method proposed to solve the log message format identification problem has to deal with the trade-off between these two conflicting goals.

4 LOG MESSAGE FORMAT IDENTIFICATION AS A MULTI-OBJECTIVE OPTIMIZATION PROBLEM

In this section, we illustrate how log message format identification can be recast as a multi-objective optimization problem and present our approach MoLFI for the solution of this problem, based on NSGA-II.

4.1 Problem Formulation

As discussed in section 3, we consider *frequency* and *specificity* as objective functions to optimize simultaneously. The multi-objective optimization formulation of the log message format identification problem entails that we find, from the set S of all feasible solutions, a set of templates $X = \{\tau_1, \ldots, \tau_n\}$, $X \subseteq S$, such that each template $\tau_i \in X$ with $i = 1, \ldots, n$, matches as many log messages as possible (high frequency) and contains as few variable tokens as possible (high specificity). More formally, the objective functions are the *frequency*: $Freq(X) = \sum_{i=1}^{n} \frac{|match(\tau_i, M)|}{n \times |M|}$, and the *specificity:* $Spec(X) = \sum_{i=1}^{n} \frac{fixed(\tau_i)}{n \times tok(\tau_i)}$, where n is the number of templates in X, M is a list of log messages, $match(\tau, M)$ denotes the list of log messages in M that match a template τ, $fixed(\tau)$ denotes the number of tokens in the fixed part of τ, $tok(\tau)$ denotes the total number of tokens in τ.

When determining a solution to this problem, there are two important aspects to assess. First, the templates contained in a (Pareto optimal) solution may not match all the log messages in M. For example, the solution $X = \{\, \langle$`FATAL instruction address 0x0000df30`$\rangle$, $\langle$`INFO generating core 135`$\rangle\,\}$ is Pareto optimal for the log messages in Figure 1, since it has the highest possible specificity ($Spec(X) = 1$). However, the templates in X match only two out of the 15 log messages ($Freq(X) = \frac{2}{15}$). Second, two different templates τ_1 and τ_2 in the same solution X may match the same log messages, i.e., $match(\tau_1, M) \cap match(\tau_2, M) \neq \emptyset$. To avoid this type of solutions, we introduce two additional constraints to the optimization problem to determine the set of *feasible solutions* S. More specifically, a solution $X = \{\tau_1, \ldots, \tau_n\} \subseteq S$ is *feasible* if it satisfies the following constraints:

$$\bigcup_{i=1}^{n} match(\tau_i, M) = M \tag{1}$$

$$match(\tau_i, M) \cap match(\tau_j, M) = \emptyset \text{ for all } \tau_i, \tau_j \in X, \tau_i \neq \tau_j \tag{2}$$

4.2 MoLFI

To solve the multi-objective optimization formulation of the log message format identification problem, we introduce our approach, named MoLFI, which tailors the standard NSGA-II to our context. In particular, we detail the encoding schema and the genetic operators (i.e., crossover and mutation) we use, the pre- and post-processing procedures we apply, and the procedure we follow to select one solution from the Pareto front.

4.2.1 Pre-processing. Before starting the search process, we first pre-process the log messages to improve the accuracy of the process; we follow the guidelines by He et al. [16, 17]. We first use regular expressions to identify trivial variable parts within the log messages based on domain knowledge, e.g., numbers, memory and IP addresses. Strings in the log messages matching these regular expressions are replaced with a special variable token `#spec#` that cannot be mutated in the later stages of the search. To reduce the computation cost of the template identification process, we filter out duplicated log messages, reducing the number of messages to consider for generating templates. The messages are then tokenized, using blanks, parentheses and punctuation characters as word-separators. Finally, messages are grouped into buckets, with each bucket containing messages that have the same number of tokens; we denote with M_L the bucket/group containing messages with exactly L tokens.

4.2.2 Encoding Schema. In our context, a solution is a set of templates $X = \{\tau_1, \ldots, \tau_n\}$ where each template τ_i corresponds to a group of pre-processed log messages having the same length and sharing all fixed tokens in τ_i. Therefore, each template τ_i is a list of tokens, where each token can be either variable (denoted by the symbols * or `#spec#`) or fixed (i.e., the tokens identified during the pre-processing step).

Although very intuitive, this encoding schema is not efficient for computing the log messages being matched by each template. Indeed, this procedure requires comparing every template against *all* log messages even if most of them have a number of tokens not compatible with what is prescribed by the template. To speed-up the matching process, we design a two-level encoding schema: a chromosome C is a set of groups $C = \{G_1, \ldots, G_{max}\}$, where each group $G_L = \{\tau_1, \ldots, \tau_k\}$ is a set of templates having the same number of tokens L. This encoding schema guarantees that the matching procedure is applied only for messages and templates of the same length.

Figure 2 shows an example of chromosome for the log messages in Figure 1 based on our encoding schema. It has four groups of templates with lengths 4, 5, 6, and 12; it also satisfies the constraints for feasible solutions.

4.2.3 Initial Population. MoLFI uses the algorithm *InitialPopulation* (Algorithm 1) for generating the initial population. The algorithm takes as input a set of pre-processed log messages M, the population size N; it returns a population P. Each chromosome is randomly generated inside the loop at lines 4–16: after initializing the chromosome C (line 4), it is iteratively filled with groups of templates (lines 5–15), one group of templates G_L for each group of pre-processed log messages $M_L \in M$ with the same length L.

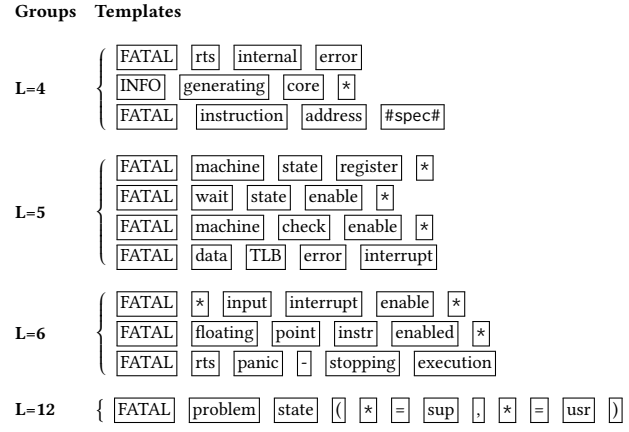

Figure 2: An example of chromosome for the log messages in figure 1.

Algorithm 1: InitialPopulation

Input: Set of pre-processed log messages M
Population size N

Result: Initial population P

```
1  begin
2      P ← ∅
3      while |P| < N do
4          C ← ∅
5          for each group M_L ∈ M do
6              G_L ← create an empty group for templates with length L
7              unmatched ← M_L
8              while |unmatched| > 0 do
9                  log_message ← randomly select one message from
                     unmatched
10                 τ ← copy(log_message)
11                 index ← random integer ∈ [1; L]
12                 τ[index] ← "*"
13                 G_L ← G_L ⋃ {τ}
14                 unmatched ← unmatched \ match(τ, M_L)
15             C ← C ⋃ {G_L}
16         P ← P ⋃ {C}
```

For each group of messages $M_L \in M$, the algorithm creates a corresponding group of templates G_L (line 6). Initially, the group G_L is empty and therefore it does not match any log message. The algorithm keeps track of the unmatched messages in the set *unmatched*, initialized with M_L (line 7). Then, a log message is randomly selected from *unmatched* (line 9) and used to generate a template τ (lines 10–12). Template τ is a copy of the original log message with the exception of one single token (randomly selected at line 11), which is replaced with the variable token "$*$" (line 12). The newly generated template is then added to the group G_L and used to update the set of unmatched log messages (line 14). The loop at lines 8–14 terminates when the templates composing the group G_L match all log messages in M_L (i.e., when the set *unmatched* is empty). Since this condition has to be satisfied for each group of messages $M_L \in M$, the chromosome C is a feasible solution. Therefore, Algorithm 1 guarantees that all chromosomes in the initial population satisfy our constraints.

4.2.4 Crossover. We implemented the *uniform crossover*, which is one of the most popular crossover operators [26, 27]. It generates two offsprings by shuffling the different characteristics (groups of templates in our case) of the parents. Let $A = \{A_1, \ldots, A_{max}\}$ and $B = \{B_1, \ldots, B_{max}\}$ be the two selected parents where each pair of groups $A_L \in A$ and $B_L \in B$ matches the same pre-processed log messages $M_L \in M$ with length L. The uniform crossover first generates a random binary vector β (called the crossover mask) with a length equal to the number of groups in A and B. Then, the two offsprings O_1 and O_2 are obtained as follows: when the binary element in β for the group with length L is zero, offspring O_1 inherits group A_L while O_2 inherits group B_L; otherwise, O_1 inherits group B_L while O_2 inherits group A_L.

Notice that this crossover operator swaps groups of templates between the two parents without changing the set of templates composing each group. Therefore, it generates offsprings that are feasible solutions: each group $A_L \in A$ and $B_L \in B$ covers all pre-processed log messages $M_L \in M$ and they do not contain overlapping templates (i.e., templates that match the same log messages). Since A_L and B_L are not modified by our crossover, the properties above are preserved independently from which offspring inherits the two groups.

4.2.5 Mutation. After crossover, offsprings are mutated using the mutation operator to randomly change the generated templates. Given a chromosome to mutate $C = \{G_1, \ldots, G_{max}\}$, each group $G_L = \{\tau_1, \ldots, \tau_k\}$ is mutated with probability $\frac{1}{max}$. A group G_L is mutated by changing one of its templates; the template is mutated by adding or removing variable tokens. In particular, let $\tau = [token_1, \ldots, token_n]$ be the template to mutate; each token is mutated with probability $\frac{1}{n}$. The token $token_i$ is mutated as follows: if it is a fixed one, it is replaced by the variable token "$*$"; if it is a variable token, it is replaced by a fixed token, which is randomly selected among all fixed tokens in position i of the log messages that match τ; if it is the special token #spec# added during the pre-processing, it is not mutated. Therefore, our mutation operator either increases or reduces the number of variable tokens in τ. In the former scenario, it likely increases the frequency of the original template τ; in the latter case it increases its specificity.

Different from the crossover, the mutation operator changes the templates within the chromosome's groups. Therefore, it does not guarantee that the mutated chromosomes satisfy the feasible solution constraints. For this reason, we developed a *correction operator* that (i) removes overlapping templates (i.e., two or more templates matching the same pre-processed log messages), and (ii) adds randomly generated templates if a mutated group G_L does not match all messages in M_L. Random templates are added following the same procedure used at lines 7–14 of Algorithm 1. Notice that the *correction operator* is applied after the mutation operator and it is applied only to the mutated chromosome's groups.

4.2.6 Post-processing. At the end of the search, NSGA-II returns a set of feasible solutions that are Pareto optimal, i.e., representing optimal trade-off between frequency and specificity. Due to the random nature of NSGA-II, Pareto optimal solutions may contain log message templates with spurious variable tokens, i.e., variable tokens that have been inserted by mutation across the generations

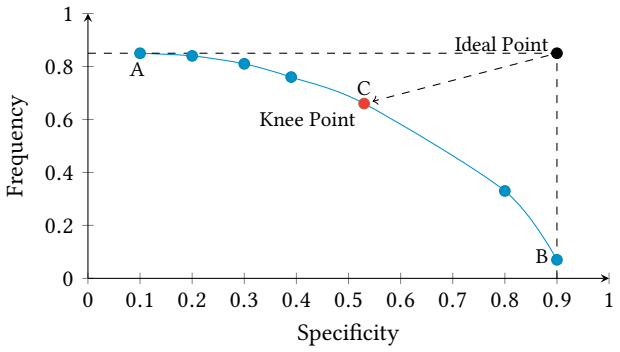

Figure 3: The concepts of Pareto front and knee point.

but that do not contribute to match more pre-processed log messages. For this reason, MoLFI post-processes the templates in each Pareto optimal chromosome with a greedy procedure, which iteratively removes all variable tokens that do not affect the frequency scores. In other words, given a template τ to post-process, the procedure temporarily removes one of its variable tokens and checks whether the set of log messages matched by τ remains unchanged. If the applied change affects the set of matched log messages, the change is reverted; otherwise it is maintained. The post-processing procedure ends once all variable tokens in τ have been verified.

4.2.7 Choosing a Pareto optimal solution. If the number of solutions in the generated Pareto front is large it may be difficult to choose one solution (best trade-off) among the different alternatives. For this reason, researchers proposed various guidelines to find and suggest points of interest in the Pareto front, such as the *knee points* [6], *mid points* [22], or the best point (*corner*) for each objective [23].

According to Branke et al. [6], the most interesting Pareto optimal solution is the knee point because any other solution in the front leading to a small improvement in one of the two objectives (e.g., *Freq*) would lead to a large deterioration in the other objective (e.g., *Spec*). To provide a graphical interpretation of the knee point, Figure 3 depicts an example of Pareto front for the log message format identification problem. The Pareto front is composed of seven non-dominated solutions: points A and B are the corner solutions of the front while the other solutions represent intermediate trade-offs. Point C can be considered as a knee point since any marginal improvement to *Freq* will correspond to a large deterioration in *Spec*, and vice versa. Therefore, the knee point leads to the lowest loss in both objectives.

To identify the knee point, we measure the distance of each Pareto optimal solution from the *ideal point* [26]. The coordinates of the ideal point correspond to the maximum objective values among all solutions in the Pareto front, considering each objective function separately. For example, for the Pareto front in Figure 3, the ideal point has the coordinates (F_{max}, S_{max}), where $F_{max} = 0.85$ (from point A) and $S_{max} = 0.9$ (from point B). More formally, given a Pareto front $P = \{C_1, \ldots, C_p\}$, the knee point $C_k \in P$ is the solution minimizing the distance $\sqrt{(F_{max} - Freq(C_i))^2 + (S_{max} - Spec(C_i))^2}$, for all $C_i \in P$.

5 EVALUATION

We have implemented the MoLFI approach as a Python program. In this section we report on the evaluation of the effectiveness of the MoLFI implementation in identifying accurate log message formats.

First, we want to assess the performance of MoLFI in comparison with state-of-the-art techniques, in terms of accuracy and efficiency. Second, there are various factors that may influence the effectiveness of MoLFI, such as (1) the number of templates to identify, (2) the population size in NSGA-II; (3) the removal of duplicate messages from the log file to analyze, performed as part of the pre-processing step; we want to understand whether and to what extent these factors affect the effectiveness of MoLFI. Last, in MoLFI we choose the knee point as most valuable solution from the Pareto front, following the general guidelines by Branke et al. [6]. However, different trade-offs in the Pareto front may provide equal or better results in our context; hence, we want to assess whether the knee point is the best Pareto optimal solution for the log message format identification problem.

Summing up, we investigate the following research questions:

RQ1: *How does MoLFI perform when compared to state-of-the-art techniques for the log message format identification problem?*
RQ2: *Which factors impact the effectiveness of MoLFI?*
RQ3: *Is the knee point the best solution to choose from the Pareto front?*

5.1 Benchmark

To evaluate MoLFI, we used a benchmark composed of six different datasets: five datasets are publicly available and have been used in previous work on the log message format identification problem [16, 17], while the last one is industrial and proprietary.

The five public datasets are HDFS, BGL, HPC, Zookeeper (shortened to "ZK") and Proxifier (shortened to "PRX"). HDFS consists of logs from the Hadoop file system that were collected from a 203-node cluster on the Amazon EC2 platform [16]. BGL contains logs generated from the Blue Gene/L (BGL) supercomputer, collected by the Lawrence Livermore National Labs (LLNL) [16]. The logs contained in the HPC dataset were collected from a high-performance cluster with 49 nodes and thousands of cores [16]. The logs in ZK were collected by He et al. [16, 17] from a 32-node cluster. PRX consists of logs generated by a standalone software [16].

The proprietary dataset (named *PR*) has been provided by one of our industrial partners, active in the aerospace industry; it contains logs produced by a complex system with more than 20 distributed processes.

All datasets contain log files of various size. For the HDFS, BGL, HPC, ZK, and PRX datasets, we used the same samples of 2K log entries used in previous studies [16, 17]. In addition, we also selected a sample of 100K log entries from BGL and a sample of 60K log entries from HDFS. As for the proprietary dataset, we considered three different log files, generated by three different sub-systems, containing 2K, 20K, and 300K log entries.

Ground truth definition. In the case of the log message format identification problem, the ground truth is represented by the actual log message templates. For our evaluation, we established the ground truth as follows.

For the log files with 2K log entries of the public datasets, we used the ground truth defined by He et al. [16, 17] and publicly available from their replication package. In the case of the BGL and HPC datasets, the original set of correct templates contains some mistakes, e.g., templates with unbalanced parentheses and missing punctuation marks. Therefore, we manually validated and fixed them before performing our evaluation.

No ground truth is available for the proprietary logs, the 100K log file from BGL, and the 60K log file from HDFS. Therefore, we had to manually establish the ground truth. Two validators independently inspected each log file and extracted the corresponding templates. Then, the two sets of templates independently extracted by the two validators were merged into a single ground truth set, by including only the templates extracted by both validators. Templates identified by only one of the two validators were discussed and further added to the ground truth only upon agreement between the validators. At the end of the validation process, we also verified that no log message in our datasets could be matched by more than one single template in the ground truth. In total, 486K log messages were manually inspected to establish the ground truth. The number of log message templates in each log file ranges from 13 (PRX) to 394 (PR with 20K messages).

5.2 Effectiveness of MoLFI

To answer RQ1, we assess the performance of MoLFI, in terms of accuracy and efficiency, in comparison with DRAIN [17] and IPLoM [19], which are the two most recent and effective tools for the log message format identification problem [16, 17]. We use the implementation of DRAIN available in [17] and the one of IPLoM available in [16].

5.2.1 Methodology. The NSGA-II algorithm used in MoLFI requires to set four parameters: crossover probability, mutation probability, population size, and stopping condition. To set these parameters, we followed the guidelines proposed in the literature. More specifically, Arcuri and Fraser [1] and Sayyad et al. [25] have empirically demonstrated that the benefits of fine-tuning the parameters of search-based algorithms often do not compensate for the required overhead; both studies recommend to use the default parameters values, since they provide competitive results.

We set the NSGA-II parameters as follows:

- *crossover probability* $p_c = 0.70$, since the recommended values are within the interval $0.45 \leq p_c \leq 0.95$ [7, 8];
- the *mutation probability* p_m is proportional to the length of the chromosome (see section 4.2.5), as recommended in the related literature [10];
- the *population size* is set to 20 individuals; according to our preliminary experiments (see section 5.3.2), this small value corresponds to the best compromise between accuracy and efficiency;
- the *stopping condition* is set to 200 generations [10].

As selection operator, we used binary tournament selection [10], which is based on dominance and crowding distance.

For DRAIN, in the case of the public datasets, we used the same parameters values used in [17]; in the case of our proprietary dataset, we used the default parameter values: *depth* = 4, *similarity* = 0.5. We also pre-processed the logs, as suggested in [17], to identify trivial variable parts within the log messages based on domain knowledge.

For IPLoM, we used the default parameter values used in [19]: *file support threshold* = 0, *partition support threshold* = 0, *upper bound* = 0.9, *lower bound* = 0.25, and *cluster goodness threshold* = 0.35.

We ran the three tools on each log file in our benchmark and collected the generated log message templates. We measured the accuracy of each tool by comparing the set of generated templates with the ground truth. Furthermore, we measured the wall-clock time for executing the complete program (including pre- and post-processing tasks for MoLFI). To measure the accuracy, we used the metrics used in previous studies [16, 17], i.e., $Precision = \frac{|CRT \cap GEN|}{|GEN|}$, $Recall = \frac{|CRT \cap GEN|}{|CRT|}$, and $F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$, where *GEN* denotes the set of templates generated by a tool and *CRT* denotes the set of templates generated by a tool which are correct, i.e., conform to the ground truth.

To account for the random nature of NGSA-II, we executed MoLFI 50 times on each log and computed the median and standard deviation of the effectiveness metrics; DRAIN and IPLoM were executed only once due to their deterministic nature. However, DRAIN generated duplicated templates, i.e., multiple templates having exactly the same fixed and variable tokens. To avoid any bias due to duplicated templates, we detected and removed them before computing the various effectiveness metrics.

Furthermore, we used the Welch's t-test to verify whether the F-measure scores achieved by MoLFI are significantly higher than those achieved by the alternative tools. The Welch's t-test is a test for statistical significance suitable for distributions with different variance. In our case, the variance for DRAIN and IPLoM is zero as they are deterministic; MoLFI may return a non-zero variance due to NSGA-II. For this test, we consider a level of significance α=0.05. Other than simply testing the statistical significance, we estimated the magnitude of the differences (effect size) using the Vargha-Delaney ($\hat{A}_{12}$) statistic [31]. $\hat{A}_{12}$ takes values in [0; 1]; $\hat{A}_{12} > 0.50$ values indicate that MoLFI outperforms the alternative tool while for $\hat{A}_{12} < 0.50$ the contrary is true. $\hat{A}_{12} = 0.50$ if the two tools are equivalent.

5.2.2 Results. Table 1 shows the results of the three tools, grouped by dataset and log file size. Column "#T" indicates the number of templates in a file; column "NTT" indicates the percentage of templates with more than one variable token in a file; columns "Prec", "Rec", "F-m", "T" indicate, respectively, the precision, recall, F-measure, and the execution time in seconds. For MoLFI, the table reports the median results achieved across the 50 runs as well as the corresponding standard deviation values.

According to our results, MoLFI obtains, on all the log files in the benchmark, a better F-measure than both DRAIN and IPLoM.

We compared the effectiveness of our approach with the two state-of-the-art tools and we present the differences in percentage points (*pp*). The difference between MoLFI and DRAIN in terms of F-measure ranges between +13 *pp* and +36 *pp*. The values for the difference are always statistically significant according to the Welch's t-test (all *p*-values are lower than 0.01) and the effect size is always *large* (i.e., $\hat{A}_{12} \approx 1$). This difference is due both to better precision and to better recall. We also remark that DRAIN crashed

Table 1: Precision (Prec), recall (Rec), F-measure (F-m), and execution time (T(s)) of the three approaches.

Dataset	Size	#T	NTT	Drain				IPLoM				MoLFI			
				Prec	Rec	F-m	T(s)	Prec	Rec	F-m	T(s)	Prec	Rec	F-m	T(s)
BGL	2K	114	31%	0.55	0.51	0.53	0.54	0.47	0.46	0.46	0.28	0.83 ±0.03	0.86 ±0.03	0.84 ±0.03	17.38 ±0.32
	100K	57	22%	0.60	0.74	0.66	32.89	0.42	0.53	0.47	5.76	0.83 ±0.03	0.78 ±0.04	0.80 ±0.03	10.91 ±0.11
HDFS	2K	16	93%	0.86	0.75	0.80	0.29	0.86	0.75	0.80	0.27	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	3.36 ±0.02
	60K	62	43%	0.79	0.68	0.73	2.11	0.56	0.45	0.50	2.87	0.94 ±0.02	0.95 ±0.01	0.94 ±0.01	10.32 ±0.09
HPC	2K	42	33%	0.49	0.61	0.54	0.32	0.37	0.52	0.43	0.27	0.92 ±0.02	0.88 ±0.04	0.90 ±0.03	7.56 ±0.32
PR	2K	286	14%	0.60	0.64	0.62	0.27	0.52	0.51	0.51	0.40	0.77 ±0.04	0.82 ±0.03	0.79 ±0.03	36.80 ±0.65
	20K	394	17%	0.61	0.57	0.59	1.51	0.54	0.47	0.50	2.38	0.71 ±0.03	0.82 ±0.02	0.76 ±0.02	79.59 ±2.86
	300K	52	80%	–	–	–	–	0.08	0.13	0.10	49.49	0.94 ±0.01	0.88 ±0.00	0.91 ±0.01	125.84 ±4.14
PRX	2K	13	61%	0.46	0.46	0.46	0.21	0.45	0.38	0.42	0.26	0.77 ±0.00	0.77 ±0.00	0.77 ±0.00	3.46 ±0.09
ZK	2K	47	30%	0.77	0.77	0.77	0.22	0.47	0.46	0.46	0.28	0.93 ±0.03	0.87 ±0.01	0.90 ±0.02	6.94 ±0.26

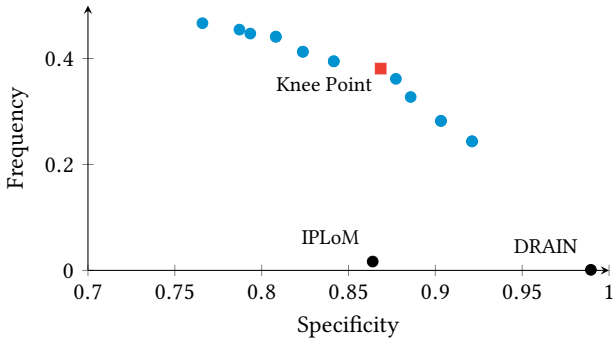

Figure 4: Pareto front generated by MoLFI and the objectives scores of the templates generated by DRAIN and IPLoM

on the 300K log from the proprietary dataset, without yielding any message template.

The difference between MoLFI and IPLoM in terms of F-measure ranges between +20 *pp* and +81 *pp*. For all logs in our study, the differences are statistically significant (p-values are always lower than 0.01) with a *large* effect size ($\hat{A}_{12} \approx 1$). Also in this case, the better F-measure is ascribable to the substantial improvements in both precision (ranging between +14 *pp* and +86 *pp*) and recall (ranging between +25 *pp* and +75 *pp*). An interesting case is represented by the 300K log from the proprietary dataset: MoLFI generates very accurate templates achieving an F-measure of 0.91 while IPLoM obtains a very low F-measure of 0.1.

Figure 4 shows an example of the Pareto front generated by MoLFI for the dataset HPC on one single run. It also displays the knee point (red point) and two further points (in black color) corresponding to the frequency and specificity values of the templates generated by DRAIN and IPLoM. The knee point dominates the templates produced by IPloM, meaning that MoLFI generates templates having both better frequency and better specificity. Instead, the templates produced by DRAIN are non-dominated neither by the knee point nor by the other Pareto optimal solutions. Indeed, their objective scores are located in one of the corners of the Pareto front, meaning that their specificity is very high (few variable tokens) but their frequency is very low. Similar results are obtained also for the other logs in the benchmark. To sum up, both IPLoM and DRAIN are not able to provide optimal compromises between the two objective functions.

In terms of efficiency, MoLFI is the slowest technique; this can be explained because of the usage of NSGA-II, which is an iterative algorithm. The fastest technique is IPLoM, which, however, is also the one with the lowest F-measure values. DRAIN is faster than MoLFI in all the cases with the only exception of the 100K log from the BGL dataset: for this file DRAIN takes 32.89 s while MoLFI takes only 10.91 s. Although MoLFI takes longer to converge than state-of-the-art tools, the increment of the running time has no practical implications since it took less than 126 s when analyzing the largest dataset.

5.3 Factors influencing the effectiveness of MoLFI

To answer RQ2, we investigate the effect of the following three factors on the effectiveness of MoLFI: (1) the number of templates to identify in a log file, (2) the population size in NSGA-II; (3) the removal of duplicate messages from the log file to analyze, performed as part of the pre-processing step. In the following, we illustrate the evaluation methodology and the results for each of these factors.

5.3.1 Number of templates. To test the effect of this factor, we used the one-way permutation test [2] to assess whether the number of templates to identify in a log file statistically interacts with the F-measure scores achieved by MoLFI. The permutation test is a non-parametric test and therefore it does not assume that the data are normally distributed. We ran this test with a very large number of iterations (i.e., 10^8), as suggested in the literature [2].

According to the one-way permutation test, there is no interaction between the number of templates to be identified in a log file

and the F-measure values obtained by MoLFI (p-value=0.08). This means that our technique yields high F-measure scores both with log files containing a low number of templates (e.g., see the 2K log from PRX in Table 1) and with log files containing a high number of templates (e.g., see the 20K log from PR in Table 1).

5.3.2 Population size. Given the nature of NSGA-II, using a large population size may significantly increase the execution time for finding the best solutions; however, using a population with few individuals may yield poor results. We test the effect of this factor by running MoLFI (on each log file of the benchmark) with a population size of 40 and 80 individuals. We repeated each run ten times and computed the median F-measure and execution time. We compared these results with those obtained by the baseline (with a population of 20 individuals, see Table 1).

Table 2 shows the results of this comparison. Column "T" indicates the median execution time in seconds; column "R" is the ratio between the execution time achieved by the new configurations and the baseline; column "F-m" is the median F-measure; column "$\Delta_{\text{F-m}}$" indicates the difference, in percentage points, between the F-measure achieved by the new configurations and the baseline. All these values are shown for the columns labeled "pop=40" and "pop=80" of Table 2.

In terms of F-measure, MoLFI performs almost equivalently for the three configurations, with an increase for the majority of log files reaching 4 *pp* for the 2K log file from BGL. We remark two exceptions where the F-measure decreased with a population size of 80: the 300K log file from PR (-5 *pp*) and the BGL log file of size 100K (-1 *pp*).

Execution time sharply increases as the population size grows. This is to be expected since a larger population size entails more fitness computations for NSGA-II.

5.3.3 Removal of duplicate messages. In the pre-processing step presented in section 4.2.1, we filter out duplicated log messages, to reduce the number of log messages to consider for generating templates. However, such a reduction may also directly affect the value of one of our two objective functions, *frequency*, which could be further reflected in changes to the shape of the Pareto front and to its knee point.

To test the effect of this factor, we ran MoLFI by disabling the routine responsible for removing duplicated log messages in the pre-processing step. As above, each run was repeated ten times and we computed the median F-measure and execution time, as well as the ratio between execution times and the difference in percentage points of the F-measure. The results are shown in the column "no filtering" of Table 2. No data are reported for the 300K log from the PR dataset, since it timed-out (> 3 hours) when completing the first generation of NSGA-II.

We compare these effectiveness scores with those reported in Table 1 (i.e., with filtering enabled). We observe that the F-measure scores obtained by the two configurations are the same for all log files, with only 1 *pp* increase for the BGL dataset. These results show that filtering out duplicated log messages during pre-processing does not significantly alter the final F-measure. However, it results in a substantial reduction of the execution time. For example, when the filter is enabled, MoLFI requires 126 s to converge for the largest log file (the 300K log file from the PR dataset), while it times out (after three hours) for the same log file when the filter is disabled.

Table 2: Comparison between three different configurations of MoLFI: with population size of 40 (column "pop=40") and 80 (column "pop=80") individuals, and without filtering the duplicated log messages (column "no filtering"). "T": median execution time in seconds, "R": ratio of execution time values, "F-m": median F-measure, $\Delta_{\text{F-m}}$: difference of F-measure in percentage points

Dataset	Size	pop=40				pop=80				no filtering			
		T (s)	R	F-m	$\Delta_{\text{F-m}}$ (*pp*)	T (s)	R	F-m	$\Delta_{\text{F-m}}$ (*pp*)	T (s)	R	F-m	$\Delta_{\text{F-m}}$ (*pp*)
BGL	2K	35.84	2.06	0.86	2	70.97	4.08	0.88	4	25.80	1.48	0.85	1
	100K	17.88	1.64	0.80	0	31.90	2.92	0.79	−1	397.76	36.46	0.81	1
HDFS	2K	6.36	1.89	1.00	0	12.48	3.71	1.00	0	13.21	3.93	1.00	0
	60K	18.26	1.77	0.94	0	34.32	3.33	0.94	0	175.09	16.97	0.94	0
HPC	2K	15.33	2.03	0.90	0	29.75	3.94	0.92	2	15.43	2.04	0.90	0
PR	2K	75.44	2.05	0.80	1	162.37	4.41	0.79	0	45.39	1.23	0.79	0
	20K	157.88	1.98	0.78	2	315.11	3.96	0.76	0	217.43	2.73	0.76	0
	300K	155.58	1.24	0.91	0	235.69	1.87	0.86	−5	>3h	–	–	–
PRX	2K	6.55	1.89	0.77	0	12.88	3.72	0.77	0	14.47	4.18	0.77	0
ZK	2K	13.86	2.00	0.90	0	26.56	3.83	0.90	0	15.05	2.17	0.90	0

5.4 Is the knee point the best solution?

To answer RQ3, we analyze, over the entire benchmark, the F-measure scores achieved by all solutions in the Pareto front. This means comparing the F-measure of the knee point with the scores achieved by the other Pareto optimal solutions. For the sake of analysis, for each log, we selected one single Pareto front among those obtained with 50 independent runs. For the selection, we first computed the F-measure for the knee point generated in each run; then, we selected the knee point having the median F-measure across the runs and its corresponding Pareto front.

Figure 5 shows, for all the log files in our benchmark, the F-measure of the knee points (indicated with red points) when compared with all the Pareto optimal solutions (represented by the boxplots). One can see that, in all cases, the F-measure score of the knee point is located at the very top of the boxplot. This confirms our conjecture that, in the context of the log message format identification problem, the knee point is the best solution to choose from the Pareto front.

6 DISCUSSION

Practical implications. As discussed in Section 5.2, MoLFI achieves a substantial higher accuracy than alternative algorithms. Such improvements in accuracy represent a considerable reduction in the time needed by analysts to inspect the generated templates, validate them and eventually modify the incorrect ones. For example, MoLFI generates 62 templates for HDFS with the 60K log; on average across runs, 60 templates are correct (as they match the ground truth). One incorrect template is T=⟨`PacketResponder * for block * *`⟩, which is too general as it matches log messages belonging to two different log events: (i) when the `PacketResponder` for a given block terminated correctly and (ii) when it has been interrupted. The log messages for these two log events are very similar as they

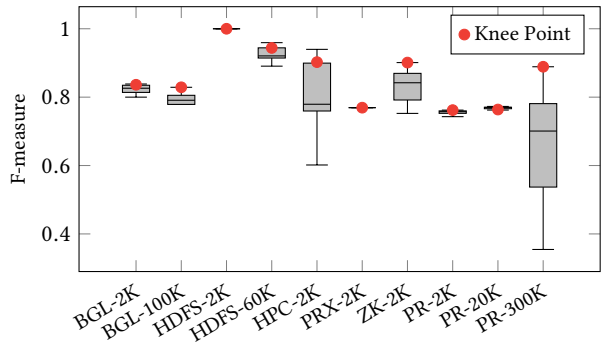

Figure 5: Comparisons between the knee-point and other Pareto front solutions in terms of F-measure

differ only by one single token. Fixing this template would need to create two templates, each one with an additional fixed token. Since T matches only those two log events, fixing it is trivial.

Sum-scalarization vs. multi-objective search. An alternative search-based solution to solve our multi-objective problem would be applying *sum scalarization* [9]. Such a strategy combines the objectives to optimize into one single function by using the sum operator and thus enabling the use of a single-objective genetic algorithm to optimize the aggregated function. In our case, the aggregation function combines frequency and specificity, i.e., $f(X) = Freq(X) + Spec(X)$. To assess this alternative search strategy, we ran a classical genetic algorithm to optimize the function $f(X)$ mentioned above on HDFS with the 60K log. The median F-measure obtained over 10 independent runs is 0.65 (± 0.02), which is statistically significantly lower than the value achieved by MoLFI (i.e., NSGA-II and the knee point), which is 0.94. Note that for the single-objective genetic algorithm we use the same parameter values as for NSGA-II.

The role of constraints. In our problem formulation, we consider two constraints: (i) each log message has to be matched by only one template in a solution X; and (ii) the templates in X have to match all log messages (100% coverage). The former constraint is straightforward since templates in a given solution X should not overlap; the latter is less intuitive but analysts, for some specific applications, could be interested in solutions not covering all log messages. However, we observed that the solutions obtained when removing the coverage constraint have only one single template. For example, if we run MoLFI on HDFS with 60K logs by disabling the coverage constraint, NSGA-II returns a knee point which is a solution with only one template having one variable token and 12 fixed tokens. Such a template has high frequency (*Freq*=0.06) and high specificity (*Spec*=12/13=0.94). No other template is included in the solution because adding any other template would penalize both frequency and specificity.

Limitations. Our approach may produce incorrect results because of the method we use to group messages. In particular, log messages whose variable part has a variable composition (e.g., because of a variable-length argument list), could lead to different templates even if they have the same fixed part.

7 RELATED WORK

Researchers have proposed various black-box techniques for the log message format identification problem. These techniques rely on clustering [12, 28], heuristics [19, 29], longest common sequence method [11], and textual similarities [17]. Recently, He et al. [16] carried-out an empirical study comparing four techniques for the log message format identification problem: SLCT [29], LKE [12], LogSig [28], and IPLoM [19]. The results of this study revealed that (i) IPLoM produces the most accurate templates, and (ii) log pre-processing (like the one applied in MoLFI) is very critical to achieving good clustering performance. In a later study, He et al. [17] introduced DRAIN, a novel technique that processes the log messages through a fixed-depth parse tree. Their empirical study showed that DRAIN generates more correct templates than IPLoM.

One limitation of the two best techniques (DRAIN and IPLoM) is that they require their parameters to be tuned for each log file. Such parameters, if not chosen carefully, will significantly affect the performance of the tool. Different from state-of-the-art techniques, our approach MoLFI uses an automated heuristic to automatically choose the best compromise between the two objectives of the log message format identification problem (frequency and specificity) without using a priori, user-defined thresholds. Our evaluation results show that MoLFI significantly outperforms both IPLoM and DRAIN, both in precision and in recall.

8 CONCLUSION AND FUTURE WORK

The *log message format identification problem* deals with the identification of the different templates used in the log messages. In this paper, we formulated this problem as a multi-objective optimization one, where the goal is to generate log message templates with high frequency (i.e., they match as many log entries as possible) and high specificity (i.e., specific for each log event). To tackle the problem, we introduced MoLFI, a tool implementing a search-based approach based on a multi-objective genetic algorithm and trade-off analysis.

An empirical study involving six real-world datasets (five publicly-available and one proprietary) showed that MoLFI (i) achieved significantly higher accuracy than DRAIN and IPLoM, two state-of-the-art tools; (ii) is highly scalable to large logs since it requires slightly above two minutes to analyze hundreds of thousands of messages.

As part of future work, we plan to improve the effectiveness of MoLFI by investigating other encoding schemas, experimenting with other formulations of the problem (e.g., by introducing the coverage of log messages as another optimization objective), and by handling semantically equivalent templates. We also plan to assess the use of MoLFI for supporting various software maintenance and testing activities, such as boosting test case generation techniques through the definition of new seeding strategies [24] based on input and output values (variable parts) observed in the logs.

ACKNOWLEDGMENTS

This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), from the Luxembourg National Research Fund (FNR) under grant agreement No C-PPP17/IS/11602677, and from a research grant by SES.

REFERENCES

[1] Andrea Arcuri and Gordon Fraser. 2013. Parameter Tuning or Default Values? An Empirical Investigation in Search-Based Software Engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[2] Rose D. Baker. 1995. Modern Permutation Test Software. In *Randomization Tests, Third Edition*. Marcel Dekker, New York, NY, USA, 391–401.

[3] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. 2014. Scalable Offline Monitoring. In *Proceedings of the 5th International Conference on Runtime Verification (RV 2014) (LNCS)*, Vol. 8734. Springer, Cham, Switzerland, 31–47.

[4] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. 2017. Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)*. IEEE, Piscataway, NJ, USA, 351–360.

[5] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. ACM, New York, NY, USA, 267–277.

[6] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. 2004. Finding Knees in Multi-objective Optimization. In *Proceedings of the 8th International Parallel Problem Solving from Nature (PPSN 2004) (LNCS)*, Vol. 3242. Springer, Berlin, Heidelberg, 722–731.

[7] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. 2006. Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-time Systems. *Genetic Programming and Evolvable Machines* 7, 2 (2006), 145–170.

[8] Helen G. Cobb and John J. Grefenstette. 1993. Genetic Algorithms for Tracking Changing Environments. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA 1993)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 523–530.

[9] Kalyanmoy Deb. 2014. Multi-objective Optimization. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Second Edition*. Springer, New York, NY, USA, 403–449.

[10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[11] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *Proceedings of the16th IEEE International Conference on Data Mining (ICDM 2016)*. IEEE, Piscataway, NJ, USA, 859–864.

[12] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM 2009)*. IEEE Computer Society, Los Alamitos, CA, USA, 149–158.

[13] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 24–33.

[14] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience Report: Log-Based Behavioral Differencing. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)*. IEEE, Piscataway, NJ, USA, 282–293.

[15] Christian W. Günther and Wil M. P. van der Aalst. 2007. Fuzzy Mining-adaptive Process Simplification based on Multi-perspective Metrics. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007) (LNCS)*, Vol. 4714. Springer, Berlin, Heidelberg, 328–343.

[16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2016. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2016)*. IEEE, Piscataway, NJ, USA, 654–661.

[17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *Proceedings of the International Conference on Web Services (ICWS 2017)*. IEEE, Piscataway, NJ, USA, 33–40.

[18] Vineet Khare, Xin Yao, and Kalyanmoy Deb. 2003. Performance Scaling of Multi-objective Evolutionary Algorithms. In *Proceedings of the 2nd International Conference on Evolutionary Multi-criterion Optimization (EMO 2003) (LNCS)*, Vol. 2632. Springer-Verlag, Berlin, Heidelberg, 376–390.

[19] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2012. A Lightweight Algorithm for Message Type Extraction in System Application Logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2012), 1921–1936.

[20] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael R. Lyu, and Hua Cai. 2013. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[21] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 2012)*. USENIX Association, Berkeley, CA, USA, 26–26.

[22] Réka Nagy, Mihai A. Suciu, and Dumitru Dumitrescu. 2012. Lorenz Equilibrium: Equitability in Non-cooperative Games. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO 2012)*. ACM, New York, NY, USA, 489–496.

[23] Annibale Panichella, Fitsum M. Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-objective Optimization Problem. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*. IEEE, Piscataway, NJ, USA, 1–10.

[24] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding Strategies in Search-based Unit Test Generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[25] Abdel Salam Sayyad, Katerina Goseva-Popstojanova, Tim Menzies, and Hany Ammar. 2013. On Parameter Tuning in Search Based Software Engineering: A Replicated Empirical Study. In *Proceedings of the 3rd International Workshop on Replication in Empirical Software Engineering Research (RESER 2013)*. IEEE Computer Society, Washington, DC, USA, 84–90.

[26] S.N. Sivanandam and S.N. Deepa. 2008. *Introduction to Genetic Algorithms*. Springer, Berlin, Heidelberg.

[27] Gilbert Syswerda. 1989. Uniform Crossover in Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA 1989)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2–9.

[28] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating System Events from Raw Textual Logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM 2011)*. ACM, New York, NY, USA, 785–794.

[29] Risto Vaarandi. 2003. A Data Clustering Algorithm for Mining Patterns from Event Logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)*. IEEE, Piscataway, NJ, USA, 119–126.

[30] Jan M. E. M. van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens, and Alexander Serebrenik. 2008. Process Discovery Using Integer Linear Programming. In *Proceedings of the 29th International Conference on Applications and Theory of Petri Nets (PETRI NETS 2008) (LNCS)*, Vol. 5062. Springer, Berlin, Heidelberg, 368–387.

[31] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[32] Shuai Wang, Shaukat Ali, Tao Yue, Yan Li, and Marius Liaaen. 2016. A Practical Guide to Select Quality Indicators for Assessing Pareto-based Search Algorithms in Search-based Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, New York, NY, USA, 631–642.

[33] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2012. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Transactions on Reliability* 61, 1 (2012), 149–169.

[34] Wei Xu. 2010. *System Problem Detection by Mining Console Logs*. Ph.D. Dissertation. University of California Berkeley.

[35] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-source Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE, Piscataway, NJ, USA, 102–112.

[36] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. IEEE, Piscataway, NJ, USA, 415–425.