A novel pattern-based edit distance for automatic log parsing

Maxime Raynal, Marc-Olivier Buob, Georges Quénot

A novel pattern-based edit distance for automatic log parsing

Maxime Raynal
Nokia Bell Labs France
Univ. Grenoble Alpes
CNRS, Grenoble INP, LIG
F-38000 Grenoble, France

Marc-Olivier Buob
Nokia Bell Labs France

Georges Quénot
Univ. Grenoble Alpes
CNRS, Grenoble INP, LIG
F-38000 Grenoble, France

***Abstract*—This work aims at inferring a set of regular expressions to parse a text file, like a system log. To this end, we propose a novel edit distance taking advantage of the pattern matching background. Edit distances are commonly used for fuzzy search and in bioinformatics, and compare two strings at the character level. By doing so, edit distances do not consider the nature of the data conveyed by the strings. To address this problem, we propose the following contributions. First, we propose to model strings at the pattern level using a dedicated data structure, called pattern automaton. Second, we design a novel edit distance, operating at the pattern level. Third, we derive a clustering algorithm optimized for this distance. Finally, we evaluate our proposal through experimental validation.**

I. Introduction

Unstructured data is ubiquitous, and its lack of structure makes it difficult to analyze. As a sequel, it often ends up being unused [1]. In practice, processing unstructured data forces to develop dedicated parsers to convert it to a more convenient format. This problem arises in network management especially when analyzing system *logs*[1] or system command outputs. Unfortunately, developing parsers is often tedious, time consuming and error prone. To address this problem, we propose to design a novel distance and a clustering algorithm to infer the regular expressions required to parse the data.

In the details, log parsing involves three main steps.

1) *Log clustering* partitions lines involved in a log file so that each group of lines conforms to a same (unknown) template.
2) *Template extraction* derives for each cluster a common "template", *e.g.*, regular expression, allowing to extract the data from each line involved in this cluster.
3) *Data extraction* exploits the template assigned to each cluster. It reads the input file line-by-line, extracts the data, and converts it to the target format.

Ideally, a log parsing tool should meet the following requirements.

- **Scalability**: the underlying algorithms must be efficient in order to process large log files.
- **Coherency**: the obtained clustering must form groups of homogeneous lines to output tractable results.
- **Generalizability**: the automatic parser must work with as little prior knowledge as possible (configuration settings must be as generic as possible) and without human intervention.

In a nutshell, our proposal takes as parameter some predefined patterns allowing to capture the nature of the data conveyed in the strings (just like Spell [2], Logmine [3] and Drain [4]) and cluster the log accordingly. This collection of patterns may be customized to improve the quality of the clustering.

The main novelty provided by our approach resides in its ability to consider every decomposition at the pattern scale. Moreover, our approach performs well by only using universal patterns (dates, numerical values, network addresses) and leads to results as accurate as Drain, which is the best existing tool according to [5].

Our proposal models each line of logs at the pattern scale using an automata-like structure called *pattern automaton* (PA). Indeed, a given string may lead to several pattern-based decompositions[2]. As a sequel, choosing an arbitrary decomposition could degrade the quality of the results, and by design, a pattern automaton considers them all.

To compare pattern automata, we introduce a novel distance, that computes a Levenshtein-like distance operating at the pattern scale. From this distance, we derive a clustering algorithm, called *pattern clustering (PC)*, and allowing to group homogeneous lines of log.

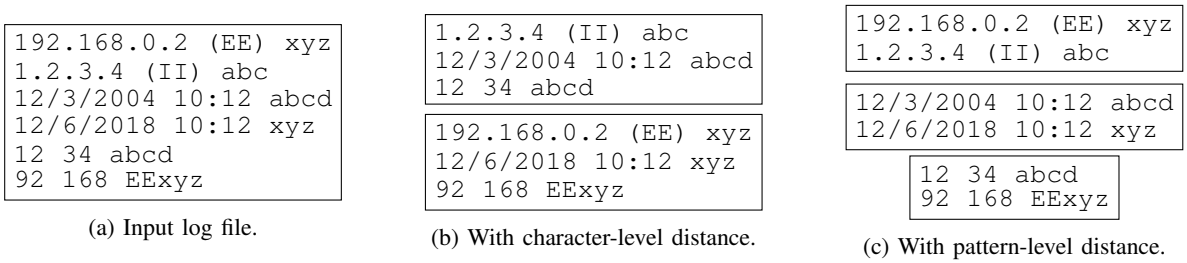
To sum up, the contributions of this paper are:

- a new pattern-based text similarity measure, obtained by generalizing the Needleman-Wunsch algorithm;
- a novel pattern-based clustering algorithm taking advantage of the pattern distance;
- an experimental validation of our proposal, and a comparison against the most relevant existing solutions.

Section II presents the related works, which includes the theoretical background as well as works related to automatic parsing. Section III defines the *pattern automaton* structure allowing to represent text at the pattern scale. Section IV introduces the *pattern distance* used to evaluate how similar

[1]Logs are text files, where each line usually corresponds to a timestamped message.

[2]See for example https://regex-generator.olafneumann.org



(a) Input log file.

(b) With character-level distance.

(c) With pattern-level distance.

Fig. 1: Toy example illustrating the log clustering problem and the inherent limitations of character based distances.

are two pattern automata. Section V presents the *pattern clustering* algorithm, obtained by optimizing the nearest neighbor clustering according to our distance. Section VI evaluates the tractability of our proposal on several public datasets and compares it to LogMine and Drain.

II. State of the art

This section presents the theoretical background related to our proposal (see Section II-A), as well as automatic parsing tools proposed in the literature (see Section II-B).

A. Edit distances on strings

Many text similarity measures have been proposed in the literature [6]. Edit distances compare two strings according to a set of edit operations, and count how many operations are required to transform the first string into the other one.

The Longest Common Subsequence (LCS) only considers character insertions and deletions [7]. [8] revisits this problem using a dedicated graph, called edit graph. [9], [10] propose various optimizations to accelerate LCS algorithm. The Levenshtein distance [11] considers an additional operation, the character substitution. The Needleman-Wunsch (NW) algorithm [12] can compute any edit distance involving character deletion, insertion and substitution. These operations may be weighted to guide how the two compared strings must be aligned. The Smith-Waterman (SW) algorithm [13] is a variation of the NW algorithm. While NW searches for the best alignment for the entire sequences (global alignment), SW searches for the best possible alignment among all the infixes of the two input strings (local alignment).

By design, LCS, NW and SW algorithms operate at the character level. To overcome this limitation, Jiang et al [14] represent RNA sequences using graphs rather than strings. This generalized edit distance captures the secondary and tertiary structures of a RNA sequence, and thus leads to more relevant alignments. For further details about edit distances, we refer the reader to [15].

B. Tools related to automatic parsing

1) Generic text parsing tools: This section covers the related works aiming at transforming unstructured text to structured data with as few assumptions as possible.

PADS [16] is an automatic inference algorithm that converts ad hoc data to a dedicated output format. PADS relies on universal patterns to infer the most appropriate types and frequency-based heuristics to infer the best structure to organize the data (e.g., using arrays, unions, enumerations, etc.). Unfortunately, PADS is no more maintained and is quite complicated to exploit, even for simple file formats.

FlashExtract [17] is a tool accelerating the extraction of data from unstructured text. It can even produce a parser. Its main limitation is that it must be guided by the user during the whole operation.

Datamaran [18] is an unsupervised tool that performs automatic data extraction from unstructured text. It can detect repeated sequences across multiple lines, which improves the quality of the output data structure. Unfortunately, Datamaran is not publicly available and relies on assumptions that may be too restrictive for some practical use cases.

2) Specialized log parsing tools: Existing works mainly focus on the log parsing aspect. From each cluster, one can derive a parsing rule. A complete overview is provided in [5]. The three proposals closest to ours are listed below.

Spell [2] uses a custom clustering algorithm based on the LCS distance.

LogMine [3] combines a nearest neighbor clustering with a custom distance. This distance compares two lines of log and depends on a list of patterns. Each occurrence of each pattern in the compared lines is substituted by a dedicated meta character. This means that the decomposition operated by LogMine depends on the order in which the patterns are listed. Then, the two resulting strings are compared character by character in linear time. This distance is very quick but assumes that the decomposition is the most appropriate one.

Finally, Drain [4] uses the same distance as LogMine, but relies on a multi-level clustering system based on directed graph to build accurate clusters. According to the experiments conducted in [5], Drain is the most accurate tool.

III. Pattern Automaton

Let Σ be an alphabet. We denote the empty word by ε. Given a word w, w_j denotes the j^{th} character of w and $w_{j:k}$ the subword $w_j \dots w_{k-1}$. If $j \geq k$, then $w_{j:k} = \varepsilon$. The Kleene star is denoted by * [19]. For any language L, L^+ is defined by $L^+ = LL^*$. A non-deterministic finite automaton (NFA) is a tuple $A = (\Sigma, Q, \delta, q_0, F)$ where Σ is its alphabet, Q the set of its states, $\delta : Q \times \Sigma \to 2^Q$ its transition function, $q_0 \in Q$ its initial state, $F \subseteq Q$ the set of its final states. Finally, for $n \in \mathbb{N}$, we denote by $\mathbb{N}_n$ the set $\{0, \dots, n\}$.

A pattern automaton aims at representing an arbitrary word at the pattern level using an automaton. We call *pattern* any language included in $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. We call *pattern collection* any set $\mathcal{P}$ of patterns such that $(\bigcup_{P \in \mathcal{P}} P)^+ = \Sigma^+$. The pattern collection definition means that any word can be split in one or more non-overlapping sequences of patterns of $\mathcal{P}$. Such a sequence of patterns is called a *decomposition* of w.

Example 1: Consider $\Sigma = \{\texttt{0}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{Z}, \texttt{.}\}$. Let be $P_{\text{int}} = \{\texttt{0}, \ldots, \texttt{9}\}^+$, $P_{\text{float}} = P_{\text{int}} \cup \{w\,\texttt{.}\,w' \mid w, w' \in P_{\text{int}}\}$, $P_{\text{hex}} = \{\texttt{0}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{F}\}^+$, $P_{\text{letters}} = \{\texttt{A}, \ldots, \texttt{Z}\}^+$, $P_{\text{dot}} = \{\texttt{.}\}$. Each of these languages is a pattern of Σ. Moreover, $\mathcal{P} = \{P_{\text{int}}, P_{\text{float}}, P_{\text{hexa}}, P_{\text{letters}}, P_{\text{dot}}\}$ forms a pattern collection (relative to Σ).

Let word w be a word of Σ^+ and $\mathcal{P}$ be a pattern collection. Consider the NFA such that its alphabet is $\mathcal{P}$, its set of states is $\mathbb{N}_{|w|}$, its initial state is 0 and its only final state is $|w|$. Its transition function δ is defined by $k \in \delta(j, P)$ if and only if $w_{j:k} \in P$, where $P \in \mathcal{P}$. This NFA is acyclic by construction. In this NFA, each path from 0 to $|w|$ corresponds to a possible decomposition of w according to $\mathcal{P}$.

The *pattern automaton (PA)* $\text{PA}(w, \mathcal{P})$ is obtained by removing from this automaton any transition (j, k) labeled by P if there exists a transition (j', k') labeled by P such that $[j, k]$ is strictly included in $[j', k']$. Intuitively, this discards the occurrences of a pattern strictly included in a larger occurrence of the same pattern. Thanks to this filtering, the PA is always deterministic. Similarly, the automaton obtained by flipping each arc of the PA is also deterministic. According to the Brzozowski algorithm [20], this shows that the PA is minimal. Thus, a PA is always deterministic and minimal. As a consequence, two PAs having mismatching transitions necessarily recognize different languages.

Figure 2 represents the pattern automaton of the word `4.57AB` using two different patterns collections. Figure 2a shows a character-based decomposition by using $\mathcal{P}_\Sigma = \{\{a\} \mid a \in \Sigma\}$, whereas Figure 2b depicts the decomposition obtained using the same pattern collection as in Example 1.

IV. Pattern distance

We extend the Needleman-Wunsch [12] algorithm to process pattern automata (see Section III). By doing so, we compute a distance operating at the pattern scale that we call *pattern distance*. This adaptation allows our edit distance to better capture the nature of the data conveyed in the two compared strings, as illustrated on Figure 1.

To compute the pattern distance, we adapt the notion of edit graph introduced by [8]. Figure 3a illustrates the *edit graph* obtained when comparing `4.57AB` and `47XY` using the Levenshtein distance [11]. This distance computes how many edit operations are required to transform w into w' when considering character insertions, deletions and updates. Each state of the edit graph corresponds to the position of two cursors on w and w'. Each *horizontal* (resp. *vertical*) arc corresponds to deletion (resp. insertion) of a character from w to obtain w' and is weighted by 1. Each *diagonal* arc corresponds to replacement operations or matching character. If a diagonal arc corresponds to a match, it is weighted by 0 and otherwise by 1. This graph structure reflects that the Levenshtein distance is character-based. The weight of a shortest path from (j, j') to (k, k') is the minimal number of edit operations required to transform $w_{j:k}$ to $w'_{j',k'}$. In particular, a shortest path from $(0, 0)$ to (w, w') (in red on the Figure 3a) materializes a minimal set of editing operations to transform w into w'.

Figure 3b depicts the generalized edit graph used at the pattern scale to process the input same input strings as Figure 3a. The underlying edit operations are the insertion of a word matched by a pattern of $\mathcal{P}$, the deletion of a word matched by a pattern of $\mathcal{P}$ and the replacement of a subword of $P \in \mathcal{P}$ by another word of P.

By doing this generalization, one can directly apply the Needleman-Wunsch algorithm on two PAs, and hence compare two strings at the pattern scale according to $\mathcal{P}$.

More formally, let be two words w and w' and a pattern collection $\mathcal{P}$. From their PAs $G = (Q, \mathcal{P}, \delta, 0, \{|w|\})$ and $G' = (Q', \mathcal{P}, \delta', 0, \{|w'|\})$, we can derive a NFA $(Q \times Q', (\mathcal{P} \cup \{\varepsilon\})^2, \Delta, (0, 0), \{(|w|, |w'|)\})$, called edit graph and denoted by $\mathcal{E}(G, G')$, where:

- $\Delta((j, j'), (P, \varepsilon)) = (k, j')$ if $w_{j:k} \in P$ (deletion);
- $\Delta((j, j'), (\varepsilon, P')) = (j, k')$ if $w'_{j':k'} \in P'$ (insertion);
- $\Delta((j, j'), (P, P)) = (k, k')$ if $w_{j:k} \in P$ and $w'_{j':k'} \in P$ (match);

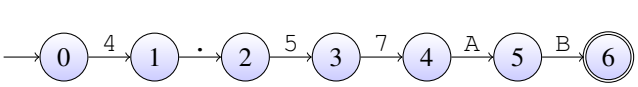
with $k = \delta(j, P)$ and $k' = \delta'(j', P)$. We can show that as G and G' are deterministic (resp. minimal), then so is $\mathcal{E}(G, G')$.

The cost function must also be adapted to our pattern-based edit operations. To quantify how "large" a language L is relatively to Σ, we introduce the language density ρ formally defined by $\rho(L) = \sum_{n \in \mathbb{N}^*} \frac{1}{2^{n-1}} \cdot \frac{|L_n|}{|\Sigma|^n}$. In particular, if $L \subseteq L'$, then $\rho(L) \leq \rho(L')$.

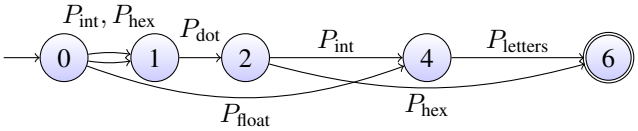
Moreover, it is worth quantifying how similar are two subwords conforming to the same pattern and thus be based on an edit distance. We have decided to use the LCS distance, denoted by d_{LCS}. Each arc of $\mathcal{E}(G, G')$ connecting a state (j, j') to (k, k') labeled by (P, P') is weighted as follows. If the arc is diagonal ($P = P'$), it is weighted by $\rho(P).d_{\text{LCS}}(w_{j:k}, w'_{j':k'})$. Otherwise, the arc is either horizontal ($P' = \varepsilon$) or vertical ($P = \varepsilon$) and is weighted by $d_{\text{LCS}}(w_{j:k}, w'_{j':k'})$. By computing the shortest path from $(0, 0')$ to $(|w|, |w'|)$ using the Dijkstra algorithm [21], we obtain the (non-normalized) pattern distance between w and w' according to $\mathcal{P}$. Note that if $\mathcal{P} = \mathcal{P}_\Sigma$ (see Figure 2a), the resulting distance exactly corresponds to $d_{\text{LCS}}/|\Sigma|$. To normalize our distance between 0 and 1, we divide it by $|w| + |w'|$ and denote it by d.

V. Pattern-based clustering

This section presents various tricks to adapt and accelerate the well-known nearest neighbor algorithm when using the pattern distance. Section V-A shows how to optimize the nearest neighbor primitive required by this algorithm. Section V-B explains how to improve the clustering procedure.

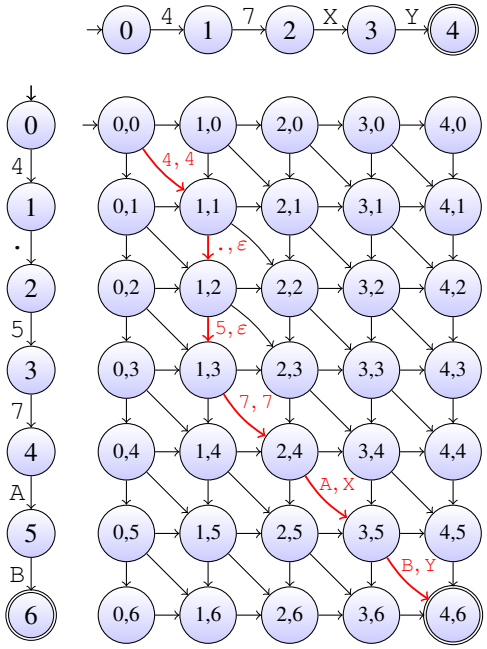


(a) Pattern automaton of `4.57AB` at the character level (≈ string).

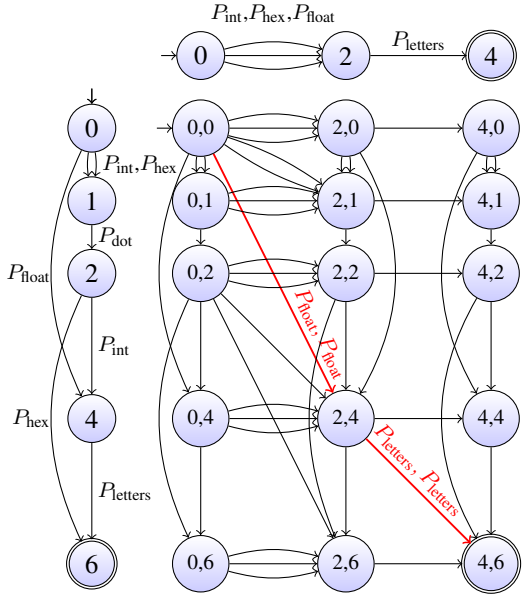


(b) Pattern automaton of `4.57AB` at the pattern level.

Fig. 2: Examples of pattern automata.



(a) Edit graph at the character level for `4.57AB` and `47XY`.



(b) Edit graph at the pattern level for $\text{PA}(\texttt{4.57AB}, \mathcal{P})$ and $\text{PA}(\texttt{47XY}, \mathcal{P})$, using the same definition of $\mathcal{P}$ as in Figure 2b.

Fig. 3: Edit graphs

A. Nearest neighbor

Finding the nearest neighbor: Given a pattern automaton G and a set of pattern automata $\mathcal{G}$, the nearest neighbor primitive determines $\mathrm{n}(G, \mathcal{G}) = \mathrm{argmin}_{G' \in \mathcal{G}}(d(G, G'))$. The simplest approach consists in computing $d(G, G')$ for each $G' \in \mathcal{G}$ to determine the nearest neighbor of G. All these distances may be computed in parallel. One could also build the union graph, obtained by gathering the edit graphs $\{\mathcal{E}(G, G'), G' \in \mathcal{G}\}$, add a source node s connected to each initial states of those edit graphs via an ε-transition weighted by 0, and run a single Dijkstra algorithm from s. When a final state is discovered, it reveals the closest neighbor of G. One can build the union graph opportunistically to save time and memory.

Finding the nearest neighbor closer than D: The sensitivity of the nearest neighbor algorithm is calibrated by tuning a threshold parameter D. This parameter prevents adding an element to a cluster if its distance to the cluster's centroid exceeds D. We thus refine n to obtain the primitive $\mathrm{n}_{\leq D}$. If $\mathrm{n}(G, \mathcal{G}) \leq D$, then $\mathrm{n}_{\leq D}(G, \mathcal{G}) = \mathrm{n}(G, \mathcal{G})$, otherwise $\mathrm{n}(G, \mathcal{G}) \leq D = \perp$. $\mathrm{n}_{\leq D}$ is obtained by interrupting the exploration of the Dijkstra's algorithm if the distance from s to the state being processed is greater than D.

B. Pattern clustering algorithm

Pre-processing: The pattern distance $d(G, G')$ only depends on G and G'. Thus, if they recognize the same language ($\mathcal{L}(G) = \mathcal{L}(G')$), they fall in the same cluster. By discarding duplicated PA, we accelerate the clustering procedure. As explained in Section III, our pattern automata are minimal and deterministic. Thus, if they have a different number of states, then necessarily $\mathcal{L}(G) \neq \mathcal{L}(G')$. Otherwise, we compare G and G' using a graph traversal algorithm to find eventual mismatching transitions. Below, we assume without loss of generality that $\mathcal{G}$ is restricted to distinct pattern automata.

The *pattern clustering (PC)* algorithm partitions $\mathcal{G}$ according to a threshold value D. Each resulting cluster is represented by an arbitrary element belonging to the cluster, called *representative*. In our case, each cluster is represented by its first inserted pattern automaton. We denote by $\mathcal{C}$ the current set of clusters and by $\mathcal{R}$ the corresponding set of representatives and initialize them to $\varnothing$. For each word $G \in \mathcal{G}'$, if $n_{\leq D}(G, \mathcal{R}) = \perp$ or if $\mathcal{C} = \varnothing$, then a new cluster $\{G\}$ is added to $\mathcal{C}$ and is represented by G. Otherwise, G is inserted in the cluster represented by $n_{\leq D}(G, \mathcal{R})$.

To apply this algorithm to a set of words, we only need to

map each pattern automaton with its corresponding word(s).

C. Regular expression inference

This section presents a greedy algorithm inferring for each cluster $C \in \mathcal{C}$ a regular expression able to parse any line involved in C. Let $\{G_1, \ldots, G_{|C|}\}$ be the PAs of C.

We denote by $\mu(G)$ the function that extracts the shortest path from a PA (resp. edit graph) G according to the density ρ and builds the corresponding PA. If G is an edit graph, each arc of the shortest path is labeled by (P, P), (P, ε) or (ε, P) with some $P \in \mathcal{P}$; the corresponding arc of $\mu(G)$ is labeled by P.

- If $|C| = 1$, $E_{|C|}$ is defined by $\mu(G_1)$. Its horizontal and vertical arcs (see Section III) are flagged as *optional*.
- If $|C| > 1$, we define recursively E_i by $E_2 = \mu(\mathcal{E}(G_1, G_2))$ and $E_{i+1} = \mu(\mathcal{E}(E_i, G_{i+1}))$ for all $2 \leq i < |C|$. In E_{i+1}, each arc is flagged as optional if and only if it is horizontal, vertical or related to an optional arc of E_i.

The regular expression characterizing C is obtained by concatenating the (possibly optional) patterns involved in $E_{|C|}$.

VI. Experimental results

A. Setup

In this section, we compare three algorithms, namely LogMine (LM) [3], Drain (DR) [4] and the pattern clustering algorithm (PC) (see Section V). We reproduce the experiments presented in [5]. They are based on the public LogHub dataset [22]. It consists in 16 real log files, involving distributed systems (HDFS), super-computers (HPC), OS logs (Linux, Mac, Windows, Android) and specific software (Proxifier, Thunderbird, Apache), etc. For more details about the Loghub dataset, we refer the reader to [23]. Compared to [5], the experimental setup is improved as follows:

- We reproduce the experiments performed in [5] using their collection of patterns, which involves dataset-dependant and non-trivial patterns tailored to improve the results. Moreover, we run our experiments using the same collection of universal patterns.
- To assess the accuracy of the evaluated algorithms, [5] computes a custom metric, called *parsing accuracy*. According to its definition, this metric only rewards clusters that exactly matches those expected in the ground truth. It is highly sensitive to small variations between the ground truth and the computed clustering. As a sequel, it may highly penalize results that are almost perfect. That is why we also computed the adjusted Rand index [24], which is by design less sensitive to small variations.
- Finally, [5] summarizes each cluster by a template (i.e., a string with some wildcards). We merged the clusters which have no reason to be separated.

The accuracy of LM, DR and PC are evaluated by using two following pattern collections:

- The *specific collection*: it corresponds to the patterns used by [5], specifically tailored for each dataset. These collections are used to evaluate the performance of each algorithm with significant prior knowledge.
- The *minimal collection*: it only involves universal patterns (recognizing integers, words, hexadecimals, punctuation marks, network addresses, etc.). We use the same minimal collection for all the datasets. It is used to evaluate whether an algorithm captures well the file structure with low prior knowledge.

Each algorithm also requires a threshold value, used to calibrate the sensitivity of the clustering algorithm. We sampled several threshold values. As in [5], we only report the best results obtained for each dataset. Our setup (ground truth, pattern collections, optimized threshold values) as well as our experimental results are publicly available on [25]. Our experiments have been realized on a 24-core AMD Ryzen @3.9 GHz CPU and 32GB of RAM.

B. Accuracy and generalizability

Figure 4 depicts the parsing accuracy obtained for each algorithm, each dataset and each collection. Using the minimal collection does not degrade significantly the results, demonstrating a good generalizability for each algorithm. According to [5], Drain is the most accurate algorithm among the proposals tested in their benchmark. The parsing accuracy is better with PC in average with both the minimal collection (0.27 for LM, 0.53 for DR and 0.71 for PC) and the specific collection (0.27 for LM, 0.72 for DR and 0.78 for PC).

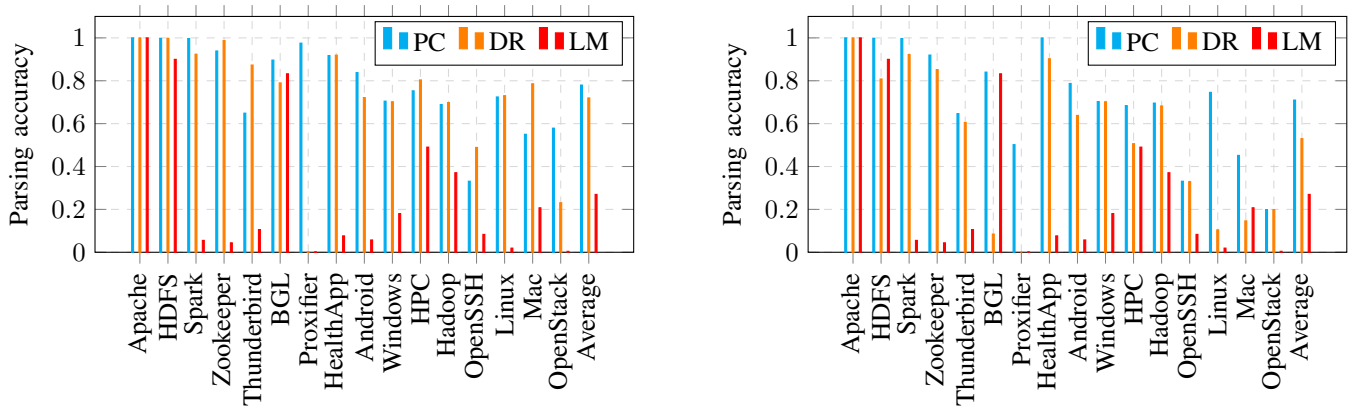
Figure 5 reports the accuracy of each algorithm by using the the adjusted Rand index, which is most robust than the parsing accuracy for small variations. The differences using the specific collections and the minimal collection are negligible. Our approach is more accurate in terms of Adjusted Rand Index, both with minimal collection (namely 0.767 for LM, 0.756 for DR and 0.97 for PC) and specific collection (namely 0.767 for LM, 0.917 for DR and 0.981 for PC).

The most striking result on Figures 4 and 5 concerns the Proxifier dataset. Contrary to the other datasets, it contains optional fields and both static and variable parts of texts. This explains why PC outperforms DR and LM. Indeed, our approach supports optional fields, only relies on pattern-based considerations, and thus does not require static part of text to perform well.

Finally, PC and LM are based on the same clustering algorithm, but distinct distances. PC is significantly more accurate showing the relevance to use a pattern-based distance.

C. Scalability

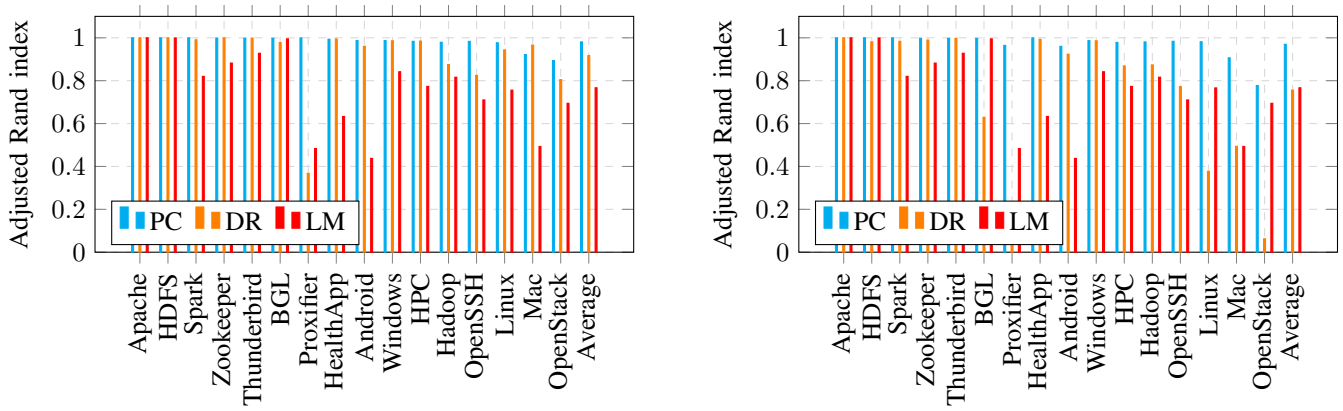
Figure 6 depicts the running time required to process the Proxifier log with each algorithm. On this dataset, our approach (PC) is roughly 10x slower than DR, but like its competitors, the runtime grows linearly with the number of lines. Thus, PC scales well with the size of the log file. This important overhead is mainly due to the "distance" used in each algorithm. Indeed, the distance used by LM and DR runs in linear time with the length of the compared lines, whereas the pattern distance used in PC is quadratic. Despite



(a) Using the specific collections proposed in [5].

(b) Using the minimal collection.

Fig. 4: Parsing accuracy results (using the best hyper parameters). The last column gathers the average on all logs.



(a) Using the specific pattern collections proposed in [5].

(b) Using the minimal pattern collection.

Fig. 5: Adjusted Rand index results (using the best hyper parameters). The last column gathers the average on all logs.

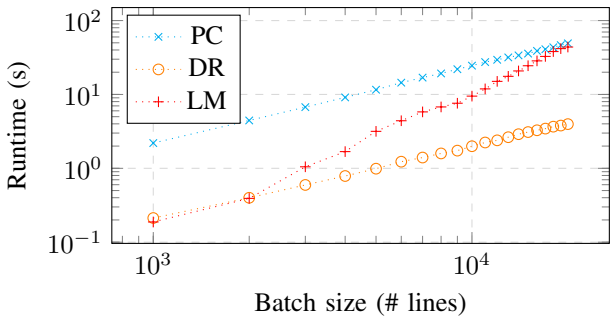


Fig. 6: Runtime obtained by various number of lines from the Proxifier dataset using the minimal pattern collection.

this important overhead, our approach computes clusters in a reasonable time, which makes it usable for practical use cases.

Further experiments show that the runtime of PC is multiplied by a factor 4 due to the LCS computations inherent to the weighting of the edit graphs. Unfortunately, using simpler weights significantly degrades the accuracy and that is why we decided to keep our cost function.

VII. Conclusion

This paper presents a novel pattern-based distance obtained by extending the Needleman-Wunsch algorithm at the pattern scale. This generalization is done by modeling strings using automata capturing any reasonable pattern-based decomposition and revisiting the notion of edit graph. Then, we proposed an optimized nearest neighbor algorithm built on top of our distance. The resulting clusters are used to infer valid regular expressions. supporting optional fields. To the best of our knowledge, this is the first log parsing approach that infers accurate regular expressions and supporting optional fields. Our experiments show that our clustering algorithm runs fast enough to be applicable for real datasets. Moreover, our results show that even with low prior knowledge, our proposal competes with the best existing approaches in terms of accuracy. In future works, we plan to improve our regular expression inference to detect repetitions and hence better capture the structure of the input file.

Acknowledgment

This work has been partially supported by MIAI@Grenoble Alpes, (ANR-19-P3IA-0003).

REFERENCES

[1] I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino, "Data wrangling: The challenging yourney from the wild to the lake." in *CIDR*, 2015.

[2] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.

[3] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1573–1582.

[4] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 33–40.

[5] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 121–130.

[6] J. Wang and Y. Dong, "Measurement of text similarity: A survey," *Information*, vol. 11, no. 9, p. 421, 2020.

[7] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM (JACM)*, vol. 25, no. 2, pp. 322–336, 1978.

[8] E. W. Myers, "An $o(nd)$ difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[9] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[10] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. IEEE, 2000, pp. 39–48.

[11] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[12] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[13] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[14] T. Jiang, G. Lin, B. Ma, and K. Zhang, "A general edit distance between rna structures," *Journal of computational biology*, vol. 9, no. 2, pp. 371–388, 2002.

[15] W. Haque, A. Aravind, and B. Reddy, "Pairwise sequence alignment algorithms: a survey," in *Proceedings of the 2009 conference on Information Science, Technology and Applications*, 2009, pp. 96–103.

[16] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: fully automatic tool generation from ad hoc data," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 421–434, 2008.

[17] V. Le and S. Gulwani, "Flashextract: a framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 542–553.

[18] Y. Gao, S. Huang, and A. Parameswaran, "Navigating the data lake with datamaran: Automatically extracting structure from log datasets," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 943–958.

[19] K. D. Joshi, *Foundations of discrete mathematics*. New Age International, 1989.

[20] J. A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," in *Proc. Symposium of Mathematical Theory of Automata*, 1962, pp. 529–561.

[21] E. W. Dijkstra, *A short introduction to the art of programming*. Technische Hogeschool Eindhoven Eindhoven, 1971, vol. 4.

[22] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[23] Loghub: A large collection of system log datasets for ai-powered log analytics. [Online]. Available: https://github.com/logpai/loghub

[24] L. Hubert and P. Arabie, "Comparing partitions," *Journal of classification*, vol. 2, no. 1, pp. 193–218, 1985.

[25] Results repository. [Online]. Available: https://github.com/raynalm/pattern_distance