Instituto Tecnologico y de Estudios Superiores de Monterrey

Puebla Campus

School of Engineering and Sciences

TECNOLÓGICO DE MONTERREY®

An Approach Based on Suffix Array to Discover Routines in User Interaction Logs

A thesis presented by

Astrid Monserrat Rivera Partida

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Monterrey, Nuevo León, June, 2021

An Approach Based on Suffix Array to Discover Routines in User Interaction Logs
by
Astrid Monserrat Rivera Partida

Abstract

Robotic Process Automation (RPA) is a fast-advancing technology that allows organizations to automate repetitive work by the use of software robots. The number of candidate processes for automation may be vast and, henceforth, it raises questions such as which processes have higher priority for automation? and which of them can be automated by means of User Interaction (UI) routines? Selecting routines amenable for automation requires distinguishing between noise and relevant events and defining the boundaries thereof. From a technical point of view, the discovery of routines starts with a log capturing the UI tasks performed by a human and partitions it into segments that are presumably candidate patterns that can be composed into routines. Existing techniques, however, fail when the UI log contains multiple intertwined routines, yet such situation occurs in real-world scenarios. This project tackles this problem leveraging techniques stemmed from the field of periodic sequence mining and, more specifically from stringology. The result is a collection of novel algorithms tailored for the problem at hand which shows good performance, resilience to noise and has been evaluated with artificial and human generated UI logs.

List of Figures

2.1 Excerpt of a UI log . . . 8
2.2 Robotic Process Mining pipeline . . . 9
2.3 Web-usage mining process . . . 14
2.4 Suffix tree for T = mississippi . . . 17
2.5 Suffix array of the string T = mississipi$ and its lcp-interval tree . . . 20
2.6 Range Minimum Query example . . . 23
2.7 The algorithm assumes a constant alphabet Σ and k ≥ 1. However the solution takes $O(mk|A| \cdot max(k, logn + occ)$ when using O(n) bits indexing data structure. . . . 24
2.8 Example of a control-flow graph . . . 25
2.9 Dominator tree from CFG . . . 26

3.1 CFG of example log *L'* . . . 32
3.2 Sequence from UI log L' . . . 33
3.3 LCP array excerpt . . . 35
3.4 Suffix array for UI log L = $\bot ABDABCDACABCDABD\top$ . . . 35
3.5 LCP Tree of UI log *L* . . . 36
3.6 Pattern Symbol for leaf $\langle(5; 0, 1), 4\rangle$ . . . 38
3.7 LCP Tree of UI log *M* . . . 41

List of Tables

3.1 Excerpt of a UI log 30
3.2 UI log normalization and translation 31

4.1 UI logs 48
4.2 Information of the interleaved UI logs 48
4.3 Comparing the quality of the discovered routines 52
4.4 Comparing the quality of the discovered routines 53
4.5 Results of the interleaved logs 54
4.6 Information of the routines found by the Graph Approach 55

Contents

Abstract v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Problem definition . . . 2
- 1.2 Hypothesis and Research Questions . . . 3
- 1.3 Objectives . . . 4
- 1.4 Organization . . . 4

2 Theoretical Framework 7
- 2.1 Robotic Process Mining . . . 7
  - 2.1.1 User Interactions log . . . 7
  - 2.1.2 Stages and Challenges in RPM . . . 8
- 2.2 Sequential Pattern Mining . . . 11
  - 2.2.1 Full vs. Partial . . . 11
  - 2.2.2 Perfect vs. Imperfect . . . 12
  - 2.2.3 Synchronous vs. Asynchronous . . . 12
  - 2.2.4 Dense Periodic Patterns . . . 13
  - 2.2.5 Approximate Periodic Patterns . . . 13
- 2.3 Web-usage Mining . . . 14
  - 2.3.1 Time-Oriented Heuristics . . . 14
  - 2.3.2 Navigation-Oriented Heuristics . . . 15
- 2.4 Correlation of event logs for process mining . . . 15
- 2.5 String Matching Basic Concepts . . . 16
  - 2.5.1 Suffix Tree . . . 16
  - 2.5.2 Suffix Array . . . 17
  - 2.5.3 Burrows Wheeler Transform . . . 17
  - 2.5.4 Repeating Substrings and Extendibility . . . 18
  - 2.5.5 Longest Common Prefix . . . 18
  - 2.5.6 Finding Maximal Repeats . . . 20
  - 2.5.7 Top-down traversals . . . 20
  - 2.5.8 Range Minimum Query . . . 22

2.5.9 Approximate String Matching 23
2.6 Related Work 25
2.7 Chapter Conclusions 27

3 Contribution **29**
3.1 UI Log Normalization 29
3.2 Discovering Exact Routines in a UI Log 32
3.2.1 Building basic structures 32
3.2.2 Algorithms to Discover Exact Patterns 36
3.3 Approximate matching 40
3.4 Chapter Conclusions 44

4 Evaluation **47**
4.1 Datasets 47
4.2 Metrics 48
4.2.1 Jaccard index 49
4.2.2 Rand index 49
4.2.3 Coverage 50
4.2.4 Effective Coverage 50
4.3 Results 50
4.3.1 Artificial and real-life logs 50
4.3.2 Interleaved logs 54
4.4 Validity Threats 55
4.5 Limitations 56

5 Conclusions and Future Work **57**
5.1 Future Work 57

Bibliography **62**

Chapter 1

Introduction

Business processes are part of the daily routine of every organization, and they must perform in the best possible way to reduce or avoid losses. A business process may take input from different devices or people and must be completed according to predefined rules to produce the desired output [34]. Seeking efficiency in the execution of business processes is not something new. As intelligent systems have evolved, their integration to typical business applications has been sought, being the primary objective the efficiency enhancement through process automatization—one of the significant trends in the robotic automation of business processes.

Robotic Process Automation (RPA) is a software solution for creating programs that mimic human workers' behavior when performing repetitive and structured tasks with information systems [9]. It is driven by simple rules and business logic while interacting with multiple information systems through graphic user interfaces. Its functionalities comprise the automation of repeatable and rule-based activities using a non-invasive software robot called a bot. Such bots have the capacity to respond to stimuli and changes in business processes. Also, RPA simplifies the way users automate tasks by interacting with multiple applications at once [22].

RPA can facilitate scaling up strategically and helping companies to offer greater value to clients and customers. Its main objective is to reduce the burden of repetitive, simple tasks on employees [47]. RPA has outperformed expectations on non-financial benefits such as accuracy, timelines, flexibility, and improved compliance, with at least 85 of the surveyed businesses reporting that RPA met or exceeded their expectations in these areas [49]. According to Gartner [17], RPA software revenue grew 63.1% in 2018 to $846 million, making it the fastest-growing segment of the global enterprise software market.

However, RPA still has some challenges to overcome. While RPA has already been successfully applied to various organizations, a significant amount of time is dedicated to programming the bots manually. The current method for identifying candidate routines is interviewing workers, walk-throughs, and detailed observation. Nevertheless, these methods are time-consuming, and they face scalability issues if the number of routines is too high. Besides, if there is an inefficient process, RPA can not find a way to optimize it [30]. Hence, it raises two important questions: which routines should be automated and how to optimize them.

A new set of tools, Robotic Process Mining (RPM), arises as a solution to both questions. Its main idea is that repetitive routines amenable to automation may be found from

the interactions between users and the applications (User Interactions or UIs) used during the process enactment. The candidate routines will be analyzed in terms of potential benefits and automation costs. RPM tools identify automatable routines from a given log of user interactions (UI log), collect their variants, standardize and streamline the identified variants, and discover an executable specification. Then, the specification will be compiled into a script and executed in an RPA tool.

The discovery of automatable routines still is an unexplored area. However, there have been recent approaches for process discovery. For example, the research in [32] presents an approach to identify the degree of automation through textual process descriptions, such as work instructions. However, this approach may lead to imprecise results due to the assumption that tasks are performed as documented. Other research proposals, such as TaskTracer [13], have addressed the problem by analyzing UI logs to identify the task performed by the user or the switches between tasks. Notwithstanding, they do not consider the inputs and outputs of each action but only the sequence of actions.

Therefore, in this research work, we address the need to discover routines from an unsegmented UI log. We, therefore, develop a collection of algorithms stemmed from the field of periodic sequence mining to identify automatable routines from a UI log. An empirical analysis of the algorithms' performance regarding their ability to discover routines was carried out. The algorithms have been evaluated with artificial logs and real processes within an educational institution.

1.1 Problem definition

Robotic Process Automation (RPA) is a fast-emerging process automation approach that uses software robots to replicate human tasks, including interacting with an enterprise application or transferring data from one application to another, and more, allowing users to focus on more critical tasks. RPA technology is suited to replace humans who perform processes where it is needed to include data from multiple inputs, and then RPA processes it using rules and uses this completed as the input for other systems [28].

RPA has been defined as a class of tools that allow users to specify deterministic routines involving structured data, rules, user interface interactions, and operations accessible via APIs [31]. These routines are encoded as scripts executed by bots, operated via control dashboards.

This technology has the purpose of eliminating or reducing costs, the need for multiple people performing large amounts of repetitive tasks, and improving business processes' quality and speed. RPA contributes to accelerate time to value, reduce human error, and increase throughput. It also helps to ensure that outputs are complete, correct, and consistent [39]. Another critical benefit of robotic process automation is that the tools do not alter existing systems, while other automation tools interact using application programming interfaces (APIs), leading to issues about maintaining the code and responding to underlying applications [28].

Despite the multiple benefits of RPA, there is still one paramount concern: How to select candidate routines for automation? Even if RPA tools are able to automate different routines, they cannot identify which routines are suitable to automate. Current methods for identifying automatable routines involve interviews and detailed observations from the workers, either in

situ or using video recordings [6]. In large organizations, this approach is time-consuming, and it faces scalability limitations if the number of routines is high.

The new concept of Robot Process Mining (RPM) arises to solve this problem. RPM is defined as a set of tools that take as input data collected (UI log) during the execution of tasks in business processes and use these data to assist analysts to identify the routines for automation, extract executable routine specifications, generate RPA scripts and monitor the execution [31]. Hence, RPM tools receive as input a UI log which has to be recorded beforehand. Given a UI log, the goal of RPM is to discover repetitive sequences of actions, which are called *routines*, and select the routines amenable for automation. Then, RPM aims to extract an executable *routine specification* for an RPA tool. From this, we are able to distinguish three main stages in RPM:

1. Collecting and pre-processing UI logs corresponding to executions of one or more tasks.

2. Identifying candidate routines.

3. Discovering executable RPA routines.

In this work, we focus on the second stage of the RPM process: identifying candidate routines for automation. Discovering routines can be enhanced via automated methods. Most existing approaches to find automatable routines assume that the UI log is already segmented, which translates to discovering frequent sequential patterns, a problem widely explored. In practice, a UI log is not segmented and consists of a single sequence of actions or UIs containing one or more routines. However, these instances of routines can also be intertwined with UIs that are not part of any routine. These UIs are called *noise*. Noise can occur when workers make mistakes while performing a task, leading to execute irrelevant actions.

Previous work on noise filtering [45] treats noise as chaotic events; however, it is not useful in case noise gathers around a specific state and could also remove relevant events that seem chaotic. The problem posed by the segmentation stage is similar to Web session mining. Nonetheless, Web session mining approaches can be used only in the context of Web interactions, while tasks in the RPM context are performed across multiple applications. The approaches [15] and [4] are tested under restrictive settings and produced inaccurate results. Another approach [29], which tackles the problem with a high degree of success, is sensitive to multiple starts and is not as accurate when different routines start with different initial UIs.

Therefore, in this work we address the problem of finding candidate routines for automation from an unsegmented UI log. To this end, the selected approach uses suffix arrays to extract exact patterns from the UI log, each representing a sequence of UIs that repeatedly appear in the unsegmented UI log. Additionally, we adapt existing string matching methods to find approximate patterns to discover routines affected by noise. Moreover, we evaluate and compare the performance of our method with the existing approach for identifying UI log segments. This method has been tested on artificial and real-life UI logs to evaluate its ability to discover quality routines.

1.2 Hypothesis and Research Questions

In this research, we are interested in the question of how to identify candidate routines for automation? However, we focus on particular questions that guide this work and we aim to

answer through our research.

Research questions

- **RQ1** What set of state-periodic mining algorithms can be adapted to be used for the problem at hand?
- **RQ2** How to measure the quality of executable routines discovered by periodic sequence mining algorithms?
- **RQ3** What is the performance of periodic sequence mining?
- **RQ4** Is the approach applicable in real-life settings? Is it capable of handling noise in the UI log?
- **RQ5** What patterns are factors in failures?

We are interested in developing an efficient method to extract routines from UI logs, therefore, we define the following hypothesis: **Using periodic sequence mining techniques will allow us to identify periodic frequently occurring significant sequences from unsegmented User Interface logs that will lead to quality candidate automatable routines for Robotic Process Mining tools**.

1.3 Objectives

The general objective of this research is **to develop and implement a noise-tolerant algorithm based on periodic sequence mining techniques to identify candidate routines from UI logs**. We expect that such a technique would allow us to find meaningful sequences and amenable for automation and enhance the quality of the executable routines for RPA tools. The particular goals to achieve as this research work is conducted are:

- To develop a new algorithm extending the state of the art techniques on periodic sequence mining to discover noise-free routines from UI logs generated synthetically.
- To develop a noise-tolerant algorithm to discover routines from UI logs extracted from real-world scenarios.

1.4 Organization

This thesis project is organized into five chapters, starting with the current chapter.

- Chapter 2 reviews the **Theoretical Framework** needed to conduct the research, going into detail about the complete process of Robotic Process Mining and the concepts behind it. We present the concept of UI log; and a review of string matching concepts and algorithms used in our proposed solution. Moreover, we analyse the only other algorithm applicable to our problem of identifying candidate automatable routines.

- Chapter 3 describes our **Contribution**. The proposed solution is divided into two parts: the first one to extract exact routines and the second one to find routines with noise.
- Chapter 4 presents the **Evaluation** of the proposed solution and compares it against the current State of the Art algorithms. To this end, we propose evaluation metrics to assess the efficiency of our approach.
- Chapter 5 contains the **Conclusions and Future Work**, where we evaluate the completion of the research objectives and address the future work.

Chapter 2

Theoretical Framework

In this section we introduce the concepts and techniques relevant for the research proposed in this manuscript. Firstly, we explore ”Robotic Process Mining” (RPM), the phases comprising it, and the challenges associated with each of those phases. Moreover, we introduce the concept of User Interactions Log (UI log) and the processing required to make it suitable for RPM. Given that the focus of this work is the noise filtering and segmentation of UI logs, we explore sequence mining techniques as we consider them relevant in formulating a solution to the problem of identifying candidate routines from the UI logs. Henceforth, we introduce both RPM and sequence mining in the following subsections.

2.1 Robotic Process Mining

As stated before, Robotic Process Mining (RPM) is a class of new tools capable of discovering automatable repetitive routines from interactions between workers and Web and desktop applications, intending to turn the discovered routines into scripts that RPA bots can execute. In this way, they will assist analysts by creating a systematic inventory of candidate routines for automation and once selected, RPM will produce executable specifications of routines. RPM can be considered as an extension of process mining. However, their focus and goal are complementary. While process mining is concerned with the analysis of event logs from enterprise systems to discover/enhance process models and supporting the analysis of the performance of the underlying business processes [46], RPM is concerned with analyzing UI logs, possibly in conjunction with data extracted from process event logs, to identify candidate automatable routines, obtain routine specifications, and monitor their executions. Then, its focus lies in identifying opportunities for automation. Before discussing the research challenges these tools present, we introduce the concept of User Interface log, which is the input for the whole RPM process.

2.1.1 User Interactions log

User Interactions (UIs) are generated as natural byproducts of the interaction of users with systems using a window-based operation. Such events usually capture information related to a user interaction executed on the system. A UI log is a timestamped sequence of UIs or

	Timestamp	Event Type	Source	Arg 1	Arg 2	Arg 3
1	20-07-2014 14:00:00	Edit text field	Web	Label: “Name”	Value: “Ralph”	
2	20-07-2014 14:01:10	Click button	Web	Label: “Save”		
3	20-07-2014 14:02:15	Go to URL	Web	URL: “https://www.facebook.com/”		
4	20-07-2014 14:02:30	Open File	File System	FileName: data.xls		
5	20-07-2014 14:03:45	Go to cell	Worksheet	SheetName: Sheet1	Cell: A2	Value: “John”
6	20-07-2014 14:04:00	Copy	Worksheet	Content: “John”		
7	20-07-2014 14:04:03	Copy	Worksheet	Content: “John”		
8	20-07-2014 14:04:15	Click text field	Web	Label: “Name”		
9	20-07-2014 14:04:21	Paste	Web	Value: “John”		
10	20-07-2014 14:05:02	Click button	Web	Label: “Save”		
...	...	...	...	...	...	...

Figure 2.1: Excerpt of a UI log

events across one or multiple applications performed by a user in a workstation [38]. Each UI is characterized by its timestamp, event type, and other parameter values. A UI’s parameters can be divided into: *context parameters* and *data parameters*. Data parameters store the values used per task execution. On the other hand, context parameters always have the same values.

In Figure 2.1 we present a fragment of a UI log to illustrate these concepts. If a UI consists of clicking a button, the button’s identifier must be stored, as seen in row 2 of the UI log. *Click button* stores the source application (Web), which is a context parameter. Similarly, if a field is edited (row 1), the attributes that should be stored include the identifier of the field and the new value assigned to that field, which is data parameters. Moreover, UIs of the same type normally have the same number of attributes. Some UIs can be merged into one *action*, i.e., *Go to cell* (row 5) and *Copy* (row 6) can be merged in one action called *Copy Cell*.

The bulk of the information associated with a UI is collectively referred to as the *payload*. Likewise, each execution of a task is represented by a *task trace*. Given that UI logs are typically voluminous and rich in detail, automated processing is generally required to extract information at a level of abstraction useful to identify repetitive sequences of actions and the routines amenable for automation.

2.1.2 Stages and Challenges in RPM

The process followed by RPM tools can be decomposed into three main phases:

1. UI log collection and pre-processing.

2. Candidate routine identification.

3. Executable routine discovery.

To provide a more detailed explanation of the elements of each phase, they are decomposed into steps. The first phase decomposes into the recording step itself, and three preprocessing steps (noise filtering, segmentation and simplification). The second phase maps into the step of candidate routine identification. The third phase is decomposed in two steps: executable routine discovery and compilation. We present the specific challenges and opportunities that are associated with each stage. To clarify the process, the diagram in Figure 2.2 shows the steps graphically with their corresponding inputs and outputs.

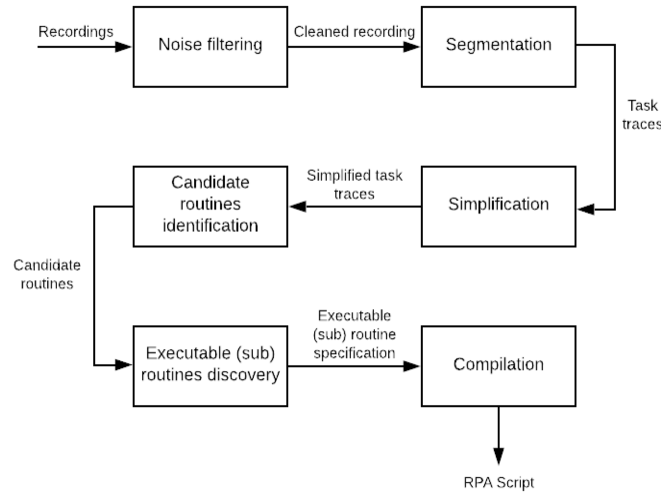

Figure 2.2: Robotic Process Mining pipeline

1. **Recording.** The main challenge regarding recordings is identifying which actions must be recorded, given that one action may be either important or irrelevant depending on the context. From this step, we extract the *recording* of a UI log. Existing UI event recording tools, such as Macro Magic and TinyTask, save the actions performed by the user at pixel coordinates level, and the UI logs generated by these tools are not useful for extracting routines. In [2] a Windows-based application that records UI actions from mouse movements to the selection of items in dialog boxes is presented; however, it supports only Microsoft Word and Adobe Reader. The tool in [14] goes beyond that and even simplifies the recorded UI logs by removing redundant events, although it supports only MS Excel and Google Chrome.

2. **Noise filtering.** A UI log may contain irrelevant events that do not belong to any task; thus, it is needed to separate this noise from the events that contribute to tasks. One solution could be to treat noise as events that can happen anywhere in process execution. But if noise gathers towards one specific task or set of tasks, it is possible not to filter it out or remove relevant events mistakenly. Thus, it is important to consider the values of the data objects involved in the actions and events.

3. **Segmentation.** Likewise, one UI log may have data from multiple tasks, where their actions and events are mixed. One task can be spread across multiple UI logs; for example, if multiple users perform the same task on different work stations. Similarly, as with noise filtering, one approach examines data values involved in the events and actions since it is most likely that events belonging to the same task have similar data values. It is also possible to use sequential pattern mining techniques [42] to search for frequent sequences of events with common data attributes or attribute values. Furthermore, an approach to tackle the problem of segmentation is presented in the research by [33]. This problem is akin to Web session mining, where the input is a set of clickstreams, and the goal is to extract sessions where a user engages with a web application to fulfill

a goal. Most traditional approaches to session identification can be used, however, only in the context of Web interactions, making this approach not valid, given that tasks are usually performed across different systems and applications, and the Web browser is just one of many applications. Another approach is to set a limit for the duration of a session through time-based heuristics or to set the maximally allowed time difference between two events. However, this approach is not reliable since users may be involved in different activities when performing the tasks.

4. **Simplification.** Even if an event is part of a task, it can still be redundant based on the context. So, simplification involves the detection of redundant events. One possible approach to eliminate redundant tasks is to use sequential pattern mining techniques to identify which events are outliers [42]. Nevertheless, if an event rarely occurs, it can be treated as an outlier, creating the need for semantic filtering.

5. **Candidate routines identification.**

   The first substep aims to identify and extract repetitive sequential patterns that represent the execution of routines. One of its challenges is that during routine execution, the user can perform other actions that do not constitute a routine. When identifying the routines, such actions have to be ignored. In this regard, sequential pattern mining techniques, in particular, the ones that work with gaped patterns [14] can be used. The second challenge involves the order of the actions performed in the routine, given that sometimes the actions that constitute a routine may be performed randomly (e.g., filling a Web form). Hence, it is difficult to identify frequently occurring patterns. One possible solution is to standardize the task traces and then identify repetitive patterns.

   This step can be mapped to the problem of frequent pattern mining. The goal of this step is to identify sequences of UIs, which can be translated as symbols. Frequent pattern mining algorithms can be divided into two types based on their output:

   - Algorithms that discover exact patterns and are vulnerable to noise.
   - Algorithms that discover patterns with gaps and are noise-resilient.

   Based on their input we can divide them into:

   - Algorithms that operate with a set of sequences of symbols and can be applied to segmented UI log.
   - Algorithms that discover frequent patterns from a sequence and may be applied directly to unsegmented UI log.

   The second substep seeks to identify routines amenable for automation. A discovered routine is considered a candidate for automation if this routine is semi or fully automatable. In this context, the challenge is how to identify if a routine is automatable or not. Geyer-Klingeberg et al. [2], describe how to assess the automation potential of a task, they propose the frequency of execution of a task as the main criterion for automation. However, if the task is frequent, there is no guarantee it is automatable.

6. **Executable routines discovery.** There can be multiple ways of executing a routine, so when discovering a routine specification, it is necessary to capture all the preconditions that may trigger the routine and its effects or postconditions. If two routines have the same effects, the best must be selected. Another challenge is to discover the data transformations within each action. Recent work [6] uses methods for automated discovery of data transformation-by-example, although these methods present scalability issues and the types of transformations they can discover are limited.

7. **Compilation.** To generate an executable RPA script, given the routine specification, it is required to identify the application elements involved in the routine. Such information is usually obtained during the recording step. However, sometimes it may be missing, for example, when the elements do not have identifiers or in nested containers. Hence, it is required an intelligent recognition of the elements.

2.2 Sequential Pattern Mining

The problem of segmentation is similar to periodic pattern mining. Pattern mining is a powerful tool for analyzing big datasets, which consists in discovering useful patterns in databases. A sequential pattern is a set of itemsets structured in sequence database which occurs sequentially with a specific order. A sequence database is a set of ordered elements or events, stored with or without a concrete notion of time. Each itemset contains a set of items which include the same transaction-time value [43]. Frequently occurring subsequences are referred to as sequential patterns. Sequential pattern mining aims at discovering and analyzing statistically relevant subsequences from sequences of events or items with a time constraint. It consists of discovering interesting subsequences in a set of sequences, where the interestingness of a subsequence can be measured in terms of various criteria such as its occurrence frequency, length, and profit [16].

A periodic pattern is the one defined as repeating behaviors at certain location with regular time interval (time property), they can provide important information to assist with decision-making [51]. At the same time, periodic patterns are classified as frequent periodic patterns and statistically significant patterns based on the frequency of occurrence. According to Sirisha et al. [42], frequent periodic patterns are classified as full or partial, perfect or imperfect, synchronous or asynchronous, dense and approximate. Below are described the characteristics of each pattern and the existing approaches to identify them.

2.2.1 Full vs. Partial

Full. A full periodic pattern is where every element in the pattern exhibits the periodicity.

ABCBCBCACD

This sequence has a full periodic pattern BC with period equal to 2.
Partial. It is a pattern in which one or more elements do not exhibit periodicity.

ABCADCACC

This sequence contains a partial periodic pattern A*C, where the second element does not exhibit the periodic behavior.

2.2.2 Perfect vs. Imperfect

Perfect. A pattern X is said to satisfy perfect periodicity in sequence S with period p if starting from the first occurrence of X until the end of S every next occurrence of X occupies p positions away from the current occurrence of X.

ABDABVABFABC

AB* is a perfect periodic pattern with a period 3. It occurs 4 times from first occurrence to the end of the sequence. In [36], an algorithm for discovering cyclic association rules is presented, in which association rules are defined as the relationships between the occurrences of items within transactions This algorithm finds the cyclic association from time stamped transactional data. A cycle *c* is a tuple *(l,o)*, where *l* is the length in multiples of the time unit and an offset *o* which represents the first time unit when the cycle occurs. An association rule has a cycle *c = (l,o)* if the association rule holds in every lth time unit starting with time unit t_0 and t_i equals to the time interval $[i * t, (i + 1) * t]$, where *t* is the time unit referring to time granularity specified by the user.
Imperfect. Imperfect periodicity means that the pattern deviates from the next expected occurrence, it is possible to have some of the expected occurrences of X missing.

ABCDBFABGABV

AB* is an imperfect pattern, it has missed the second of its expected positions. Han et. al [23] explored properties related to partial periodicity and proposed an algorithm called *max-subpattern hit set*, creating a tree to mine full imperfect periodic patterns and partial imperfect periodic patterns. For a given period, the time series is segmented in smaller segments where the length of their period is equal the given period. The time series is scanned once and all *1-patterns* which are found to be frequent in the segmented time series are reported. A 1-pattern is the one where only one position in the pattern is defined, i.e. *(a, *, *)* or *(*, a, *)* or *(*, *, a)*. Then, these patterns are joined to form a maxpattern and the time series is scanned for the second time. During the second scan each segment is intersected with the maxpattern. The result of the intersection, called *max-subpattern*, is inserted into the tree or, if it is already in the tree its node count is incremented. At last, the max-subpattern tree is traversed and all the patterns whose count is greater than a user defined threshold are shown as frequent periodic patterns.

2.2.3 Synchronous vs. Asynchronous

Synchronous. A pattern that occurs periodically without any misalignment or with no intervention of random noise is called a synchronous periodic pattern.

ABCADCBDCBAC

**C is a synchronous periodic pattern with a period 3.
Asynchronous. Asynchronous periodic patterns mean the patterns might be misaligned due to the intervention of random noise. The misalignment is accepted only up to a certain threshold value.

ABCDBCCBABC

*BC is an asynchronous periodic pattern due to the insertion of random noise events CB between the second and third occurrences of the patterns. Asynchronous periodic pattern mining can deal with the shift and distortion due to the presence of random noises in the periodic sequence.

Longest Sequence Identification (LSI) is the pioneering algorithm to mine asynchronous periodic patterns. For an asynchronous periodic pattern, the algorithm detects the longest subsequence containing it. It works in three phases. In the first phase it detects all potential periods, which requires single scan of the sequence. In the second phase all candidate 1-patterns are validated. An *i-pattern* is a pattern where *i* positions in the pattern are defined for e.g. *ab** is a 2-pattern with periodicity 3 where 2 positions out of 3 are defined. In the third phase the candidate i-patterns are formed from (i-1)-patterns. Validation of these i-patterns require single scan of the sequence. The second and third phases of LSI require multiple scans of the sequence [50].

2.2.4 Dense Periodic Patterns

A dense periodic pattern is the one in which the periodicity is focused on part of the time series or a pattern which occurs in small segments of time. Sheng et al. [36] have proposed an algorithm to find dense fragments from time series, where a dense fragment is a segment of time series where the distance between every pair of consecutive occurrences of each unique symbol is less than the distance threshold. All the dense fragments with lengths greater than the minimum length defined by the user, are retained. Then a max-subpattern tree is used to generate the partial periodic patterns. The algorithms prunes periods by finding the lower bound period for each symbol in its dense fragment. Periods below lower bound are safely pruned.

2.2.5 Approximate Periodic Patterns

Noise and imprecision exists in most of the real world data, so there is the need to mine approximate patterns. The previous algorithms which tolerate replacement, insertion, and deletion noise mine approximately repeating periodic patterns [42]. Y.L. Zhu el al. have mined approximate periodic patterns from hydrological time series [52]. Based on asynchronous periodic pattern and partial period mining with suffix tree, to mine multi-event asynchronous periodic patterns based on modified suffix tree representation and traversal, with a dynamic candidate period intervals adjusting method, which avoids period omissions and discover more local periodic patterns. The majority of the approaches to discover periodic patterns require other information such as the length of the pattern to discover or they need to assume a period (e.g. hour, day, week). This makes the adaptation of such patterns not ideal for the problem of UI log segmentation, where the length of the routines is unknown.

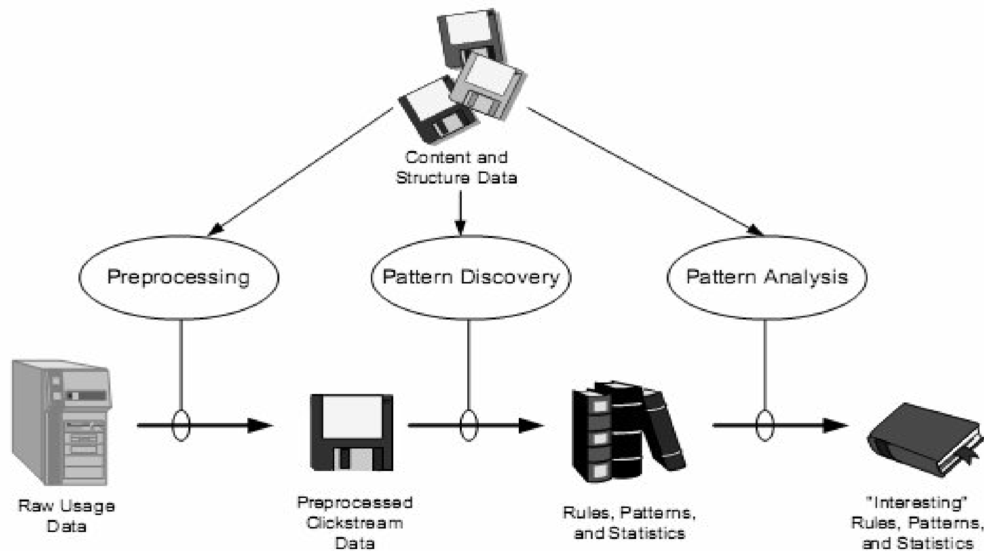

Figure 2.3: Web-usage mining process

2.3 Web-usage Mining

Web-usage Mining deals with understanding the user behavior, when interacting with a web site. This information is stored in Web server logs. It employs data mining techniques to extract patterns from the Web server logs [44]. Data collection is the first step of in Web-usage mining, which consists in gathering data from sources like Web servers, Cookies, user input from forms and proxy servers. Web-usage mining is composed from three stages that are similar to the aforementioned stages of RPM, besides the step of data collection [41]. In Figure 2.3 :

1. **Data preprocessing** contains three steps: data cleaning, user identification and session identification.

2. **Pattern discovery** applies data mining techniques to processed data with the goal of extracting relevant patterns.

3. **Pattern analysis** Uninteresting rules are ruled out and analysis is done using knowledge query mechanism.

Furthermore, a session can be defined as a set of pages visited by a single user during one particular visit to a website [35]. Sessions are usually identified on time or navigation. Traditional session identification algorithms are based on a fixed timeout.

2.3.1 Time-Oriented Heuristics

One approach to identify sessions is through time-oriented heuristics. Kapustra et al. [24] assume that the user visits multiple web pages in one session until the user finds the content page with the required information. There, the user spend a considerable amount of time compared with the navigation pages. The content page is considered the end of the session.

Chen et al. [10], designed an algorithm whose purpose is to find maximal forward references (longest sequences of Web pages visited by a user without revisiting some previously visited page in the sequence) from very large Web logs. The approach considers two types of sessions:

- α**-interval session:** ensures that the duration of a session may not exceed a threshold of 30 minutes. β**-gap session:** ensures that the time between any two consecutively visited pages may not exceed a threshold of 20 minute.

The algorithms define a URL node structure which stores the URL, the user's access time and a pointer to the next URL node. Next, the maximal forward reference session is calculated using interval session and gap session.

These methods could be applied in the context of RPM. However, in the case of the algorithms based on timeout, users may be involved in different activities when performing a task. Moreover, users usually perform tasks in batches and the time difference between two tasks may be smaller than the time difference between events in the same task, causing an incorrect segmentation.

2.3.2 Navigation-Oriented Heuristics

Navigation-oriented heuristics do not consider the time a user spends on a Web page, instead exploit the fact that users reach pages through hyperlinks rather than typing URLs. According to Cooley et al. [11] a requested Web page *w* that is not reachable from previously visited pages, should be assigned to a different session. The heuristic accounts for the fact that *w* may not be accessible from the page immediately accessed before it and the user can backtrack to a page visited earlier, from which *w* is reachable. The backwards moves are not always accessible and in this scenario the heuristic reconstructs the shortest path of backward moves to *w* and adds it to the user's session.

Cooley et al. [12] proposed another heuristic based on the referrer information of a URL request, which is the page from which the request was issued. This heuristic states that the referrer of a requested page should be a page in the session; else, the page is assigned to another session. If the page has an empty referrer, then it is likely to be the first page of a new session.

These approaches can only be used in the context of Web interactions, given that they are based on Web specifics, as the approaches proposed in [11] and [12]. Therefore, the adaptation of such techniques to work with multiple applications could be challenging.

2.4 Correlation of event logs for process mining

As evidence of processes execution information systems produce event logs. An event log consists of a set of events. Each event represents an executed activity in the business process. An event has a case identifier, a timestamp, among other context data [5]. The event log is comparable to the UI log, where the events are equivalent to UIs. When the events in an event log do not contain explicit case identifiers, they are said to be uncorrelated.

The approach in [5] deduces the case identifiers of the unlabeled events, and generates a set of labeled event logs with ranking scores. In a preprocessing step the algorithm constructs a relationship matrix that represents the relations among the activities to handle cyclic behavior. It has as inputs:

1. A process model.
2. An unlabeled event log with activity and timestamp.
3. Activity heuristics are data about the execution duration of each activity.
4. Threshold ranking-score, which is used to suppress generated labeled logs with a ranking score less than the threshold.

The algorithm is based on some assumptions in order to work correctly. First, there is no waiting time between activities. Each event has a timestamp that represents the completion time of an activity and the start time of the next activity. The second assumption is that the input process model must be dead- and live-lock free. The third assumption is that there is one start activity not contained in any loop.

This approach assumes that the process model is given as input. However, a process model is not available since the objective is to identify the routines in the log.

2.5 String Matching Basic Concepts

Given a UI log, which has been defined as a sequence of UIs, segmentation can be mapped to that of string matching, treating the UI log as a string and applying string matching methods to extract repetitive routines from the unsegmented UI log. Finding the occurrences of a pattern in a given text is also widely studied in bioinformatics. Suffix trees are important data structures in string processing and comparative genomics. However, suffix trees suffer from relatively large memory requirements; even recently improved implementations of linear time constructions still require 20 bytes per input character in the worst-case [26]. This problem has repercussions in large-scale applications like the repeat analysis of whole genomes [27]. More efficient structures exist; among them is the suffix array.

2.5.1 Suffix Tree

A suffix tree of a sequence *T*, where $n = |T|$, is an index structure than can be computed and stored in $O(n)$ time and space [48]. Its applications can be classified into tree traversals [20]:

- Bottom-up traversal of the complete sufix tree.
- Top-down traversal of a subtree.
- Traversal of suffix tree using suffix links.

i	**Suffix**
0	mississippi$
1	ississippi$
2	ssissippi$
3	sissippi$
4	issippi$
5	ssippi$
6	sippi$
7	ippi$
8	ppi$
9	pi$
10	i$
11	$

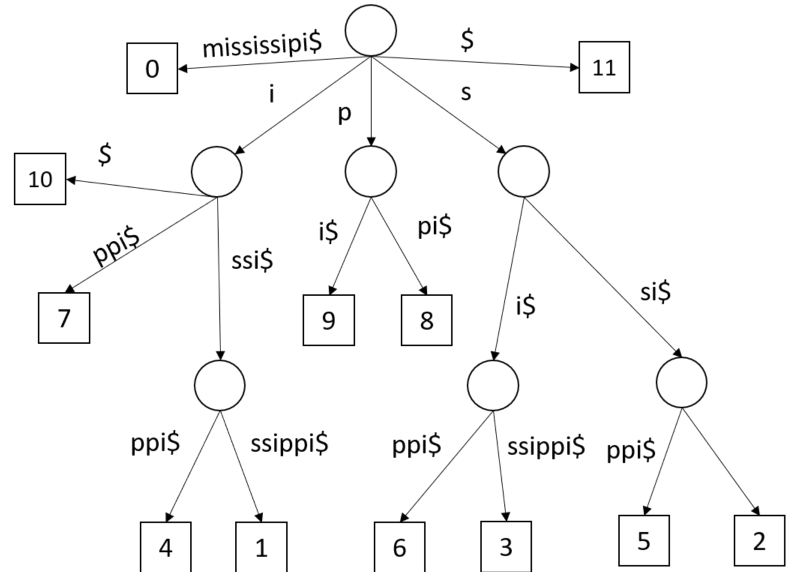

Figure 2.4: Suffix tree for T = mississippi

A suffix tree for the sequence *T* is a rooted directed tree with $n+1$ leaves. Except for the root, the internal nodes have at least two children. Each edge is labeled with a nonempty substring of $T\$$. When two edges start at the same node, they cannot begin with the same character. Its key feature is that for any leaf l, the concatenation of the edge-labels on the path from the root to leaf l spells out the string $T[i \dots n-1]\$$. Figure 2.4 shows the suffix tree for T = mississippi as an example.

2.5.2 Suffix Array

The suffix array of a sequence *T*, where n is the length of *T* $n = |T|$, requires 4 bytes per input character, and it can be constructed in $O(n)$ time in the worst case. Queries of the type "Is P a substring of T?", where $m = |P|$ can be answered in optimal $O(m)$ time [20].

Let Σ be a finite ordered alphabet. The suffix array SA of a string *T* is an array of integers from 0 to *n*, containing the indexes of the lexicographically ordered *n+1* suffixes of the string *T$*, where $ is considered as the lexicographically last symbol of the alphabet. It requires *4n* bytes. $T[i]$ denotes the character at position *i* in *T*, for $0 \leq i \leq n$. Then, $T_i = T[i \dots n-1]\$$ denotes the *i*th nonempty suffix of the string *T*$. In Figure 2.5 column *SA* contains the index in *T* of the suffixes in column $T_{SA[i]}$. For example, when $i = 0$, $SA[i] = 7$ and the character at position 7 in *T* is "*i*". So, $T_{SA[i]} = T[SA[i] \dots n-1] =$ "*ippi*$" The inverse suffix array *iSA* is a structure of size *n+1* such that $iSA[SA[q]] = q$ for $0 \leq q \leq n$.

2.5.3 Burrows Wheeler Transform

The Burrows-Wheeler Transform (BWT) is an algorithm that takes a block of data and rearranges it using a sorting algorithm [1]. The resulting output block contains the same data elements that it started with, differing only in their ordering. The transformation is reversible,

meaning the original ordering of the data elements can be restored with no loss of fidelity.

This may be defined in relation to the suffix array as follows. The result is a BWT array such that $BWT[i] = T[SA[i] - 1]$ if $SA[i] > 0$, otherwise is the last symbol of the original string. The BWT is stored in n bytes and constructed over the suffix array in $O(n)$ time.

In Figure 2.5, when $i = 0, BWT[0] = T[6] = s$. We can analyze the case when the condition $SA[i] > 0$ is not met at $i = 4, BWT[4] = \$$.

2.5.4 Repeating Substrings and Extendibility

Abouelhoda et al. [1], state that pair of substrings $R = ((i_1, j_1), (i_2, j_2))$ is a repeat iff $(i_1, j_1) \neq (i_2, j_2)$ and $T[i_1 \ldots j_1] = T[i_2 \ldots j_2]$. The length of R is $j_1 - i_1 + 1$. A repeat is *left-extendible* if $T[j_1 - 1] \neq T[i_2 - 1]$ and a *right-extendible* (RE) $T[j_1 + 1] \neq T[i_2 + 1]$. It is a *maximal* or *nonextendible* (NE) if is both not LE (NLE) and not RE (NRE). For example, in string T = mississippi$, substrings "issi", "iss" and "ssi" are repeats of length 4, and 3 respectively. Substring "issi" is a maximal repeat but "iss" and "ssi" are not. However, "ssi" is left-extendible: if we extend it by the character to the left, the character that precedes the substring is *i*. Analogously, the substring *iss* is right-extendible. In Algorithm 1 we present the algorithm used to obtain all the NE repeats.

Algorithm 1 Determine if the repeat is NLE [19]

```
1: function NLE(i,j)
2:     λ ← BWT[i]
3:     i' ← i + 1
4:     while i' ≤ j and λ = BWT[i'] do
5:         i' ← i' + 1
       return (i' ≤ j)
```

2.5.5 Longest Common Prefix

The *Longest Common Prefix* (LCP) has been an important tool in pattern matching. Some of its applications [3] include: compression of texts, calculating the Edit Distance between two strings by computing mismatches, finding maximal repeats in genomic sequences, among other applications. Due to LCP's importance in our approach, we start this subsection with definitions based on the LCP and derive algorithms that exploit its properties.

Let *A* and *B* be strings over Σ. The LCP between the two strings is the longest substring shared by them. For future reference, we denote the length of the LCP of A and B as *lcp(A, B)*. We can obtain the LCP via suffix tree construction, Lowest Common Ancestor queries, or suffix array construction and LCP queries.

For example, between suffixes $T[SA[1]] =$ "issippi$" and $T[SA[2]] =$ "ississippi$" the LCP of both suffixes is "issi" with length 4; hence, $lcp(T[SA[1]], T[SA[2]]) = 4$.

LCP array

The LCP array is an array of integers in the range 0 to n, which stores the lengths of the LCPs between consecutive sequences in the suffix array $LCP[k] = lcp(SA_{[k-a]}, SA_{[SA_k]})$ for

$1 \leq k \leq n$. LCP array can be computed during the construction of the suffix array or in linear time from the suffix array. It requires $4n$ bytes in the worst case.

In the column LCP of Figure 2.5 we have the LCP array. $LCP[0] = 0$. $LCP[1]$ stores $LCP(T[SA[0]], T[SA[1]]) = 1$, $LCP[2] = LCP(T[SA[1]], T[SA[2]]) = 2$ and so on.

LCP intervals

Abouelhoda et al. [1] introduced the concept of *lcp-intervals*. An interval (i, j), where $0 \leq i < j \leq n$ is called an lcp-interval (denoted by $(l; i, j)$) of lcp-value *l* if:

1. $LCP[i] < l$
2. $LCP[k] \geq l$ for all *k* with $i + 1 \leq k \leq j$
3. $LCP[k] = l$ for at least one *k* with $i + 1 \leq k \leq j$
4. $LCP[j + 1] < l$

Kasai et al. [25] present an algorithm to simulate bottom-up traversal of a suffix tree with a suffix array and the LCP array. It computes all the lcp-intervals of the LCP array. In the

Algorithm 2 Algorithm to compute the lcp-intervals

```
1: push(⟨0; 0, ⊥⟩)
2: for k ∈ (1,n) do
3:     lb ← k - 1
4:     while LCP[k] < top.lcp do
5:         top.rb ← k - 1
6:         interval ← pop
7:         Report(interval)
8:         if LCP[k] > top.lcp then
9:             push(⟨LCP[k]; lb, ⊥⟩)
```

algorithm, we denote the lcp-intervals as the tuple $\langle lcp; lb, rb \rangle$. As an example, the interval (0, 3) in Figure 2.5 has an lcp-value of 1, so the lcp-interval is represented by (1; 0, 3). The interval (1, 2) has an lcp-value of 4, represented as (4; 1, 2).

LCP tree

The lcp-interval tree of a suffix array is not built but allows us to simulate tree traversals. Algorithm 2 pushes the first element onto the stack; $\perp$ stands for undefined value for interval's right boundary. The algorithm has the usual *push* and *pop* operations; *top* provides a pointer to the stack.

An interval (m;l,r) is said to be embedded in (n;i,j) if $i \leq l < r \leq j$ and $n > m$, when an interval (m;l,r) is embedded in (n;i,j), then (m;l,r) is a *child interval* of (n;i,j). This parent-child relationship constitutes a virtual tree, the *lcp-interval tree*. Its root is the interval (0;0,n). Intervals where $[l, l]$ are *singleton intervals* and they are left implicit on the tree. Continuing the example of Figure 2.5 in the leftmost subtree represented with the lcp-interval (1;0,3), the child interval is (4;1,2) and the singleton intervals [0,0] and [2,2].

i	SA	LCP	BWT	$T_{SA[i]}$
0	7	0	s	ippi$
1	4	1	s	issippi$
2	1	4	m	ississippi$
3	10	1	p	i$
4	0	0	$	mississippi$
5	9	0	p	pi$
6	8	1	i	ppi$
7	6	0	s	sippi$
8	3	2	s	sissipi$
9	5	1	i	ssippi$
10	2	3	i	ssissippi$
11	11	0	i	$

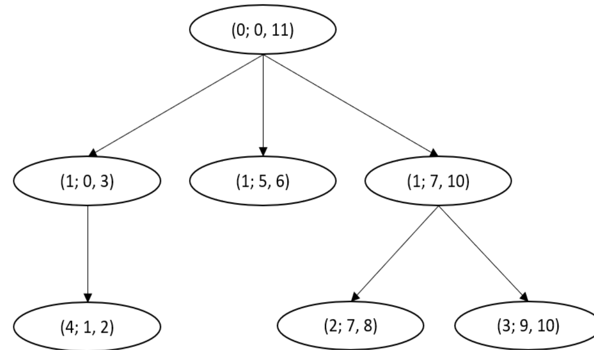

Figure 2.5: Suffix array of the string T = mississipi$ and its lcp-interval tree

2.5.6 Finding Maximal Repeats

The algorithm in [37] makes use of the concepts of LCP array and the BWT array. It requires *5n* bytes of storage, plus the stack space. It outputs a set of lcp-intervals.

Given a threshold p_{min}, which is the minimum lcp-value of the interval, and a range in SA, Algorithm 3 outputs complete NE repeats. LB (Left Boundary) is the stack to store the leftmost positions *i* at which there is an increase in the LCP value. Intervals (j, q) are pushed onto LB at position *j* when the lcp increases and popped when it decreases. When a repeat is popped, the algorithm identifies the repeat as NRE. To verify if a repeat is also NLE, we use the function NLE in Algorithm 1. Variable *prevNE* stores the lefthand position of the repeat (p; i, j).

2.5.7 Top-down traversals

In order to simulate the parent-child relations of the lcp-tree using a suffix array, we must enhance it with additional information. The child-table is a table of size n+1 from 0 to n and each entry contains three values: up, down, and nextIndex. Each of these three values requires 4 bytes in the worst case. The child-table stores the parent-child relationship of the lcp-intervals.

It can be computed in linear time by a bottom-up traversal of the lcp-interval tree, computing the up/down values and the nextIndex value of the child-table. The Algorithm 4 in [1] scans the lcp-table and pushes the current index on the stack if its lcp-value is greater than or equal to the lcp-value of top. Otherwise, elements of the stack are popped as long as their lcp-value is greater than that of the current index. Based on a comparison of the lcp-values of top and the current index, the up and down fields of the child-table are filled with elements that are popped from the stack during the scan.

Algorithm 3 Algorithm to compute all NE repeats of period $p \geq p_{min}$

1: $j, p, q, prevNE \leftarrow 0, -1, 0, 0$
2: push(LB;0,0)
3: **while** $j < n$ **do**
4:     **repeat**
5:         $j, p, q \leftarrow j + 1, q, LCP[j + 1]$
6:         **if** $q > p$ and $q \geq p_{min}$ **then**
7:             push(LB, j, q)
8:     **until** $p > q$
9:     **repeat**
10:         **if** $prevNE \geq i$ **then**
11:             Report (p; i, j)
12:         **else if** NLE(i,j) **then**
13:             $prevNE \leftarrow i$
14:             Report (p; i, j)
15:     **until** $top(LB).lcp \leq q$
16:     **if** $top(LB).lcp < q$ and $q \geq p_{min}$ **then**
17:         push(LB, i, q)

Algorithm 4 Algorithm to build the columns up/down values from child-table

$lastIndex \leftarrow -1$
$push(0)$
for $i \in (1, n)$ **do**
    while $LCP[i] < LCP[top]$ **do**
        $lastIndex \leftarrow pop$
        if $LCP[i] \leq LCP[top]$ and $LCP[top] \neq LCP[lastIndex]$ **then**
            $childtab[top].down \leftarrow lastIndex$
    if $lastIndex \neq -1$ **then**
        $childtab[i].up \leftarrow lastIndex$
        $lastIndex \leftarrow -1$
    $push(i)$

Algorithm 5 of the *nextIndex* verifies if the condition $LCP[i] = LCP[top]$ is true. If it is true then *i* is assigned to $childtab[top].nextIndex$. The child-table can be constructed in linear time and space. However, it is possible to reduce the space requirement of the child-table.

Algorithm 5 Algorithm to construct the nextIndex field of child-table

```
push(0)
for i ∈ (1, n) do
    while LCP[i] < LCP[top] do
        pop
        if LCP[i] = LCP[top] then
            lastIndex ← pop
            childtab[lastIndex].nextIndex ← i
    push(i)
```

Once we built the child-table, to locate the child intervals of (l; i, j) in constant time is to find the minimum lcp in the interval[i, j]. The child intervals of [i, j] are the intervals $[i, i_1 - 1], [i_1, i_2 - 1], \ldots, [i_k, j]$.

Abouelhoda et al. [1] proposed an algorithm *getChildIntervals* to calculate the child intervals of a given interval [i, j] in time $O(|\Sigma|)$. This function can be modified to a function *getInterval*, with an lcp-interval [i, j] and a character $c \in \Sigma$ and returns the child interval [l, r] of [i, j]. Suffixes in [l, r] share the same prefix.

Algorithm 6 Algorithm to find the child intervals of a given interval [i, j]

```
 1: intervalList ← []
 2: if i < childtab[j + 1].up ≤ j then
 3:     i_1 ← childtab[j + 1].up
 4: else
 5:     i_1 ← childtab[i].down
 6: add(intervalList, (i, i_1 − 1))
 7: while  do
 8:     i_2 ← childtab[i_1].nextIndex
 9:     add(intervalList, (i_1, i_2 − 1))
10:     i_1 ← i_2
11: add(intervalList, (i_1, j))
```

2.5.8 Range Minimum Query

A Range Minimimum Query ($RMQ(i, j)$) for an array L, asks for an index *k*, such that $i \leq k \leq j$ and $L[k] = minL[q]|i \leq q \leq j$. An RMQ can be answered in constant time provided that the array is appropriately preprocessed [1]. Possible approaches to solve the RMQ task are listed below:

1. **Sqrt-decomposition** answering each query in $O(\sqrt{n})$. Preprocessing is done in $O(n)$

$RMQ_A(2,7) = 3$

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	4	3	1	6	7	8	9	1	7

Figure 2.6: Range Minimum Query example

2. **Segment tree** answering each query in $O(logn)$. Preprocessing is done in $O(n)$
3. **Sparse table** answering each query in $O(1)$. Preprocessing is done in $O(nlogn)$.
4. **Cartesian Tree** answering each query in $O(1)$. Preprocessing is done in $O(n)$.

The applications of RMQ include computing the Lowest Common Ancestor (LCA) and finding the LCP.

2.5.9 Approximate String Matching

The k-difference problem consists of finding all occurrences of a pattern P in a string T that have an edit distance at most k from P. Huynh et al. [21] study the case in which T is fixed and then preprocessed into an indexing data structure (suffix array) so that any pattern query can be answered faster. One of the motivating applications of the offline version of the problem is DNA sequence searching. This application requires finding DNA subsequences over some known DNA genome sequences like the human genome. Since the genome sequence is very long, it would be desirable to preprocess it to accelerate pattern queries. This problem presents similarities with the one presented previously in the current research. An algorithm that tackles this problem is described next.

Consider a text T, with $|T| = n$ and a pattern P, with $|P| = m$, both strings over a fixed finite alphabet Σ. The k-difference problem is to find all j such that the edit distance between P and some substring starting at j in T is k. Let $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_m$ be strings over σ. The edit distance between X and Y, denoted by $dist(X, Y)$, is the minimum number of character deletions, replacements, and insertions to convert X to Y. Distance $dist(X, Y)$ can be evaluated in time O(mn) by using a very simple form of dynamic programming. The method evaluates a $(m + 1) \times (n + 1)$ table *e* such that $e[i, j] = dist(x_1 \ldots x_i, y_1 \ldots y_j)$. Entries in one column are used to evaluate entries in the next column.

The editing trace from X to Y is any sequence $T = \tau_1\tau_2 \ldots \tau_q$ of character operations applied on positions in X to get Y, ordered from the rightmost position in X to the leftmost position, where each τ_q is either *d (delete)*, *r (replace or change)*, *u (unchange)*, or *i (insert)*. Additionally, the editing trace does not store information about the character value. For example, a trace of the string $x = aaaaa$ to $Y = aaacb$ is represented as "*riduuu*", in which the *r* operation is to replace the rightmost character "a" by "b", the *i* operation is to insert "c", the d operation is to delete the second rightmost "a", and the remaining three *u* operations are to

1. Construct $F_{st}[1..m+1]$ and $F_{ed}[1..m+1]$, such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$. If the interval $[F_{st}[1]..F_{ed}[1]]$ is valid, then P has exact occurrences in T, we report occurrences in this interval.
2. $s' := 0$, $e' := n$
3. For i :=1 to $m+1$
   (a) (when $i \leqslant m$, deletion at i, ignored for 1-mismatch problem)
      i. $P' = P[1..i-1]P[i+1..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$ and the interval $[F_{st}[i+1]..F_{ed}[i+1]] = range(T, P[i+1..m])$, find $[st..ed] = range(T, P')$.
      iii. Report $[st..ed]$ if exist.
   (b) (when $i \leqslant m$, replacement at i) for each $c \neq P[i]$ in A
      i. $P' = P[1..i-1]cP[i+1..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$, find $[s''..e''] = range(T, P[1..i-1]c)$.
      iii. Given the interval $[s''..e''] = range(P[1..i-1]c)$ and the interval $[F_{st}[i+1]..F_{ed}[i+1]] = range(T, P[i+1..m])$, find $[st..ed] = range(T, P')$.
      iv. Report $[st..ed]$ if exist.
   (c) (insertion at i, ignored for 1-mismatch problem) for each c in A
      i. $P' = P[1..i-1]cP[i..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$, find $[s''..e''] = range(T, P[1..i-1]c)$.
      iii. Given the interval $[s''..e''] = range(P[1..i-1]c)$ and the interval $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$, find $[st..ed] = range(T, P')$.
      iv. Report $[st..ed]$ if exist.
   (d) (when $i \leqslant m$) Given $[s'..e'] = range(T, P[1..i-1])$, find $[s..e] = range(T, P[1..i])$. If not exist, exit out of the loop.
      $s' := s$, $e' := e$

Figure 2.7: The algorithm assumes a constant alphabet Σ and k $\geq$ 1. However the solution takes $O(mk|A| \cdot max(k, logn + occ)$ when using O(n) bits indexing data structure.

keep the remaining "a" unchanged. There is exactly one operation (u, d, or r) for each character in X, whereas there may be zero or more *i* operation. The cost $c(T)$ of T is the number of delete, insert or replace operations in T. Thus *dist(X, Y)* is the minimum possible cost of a trace from X to Y. The order of the operations is as follows: $u < d < r < i$. Figure 2.7 shows the algorithm for the k-difference problem. The algorithm is based on the following lemma:

- **Lemma 1.** Given a text T with $|T| = n$ and its SA, assume for a prefix P an interval in SA $[s, e]$. Then, for a character c, the interval $[s', e']$ of Pc can be computed in $O(logn)$ time

- **Lemma 2.** Given the prefixes P_1 and P_2 with their respective interval $[st_1, ed_1]$ and $[st_2, ed_2]$ we can find the interval $[st, ed]$ using SA and iSA such that P_1P_2.

- **Lemma 3.** Given SA and iSA, for a prefix P with interval $[s, e]$ and a character c we can find the interval for $[s', e']$ for cP, assuming that we have in advance an array C, such that for any c in alphabet, $C[c]$ stores all the total number of occurrences of c' in T, where $c' \leq c$.

For the same k-difference problem, Ghodsi [19] proposes a backtracking algorithm for discovering DNA seeds that finds approximate matches of a pattern in a large indexed text adapted for suffix arrays, which takes theoretically sublinear time. It is based on the affirmation that for every large pattern *m*, the pattern can be broken into smaller pieces and each piece can be searched for independently. The algorithms takes advantage of the main property of the suffix array: if some prefix of the suffix pointed to by the *i*th element in the suffix is equal

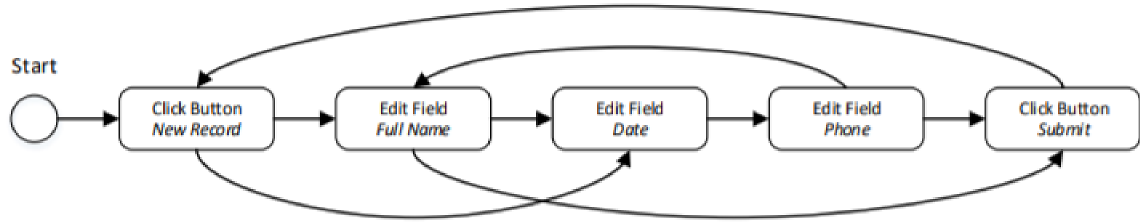

Figure 2.8: Example of a control-flow graph

to the element pointed to by *j*th, then for every $k\epsilon[i \dots j]$, the all elements in the suffix array pointed to by *k*th element share the same prefix.

The recursive algorithm over suffix array is described as follows: we keep two aligned windows of two suffix arrays and the length *lcp* of the prefix. The distance between the shared prefix of suffixes in the windows is stored in a global dynamic programming table. If the symbols from both sides of the window match, the dynamic programming table is updated and search for the next symbol after lcp. If the symbols do not match we divide the window in two smaller equal windows and use recursion to find the matches. The size of the window is greater or equal to one.

2.6 Related Work

The discovery of candidate routines for automation with RPA tools is a new research area. Before, we have described some of the approaches with a context similar to ours, and could, therefore, be applicable to the problem. However, after performing an analysis of the literature, we could only identify one work directly related with identifying routines and we describe it below.

The approach in [29] follows the RPM pipeline in Figure 2.2. It takes as input a preprocessed UI log, and its output is a set of routines candidate for automation. First, the UI log is decomposed into segments (segmentation). Then, candidate routines are identified by mining frequent sequential patterns from segments (candidate routine identification).

A preprocessing step is applied to the log to reduce noise impact. In this step, redundant UIs produce a log where most UIs are unique because they have a different payload. To discover the start and end of each task execution, we need to detect all the UIs that even having different payloads represent the same action. To this end, the UI log is normalized (a set of context parameters characterizes each UI). Given that a normalized log consists of context parameters, multiple executions of the same routine likely have the same sequence of UIs.

With the normalized log, the segmentation step starts by constructing a *Control-flow Graph* (CFG), where each vertex of the CFG maps to a normalized UI and the edges represent a directly-follows relation between the UIs mapped. The algorithm then detects the back-edges of the CFG by analyzing the *Strongly Connected Components* (SCCs).

The segmentation starts when the CFG is built. In an ideal scenario, once the task execution ends, the next UI should already be in the CFG, and the loop will be generated. In

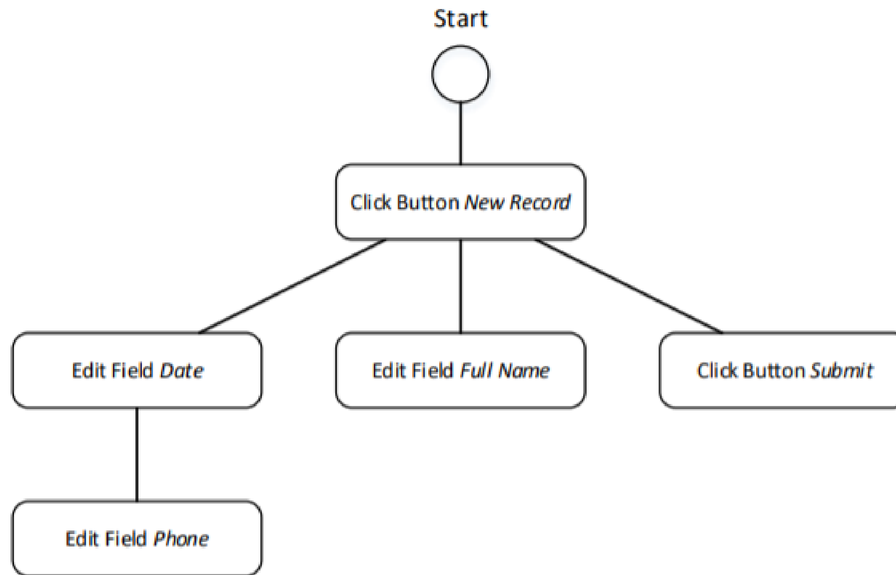

Figure 2.9: Dominator tree from CFG

such case, all the vertices in the loop correspond to a UI in the executed task. If several tasks are captured in the UI log, the graph would contain disjoint loops. Alternatively, if a task has repetitive subtasks, there would be nested loops. Figure 2.8 shows an example of CFG.

Then, back-edges are identified by analyzing the SCCs of CFG. Given a CFG, the dominator tree Θ is built. It captures the domination relations between vertices of CFG.

The non-trivial SCCs are discovered with the algorithm in [40] removing the trivial SCCs. For each SCC, the algorithm finds its header (the vertex that dominates all the other SCC vertices). The back-edges are identified to look for nested SCCs until there are no more back-edges and SCCs. In case an SCC does not have a header, back-edges cannot be detected; instead, the loop-edges are collected via depth-first search. Out of the loop edges, the algorithm stores the one which contains the target and source connected by the longest simple path.

As all of the back-edges have been collected, the targets and sources of the back-edges are retrieved. Every UI corresponding to a target is a possible starting point for a segment. Similarly, the UIs that are the source of a back-edge is possible ending points for the segment. Next, the algorithm scans all the UIs in the log, and when a starting point is found, a new segment is created. This strategy assumes that the same starting UI will follow the ending UI of a segment. If the segment is not a start UI, then the algorithm checks if it is within the segment, discarding it as noise if not. If the condition is true, the algorithm appends it to the segment and verifies an ending UI. When it is true, the segment is added to the set of segments; otherwise, the algorithm continues analyzing the UIs.

Finally, with the segmented UI log, the algorithm identifies candidate routines. The candidate routines identification step is based on sequence mining algorithms. A sequential pattern within a UI log is a sequence of normalized UIs occurring in the same order in different segments but with possible gaps between UIs belonging to a pattern.

The approach is capable of discovering multiple variants of the same routine when the

UIs occur in different orders. However, while the approach can handle multiple ends, all routine executions should start with the same UI.

2.7 Chapter Conclusions

We reviewed the concepts necessary to address the problem of discovering RPA routines, covering the concept of UI log and the pipeline of Robotic Process Mining, addressing the challenges and possible approaches to solve them. Besides, this chapter has addressed current approaches to tackle the problem of finding candidate routines for automatization from unsegmented UI logs. We have analyzed techniques in similar contexts, such as Web Usage Mining, and presented how they could be applied to the current problem besides the problems they could present. Some of them consider that a model is given, some are not possible to adapt to work with multiple applications (which is required in our context). In contrast, others rely on the time to perform the segmentation step (this could lead to imprecision).

Moreover, we present string matching concepts applicable to our approach, where the input UI log may be treated as a string. Approximate string matching algorithms were introduced also. Finally, the Graph approach is the only existing technique in the context of RPM. In the following sections, we will further analyze its results by comparing the graph approach and our approach. Our approach will be using string matching techniques to tackle the problem. In the next chapter, we detail the approach developed in this work.

Chapter 3

Contribution

This chapter describes our approach for identifying candidate routines from an unsegmented UI Log. First, we discuss how a UI log can be mapped into a sequence of symbols, referred to as a normalized log, that can be used as input for pattern mining techniques. We show how string matching algorithms can be used for extracting exact routines from the normalized log. This step generates a collection of exact routines. Then, we present an approach to finding approximate routines from the exact routines. Overall, our approach consists of a collection of algorithms that takes as input a UI log and generates as output a collection of routines. In the following sections, we describe the steps of the process in detail, including the normalization of the UI log.

3.1 UI Log Normalization

The interaction of a worker with one or more applications in a workstation is recorded by a software agent in a UI log. We part from the assumption that only one action can be performed at a specific time, and therefore, the UI log can be considered a sequence of UIs chronologically ordered based on their timestamps. The concept of UI can be better explained with the excerpt of a UI log in Table 3.1.

The example described is a simple yet realistic UI log based on one of the cases we use to test our approach in Chapter 4, which consists of the admission process of international and domestic students in a university. The software agent recollects information from two applications: Excel and Web browser. Moreover, rows are chronologically ordered by their *Timestamp*. Each row in Table 3.1 corresponds to an instance of a UI between the worker and the application.

The sequence of UIs emulates transferring the students' data from a spreadsheet in Excel to a Web form. There are three complete executions of a task (task traces) belonging to two records of different students, "John", "Mary", and "Jane". The main goal of the task captured in the example is to generate a new record of a student in the Web form. Hence, the end UI references the actual creation of a record by clicking the button "Submit". Please notice that when filling the Web form for "Jane", the checkbox for international students is not clicked, meaning that "Jane" is a domestic student. Thus, there are two variations of the task: one for international students and the other for domestic students. The start UI of the task trace

is colored in green, and the end UI in red. Such UIs are represented by the Timestamp, UI Type, a set of parameters, and their values. With this information we define a UI as a tuple $u = (t, \tau, P, V)$ where t is the timestamp, τ is the UI type, P is the set of parameters and V is the set of parameter values.

i	Timestamp	UI type	Application	Element Label	Column	Row	Value
...							
56	2019-02-21 16:07:21	copyCell	Excel		A	1	John
57	2019-02-21 16:07:24	clickTextField	Web	Name			
58	2019-02-21 16:07:28	paste	Web				
59	2019-02-21 16:07:30	clickCheckBox	Web	International			
60	2019-02-21 16:07:35	clickButton	Web	Submit			
61	2019-02-21 16:07:40	copyCell	Excel		A	2	Mary
62	2019-02-21 16:07:49	clickTextField	Web	Name			
63	2019-02-21 16:07:53	paste	Web				
64	2019-02-21 16:08:01	clickCheckBox	Web	International			
65	2019-02-21 16:08:05	clickButton	Web	Submit			
66	2019-02-21 16:08:12	copyCell	Excel		A	3	Jane
67	2019-02-21 16:08:16	clickTextField	Web	Name			
68	2019-02-21 16:08:22	paste	Web				
69	2019-02-21 16:08:35	clickButton	Web	Submit			
...							

Table 3.1: Excerpt of a UI log

To discover the routine, first, we need to identify the UIs that, having different payloads, represent the same action within a task. Different parameters will be relevant for its characterization accordingly to the UI type. Two types of parameters are identified within this framework:

- Context parameters store the information about the location where the UI was performed, namely, the application and its location within it.
- Data parameters capture the values used during the execution, e.g., the value of text fields.

Context parameters are likely to have the same values within a task execution, such as the *Application* field in Table 3.1. Data parameters have different values per task execution, i.e., *Value* field, which stores the value of the copied cell. Hence, we must select a set of relevant parameters to characterize the UIs. For example, *copyCell*, which represents the action of copying a cell from a spreadsheet in Excel, we have the context parameters *Application* and *Column*. The value of both parameters does not change among task executions. However, if the user wants to copy the student's last name besides the name, *Row* field would be the same for the name and the last name, yet the columns would differ. In this scenario, we would have two different actions: copy name and copy last name. Therefore, it is necessary to select parameters that properly identify a UI. To this end, for *copyCell* we concatenate the UI type with the column Row, as shown in column λ Table 3.2. Each UI is then characterized by the combination of different attributes needed to reconstruct the activity. We call this process the *normalization* of a UI.

UI	λ	λ'
...		
u_{56}	copyCell_A	A
u_{57}	clickTextField_Name	B
u_{58}	paste	B’
u_{59}	clickCheckBox_International	C
u_{60}	clickButton_Submit	D
u_{61}	copyCell_A	A
u_{62}	clickTextField_Name	B
u_{63}	paste	B’
u_{64}	clickCheckBox_International	C
u_{65}	clickButton_Submit	D
u_{66}	copyCell_A	A
u_{67}	clickTextField_Name	B
u_{68}	paste	B’
u_{69}	clickButton_Submit	D
...		

Table 3.2: UI log normalization and translation

Given the UI log, we normalize it by normalizing every UI in it. In a normalized log, the possibilities that multiple executions of the same task have the same sequence are high. Table 3.2 shows the normalization of the excerpt of the UI log. The first column contains the UI’s identifier, which is unique for each UI, λ is the label produced by the normalization of the UI. The last column, λ' is the mnemonic used for simplification in the following examples. Besides, for this reason, we merge B and B' to produce a simplified UI log excerpt. By inspecting λ we can clearly observe two routines within the UI log, with the initial and end UIs equally colored in green and red, respectively. The routines showed in the example using λ' are:

- $ABCD$ for international students, where C corresponds to the activity of ”clickCheckbox_International”. Two traces in the running example match this routine. Trace 1: $u_{56}, u_{57}, u_{58}, u_{59}, u_{60}$ and Trace 2: $u_{61}, u_{62}, u_{63}, u_{64}, u_{65}$.
- ABD for domestic students ”clickCheckbox_international” is not performed. One trace in the running example matches this routine. Trace 1: $u_{66}, u_{67}, u_{68}, u_{69}$.

$$ABCD$$

In this approach, a domain expert manually selects the set of context parameters that will constitute the UIs’ labels. The above is formally defined as follows.

Definition 3.1.1 (User Interface log)**.** *Let U be the set of UIs $U = u_0, \dots, u_{n-1}$, where a UI log L is a subset of UIs in U with a total order defined by the binary relation ”happens before” $<$ such that $u_i < u_{i+1}$. Similarly, the function $\lambda : U \to \Sigma$ provides the label resulting from the combination of relevant parameters for a UI, such that $\lambda(u)$ is the label associated with the UI **u**. Σ is a finite alphabet of labels. For the sake of simplicity $L[0 \dots n-1] = \lambda(u_0) \cdot \lambda(u_1) \cdot \dots \cdot \lambda(u_{n-1})$.*

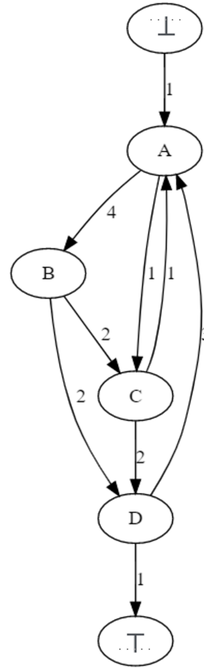

Figure 3.1: CFG of example log *L'*

With these concepts in mind, we may proceed to explain the main approach. The following section describes the algorithm to find the exact routines in a log.

3.2 Discovering Exact Routines in a UI Log

Before proceeding to explain the algorithms, in this section, we review some basic concepts from Chapter 2 and how they can be applied to the concept of UI log as seen in the previous section. Then, we explain the algorithms to find exact patterns using a running example.

3.2.1 Building basic structures

Intuitively, the suffix array (SA) is a data structure that allows us to identify the position of all the suffixes in a sequence, which are ordered lexicographically. Accordingly, a suffix is not stored explicitly in SA, but only the suffix index. Usually, the suffix array is found in conjunction with the *Longest Common Prefix* (LCP) and *Burrows-Wheeler Transform* (BWT). The LCP array is an auxiliary data structure to the SA. Considering two consecutive suffixes in SA, the LCP array stores the number of matching symbols between both suffixes. The BWT stores the preceding symbol of each suffix. We rely on these data structures to identify the repetitive sequences in the UI log. These concepts can be applied to the UI log, based on Definition 3.1.1, where the UI log is defined as a sequence of labels, where each label is comparable to a symbol in a string. Consider the UI log $L = ABDABCDACABCDABD$.

$$L' = \perp ABDABCDACABCDABD\top$$

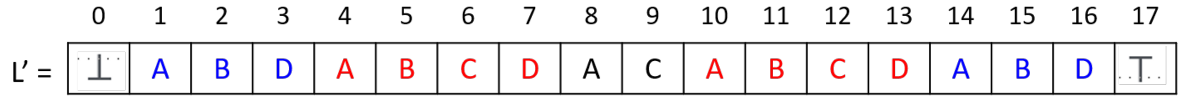

Figure 3.2: Sequence from UI log L'

$\perp(\top)$ corresponds to an artificial start UI (end UI) which is a prepended (appended) to the sequence for convenience. Therefore, we consider that both $\perp$ and $\top$ are included in the alphabet Σ but are not part of *L*. The subsequences of interest in this example are **ABCD** and **ABD**[1], which represent the college admission process of international and domestic students, respectively. There is an extra step *C* in the running example for international students, which represents clicking the "international" checkbox.

Ideally, the recording of the UI log should capture only the execution of the task. Nevertheless, usually, there are UIs that do not bring any value to the recorded task. Consequently, we added an error to the original UI log, where the user begins to execute the routine, continues with the next action, but instead of *B* performs *C* and needs to restart the sequence. To illustrate the process performed in *L*, we have generated a CFG of *L'* in Figure 3.1 only for visualization purposes. The CFG provides a graphical representation of the paths followed. The error is shown in the CFG as a loop between *A* and *C* performed only one time.

To extract exact patterns, we must calculate all the structures mentioned before in order to generate the LCP tree. But before, we must review the concept of repeat. We are interested in analyzing the sequences $l_1 = ABCD$ for international students and $l_2 = ABD$ for domestic students, and we will use them as a running example to illustrate the concept of repeat.

A repeating subsequence in a UI log is a subsequence that occurs more than one time within the log. Given that l_1 and l_2 appear twice in *L*, they can be considered repeating subsequences. Figure 3.2 presents UI log *L'*, where each symbol is the representation of a UI and has a specific position in the sequence. Subsequences of interest, l_1 and l_2, are colored in red and blue, respectively, while *AC* may be considered as noise.

A repeat is a set of locations in the UI log at which a repeating subsequence appears. To characterize the repeat, we need the length of the subsequence and the positions in the sequence where it occurs. For example, in the case of international students $|l_1| = 4$ and it occurs in positions 4 and 10 in L'. Thus, the tuple $(4; 4, 10)$ denotes the repeat for international students. In a similar manner, for domestic students $|l_2| = 3$, starting in positions 1 and 14 in L', the tuple $(3; 1, 14)$ denotes the repeat. As another example, the repeating subsequence $l = AB$, which is the longest common prefix shared by l_1 and l_2 is denoted by the repeat $(2; 1, 4, 10, 14)$. The set of positions of l contains the positions of l_1 and l_2.

Definition 3.2.1 captures formally the concept of repeat in the context of RPM.

Definition 3.2.1 (Repeats on an UI log)**.** *Let L be a UI log, and u is a repeating subsequence if it occurs at least twice in L. A repeat in L is a set of locations in L at which u occurs. It can be specified by the length of u,* $p = |u|$ *and the locations* $(p; i_1, i_2, i_3, \ldots, i_k), k \geq 2$.

[1]Without loss of generality, we merge UIs B and B' from the example in Table 3.1 as B only to produce a simplified UI log excerpt

The positions in the repeat can be expressed as a range using the suffix array, while the lengths of the repeats are calculated with the lcp. Next is detailed how the suffix array can be calculated and aids in representing the repeats.

Definition 3.2.2 (Suffix Array of a UI log)**.** *Let L be a UI log. We refer to the suffix $L[i \dots n]$, $i \in 0 \dots n-1$, as suffix **i**. Then, the suffix array **SA** is an array of integers in the range 0 to $n-1$. $SA[j] = i$ iff the suffix i is in the position j among all the ordered suffixes of L.*

Figure 3.4 shows the suffix array of $L' = \perp ABDABCDACABCDABD\top$ in the second column (SA).

Besides the suffix array, we need other structures for the algorithm, such as the LCP array. Towards this end, we define *lcp*. Below we define the concept of *longest common prefix*. The LCP array can be calculated during the construction of SA or in linear time from SA.

Definition 3.2.3 (Longest Common Prefix (LCP) and LCP Array)**.** *Let i_1 and i_2 be two suffixes in the UI log L. The longest common prefix of suffixes i_1 and i_2 is denoted by $lcp(i_1, i_2)$. The LCP array for L' is an array of integers in the range of 0 to $n-1$. in which $LCP[0] = 0$ and $LCP[j] = |lcp(SA[j-1], SA[j])|$.*

Continuing with the above example, Figure 3.4 gives the LCP array of L'. The lcp between the first two suffixes $lcp(SA[0], SA[1]) = ABCDAB$, then $LCP[1] = |ABCDAB| = 6$. Figure 3.3, shows the lcp for each pair of the first four suffixes in SA and the length of the lcp, stored in the LCP array. Consequently, *lcp-intervals* can be computed from the SA array and LCP array.

An *lcp-interval* is denoted as $(p; l, r)$ where p is the length of the repeat and *l* and *r* are the left and right boundaries, indicating the initial and last position of the repeat in SA. For example, the repeat for international students l_1 is represented by the tuple $(4; 4, 10)$. In SA, positions 4, 10 is stored in the range from $i = 0$ to $i = 1$ with SA[0] = 10 and SA[1] = 4. For l_1, the range in SA is given by the interval $(0, 1)$, where 0 is the left boundary and 1 the right boundary. Therefore, $l_1 = (4; 0, 1)$ as shown in Figure 3.4 with ABCD. While the repeat for domestic students l_2 represented by the tuple $(3; 1, 14)$ is stored in SA[2] = 1 and SA[3] = 3. Thus, its lcp-interval is $(3; 1, 14)$, as seen in Figure 3.4, with ABD. The repeat $l = AB$, $(2; 1, 4, 10, 14)$ is stored in the range from $i = 0$ to $i = 3$ in SA. in this case the lcp-interval is represented by the tuple $(2; 0, 3)$.

An lcp-interval $(p; l, r)$ in the UI log *L* is *left-extendible (LE)* if and only if all the UIs preceding *u* are equal, such that

$$L'[SA[l] - 1] = L'[SA[l+1] - 1] = \dots = L'[SA[r] - 1]$$

It is *right-extendible (RE)* if and only if all the UIs succeed *u* are equal, such that

$$L'[SA[l] + p] = L'[SA[l+1] + p] = \dots = L'[SA[r] + p]$$

A repeat is *nonextendible (NE)* or *maximal* if and only if it is not LE (NLE) and not RE (NRE). Following the example of international and domestic students, the repeat associated with international students $l_1 = ABCD = (4; 4, 10)$ is RE and NLE, so it would not be a

i	LCP																	
0	0	A	B	C	D	A	B	D	⊤									
1	5	A	B	C	D	A	C	A	B	C	D	A	B	D	⊤			
2	2	A	B	D	A	B	C	D	A	C	A	B	C	D	A	B	D	⊤
3	3	A	B	D	⊤													

Figure 3.3: LCP array excerpt

i	SA	LCP	BWT	L[SA[i]:]	Maximal
0	10	0	C	**AB**CDABD⊤	
1	4	5	D	**AB**CDACABCDABD⊤	
2	1	2	⊥	**AB**DABCDACABCDABD⊤	*
3	14	3	D	**AB**D⊤	*
4	8	1	D	ACABCDABD⊤	
5	11	0	A	BCDABD⊤	
6	5	4	A	BCDACABCDABD⊤	
7	2	1	A	BDABCDACABCDABD⊤	
8	15	2	A	BD⊤	
9	9	0	A	CABCDABD⊤	
10	12	1	B	CDABD⊤	*
11	6	3	B	CDACABCDABD⊤	*
12	3	0	B	DABCDACABCDABD⊤	
13	13	3	C	DABD⊤	
14	7	2	C	DACABCDABD⊤	
15	16	1	B	D⊤	
16	0	0	⊤	⊥ABDABCDACABCDABD⊤	
17	17	0	D	⊤	

Figure 3.4: Suffix array for UI log $\mathtt{L} = \bot ABDABCDACABCDABD\top$

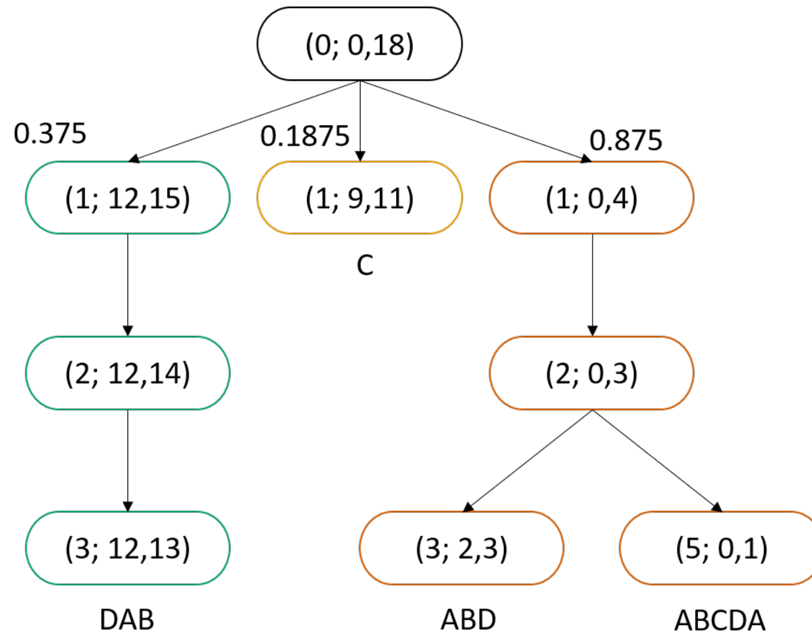

Figure 3.5: LCP Tree of UI log *L*

maximal repeat; since $l_2 = ABD = (3; 1, 14)$ is NRE and NLE, it is NE. The repeat $l = AB$ is NE, given that it is NRE and NLE. In the example, $ABD, ABCDA, C$ and DAB are all maximal repeats.

Through the Burrows-Wheeler Transform, it is possible to identify NLE repeats. Column BWT in Figure 3.4 contains the *Burrows-Wheeler transformation*. BWT table is stored in *n* bytes and constructed in one scan over the suffix array in *O(n)* time. The Burrows-Wheeler Transform is defined for a log *L* as a table such that for every $j, 0 \leq j \leq n, BWT[j] = L[SA[j] - 1]$ if $SA[j] \neq 0$; else if $SA[i] = 0$ we set $BWT[i] = \top$.

We introduce a critical concept for our algorithm: the *lcp-interval tree*. To shorten, we call it *lcp tree*. Two intervals can have a parent-child relationship, if the child interval $(l; i, j)$ is a subinterval of the parent interval $(p; m, n)$; $(l; i, j)$ is a subinterval of $(p; m, n)$ if $i \leq m < j \leq n$ and $p < l$. The intervals where only one repetition appears are called *singleton intervals* $[k, k]$ and are left implicit in the *lcp tree*.

From this parent-child relationship between LCP intervals a *lcp tree* can be built. The root of this tree is the interval $(0; 0, n - 1)$. The shorter prefixes appear near the root with longer intervals and longer prefixes with shorter intervals towards the leaves. Each of the lcp-intervals represented in the tree is called a *node*. The nodes with children are *intermediate nodes*, and the ones without children are the *leaves*. Although the lcp trees are usually only conceptual, we use them for explanation purposes. For example, AB $(2; 0, 3)$ would be the parent interval of $(3; 2, 3)$ and $(5; 0, 1)$ as seen in the rightmost subtree of Figure 3.5.

3.2.2 Algorithms to Discover Exact Patterns

We have the necessary ingredients to explain our approach with the structures described above. Our algorithm computes the enhanced suffix array and finds all the maximal repeats through

the Algorithm 3, defined in Chapter 2 in order to build the lcp tree. Figure 3.5 presents the lcp tree with only the maximal repeats for UI log $L = \bot ABDABCDACABCDABD\top$. Please, notice that the *lcp tree* as a simplified example contains only three branches (subtrees).

However, the general case usually presents a more significant number of subtrees. It is thus necessary to design pruning and selection strategies that identify the routines that are likely to produce quality routines. Consequently, we define heuristics to select the most relevant subtree to analyze. The intuition followed is to choose the subtree that legitimizes the routine or routines that cover the highest percentage of the UI log. The first step is to prune the tree; we remove the sequences containing a *leaf cycle*.

Definition 3.2.4 (Leaf cycle)**.**

$$l_c(p; l, r) = \arg\min_k \; 0 < k < p \land \exists n : l \leq n \leq r \land L[SA[n]] == L[SA[n] + k] \lor k = p$$

As explained in Definition 3.2.4, a leaf cycle l_c represents the length of the sequence associated with a node upto the point it starts again. To illustrate this concept, consider the leaves of the rightmost subtree in Figure 3.5. The sequence *ABD* corresponding to the leaf $(3; 2, 3)$ with $p = 3$ does not have a repeated symbol, therefore, $l_c = p = 3$. As counterexample, the sequence *ABCDA* in the leaf $(5; 0, 1)$ starts again in the fifth position. Hence, $l_c = 4$ cuts the repeating symbol *A* out of the sequence of interest, which becomes *ABCD*. The leaves in an lcp tree are then characterized by an lcp-interval and a leaf cycle.

After pruning the tree, we keep only intermediate nodes that do not present a cycle, yet the leaves may or may not present a cycle. Each node (intermediate or leaf) has a set of symbols covered by their lcp intervals. This is formally defined in Definition 3.2.5.

Definition 3.2.5 (Pattern Symbols, Strict Coverage)**.** *Let $n = \langle p; l, r\rangle$ be a node in the LCP Tree, and l_c be the length of the sequence upto the point it starts to repeat as shown in Definition 3.2.4. Moreover, we assume that SA is the underlying suffix array. The set of symbols covered by the underlying pattern, denoted as $PS(n, l_c)$ can be defined as follows:*

$$PS(n, l_c) = PS(\langle p; l, r\rangle, l_c) = \begin{cases} \bigcup_{i=l}^{r} \bigcup_{j=0}^{l_c} SA[i] + j & \text{if } n \text{ is a leaf node} \\ \bigcup_{d \in Desc(n)} PS(d, l_c) & otherwise \end{cases}$$

Where the function Desc(n) returns all the children of the node n. Moreover, the strict coverage of node n is given by

$$SCov(n, l_c) = \frac{|PS(n, l_c)|}{|L|}$$

For each legitimized leaf $\langle p; l, r\rangle$, with leaf cycle l_c its *Pattern Symbol (ps)* is given by the indexes stored in SA corresponding to each suffix within the length of the interval $[l, r]$. Taking into account the lcp tree used in previous examples, $n = (3; 2, 3)$ with $l_c = 3$. Then, $PS(n, l_c) = 1, 2, 3, 14, 15, 16$, corresponding to the two times the sequence ABD appears in L'. Figure 3.6 shows *PS* for both leaves of the subtree $(1; 0, 4)$.

However, for intermediate nodes *ps* is given by union of the PS of all its descendants returned by function $Desc(n)$; thus, PS would propagate bottom-up within a lcp tree. For

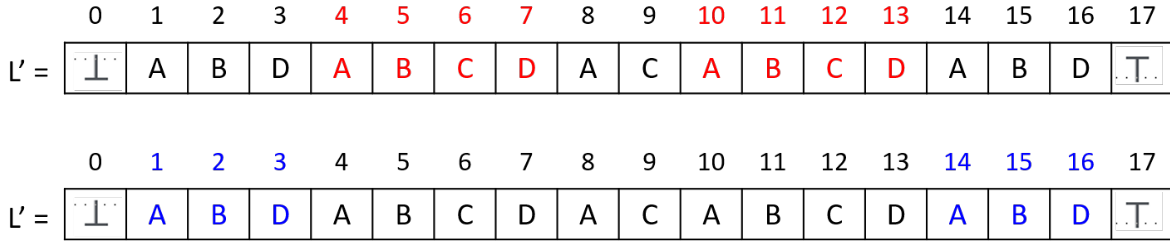

Figure 3.6: Pattern Symbol for leaf $\langle(5;0,1),4\rangle$

the case of node $(1;0,4)$, *ps* would be the union of $PS(\langle 3;2,3\rangle,3)$ with $PS(\langle 3;0,1\rangle,4)$, covering almost all the symbols in L, except for AC.

On the other hand, the *Strict Coverage (scov* calculates the percentage of the UI log L that is covered by a node. We can see in Figure 3.5 the *scov* above the root of each subtree, being the rightmost subtree the one with the highest *scov*. This process is the basis for our approach to finding the exact routines, which is described by Algorithm 7 and Algorithm 8.

Algorithm 7 DFS of LCP Tree

1: **procedure** TRAVERSE(current, **var** patterns, **var** ps)
2:     **if** current is LEAF **then**
3:         $l_c \leftarrow$ PatternSize(current)
4:         patterns $\leftarrow$ patterns $\cup$ $\{\langle current \mapsto ExtractPattern(current, l_c)\rangle\}$
5:         ps $\leftarrow$ ps $\cup$ $\{\langle current \mapsto LeafPS(current, l_c)\rangle\}$
6:     **else**
7:         nodePatterns, nodePS $\leftarrow \emptyset, \emptyset$
8:         **for** child $\in$ ChildrenOf(current) **do**
9:             Traverse(child, patterns, ps)
10:             nodePatterns $\leftarrow$ nodePatterns $\cup$ patterns(child)
11:             nodePS $\leftarrow$ nodePS $\cup$ ps(child)
12:         patterns $\leftarrow$ patterns $\cup$ nodePatterns
13:         ps $\leftarrow$ ps $\cup$ $\{\langle current \mapsto nodePS\rangle\}$

Algorithm 7, the procedure *Traverse* performs a depth first search on the lcp tree, calculating recursively the exact routines and *PS* associated with each node. It receives as input the current node (*current*), and two variables *patterns* and *ps*. *Traverse* has two cases, the same cases as in Definition 3.2.5:

- Current node is a leaf node.
- Current node is an intermediate node.

Initially, *Traverse* receives a node, being the root of a tree. The first case we explore is when the node is a leaf. The length of the cycle l_c is returned by the function *Pattern Size*. Then, *ExtractPattern* returns the exact routine found in the interval of *current*; the routine length is given by l_c. The routines are stored in the variable *patterns* and *ps* actualizes with the the current leaf *ps*.

When *current* is an intermediate node, the algorithm explores all of its children recursively and updates *patterns* and *ps* with each child's *patterns* and *ps*. In Figure 3.5 the patterns of the root of each subtree are the patterns of their leaves that are propagated through the intermediate nodes performing a bottom-up traversal. For example, the *patterns* associated with the root of the rightmost subtree (root = (1;0,4)) are ABD and ABCD. The routines found with Algorithm 7 are the same we presented at the beginning, corresponding to domestic and international students.

Algorithm 8 Algorithm to extract the exact patterns

```
 1: function EXACTPATTERNS(Root)
 2:     patterns, ps, cover, patternSet ← ∅, ∅, ∅, ∅
 3:     Traverse(Root, patterns, ps)
 4:     open ← Desc(root)
 5:     while SCov(cover)< α ∧ open ≠ ∅ do
 6:         Let s ∈ open w. highest coverage
 7:         if |ps(s) ∩ cover|/|ps(s) ∪ cover| < β then
 8:             patternSet ← patternSet ∪ patterns(s)
 9:             cover ← cover ∪ ps(s)
10:         open ← open \{s}
          return patternSet
```

In Algorithm 8, function *ExactPatterns* receives as input the root of the complete lcp tree (*Root*). The algorithm calls procedure *Traverse*, which receives *Root* and maps each node in the tree with their respective *patterns* and *ps*. Function *Desc(root)*, returns the roots of the subtrees, i.e. the children of *Root*. In the example, variable *open* stores $(1; 12, 15), (1; 9, 11), (1; 0, 4)$. While there are nodes in *open* and the minimum coverage threshold α has not been achieved, the algorithm will continue to explore the nodes in *open*. Next, the algorithm selects the subtree *s* with the highest coverage. Variable *cover* stores the *ps* of the discovered routines, whereas *patternSet* stores the discovered routines per se.

Up to this point, we have discovered the exact routines of one task that might have variations (e.g., the international and domestic students' variations). However, in a UI log, there could be multiple routines completely or partially different among them (in terms of UIs). For example, a UI log might have two routines $r1 = ABCD$ and $r2 = XYZ$ and to achieve high coverage, we should be able to find both of them.

Thus, it it critical to assess if other subtrees contain different routines. To this end, we use the Jaccard Index to measure the similarity between *cover* and *ps(s)*. Consider subtree roots $s_1 = \langle 1; 0, 4 \rangle$, which is the subtree with the highest coverage and $s_2 = \langle 1; 12, 15 \rangle$ the second highest coverage. In the first loop iteration, the exact routines $patterns(s_1)$ are added to *patternSet* and its *ps* to *cover*. So,

$$cover = \{1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16\}$$

Then, in the second iteration, if the threshold α is higher than 0.875, the node with the highest coverage is s_2. With

$$ps(s_2) = \{3, 4, 5, 13, 14, 15\}$$

The algorithm computes the Jaccard Index, measuring the number of symbols shared by *cover* and *ps*. It ranges from 0 to 1, 0 if they are completely different from each other, and one if they are the same. If the Jaccard Index is smaller than the threshold β, the routines generated from the subtree s_2 are legitimized; otherwise, the tree is not considered in the analysis. Continuing the example,

$$\frac{|ps(s_2) \cap cover|}{|ps(s_2) \cup cover|} = \frac{6}{14} = 0.43$$

In this case, they share a high amount of pattern symbols, so it is likely s_2 would be discarded from the analysis. The algorithm returns the discovered routines, which for the example, are ABD and $ABCD$. They correspond to the routines we are looking for. Please notice the missing subsequence *AC*, considered noise, corresponding to the indexes 8, 9. This situation can be appreciated better in Figure 3.6.

The algorithms present a greedy approach since it finds the local optima among the subtrees, relying on two heuristics:

- Selecting the subtree that maximizes the *Strict Coverage*
- Selecting only the subtrees that minimize the number of UIs shared among them (Jaccard Index).

Routines may present nested loops, hindering the identification of exact routines due to the definition of l_c. Therefore, to remove subtrees with nested loops from the analysis, we calculated the number of instances from each leaf in a subtree. Then, we calculate the interquartile range (IQR) and apply the formula:

$$bottom = Q1 - 1.5 * IQR; top = Q3 + 1.5 * IQR;$$

Consequently, subtrees with nested loops appear as outliers and are discarded.

With our approach in the example, we have achieved a coverage of 0.875. However, in a real-life UI log, the coverage could be seriously reduced if there are permutations in the order of the actions or mistakes. Therefore, it is critical to develop an algorithm able to identify approximate candidate routines. In the following section, we address this issue and present the approach we developed using approximate string matching techniques.

3.3 Approximate matching

At this point, we have collected all the exact candidate routines in the log. However, the exact routines found in the previous example do not cover the UI log completely. The reason lies in the human errors performed during the execution of the routine or variations in the order (permutations). In Figure 3.2, the subsequence *AC* is considered as noise, and these symbols are lost from the coverage. Hence, finding all the exact routines is not sufficient to cover the full extension of the UI log. To solve this problem, we part from the assumption that the errors occur less frequently than the correct exact routines, but they may happen more than once.

Noise may be identified in two cases:

- Leaves with very low frequency. These type of leaves are not legitimized as exact routines by the *ExactPatterns* algorithm. Therefore, they are considered as noise.

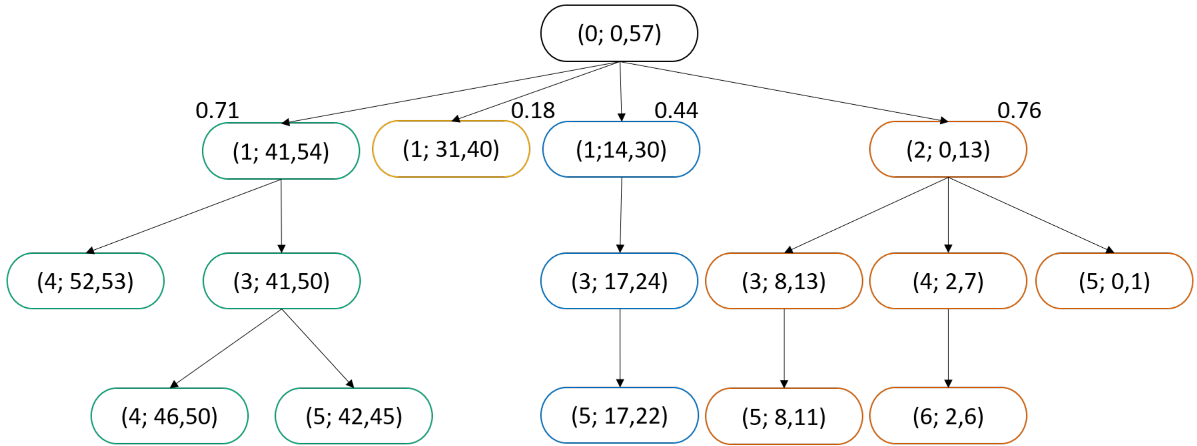

Figure 3.7: LCP Tree of UI log *M*

- Singletons. They are not shown in the LCP tree and are represented as an interval with only one occurrence in the form of $[l, l]$

The lcp tree of UI log M is in Figure 3.7. Assuming that we have already applied Algorithm 8 and extracted the exact routines. We work with the right subtree, given that it is the one with the highest coverage. The right node $(5; 0, 1)$ occurs two times while the sequences associated with the other leaves occur 4 and 5 times. Hence, node $(5; 0, 1)$ is considered noise. Furthermore, the second case of noise is also present in the same subtree. Please notice that between node $(3; 8, 13)$ and its respective child, the interval $[12, 13]$ is missing. The two existing singleton intervals $[12, 12]$ and $[13, 13]$ share with its parent the first three symbols *ABD*, which is the lcp-value in the interval $[8, 13]$, yet the next symbols differ. This scenario happens once again between node $(4; 2, 7)$ and its child $(6; 2, 6)$, with another missing singleton $[7, 7]$, sharing the first four symbols with its parent.

To retrieve the missing sequences, we use Algorithm 12. Our approach takes as input:

- *n* the length of the exact routine
- *k* the maximum difference
- *query* pattern the interval corresponding to the exact routine
- *approximate interval* or the search interval

Our algorithm does not exist in the literature; it is a non-trivial adaptation from the algorithm for the k-difference problem by Huynh et al. [21] and the algorithm to answer decision queries in [1]. This algorithm was not intuitively deduced and required an in-depth analysis through multiple stages of experimentation with UI logs of varying degrees of complexity. In fact, our algorithm parts from **Lemma 2**. Lemma 2 uses the suffix array (SA) and the inverse suffix array (iSA). iSA is a table such that $iSA[SA[q]] = q$ for any $0 \leq q \leq n$. Given two prefixes with their lcp-intervals $P_1 = (m_1; st_1, ed_1)$ and $P_2 = (m_2; st_2, ed_2)$, Lemma 2 finds the interval $[st, ed]$ such that P_1P_2. It receives as input both intervals and the length of P_1. As an example, take $P_1 = AB$, $m_1 = 2$ and $P_2 = C$, with intervals $[0, 13]$ and $[31, 40]$, respectively.

Algorithm 9 Given two prefixes P_1 and P_2, Lemma 2 finds the interval $[st, ed]$ for which P_1P_2

1: **function** LEMMATWO($[st_1, ed_1]$, $[st_2, ed_2]$, m_1)
2: st←min([$\forall st \in [st_1, ed_1]$ s.t. $st_2 \leq iSA[SA[st] + m_1] \leq ed_2$])
3: ed←max([$\forall ed \in [st_1, ed_1]$ s.t. $st_2 \leq iSA[SA[ed] + m_1] \leq ed_2$])
4: **return** $[st, ed]$

We want to know the interval for $P_1P_2 = ABC$. The result of $LemmaTwo([0, 13], [31, 40], 2)$ is the interval $[2, 7]$ corresponding to the sequence ABC.

Function *getInterval* receives as input a symbol in M and returns its interval. This function is an adaptation of Algorithm 6. For example, if we input A in $getInterval$, the result is the interval in SA of symbol A. Hence, $getInterval(A) = [0, 13]$. We use ***getInterval*** in ***Deflate*** in Algorithm 10. ***Deflate*** computes the interval of the symbol ***c*** in position ***m*** within the interval $[st_2, ed_2]$ in the UI log M. For ***c***, $[st_c, ed_c] = getInterval(c)$. Then, LemmaTwo obtains the interval for which P_1c, with intervals $[st_1, ed_1]$ and $[st_c, ed_c]$. The length of P_1 is given by *offset* + *m*.

Algorithm 10 Function Deflate

1: **function** D($[st_1, ed_1]$, $[st_2, ed_2]$, offset,m)
2: **return** LemmaTwo($[st_1, ed_1]$, getInterval($M[SA[st_2] + m]$), offset $+ m$)

Function *Translate* in Algorithm 11, takes as input an interval and the offset. The input interval $[st, ed]$ is an interval over the *local* subtree (the one with the highest coverage). The function *translates* $[st, ed]$ from the *local* subtree to an interval in the subtree of the symbol in positions *SA[st] + offset* and *SA[ed] + offset*. In our running example, we will use as input for *Translate* the interval $[0, 1]$ with an offset = 2. The result of $Translate([0, 1], 2)$ is the interval $[22, 23]$. The prefix of the suffixes in interval $[0, 1]$ is AB, then, the symbol in position *SA[st] + offset* is B. This resulting interval is located in the subtree beginning with B.

Algorithm 11 Function Translate

1: **function** T($[st, ed]$, offset)
2: **return** ($iSA[SA[st] +$ offset$]$, $iSA[SA[ed] +$ offset)

Now, with the necessary functions defined, we introduce the main algorithm to compute the approximate patterns from a given subtree. The approach takes into account only the insertions and deletions but not the replacements. Algorithm 12 receives two intervals: the interval $[st_q, ed_q]$ for a legitimized routine extracted with Algorithm 8 (exact interval) and the approximate interval $[st_a, ed_a]$. Besides the two intervals, the algorithm needs the length of the exact routine *n* and the maximum number of differences *k*. We use a heap *h* to store the *candidate* intervals and a list *closed* to store the approximate intervals found. Below, we enumerate the variables used in the algorithm and what they represent.

- **edop**: number of current edition operations. It should be less than *k*.

- $\mathbf{m_q}$: position in the approximate pattern.
- $\mathbf{m_a}$: position in the query pattern.
- $\mathbf{[lst_a, led_a]}$: approximate interval in the local subtree.
- **summary**: string that summarizes the operations performed.

The algorithm initializes the heap with the approximate interval. The exact interval is *translated* as $[tst_q, ted_q]$; if m_q = 0 then $[tst_q, ted_q] = [lst_a, led_a]$, which applies for the first iteration of the loop. The function *getChildIntervals* has been defined in Algorithm 6 and returns all the child local intervals $[i_o, j_o]$ of a given node. We translate each child and compute the lcp between the translated query interval and the translated child. Next, we have three cases:

- if the complete pattern has been completely covered we report the interval after *deflating* it (conditional in line 10).
- if the translated query interval is a subinterval of the translated child interval (conditional in line 14).
- if the number of edit operations is less than the maximum number of operations (conditional in line 19).

In the first case, $m_q + l$ is the number of symbols covered in the query pattern, and if the sum is greater than the length of the query pattern, we report the deflated approximate interval. Please notice that the interval $[lst_a, led_a]$ is an interval in the local subtree, while the interval $[i, j]$ is located in a translated subtree. The result of this function should be an interval in the local subtree. The discovered interval is added to the *closed* list to avoid analyzing the same pattern in future iterations. When a pattern is found, we obtain its PS and add it to the coverage of the exact routine.

The second case addresses the situation where the translated query interval $[tst_q, ted_q]$ is a subinterval of the translated child interval. If the number of edit operations is zero, we push into the heap *h* a tuple with the local child interval, which corresponds to an extended approximate interval. In the case where the number of edit operations is greater than zero, we push the child's local interval into the heap, yet actualizing the variables m_q and m_a by advancing the length of the lcp.

If the previous two cases are not true, we verify if the query pattern has already been covered and report it, following the same process as with the first case. If the pattern has not been covered, we push the tuples in the heap for the operations *insert* and *delete*. The operation *insert* advances one symbol on the side of the approximate pattern ($m_a + l + 1$). *Delete* advances one symbol on the side of the query pattern ($m_q + l + 1$). The number of edit operations is increased by one, and the values of m_q, m_a and the local approximate interval (through the *Deflate* function are actualized.

To illustrate this process, we use our running example in Figure 3.7. Given that the rightmost subtree has the highest coverage the algorithm selects it. The nodes with the highest coverage (5;8,11) and (6;2,6) are validated as exact routines. In the running example, the root of the subtree [0, 13] is the approximate interval, we take as exact routine the leaf (6; 2, 7) with length $n = 4$ and $k = 1$. Hence, $[st_a, ed_a] = [0, 13]$ and $[st_q, ed_q] = [2, 6]$. Please notice

in Figure 3.7 that $[0, 13]$ has three children. We will analyze the cases of the children $(4; 2, 7)$ and $(5; 0, 1)$.

Initially, all the variables are zero. We know the *Translate* function for every interval will produce the same input interval given that $m_a = m_q = 0$. The child to be analyzed is $[0, 1]$; lcp between $[0, 1]$ and $[2, 6]$ is $l = 2$. The first condition is not met and the exact pattern is not completely covered given that $m_q + l < n$. The second condition is not met either. The third condition is fulfilled but since $m_q + l + 1 < n$, we push the tuples for insert and delete operations. Function *Deflate* $D([0, 13], [0, 1], 0, 3)$, finds the interval for $M[SA[0] + 3] = C$. So, $LemmaTwo([0, 13], [31, 40], 3)$. Hence, we push into the heap: $\langle 1, 2, 3, [0, 12], \text{'- -i '}\rangle$. A similar situation applies to the *delete* operation.

For the second child, with interval $[2, 7]$ when we compute its lcp with the exact routine $[2, 6]$, $l = 4$ and $[2, 6]$ is a subinterval of $[2, 7]$. The condition $m_q + l \geq n$ holds; thus, we report the interval and summary. The summary is similar to the editing trace mentioned in Chapter 2, as it stores the edit operations performed to identify the approximate pattern. The edit operations insert and delete are represented as *"i"* and *"d"*. If symbols in the exact and approximate pattern are the same, we represent the unchanged symbols with *"-"*. This process repeats until there is no element in the heap.

Up to this point, we have discovered the exact and approximate patterns. Now, we have to assess the quality of the discovered routines and determine if they are significant. This is a vital step to evaluate the effectiveness of our approach and it will be the subject of the next chapter.

3.4 Chapter Conclusions

This chapter introduced the two developed methods to identify routines from a UI log: the exact matching approach and the approximate matching approach. The first one is a heuristic-based algorithm seeking to maximize the *Strict Coverage* and minimize the *Jaccard*. Moreover, we have seen that the algorithm to find the exact routines is the basis of the approximate algorithm due to the extracted exact patterns being the input for the approximate algorithm. Approximate matching techniques allow us to analyze the patterns with noise that the exact matching method is not able to identify.

The chapter focused on explaining our methods and the basis behind them. However, to assess the effectiveness of our approaches, we must answer whether the extracted patterns are significant or not, leading us to select appropriate metrics to evaluate the routines. On that account, we devote the next chapter to analyze the results of both approaches and comparing them to the State of the Art approach, already reviewed in Chapter 2.

Algorithm 12 Algorithm to find approximate patterns

```
1:  function APPROXIMATE MATCHING([st_q, ed_q], [st_a, ed_a], n, k)
2:      h, closed ← ∅, ∅
3:      offer(h,⟨0, 0, 0, [st_a,ed_a],''⟩)
4:      while h ≠ ∅ do
5:          edop, m_q, m_a, [lst_a, led_a], summary ← take(h)
6:          [tst_q, ted_q] ← T([st_q, ed_q],m_q)
7:          for [i_o, j_o] ∈ getChildIntervals([lst_a, led_a]), s.t. [i_o, j_o] ⊈ closed do
8:              [i, j] ← T([i_o, j_o], m_a)
9:              l ← LCP(min(tst_q, i), max(ted_q, j))
10:             if m_q + l ≥ n then
11:                 newInt ← D([lst_a, led_a],[i,j],m_a,n-m_q)
12:                 REPORT (newInt, summary + '-'*(n-m_q))
13:                 closed ← closed ∪ newInt
14:             else if [i, j] ⊇ [tst_q, ted_q] then
15:                 if edop = 0 then
16:                     offer(h,⟨0, 0, 0,[i_o, j_o],''⟩)
17:                 else
18:                     offer(h,⟨edop, m_q + l, m_a + l, [i_o, j_o], summary+'-'*l⟩)
19:             else if edop < k then
20:                 if m_q + l + 1 = n then
21:                     newInt ← D([lst_a, led_a],[i,j],m_a, l)
22:                     REPORT(newInt, summary+'-'*l+'d')
23:                     closed ← closed ∪ newInt
24:                 else
25:                     offer(h,⟨edop+1, m_q+l,m_a+l+1, D([lst_a,led_a],[i,j],m_a,l+1), summary+'-'*l+'i'⟩)
26:                     offer(h,⟨edop+1, m_q+l+1, m_a+l, D([lst_a,led_a],[i,j], m_a,l), summary+'-'*l+'d'⟩)
        return results
```

Chapter 4

Evaluation

To assess the effectiveness of the algorithms, we apply our approach to artificial and real-life UI logs in a controlled environment and evaluate the quality of the resulting routines. First, we describe the datasets used for the experiments. Besides the artificial and real-life logs, we generated more complex logs to test the algorithms. Moreover, we select adequate metrics and describe the process to adapt and calculate them to evaluate the quality of the discovered routines. Next, we analyze the results obtained based on the provided ground truths. Furthermore, we compare our results against the ones obtained with the Graph Approach.

4.1 Datasets

We rely on 11 UI logs, which can be divided in: nine artificial logs and two real-life logs in a controlled environment. The artificial logs (CPN1-CPN9) are noise-free and segmented. These logs were generated from Coloured Petri Nets (CPNs) [7]. The logs contain a different number of routines of varying complexity. The first six CPNs have a low degree of complexity, with clear routines with a specific goal. CPN1 represents the sequence where the user opens a file, opens a webpage, logs in, awaits the server's response, and copy data from the webpage to the file. CPN2 follows the same sequence, although it includes an error in the login action: the user entering the wrong credentials. CPN3 is an extension of CPN2, where the user performs unsuccessfully the login action multiple times until they cancel the procedure. CPN4 stems from CPN1, injecting non-deterministic actions such as random button clicks and user inputs with different login credentials for each routine trace. For CPN5 and CPN6, the number of non-deterministic actions increases. The last three CPNs have the highest complexity. CPN8 includes a conditional, where a routine is only executed if a condition is met. CPN9 merges CPN5 and CPN6 and includes a conditional based on input data.

The real logs in a controlled environment *Student Records* (SR) and *Reimbursement* (RT) were recorded with the tool Action Logger. The tool stores all context parameters associated with each action. SR simulates copying students' data from a spreadsheet to a Web form, while RT simulates the task of filling reimbursement requests. Both logs contain fifty task traces, where some of the task traces contain noise representing user mistakes.

Table 4.1 shows the characteristics of the logs we use to test our approach. Particularly, we show the number of routine variants as reported by the user.

UI log	# Reported Routines	#Routine Variants	#Taks Traces	#Total UIs
CPN1	1	1	100	1400
CPN2	3	2	1000	14804
CPN3	7	6	1000	14583
CPN4	4	1	100	1400
CPN5	36	12	1000	8775
CPN6	2	2	1000	9998
CPN7	14	7	1500	14950
CPN8	15	8	1500	17582
CPN9	38	18	2000	28358
Student Records (SR)	2	2	50	1539
Reimbursement (RT)	1	1	50	3114

Table 4.1: UI logs

UI log	Length	RT	SR
$IL40_1$	40	22	18
$IL40_2$	40	20	20
$IL50_1$	50	21	29
$IL50_2$	50	25	25
$IL60_1$	60	28	32
$IL60_2$	60	33	27
$IL80_1$	80	36	44
$IL80_2$	80	30	50

Table 4.2: Information of the interleaved UI logs

Besides the 11 logs presented in Table 4.1, we use: eight more logs built from SR and RT obtaining random samples from both logs, simulating when a user works simultaneously on two tasks; and the concatenation of SR and RT labeled $SRRT_+$ and $RTSR_+$, i.e., the user performs one task first and then the other. The information regarding the length and number of instances from SR and RT is shown in Table 4.2.

4.2 Metrics

To answer the question of how we can assess the quality of a candidate routine, we collected all the ground truth routines for each UI log. The ground truth routines for CPNs were extracted from the variants observed in the logs. The number of ground truths for each artificial log is stated in column *Number of routine variants* from Table 4.1. Column *Number of routines reported* is the number of routines per UI log in [29]. For SR and RT, we know the routines performed by the user, and they represent the ground truth to evaluate the approach.

To assess the quality of the discovered routines, we measure their similarity with the ground truth routines. There are different ways to calculate the similarity between the two

routines. We use three different metrics, Jaccard index (JI), Rand index (RI), and Coverage, the last one has two variants. The first two metrics measure the degree of similarity, while the third one measures how much of the log behavior is captured by the routines. Neither JI nor RI penalize the order of the UIs in a routine, following the assumption that a routine may have permutations in the order of the UIs are executed. To introduce the metrics we have to calculate the confusion matrix. Consequently, we describe how we calculate the values for the confusion matrix in the context of UI log.

- *True positive (TP)* values are the UIs shared by the ground truth and the discovered routines.
- *False positives (FP)* represent UIs in the discovered routine but not in the ground truth.
- *True negatives (TN)* are UIs not in the ground truth and not in the routine.
- *False negatives (FN)* are in the ground truth but not in the discovered routines.

Given the set of discovered routines and the set of ground truth routines, we calculate the Levenshtein Edit Distance of each routine with the ground truths, and assign the closest match. With the closest match we calculate the value of the metrics of interest and assign the results of the metrics as a quality score for the routines.

4.2.1 Jaccard index

Jaccard index, which measures the degree of similarity between the discovered routine and the ground truth. The Jaccard index between the routine found by the algorithm and the ground truth is given by the ratio

$$JC = \frac{TP}{TP + FP + FN}$$

In the RPM context, the JI between the closest match and the routine can also be calculated as the number of UIs contained in both routines over the sum of the lengths of the two routines. It has a value between 0 and 1, with 0 indicating that there is no similarity and 1 that the routines are exactly the same.

4.2.2 Rand index

Besides JI, we use Random Index to measure the similarity between ground truths and the discovered routine. We consider this metric, given that JI disregards the elements that are different in both routines (TN values), as we consider the relevance of the TN values in the routines.

$$RI = \frac{TP + TN}{TP + TN + FP + FN}$$

Similarly to the JI, RI result ranges from 0 to 1, 0 indicating no similarity and 1 indicating total correspondence.

4.2.3 Coverage

The pattern coverage considers the percentage of the log that is covered by the pattern occurrence. The goal is to reach a very high coverage with fewer routines. We defined the coverage as the *Strict node coverage* in Definition 3.2.5. The coverage associated with each routine (leaf coverage), we use:

$$Coverage = \frac{|PS(node, l_c)|}{|L|}$$

When we want the total coverage with the routines found by the exact routines,

$$TotalCoverage = \frac{|PS(root_{subtree})|}{|L|}$$

If there are approximate patterns, they are treated as a leaf and their *ps* is added to the exact routine *ps*. Then, we can calculate the *TotalCoverage* or the *coverage per routine* of the exact routines including the approximate *ps*.

4.2.4 Effective Coverage

We defined an additional metric to make the coverage of Graph Approach comparable with the coverage achieved by our method. The *effective coverage* counts the exact occurrences of the reported patterns in the UI log. The effective coverage is calculated per pattern and is defined as the ratio

$$\frac{n * |P|}{|L|}$$

Where n is the number of occurrences given a pattern P.

4.3 Results

In the following sections, we divided the results of the artificial logs and the real-life logs. Initially, we use three evaluation metrics: Jaccard Index, Random Index, and Coverage (node coverage and effective coverage). We present the coverage achieved by our approach in two columns, *Exact Cov* refers to the coverage finding only the exact patterns. *App Cov* shows the coverage after retrieving approximate routines using approximate pattern matching techniques. All the approximate matching tests used the maximum number of edit operations $k = 7$ We implemented our algorithm in C++ and the experiments were executed on a Windows 10 laptop with an Intel Core i7-6700HQ CPU 2.60GHz and 8GB RAM.

4.3.1 Artificial and real-life logs

We compare the results of our algorithms *Exact* and *Approximate* against the Graph Approach in Table 4.3 and Table4.4, which implemented four criterion to select the candidate routines: *Length*, *Frequency*, *Cohesion*, and *Coverage*. Artificial logs have very similar values for the exact and approximate coverage. When the perfect coverage is not achieved through Approximate Matching, the identified subtree does not correspond to the first symbol of the

routine, and the first instance of the routine is lost. For CPN9, we legitimized only six routines with the highest number of instances among the twenty-nine the algorithm finds due to the low coverage of the left-out routines. In the case of CPN5, the 36 routines had a significant number of occurrences in the log.

For the real-life logs, the difference between the achieved coverage of the exact method and the approximate is wide, given that some noise was introduced to the logs. The results for the concatenated logs $SRRT_+$ and $RTSR_+$ are the same.

For the CPN logs, the metrics JI and Coverage metrics were similar. However, as the complexity of the logs increases, we may notice that our approach, with the approximate matching algorithm, has better performance, yet the difference between both approaches is low. The best result from the Graph Approach is achieved by the *Cohesion* criterion, being *Length* almost always as high as *Cohesion*.

UI log	#Selection Criterion	Discovered Routines	Routine Length	Coverage	JI
CPN1	Frequency	1	14.00	1.00	1.00
	Length	1	14.00	1.00	1.00
	Coverage	1	14.00	1.00	1.00
	Cohesion	1	14.00	1.00	1.00
	Exact	1	14.00	0.99	1.00
	Approximate	1	14.00	1.00	1.00
CPN2	Frequency	3	6.33	0.99	1.00
	Length	2	14.50	0.95	1.00
	Coverage	2	14.00	0.99	1.00
	Cohesion	2	14.50	0.95	1.00
	Exact	2	14.50	0.99	1.00
	Approximate	2	14.50	0.99	1.00
CPN3	Frequency	4	5.75	0.95	0.51
	Length	3	14.33	0.95	1.00
	Coverage	3	9.67	0.96	0.83
	Cohesion	3	14.33	0.93	1.00
	Exact	6	17.00	0.99	1.00
	Approximate	6	17.00	1.00	1.00
CPN4	Frequency	1	12.00	0.86	0.86
	Length	2	14.00	1.00	1.00
	Coverage	1	13.00	0.93	0.93
	Cohesion	2	14.00	1.00	1.00
	Exact	2	14.00	0.99	1.00
	Approximate	2	14.00	0.99	1.00
CPN5	Frequency	6	1.67	0.86	0.21
	Length	7	7.29	0.83	0.85
	Coverage	4	3.75	0.80	0.46
	Cohesion	8	7.5	0.86	0.91
	Exact	36	8.833	0.99	0.85
	Approximate	36	8.883	0.99	0.85
CPN6	Frequency	3	4.67	1.00	0.49
	Length	2	10.00	1.00	1.00
	Coverage	3	4.67	1.00	0.49
	Cohesion	2	10.00	1.00	1.00
	Exact	2	10.00	0.99	1.00
	Approximate	2	10.00	1.00	1.00

Table 4.3: Comparing the quality of the discovered routines

UI log	#Selection Criterion	Discovered Routines	Routine Length	Coverage	JI
CPN7	Frequency	7	2.43	0.91	0.26
	Length	7	9.57	0.88	0.99
	Coverage	6	3.67	0.91	0.39
	Cohesion	7	9.43	0.93	0.97
	Exact	14	10.10	0.99	1.00
	Approximate	14	10.10	1.00	1.00
CPN8	Frequency	5	4.20	0.75	0.33
	Length	6	10.67	0.91	0.97
	Coverage	5	7.60	0.89	0.62
	Cohesion	6	10.67	0.91	0.97
	Exact	15	10.67	0.99	1.00
	Approximate	15	10.67	1.00	1.00
CPN9	Frequency	5	5.20	0.82	0.40
	Length	6	14.67	0.95	1.00
	Coverage	5	6.60	0.88	0.51
	Cohesion	6	14.67	0.95	1.00
	Exact	6	14.69	0.97	0.99
	Approximate	6	14.69	0.99	0.99
SR	Frequency	3	10.00	0.96	0.36
	Length	3	28.33	0.98	0.94
	Coverage	2	15.50	0.96	0.53
	Cohesion	3	28.33	0.98	0.94
	Exact	2	30.00	0.78	0.97
	Approximate	2	30.00	0.98	0.97
RT	Frequency	3	18.67	0.90	0.29
	Length	3	56.33	0.96	0.83
	Coverage	2	30.50	0.45	0.45
	Cohesion	3	56.33	0.96	0.83
	Exact	2	62.00	0.75	0.95
	Approximate	2	62.00	0.98	0.95
SRRT$_+$	Frequency	5	16.80	0.90	0.37
	Length	4	45.25	0.91	0.93
	Coverage	2	42.50	0.86	0.92
	Cohesion	4	45.25	0.91	0.93
	Exact	4	46.00	0.76	0.96
	Approximate	4	46.00	0.97	0.96
RTSR$_+$	Frequency	5	16.80	0.90	0.37
	Length	4	45.25	0.91	0.93
	Coverage	2	42.50	0.86	0.92
	Cohesion	4	45.25	0.91	0.93
	Exact	4	46.00	0.76	0.96
	Approximate	4	46.00	0.97	0.96

Table 4.4: Comparing the quality of the discovered routines

UI log	Method	JI	RI	# Routines	Average Length	Cover	Eff Cover	Exec Time (s)
$IL40_1$	Graph	0.8	0.871	4	53.8	0.89	0.51	5.202
	SA_{Exact}	0.834	0.896	8	39.5	0.65		0.46
	SA_{Approx}					0.81		0.467
$IL40_2$	Graph	0.923	0.950	5	45.75	0.88	0.70	4.771
	SA_{Exact}	0.828	0.841	5	68.2	0.592		0.055
	SA_{Approx}					0.592		0
$IL50_1$	Graph	0.863	0.912	5	44.60	0.87	0.65	8.58
	SA_{Exact}	0.748	0.839	5	47.6	0.799		0.075
	SA_{Approx}					0.97		0.31
$IL50_2$	Graph	0.760	0.847	5	60.2	0.89	0.51	5.866
	SA_{Exact}	0.849	0.885	7	58.91	0.78		0.12
	SA_{Approx}					0.92		1.100
$IL60_1$	Graph	0.706	0.88	5	57.20	0.89	0.53	5.692
	SA_{Exact}	0.832	0.850	5	55.80	0.66		0.106
	SA_{Approx}					0.95		0.945
$IL60_2$	Graph	0.807	0.811	6	54.67	0.89	0.84	7.776
	SA_{Exact}	0.798	0.813	4	62.00	0.741		0.178
	SA_{Approx}					0.97		0.641
$IL80_1$	Graph	0.797	0.865	6	57.5	0.84	0.70	5.729
	SA_{Exact}	0.956	0.973	4	49.4	0.72		0.157
	SA_{Approx}					0.94		0.468
$IL80_2$	Graph	0.799	0.880	6	47.00	0.87	0.77	6.828
	SA_{Exact}	0.954	0.978	7	47.09	0.86		0.123
	SA_{Approx}					0.94		1.15

Table 4.5: Results of the interleaved logs

4.3.2 Interleaved logs

We presented the information of the interleaved logs in Table 4.2, where the logs were built by alternating routines from RT and SR through a sampling process. Below we compare the results of the algorithms developed and the Graph Approach. Nevertheless, to avoid cluttering, we compare only the results of our two methods only against their best approach, namely the cohesion-based variant and, given that the values of SA_{Exact} and SA_{Approx} are the same except for the coverage, we only added the value for the column corresponding to coverage. As seen in Table 4.5, our approach has a good performance in terms of coverage, when the UI logs are larger, yet with the logs with 40 instances of SR and RT, the coverage is low, specially in the log $IL40_2$.

We observe a drop in the effective coverage obtained from the Graph Approach. Moreover, as we examine in great detail, we found that there are even cases where a routine has no occurrence observed in the UI log. To elaborate further, we break down the effective coverage of UI log $IL60_1$. Five routines are the result of the Graph Approach. Information regarding

each routine is in Table 4.6. The *Coverage* column shows the coverage reported by the algorithm, while the *Effective Coverage* column coverage of the exact occurrences of the routines. Please notice that only two of these routines have real occurrences in the log. Hence, we may conclude that some of the reported routines are approximate patterns.

Pattern	**Length**	**Coverage**	**Effective Coverage**
P1	61	0.29	0.318
P2	100	0.2	0
P3	31	0.18	0.216
P4	51	0.14	0
P5	40	0.08	0

Table 4.6: Information of the routines found by the Graph Approach

4.4 Validity Threats

After analyzing the reported coverage and the effective coverage, we notice the drop in the value of the second metric for every UI log due to the exact occurrences of the reported routine being low. Some of the routines do not occur in the UI log, being approximate patterns. By contrast, our approach can back up the coverage results with an exact reference of the observed instance in the UI log and characterize the discrepancy of the approximate routines through the exact number of edit operations. The exact approach reports higher levels of actual routines found in the log, and the Graph Approach presents uncertainty in the reliability of the reported data.

At first glance, we could be prompted to dismiss the validity of the Graph Approach's discovered routines. Nevertheless, there are aspects in the evaluation that are not transparent. With the analysis, we cannot gauge the degree of uncertainty in their results. An alternative to this situation search for approximate patterns and quantify the number of edition operations that would legitimize every instance and its respective contribution to the coverage, similarly to the approximate matching technique. Furthermore, we can either determine the level of overlapping in the routines only using the effective coverage metric. To go through this level of detail, we would need to analyze thoroughly the code implemented in their approach.

Since there exists a degree of uncertainty in the Graph Approach's reported results, due to their high-level evaluation, a more detailed evaluation of the routines is needed. We consider that a further examination is required to assess in an objective manner and with more precision the quality of their method, given that the patterns they are reporting, are approximate yet they do not report the edit distance, nor the duplicate symbols or the symbols in a different order. We would need to redefine their high-level vision, which could possibly turn into a biased evaluation.

It is then required to create a deeper evaluation, possibly including the design of new evaluation metrics that allow us to assess the quality of the routines objectively in the RPM context. Consequently, such analysis should not be conducted unilaterally to avoid perception bias, so a collaboration with another university would be desirable. We foresee an opportunity

to work with the authors of the State of the Art algorithm. The detailed analysis in collaboration with another research team is beyond the work of the present Thesis project, and we consider it as future research work.

4.5 Limitations

Our approach can discover interesting routines from UI logs where the user performs one or more tasks. It is capable of discovering more than one routine in the UI log. However, when the routines are interleaved, this is applicable if the routines are executed more than one consecutive time. Otherwise, if two routines are interleaved among each other, performing only one time each routine, our approach likely detects them as one routine composed by the concatenation of both.

The approach identifies multiple variants of the same routine if there is a slight variation in the UIs through approximate matching techniques. Since our approach is designed for logs where the routines instances are consecutive, if the user performs multiple routines simultaneously, i.e., interleaving UIs of one routine and then UIs of the other, the algorithm won't be able to discover significant routines. The situation is similar to when the routines are interleaved.

Furthermore, the algorithm is capable of discovering multiple routines, which has been proven with the concatenated and interleaved logs. However, the algorithm can discover a routine only if it appears in the log at least twice. If a routine has multiple variants, and each variant does not have enough support, the algorithm will not be able to find it.

Finally, the approach can handle multiple variants, nevertheless, in the case of permutations (sections of the routine that can be performed in a different order), if the number of interchanged activities is high (more than the maximum edit operations), the algorithm would not discover the routine and some of the occurrences could be lost. Even increasing the maximum number of edit operations would increase the execution time. Therefore, we would need to adequate the algorithm to handle permutations more efficiently.

Chapter 5

Conclusions and Future Work

This research presented an approach to discover candidate routines from UI logs for automatization in Robotic Process Automation. The project's main contribution is developing an exact and approximate approach to identify candidate routines for RPA. As a first step, our approach extracts exact repetitive routines. The UI log is treated as a sequence of labels, where a label represents a UI. Each label is composed of a combination of relevant parameters, which where selected manually. Consequently, the UI log can be processed with string matching techniques. The algorithm has a greedy point of view based on two heuristics that allow us to select the best local optimal subtree or subtrees (by maximizing its coverage and minimizing Jaccard) and legitimize the routines with coverage higher than a certain threshold. The second part of the approach consists of, once having the best subtree (or subtrees), finding all the approximate patterns within a difference of k symbols. Moreover, we selected a set of metrics, that we consider appropriate to evaluate the routines. The empirical evaluation shows that the algorithm is capable of finding relevant routines and is robust to a certain degree of noise. The approach has been implemented as in C++ and evaluated with real-life and synthetic logs. There are still limitations that we have addressed before and constitute the basis for future work.

5.1 Future Work

In the previous chapter, we have analyzed the results of both approaches. Based on the results, we presented limitations the algorithm has to tackle. We aim to address them as future work. Below are the main tasks to work on to improve our current algorithm:

1. We plan to develop an algorithm to select relevant parameters that characterize the UIs automatically. This step has proven to be crucial for the discovery of candidate routines. Greedy approaches can be explored to propose a plausible solution for this problem, particularly the Gradient Descent algorithm. Other optimization algorithms can be used in this scenario.

2. We aim to complement our approach by improving our algorithm to be able to handle the permutations of the UIs in the routine (actions are performed in a different order). A possible approach is to use more flexible notions of patterns such as in [8] that do not consider the order of the events in an event log.

3. We plan to collaborate with another university (especially with the authors of the Graph Approach) to conduct a more detailed evaluation to assess the quality of the routines and, consequently, the effectiveness of both approaches. Specific evaluation metrics should be developed to this end.

Another task to fulfill is related to the step in the RPM process pipeline, ”Candidate routines identification”, which can be decomposed into two substeps. The first substep refers to the problem already tackled with this approach. However, to address the second substep, ”Identifying routines amenable for automation”, we develop a metric to assess the extent to which a routine is automatable and classify the routines as semi- or fully automatable routines. Some approaches [18] use the frequency of execution of a task to decide whether the task is automatable or not. Nevertheless, this approach is unreliable. Leno et al. [31] propose the use of the concept of determinism. A bot would be able to execute a fully deterministic routine, but a semi deterministic routine can be split into two automatable actions if a non-deterministic action happens in the middle. Hence, there is a need to develop an approach to measure the automatability of candidate routines.

Bibliography

[1] ABOUELHODA, M. I., KURTZ, S., AND OHLEBUSCH, E. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2*, 1 (2004), 53 – 86. The 9th International Symposium on String Processing and Information Retrieval.

[2] ALEXANDER, J., COCKBURN, A., AND LOBB, R. Appmonitor: A tool for recording user actions in unmodified windows applications. *Behavior Research Methods 40*, 2 (May 2008), 413–421.

[3] AMIR, A., APOSTOLICO, A., LANDAU, G. M., LEVY, A., LEWENSTEIN, M., AND PORAT, E. Range lcp. *Journal of Computer and System Sciences 80*, 7 (2014), 1245–1253.

[4] BAYOMIE, D., AWAD, A., AND EZAT, E. Correlating unlabeled events from cyclic business processes execution. In *Advanced Information Systems Engineering* (Cham, 2016), S. Nurcan, P. Soffer, M. Bajec, and J. Eder, Eds., Springer International Publishing, pp. 274–289.

[5] BAYOMIE, D., AWAD, A., AND EZAT, E. Correlating unlabeled events from cyclic business processes execution. In *CAiSE* (2016).

[6] BOSCO, A., AUGUSTO, A., DUMAS, M., LA ROSA, M., AND FORTINO, G. Discovering automatable routines from user interaction logs. In *Business Process Management Forum* (Cham, 2019), T. Hildebrandt, B. F. van Dongen, M. Röglinger, and J. Mendling, Eds., Springer International Publishing, pp. 144–162.

[7] BOSCO, A., AUGUSTO, A., DUMAS, M., ROSA, M. L., AND FORTINO, G. Discovering automatable routines from user interaction logs. In *BPM Forum* (2019).

[8] BOSE, R. P. J. C., AND AALST, W. Abstractions in process mining: A taxonomy of patterns. In *BPM* (2009).

[9] CHACÓN MONTERO, J., JIMENEZ RAMIREZ, A., AND GONZALEZ ENRÍQUEZ, J. Towards a method for automated testing in robotic process automation projects. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)* (May 2019), pp. 42–47.

[10] CHEN, Z., FOWLER, R., AND FU, A. Linear time algorithms for finding maximal forward references. *Proceedings ITCC 2003. International Conference on Information Technology: Coding and Computing* (2003), 160–164.

[11] COOLEY, R., MOBASHER, B., AND SRIVASTAVA, J. Data preparation for mining world wide web browsing patterns. *Journal of Knowledge and Information Systems 1* (04 1999).

[12] COOLEY, R., TAN, P.-N., AND SRIVASTAVA, J. Discovery of interesting usage patterns from web data. vol. 1836.

[13] DRAGUNOV, A., DIETTERICH, T., JOHNSRUDE, K., MCLAUGHLIN, M., LI, L., AND HERLOCKER, J. Tasktracer: A desktop environment to support multi-tasking knowledge workers. pp. 75–82.

[14] ELFEKY, M. G., AREF, W. G., AND ELMAGARMID, A. K. Stagger: Periodicity mining of data streams using expanding sliding windows. In *Sixth International Conference on Data Mining (ICDM'06)* (Dec 2006), pp. 188–199.

[15] FERREIRA, D. R., AND GILLBLAD, D. Discovering process models from unlabelled event logs. In *Business Process Management* (Berlin, Heidelberg, 2009), U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, Eds., Springer Berlin Heidelberg, pp. 143–158.

[16] GAN, V. W., LIN, C.-W., FOURNIER VIGER, P., CHAO, H.-C., AND YU, P. A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data 13* (06 2019), 1–34.

[17] GARTNER. Gartner says worldwide robotic process automation software market grew 63 in 2018, June 2019.

[18] GEYER-KLINGEBERG, J., NAKLADAL, J., BALDAUF, F., AND VEIT, F. Process mining and robotic process automation: A perfect match. In *BPM* (2018).

[19] GHODSI, M. Approximate string matching using backtracking over suffix arrays .

[20] GUSFIELD, D. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News 28*, 4 (1997), 41–60.

[21] HUYNH, T. N., HON, W.-K., LAM, T.-W., AND SUNG, W.-K. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science 352*, 1-3 (2006), 240–249.

[22] IVANČIĆ, L., SUŠA VUGEC, D., AND VUKSIC, V. *Robotic Process Automation: Systematic Literature Review*. 08 2019, pp. 280–295.

[23] JIAWEI HAN, GUOZHU DONG, AND YIWEN YIN. Efficient mining of partial periodic patterns in time series database. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)* (March 1999), pp. 106–115.

[24] KAPUSTA, J., MUNK, M., AND DRLÍK, M. Cut-off time calculation for user session identification by reference length. *2012 6th International Conference on Application of Information and Communication Technologies (AICT)* (2012), 1–6.

[25] KASAI, T., LEE, G., ARIMURA, H., SETSUO, A., AND PARK, K. Linear-time longest-common-prefix computation in suffix arrays and its applications. vol. 2089, pp. 181–192.

[26] KURTZ, S. Reducing the space requirement of suffix trees. *Softw. Pract. Exper. 29*, 13 (Nov. 1999), 1149–1171.

[27] KURTZ, S., CHOUDHURI, J. V., OHLEBUSCH, E., SCHLEIERMACHER, C., STOYE, J., AND GIEGERICH, R. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research 29*, 22 (11 2001), 4633–4642.

[28] LACITY, M., AND WILLCOCKS, L. Robotic process automation at telefónica o2. 21–35.

[29] LENO, V., AUGUSTO, A., DUMAS, M., ROSA, M. L., MAGGI, F., AND POLYVYANYY, A. Identifying candidate routines for robotic process automation from unsegmented ui logs, 2020.

[30] LENO, V., DUMAS, M., MAGGI, F. M., AND ROSA, M. L. Multi-perspective process model discovery for robotic process automation.

[31] LENO, V., POLYVYANYY, A., DUMAS, M., LA ROSA, M., AND MAGGI, F. M. Robotic process mining: Vision and challenges. *Business and Information Systems Engineering* (Mar 2020).

[32] LEOPOLD, H., VAN DER AA, H., AND REIJERS, H. A. Identifying candidate tasks for robotic process automation in textual process descriptions. In *Enterprise, Business-Process and Information Systems Modeling* (Cham, 2018), J. Gulden, I. Reinhartz-Berger, R. Schmidt, S. Guerreiro, W. Guédria, and P. Bera, Eds., Springer International Publishing, pp. 67–81.

[33] LIU, B. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. 01 2007.

[34] MADAKAM, S., HOLMUKHE, R. M., AND JAISWAL, D. K. The Future Digital Work Force: Robotic Process Automation (RPA). *JISTEM - Journal of Information Systems and Technology Management 16* (00 2019).

[35] MOBASHER, B., COOLEY, R., SRIVASTAVA, J., ZHOU, B., CHEUNG, S., DOMENECH, J., MIHARA, K., TERABE, M., HASHIMOTO, K., HEYDARI, M., HELAL, R. A., GHAUTH, K. I., BAYIR, M., TOROSLU, I. H., COSAR, A., AND FIDAN, G. A novel technique for sessions identification in web usage mining preprocessing.

[36] OZDEN, B., RAMASWAMY, S., AND SILBERSCHATZ, A. Cyclic association rules. pp. 412 – 421.

[37] PUGLISI, S., SMYTH, W., AND YUSUFU, M. Fast, practical algorithms for computing all the repeats in a string. *Mathematics in Computer Science 3* (06 2010), 373–389.

[38] RODRIGUEZ, C., ENGEL, R., PISONI, G., DANIEL, F., CASATI, F., AND AIMAR, M. Eventifier: Extracting process execution logs from operational databases.

[39] ROMAO, M., COSTA, J., AND COSTA, C. J. Robotic process automation: A case study in the banking industry. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)* (June 2019), pp. 1–6.

[40] SHARIR, M. A strong-connectivity algorithm and its applications in data flow analysis.

[41] SHARMA, N., AND MAKHIJA, P. Web usage mining:a novel approach for web user session construction. *Global journal of computer science and technology 15* (2015).

[42] SIRISHA, N., SHASHI, M., AND PADMA RAJU, G. Periodic pattern mining-algorithms and applications. *Global Journal of Computer Science and Technology Software and Data Engineering 13* (01 2013).

[43] SLIMANI, T. Sequential mining: Patterns and algorithms analysis. *International Journal of Computer & Electronics Research (IJCER) 2* (01 2013), 639–64.

[44] SPILIOPOULOU, M., MOBASHER, B., BERENDT, B., AND NAKAGAWA, M. A framework for the evaluation of session reconstruction heuristics in web usage analysis. *INFORMS Journal on Computing 15* (04 2003).

[45] TAX, N., SIDOROVA, N., AND VAN DER AALST, W. M. P. Discovering more precise process models from event logs by filtering out chaotic activities. *Journal of Intelligent Information Systems 52*, 1 (Feb 2019), 107–139.

[46] VAN DER AALST, W., AND WEIJTERS, A. Process mining: a research agenda. *Computers in Industry 53*, 3 (2004), 231 – 244. Process / Workflow Mining.

[47] VAN DER AALST, W. M. P., BICHLER, M., AND HEINZL, A. Robotic process automation. *Business and Information Systems Engineering 60*, 4 (Aug 2018), 269–272.

[48] WEINER, P. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)* (1973), IEEE, pp. 1–11.

[49] WRIGHT, D. The robots are ready. are you? untapped advantage in your digital workforce. *Deloitte Consulting Group S.C* (2018).

[50] YANG, J., WANG, W., AND YU, P. Mining asynchronous periodic patterns in time series data. vol. 15, pp. 275–279.

[51] ZHANG, D., LEE, K., AND LEE, I. Periodic pattern mining for spatio-temporal trajectories: A survey. In *2015 10th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)* (Nov 2015), pp. 306–313.

[52] ZHU, Y., LI, S., BAO, N., AND WAN, D. Mining approximate periodic pattern in hydrological time series. In *EGU General Assembly Conference Abstracts* (Apr 2012), EGU General Assembly Conference Abstracts, p. 515.