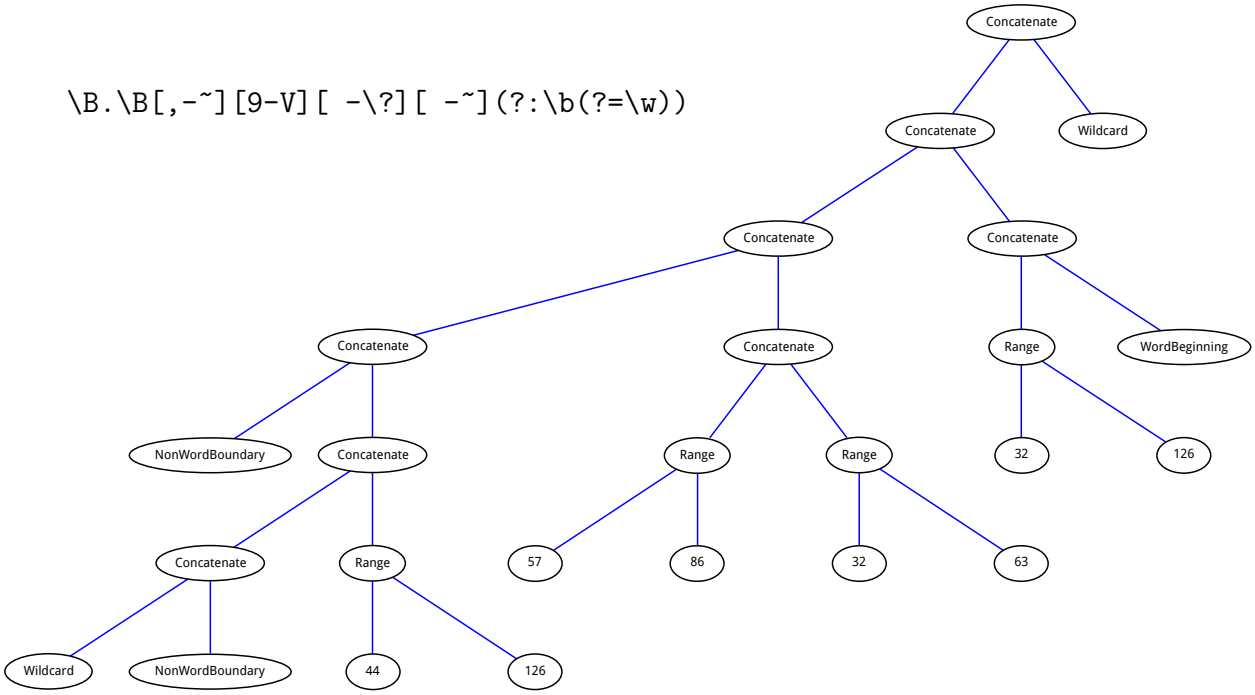


A genetic programming approach to finding discrepancies in log files

Master's thesis in Complex Adaptive Systems

MARTIN GULLIKSSON

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

A genetic programming approach to finding discrepancies in log files

MARTIN GULLIKSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Algebra and Geometry
Chalmers University of Technology
Gothenburg, Sweden 2021

A genetic programming approach to finding discrepancies in log files
MARTIN GULLIKSSON

Supervisor: Elin Helgegren, Ascom
Examiner: Martin Raum, Department of Mathematical Sciences

Master's Thesis 2021
Department of Mathematical Sciences
Division of Algebra and Geometry
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A regular expression represented as a tree structure.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

A genetic programming approach to finding discrepancies in log files
MARTIN GULLIKSSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

An evolutionary algorithm is used to find discrepancies in log files. From a set of error-free reference logs, a set of regular expression patterns describing the logs' general structure is generated using genetic programming. The patterns can then be checked against logs containing errors, with the goal being that added, removed and reordered lines are detected. Using a regex-oriented approach allows for grouping lines together even though the contents are not exactly the same in every instance. The approach works well so long as the log files provided do not contain too much noise.

Keywords: log analysis, evolutionary algorithm, genetic programming, regular expressions, openmpi.

Acknowledgements

This work is dedicated to all my friends and loved ones, who are always rooting for me.

Additionally, I'd like to give special thanks to my examiner Martin Raum and my supervisor Elin Helgegren for their valuable help and encouragement over the course of the project, and to Ascom for ultimately providing me with the opportunity to perform this thesis project.

Martin Gulliksson, Gothenburg, May 2021

Contents

List of Figures

List of Tables

1

Introduction

In software development, a large portion of time is usually spent troubleshooting problems. When a user or tester reports an issue and provides a set of log files, the developers have to analyze these files to find the cause of the issue. Depending on the size of the logs and the nature of the problem, this can be a tedious and time-consuming process. If it is possible to automate parts of this workflow, it will allow the software developers to spend more time with feature development as opposed to maintenance.

This project has been performed together with Ascom, a global company that provides solutions focused on healthcare ICT and mobile workflows.

1.1 Related work

Several tools for automated log analysis have been developed over the years[1][2]. Typical methods for finding discrepancies in log files include *principal component analysis*[3] and *support vector machines*[4]. These types of classifiers generally work well but they can sometimes be difficult to debug due to them being black box models. It is difficult to correlate the input data and output data by simply analysing the models's structure. Similarly, it is difficult to predict how the output of the model changes as the input changes. Explainable types of learning algorithms, however, can be easily understood, predicted and debugged[5].

The method used in this project uses genetic programming to autonomously generate a list of regular expressions (regex) from (error-free) training data which is then used to find discrepancies in error logs. Generating regex patterns from a list of strings has been done in the past[6][7][8], but distinctive for this project is that the data to be matched is not known beforehand. It is up to the algorithm to extract similar strings from the input data and generate regular expressions to match these strings.

1.2 Aim

The aim of the project is to investigate the feasibility of using an evolutionary algorithm approach to find discrepancies in log files. Using log files from normal operation as training data, the algorithm should learn the types, ordering and quantities of log entries the logs usually contain. When a log file containing discrepancies is provided, it should be able to find these inconsistencies and present them to the

developer in a meaningful way. The purpose is to prevent the developer from having to sift through large portions of information which is irrelevant to the problem at hand, which ultimately saves time and effort. Potentially the algorithm could also find discrepancies that the developer would not have found on their own. This kind of algorithm is especially helpful to track down errors which occur seemingly randomly.

1.3 Limitations

The log file analyzer program should (theoretically) be able to analyze any kind of plaintext file, but the the scope for this project is largely limited to analyzing logs from Logcat, the logging system built into the Android operating system. Logcat entries have a predetermined format with the following structure:

```
[date] [time] [process-id] [thread-id] [priority] [tag]: [log message]
```

The log files to be compared are assumed to be of similar nature and duration. For example, the training dataset may consist of 100 log files from successful system startups, while the error logs are from system startups where a certain background service fails to start due to some unknown reason. The goal in this case would be for the program to correlate the service failing to something else happening in another part of the system, likely the explanation for the crash.

Log files containing errors are assumed to be clearly labelled as such, and kept separate from the error-free reference data logs.

Log files do not have to be analyzed in realtime. That being said, the computational performance should still be sufficient enough to not be a hindrance.

1.4 Ethical considerations

The purpose of the log file analyzer described in this report is to quickly and effortlessly find the root cause of observed issues. The purpose is *not* to map out the behaviour patterns of users or infringe on their privacy. The program described in this report is released as free, open source software. Since such software can be used by anyone for any purpose, it may be used in ways which is in conflict with the sentiment described here.

Training data for the project has been gathered from in-house at Ascom. This means that privacy from end users is preserved.

1.5 Thesis outline

This report is divided into theory, methods, results and conclusions. Section 2 discusses the theory behind regular expressions and genetic programming. Section 3 explains how to combine these two concepts into a functional log file analyzer. Then, in Section 4, its performance is evaluated on three different datasets, created from

three different bug reports. Finally, Section 5 summarizes which parts work well, which parts do not, and possibilities for the future.

2

Theory

This section describes the theory behind genetic programming and regular expressions, which are the building blocks needed for implementing the log file analyzer.

2.1 Genetic programming

Genetic programming (GP) is a type of evolutionary algorithm (EA), an optimization algorithm which mimics evolution in nature[9]. The algorithm creates a random set (*population*) of *individuals* and assigns a *fitness score* to each one depending on how well it satisfies certain criteria. New individuals are created based on the best ones in previous set (*selection* and *reproduction*), and this process is repeated many times (*generations*) in order to maximize the fitness score. The training is typically stopped after a specified number of generations, after a specified amount of time or when the maximum fitness score hasn't increased for a certain number of generations.

Genetic programming is an evolutionary algorithm which specifically focuses on evolving computer programs. Individuals in genetic programming are typically represented as tree structures consisting of operators and values. Each individual is given a fitness score depending on how well the corresponding computer program solves the task at hand.

New individuals are created using *crossover* and *mutation*. Crossover is performed by swapping two random subtrees between two individuals. Mutation is performed by modifying an individual in some way, for example replacing/adding/removing a random subtree or replacing a random operator/input value.

2.2 Regular expressions

A regular expression (regex) pattern is a character sequence used to extract data from a stream of text, commonly used in conjunction with the UNIX command line utilities `grep` and `sed`. Regular expressions can be written to extract e-mail addresses, hyperlinks or any other kind of content[10].

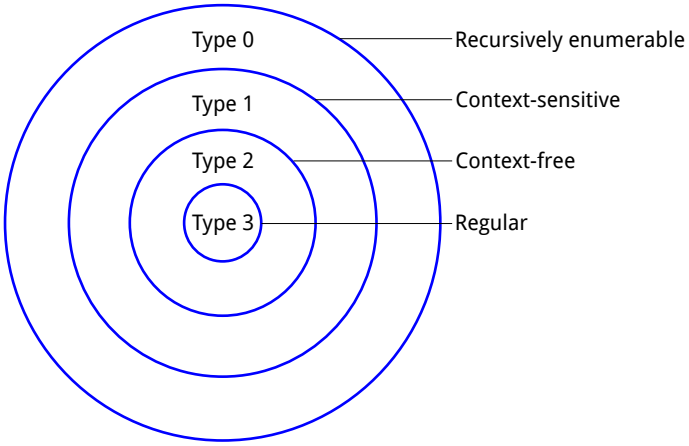


Figure 2.1: The Chomsky hierarchy of formal grammars. All regular languages are context-free, all context-free languages are context-sensitive and all context-sensitive languages are recursively enumerable. The language of regular expressions is an example of a regular language.

2.2.1 Theoretical background

The language of regular expressions is made up of a type 3 grammar according to the Chomsky hierarchy, see Figure 2.1. This means that a regular expression can be implemented using finite state automation[11][12][13]. Type 2 grammars (e.g. XML) can be implemented using pushdown automation, a type of automation which uses stack memory. Type 1 grammars (e.g. most programming languages) can be implemented using linear bounded automaton, a type of automation which uses a finite amount of arbitrary memory. Type 0 grammars can be implemented using a Turing machine, a type of automation which has an infinite amount of arbitrary memory[14].

Using less expressive languages like regular expressions, despite not being as flexible, has its advantages. The language is faster and easier parse. Additionally, creating well-formed programs using stochastic, automated means (e.g. an evolutionary algorithm) is more feasible if the grammar is more restrictive.

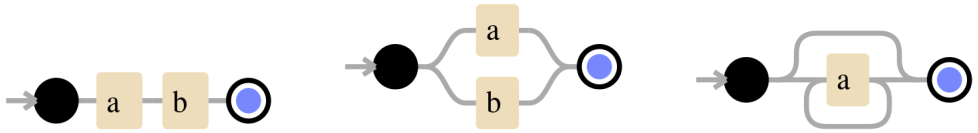
A regular expression is created using one of the following constants:

- $\varnothing$: empty set
- ε: empty string
- $a, b, ...$: single literal character

and new regular expressions can be created from existing ones R, S using the operations:

- RS: concatenation of R and S
- $R|S$: alternation (match either R or S)
- $R*$: match zero or more of R (Kleene star)

The operations are visualized in FSM form in Figure 2.2 and Figure 2.3. These



(a) Concatenation **(b)** Alternation **(c)** Kleene star

Figure 2.2: Visualization of the three primitive regex operations, `ab`, `a|b` and `a*`, as finite state machines. In each case, the regex engine starts out in the state represented by the black disk on the left. The regex engine then loops through the input string and changes state based on the current character in the string. If the state machine stops at the state represented by the encircled blue disk on the right, the regex pattern successfully matches the input string.

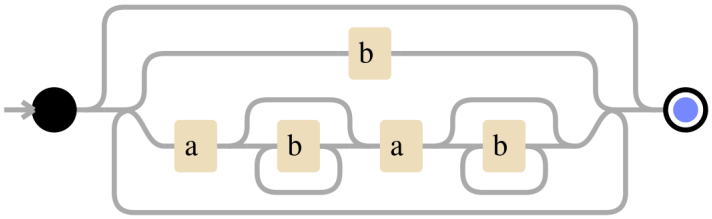


Figure 2.3: Visualization of a more complex regex pattern, `(b|ab*ab*)*`. This pattern matches input strings containing any number of a's and b's so long as the number of a's is an even number $(0, 2, 4, \dots)$. If the input string is empty, the state machine immediately reaches the target state by using the topmost connection. If the input string contains a single b, the state machine reaches the target state by using the connection below the topmost one. If there are multiple b's, the state machine can return to the starting state using the connection at the bottom. The connection in the middle containing the two a's matches two a's in the input string at a time. If the input string contains an odd number of a's, there is eventually only a single one left to match. This causes the state machine gets stuck in the middle of the connection since it's forced to match an a, but there is none.

figures were generated using the regex debugger and visualizer Debuggex[1].

All additional regular expression syntax (shown in Section 2.2.2) can be derived using the three elementary operations shown in this section. As examples, a regex range `[a-f]` can be written as `(a|b|c|d|e|f)` and an optional character `a?` can be written as `(a|ε)`.

2.2.2 Regular expression syntax

A regex pattern may consist of *atoms*, *quantifiers*, *anchors* and *lookarounds*. Atoms specify *what* is matched and quantifiers specify *how many times* each atom should match. An anchor matches a position in the input string and lookarounds assert whether a certain sub-pattern matches the input string at a given position.

List of atoms:

[1] `https://www.debuggex.com`

- `a` matches 'a' (*character literal*)
- `(?:abc|def|ghi)` matches one string of either 'abc', 'def' or 'ghi' (*alternation*)[2]
- `[abc]` matches one character of either 'a', 'b' or 'c' (*character set*)
- `[^abc]` matches one character that is *not* 'a', 'b' or 'c' (*negated character set*)
- `[A-f]` matches one character in the ASCII range from 'A' to 'f' (*character range*)
- `[^A-f]` matches one character which is *not* in the ASCII range from 'A' to 'f' (*negated character range*)
- `.` (dot) matches any character (*wildcard*)
- `\d` matches `[0-9]` (*digit character*)
- `\D` is the inverse of `\d` (*non-digit character*)
- `\s` matches space, tab, etc. (*whitespace character*)
- `\S` is the inverse of `\s` (*non-whitespace character*)
- `\w` matches `[a-zA-Z0-9_]` (*word character*)
- `\W` is the inverse of `\w` (*non-word character*)

List of quantifiers:

- `a?` 'a' is optional
- `a+` 'a' occurs one or more times
- `a*` 'a' can occur any number of times (including zero)
- `a{4}` 'a' occurs exactly 4 times in a row (same as 'aaaa')
- `a{4,8}` 'a' occurs between 4 and 8 times in a row

By default, quantifiers are *greedy*, i.e. they try to match as many characters as possible. Quantifiers can be made *lazy* by appending a question mark (`?`) which makes them match as few characters as possible.

List of anchors:

- `^` matches the beginning of the line
- `$` matches the end of the line
- `\b` matches the beginning or end of a word (*word boundary*)
- `\B` is the inverse of `\b` (*non-word boundary*)

List of lookarounds:

[2] In regular expression syntax, parenthesis `(...)` can be used to specify operator precedence. Parenthesis also signifies a *capture group*, i.e. the string matched by it is stored in a register for later use (e.g. backreferences). If only the operator precedence functionality of the parenthesis is needed, it is better to instead use a *non-capturing group* `(?:...)` for increased computational efficiency.

- `a(?=<regex>)` asserts that 'a' is followed by `<regex>`, but `<regex>` is not consumed (*positive lookahead*)
- `a(?!<regex>)` asserts that 'a' is not followed by `<regex>`, `<regex>` is not consumed (*negative lookahead*)
- `(?<=<regex>)a` asserts that 'a' is preceded by `<regex>`, but `<regex>` is not consumed (*positive lookbehind*)
- `(?<!<regex>)a` asserts that 'a' is not preceded by `<regex>`, `<regex>` is not consumed (*negative lookbehind*)

Not all of the regex syntax mentioned in this section is used in the project, but is included for here for the sake of completeness.

3

Methods

This section describes the implementation of the log file analyzer, which is named *Regea*. Its source is available at GitHub[1]. The program consists of two separate Python scripts: `regea.py` for generating regular expression patterns to match the training data and `regea_diff.py` for finding added, removed and reordered lines using the generated patterns. This report describes the implementation of Regea as of version 1.0.0.

Regea uses the Python module `DEAP`[15][2] for implementing genetic programming, a combination of the python module `regex`[3] and the stand-alone application `ripgrep`[4] (PCRE2[5] back-end) for evaluating regex patterns, and `OpenMPI`[6] for parallelization.

3.1 Input data and preprocessing

Regea expects two sets of log files as its input: error-free training data logs and one or more log files which are known to contain discrepancies. The training logs (or *reference logs*) are used to learn the types and quantities of usually occurring log entries and their ordering. Each log file in the training data should be a record of the same test or event, and there should be exactly one test per file. As an example, the input files may be organized according to:

- Error files (contain discrepancies)
  - xx63.log
  - xx96.log
- Reference files (error-free training data)
  - xx01.log
  - xx02.log
  - ...

[1] `https://github.com/gullikx/regea`
[2] `https://github.com/DEAP/deap`
[3] `https://bitbucket.org/mrabarnett/mrab-regex`
[4] `https://github.com/BurntSushi/ripgrep`
[5] `https://www.pcre.org`
[6] `https://www.open-mpi.org`

- xx94.log
- xx95.log
- xx97.log
- xx98.log

Since this project aims to analyze logs from Andorid logcat, the structure of the log messages is known. To make it easier to compare and classify the log entries, dates, times, and process id:s are stripped from the data. For example, the log entry:

```
09-22 23:55:08.142 723 723 I PackageManager: Time to scan packages: 0.515 seconds
```

is simplified to:

```
I PackageManager: Time to scan packages: 0.515 seconds
```

using string manipulation prior to beginning the training.

3.2 Generating regular expression patterns

This section describes the implementation of the Python script `regea.py` in the Regea source code.

Regea learns the structure of the training data by generating regex patterns for each line in the data.

For each log entry, for example:

```
I PackageManager: Time to scan packages: 0.515 seconds
```

the program attempts to create a corresponding regex pattern, for example:

```
^I PackageManager: Time to scan packages: [0-9]+\.[0-9]+ seconds$
```

This way, it is possible to cross-check how often and at what positions this log entry occurs across the training data, even though the contents are not exactly the same in every instance. Log entries typically consist of a constant string with a few variable parts inside of it, e.g. time stamps or memory addresses. A visual overview of the method is shown in Figure 3.1.

3.2.1 Training loop

A visual overview of the training loop is shown in Figure 3.2. The training data logs are concatenated together and then iterated. A list of regular expression patterns is kept during training. For each line, Regea checks whether any of the regex patterns in the pattern list match the line. If at least one matching regex is found, the line is assumed to have been seen before and the loop continues to look at the next line. If no matching regex is found, a new regular expression is created using genetic programming as described in Section 3.2.2. After having iterated though all of the training data logs, all log entries in the training data are matched by at least one regex in the regular expression list. The list containing regular expressions is then exported and can be checked against a log file containing errors.

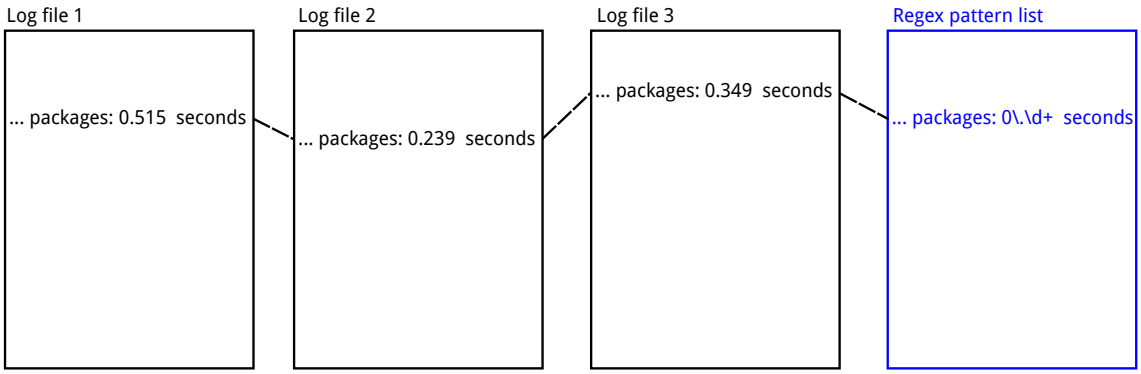


Figure 3.1: Visual overview of the method. The idea is to find similar log entries across a set of log files, and create a regular expression pattern which matches these entries. The list of regular expressions can then be used to detect discrepancies (added, removed or reordered lines) in a log file containing errors.

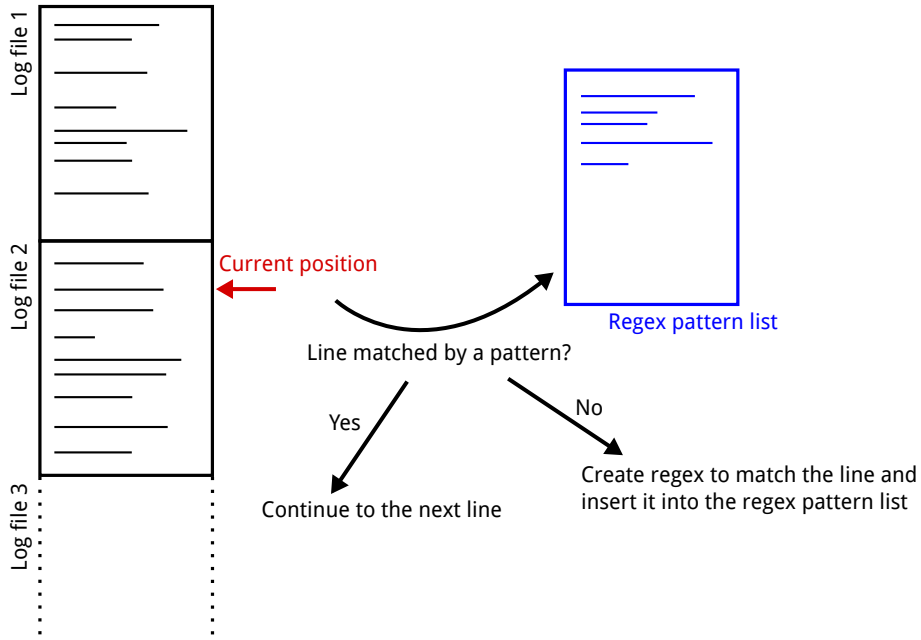


Figure 3.2: Visual overview of the training loop. The log files are concatenated together and iterated sequentially. During training, regular expressions for all previously unmatched log entries are generated using genetic programming. The end result is a list of regular expressions which describes the contents of the training data.

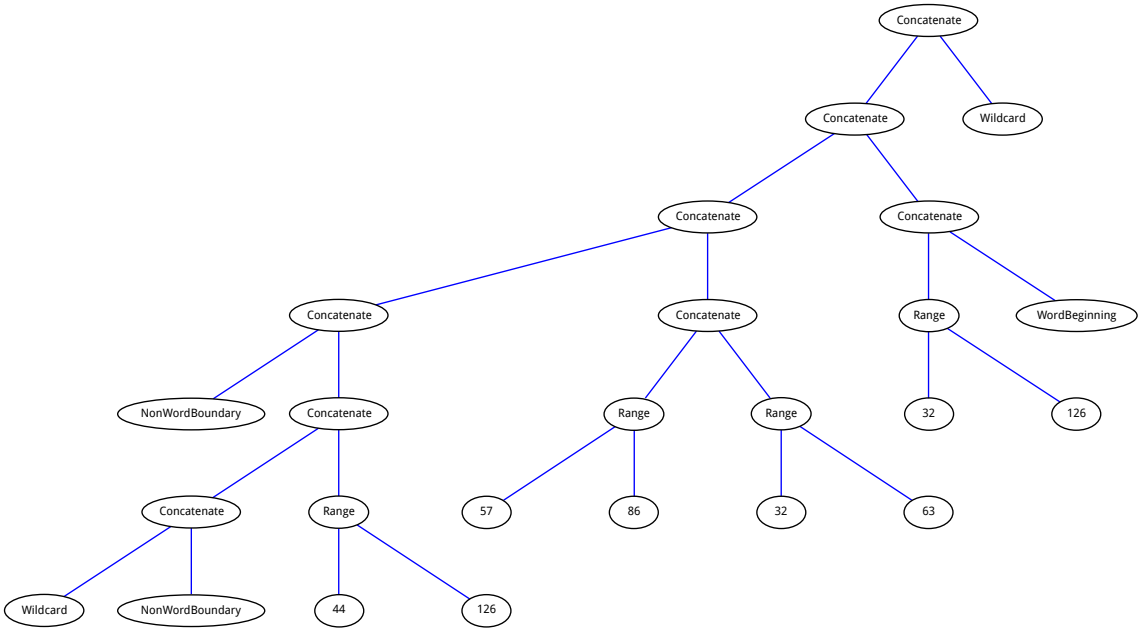


Figure 3.3: Tree representation of an example regex pattern, `\B.\B[,-~][9-V][ -\?][ -~](?:\b(?=\w))`. The tree is traversed in depth-first order in order to construct the regex pattern.

3.2.2 Genetic programming implementation

Regex patterns are generated using genetic programming, a type of evolutionary algorithm which is used to evolve computer programs. Individuals in genetic programming are usually represented as tree structures, consisting of different types of nodes. An example of a regex pattern represented as a tree structure is shown in Figure 3.3.

The genetic programming algorithm is run for each (previously unmatched) entry in the logs, which means that the number of generated regex patterns will ideally be about the same as the number of distinct types of log messages.

During the evolution of a regex pattern, all individuals are padded with wildcard characters and anchored with '`^`' and '`$`' so that they match whole lines. For example, if the evaluation of the tree structure for an individual gives the regex pattern:

```
^example\s\d$
```

it may be padded to

```
^.......example\s\d..................$
```

The number of wildcard characters to add is chosen so that the regex matches the target string (the log line at the 'current position' in Figure 3.2). The motivation for this is that the regex should match log lines which are similar to the target string, which have a similar length. However, requiring exactly the same length is a bit too strict (a number can have a variable number of digits, for example). Instead, the number of wildcard is specified as a range:

```
^.{5,11}example\s\d.{16,22}$
```

The size of these ranges can be set as an input argument to Regea.

3.2.2.1 Node types

Tree structures in genetic programming are made up of different types of nodes. A node can either be a terminal, which takes no inputs, or an operator which takes a set of other nodes as inputs. The terminal node types can be further divided into constants and ephemerals (Python DEAP terminology). A constant terminal type only has a single possible value, while an ephemeral can have one of many different possible values. The terminal node types implemented in Regea are shown in Table 3.1 and the operator node types implemented in Regea are shown in Table 3.2. Each node has specified input and output data types. The genetic programming implementation takes this into consideration when generating new individuals.

Some of the operators mentioned in Section 2.2.2 are not present in Table 3.1 or Table 3.2 and are also not implemented in the code of Regea. The Kleene star (`*`) and plus (`+`) are not implemented because they can match a variable number of characters, making it difficult to assign a fitness value. They are also not easily mutated into other structures. Negative lookarounds are not implemented as it is too easy for the algorithm to illicitly gain large amounts of fitness using them. Positive lookbehinds are not implemented due to performance issues. Variable-size lookbehinds are also not implemented by PCRE2.

Table 3.1: Available terminal node types for creating individuals in the genetic programming algorithm. Some terminal types can have one of many different values (ephemeral), while other terminal types can only have one possible value (constant).

Node type	Possible value(s)	Output type
Ascii code	`(32, 33, ..., 126)`	ascii code
Boolean	`(True, False)`	boolean
Specific character	`( , !, ..., 0...z, ..., ~)`	regex
Empty string		regex
Wildcard	`.`	regex
Word boundary	`\b`	regex
Non-word boundary	`\B`	regex
Word beginning	`(?:\b(?=\w))`	regex
Word end	`(?:(?<=\w)\b)`	regex
Digit	`\d`	regex
Non-digit	`\D`	regex
Word character	`\w`	regex
Non-word character	`\W`	regex
Whitespace character	`\s`	regex
Non-whitespace character	`\S`	regex

Table 3.2: Available operator node types for creating individuals in the genetic programming algorithm. The numbers next to the input types specifies that there are multiple inputs of that type.

Node type	Input type(s)	Output type
Identity	(any)	(same as input)
Concatenate	regex (2)	regex
Optional	regex	regex
Range	ascii code (2)	regex
Negated range	ascii code (2)	regex
Set	boolean (95)	regex
Negated set	boolean (95)	regex
Positive lookahead	regex	regex

3.2.2.2 Generating the initial population

The initial population is generated using wildcard characters and concatenation operators. The amount of wildcard characters depend on the size of the input string. To aid with later mutation, some wildcard characters are chosen randomly to be replaced with filled ASCII ranges, `[ -~]`.

By creating the initial population this way, one gets large, non-specific individuals which the genetic programming algorithm can make more specific over time. If the initial population was created randomly, any valid individuals would be very small (only one or two characters) since the probability of a long, randomly generated regex matching a certain string is very small.

3.2.2.3 Fitness evaluation

The choice of fitness function is very important for generating regex patterns of reasonable generality. If the patterns are too specific, e.g.

```
^I PackageManager: Time to scan packages: 0\.515 seconds$
```

instances of the same log entry will be classified as entirely different entries (overfitting). If the patterns are too broad, e.g.

```
^. P.........g............[a-x]... p.....................$
```

different types of log entries will be classified as the same type of entry (underfitting).

The fitness function for a regular expression in Regea has the form:

$$f = f_{\mathrm{m}} f_{\mathrm{c}} \tag{3.1}$$

where f is the fitness. The component f_{m} is hereafter called the *matching fitness* and the component f_{c} is hereafter called the *complexity fitness*.

The matching fitness $f_{\mathrm{m}} \in \{0, 1\}$ varies according to the number of matches in the

reference files for the regex pattern. It is calculated according to:

$$f_{\mathrm{m}} = \frac{1}{N} \sum_{i=1}^{N} \begin{cases} 1/n_i & n_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

where n_i is the number of times the regex matches in reference file i, and N is the total number of reference files. If there is exactly one match in each of the N files in the reference data, the matching fitness has a perfect value of 1:

$$f_{\mathrm{m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1} = \frac{N \cdot 1}{N} = 1$$

If any reference files have more than one match or none at all, the matching fitness f_{m} will be lower. The motivation for this type of fitness function is that it encourages regex patterns which are specific enough to match as few times as possible in the reference files, but not so specific that some reference files are excluded.

The matching fitness does not work well on its own, since it has (at least) two issues. The first issue is that the fitness value has a strict maximum. If the matching fitness is 1, the regex pattern cannot get anymore specific since the fitness cannot get any higher. The second issue is that small advancements in specificity are not rewarded. If a regex range `[a-t]` is condensed to `[b-g]` it has gotten more specific, but if the number of matches across the reference data is the same the fitness will also be the same. To counteract these issues, the complexity fitness is introduced.

The complexity fitness is calculated by looking at the contents, the atoms and operators, of the regex patterns. Remember that the regex patterns are represented as tree structures. The tree is iterated and each node in the tree has a fitness associated with it. The fitness assigned to each node type is shown in Table 3.3. The total complexity fitness f_{c} for a regex pattern is the sum of the contributions from all nodes in the tree.

Table 3.3: Contribution to the complexity fitness f_c in (3.1) for the different types of regex nodes listed in Table 3.1 and Table 3.2. Node types above the horizontal line are operators while node types below the horizontal line are terminals. The fitness contributions are tuned such that more specific nodes provide more fitness. Some operators, like the lookarounds, do not provide any fitness. The reasoning is that the nodes connected to the lookarounds (the input arguments) provide the fitness instead. The fitness for optional nodes is negative as to prevent the algorithm from gaining fitness from inserting large amounts of optional expressions (though this can still happen in some instances).

Node type	Fitness contribution
Identity	0
Concatenate	0
Optional	-1
Range	1/(number of allowed characters)
Negated range	1/(number of allowed characters)
Set	1/(number of allowed characters)
Negated set	1/(number of allowed characters)
Positive lookahead	0
Ascii code	0
Boolean	0
Specific character	1
Wildcard	1/(number of printable characters) $= \frac{1}{100} = 0.01$
Word boundary	0.5
Non-word boundary	0.5
Word beginning	1
Word end	1
Digit character	1/(number of digit characters) $= \frac{1}{10} = 0.1$
Non-digit character	1/(number of non-digit characters) $= \frac{1}{90} \approx 0.0111$
Word character	1/(number of word characters) $= \frac{1}{63} \approx 0.0159$
Non-word character	1/(number of non-word characters) $= \frac{1}{37} \approx 0.0270$
Whitespace	1/(number of whitespace characters) $= \frac{1}{6} \approx 0.166$
Non-whitespace	1/(number of non-whitespace characters) $= \frac{1}{94} \approx 0.0106$

The reason that the complexity fitness f_c and the matching fitness f_m are multiplied together in the fitness function (3.1) is to make the fitness zero if there are no matches in any of the reference files (because f_m would be zero).

Additionally, there are a few situations where the fitness value of a regex pattern is immediately evaluated to zero. These include:

- The regex does not match the target string (the log line at the 'current position' in Figure 3.2).
- The regex contains two or more anchors in a row, e.g. `\b\b` and `(?:\b(?=\w))(?:\b(?=\w))`.
- Padding the regex pattern with wildcard characters fails to produce a pattern which matches the target string (unlikely, but can happen for patterns which

contain certain combinations of operators).

This list would also include regex patterns which fail to compile due to syntax errors, but care has been taken so that all regex pattern generated by Regea are well-formed.

3.2.2.4 Generating new individuals

Individuals are selected for reproduction using tournament selection and new individuals are created using one-point crossover and several types of mutation. The types of mutation are as follows, each with its own adjustable probability:

- Uniform – replace a random subtree with a new, randomly generated subtree
- Node replacement – replace a terminal with another one of the same output type
- Regenerate ephemeral – randomize the value of one of the ephemeral terminals in the tree (e.g. change a specific character to another specific character)
- Regenerate ephemeral all – randomize the value of all ephemeral terminals in the tree
- Insert – insert a randomly generated subtree
- Shrink – remove a random subtree

This creates a new set of individuals to make up the next generation of the evolution, and the fitness evaluation and selection process are repeated.

The evolution process for a regex pattern is time-limited, i.e. it is stopped after a specified number of seconds (or minutes). This is in contrast to running the evolution for a specified number of generations, or until a specified fitness target is reached. The motivation for the current method is that it prevents inefficient-to-evaluate regex patterns from severely slowing down the program. This issue has been largely mitigated in later versions of Regea, but the time-limited evolution cutoff has still been kept.

3.2.3 Implementation details

Pseudocode:

```
def generatePattern(line, fileContents):
    population = initializePopulation()

    timeStart = time.time()
    while time.time() - timeStart < timeLimit:
        population.evolve()
    return getBestIndividual(population)

fileContents = [None] * nInputFiles
for iFile in range(nInputFiles):
    fileContents[iFile] = inputFiles[iFile].read().splitlines()

regexPatterns = []
regexPatterns.extend(getIdenticalLinesAcrossAllFiles(fileContents))
```

```
fileContentsConcatenated = []
for iFile in range(nInputFiles):
    fileContentsConcatenated.extend(fileContents[iFile])

for line in fileContentsConcatenated:
    if not anyMatch(regexPatterns, line):
        patternNew = generatePattern(line, fileContents)  # performed in parallel
        regexPatterns.append(patternNew)

outputFile.write(regexPatterns)
```

Regea keeps two lists during training, see Figure 3.2. One list contains the log entries from all log files concatenated together. The other list contains the as-of-yet generated regex patterns. The program iterates through the log entry list and checks if any of the patterns in the regex list matches the current line. If not, a new pattern is created and added to the regex list. The genetic programming logic is implemented using the Python module `DEAP`.

This project uses a combination of the Python module `regex` and the stand-alone application `ripgrep` with the PCRE2 back-end for implementing regular expression operations. `ripgrep` is very efficient but calling an external application has extra overhead. Using `python-regex` is therefore faster for smaller data sizes while `ripgrep` is faster for larger data sizes.

Lines which are identical across all log files are added directly to the regex list when starting the program. This saves time since fewer patterns need to be generated by the evolutionary algorithm. Finding identical lines is done efficiently by sorting the file contents alphabetically.

Generating regex patterns is parallelized using OpenMPI. The nodes (processes/threads) are organized in a master-worker configuration with a single node iterating through the concatenated log entries list while the worker nodes are generating regex patterns. If the master node finds a log entry which isn't matched by any pattern, it sends it to an idle worker node and then moves on to the next entry. When a worker node is finished generating a pattern, it sends the pattern back to the master node and receives a new log line in return.

Since generating regex patterns is performed in parallel, two or more worker nodes may simultaneously be creating regex patterns for very similar or identical lines. This can waste a bit of time but it does not affect the final result negatively.

When the program has iterated through the entire list with log entries, it writes the regex list to a file called `regea.output.patterns`. This list of regex patterns is used in Section 3.3 and Section 3.4 for finding discrepancies.

3.2.4 Time complexity analysis

This section evaluates the time complexity of generating regular expressions to match the training data.

Notation:

- N – number of log files in the training data
- l – number of lines per file
- c – number of characters per line
- R – number of regex patterns generated during the training
- M – total number of individuals across all generations during the evolution of a single regex pattern

Assume that the training data consists of N log files, each containing l lines consisting of c characters. The total number of log lines across the entire training set is therefore Nl. This concatenated list of log lines is iterated and for each of the lines, Regea checks if any of the previously generated regex patterns match the line. If any of the previously generated regex patterns match, Regea continues to the next line. If no regex pattern match, a new regex pattern is generated to match the line before continuing. Assume that R number of regex patterns are created during training, each with a size and evaluation time proportional to the line length c. The time complexity to check whether each log line is matched by any pattern is therefore $\mathcal{O}(NlRc)$.

The genetic programming algorithm is run R times, since a total of R regex patterns are created. Assume that the total number of individuals across all generations is M. To calculate the matching fitness f_{m} for each individual, the number of matches for all files in the training data is evaluated. This requires Nl evaluations for each individual resulting in a total of NlM evaluations and an $\mathcal{O}(NlMc)$ time complexity. To calculate the complexity fitness f_{c} for an individual, the individual has to be iterated. The size of each individual is derived from the length of the log line, c. The time complexity for calculating the complexity fitness is therefore $\mathcal{O}(Mc)$.

The combined time complexity is:

$$T(N, l, c, R, M) = \mathcal{O}(NlRc + R(NlMc + Mc)) = \mathcal{O}(RcNlM)$$

The best case time complexity occurs when all l log lines in all N log files are completely identical. In this case, a single regex pattern is created to match the first line in the first file and then all subsequent lines in the training data will be matched by that one pattern. No more patterns are created, leading to $R = 1$ and the time complexity

$$T(N, l, c, M) = \mathcal{O}(1 \cdot lcNM) = \mathcal{O}(lcNM) \tag{3.2}$$

A scenario which is probably more realistic is that the N log files have similar contents, but the log lines within each file are distinct from one another. When iterating through the first log file, new regex patterns are generated for most lines. When iterating through the subsequent files, most of the lines are already matched by at least one pattern (since the contents are very similar to the previous files). This means that the number of generated regex patterns R after iterating through all log files is close to the log length l, regardless of the number of files N. The time complexity becomes

$$T(N, l, c, M) = \mathcal{O}(l \cdot cNlM) = \mathcal{O}(l^2cNM) \tag{3.3}$$

resulting in a time complexity with an additional factor l relative to (3.2).

The worst case time complexity occurs when the contents of the N log files are completely unique. This would mean that the regex patterns generated by iterating through the first log file does not match any of the log lines in the subsequent files. This means that a new regex pattern has to be generated for every line in every file, leading to $R \approx Nl$. The time complexity becomes

$$T(N, l, c, M) = \mathcal{O}(Nl \cdot cNlM) = \mathcal{O}(l^2 cN^2 M)$$

resulting in a time complexity with an additional factor N relative to (3.3).

3.3 Finding reordered log entries

This section, together with Section 3.4, describes the implementation of the Python script `regea_diff.py` in the Regea source code.

The ordering of the log entries are compared by converting each log file into an array of integers, by substituting each entry with the index/indices of the regex pattern(s) matching it. As an example, the log entries

```
I PackageManager: Time to scan packages: 0.515 seconds
I PackageManager: Time to scan packages: 0.318 seconds
I WifiManager: Disconnected from wifi
I PackageManager: Time to scan packages: 0.826 seconds
```

may produce an array $[10, 10, 8, 10]$. This indicates that the lines 1, 2 and 4 are matched by the same regex pattern whereas line 3 is matched by a different pattern. If a line is matched by multiple patterns, all pattern indices are inserted into the array, for example $[10, 10, [2, 8, 45], 10]$.

Regea now generates ordering rules for how these integers relate to each other. A rule can be one of the following types:

- "Pattern A always matches *before all* matches for pattern B"
- "Pattern A always matches *after all* matches for pattern B"
- "Pattern A always matches *before some* match for pattern B"
- "Pattern A always matches *after some* match for pattern B"
- "Pattern A always matches *directly before* pattern B"
- "Pattern A always matches *directly after* pattern B"

Rules are created for how each pattern relates to every other pattern according to the above rule types. The rules are checked against the training set for their validity. The validity for a rule is calculated by evaluating how large proportion of the training data logs satisfy the rule. All rules which have a validity above a certain threshold (default: 0.90) are added to a set.

After training, the generated valid rules are evaluated for an error log. A colored heatmap of the error log is generated and outputted in HTML-format, which can be opened using any web browser. The file highlights each log line according to the

number of violated ordering rules for that line. Lines which have been dislocated further distances are more likely to have a more intense highlighting, as they have been dislocated relative to many other log lines. Also note that different log lines have a different number of ordering rules associated with them. The more the training data agrees with where a line should be positioned, the more valid ordering rules can be created for that line. A line with more ordering rules associated with it will quickly get an intense highlighting if dislocated, since there are a lot of ordering rules to potentially be violated.

3.3.1 Implementation details

Pseudocode[7]:

```
# Generate ordering rules
rulesValid = set()
for pattern in regexPatterns:
    for patternOther in regexPatterns:
        if pattern == patternOther:
            continue
        for ruleType in ruleTypes:
            rule = Rule(pattern, patternOther, ruleType)
            ruleValidity = 1.0
            for referenceFile in referenceFiles:
                if not rule.evaluateForFile(referenceFile):
                    ruleValidity -= 1.0 / len(referenceFiles)
                    if ruleValidity < threshold:
                        break
            else:
                rulesValid.add(rule)

# Highlight log lines with deviations
orderingHeatmap = zeros(len(errorFile))
for line in errorFile:
    for rule in getRulesForLine(rulesValid, line):
        if not rule.evaluateForLine(errorFile, line):
            orderingHeatmap[line] += 1
```

The list of generated regex patterns is iterated in a double for-loop, and rules are created which relates every pattern to every other pattern. The validity of each rule is calculated by evaluating it for the training data. OpenMPI is used to create and evaluate ordering rules in parallel. Creating and evaluating ordering rules is very computationally efficient since the rules are evaluated using integer comparisons, rather than performing any regex operations.

3.3.2 Time complexity analysis

This section evaluates the time complexity of creating and evaluating ordering rules.

Notation:

- N – number of log files in the training data

[7]The `for...else` syntax means that the code inside the `else` clause is executed if and only if the for-loop is ran to its completion, i.e. it is not exited using a `break` statement.

- l – number of lines per file
- c – number of characters per line
- R – number of regex patterns generated during training
- m – number of lines per file matched by each regex pattern

Maps are generated which describe which lines are matched by which patterns at which positions (and vice versa) according to Section 3.3. After this, no further regex operations need to be performed it is possible to perform $\mathcal{O}(1)$ (amortized) lookup of this information. To create these maps, every log line in the training data must be checked against every generated regex pattern, giving the time complexity

$$T(N, l, R, c) = \mathcal{O}(NlRc)$$

This assumes that the time required to evaluate a regex pattern is proportional to c.

Rules are created which describe the position of the matches for every pattern relative to every other pattern in a log file. Thus, the total number of possible ordering rules is R^2 (the fact that there are multiple types of rules is left out here for simplicity, but it would add a constant factor to the expression).

All of the possible ordering rules are evaluated for their validity across the training data logs. Each ordering rule encompasses two regex patterns, and these two patterns can match any number of lines m within the file. The positions of all the m lines matched by the first pattern must be compared to the positions of all of the m lines matched by the second pattern. The total number of positions to compare for all R^2 ordering rules in a single file is therefore R^2m^2. Assume that $l = Rm$. If $R \approx 1$, then $m \approx l$. If $R \approx l$, then $m \approx 1$. The number of comparisons R^2m^2 is therefore approximately l^2 for both $R \approx 1$ and $R \approx l$. The time complexity of evaluating all possible ordering rules for N files is therefore

$$T(N, l, R, m) = \mathcal{O}(R^2m^2N) = \mathcal{O}(l^2N) = \mathcal{O}(Nl^2)$$

3.4 Finding added/removed log entries

This section, together with Section 3.3, describes the implementation of the Python script `regea_diff.py` in the Regea source code.

With the regex patterns generated, the mean and standard deviation of the number of matches across the training set are calculated, see Table 3.4.

Table 3.4: The number of matches in the training data logs and a corresponding error log for the generated regex patterns. In this example, pattern 2 points to a discrepancy since the number of matches in the error log deviates significantly from the training data logs.

Regex pattern	Matches in training logs (mean, std. dev.)	Matches in error log
<pattern 1>	$\mu = 1$, $\sigma = 0$	1
<pattern 2>	$\mu = 5$, $\sigma = 2$	30
<pattern 3>	$\mu = 50$, $\sigma = 5$	47
⋮	⋮	⋮

If the number of matches in the error log for a certain pattern deviates from the training data (pattern 2 in Table 3.4), it is assumed to be a discrepancy. Entries in the error log which are not matched by any pattern are immediately assumed to be discrepancies (since no similar log lines appear in the training data).

Regea creates an output file similar to a `diff`-file, but in HTML-format. The output file shows lines which have been added in green and lines which have been removed in red. The intensity of the color depends on the certainty (the multiple of standard deviations from the mean). Green lines are simply highlighted, but red lines need to be imported from the reference data (since they do not exist in the error log).

If the number of matches for a regex pattern in the error log is less than one standard deviation below the mean of the training data, a random line matched by the pattern is imported from the training data and inserted into the error log. All lines matched by a regex pattern should be very similar, which means that the exact line to be inserted is not of great importance. Finding the most optimal position to insert the line into the error log is done by evaluating the ordering rules used for finding reordered lines (see Section 3.3). Regea evaluates the number of violated ordering rules for each possible insertion position in the error log. The line is then finally inserted into the position where it violates the least number of rules. If there is a tie, the line is inserted into the first of the tied positions (lowest index). A side effect of this is that lines to be inserted which do not have any valid ordering rules applicable to them will always be inserted at the top of the file (since Regea has no information about where they should be put).

3.4.1 Implementation details

Pseudocode:

```
# Calculate matching frequencies for all regex patterns
freqMeans = {}
freqStddev = {}
errorFreq = {}

for pattern in regexPatterns:
    nMatchesPerReferenceFile = countMatches(pattern, referenceFiles)
    freqMeans[pattern] = mean(nMatchesPerReferenceFile)
    freqStddevs[pattern] = stddev(nMatchesPerReferenceFile)
    errorFreq[pattern] = countMatches(pattern, errorFile)
```

```
# Highlight added lines in green
diffHeatmap = zeros(len(errorFile))
for line in errorFile:
    patternsMatchingLine = getPatternsMatchingLine(line, patterns)
    if len(patternMatchingLine) == 0:
        diffHeatmap[line] = <large value>  # line unique to error log
    else:
        for pattern in patternsMatchingLine:
            if errorFreq[pattern] > freqMeans[pattern]:
                diffHeatmap[line] += countStddevs(freqMeans[pattern],
                        freqStddevs[pattern], errorFreq[pattern])
        diffHeatmap[line] /= len(patternsMatchingLine)

# Import and insert missing lines (highlighted in red)
for pattern in regexPatterns:
    if errorFreq[pattern] < freqMeans[pattern]:
        if (countStddevs(freqMeans[pattern],
                    freqStddevs[pattern], errorFreq[pattern]) > threshold):
            line = getRandomLineToInsert(pattern, referenceFiles)

            violationsPerPosition = getRuleViolationsPerPosition(line, errorFile)
            insertPositionBest = argmin(violationsPerPosition)
            errorFile.insert(insertPositionBest, line)
            diffHeatmap[line] = <large negative value>
```

The mean and standard deviation for the number of matches for each pattern is evaluated. These values are then compared to the number of matches in the error log for each pattern. Lines which are matched by patterns occurring more often than the mean are highlighted in highlighted in green. The strength of the color depends on the multiple of standard deviations from the mean. For each pattern, a line is imported from the reference data and inserted if it occurs too few times relative to the training data. The line is inserted at the position where it violates the least number of ordering rules and is highlighted in red. OpenMPI is used to calculate pattern match frequencies and insert lines in parallel.

3.4.2 Time complexity analysis

This section evaluates the time complexity of performing statistical analysis of the regex patterns, highlighting added lines in an error log and importing removed lines from the training data.

Notation:

- N – number of log files in the training data
- l – number of lines per file
- c – number of characters per line
- R – number of regex patterns generated during training

The same lookup maps as in Section 3.3.2 are used, which are created with time complexity $\mathcal{O}(NlRc)$.

The mean and standard deviation of the number of matches for each pattern across the training data is evaluated. Looking up the number of matches for a single pattern

in a single file is performed with time complexity $\mathcal{O}(1)$ amortized. Evaluating the number of matches for all regex patterns for all training data files therefore has the time complexity $\mathcal{O}(NR)$. Evaluating the number of matches for each of the R patterns in a single error file is done in the same way except that there is only a single file, leading to the time complexity $\mathcal{O}(R)$. Comparing the number of matches for each pattern against the mean and standard deviations of the training data has the time complexity $\mathcal{O}(R)$. Additionally, each line in the error log is checked whether it has at least one regex pattern matching it. This has time complexity $\mathcal{O}(1)$ for each line, and therefore $\mathcal{O}(l)$ for all l lines. The total time complexity for finding added/removed lines becomes:

$$T(N, R, l) = \mathcal{O}(NR + R + R + l) = \mathcal{O}(NR + l)$$

Following the reasoning in Section 3.2.4, the best case time complexity ($R \approx 1$) is

$$T(N, l) = \mathcal{O}(N + l)$$

and the worst case time complexity ($R \approx Nl$) is

$$T(N, l) = \mathcal{O}(N^2 l^2)$$

Log entries which are deemed by Regea to be missing in the error log are imported from the training data and inserted into the error log. Assuming that a large portion of the R^2 total possible ordering rules are valid, evaluating the valid rules for an error log has time complexity $\mathcal{O}(R^2 m^2) = \mathcal{O}(l^2)$ (see Section 3.3.2). Checking all l locations in the error log for the best position to insert a line has the time complexity $\mathcal{O}(l^3)$. Inserting l log lines has the time complexity:

$$T(l, R, m) = \mathcal{O}(R^2 m^2 l^2) = \mathcal{O}(l^4)$$

4

Results

Regea has been tested on a variety of datasets. This section shows the performance on three different sets, designated A, B and C. Datasets A and B were generated using an automated testing system, while dataset C was generated by human operators. Dataset A contains logs from a bug which caused missing audio during phone calls, dataset B contains contains logs from a bug which caused missing notifications, and dataset C contains logs from a bug which caused sub-par phone call audio quality when Bluetooth was active. The properties of the three datasets are shown in Table 4.1.

Table 4.1: Properties of the three datasets used to quantify the performance of the log file analyzer. Here, N denotes the number of training log files in the dataset, l denotes the number of log lines per file, l_d denotes the number of duplicate lines per file (lines which are found at least once in all logs in the dataset), and c denotes the number of characters per log line. The values are written using their mean μ and their standard deviation σ across the dataset. Dataset C was generated by human operators (rather than automated testing) which explains the larger standard deviation σ for the log length l.

Dataset	N	l	l_d	c
A	96	$(\mu \approx 1687, \sigma \approx 283)$	$(\mu \approx 782, \sigma \approx 21)$	$(\mu \approx 102, \sigma \approx 69)$
B	88	$(\mu \approx 3538, \sigma \approx 140)$	$(\mu \approx 1588, \sigma \approx 50)$	$(\mu \approx 112, \sigma \approx 70)$
C	26	$(\mu \approx 1749, \sigma \approx 1144)$	$(\mu \approx 715, \sigma \approx 95)$	$(\mu \approx 92, \sigma \approx 55)$

4.1 Finding discrepancies

The results from training on dataset A are shown in Listing 4.1, Listing 4.2, Figure 4.1 and Figure 4.2. Listing 4.1 shows a heatmap of an error log which highlights reordered lines while Listing 4.2 shows a heatmap of an error log which highlights added/removed lines, Figure 4.1 and Figure 4.2 show the heatmaps as histograms. These figures also contain the histogram for an error-free reference log, to show that it does not show as large discrepancies as a log containing errors. Similarly, the results from training on dataset B are shown in Listing 4.3, Listing 4.4, Figure 4.3 and Figure 4.4. The results from training on dataset C are shown in Listing 4.5, Listing 4.6, Figure 4.5 and Figure 4.6. A comparison of the results across all datasets is shown in Table 4.2 and Table 4.3.

```
D hwcomposer: (0:2) Layer+ (mva=0x0/sec=0/prot=0/alpha=1:0xff/blend=0002/dim=0/fmt
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
D AudioPolicyServiceCustomImpl: AudioCommandThread() processing set volume stream
V Ascom Phone (VoIP): Call cleanup
V Ascom Phone (VoIP): Audio cleanup
I SurfaceFlinger: [Built-in Screen (type:0)] fps:9.176484,dur:1198.72,max:212.05,m
D PhoneSip: sip_call::change_media_state() [0x8E9] await_answer -> answer_received
D PhoneSip: VOIP_CHAN.0 -> AC_CH.0 : CHANNEL_MEDIA_SENDRECV
D PhoneSip: VOIP_CHAN.0 -> AC_CH.0 : CHANNEL_MEDIA_CONFIG (0x10,172.20.17.2:25506,
D PhoneSip: sip_call::change_state() [0x8E9] Alerting -> Connected
D PhoneSip: VOIP_CHAN.0 -> AC_CH.0 : CHANNEL_MEDIA_CONNECTED
D PhoneSip: sip_call::change_media_state() [0x8E9] answer_received -> idle
D PhoneSip: VOIP_CHAN.0 -> AC_CH.0 : CHANNEL_MEDIA_SENDRECV
D PhoneSip: VOIP_CHAN.0 -> AC_CH.0 : CHANNEL_MEDIA_CONFIG (0x10,172.20.17.2:25506,
```

Listing 4.1: An excerpt of a heatmap describing the differences in ordering between an error file and an average reference file in dataset A. Lines which are more yellow violate more ordering rules. For the bug searched for in this case, the issue is that the function "`V Ascom Phone (VoIP): Audio cleanup`" is being run at the wrong point in time, which Regea successfully highlights. In this case many lines are highlighted, since the bug causes a lot of secondary effects in the ordering of the log lines. Lines which are longer than what fits on the page are shown as truncated.

```
V Ascom Phone (VoIP): enterWifiInCallMode - mocked out
I Telecom : InCallController: Calling onAudioStateChanged, audioState: [AudioState
V Ascom Phone (VoIP): createStartInCallActivityIntent: open extra action = 0.
D PhoneSip: VoipCall.2280 -> AC_CH.0 : PH-TONE-ON
D PhoneSip: AC_CH.0[-]: PhTone on 2 0 23 0x0 500 1000/3000 0/0 0/0 0/0
D PhoneSip: AC-DSP0: update_dsp(1) 00 0 0 0
V AudioPolicyIntefaceImpl: getOutput()
D AudioTrack: InitializeMTKLogLevel: default level[2]
D AudioTrack: set(): 0x7bd8b2dc00, streamType -1, sampleRate 0, format 0x1, channe
D AudioTrack: Building AudioTrack with attributes: usage = 3, content = 4, flags =
D AudioALSAStreamManager: +closeInputStream(), in = 0xecbcb800, size() = 1
D AudioTrack: set: 0x7bd8b2dc00, Create AudioTrackThread, tid = 3321
D AudioDetectPulse: setDetectPulse, mIsDetectPulse 0, 0x7c61fad0e8
V Ascom Phone (VoIP): onCallAudioStateChanged 0: [AudioState isMuted: false, route
V AudioPolicyIntefaceImpl: getOutputForAttr()
D AudioPolicyManagerCustomImpl: getOutputForAttr() device 0x4, sample_rate 0, form
D AudioFlinger_Threads: Client defaulted notificationFrames to 256 for frameCount
D AudioFlinger: track(0xed990d00): mIsOut 1, mFrameCount 960, mSampleRate 48000, m
D AudioTrackShared: InitializeMTKLogLevel: default level[2]
D AudioFlinger: track(0xed990d00): mFastIndex 3, mStreamType 8, mName -1
```

Listing 4.2: An excerpt of a heatmap describing the difference between an error file and an average reference file in dataset A. Lines deemed by Regea to have been added in the error log relative to the reference data are highlighted in green, with the strength of the color depending on the certainty. Lines which are deemed to be missing from the error log are imported from the reference data and inserted at reasonable positions. These are shown in red. In this case, the error log shows discrepancies related to the Android audio system, caused by the reordered log line mentioned in Listing 4.1.

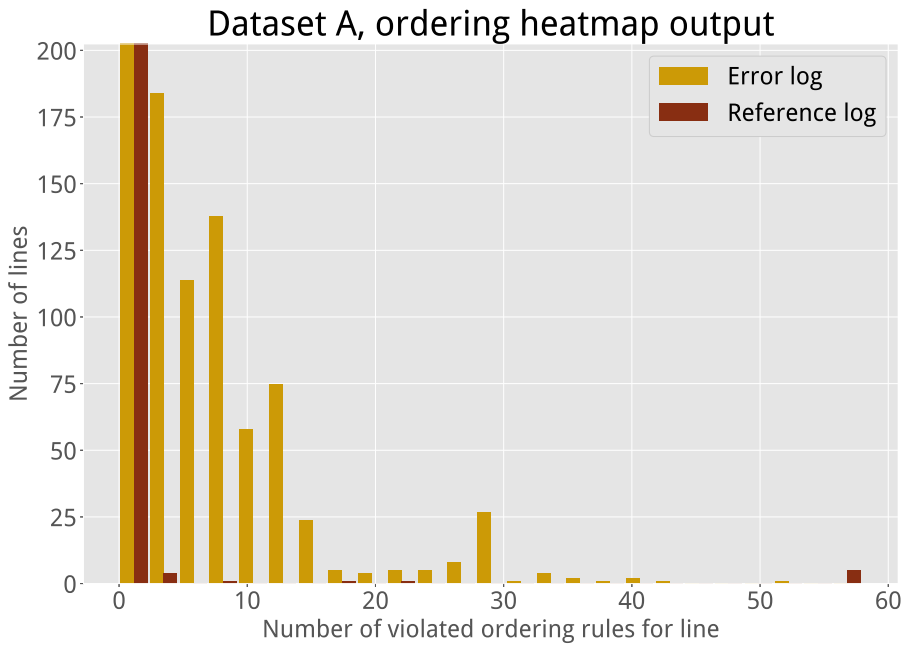


Figure 4.1: The heatmap in Listing 4.1 shown as a histogram, both for an error log and for an error-free reference log. The histogram bins at $x = 0$ have been truncated for readability. Their actual values are 698 for the error log and 1452 for the reference log. The value for the reference log at $x = 60$ is caused by an Android background service which was performing tasks while this log was being recorded.

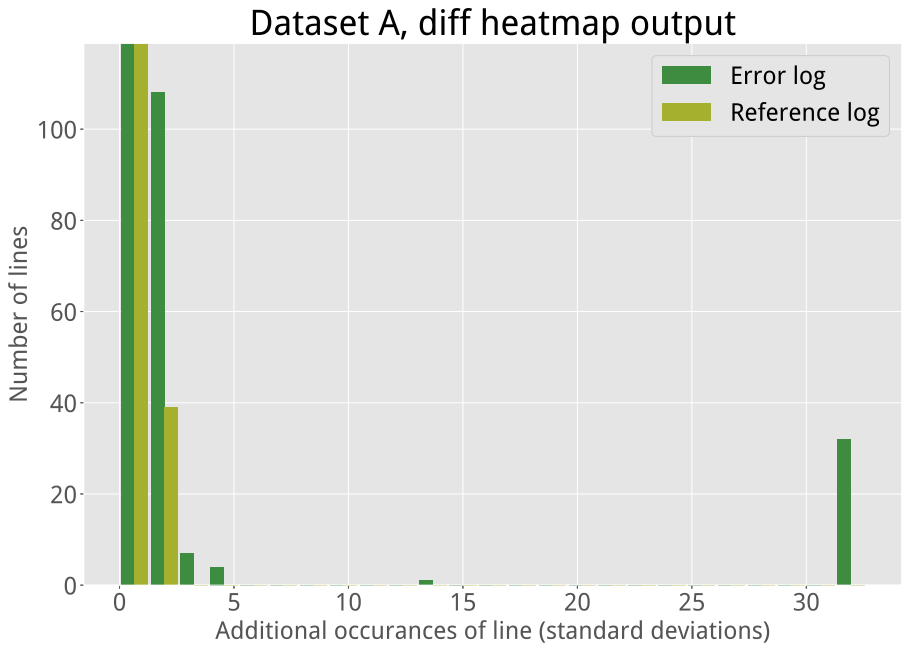


Figure 4.2: The heatmap in Listing 4.2 shown as a histogram for an error log and for an error-free reference log. Note that the histogram only includes lines which Regea perceives to be added (shown in green in Listing 4.2). Removed lines (shown in red in Listing 4.2) are not included since they are not part of the actual error log. The actual (non-truncated) values for the histogram bins at $x = 0$ are $(1205, 1425)$.

```
E AudioSystem-JNI: Command failed for android_media_AudioSystem_setStreamVolumeInd
D APM_AudioPolicyManager: setVolumeCurveIndex: wrong index 0 min=1 max=15
I A       : Refresh account policy.
E APM_AudioPolicyManager: setVolumeIndexForAttributes failed to set curve index fo
E AudioSystem-JNI: Command failed for android_media_AudioSystem_setStreamVolumeInd
I Telecom-NewOutgoingCallBroadcastIntentReceiver: Received new-outgoing-call-broad
D APM_AudioPolicyManager: setVolumeCurveIndex: wrong index 0 min=1 max=15
E APM_AudioPolicyManager: setVolumeIndexForAttributes failed to set curve index fo
E AudioSystem-JNI: Command failed for android_media_AudioSystem_setStreamVolumeInd
I hwservicemanager: getTransport: Cannot find entry vendor.mediatek.hardware.pplag
W SearchServiceCore: Abort, client detached.
I Telecom-NewOutgoingCallIntentBroadcaster: Placing call immediately instead of wa
I Telecom-CallsManager: Creating a new outgoing call with handle: tel:****: NOCBIR
I Telecom-CallsManager: isSpeakerphoneEnabledForTablet: NOCBIR.oR@rzE
I Dialer  : InCallActivity.getShouldShowVideoUi - null call
I Dialer  : InCallActivity.getShouldShowRttUi - null call
I Dialer  : InCallFragment.onCreateView
I SetupWizard: [aru] [logReason:Get Notification Priority, UserDismissed: false, P
I Telecom-PhoneAccountRegistrar: getSimCallManager: SimCallManager for subId -1 qu
I Telecom-CreateConnectionProcessor: Trying attempt CallAttemptRecord(ComponentInf
```

Listing 4.3: An excerpt of a heatmap describing the differences in ordering between an error file and an average reference file in dataset B. For this dataset, Regea only found minor discrepancies in the ordering of the log entries.

```
I Dialer  : InCallFragment.setCallState - PrimaryCallState, state: 13, connectionL
I Dialer  : VideoPauseController.onPrimaryCallChanged - new call: [DialerCall_42,
D AudioTrack: ~AudioTrack(3498): 0x75b2b57800
I BufferQueueConsumer: [Toast#0](this:0x7823f49000,id:7506,api:0,p:-1,c:478) setDe
E AxessManagementService: at com.ascom.services.OkHttpClientProvider$InternetConne
I Dialer  : InCallPresenter.setBoundAndWaitingForOutgoingCall - setBoundAndWaiting
I Dialer  : ProximitySensor.updateProximitySensorMode - screenOnImmediately: true,
I Dialer  : ReturnToCallController.hide - hide() called without calling show()
I Dialer  : InCallFragment.onButtonGridCreated - InCallUiReady
I Dialer  : InCallFragment.setAudioState - audioState: [AudioState isMuted: false,
I MediaFocusControl: requestAudioFocus() from uid/pid 10156/1741 clientId=android.
D AlertIndicatorService: set value:000000000000000
I BufferQueue: [unnamed-478-7850](this:0x7824076800,id:7850,api:0,p:-1,c:-1) Buffe
I BufferQueueConsumer: [unnamed-478-7850](this:0x7824076800,id:7850,api:0,p:-1,c:4
I BufferQueueConsumer: [unnamed-478-7850](this:0x7824076800,id:7850,api:0,p:-1,c:4
I BufferQueueConsumer: [com.android.dialer/com.android.incallui.InCallActivity#0](
I BufferQueueConsumer: [com.android.dialer/com.android.incallui.InCallActivity#0](
D Surface : Surface::allocateBuffers(this=0x7595267000)
W ActivityManager: Slow operation: 60ms so far, now at startProcess: building log
I BufferQueueProducer: [StatusBar#0](this:0x7823fa9800,id:2,api:1,p:1050,c:478) qu
```

Listing 4.4: An excerpt of a heatmap describing the difference between an error file and an average reference file in dataset B. There are a few lines missing related to notifications and the audio system, which is very plausible considering the bug is related to missing notifications. A few lines can also be seen being faintly highlighted in green.

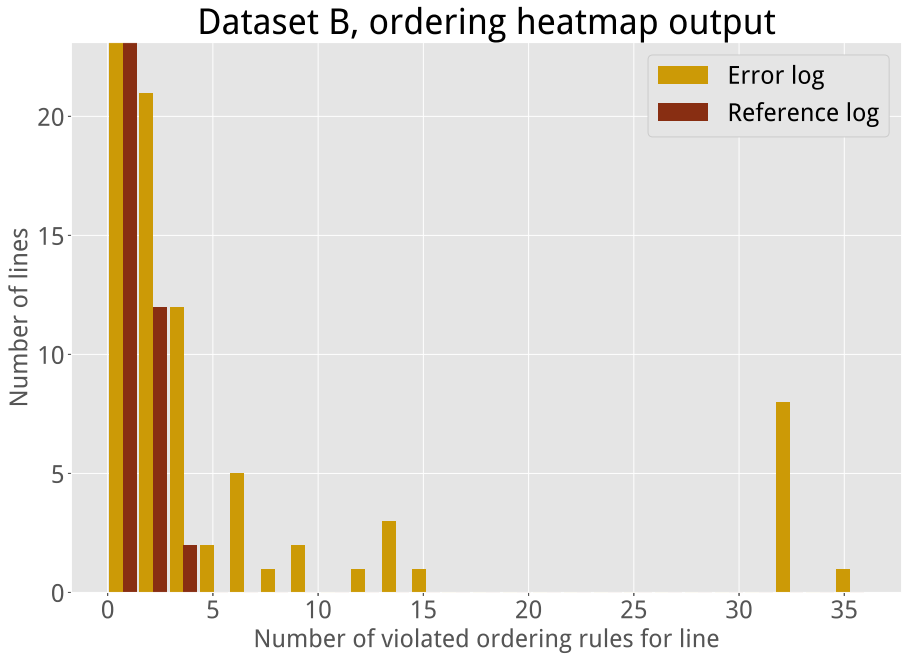


Figure 4.3: The number of violated ordering rules per line for dataset B shown as a histogram. The actual (non-truncated) values for the histogram bins at $x = 0$ are $(3521, 3650)$.

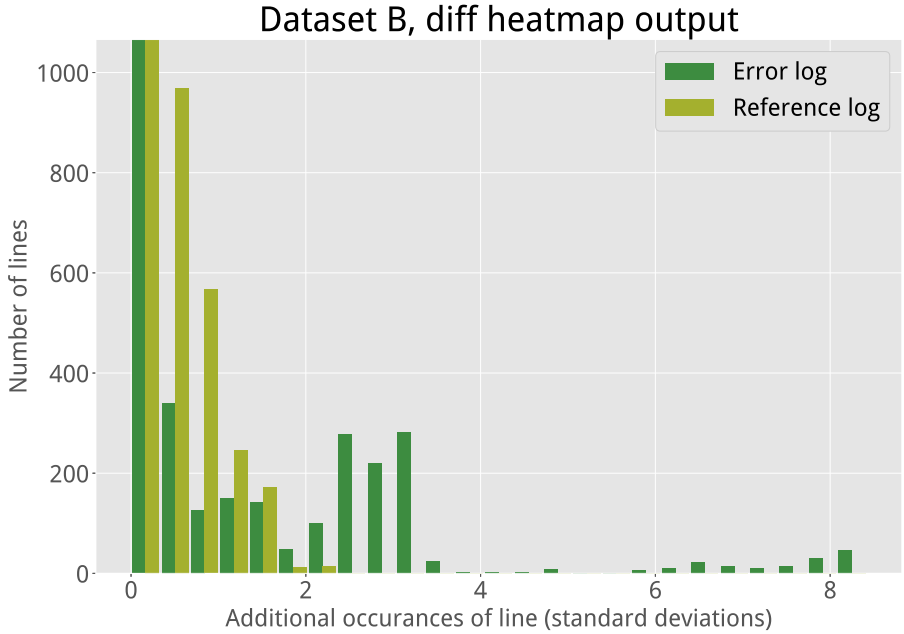


Figure 4.4: The number of extra occurrences for each log line for dataset B shown as a histogram. The actual (non-truncated) values for the histogram bins at $x = 0$ are $(1691, 1680)$.

```
I AudioPolicyManagerCustomImpl: -computeGainTableCustomVolume volume 1.000000 stre
D AudioPolicyClientImpl: invalidateStream: type 10
D APM_AudioPolicyManager: stopOutput portId 20
D APM_AudioPolicyManager: stopOutput() output 13, stream 1, session 25
I AudioPolicyManagerCustomImpl: -computeGainTableCustomVolume volume 1.000000 stre
D View : [Warning] assignParent to null: this = DecorView@836e3f9[dialer]
D KeyguardUpdateMonitor: handlePhoneStateChanged() - mPhoneState = 0
D Surface : Surface::disconnect(this=0x7c89c5a000,api=2)
I BufferQueueProducer: [Splash Screen com.android.dialer#0](this:0x7d6752b800,id:2
I InputTransport: Destroy ARC handle: 0x7cfc20b620
I BufferQueueProducer: [FramebufferSurface](this:0x7d4f9ef000,id:0,api:1,p:495,c:4
I GED : ged_boost_gpu_freq, level 100, eOrigin 2, final_idx 1, oppidx_max 1, oppid
I Ascom Phone (VoIP): Call: 7000
V Ascom Phone (VoIP): enterWifiInCallMode - mocked out
D PhoneSip: VoipCall.34 -> AC_CH.0 : PH-TONE-ON
D PhoneSip: AC_CH.0[-]: PhTone on 2 0 23 0x0 500 1000/3000 0/0 0/0 0/0
D PhoneSip: AC-DSP0: update_dsp(1) 00 0 0 0
V Ascom Phone (VoIP): Outgoing call 0x1022 to 7000. Has properties=true.
V Ascom Phone (VoIP): onCallAudioStateChanged 0: [AudioState isMuted: false, route
I AscomEventLog: SIP: Outgoing call (0x1022)
```

Listing 4.5: An excerpt of a heatmap describing the differences in ordering between an error file and an average reference file in dataset *C*. This dataset did not contain any major ordering discrepancies.

```
D AudioALSAStreamIn: getCapturePosition(), timestamp not change, update time 25100
I Proximity: distance = 2
I hwcomposer: [HWCDisplay] [Display_0 (type:1)] fps:34.822975,dur:1033.80,max:149.
D PhoneSip: AC_CH.0[0]: RTP packet loss 36 0
V Ascom Phone (VoIP): exitWifiActiveMode - mocked out
I Dialer : ProximitySensor.updateProximitySensorMode - screenOnImmediately: false,
D AudioALSAStreamIn: getCapturePosition(), timestamp not change, update time 25104
D AudioALSAStreamIn: getCapturePosition(), timestamp not change, update time 25105
I BufferQueueProducer: [com.android.dialer/com.android.incallui.InCallActivity#0](
D AudioALSAStreamIn: getCapturePosition(), timestamp not change, update time 25106
I Proximity: distance = 1
D lights : write_int open fd=7
D PQ : ccorr table index=0
D PQ : [PQ_SERVICE] setPQParamWithFilter configFlag: 1
D AAL : onBacklightChanged: 409/1023 -> 0/1023(phy:0/4095)
D AAL : onBacklightChanged: change screen state 3(On) -> 0(Off)
D AALLightSensor: AALLightSensor setEnabled 0-->0
D AAL : 02-26 06:08:24.680 BL= 0,ESS= 256,
I BufferQueueProducer: [StatusBar#0](this:0x7d4a58f800,id:2,api:1,p:1209,c:495) qu
D SurfaceFlinger: Setting power mode 0 on display 0
```

Listing 4.6: An excerpt of a heatmap describing the difference between an error file and an average reference file in dataset *C*. As these log files were recordings of phone calls performed by human operators, they contained a lot of extra noise. This excerpt shows an example of this: readings from the proximity sensor, used to turn the phone's display off when holding it against the ear. The log line "`D PhoneSip: AC_CH .0[0]: RTP packet loss 36 0`" is related to the actual bug associated with this dataset, and is faintly highlighted in green. The occurrence of packet loss points to the call quality issue being networking-related, which provides a suitable starting point for further debugging. However, the root cause of the packet loss itself is not found within the logs.

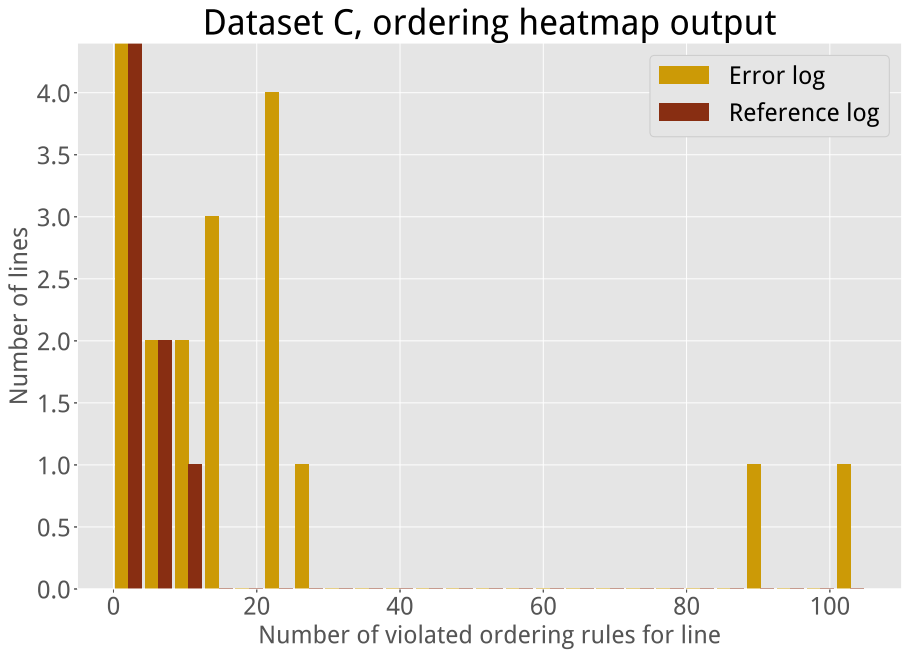


Figure 4.5: The number of violated ordering rules per line for dataset C shown as a histogram. The actual (non-truncated) values for the histogram bins at $x = 0$ are $(1498, 1294)$.

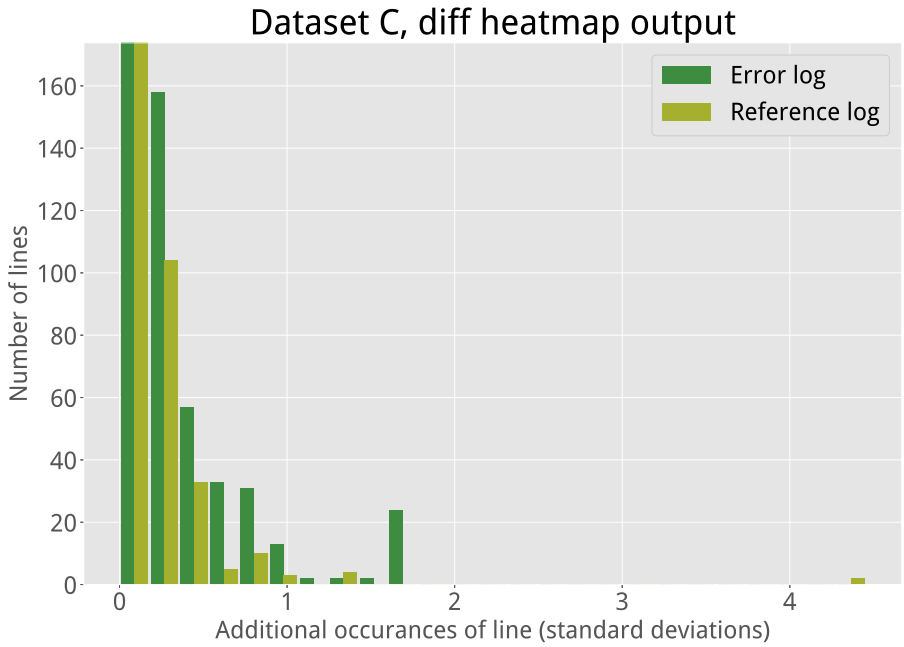


Figure 4.6: The number of extra occurrences for each log line for dataset C shown as a histogram. The actual (non-truncated) values for the histogram bins at $x = 0$ are $(1190, 1136)$.

Table 4.2: The number of added/removed lines for an error log in each dataset. In this table, a line is assumed to have been added or removed if the number of occurrences for the line in the error log deviates more than one standard deviation from the training data mean. A line is counted as reordered if it violates at least one valid ordering rule.

Dataset	Log length	Added lines	Removed lines	Reordered lines
A	1357	297 (21.8 %)	321 (23.6 %)	953 (70.2 %)
B	3578	1420 (39.6 %)	291 (8.1 %)	225 (6.3 %)
C	1512	280 (18.5 %)	9 (0.6 %)	287 (19.0 %)

Table 4.3: The number of added/removed lines for an error-free reference log in each dataset. The number of added, removed and reordered lines are much fewer than in Table 4.2, except for dataset C.

Dataset	Log length	Added lines	Removed lines	Reordered lines
A	1464	63 (4.3 %)	41 (2.8 %)	111 (7.6 %)
B	3664	458 (12.5 %)	36 (1.0 %)	110 (3.0 %)
C	1297	254 (19.6 %)	23 (1.8 %)	213 (16.4 %)

4.1.1 Analysis

As shown in Table 4.2 and Table 4.3, Regea successfully detects that there are more discrepancies in the error logs than in the reference logs for datasets A and B. This is not true for dataset C, likely due to the extra noise in the data. This shows that the accuracy of the results are highly dependent on the quality of the dataset. It is also important to point out that different types of bugs appear differently in the log files, and some more easily detectable than others. The bug related to dataset C was caused by a low level driver issue, and it may therefore not have been visible in the Android logcat output.

An issue with quantifying the accuracy of the results is that there exists no ground truth for the datasets. While it is possible to observe the amount of discrepancies detected, it is difficult to say how many of these discrepancies are related to the actual issue at hand and how many are false positives. Another issue is that an error occurring often causes several secondary effects to happen. If a set of discrepancies is found, it is difficult to determine which ones of these discrepancies are the cause of the error occurring and which ones are an effect of the error occurring.

4.2 Performance benchmarking

This section contains a set of performance benchmarks see how the practical time complexity compares to the theoretical analysis. It is important to point out that Regea, being based on evolutionary algorithms, does not have a well-defined runtime. Letting each evolution run for more generations increases the accuracy of the result, at the cost of more computation time. Regea also has an extensive set of adjustable parameters, with the default ones being relatively arbitrary. Therefore, it would be wise to not fixate too much on the exact runtimes when reading this section, but instead focus more on the differences in runtime depending on the size of the input data. For reference, the performance benchmarks in this section were conducted on a dedicated, 46 core OpenMPI cluster consisting of a mixed set of Intel desktop CPU:s, Sandy Bridge and newer.

4.2.1 Generating regular expression patterns

This section aims to analyze the practical time complexity of Regea's regex generation, as a complement to the theoretical time complexity analysis in Section 3.2.4. For this purpose, the computation time has been measured for a varying number of log files and varying log file sizes. Dataset B specified in Table 4.1 was used for this purpose.

The default behavior in Regea is for the evolution of a regex pattern to be terminated after a specified amount of time (see Section 3.2.2.4). However, the true time complexity would be more accurately measured by having each evolution run for a specified number of generations instead. Therefore, the number of generations has been set to a fixed value (100) for the benchmarks in this section. Figure 4.7 shows the computation time for training on dataset B depending on how many log files were used, and the size of the files. Figure 4.8 shows corresponding number of regex patterns generated during the training. Finally, Figure 4.9 uses the data shown in the two aforementioned figures to approximate the practical time complexity of Regea's regex pattern generation.

4.2.1.1 Analysis

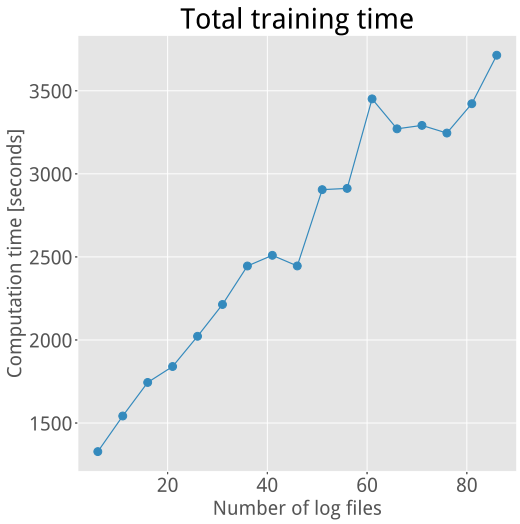
Recall from Section 3.2.4 that the theoretical best case time complexity for generating regular expression patterns is:

$$T(N, l, c, M) = \mathcal{O}(lcNM)$$

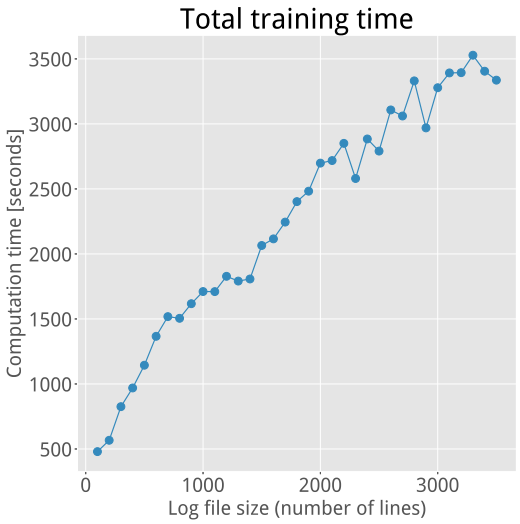
and the theoretical worst case time complexity is:

$$T(N, l, c, M) = \mathcal{O}(l^2cN^2M)$$

where N is the number of log files, l is the number of lines per file, c is the number of characters per line and M is the total number of individuals across all generations during the evolution of a single regex pattern.



(a) Training time depending on the number of log files in the training data.

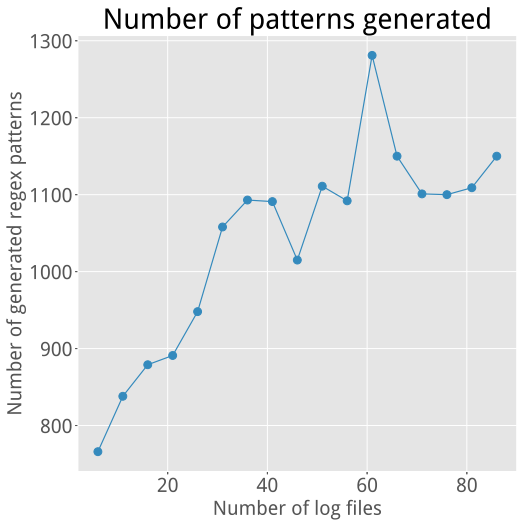


(b) Training time depending on the size of the log files in the training data.

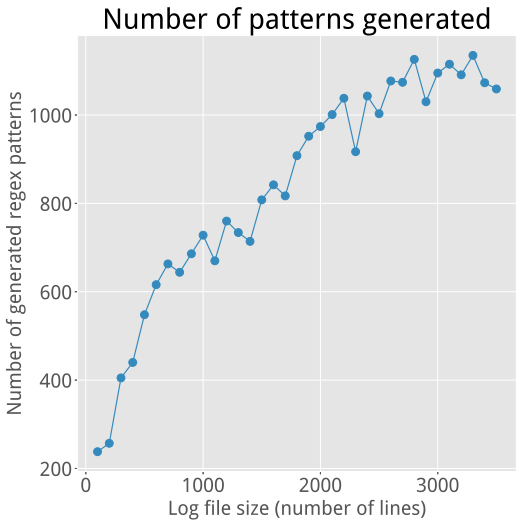
Figure 4.7: Training time for dataset B with the evolution for each regex pattern for 100 generations. Figure (a) shows the training time depending on the number of log files, and (b) shows the training time depending on the size of the log files. For (a), the full log files were used ($\sim$ 3538 lines), and for (b), all 88 log files were used. For figure (b), the size of the log files were simply truncated to the specified number of lines.

Looking at Figure 4.7, the computation time seems to increase about linearly with both the number of log files and the size of the files. Figure 4.8 shows how the number of generated regex patterns seems to converge for a larger set of log files and (to a lesser extent) also for larger log file size. Meanwhile, Figure 4.9b shows how the training time seems to depend quadratically on the amount of regex patterns created during the training session. Figure 4.9a shows the reason for the quadratic behavior, as the rate of generating regex patterns slows down over the course of any given training session.

The fact that the number of generated regex patterns starts to converge for larger data sizes is plausible. Dataset B contains a total of 88 log files. When Regea iterates through the first log file, a regex pattern is generated for many of the lines. When iterating through the second file, most lines will already be matched by a regex pattern. Thus, fewer patterns are generated when iterating through the second file compared the first file (assuming the files are of the same length). When iterating through the third file, even fewer patterns will be generated. Since the different log files are recorded from the same test case they have a lot of similarities, leading to quick convergence. The same is true (but to a lesser extent) for larger log file sizes. The more regex patterns that have already been generated, the higher the probability that a line is matched by at least one previously-generated pattern. This is true even for a dataset where the log files are not at all correlated with each other, since there only exists a finite number of possible strings of finite length.



(a) Number of generated regex patterns depending on the number of log files in the training data.



(b) Number of generated regex patterns depending on the size of the log files in the training data.

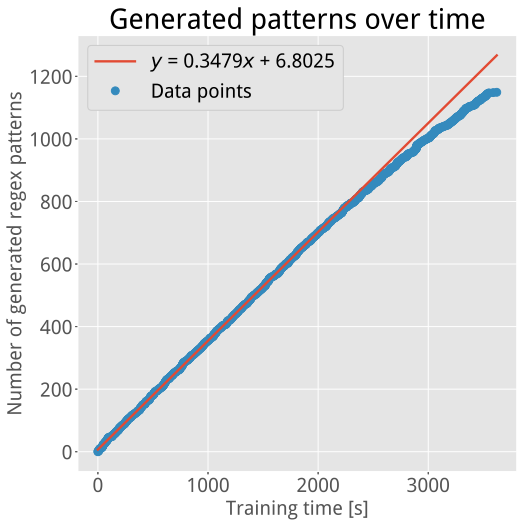
Figure 4.8: Number of generated regex patterns during training for dataset B. Figure (a) shows the number of generated patterns depending on the number of log files, while (b) shows the number of generated patterns depending on the size of the log files.

The fact that the training time depends quadratically on the number of generated regex patterns also seems plausible. Assume that R regex patterns are generated during a training session. Before generating a regex pattern for a line, Regea checks whether any of the already generated regex patterns match the line. In the beginning of the training session there are very few regex patterns to check, but at the end there are R patterns to check. This causes the training to slow down over time, see Figure 4.9b. Combining this with the fact that the R patterns also have to be generated leads to a time complexity which depends quadratically on R.

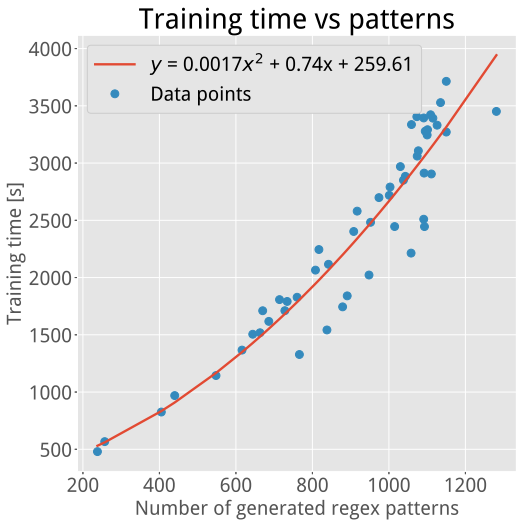
To summarize, the benchmark results are consistent with the best case time complexity described in Section (3.2.4). Figure 4.8b shows how $R < l$, and that the difference between them increases for larger log files. This causes the computation time to increase linearly with the data size rather than quadratic. To analyze the time complexity in more detail, the benchmarks will have to be performed multiple times and have their results averaged. This will mitigate the effects of noise in the data, making it easier to observe the time complexity behavior.

4.2.2 Creating and evaluating ordering rules

This section aims to analyze the practical time complexity of creating and evaluating ordering rules, as a complement to the theoretical time complexity analysis in Section 3.3.2. The time required to evaluate all possible ordering rules for dataset B depending on the file size and the number of patterns is shown in Figure 4.10.



(a) The cumulative number of generated regex patterns as a function of time over the course of a single training session.



(b) The total training time depending on the number of generated regex patterns for all training sessions in Figure 4.7.

Figure 4.9: Figure (a) shows how the list of generated regex patterns grows as a function of time during an example training session. This rate starts out close to linear, but decreases over time. Figure (b) shows how the total training time depends on how many regex patterns were generated during the training. Note that (a) and (b) have their x-axis and y-axis mirrored relative to each other.

4.2.2.1 Analysis

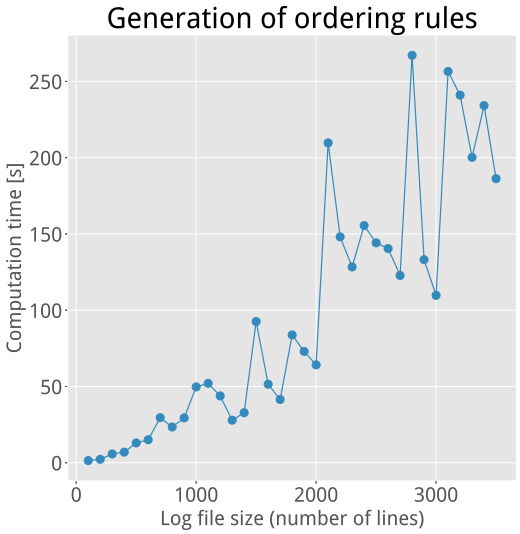
Recall from Section 3.3.2 that the theoretical time complexity of evaluating all possible ordering rules for a dataset is

$$T(N, l, R, m) = \mathcal{O}(R^2 m^2 N) = \mathcal{O}(l^2 N) = \mathcal{O}(N l^2)$$

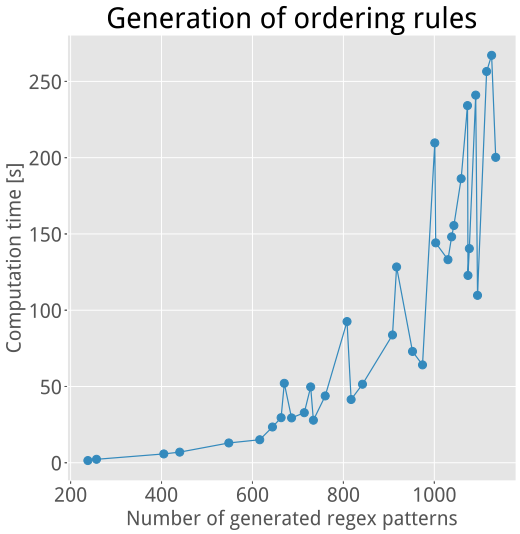
As shown in Figure 4.10, the computation time seems to increase quadratically with both the file size and the number of regex patterns. This is in line with the theoretical time complexity analysis.

4.2.3 Insertion of missing log entries

This section provides a practical benchmark of the practical time complexity of inserting missing log entries, as a complement to the theoretical time complexity analysis in Section 3.4.2. The time required to insert missing log lines depending on the file size and the number of generated regex patterns is shown in Figure 4.11.



(a) Computation time required to evaluate all possible ordering rules depending on the log file size.



(b) Computation time required to evaluate ordering rules depending on the number of generated regex patterns.

Figure 4.10: Computation time required to evaluate all possible ordering rules for dataset B depending on the log file size and the number of generated regex patterns. The patterns used for creating these figures were the ones generated from creating Figure 4.7b and Figure 4.8b.

4.2.3.1 Analysis

Recall from Section 3.4.2 that the theoretical time complexity of inserting missing lines into an error log has the time complexity

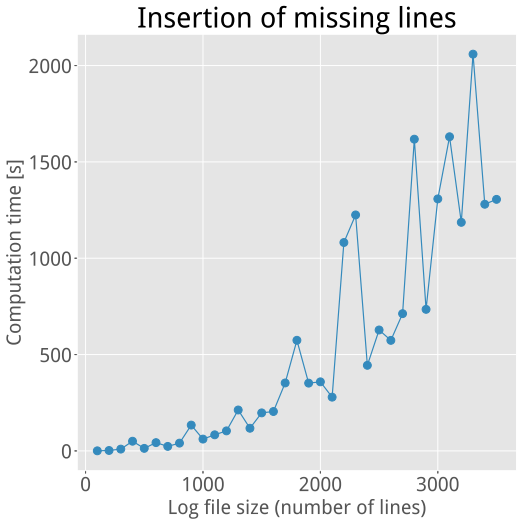
$$T(l, R, m) = \mathcal{O}(R^2 m^2 l^2) = \mathcal{O}(l^4)$$

As shown in Figure 4.11, the computation time seems to increase (at least) quadratically with both the file size and the number of regex patterns. Notice that the computation times are an order of magnitude higher than in Figure 4.10, which points to the time complexity being higher than $\mathcal{O}(l^2)$. This is in line with the analysis in Section 3.3.2.

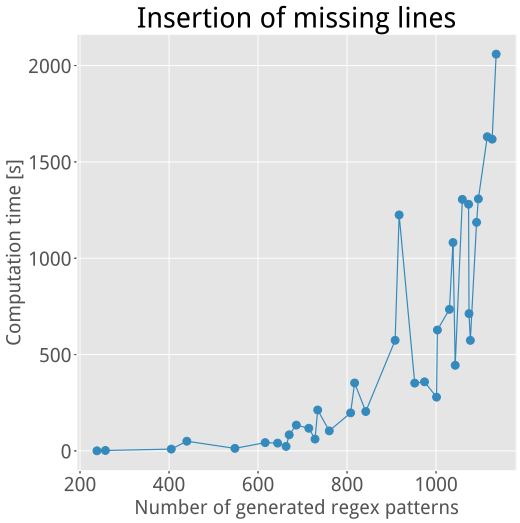
4.3 A closer look at the inner workings of Regea

This section features a set of miscellaneous results which do not fit neatly into the previous sections, but are still important for understanding how Regea works.

As explained in Section 3.2.2, Regea uses genetic programming to generate regular expression patterns to match the lines in the reference files. The specificity and structure of the regex patterns depends on the training time, the size of the log files, and how similar the log files are to each other. Below is an example of what an excerpt of a generated list of regular expressions may look like after training:



(a) Computation time required to insert missing log entries depending on the log file size.



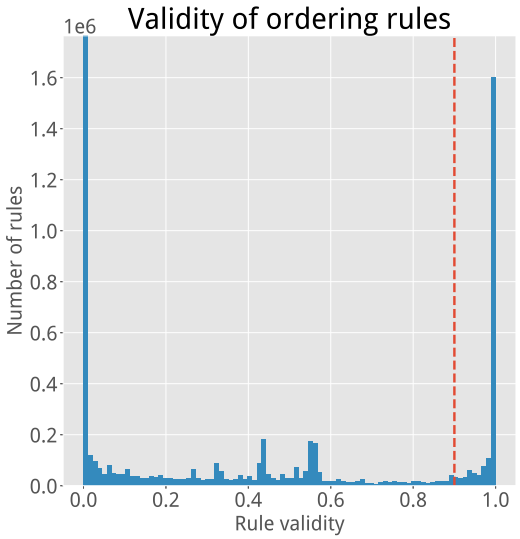
(b) Computation time required to insert missing log entries depending on the number of generated regex patterns.

Figure 4.11: Computation time required to insert all missing lines for an error log in dataset B depending on the log file size and the number of generated regex patterns.

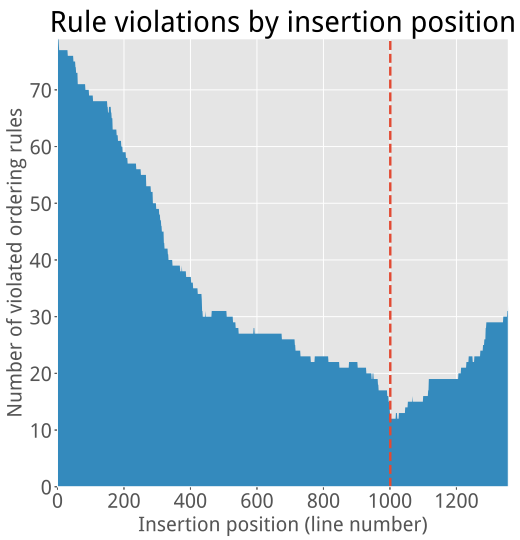
```
^.{15,21}[!-y].[ -a](?:\b(?=\w)).[1-\~]\B[!-M]\B[ -y].[ -\^].\B.{30,36}$
^.{34,40}\B(?=(?=[7-F]))\B[3-E].\b.\B..\B[,-\~]...\b(?:(?<=\w)\b).{0,3}$
^.{0,3}\b(?:\b(?=\w))\b\S[^%-\~][L-z]\B[M-\~]\S\B[\^-\~][ -v]\S[ -C].{34,40}$
^.{30,36}\B.[ -\~].\B\w\B[ -\~](?:(?<=\w)\b).\W[ -\~] [ -9][ -\~][\\-\~]\w.{3,9}$
^.{0,4}(?=(?:(?<=\w)\b)\s[Z-v])(?:(?<=\w)\b)(?=(?:(?<=\w)\b)\b.[Z-v]).\w.{40,46}$
^.{22,28}(?=.)\W[ -4]..[ -s][ -\~]\B\S..[ -\~]\S.\B. [ -\~]\B\w[ -x][\\-\~].{0,6}$
```

As explained in Section 3.3, Regea generates a set of ordering rules according to the training data. The rules are generated randomly and a validity is assigned to each rule depending on the proportion of training data logs which satisfy the rule. By default, a validity of at least 0.90 is required for an ordering rule to be considered valid. An example distribution of the validities for a set of randomly generated ordering rules is shown in Figure 4.12a. As can be seen in the figure, all randomly generated rules have a validity close to zero or one, i.e. they are likely to either match (almost) all of the training data logs or (almost) none.

As mentioned in Section 3.4, log lines which Regea deems to be missing from an error log are imported from the training data and inserted into the error log. To find a reasonable location to insert a line, Regea evaluates the ordering rules for the line for all positions in the error log. The line is inserted at the position where it violates the least number of ordering rules. Figure 4.12b shows an example for how the number of violated ordering rules may depend on the insertion position of a log line.



(a) Histogram of calculated rule validities for all possible ordering rules for a set of patterns generated for dataset B. The vertical red line shows the default threshold for an ordering rules to be considered valid (0.90). The proportion of valid rules in this instance is approximately 13.9 %. The bin at $x = 0$ has been truncated for better readability (actual value $\approx 9.3 \cdot 10^6$).



(b) Example of how the number of violated ordering rules depends on the insertion position when inserting a missing line into the error log. The vertical red line shows the chosen position to insert the log line ($x = 1001$).

Figure 4.12: Figure (a) shows the distribution of rule validities for a set of randomly generated ordering rules, while (b) shows an example of the number of violated ordering rules depending on the position when inserting a missing line into the error log.

5

Conclusion

A log file analyzer, called *Regea*, has been implemented which uses genetic programming to generate regular expression patterns. These regex patterns are then used to classify log entries and perform statistical analysis. Any discrepancies are presented to the user.

Regea is able to detect discrepancies in log files so long as the files are similar enough, for example from automated, repeated tests.

The main issue with Regea in its current form is that it only works for very specific cases. The program requires a large amount of log files for training (up to about a hundred), the files have to be free of extra noise, and they have to labelled correctly (error/no error). Creating this kind of dataset by hand is a time-consuming task and will in many cases be slower than simply studying the log files manually. This limits Regea to mostly work on log files created by automated testing scenarios. Additionally, the provided log files have to contain at least some data which is related to the issue at hand. If nothing related to the error is logged, then nothing can be found.

Aside from this fundamental issue, there are (at least) three problems with the current implementation of Regea: the fitness evaluation of the regex patterns, the output format and the computational performance.

The genetic programming algorithm which generates regular expressions uses a fitness function to evaluate how well each regex pattern performs, as described in Section 3.2.2.3. A fitness function for an evolutionary algorithm has to explicitly incorporate all aspects of what one wants the computer to optimize. This can be a very a difficult thing to do, and failure to do so will cause the algorithm to gain fitness in ways that were not intended[16]. Aside from this, the fitness function also needs to be computationally efficient to evaluate, since it is run for each individual in each generation of the evolution. The fitness function for Regea has been adjusted and rewritten several times over the course of the project. In its current state, it aims to create as specific regex patterns as possible which match as few times as possible per log file (but at least once per file). This provides a reasonable middle-ground between specificity and generality of the generated regex patterns. However, the current fitness function requires performing regex operations on all training data files for every individual in the population for every generation of the evolution. This is not computationally efficient, and accounts for the majority of Regea's runtime. Aside from this, the algorithm currently has the option of exploit-

ing optional nodes to gain an unjustified amount of fitness. Inserting an optional node into an individual (remember that individuals are represented as tree structures) causes that whole subtree to be regarded as optional. The optional node itself gives a negative fitness (see Table 3.3) but all nodes connected to the optional node will give a positive fitness, likely resulting in a net-positive fitness contribution for the optional subtree. A possible solution to this is to apply a modifier to the whole optional subtree, instead of just the optional node itself. Negative lookaheads have the same problem as optional nodes, but have more possibilities of being inserted (since they do not consume characters). Negative lookaheads are therefore excluded from Regea to prevent too much fitness exploitation.

The output format is another part of Regea which has changed a lot over the course of the project. The current output format consists of HTML-based, colored heatmaps, see Listing 4.1 and Listing 4.2. Regea takes an error log file and highlights each line depending on how much it deviates from the training data. This provides an output file which is easy and intuitive for a user to understand (a more strongly colored line points to a more certain discrepancy). However, it hides certain information which may be of importance. For example, in an ordering heatmap it is possible to see that one line violates more ordering rules than another, but it is not possible to see the exact number of violated rules and also not which ordering rules were violated. This type of data may be relevant depending on the nature of the issue at hand. Another output format which was tried was JSON. In this case, each line in the error log had its violated ordering rules as subelements. This worked well but it was difficult to get an overview by quickly looking through the file. Perhaps Regea could support multiple different output formats and let the user decide what is the most appropriate format for each situation.

Another issue is the computational efficiency (or lack thereof). As showed in Figure 4.7, training on the entirety of dataset B takes about one hour on a 46 node OpenMPI cluster. This means that the same computation will take several hours on an average desktop computer. Long log lines are also an issue since larger regex patterns are slow to evaluate. Performing mutation or crossover on such patterns also likely leads to patterns which do not match the input data, leading to slow evolution.

5.1 Future work

The current implementation of Regea has many possibilities for improvements which have been left unexplored. One such topic is the intricacies of the genetic programming algorithm. Regea has a large set adjustable parameters, including crossover probability and several mutation probabilities for different types of mutation. These currently have to be set manually by the user (or left at their default values). It was planned to have Regea optimize these parameters automatically for the current dataset, but this optimization logic was not able to be implemented in time for this report. It will likely be included in a future version of Regea. When initializing the population, individuals are very generic and grow more specific over time. Another possibility is to have individuals start very specific and grow more general over

time. When calculating the fitness for a regex pattern, the evaluation time for the pattern can be taken into account to discourage the algorithm from creating regex patterns which are very slow to evaluate. The simplest way to do this is to divide the fitness value calculated in Section 3.2.2.3 by the time required to calculate said fitness. Apart from improving computational efficiency, this can also act as a way to prevent the algorithm from creating individuals with an abundance of optional or otherwise unnecessary regex segments.

Another possibility of improvement is the statistical calculations used to determine whether a line has been added or removed. Currently, the mean and standard deviations of the number of matches for each regex pattern in evaluated across the training set and is compared to the number of matches in an error log. This generally works well enough, but has issues in some scenarios. If a pattern has a mean close to zero and a high standard deviation, this may cause the difference between the mean and zero to be less than one standard deviation. It is currently not possible to detect whether an occurrence of such a pattern is missing from an error log since the number of occurrences cannot be less than zero. A possible improvement is to not only compare the number of matches for a pattern in the error log to the mean and standard deviations for the training data, but also compare it to the min and max occurrences in the training data.

Another point of interest is that of computational performance. Optimizing the logic within the fitness function is the most obvious way to do this, as it accounts for the majority of Regea's runtime. Other ways include rewriting the application in a fast, compiled language like C. This way, it will also be possible to use efficient regex libraries like PCRE2 directly, without having to spawn an external process like with Python. Furthermore, it will also possible to implement efficient multi-processing using C11 threads or POSIX threads, bypassing the overhead of using OpenMPI (though those methods do not allow for parallelizing computations across multiple computers). Another possibility is to utilize graphics cards (GPU:s) to speed up calculations. For example, there already exists an efficient implementation of `grep` using CUDA[17]. Computational performance is important for evolutionary algorithms since letting the algorithm run for longer periods of time yields more accurate results.

As of right now, Regea uses genetic programming almost exclusively to generate its regular expression patterns. The reasoning for this choice was to explore the possibilities of using machine learning for analyzing log files. In hindsight, combining machine learning techniques with more naive methods will probably result in a more favorable outcome. For example, log lines could be grouped together by using sting comparison to check for common substrings. Machine learning techniques could then instead be used to analyze the parts where naive string comparisons are not enough. This will likely result in a log file analyzer which is more robust, more manageable and more computationally efficient than the current implementation.

Bibliography

[1] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2018. DOI: 10.1109/TDSC.2017.2762673.

[2] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 207–218. DOI: 10.1109/ISSRE.2016.21.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs.," Jan. 2010, pp. 37–46. DOI: 10.1145/1629575.1629587.

[4] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in IBM BlueGene/L event logs," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, 2007, pp. 583–588. DOI: 10.1109/ICDM.2007.46.

[5] D. Gunning, M. Stefik, J. Choi, T. Miller, S. Stumpf, and G.-Z. Yang, "Xai—explainable artificial intelligence," *Science Robotics*, vol. 4, no. 37, 2019. DOI: 10.1126/scirobotics.aay7120.

[6] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, 2016. DOI: 10.1109/TKDE.2016.2515587.

[7] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Playing regex golf with genetic programming," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 1063–1070. DOI: 10.1145/2576768.2598333.

[8] P. M. Stahl. (2021). "Grex – a command-line tool and library for generating regular expressions from user-provided test cases," [Online]. Available: https://github.com/pemistahl/grex (visited on 04/10/2021).

[9] M. Wahde, *Biologically Inspired Optimization Methods*. WIT Press, 2008, ISBN: 9781845643447.

[10] J. Friedl, *Mastering Regular Expressions*. O'Reilly, 2006, ISBN: 9780596550028.

[11] G. Berry and R. Sethi, "From regular expressions to deterministic automata," *Theoretical Computer Science*, vol. 48, pp. 117–126, 1986, ISSN: 0304-3975. DOI: 10.1016/0304-3975(86)90088-5.

[12] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 1, pp. 39–47, 1960. DOI: 10.1109/TEC.1960.5221603.

[13] A. Brüggemann-Klein, "Regular expressions into finite automata," *Theoretical Computer Science*, vol. 120, no. 2, pp. 197–213, 1993, ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90287-4.

[14] G. Jäger and J. Rogers, "Formal language theory: Refining the chomsky hierarchy," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1598, pp. 1956–1970, 2012. DOI: 10.1098/rstb.2012.0077.

[15] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012.

[16] J. Lehman, J. Clune, D. Misevic, C. Adami, L. Altenberg, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson, N. Cheney, P. Chrabaszcz, A. Cully, S. Doncieux, F. C. Dyer, K. O. Ellefsen, R. Feldt, S. Fischer, S. Forrest, A. Fŕenoy, C. Gagńe, L. Le Goff, L. M. Grabowski, B. Hodjat, F. Hutter, L. Keller, C. Knibbe, P. Krcah, R. E. Lenski, H. Lipson, R. MacCurdy, C. Maestre, R. Miikkulainen, S. Mitri, D. E. Moriarty, J.-B. Mouret, A. Nguyen, C. Ofria, M. Parizeau, D. Parsons, R. T. Pennock, W. F. Punch, T. S. Ray, M. Schoenauer, E. Schulte, K. Sims, K. O. Stanley, F. Taddei, D. Tarapore, S. Thibault, R. Watson, W. Weimer, and J. Yosinski, "The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities," *Artificial Life*, vol. 26, no. 2, pp. 274–306, May 2020. DOI: 10.1162/artl_a_00319.

[17] B. K. Manish Burman. (2016). "Grep on cuda," [Online]. Available: https://github.com/bkase/CUDA-grep (visited on 05/17/2021).

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY