Efficient classification of strings using regular expressions

Mowbray, Miranda; Horne, William; Rao, Prasad

Hewlett Packard Labs
HPE-2017-03

Keyword(s):
pattern matching; regular expressions; network security; bioinformatics

Abstract:
This paper presents a new method of compiling a large set of patterns, described in regular expression syntax, into a new data structure, in order to efficiently determine at runtime which of the patterns is matched by an input string. This problem arises in network security and a wide variety of other businesses. In evaluation tests on data sets used for security information and event management and in bioinformatics, our method outperforms the most commonly-used method by a factor of 6.8 or more.
Efficient classification of strings using regular expressions
Miranda Mowbray, William Horne, Prasad Rao

Abstract
This paper presents a new method of compiling a large set of patterns, described in regular expression syntax, into a new data structure, in order to efficiently determine at runtime which of the patterns is matched by an input string. This problem arises in network security and a wide variety of other businesses. In evaluation tests on data sets used for security information and event management and in bioinformatics, our method outperforms the most commonly-used method by a factor of 6.8 or more.

Problem statement
The challenge addressed in this abstract is to compile a large set of patterns, described in regular expression syntax, into a new data structure in order to efficiently determine at runtime which of the patterns is matched by an input string. The set of patterns has the property that every input string matches at least one pattern; for some applications an input string may match more than one pattern. This is a problem of efficient classification, for a large number of known classes, where each class is specified by a regular expression such that an input string matches the regular expression if and only if it is a member of the class. We originally studied this problem for an application in network security, however it also arises in document classification, routing, e-service discovery, and bioinformatics. Solutions may also be applicable to (for example) efficiently determining which policies should be applied to a file within a computer network, or the efficient classification of customers in CRM applications. Regular expressions are a highly expressive pattern syntax, and the input strings may be over any finite alphabet, so patterns expressed in other syntaxes are quite likely to be translatable into regular expressions. We believe that our solution to this problem is of potential interest to a wide variety of businesses.

Our solution
The most commonly-used approach to this problem is sequential matching, that is, checking each input string against each pattern in turn. The problem with this approach is that the running time for the algorithm is proportional to the number of patterns. Moreover, we must do a full pattern match, which can be computationally expensive depending on the complexity of the patterns being tested. The question is whether we can do something more efficient by testing strings against fewer, simpler patterns.

In our approach, we do some simple initial tests on input strings at runtime. Our candidate initial tests are chosen to be swift to carry out. In our experiments we have used initial tests based on the length of the string, its first character, and whether a particular character is present somewhere in the string. We use the information gained by these initial tests to eliminate some patterns. For example, if an input string fails an initial test that the string contains “a”, we can deduce that the string cannot match any pattern that is only matched by strings which contain “a”. We then check for matches to the remaining patterns.

Carrying out all possible initial tests would potentially lead to a huge number of different sets of test results, and an impractically large data structure. Our key innovation is to use training data to compile a decision tree specifying which initial tests to do, and in which order.

The training data may be a set of randomly-generated strings that match each of the patterns. We have previously developed a method that generates random strings matching a given regular expression [2], which we use in our solution.

If it is known that some patterns are likely to be matched more often than others, then it makes sense to generate more strings matching the popular patterns when constructing the training set. For example, some patterns will correspond to devices that are present in many customer systems, and some to much rarer devices. If an approximate distribution of matched patterns is known or can be
guessed, this distribution can be used to determine how many matching strings to generate for each pattern. If no such information is known, the default option is to generate the same number of matching strings for each pattern: this is what we do in the evaluation experiments reported in this paper.

An alternative (and preferable) way of obtaining training data is to use a set of input strings collected in the past, if such a set is available. For example, one use case in network security is the testing of security-related event reports sent from a collection of different sources, in which case the input strings collected in the past are event reports previously sent from the same set of sources. The distribution of input strings will vary between different domains and installations – for instance, it will be different for different installations of the same network security product - and so ideally the training set will be a large set of strings that have previously been input into the same installation. Since the decision tree is “learned” from the training data, different domains and installations will generate different decision trees, optimized for their own distributions of input strings. When an installation is first set up there will be no past input strings available to use in a training set, and so a training set of randomly-generated strings matching the patterns can be used to bootstrap the system.

As new input strings come into the system they can be recorded or sampled to create a fresh training set, to enable periodic recompilation of the decision tree to reflect more recent input. In this way, the decision tree will evolve as the input data sources do. The compilation step can be completely automated, but for performance reasons recompilation will probably have to be periodic rather than occurring after every input string. A very small change in the training set may result in a large change in the optimum decision tree corresponding to the training set, especially if the change affects the best choice for the test at the root of the decision tree. So periodic recompilation rather than recompilation after every input string is more suitable for applications - for instance, security information and event management (SIEM) in large networks - for which there may be a very high throughput of input strings. To reduce compilation time, we bound the size of the decision tree. The decision tree does not have to be very large for good results: in the evaluation experiments reported in the next section, the decision tree has the same number of internal nodes as there are characters in the alphabet used for the input strings (20 or 128).

Recompilation of the decision tree will also be necessary if the pattern set changes. For example, when a SIEM manufacturer makes the decision to support a new device by a third party, this will mean the addition of new patterns to the pattern set. In the applications for which the problem we are addressing arises, the pattern set typically remains stable for weeks or months, whereas the inter-arrival time between input strings may be less than a second.

To compile the decision tree, at each step we find the leaf node with the maximal value of the product of the number of training strings that reach this node, and the number of tests to be done on these strings, counting both initial tests carried out on the way to this node, and the remaining patterns that would need to be checked for strings that reach this node. (If the same pattern occurs more than once in the pattern set, it is only checked once.) The intuition here is that this will minimize the total number of pattern-checking tests we have to make on average. We determine the candidate initial test that would eliminate the largest number of these checks. We add two new branches to the node leading to two new leaves, one to be reached by input strings that pass the test and the other by those that fail it, and determine which patterns can be eliminated at each of these new nodes. The step is repeated until the decision tree reaches the required size.

In more detail, the compilation algorithm first determines the functions MP(p, t) MS(s, t) from the set of all possible initial tests to {1, -1, 0}, for each different pattern p in the pattern set and each index s of a string in the training set, where for each possible initial test t these functions are defined as

\[ MP(p, t) = 1 \text{ if all strings that match } p \text{ pass } t, \]
\[ MP(p, t) = -1 \text{ if all strings that match } p \text{ fail } t, \]
\[ MP(p, t) = 0 \text{ otherwise} \]

(“all strings” in this definition means all finite strings of characters in the alphabet, not just the strings
in the training set), and
\[ MS(s,t) = 1 \text{ if the string indexed by } s \text{ passes } t, \]
\[ MS(s,t) = -1 \text{ if the string indexed by } s \text{ fails } t. \]
The reason we use an indexing set for the training strings rather than using the training strings directly is that the same string may appear multiple times in the training set. The decision tree is initialized as a single node \( n_0 \), and grown iteratively. As the decision tree is grown, for each new node \( n \) the compilation algorithm will calculate the sets \( S(n) \) and \( P(n) \), which are, respectively, a set indexing the training strings that will reach node \( n \), and the set of remaining patterns that need to be checked for the strings indexed by \( S(n) \). The set \( S(n_0) \) indexes the whole training set, and the set \( P(n_0) \) is the pattern set after removal of duplicate patterns. For all nodes \( n \) other than \( n_0 \), the decision tree records the parent node \( \text{Parent}(n) \), the test \( t(n) \) applied at node \( \text{Parent}(n) \), and the value \( e(n) = MS(s,t(n)) \) of the test when applied to strings in \( S(n) \): this value will either be 1 for all strings \( s \) indexed by \( S(n) \), or -1 for all strings indexed by \( S(n) \), according to which branch is taken from \( \text{Parent}(n) \) to reach node \( n \).

The sets \( S(n) \) and \( P(n) \) can be determined from the decision tree by iteratively using the identities
\[ S(n) = \{ s \in S(\text{Parent}(n)) : MS(s,t(n)) = e(n) \} \]
\[ P(n) = \{ p \in P(\text{Parent}(n)) : MP(p,t(n)) \neq -e(n) \} \]

The intuition behind the equation for \( P(n) \) is that in order to obtain the set of patterns that remain to be checked for strings that reach \( n \), you take the set of patterns that remain to be checked for strings that reach the parent of \( n \), and remove those patterns that only are matched by strings which give a different result than \( e(n) \) under test \( t(n) \) than the strings indexed by \( S(n) \) do.

At each iteration step, the algorithm picks a leaf node \( n \) such that
\[ |P(n)| \cdot |S(n)| \]
is maximal, and then for this \( n \) picks an initial test \( t \) such that the value
\[ \{ (s,p) \in S(n), p \in P(n) : MP(p,t) \cdot MP(p,t) = -1 \} \]
is maximal. The iteration step adds to the decision tree two new leaf nodes, \( n^+ \) and \( n^- \), attached to \( n \) by two new branches (so that \( n \) is no longer a leaf node, but is now an internal node), with \( t \) as the test that will determine which of the two branches will be followed. When an input string is processed by the decision tree, if it reaches node \( n \) of the tree, the processing will then carry out test \( t \). The iteration step sets
\[ \text{Parent}(n^+) = \text{Parent}(n^-) = n \]
\[ t(n^+) = t(n^-) = t \]
\[ e(n^+) = 1 \]
and \[ e(n^-) = -1 \]

If the number of internal nodes in the decision tree is now equal to the bound \( N \), the algorithm terminates; otherwise a new iteration step is carried out. (As mentioned above, in the evaluation section \( N \) is chosen to be the number of characters in the alphabet used for the input strings, i.e. 20 or 128. This is a somewhat ad hoc choice, and one potential enhancement of the algorithm would be a better-founded method of choosing a good value for \( N \).

Note that the value that is maximized by the test \( t \) is equal to
\[ |P(n)| \cdot |S(n)| - |P(n^+)| \cdot |S(n^+)| - |P(n^-)| \cdot |S(n^-)| \]

To process an input string using the decision tree, initial tests are applied to the string in the order indicated by the decision tree based on the answers to the previous tests, until the string reaches a leaf node \( n \). Then the string is checked against each of the patterns in the set \( P(n) \), to see which of these it matches. The matched patterns are output.
Evidence the solution works

We compiled decision trees for a set of patterns used by ArcSight network security products, and a set of protein patterns used in bioinformatics (PROSITE v.14.0, [3]). We have implemented our solution to the extent necessary to carry out the evaluation experiments reported here, but not to a sufficient extent to be able to do performance testing and a set of protein patterns used in bioinformatics (PROSITE v.14.0, [3]). The alphabet for the ArcSight input strings is 128-character ASCII, and the alphabet for the proteins consists of 20 characters corresponding to the 20 amino acids. In both cases we used a training set consisting of one random match to each pattern in the pattern set. The choice of using only one random match to each pattern was conservative: using a larger training set with more than one match to each pattern would have resulted in more accurate training. As will be seen, we obtained satisfactory results even with this small training set, but larger training sets could be used in practice. We tested the decision trees using an independently-generated testing set five times larger than the training set.

As a metric for the efficiency of a given approach, we used the total number of tests required to process the testing set using sequential matching (where sequential matching just means checking each input string against each pattern in turn) divided by the total number of tests required by the approach, counting both initial tests and tests of remaining patterns. If all tests took the same time, a value $m$ of this metric would indicate that the approach was $m$ times more efficient than sequential matching. Our candidate initial tests are chosen to be rapid to carry out, so this metric may underestimate our method.

The results are summarized in the Table 1, which also shows the minimum, median and maximum values of the metric on individual strings in the testing set. (The value of the metric for an individual string is just the size of the pattern set divided by the total number of tests of that string that are required by the approach being considered.) Figures 1 and 2 give more detail, showing the frequencies with which different values of the metric occurred: for these two Figures the metric value is rounded to the nearest integer.

We tested dependency on training data by re-running the experiment for the protein patterns using two other training sets. There was less than 3% difference in the overall metric values obtained with the three different training sets.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Pattern Set</th>
<th># patterns</th>
<th>min</th>
<th>median</th>
<th>max</th>
<th>metric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>ArcSight</td>
<td>16795</td>
<td>1.3</td>
<td>8.2</td>
<td>28.4</td>
<td>6.8</td>
</tr>
<tr>
<td>Ours</td>
<td>Proteins</td>
<td>1277</td>
<td>11.9</td>
<td>14.8</td>
<td>23.6</td>
<td>15.2</td>
</tr>
<tr>
<td>Sequential</td>
<td>any</td>
<td>any</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1: Frequencies of different metric values: Our approach, ArcSight pattern set
Competitive approaches

As mentioned before, the most commonly-used approach to the problem is sequential matching. There is a large literature (eg [4]) on methods for increasing the speed of checking whether a single input string matches a single regular expression, which speeds up sequential matching. This work is complementary to our solution, as it can be used to speed up the final step of our approach, which is to check for matches to the patterns that remain after some have been eliminated.

Another approach to the problem is to combine all the patterns with an “OR” operator, build a finite automaton corresponding to the resulting very large regular expression, label each accepting state to indicate the corresponding pattern, and then use standard automaton processing to determine all the accepting states reachable by an input string. For nondeterministic automata, this gives no improvement in runtime over sequential matching. For deterministic automata, this approach is very efficient for small pattern sets, but it does not scale well: in the worst case, the storage and memory requirement scales exponentially with the number of patterns. In practice worst-case scaling almost never occurs, and Patel et al’s implementation of this approach includes techniques that can significantly reduce storage and memory requirements [5], however scalability is still a potential issue.

A technique that is used to search for web pages that contain matches to one or more of a set of specified patterns is to identify substrings that occur often (or always) in matches to one of the patterns but occur rarely in input strings, and examine input strings for these substrings as an initial step [9]. (For the application to web search, the input strings are HTML descriptions of web pages.)
This technique works well if most input strings do not match any pattern, but it has exactly the wrong behaviour for our classification problem in which all input strings match at least one pattern, as it increases the processing time for such strings.

A different type of approach, which can give significantly better performance than sequential matching, is to exploit a parallel architecture by implementing a match-checker for each pattern in a different virtual machine or Graphical Processing Unit, and checking for matches to each pattern in parallel. Using our approach, it is possible to exploit available parallelism and obtain higher throughput than can be obtained by checking different patterns in parallel. This can be done by compiling the patterns into code that identifies which patterns are matched by an input string more efficiently than sequential matching, implementing the same code in each of the virtual machines/GPUs, and processing different input strings in parallel instead of checking different patterns in parallel. Thus, our approach is complementary to the use of a parallel architecture.

Two approaches in the literature, like ours, also do initial testing to eliminate some patterns and then test for matches to the remaining patterns. Neither uses training data. These two approaches produce particularly compact compiled data structures. Chan et al.’s RE-tree approach [6, 7] compiles the patterns into a tree with a pattern at each leaf and a compact regular expression at each non-leaf node, such that if an input string does not match the expression at a node, all patterns at leaves below it can be eliminated. Gentile’s approach [8] compiles intervals for each pattern, using a map from strings to the unit interval, such that if the image of an input string lies outside the intervals for a pattern, that pattern can be eliminated. It is not straightforward to implement either of these approaches for ArcSight’s patterns: the RE-tree method is ambiguously described, and Gentile’s paper assumes that the patterns use a more restricted regular expression syntax than ArcSight’s patterns do.

The two RE-tree papers [6, 7] report results for the RE-tree approach on synthetic pattern sets generated using the same algorithm with different parameter values. The alphabet used has 20 characters. The RE-tree conference paper [6] indicates a metric value of approximately 2 for a data set constructed using parameter values \( \rho=0.75, \ cnum=50, \ csize=25, \ \alpha=10 \), and the journal version [7] indicates a metric value of approximately 3 for parameter values \( \rho=0.5, \ cnum=250, \ csize=200, \ \alpha=10 \). We implemented Gentile’s and our approaches on two pattern sets generated in the way specified in the two RE-tree papers, with these two sets of parameter values, and tested them using testing data (and in addition, for our approach, independently-generated training data) that were derived following the method used for obtaining testing data in the RE-tree papers.

The evaluations reported in [6,7] used sets of 1,000 input strings for testing, but for greater certainty we used sets of 5,000 input strings to test Gentile’s and our approaches. The two training data sets used for our approach each consisted of 1,000 strings. Since Gentile’s initial tests can be done quickly using a known technique for fast interval matching, to be conservative we only counted the tests of remaining patterns when calculating the value of the metric for Gentile’s approach. Thus in addition to potentially underestimating the performance of our approach, the metric used slightly overestimates the performance of Gentile’s approach.

The results are summarized in Table 1. The larger RE-tree data set (“RE-trees 2”) contains many duplicate patterns - after deduplication, the size of the pattern set decreases from 50,000 to just 13,940 – and an early step of our approach is to deduplicate the pattern set. So caution should be taken into reading too much into the large metric (21.3) obtained for our approach on this data set, as this level of efficiency may not be easily obtained for pattern sets containing few duplicates. However, the results in Table 1 indicate that, for these synthetic data sets at least, our approach is noticeably more efficient than either Gentile’s approach or Chen et al.’s RE-tree approach.

Figures 3 to 6 give more detail. They show the frequencies with which different values of the metric occurred for testing strings when using our approach (Figures 3 and 4) and Gentile’s approach (Figures 5 and 6) on these two data sets. As for Figures 1 and 2, for Figures 3 to 6 the metric value is rounded to the nearest integer.

Table 1: Metric values for different approaches and pattern sets
<table>
<thead>
<tr>
<th>Approach</th>
<th>Pattern Set</th>
<th># patterns</th>
<th>min</th>
<th>median</th>
<th>max</th>
<th>metric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>RE-trees 1</td>
<td>1250</td>
<td>3.5</td>
<td>15.4</td>
<td>54.3</td>
<td><strong>14.1</strong></td>
</tr>
<tr>
<td>Ours</td>
<td>RE-trees 2</td>
<td>50000</td>
<td>6.8</td>
<td>20.3</td>
<td>70.8</td>
<td><strong>21.3</strong></td>
</tr>
<tr>
<td>Gentile’s</td>
<td>RE-trees 1</td>
<td>1250</td>
<td>2.4</td>
<td>3.9</td>
<td>29.7</td>
<td><strong>4.2</strong></td>
</tr>
<tr>
<td>Gentile’s</td>
<td>RE-trees 2</td>
<td>50000</td>
<td>2.4</td>
<td>3.1</td>
<td>56.7</td>
<td><strong>3.7</strong></td>
</tr>
<tr>
<td>RE-trees</td>
<td>RE-trees 1</td>
<td>1250</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>approx 2</td>
</tr>
<tr>
<td>RE-trees</td>
<td>RE-trees 2</td>
<td>50000</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>approx 3</td>
</tr>
<tr>
<td>Sequential</td>
<td>any</td>
<td>any</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

Figure 3: Frequencies of different metric values: Our approach, RE-tree patterns 1

![Figure 3](image1.png)

Figure 4: Frequencies of different metric values: Our approach, RE-tree patterns 2

![Figure 4](image2.png)

Figure 5: Frequencies of different metric values: Gentile’s approach, RE-tree patterns 1

![Figure 5](image3.png)
Possible refinements and next steps

In addition to the functionality described in this paper, there are some possible refinements that might be added to our solution. For example, rather than treating all possible initial tests equally, the decision tree compilation algorithm might take into account the time that it will take to carry out an initial test. Although the possible initial tests are chosen to be rapid to carry out, they may vary in just how rapid they are. It will generally take longer to test whether an input string contains the character “a” anywhere in the string than whether its first entry is “a”, and as a result it may in some circumstances it may be more efficient to try the second test than the first, even if the second test will not eliminate as many checks.

Another possible refinement is automatic recompilation. Recompilation of the decision tree might be set to occur at regular time intervals or after a fixed number of input strings; or might be automatically triggered when a particular measure passes a threshold value. The most straightforward way to measure the effectiveness of the decision tree over time is to measure the long-term average processing time for input strings. A rise in this average time may indicate that the input data sources have changed and that a recompilation is due. Or, a measure of the difference between the distribution of more recent input strings and the training set that was used in the last compilation might be calculated directly and updated on receipt of each new input string, triggering a recompilation if this measure passes a given threshold.
A slightly different technical problem is one in which we try to determine whether there is at least one pattern matched by a given string, instead of finding all the patterns matched by a given string. This could be useful, for example, in devices that look for signatures of malware in data and block that data if detected. In such a case, it may not matter which patterns match: they are all bad. So, finding at least one match is sufficient. Although this is a subtle difference in the problem statement, it could lead to very different kinds of solutions. In particular, it at least seems feasible that given n such patterns, a matching pattern could be found in $O(\log(n))$ steps.

We also have been giving some consideration as to whether it is possible to improve the choice of candidate initial tests.

We have implemented our solution to the extent necessary to carry out the evaluation experiments reported here, but not to a sufficient extent to be able to do performance testing. The next technical step for this work will be to create a fuller implementation, ready for performance testing.

As we said at the beginning of this paper, this work addresses a rather general problem with a wide variety of applications, and so we believe that our solution to this problem is of potential interest to a wide variety of businesses.

References


