Optimizing Unicode Regular Expression Evaluation with Previews

Howard Chivers[1]

Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK

SUMMARY

The *jsre* regular expression library was designed to provide fast matching of complex expressions over large input streams using user-selectable character encodings. An established design approach was used: a simulated non-deterministic automaton (NFA) implemented as a virtual machine, avoiding exponential cost functions in either space or time. A deterministic automaton (DFA) was chosen as a general dispatching mechanism for Unicode character classes and this also provided the opportunity to use compact DFAs in various optimization strategies. The result was the development of a regular expression *Preview* which provides a summary of all the matches possible from a given point in a regular expression in a form that can be implemented as a compact DFA and can be used to further improve the performance of the standard NFA simulation algorithm. This paper formally defines a preview and describes and evaluates several optimizations using this construct. They provide significant speed improvements accrued from fast scanning of anchor positions, avoiding retesting of repeated strings in unanchored searches, and efficient searching of multiple alternate expressions which in the case of keyword searching has a time complexity which is logarithmic in the number of words to be searched.

KEY WORDS: regular expression, finite automaton, non-deterministic, optimization, searching, forensics

1. INTRODUCTION

The work described here began with an apparently straightforward need: a regular expression matching capability that had a good time and space performance on complex expressions over large datasets, was Unicode compliant but allowed the user to decide what encodings are searched.

The initial application was the extraction of computer forensics artefacts from disks and files and Python was the language of choice for portability and compatibility with existing software. However, the regular expression libraries in Python and other common languages (e.g. Java, Perl, Ruby and the widely used PCRE library) are implemented using recursive backtracking, with exponential asymptotic execution times [1] that resulted in unpredictably long execution times for forensic pattern matching.

To meet the needs of this application a new regular expression library, *jsre* [2], was developed. As part of this development novel optimisations and enhancements were made to an established expression evaluation algorithm by Thompson [3]; these enhancements are documented here and are the primary contribution of this paper. A further contribution is the description of some novel aspects of the how the Thompson algorithm was implemented, including the use of loop counting to implement counted repetitions within an expression, and support for backreferences.

The design of a regular expression matching algorithm is a balance between execution time, space and functionality. The underlying theory is well established; a regular expression (RE) may be converted into a non-deterministic finite automata (NFA) which encodes the set of words denoted by the expression and is used to test for matching words in an input string [4]. Non-determinism is present because at any position in the input string there may be several prefixes that are valid choices, and several alternative suffixes that cannot yet be distinguished.

There are three standard approaches to implementing non-determinism for regular expression evaluation: backtracking, NFA simulation, and conversion to a deterministic finite automaton (DFA); each has a distinctive profile in terms of space and time complexity.

[1] E-mail: hrchivers@iee.org

The backtracking approach to NFAs is common in current regular expression libraries (e.g. Java, Python and Perl); however, given a finite regular expression R of length n and an input sequence S of length m, the time complexity of a backtracking implementation is $O(2^n)$ [5]. A common expression used to evaluate asymptotic time performance is `(a?){i}a{i}` for increasing values of i against an input string `a...a` of length i. This benchmark exposes the worst-case performance of backtracking algorithms because every possibility in the prefix `(a?){i}` is tested before the final match is achieved against the pattern `a{i}`. This catastrophic performance is not just a theoretical problem; the poor performance of standard language libraries on forensic applications was sufficient to motivate the work described here, and researchers have even speculated on the possibility that crafted input strings could be used as denial-of-service attacks against intrusion detection systems that use regular expressions to implement signature matching [6].

The NFA simulation approach was introduced by Thompson [3]. In essence the input string is processed one character at a time and at each position a list is maintained of all the positions in the expression with valid prefixes. The execution time is therefore related to the product of the input stream and the size of the expression (i.e. the maximum possible number of prefixes), resulting in a time complexity of O(nm) [7] with space complexity O(n).

The third alternative is to transform the NFA into a DFA; the standard approach uses the powerset construction introduced by Rabin and Scott[8] and since a DFA has only one transition possible at each step its time complexity is proportional to the length of the input stream: O(m). This performance is at the expense of the size of the state table needed to encode all valid prefixes at any given point in the input stream, resulting in an exponential space complexity of $O(2^n)$ [9]. Expressions that expose this space complexity have a form of `x[xy]{n}y` where `[xy]{n}` signifies n repetitions of either *x* or *y*; in essence, the DFA transition table has to record if an *x* occurred n characters before the final *y*.

In systems where asymptotic performance, as opposed to the widest possible range of features, is essential a balanced design approach is to use Thomson's NFA simulation. This guarantees that neither space nor time complexity is exponential, which in turn means that performance is less likely to be disproportionately sensitive to the expressions used in practice. jsre is therefore based on a NFA simulation; however, its practical performance is significantly enhanced by the use of the optimisations which are described here. In many existing regular expression libraries single byte look-ahead tests are used to avoid unproductive branches; previews generalise this idea and because of their generality can be used in a wider range of circumstances.

Since the contribution of this paper is the optimisation and implementation of the NFA simulation algorithm primarily, the approach to evaluation is to consider each of the preview optimisations separately and show how they contribute to regular expression performance. The performance of a regular expression library can be judged only in the context of a specific application; however, to provide an indication of the value of the NFA simulation approach the performance of jsre is also contrasted with the standard Python regular expression library *re*.

Where a test corpus is needed the public forensic corpus GovDocs is used as described in the sequel. All evaluation is carried out using a single CPU; however, compiled jsre instances are sufficiently portable and compact to distribute across many CPUs and are often deployed in this way.

The remainder of this paper is organised as follows. Section 2 introduces regular expressions, how they are interpreted as NFAs and their programmatic implementation. The overall design and asymptotic performance of jsre is described in section 3 which also describes novel implementation features specific to NFA simulation. Section 4 defines previews and explains their relation to the semantics of regular expressions, and this is followed in section 5 by a description and evaluation of optimisations that use the preview construct. The paper discusses related work in section 6, and is concluded in section 7.

2. REGULAR EXPRESSIONS AND AUTOMATA

This section introduces the semantics of regular expressions and describes how they are interpreted as non-deterministic finite automata (NFAs). It then describes the Thompson NFA simulation algorithm and how NFA simulation can be implemented as a program in a specialised language.

2.1. Regular Expression Semantics

The standard approach to regular expression semantics is to recursively develop the *set of words*, or *language*, denoted by the expression [4]. For a given alphabet Σ, a word is a sequence of symbols from Σ, or the empty sequence <ε>.

The base case is that for any symbol x in the alphabet Σ, `x` is a regular expression denoting the language {<x>}, the set containing a single sequence comprising the symbol x.

Three basic operations are defined on regular expressions: *concatenation*, *union* and *star,* which is also known as *closure* or *Kleene closure*. Consider that r and s are regular expressions denoting the languages R and S respectively. The concatenation of r and s, written `rs`, results in the language $\{xy \mid x \in R \wedge y \in S\}$ which is formed by concatenating every word in R with every word in S. Union, usually written `r|s`, results in the language $R \cup S$. Star, usually written `r*`, generates a language by concatenating zero or more words from R.

Other regular expression constructs may be defined in terms of these basic operations, for example `r+` ≡ `r(r*)` and the notation for a character set `[ab]`, where `a` and `b` are characters, is equivalent to `(a|b)`. `r?` signifies zero or one occurrences of the expression `r`. Table 1 provides examples of the language generated by different regular expression operators.

Table 1. Regular Expression Operations and the Resulting Regular Language

Regular Expression	*Words Matched in the Resulting Regular Language*
abc	abc
abc|de	abc, de
(de)*	ε (the empty string), de, dede, dededе, ...etc
[ab]	a, b
(de)?	ε (the empty string), de

2.2. Nondeterministic Finite Automata

A regular expression can be interpreted as a state transition graph [4] where each transition either consumes a character from the input stream, or changes state without consuming any characters, signified by the empty sequence ε. Each regular expression operation specifies what transitions are possible to following states, as shown in Figure 1.

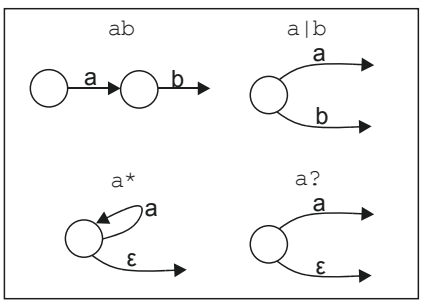

Figure 1. Transitions from Basic Operations

The closure operation may transition to the following state without consuming an input, and union results in two or more possible next states. This is the source of non-determinism; the following states are not determined uniquely by the input stream so far, so all possibilities must be explored. The three possible approaches to evaluation described above, backtracking, simulation and conversion to a deterministic automaton, are alternative approaches to exploring the states that result from this inherent non-determinism.

Figure 2 shows how these basic state transitions are combined to evaluate the expression `a*(a?ab|b)`. The states in this NFA are numbered for reference in the sequel. In this presentation the transitions following state 1 are shown separately and consume no input in order to show how basic operations can be combined directly into a NFA. The reader should not assume that the NFA transitions shown here proceed in lock step over the input string; an implementation may process a group of transitions at a time, such that different paths have reached different input positions at a given point in the simulation.

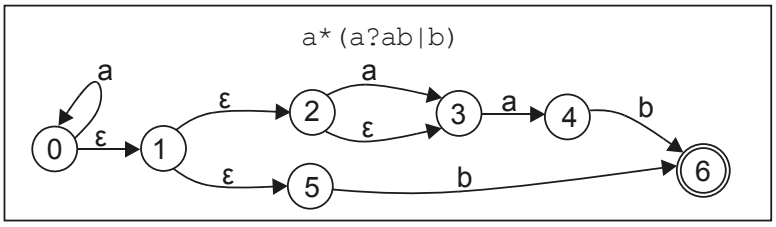

Figure 2. NFA for the expression `a*(a?ab|b)`

2.3. NFA Simulation

Thompson's NFA simulation algorithm [3] maintains a list of all valid possible states of the NFA at each character in the input stream. For example, using the states labelled in Figure 2 and the input 'aab', the following sequence of states is generated:

Input	*Possible Transitions consuming the Input Character*	*Resulting States*
a	0-0, 0-1-2-3, 0-1-2-3-4	0, 3, 4
a	0-0, 0-1-2-3, 0-1-2-3-4, 3-4	0, 3, 4
b	0-1-5-6, 4-6	6

The maximum number of resulting states processed by this algorithm is the product of the input length and the number of states in the NFA, hence the O(nm) complexity described above. The second and third steps of this example result in duplicate states - two ways of reaching the same position in the regular expression. Such duplicates are merged, since they will follow identical paths to completion. This state merging is why the algorithm has a good asymptotic performance, unlike recursive backtracking which would try every possibility sequentially.

From the perspective of the basic NFA simulation algorithm there is no distinction between two identical states at the same point in the input stream since any subsequent matching will be identical. In practice, however, if the history of the path through the NFA must be maintained to record the position of groups within the expression, when states are merged it may be necessary to prefer one history in preference to another. This is discussed further in section 3.5.

2.4. Programmatic Implementation

An NFA, such as the graph in Figure 2, can be evaluated by a specialised computer language in which the non-deterministic choices give rise to alternative execution threads, which are then evaluated in parallel. The basic operations in Figure 1 can be implemented by instructions that test the value of a character or create new threads.

Program execution continues until the final state is reached or there are no remaining threads because they have been eliminated by failed character tests. Execution is ordered to implement the NFA simulation, so threads that reference earlier positions in the input stream are executed first and threads that reference the same input position and result in the same NFA state are merged.

Figure 3 shows a program that implements the NFA given in Figure 2, together with the execution order that results from the input 'aab'. The instructions in this program are numbered to correspond with NFA state numbers in Figure 2.

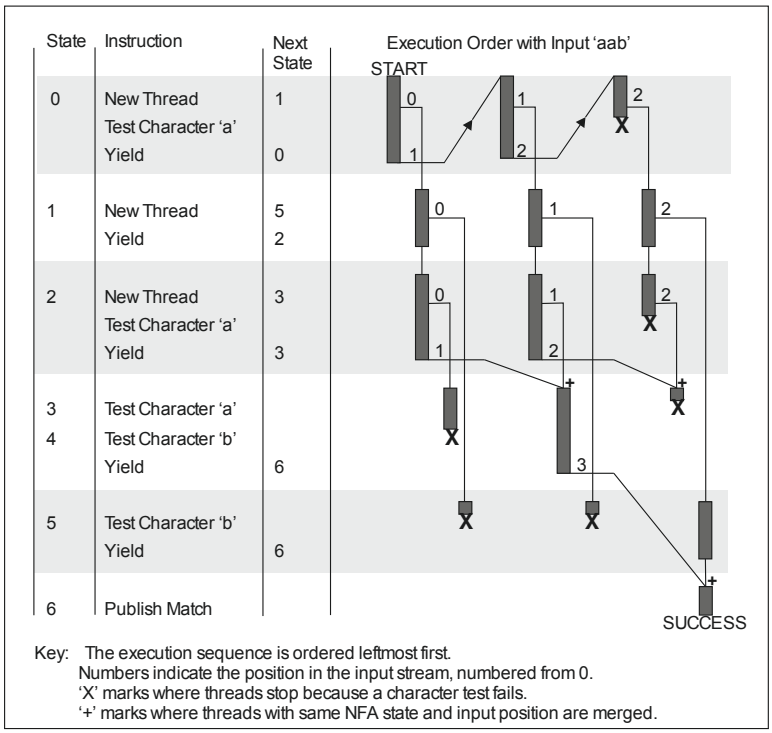

Figure 3. Implementation of the expression a*(a?ab|b) as a program showing the execution order for the input 'aab'.

Referring to Figure 3, the expression a* is implemented at state 0 with three instructions: a new thread is started to transition to the following state without consuming input, the character 'a' is tested and, if successful, transitions to the same state after yielding to threads that reference earlier positions in the input stream. For example, the first execution from state 0 creates a new thread to transition to state 1 with input position 0, then tests the character 'a' incrementing the input position associated with the current thread to 1. The yield instruction causes the execution of the current thread to wait until the processing of input position 0 has completed.

The three columns of execution shown left to right correspond to the three input characters, deferring the execution of threads that have advanced to the next input position; threads that arrive at the same state and input position are merged (for example, see state 3).

The example also shows an extension to the NFA simulation algorithm in the treatment of states 3 and 4. State 4 is entered only from state 3; there is therefore no possibility that a thread at state 3 can be merged with another thread before state 4 so there is no need to yield at this point. State 4 therefore follows state 3 directly, and this results in forward out-of-order testing of the input. Any test may result in failure and avoidance of further processing, so it is advantageous to test characters as early as possible where it is safe to do so. No additional threads result from this out of order testing because it is limited to deterministic state transitions.

In this example, if strict input order were maintained between states 3 and 4, on the first execution of state 3 the thread would yield, deferring processing of state 4 until the first input character had been

fully consumed. This optimisation saves a yield instruction in the NFA and a yield operation during execution.

3. IMPLEMENTATION AND PERFORMANCE

jsre uses a bespoke virtual machine (VM) which is implemented as a C extension to Python, providing the execution object which carries out expression matching. This section outlines the design of the VM, evaluates its asymptotic performance as a matching engine, and describes implementation details that modify or extend the simulated NFA algorithm.

3.1. Executable Architecture

As outlined above, the NFA simulation approach to RE matching steps through the input stream a symbol at a time maintaining a list of the expression positions which match at that point. This was straightforward in early implementations where symbols were single byte characters; modern symbols are multi-byte UNICODE encodings and so a matching algorithm requires a more general approach capable of matching variable length multi-byte characters and associated character classes.

The straightforward solution is to dispatch each character to a small DFA which implements the matching of individual characters or character classes. The design of character encodings guarantees that the DFA representing a single symbol is compact. The more complex DFAs result from character classes which have symbols in many different scripts; however, even these are constrained in size. For example, the representation of the UTF-8 encoding of 'alphabetic' requires only 267 states.

The NFA element of the matching engine is implemented as a bytecode virtual machine, resulting in the architecture shown in Figure 4.

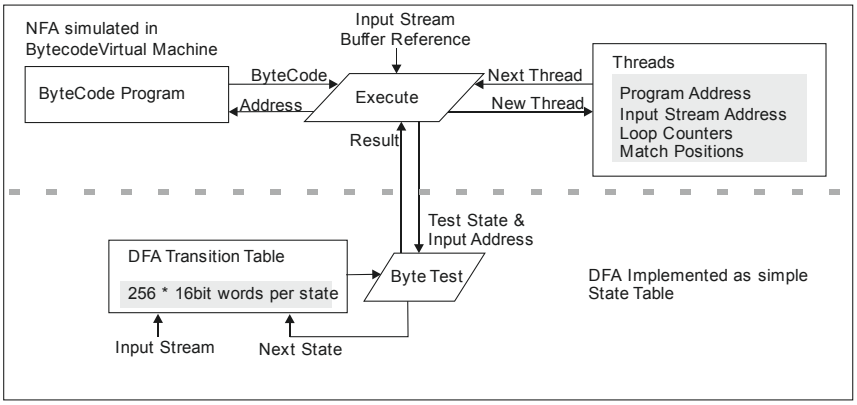

Figure 4. jsre Virtual Machine Architecture for Regular Expression Matching

Figure 4 shows the main data structures in the virtual machine and the machine architecture. The size of the memory image is dominated by the DFA table; the maximum memory footprint is limited to below 10MB (2^{14} states) which provides ample space for practical expression matching.

The NFA is simulated using a bytecode virtual machine using instructions similar to those described in section 2.4 above. Additional specialised instructions include the recording of the start and end of groups within a regular expression, loop counting, and testing characters before and after the current input position without consuming input.

The initial state of the machine is determined by an entry in a start list which includes the program address, the number of groups in the expression, and how the anchor (the start point for testing a match) will be moved through the input stream. Usually each entry in the start list will be searched in turn, allowing different encodings and other variants to be tested against the same input stream. This feature is not intended to replace alternative patterns within a regular expression, rather to allow major variants of encoding or anchor management to be evaluated without the overhead of returning to the calling program.

3.2. *Asymptotic Performance*

The timing results quoted in this paper were carried out in a single thread on a i7-3517Y laptop processor; a modern but not particularly highly specified CPU.

As noted in section 1 above, the expected complexity of a simulated NFA is O(nm) where n is the length of the expression and m is the length of the input stream. The time to match the benchmark `(a?){i}a{i}` described above should therefore be square law, since the length of the expression and the input stream grow at the same rate.

The measured performance of jsre against this benchmark is show in Figure 5.

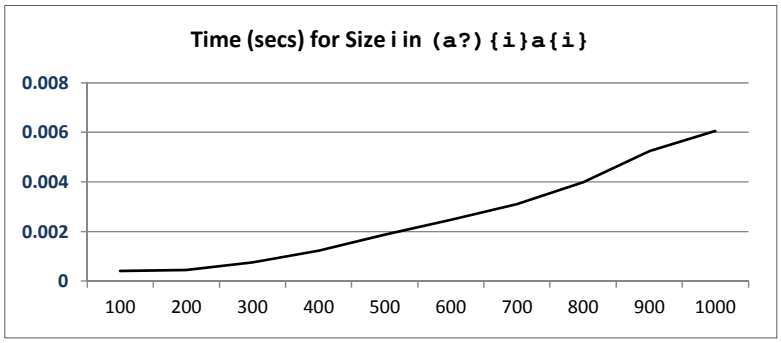

Figure 5. Performance against the Benchmark Expression `(a?){i}a{i}`

This curve appears to fall below the expected square law complexity. However, a polynomial fitted regression provides a close match ($R^2 = 0.9975$) with $t = 6.10^{-9}i^2 + 2.10^{-7}i + .0003$ confirming that the execution time is asymptotically square law. The two components of time are directly related to the expression; the first part `(a?){i}` is linear in *i*, however it is slow since it involves setting and managing *i* new threads. The final part `a{i}` is repeated for every thread generated by the first part, and is therefore responsible for the square law component of the time equation; however, it is fast because it performs character testing in the DFA. Very large values of *i* are therefore required before the square law is dominant.

Three distinctive operations that contribute to performance were characterised by adding each element in turn to a program with 10^8 repetitions and known performance; the results are given in Table 2.

Table 2. Execution time of basic operations.

Program Operation	*Measured time per operation (μsecs)*	*Equivalent rate (MHz)*
Test a single byte using the DFA state machine.	.0024	416
Execute a single bytecode instruction.	.0073	137
Create and store a new thread, retrieve it and begin execution.	.092	11

These figures support the intuitive understanding that to optimise performance in practice it is necessary to maximise the use of the DFA, and as far as possible avoid creating new threads. It is normal in regular expression compilers to place tests ahead of branches to determine if the first byte in the new thread can be matched in the input stream, avoiding the overhead if it is unlikely to be successful (see Henry Spencer's description in [10]) . The preview optimisations described below build on this idea, enabling effective use of a DFA without requiring a large state table.

The relative performance of NFA Simulation against backtracking is presented in Figure 6. The same benchmark is used, however, the size of the input is restricted to make the comparison feasible, and a logarithmic scale is necessary to show both together. As noted in the introduction, this benchmark exposes the asymptotic exponential time complexity of backtracking expressions.

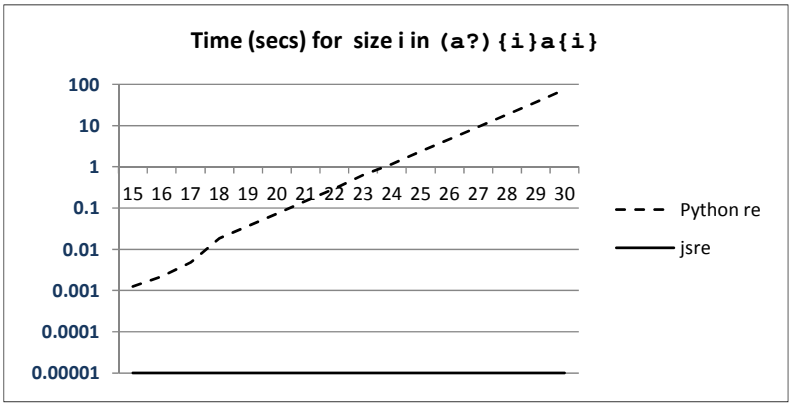

Figure 6. Comparative Performance of NFA Simulation (*jsre*) with backtracking (Python re)

Practical implementations of NFA simulation have to take account of features that have become standard in language libraries, including sub-group matching and backreferences. These and other extensions to the underlying NFA simulation are described briefly in the following sections.

3.3. Counted Repetitions and NFA State

The RE closure operator results in a repetition which may be repeated indefinitely. Regular expression syntax also includes counted repetitions where a given expression is repeated a fixed number of times; formally these are interpreted as concatenation, for example the expression `a{4}` is equivalent to the expression `aaaa`.

The expression compiler can interpret counted loops by simply expanding the repetition. Where a single character or a character set is repeated this is often the best approach, it occupies little space and executes as quickly as possible; however, the expression within the repetition may be an arbitrarily complex deeply nested expression requiring many repeats. To support these more complex cases jsre uses counted loops.

Code Example 1 shows a program fragment that illustrates the use of a counted loop to implement the start of a simplified expression for IP addresses: `([0-9]{1,3}\.){3}` This expression matches between one and three decimal digits followed by '.', all of which is repeated three times.

```
Program   Instruction                          Branch
Counter                                        Address
...
1         count_0 = 3
2         test char [0-9]
3         test char [0-9]: jump if fail        5
4         test char [0-9]: jump if fail        5
5         test char '.'
6         yield
7         decrement count_0: jump if > 0       2
...
```

Code Example 1. Program Fragment to Evaluate `([0-9]{1,3}\.){3}`

The code examples presented here provide descriptive instructions for the sake of readability, rather than actual bytecode.

The outer repeat of 3 times is implemented by setting counter 0 at statement 1, and the loop test at statement 7 which decrements the counter and branches to the start of the loop if the result is not zero. The character class [0-9] and the final character'.' are tested in order in the inner statements. The first test at statement 2 is compulsory, failure here will abort the match, while the tests at statements 3 and 4 are optional and both jump on failure to statement 5 which continues by testing '.'.

The optional repeats in the inner loop (statements 3 and 4) jump on failure to the loop end, rather than creating new threads before each test as would be expected. This optimisation is justified in section 5.5 below.

Counted loops are straightforward to implement, but introduce the problem of how to relate program instructions to NFA states. Consider the implementation of the expression `a{4}`. The formal interpretation of this is the concatenation `aaaa`, each of which would be a distinct NFA state. However, in a program loop the character 'a' would be tested at each repetition by the same instruction so the instruction address is not a valid proxy for an NFA state.

This distinction matters, because NFA simulation merges threads that have the same input position and NFA state, so to avoid merging threads that are not in the same state it is necessary to label states using both the program counter and the loop counter. Threads may be merged if they are at the same point in the input stream and both program counter and any loop counters have the same value.

The practical implementation in jsre allows a small number of loop counters which can be re-used at different parts of the expression; the number of counters place a limit on the number of nested counting loops that are supported by the implementation. The counters are always included in the NFA state, but are set to zero outside the scope of any program loops.

3.4. Backreferences and NFA states

A backreference is a common feature in modern expression languages, although it is not a regular feature in the sense that it can be constructed using the basic operations described above. For example in the expression `(abc?).*\1` the first group will match `ab` or `abc`, `.*` will match any characters and `\1` will then match whatever string was matched by group 1. Note that the backreference specifies the specific string matched by the numbered group, not just a repetition of the expression within that group; for example the above expression will match `abcxxab` but not match `abxxabc`.

Consider the expression `(abc?).*\1` and the input string `abcxxab`. When the third input character, c, is consumed there are two possible prefixes that match: `abc` and `ab.`, where '.' is the first repetition of the closure. Usually in NFA simulation the next character will result in the merging of these alternatives since they are at the same input position and NFA state (the closure). However, the backreference at the end of the expression may match either of the alternatives for the first group (`ab` or `abc`), so merging execution threads at this point is unsafe, it may discard the group which subsequently matches the input.

The solution in NFA simulation is to avoid merging threads if they contain groups which are later backreferenced, where those groups have matched different character sequences. Formally this could be regarded as creating several NFA simulations, one for each set of characters actually matched by a group which will later be backreferenced.

This solution allows backreferences to be supported naturally by an NFA simulation, at the expense of maintaining more parallel threads. The benefit in doing so is that it provides facilities usually supported only by recursive backtracking. Both simulation and backtracking will fail if a backreference expression results in a very large number of options for the groups that are referenced: backtracking will fail in time and simulation will fail by exhausting the space available for threads. At present jsre limits the total number of threads that can be allocated to 8192; this appears to be ample for important practical cases such as matching html tags but the optimum balance between size and performance here is still an open question.

To summarise, in this implementation NFA states are identified with program position, loop counter value and matched backreference group content. If two program threads match in all three components at the same input position then the NFA simulation algorithm will merge those threads.

3.5. *Groups and Thread Merging Policy*

Another standard feature in regular expression libraries is the ability to define groups within an expression whose matching sub-strings are reported. For example, the expression `the quick (.*) dog` would match the input `the quick brown dog` and also report the first group as matching `brown`.

In some expressions there may be several possible options for the groups reported against certain strings; in the event of a choice jsre returns the leftmost-longest groups, meaning that groups that match earliest have priority followed by those that are longest. For example the expression `(ab)c|(abc)` will match the input `abc` choosing to report the second match because it is the longer.

It is straightforward in the programmatic interpretation of an NFA to insert statements that record the start and end of groups and report these positions. However, threads that arrive at the same input position and NFA state may have different histories, and therefore have recorded different group matches. When such threads are merged it is necessary to decide which group history to preserve, and thus the choice of which thread to preserve is not arbitrary, it is subject to a thread policy which implements leftmost-longest group reporting.

4. PREVIEW DFAS

Previews are the formal construct used to implement the optimisations described in the next section. This section defines a preview and its relationship to the language denoted by a regular expression.

4.1. *Preview Definition*

The standard semantics for regular expressions are described in section 2.1. A preview provides a different interpretation of a regular expression, essentially an alternative language. A preview is a sequence of sets constructed in such a way that the i_{th} symbol in any word of the language normally defined by a regular expression is a member of the i_{th} set of the preview.

The formal notation here, in particular the definition of seq(), follows the definitions given in the Z mathematical toolkit [11]. A preview p: seq ($\mathbb{P}\ \Sigma$) corresponds to a regular language L: $\mathbb{P}$ seq (Σ) if:

$$\forall s: \text{seq}\ (\Sigma);\ n: \mathbb{N} \bullet s \in L \Rightarrow s(n) \in p(n)$$

For example, consider the regular expression `(ab|c)(d|e)` which would denote the regular language {<abd>,<abe>,<cd>,<ce>}. The preview for this expression would be the sequence <{a,c}{b,d,e}{d,e}>. This has the property that the sequence of symbols in any word denoted by the regular expression are members of the corresponding sets in the preview.

The reverse is not necessarily true; for example, the symbols of the word <cbe> can be found in the preview, but this is not a word in the regular language.

4.2. *Preview Construction*

The following examples will clarify how a preview is constructed by giving a new interpretation to the regular expression constructs of concatenation, union, and star. The software compiler used to generate a preview exactly follows this recursive structure in practice.

The base case is that if symbol x is in the alphabet Σ, then the preview of the regular expression `x` is <{x}>; a sequence containing a single set with the member x.

Consider the regular expressions `r = abcd|ef` and `s = ghi|j`, which denote the regular languages R = {<abcd>,<ef>} and S = {<ghi>,<j>}. The previews corresponding to these expressions are *Preview(*`r`*)* = <{a,e}{bf}{c}{d}> and *Preview(*`s`*)* = <{g,j}{h}{i}>.

The union operation results in a union of the corresponding sets within the preview. For example Preview(`r|s`*)* = <{a,e,g,j},{b,f,h}{c,i}{d}>.

It is necessary to record the minimum length of a word denoted by a preview (the *validity length*), since any input which matches this length may be a valid word in the corresponding regular language. For example, given the above definitions the shortest word denoted by Preview(`r`) is 2, and by Preview(`s`) is 1. The validity length also requires combining rules corresponding to the regular expression constructs. The validity length for the base case is 1, and for the union operator is the minimum of the validity lengths being combined.

When two regular expressions are concatenated their previews must take account of all the possible positions at which the concatenation may take place. The first position is given by the validity length. For example Preview(`rs`) is calculated as follows, the first concatenation position being the validity length of `r`, which is 2:

```
 <{a,e}{b,f} {c}   {d}>
             <{g,j} {h}  {i}>
                         <{g,j} {h}{i}>
=⟨{a,e}{b,f}{c,g,j}{d,h}{g,j,i}{h}{i}⟩
```

This preview can be contrasted with the regular language denoted by the expression `rs`, which is {⟨abcdghi⟩,⟨abcdj⟩,⟨efghi⟩,⟨efj⟩}.

The validity index of the resulting expression is the sum of the validity indexes of the concatenated previews (3 = 2 + 1). In this case the new validity index is 3 which correctly matches the length of the shortest word in the regular language.

The star operator follows the same pattern by concatenating a preview with itself an arbitrary number of times. Since this closure is also concatenated with the empty sequence <ε> the validity length of star is 0. In practical optimisation only the first few character sets in the preview are required and this resolves the difficulty that the size of a preview for the star operation is unbounded.

4.3. From Previews to DFAs

Constructing a DFA from a preview is essentially trivial. Each set in the preview is a character class and can therefore be encoded in the same way as other classes; the resulting sequence of DFAs is combined into a single automaton by redirecting the terminal states from each DFA to the root of the next. The size of the resulting DFA is therefore the sum of the sizes of the individual classes.

The individual character class DFAs have the same characteristics as those for user defined character classes; as discussed above, the worst cases occur when symbols are distributed across the UNICODE code space and even these cases occupy only a few hundred states.

Previews may therefore be compiled into compact DFAs which can be used to predict the words that may be denoted by a regular expression. There are no false negatives by construction, but there are false positives; a preview defines a superset of the words specified by the corresponding regular language.

5. OPTIMISATION WITH PREVIEWS

Previews are used in two general ways to improve expression evaluation performance:

- as an advance test to check if a more expensive operation is worthwhile, and
- to allow the compiler to determine if elements of a regular expression are disjoint in the sense that they cannot match the same character sequence.

The most basic use of a preview is to test if a match may be possible after a non-deterministic branch before incurring the overhead of launching the corresponding new thread. This test requires an extra bytecode instruction to implement the test. Section 3.2 provides relative times for the execution of a new thread as opposed to a byte code instruction of approximately 3:1, so provided such a preview test fails more than 33% of the time there will be a net speed benefit. The value of this approach depends on the application; in practice jsre always tests in this way because the cost of

calculating the preview in the compiler is low and previews of over 3 characters are effective in maintaining a positive benefit in a range of applications.

However, the performance advantage gained by this basic approach is relatively small; if the non-deterministic branch was always avoided and was the dominant cost in the expression the improvement in speed would be a factor of only 3.

This section describes optimisations that provide more significant improvements. The anchor scan in section 5.1 and multiple expression matching in section 5.3 use advance preview tests in a more efficient way; the anchor scan avoids the whole matching process in unanchored searches and multiple expression matching simultaneously tests many alternative sub-expressions.

The second use of previews, to allow the compiler to detect situations where non-determinism can be avoided, is described in its basic form in section 5.5. This is extended in sections 5.6 to 5.8 to use match failures within counted or uncounted repeats to eliminate unnecessary evaluation in unanchored searches.

5.1. Anchor Preview Scan

One performance problem with expressions used to search long input streams is that every position in the input stream has to be tested as a potential anchor, the position from which matching is attempted. One solution to anchor movement is to use a DFA preview as a primary test to find anchor positions that match the first few characters of the regular language and then use the simulated NFA to test for a match as usual. This approach has the advantage of exploiting the inherent speed advantage of the DFA to find candidate anchor positions.

The number of character sets required in the preview need to be chosen to ensure that the DFA speed dominates the anchor search. Choice of the size of the preview depends on assumptions about the application; in practice the assumption that the input stream is random is usually over-optimistic because biases within the input stream result in many more single character matches than would be expected at random. In practice therefore the preview needs to encode several characters to ensure that DFA performance dominates. jsre uses an anchor scan preview size of 3 characters which has proved effective in a range of applications. The next section shows the performance calculation in detail.

5.2. Anchor Preview Scan Performance

As noted above, testing against biased input data is more conservative than testing against random data. For this reason tests in this section were carried out against a public corpus biased towards text-based material. Garfinkel's Digital Corpora [12] includes a large collection known as GovDocs, many of which are text based, but which also include items with a more random character distribution, such as image files. The 'threads' packaging of this corpora provide a cross-section of document types and a test file was generated by combining thread0 and thread1 into a test input of approximately 1GB. As above, all tests were conducted using a single thread on a i7-3517Y laptop processor.

The anchor scan performance was tested using the expression: `([0-9]{1,3}\.){3} [0-9]{1,3}` which is a simplified expression to match IP addresses. The jsre virtual machine is able to provide a debugging trace which was used to profile execution; Code Example 2 profiles the start of the program which implements this expression with and without the preview scan.

The program without the scan instruction first marks the start of the match; this mark is automatically inserted by scan instruction so this instruction is not required in the program that uses the scan. In contrast with Code Example 1 the first three character tests are enumerated and not placed within a loop to avoid the instruction overhead of counter setting.

```
Profile   Profile  Program  Instruction                        Branch
without   with     Counter                                     Address
scan      scan
N/A       0.026    0        preview scan
1.0       N/A      0        mark-start-0
1.0       0.026    1        test char [0-9]
0.072     0.026    2        test char [0-9]: jump if fail      4
0.033     0.024    3        test char [0-9]: jump if fail      4
0.072     0.026    4        test char '.'
0.005     0.005    5        ...
...
```

Code Example 2. Execution Profiles with and without Preview Scan

The profile documents the number of times each instruction was executed as a proportion of the total number of bytes in the input stream. Without the preview scan the first character test is conducted against every input byte, succeeding in 0.072 cases: approx 1 in 14 of the bytes are decimal. Decimal characters are likely to be followed by similar characters, so the second test (line 2) succeeds nearly half the time. The final test against '.' reduces the number of positions that match `[0-9]{1,3}\.` to .005, or one in 200. The full expression matched 3173 IP addresses in the corpus.

The profile using a scan shows the same test using the preview DFA. Because the scan moves the anchor point the program is entered 1 byte in 38 (1/0.026). The preview is 3 symbols long, so the character tests at lines 1-3 do not significantly filter the match before the test of '.' at line 4 reduces the instruction rate to the same as the reference program.

Taking the time of a DFA test as unity, and the relative time of a NFA instruction as 3.04 from Table 2, the performance advantage of the preview scan can be predicted:

Instruction cost per byte without scan	= 2.182 (NFA instructions executed) * 3.04
	= 6.63
Instruction cost per byte with scan	= 1 (DFA) + 0.133 (NFA instructions) * 3.04
	= 1.40

This predicts a speed advantage due to the preview scan of 4.7 (6.63/1.4).

The measured execution times of 3.37 sec with preview scan and 16.43 seconds without, confirm that this improvement is achieved practice with a speed advantage resulting from the preview scan of 4.8. The small difference between prediction and practice is due to timing differences between different virtual machine instructions.

5.3. Matching Multiple Alternate Expressions

An important use case for regular expressions is matching several expressions in parallel by joining them into a single expression as alternatives; examples include searching simultaneously for emails and URLs, or where the expressions are keywords and an input stream needs to be scanned for a potentially large number of alternate words. This section describes how previews are used to optimise the matching of such expressions.

The elements of the optimisation strategy have already been described. Similar to the anchor scan a preview can be used to test if it is possible that a (sub) expression may match before the overhead of creating a new thread to test the alternative. The preview construction applies to an arbitrary expression, so this test can be applied to individual alternative expressions or groups of such alternatives.

It is therefore straightforward to arrange an efficient search strategy by re-ordering the alternative expressions into a tree and using preview tests at each node. The degree (number of branches) of each node is an empirical choice balancing depth of tree with the average number of tests required at each level; jsre uses a degree of 4.

5.4. *Matching Alternate Expressions - Keyword Performance*

Alternate expression optimisation applies to all expressions; however, its performance is most easily evaluated using random keywords.

A set of regular expressions were constructed by sampling keywords at random from the 10000 most common English words at least 7 characters long. Short words were avoided to prevent the measurement of matching time being dominated by match reporting. The sample was carried out without replacement; for example the test for 256 keywords contained the words in the 128 keyword test together with 128 additional words chosen at random. This procedure was adopted so that any observed non-monotonic behaviour would be due to expression evaluation rather than the structure of the test cases. The resulting expression was used in case-insensitive mode, again to provide the more demanding test.

The GovDocs test corpus described above was used as the target input; its bias toward textual content was expected to provide a strenuous test, and this was confirmed by the large number of results reported; the case insensitive test of 1024 keywords resulted in 443148 matches in the 1GB input sample.

The time taken to perform matching for various numbers of keywords is shown in Figure 7.

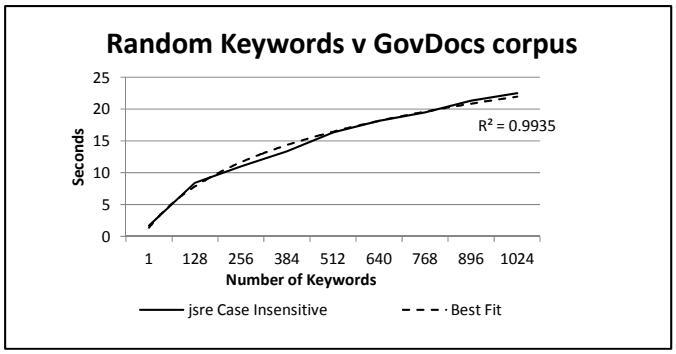

Figure 7. Scalability of Keyword Matching

The tree construction should ensure that the time taken to evaluate a set of keywords will scale logarithmically, and this is confirmed by the best fit regression line fitted to these results which is logarithmic with a coefficient of determination, R^2, of 0.993.

As remarked above, the test expressions were constructed to ensure that matches increase monotonically and this is confirmed by the number of matches reported. The variation above and below the logarithmic regression is a feature of the matching process, inflections corresponding to the addition of new layers in the tree.

Figure 8 contrasts this logarithmic performance with that of the standard Python re module, which is linear in the number of keywords tested.

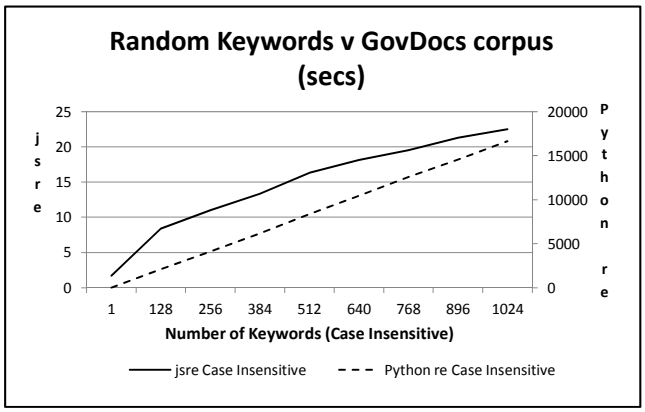

Figure 8. Comparative Performance of jsre with Python re on Keyword Matching

Figure 8 uses linear axes in order to contrast the complexity of these two approaches. However, note the difference of 800 in scale between the two vertical axes; there is a substantial performance difference between the two evaluation approaches. A significant element of the time incurred by the backtracking system in this example is in evaluating case differences, while the use of DFAs to evaluate symbols in jsre means that alternate symbols incur little additional overhead.

The performance of the tree construction against arbitrary expressions depends on the form of the individual (sub) expressions; expressions with relatively small preview sets will result in a similar asymptotic performance as demonstrated here with keywords.

5.5. Loop Optimisation

Code Example 1 used conditional jumps as opposed to new threads to implement optional repeats. The optional character tests in this example jump on failure to the end of the loop, as opposed to creating a new thread to evaluate each optional path.

This simplification is valid because the compiler is able to detect that the character set tested within the loop [0-9] is disjoint from the character set that immediately follows [\.].

In this case it is self evident that these sets are disjoint; however, in more complex expressions there may be an arbitrarily complex expression following a repeat. The first element of the preview of the expression following the repeat is the union of all characters that may match immediately after that repeat. A preview therefore provides the character set that is tested to check if the repeat is disjoint from its suffix. If the character set tested in the program loop is disjoint from the first set in the preview then the optimisation shown in Code Example 1 can be used. In terms of NFA states, the closure is modified as shown in Figure 9.

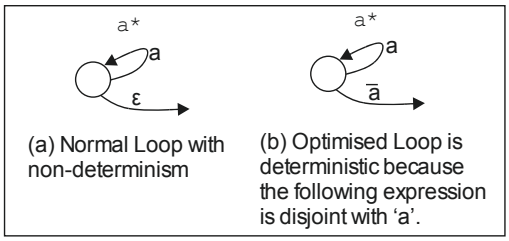

Figure 9. Loop Optimisation

5.6. String Prefix Optimisation

Efficient string search algorithms, such as the Knuth-Morris-Pratt search[13], exploit the information provided when a match attempt fails to determine the next anchor point from which to continue the search. The algorithm is effective since the start of the next search is moved as far forward as possible, rather than starting at the next character position.

String search algorithms have been applied to regular languages expressed as a DFA [14]; however, the concept of exploiting a prefix which is followed by a failure can also be applied in a simulated NFA.

Consider a closure of a character set [*x*] or the repeated test of a character set [*x*] with a disjoint suffix *y*: `[x]*y`. In both cases if the repeated character test stops and the suffix fails to match, then any subsequent anchor point that tests [x] in a position from which it has previously failed will also fail.

The proof is that in the case of a repeated character, starting the repeat later will reach the same failure, in the case of a closure the suffix will already have been tested at every point in the repeat and known to fail.

For example, consider the input string `aaaaxaaxy` and the regular expression `a*xy`. The first attempt to match fails on the sixth character:

```
Input:              aaaaxaaxy
Match attempt 1:    aaaax
```

Any anchor point chosen before the 'x' in the input stream will fail at the same position: the loop cannot match at this position and the suffix is known to fail starting here. Therefore the next position for the anchor is the symbol beyond 'x', from which the match succeeds.

```
Input:              aaaaxaaxy
Match attempt 2:         aaxy
```

The necessary disjoint property is established using a preview as described in section 5.5.

This algorithm could be applied directly to expressions that begin with repeated characters, resulting in optimal anchor movement for the whole regular expression. However, this would limit its usefulness because many practical expressions are the union of several sub-expressions. The alternative is to use this algorithm as a guard before any sub-expressions that begin with a repeated character set; this is the approach used by jsre.

This process requires two new instructions; a guard test which checks if the input pointer is in a guarded area and a character test which updates guarded areas if the character test fails. If the character at the current position p does not match the associated character class then this adjusts the guarded area so that any subsequent test of this guard will fail if it occurs before position p+1.

For example, the start of a program that matches the expression `a*x|b*y` is shown in Code Example 3.

```
Program   Instruction                              Branch
Counter                                            Address
0         preview scan
1         test guard 0; jump if guarded            4
2         test preview 0; jump if fail             4
3         new thread                               8
4         test guard 1: jump if guarded            7
5         test preview 1; jump if fail             7
6         new thread                               4e
7         halt
8         test char 'a'; if fail set guard 0
...
```

Code Example 3. Guarding Alternate Sub-Expressions.

Consider the first alternate sub-expression `a*x`. After the preview scan for the whole expression a guard is tested (instruction 1), if the anchor is within a guarded area then execution jumps to test preconditions for the second branch starting at instruction 4 and this sub-expression is not evaluated. Following the guard test a preview test is conducted for the sub-expression (instruction 2) and if the input passes this test then a new thread is started (instruction 4) to evaluate the sub-expression. Character tests in the sub-expression set the guard area on failure (instruction 8).

The same test pattern is used for the second sub-expression, `b*y` (instructions 4-6); a separate guard is associated with each alternate branch in an expression.

5.7. Basic String Prefix Optimisation Performance

String prefix optimisation provides performance improvements where symbols in a repeated character set prefix are common in the input string. One such example is an expression to match email addresses[2]:

[2] This is simplified; for example, it doesn't allow an IP address in place of the domain name.

```
[a-zA-Z0-9\-\.]{1,20}@[a-zA-Z0-9\-\.]{2,20}\.[a-zA-Z]{2,7}
```

The repeated character set at the start of this RE can be expected to match a large proportion of Latin text. Code Example 4 lists the start of the resulting program profiled with and without the loop guard; the input stream was the test corpus described above. Since there are no alternate branches there is only a single guard.

```
Profile  Profile  Program  Instruction                          Branch
without  with     Counter                                       Address
guard    guard
.096     .096     0    preview scan
N/A      .096     1    test guard 0; halt if guarded
.096     .033     2    test char [a-zA-Z0-9\-\.];
                       if fail set guard 0
                       ...
```

Code Example 4. Execution Profiles with and without Prefix Guard.

The code example shows only the first character test in the repeated expression. The guard test at line 1 implements the string prefix optimisation as described above, the match is abandoned if the input pointer is within the guarded area.

The immediate effect of the guard is to reduce the number of attempted matches by a factor of 2.9 (.096/.033). A similar performance prediction can be carried out as above, confirming a speed improvement observed in practice of 1.4 (11.7 seconds without string prefix, 8.4 seconds with).

This performance improvement is modest; however, the approach is able to make a more substantial contribution to the evaluation of unanchored searches in expressions that include closures.

5.8. Avoiding Quadratic Complexity

If an expression matches a large portion of the input stream, retrying the anchor point from every position in the stream results in an asymptotic execution time proportional to the square of the input size. One solution to this problem is to prefix an expression with `.*` (where '.' here signifies any character). This is able to exploit thread merging which is the basis of the efficiency of the NFA simulation algorithm to avoid the cost of completely retesting the expression from every position. The thread diagram in Figure 3 shows how threads originating in different positions of an initial repeat may later be merged (see state 3).

The disadvantage of this approach is that if the start of the expression is able to consume a large number of input characters in a linear search, the .* will result in a number of parallel threads equivalent to the number of input positions consumed. For example, an expression that begins `x{100}` (ie match 'x' exactly 100 times) is normally a linear search but when prefixed by `.*` will generate 100 parallel threads which are tested at every position in the input stream. At best thread handling may result in slower evaluation due to thread overheads, at worst this may result in early exhaustion of the available thread space.

For these reasons jsre leaves the decision to adopt this optimisation to the designer of the expression. However, the string prefix optimisation described above is able to achieve linear scans without this prefix in the common case where the non-determinism is introduced by closures embedded in the expression.

Consider the following regular expression, with correspondingly numbered NFA states, and the input string 'abcabczzzxyy':

```
Expression:                abc.*xyz
NFA State:                 0123 456
```

Attempting to match from the first anchor position results in state 3, the closure, failing at the end of the input; the suffix to this closure matches x and y (states 4 and 5) then fails at the final character. This results in a guarded area for the closure corresponding to all the positions matched by that repeat:

```
Input string:                 abcabczzzxyy
NFA states reached:           012333333333X
                    :                   45X
Guarded input for state 3:       GGGGGGGGG
```

When attempting to match from the second anchor position the anchor scan finds a possible match starting at the second 'abc'; when the match is attempted from this anchor, state 3 is reached within the guarded area, and since this is known to fail the evaluation from this anchor is halted.

```
Input string:                 abcabczzzxyy
Guarded input for state 3:       GGGGGGGGG
NFA states reached:            ..012X
```

This process is equivalent to thread merging in the NFA simulation; NFA states that lead to failure are recorded and when a match is started from a new anchor point new execution threads are merged into those that have already failed.

This expression can be used to explore how effective this process is in merging states, in particular if it avoids quadratic complexity. The expression is tested against an input string composed of n repeated copies of 'abc' followed by 'xyy'; the results are shown in Figure 10 for values of n between 1000 and 20000 and contrasted with those for the standard Python re package.

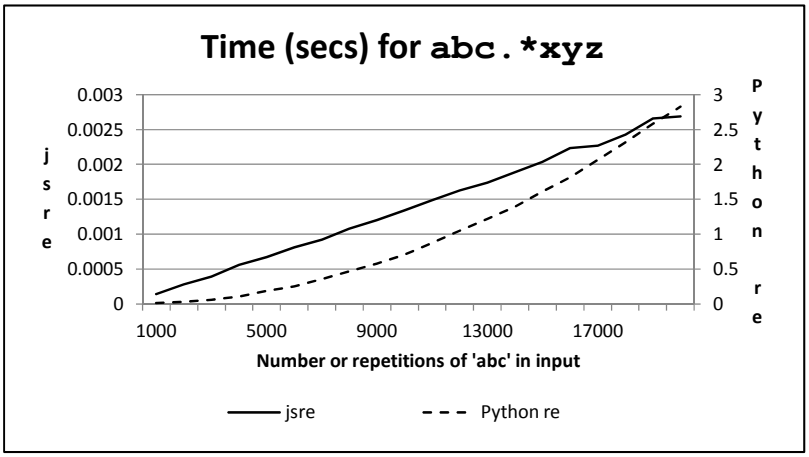

Figure 10. Performance of `abc.*xyz` for input of n (1000 to 20000) repeats of 'abc' followed by xyyy

Fitting a polynomial regression to these results confirms the expectation that jsre achieves linear performance ($R^2 = 0.998$) while the Python re is quadratic ($R^2 = 0.999$). Please note that these results are shown on linear axis to contrast their complexity; however, there is also a factor of 1000 difference in the two vertical axes.

6. RELATED WORK

The underlying theory of how to convert a regular expression into an automata and the complexity of the various implementation options is well established. The first implementation of a regular expression as a simulated NFA is due to Thompson [3], who gives an algorithm in IBM 7094 machine code; the construction of an equivalent DFA is due to Rabin and Scott [8]. Aho, Hopcroft and Ullman provide a proof that an NFA can be constructed to accept the same language as a given regular expression [4]. Their construction introduced the empty string ε to allow alternative paths in NFAs to be modelled as transitions that are not prompted by an input character. Following this construction directly introduces unnecessary states and is avoided by jsre in favour of the straightforward programmatic interpretation that non-determinism may allow more than one state to follow from a transition.

The space complexity of using DFAs to implement regular expression matching is a significant practical barrier to their use; for example Liu et al report that a DFA for the L7-filters requires over

16GB of memory, when the sum of the expressions taken individually consume only 9 MB [15]. Researchers have attempted to mitigate this problem by sparse matrix compression [16], or by exploiting similarities in transition patterns within the DFA. Kumar et al take the second approach, they observe that the transition tables following any given state tend to be similar by construction so compression is possible by recording a default state and its variations [17]. One consequence of this approach is that the resulting structure requires more than one transition per input symbol; this problem is bounded but not eliminated in subsequent work by Becchi and Crowley [18] and further developed by Ficara et al with a compression which achieves a single transition state per input symbol [19]. None of these approaches resolve the underlying problem of exponential space complexity; results reported by these authors typically report space compressions of 90-95%. Perhaps a motivation for the use of DFAs is to implement regular expression matching in FPGAs. In contrast one objective of the implementation described here is the efficient utilisation of the power of modern CPUs.

Basic architectural features of the jsre implementation follow what can be regarded as standard implementation approaches. Several researchers have proposed a hybrid architecture which uses DFAs at its inner level, with a table or NFA driven control structure [20, 21]. Evaluating regular expressions by compiling the expression into bytecode and execution in a virtual machine is also well established; many libraries are based on code originally written by Henry Spencer[10] including the backtracking algorithms in Python, Java, Ruby and Perl. Cox has described how to adapt this approach to a simulated NFA [22], and regular expressions may also be compiled directly into general purpose bytecode such as Java [23].

Look-ahead approaches have also been used in practice; an early optimisation was to search for anchor points by checking the first character, in this case one byte, in the expression [10]. The multi-character preview approach described here can be regarded as a principled generalisation of such approaches, is applicable to a much wider range of expressions and is more efficient in dealing with non-random input streams. A different look-ahead approach is implemented in Hyperscan (see Prefiltering at [24]), in which unsupported constructs such as backreferences are rewritten into weaker expressions. This is conceptually similar to a preview, but no attempt is made to use the construct except as a primary test; matches resulting from prefiltering must subsequently be confirmed by the application using a different regular expression library.

7. CONCLUSION

The jsre regular expression library provides fast matching of complex expressions over large input streams using many different character encodings. The chosen architecture avoids exponential cost functions in either space or time by applying a simulated NFA approach to expression matching.

In a Unicode environment it is necessary to dispatch multi-byte characters, and the use of DFAs to implement character matching allows the unification of characters and classes and provides an opportunity to develop the preview construct as a principled look-ahead of the possible suffixes that may be matched from a given point in an expression.

The preview proves to be an important general mechanism, with much better applicability than optimisations limited to single characters or bytes since it can easily be computed recursively for an expression of arbitrary complexity, and compiles into a compact DFA. This paper has introduced previews and highlighted some important optimisations that can be carried out using this mechanism; they include anchor position scanning, loop optimisation, avoiding retesting of repeated strings in unanchored searches, and the efficient evaluation of expressions that comprise a list of alternate sub-expressions.

8. ACKNOWLEDGEMENTS

I am grateful to the reviewers of this paper for their constructive and helpful feedback, and particularly to Russ Cox for his detailed and knowledgeable comments and suggestions.

9. REFERENCES

1. Cox, R., *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby,...).* URL: https://swtch.com/~rsc/regexp/regexp1.html, 2007.
2. Chivers, H. *Fast Regular Expression module for Forensics and Big Data*. 2015; Available from: https://pypi.python.org/pypi/jsre.
3. Thompson, K., *Programming techniques: Regular expression search algorithm.* Communications of the ACM, 1968. **11**(6): p. 419-422.
4. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974. **4**: p. 1.2-4.3.
5. Berglund, M., F. Drewes, and B. van der Merwe, *Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching.* arXiv preprint arXiv:1405.5599, 2014.
6. Kirrage, J., A. Rathnayake, and H. Thielecke, *Static analysis for regular expression denial-of-service attacks*, in *Network and System Security*. 2013, Springer. p. 135-148.
7. Sedgewick, R. and K. Wayne, *Algorithms*. Fourth ed. 2011, Boston: Pearson Education, Inc.
8. Rabin, M.O. and D. Scott, *Finite Automata and their Decision Problems.* IBM journal of research and development, 1959. **3**(2): p. 114-125.
9. Meyer, A. and M. Fischer, *Economy of Description by Automata, Grammars, and Formal Systems.* grammars, 1971. **1**: p. 10.
10. Schumacher, D., *Software solutions in C*. 1994: Academic Press Professional, Inc.
11. Spivey, J.M. and J. Abrial, *The Z notation*. 1992: Prentice Hall Hemel Hempstead.
12. Garfinkel, S., et al., *Bringing science to digital forensics with standardized forensic corpora.* digital investigation, 2009. **6**: p. S2-S11.
13. Knuth, D.E., J. Morris, James H, and V.R. Pratt, *Fast pattern matching in strings.* SIAM journal on computing, 1977. **6**(2): p. 323-350.
14. Watson, B.W. and R.E. Watson, *A Boyer–Moore-style algorithm for regular expression pattern matching.* Science of Computer Programming, 2003. **48**(2): p. 99-117.
15. Liu, T., et al. *An efficient regular expressions compression algorithm from a new perspective*. in *INFOCOM, 2011 Proceedings IEEE*. 2011: IEEE.
16. Jiang, L., J. Tan, and Q. Tang, *An efficient sparse matrix format for accelerating regular expression matching on field-programmable gate arrays.* Security and Communication Networks, 2015. **8**(1): p. 13-24.
17. Kumar, S., et al., *Algorithms to accelerate multiple regular expressions matching for deep packet inspection.* ACM SIGCOMM Computer Communication Review, 2006. **36**(4): p. 339-350.
18. Becchi, M. and P. Crowley. *An improved algorithm to accelerate regular expression evaluation*. in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. 2007: ACM.
19. Ficara, D., et al., *An improved DFA for fast regular expression matching.* ACM SIGCOMM Computer Communication Review, 2008. **38**(5): p. 29-40.
20. Myers, G., *A four russians algorithm for regular expression pattern matching.* Journal of the ACM (JACM), 1992. **39**(2): p. 432-448.
21. Becchi, M., C. Wiseman, and P. Crowley. *Evaluating regular expression matching engines on network and general purpose processors*. in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2009: ACM.
22. Cox, R., *Regular expression matching: the virtual machine approach.* URL: https://swtch.com/~rsc/regexp/regexp2.html, 2009.
23. Karakoidas, V. and D. Spinellis, *FIRE/J—optimizing regular expression searches with generative programming.* Software: Practice and Experience, 2008. **38**(6): p. 557-573.
24. Langdale, G. *Hyperscan Documentation*. 2015; Available from: http://01org.github.io/hyperscan/dev-reference/compilation.html.