A streaming full regular expression parser

Line Bie Pedersen 300976

Abstract

In this paper we present a design and a working prototype of a regular expression engine. It is able to match and extract the values of captured groups. The design splits the process into several components. Our components are streaming and use constant memory for a fixed regular expression, with the exception of one non-streaming component. We also evaluate the results and compare our regular expression engine with existing implementations.

Missing abstract

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Definitions, conventions and notation . . . 1
1.3 Objectives and limitations . . . 1
1.3.1 Limitations . . . 2
1.4 Summary of contributions . . . 2
1.5 Thesis overview . . . 2

2 Regular expressions and finite automatons **3**
2.1 Regular expressions . . . 3
2.2 Extensions to the regular expressions . . . 4
2.3 Finite automatons . . . 6
2.4 Regular expression to NFA . . . 6
2.4.1 Thompson . . . 6
2.5 Matching . . . 10
2.6 Summary . . . 11

3 Designing a memory efficient regular expression engine **12**
3.1 Architecture . . . 12
3.1.1 Constructing an NFA . . . 12
3.1.2 Dub and Feeley . . . 12
3.1.3 Bit-values and mixed bit-values . . . 13
3.1.4 After MBV . . . 13
3.1.5 Solutions . . . 14
3.2 Protocol specification . . . 15
3.3 Filters . . . 18
3.3.1 The 'match' filter . . . 18
3.3.2 The 'trace' filter . . . 18
3.3.3 The 'groupings' filter . . . 19

4 Implementing a regular expression engine **24**
4.0.4 Character classes . . . 24
4.1 Regular expression to NFA . . . 25
4.2 The simulator . . . 26
4.3 Filters . . . 28
4.3.1 Groupings . . . 28
4.3.2 Trace . . . 29
4.3.3 Serialize . . . 29

5 Optimizations **30**
5.1 Finding out where . . . 30
5.1.1 Memory usage . . . 30
5.1.2 Output sizes . . . 30
5.1.3 Runtimes . . . 31
5.1.4 Profiling . . . 31
5.2 Applying the knowledge gained . . . 34
5.2.1 ε-lookahead . . . 34
5.2.2 Improved protocol encoding . . . 34
5.2.3 Buffering input and output . . . 36
5.2.4 Channel management in `trace` . . . 37

6 Evaluation **38**
6.1 A backtracking worst-case . . . 38
6.1.1 Runtimes . . . 39
6.1.2 Memory usage . . . 40
6.2 A DFA worst-case . . . 41
6.2.1 Runtimes . . . 41
6.2.2 Memory usage . . . 43
6.3 Extracting an email-address . . . 44
6.3.1 Runtimes . . . 44
6.3.2 Memory usage . . . 44
6.4 Extracting a number . . . 47
6.4.1 Runtimes . . . 47
6.4.2 Memory usage . . . 48
6.5 Conclusion . . . 50

7 Related work **51**
7.1 Constructing NFAs . . . 51
7.2 Simulating NFAs . . . 51
7.2.1 Frisch and Cardelli . . . 51
7.2.2 Backtracking . . . 51
7.3 Virtual machine . . . 53

8 Future work **56**
8.1 Extending the current regular expression feature-set . . . 56
8.2 Internationalization . . . 57
8.3 More and better filters . . . 57
8.4 Concurrency . . . 57

A Test computer specifications **61**

B Huffman trees **61**

C Experiments **61**
C.1 Perls debug output 61
C.1.1 `regexmach.pl` 61

D Optimization scripts **61**
D.1 Memory usage 61
D.1.1 `memoryusage.pl` 61
D.2 Runtimes 64
D.2.1 `runtimes.pl` 64
D.3 Profiling 66
D.3.1 `profiling.pl` 66

E Benchmark scripts **67**
E.1 `backtrackingworstcase.pl` 67
E.2 `backtrackingworstcase_mem.pl` 69
E.3 `dfaworstcase.pl` 70
E.4 `dfaworstcase_mem.pl` 72
E.5 `email.pl` 73
E.6 `email_mem.pl` 76
E.7 `email_mbvsize.pl` 78
E.8 `number.pl` 79
E.9 `number_mem.pl` 81
E.10 `number_mbvsize.pl` 82

List of Figures

1 Fragment accepting a single character *a* 7
2 Fragment accepting the empty string 7
3 Alternation R|S 7
4 Concatenation RS 8
5 Repetition R* 8
6 Individual fragments when converting a*b|c to NFA 9
7 NFA for the regular expression a*b|aab 11
8 Architecture outline 15
9 Automaton with bitvalues for regular expression `a*` 16
10 Capturing under alternation 21
11 A simple character class-transition example 24
12 ε-cycles 27
13 Output of `gprof` running `main` 32
14 Output of `gprof` running `groupings_all` 32
15 Output of `gprof` running `trace` 33
16 Output of `gprof` running `serialize` 33
17 A backtracking worst-case: Main, Tcl and RE2 runtimes. . . 39
18 A backtracking worst-case: Perl runtime on a logarithmic scale. 40
19 A backtracking worst-case: Total, Tcl and RE2 memory usage. 40
20 A backtracking worst-case: Perl memory usage. 41
21 A DFA worst-case: Main, RE2 and Perl runtimes. 42
22 A DFA worst-case: Tcl runtimes. 42
23 A DFA worst-case: Main and RE2 memory usage. 43
24 A DFA worst-case: Perl memory usage. 43
25 A DFA worst-case: Tcl memory usage. 44
26 Extracting an email-address: Runtimes. 45
27 Extracting an email-address: Memory usage. 45
28 Extracting an email-address: Individual programs memory usage. 46
29 Extracting an email-address: Sizes of output from individual programs. 46
30 Extracting an email-address: The relationship between input size to `trace` and size of input string. 47
31 Extracting a number: Runtimes. 48
32 Extracting a number: Memory usage. 48
33 Extracting a number: Individual programs memory usage. . 49
34 Extracting a number: Sizes of output from individual programs. 49
35 Extracting a number: The relationship between input size to `trace` and size of input string. 50
36 NFA for a* 52
37 Huffman tree for frequencies in table 4, .* 62

38 Huffman tree for frequencies in table 4, (?:(?:(?:[a-zA-Z]+ ?)+[,.;:] ?)*..)* 63

List of Tables

1 Peak memory usage 30
2 Sizes of output 31
3 Runtimes 31
4 Frequencies of operators in a mixed bit-value string 35
5 Huffman encoding 35
6 Runtimes for different buffersizes 36
7 Runtimes for different buffersizes using non-thread-safe functions 37
8 Code sequences 54

List of definitions

1 Definition (Regular language) 3
2 Definition (Regular expression) 3
3 Definition (The groupings filter rewriting function) 23

List of examples

1 Example (Regular expression) 4
2 Example 5
3 Example (Converting a regular expression to a NFA) 8
4 Example (Matching with a NFA) 10
5 Example (Protocol) 16
6 Example 18
7 Example 19
8 Example (Simple groupings filter) 19
9 Example (Capturing under alternation) 20
9 Example (continuing from p. 20) 21
10 Example 23
11 Example (Backtracking) 52
12 Example (Virtual machine) 54

1 Introduction

This masters thesis presents a design and an implementation of a regular expression engine with focus on memory consumption.

Regular expressions is an important tool for matching strings of text. In many text editors and programming languages they provide a concise and flexible way of searching and manipulating text. They find their use in areas like data mining, spam detection, deep packet inspection and in the analysis of protein sequences, see [12].

1.1 Motivation

Regular expressions is a popular area in computer science and has seen much research. They are used extensively both in academia and in business. Many programming language offer regular expressions in some form, either as an embedded feature or as a stand alone library. There are many different flavors of regular expression and implementations, each adapted to some purpose.

Many of the existing solutions gives no guarantees on their memory consumption. In this project we will focus on a streaming solution, that is we will, where possible, use a constant amount of memory for a fixed regular expression. We will build a general framework to this purpose.

1.2 Definitions, conventions and notation

The empty string is denoted as ε. Σ is used to denote the alphabet, or set of symbols, used to write a string or a regular expression.

Automatons are represented as graphs, where states are nodes and transitions are edges. The start state has an arrow starting nowhere pointing to it. The accepting state is marked with double circles. Edges has an attached string, indicating on which input symbol this particular transition is allowed.

Regular expressions will be written in sans serif font: $\mathsf{a|b}$ and strings will be written slanted: *The cake is a lie*.

1.3 Objectives and limitations

The objectives of this thesis is to extend existing theory and design, implement and evaluate a prototype. We will be extending theory by Dubé and Feeley [6] and Henglein and Nielsen [8]. The extended theory will be used in designing a streaming regular expression engine. The design will be implemented in a prototype and finally we will evaluate and compare with existing solutions.

We aim to address these topics in this thesis:

- Extend existing theory by Dubé and Feeley [6] and Henglein and Nielsen [8].
- Create a prototype implementation
- Compare the prototype with existing solutions
- Conclude and propose extensions and improvements on the work

1.3.1 Limitations

The focus is on designing and implementing a streaming regular expression engine. There are many general purpose features and optimizations that can be considered necessary in a full-fledged regular expression engine that are only peripherally considered here. In situations where we are faced with a choice, we have generally favored simplicity and robustness.

1.4 Summary of contributions

The main contribution of this thesis work is a streaming regular expression engine based on Dubé and Feeley [6] and Henglein and Nielsen [8]. We present, implement and evaluate a working prototype that demonstrates that our solution is both technically viable and in many cases preferable from a resource-consumption standpoint compared to existing industry solutions.

1.5 Thesis overview

Section 2 gives an introduction to regular expressions and finite automatons.
Missing

2 Regular expressions and finite automatons

A regular language is a possibly infinite set of finite sequences of symbols from a finite alphabet.

A formal definition of regular languages:

Definition 1 (Regular language)**.** The regular language over the alphabet Σ is defined recursively as:

- The empty language $\emptyset$.
- The empty string language $\{\varepsilon\}$.
- The singleton language $\{a\}$, for any symbol $a \in \Sigma$.
- If L_r and L_s are both regular languages then the union $L_r \cup L_s$ is also a regular language.
- If L_r and L_s are both regular languages then the concatenation $L_r \bullet L_s$ is also a regular language.
- if L is a regular language then the Kleene star $L*$ is also a regular language.

*

2.1 Regular expressions

Regular expressions are used to denote regular languages. They are written in a formal language consisting of two types of characters: Meta characters and literal characters. The meta characters have special meaning and are interpreted by a regular expression engine. The basic meta characters are parenthesis, the alternation operator and the Kleene star. Parenthesis provides grouping, alternation allows the choice between different text strings and the Kleene star repeats. The literal characters have no special meaning and match literally.

A formal definition of regular expressions:

Definition 2 (Regular expression)**.** A regular expression over an alphabet Σ can be defined as follows:

- An empty string, ε, and any character from the alphabet Σ
- If r_1 and r_2 are regular expressions, then the concatenation $r_1 r_2$ are also a regular expression
- If r_1 and r_2 are regular expressions, then the alternation $r_1|r_2$ is also a regular expression

- If r is a regular expression, then so is the repetition $r*$

Any expression is a regular expression if it follows from a finite number of applications of the above rules. *

The precedence of the operators are: repetition, concatenation and alternation, from highest to lowest. Concatenation and alternation are both left-associative.

Example 1 (Regular expression)**.** Here we have a somewhat complicated example of a regular expression that demonstrates the basic operators. Consider the sentence:

This book was written using 100% recycled words.[1]

Other writings such as papers and novels also use words. If we want to catch sentences referring to these writings as well, we can use the regular expression: `(book|paper|novel)`.

To match the number 100 in the sentence, we could use the regular expression `100`. In most cases however we will not know beforehand how many words are recycled, so we may want to use the regular expression `(0|1|2|3|4|5|6|7|8|9)*`, which will match any natural number.

With this in mind we can write a regular expression to match our sentence:

```
This (book|paper|novel) was written using
(0|1|2|3|4|5|6|7|8|9)*% recycled words.
```

*

2.2 Extensions to the regular expressions

Many tools extend the regular expressions presented in the previous section. They add new notation to make it easier to specify patterns. In this section we present the extensions to definition 2 on the preceding page we have made: Extra quantifiers, character classes, a quoting character, a wild card and some non-capturing parenthesis.

- The quantifier $+$ causes the regular expression r to be matched one or more times. This can also be written as $rr*$
- The quantifier ? causes the regular expression r to be matched zero or one times. This can also be written as $\varepsilon|r$
- A character class is delimited by [] and matches exactly one character in the input string. Special characters loose their meaning inside a character class; $*$, $+$, ?, (,) and so on are treated as literals.

[1] Terry Pratchett, Wyrd Sisters

Characters can be listed individually, e.g. `[abc]`, or they can be listed as ranges with the range operator: $-$, e.g. `[a-z]`. These can be rewritten in terms of our original regular expression: `a|b|c` and `a|b|c...x|y|z` respectively.

To match characters not within the range, the complement operator is used. `^` used as the first character in a character class, elsewhere it will simply match literally, indicates that only characters not listed in the character class should match. E.g. `[^^]` will match anything but a `^`

- The quoting character \ will allow the operators to match literally. We use * to match a *.
- The wild card `.` will match any character, including a newline.
- For the non-capturing parenthesis we have the choice of notation. Here we will list some of the options, where r is some regular expression:
  - The industry standard, to which Perl, Python, RE2 and most others adhere is: $(?:r)$.
  - Perl 6 [15] suggests use of square parenthesis instead: $[r]$. These are however already in use by the character classes.
  - A more intuitive notation could be using single parenthesis for non-capturing, (r), and double parenthesis for capturing, $((r))$.
  - Since we are not using $\{r\}$ as special notation, this could be a good use, it surely would be the simplest to implement. This is however used in the repetition notation in industry standard.

  Since there is a standard, we will adhere to it, and use $(?:r)$ for non-capturing parenthesis.

Example 2. As we saw in example 1, we can match a natural number with the regular expression `(0|1|2|3|4|5|6|7|8|9)*`. Using the expansions to regular expressions above, we can rewrite this as:

`[0-9]*` This literally means the same thing.

`[0-9]+` We can use a different repetition operator and require there be at least one digit.

`[1-9][0-9]*` This matches any natural number as well, but it will not match any preceding zeros. This is a refinement, in that it will match fewer text strings than the first expression. This is however not always an advantage.

*

2.3 Finite automatons

Finite automatons are used to solve a wide array of problems. In this thesis we will focus on finite automatons as they are used with regular expressions. A finite state machine consists of a number of states and transitions between states. One state is marked as the initial state, a set of states is marked as final. To each transition there is attached a condition. Input is consumed in sequence, for each symbol transitions are taken when their attached condition are met. If the simulation ends in a final state, the finite automaton is said to accept the input.

Finite automatons can be divided in two categories: The deterministic (DFA) and the non-deterministic (NFA) finite automaton. This distinction is mostly relevant in practice, as they are equivalent. NFAs and DFAs recognize exactly the regular languages.

NFA For each pair of input symbol and state, there may be more than next states. This means that there may be more than one path through an NFA for an input string.

The ε-transitions are an extension of the NFA. These are special transitions that can be taken without consuming any input symbols. This also has mainly practical implications, NFAs with and without ε-transitions are equivalent.

DFA For each pair of input symbol and state, there may be only one next state. This means there is only one path through the DFA for an input string.

2.4 Regular expression to NFA

Every regular expressions can be converted to a NFA matching the same language.

2.4.1 Thompson

The method described in this section first appeared in Ken Thompsons article from 1968 [13]. The descriptions given in for example [9], [1] and [4] are considered more readable and we will be basing our description on these.

The NFA will be build in steps from smaller NFA fragments. A NFA fragment has an initial state, but no accepting state, instead it has one or more dangling edges leading nowhere (yet).

The base fragment corresponds to the regular expression consisting only of a single character a. The NFA fragment is shown in figure 1 on the next page. One state with a single edge, marked with the character a is added. The new state is the initial state for this fragment and the edge is left dangling.

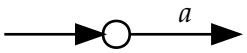


Figure 1: Fragment accepting a single character *a*

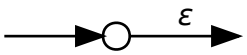


Figure 2: Fragment accepting the empty string

The second base fragment corresponds to the empty regular expression. The NFA fragment is shown in figure 2. One state with a single edge marked as a ε-edge is added. The new state is the initial state for this fragment and the edge is left dangling. This fragment is used for the empty regular expression and for alternations with one or more options left empty.

The first compound fragment is alternation, see figure 3. The two subfragments R and S are automatons with initial states and some dangling edges. What else they are composed of, is of no consequence. One new state is added, this is made the initial state for this fragment. The initial state has two ε-edges leaving, connecting to the initial states of R and S. The dangling edges for the new fragment is the sum of the dangling edges leaving R and S.

Concatenation of two regular expressions R and S is achieved as shown in figure 4 on the following page. The dangling edges of R is connected to the initial state of S. The initial state for the new fragment is the initial state of R and the dangling edges of S is still left dangling.

Zero or more times repetition is shown in figure 5 on the next page. One new, initial, state is added. It has two ε-edges leaving, one is connected to

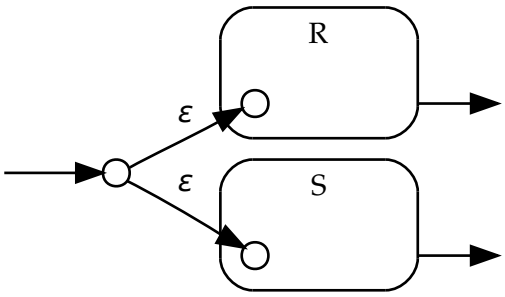


Figure 3: Alternation R|S

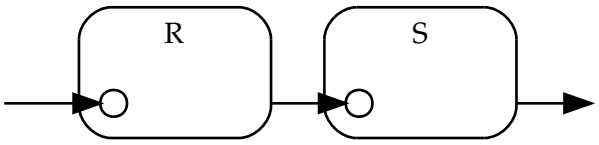


Figure 4: Concatenation RS

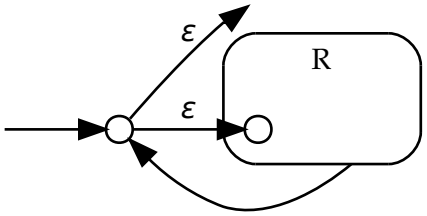


Figure 5: Repetition R*

the initial state of R and one is left dangling. The dangling edges of R is connected to the new initial state.

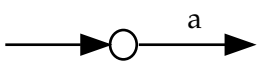
Finally an accepting state is patched into the NFA. All edges left dangling is connected to the accepting state.

Properties NFAs created with Thompsons method has these properties:

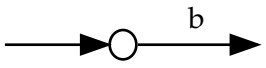
- At most two edges is leaving a state
- There are no edges leaving the accepting state
- There are no edges leading into the starting state

Example 3 (Converting a regular expression to a NFA)**.** In this example we will be converting the regular expression `a*b|c` to a NFA using Thompsons method.

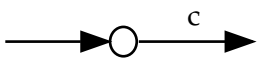
- Top level we have the alternation operator, but before we can complete this fragment, we need to convert `a*b` and `c` to fragments.
  - `a*b` is complicated since we have one operator, two literals and a hidden concatenation. Top level we have the concatenation operator, concatenating `a*` and `b`. These needs to be converted before we can concatenate.
    - `a*` needs to be broken further down. Top level we have the Kleene star, but we can not apply the rule for converting this to a NFA fragment before we have converted `a`.



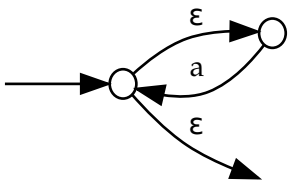
(a) Fragment for a



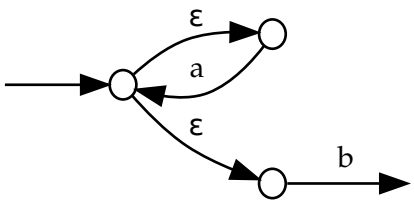
(b) Fragment for b



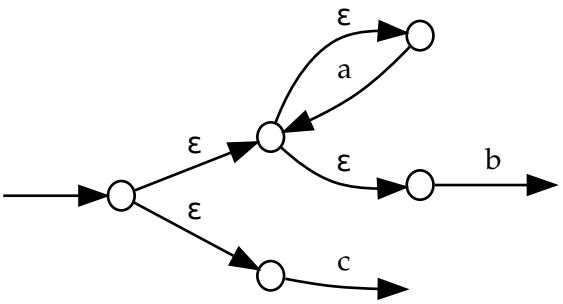
(c) Fragment for c



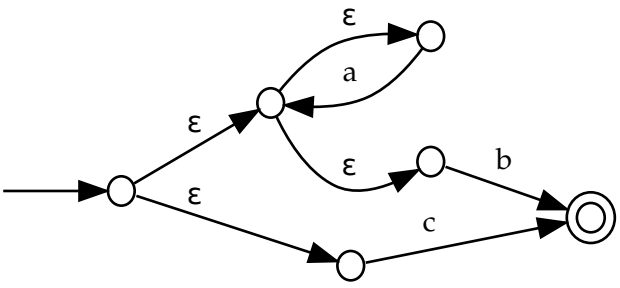
(d) Fragment for a*



(e) Fragment for a*b



(f) Fragment for a*b|c



(g) Final NFA for a*b|c

Figure 6: Individual fragments when converting a*b|c to NFA

· a is straightforward, we just apply the rule for transforming literals and we have the fragment in figure 6(a)

Using this fragment to complete the Kleene star, we have the fragment in figure 6(d).

∗ b is straightforward, we just apply the rule for transforming literals and we have the fragment in figure 6(b).

Now we are ready to concatenate, fragments 6(d) and 6(b) are concatenated and we have the fragment in figure 6(e)

– c is straightforward, we just apply the rule for transforming literals and we have the fragment in figure 6(c).

With these expressions converted to fragments we can apply the alternation conversion rule. We have the resulting fragment in figure 6(f)

All that is left now is to connect the dangling edges to an accepting state. We have the final result in figure 6(g) on the preceding page

∗

2.5 Matching

The NFAs constructed as described in section 2.4 on page 6 can be used to match a regular expression with a string, i.e. to decide if a string belongs to the language of the regular expression.

Once the NFA is generated, simulating it is a straightforward task. The method we will be using is attributed to Thompson [13]. We maintain a set of active states and a pointer to the current character in the string. At the beginning only the start state belongs to the set of active states. The string is read from left to right, taking each character in turn. When a character is read from the input string, all legal transitions from the states in the active set is followed. A transition is legal if it is a ε-transition or if the mark on the transition matches the character read from the input string. The new set of active states is the set of end states for the transitions followed. If the accepting state is included in the active set when the string is read, the string matches the regular expression.

With this method we only ever add a state to the active set once per iteration and we only read each character from the input string once.

Example 4 (Matching with a NFA)**.** In this example we will demonstrate how the regular expression a*b|aab is matched with the string *aab*. In figure 7 on the following page we have the corresponding NFA. Each state is marked with a unique number which we will be referring to in the table below.

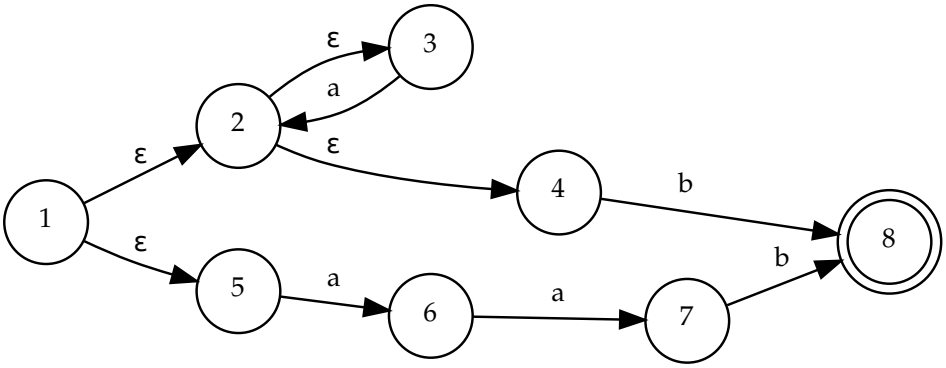


Figure 7: NFA for the regular expression a*b|aab

Active set	SP	Explanation
1	*a*ab	Initially we have the start state in the active set and SP points to the start of the string.
3, 4, 5	*a*ab	Following all ε-transitions.
2, 6	a*a*b	Reading the first *a* from the input string, states 3 and 5 have legal transitions on *a*.
3, 4, 6	a*a*b	Following all ε-transitions.
2, 7	aa*b*	Reading the second *a* from the input string, states 3 and 6 have legal transitions on *a*.
3, 4, 6	aa*b*	Following all ε-transitions.
8	aab_	Reading the last character from input string: *b*, states 4 and 7 have legal transitions on *b*.
8	aab_	No ε-transitions to follow.

After reading the string we can see that the accepting state is in the active set: We have a match!

*

2.6 Summary

Regular expressions are a widely used and popular tool. The features offered and the semantics vary. For example some will offer backreferencing and others will not, some will offer a leftmost match in alternations others will offer a longest match. The underlying implementation and performance vary widely. A regular expression engine will solve some problems more efficiently than others.

3 Designing a memory efficient regular expression engine

Missing introduction

3.1 Architecture

In this thesis we will build a general framework for matching regular expressions with strings. Our vision is a flexible architecture where the user is in control. Regular expression matching is a sequence of operations, where not all operations are needed at all times. This leads to the idea that we can split the regular expression engine into several dedicated parts. This can be demonstrated by considering the tasks of simple acceptance and extractions of groupings, the first only reports if a string matches a regular expression and the latter will also report on any groupings. By pulling this functionality out of the regular expression engine, we make the job of reporting simple acceptance simpler.

Before moving on, there are some prerequisites that must be discussed. This leads us to a discussion on possible mechanisms that would allow us to separate each task. We require several things: a mechanism to construct a NFA, a means of generate a syntax tree and a compact means of passing on the current state of the match process.

3.1.1 Constructing an NFA

In this thesis we have chosen to use Thompsons method of constructing NFAs. The NFAs constructed in this manner exhibit desirable properties: All states has no more than two outgoing transitions and the number of states grows linear in the size of the regular expression. Typically you would take this one step further and in some way build a DFA from the NFA, since these has much better traversal properties. We will not be doing this, the worst-case behavior of building a DFA is exponential both in time and space, as we will see in the evaluation section.

3.1.2 Dub and Feeley

One way of communicating the current state of the match process, would be to send the whole parse tree. An efficient algorithm for parsing with regular expressions is presented by Dub and Feeley in their paper from 2000 [6]. The algorithm produces a parse tree, describing how string w matches regular expression r. For a fixed regular expression the algorithm runs in time linear in the size of w.

To build the parse tree, we first construct a NFA corresponding to r. The article specifies a method for construction, but this can be any NFA constructed so that the number of states is linear in the length of r, this includes those constructed with Thompsons method from [13]. This restriction ensures the runtime complexity. Until this point there is no difference from a standard NFA, but Dub and Feeley then add strings to some of the edges. These strings are outputted whenever the associated edge is followed. When the outputted strings are then read in order they form a parse tree.

The idea of having output attached to edges is further developed in the paper [8]. The parse trees Dub and Feeleys method gives is rather verbose and can be more compactly represented: Whenever a node has more than one edge leaving, a string is added to the edges leaving, containing just enough information to decide which edge was taken.

3.1.3 Bit-values and mixed bit-values

Henglein and Nielsen introduce the notion of bit-values in [8]. A bit-value is a compact representation of how a string matches a regular expression. In itself it is just a string of `0`s and `1`s and has no meaning without the associated regular expression. The actual bit-value for a string is not unique and will depend on the choice of regular expression. If the regular expression is ambiguous and matches the string in more than one way, there will also be more than one bit-value for this string and regular expression.

If we rely on the properties of the NFAs generated by Thompsons method, that no state has more than two outgoing transitions, we have a perfect mapping for the bit-values. Instead of mapping syntax tree constructors to bit-values, we will map the outgoing transitions in split-states to bit-values. Each time we are faced with a choice when traversing the NFA, we will record that choice with a bit-value. This will enable us to recreate the exact path through the NFA. See also [6].

For reasons we will discuss later we will introduce the notion of mixed bit-values in this thesis. When simulating the NFA we will simultaneously be creating many bit-values which may or may not end up in an actual match. the individual bit-values will be referred to as a channel. Mixed bit-values are all these channels and they are simply a way of talking about multiple paths through the NFA.

3.1.4 After MBV

We have now introduced the bit-values. The bit-values enables us to split up the work in several tasks.

- The first task will be to create the mixed bit-values describing the paths the Thompson matching algorithm takes through the NFA. The

first task will need the regular expression to form the NFA and the string for the matching. Note that there is no need to store the whole string, the matching processes the characters in the input string in a streaming fashion.

- The next and also last step in a simple acceptance match, would be to check the mixed bit-values for a match. Simply scan the bit-values for acceptance.
- In extracting the values of groupings, we would need more tasks. We could form a task that cuts away unneeded parts of the parse tree. Only the parts concerned with contents of the groupings would be needed to actually extract the values. To do this we would require the regular expression to form an NFA annotated so that we could recognize the relevant parts of the syntax tree.
- We have a stream of mixed bit-values. It would be necessary at some point to extract the channel that makes up the actual match, if there is one. This can not be done in a streaming fashion. When first encountering a new channel, we need to know whether or not it has a match. The only way to know this is to read the whole stream of mixed bit-values. This task would only need the stream of mixed bit-values, it has no need for the regular expression.
- The last step in extracting the values of the groupings would be to output the actual values in some format. To do this we would require the bit-values from the match and the regular expression. The regular expression is so that we can form an NFA annotated with the positions of the groupings.

3.1.5 Solutions

There are two main methods of realizing this design. We can make the tasks be small separate programs that communicate through pipes or we can make one program where the tasks will be processes that communicate through some inter-process communication model. The separate programs model has the advantage of being simpler, in that the communication framework is already in place, we would not have to worry about synchronization and such. The processes model would probably have the advantage of being much faster in communicating and a generally lower overhead, all according to which model for inter-process communication was chosen. We have chosen the separate programs model, because of the ease with which you can combine the separate programs and the much simpler communication model. This also opens up for the possibility to store the output from one task for later use, or perhaps even piping the output to a completely different system with for example `netcat`.

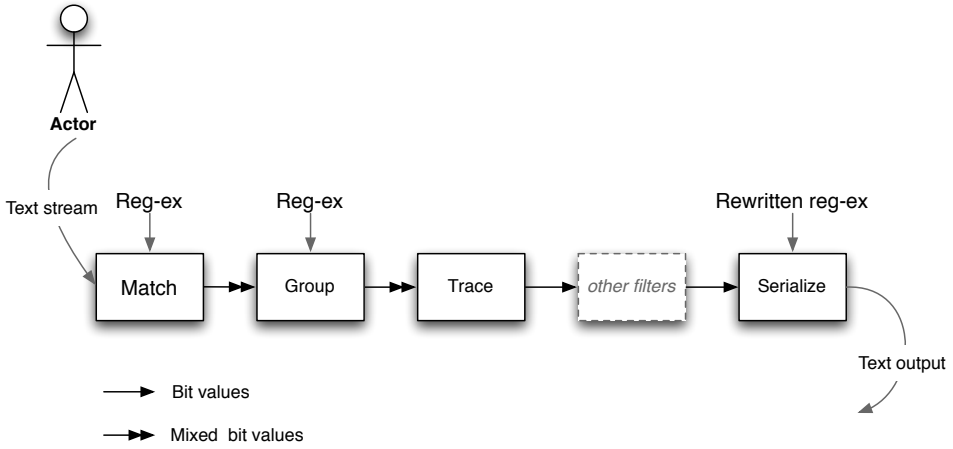


Figure 8: Architecture outline

The tasks will in some sense be projections performed on the mixed bit-values and the bit-values. The programs will therefore be called filters.

We present the overall architecture in figure 8. In the first program we have the matcher, this program will take the regular expression as an argument and have the string piped in and will output the mixed bit-values that comprises the match. The second program will take the mixed bit-values from the first and filter out those mixed bit-values relevant to the capturing groupings only. The third program takes mixed bit-values and filters out the bit-values relevant to the actual match, if there is no match, the output will be the empty string. The fourth program takes bit-values and constructs the string that was matched with those bit-values. If you rewrite the regular expression so that it only consists of the capturing groups and adjust your bit-values accordingly, this will result in the capturing groups being outputted. We have put in a fifth program in the design to signal that you could have more filters and place them anywhere in the chain they make sense.

3.2 Protocol specification

In this section we will define a protocol that can communicate information between our programs. The information consists of the mixed bit-values generated by the NFA simulator and the filters. The protocol should enable us to recreate paths taken through an NFA. To this purpose we need the protocol to support the following operators:

| The end of the channel list is reached and we should set the active channel to the first channel. This coincides with reading a new character. It is not a strictly necessary operator, we can make do with the change

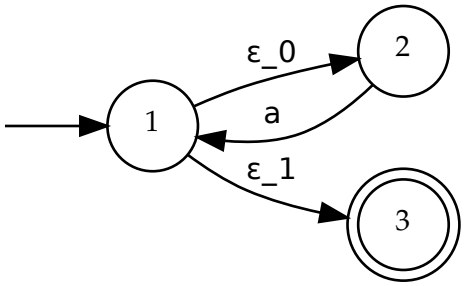


Figure 9: Automaton with bitvalues for regular expression `a*`

channels action. We choose to keep a separate action for end of list, because it adds to readability and redundancy.

: Whenever we change channels we put a :. There may be more than one or perhaps even no bits output on a channel for any given character from the string

= Copying of a channel. One channel is split into two, the paths taken through the NFA will be identical up to the point of splitting. The newly created channel is put in front of the rest of the channels

0,1, $\backslash a$ The actual bit values. The character classes needs extra care, we need to know on which character we passed them, so we put the value we pass on, escaped in the output

b A channel is abandoned with no match

t A channel has a match

Example 5 (Protocol)**.** In figure 9 we have an automaton for regular expression `a*`. When matching this regular expression with the string *aa* we generate some mixed bit-values. This example will in detail demonstrate how the mixed bit-values are generated.

Initial step: Initially the start state of the automaton is added to the active list. All ε-edges are followed and the following is output:

1. Node 1 is a split-node, a = is output, and we follow the ε-edge to node 2 and output a `0`. We can not make any further progress on this channel. We output a **:** and switch to the next channel.
   Output so far: `=0:`.
   List of active channels: $\{2, 1\}$.
2. The active channel is now in node 1, we follow the ε-edge from node 1 to 3 and output a `1`. We can not make any further progress

on this channel. This is the last channel in the channel list, so we output a `|` and reset the active channel.
Output so far: `=0:1|`.
List of active channels: $\{2, 3\}$.

First *a* is read

1. Node 2 has a transition marked `a`, we follow this back to node 1. Node 1 is a split-node, a = is output, and we follow the ε-edge to node 2 and output a `0`. We can not make any further progress on this channel. We output a `:` and switch to the next channel.
   Output so far: `=0:1|=0:`.
   List of active channels: $\{2, 1, 3\}$.
2. The active channel is now in node 1, we follow the ε-edge from node 1 to 3 and output a `1`. We can not make any further progress on this channel. We output a `:` and switch to the next channel.
   Output so far: `=0:1|=0:1:`.
   List of active channels: $\{2, 3, 3\}$.
3. Node 3 is the accepting node and does not have any transitions. We abandon this channel and output a `b`. This is the last channel in the channel list, so we output a `|` and reset the active channel.
   Output so far: `=0:1|=0:1:b|`.
   List of active channels: $\{2, 3\}$.

Second *a* is read This is the final step.

1. From node 2 we can make a transition on `a` back to node 1. This is a split node, so we output a `=` and transition on the the ε-edge to node 2 and output a `0`. We can not do further transitions and this is not the accepting node, we abandon this channel and output a `b`. We switch to the next channel and output a `:`.
   Output so far: `=0:1|=0:1:b|=0b:`.
   List of active channels: $\{1, 3\}$.
2. Node 1 has a ε-transition to node 3, we take it and output a `1`. We can not do further transitions and since this is the accepting node, we output a `t`. We have one channel left, so we output a `:` and switch.
   Output so far: `=0:1|=0:1:b|=0b:1t:`.
   List of active channels: $\{3\}$.
3. Node 3 has no available transitions. We abandon this channel and output a `b`.
   Output so far: `=0:1|=0:1:b|=0b:1t:b`.
   List of active channels: $\{\}$.

*

3.3 Filters

We have now established that filters are stand-alone programs that takes input, performs some projection and outputs the result. In this subsection we will give a more detailed description of the filters developed for this thesis.

The filters can be combined to make a whole, naturally some orders of combining makes more sense than others. For example will it make sense to have filters removing unnecessary information as early as possible, to reduce the input data-sizes for downstream filters.

3.3.1 The 'match' filter

Input Any mixed bit-values or bit-values

Output A single value indicating match or no match.

This is a simple filter. The input is scanned for a `t` control character, if present we output a `t` otherwise we output a `b`. In the case of empty input, we will output an error message, this is because the empty input is most likely due to an error in the previous programs. To save time on processing, we will assume the input format is correct.

Example 6. The regular expression a* matches the string *aaa*:

```
$ echo -n 'aaa' | ./main 'a*' | ./ismatch
t
```

a* does *not* match *bbb*:

```
$ echo -n 'bbb' | ./main  'a*' | ./ismatch
b
```

Since we do not check the correctness of the input, the sentence: "the cake is a lie" which is clearly not in the correct input format with regards to the protocol defined section 3.2 on page 15, will also produce a positive answer from the filter:

```
$ echo -n 'the cake is a lie' | ./ismatch
t
```

*

3.3.2 The 'trace' filter

Input Mixed bit-values

Output Bit-values

The mixed bit-values is a way of keeping track of multiple paths through the NFA. This filter will remove all channels from the mixed bit-values, except the one that has a match. We are using Thompsons method for matching, so we can be sure there is at most one channel with a match.

This will be a non-streaming filter. This problem can not be solved without in some way storing the mixed bit-values: We need knowledge of whether or not a channel has a match at the beginning, but we will not have that knowledge until the end.

Example 7. In the previous example we saw that the regular expression a* matches the string *aaa*. The NFA for the regular expression is in figure 9 on page 16, marked with state numbers and bit-values. This particular match will generate the following mixed bit-values: `=0:1|=0:1:b|=0:1:b|=0b:1t:b`. The filter should then only return the bit-values `0001`, which represents the match. The filter should return the empty string if there is no match. ∗

3.3.3 The 'groupings' filter

Input Mixed bit-values

Output Mixed bit-values for rewritten regular expression

This filter facilitates reporting the content of captured groups. The filter outputs the mixed bit-values associated with the groupings. By this we mean that all mixed bit-values generated while inside a captured group should be sent to output and all mixed bit-values generated outside a group should be thrown away. By throwing away the unnecessary bit-values we hope to make the mixed bit-values sequence shorter. This will be an advantage when the time comes to apply the trace filter, which is non-streaming, described in section 3.3.2 on the previous page.

Example 8 (Simple groupings filter)**.** Here we have a few simple examples of what the groupings filter should do.

- For regular expression (a|b) matched with *a* the mixed bit-values are `=0:1|t:b`. Since the whole regular expression is contained in a capturing parenthesis, nothing should be thrown away. Output should contain `=0:1|t:b`.
- For regular expression (?:a|b)(c|d) matched with *ac* the mixed bit-values are `=0:1|=0:1:b|t:b`. This time the first part of the regular expression is contained only in a non-capturing parenthesis and the associated bit-values should be thrown away. We want to keep only

the bit-values from the second alternation. Output should contain `=:|=0:1:b|t:b`.

In this example we have only dealt with simple examples. Regular expressions containing parenthesis under alternation and repetition, e.g. (a)|b and (a)*, require extra care and will be discussed later. ∗

The output of the groupings filter can be used to navigate the NFA for the regular expression altered in a similar manner: Everything not in a capturing parenthesis is thrown away. From example 8 on the preceding page we have the regular expression (?:a|b)(c|d), if we throw away everything not in a capturing parenthesis we have left (c|d). Stated in a more formal manner, we can define our first naive rewriting function G':

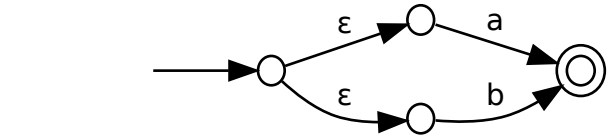
$$\begin{aligned} G'[[\varepsilon]] &= \varepsilon \\ G'[[a]] &= \varepsilon \\ G'[[[...]]] &= \varepsilon \\ G'[[r'r_2]] &= G'[[r']]G'[[r_2]] \\ G'[[r'|r_2]] &= G'[[r']]G'[[r_2]] \\ G'[[r*]] &= G'[[r]] \\ G'[[r+]] &= G'[[r]] \\ G'[[r?]] &= G'[[r]] \\ G'[[(? : r)]] &= G'[[r]] \\ G'[[(r)]] &= (r) \end{aligned} \tag{1}$$

Capturing under alternation As is seen, G' basically throws away anything not in a capturing parenthesis. There are however a few problems with this definition, as hinted earlier. Our first problem is regular expression with a capturing parenthesis under alternation. When the capturing parenthesis is under the alternation and we throw away the alternation, we lose a vital choice: There is no longer a way to signal whether or not a group participates in a match.

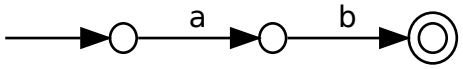
Example 9 (Capturing under alternation)**.** In matching the regular expression (a)|(b), see figure 10(a) on the next page for the NFA, with the string *a* we obtain these mixed bit-values:

`=0:1|t:b`

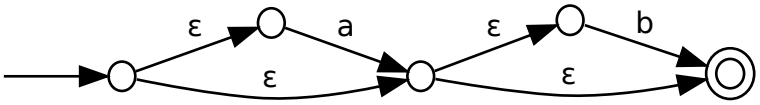
What these mixed bit-values are saying is that we have 2 channels, one that go through a and succeeds and one that go through b and fails. The succeeding channel never goes through b, the contents of that group is not defined.



(a) (a)|(b)



(b) (a)(b)



(c) (?:|(a))(?:|(b))

Figure 10: Capturing under alternation

Rewriting the regular expression (a)|(b) according to G' we have:

$$\begin{aligned} G'[[\text{(a)|(b)}]] &= G'[[\text{(a)}]]G'[[\text{(b)}]] \\ &= \text{(a)(b)} \end{aligned}$$

In this regular expression there is only one way: The one going through both the groups. See figure 10(b) for the NFA of the expression. This is bad news for our rewriting function and our filter, since we need some way of skipping groups: Each channel goes through only one group. $*$

In example 9 we saw an example of how undefined groups are not handled. To solve this problem we need some way of signaling if a group participates in a match or not. We define a new rewriting function G'' it is identical to G' except for equation 1 which is changed to:

$$G''[[(r)]] = (? : |(r))$$

This change will enable us to choose which groups participates in a match. This comes at a cost: Extra bits will have to be added to the mixed bit-values output and extra alternations to the rewritten regular expression.

Example 9 (continuing from p. 20)**.** With the changed equation 1 we can continue our example from before. Again we rewrite regular expression

(a)|(b), this time according to G'':

$$\begin{aligned} G''[[\text{(a)|(b)}]] &= G''[[\text{(a)}]]G''[[\text{(b)}]] \\ &= \text{(?:|(a))(?:|(b))} \end{aligned}$$

See figure 10(c) on the previous page for the NFA. As is clear from the rewritten regular expression and the NFA, there is now a way around the groups. Taking this into account, the output for the groupings filter should be:

```
=1:01|0t:b
```

What these mixed bit-values are saying is that we have two channels, one picks the route through a, around b and succeeds and the other picks the route around a, through b and fails. *

As needed, we now have a way of signaling if a particular group is in a match: Insert a 1 in the mixed bit-values and the group participates or insert a 0 and it does not.

Capturing under repetition The other problem we hinted at has to do with capturing under repetition. When using a capturing subpattern, it can match repeatedly using a quantifier. For example matching (.)* with the string *abc*, the first time we apply the * we capture a *a* the second time a *b* and the last time a *c*. In such a case we have several options when reporting the strings that was captured:

- The first
- The last, this is the what most backtracking engines like Perl do
- All, this is what a full regular expression engine do

Only two of these options are available to a streaming filter: All and the first. In order to return the last match, we would have to save the latest match when matching with the quantifier, it is potentially the last and we can not know until we are done matching with the quantifier.

Returning the first string that was captured by the quantifier, forces us to throw away mixed bit-values generated in a capturing parenthesis. We would only need the mixed bit-values generated by the first iteration of the quantifier.

To return all the strings captured by a group, we simply output all the mixed bit-values generated while in the capturing parenthesis. However, this causes problems with the rewriting function. Rewriting (.)* according to G'' we have (.). This regular expression accepts one single character.

In no way can we make mixed bit-values, fitting this regular expression, that represent a list of matched strings. Therefore we add the following equations:

$$G''[[(r)*]] = (r)*$$
$$G''[[(r)+]] = (r)+$$

We should now also keep the mixed bit-values that glues the iterations together, even though they are outside the capturing group.

We are now ready to present the final rewriting function: Definition 3.

Definition 3 (The groupings filter rewriting function)**.** For regular expressions r, r_1, r_2, defined over alphabet Σ, and a, any character from Σ, let G be defined by:

$$\begin{aligned}
G[[\varepsilon]] &= \varepsilon \\
G[[a]] &= \varepsilon \\
G[[[...]]] &= \varepsilon \\
G[[r_1 r_2]] &= G[[r_1]]G[[r_2]] \\
G[[r_1|r_2]] &= G[[r_1]]G[[r_2]] \\
G[[r*]] &= G[[r]] \\
G[[r+]] &= G[[r]] \\
G[[r?]] &= G[[r]] \\
G[[(? : r)]] &= G[[r]] \\
G[[(r)]] &= (? : |(E)) \\
G[[(r)*]] &= (? : |(r)*) \\
G[[(r)+]] &= (? : |(r)+)
\end{aligned}$$

*

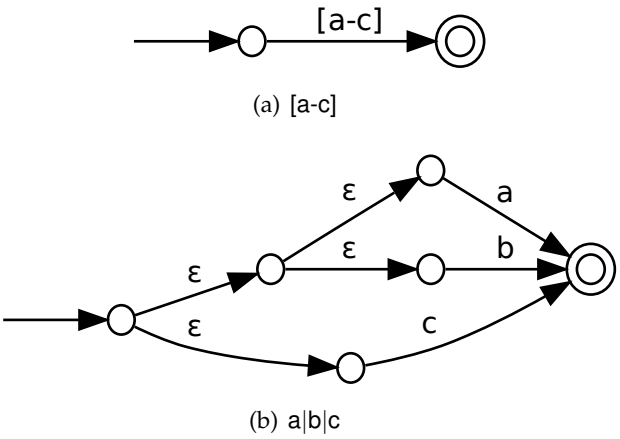
Example 10. Here follows a few examples of how the groupings filter rewriting function, G, works.

$$\begin{aligned}
G[[(\mathsf{a})|\mathsf{b}]] &= G[[(\mathsf{a})]]G[[\mathsf{b}]] \\
&= (? : (\mathsf{a}))\varepsilon \\
&= (? : (\mathsf{a}))
\end{aligned}$$

$$G[[((\mathsf{cup})\mathsf{cake})]] = (? : |((\mathsf{cup})\mathsf{cake}))$$

$$G[[(\mathsf{a}|\mathsf{b})*]] = (\mathsf{a}|\mathsf{b})*$$

*



(a) [a-c]

(b) a|b|c

Figure 11: A simple character class-transition example

4 Implementing a regular expression engine

The task of implementing a regular expression engine can be undertaken in steps. The first steps is converting the regular expression to a NFA. The next step is to simulate the NFA. The last step in our implementation is to build filters.

4.0.4 Character classes

Character classes are part of the extension we made to the regular expression definition. When implementing, we have the choice of rewriting character classes in terms of the original regular expressions, but as we can see in figure 11, this quickly becomes unwieldy. When we rewrite we add almost two states per character matched by the character class, instead of adding just one state for the whole character class. What we want is a NFA similar to figure 11(a), not figure 11(b).

There are several ways of obtaining this goal. Perl uses a bitmap to indicate membership of a range, for each character in the character set there is a bit in the bitmap. To decide membership the bit corresponding to the character is looked up. RE2 uses a balanced binary tree, each node in the tree corresponds to either a whole range or a literal character, the tree is then searched when deciding membership. Each method has its advantages and drawbacks. The bitmap is of constant size, so for small character classes, it will be unnecessarily large, but the time to look up a value in the bitmap is also constant and very fast. The balanced binary tree, has its advantages for character classes with few ranges and literal characters, since it will then be small in size and look up times. The drawbacks are of course that it grows in size and look up times with the character class.

For this project an even simpler solution was chosen: A simple linked list of ranges. The literal characters will be represented as ranges of length one. On other words, we will have one linked list per character class, and the number of elements in each linked list is the number of literals and ranges in the character class. Worst case we will have to look through all members of a linked list to decide membership of a character class. This is simplistic, but sufficient.

4.1 Regular expression to NFA

The first step in our regular expression engine is the regular expression to NFA converter. As discussed in section 2.4.1 on page 6, the NFA is built from the regular expression in steps from smaller NFA fragments. In order for this method, used directly, to be successful, the regular expression has to be in a form where the meta characters and the literals are presented in the right order. Regular expressions with for example | can not simply be read from left to right and be converted correctly. The problem with the alternation operator is that it is an infix operator, so we only have the left hand side and not the right hand side when we read the | and can therefore not complete the fragment.

Converting the regular expression to reverse polish notation, with an explicit concatenation operator, or making a parse tree will solve these problems. For this project neither is chosen. A third solution to this problem is maintaining a stack where fragments and operators are pushed and popped. This is the method that is implemented. We tried determining the quality of the decision by comparing run times with Russ Cox's example code [3]. This did not go well due to several reasons. The main reason is that the example code does not do well on large examples[2] and large examples is needed to do a reasonable comparison.

We followed Russ Cox' method from [4], when converting the regular expression to NFA. Russ Cox rewrites the regular expression to reverse polish notation with an explicit concatenation operator, so some changes will be necessary. There are tree main areas that needs to be changed:

Concatenation While constructing the NFA, NFA fragments are pushed onto a stack. Whenever the concatenation operator is encountered, the two top fragments are popped and patched together, see figure 4 on page 8. We do not have the advantage of an explicit concatenation operator. Instead we will be trying to pop the top two NFA fragments and patching them together as often as possible. As often as possible is after a character is read, but before any action is taken on the character read. The exception to this rule is the quantifiers, which binds tighter than concatenation.

[2] There are constants in the source code and a naive list append function

Parentheses The binding of the operators can be changed with parentheses. Not using a tree structure or reverse polish notation with an explicit concatenation operator, there is nothing showing the structure of how everything binds when simply reading the regular expression from left to right. We need some way of connecting the left parentheses to the matching right parentheses. For this we will be using the stack, we will expand it to also accept operators. Every time we read a left parenthesis in the regular expression, a left-parenthesis-fragment is pushed onto the stack. When we later on read a right parenthesis we simply pop fragments of the stack and patch them together till we reach a left-parenthesis-fragment.

Alternation When reading the regular expression left to right, we only have the left NFA fragment ready when reading the alternation operator. Therefore we simply push the alternation operator on the stack. Whenever possible we pop the alternation operator and associated NFA fragments and patch them together, see figure 3 on page 7. This is probably not very often, as it will only happen after reading a right parenthesis or the end of the regular expression.

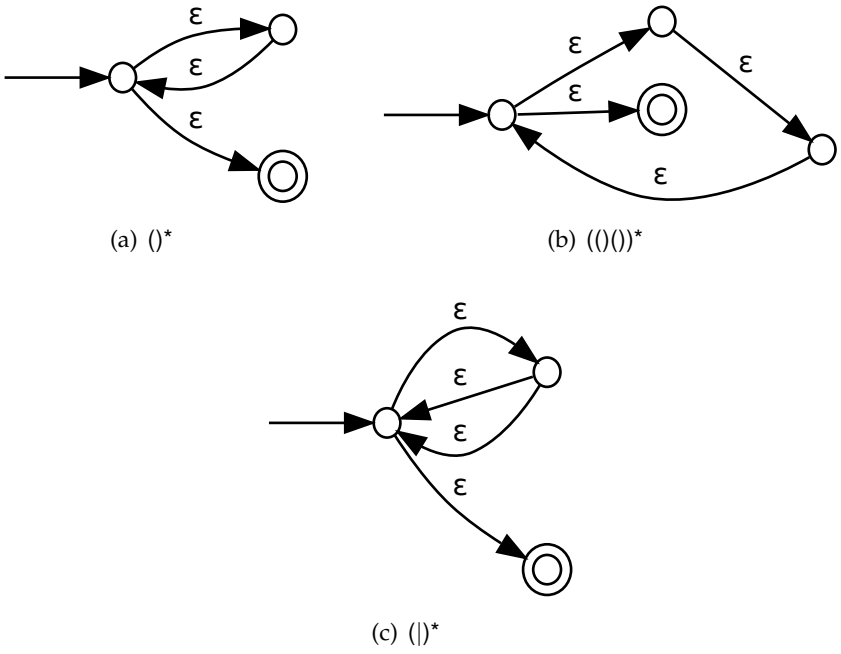
We have two important helper functions: `maybe_concat` and `maybe_alternate` . The first concatenates the top two fragments if possible, also see figure 4 on page 8. The second alternates the top fragments, if possible, so also figure 3 on page 7. `maybe_alternate` will pop alternate markers from the stack. These are called as often as possible to keep the stackdepth at a minimum and to avoid postponing all the concatenating and alternating till the end. Supplying a regular expression consisting entirely of left parenthesis will still make the stackdepth grow to a maximum.

4.2 The simulator

We have built the NFA and the next step is to simulate it. This requires keeping track of a set of active states. In a basic implementation of the Thompson simulation algorithm [13], a state is only added to the active set once. This will throw away matches, e.g. when the regular expression a|a is matched with the string *a* there are two possible routes through the NFA, but only one will be reported, since the final state will only be added once to the set of active states.

We had to adjust the basic implementation to our needs; we need to generate mixed bit-values.

We have implemented the standard Thompson simulation algorithm, but it is possibly to report all matches, you just need to allow a state to be added more than once to the active set. This will give rise to infinite loops when matching regular expressions like ()*, see figure 12(a) for the NFA. The simulator will go into a infinite loop generating these bit-values:



(a) ()* (b) (()())* (c) (|)*

Figure 12: ε-cycles

```
=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0...
```

This is because there is a cycle of ε-transitions in the NFA. This would not be desirable behavior and we would need to stop the simulation before it goes into an infinite loop.

cut out all this DFS stuff?

From [2] we have the depth-first search (DFS) algorithm. This algorithm can be modified to detect cycles. In short it works by initially marking all vertexes white. When a vertex is encountered it is marked gray and when all its descendants are visited, it is marked black. If a gray vertex is encountered, then we have a cycle and do not need to explore further on this path. The algorithm terminates when all vertexes are black. The algorithm will terminate, as we color one vertex each step and we always color the vertexes darker.

To see why this algorithm detects cycles, suppose we have a cycle containing vertex a. Then a is reachable from at least one if its descendants. When we reach a from this descendant, it will still be colored gray, since we are not done exploring a's descendants. Thus the cycle is detected.

Our problem is slightly different: We need to detect if we are in a cycle of ε-transitions. The DFS algorithm solution is still applicable, with slight modifications, as we do a depth first search when we explore the ε-edges. There will be no white states. Instead we will have a counter that is incremented every time a character is read from the input string. Every time a

state is encountered it is stamped with the counter. We can only trust the color of the state if the counter and the stamp are identical. The gray and the black states work in much the same way.

In figure 12 on the preceding page we have some of the NFAs we encounter. We have an example of a long cycle in figure 12(b), more parenthesis adds more ε-transitions. We also have an example of how more channels can be created in the loop in figure 12(c) by adding alternations.

4.3 Filters

4.3.1 Groupings

As we described in section 3.3.3 on page 19, this is the filter that should (more or less) throw away any mixed bit-values not generated in a capturing parenthesis. In order to do this we need to know which values are generated in a capturing parenthesis and which are not. We look to Laurikari [10] for inspiration. We will be using a NFA augmented with extra ε-transitions. The extra transitions will be used to mark the beginning and end of a capturing parenthesis. We will use the mixed bit-values to navigate the NFA, whenever we are inside a capturing parenthesis we will copy the mixed bit-values to output.

Add example of the extra epsilon-edges

We rewrote the regular expression to allow for capturing under alternation. We will need to insert a `1` when a group participates and a `0` when it doesn't. When exiting the upper arm of an alternation we need to know how many top level capturing groups there are in the lower arm and when entering the lower arm we need to know how many top level capturing groups there in the upper arm. Again we solve this problem by augmenting the NFA. We insert the extra information in the split-state marking the entrance to an alternation and add an extra state at the end of the upper arm.

Add example of the extra node and information

We adopt a similar strategy to solve the problem of reporting only the first match in capturing under a quantifier. We will again augment the NFA with necessary information. A state is inserted at the end of the quantifier, so that this state is the last state that is met in a iteration of the quantifier. When we pass this state and do another iteration we will know that we have already been there at least once and should not output any more bit-values.

Add example of star iteration count thingy

We could also have solved the problem of keeping track of how many times we have matched a quantifier by simply rewriting the regular ex-

pression. For example would we rewrite (a)* to (|a)a*. This was dropped because it can not be done easily on the fly by the NFA generator. The fragment formed by (a) could no longer be considered a finished fragment that was just plugged into the rest. We use it with and without the capturing parenthesis in the rewrite and would therefore need to open up the fragment and remove the capturing parenthesis for parts of the rewrite.

4.3.2 Trace

We will limit this filter to only output one channel with a match. In the current system this is not actually a limitation, as we are using Thompsons method for matching, there will only ever be one channel with a match.

All channels are read and the bit-values are saved separately. Every time we read a channel-split operator we will have to allocate a new chunk of memory and copy the bit-values we have accumulated up to this point. When the chunk of memory becomes to small, we will enlarge to a chunk twice the size. When we reach the end of the input stream, we will know if there is a match and be able to output the bit-values that make up the match.

4.3.3 Serialize

This is the filter that outputs what is matched. We will need the regular expression, that matches the bit-values, to form the NFA. The NFA is traversed using the bit-values. As we go along we output the symbols the transitions are marked with and the escaped symbols in the bit-values. This should result in a string being outputted of what was matched.

Table 1: Peak memory usage

Program	Size (KB)	Program	Size (KB)
`main`	3.43	`groupings_all`	3.43
`trace`	1500.16	`serialize`	3.43
`ismatch`	3.43		

5 Optimizations

The program as described in section 4 on page 24 is unoptimized and written for readability and simplicity. This section deals with potential and realized optimizations. Optimizing one aspect of performance can often hurt another aspect. See section A on page 61 for the specifications of the test computer. Programs were translated with `gcc` version 4.4.5 and the following flags: `-O3 -march=i686`.

5.1 Finding out where

The first step in optimizing should be finding out where. To this purpose we have set up a few experiments to decide the memory usage and runtimes of the programs. In all the following experiments, the first program, `main`, in our pipe is called with a regular expression matching capturing all: (.*) and about 114KB of text generated by `lipsum.com` to be matched.

5.1.1 Memory usage

In table 1 we have the peak memory usage charted. These numbers were collected with `valgrind` using the parameters `--tool=massif --stacks=yes`. These parameters mean we collect information on stack and heap usage. We can see from the table that all programs except `trace` use a negligible amount of memory. The Perl script in section D.1.1 on page 61 was used for collecting the data in this paragraph.

5.1.2 Output sizes

The sizes of the output from the previous experiment on memory usage is plotted in figure 2 on the following page. We can see that there is a big size difference between input to `main` and output; the output is 9 times bigger than the input. The same goes for the output of `groupings_all`, this is because there is nothing to be removed by this filter with this particular regular expression. The output of `trace` is 3 times bigger than the original string of text.

Table 2: Sizes of output

Program	Size (KB)	Program	Size (KB)
`main`	1022.5	`groupings_all`	1022.5
`trace`	340.8	`serialize`	113.6
`ismatch`	0		

Table 3: Runtimes

Program	Runtime (s)	Program	Runtime (s)
`main`	1.010	`groupings_all`	2.168
`trace`	18762.863	`serialize`	0.443
`ismatch`	0.773		

5.1.3 Runtimes

We made a small experiment to decide the runtimes of each program. We used the Perl script in section D.2.1 on page 64 to measure runtimes. We used the regular expression (.*) and a logfile of suitable size as inputs. In table 3 the runtimes for the different programs is seen. `trace` takes more than 5 hours to complete.

5.1.4 Profiling

We need to analyze the runtime behavior of the programs, to see which functions takes up the runtime of the programs. This is what a profiler is for. There are many to chose from. We chose to use `gprof` as it can give us an overview of which functions are called and how much time is spent in each.

The programs was compiled and linked with the `-pg` option to enable profiling data to be collected for `gprof`. We need to extend the runtimes to get better results from `gprof`, so instead of the text from `lipsum.com` we used a logfile of suitable size. The size was chosen to be small enough to fit in memory, but big enough to produce longer runtimes. In section D.3.1 on page 66 we have the perl script used for the profiling.

In figures 13, 14, 15 and 16 we have the output from `gprof` flat profile column marked `% time`. This column describes the percentage of the total running time used by this function. Only functions taking up more than 5% of the total runtime is included, the rest is bunched together in the other column.

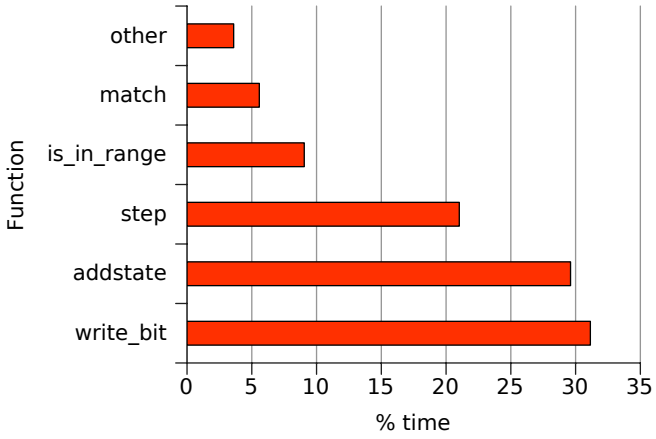


Figure 13: Output of `gprof` running `main`

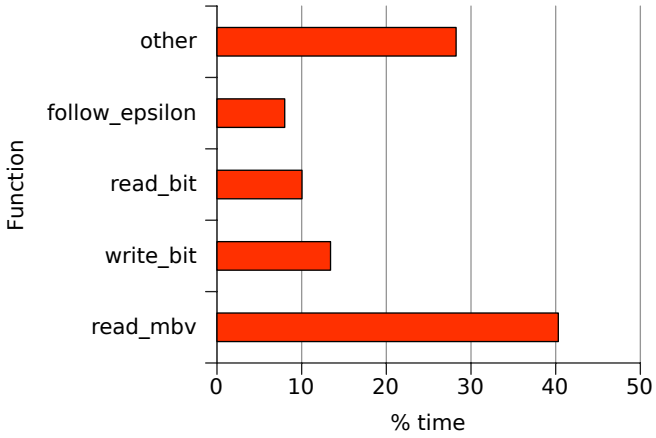


Figure 14: Output of `gprof` running `groupings_all`

`main` In figure 13 we have the values for running `main`. Functions `add_state` , `step`, `is_in_range` and `match` is all called in the process of simulating the NFA, this takes up about 65% of the total runtime. The rest is taken up by IO: Function `write_bit`. For this example, the process of creating the NFA is near instantaneous. We can also see the penalty for choosing a simple solution to the character class problem, the function for deciding membership `is_in_range` takes up about 9% of the total runtime.

`groupings_all` In figure 14 we have the values for running `groupings_all`. The main loop function, `read_mbv` takes up about 40% of the total runtime. We also see a large amount of runtime being taken up by functions with a small amount of runtime each, these are helper functions to the main loop

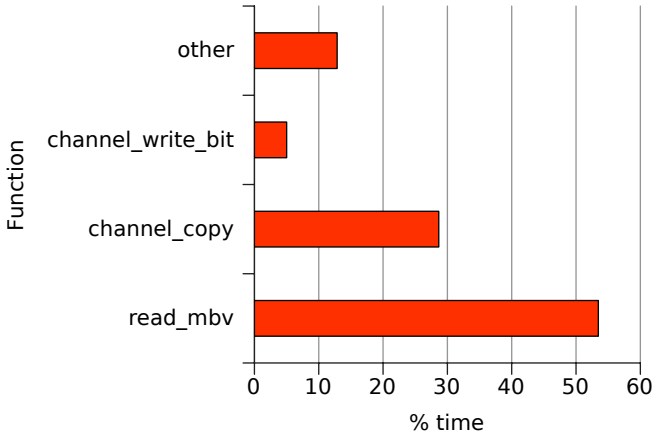


Figure 15: Output of `gprof` running `trace`

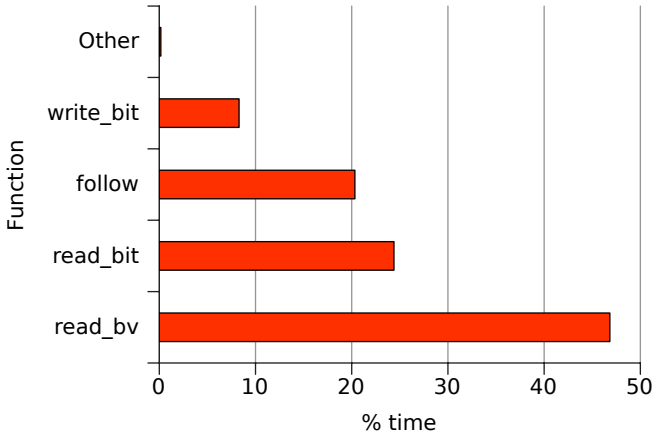


Figure 16: Output of `gprof` running `serialize`

and functions to do with keeping track of the channels. Again does the IO functions `write_bit` and `read_bit` take up a fair amount of runtime, about 23%.

`trace` In figure 15 we have the values for running `trace`. The main loop function `read_mbv` takes up more than half the total runtime. We spend a lot of time copying, writing to, appending and freeing channels, about 40% of the total runtime. Functions prepended with a `channel_` deal with channel management. Here the IO functions `read_bit` and `write_bit` take up a relatively little amount of runtime, about 5% in total.

`serialize` In figure 16 on the previous page we have the values for running `serialize`. Again the main loop function takes up a lot of time, about 47% of the total runtime. I/O functions `read_bit` and `write_bit` takes up about a third of the total runtime.

5.2 Applying the knowledge gained

In the previous section we identified a few troublespots, these are listed here.

IO From the output of `gprof` we can see that a lot of our runtime, in most programs, is used in the two I/O functions `read_bit` and `write_bit` .

`trace` `trace` takes to long to complete, a good place to start would be to look for a way around all the channel management.

Main loops A lot of the runtime is spend in main loops. This is not necessarily a good place to start optimizing, this could just as well be because we were not good at spreading out the workload in smaller functions.

5.2.1 ε-lookahead

Each ε-transition is marked with a look-ahead symbol. The ε-transition can then only be taken if the look-ahead symbol matches the next character in the input string. This optimization will have double effect, it will both reduce the number of states in the active set when simulating the NFA and it will reduce the mixed bit-values output at the cost of extra memory for and time spend constructing the NFA.

This optimization has not been implemented.

5.2.2 Improved protocol encoding

The protocol used for transmitting data is text-based, using a binary protocol would make the content more terse, but also nigh impossible to read for a human.

To transmit one operator in the text based protocol we always use 8 bits. Since we only have 8 different operators, this can be done using less bits. A widely used and effective technique for lossless compression of data is Huffman codes [2]. To encode our data efficiently with Huffman codes we need to analyze our data; we need to know the frequency with which the operators appear. In table 4 on the following page we have some frequencies of the operators using different regular expressions on a text file generated by `www.lipsum.org` of size 114KB. We have a simple and

Table 4: Frequencies of operators in a mixed bit-value string

Operator	Match all[a]	Match words[b]
`0`	11%	13%
`1`	11%	13%
`:`	22%	27%
`|`	11%	2%
`=`	11%	13%
`\`	11%	8%
`b`	11%	13%
`t`	0%	0%

[a] .*

[b] (?:(?:(?:[a-zA-Z]+ ?)+[,.;:] ?)*..)*

Table 5: Huffman encoding

Operator	Match all[a]	Match words[b]
`0`	1110	010
`1`	010	110
`:`	00	10
`|`	011	01110
`=`	100	111
`\`	101	0110
`b`	110	00
`t`	1111	01111

[a] .*

[b] (?:(?:(?:[a-zA-Z]+ ?)+[,.;:] ?)*..)*

a somewhat complex example. Since this is a table of operator frequencies, we have left out the escaped characters, this means the numbers will not sum to 100. What is missing is the escaped characters, these have the same frequency as the escape operator.

In figures 37 on page 62 and 38 on page 63 we have the corresponding Huffman trees, they yield the encoding in table 5. We observe that as the regular expression gets more complicated, we use | and \ less and `0`, `1`, `:`, = and `b` more. This is also reflected in the Huffman encoding. Since most regular expression will be more complex than .*, we choose the encoding in the match words column. This gives us a compression ratio of 0.39, for the match words case and a compression ratio of 0.44 for the match all case.

We can not make assumptions beforehand as to the frequencies of the escaped characters. Their frequency depends highly on the text being matched.

Table 6: Runtimes for different buffersizes

Buffer size (B)	Runtime (s)	Buffer size (B)	Runtime (s)
0	8.786	128	0.391
2	4.507	256	0.338
4	2.559	512	0.320
8	1.354	1024	0.313
16	0.886	2048	0.314
32	0.618	4096	0.310
64	0.437	8196	0.324

If the escaped characters all have the same frequency, then the Huffman method can achieve no compression. Instead we can look to the character class that generated the escaped character. By rewriting the character class with the | operator and creating the corresponding partial NFA as a balanced tree and use this tree as we would a Huffman tree, we can compress the escaped characters. How efficient this method is depends on how many characters is matched by the character class. For example if we match all characters, then no compression would be obtained, if on the other hand we had a small character class like [a-d] we could encode characters matched by this class using only 2 bits.

We did not implement this optimization.

5.2.3 Buffering input and output

The two functions called when doing I/O is `fputc` and `fgetc`. These are included with the `stdio.h` header file. Reading up on those two reveal that they are buffered and thread-safe. There is even a function for manipulating the buffering method: `setvbuf`. We did a bit of experimenting with `setvbuf`, the results are shown in figure 6. The runtimes in the graph is the combined runtime of all programs, i.e. we measured the runtime for all programs combined with pipes:

```
echo lipsum | ./main regex | ./groupings_all regex  |  \
./trace | ./serialize regex'
```

`regex` and `lipsum` are the same as used in section 5.1 on page 30. We can see that not much is gained from a buffer size exceeding 1024 bytes.

Thread-safety We also looked into thread safety. There are non-thread-safe variants of `fputc` and `fgetc`, `fputc_unlocked` and `fgetc_unlocked` respectively. Note that the man-pages does state that these thread-unsafe functions probably should not be used. The results from an experiment

Table 7: Runtimes for different buffersizes using non-thread-safe functions

Buffer size (B)	Runtime (s)	Buffer size (B)	Runtime (s)
0	8.597	128	0.248
2	4.355	256	0.204
4	2.288	512	0.181
8	1.240	1024	0.177
16	0.716	2048	0.164
32	0.463	4096	0.151
64	0.308	8192	0.162

similar to the previous, only using the non-thread-safe functions for IO, are in table 7. The non-thread-safe functions are on average 0.16 seconds faster.

5.2.4 Channel management in `trace`

The problem with `trace` is that we do not know which channel has a match, so we need to keep track of the bit-values on all of them. If we instead read the mixed bit-values backwards, the first character we would read on a channel would be the `t` or `b` operator, precluding the problem of knowing which channel has a match. This does require us to read the whole string of mixed bit-values and reversing it. Because this filter already is non-streaming, it will not become a problem reading and storing the whole string of mixed bit-values.

This optimization is implemented. Running the experiments determining runtime and memory usage for the improved `trace` gives us a runtime of just 1.309 seconds and a memory usage of 1536KB. Comparing this to the values for the old `trace` we see a huge performance gain in runtime and a slight increase in memory consumption. This optimization is well worth it.

We would expect no performance gain on this optimization when the regular expression consists of a string of literals, that is we only create one channel when simulating the NFA. This is however not a very useful application of regular expressions, there are faster ways of comparing two strings.

6 Evaluation

In this section we will be looking at how our programs compare to other implementations. Our implementation will be denoted as Main in the graphs. We have chosen a few languages and libraries that we feel are interesting:

RE2 RE2 is a new open source library for C++ written by Russ Cox. It is only a little more than a year old. It uses automata when matching. It does not offer backreferences.

TCL TCL added regular expression support in a release in 1999. The regular expression engine is written by Henry Spencer. It uses a hybrid engine. It is a interpreted language.

Perl Perl is from 1987 and is written by Larry Wall. It uses backtracking and virtual machines when matching. It is an interpreted language.

They are few in numbers, but they cover the basics in underlying technology and performance.

The benchmarks here can not be considered exhaustive, instead we have tried picking a few that would show interesting features of our programs.

Input method Our chosen method of input has a drawback, namely the upper limit on size for command line input. The system we tested on, see also A on page 61, has a upper limit on command line input of 2MB. This can be ascertained (on our test system at least) by issuing the following command:

```
$ getconf ARG_MAX
2097152
```

In the unlikely event that a user will need to match with a regular expression exceeding this 2MB limit, there is always the option to use a file instead. Files only suffer the limit that they need to fit in memory.

6.1 A backtracking worst-case

Our first benchmark is taken from [4], it demonstrates the worst-case behavior of the backtracking algorithm. Using superscripts to denote string repetition, we will be matching $\mathsf{a?}^n\mathsf{a}^n$ with the string a^n. For example will $\mathsf{a?}^3\mathsf{a}^3$ translate to a?a?a?aaa. We expect Perl to do poorly in this benchmark, while the rest should do well.

For this experiment we used the programs `main` and `ismatch`. The script used for the backtracking worst case is in sections E.1 on page 67 and E.2 on page 69.

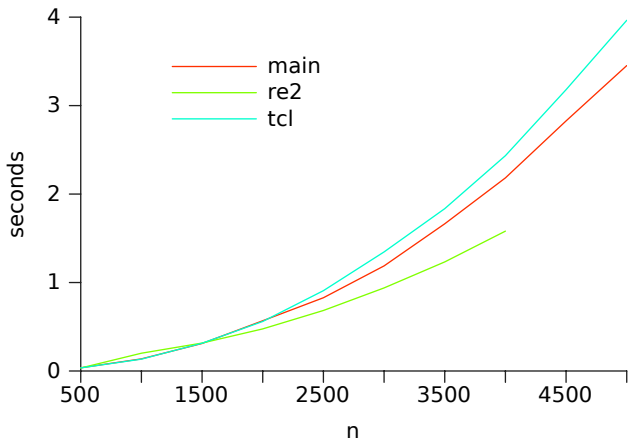


Figure 17: A backtracking worst-case: Main, Tcl and RE2 runtimes.

6.1.1 Runtimes

The runtimes can be seen in figures 17 and 18. As anticipated: Perl exhibits very poor performance. The slope on figure 18 on the following page and the logarithmic scale suggest that Perl runs in time exponential in n. This is not surprising considering the following. The ? matches greedily, meaning it will first try to consume a character from input. The only way that the regular expression matches the string is if all the quantifiers consume no input. There are 2^n possible ways for the quantifiers to consume and not consume a character. The backtracking engine has to search through all the 2^n possible solutions to find the matching one, since it matches greedily.

In figure 17 we do not see much difference in the performance of Main, RE2 and Tcl. RE2 stops before the others, it has a upper limit on how much memory it will consume. The limit is user defined, see documentation in the RE2 headerfile. This limit could have been set to a value that would allow RE2 to continue matching with Main and Tcl. We chose not to do this, because we wanted to demonstrate this feature in RE2. This feature is especially useful in setups where memory is very tight or you accept regular expressions from untrusted sources, but can be considered a nuisance in other situations where you do not want to fiddle with this limit, but just want to make RE2 do your matches.

As a side note, we found that if we added a b at the end of the regular expression we would suddenly see marked improvements in the runtimes of Perl. Using a backtracking algorithm it would still take time exponential in n to decide they did not match, but Perl scans the input string for all literals in the regular expressions, and it quickly discovers that there is no *b* in the input string, so therefore the regular expression can not match the string.

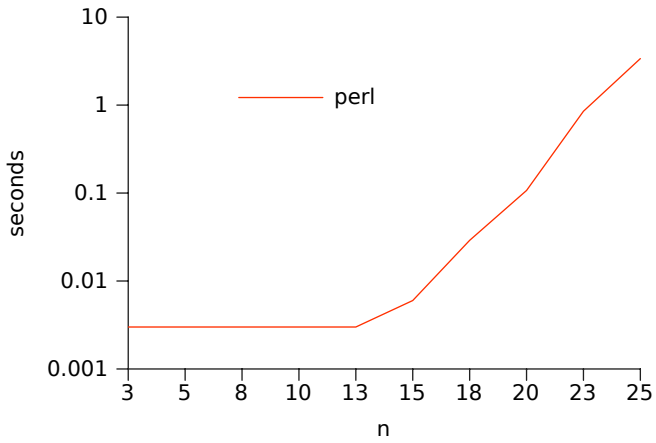


Figure 18: A backtracking worst-case: Perl runtime on a logarithmic scale.

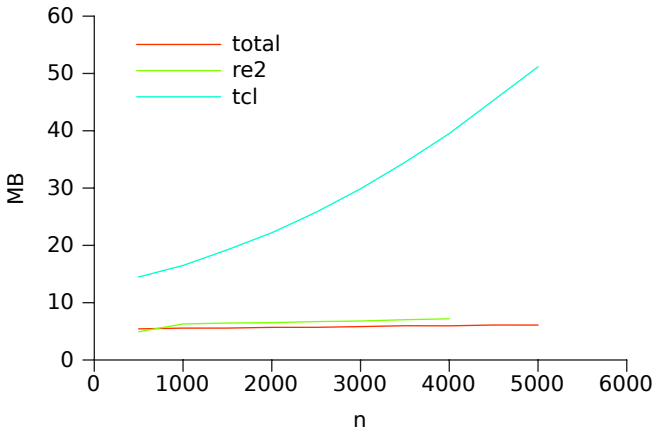


Figure 19: A backtracking worst-case: Total, Tcl and RE2 memory usage.

6.1.2 Memory usage

Memory usage is depicted in figures 19 and 20. For our program we have added up the memory usage of the individual programs and displayed them under the total header.

In figure 19 we note that RE2 stops before the other two, see above section for an explanation. The memory usage of our programs and RE2 does not appear to be more than linear in the input size, Tcl looks more like some quadratic function. It is hard to give a good explanation to this without knowing Tcls regular expression better, even if it did use a NFA for this match the size should still be linear in the input. Tcl has the same asymptotic runtime performance as Main and RE2, but with bigger constants, all

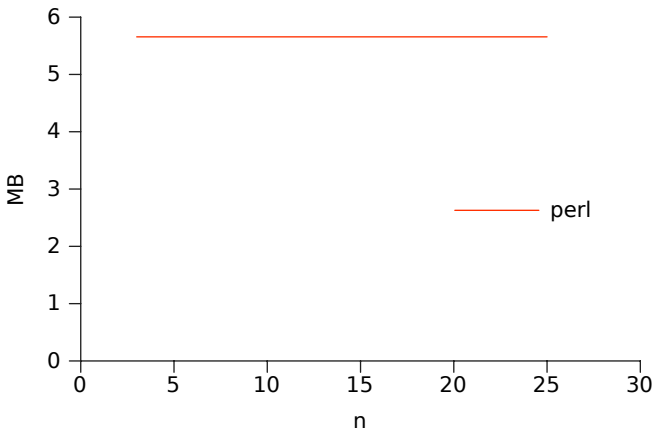


Figure 20: A backtracking worst-case: Perl memory usage.

that extra memory spent is not being put to good use.

Figure 20 shows Perls memory usage. It is hard to say anything useful based on that graph, the values for n are too small, but due to the exponential nature of the problem is it not viable to increase them.

6.2 A DFA worst-case

Using superscripts to denote string repetition, constructing a DFA from the regular expression (a|b)*a(a|b)^n results in a exponential blow up of the state count. For example will (a|b)*a(a|b)^3 translate to (a|b)*a(a|b)(a|b)(a|b). Acceptance with a DFA is decided in time linear to the size of the input string, but we would still have to store the DFA which takes space exponential in the size of the regular expression. We expect any regular expression engine using DFAs to do poorly on this benchmark. The engines that can be expected to use DFAs are RE2 and Tcl, but both can switch method according to need.

For this experiment we used the programs `main` and `ismatch`. The script used for the backtracking worst case is in sections E.3 on page 70 and E.4 on page 72.

6.2.1 Runtimes

In figures 21 and 22 we have the figures displaying the runtimes of the various programs. Again we note that RE2 stops before the others, see above. Tcl stands out with significantly lower performance, but none appears to have exponential or worse asymptotic behavior. This would suggest that Tcl chooses to use a DFA in some form for this match and RE2 falls back on something else.

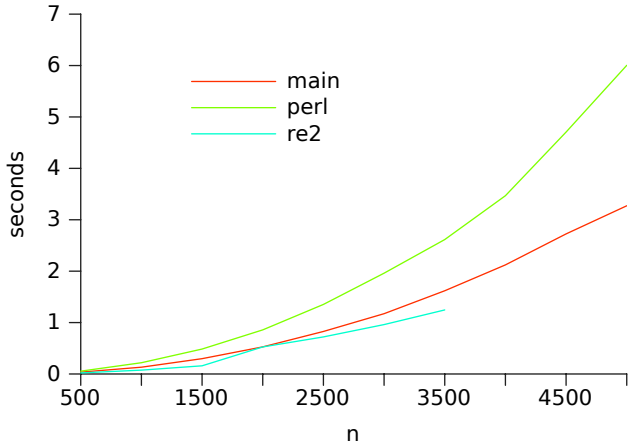


Figure 21: A DFA worst-case: Main, RE2 and Perl runtimes.

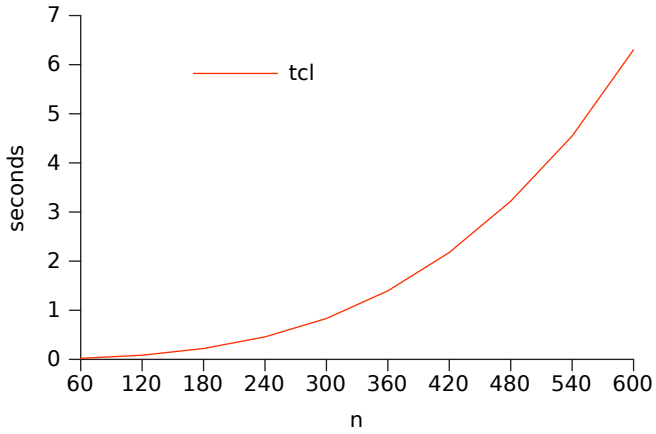


Figure 22: A DFA worst-case: Tcl runtimes.

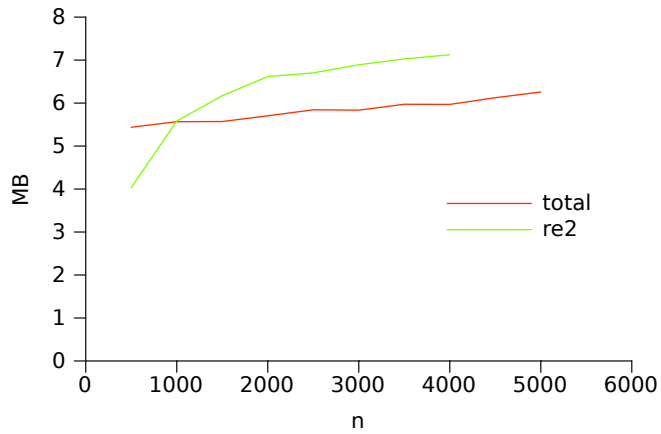


Figure 23: A DFA worst-case: Main and RE2 memory usage.

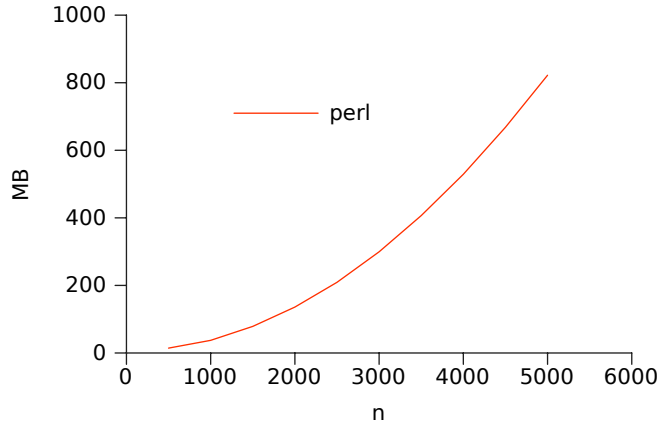


Figure 24: A DFA worst-case: Perl memory usage.

6.2.2 Memory usage

The memory usage proved to be a more complicated matter than the runtimes, see figures 23, 24 and 25 display.

Our programs and RE2 appears to be using memory linear in the size of the regular expression. There is a sharp rise in memory consumed by RE2 for n smaller than about 1000. This would indicate that RE2 uses a DFA until the exponential factor becomes to big and forces it to switch method.

Perls memory usage is mapped in figure 24. Compared to our programs Perl uses rather a lot of memory. It seems to be increasing in a quadratic manner.

investigate further

In figure 25 on the next page we have Tcls memory usage mapped. Note the logarithmic scale. Our suspicion that Tcl uses a DFA is confirmed by the

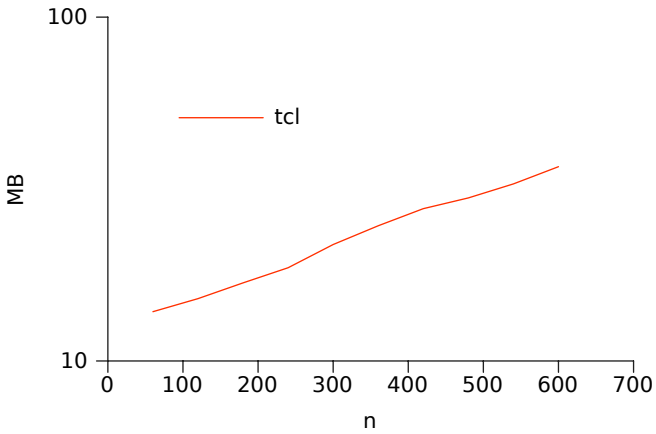


Figure 25: A DFA worst-case: Tcl memory usage.

memory usage, which appears to be exponential in the size of the regular expression.

6.3 Extracting an email-address

This is the first of our real world benchmarks. We will be extracting an email-address from a string of text. Since we can not do partial matches, we will be constructing strings of increasingly long email-addresses. The regular expression is taken from [14]. Unlike the two previous benchmarks, the regular expression is kept constant and does not grow.

Since we are extracting a value we are using `main`, `groupings_all`, `trace` and `serialize` for this match. The scripts can be found in sections E.5 on page 73, E.6 on page 76 and E.7 on page 78

6.3.1 Runtimes

In figure 26 on the next page we have the runtimes. All appear to be running in time linear to the input string, with Tcl clearly having the lowest constants and our programs the biggest.

6.3.2 Memory usage

In figure 27 on the following page we have the memory usage of the programs. All programs except ours seem to be using memory linear in the input string. We seem to be using memory in a stepped manner. This correlates well with our scheme for memory management in `trace`: We double the amount of memory used every time we run out. This is confirmed by figure 28 on page 46, which displays the memory usage for the individual

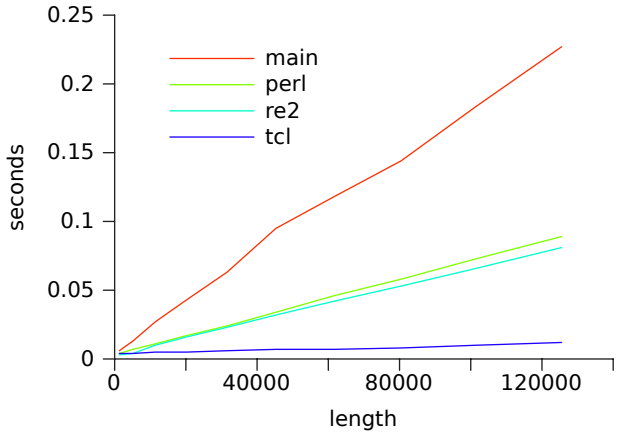


Figure 26: Extracting an email-address: Runtimes.

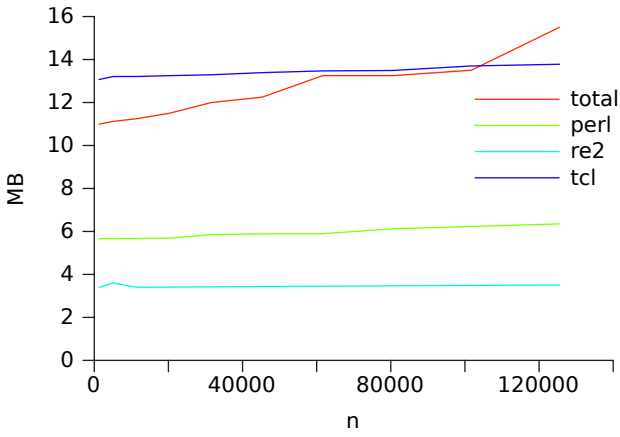


Figure 27: Extracting an email-address: Memory usage.

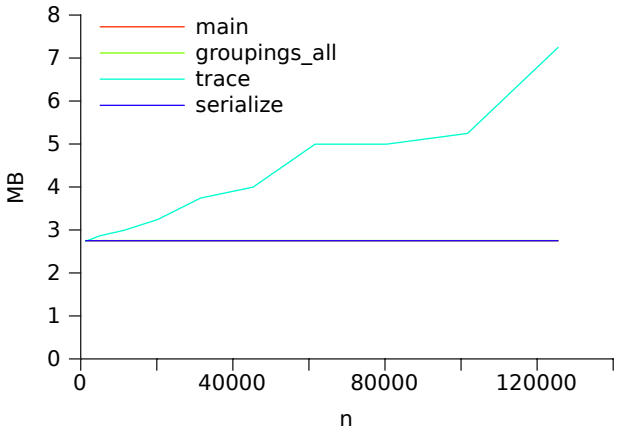


Figure 28: Extracting an email-address: Individual programs memory usage.

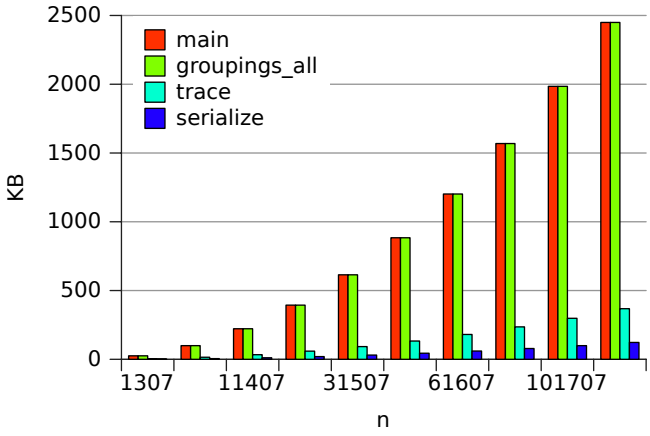


Figure 29: Extracting an email-address: Sizes of output from individual programs.

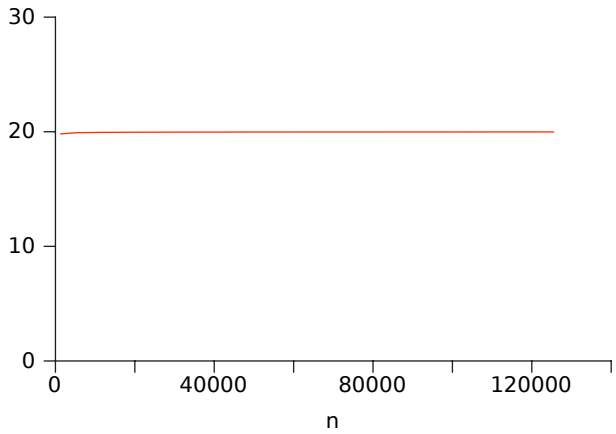


Figure 30: Extracting an email-address: The relationship between input size to `trace` and size of input string.

programs. Here we see that all our programs except `trace` use a constant amount of memory. It is hard to tell from figures 28 and 27 on page 45 if the memory used is linear in the input string. What we need to know is the size of the mixed bit-values compared to the input string. This we have displayed in figure 29 on the previous page, where we have the sizes of the output from the individual programs. We did not put in a separate column for the size of the input string, since this is exactly the same as the size of the output from `serialize`. There is a linear relationship between the output of `groupings_all` and `serialize`: The first is 20 times bigger than the latter. See figure 30.

6.4 Extracting a number

Our fourth and last benchmark is also a real world example taken from [14]. This one extracts a number from a string. We can not do partial matches, so again we will be using a string consisting of increasingly large numbers. The regular expression is constant.

We will be extracting a number, so we will be using programs `main`, `groupings_all`, `trace` and `serialize` for this match. The scripts can be found in sections E.8 on page 79, E.9 on page 81 and E.10 on page 82.

6.4.1 Runtimes

In figure 31 on the following page we have the runtimes for this benchmark. They all appear to be linear in the size of the input string. Our programs clearly have bigger constants than the rest.

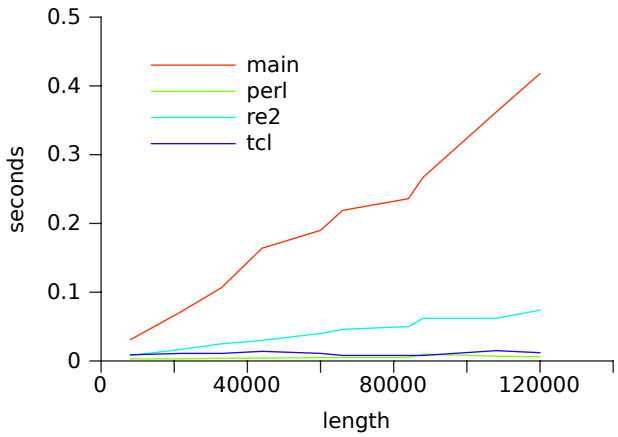


Figure 31: Extracting a number: Runtimes.

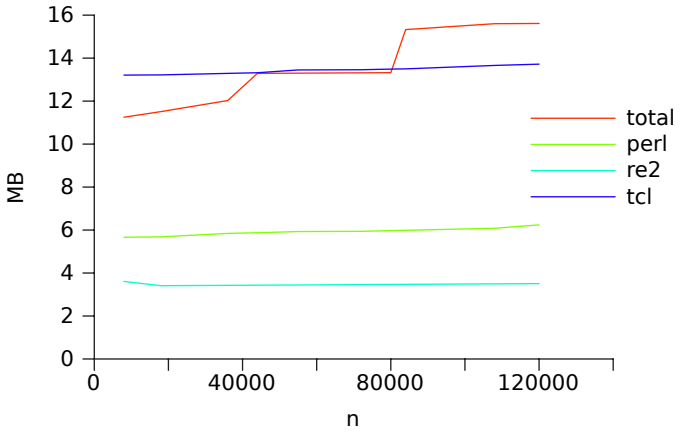


Figure 32: Extracting a number: Memory usage.

6.4.2 Memory usage

In figure 32 we have the memory usage. We see, perhaps even more clearly, the same pattern as in the extracting an email address benchmark, that our programs use memory in a stepped manner and the rest use memory linear in the size of the input string. We investigate further, see figure 33 on the following page for the individual programs memory usage. All our programs, except `trace` use a constant amount of memory. The steps in the memory usage of `trace` is even more pronounced in this figure. See above note on memory management strategy in `trace` for explanation. The relationship between the sizes of the output from the different programs is displayed in figure 34 on the next page. The size of the input string is exactly the same as the output from `serialize`. For greater clarity we

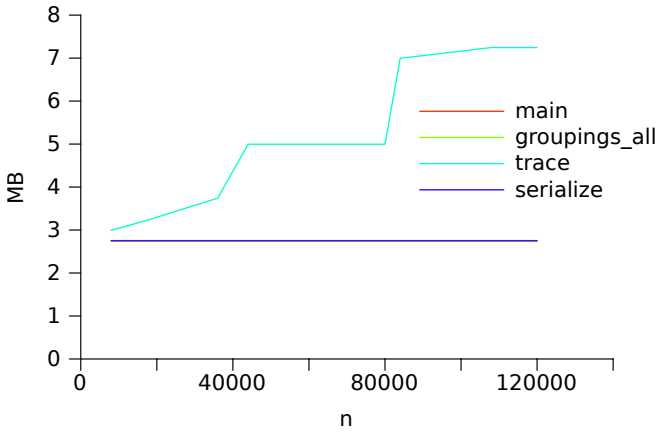


Figure 33: Extracting a number: Individual programs memory usage.

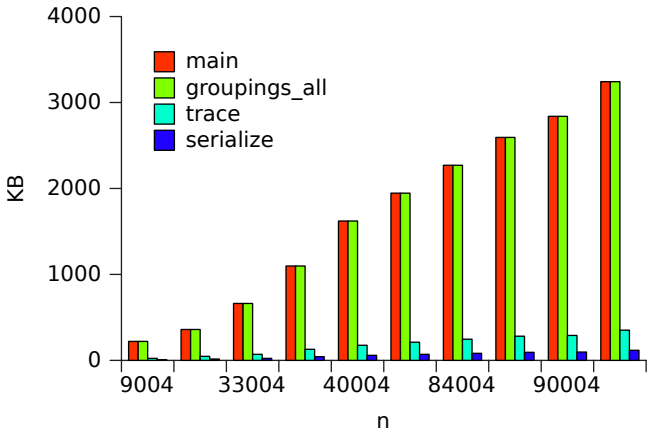


Figure 34: Extracting a number: Sizes of output from individual programs.

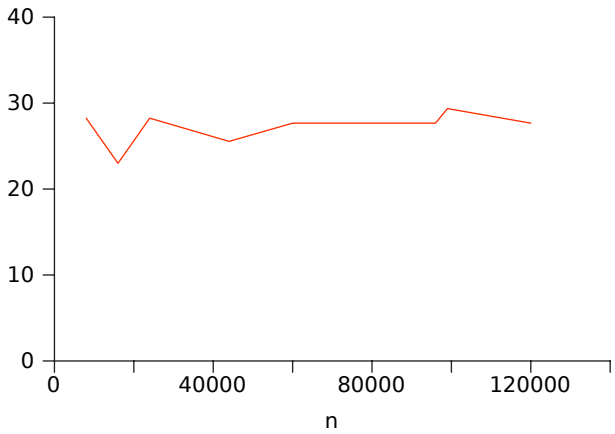


Figure 35: Extracting a number: The relationship between input size to `trace` and size of input string.

have the relationship between the input size to `trace` and the size of the input string displayed in figure 35. We do not observe the same fixed relationship between the two as we did in the extracting an email address benchmark. There seem to be no increasing trend. We generate between 20 and 30 mixed bit-values per byte in the input string.

6.5 Conclusion

7 Related work

Opsummer hvad vi har haft af related work i rapporten of inkluder det der ikke passede ind

The groundwork of regular expression has been known for decades. Current research is often focused generally; for example to improve matching speeds, but there is also heavy activity towards more specialized purposes, such as XML parsers, hardware based spam detection or even such fields as biology where regular expressions are often used to find patterns of amino acids.

The intended aim of this particular project is towards the general realm (the curios reader may wish to read if they are interested in special usages of regular expressions). For this reason, the remainder of this section will therefore concentrate on likewise general research.

cite-dig-selv

blah blah om ting du ikke har snakket naer-mere om

7.1 Constructing NFAs

It is possible to construct different NFAs accepting the same language. These NFAs will have different properties.

Find out other methods for constructing NFAs

7.2 Simulating NFAs

7.2.1 Frisch and Cardelli

Frisch and Cardelli presents a way of simulating a NFA in their paper [7]. It works in two passes over the input string. The first pass annotates the input string with enough information to decide which branch to pick in alternations and how many times to iterate a quantifier. This is done by reading the input from right-to-left and working our way backwards in the NFA, the visited states are annotated and stored with the input string. In the second and main pass, where we read the input string left-to-right and work our forwards in the NFA, these annotations are used to decide which branch to take in an alternation and how many times to iterate a star. This method is not suitable for a streaming regular expression engine. The input string is read first from end-to-front and then from front-to-end, this can not be done without storing the string.

7.2.2 Backtracking

Backtracking is a way of simulating a NFA. It is the method employed by many programming languages and libraries, such as Perl and PCRE. Compared to other methods, this has the advantage of allowing backreferences,

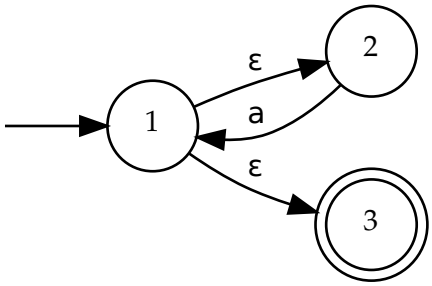


Figure 36: NFA for a*

but it is also a worst-case exponential-time algorithm. An example of a matching that exhibits worst-case behavior is the regular expression $\mathsf{a?}^n\mathsf{a}^n$ and the string a^n, where superscripts denotes string repetition. This example is also known from the article [4].

A backtracking algorithm works depth-first. It has one active state (`AS`) and a string pointer (`SP`) and a stack of save-points. Every time we have to make a choice as to which transition to take when traversing the NFA, we save the state so we can later return and explore the alternate routes. Each save-point consists of a state and a pointer to the string.

Example 11 (Backtracking)**.** In this example we will be matching a* to the string *aa*. The NFA we will be using for this example is in figure 36. The process of matching could look like:

AS	SP	Save-points	Explanation
1	$\underline{a}a$		Initially AS is set to the start state and SP is set to the first character in the string
2	$\underline{a}a$	(1, $\underline{a}a$)	Two paths are available, so we save the state and take the ε-transition to state 2.
1	$a\underline{a}$	(1, $\underline{a}a$)	We take the transition back to state 1 and consume the first *a*.
2	$a\underline{a}$	(1, $\underline{a}a$) (1, $a\underline{a}$)	Two paths are available, so we save the state and take the ε-transition to state 2.
1	$aa\_$	(1, $\underline{a}a$) (1, $a\underline{a}$)	We take the transition back to state 1 and consume the second *a*.
2	$aa\_$	(1, $\underline{a}a$) (1, $a\underline{a}$) (1, $aa\_$)	Two paths are available, so we save the state and take the ε-transition to state 2.
1	$\underline{a}a$	(1, $a\underline{a}$) (1, $aa\_$)	No transitions are available from state 2, so this path is abandoned. We backtrack and pop a save-state.
3	$\underline{a}a$	(1, $a\underline{a}$) (1, $aa\_$)	The other available ε-transition to state 3 is taken.
1	$a\underline{a}$	(1, $aa\_$)	No transitions are available from state 3, so this path is abandoned. We backtrack and pop a save-state.
3	$a\underline{a}$	(1, $aa\_$)	The other available ε-transition to state 3 is taken.
1	aa		No transitions are available from state 3, so this path is abandoned. We backtrack and pop a save-state.
3	$a\underline{a}$		The other available ε-transition to state 3 is taken. We have reached the end of the string and are in an accepting state: We have a match!

*

7.3 Virtual machine

Another popular method of matching regular expressions to text is the virtual machine approach [5]. Instead of constructing a automaton, we generate bytecode for an interpreter.

A simple virtual machine would have the ability to execute threads, each thread consisting of a regular expression program. Each thread would

Table 8: Code sequences

a	`char` a
e_1e_2	*codes for* e_1 *codes for* e_2

$e_1 \mid e_2$	`split L1, L2`
	`L1:` *codes for* e_1
	`jmp L3`
	`L2:` *codes for* e_2
	`L3:`

$e*$	`L1:` `split L2, L3`
	`L2:` *codes for* e
	`jmp L1`
	`L3:`

maintain a program counter (`PC`) and a string pointer (`SP`). A regular expression program could for example consist of the following instructions:

`char` c If the `SP` does not point to a c character, then this thread of execution is abandoned. Otherwise, the `SP` and the `PC` is advanced.

`match` Stop thread, we have a match.

`jmp` x `PC` is set to x.

`split` x`,` y Split the thread of execution. The new threads `PC` is set to x and the old threads `PC` is set to y.

With these few and simple instructions we are able to compile regular expressions with concatenation, alternation and repetition, see table 8.

Example 12 (Virtual machine)**.** We are now ready for a small example match, we can match the regular expression `a*` with the string *aa*. The regular expressions compiles to

```
0 split 1, 4
1 char a
2 jmp 0
4 match
```

Running this on a virtual machine could look like

Thread	PC	SP	Explanation
T1	0	*a*a	Create thread T2 with PC set to 4 and SP at *a*a. T1 continues execution at 1.
T1	1	*a*a	Character matches, SP and PC is advanced.
T1	2	*aa*	PC is set to 0.
T1	0	*aa*	Create thread T3 with PC set to 4 and SP at *aa*. T1 continues execution at 1.
T1	1	*aa*	Character matches, SP and PC is advanced.
T1	2	*aa*	PC is set to 0.
T1	0	*aa*_	Create thread T4 with PC set to 4 and SP at *aa*_. T1 continues execution at 1.
T1	1	*aa*_	Character does not match and this thread is abandoned.
T2	4	*a*a	We have a match.
T3	4	*aa*	We have a match.
T4	4	*aa*_	We have a match.

*

We believe this is how Perl matches [11]. Running the program with debug mode on makes Perl print a textual representation of the bytecode the regular expression is compiled into. In section C.1.1 on page 61 we have a small Perl experiment demonstrating this feature. The results of running the program:

```
$ ./regexmach.pl
Compiling REx "a*"
Final program:
   1: STAR (4)
   2:   EXACT <a> (0)
   4: END (0)
minlen 0
Freeing REx: "a*"
```

8 Future work

Introduction

8.1 Extending the current regular expression feature-set

Regular expressions in real world usage is more complicated than what we have described in this thesis. Here is a list with some of the features that the regular expressions presented here could be extended with.

Counted repetitions A shorthand for matching at least n, but no more than m times can easily be implemented. Using industry standard notation, the repetition $e\{3\}$ expands to eee, $e\{2,5\}$ expands to $eee?e?e?$ and $e\{2,\}$ expands to $eee*$, where e is some regular expression.

Non-greedy or lazy quantifiers Traditionally the quantifiers will match as much as possible, they are greedy. Non-greedy quantifiers will match as little as possible.

Character class shorthands Often used character classes, like [A-Za-z0-9_] for a word character, often has shorthands. It makes for shorter and more readable regular expressions.

Unanchored matches In this thesis we have assumed that the regular expression will match the whole of the string. In practice, it is often useful to find out if the regular expression matches a substring of the input string. An unanchored match has implicit non-greedy .* appended and prepended. Here it would also be very useful to have the ability to restart matching where the previous left of.

More escape sequences Special escape sequences for characters like tab, newline, return, form feed, alarm and escape are useful. They are more readable in a regular expression than the actual characters themselves, because they will show as blank space in most editors.

Assertions Assertions does not consume characters from the input string. They assert properties about the surrounding text. Most languages and libraries provide the start and end of line and word boundary assertions. There are also general assertions, like the lookahead assertion (?=r) which asserts that the text after the current matches r.

Case insensitive matching This could be implemented as the poor man's version of the case insensitive match: Every character a matched case insensitively is expanded to a character class $[aA]$. This might not be the best idea performance wise, since the character classes are so expensive to simulate.

8.2 Internationalization

What this long word covers over is basically integrating other character sets than the ASCII. Internationalization also makes character class shorthands mean different things. The word character class from above would vary according to locale, for example in a danish setting it would make more sense defining it as: [A-a-0-9_].

8.3 More and better filters

Copy A copy filter can easily be implemented, just write input to two output channels. These channels could be standard input and error. An effect very similar can already be achieved with the utility `tee`, this will read from standard input and write to standard output and files.

Serialize The serialization filter dumps the contents of the captured groups with no formatting. It would be beneficial to the user to have some kind of formatting to distinguish the different captured groups.

8.4 Concurrency

We can not split up neither the regular expression nor the input string for concurrent processing. To split up the regular expression, we would need to know exactly how much the sub-regular expressions each consume of the input string. This we cannot know unless we actually do the match. Splitting up the string instead would also require knowledge we can only gain by actually performing the match, we would need to know in what state the simulation process should start at this particular input string symbol.

The current setup is a pipeline. Only the streaming filters are able to process data upon reception, the non-streaming filters gathers all data in the input stream before beginning the processing. On most Unix-like systems it is possible to chain programs together with pipes. The output of a program is directly fed to the input of the next program in the chain. This is usually implemented so that all the programs in the chain is started at the same time. The scheduler is then responsible for managing the processes. Although the problem is not parallel in nature, by dividing the solution

into discrete components we can at least utilize each streaming component simultaneously.

To gain more control over the concurrency we could implement the programs as processes. This would allow us to tweak the scheduling even further. It is however doubtful that this will give a performance boost based on better control over the concurrency, as the scheduler already does a good job of this. The performance boost will more likely come from faster communication channels.

In a Unix environment there will be no loss of data in a pipeline, if for example a program can produce data faster than the receiving program can read. The data will be buffered until the receiving program is ready to read the data. If the buffer fills up, the producer will be suspended until there again is room in the buffer. This could mean we are buffering data twice, once in the buffer used by the operating system between programs in a pipeline and once in the buffer used by the programs own input and output.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] Russ Cox. Regular expression implementation, 2007. `http://swtch.com/~rsc/regexp/nfa.c.txt`.

[4] Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007. `http://swtch.com/~rsc/regexp/regexp1.html`.

[5] Russ Cox. Regular Expression Matching: the Virtual Machine Approach, 2009. `http://swtch.com/~rsc/regexp/regexp2.html`.

[6] Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, September 2000.

[7] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 618–629. Springer, 2004.

[8] Fritz Henglein and Lasse Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). 2010.

[9] Jeffrey D. Hopcroft, John E. And Motwani, Rajeev And Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd editio edition, 2001.

[10] Ville Laurikari. Efficient submatch addressing for regular expressions. 2001.

[11] Yves Orton. perlreguts, 2006. `http://perldoc.perl.org/perlreguts.html`.

[12] Line Bie Pedersen. Regular expression libraries, tools and applications. 2010.

[13] Ken Thompson. Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[14] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:498–507, 2010.

[15] Larry Wall. Apocalypse 5: Pattern Matching, 2002. `http://dev.perl.org/perl6/doc/design/apo/A05.html`.

A Test computer specifications

In this section we put the relevant parts of the specifications of the computer that was used for testing and benchmarking.

Software versions

Operating system Ubuntu 10.10 - the Maverick Meerkat

`gcc` (Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5

`perl` 5.10.1 (*) built for i686-linux-gnu-thread-multi

`tcl` 8.4.16-2

`re2` Version present in the repository at the date of fetching: 2. March 2011.

`g++` Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5

Technical specifications

CPU Intel(R) Core(TM) i3 CPU M 330 @ 2.13GHz

Memory 1938 MB

Storage Samsung HM250HI

B Huffman trees

C Experiments

C.1 Perls debug output

C.1.1 `regexmach.pl`

```
#! /usr/bin/perl -Wall

use strict;
use re Debug => 'DUMP';

"aaa" =~ /a*/;
```

D Optimization scripts

D.1 Memory usage

D.1.1 `memoryusage.pl`

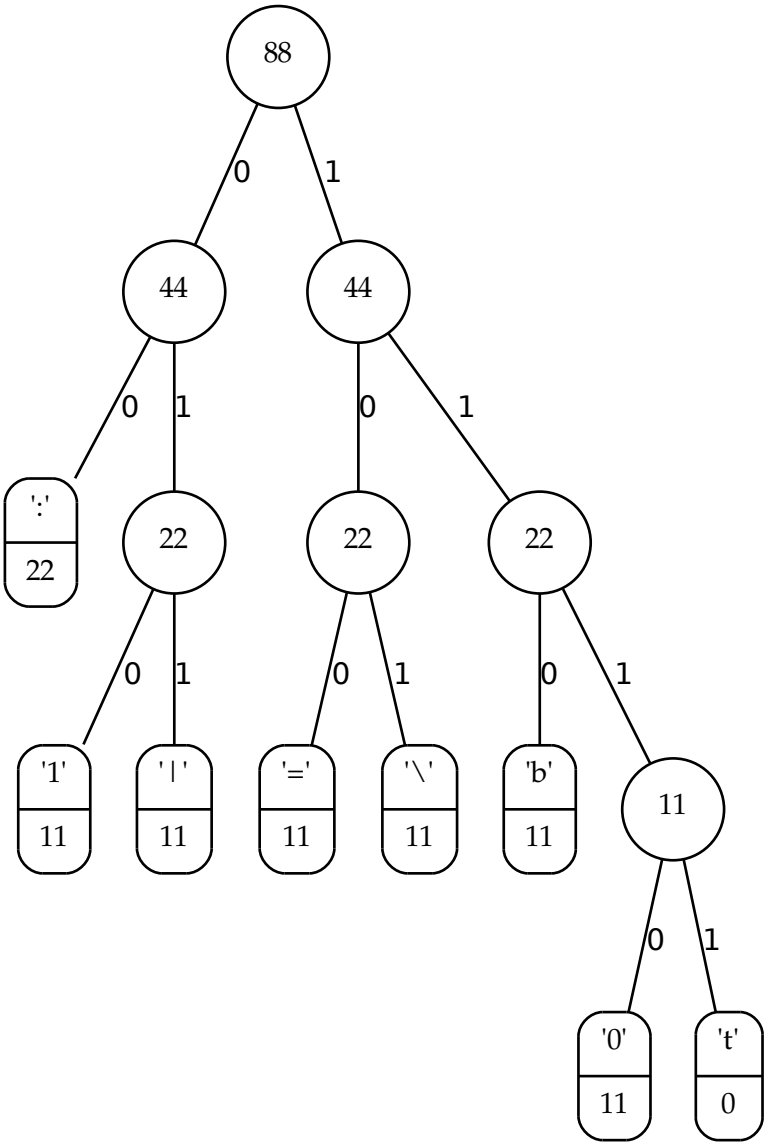


Figure 37: Huffman tree for frequencies in table 4, .*

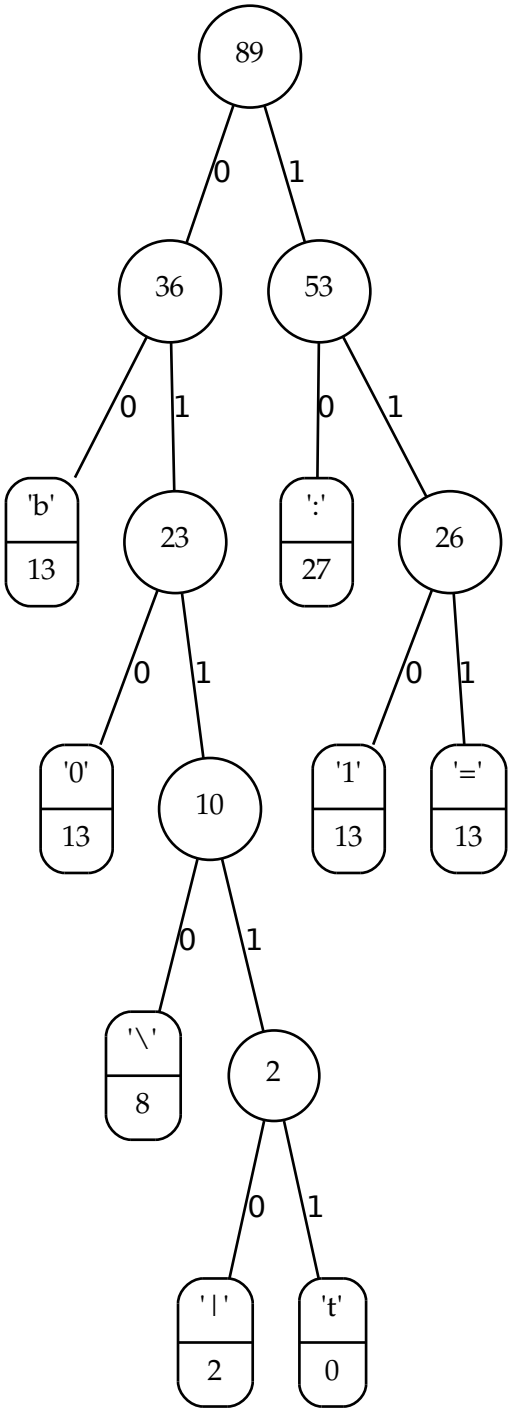


Figure 38: Huffman tree for frequencies in table 4, (?:(?:(?:[a-zA-Z]+ ?)+[,.;:] ?)*..)*

```
#! /usr/bin/perl -Wall

use strict;

my $regex = '(.*)';
my $regex2 = '|(.*)';

# Generate the files
`cat lipsum.txt | ./main '$regex' > ~/speciale/memory.main.
   mbv`;
`./groupings_all '$regex' <  ~/speciale/memory.main.mbv > ~/
   speciale/memory.groupings_all.mbv`;
`./trace < ~/speciale/memory.groupings_all.mbv > ~/speciale/
   memory.trace.mbv`;
`./serialize '$regex2' <  ~/speciale/memory.trace.mbv > ~/
   speciale/memory.serialize.mbv`;

# Collect memory usage data using massif
`cat lipsum.txt | valgrind --tool=massif --stacks=yes ./main
    '$regex'`;
`valgrind --tool=massif --stacks=yes ./ismatch < ~/speciale/
   memory.main.mbv`;
`valgrind --tool=massif --stacks=yes ./groupings_all '$regex
   ' < ~/speciale/memory.main.mbv`;
`valgrind --tool=massif --stacks=yes ./trace2 < ~/speciale/
   memory.groupings_all.mbv`;
`valgrind --tool=massif --stacks=yes ./trace < ~/speciale/
   memory.groupings_all.mbv`;
`valgrind --tool=massif --stacks=yes ./serialize '$regex2' <
    ~/speciale/memory.trace.mbv`;
```

D.2 Runtimes

D.2.1 `runtimes.pl`

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';


my $regex = "(.*)";
my $regex2 = "|(.*)";
my $startTime, my $endTime, my $result;

# Generate the files
`cat ~/speciale/shorttrace | ./main '$regex' > ~/speciale/
   runtime.main.mbv`;
`./groupings_all '$regex' <  ~/speciale/runtime.main.mbv >
   ~/speciale/runtime.groupings_all.mbv`;
```

```
`./trace < ~/speciale/runtime.groupings_all.mbv > ~/speciale
    /runtime.trace.mbv`;
`./serialize '$regex2' <  ~/speciale/runtime.trace.mbv > ~/
    speciale/runtime.serialize.mbv`;

my $runs = 5;
my $i;
my $best;

for ($i = 0; $i < $runs; $i++){
    print $i;
    $startTime = time();
    `cat ~/speciale/shorttrace | ./main '$regex'`;
    $endTime = time();

    if($i == 0 || ($endTime - $startTime) < $best){
        $best = $endTime - $startTime;
    }
}

printf("Main: takes %.3f seconds.\n", $best);

for ($i = 0; $i < $runs; $i++){
    print $i;
    $startTime = time();
    `./groupings_all '$regex' <  ~/speciale/runtime.main.mbv
    `;
    $endTime = time();

    if($i == 0 || ($endTime - $startTime) < $best){
        $best = $endTime - $startTime;
    }
}

printf("groupings_all: takes %.3f seconds.\n", $best);

for ($i = 0; $i < $runs; $i++){
    print $i;
    $startTime = time();
    `./ismatch <  ~/speciale/runtime.main.mbv`;
    $endTime = time();

    if($i == 0 || ($endTime - $startTime) < $best){
        $best = $endTime - $startTime;
    }
}

printf("ismatch: takes %.3f seconds.\n", $best);
```

```
for ($i = 0; $i < $runs; $i++){
    print $i;
    $startTime = time();
    `./trace <  ~/speciale/runtime.groupings_all.mbv`;
    $endTime = time();

    if($i == 0 || ($endTime - $startTime) < $best){
        $best = $endTime - $startTime;
    }
}

printf("trace: takes %.3f seconds.\n", $best);


$startTime = time();
`./trace2 <  ~/speciale/runtime.groupings_all.mbv`;
$endTime = time();

$best = $endTime - $startTime;

printf("trace2: takes %.3f seconds.\n", $best);

for ($i = 0; $i < $runs; $i++){
    print $i;
    $startTime = time();
    `./serialize '$regex2' <  ~/speciale/runtime.trace.mbv`;
    $endTime = time();

    if($i == 0 || ($endTime - $startTime) < $best){
        $best = $endTime - $startTime;
    }
}

printf("serialize: takes %.3f seconds.\n", $best);
```

D.3 Profiling

D.3.1 profiling.pl

```
#! /usr/bin/perl -W

use strict;


my $regex = '(.*)';
my $regex2 = '|(.*)';


`cat ~/speciale/xac | ./main '(.*)'`;
```

```
`mv gmon.out gmon.main.out`;
`cat ~/speciale/xac | ./main '(.*)' | ./groupings_all '(.*)'
    `;
`mv gmon.out gmon.groupings_all.out`;
`cat ~/speciale/shorttrace | ./main '(.*)' | ./groupings_all
    '(.*)' | ./trace2`;
`mv gmon.out gmon.trace.out`;
`cat ~/speciale/xac | ./main '(.*)' | ./groupings_all '(.*)'
    | ./trace | ./serialize '|(.*)'`;
`mv gmon.out gmon.serialize.out`;
```

E Benchmark scripts

E.1 backtrackingworstcase.pl

```
#! /usr/bin/perl -Wall

use POSIX;
use strict;
use Time::HiRes 'time';


my $startTime, my $endTime, my $result;

my $runs = 10;
my $i, my $j;
my $best;
my $n;

print "main";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `echo -n  '$text' | ./main '$regex' | ./
   ismatch`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
```

```
}

print "perl";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = ceil($i * 2.5);
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./perlmatchonly.pl '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}

print "re2";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./re2matchonly '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}


print "tcl";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
```

```
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./tclmatchonly.tcl '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}
```

E.2 backtrackingworstcase_mem.pl

```
#! /usr/bin/perl -Wall

use POSIX;
use strict;
use Time::HiRes 'time';


my $startTime, my $endTime, my $result;

my $i, my $j;
my $best;
my $n;

for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    `echo -n '$text' | valgrind --tool=massif --pages-as-
   heap=yes --massif-out-file='backtrackingworstcase/main.
   $n.out' ./main '$regex' | valgrind --tool=massif --pages
   -as-heap=yes --massif-out-file='backtrackingworstcase/
   ismatch.$n.out' ./ismatch`;

}

for($i = 1; $i <= 10; $i++){
    $n = ceil($i * 2.5);
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;
```

```
    `valgrind --tool=massif --pages-as-heap=yes --massif-out
   -file='backtrackingworstcase/perl.$n.out' ./
   perlmatchonly.pl '$regex' '$text'`;
}

for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    `valgrind --tool=massif --pages-as-heap=yes --massif-out
   -file='backtrackingworstcase/re2.$n.out' ./re2matchonly
   '$regex' '$text'`;
}


for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "a?" x $n . "a" x $n;
    my $text = "a" x $n;

    `valgrind --tool=massif --pages-as-heap=yes --massif-out
   -file='backtrackingworstcase/tcl.$n.out' ./tclmatchonly.
   tcl '$regex' '$text'`;
}
```

E.3 dfaworstcase.pl

```
#! /usr/bin/perl -Wall

use POSIX;
use strict;
use Time::HiRes 'time';


my $startTime, my $endTime, my $result;

my $runs = 10;
my $i, my $j;
my $best;
my $n;

print "main";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "(a|b)*a" . "(a|b)" x $n;
    my $text = "a" . "a" x $n;

    for ($j = 0; $j < $runs; $j++){
```

```
        $startTime = time();
        $result = `echo -n '$text' | ./main '$regex' | ./
    ismatch`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}

print "perl";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "(a|b)*a" . "(a|b)" x $n;
    my $text = "a" . "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./perlmatchonly.pl '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}

print "re2";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "(a|b)*a" . "(a|b)" x $n;
    my $text = "a" . "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./re2matchonly '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
```

```
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}

print "tcl";
print "   n   sec";
for($i = 1; $i <= 10; $i++){
    $n = $i * 60;
    my $regex = "(a|b)*a" . "(a|b)" x $n;
    my $text = "a" . "a" x $n;

    for ($j = 0; $j < $runs; $j++){
        $startTime = time();
        $result = `./tclmatchonly.tcl '$regex' '$text'`;
        $endTime = time();

        if($j == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp $result;
    print "ARGH" unless $result eq 't';
    printf("%4i %.3f\n", $n, $best);
}
```

E.4 dfaworstcase_mem.pl

```
#! /usr/bin/perl -Wall

use POSIX;
use strict;
use Time::HiRes 'time';


my $result;

my $i, my $j;
my $n;

for($i = 1; $i <= 10; $i++){
    $n = $i * 500;
    my $regex = "(a|b)*a" . "(a|b)" x $n;
    my $text = "a" . "a" x $n;

    $result = `echo -n '$text' | valgrind --tool=massif --
   pages-as-heap=yes --massif-out-file='dfaworstcase/main.
   $n.out' ./main '$regex' | valgrind --tool=massif --pages
```

```
    -as-heap=yes --massif-out-file='dfaworstcase/ismatch.$n.
    out' ./ismatch`;
     chomp $result;
     print "ARGH" unless $result eq 't';
}

for($i = 1; $i <= 10; $i++){
     $n = $i * 500;
     my $regex = "(a|b)*a" . "(a|b)" x $n;
     my $text = "a" . "a" x $n;

     $result = `valgrind --tool=massif --pages-as-heap=yes --
    massif-out-file='dfaworstcase/perl.$n.out' ./
    perlmatchonly.pl '$regex' '$text'`;

     chomp $result;
     print "ARGH" unless $result eq 't';
}

for($i = 1; $i <= 10; $i++){
     $n = $i * 500;
     my $regex = "(a|b)*a" . "(a|b)" x $n;
     my $text = "a" . "a" x $n;

     $result = `valgrind --tool=massif --pages-as-heap=yes --
    massif-out-file='dfaworstcase/re2.$n.out' ./re2matchonly
     '$regex' '$text'`;

     chomp $result;
     print "ARGH" unless $result eq 't';
}

for($i = 1; $i <= 10; $i++){
     $n = $i * 60;
     my $regex = "(a|b)*a" . "(a|b)" x $n;
     my $text = "a" . "a" x $n;

     $result = `valgrind --tool=massif --pages-as-heap=yes --
    massif-out-file='dfaworstcase/tcl.$n.out' ./tclmatchonly
    .tcl '$regex' '$text'`;

     chomp $result;
     print "ARGH" unless $result eq 't';
}
```

E.5 email.pl

```
#! /usr/bin/perl -Wall

use strict;
```

```
use Time::HiRes 'time';
use String::Random;


my $regex = '([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A-
    Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-z
    ]+))';
my $regex2 = '|([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A
    -Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-
    z]+))';
my $startTime, my $endTime, my $result;

my $foo = new String::Random;
$foo->{'A'} = [ 'A'..'Z', 'a'..'z', '0'..'9' ];
$foo->{'B'} = [ '_', '.', '-' ];
$foo->{'D'} = [ '.', '-' ];
$foo->{'E'} = [ 'A'..'Z', 'a'..'z' ];
$foo->{'F'} = [ '@' ];
$foo->{'G'} = [ '.' ];

my $n;

#my $email = $foo->randpattern("A" . ("B" x int(rand(2)) . "
    A" x int(rand($n))) x int(rand($n)) . "FA" . ("D" x int(
    rand(2)) . "A" x int(rand($n))) x int(rand($n)) . "GEE"
    . "E" x int(rand($n)));
#my $email = $foo->randpattern("A" . ("B" . ("A" x $n)) x $n
     . "FA" . "D" . (("A" x $n) x $n) . "GEE" . ("E" x $n));
my @email;
my $runs = 10;
my $i, my $j;
my $best;
my $factor = 25;

for($i = 1; $i <= 10; $i++){
    $n = $i * $factor;
    $email[$i] = $foo->randpattern("A" . ("B" . ("A" x $n))
   x $n . "FA" . "D" . (("A" x $n) x $n) . "GEE" . ("E" x
   $n));
}


print "main";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `echo -n '$email[$j]' | ./main '$regex' |
   ./groupings_all '$regex'  | ./trace | ./serialize '
```

```
    $regex2' `;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    print "ARGH" unless $result eq $email[$j];
    printf("%7i %.3f\n", length($email[$j]), $best);
}


print "tcl";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./tclmatch.tcl '$regex' '$email[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp($result);
    print "ARGH" unless $result eq $email[$j];
    printf("%7i %.3f\n", length($email[$j]), $best);
}


print "re2";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./re2match '$regex' '$email[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    print "ARGH" unless $result eq $email[$j];
    printf("%7i %.3f\n", length($email[$j]), $best);
}


print "perl";
print "    len   sec";
```

```
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./perlmatch.pl '$regex' '$email[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    print "ARGH" unless $result eq $email[$j];
    printf("%7i %.3f\n", length($email[$j]), $best);
}
```

E.6 email_mem.pl

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';
use String::Random;


my $regex = '([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A-
    Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-z
    ]+))';
my $regex2 = '|([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A
    -Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-
    z]+))';
my $result;

my $foo = new String::Random;
$foo->{'A'} = [ 'A'..'Z', 'a'..'z', '0'..'9' ];
$foo->{'B'} = [ '_', '.', '-' ];
$foo->{'D'} = [ '.', '-' ];
$foo->{'E'} = [ 'A'..'Z', 'a'..'z' ];
$foo->{'F'} = [ '@' ];
$foo->{'G'} = [ '.' ];

my $n;

#my $email = $foo->randpattern("A" . ("B" x int(rand(2)) . "
    A" x int(rand($n))) x int(rand($n)) . "FA" . ("D" x int(
    rand(2)) . "A" x int(rand($n))) x int(rand($n)) . "GEE"
    . "E" x int(rand($n)));
#my $email = $foo->randpattern("A" . ("B" . ("A" x $n)) x $n
     . "FA" . "D" . (("A" x $n) x $n) . "GEE" . ("E" x $n));
my @email;
my $i, my $j;
my $factor = 25;
```

```
for($i = 1; $i <= 10; $i++){
    $n = $i * $factor;
    $email[$i] = $foo->randpattern("A" . ("B" . ("A" x $n))
   x $n . "FA" . "D" . (("A" x $n) x $n) . "GEE" . ("E" x
   $n));
}

for($j = 1; $j <= 10; $j++){
    my $n = length($email[$j]);
    $result = `echo -n '$email[$j]' | valgrind --tool=massif
     --pages-as-heap=yes --massif-out-file='email/main.$n.
   out' ./main '$regex' | valgrind --tool=massif --pages-as
   -heap=yes --massif-out-file='email/groupings_all.$n.out'
    ./groupings_all '$regex'  | valgrind --tool=massif --
   pages-as-heap=yes --massif-out-file='email/trace.$n.out'
    ./trace | valgrind --tool=massif --pages-as-heap=yes --
   massif-out-file='email/serialize.$n.out' ./serialize '
   $regex2' `;
    print "ARGH" unless $result eq $email[$j];
}


for($j = 1; $j <= 10; $j++){
    my $n = length($email[$j]);
    $result = `valgrind --tool=massif --pages-as-heap=yes --
   massif-out-file='email/tcl.$n.out' ./tclmatch.tcl '
   $regex' '$email[$j]'`;
    chomp($result);
    print "ARGH" unless $result eq $email[$j];
}


for($j = 1; $j <= 10; $j++){
    my $n = length($email[$j]);
    $result = `valgrind --tool=massif --pages-as-heap=yes --
   massif-out-file='email/re2.$n.out' ./re2match '$regex' '
   $email[$j]'`;
    print "ARGH" unless $result eq $email[$j];
}


for($j = 1; $j <= 10; $j++){
    my $n = length($email[$j]);
    $result = `valgrind --tool=massif --pages-as-heap=yes --
   massif-out-file='email/perl.$n.out' ./perlmatch.pl '
   $regex' '$email[$j]'`;
    print "ARGH" unless $result eq $email[$j];
}
```

E.7 email_mbvsize.pl

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';
use String::Random;


my $regex = '([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A-
    Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-z
    ]+))';
my $regex2 = '|([A-Za-z0-9](?:(?:[_.-]?[a-zA-Z0-9]+)*)@(?:[A
    -Za-z0-9]+)(?:(?:[.-]?[a-zA-Z0-9]+)*)\.(?:[A-Za-z][A-Za-
    z]+))';
my $startTime, my $endTime, my $result;

my $foo = new String::Random;
$foo->{'A'} = [ 'A'..'Z', 'a'..'z', '0'..'9' ];
$foo->{'B'} = [ '_', '.', '-' ];
$foo->{'D'} = [ '.', '-' ];
$foo->{'E'} = [ 'A'..'Z', 'a'..'z' ];
$foo->{'F'} = [ '@' ];
$foo->{'G'} = [ '.' ];

my $n;

my @email;
my $runs = 1;
my $i, my $j;
my $best;
my $factor = 25;

for($i = 1; $i <= 10; $i++){
    $n = $i * $factor;
    $email[$i] = $foo->randpattern("A" . ("B" . ("A" x $n))
    x $n . "FA" . "D" . (("A" x $n) x $n) . "GEE" . ("E" x
    $n));
}


print "         main groupings_all         trace
    serialize";
for($j = 1; $j <= 10; $j++){
    my $n = length($email[$j]);
    `echo -n '$email[$j]' | ./main '$regex' > main.mbv.tmp`;
    `./groupings_all '$regex' < main.mbv.tmp > groupings_all
    .mbv.tmp`;
    `./trace < groupings_all.mbv.tmp > trace.mbv.tmp`;
```

```
    `./serialize '$regex2' < trace.mbv.tmp > serialize.mbv.
   tmp`;


    printf("%13i %13i %13i %13i\n", -s 'main.mbv.tmp',
           -s 'groupings_all.mbv.tmp', -s 'trace.mbv.tmp',
           -s 'serialize.mbv.tmp');
}
```

E.8 number.pl

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';

my $regex = '([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[eE
   ][+-]?[0-9]+)?)';
my $regex2 = '|([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[
   eE][+-]?[0-9]+)?)';
my $startTime, my $endTime, my $result;

my $n;

my @number;
my $runs = 1;
my $i, my $j;
my $best;
my $factor = 1000;

for($j = 1; $j <= 10; $j++){
    $n = $factor * $j;
    $number[$j] = '+' . int(rand($n)) x $n . '.' . int(rand(
   $n)) x $n . 'E-' . int(rand($n)) x $n;
}

print "main";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./main '$regex' '$number[$j]' | ./
   groupings_all '$regex' | ./trace | ./serialize '$regex2'
    `;
        $endTime = time();
        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    print "ARGH" unless $result eq $number[$j];
```

```
    printf("%7i %.3f\n", length($number[$j]), $best);
}


print "tcl";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./tclmatch.tcl '$regex' '$number[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    chomp($result);
    print "ARGH" unless $result eq $number[$j];
    printf("%7i %.3f\n", length($number[$j]), $best);
}


print "re2";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./re2match '$regex' '$number[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
        }
    }
    print "ARGH" unless $result eq $number[$j];
    printf("%7i %.3f\n", length($number[$j]), $best);
}


print "perl";
print "    len   sec";
for($j = 1; $j <= 10; $j++){
    for($i = 0; $i < $runs; $i++){
        $startTime = time();
        $result = `./perlmatch.pl '$regex' '$number[$j]'`;
        $endTime = time();

        if($i == 0 || ($endTime - $startTime) < $best){
            $best = $endTime - $startTime;
```

```
        }
    }
    print "ARGH" unless $result eq $number[$j];
    printf("%7i %.3f\n", length($number[$j]), $best);
}
```

E.9 number_mem.pl

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';

my $regex = '([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[eE
    ][+-]?[0-9]+)?)';
my $regex2 = '|([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[
    eE][+-]?[0-9]+)?)';
my $startTime, my $endTime, my $result;

my $n;

my @number;
my $i, my $j;
my $factor = 1000;

for($j = 1; $j <= 10; $j++){
    $n = $factor * $j;
    $number[$j] = '+' . int(rand($n)) x $n . '.' . int(rand(
    $n)) x $n . 'E-' . int(rand($n)) x $n;
}

for($j = 1; $j <= 10; $j++){
    my $n = length($number[$j]);
    `valgrind --tool=massif --pages-as-heap=yes --massif-out
    -file='number/main.$n.out' ./main '$regex' '$number[$j]'
     | valgrind --tool=massif --pages-as-heap=yes --massif-
    out-file='number/groupings_all.$n.out' ./groupings_all '
    $regex' | valgrind --tool=massif --pages-as-heap=yes --
    massif-out-file='number/trace.$n.out' ./trace | valgrind
     --tool=massif --pages-as-heap=yes --massif-out-file='
    number/serialize.$n.out' ./serialize '$regex2' `;
}


for($j = 1; $j <= 10; $j++){
    my $n = length($number[$j]);
    `valgrind --tool=massif --pages-as-heap=yes --massif-out
    -file='number/tcl.$n.out' ./tclmatch.tcl '$regex' '
    $number[$j]'`;
}
```

```
for($j = 1; $j <= 10; $j++){
    my $n = length($number[$j]);
    `valgrind --tool=massif --pages-as-heap=yes --massif-out
   -file='number/re2.$n.out' ./re2match '$regex' '$number[
   $j]'`;
}


for($j = 1; $j <= 10; $j++){
    my $n = length($number[$j]);
    `valgrind --tool=massif --pages-as-heap=yes --massif-out
   -file='number/perl.$n.out' ./perlmatch.pl '$regex' '
   $number[$j]'`;
}
```

E.10 number_mbvsize.pl

```
#! /usr/bin/perl -Wall

use strict;
use Time::HiRes 'time';

my $regex = '([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[eE
   ][+-]?[0-9]+)?)';
my $regex2 = '|([+-]?(?:[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)(?:[
   eE][+-]?[0-9]+)?)';

my $n;

my @number;
my $i, my $j;
my $factor = 1000;

for($j = 1; $j <= 10; $j++){
    $n = $factor * $j;
    $number[$j] = '+' . int(rand($n)) x $n . '.' . int(rand(
   $n)) x $n . 'E-' . int(rand($n)) x $n;
}

print "         main groupings_all         trace
   serialize";
for($j = 1; $j <= 10; $j++){
    `echo -n '$number[$j]' | ./main '$regex' > main.mbv.tmp
   `;
    `./groupings_all '$regex' < main.mbv.tmp > groupings_all
   .mbv.tmp`;
    `./trace < groupings_all.mbv.tmp > trace.mbv.tmp`;
```

```
    `./serialize '$regex2' < trace.mbv.tmp > serialize.mbv.
   tmp`;


    printf("%13i %13i %13i %13i\n", -s 'main.mbv.tmp',
           -s 'groupings_all.mbv.tmp', -s 'trace.mbv.tmp',
           -s 'serialize.mbv.tmp');
}
```