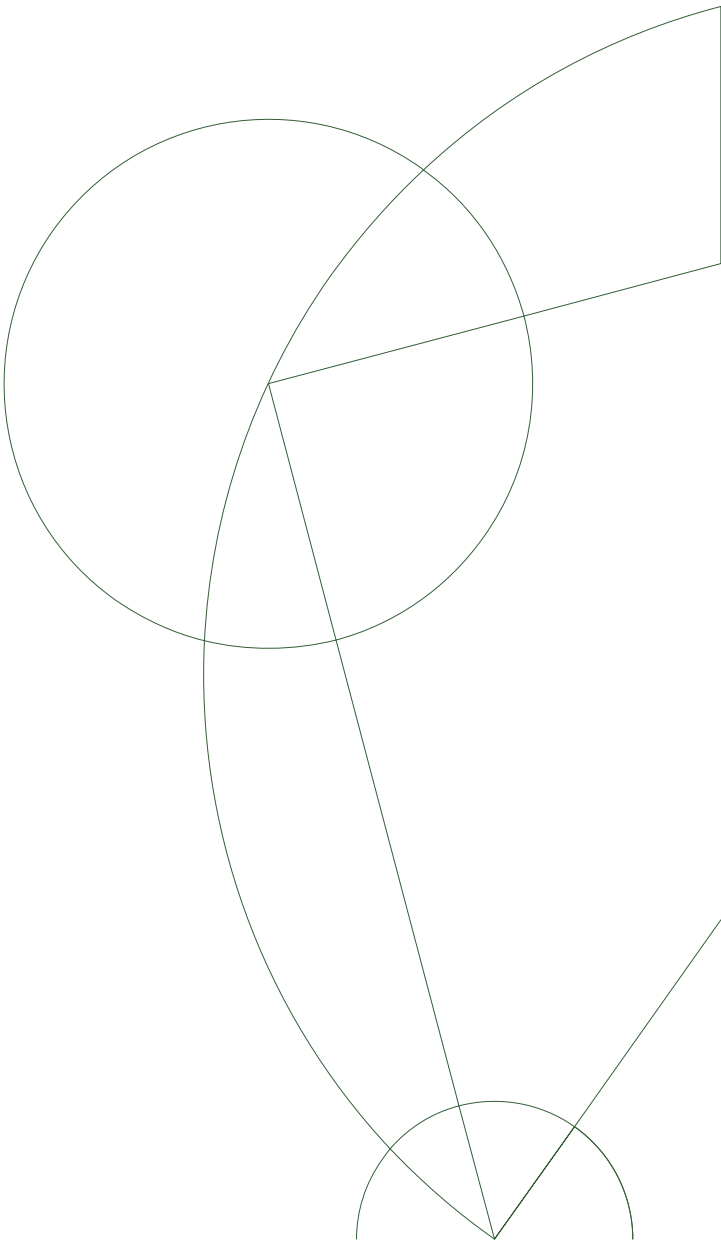
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor's Thesis

Arni Asgeirsson & Peter Troelsen

Regular Expression Parsing

From Two Phases to $1 + \epsilon$ Phases



Fritz Henglein & Niels Bjørn Bugge Grathwohl

09/06 - 2014

Abstract

We present an improved algorithm for producing greedy parses for regular expressions in a semi-streaming fashion. Our algorithm improves on a recently published algorithm, called the lean-log algorithm, by the KMC group at DIKU, by minimizing the number of active states, an thereby the memory usage during parsing. Our algorithm furthermore introduces commit points as the ability to recognize and generate correct partial parse trees on the fly. We perform a minor analyze on the frequency of commit points on real-life data and finds that the applicability of algorithm varies by the regular expressions, but still looks very promising.

Contents

1 Introduction

Regular expression (RE) parsing is an extension to the traditional RE matching. The difference between typical RE matching and RE parsing is that when doing parsing, one is not only interested in whether the input string is accepted, but *how* it matches the RE, by listing all possible parse trees. The reason for possibly having multiple parse trees is the fact that REs can be ambiguous, however normally one is not interested in all the parse trees, but just *one* based on some deterministic choice.

The KMC group at DIKU have published a new algorithm [3] for doing greedy left-most RE parsing. The greedy left-most disambiguation strategy tries to parse as much of the input string as early in the RE as possible. This new algorithm uses a new variant of the Thompson NFA, called augmented NFA (aNFA), to create a symmetric construction, with a one-to-one correspondence with the parse tree and the underlying RE. The algorithm is very memory efficient, but suffers from being a *two phase* algorithm, by having problems dealing with very long input strings.

In this Thesis we work with an improvement of the algorithm, based on unpublished work done by the KMC group, and present the following contributions:

1. A discussion and definition of a new simulation strategy which when used in conjunction with the aNFA gives rise to *commit points*, which is the ability to return correct partial parse trees during the first phase of the parse, allowing for the algorithm to reduce the two phases to "$1+\epsilon$ phases", meaning dividing the task into smaller task, such that one do not need to finish the entire first phase, before starting the second phase.

2. An empirical analysis on the number of commit points when using the improved algorithm with "real-life" data, showing how we found the amount of commit points to heavily depend on the construction of the RE and the length of the input.

Furthermore we will have a discussion on the pitfalls of the improved algorithm and how it does not catch all desired commit points. We will discuss the extension of using the new algorithm and simulation strategy to build a semi-streaming system, and show a simple proof-of-concept implementation.

2 Regular Expressions

Before defining a regular expression, we give a short introduction to languages and alphabets. An alphabet Σ is a finite set of characters, which can be used to form words in a language. An example of an alphabet could be $\{a, b\}$, and so a word from this alphabet could be anything containing only these elements, like a and baa. It is clear that the set of all words from any given alphabet is not finite, as we can produce a infinite number of combinations of symbols from a nonempty alphabet. A language is then defined as a subset of all words that can be produced from this alphabet [1].

Given an alphabet Σ, a regular expression is an algebraical notation. One can think of regular expressions as an abstract syntax definition. Regular expressions are defined as follows [2, p. 2]:

Definition 1. A *regular expression* (RE) over an alphabet Σ is an expression of the form

$$E = 0 \mid 1 \mid a \mid E_1 + E_2 \mid E_1 E_2 \mid E_0^*,$$

where $a \in \Sigma$, and E and F are both Regular Expressions[1]. □

[1] Sometimes a '|' operator is used instead of '+', both should be pronounced 'or'.

This definition does not tell much about how to express a language. Instead we can give a language interpretation:

Definition 2. The *language* $\mathcal{L}(E)$ produced from some alphabet Σ denoted by E is:

$$\begin{array}{rclcrcl} \mathcal{L}(0) & = & \emptyset & & \mathcal{L}(E+F) & = & \mathcal{L}(E) \cup \mathcal{L}(F) \\ \mathcal{L}(1) & = & \{\epsilon\} & & \mathcal{L}(EF) & = & \mathcal{L}(E) \odot \mathcal{L}(F) \\ \mathcal{L}(a) & = & \{a\} & & \mathcal{L}(E^*) & = & \bigcup_{i\geq 0} (\mathcal{L}(E))^i, \end{array}$$

where $\mathcal{L}(E) \odot \mathcal{L}(F) = \{w_1 w_2 \,|w_1 \in \mathcal{L}(E),\, w_2 \in \mathcal{L}(F)\}$, $E^0 = \{\epsilon\}$, $E^{i+1} = E \odot E^i$ and $a \in \Sigma$. [4, def. 3] □

With this definition we can create any regular language, since a regular language is defined as a language that can be expressed from a RE.

Example 1. Looking back at the example from before, with the alphabet $\Sigma = \{a, b\}$. From Σ we could easily define a language:

$$L = \{w \mid w \text{ starts with } a\text{followed by an arbitrary amount of } b\text{s}\}.$$

This could be expressed as the RE $E_1 = ab^*$. We can then try to define the language denoted by E_1 as

$$\begin{aligned} \mathcal{L}(E_1) &= \mathcal{L}(ab^*) \\ &= \{w_1 w_2 \mid w_1 \in \mathcal{L}(a),\, w_2 \in \mathcal{L}(b*)\} \\ &= \left\{ a\, w_2 \,\middle|\, w_2 \in \bigcup_{i\geq 0} (\mathcal{L}(b))^i \right\} \\ &= \left\{ a\, w_2 \,\middle|\, w_2 \in \bigcup_{i\geq 0} (b)^i \right\} = \{a, ab, abb, abbb, ...\} = L, \end{aligned}$$

where i is an arbitrary number, giving $L = \mathcal{L}(E_1)$. □

Example 2. We can also define a RE for all natural numbers $n > 0$ from the alphabet of all digits $\{0..9\}$, as

$$E_2 = (1+2+3+4+5+6+7+8+9)\,(0+1+2+3+4+5+6+7+8+9)^*$$

□

The above examples show that REs can be used for specifying languages. For a RE E we say that an input string s *matches* E iff $s \in \mathcal{L}(E)$.

Type interpretation are useful as interpretations for *how* a word matches a RE. This could also be referred to as the *path* one needs to take through the RE, which is defined as follows:

Definition 3. The *type interpretation* $\mathcal{T}(\cdot)$ of REs is denoted by:

$$\begin{array}{rcll} \mathcal{T}(0) & = & \emptyset & \text{empty type} \\ \mathcal{T}(1) & = & \{()\} & \text{unit type} \\ \mathcal{T}(a) & = & \{a\} & \text{singleton type} \\ \mathcal{T}(E+F) & = & \mathcal{T}(E) + \mathcal{T}(F) & \text{sum type} \\ \mathcal{T}(EF) & = & \mathcal{T}(E) \times \mathcal{T}(F) & \text{product type} \\ \mathcal{T}(E^*) & = & \{[v_0, \ldots, v_n] \,|v_i \in \mathcal{T}(E)\} & \text{list type,} \end{array}$$

where $T + U = \{\mathsf{inl}\ v \mid v \in T\} \cup \{\mathsf{inr}\ w \mid w \in U\}$ and $T \times U = \{(v, w) \mid v \in T \wedge w \in U\}$ [3, fig. 1(a)]. □

The inl constructor can be interpreted as "taking the left option", just like the inr constructor can be interpreted as "taking the right option".

The list type is a shorthand for the head of the list combined with the tail of the list or the empty list. This interpretation can be written as:

$$\mathcal{T}(E^*) = \mathcal{T}(EE^*) + \mathcal{T}(1),$$

so writing this we have the empty list 1 as $\mathsf{inr}\ ()$ or the head x and the tail xs written as $\mathsf{inl}\ (x, xs)$. This can be compared to the Haskell List type, defined as `data [] a = a : [a] | []`.

Example 3. An example of rewriting the list type is the expression $E_2 = z^*$, which giving an input of a arbitrary amounts of z's would give the parse tree $[z, ..., z]$, by the following steps

$$\begin{aligned}\mathcal{T}(E_2) &= \mathcal{T}(z*) \\ &= \{[a_0, ..., a_n] \mid a_i \in \mathcal{T}(z)\} \\ &= \{[z_0, ..., z_n]\} \text{ for any } n \geq 0,\end{aligned}$$

this could however also be written using the sum operator:

$$\begin{aligned}\mathcal{T}(E_2) &= \mathcal{T}(z^*) \\ &= \mathcal{T}(zz^*) + \mathcal{T}(1) \\ &= \mathsf{inl}\ (\mathcal{T}(z), \mathcal{T}(z^*)) \\ &= \{\mathsf{inl}\ (z, \mathsf{inr}\ ()), \mathsf{inl}\ (z, \mathsf{inl}\ (z, \mathsf{inr}\ ()))\, , ...\}\end{aligned}$$

□

Let us define a function flat$(\cdot)$, for going from parse tree, to the underlying strings.

Definition 4. The function flat$(\cdot)$, flatting a value to its underlying string, is defined as:

$$\begin{aligned}\mathrm{flat}\,(()) &= \epsilon & \mathrm{flat}\,(\mathsf{inl}\ v) &= \mathrm{flat}\,(v) \\ \mathrm{flat}\,(a) &= a & \mathrm{flat}\,(\mathsf{inr}\ v) &= \mathrm{flat}\,(v) \\ \mathrm{flat}\,((v, w)) &= \mathrm{flat}\,(v)\,\mathrm{flat}\,(w)\,.\end{aligned}$$

[4, def. 5] □

Example 4. Let us combine the above definitions in an example. Consider the RE $E_3 = (a + 1)(ab + b)$ where $\mathcal{L}(E_3) = \{b, ab, aab\}$. The set of parse trees to E_3 is:

$$\begin{aligned}\mathcal{T}(E_3) &= \mathcal{T}((a + 1)(ab + b)) \\ &= \mathcal{T}(a + 1) \times \mathcal{T}(ab + b) \\ &= \{\mathsf{inl}\ a, \mathsf{inr}\ ()\} \times \{\mathsf{inl}\ (a, b), \mathsf{inr}\ b\} \\ &= \{(\mathsf{inl}\ a, \mathsf{inl}\ (a, b)), (\mathsf{inl}\ a, \mathsf{inr}\ b), (\mathsf{inr}\ (), \mathsf{inl}\ (a, b)), (\mathsf{inr}\ (), \mathsf{inr}\ b)\}\,.\end{aligned}$$

We can find the underlying strings, using flat(v) on each element $v \in \mathcal{T}(E_3)$, which gives us the

following set of strings

$$\begin{aligned}
\text{flat}\,(\mathcal{T}\,(E_3)) &= \text{flat}\,(\{(\mathsf{inl}\ a, \mathsf{inl}\ \ (a,b))\,, (\mathsf{inl}\ a, \mathsf{inr}\ b)\,, (\mathsf{inr}\ \ ()\,, \mathsf{inl}\ \ (a,b))\,, (\mathsf{inr}\ \ ()\,, \mathsf{inr}\ b)\}) \\
&= \{\text{flat}\,(\mathsf{inl}\ a, \mathsf{inl}\ \ (a,b))\,,\ \text{flat}\,(\mathsf{inl}\ a, \mathsf{inr}\ b)\,, \\
&\quad\ \text{flat}\,(\mathsf{inr}\ \ ()\,, \mathsf{inl}\ \ (a,b))\,,\ \text{flat}\,(\mathsf{inr}\ \ ()\,, \mathsf{inr}\ b)\} \\
&= \{\text{flat}\,(\mathsf{inl}\ a)\ \text{flat}\,(\mathsf{inl}\ \ (a,b))\,, \text{flat}\,(\mathsf{inl}\ a)\ \text{flat}\,(\mathsf{inr}\ b)\,, \\
&\quad\ \text{flat}\,(\mathsf{inr}\ \ ())\ \text{flat}\,(\mathsf{inl}\ \ (a,b))\,, \text{flat}\,(\mathsf{inr}\ \ ())\ \text{flat}\,(\mathsf{inr}\ b)\} \\
&= \{\text{flat}\,(a)\ \text{flat}\,(a)\ \text{flat}\,(b)\,, \text{flat}\,(a)\ \text{flat}\,(b)\,, \\
&\quad\ \text{flat}\,(())\ \text{flat}\,(a)\ \text{flat}\,(b)\,, \text{flat}\,(())\ \text{flat}\,(b)\} \\
&= \{aab,\ ab,\ ab,\ b\} = \mathcal{L}\,(E_3)
\end{aligned}$$

□

The above example shows that there are different ways to express the same string, using the same RE. As mentioned in the introduction to this Thesis, REs can be ambiguous, and this is an example of exactly that. An ambiguous RE is defined as the RE E with more than one parse tree for a word in $\mathcal{L}(E)$.

Definition 5. A RE E is ambiguous iff multiple elements from $\mathcal{T}\,(E)$ flatten to the same string:

$$\exists v, w \in \mathcal{T}\,(E) \text{ s.t. } v \neq w \wedge \text{flat}\,(v) = \text{flat}\,(w)\,.$$

□

As long as we are only doing matching, it does not matter if the regular expression is ambiguous or not. This can however cause problems when doing RE parsing, as we will explain in Section 4.

2.1 Regular Expression Parsing and Bit-Coding

We are now ready to define a method for regular expression parsing. Regular expression parsing is a question of finding the parse tree for an input string s relative to some RE.

Definition 6. The result of a parse of the string s under the RE E, is *some* parse tree $v \in V$ where

$$V = \{v \mid s = \text{flat}\,(v) \wedge v \in \mathcal{T}\,(E)\}\,.$$

It can be noted that given Definition 5, if V contains more than one element, E is ambiguous. □

A parse tree holds redundant information, since we already know the regular expression and the input string, the only *new* information that lies in the parse tree is the inl and inr constructors. As explained earlier, inl can be interpreted as taking the left option, just as inr can be interpreted as taking the right option. Since taking the left or right option is a binary choice, it is possible to define a binary bit-coding for representing a parse tree.

Definition 7. The function $\mathcal{B}\,(\cdot)$ that compresses a parse tree for a RE E to bit-code is defined as follows:

$$\begin{aligned}
\mathcal{B}\,(()) &= \epsilon & \qquad \mathcal{B}\,(\mathsf{inl}\ v) &= 0\,\mathcal{B}\,(v) \\
\mathcal{B}\,(a) &= \epsilon & \qquad \mathcal{B}\,(\mathsf{inr}\ v) &= 1\,\mathcal{B}\,(v) \\
\mathcal{B}\,((v,w)) &= \mathcal{B}\,(v)\,\mathcal{B}\,(w)\,, & &
\end{aligned}$$

where $v, w \in \mathcal{T}(E)$ [3, sec. 2]. □

From the definition above, we can now generate a bit-code describing the path through the regular expression, and likewise by having only the RE and the bit-coding we can figure out what the input string was.

Example 5. Consider the regular expression $E_4 = (ab)^* (c + d)$, and the input string $s = ababd$. Going through all elements of $\mathcal{T}(E_4)$ we find a parse tree v_1 using Definition 6:

$$v_1 = (\mathsf{inl}\ ab, (\mathsf{inl}\ ab, (\mathsf{inr}\ (), (\mathsf{inr}\ d)))),$$

which makes it possible to determine the bit-code as:

$$\begin{aligned}
\mathcal{B}(v_1) &= \mathcal{B}((\mathsf{inl}\ ab, (\mathsf{inl}\ ab, (\mathsf{inr}\ (), (\mathsf{inr}\ d)))))\\
&= 0\,\mathcal{B}((\mathsf{inl}\ ab, (\mathsf{inr}\ (), (\mathsf{inr}\ d))))\\
&= 00\,\mathcal{B}((\mathsf{inr}\ (), (\mathsf{inr}\ d)))\\
&= 001\,\mathcal{B}((\mathsf{inr}\ d))\\
&= 0011.
\end{aligned}$$

□

Note that a bit-code is relative to the RE, since the bit-code would most likely match other REs as well.

Example 6. Let us consider yet another RE $E_5 = ((a + b) + c)((d + e) + (f + g))$, and try to match the input string $s = ag$, to create some bit-code b. First we get the parse tree:

$$v_2 = ((\mathsf{inl}\ (\mathsf{inl}\ a,)), (\mathsf{inr}\ (\mathsf{inr}\ g))),$$

and then the bit-code $b = \mathcal{B}(v_2)$ by:

$$\begin{aligned}
\mathcal{B}(v_2) &= \mathcal{B}(((\mathsf{inl}\ (\mathsf{inl}\ a,)), (\mathsf{inr}\ (\mathsf{inr}\ g))))\\
&= \mathcal{B}(\mathsf{inl}\ (\mathsf{inl}\ a,))\,\mathcal{B}(\mathsf{inr}\ (\mathsf{inr}\ g))\\
&= 0\,\mathcal{B}(\mathsf{inl}\ a,)\ 1\,\mathcal{B}(\mathsf{inr}\ g)\\
&= 0011.
\end{aligned}$$

We now see how the bit-code from Example 5 and b are equal even though the REs and input strings are different, hence we need the RE to get any useful information from the bit-code.

□

The core purpose of doing bit-coding is to save space, although we first have to construct a full parse tree $v \in \mathcal{T}(E)$ before compressing it with $\mathcal{B}(v)$. This can be avoided as we will explain in Section 4.

3 Finite Automata

3.1 Non-Deterministic Finite Automata

We give the notion of a non-deterministic finite automaton.

Definition 8. A Thompson *non-deterministic finite automaton* (NFA) is a 5-tuple $M = \left(Q, \Sigma, \Delta, q^s, q^f\right)$ where Q is the set of states. Σ is the input alphabet. q^s is the initial state and q^f is the accepting state. The transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ contains directed, labeled transitions, i.e.

Δ is the set of all transitions and $(q, \gamma, q') \in \Delta$ is the transition from q to q' with the label γ, written $q \xrightarrow{\gamma} q'$. We write $q \overset{\delta}{\rightsquigarrow} q'$ when there is a path from q to q' consisting only of 0 or more δ transitions, where δ is some label. We write $\mathcal{N}\left\langle E, q^s, q^f \right\rangle$ to be the Thompson NFA corresponding to E with initial state q^s and accepting state q^f.

Thompson NFAs are constructed according to the following rules:

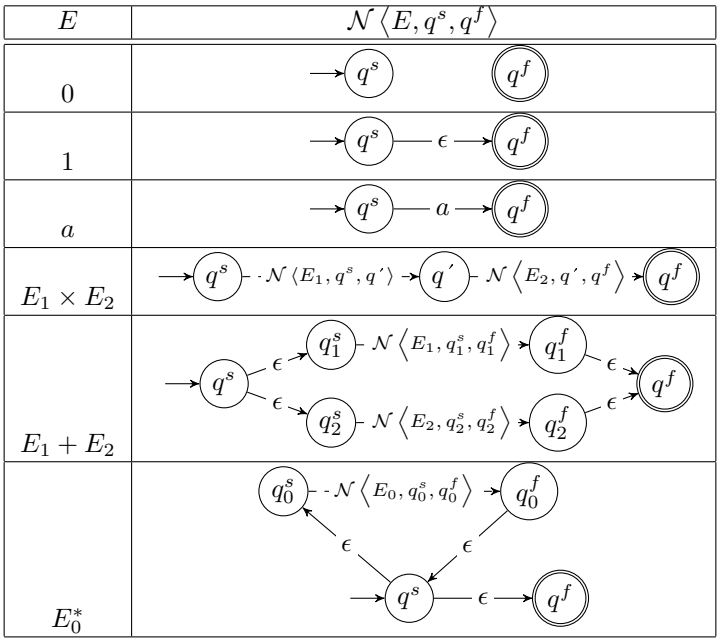


Figure 1: Construction rules of the Thompson NFA.

[6, def. 2] □

The Thompson NFA construction transforms a given RE E to a state machine that can be used to match any input string against E.

3.1.1 NFA-Simulation

Definition 9. Simulation of a NFA N with initial state q^s and accepting state q^f on the input string s is to check $q^f \in Reach(Close(\{q^s\}), s)$ where:

$$\begin{aligned} Reach(S, \epsilon) &= S \\ Reach(S, as') &= Reach(Reach(S, a), s') \\ Reach(S, a) &= Close(Step(S, a)) \\ Step(S, a) &= \{q' \mid q \in S, q \xrightarrow{a} q'\} \\ Close(S) &= \{q' \mid q \in S, q \overset{\epsilon}{\rightsquigarrow} q'\}, \end{aligned}$$

we also call the function $Close(\cdot)$ for ϵ-closure.[3, sec. 4] □

Note that standard NFA-simulation does not care about nor deal with ambiguity, its goal is to check whether the accepting state is reached by the end of the given input.

Example 7. Consider the RE $E_6 = (aa + a)^*$ and the corresponding NFA shown in Figure 2, where $q^s = 0$, $q^f = 8$ and the input string $s = aaa$.

To know whether s matches E_6 we must check $\{q^f\} \in Reach(Close(\{q^s\}), s)$:

$$\begin{aligned}
Reach(Close(\{0\}), s) &= Reach(\{0,1,2,5,8\}, aaa) \\
&= Reach(Reach(\{0,1,2,5,8\}, a), aa) \\
&= Reach(Close(Step(\{0,1,2,5,8\}, a)), aa) \\
&= Reach(Close(\{3,6\}), aa) \\
&= Reach(\{3,6,7,1,2,5,8\}, aa) \\
&= Reach(Reach(\{3,6,7,1,2,5,8\}, a), a) \\
&= Reach(Close(Step(\{3,6,7,1,2,5,8\}, a)), a) \\
&= Reach(Close(\{4,3,6\}), a) \\
&= Reach(\{4,7,1,2,5,8,3,6\}, a) \\
&= Reach(Reach(\{4,7,1,2,5,8,3,6\}, a), \epsilon) \\
&= Reach(Close(Step(\{4,7,1,2,5,8,3,6\}, a)), \epsilon) \\
&= Reach(Close(\{3,6,4\}), \epsilon) \\
&= Reach(\{3,6,7,1,2,5,8,4\}, \epsilon) \\
&= \{3,6,7,1,2,5,8,4\}.
\end{aligned}$$

As the accepting state 8 is in the final active state set we can conclude that s is contained in $\mathcal{L}((aa + a)^*)$.

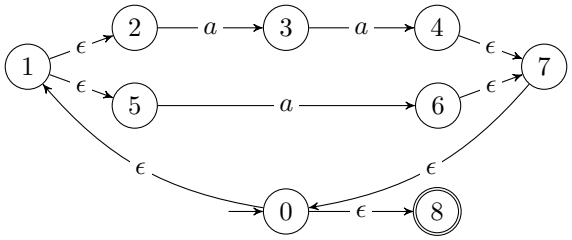


Figure 2: The Thompson NFA representation of the RE $(aa + a)^*$.

□

Example 8. Consider the RE $E_7 = (abc^*)^*d$ and the NFA N shown in Figure 3 where $q^s = \{0\}$ and $q^f = \{8\}$.

Trying to simulate N with the input $s = abcad$, we can see that the accepting state is unreachable as, after matching the second a character, there is no outgoing character transition with label d and hence the input is rejected:

$$
\begin{aligned}
Reach(Close(\{0\}), s) &= Reach(\{0,1,7\}, abcad) \\
&= Reach(Reach(\{0,1,7\}, a), bcad) \\
&= Reach(Close(Step(\{0,1,7\}, a)), bcad) \\
&= Reach(Close(\{2\}), bcad) \\
&= Reach(\{2\}, bcad) \\
&= Reach(Reach(\{2\}, b), cad) \\
&= Reach(Close(Step(\{2\}, b)), cad) \\
&= Reach(Close(\{3\}), cad) \\
&= Reach(\{3,4,6,0,1,7\}, cad) \\
&= Reach(Reach(\{3,4,6,0,1,7\}, c), ad) \\
&= Reach(Close(Step(\{3,4,6,0,1,7\}, c)), ad) \\
&= Reach(Close(\{5\}), ad) \\
&= Reach(\{5,3,4,6,0,1,7\}, ad) \\
&= Reach(Reach(\{5,3,4,6,0,1,7\}, a), d) \\
&= Reach(Close(Step(\{5,3,4,6,0,1,7\}, a)), d) \\
&= Reach(Close(\{2\}), d) \\
&= Reach(\{2\}, d) \\
&= Reach(Reach(\{2\}, d), \epsilon) \\
&= Reach(Close(Step(\{2\}, d)), \epsilon) \\
&= Reach(Close(\{\}), \epsilon) \\
&= Reach(\{\}, \epsilon) \\
&= \emptyset
\end{aligned}
$$

As the final state set is empty then q^f is unreachable and s is not contained in $\mathcal{L}(E_7)$.

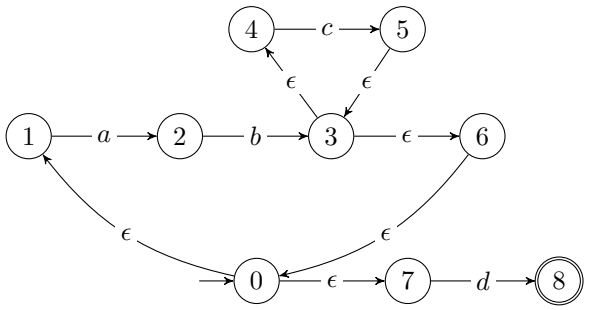


Figure 3: The Thompson NFA representation of the RE $(abc^*)^*d$.

□

3.1.2 Ambiguity & Disambiguation Strategies

If we look at the NFA construction for E_6, shown in Figure 2, and we are given the input string aaa, we cannot know whether we should first match an aa then an a or the other way around,

or even three a's.

This is the *non-deterministic* part of NFA and we need some disambiguation strategy to deal with this. We will use the greedy left-most disambiguation strategy throughout the rest of this Thesis. In the case of the previously example we would first parse an aa followed by a single a.

Greedy Left-Most Disambiguation Strategy

When using the greedy left-most disambiguation strategy, one deals with ambiguity by always trying the left choice before trying the right choice. This gives the behavior of a regular backtracking or recursive descent parser in the sense that if the left choice was not applicable, then we go back to the most recent split and try the right choice instead.

3.2 Augmented NFA

The *augmented non-deterministic finite automaton* (aNFA) was first described in [3]. It introduces the bit-coding representation described in 2.1 to the Thompson NFA. The aNFA makes small alterations to the Thompson NFA construction to incorporate the bit-coding.

3.2.1 aNFA Construction

Definition 10. aNFA extends upon Definition 8 on the Thompson NFA by changing the transition relation Δ to be $\Delta \subseteq Q \times (\Sigma \cup \{1, 0, \bar{1}, \bar{0}, \epsilon\}) \times Q$.

We call the transition labels in $\{0, 1\}$ *output labels* and labels in $\{\bar{0}, \bar{1}\}$ *log labels*. We call transitions with output labels *logging edges* and transitions with log labels *choice edges*. We call state $q \in Q$ a *join state* when $\exists q_1, q_2 \in Q\,(q_1, \bar{0}, q), (q_2, \bar{1}, q) \in \Delta$ and $q \in Q$ a *split state* when $\exists q_1, q_2 \in Q\,(q, 0, q_1), (q, 1, q_2) \in \Delta$.

To build an aNFA based on some given RE we use the same procedure as building a regular Thompson NFA, although we use the *augmented* construction rules instead, which are shown in Figure 4. We write $\mathcal{M}\left\langle E, q^s, q^f \right\rangle$ to be the aNFA corresponding to E with start state q^s and end state q^f.

aNFAs differ from Thompson NFAs in the construction of Kleene stars, for the reason of maintaining the symmetry of the bit codes.

When working with aNFA it is implicit that ϵ-closure is redefined to be $\{q' \mid q \in S, q \overset{p}{\rightsquigarrow} q', p \in \{0, 1, \bar{0}, \bar{1}, \epsilon\}\}$.[3, def. 1,2] □

Example 9. Figure 5 shows the aNFA for the RE $E_1 = ab^*$.

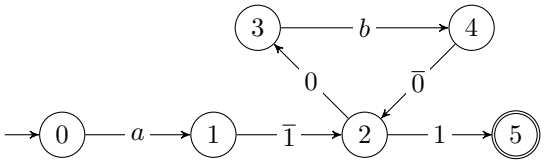


Figure 5: ab^* constructed as an aNFA.

□

Example 10. Figure 6 shows the aNFA for the RE $E_6 = (aa + a)^*$.

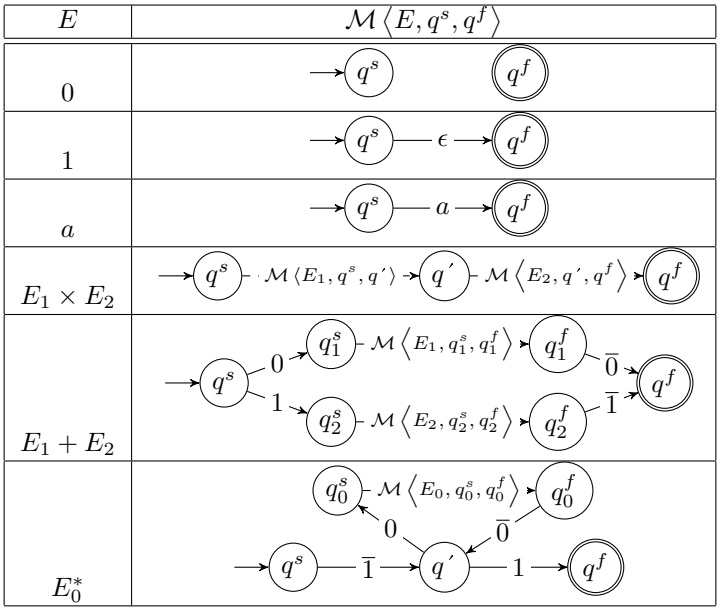


Figure 4: Construction rules of the augmented NFA.

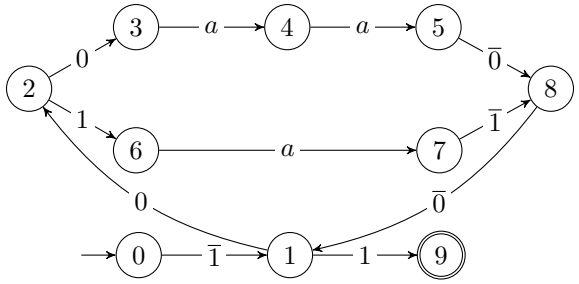


Figure 6: $(aa + a)^*$ constructed as an aNFA.

□

4 Two-Pass Greedy Regular Expression Parsing

This section is a description of the relatively new lean-log algorithm presented in [3] and further described in [4], and therefore this section is purely based on [3] and [4].

First of all, this algorithm is a two phase algorithm, which in this case means that the algorithm is constructed from a forward pass and a backward pass. The purpose of the forward pass is to construct a *lean-log stack,* explained later in this section, making the backward pass able to construct a bit-code, like the one explained in Section 2.1.

4.1 Forward pass

The goal of the forward pass is to construct a lean log stack, used to describe a path from the accepting state, to the initial state. The forward pass follows a structure of initially doing an

ϵ-closure, followed by repeatedly doing character transactions and ϵ-closures.

We call this structure *left projection*, as we after each iteration end up on the left side of the character transition.

Definition 11. Let *left projection* be defined as computing $Close(Step(S, a))$ where S is the current set of active states and a is the next character in the input. □

Definition 12. We could likewise define *right projection* to be computing $Step(Close(S), a)$ where S is the current set of active states and a is the next character in the input. □

When doing RE parsing, we need to be able to define the path taken through the aNFA. One way of doing this is by using the bit-coding, as explained in Section 2.1. Another more intuitive way to describe the path is to list all visited states, and the transitions crossed to do this.

Definition 13. Let a *path* be a list of one or more states and the transitions crossed in the parse. Two states next to each other in the list needs to be states directly connected by one transition $\xrightarrow{a}$, where a is the label of that transition. □

Example 11. Looking at the RE $E_8 = (a + a)(b + bb)$, we could consider parsing the string abb. For the aNFA describing E_8, see Figure 7.

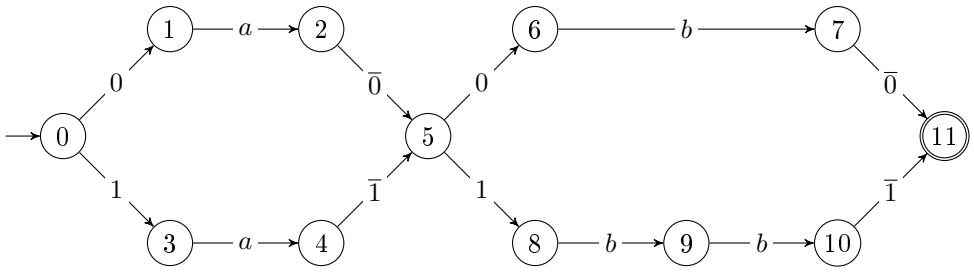


Figure 7: The regular expression E_8 as a $aNFA$

We start by doing an ϵ-closure from the initial state 0, to all other nodes, giving the three paths:

$$\{0, \quad 0 \xrightarrow{0} 1, \quad 0 \xrightarrow{1} 3\}.$$

It is trivial from the aNFA constructions that it is not possible to have both a character transition edge and something that is not a character transition edge going from the same state. This means we only have to keep the paths where the end state is not present in any other path, reducing the amount of paths to the two paths, this is implicit done in the following:

$$\{0 \xrightarrow{0} 1,\ 0 \xrightarrow{1} 3\}.$$

We do a character transition followed by an ϵ-closure, giving the four possible paths:

$$\begin{aligned}\{&0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\overline{0}} 5 \xrightarrow{0} 6,\\ &0 \xrightarrow{1} 3 \xrightarrow{a} 4 \xrightarrow{\overline{1}} 5 \xrightarrow{0} 6,\\ &0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\overline{0}} 5 \xrightarrow{1} 8,\\ &0 \xrightarrow{0} 3 \xrightarrow{a} 4 \xrightarrow{\overline{1}} 5 \xrightarrow{1} 8\}.\end{aligned}$$

Then, parsing a b gives us:

$$\begin{aligned}\{&0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\bar{0}} 5 \xrightarrow{0} 6 \xrightarrow{b} 7 \xrightarrow{\bar{0}} 11,\\ &0 \xrightarrow{1} 3 \xrightarrow{a} 4 \xrightarrow{\bar{1}} 5 \xrightarrow{0} 6 \xrightarrow{b} 7 \xrightarrow{\bar{0}} 11,\\ &0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\bar{0}} 5 \xrightarrow{1} 8 \xrightarrow{b} 9,\\ &0 \xrightarrow{1} 3 \xrightarrow{a} 4 \xrightarrow{\bar{1}} 5 \xrightarrow{1} 8 \xrightarrow{b} 9\}.\end{aligned}$$

Finally parsing the last character b gives us the final paths:

$$\begin{aligned}\{&0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\bar{0}} 5 \xrightarrow{1} 8 \xrightarrow{b} 9 \xrightarrow{b} 10 \xrightarrow{\bar{1}} 11,\\ &0 \xrightarrow{1} 3 \xrightarrow{a} 4 \xrightarrow{\bar{1}} 5 \xrightarrow{1} 8 \xrightarrow{b} 9 \xrightarrow{b} 10 \xrightarrow{\bar{1}} 11\}.\end{aligned}$$

We can express these paths as parse trees, using the knowledge gained in the previous sections. What we do, is logging going left as inl and going right as inr, furthermore we create tuples by the defined rules for the product type.

$$\begin{aligned}\{&(\mathsf{inl}\ a, \mathsf{inl}\ (b, b)),\\ &(\mathsf{inr}\ a, \mathsf{inl}\ (b, b))\},\end{aligned}$$

showing that E_8 is ambiguous. □

Notice how holding these paths or parse trees can take up a lot of memory while parsing, and this is where the lean-log stack and the backward pass come into play. The idea of the lean-log stack is to use the symmetry of the aNFA to store only very little information on what path is taken. Since there is a clear one-to-one relation between the split states and the join states, it is possible to make the decision on which path that should be taken between these two states, as soon as the join state exists in multiple paths. The relation also means that disambiguating one join state only disambiguates the corresponding split state.

The only thing needed to be stored are the decisions made throughout the path. The logging edges are named as bits, which means the only thing we need to store for each joining state per character consumed is one bit, giving us a nm bit-stack where n is the number of join states, and m is the number of characters in the input string.

We can define an algorithm for doing a forward pass, as presented in Algorithm 1.

The algorithm shows how the step function, takes a transition label, a list of states and an aNFA and returns a list of states.

After each ϵ-closure we get a tuple containing a list of states and a list of labels (one per state), that shows the label of the edge going into these states. The function keepLoggingEdges throws away any edge, and corresponding state, that do not have one of the labels in $\{\bar{0}, \bar{1}\}$.

The only non-trivial part is the epsClosure, since aNFA can hold cycles for the closure to

Algorithm 1: Forward pass

```
input : inputString string that needs to be passed
        NFA aNFA generated from the given RE
output: bitStack computed lean-log stack

bitStack ← empty stack
states, edges ← epsClosure(NFA.startState(), NFA)
bitStack.push(keepLoggingEdges(states, edges))
foreach c in inputString do
    states ← step(c, states, NFA)
    if states.isEmpty() then
        abort
    states, edges ← epsClosure(states, NFA)
    bitStack.push(keepLoggingEdges(states, edges))
if NFA.acceptState() is in states then
    return bitStack
else
    abort
```

travel in. The function epsClosure is defined in Algorithm 2.

Algorithm 2: epsClosure

```
input : state state from where we should do closure
        NFA aNFA generated from the given RE
        visited if not set visited ← empty list
        edges if not set edges ← empty list
output: visited a list containing nodes
        edges a list containing belonging edges

foreach l in [0, 1, 0̄, 1̄, ϵ] do
    state ← step(l,state,NFA)
    if state != NULL and state not in visited then
        s, e ← epsClosure(nextstate, visited.append(state), edges.append(l), NFA)
        visited.append(s)
        edges.append(e)
return visited, edges
```

The function is recursive, and it is keeping track of all edges visited so far. From the construction of the aNFA we know that from doing a step with *one* label, makes sure we can only get one or zero new states. It is important that the list of labels is sorted the way shown in the algorithm, as this is part of the greedy choice.

Example 12. We try to parse the input string *abb*, with E_8. We keep the idea of our active states, but it is no longer necessary to keep the entire path behind the state, hence it does not make any sense keeping the same state twice. Further we need to identify the join states to keep these in the stack. In our example we only have two join states: 5 and 11. Notice how a state can be both a join state and a split state at the same time, like state 5. We can create an *active state tree* by showing the active states after each character transition, furthermore the graph is ordered, so the leftmost choice in the aNFA is placed highest, see Figure 8.

Just like before, we start at the initial state. After the first ϵ-closure, we stand at the states

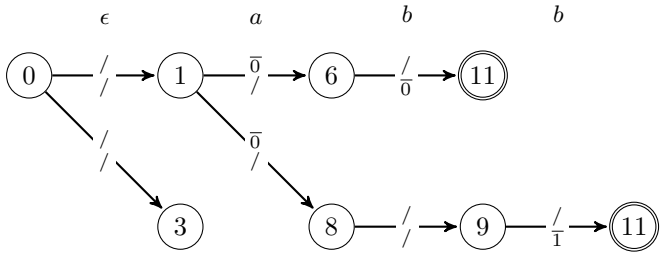


Figure 8: Active state tree for parsing the string abb on the RE $(a + a)(b + bb)$.

1 and 3, where 1 is the leftmost choice, thereby placed highest. Since none of the paths crossed a logging edge, both lines are marked with $\frac{/}{/}$, where the upper sign refers to the state 5, and the lower one refers to state 11. After doing the first character transition, we see that the path starting at state 1 crosses the logging state 5 by the logging edge $\overline{0}$, marked in the figure. We also see that the path going to the state 3 ends. This is because the greedy choice is made already here, and can be interpreted as the following: Since the state 1 is placed higher than 3, we first follow the paths starting at 1, after this, we follow the paths starting at 3 and we get the active states 6 and 8, but since these are already reached, we simply drop the path. Figure 9 shows how we for each character transition add an extra frame to the stack, describing the logging edges crossed.

	ϵ	a	b	b
5	$\frac{/}{/}$	$\overline{0}$	$\frac{/}{/}$	$\frac{/}{/}$
11	$\frac{/}{/}$	$\frac{/}{/}$	$\overline{0}$	$\overline{1}$

Figure 9: Lean-log stack from parsing the string abb on E_8.

□

Notice that it is no longer necessary to hold on to the input string, as all information needed to travel back is now placed in the stack.

4.2 Backward pass

For doing a backward pass, the only thing needed is the aNFA and the lean-log stack, produced by the forward pass. The backward pass starts at the accepting state, then travels backwards, in opposite direction of the directed edges, until meeting a join state where a simple look-up in the lean-log stack is made to determine which way to go. The backward pass only looks at the current stack frame, where each frame in the stack is a list of length n, and m is the amount of join states in the aNFA. Whenever the backward pass meets a character transition it pops the top element of the stack.

The only thing left to do in the backward pass, is to log the labels of the edges, as the path is followed back. In the end stripping away everything that is not choice edges and reversing this list gives us the bit-code representation of the correct parse tree. For a detailed description of the algorithm described see Algorithm 3.

Example 13. From the RE E_8 (Figure 7), the input string abb and the lean-log stack produced (Figure 9), we can now start a backward pass at the accepting state. Already at the first state, we have to do a look up in at the lean-log stack making us follow the logging edge $\overline{1}$. Then we

Algorithm 3: Backward pass

```
input  : bitStack computed lean-log stack
         NFA aNFA generated from the given RE
output: bitCode that denoted the way through the aNFA

bitCode ← empty list
state ← NFA.acceptState()
while state != NFA.startState() do
    label ← NFA.getIngoing(state)
    if label.isList() then
        label ← bitStack.getEdge(state)
    else if label.isChar() then
        bitStack.pop()
    bitCode.append(label)
    state ← NFA.followLabel(label)
return reverse(keepOnlyChoise(bitCode))
```

meet two character transitions, making us pop the two last columns. Then we pass the choice edge 1, before having to do a look up in the lean-log stack again. Finally we cross another character transition and the choice edge 0, reversing these gives us the final bit-code

$$01.$$

Of course we can now verify this by following this bit string through the aNFA, and checking that the result is the string *abb*. □

5 Committing

This section is based on unpublished work done by the KMC group at DIKU. The work on single-state committing, as presented in section 5.2 is not yet proven to be correct, but we will show how this seems intuitive. Furthermore we will show cases where the algorithm does not commit, even though this could be possible.

5.1 Middle Projection

Recall that the lean-log algorithm is implemented by repetitively doing a character transition followed by an epsilon closure. As explained earlier this is called left projection, and likewise we could implement the lean-log algorithm using right projection. Instead, we consider a third alternative.

We define another closure method and projection strategy with the goal of keeping as few active states as possible between the character transitions.

Definition 14. Let *Contract closure* be defined as:

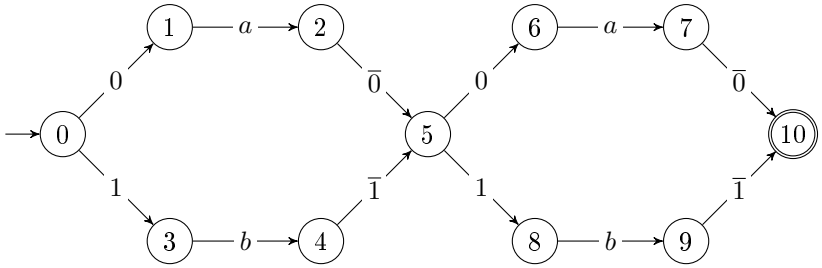
$$Contract(S) = \left[q' \mid q \in S,\, q \overset{p}{\rightsquigarrow} q',\, p \in \{\overline{0}, \overline{1}, \epsilon\} \right],$$

where S is the current list of active states. □

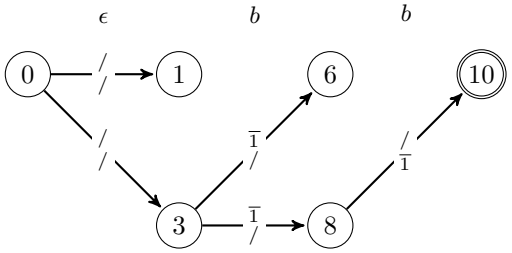
Definition 15. Let *middle projection* be defined as computing $Contract(Step(Close(S), a))$ where S is the current set of active states and a is the next input character. □

The reason for the use of contract closure is that it follows all trivial edges which are logging and ϵ-transition edges, i.e. all edges where no alternative can be chosen. This makes it possible to hold fewer states between parsing characters, since some paths might meet at some point. From the definitions we see how a contract closure does nothing more than part of an ϵ-closure, and how a contract closure is always followed by an ϵ-closure, which overrules what is done in the contract closure. This means that the active states after the ϵ-closure is the same whether we did a contract closure or not. Hence, not considering logging, the outcome of the algorithm does not change when using middle projection instead of left (or right) projection.

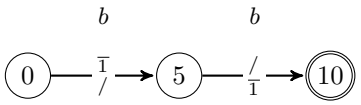
Example 14. Consider a RE $E_9 = (a + b)(a + b)$, shown as an aNFA in Figure 10a. Parsing the string $s = bb$ using left projection would result in the active state tree shown in Figure 10b. However doing the same parse using middle projection would result in the active state tree shown in Figure 10c. So even though we end up with the same lean log, we end up with two different active state trees, and we can see how, in this example doing middle projection, saves some space.



(a) E_9 as an aNFA.



(b) Using left projection.



(c) Using middle projection.

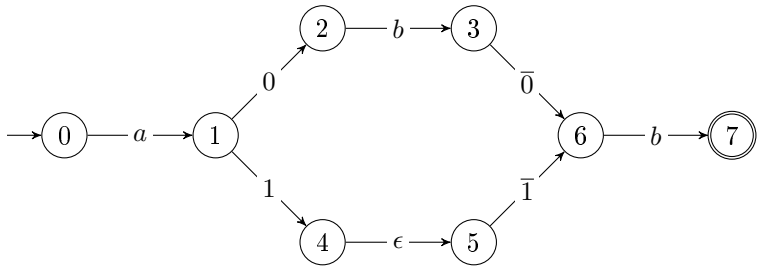
Figure 10: E_9 as aNFA and active state trees for paring bb on E.

□

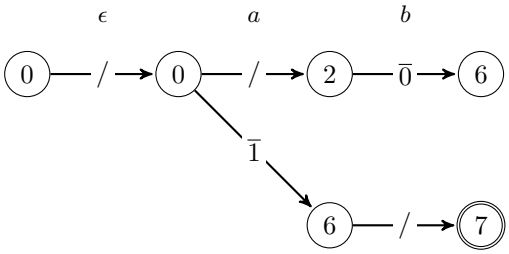
Until now, we have seen how this new projection strategy does not have any impact on the algorithm. This has been under the assumption that the ϵ-closure does not cross any logging edges. The problem is that the logging edges passed by the ϵ-closure needs to be logged in the previous log frame, since it is a result of the last character transition. This happens in two different cases. First of all we cross a logging edge for every Kleene star we meet, since the ϵ-closure can cross the label $\overline{1}$, no matter if we enter the star or not. The other case, is if the REs containing an ϵ-transition, since the ϵ-closure can pass this transition.

Example 15. Let us consider the RE $E_{10} = a(b+1)b$, see Figure 11a. Like in Example 14 we try to parse a string $s = ab$, by using left projection and middle projection. First we parse using left

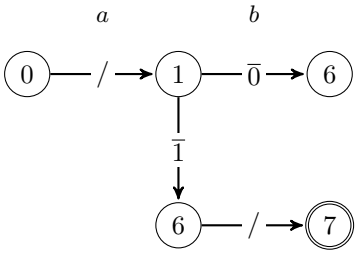
projection, which gives us the parse tree shown in Figure 11b. When we write this information to the lean-log we get Figure 11d, just as we have learned from the previous example. However, when parsing using middle projection, we see how after parsing a, we have the ϵ-closure meeting the joining edge between state 5 and 6. As explained this information needs to be written in the log frame belonging to the parse of a, hence we do a vertical line in the parse tree, and writes the information in the previous log frame, giving the final lean log stack, shown in Figure 11e.



(a) E_{10} as an aNFA.



(b) Using left projection.



(c) Using middle projection.

	ϵ	a	b
6	/	$\bar{1}$	$\bar{0}$

(d) lean-log stack after doing left projection.

	a	b
6	$\bar{1}$	$\bar{0}$

(e) lean-log stack after doing middle projection.

Figure 11: E_{10} as aNFA and active state trees for paring ab on E.

Even though the two lean log stacks do not look completely alike, they still provide the same information to the backward pass, meaning nothing changes in this part of the algorithm. □

Looking at the example above, one might wonder why the old strategy produces one more log frame than doing the exactly same parse using middle projection. This is because, using left projection, the only case where the initial ϵ-closure produces any information to the lean-log stack, is the case where this closure crosses a logging edge i.e. the first thing in the regular expression is something like $E = (E' + 1)...$, or $E = E'^*...$. In this case, what we do using middle projection, is pushing the information to the previous frame, but since no frame is yet pushed to the lean-log, we simply create a new frame. This causes the two lean-logs to look exactly alike.

5.2 Single-State Committing

The motivation for using middle projection is the fact that we want to minimize the amount of states between each character transitions. The best scenario, memory wise, is the case where

we only keep a single state, but only having one state tells something about the parse as well. Having only one state after paring some substring s' means that *all* possible paths goes through this state, after the parse of s'. Furthermore we know, that no matter what happens after this state in the parse, this does not change. This is guaranteed by the symmetry of the aNFA: Disambiguating one join state disambiguates the corresponding split state, and this is what we will try to show in this section.

First, we need to define a function for flattening a path to the input string parsed.

Definition 16. pFlat$(\cdot)$ is a function, that flattens a path to the input string parsed:

$$\begin{aligned} \text{pFlat}\left(q_0 \overset{a_0}{\to} q_1 \overset{a_1}{\to} \ldots \overset{a_{n-1}}{\to} q_n\right) &= \text{pFlat}\left(\overset{a_0}{\to}\right)\text{pFlat}\left(\overset{a_1}{\to}\right)\ldots\text{pFlat}\left(\overset{a_{n-1}}{\to}\right) \\ \text{pFlat}\left(\overset{a}{\to}\right) &= \begin{cases} a & \text{if } a \in \Sigma \\ \epsilon & \text{otherwise,} \end{cases} \end{aligned}$$

where Σ is the input alphabet. If the path is just a single state then the empty string is returned. □

Definition 17 (Single-State Commit Points)**.** Let P be a set of all accepted paths, from parsing an input sting s on some RE E. If we at some point during the parse had one active state q_k after parsing the first parts of the input string s, we know that all paths contain q_k, and we can separate the set P into P' and P'', where all paths $p \in P$ have the structure $\left(q_0 \overset{a_0}{\to} \ldots q_k \ldots \overset{a_{n-1}}{\to} q_n\right)$ and can be divided such that

$$\left(q_0 \overset{a_0}{\to} \ldots \overset{a_{k-1}}{\to} q_k\right) \in P' \text{ and } \left(q_k \overset{a_k}{\to} \ldots \overset{a_{n-1}}{\to} q_n\right) \in P''.$$

Doing this, we further know

$$\forall p' \in P' \ \text{pFlat}(p') = s' \wedge \forall p'' \in P'' \ \text{pFlat}\,(p'') = s'',$$

where s'' is the remaining input sting after parsing s'. Whenever this is the case, we call q_k a commit point. □

From the definition, we see how P' is not dependent on anything in P'', and so we can divide the set of paths into sets of subpaths, while doing the forward pass, without knowing anything about the rest of the parse. The benefit of dividing the paths into smaller sets of subpaths, is that we are able to do the backward pass on partial paths, by treating the first node of the path as the initial state, and the last state as the accepting state.

Example 16. Consider the RE $E_{11} = (a + a)(b + bc)$, shown as an aNFA in Figure 12. We try to parse the string $s = abc$, and see how we only hold one state after the the first a. After the first a, standing at node 5, we have the two paths $0 \overset{0}{\to} 1 \overset{a}{\to} 2 \overset{\overline{0}}{\to} 5$ and $0 \overset{1}{\to} 3 \overset{a}{\to} 4 \overset{\overline{1}}{\to} 5$. Since both paths end in the same state, we only have one active state, and we can separate these pats, into their own set. After parsing the next character b, we have two paths $5 \overset{0}{\to} 6 \overset{b}{\to} 7 \overset{\overline{0}}{\to} 11$ and $5 \overset{1}{\to} 8 \overset{b}{\to} 9$, and since these do not end up in the same state we cannot divide these into their own set. Finally, after parsing c, we only have the path $5 \overset{1}{\to} 8 \overset{b}{\to} 9 \overset{c}{\to} 10 \overset{\overline{1}}{\to} 11$. After the last character we only consider paths ending at the accepting state, hence we always have the single state property at the end.

Sine we have two sets of subpaths, we can consider the sets separately, and do the disambiguation of the first set of subpaths independent of the second set. Using the greedy choice, we

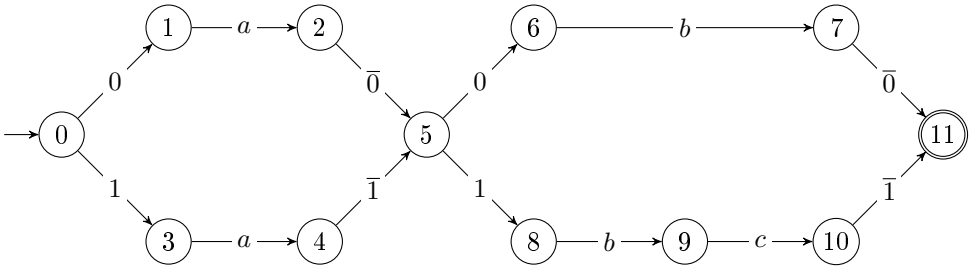


Figure 12: E_{11} as a aNFA.

get the path $0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\overline{0}} 5$ from the first set and since we only have own element in the second set we get the final path:

$$p = 0 \xrightarrow{0} 1 \xrightarrow{a} 2 \xrightarrow{\overline{0}} 5 \xrightarrow{1} 8 \xrightarrow{b} 9 \xrightarrow{c} 10 \xrightarrow{\overline{1}} 11,$$

where $\text{pFlat}(p) = s = abc$. □

Since paths and bit-codes describe the exact same thing, we can easily transfer the single-state commit point property from parsing using paths to parsing using bit-codes. This means we can incorporate the committing property as part of the lean-log algorithm. The algorithm need to be very alike the forward pass, shown in Algorithm 1, with the few modifications, that it uses middle projection, and after each contract closure should check for the singe-state property. We write this as Algorithm 4.

We introduce a function contractClosure, with a similar definition to epsClosure, where the only difference is that we only cross $\left[\overline{0}, \overline{1}, \epsilon\right]$ instead of $\left[0, 1, \overline{0}, \overline{1}, \epsilon\right]$. Furthermore we introduce a if-statement, checking whether the lengths of the list of current states is equal to one, if this is the case, we have a commit point, meaning we do the backward pass, and collect the bit code returned. The backward pass also needs to be slightly modified, such that the backward pass only moves between two given states, instead of the accepting state and the initial state in the entire aNFA. For this we keep a variable lcn, denoting the last commit node i.e. what should be treated as initial state.

Example 17. Consider the RE $E_{12} = (a + b)^* c$, shown as an aNFA in Figure 13a. We parse the string $s = abc$ using middle projection, giving us the active state tree shown in Figure 13b. While parsing, we construct the lean-log stack, and each time we have only one active state, we do a backwards pass, from the new single state to the last single state in the parse.

After parsing the first character a, we only have one active state, and do a backward pass with the first part of the stack, interpreting state 1 as the accepting state and 0 as the initial state. This returns the bit-code 00. After parsing the next b we treat state 1 as both the accepting and initial state, now the lean-log stack has only one frame, (third column in Figure 13c) and this gives the result 01. Finally we parse a c, and here we treat state 9 as the accepting state and state 1 as the initial state. In this case the lean-log stack again consists of a single frame, and the backward pass returns 1.

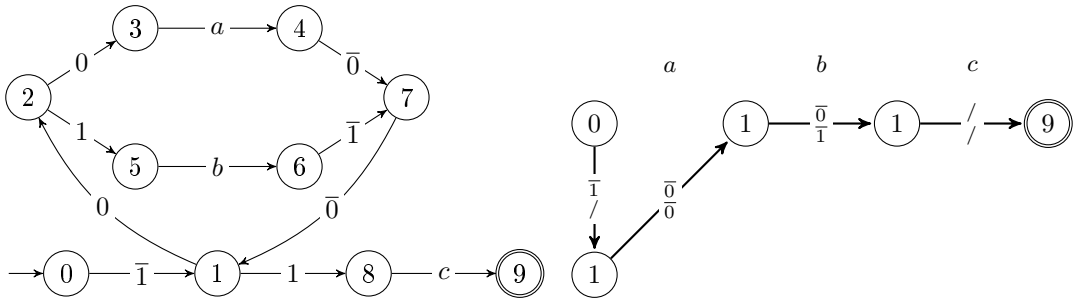
Concatenating the results, we get the final bit-code 00011, and if we follow this string through the aNFA we get the input string $s = abc$, showing the bit code is correct. □

Since what we essentially do is dividing the task into smaller task whenever we have a single state, we call this *single-state committing*.

Algorithm 4: Pass

```
input  : inputString string that needs to be parsed
         NFA aNFA generated from the given RE
output: bc the accumulated bit code

bitStack ← empty stack
bc ← empty code
lcn ← NFA.startState()
foreach c in inputString do
    states, edges ← epsClosure(states, NFA)
    frame ← keepLoggingEdges(states, edges) concatenated with bitStack.pop()
    bitStack.push(frame)
    states ← step(c, states, NFA)
    if states.isEmpty() then
        abort
    states, edges ← contractClosure(states, NFA)
    bitStack.push(keepLoggingEdges(states, edges))
    if states.length = 1 then
        bc ← bc + backwardPass(bitStack, NFA, lcn, states)
        bitStack ← empty stack
        lcn ← states
states, edges ← epsClosure(states, NFA, lcn, states)
frame ← keepLoggingEdges(states, edges) concatenated with bitStack.pop()
bitStack.push(frame)
if NFA.acceptState() is in states then
    return bc + backwardPass(bitStack, NFA, lcn, NFA.acceptState())
else
    abort
```



(a) E_{12} as a aNFA.

(b) Active state tree.

	ϵ	a	b	c
1	$\overline{1}$	$\overline{0}$	$\overline{0}$	/
7	/	$\overline{0}$	$\overline{1}$	/

(c) Lean-log stack.

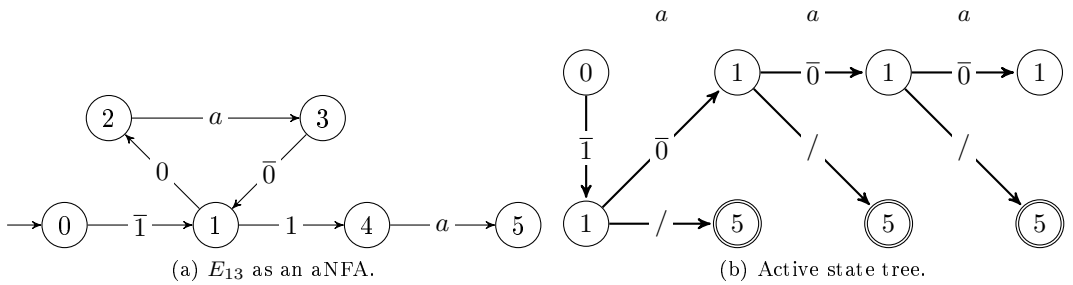
Figure 13: Lean-log example of single-state committing.

5.3 Other Commit Points

In this section we will explore other commit points that one might wish to catch, but are not detected by single-state committing. These examples are not going to show that single-state committing is not working correctly, but instead how the commit not always happens as early as one might wish for.

To study REs where single-state committing does not detect all commit points, we carefully create a RE where is it always the case that we have two active states.

Example 18. Consider the RE $E_{13} = a^*a$, shown in Figure 14a. We parse the input string $s = aaa$, and get the active state tree shown in Figure 14b., and the lean-log stack shown in Figure 14c.



(a) E_{13} as an aNFA.

(b) Active state tree.

	ϵ	a	a	a
1	$\overline{1}$	$\overline{0}$	$\overline{0}$	$\overline{0}$

(c) Lean log stack.

Figure 14: Example of a problematic RE.

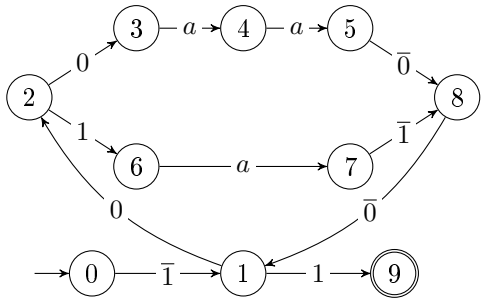
We see how we after each parsed a end up with the two active states 1 and 5. This means

that we are not able to commit before after the last a is parsed, since the path now ending at state 1 is not a legal path.

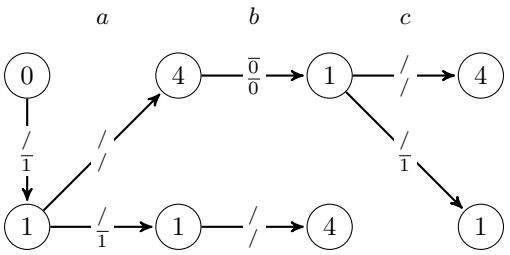
Ideally what we want is for the algorithm to place a commit point after n as, if we know character $n+1$, is also a a. In other words, we would like to look one character ahead *or* commit only part of the parsed input. □

This example shows how something that we by certainty knows does not change, is actually not committed.

Example 19. Consider the RE $E_{14} = (aa+a)^*$ shown in Figure 15a. We parse the input string $s = aaa$, and get the active state tree shown in Figure 15b.



(a) E_{14} as an aNFA.



(b) Active state tree.

	ϵ	a	a	a
1	$\frac{/}{\overline{1}}$	$\frac{/}{\overline{1}}$	$\frac{\overline{0}}{0}$	$\frac{/}{\overline{1}}$
8	$\frac{/}{\overline{1}}$	$\frac{/}{\overline{1}}$	$\frac{\overline{0}}{0}$	$\frac{/}{\overline{1}}$

(c) Lean log stack.

Figure 15: Another problematic RE.

In this example we also get the same pattern where we keep having one branch of the active state tree that dies, just as a new is formed, making sure it is not possible to commit using single-state committing. In the active state tree we can also see how what we optimal wish for is a commit after every second a, to satisfy the greedy choice. □

The above examples might seem disappointing, and question the quality of single-state committing. However both examples have been carefully constructed to show how single state committing does not catch all commit points and are here 'unrealistic'.

The RE E_{13}, can easily be rewritten to $E_{13'} = aa^*$, and the RE E_{14} is something one would not expect a programmer to write, since $E_{14'} = a^*$ would evaluate to the exact same language.

One might notice how our desired commit points in Figure 14b and 15b is placed where we at some later point only have *one* active state (since the other branch is dead). This could be something to consider if one were to expand this single-state committing algorithm, to catch more of the desired commit points.

What this section shows, is how the effectiveness from the single-state algorithm depends very much on *how* the RE is constructed.

6 Implementation

In the following section we will discuss the main concepts of the given lean-log algorithm implementation and the changes we made to implement the use of middle projection and committing. Furthermore we will briefly discuss our testing strategy and results, and present another main use of the commit points.

6.1 Lean-log Implementation

The original implementation includes a complete implementation of the lean-log algorithm, some other implementations for reference and a complete test suite. The primary part of the implementation of the lean log algorithm is placed in `lean-log-impl/src/KMC/TwoPhaseREParser/LeanLog/LeanLogStack.hs`. The implementation leans very close to the overall algorithms explained in Section 4, but there are a few differences which we will try to clarify.

Before beginning the iterative part of the forward pass, something called `adjMemo` and `closureMemo` is computed. `adjMemo` is a data structure keeping all incoming and outgoing edges for each of the nodes, including what state the edge comes from or goes to. `closureMemo` is a data structure that keeps a list of all states that can be reached by an ϵ-closure, for each state in the aNFA. These two data structures are precomputed to avoid look ups in the aNFA and to speedup computing of closures.

After the data structures have been computed, an ϵ-closure is computed and the function `step` is called. `step` (not to be confused with the function step in the algorithms) is the recursive function, that repeatably takes a character transition and performs an ϵ-closure. The ϵ-closure implementation `computeClosure` takes a list of states S and computes the closure for each s in S by following all $\{0, 1, \overline{0}, \overline{1}, \epsilon\}$ edges in `closureMemo`, while keeping track of visited states, to avoid looping. After each ϵ-closure the result of the paths are made into a bit frame by removing anything but the logging edges.

When the last input character has been parsed, the forward pass checks that the accepting state is included in the last ϵ-closure, before returning the bit stack. Otherwise the parse fails.

The backward pass is implemented in such a way that it starts at the accepting state, and then simply looks up the ingoing edges from `adjMemo`, until it reaches initial state. In the case that an edge is labeled with a character a frame is popped from the bit stack, otherwise the edge is added to the result list. Finally if there are multiple paths, the ambiguity is solved by a look-up in the current stack frame, the label is added to the list and that path is followed. Finally the entire list of edges is sorted, before all non-choice edges are filtered out, giving the final bit-code.

6.2 Single-State Committing Implementation

The majority of the new implementation is placed in `lean-log-impl/src/KMC/TwoPhaseREParser/EpsLog/EpsLogStack.hs`, along with minor alterations scattered around the project.

The first thing we did was making sure that the implementation used the defined middle projection strategy. This was done by removing the ϵ-closure being computed before the initial call to `step`, changing `step` to do an ϵ-closure, a character transition and a contract closure furthermore adding a final ϵ-closure after the last input character have been parsed. Before this was possible, we had to add a new stepping strategy and closure function in `Closure.hs`[2].

As discussed earlier in this Thesis the information gained in the contract closure and the following ϵ-closure, needs to be pushed to the same frame in the bit stack. As a more simple

[2] `lean-log-impl/src/KMC/Utils/Closure.hs`

solution we decided to let the ϵ-closure log everything and only log the information gained from the contract closure if the contract closure resulted in a single state. Furthermore after each contract closure the function `commit` is called and it checks if a commit point occurs. The function takes two copies of the bit stack, one *without* the information from the contract closure, and one with the extra stack frame from the contract closure. If a commit has occurred we need the final frame when doing the backward pass and an empty bit stack is returned to the caller along with the bit-code returned from `backwardPass`. If there is no commit, we need to return the stack without the extra frame as the subsequent ϵ-closure will do all the logging. As `commit` becomes responsible for calling `backwardPass` each time a commit has occurred we needed to make a small change to the `backwardPass` function, so instead of always going from the accepting state to the initial state, is goes between two given nodes. We made `forwardPass` responsible for gathering the bit-codes returned from `commit`, allowing `forwardPass` to accumulate and return the results from all the commits. Finally we have renamed `forwardPass` to `Pass` as there is no longer the need for manually calling `backwardPass` after `forwardPass`.

6.3 Tests

We assume that the given lean-log algorithm works as intended and therefore use this as the standard for correctness. As already mentioned, the test framework used to test the lean-log algorithm is included in the project and we use this as the backbone for our testing. The test framework is a set of QuickCheck[3] test cases and a set of HUnit[4] test cases. Whenever a problematic scenario/case has presented itself throughout the development process, it would be added as a unit test case along with the others. These are all run with our implementation against the lean-log algorithm, and all the unit test cases pass and we have yet to find an unsuccessful case in the randomized tests with our current build. Based on these results we assume that our implementation returns the expected results.

We do not test that the implementation commits the expected amount of times or that it commits after the expected input characters have been parsed. The reason for this is that it is not trivial to precompute how many or when a commit should occur. This problem is one of the main reasons why we perform the analysis described in Section 7.

6.4 Implementing Semi-Streaming

When using the middle projection strategy and performing the commits when applicable, the receiver of the partial bit-codes returned from the `commit` function can be changed from being the same thread to be some other thread or process without breaking the algorithm. This opens up for the opportunity to setup a semi-streaming-like system where one process parses the input while another deals with the bit-codes as they come.

The `lean-log-impl/src/KMC/TwoPhaseREParser/EpsIOLog/EpsIOLogStack.hs` file along with the function `elsGetNfaPathIO` defined in `lean-log-impl/src/KMC/TwoPhaseREParser/TwoPhaseREParser.hs` show a simple implementation of this concept, which does not focus on performance nor strive to be some advanced cross-process implementation, but rather a simple working example, meant as a proof-of-concept.

This implementation is built upon the single-state committing implementation and does only make minor alterations to introduce threading. When the user calls the top level function

[3] `http://hackage.haskell.org/package/QuickCheck-2.6`
[4] `http://hackage.haskell.org/package/HUnit`

`elsGetNfaPathIO`, for parsing an input with a given RE, a new thread t is spawned for computing the `pass` function, while the main thread acts as the new receiver and simply waits until it has received all the partial bit-codes from t and returns them to the user when t is done. t proceeds as usual, but instead of returning the partial bit-codes from the backwards passes to the function `forwardPass` it is sent to the main thread.

In our implementation we further try to optimize the process, with parallelism in mind, by computing each partial backward pass in a new thread.

The receiving thread or process could easily be changed from being the main thread to something else by letting it be a passable argument as another Haskell process, a socket, Remote Procedure Call or even some other type of network service.

This setup does not yield much, if not worse, performance gain when performed on a machine with a single core, or when the input string is small. This is due to the extra threading and communication overhead. Instead if a very large file needs to be parsed and some actions needs to be taken on the results, then the procedure could benefit a lot from using such a semi-streaming like implementations, and lower the overall execution time by working in parallel across machines.

Another noteworthy comment is that with such an implementation the receiver should also be ready to cast away any work done, if desired, if it turns out that the given input string does not match even after multiple commits.

Testing of such a setup has not been part of our overall scope, and because of that we have not been investigating the difference in performance due to our changes.

7 Analysis of Commit Point Occurrence

In the following we will try to perform a minor analysis on how often commit points occur in real-world examples. In the analysis we will parse a set of unit test cases followed by a set of log files and count the number of commit points that occurred in the parses.

7.1 POSIX

It should be noted that *POSIX* refer to the *AT&T*[5] interpretation[6] of the POSIX regex standard described in [5, sec. 12].[7]

All test cases where our implementation does not support the RE syntax are not used in the following statistics, nor are the test cases that are supposed to fail, as these do not resemble real-world examples. This leaves us with 254 valid test cases. When parsing all the valid test cases and counting the number of commit point occurrences we arrive at a total of 695 occurrences and an average of 2.74 commit point occurrences per test case, this is including the final commit point that *all* successful parses perform after parsing the last character of the input.

This number is very low and is most likely explained by the average length of the input REs and input strings, these are 12.78 and 4.53, respectively.

If we look at the number of commit points as a function of the length of the input RE, we see in Figure 16 that less is, surprisingly, more. This is most likely due to the fact that most of the small REs do not contain a nested Kleene star or something more complex and therefore allow for many commit points to occur.

[5] `http://www.research.att.com/editions/201406_home.html`

[6] A discussion of the intepretation can be found in `http://www2.research.att.com/~astopen/testregex/re-interpretation.html`.

[7] The individual files are downloaded from `http://www2.research.att.com/~astopen/testregex/testregex.html`

Instead, if we look at the number of commit points as a function of the length of the given input string shown in Figure 17, then we can see an increasing number of commit points when the length of the input increases, apart from the outlier which has an input length of 81, but only 1 commit point. The reason for this is due to the nature of its RE $E = (a + 1)(ab + ba)^*$. It does not allow for any commits until the entire input has been read for reasons discussed in Section 5.3.

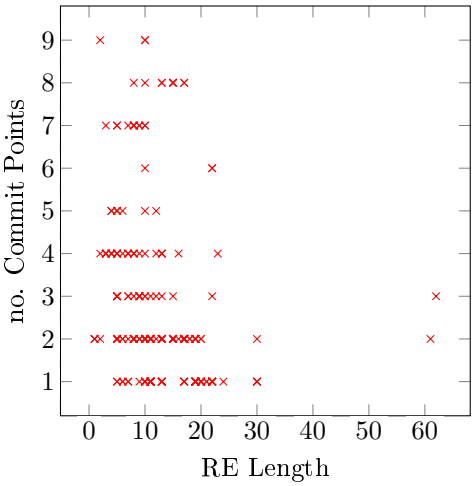


Figure 16: Results showing the number of commit points as a function of the length of the given RE.

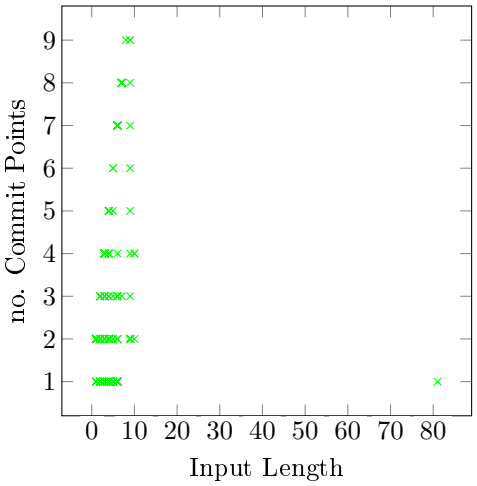


Figure 17: Results showing the number of commit points as a function of the length of the given input.

7.2 DirectX Log File

We have selected a sample log file `DirectX.log`[8] from a Windows machine to try and have a more realistic example. A snippet of the log file can be seen in Figure 18.

```
05/16/14 17:02:45: dsetup32: Extracted file C:\Users\ArniA\AppData\Local\Temp\DXD558.tmp\dxupdate.inf from cab
05/16/14 17:02:45: dsetup32: Extracted file C:\Users\ArniA\AppData\Local\Temp\DXD558.tmp\dxupdate.cif from cab
05/16/14 17:02:45: dsetup32: GetCDXUpdate(): Loading dxupdate.dll in C:\Users\ArniA\AppData\Local\Temp\DXD558.tmp\.
05/16/14 17:02:45: dsetup32: DirectXSetupIsJapanese == 0
05/16/14 17:02:45: dsetup32: DirectXSetupIsJapanNec == 0
05/16/14 17:02:45: dsetup32: IsWow64(): running on Wow64.
05/16/14 17:02:45: dxupdate: -----
05/16/14 17:02:45: dxupdate: DirectXUpdateInstallPlugIn(): Plug-In dxdllreg_x86.cab does not exist.
05/16/14 17:02:45: dxupdate: -----
05/16/14 17:02:45: dxupdate: DirectXUpdateInstallPlugIn(): Plug-In RGB9Rast_1_x86.cab does not exist.
05/16/14 17:02:45: dxupdate: -----
05/16/14 17:02:45: dxupdate: DirectXUpdateInstallPlugIn(): Plug-In RGB9Rast_1_x64.cab does not exist.
```

Figure 18: A snippet of text from the `DirectX.log` file.

The following REs have been constructed:

[8] Primary log file used by the DirectX APIs. See `http://en.wikipedia.org/wiki/DirectX`.

$E_{15} = (\#^*\texttt{\textbackslash n})^*$
$E_{16} = ((timestamp\texttt{:}\ (\texttt{dsetup32} + \texttt{infinst} + \texttt{DXSetup} + \texttt{DSETUP} + \texttt{dxupdate})\texttt{:}\ \#^*\texttt{\textbackslash n}) + \#\#^*\texttt{\textbackslash n} + \texttt{\textbackslash n})^*$
$E_{17} = ((timestamp\texttt{:}\ (\texttt{dsetup32} + \texttt{infinst} + \texttt{DXSetup} + \texttt{DSETUP} + \texttt{dxupdate})\texttt{:}\ .^*) + \#\#^*\texttt{\textbackslash n} + \texttt{\textbackslash n})^*$

where $timestamp = dd/dd/dd\ \ dd\texttt{:}dd\texttt{:}dd$, where $d = [0-9]$, # is a shorthand for anything but \n and . is shorthand for anything.

Their purpose is to try and imitate a real-world usage of the log file where some user might want to extract some information from the file, i.e. some specific parse tree. We decided to plot the number of commit points as a function of the length of the given input string and have therefore additively increased the number of lines parsed from the file and counted the number of commit points. The results can be seen in Figure 19. Here we can see that there is a linear increase of commit points when increasing the given input for all the constructed REs. It should be clear that at least one commit occurs per new line and if we look at the plot we can see that the number of commit points follow closely the number of parsed lines when considering E_{16}. When looking at the results from using E_{15} we can see that there is just about as many commits as there are as characters in the file. E_{17} does only include a very subtle difference from E_{16} and we can see that the amount of commits fall drastically.

7.3 DISM Log File

We selected another log file `dism.log`[9] from a Windows machine and tried to perform the analysis with the same procedure as above. A snippet of the log file can be seen in Figure 20.

```
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Succesfully registered commands for the provider: IntlManager.
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Attempting to add the commands from provider: DriverManager
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Succesfully registered commands for the provider: DriverManager.
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Attempting to add the commands from provider: DISM Unattend Manag
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Succesfully registered commands for the provider: DISM Unattend M
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Attempting to add the commands from provider: DISM Log Provider
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Attempting to add the commands from provider: SmiManager
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Attempting to add the commands from provider: Edition Manager
2014-05-16 00:18:02, Info      DISM   DISM Transmog Provider: PID=3288 Current image session is [ONLINE] - CTrans
2014-05-16 00:18:02, Info      DISM   DISM.EXE: Succesfully registered commands for the provider: Edition Manager
2014-05-16 00:18:02, Info      DISM   DISM Provider Store: PID=3288 Getting Provider DISM Package Manager - CDISM
2014-05-16 00:18:02, Info      DISM   DISM Provider Store: PID=3288 Provider has previously been initialized.  Re
```

Figure 20: A snippet of text from the `dism.log` file.

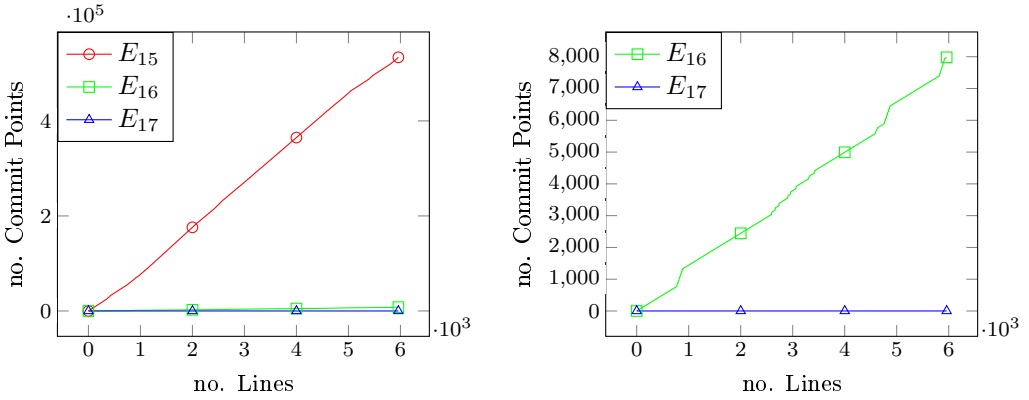
The REs constructed for the log file are:

$$E_{18} = (\#^*\texttt{\textbackslash n})^*$$
$$E_{19} = ((timestamp'\texttt{,}\ \ (\texttt{Info} + \texttt{Warning})\#^*\texttt{\textbackslash n}) + \texttt{\textbackslash n})^*$$
$$E_{20} = ((timestamp'\texttt{,}\ \ (\texttt{Info} + \texttt{Warning}).^*) + \texttt{\textbackslash n})^*$$

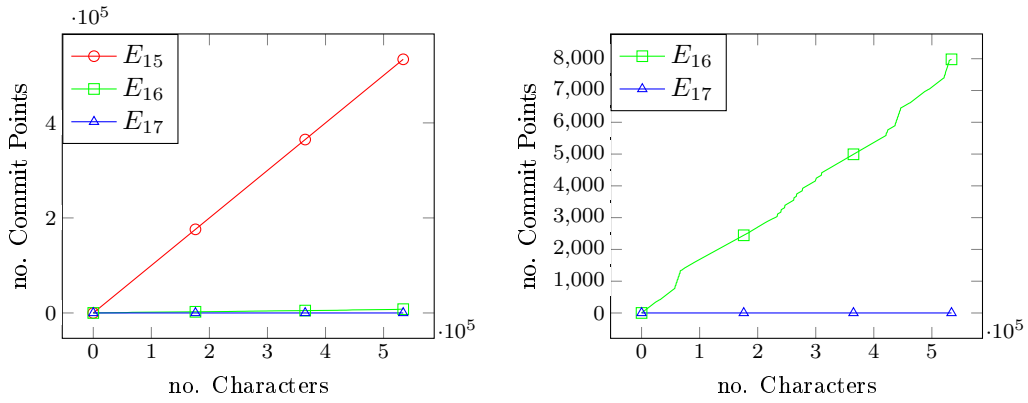
where $timestamp' = dddd\text{-}dd\text{-}dd\ \ dd\texttt{:}dd\texttt{:}dd$, where $d = [0-9]$, # is a shorthand for anything but \n and . is shorthand for anything.

The results produced are very alike the ones found with the `DirectX.log` file as we can see in Figure 21. When we look at the results from E_{18} then we can see that the number of commit

[9] Log file used by the Deployment Image Servicing & Management Tool for Windows 7 & 8. See `http://www.thewindowsclub.com/deployment-image-servicing-management-tool-in-windows-7`.



(a) Number of commit points as a function of the number of lines given as input.



(b) Number of commit points as a function of the number of characters given as input.

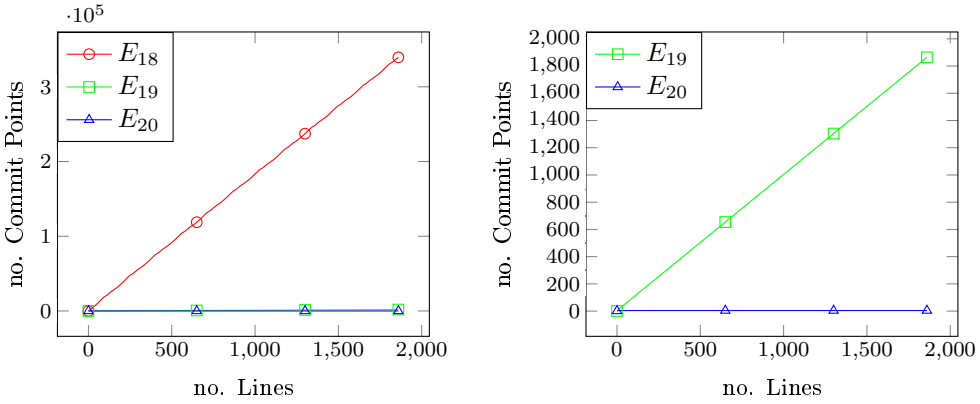
Figure 19: Results from parsing `DirectX.log` with E_{15}, E_{16} and E_{17}.

points roughly match the number of characters in the given input. If we look at the result from E_{19} we can see that it produces about the same number of commit points as there are lines in the file. Finally E_{20} produces only a single commit at the very end on the input.

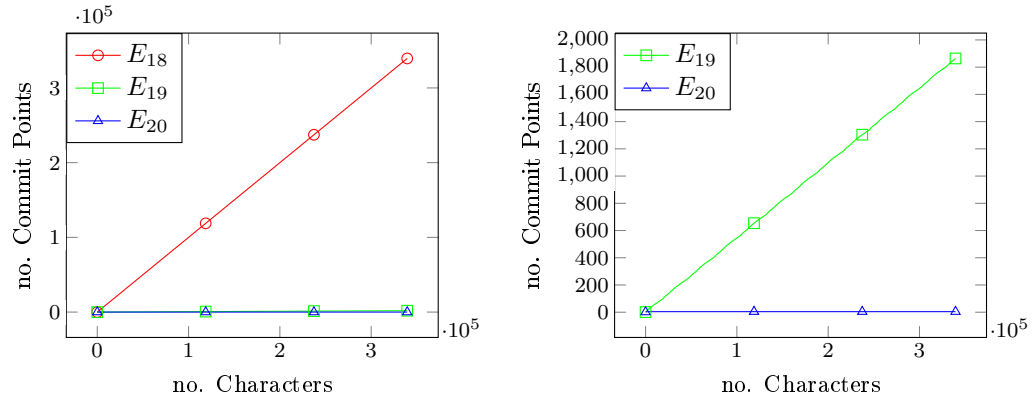
7.4 Discussion of Results

One should be careful when concluding on the results from the POSIX test cases. These are made from a unit testing perspective and are made for testing correctness of a RE matching implementation and therefore only include the necessary. This does make the test cases very far from almost any real world application, but they do give us a hint, and in this case they do show promising results. The log files do give a more realistic view on the matter, as they are unaltered log files taken from the "real world". These clearly show that there is a linear increase in the number of commit points when increasing the input size. Most of the commits arise from the use of the newline in the input and the outer Kleene star in the REs. This makes sure that there is at least one commit per line, and the plots shows that this is the main source of commit points. This shows that files using this kind of structure goes along well with committing.

The results clearly show that the number of commit points also depends on the RE being used.



(a) Number of commit points as a function of the number of lines given as input.



(b) Number of commit points as a function of the number of characters given as input.

Figure 21: Results from parsing `Dism.log` with E_{18}, E_{19} and E_{20}.

E_{15} produces a massive amount of commit points, but does not yield any more information than the number of line breaks in the file. Whereas E_{16} gives a much lower count, but also provides the user with some useful information about the content of the file.

The above results further indicate that to gain full use of the commit points, one has to write appropriate REs, as we can see in the different results from E_{16}, E_{17}, E_{19} and E_{20}. This shows that even though there might be a lot of potential for meaningful commit points, if the user is not able to write a fitting RE then he or she won't be able to get a lot of meaningful commit points.

8 Summary & Future Work

We presented an improved algorithm to the one described in [3]. The improvement allowed for the ability to output correct partial parsing trees during the forward pass of the algorithm. An analysis showed that the number of commit points depend on both the input string and the RE used. Future work could consist of improving the algorithm even further to be able to recognize even more commit points, as discussed in Section 5.3. Further interesting work could be to extend the analysis performed to include different types of files to cover a broader spectrum of

the "real-world" and get a deeper insight into the usage of the middle projection strategy. A C-implementation of the single-state committing algorithm for performing a performance test of algorithm against common RE parsing and matching tools would be a natural next step.

References

[1] Samarjit Chakraborty. Formal Languages and Automata Theory-Regular Expressions and Finite Automata. *Computer Engineering and Networks Laboratory*, 2003.

[2] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer, 2011.

[3] Lasse Nielsen Ulrik Terp Rasmussen Niels Bjørn Bugge Grathwohl, Fritz Henglein. Two-Pass Greedy Regular Expression Parsing. In Stavros Konstantinidis, editor, *Proceedings 18th International Conference on Implementation and Application of Automata (CIAA)*, volume 7982 of *Lecture Notes in Computer Science*, pages 60–71. Springer, July 2013.

[4] Ulrik Terp Rasmussen Niels Bjørn Bugge Grathwohl. A Crash-Course in Regular Expression Parsing and Regular Expressions as Types.

[5] IEEE Computer Society. *Standard for Information Technology - Portable Operating System Interface (POSIX), Base Specifications, Issue 7*. IEEE, 2008. IEEE Std 1003.1.

[6] Guangming Xing. Minimized thompson nfa. *International Journal of Computer Mathematics*, 81(9):1097–1106, 2004.