Regular expression-based parsing with PATRICIA tries

Authors: Emil Heckermann Pedersen
Rune Filtenborg Hansen

Supervisors: Niels Bjørn Bugge Grathwohl
Ulrik Terp Rasmussen
Fritz Henglein

June 7, 2015

Abstract

We show that PATRICIA tries are an efficient data structure for the set of bit strings maintained during transducer simulation performing streaming regular expression-based parsing with greedy disambiguation, by implementing it and comparing it to alternative implementations by measuring their throughput on different expressions and inputs. Because of the features of the PATRICIA trie data structure, the implementation that used the PATRICIA trie obtained a linear run time on all regular expressions, even when other implementations did not, and was between 12% slower and 20% faster than other implementations in cases where they both had a linear run time.

Contents

Chapter 1

Introduction

Regular expressions are widely used for data parsing, filtering, extraction and transformation, not just input validation. Highly efficient streaming regular expression-based parsing with greedy disambiguation can be accomplished by compiling a regular expression to a deterministic finite state transducer[1]. This can result in exponentially-sized transducers when transforming semantically large expressions, which can be abused to cause Regular Expression Denial of Service (REDoS) attacks. This blow-up can be avoided by simulating non deterministic finite-state transducers whose paths characterize parse trees [6, 2].

In this report we describe the theory behind regular expression matching and parsing by non-deterministic transducer simulation, and argue that a PATRICIA trie is an efficient data structure for storing the set of bit strings maintained during the simulation, allowing us to reach a linear run time over the length of the input for any regular expression.

We implement three implementations of the above mentioned non deterministic finite state transducer simulation. One uses PATRICIA tries for the set of bit strings maintained during the simulation, the other two uses an array of string and a list of string respectively, and are for comparison. We compare the performance of the different implementations on different inputs and for different regular expressions. We explain our testing procedures and present the results of our testing.

Chapter 2

Regular expressions

A regular expression can be seen as a formal description of a set of strings satisfying certain criteria. Before defining regular expressions formally, we give a short description of languages and alphabets. All definitions in the following chapters are taken directly from or heavily inspired by "A Crash-course in Regular Expression Parsing and Regular Expressions as Types"[7]

Definition 1. *An* alphabet Σ *is a not empty finite set of symbols.*

Example 1. *An example of an alphabet is* $\Sigma_1 = \{\mathtt{a}, \mathtt{b}\}$

Definition 2. *A* language *using an alphabet* Σ *is a subset* $\mathcal{L} \subseteq \Sigma^\star$*, where* $\Sigma^\star$ *is the set of all strings that can be formed using symbols from* Σ*.*

Given two languages $\mathcal{L}_1$ *and* $\mathcal{L}_2$*, the* concatenation $\mathcal{L}_1\mathcal{L}_2$ *is the set of all strings that are a string from* $\mathcal{L}_1$ *concatenated with a string from* $\mathcal{L}_2$*.*

$$\mathcal{L}_1\mathcal{L}_2 = \{w_1 w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}.$$

Given a language $\mathcal{L}$*, let:*

- $\mathcal{L}^0 = \{\epsilon\}$*, where* ϵ *is a special character that occur in no alphabet and represents the empty string, and*
- $\mathcal{L}^n = \underbrace{\mathcal{L}\mathcal{L}\ldots\mathcal{L}}_{(n\ times)}$.

Then the star operation is defined as star $\mathcal{L}^\star = \bigcup_{n=0}^{\infty} \mathcal{L}^n$

Example 2. *Examples of languages that use the alphabet* Σ_1 *are* $\mathcal{L}_1 = \{\mathtt{a}, \mathtt{abb}, \mathtt{bbbb}\}$ *and* $\mathcal{L}_2 = \{\mathtt{aa}, \mathtt{bb}\}$ *Examples of concatenation of languages:*

- $\mathcal{L}_1\mathcal{L}_2 = \{\mathtt{aaa}, \mathtt{abb}, \mathtt{abbaa}, \mathtt{abbbb}, \mathtt{bbbbaa}, \mathtt{bbbbbb}\}$
- ${\mathcal{L}_2}^\star = \{\epsilon, \mathtt{aa}, \mathtt{bb}, \mathtt{aaaa}, \mathtt{aabb}, \mathtt{bbaa}, \mathtt{bbbb}, \mathtt{aaaaaa}, \mathtt{aaaabb}, \mathtt{aabbaa}, \ldots\}$

Definition 3. *The syntax of regular expressions is described by the grammar:*

$$E_\Sigma ::= 0 \mid 1 \mid \underline{\boldsymbol{a}} \mid (E_1|E_2) \mid E_1^\star \mid E_1E_2$$

where it is assumed that a finite alphabet Σ is given, where $\underline{\boldsymbol{a}} \in \Sigma$, and where E_1 and E_2 are both regular expressions.

"Kleene star" ($E_1^\star$) binds stronger than "Concatenation" (E_1E_2), and both bind stronger than "Alternative" ($E_1|E_2$). So the RE $\underline{\mathbf{ab}}^\star$, is to be understood as $\underline{\mathbf{a}}(\underline{\mathbf{b}}^\star)$, not $(\underline{\mathbf{ab}})^\star$.

Example 3. *Examples of RE's using the alphabet Σ_1:*

- *A literal expression:* $\underline{\boldsymbol{a}}$
- *A concatenation of two literals:* $\underline{\boldsymbol{ab}}$
- *An alternative between two literals:* $\underline{\boldsymbol{a}} \mid \underline{\boldsymbol{b}}$
- *A concatenation between two literal and a Kleene star:* $\underline{\boldsymbol{ab}}^\star$
- *An expression we will work with is* $E_{ex} = (\underline{\boldsymbol{a}}^\star \underline{\boldsymbol{b}} | \underline{\boldsymbol{ab}}^\star)^\star$

Syntactic shorthands:

- $[\underline{\mathbf{a}}, \underline{\mathbf{b}}, \underline{\mathbf{c}}] \stackrel{\text{def}}{=} \underline{\mathbf{a}} \mid \underline{\mathbf{b}} \mid \underline{\mathbf{c}}$
- $E? \stackrel{\text{def}}{=} E \mid 1$
- $E^+ \stackrel{\text{def}}{=} EE^\star$

emailex

Example 4. *Examples of different expressions and what they represent.*

- *All digits can be given by,* $E_d = [\underline{\boldsymbol{0}}, ..., \underline{\boldsymbol{9}}]$.
- *All latin lower case letters,* $E_{lower} = [\underline{\boldsymbol{a}}, ..., \underline{\boldsymbol{z}}]$.
- *All latin upper case letters,* $E_{upper} = [\underline{\boldsymbol{A}}, \ldots, \underline{\boldsymbol{Z}}]$.
- *All latin letters,* $E_l = E_{lower}|E_{upper}$.
- *Up to 255,* $E_{255} = \underline{\boldsymbol{0}}|\underline{\boldsymbol{1}}E_dE_d|\underline{\boldsymbol{2}}([\underline{\boldsymbol{0}}, \ldots, \underline{\boldsymbol{4}}]E_d|\underline{\boldsymbol{5}}[\underline{\boldsymbol{0}}, \ldots, \underline{\boldsymbol{5}}])$
- *Ipv4 address,* $E_{ip} = \underline{\boldsymbol{[}}E_{255}\underline{\boldsymbol{.}}E_{255}\underline{\boldsymbol{.}}E_{255}\underline{\boldsymbol{.}}E_{255}\underline{\boldsymbol{]}}$

Example 5. *Example of an email address given as an expression:*

$$E_{email} = (E_l|E_d)^+ \underline{\boldsymbol{@}}(E_{ip}|((E_l|E_d)^+\underline{\boldsymbol{.}}(E_l|E_d)^+))$$

The language of an RE E is the set of strings containing exactly all possible sentences that can be written in the grammar for E. The language of a regular expression is defined by the function L.

Definition 4. *The* language interpretation $\mathcal{L}(\cdot)$ *of a regular expression is defined as:*

$$\begin{aligned}
\mathcal{L}(0) &= \emptyset \\
\mathcal{L}(\underline{\boldsymbol{a}}) &= \{\mathtt{a}\} \\
\mathcal{L}(1) &= \{\epsilon\} \\
\mathcal{L}(E_1 \mid E_2) &= \mathcal{L}(E_1) \cup L(E_2) \\
\mathcal{L}(E_1E_2) &= \mathcal{L}(E_1)\mathcal{L}(E_2) \\
\mathcal{L}(E^\star) &= \mathcal{L}(E)^\star
\end{aligned}$$

From the definition it is understood that concatenation describes the act of appending the set $\mathcal{L}(E_1)$ with all string in the set $\mathcal{L}(E_2)$.

Example 6. *Some examples of language interpretations:*

- $\mathcal{L}(\underline{\boldsymbol{ab}}) = \mathcal{L}(\underline{\boldsymbol{a}})\mathcal{L}(\underline{\boldsymbol{b}}) = \{\mathtt{a}\}\{\mathtt{b}\} = \{\mathtt{ab}\}$
- $\mathcal{L}(\underline{\boldsymbol{a}} \mid \underline{\boldsymbol{b}}) = \mathcal{L}(\underline{\boldsymbol{a}}) \cup \mathcal{L}(\underline{\boldsymbol{b}}) = \{\mathtt{a}\} \cup \{\mathtt{b}\} = \{\mathtt{a},\ \mathtt{b}\}$
- $\mathcal{L}(\underline{\boldsymbol{ab}}^\star) = \mathcal{L}(\underline{\boldsymbol{a}})\mathcal{L}(\underline{\boldsymbol{b}}^\star) = \{\mathtt{a}\}\left(\bigcup_{n=0}^{\infty} \mathcal{L}(\underline{\boldsymbol{b}})^n\right) = \{\mathtt{a}\}\left(\bigcup_{n=0}^{\infty} \{\mathtt{b}\}^n\right) = \{\mathtt{a}\}\{w_1 w_2 \ldots w_n \mid w_i = \mathtt{b}, n \geq 0\} = \{\mathtt{a}w_1 w_2 \ldots w_n \mid w_i = \mathtt{b}, n \geq 0\}$
- $$\begin{aligned}
\mathcal{L}((\underline{\boldsymbol{a}}^\star \underline{\boldsymbol{b}} | \underline{\boldsymbol{ab}}^\star)^\star) &= \bigcup_{n=0}^{\infty} \left(\mathcal{L}\left(\underline{\boldsymbol{a}}^\star \underline{\boldsymbol{b}} | \underline{\boldsymbol{ab}}^\star\right)\right)^n \\
&= \bigcup_{n=0}^{\infty} \left(\mathcal{L}(\underline{\boldsymbol{a}}^\star \underline{\boldsymbol{b}}) \cup \mathcal{L}(\underline{\boldsymbol{ab}}^\star)\right)^n \\
&= \bigcup_{n=0}^{\infty} \left((\mathcal{L}(\underline{\boldsymbol{a}}^\star)\mathcal{L}(\underline{\boldsymbol{b}})) \cup (\mathcal{L}(\underline{\boldsymbol{a}})\mathcal{L}(\underline{\boldsymbol{b}}^\star))\right)^n \\
&= \bigcup_{n=0}^{\infty} \left(\left(\left(\bigcup_{m=0}^{\infty} \mathcal{L}(\underline{\boldsymbol{a}})^m\right)\{\mathtt{b}\}\right) \cup \left(\{\mathtt{a}\}\left(\bigcup_{k=0}^{\infty} \mathcal{L}(\underline{\boldsymbol{b}})^k\right)\right)\right)^n \\
&= \bigcup_{n=0}^{\infty} \left(\{\mathtt{b}, \mathtt{ab}, \mathtt{aab}, \ldots\} \cup \{\mathtt{a}, \mathtt{ab}, \mathtt{abb}, \mathtt{abbb}, \ldots\}\right)^n \\
&= \{\mathtt{a}, \mathtt{b}\}^\star
\end{aligned}$$

Chapter 3

Finite state transducer

To be able to assess whether a given word is in a language, we present a method here which we will use going forward.

Definition 5. *A* finite state transducer *(FST)* $M = (Q, \Sigma, \Gamma, \Delta, q_0, F)$*, where:*

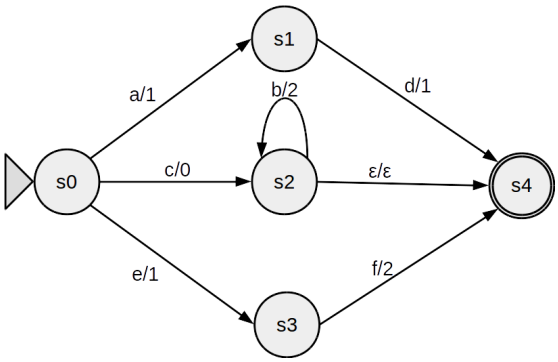
- Q *is a finite set of* states*,*
- Σ *is the expression alphabet,*
- Γ *is the output alphabet,*
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^\star \times Q$ *is a finite set of* transitions*,*
- $(q, x, y, q') \in \Delta$*,*
- $q_0 \in Q$ *is the* initial state*, and*
- $F \subseteq Q$ *is the* accepting states*.*

If a transition (q, x, y, q') exists, when Δ is understood we give the shorthand $q \overset{x/y}{\rightarrow} q'$. We can draw such a transducer, by representing the states as circles and the transitions as arrows, with their labels written beside the transition arrows.

Example 7. *Example of an FST* $M_1 = (Q, \Sigma, \Gamma, \Delta, q_0, \{q_f\})$

- $Q = \{s_0, s_1, s_2, s_3, s_4\}$
- $\Sigma = \{a, b, c, d, e, f\}$
- $\Gamma = \{\epsilon\}$
- $\Delta = \{(s_0, a, 1, s_1), (s_0, c, 0, s_2), (s_0, e, 1, s_3), (s_1, b, 1, s_4), (s_2, d, 2, s_2), (s_2, \epsilon, \epsilon, s_4), (s_3, f, 2, s_4)\}$
- $q_0 = s_0$
- $F = \{s_4\}$

We can draw an FST as a graph like the one below, where circles represent states and arrows represent transitions between states. The triangle points at the initial state and the accepting state has a double lined edge.



Definition 6. *A* path *in FST* $M = (Q, \Sigma, \Gamma, \Delta, q, F)$*, is a sequence of transitions*

$$p = [(q_1, \alpha_1, y_1, q_2), (q_2, \alpha_2, y_2, q_3), \ldots, (q_{n-1}, \alpha_{n-1}, y_{n-1}, q_n)]$$

Where:

- $\alpha_1, \alpha_2, \ldots, \alpha_n \in \Sigma \cup \{\epsilon\}$
- $w = \alpha_1 \alpha_1 \ldots \alpha_n$
- $v = v_1 v_2 \ldots v_n \in \Gamma^\star$
- $q_i \overset{\alpha_i/v_i}{\rightarrow} q_{i+1}, i \in \mathbb{N}$
- $q_1, \ldots, q_n \in Q$

We will write $p : q_1 \overset{w/v}{\rightsquigarrow} q_n$*, when such a path exists.*

Example 8. *An example of a path could be the path* $P_{cdd} = s_0 \overset{\texttt{cdd}/022}{\rightsquigarrow} s4$

Definition 7. *A path* $p : q \overset{\epsilon/v}{\rightsquigarrow} q$ *is an* ϵ*-cycle.*

Definition 8. *A* non-problematic path *is a path that does not contain any* ϵ*-cycles.*

We use the shorthand $q_1 \overset{w/v}{\rightsquigarrow}_{np} q_n$ to refer to a non-problematic path from q_1 to q_n.

For every path, there is a non problematic path with the same input label, and the number of non problematic paths for a given input is finite[3].

Definition 9. *We say that an FST* $M = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ accepts *a string* s *when there exists path* $P = q_0 \overset{w/v}{\rightsquigarrow} q_f$*, where* $q_f \in F, \gamma \in \Gamma^\star$

Chapter 4

Regular Expression Matching

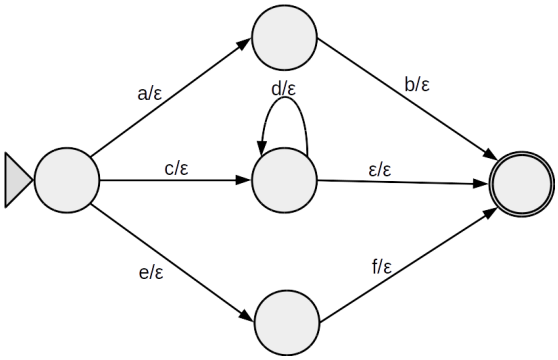
When given a regular expression the purpose of it can simply be to figure out if a given word matches with it.

Definition 10. *We say that a string s* matches *an RE E, if $s \in L(E)$*

Matching can be done by converting the RE into an FST that only accepts inputs that match the RE, and simulating that transducer to determine if it accepts the input we know this from Kleene's Representation theorem[10]. As such we know that:

$$\forall s \in \Sigma^\star.\ s \in \mathcal{L}(e) \iff q_0 \overset{s/\epsilon}{\rightsquigarrow} q_f, q_f \in F.$$

Example 9. *Example of an FST representing the RE* $\underline{\boldsymbol{ab}} \mid \underline{\boldsymbol{cd}}^\star \mid \underline{\boldsymbol{ef}}$



The strings `ab` *and* `cdd` *match the expression, and would be accepted.*

Chapter 5

Regular Expressions as Types

Intuitively an RE contains some structure, this structure corresponds to a type. Strings that match an RE E can be thought of as values of a type $\mathcal{T}(E)$, where $\mathcal{T}(E)$ corresponds to the structure of E. These values are also called parse trees.

Definition 11. *The syntax of* values *is described by the grammar:*

$$v_1, \ldots, v_n \in Val ::= () \mid a \in \Sigma \mid \text{inl}\, v_1 \mid \text{inr}\, v_1 \mid \langle v_1, v_2 \rangle \mid [v_1, \ldots, v_n]$$

Where Val(Σ) is the set of all values over Σ.

Definition 12. *Given two subsets $A, B \in Val$, the Cartesian product of A and B is:*

$$A \times B = \{\langle v_1, v_2 \rangle \mid v_1 \in A, v_2 \in B\}$$

The disjoint union is given by:

$$A + B = \{\text{inl}\, v_1 \mid v_1 \in A\} \cup \{\text{inr}\, v_2 \mid v_2 \in B\}$$

And for $A^\star$:

$$A^n = \{[v_1, ..., v_n] \mid v_i \in A\}, n \geq 0$$

$$A^\star = \bigcup_{n=0}^{\infty} A^n$$

Definition 13. *The set of* structured values *denoted by an RE E is the set $\mathcal{T}(E) \subseteq Val(\Sigma)$ defined as:*

$$\begin{aligned}
\mathcal{T}(0) &= \emptyset \\
\mathcal{T}(1) &= \{()\} \\
\mathcal{T}(\underline{a}) &= \{a\} \\
\mathcal{T}(E_1 \mid E_2) &= \mathcal{T}(E_1) \ + \ \mathcal{T}(E_2) \\
\mathcal{T}(E_1 E_2) &= \mathcal{T}(E_1) \times \mathcal{T}(E_2) \\
\mathcal{T}(E^\star) &= \mathcal{T}(E)^\star,
\end{aligned}$$

Example 10. *Examples of regular expressions as types.*

- $\mathcal{T}(\underline{\boldsymbol{ab}}) = \mathcal{T}(\underline{\boldsymbol{a}}) \times \mathcal{T}(\underline{\boldsymbol{b}}) = \{\mathtt{a}\} \times \{\mathtt{b}\} = \{\langle \mathtt{a}, \mathtt{b} \rangle\}$
- $\mathcal{T}(\underline{\boldsymbol{a}}|\underline{\boldsymbol{b}}) = \mathcal{T}(\underline{\boldsymbol{a}}) \ + \ \mathcal{T}(\underline{\boldsymbol{b}}) = \{\text{inl } \mathtt{a}\} \cup \{\text{inr } \mathtt{b}\} = \{\text{inl } \mathtt{a}, \text{inr } \mathtt{b}\}$
- $$\begin{aligned}\mathcal{T}(\underline{\boldsymbol{ab}}^\star) &= \mathcal{T}(\underline{\boldsymbol{a}}) \times \mathcal{T}(\underline{\boldsymbol{b}}^\star) \\ &= \{\mathtt{a}\} \times (\mathcal{T}(\underline{\boldsymbol{b}})^\star) \\ &= \{\mathtt{a}\} \times (\{\mathtt{b}\}^\star) \\ &= \{\mathtt{a}\} \times \{[v_1, \ldots, v_n] \mid v_i \in \{\mathtt{b}\}, n \geq 0\} \\ &= \{\langle \mathtt{a}, v \rangle \mid v \in \{[v_1, \ldots, v_n] \mid v_i = \mathtt{b}, n \geq 0\}\}\end{aligned}$$

- $\mathcal{T}(\underline{\boldsymbol{ab}}) = \mathcal{T}(\underline{\boldsymbol{a}}) \times \mathcal{T}(\underline{\boldsymbol{b}}) = \{\mathtt{a}\} \times \{\mathtt{b}\} = \{\langle \mathtt{a}, \mathtt{b} \rangle\}$
- $\mathcal{T}(\underline{\boldsymbol{a}}|\underline{\boldsymbol{b}}) = \mathcal{T}(\underline{\boldsymbol{a}}) \ + \ \mathcal{T}(\underline{\boldsymbol{b}}) = \{\text{inl } \mathtt{a}\} \cup \{\text{inr } \mathtt{b}\} = \{\text{inl } \mathtt{a}, \text{inr } \mathtt{b}\}$
- $$\begin{aligned}\mathcal{T}(\underline{\boldsymbol{ab}}^\star) &= \mathcal{T}(\underline{\boldsymbol{a}}) \times \mathcal{T}(\underline{\boldsymbol{b}}^\star) \\ &= \{\mathtt{a}\} \times (\mathcal{T}(\underline{\boldsymbol{b}})^\star) \\ &= \{\mathtt{a}\} \times (\{\mathtt{b}\}^\star) \\ &= \{\mathtt{a}\} \times \{[v_1, \ldots, v_n] \mid v_i \in \{\mathtt{b}\}, n \geq 0\} \\ &= \{\langle \mathtt{a}, v \rangle \mid v \in \{[v_1, \ldots, v_n] \mid v_i = \mathtt{b}, n \geq 0\}\}\end{aligned}$$

We can *flatten* parse trees by removing all the branches and only keeping the leaves.

Definition 14. *The* flattening *of a parse tree v is denoted $|\cdot|$ and is defined as:*

1. $|()| = \epsilon$
2. $|\mathtt{a}| = \mathtt{a}$
3. $|\langle v_1, v_2 \rangle| = |v_1||v_2|$
4. $|\text{inl } v| = |v|$
5. $|\text{inr } v| = |v|$
6. $|[v_0, \ldots, v_n]| = |v_0| \ldots |v_n|$.

Given an REs type definition and a string that matches the RE, we can create parse trees that represent that string given the RE.

Example 11. *For example value* $\text{inl } \mathtt{a} \in \mathcal{T}(\underline{\boldsymbol{a}}|\underline{\boldsymbol{b}})$*, or given RE* $\underline{\boldsymbol{ab}}|\underline{\boldsymbol{c}}^\star$ *the input* `ccc` *is represented by the value*

$$\text{inr } [\mathtt{c}, \mathtt{c}, \mathtt{c}] \in \mathcal{T}(\underline{\boldsymbol{ab}}|\underline{\boldsymbol{c}})^\star$$

Example 12. *For example when we flatten the value* $\text{inl}\ \mathtt{a}$*, Where* $v \in T(a|b)$*, we get* $|v| = |inla| = |a| = a$

Another example of flattening, $|\text{inl}\ \langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle\rangle|$*:*

$$\begin{aligned}
|\text{inl}\ \langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle\rangle| &= |\langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle\rangle| \\
&= |\mathtt{a}||\text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\mathtt{b}||\text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\text{inl}\ \langle \mathtt{c},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\langle \mathtt{c},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\mathtt{c}||\text{inr}\ ()| \\
&= \mathtt{abc}|\text{inr}\ ()| \\
&= \mathtt{abc}|()| \\
&= \mathtt{abc}.
\end{aligned}$$

Definition 15. *An RE E is* ambiguous *when two or more parse trees in* $\mathcal{T}(E)$ *flatten to the same string:*

$$\exists v_1, v_2 \in \mathcal{T}(E).v_1 \neq v_2 \wedge |v_1| = |v_2|.$$

Example 13. *Example the RE* $\underline{\mathtt{abb}}|\underline{\mathtt{ab}}^\star$ *is ambiguous when matched with* $\mathtt{abb}$

$$\begin{aligned}
|\text{inl}\ \langle \mathtt{a},\ \langle \mathtt{b},\ \mathtt{b}\rangle\rangle| &= |\langle \mathtt{a},\ \langle \mathtt{b},\ \mathtt{b}\rangle\rangle| \\
&= |\mathtt{a}||\langle \mathtt{b},\ \mathtt{b}\rangle| \\
&= \mathtt{a}|\langle \mathtt{b},\ \mathtt{b}\rangle| \\
&= \mathtt{a}|\mathtt{b}||\mathtt{b}| \\
&= \mathtt{ab}|\mathtt{b}| \\
&= \mathtt{abb}.
\end{aligned}$$

$$\begin{aligned}
|\text{inr}\ \langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle\rangle| &= |\langle \mathtt{a},\ \text{inl}\langle \mathtt{b},\ \text{inl}\ \langle\ \mathtt{b},\ \text{inr}\ ()\rangle\rangle\rangle| \\
&= |\mathtt{a}||\text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle| \\
&= \mathtt{a}|\mathtt{b}||\text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\langle \mathtt{b},\ \text{inr}\ ()\rangle| \\
&= \mathtt{ab}|\mathtt{b}||\text{inr}\ ()| \\
&= \mathtt{abb}|\text{inr}\ ()| \\
&= \mathtt{abb}|()| \\
&= \mathtt{abb}.
\end{aligned}$$

Given a parse tree, we can minimize the amount of information we need to store, by also knowing which regular expression it is generated from.

Definition 16. *Given a parse tree t, $\mathcal{B}(t)$ is the bitcoding:*

1. $\mathcal{B}(\underline{\boldsymbol{a}}) = \epsilon$
2. $\mathcal{B}(()) = \epsilon$
3. $\mathcal{B}(\langle v_1,\ v_2\rangle) = \mathcal{B}(v_1) \cdot \mathcal{B}(v_2)$
4. $\mathcal{B}(\text{inl}\ v_1) = 0 \cdot \mathcal{B}(v_2)$
5. $\mathcal{B}(\text{inr}\ v_1) = 1 \cdot \mathcal{B}(v_2)$
6. $\mathcal{B}([v_1, \ldots, v_n]) = 0 \cdot \mathcal{B}(v_1) \cdot 0 \cdot \mathcal{B}(v_2) \cdot \ldots \cdot 0 \cdot \mathcal{B}(v_n) \cdot 1,$

where $v_1, \ldots, v_n \in \mathcal{T}(E)$, and $\cdot$ denotes string concatenation.

Example 14. *We can find the bitcode values of the previous example:*

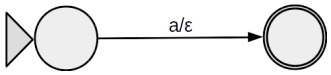
$$\begin{aligned}
\mathcal{B}(\text{inr}\ \langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle\rangle) &= 1 \cdot \mathcal{B}(\langle \mathtt{a},\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle\rangle) \\
&= 1 \cdot \mathcal{B}(\mathtt{a}) \cdot \mathcal{B}(\ \text{inl}\ \langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle) \\
&= 10 \cdot \mathcal{B}(\langle \mathtt{b},\ \text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle\rangle) \\
&= 10 \cdot \mathcal{B}(\mathtt{b}) \cdot \mathcal{B}(\text{inl}\ \langle \mathtt{b},\ \text{inr}\ ()\rangle) \\
&= 100 \cdot \mathcal{B}(\langle \mathtt{b},\ \text{inr}\ ()\rangle) \\
&= 100 \cdot \mathcal{B}(\mathtt{b}) \cdot \mathcal{B}(\text{inr}\ ()) \\
&= 1001 \cdot \mathcal{B}(()) = 1001,
\end{aligned}$$

$$\mathcal{B}(\text{inl}\ \langle \mathtt{a},\ \langle \mathtt{b},\ \mathtt{b}\rangle\rangle) = 0.$$

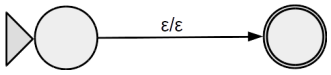
Chapter 6

Thompson NFA

A *non deterministic finite automaton* (NFA), is an FST without output. Thompson NFA [11] generation is a specific method for generating an NFA representation of an RE. When using the Thompson algorithm, we construct an NFA from the RE E bottom-up, i.e. atomic expressions are converted first, and compound expressions are converted by combining the automata resulting from converting its sub-expressions. We use a slightly modified version of the algorithm to generate an FST instead of an NFA, which outputs a bit string representing the path taken through the FST[6]. The conversion of atomic expressions are simple. E = **a** becomes:

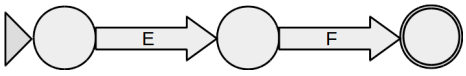


E = `1` becomes:

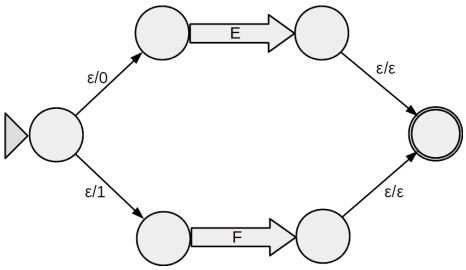


$E = 0$ becomes:

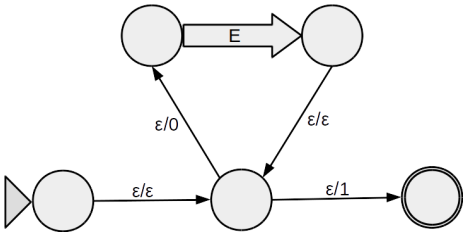
For compound expressions the arrow notation is not to be understood literally, but as where the recursive subexpressions automata is to be inserted. Here $E_0, E,\ and\ F$ are regular expressions.
$E_0 = EF$ becomes:



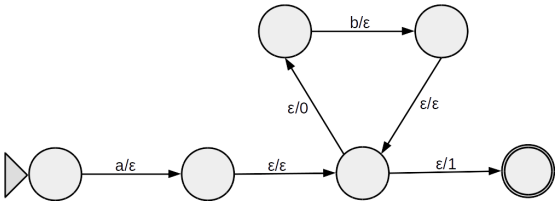
$E_0 = E|F$ becomes:



$E_0 = E^\star$ becomes:



Example 15. *NFA generated from the RE* $\underline{\boldsymbol{ab}}^\star$

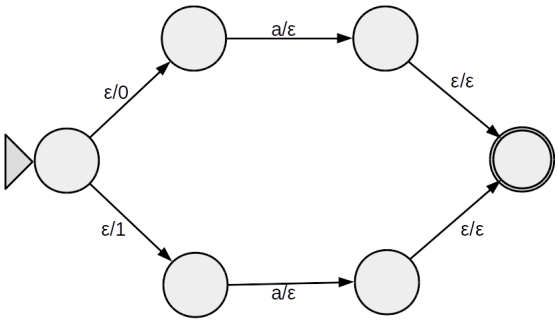


Because of the way the Thompson algorithm creates the NFA from the regular expression, a path through the Thompson NFA matches a parse tree. And its output bit string will correspond to the parse tree's bit string[6].

Just as when we could get several parse trees for the same expression which flattens to the same, the FST can also be ambiguous.

Definition 17. *An FST is* ambiguous *if there exists more than one path* $q_i \overset{w/v}{\rightsquigarrow} q_j$.

Example 16. *ambiguous NFA generated from the ambiguous RE* $\underline{\boldsymbol{a}}|\underline{\boldsymbol{a}}$



We solve this problem by always choosing a left branch over a right branch in a parse tree, this corresponds to always choosing the leftmost argument for an expression. This will correspond to outputting the lexicographically least bit string as described in Definition 18.

Chapter 7

Regular Expression Parsing

Now we can use the structures we have showed to find the given parse tree for a specific input, we call this *parsing.*

Definition 18. *RE-based* parsing *of a string s on RE E, is the process of constructing a parse tree v, where $v \in \mathcal{T}(E)$ and $|v|$=s.*

If the RE is ambiguous we will choose the parse tree with the lexicographically least bit string.

Definition 19. *The* lexicographical *order on bit strings.*

1. *$\epsilon < s$ if $s \neq \epsilon$*
2. *$0s < 1s$'*
3. *$0s < 0s'$ if $s < s'$*
4. *$1s < 1s'$ if $s < s'$*

We say that $\text{MIN}(\{v_0, \ldots, v_n\})$ is given in the usual sense. We can perform parsing by simulating a transducer $M = (Q, \Sigma, \Gamma, \Delta, q_0, q_f)$ created from the RE, using the Thompson algorithm.

Example 17. $\text{MIN}\,(\{01011, 00111, 100, 1101\}) = 00111$

The following algorithm and functions are from [8].

$$\text{closure}_\Delta(W) :\ \mathcal{P}(\{0,1\}^\star \times Q) \to \mathcal{P}(\{0,1\}^\star \times Q)$$
$$\text{closure}_\Delta(W) = W \cup \left\{ (v \cdot o, q') \mid (v, q) \in W \wedge q \overset{\epsilon/o}{\rightsquigarrow}_{np} q' \right\}$$
$$\text{step}_\Delta(W, a) :\ \mathcal{P}(\{0,1\}^\star \times Q) \times \Sigma \to \mathcal{P}(\{0,1\}^\star \times Q)$$
$$\text{step}_\Delta(W, a) = \{(v \cdot o, q') \mid (v, q) \in W \wedge (q, a, o, q') \in \Delta\}$$
$$\text{filter}(W) :\ \mathcal{P}(\{0,1\}^\star \times Q) \to \mathcal{P}(\{0,1\}^\star \times Q)$$
$$\text{filter}(W) = \{(v, q) \mid (v, q) \in W \wedge \forall v', (v', q) \in W \Rightarrow v \leq v'\}$$
$$\text{split}(W) :\ \mathcal{P}(\{0,1\}^\star \times Q) \to (\{0,1\}^\star, \mathcal{P}(\{0,1\}^\star \times Q))$$
$$\text{split}(W) = (v, W') \text{ with longest } v \text{ s.v. } W = \{(v \cdot v', q') \mid (v', q') \in W'\}$$

Where:

- $v, v', o \in \Gamma^\star$
- $a \in \Sigma$
- $q, q' \in Q$
- W is a set of (bit string, state) pairs
- $W \in \mathcal{P}(\{0,1\}^\star \times Q)$

Some intuition about the purpose of these functions.

- $\text{closure}_\Delta(W)$ is a function that returns the epsilon closure of W.
- $\text{step}_\Delta(W, a)$ is a functions that returns the set of states that can be reached by crossing a transition with label a, from a state in W.
- $\text{filter}(W)$ is a function that returns only the lexicographically least path for each state q in W, we can only find the least from a finite set, which is why we only use non-problematic paths.
- $\text{split}(W)$ is a function that returns a pair (v, W') where t is the longest common prefix among all bit strings in W, and W' is W with v removed from the beginning of each bit string.

Algorithm 1: Parsing

```
input  : Δ, s
output: Lexicographical least accepting path
W := filter(closure_Δ({(ϵ, q_0)}
for a ∈ s do
    W := filter(closure_Δ(step_Δ(W, a)))
    if W = ∅ then
        terminate with error
    (t, W) := split(W);
    output v
if ∃v.(v, q_f) ∈ W then
    output v and terminate with success
else
    terminate with error
```

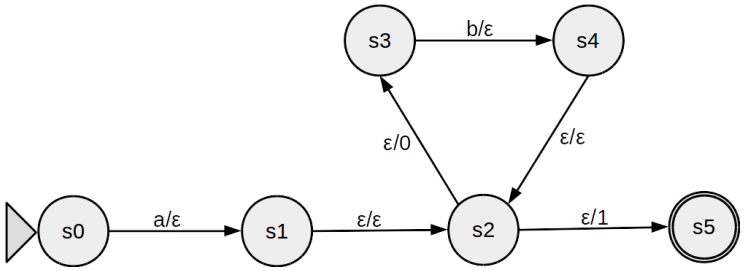
We study the behavior of the set W in this algorithm. The first lines gives us an epsilon closure on the initial state followed by a filter operation. This first gives us a set of all the states reachable from the initial state with epsilon transitions. The epsilon closure is followed by a filter operation, as described above filter returns only one path for each state. Therefore W must be a finite set, with at most one element for each state in the FST. The first line of the loop will again call filter on an epsilon closure and a step operation. As such W will again contain at most one element for each state, each containing the lexicographically least output for a path to that state. After the if statement we call *split*(W), which will never add new elements to W. From these observations we can conclude that $|W| \leq |Q|$. Since the paths are always non problematic, we know that the number of these paths are finite and that each output string will always have a finite length. $\forall (v, q) \in W, |v| < \infty$

We define the function *StepComplete* as this is closer to what our actual implementations look like.

$$\begin{aligned}
&\mathit{StepComplete}(W, \alpha) : \mathcal{P}(\{0,1\}^\star \times Q) \times \Sigma \cup \epsilon \to \mathcal{P}(\{0,1\}^\star \times Q).\\
&\mathit{StepComplete}(W, \epsilon) = \text{filter}(\text{closure}_\Delta(W))\\
&\mathit{StepComplete}(W, a) = \text{filter}(\text{closure}_\Delta(\text{step}_\Delta(W, a)))
\end{aligned}$$

The run time of the algorithm is linear to the input size. The *Closure*, *step*, *split* and *filter* operation depends on the FST size, as they scale by the size of Q, therefore they run in time $O(|Q|)$. We repeat these operations a number of times equal to at most the input size, as we terminate early if no further operations can be made. Therefore we have an expected time of $O(|Q||s|) = O(|s|)$, as the automaton remains constant through any given simulation.

Example 18. *Simulating Thompson NFA of RE* $\underline{\boldsymbol{ab}}^\star$ *with input* `ab`



- $Q = \{s0, s1, s2, s3, s4, s5\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{0, 1\}$
- $\Delta = \{(s0, a, \epsilon, s1), (s1, \epsilon, \epsilon, s2), (s2, \epsilon, 0, s3), (s3, a, \epsilon, s4), (s4, \epsilon, \epsilon, s2), (s2, \epsilon, 1, s5)\}$
- $q_0 = s0$
- $g_f = q5$

1.

$$\begin{aligned}
W &:= \text{filter}(\text{closure}_\Delta(\{(\epsilon, s0)\}))\\
W &:= \text{filter}(\{(\epsilon, s0)\})\\
W &:= \{(\epsilon, s0)\}
\end{aligned}$$

2. start loop, there is input to read, read `a`*.*

3.

$$\begin{aligned}
W &:= \text{filter}(\text{closure}_\Delta(\text{step}_\Delta(\{(\epsilon, s0)\}, a)))\\
W &:= \text{filter}(\text{closure}_\Delta(\{\epsilon, s1\}))\\
W &:= \text{filter}(\{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s1)\})\\
W &:= \{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s1)\}
\end{aligned}$$

4.

$$(t, W) = \text{split}(\{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s1)\})$$
$$(t, W) = (\epsilon, \{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s1)\})$$

5. ***output*** ϵ*, there is still input, read b.*

6.

$$W := \text{filter}(\text{closure}_\Delta(\text{step}_\Delta(\{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s1)\}, b)))$$
$$W := \text{filter}(\text{closure}_\Delta(\{0, s4\}))$$
$$W := \text{filter}(\{(00, s3), (01, s5), (0, s2), (0, s4)\})$$
$$W := \{(00, s3), (01, s5), (0, s2), (0, s4)\}$$

7.

$$(t, W) = \text{split}(\{(00, s3), (01, s5), (0, s2), (0, s4)\})$$
$$(t, W) = (0, \{(0, s3), (1, s5), (\epsilon, s2), (\epsilon, s4)\})$$

8. ***output*** *0, there is no input, check if* q_f *exists in set.*

9. q_f *is* $s5$*,* ***output*** *1 and terminate with success*

The Complete output becomes 01.

Chapter 8

PATRICIA tries

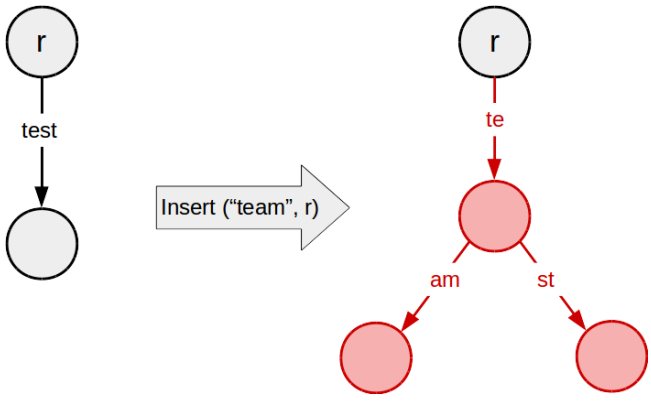
PATRICIA tries [4] are a compact datastructure for storing strings. It is a labeled tree with each branch representing a common prefix of two or more strings and each leaf representing a string.

Definition 20. *A* PATRICIA trie *is a tree with the following properties:*

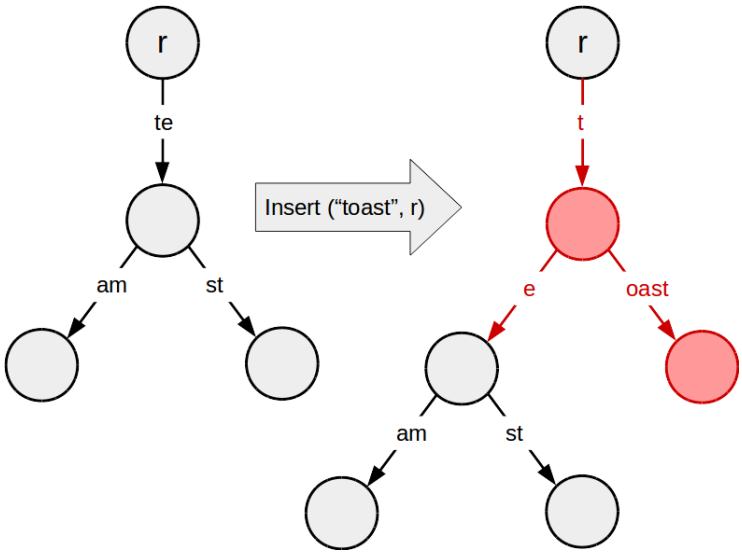
- *There is exactly one leaf for each string stored in the trie.*
- *Branch nodes represent the longest common prefixes of the strings its children represent.*
- *Any node is either a leaf or it has at least one child.*

The operation Insert(s, r) inserts string s into the tree with root r. The following examples are inspired by examples found on the Wikipedia page for radix trees [5].

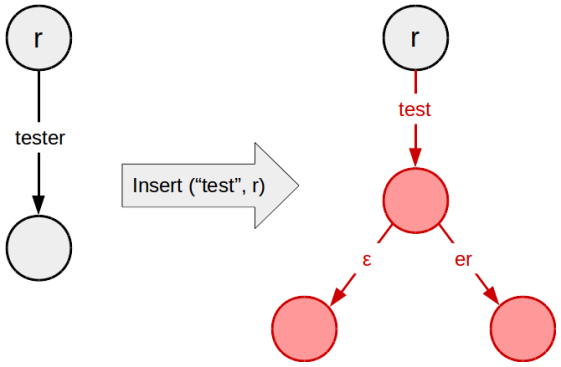
Example 19. *Inserting the string* `team` *into a trie which contains the string* `test`*, making a new node with common prefix* `te`*.*



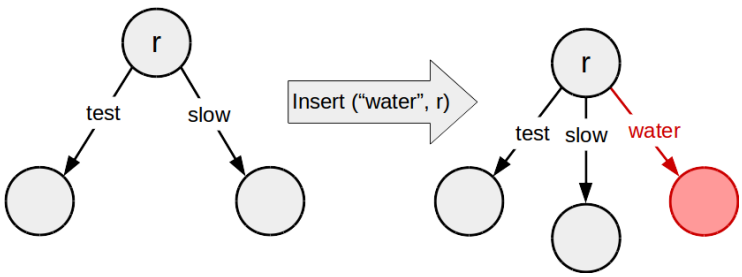
Inserting the string `toast` *into the above, making a new sub-node with common prefix* `t`*.*



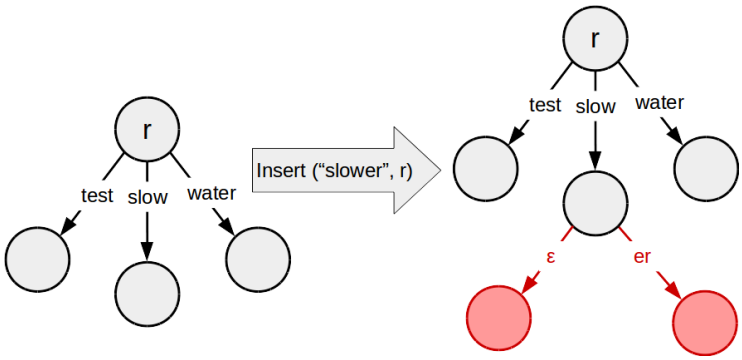
Inserting string `test` *into the trie with string* `tester`*, creating a new null node to represent the common prefix* `test`*.*



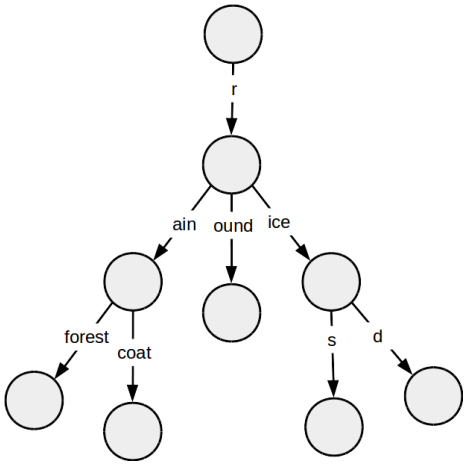
Inserting the string `water`*, into the trie containing the strings* `test` *and* `slow`*, will create a new substring containing* `water`*.*



Inserting string `slower` *on the above, again the the common prefix with* `slow` *is kept as an empty node.*

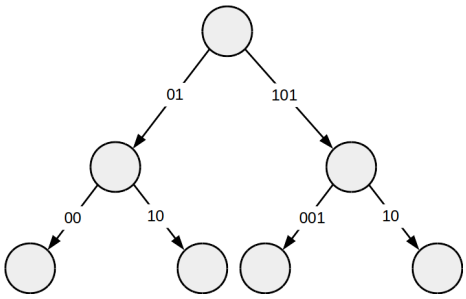


Example 20. *For example the PATRICIA trie containing the strings* `raincoat`, `rainforest`, `round`, `rices` *and* `riced` *would look like this:*



When storing only bit strings, each branch node will have exactly two children. There will be a branch the first time two strings differ.

Example 21. *Example of a trie containing the bit strings* `0100`, `0110`, `101001` *and* `10110`

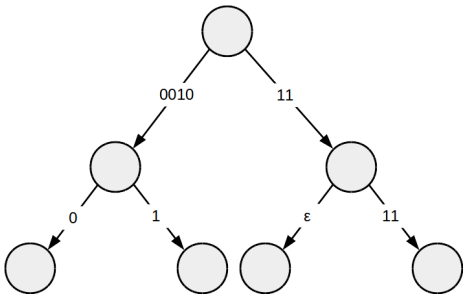


We implement PATRICIA tries as a record representing a node in the tree, with pointers *left*, *right*, *parent* and *string*. The string that a node points to contains only the suffix that this node adds to its parents string. So the root will always contain the empty string and one have to traverse the tree all the way from the root down to one of the nodes to read the string represented by that

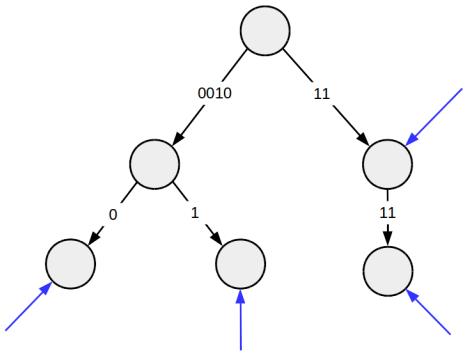
node. The left childs string always starts with a 0 and the right childs string always starts with a 1, thus the leftmost string is always the lexicographically least.

Our implementation differs from the PATRICIA trie definition in that, strings that are prefixes of other strings, which would normally be represented by an empty leaf, are represented by a branch node which may have only one child. We use a flag in the node structure to keep track of which nodes represent strings and which are just common prefixes.

Example 22. *Let* $W = \{00101, 00100, 1111, 11\}$. *Storing the set* W *in a PATRICIA trie looks like this:*



Storing the set W *in our implementation of PATRICIA tries looks like this. Where the blue arrows point to nodes that represent strings stored in the trie.*



Leaves that do not represent a string in the set are considered dead. The *clean-up* function will recursively go through the trie and remove dead leaves in a bottom-up fashion. When called on a given node, it will then call *clean-up* on its children. After the recursive calls have finished, it will whether the current node is alive, if this is the case nothing needs to be done. If it is not, it will check if it is a leaf. A dead leaf will naturally be deleted. If this was not the case it will check whether it has only one child, if this is the case, it can merge these two nodes into one. This is to maintain the PATRICIA structure after

other nodes have been deleted or inserted.

Algorithm 2: Cleanup

input : $node, T$
Cleanup($node.left, T$)
Cleanup($node.right, T$)
if $node \in T$) **then**
  terminate with success
if *node is a leaf* **then**
  delete reference from parent.
  delete($node$)
if *node has one child* **then**
  $child.string = node.string \cdot child.string$
  $child.parent = node.parent$
  if node has a parent it is either the left or right, we call this rift.
  $node.parent.rift = child$
  delete($node$)
terminate with success

Split works simply by checking whether all active nodes are contained within one branch from the root. If the root node is not active and it only has one child, we know that all bit strings in a set of node pointers T have the common prefix represented by that child.

Algorithm 3: Split

input : $root$
output: $(v, newRoot)$
if $newRoot \in T$ **then**
  output $(root, \epsilon)$
  terminate with success
if *root has one child* **then**
  output $(child.string, child)$
  $child.string = \epsilon$
  delete($root$)
  terminate with success
output $(root, \epsilon)$
terminate with success

Chapter 9

Implementations

We have made a few different implementations based on Algorithm 1. The difference between the algorithms primarily lies in the way we treat the set W.

9.1 The simple implementation

Here we will describe a simple way to implement Algorithm 1. Remember that W is a set of (v, q) pairs, where q is a state number and v is the lexicographically least output for a path $q_0 \overset{w/v}{\rightsquigarrow}_{np} q$, where $|W| \leq |Q|$ and the number of non problematic paths are finite.

Since there is at most one element in W for each state, and both the number of states and the lengths of outputs of non-problematic paths are finite, we can use an array of bit strings, indexed by states, to represent the set W. Bit string array $ArrW$ is the array representations of W, it is an array of bit strings indexed by state numbers.

So bit string $ArrW(q)$ is the lexicographically least output v for a path $q_0 \overset{w/v}{\rightsquigarrow}_{np} q$ from initial state to q, where w is the input read so far. If no such path exists we give it the special value NULL.

The simple implementation is implemented as follows:

Algorithm 4: Parsing with arrays

input : M, s
output: *Lexicographical least accepting path*
$ArrW :=$ StepComplete($\{(\epsilon, q_0)\}, \epsilon$)
for $a \in s$ **do**
  $ArrW :=$ StepComplete($ArrW, a$)
if $ArrW(q_f) \neq$ NULL **then**
  output $ArrW(q_f)$ and terminate with success
else
  terminate with error

The algorithm is as Algorithm 1, though we do not use the split operation. For each state in the state set it will find the lexicographical least output, from a given input letter and assign that as the new output string for that path.

We compute M, which is an array of relation matrices generated from the

NFA during preprocessing which is unaffected by the length of the input string. Bit string $M(a)_{q_i,q_o}$ is the lexicographically least output v from a path $q_i \overset{a/v}{\rightsquigarrow}_{np} q_o$ that reads the character a. Bit string array $M(a)_{q_i}$ corresponds to an array implementation of the set filter(closure$_\Delta$(step$_\Delta$($\{(\epsilon, q_i)\}, a$))), $a \in \Sigma$, similar to how we used $ArrW$ to store the set W.

In the case of $a = \epsilon$, $M(\epsilon)_{q_i}$ corresponds to the array implementation of filter(closure$_\Delta(\epsilon, q_i)$).

The MIN operation repeated for each state, corresponds to filter(W) and $ArrW(q_i) \cdot M(a)_{q_i,q_o}$ repeated for each state corresponds to closure$_\Delta(step_\Delta(W, a))$.

Algorithm 5 is the StepComplete used in Algorithm 4.

Algorithm 5: StepComplete

input : $ArrW, a$
output: $newW$
Let $newW$ be a new array of size $|Q|$
for $q_o \in Q$ **do**
  $newW_{q_o} :=$ MIN($\{ArrW(q_i) \cdot M(a)_{q_i,q_o} \mid q_i \in ArrW\}$)
return $newW$

9.2 PATRICIA implementation

In the PATRICIA implementation the set W is implemented as a PATRICIA trie data structure. $PatW$ is a sorted list of (p, q) pairs, where $q \in Q$ and p is a pointer to a node in the PATRICIA trie, representing the lexicographically least output v in a path $q_0 \overset{w/v}{\rightsquigarrow}_{np} q$, where w is the input read so far. The elements in $PatW$ are sorted by the bit strings, pointed to by the p, in lexicographically increasing order. We use a sorted list for $PatW$ because it lets us implement the filter operation efficiently. As the lexicographical order depends on the leftmost bit first, when we did the operation $ArrW(q_i) \cdot M(a)_{q_i,q_o}$ the path $q_0 \overset{a/v}{\rightsquigarrow}_{np} q_i$ with the lexicographically least output will be the path that is extended to become the path $q_0 \overset{a/v}{\rightsquigarrow}_{np} q_o$ with the lexicographically least output.

lM is an array of relation list arrays generated during preprocessing, which is unaffected by the length of the input string. $lM(a)_{q_i}$ is a list of (v, q_o) pairs, where t is the lexicographically least output from a path $q_i \overset{a/v}{\rightsquigarrow}_{np} q_o$, reading the character a. Like $PatW$, each list in $lM(a)$ is sorted by v in lexicographically increasing order. The suffixes in $lM(a)$ are sorted in the same way in case there are more paths from q_i to q_o where we always need the one with the lexicographically least output.

Initially the PATRICIA trie storing W is empty and consists of just one node called "root", whose bit string is ϵ. Since we are working with bit strings a node can have at most 2 children. So we refer to them as "node.left" and "node.right". "node.string" is the string that this node appends to its parents string.

The following is the algorithm used when using PATRICIA tries. We now use the split function, and we use a clean-up function. StepCompleteP has the same purpose as the function used for the array implementation, though the algorithm differs slightly. Clean-up is a function that maintains the PATRICIA trie by removing dead branches and merging inactive nodes with only one child with their child, a node is inactive if PatW does not contain a pointer to this

node. Split does as the original implementation describes. StepCompleteP checks for each pair $(p, q_i) \in W$ whether there is a path $q_i \overset{w/v}{\rightsquigarrow}_{np} q_o$ for any q_o not already present in a pair in $NewW$, If there is, we add it to $NewW$ paired with a pointer to the concatenation of the bit string pointed to by p and the output v, which we have inserted in the PATRICIA trie. We insert the new string by calling insert(v, p) which inserts the suffix at the end of the string pointed to by p.

Algorithm 6: StepCompleteP

input : $PatW, a$
output: $newW$
Let $NewW = []$
for $(p, q_i) \in W$ **do**
  for $(v, q_o) \in lM(a)_{q_i}$ **do**
    if *not* $(\exists p'.(p', q_o) \in newW)$ **then**
      $NewW \cdot (\text{insert}(v, p), q_o)$
return $newW$

The corresponding function to Algorithm 1, with lists.

Algorithm 7: Parsing PATRICIA Implementation

input : M
output: *Lexicographical least accepting path*
$PatW :=$ StepCompleteP$([(root, q_0)], \epsilon)$
for $a \in s$ **do**
  $PatW :=$ StepCompleteP$(PatW, a)$
  Cleanup$(root)$
  $(root, v) =$ Split$(root)$
  output v
if $\exists p.(p, q_f) \in PatW$ **then**
  output getString(p) and terminate with success
else
  terminate with error

Where getString is the function that gets a string from a PATRICIA trie, by recursively going up to nodes to the root. The clean-up function is as defined in the PATRICIA section, likewise with the split function.

9.3 Simple implementation with lists

We made a third implementation, like the PATRICIA implementation it uses lists for the set W, but it does not store the strings in a PATRICIA trie, instead, the pointers point directly to strings. $lM(a)$ is implemented exactly like in the PATRICIA implementation. We also perform split on this implementation, it works as described by Algorithm 3.

The purpose of this implementation was to isolate the effect of storing the strings in a PATRICIA trie by reducing the difference between the PATRICIA implementation and the reference implementation. We run most of our tests on all three implementations.

Chapter 10

Testing

In this section we will explain our testing procedures and the observations we made. We use the following shorthands when refering to the three different implementations:

- *Array implementation* refers to the implementation that uses an array of strings to store the set of bitstrings and matrices for storing the transitions relations.
- *PATRICIA implementation* refers to the implementation that stores the bitstrings in a PATRICIA trie and uses lists instead of matrices for the transition sets.
- *List implementation* refers to the implementation that uses the same list structure as the PATRICIA implementation but does not store the strings in a PATRICIA trie and instead uses pointers directly to the full strings.

We used unit tests to confirm that our implementations are correct. The unit test consists of a long list of RE, input string, expected output, triples. We used test cases from a test suite made by Andrew Svetlov[12], in addition to some that we created ourselves. We modified these test by also appending the expected output. All our implementations passes the unit test.

Unfortunately there are no existing implementations that we know of that produces the same outputs that we do, most implementations available online only perform matching. It would have been preferable to compare our PATRICIA implementation with implementations made by others, but in lack of better options we had to use our own alternative implementations for comparison. To compare the inplementations we used 4 performance tests which differ in the RE used.

- The first test used the RE

  $$E_{test1} = \Big([\underline{\mathbf{a}}, \ldots, \underline{\mathbf{z}}] \Big([\underline{\mathbf{a}}, \underline{\mathbf{b}}, \underline{\mathbf{c}}]^+ \mid [\underline{\mathbf{a}}, \ldots, \underline{\mathbf{w}}]\Big)?\Big)^\star$$

  It is an RE which uses the entire English alphabet as its alphabet, it has a star height of 2 and does not contain any problematic paths. The language for E_{test1} contains every combination of letters from the English alphabet and we generated inputs for it by writing the desired number of random letters to a file.

- We also designed a test which uses the RE

$$E_{test1}|E_{test1}$$

to show how the run time scales with the expression. In this test, we do not benefit from the split operation at all. Because any accepted input will result in two paths that are identical with the exception of the very first bit, which will prevent us from outputting a longest common prefix. Its run time should be about twice that of running E_{test1}, because we have to do the same work for each side of the expression. It takes the same input as the first test.

- Another test for showing how run time scales with the expression is one that uses the RE

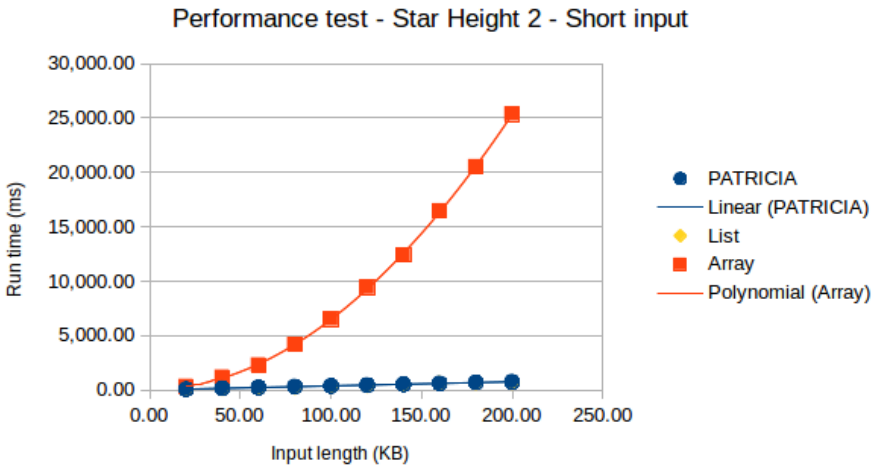
$$E_{SH5} = (((((((a+)b)+)c)+)d)+)e)+$$

which has a star height of 5. We also generated random inputs for this one, but the process is a little more complicated than for the previous ones as it does not simply accept all combinations of characters.

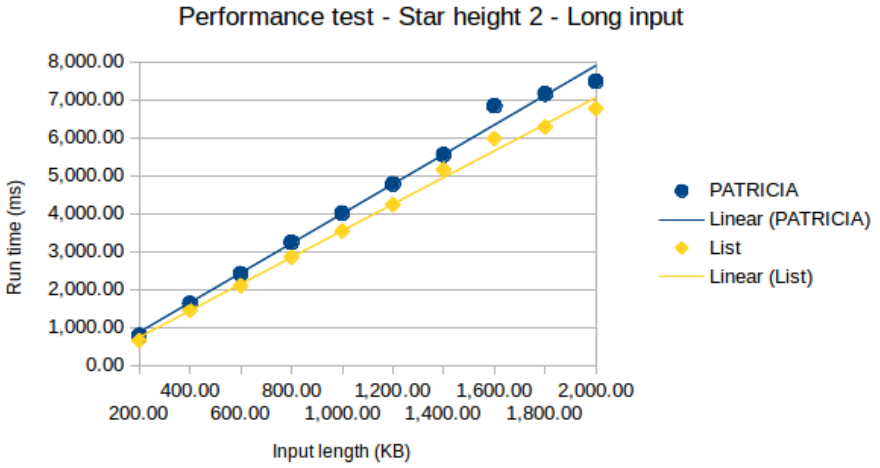
- The last type of test we did is an attempt to make a realistic case and uses an RE that would often be used in the real world. The RE used in this test matches any email address. It is a modified RE_{email} from Example 5, though instead of using E_{255} we use $E_{299} = [\underline{\mathbf{0}}, \ldots, \underline{\mathbf{2}}][\underline{\mathbf{0}}, \ldots, \underline{\mathbf{9}}][\underline{\mathbf{0}}, \ldots, \underline{\mathbf{9}}]$. The input for this is also generated with randomly assigned letters and numbers. The probability for it to end with an IP address is 50 percent.

The test results can be seen in Appendix A, we will present the graphs for the results here.

For E_{test1} we generated 10 input files with between $200,000$ and $2,000,000$ random alphabet characters respectively, with each file having 200.000 more characters than the previous one. For both the Simple List and PATRICIA implementation, we performed 10 simulations and registered the average run time for each file:

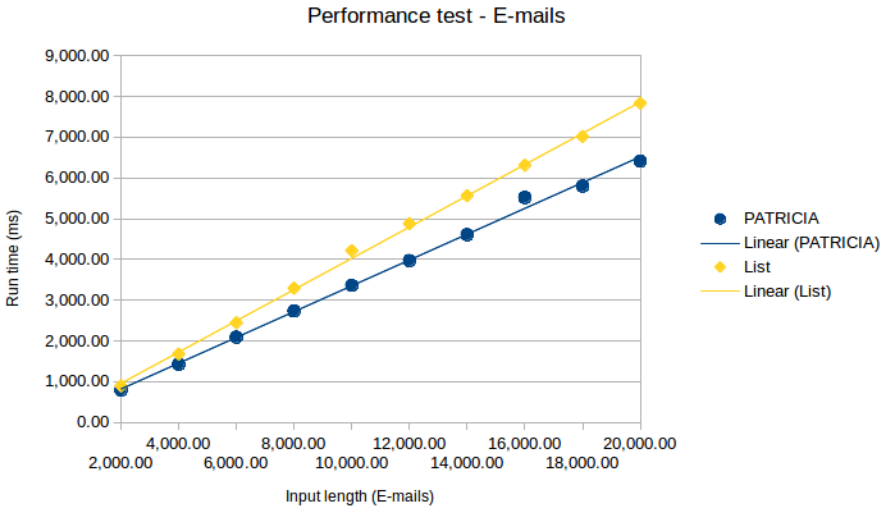


We see that the Array implementation has an exponential asymptotic run time, while both our PATRICIA and List implementations appear to be linear as expected. The exponential increase in run time of the Array implementation could be do to an error in the implementation causing the *(*Split) function to not work. If *split* is not used the length of the bit strings that have to be compared and concatenated will increase with the input length.



We see that both implementations runs in linear asymptotic time as expected. Also our PATRICIA implementation is a little slower in this case with a throughput of 0.25MB/s compared to the List implementations 0.28MB/s.

Here we perform the same test on the E-mail expression. We generated files full of randomly generated E-mails for input, each E-mail having an average size of 205 bytes.



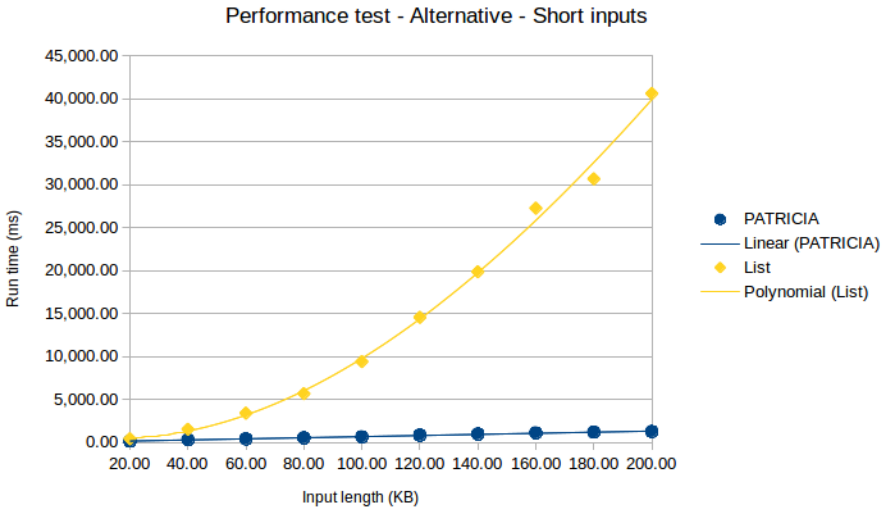
Again both run in linear time, but for this expression the PATRICIA implementation is faster with a throughput of 0.6MB/s compared to the 0.5MB/s of

the List implementation.

Here are the results for the star height 5 expression:

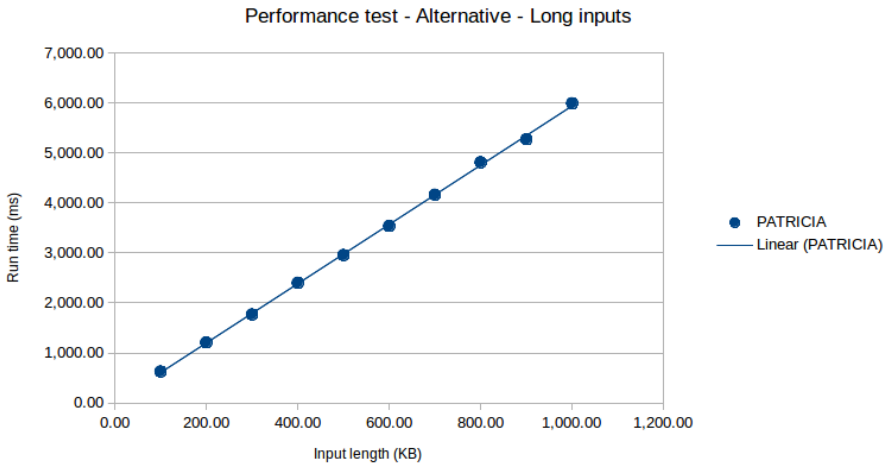
Again both run in linear time. This time the PATRICIA implementation is faster again with a throughput of 0.6MB/s compared to the 0.5MB/s of the List implementation.

The test using the expression $E_{test1}|E_{test1}$ was more interesting.



This time the LIST implementation runs in exponential asymptotic time while the PATRICIA implementation appears to still run in linear time. Without the split operation, the bit-strings maintained by the List implementation become longer with the input, while the PATRICIA implementation still benefits from the *Cleanup* operation maintaining the PATRICIA trie datastructure.

To get a better view of the run time of the PATRICIA implementation on this expression, we run it by itself and on some larger inputs.



We see that the PATRICIA implementation still runs in linear time, and with a throughput of 0.14MB/s parsing with this expression takes twice the time of parsing with just E_{test1} as expected.

Chapter 11

Conclusion and future work

From our testing we can conclude that our implementation performs in the expected asymptotic run time, in that our PATRICIA implementation runs linearly by the input given. Its run time also increases by the number of states that it has to maintain, as showed by our second example. It can be seen that PATRICIA tries can work as an efficient method of storing the outputs from a path through the FST by a given input. The performance of this implementation, beats the implementations, that simply stores their output in a string, by reducing the amount of memory and work needed for strings, and in some cases even suffixes of strings, with a common prefix.

Future work would be testing memory usage and comparing the PATRICIA implementation to more implementation, preferably some that are made by other people. It would be interesting to compare it with a python implementation that uses pythons "re" library.

Further improvements could be made in the FST generation, some cases can be made more efficient, however this is not a necessary improvement, and it could improve the performance of all implementations, so it will not tell much about the efficiency of PATRICIA tries. These improvements would lie in the preprocessing, and could be to determine when given $E_1|E_2$, where $\mathcal{L}(E_2) \subseteq \mathcal{L}(E_1)$, to just use E_1 as it will always contain the shortest prefix. As the code was written in c++, we might be able to improve the performance in some areas by constructing our own structures, and store the memory used more efficiently, such that accesses can be performed faster. Other ways to continue the work could be to find a more efficient method of determining for any given state, whether all continuing paths have been found. Or to find another way to do the update function, as our current solution is to preprocess the FST and generate the extension paths, we could do this in other ways, such as running on the FST itself. Other data structures could also be implemented, and tested against our implementation. Such as using matrix multiplication on our generated matrices which could be performed multi-threaded.

Bibliography

[1] Grathwohl, Henglein, Rasmussen, Søholm, Tørholm, "High-performance streaming regular expression processing using streaming string transducers", under preparation, 2015.

[2] Grathwohl, Henglein, Rasmussen, "Two-pass greedy regular expression parsing", CIAA, 2013.

[3] Grathwohl, Henglein, Rasmussen, "Optimally streaming greedy regular expression parsing", ICTAC, Springer, September 2014.

[4] Morrison, "PATRICIA–Practical Algorithm To Retrieve Information Coded In Alphanumeric", JACM, 1968.

[5] See Wikipedia, "Radix tree", for a basic description and for some links to implementations, e.g. in the Linux kernel.

[6] Lasse Nielsen and Fritz Henglein, "Bit-coded Regular Expression Parsing", In Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA), Lecture Notes in Computer Science (LNCS), Springer, May 2011.

[7] Grathwohl, Henglein, Rasmussen, "A Crash-course in Regular Expression Parsing and Regular Expressions as Types", draft, 2014.

[8] Henglein, "Streaming computation of lexicographically least output of transducers", notes, 2015.

[9] Grathwohl, Henglein, Rasmussen, "Optimally streaming greedy regular expression parsing", In Gabriel Ciobanu and Dominique Méry, editors, Proc. 11th Interna- tional Colloquium on Theoretical Aspects of Computing (ICTAC), volume 8687 of Lecture Notes in Computer Science, pages 224–240. Springer International Publishing, September 2014.

[10] S.C. Kleene, "Representation of events in nerve nets and finite automata", Automata studies, 1956.

[11] K. Thompson, "Programming techniques: regular expression search algorithm", Commun. acm, 1968.

[12] `<https://hg.python.org/cpython/file/178075fbff3a/Lib/test/re_tests.py>`

Appendix A

Test results

For this appendix we will use the following shorthands for the machine that ran the test: Runes' machine:
Ubuntu version 15.04 64bit running in a virtual box with access to 4gb RAM and 4 cores with 2.3 ghz. Emils' machine:
Ubuntu version 15.04 64bit on a machine with 3.7gb RAM and 4 cores with 2.5 ghz.

We use the following shorthands for naming the implementations in the schema below:

- Simple = The Simple implementation that uses an array of strings for the set og bit strings.
- Patricia = The implementation that uses a PATRICIA trie for storing the set of bit strings. And has the transition arrays replaced by lists.
- List = The Simple implementation that has had the array structure replaced by a list structure.

Star height 2 test with short inputs on Runes machine (VB):

Input size (KB)	20	40	60	80	100	120	140	160	180	200
Array	331	1,166	2,286	4,180	6,544	9,444	12,410	16,470	20,527	25,370
PATRICIA	82	163	246	327	384	461	535	621	691	770
List	74	148	208	274	352	424	520	626	688	706

Star height 2 test with long inputs on Runes machine (VB):

Input Size(KB):	200	400	600	800	1,000	1,200	1,400	1,600	1,800	2,000
PATRICIA(ms)	783	1627	2411	3240	4010	4778	5547	6836	7153	7480
Simple list(ms)	650	1448	2097	2848	3537	4231	4152	5975	6277	6764

Email tests on Emils machine

Emails:	2,000	4,000	6,000	8,000	10,000	12,000	14,000	16,000	18,000	20,000
Filesize(KB):	409	815	1,224	1,631	2,049	2,451	2,852	3,263	3,674	4,080
PATRICIA(ms)	793	1424	2092	2735	3362	3973	4607	5521	5801	6410
Simple list(ms)	895	1675	2444	3294	4217	4879	5565	6313	7017	7835

Star height five test on Runes machine (VB)

Input Size(KB):	59	122	187	252	310	370	439	501	558	620
PATRICIA(ms)	106	232	314	418	519	618	726	828	914	1041
Simple list(ms)	129	248	352	473	584	696	821	945	1040	1150

The alternative test on small inputs on Runes machine (VB)

Input Size(KB):	20	40	60	80	100	120	140	160	180	200
PATRICIA(ms)	150	315	416	543	650	846	984	1116	1195	1272
Simple list(ms)	426	1532	3409	5676	9407	14565	19868	27275	30687	40617

The alternative on long inputs, only PATRICIA on Emils machine

Input Size(KB):	100	200	300	400	500	600	700	800	900	1,000
PATRICIA(ms)	628	1209	1768	2401	2955	3539	4163	4811	5270	5990