FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Genome pattern matching using regular expressions

Simon Nicolai Lefoli Maibom - xvm226

Arinbjörn Brandsson - hkt789

Martin Simon Haugaard - cdl966

Supervisors

Rasmus Fonseca
Niels Bjørn Bugge Grathwohl
Ulrik Rasmussen
Martin Asser Hansen

June 6, 2015

Abstract

We examine a regular expression based method for pattern-matching which support errors, as an alternative to scan_for_matches currently used by biologists. We present an altered Thompson NFA, the tagged NFA which introduces three new types of transitions, and examine preformance based on real-life human genome data. Currently we cannot match the preformace of scan_for_matches, but further optimization is needed.

Contents

1 Introduction

When the Human Genome Project [**?**] (a project which had the goal of sequencing all 22 chromosomes of the human genome) was launched in 1990, the project was budgeted to cost 3 billion dollars and was estimated to take fifteen years to complete. However as technology progressed, the project managed to complete its goal two years earlier than expected, in 2003. This was made possible because of the rapid advancements in genome sequencing, and the advancement has not stopped since. This has led to decreasing costs of sequencing RNA and DNA, meaning biologists has access to greater amounts of data than before. However the technology to process these amounts of data have not progressed at the same pace as sequencing. Scan_for_matches is a tool for pattern-matching, which searches through data files to match a pattern specified by a user. While scan_for_matches has proven to be a fast and reliable tool, due to the amount of data it shifts through, a faster alternative is desired.

In this thesis, we provide an alternative to scan_for_matches based on automata theory and regular expressions. While the implementation currently does not match scan_for_matches' speed, with optimization it will. We will discuss the implementation's strengths and weaknesses, and describe what future work with the implementation will involve. Our implementation can be found at "`https://github.com/smaibom/bach_2015/tree/master/Implementation/src`".

2 Problem Analysis

The functionality of scan_for_matches dictates what our solution must be able to do. While a more indepth analysis of the functionality of scan_for_matches can be found in section 7, the requirements for our solution are as follows:

1. Read a data file
2. Match
   (a) with errors allowed
   (b) a previously found match
   (c) a modified pattern
3. Return matches with their position

While some of the functionality (items 1 and 3) will be trivial to implement since they are standard functions of most programming languages, there are some challenges to be found in regards of what we must match. Matching with errors allowed are not supported natively in regular expressions, and matching a modified text, which may first be determined at runtime, will be challenging to implement with automata.

3 State-of-the-Art

The current tools readily available that provides scan_for_matches like functionality are RE2, Google's regular expression library because of how it handles alternations as well as its linear running time and TRE, python's regex library and non-deterministic grep (NR-grep), since all three allows errors in its results and supports backreferencing.

3.1 RE2

RE2 is Google's regular expression library written in C++. It does not support backreferencing, but claims to run faster if a pattern has a high degree of alternations [?]. Due to RE2's lack of backreferencing and support for matching with errors, it would be unable to properly reproduce scan_for_matches' functionality. The project can be found on its github page: `https://github.com/google/re2`.

3.2 TRE

TRE was created by Ville Laurikari for his master's thesis in 2001[?], and is a regular expression engine which supports backreferences and matching with errors. Because of this, TRE is the best candidate for modification in order to simulate scan_for_matches functionality (see Section 8).

3.3 Python's Regex Library

The python module regex [?] is an alternative regular expression module to the native python module re created by Matt. It allows the specification of mismatches in its search terms, however because of the lack of theoretical documentation pertaining to the module and the nature of the language the module was written for, we didn't see the regex library as a potential alternative for scan_for_matches.

3.4 NR-grep

NR-grep [?] is a pattern matching tool written in C by Gonzalo Navarro in 2000. It allows backreferences as well as matching with errors, and would have been a candidate for modification alongside TRE had we learned about it earlier. However, because we learned about the tool late in the process, we did not have time to work with it.

4 Deoxyribonucleic and Ribonucleic Acids

4.1 DNA

Deoxyribonucleic acid (DNA) is a macro molecule composed of nitrogenous bases joined by a deoxyribose-phosphate. DNA is mostly found in nature as helixes, where two strands have bonded. Similarly to RNA, DNA has four nitrogenous bases, and shares three of the four that RNA have, (guanine, adenine and cytosine). However instead of uracil, the fourth base is thymine (T).

4.2 RNA

Ribonucleic acid (RNA) is a macro molecule composed of nitrogenous bases joined by ribose-phosphate backbone into long strands. The possible nitrogenous bases, or bases for short, that can be joined by the backbone are guanine (G), adenine (A), uracil (U) and cytosine (C). In nature, the predominant form of RNA are as a single-stranded chain that can fold back on itself or bundled with other chains to form a structure. This flexibility of the backbone that allows for the chain to fold in on itself is possible because the RNA's backbone is composed of a sugar called ribose, which allows more flexibility compared to its other form, deoxyribose, used in deoxyribonucleic acid (DNA).

The bases found in RNA can form hydrogen bonds with each other, though not all bases can form bonds with each other. Bases that can bond with each other are G with C, and A with U. Basepairing dictates the shape of the RNA molecules, and will be elaborated on in section 4.3.

4.3 Secondary Structure

The secondary structure of DNA and RNA describes how the bases of the strand has bonded to itself. The secondary structure can change if the strand is damaged or has mutated, causing it to gain or lose bases. Below are examples of three common secondary structures.

Bulge

A bulge occurs when one or more bases have no base to bond with, and these bases are surrounded by bases that have bonded. This causes the bases to get pushed out slightly, resembling a bulging growth. This type of structure occurs when one or more bases has been inserted or deleted. If a base has been inserted then it will have no base to bond with, and if a base has been deleted then the previously-bonded base will have no base to bond with. Figure 1 shows a bulge.

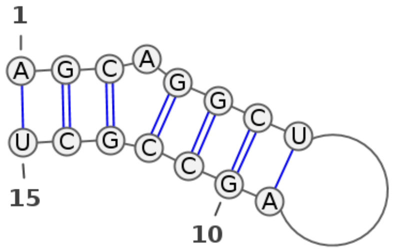


Figure 1: The RNA sequence AGCAGGCUAGCCGCU. Note the bulging A at position 4.

4.3.1 Interior Loop

An interior loop is when two or more opposing bases are not complementary and can not bond, causing them both to bulge. This occurs when one or more consecutive bases mutate to another base. Figure 2 shows an interior loop.

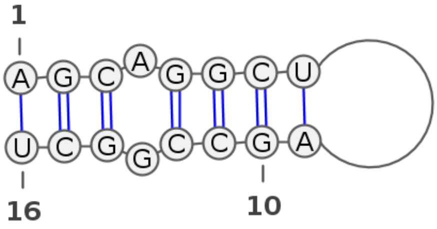


Figure 2: The RNA sequence AGCAGGCUAGCCGGCU. Note the bulging A at position 4 and G at position 13 creating a loop inside the bonded strand.

These interior loops vary in size, and can have differing amount of bases on either side of the strands.

4.3.2 Stem Loop

A stem loop, also known as a hairpin loop, occurs when a strand bonds with itself, but leaves a sequence of bases sticking out that does not bond with anything. This kind of loop occurs typically in RNA as they are single-stranded, but may happen in single stranded DNA. Figure 3 shows a stem loop. An important thing to take note of is how the sequence can be seen as one long strand that starts from the adenine bases that binds with the uracil bases, loops around without binding to anything and finally become the uracil bases that the adenine bases from the start binds with. This means that the stem loop can be written as one continuous sequence of bases; `AAAAUUGGUCUUUU`. Since we can define a stem loop, we can, with the right tools, search through a file documenting the bases of a nucleic acid and find all stem loops.

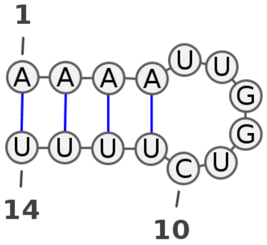


Figure 3: A stem loop of the RNA sequence `AAAAUUGGUCUUUU`.

5 Regular expressions

A regular expression(RE) is sequence of characters that define a search pattern. To explain what a RE is, we must first introduce languages and alphabets. All literals will be written using the typewriter font, to distinguish between literals and other text.

Definition 1. *An alphabet Σ is a finite non-empty set of letters.*

Definition 2. *A language L is a subset of all strings formed over Σ : $L \subseteq \Sigma^*$*

Example 5.1. *If we have a DNA sequence string, the alphabet Σ consists of the literals* {`t`,`g`,`c`,`a`}. *and the language contains strings formed by the literals from this alphabet. For example* `"gtcaaa"` *or* `"gtcaaat"`.

Definition 3. *A regular expression is described by the following grammar:*

$$E ::= a|0|1|E_1 + E_2|E_1E_2|E^*$$

where E_1 and E_2 are RE's and $a \in \Sigma$

Definition 4. *The language interpation L(E) of a regular expression is:*

$$\begin{aligned}
L(0) &= \emptyset \\
L(1) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(E_0 + E_1) &= L(E_0) \cup L(E_1) \\
L(E_1E_2) &= \{w_1w_2 | w_1 \in L(E_1), w_2 \in L(E_2)\} = L(E_1)L(E_2) \\
L(E)^0 &= \{\epsilon\} \\
L(E)^n &= \underbrace{L(E)L(E)...L(E).}_{n\ times} \\
L(E^*) &= \bigcup_{n=0}^{\infty} L(E)^n
\end{aligned}$$

*[**?** , p.5 def. 3]*

With definition 4, we can now form regular languages.

Example 5.2. *Natural numbers can be described as a regular expression. Natural numbers have the alphabet $\Sigma =$* {`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`}*, so the regular expression for natural numbers would look like:*

$$E_{nat} = (\mathtt{1}+\mathtt{2}+\mathtt{3}+\mathtt{4}+\mathtt{5}+\mathtt{6}+\mathtt{7}+\mathtt{8}+\mathtt{9})(\mathtt{0}+\mathtt{1}+\mathtt{2}+\mathtt{3}+\mathtt{4}+\mathtt{5}+\mathtt{6}+\mathtt{7}+\mathtt{8}+\mathtt{9})^*$$

Example 5.3. *Given the alphabet $\Sigma =$* {`a`,`b`,`c`}*, with this alphabet, a regular expression $E =$* `c`*(*`a`*+*`b`*)* could be formed. The language L that is produced from this expression would consist of strings that is a* `c` *followed by zero or more as or bs for example* `"caaaa"`,`"cbb"`,`"cababba"` *and so forth. The language interpretation of E is*

$$\begin{aligned}
L(E) &= L(\texttt{c}(\texttt{a}+\texttt{b})^*) \\
&= \{w_1w_2|w_1 \in L(\texttt{c}), w_2 \in L(\texttt{a}+\texttt{b})^*)\} \\
&= \{w_1w_2|w_1 \in \{\texttt{c}\}, w_2 \in \bigcup_{n=0}^{\infty} L(\texttt{a}+\texttt{b})^n\} \\
&= \{w_1w_2|w_1 \in \{\texttt{c}\}, w_2 \in \bigcup_{n=0}^{\infty} (L(\texttt{a}) \cup L(\texttt{b}))^n\} \\
&= \{w_1w_2|w_1 \in \{\texttt{c}\}, w_2 \in \bigcup_{n=0}^{\infty} (\{\texttt{a}\} \cup L(\texttt{b}))^n\} \\
&= \{w_1w_2|w_1 \in \{\texttt{c}\}, w_2 \in \bigcup_{n=0}^{\infty} (\{\texttt{a}\} \cup \{\texttt{b}\})^n\} \\
&= \texttt{c} \bigcup_{n=0}^{\infty} \{\texttt{a}, \texttt{b}\}^n
\end{aligned}$$

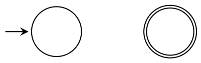
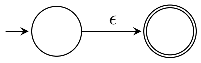
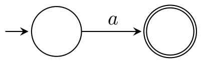
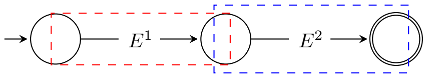
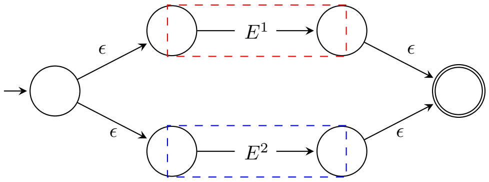
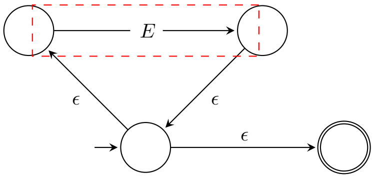
6 Nondeterministic Finite Automaton

A nondeterministic finite automaton(NFA) can be used to determine if a input string is matches a particular set of strings.

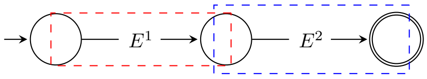
Definition 5. *An nondeterministic finite automaton (NFA) is a 5-tuple* $(Q, \Sigma, \Delta, q^s, q^a)$*, where* Q *is a finite set of states,* Σ *is the input alphabet, the initial state* $q^s \in Q$*, the accepting state* $q^a \in Q$ *and* $\Delta \subseteq Q \cdot (\Sigma \cup \{\epsilon\}) \cdot Q$ *is the transition relation.*

Each RE can be converted to an NFA, and vice versa. Table 1 shows how each RE can be converted into an NFA. The states in Q are represented as circles. The initial state q^s is shown by adding a small arrow pointing to it, and the accepting state q^a is shown as a double circle. Transitions in Δ are represented as $(q, a, q') \in \Delta$ or $(q, \epsilon, q') \in \Delta$, we write these transitions as $q \xrightarrow{a} q'$ or $q \xrightarrow{\epsilon} q'$. For subexpressions we use boxes and split the transition arrow, marking it with an E, to denote the NFA resulted from converting the expression.

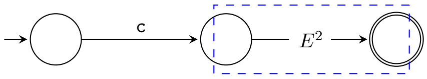
Table 1: Translation table from regular expressions to NFA

0	
1	
a	
E^1E^2	
$E^1 + E^2$	
E^*	

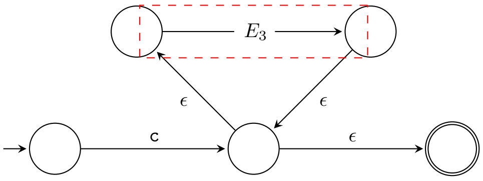
Example 6.1. *In example 5.3, the expression* $\mathtt{c}(\mathtt{a}+\mathtt{b})^*$ *was used. The NFA buildup is done like the language interpretation in the example. We first construct the* E_1E_2 *expression, where* $E_1 = \mathtt{c}$ *and* $E_2 = (\mathtt{a}+\mathtt{b})^*$.



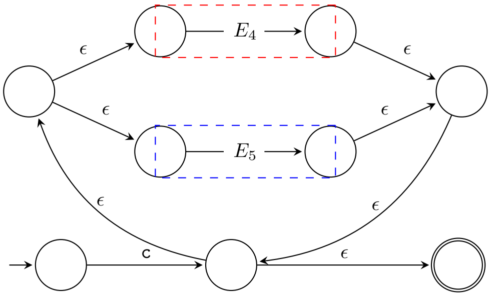
We first construct E_1



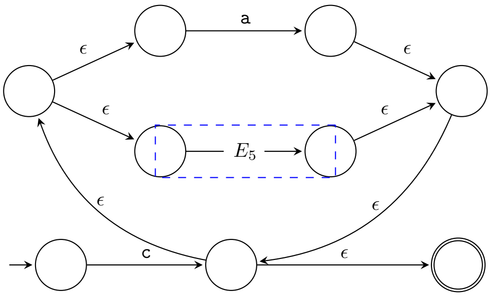
Constructing E_2 *results in a new subexpression* $E_3 = \texttt{a} + \texttt{b}$.



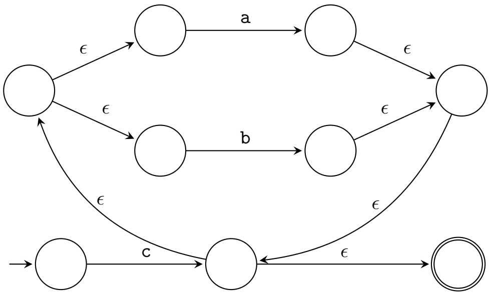
E_3 *produces the two new expressions* $E_4 = \texttt{a}$ *and* $E_5 = \texttt{b}$.



After constructing E_4



After constructing E_5 *we end up with the NFA of the expression* E



6.1 Matching using NFA

To match if a given string is accepted in an NFA, two functions *ϵ-closure* and *reachable* of the simulation algorithm 1 are introduced.

Definition 6. *Given a set of NFA states M, the ϵ-closure of M is a set of states that are reachable from states in M by following any number of ϵ-transitions in Δ.*

$$\epsilon\text{-}closure(M)\ = M \cup \{q' | q \in \epsilon\text{-}closure(M) \text{ and } (q, \epsilon, q')\ \in\ \Delta\}$$

*[**?** , p. 34, def 2.2]*

Definition 7. *Given a set of NFA states and a input symbol* `a`*, the reachable states of M are a set of states that are reachable from states in M by following transitions in Δ which match the input symbol a.*

$$reachable(M, a)\ = \{q' | q \in M, (q, a, q') \in \Delta\}$$

Algorithm 1 NFA simulation

```
Require: $N$ is a NFA and $s$ is a string
Ensure: True if $s$ is accepted in $N$, False if $s$ is rejected
function SIMULATION($N(Q, \Sigma, \Delta, q^s, q^a), s$)
    $stateset \leftarrow \{q^s\}$
    for each symbol in $s$ do
        if $stateset = \emptyset$ then
            return False
        $next \leftarrow \emptyset$
        $states \leftarrow \epsilon$-closure($stateset$)
        $next \leftarrow reachable(states, symbol)$
        $stateset \leftarrow next$
    if $q^a \in stateset$ then
        return True
    return False
```

Example 6.2. *Given the the RE $E =$* `c(ab + a)*b`*, the resulting NFA N is seen in figure 4*

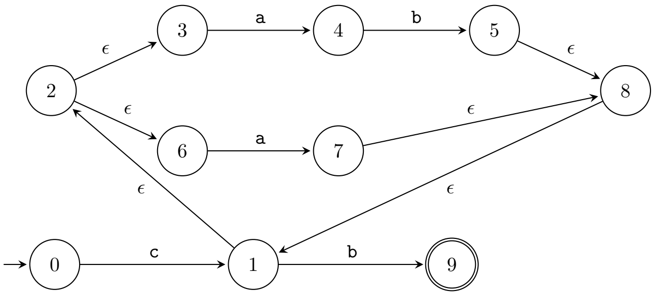


Figure 4: Figure of the NFA from the $E =$ `c(ab+a)*b`

We now want to see if the input string `caaabb` *is accepted into N. The initial stateset $\{q^s\} = \{0\}$*

Simulate(N,`caaabb`*)*

symbol `c`*:*	ϵ-*closure*$(\{0\})$	$= \{0\}$
	reachable$(\{0\}, \texttt{c})$	$= \{1\}$
symbol `a`*:*	ϵ-*closure*$(\{1\})$	$= \{1, 2, 3, 6\}$
	reachable$(\{1, 2, 3, 6\}, \texttt{a})$	$= \{4, 7\}$
symbol `a`*:*	ϵ-*closure*$(\{4, 7\})$	$= \{1, 2, 3, 4, 6, 7, 8\}$
	reachable$(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{a})$	$= \{4, 7\}$
symbol `a`*:*	ϵ-*closure*$(\{4, 7\})$	$= \{1, 2, 3, 4, 6, 7, 8\}$
	reachable$(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{a})$	$= \{4, 7\}$
symbol `b`*:*	ϵ-*closure*$(\{4, 7\})$	$= \{1, 2, 3, 4, 6, 7, 8\}$
	reachable$(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{b})$	$= \{5, 9\}$
symbol `b`*:*	ϵ-*closure*$(\{5, 9\})$	$= \{1, 2, 3, 5, 6, 8, 9\}$
	reachable$(\{1, 2, 3, 6, 5, 8, 9\}, \texttt{b})$	$= \{9\}$

After the final input symbol, it can be seen that the accepting state q^a is in the final stateset. So the string caaabb is accepted in N.

Example 6.3. *Using the NFA N from example 6.2 where $q^s = 0$, the simulation attempts to check if string* `cabbbb` *is accepted in N.*

Simulate(N,`cabbbb`*)*

symbol `c`*:*	ϵ-*closure*$(\{0\})$	$= \{0\}$
	reachable$(\{0\}, \texttt{c})$	$= \{1\}$
symbol `a`*:*	ϵ-*closure*$(\{1\})$	$= \{1, 2, 3, 6\}$
	reachable$(\{1, 2, 3, 6\}, \texttt{a})$	$= \{4, 7\}$
symbol `b`*:*	ϵ-*closure*$(\{4, 7\})$	$= \{1, 2, 3, 4, 6, 7, 8\}$
	reachable$(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{b})$	$= \{5, 9\}$
symbol `b`*:*	ϵ-*closure*$(\{5, 9\})$	$= \{1, 2, 3, 5, 6, 8, 9\}$
	reachable$(\{1, 2, 3, 6, 5, 8, 9\}, \texttt{b})$	$= \{9\}$
symbol `b`*:*	ϵ-*closure*$(\{9\})$	$= \{9\}$
	reachable$(\{9\}, \texttt{b})$	$= \emptyset$

The $\emptyset$ is reached at the 5'th input symbol of `cabbbb`*, resulting in the simulation to fail.*

6.2 Tagged NFA

The tagged NFA(TNFA) is introduced in [**?**]. It introduces the concept of tagging NFA transitions. A TNFA adds a new transition table which contains ϵ transitions for the 3 different missmatch types.

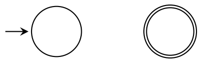
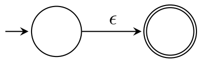
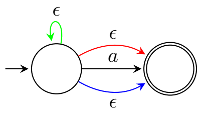
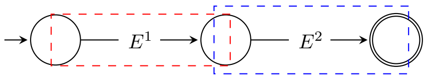
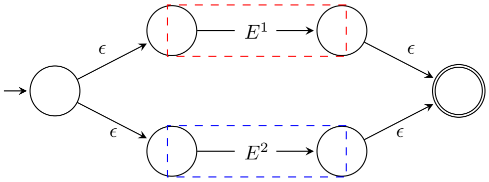
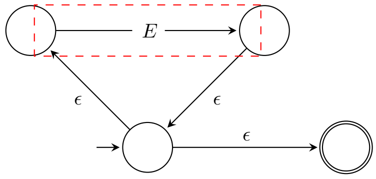
Definition 8. *A mismatch M is one of 3 types, insertion, deletion and alteration. Given a transition $(q, a, q') \in \Delta$ missmatches adds new ϵ transitions in the TNFA, where insertion is defined as (q, ϵ, q), deletion and alterations is defined as (q, ϵ, q')*

Definition 9. *A tagged NFA is a 6 tuple $(Q, \Sigma, \Delta, q^s, q^a, \Delta')$ where the first 5 elements is a standard NFA and Δ' is a set of 4 tuples containing ϵ transitions for missmatches. The type of Δ' is $\Delta' \subseteq Q\ x\ \{\epsilon\}\ x\ Q\ x\ M$.*

6.3 Constructing TNFA

TNFA adds a new literal construction literal rule. A new set of ϵ-transitions is added as shown in tabel 2 on the new literal construction rule. The new transitions added in Δ' is shown as a red arrow which denotes a deletion transition, a green arrow for the insertion transition and a blue arrow for the alteration transition.

Table 2: Translating table for literal construction of TNFA

0	
1	
a	
E^1E^2	
$E^1 + E^2$	 E^1 ϵ ϵ ϵ E^2 ϵ
E^*	

6.4 Simulating TNFA

TNFA simulation adds an additional argument for amount of mismatches allowed. The stateset is a 4 tuple of (Q, i, d, a) where i,d and a is a count of how many of each transition has been used. $\epsilon - closure$ and *reachable* transfers this count over to each of the states reached in these functions.

Algorithm 2 TNFA simulation

```
Require: $N$ is a TNFA and $x$ is a string,M is a 3-tuple of mismatches allowed
function SIMULATION($N(Q, \Sigma, \Delta, q^s, q^a, \Delta'), x, M$)
    $stateset \leftarrow \{q^s\}$
    for each symbol in $x$ do
        if $stateset = \emptyset$ then
            return False
        $next \leftarrow \emptyset$
        $states \leftarrow \epsilon$-closure($stateset$)
        $next \leftarrow reachable(states, symbol)$
        if $next = \emptyset$ then                                   ▷ Mismatch
            $next \leftarrow TNFA - trans(\Delta', states, M)$
        $stateset \leftarrow next$
    if $q^a \in stateset$ then
        return True
    return False
```

Algorithm 3

```
Require: $\Delta'$ is a tagged transition table, states is a set of 4 tuples with a state q and mismatches
    occured, M is a 3-tuple of mismatches allowed
function TNFA-TRANS($\Delta', states, M(ins, del, alt)$)
    $stateset \leftarrow \emptyset$
    for each state(s,i,d,a) in $states$ do
        if $i < ins$ then
            $stateset \leftarrow stateset \cup (tagreach(state, I)$
        if $d < del$ then
            $stateset \leftarrow stateset \cup (tagreach(state, D))$
        if $a < alt$ then
            $stateset \leftarrow stateset \cup (tagreach(state, A))$
    return $stateset$
```

7 Scan_For_Matches

Scan_for_matches is a pattern-matching tool created by Ross Overbeek, David Joerg and Morgan Price in C which searches through data files[1]. Users specify what they want to search for by defining a pattern, and scan_for_matches returns all matches that corresponds to the specified pattern.

Definition 10. *Let Σ denote an alphabet. Then we can define a* pattern unit *as follows:*

[1] http://blog.theseed.org/servers/2010/07/scan-for-matches.html

Pattern Unit	***Function of Pattern Unit***
h	*Match the sequence h, where h ∈ Σ∗*
n...m	*Match n to m characters where 0 ≤ n ≤ m*
x=n...m	*Match n to m characters, and label the sequence x*
x | y	*Match either pattern x or pattern y*
x[n,m,l]	*Match pattern x, allowing for n mismatches, m deletions and l insertions where n,m,l ≥ 0*
length(x+y) < n	*The length of patterns x+y<n where n > 0*
z={uv, vu}	*Create a pattern rule where u is the complement of v, and v is the complement of u, where u, v ∈ Σ, and call the rule z*
<x	*Match the reverse of pattern x*
∼x	*Match the reverse complement of pattern x using the G-C, C-G, A-T and T-A pairing rule*
z∼x	*Match the reverse complement of pattern x using pattern rule z={uv,vu}*
^ x	*Match only pattern x if it is at the start of a string*
x $	*Match only pattern x if it is at the end of a string*

Definition 11. *Let Λ be any pattern unit in definition 10. Let* $\mathtt{E} \in \Lambda$*. Let* `0` *be the empty string. Let P be a pattern that we are processing. A pattern may then be constructed as such:*

$$\mathtt{P} \;= \mathtt{E}\ \mathtt{P} \mid \mathtt{0}$$

Definition 11 states that a pattern may be any combination of the pattern units defined in definition 10.

Definition 12. *Let* Σ *be an alphabet. Let* $\mathtt{a} \in \Sigma$*. Let* ϵ *be the empty string. Then the language interpretation of definition 10 is defined as follows:*

$$\begin{aligned}
L(\epsilon) &= \emptyset \\
L(\mathtt{a}) &= \{\mathtt{a}\} \\
L(\mathtt{E}_1\ \mathtt{E}_2) &= L(\mathtt{E}_1)\ L(\mathtt{E}_2) \\
L(\mathtt{E}_1 \mid \mathtt{E}_2) &= L(\mathtt{E}_1)\ \cup\ L(\mathtt{E}_2) \\
L(\mathtt{E})^0 &= L(\epsilon) \\
L(\mathtt{E})^n &= \underbrace{L(\mathtt{E})L(\mathtt{E})...L(\mathtt{E})}_{n} \\
L(\mathtt{n}...\mathtt{n}) &= L(\mathtt{E})^n \\
L(\mathtt{n}...\mathtt{m}) &= L(\mathtt{n}...\mathtt{n}) \cup L(\mathtt{n}+1...\mathtt{n}+1) \cup ... \cup L(\mathtt{m}-1...\mathtt{m}-1) \cup L(\mathtt{m}...\mathtt{m}) \;=\; \bigcup_{n=\mathtt{n}}^{\mathtt{m}} L(n...n) \\
f(L(\Sigma)) &= \bigcup_{x\in\Sigma} f(L(x)) \\
L(<\mathtt{E}) &= \{w^R \mid w \in E\} \\
L(\sim\mathtt{E}) &= f(L(<\Sigma)) \\
L(\mathtt{length}(\mathtt{E} < \mathtt{n})) &= \{|\mathtt{v}| < \mathtt{n} \mid \mathtt{v} \in \mathtt{E},\ 0 < \mathtt{n}\}
\end{aligned}$$

Definition 12 defines the language interpretation of scan_for_matches. $f(L(\Sigma))$ defines a mapping [? , p. 60] which substitutes the current character with the complementary in alphabet Σ. Below is an example of a scan_for_matches pattern.

Example 7.1. *Say we want to write a pattern that finds the sequence* `GUUC`*, allowing one mismatch, followed by a random sequence which has a length between 3 and 5, followed by the reverse complement of the first sequence that we found. We can then write this as*

```
p1=GUUC[1,0,0] 3...5 ~p1
```

Example 7.1 matches a stem loop as described in section 4.3. Note that if we wanted to find all stem loops in a file where the bonded bases are of length 4, we would replace `GUUC[1,0,0]` with an arbitrary sequence of characters of length 4 by writing `p1=4..4 3..5 ~p1`.

8 TRE

Recall that in section 2, we determined that the challenge of our solution would lie in matching:

1. with errors allowed,
2. a previously found match, and
3. a modified pattern.

An implementation based on TRE would address two of the three previously mentioned problems. Item 1 would require approximate matching support, and TRE fully supports approximate matching. Item 2 could be resolved with backreferences (even if backreferences are computationally inefficient and not regular) and TRE also supports this. Item 3 means that sometimes we want to match a modification of a pattern found previously - like the reverse complement of a pattern. This would have to be implemented in TRE's parser (to denote a symbol for the modified patterns, e.g. the reverse or the complement of a pattern) as well as in TRE's basic functionality.

When analyzing TRE so we could start modifying the program, we discovered that TRE would define every newline as a delimiter. The delimiter specifies how the data should be broken up, so for every new line the current line would be loaded into the buffer and be processed. This in itself would not be a problem if not TRE would discard any match that was currently being processed when it reached a delimiter, causing matches that wrapped around two lines to be discarded. The fix to this was easy to make however; if a wrapper was created which would feed the text data to TRE, ignoring all newlines, then TRE would load the entire file into its buffer and would no longer cause it to discard potential matches that continued to the next line.

When we then tried to run a file through TRE which had no newlines, we discovered another feature of TRE which would not work for our project; TRE would only match one match per delimiter - the earliest, best-matching match (where a best-matching match is a match with the least amount of errors). Since we had to trim the text files for TRE so there were no newlines, TRE would only return one match. TRE was built around this feature, which led to the following design choices;

- the program runs through the data once to determine how many errors the best-matching match has, and then runs through the text file again, stopping when the best-matching match has been found,
- TRE ignores any matches that are not best-matching, meaning there is no way for TRE to identify and output acceptable matches.

This coupled with little documentation of how the code worked meant that it would take a long time to properly analyze TRE in order to find out how we could modify it to suit our project. At this point we decided that creating our own solution would be less time-consuming, while also allowing us to design our solution ourselves.

9 Our Implementation

We constructed a simple program, which would create a TNFA using the methods description in section 6.2 and use it to search data files for matches for a given pattern.

Our implementation supports a series of regular expression symbols, including +,* , |, ? along with concatenation of characters. This allows for construction of simple TNFAs from regular expressions, for example the regular expression "(GAT)+" would produce a structure as shown in figure 5

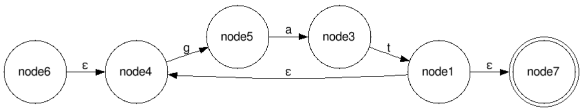


Figure 5: Example of how our implementation builds TNFA from regular expression (GAT)+

Each node in the figure has a number corresponding to the time it was created in the code, examining figure 5 we can see that the part having GAT was constructed first, and then the surrounding nodes responsible for the + were added onto that, much like one would expect from the description in section 6.2.

When there's a TNFA, our implementation will try each possible transition when matching, causing new states to be made every time two or more possible transitions are viable. For example if there's two epsilon transitions, as in node1 in figure 5, one state will move to node7 and terminate, and another to node4 and continue to match input until it either matches the pattern or is terminated.

If a state can not match a character directly, but it has available insertions, mutations or deletions, it will, for each allowed mismatch, create a new state, and move accordingly in the TNFA. This way we can guarantee that we will find every possible match for a given pattern. However it also causes an increased runtime given an increase in mismatches, causing one state to spawn up to three new states, and in worst case cause an exponential increase of states until the number of mismatches is exhausted, at which point the states will either match the pattern or be terminated.

10 Experimental Results & Tests

For this a virtual machine is created, using Oracle VirtualBox[2]. The machine running the virtualbox is running Windows 8.1 Pro x64 on an SSD, with 8,00 GB RAM, an AMD FX 4300 Quad-Core Processor 3.80 GHz, of which 1 core and 4096MB RAM was given to the virtual machine, which would run Ubuntu 14.04.2 LTS 64 bit.

We choose to test four different tools, scan_for_matches, TRE, Python using the Regex module and finally our own implementation. Since our implementation is focused on supporting insertion, deletion and mutation on a sequence, a simple DNA sequence $TGCAAGCGTTAAT$ with variable insertions is chosen as the search pattern. Each test was executed on a series of data files, a total of of 10 times, given an average runtime which was used in the following results.

The testing data was selected to be in the form of the human genome chromosome sequences. Figure 6 shows a series of fasta files, these files include nucleotide sequences, which all differ in size, decreasing from chr1 to chr22[3]. These fasta files are the kind of data which scan_for_matches is expected to run, and thus excellent for benchmark testing. It is worth noting however, that each of these files' lines are 50 characters long, as TRE matches through lines separately, and longer or shorter line sizes might affect the performance of TRE in running time and hits.

[2] virtualbox.org

[3] http://hgdownload.cse.ucsc.edu/goldenPath/hg18/chromosomes/ JUNE 2015

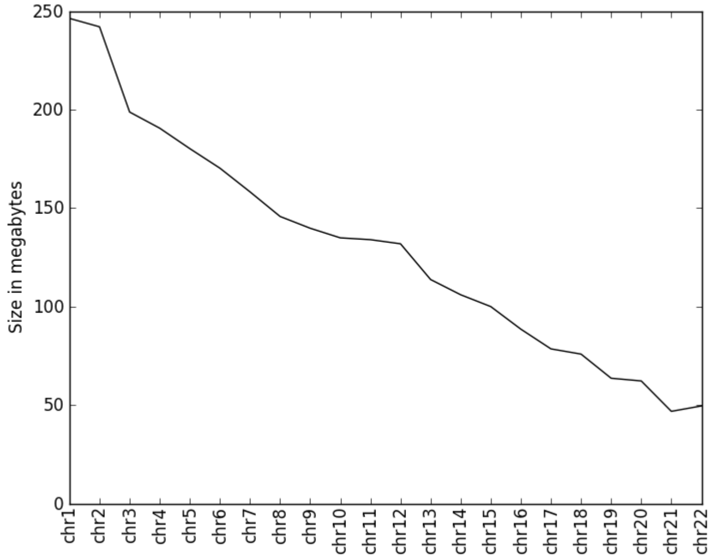


Figure 6: Sample files used for benchmarking, files range from 246.3 Megabytes to 46.8 Megabytes in size

First test was to see the runtime of each program, having no mismatches in the mentioned pattern $TGCAAGCGTTAAT$. Figure 7 displays the results, and it is evident that scan_for_matches is faster all the alternatives. Our implementation and python has slightly different runtime, our implementation being slightly faster, and finally TRE is the slowest tool. Common to all four tools is their decrease in runtime matches the sizes of the data files.

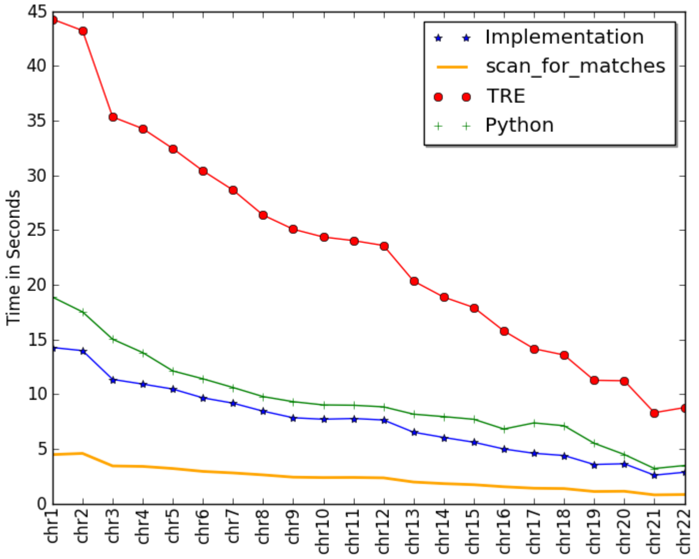


Figure 7: Running time of search through fasta files mentioned in Figure 6 looking for pattern TGCAAGCGTTAAT with no mismatches

From Figure 8 it is evident that there is an increase in the runtime for all four tools, scan_for_matches continues to be the fastest tool, while our implementation is the second fastest. Python ran at about double the speed of our implementation, and TRE continues to be the slowest tool.

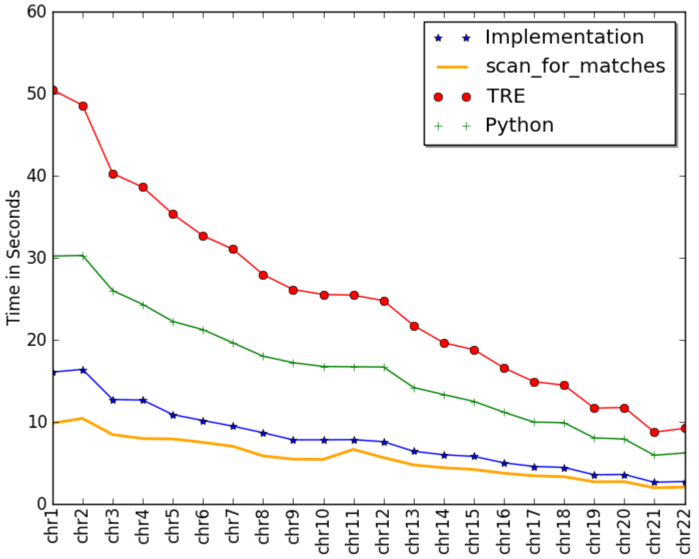


Figure 8: Running time of search through fasta files mentioned in Figure 6, allowing one insertions on pattern TGCAAGCGTTAAT

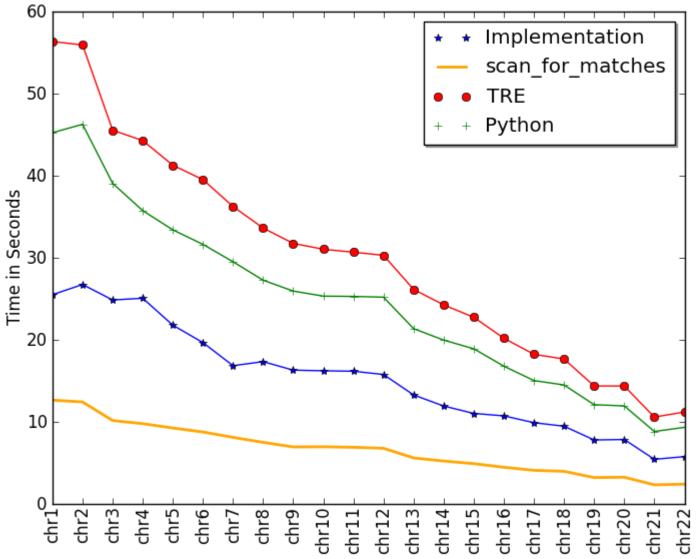


Figure 9: Running time of search through fasta files mentioned in Figure 6, allowing two insertions on pattern TGCAAGCGTTAAT

The next test was to see the runtime of two insertions instead of one. Looking at Figure 9, scan_for_matches did increase its runtime slightly compared to Figure 8, but the second insertion greatly affected our implementation, resulting in it running at about half the speed of TRE, but still faster than Python. And while TRE also had its runtime slightly increased, it's almost unchanged from one insertion.

Testing with three insertions, Figure 10 python is now the slowest tool, and our implementation is once again slower compared to TRE, which is now the second slowest tool, and scan_for_matches is still the fastest tool.

From these three tests its clear to see that our implementation has a problem with increasing

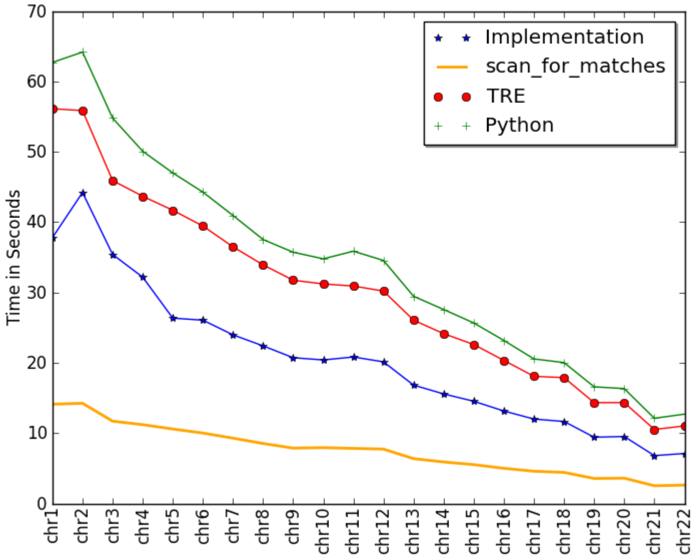


Figure 10: Running time of search through fasta files mentioned in Figure 6, allowing three insertions on pattern TGCAAGCGTTAAT

number of insertions, affecting its runtime at a much higher rate than both scan_for_matches and TRE. It does seem, however, that Python has a similar problem. From this we can conclude that our current implementation has a major flaw, should it be used with more advanced patterns.

We suspect that flaw may be the rate of which new states are created. To test this claim, we ran our code again, on fasta file chr22.fa, measuring how the number of states, and thereby the amount of work needed to pattern-match, relates to the number of mismatches allowed.

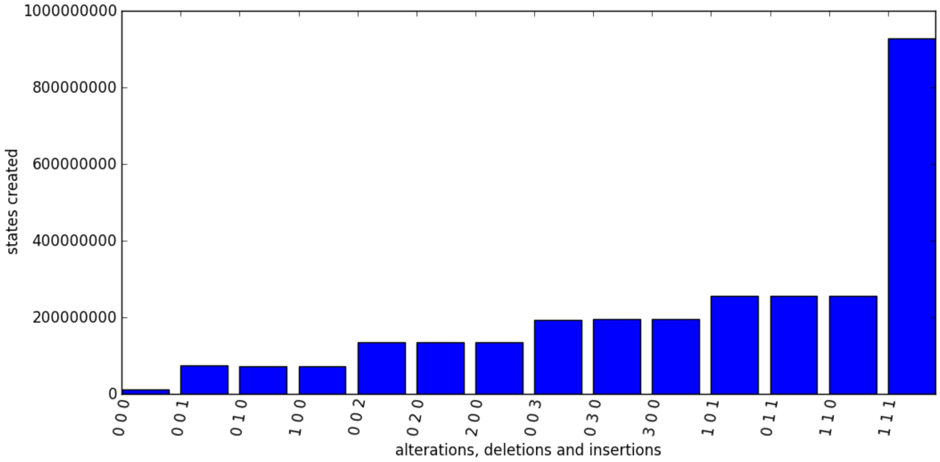


Figure 11: Bar chart showcasing the number of states created corresponding to the number of mismatches allowed.

The chart in Figure 11 displays an increasing number of states, corresponding to the increasing number of mismatches. We can observe how allowing any number of mismatches of the same type, be it insertion, alternation or deletion, yields roughly the same number of states, actually deletions and alternations gives exactly the same amount of states, as their behavior is exactly the same.

Another observation is how two mismatches of different types result in more new states than two or even three mismatches of the same kind would. This can be explained due to the implementation creating two new states every time a mismatch is encountered in the data. Causing an exponential growth of states, opposed a linear growth. Finally there is a measurement with a mismatch of every kind, causing a massive growth in states, once again being due to an exponential growth, now making three states per mismatch, instead of two or one.

Another interesting thing to test for was the number of hits when searching the files, in table 3 the number of hits which came up when searching on file chr1.fa are shown.

	1 insertion	2 insertions	3 insertions
Our implementation	5	48	235
TRE	1	19	76
scan_for_matches	5	43	192
Python	5	48	235

Table 3: Number of hits in fasta file chr1, using the mentioned benchmark tests.

The primary reason that our implementation gets more results than scan_for_matches is that our implementation finds every single match in the file, including overlapping matches, while scan_for_matches, by default, only finds matches which do not overlap. TRE has the major disadvantage here that it does not match across newlines, causing it to miss a lot of matches. Finally Python has the same amount of matches as our implementation.

11 Alternative soloutions

11.1 Forming patterns using Regular expressions

The initial goal of this project was to use regular expressions to match the sequences. The problem of only using regular expressions, is the amount of patterns explode in size when adding mismatching. Following is a description of how many patterns are formed from a pattern of length n. When a new pattern is formed, it constructs them into 1 regular expression using the alternation operator separating each of the new expressions.

Mutations are done by having a character replaced by a wildcard. This is done for every character in the pattern. When adding multiple mutations, characters which are already wildcards are not changed. The formula for the amount of patterns formed from mutations is the number of combinations that can be formed from the amount of mutations in t. This is the binomial coefficient[4].

An insertion is a wildcard added between the characters in the pattern, so for each pattern $n-1$ new patterns occur.

A deletion is removing a character from the pattern. It is not allowed to remove a character next to an insertion, as this cannot occur in RNA and DNA strings. Given multiple insertions, they will be spread out throughout the pattern in most cases, so an approximation of how many patterns formed would be $(n - insertions * 2)$.

The final formula looks like $\binom{n}{m} * (n-1) * (n - i * 2)$, where n is the length of the string, m is amount of mutations and i is amount of insertions.

Example 11.1. *Given a pattern of size 30, with 2 mutations, 1 deletion, 1 insertion. It would produce the following amount of patterns:*

[4] http://en.wikipedia.org/wiki/Binomial_coefficient

$$\textit{After mutation: } \binom{30}{2} = 435$$
$$\textit{After insertion: } 435 * (30 - 1) = 12615$$
$$\textit{After Deletion: } 12615 * (30 - 2) = 353220$$

As shown in example 11.1, the amount of patterns formed from using regular expressions could be too large for a regular expression matcher to find in a reasonable time.

12 Summary & Future Work

We presented an alternative type of NFA in Section **??**. The alternation allowed for insertions, deletions and mutations when matching a pattern. A runtime analysis showed that the new NFA type caused an increase in states which would grow as the errors were increased. Future work would consist of preventing said growth, by making sure states which does repeated work are handled as a single state. Further work would extent on the TNFA to support more SFM features, and backreferenceing.

Further more a research into preproccesing using suffix trees[5] could allow posible decrease in runtime.

[5] `http://en.wikipedia.org/wiki/Suffix_tree`