Genome pattern matching using regular expressions

Simon Nicolai Lefoli Maibom - xvm226

Arinbjörn Brandsson - hkt789

Martin Simon Haugaard - cdl966
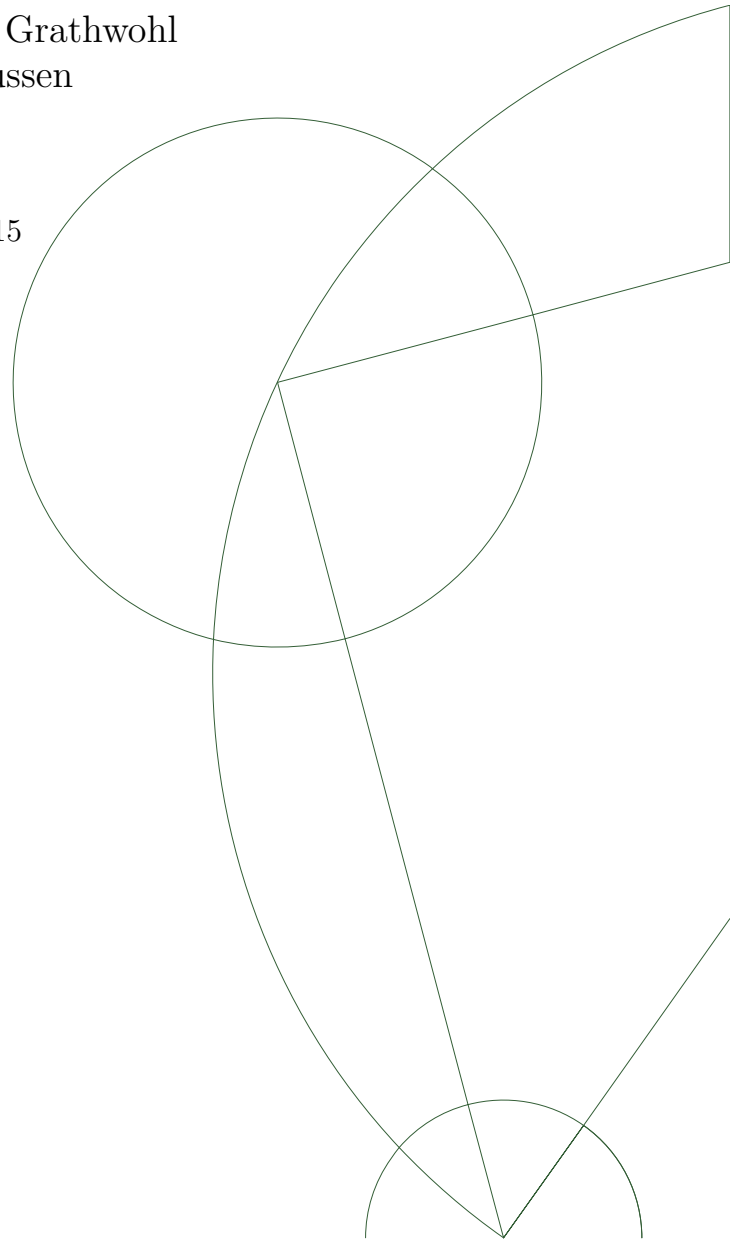
Supervisors

Rasmus Fonseca

Niels Bjørn Bugge Grathwohl

Ulrik Rasmussen

June 2, 2015

Abstract

Contents

1   Motivation

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 hasn't stopped since. This has lead to decreasing costs of sequencing RNA and DNA, meaning biologists has access to greater amounts of data than ever before. However the technology to process these amounts of data haven't progressed at the same pace as sequencing has. A tool for this is scan_for_matches, a pattern-matching tool that searches through text files to match a pattern specified by a user. While scan_for_matches has proven to be a fast and reliable tool, because of the amount of data it shifts through, a faster alternative is desired.

After hearing about this problem, we thought that there must be a better way of searching through text that is also theoretically sound. Our first thought was using automata-based searching methods, since this provides a calculable best- and worst-case run time while being theoretically sound. Since regular expressions uses an automata-based way of searching, we hypothesized that implementing regular expressions that would have the same functions as scan_for_matches would lead to faster run times.

2   Problem Analysis

The functionality of scan_for_matches dictates what our solution to the problem 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 text file,

2. Match

    (a) with errors allowed,

    (b) a previously found match,

    (c) a modified pattern, and

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

This section details the current tools readily available that closest match what our project aims to do, as well as to see if there are any existing tools we can use for our project. For this we will talk about RE2/J, Google's regular expression library because of how it handles alternations as well as its linear running time, and TRE, which allows errors in its results, can simulate backtracking, and supports backreferencing.

3.1   RE2/J

RE2/J is Google's regular expression library written for java. It does not support backtracking or backreferencing since neither can be implemented efficiently. Since RE2/J doesn't support matching with errors or backreferencing, it would be unable to properly reproduce scan_for_matches' functionality. The project can be found on its github page[1].

3.2   TRE

TRE was created by Ville Laurikari for his master's thesis in 2001, and is a regular expression engine which can simulate backtracking and supports backreferencing and matching with errors. Because of this, TRE is the best candidate for modification in order to make it work with patterns. We expand on this in section 8.

---

[1]`https://github.com/google/re2j`

4   Nucleic Acids

Nucleic acids are one of the building blocks of life, and are composed of a backbone made of a type of sugar, with bases that can bond with one another. A series of these bases on a backbone is called a sequence. The type of sugar as well as the bases depends on the nucleic acid.

4.1   Ribonucleic acid

Ribonucleic acid (RNA) is a large molecule composed of nitrogenous bases nested on a ribose-phosphate backbone. The possible nitrogenous bases, or bases for short, that can be nested on the backbone are guanine (G), adenine (A), uracil (U) and cytosine (C). In nature, the predominant form of RNA are as a singlestranded chain that can fold in on itself, 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 uses a sugar called ribose in its backbone, which is unstable compared to its other form, deoxyribose, used in deoxyribonucleic acid (DNA), but is more flexible, allowing the RNA chain to bend in ways that DNA can not.

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 guanine with cytosine, and adenine with uracil. These bonds are complementary of each other, and forms the structure of each RNA. When two bases bond with each other, they will stick to each other which changes the form of the RNA chain. However sometimes a base will have no complementary base to bond with, causing the base to stick out, giving rise to certain characteristic forms. This will be elaborated on later.

4.2   Deoxyribonucleic acid

Deoxyribonucleic acid (DNA) is a large molecule composed of nitrogenous bases nested on a deoxyribose-phosphate. DNA is mostly found in nature as helixes, where two strands has 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).

4.3   Secondary Structures of Nucleic Acids

The secondary structure of a nucleic acid describes how the bases of the nucleic acid has bonded. The secondary structure of nucleic acids can change if the nucleic acid is damaged or has mutated, causing it to gain or lose bases. When two bases bonds they hold onto each other, causing bases that have no complementary base to bond with to stick out. Below are examples of three common secondary structures as well as a brief analysis on how to identify them.

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, since 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`.

Interior Loop

An interior loop is when two or more opposing bases aren't complementary and can't 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` and `G` 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, making them hard to generalize.

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 doesn't bond with anything. This kind of loop occurs typically in RNA as they are single-stranded, but may happen in DNA if the two strands of the DNA has been separated. Figure 3 shows a stem loop.

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

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 the nucleic acid has.

5 Regular expressions

To explain what a regular expression 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:*

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

[**?** , 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`$\}$, the regular expression for natural numbers would look like:*

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

Example 5.3. *Given the alphabet $\Sigma = \{$`a`,`b`,`c`$\}$. With this alphabet, one could form a regular expression that could look like $E = $`c`$(\texttt{a}+\texttt{b})^*$. The language L that is produced from this expression, would consist of strings that is a c followed by 0 or more a or bs for example* `"caaaa"`,`"cbb"`,`"cababba"` *and so forth. The language interpertation of E is*

$$L(E) = L(c(a+b)^*)$$

$$= \{w_1\ w_2 | w_1 \in L(c), w_2 \in L((a+b)*)\}$$

$$= \{w_1\ w_2 | w_1 \in c, w_2 \in \bigcup_{n=0}^{\infty} L(a+b)^n\}$$

$$= \{w_1\ w_2 | w_1 \in c, w_2 \in \bigcup_{n=0}^{\infty} (L(a) \cup L(b))^n\}$$

$$= \{w_1\ w_2 | w_1 \in c, w_2 \in \bigcup_{n=0}^{\infty} (a \cup L(b))^n\}$$

$$= \{w_1\ w_2 | w_1 \in c, w_2 \in \bigcup_{n=0}^{\infty} (a \cup b)^n\}$$

$$= c \bigcup_{n=0}^{\infty} (a \cup b)^n$$

6 Nondeterministic Finite Automaton

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 Δ that is a set of 3 tuples containing all the transitions in Q, with the type of $\Delta \subseteq Q \ x \ (\Sigma \cup \{\epsilon\}) \ x \ Q$.*

6.1 Conversion from RE to NFA

Each expression can be converted to an NFA, and vice versa. Table 1 shows how each RE can be converted into a NFA. The states in Q is 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 is 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.
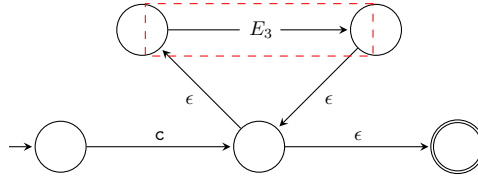
Table 1: Translation table from regular expressions to NFA
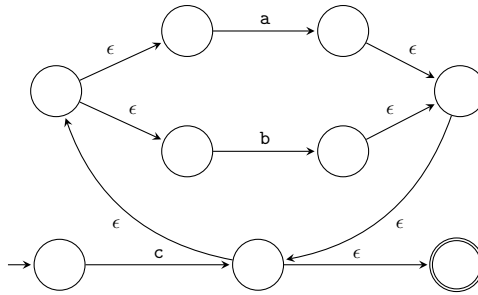

Example 6.1. *In example **??**, the expression $c(a + b)^*$ is used. The NFA buildup is done like the language interpetation in the example. We first construct the $E_1 E_2$ expression, where $E_1 = c$ and $E_2 = (a + b)^*$.*
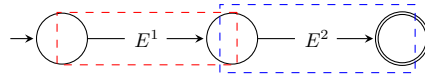
Now we construct the subexpressions, E_1 and E_2. E_2 results in a new subexpression $E_3 = a + b$.

The last part is constructing E_3, we skip ahead and construct the literals `a` and `b` as well.

6.2 Matching using NFA

Definition 6. *Given a set of NFA states M. The $\epsilon-closure$ of M, is a set of states that are reachable by following the ϵ-transitions from*

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

Algorithm 1 NFA simulation

Require: N is a NFA and x is a string

1:
2: **function** SIMULATION($N(Q, \Sigma, \Delta, q^s, q^a), x$)
3:      $stateset \leftarrow \{q^s\}$
4:      **for each** symbol **in** x **do**
5:          **if** $stateset = \emptyset$ **then**
6:              **return** False
7:          $next \leftarrow \emptyset$
8:          $states \leftarrow \epsilon\text{-closure}(stateset)$
9:          $stateset \leftarrow next$
10:      **if** $q^a \in stateset$ **then**
11:          **return** True
12:      **return** False

6.3 Tagged NFA

Before introducing tagged NFA, we must define what a missmatch is.

Definition 7. *A missmatch M in a string n with a alphabet Σ is one of 3 types, insertion, deletion and alteration. An insertion is where a symbol in Σ is added to n. A deletion is where a symbol in n is removed. A alteration is where a symbol in n is changed with a symbol in Σ.*

NFA's do not handle missmatching in strings per default, to introduce this, a new set of transitions is added to the NFA. This type of NFA is called a Tagged NFA.

Definition 8. *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 $\Delta' \subseteq Q \ x \ \{\epsilon\} \ x \ Q \ x \ M$.*

6.4 Constructing TNFA

TNFAs are constructed the same way as shown in tabel 1, with the exception when constructing literals. A new set of ϵ-transitions is added as shown in tabel 2 as 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

13
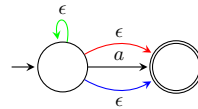
7 Scan_For_Matches

Scan_for_matches is a string-searching tool created by Ross Overbeek, David Joerg and Morgan Price in C which searches through text files. 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 9. *Let Σ denote an alphabet. Then we can define a pattern unit as follows:*

`h`	*Match the sequence h, where $h \in \Sigma*$*
`n...m`	*Match n to m characters where $n \leq m$*
`x=n...m`	*Match n to m characters, and call 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 insertions where $n,m,l \geq 0$*
`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, $v \in \Sigma$, 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*
`^ 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 10. *Let Λ be any pattern unit in definition 9. Let $E \in \Lambda$. Let 0 be the empty string. Let A be a pattern that we are processing. A pattern may then be constructed as such:*

$$A = A' \; A \mid 0$$
$$A' = E$$

Definition 10 states that a pattern may be any combination of the pattern units defined in definition 9. Using these pattern units, it is possible to make very specific or very broad searches in a text file.

Definition 11. *The language interpretation of definition 9 is defined as follows:*

$$L(0) = \{\epsilon\}$$
$$L(\texttt{a}) = \{\texttt{a}\}$$
$$L(\texttt{n...n}) = \underbrace{L(\texttt{a})L(\texttt{a})...L(\texttt{a})}_{n}$$
$$L(\texttt{n...m}) = L(\texttt{n...n}) \cup L(\texttt{n+1...n+1}) \cup ... \cup L(\texttt{m-1...m-1}) \cup L(\texttt{m...m}) = \bigcup_{n=\texttt{n}}^{\texttt{m}} L(n...n)$$
$$L(\texttt{E}_1 \; \texttt{E}_2) = L(\texttt{E}_1) \; L(\texttt{E}_2)$$
$$L(\texttt{E}_1 \mid \texttt{E}_2) = L(\texttt{E}_1) \cup L(\texttt{E}_2)$$
$$L(\sim\texttt{E}) = \sim L(\texttt{E})$$
$$L(< \texttt{E}) = \; < L(\texttt{E})$$
$$L(\texttt{length}(\texttt{E}_1) + \texttt{E}_2) = \texttt{length}(L(\texttt{E}_1)+L(\texttt{E}_2))$$

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 will match 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 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 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 analysing TRE to see how it worked 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 because 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 has lead to the following design choices;

- the program runs through the data once to deetermine 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's no way for TRE to identify and output acceptable matches.

This coupled with little documentation of how the code works meant that it would take a long time to properly analyse 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 tagged NFA using the methods description in section 6.1 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 NFA's from regular expressions, for example the regular expression "(GAT)+" would produce a structure as shown in figure 4
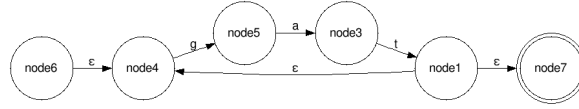

Figure 4: Example of how our implementation builds NFA got regular expression (GAT)+

Each node in the figure has a number corresponding to the time it was created in the code, examining figure 4 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 6.1.

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 4, 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 cannot 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

Using our implementation, TRE and scan_for_matches, we were able to do some benchmarking in order to compare performances.

For this a virtual machine is created, using Oracle VirtualBox[2]. The machine running the virtualbox is running Windows 8.1 Pro x64 of 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.

For the benchmarking some data was required, figure 5 shows a series of fasta files, these files include nucleotide sequences, which all differ in size, decreasing from chr1 to chr22. These fasta files are the kind of data which scan_for_matches are expected to run, and thus excellent for benchmark testing, it's worth noting however, that each of these files are 50 characters long, as TRE matches searches through lines separately, and longer or shorter line sizes might affect the performance of TRE, this was not tested.
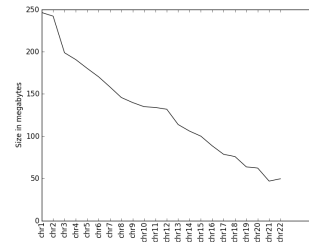

Figure 5: Plot of sample files used for benchmarking, these files range from 246.3 Megabytes to 46.8 Megabytes in size

---

[2]virtualbox.org

For testing, a suitable pattern is required. As
our implementation has primarily been focused on supporting insertion, deletion and mutation on
a sequence, a simple RNA sequence $TGCAAGCGTTAAT$ with variable insertions is chosen as the
search pattern.

Each test would be executed on each of the mentioned fasta files a total of 10 times, given an
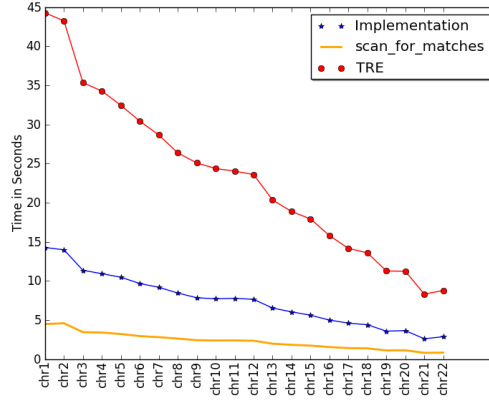average runtime which was used in the following results.

Figure 6: Running time of search through fasta files mentioned in figure 5 looking for pattern TG-
CAAGCGTTAAT with no missmatches

First test was to see the runtime of each program, having no missmatches in the mentioned se-
quence RNA $TGCAAGCGTTAAT$, figure 6 displays the results, and its clear to see that scan_for_matches
is faster than both our implementation and TRE, and that all three programs have a decreased run-
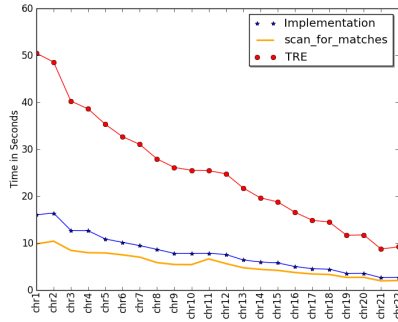time as the data sizes lowers.

Figure 7: Running time of search through fasta
files mentioned in figure 5, allowing one insertions
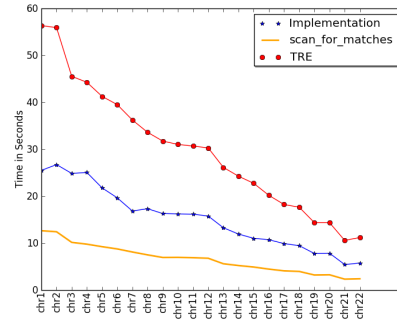on pattern TGCAAGCGTTAAT

Figure 8: Running time of search through fasta
files mentioned in figure 5, allowing two insertions
on pattern TGCAAGCGTTAAT

From figure 7 its visible that there's an increase in the runtime for scan_for_matches when adding
an insertion to its pattern, and while both our implementation and TRE also has an increased runtime,
it's not to the same scale as scan_for_matches.

Next test was to see the runtime of two insertions instead of one. Looking at figure 8 scan_for_matches
did increase its runtime slightly compared to figure 7, but the second insertion really affected our
implementation, resulting in it running at about half the same speed of TRE. And while TRE also
had its runtime slightly increased, it's almost unchanged from one insertion.

Testing with three insertions, figure 9 shows that once again our implementation had the worst
increase in time compared to the two other options, almost reaching the same runtime as TRE.
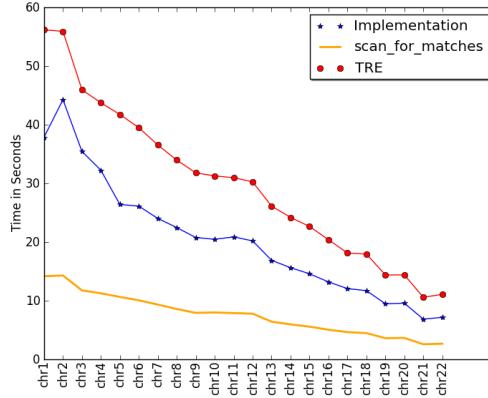
18

Figure 9: Running time of search through fasta files mentioned in figure 5, allowing three insertions on pattern TGCAAGCGTTAAT

From these three tests its clear to see that our implementation has a problem with increasing number of insertions. Increasing its runtime at a much higher rate than both scan_for_matches and TRE, from this we can conclude that our current implementation has a major flaw somewhere, should it be used with more advanced patterns.

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	47	235
TRE	1	19	76
scan_for_matches	5	43	192

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

The primary reason for our implementation to get 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 does not overlap. TRE has the major disadvantage here that it doesn't match across newlines, causing it to miss a lot of matches.

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 missmatching. Below 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 seperating each of the new expressions.

Mutations are done by having a character is replaced by a wildcard, this is done for every character in the pattern. When adding multiple mutations, characters which is 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[3].

---

[3]`http://en.wikipedia.org/wiki/Binomial_coefficient`

A 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 insertion, they will be spread out thoughtout the pattern in most cases, so an appoximation of how many patterns formed would be (n - insertions * 2).

So the final formula would look something 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:*

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

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

11.2 Preprocessing data

Preproccesing data gives certain advantages, it allows for faster lookups into the string that is being searched on. With a structure like suffix tree's[4], to store the location of all the substrings. It would be posible to do lookups in constant time given the correct choice of data structures. This would allow using the large regular expressions with missmatches to be run in a reasonable time. The disadvantage of preprocessing data, is the time it takes to construct a indexed structure of the string, and it may produce a tree larger than the original string, which could be an issue in some of the larger data files which are several GB in size. If the file have to be reused many times, it may be justified to create one.

---

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