FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

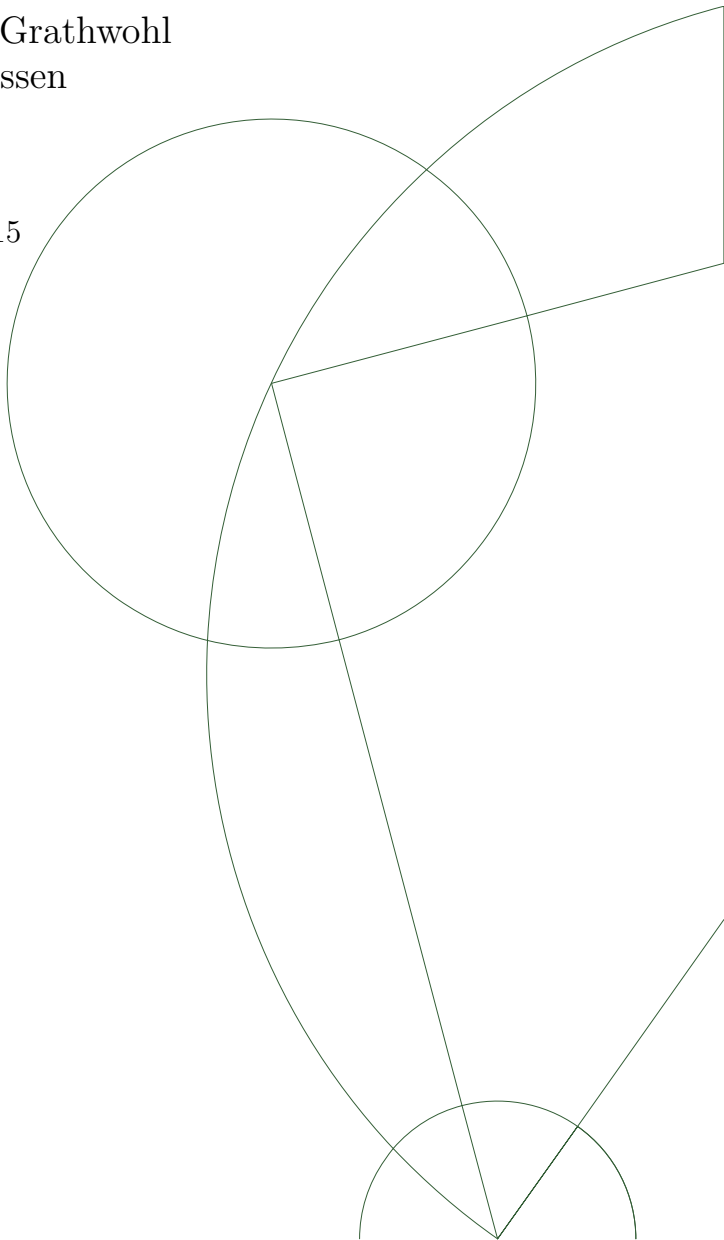
Genome pattern matching using regular expressions

Simon Nicolai Lefoli Maibom - xvm226
Arinbjörn Brandsson - hkt789
Martin Simon Haugaard - cdl966

Supervisors

Rasmus Fonseca
Niels Bjørn Bugge Grathwohl
Ulrik Rasmussen

May 20, 2015



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. One of these tools 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 hypothesize that implementing regular expressions that would have the same functions as scan_for_matches would lead to faster run times, benefiting scientists that uses scan_for_matches.

2 Nucleic Acids

((TODO: TELL WHY NUCLIEC ACIDS))

2.1 Ribonucleic acid

((TODO: ADD FIGURES, TALK ABOUT DIFFERENT KINDS OF RNA))

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.

2.2 Deoxyribonucleic acid

((TODO: ADD FIGURES, TELL MORE ABOUT DNA))

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).

2.3 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. Below is an example of a bulge:

Example 1. *The RNA sequence* `AGCAGGCUAGCCGCU`*. Note the bulging* `A`*.*

Since we know that a bulge occurs when a base is inserted or deleted, we can define a bulge as follows:

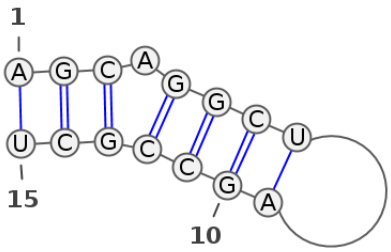


Figure 1: Example of a bulging A

Definition 1. *Let S be a bulge. Let E be a sequence of characters. Let Σ be an alphabet. Let $a \in \Sigma$. Let 0 be the empty string. Let Λ be a map that holds the complement to each base described in the alphabet. Let E^{-1} denote the reverse of the sequence. Then we can define a stem loop as follows;*

$$\begin{aligned} S &= E' \ f(E^{-1}) \mid E \ f(E^{-1\prime}) \\ E &= \{a\} \ E \mid 0 \\ E' &= E \ E' \mid E \\ f(E) &= \Lambda(E) \end{aligned}$$

((NOTE: ABOVE DEFINITION NEEDS WORK, ISN'T CORRECT))

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. Below is an example of an interior loop:

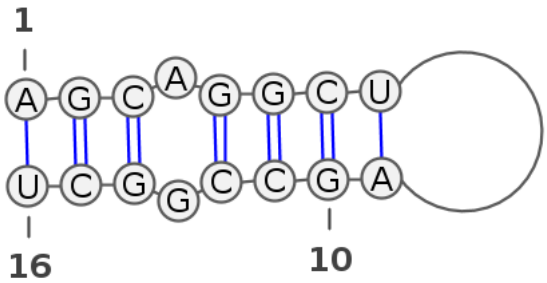


Figure 2: Example of an Interior Loop A-G

Example 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.

((TODO: ADD A BETTER EXPLANATION AND A DEFINITION OF AN INTERIOR LOOP))

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. Below is an example of a stem loop:

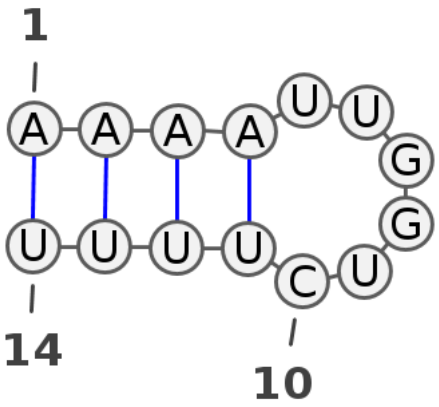


Figure 3: Example of a Stem Loop, UUGGUC

Example 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`. This specific kind of loop adheres to a specific pattern that reflects from the example, that can be defined as follows:

Definition 2. *Let S be a stem loop. Let E_n be a sequence of characters, where if in E_n and E_m $m = n$, then $E_n = E_m$. Let Σ be an alphabet. Let $a \in \Sigma$. Let Λ be a map that holds the complement to each base described in the alphabet. Let E_n^{-1} denote the reverse of the sequence. Then we can define a stem loop as follows;*

$$S = E_0 E_1 f(E_0^{-1})$$
$$E_n = \{a\}$$
$$f(E_n) = \Lambda(E_n)$$

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.

3 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 wish to search for by defining a pattern, and scan_for_matches returns all matches that corresponds to the specified pattern.

Definition 3. *Let* `E` *be any pattern that's in the alphabet Σ as defined in Example 4. Let 0 be the empty string. Let* `A` *be a string that we are processing to see if the pattern is valid. A pattern may then be constructed as such:*

Example 4. *Let Σ denote an alphabet. Then we can define a pattern 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 x or y*
`x[n,m,l]`	*Match 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={hl, lh}`	*Create a pattern rule where h is the complement of l, and l is the complement of h, where $h \in \Sigma$, $l \in \Sigma$, and call it 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 3 states that a pattern may be any combination of the alphabet defined in example 4. Using these patterns, it is possible to make very specific or very broad searches in a text file.

Example 5. *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 5 will match a stem loop as described in section 2.3. 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`.

4 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 4. *An alphabet Σ is a finite set of letters.*

Definition 5. *A language is a subset of all strings, formed over Σ : $L \subseteq \Sigma^*$*

Example 6. *If we have a DNA sequence string, the alphabet Σ consists of the literals $\{\mathtt{t}, \mathtt{g}, \mathtt{c}, \mathtt{a}\}$. and the language contains strings formed by the literals from this alphabet. For example "gtcaaa" or "gtcaaat".*

Definition 6. *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 7. *The language interpation L(E) of a regular expression is:*

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

[? , p.5 def. 3]

With definition 7, we can now form regular languages.

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

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

5 Nondeterministic Finite Automaton

((NOTE: SHARPEN UP SECTION TO A MORE PRECISE DEFINITION OF NFA))

Definition 8. *An nondeterministic finite automaton (NFA) consists of a finite set of states, Q, with a finite number of transitions, T, between them.*

In Q there's one starting state, and a subset of accepting (final) states.

Each transition in T is a has a starting state, and a destination state, and is labeled either by a character, or by ϵ, which indicates an epsilon-transition (empty transition).

5.1 Conversion from RE to NFA

One of the main reasons for using NFAs when working with regular expressions is the direct correlation between regular expressions and NFAs. Each expression can be converted to an NFA, and vice versa.
((NOTE: We'll include a table with conversions))

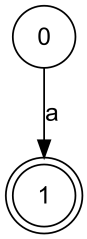


Figure 4: NFA of the expression a

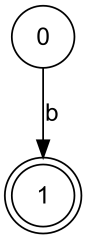


Figure 5: NFA of the expression b

For example two NFAs of regular expression a and b will appear as shown in Figure 4 & 5, each of them starts in node 0 and ends in node 1, and these nodes are connected by a one-way transition with a value a or b.

To construct the NFA for $A|B$, one will first have to construct an NFA for both A and B, which will then be combined, making the full NFA. The | operator this is achieved by constructing two new nodes, the first having epsilon-transition pointing to the start node of each of the two NFAs A and B, then for each NFA A and B the ending node will instead of ending the NFA, have a new epsilon-transition to the second new node, in the figure below, Figure 6, the two new nodes are labeled 0 and 5:

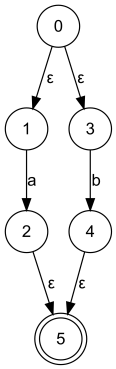


Figure 6: NFA of the expression $a|b$

Once a NFA-structure has been build, we can start matching a text against the NFA.

This is achieved by having a set of states, each state is looking at a node in the structure, while keeping track of previous nodes traversed by the state. Whenever a new character is being checked to match, each state will look at its node, and determine if it's possible to accept the character in the NFA, either by taking a transition labeled with the character, or alternatively via an epsilon-transition. When two possible transitions are viable, new states will be created in the set of states, so that each possible transition will be explored.

Any state that will fail to match the character will be removed from the set of states.

Only if a state reaches the ending node of the NFA, there's a match.

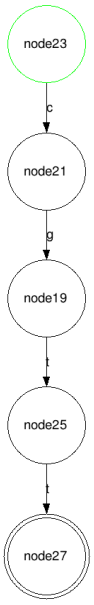


Figure 7: NFA of $CGTT$, with a state looking at node23.

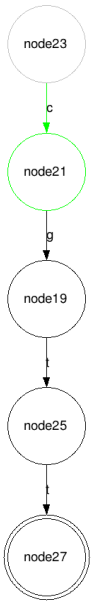


Figure 8: Next iteration of the state on Figure 7, the state is now in node21.

5.2 Insertions, Deletions and Mutations

NFAs do not support mismatching by default, although a regular expression can be built to handle mismatches. While doing handling mismatching while constructing the regular expression, we are interested in having an NFA, and allowing matching while with support for insertions, deletions and mutations, hence we need to define a way to handle this.

We can do this by adding a counter for insertions, deletions and mutations, and when a transition is not possible in the NFA, we decrease these counters and preform alternative transitions to emulate these conditions.

For insertions, the state will remember the unmatched character, but the state won't move from its current node.

For deletions, the state wont remember the unmatched character, but it will take every transition going on from the node.

For mutations, the same happens as in a deletion, but now the state also remembers the unmatched character.

Figure 9 illustrates how states move through a NFA when mismatches occur.

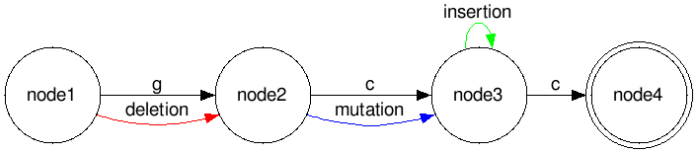


Figure 9: Simple NFA of GCC, showing behaviour of insertion, mutation and deletion

6 TRE

((TODO - UNCHANGED FROM LAST TIME))

For our implementation of scan_for_matches patterns in TRE, we needed to first make a brief analysis of the program to see what it could and couldn't do. What we discovered after our initial tests was that TRE defined each new line as a delimiter (a delimiter defines at which points the text will be separated, so each chunk can be evaluated individually). This caused a small, but surmountable, problem; chromosome data should be read as one continous line that spans over many lines. Since TRE split the data up per line, if a match spanned over more than one line, it wouldn't find it. To this problem there was an easy fix; we would make a small wrapper which would feed the text file to TRE, removing any newlines as they occurred.

With this problem solved, we did a trial test to check the accuracy and running time of TRE. Here we encountered a second, greater problem; TRE would find only one match per delimiter (the first longest match with fewest errors, prioritizing exact matches). This was a harder problem to fix, because this feature was deeply entrenched in the code, and everything had seemingly been designed around this feature. The searching tool that TRE uses, agrep, only receives the best match that the other tools of TRE has found, but throughout TRE's code, it won't save the matches it has found, but instead repeatedly discard old matches. The problem is compounded by the fact that TRE will immediately stop searching the text if it finds an exact match, leaving behind potential matches. In light of these problems, we decided that attempting to fully understand TRE's design and subsequently heavily modding it in order for it to return all matches would take too long, and that we would instead create our own solution.

7 Our Implementation

((TODO)) We constructed a simple program, which would take a regular expression and using the method description outlined in section 5.1 in order to match a file for a desired RNA section.

8 Experimental Results & Tests

((NOTE : Extensive testing will be included)) Using our implementation, TRE and scan_for_matches, we were able to do some benchmarking in order to compare performances. ((TODO: BETTER BENCHMARKING INCOMING)) Searching for pattern $GGAGTGCAAGCGTT$ with 1 insertion, 11 times each on a 40MB, 80MB and a 194MB file, we got the following average running times (in seconds):

	scan_for_matches	TRE	Our Implementation
40MB	2.2	13.6	22.0
80MB	4.4	27.8	45.2
194MB	12.2	71.3	114.6

References