Genome pattern matching using regular expressions

Simon Nicolai Lefoli Maibom - xvm226

Arinbjörn Brandsson - hkt789

Martin Simon Haugaard - cdl966
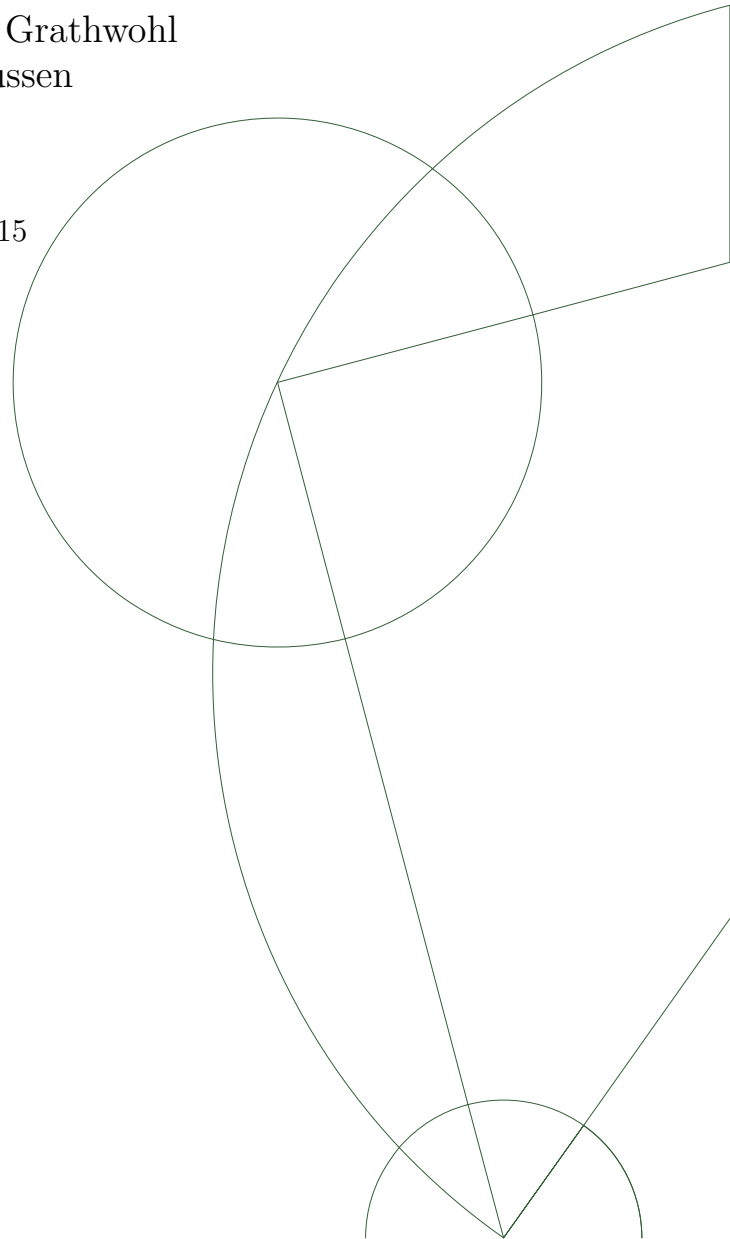
Supervisors

Rasmus Fonseca

Niels Bjørn Bugge Grathwohl

Ulrik Rasmussen

May 27, 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. 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   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

3.1   RE2

RE2 is Google's regular expression engine written in C++. It supports backtracking, but not backreferencing since backreferencing can't be implemented efficiently according to [**?**]. Since RE2 doesn't support matching with errors or backreferencing, it would be unable to properly reproduce scan_for_matches' functionality.

3.2 TRE

TRE was created by Ville Laurikari for his master's thesis in 2001, and is a regular expression engine supporting backtracking, 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.

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. Below is an example of a bulge:
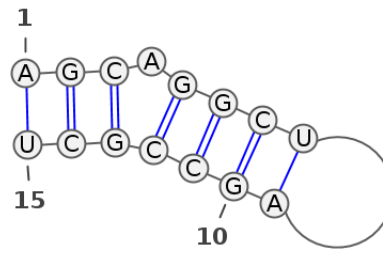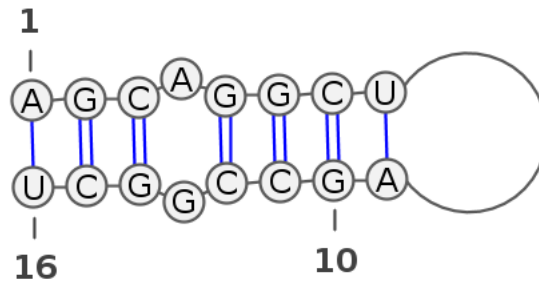
Example 1.

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. Below is an example of an interior loop:

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.

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:
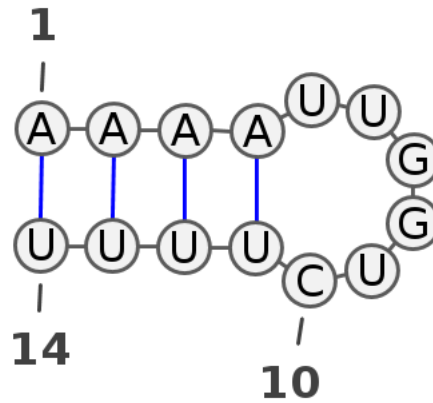
Example 3.

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. 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 4. *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:*

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

[1, p.5 def. 3]

With definition 4, we can now form regular languages.

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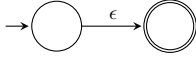
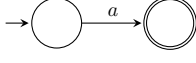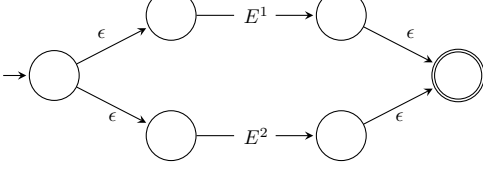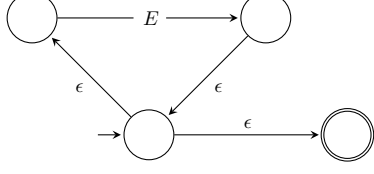$$E_{nat} = (1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*$$

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 is the set that contains all the transitions. Transitions in Δ is shown $q^1 \xrightarrow{a} q^2$ where $q^1 \in Q, q^2 \in Q$ and the label a is either $a \in \Sigma$ or the empty transition ϵ*

6.1 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. Table 1 shows the correlation between regular expressions and NFA for the most common expressions.

Table 1: Translation table from regular expressions to NFA

Example 6. *Given the expression*

6.2 Matching using NFA

Given a NFA and a input string, to see if the input string is accepted in the NFA the following algorithm is used.

//n is a NFA

```
//m is the input string
Simulate(n(Q, Σ, Δ, q^s, q^a), m)
    stateSet := (q^s)
    inputLen := strlen(m)
    for i from 0 to inputLen−1
      //No valid transitions from previous input symbol
        if size(stateSet) == 0
            return false
        symbol := m[i]
        nextStateSet := ()
        for each state in stateSet
            states := StateTransitions(Q, Δ, symbol)
            nextStateSet := union(nextStateSet, states)
        stateSet = nextStateSet
    if q^a in stateSet
        return true
    return false
```

6.3 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 1 illustrates how states move through a NFA when mismatches occur.
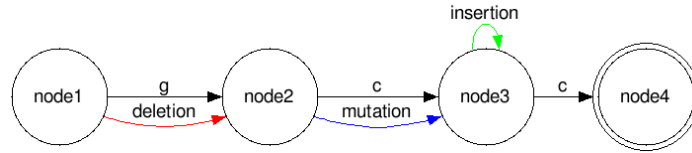
Figure 1: Simple NFA of GCC, showing behaviour of insertion, mutation and deletion

7 Scan_For_Matches

((TODO: UNCHANGED))
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 6. *Let E be any pattern that's in the alphabet Σ as defined in definition 7. 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:*

Definition 7. *Let Σ denote an alphabet. Then we can define a pattern as follows:*

h	*Match the sequence h, where h ∈ Σ*
n...m	*Match n to m characters where n ≤ 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 ≥ 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 ∈ Σ, l ∈ Σ, 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 6 states that a pattern may be any combination of the alphabet defined in definition 7. Using these patterns, it is possible to make very specific or very broad searches in a text file.

Example 7. *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*

$$\text{p1=GUUC[1,0,0] 3...5 ~p1}$$

Example 7 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

((TODO)) We constructed a simple program, which would take a regular expression and using the method description outlined in section 6.1 in order to match a file for a desired RNA section.

10 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

11   Results

chr1.fa	chr2.fa	chr3.fa	chr4.fa	chr5.fa	chr6.fa	chr7.fa	chr8.fa	chr9.fa	chr10.fa	chr11.fa
246.284	242.003	198.723	190.526	180.152	170.233	158.202	145.704	139.726	134.846	133.928

chr12.fa	chr13.fa	chr14.fa	chr15.fa	chr16.fa	chr17.fa	chr18.fa	chr19.fa	chr20.fa	chr21.fa	chr22.fa
131.833	113.698	105.954	99.947	88.481	78.468	75.820	63.563	62.193	46.761	49.498

Table 2: Fasta files from http://hgdownload.cse.ucsc.edu/goldenPath/hg18/chromosomes/ with sizes in KB

Looking at Table 2, we have a series of fasta files, ranging from 246.284KB to 46.761 KB in size, each fasta file has a number, and besides $chr22.fa$, every number is decreasingly lower in size as to the previous.
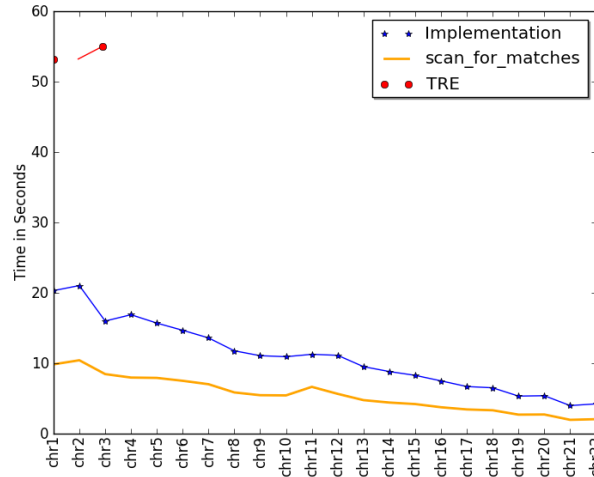

Figure 2: Running time of search through fasta files mentioned in Table 2, allowing one insertions on pattern TGCAAGCGTTAAT

12   Alternative soloutions

12.1   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.
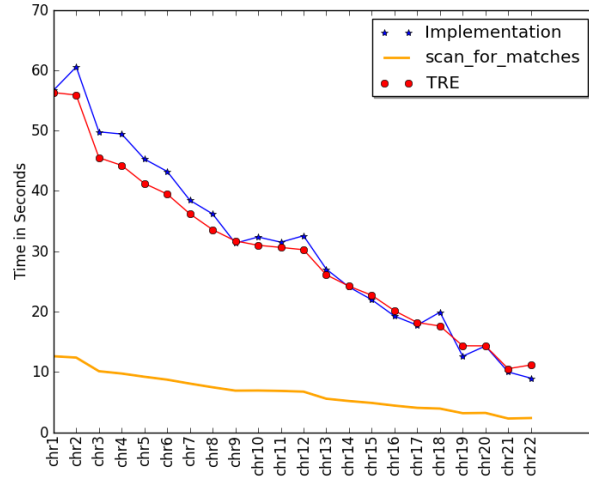
Figure 3: Running time of search through fasta files mentioned in Table 2, allowing two insertions on pattern TGCAAGCGTTAAT

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[1].

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

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

As shown in example 8, 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.

12.2 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[2], 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,

---

[1] http://en.wikipedia.org/wiki/Binomial_coefficient
[2] http://en.wikipedia.org/wiki/Suffix_tree

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.

References

[1] Niels Bjørn Bugge Grathwohl& Ulrik Terp Rasmussen Fritz Henlein. A crash-course in regular expression parsing ans regular expressions as types.

[2] Kolman, Busby, and Ross. *Discrete Mathematical Structures, sixth edition.* Pearson, 2009.

[3] Gunnar Forst. Noter til kombinatorik og grafteori. 2, 2006.

[4] William Cherowitzo. Graph theory lecture notes 6. `http://www-math.ucdenver.edu/~wcherowi/courses/m4408/gtln6.htm`, January 2015.

[5] Wikipedia. Chromatic polynomial. `http://en.wikipedia.org/wiki/Chromatic_polynomial`, January 2015.

[6] Wikipedia. Ribonucleic acid. `http://en.wikipedia.org/wiki/RNA`, May 2015.

[7] Wikipedia. Deoxyribonucleic acid. `http://en.wikipedia.org/wiki/DNA`, May 2015.

[8] Wikipedia. Nucleic acid secondary structure. `http://en.wikipedia.org/wiki/Nucleic_acid_secondary_structure`, May 2015.

[9] Ville Laurikari. Efficient submatch addressing for regular expressions. November 2001.

[10] Torben Ægidius Mogensen. *An Introduction to Compiler Design.* Springer, 2001.

[11] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press.

[12] Ville Laurikar. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. 2000.

[13] R. A. Baeza-Yates & G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *ACM, vol. 43*, 1996.

[14] Pang Ko & Sromovas Aluru. Suffix tree applications in computational biology.

[15] Wikipedia. Levenshtein automaton. `http://en.wikipedia.org/wiki/Levenshtein_automaton`, April 2015.

[16] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). January 2007.

[17] Ville Laurikari. Tre documentation. `http://laurikari.net/tre/documentation`.

[18] Mohammadreza Ghodsi. Approximate string matching using backtracking over suffix arrays. 2009.