University of Zagreb

Faculty of Electrical Engineering and Computing

Master Thesis Assignment No. 1004

Global sequence alignment tool

Goran Žužić

Advisor: *Prof. Mile Šikić*

February 5, 2015

Acknowledgements

Before anything else I would like to thank my thesis advisor Mile Šikić for his exceptional communication skills, positive attitude, enthusiasm and patience he has shown me. Furthermore, I would like to thank Filip Pavetić for numerous fruitful discussions that enabled me to refine my ideas.

Parts of the source code for this thesis were developed in collaboration with Filip Pavetić for previous projects and under the mentorship of Mile Šikić. Some of the material in this thesis has been taken from a paper authored by Filip Pavetić, Mile Šikić and me (see [1]).

Contents

1 Introduction **3**

2 Biological Primer **4**
2.1 The discovery and structure of a cell 4
2.2 Early genomics 4
2.3 The position of genes within a cell 5
2.4 From chromosomes to proteins 5
2.5 Proteins and how they are made 8
2.6 Analyzing a DNA 8
2.7 Mutation and species diversity 9

3 Algorithmic background **11**
3.1 Suffix arrays 11
3.2 Fenwick trees 12
3.3 Longest common subsequence (LCS) 14
3.4 LCSk++ 15
3.4.1 An efficient algorithm for the $LCSk++$ computation . . 16
3.5 Smith-Waterman algorithm 18

4 A new global sequence alignment tool **19**
4.1 Problem statement 19
4.2 Alignment algorithm 20
4.3 Sparse Smith-Waterman approximation 20
4.4 Theoretical background 22

5 Results **24**

6 Conclusion **26**

1 Introduction

Proteins, one of the most important molecules that make life possible, have been studied from the beginning of the 19th century. However, only in the 1950s has there been a method that could reliably determine the structure of such molecules[1]. From that point onward it was discovered that similar proteins (in terms of their structure) share a surprisingly similar function. Around that time, scientists found out that the proteins were intricately tied together with a formerly mysterious molecule in the body called the *deoxyribonucleic acid* (DNA). The idea was that uncovering the structure of the DNA could not only unlock secrets of the proteins but also point to hidden similarities across otherwise unrelated species.

The idea turned out to be a exceptionally fruitful one. Careful analysis of the structure of the DNA revealed a 99% similarity between all mammals, thereby pointing to the conclusion that most of the life on Earth had a common ancestor. This started the field of comparative genomics, uncovering evolutionary trees revealing even more striking similarities between seemingly unrelated organisms.

The main motivation behind this thesis is the development of a tool that will aid in the task of comparing the DNA of two organisms, namely the problem of "global DNA sequence alignment". But before I formally state the problem the tool is trying to solve, it is necessary to formalize our notation. Firstly, as the DNA is a long chain of four different nucleic acids, one can mathematically think about the DNA as a string over the alphabet of size four, namely $\{A, C, G, T\}$[2]. The problem of global sequence alignment can be introduced as follows: given two strings A and B over an alphabet $\{A, C, G, T\}$, find a mapping between them that will map regions of one DNA string into their homologous[3] regions on the other DNA.

The precise mathematical definition of this will be stated after some fundamental concepts have been introduced. The next section contains a biological primer that will serve as a brief introduction to the biological underpinnings of the problem. Further on, an introduction to the algorithmic and mathematical concepts will be presented. Finally, the tool and its methodology will be described, as well as the results.

[1] this was the famous Sanger's method

[2] the characters correspond to individual nucleic acids: adenine, cytosine, guanine and thymine

[3] homologous, as in related by function or lineage

2 Biological Primer

In order to understand the material presented in this thesis, one should have a modest understanding of fundamental biological concepts like DNA, RNA and proteins. I'll present a brief history of genetics as well as an overview of biological concepts.

Even though the field of bioinformatics is vast, the minimum biological background for almost any bioinformatics book can be fitted just in several pages. The following historical overview is mostly based on [2].

2.1 The discovery and structure of a cell

Initially, biology was primarily concerned with botany and zoology. Animal anatomy, medicine, pharmacology and basic taxonomy were the main topics that were studied. This, however, changed in 1665 when microscopy pioneer and public animal dissection performer Robert Hooke discovered that organisms are composed of individual parts that were named cells. This discovery changed turned biology into a science beyond the reach of the naked eye. To quote [2], "In many ways, the study of life became the study of cells".

A normal animal cell has a variety of mechanisms that control its behavior: when to replicate, synthesize a compound or even die that are called *pathways*. They are complex networks of chemical reactions that fit together to form a remarkably reliable and complex algorithm that controls the life of a cell and is "still beyond our comprehension".

Despite its complexity, the cell can be adequately analyzed by noting its three major components that are common to all living organisms: the DNA, RNA and the proteins. The DNA acts as a library, storing all the information required to replicate a new cell from scratch; the RNA is used as a mean to transfer information and materials around the cell; while the proteins do the actual work in the cell - they are the compounds that build the cell, they make chemical reactions happen and are otherwise the main working force of the cell.

2.2 Early genomics

Early scientists were often intrigued by the hereditary properties of living beings. Even ancient Jews had a law that states if a mother had lost her two male siblings due to circumcision, her son would not be circumcised until a later age, as there was a massive risk of hemophilia.[3]

In the 19th century an Augustinian monk Gregor Mendel conceived an experiment that would earn him the title of "father of modern genomics". He planted a plethora of pea plants in the monastery's garden and observed a lot of hereditary properties. In the most notable instance, he looked at the seed color of the pea plant. He noted that when a yellow pea and a green pea breed together, the offspring were always yellow. However, in the next generation of plants, the green peas reappeared in a ratio close to $1 : 3$. To explain this phenomenon Mendel postulated that each plant had two distinct objects of hereditary material that coded its color. He called this objects genes. He argued that the yellow seed color was "dominant" to the green color (which was "recessive"), but the recessive gene was still present in the organism.[4]

2.3 The position of genes within a cell

As was hinted in the introduction, the hereditary material of a living organism is stored in the DNA within a cell. Ironically, scientists long ignored the DNA as it was a long linear molecule comprised of only four simple chemical compounds: nitrogenous bases adenine (A), guanine (G), cytosine (C) and thymine (T). They were under the impression that a long repetitive molecule couldn't hold the secrets of life and they looked for genetic material in the proteins that actually manifested the observable behavior.

It was only in the first part of the 20th century that Oswald Avery proved that one should look to the DNA in search of hereditary properties. He isolated two strains of pneumonia, an R strain and an S strain, which could be identified via a microscope. The S strain had the ability to transform the R strain in the S strain, even if the S strain wasn't alive. Avery extracted the building parts of a dead S strain cell and each time tested could the new mixture cause the transformation. He found that removing the DNA from the S strain bacteria would block the ability of the R strain to transform, so he concluded that the DNA must be holding the information required to build the proteins unique to the S strain.[5]

As the DNA is a long linear molecule the conclusion was that parts of it must be responsible for hereditary behavior. DNA usually came in pairs of two, which gave an explanation to Mendel's observation that every organism contained two distinct objects of genetic information, one of which dominates the other.

Another famous experiment that predates Avery's discoveries bears witness that genes were coded into the DNA. In the 1920s Thomas Morgan conducted similar experiments as Mendel, only with fruit flies which feature short life spans and produce numerous offspring. The white-eyed male fly was mated with its red-eyed sisters and its progeny was scrutinized closely for a few generations. The researchers observed that that white eyes appeared dominantly in males, which suggested that the "male gene" and the eye color gene were closely related. Further analysis revealed severals pairs of genes that occurred in strict dependence of each other. By examining the relative occurrence of two genes one could infer the "order" of the genes and conclude that there was a linear order between them. This order served as evidence that they reside on some linear molecule.[2]

2.4 From chromosomes to proteins

The DNA of a cells resides in a thick chromatic structure called the chromosome. By the early 1940s, researchers understood that a cell's genetic traits were hereditary and that they were organized in genes that resided on the chromosomes. They didn't, however, know what do the genes do to give raise to the cell's traits. George Beadle and Edward Tatum were the first to identify that genes actually contain information to code for a particular protein. They worked with the bread mold *Neurospora* that can survive on a very limited died of sucrose and salt. In 1941 they irradiated the mold with x-rays and examined their growth afterwards. As expected, some of the spores lost their ability to survive on the medium, so Beadle and Tatum postulated that the x-rays mutated one of the genes responsible for processing these simple nutrients into more complex compounds like amino acids. This gave raise to a simple conclusion

that one gene codes for one protein. This was a predominant theory during the next 50 years until scientists discovered that it is possible for a gene to produce a multitude of proteins.

The question of how exactly do the genes on the chromosome code the proteins remained. The DNA can be visualized as a long string of characters over the alphabet of acids $\{A, C, G, T\}$. A protein is a potentially long chain of amino acids, so one can think about them as a string over an alphabet of 20 as there are 20 different amino acids. The only feasible way for the DNA to code for a protein would be to use a substring of length at least three to code for a single amino acid as $4^3 > 20$. This is exactly how it works: each amino acid is coded with a contiguous substring of length exactly three this mapping is summarized in table 1. The coding three letter substrings of the DNA are called *codons*.

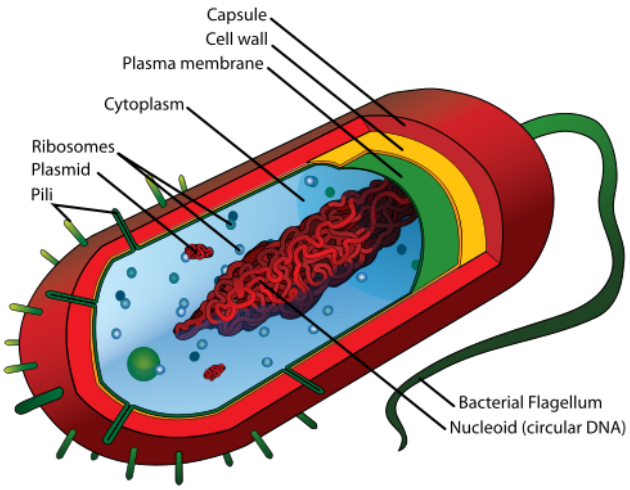
Amino acid	Codon
Ala/A	GCT, GCC, GCA, GCG
Leu/L	TTA, TTG, CTT, CTC, CTA, CTG
Arg/R	CGT, CGC, CGA, CGG, AGA, AGG
Lys/K	AAA, AAG
Asn/N	AAT, AAC
Met/M	ATG
Asp/D	GAT, GAC
Phe/F	TTT, TTC
Cys/C	TGT, TGC
Pro/P	CCT, CCC, CCA, CCG
Gln/Q	CAA, CAG
Ser/S	TCT, TCC, TCA, TCG, AGT, AGC
Glu/E	GAA, GAG
Thr/T	ACT, ACC, ACA, ACG
Gly/G	GGT, GGC, GGA, GGG
Trp/W	TGG
His/H	CAT, CAC
Tyr/Y	TAT, TAC
Ile/I	ATT, ATC, ATA
Val/V	GTT, GTC, GTA, GTG
START	ATG
STOP	TAA, TGA, TAG

Table 1: Mapping between the codons and the amino acids

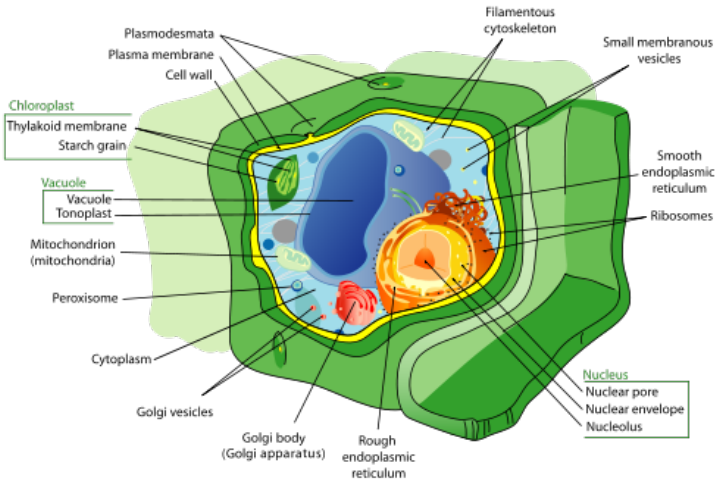
Understanding the connection between DNA and proteins began with the realization that proteins could not be made directly made from DNA as protein synthesis happen outside the cell nucleus, while DNA resides in it. The first to explain this mechanism were Benjamin Hall and Sol Spiegelman that demonstrated that a molecule called RNA can bind with the DNA and then transfer the information to the protein factories called the *ribosomes*. Thus, DNA served as a template used to copy a particular gene's genetic information to the ribosome to make a particular protein. The process of of transferring DNA information to the RNA is called *transcription*.

Another important discovery about the DNA came from the realization that not all of the information on the DNA was always used to make proteins. The

cells that have a well defined nucleus are called *eukaryotic* and they have both coding regions called *exons* and non-coding regions called *introns* in the DNA. The other type of cells that do not have a nucleus are called *prokaryotic* and they usually aren't found in multi-cell organisms[4]. Their DNA, contrary to the eukaryotic cells, have no introns - meaning that all of the DNA material is efficiently used to code for proteins. This can be explained with natural selection as using introns is energetically inefficient, which is important in organisms with a short life span.



(a) A typical prokaryotic cell, [7]



(b) A plant eukaryotic cell, [8]

[4]some cyanobacteria may be multicellular, see [6]

2.5 Proteins and how they are made

As stated previous, proteins are long chains of amino acids that play many critical roles in a cell. Unlike the DNA that is highly regular, protein structure is highly variable and intricately interlaced with their function. Here presented is a short describing a few examples of their function:

Antibody proteins Antibodies bind to specific foreign particles, such as viruses and bacteria, to protect the cell and its host.

Enzyme Enzymes are the catalyzers of chemicals reactions that take place inside a cell.

Structural component Those proteins build structural components of the cell such as the cell well and can form complex tissues such as elastic fibers and muscles.

In 1820 Henri Braconnot identified the first amino acid, glycine and by the early 1900s all twenty amino acids have been discovered [2]. They are combined together in a cell organelle ribosome to form a protein in a processes called translation. The ribosome gets the required information what protein to form from the RNA that was transcribed from the DNA. This flow of information,

$$\text{DNA} \rightarrow \text{transcription} \rightarrow \text{RNA} \rightarrow \text{translation} \rightarrow \text{protein},$$

is referred to as the central dogma in molecular biology. This pathway justifies the vast interest of researchers in DNA sequencing and analyzing technologies which will be presented in the next section.

2.6 Analyzing a DNA

I will present the most important biological tools that are used to extract and analyze an organisms DNA sequence.

DNA copying Experimental techniques such as gel electrophoresis, used for measuring DNA length, require a large amount of identical copies for any statistical significance. Since it is difficult to extract a single or even hundreds of molecules with modern instruments, amplifying DNA to yield billions or more copies is often a prerequisite of further analysis. This is achieved via *cloning*. The process starts with breaking DNA into small pieces and then inserting each individual piece inside a viral DNA. The virus does not lose its ability of self-replication, but now replicates with the inserted DNA string.

Measuring DNA length The previously mentioned gel electrophoresis technique allows one to measure the length of a DNA fragment without actually knowing the sequence in advance. As the DNA is a negatively charged molecule, it will migrate in a gel when subjected to an electric field. The gel acts as molecular "brakes" so that heavier (in this case also longer), molecules move slower than faster ones.

Cutting a DNA Restriction enzymes are compounds that act as molecular separators that cut the DNA at occurrences of a certain string (recognition sites). One example is the *BamHI* enzyme that cuts at every occurrence of the string GGATCC.

Probing The task of finding whether a particular DNA fragment occurs in a DNA solution can be solved by the processes of *hybridization*. A complementary DNA strand with fluorescent or radioactive tags is inserted into a single strand solution, the un-hybridized strands are washed away the solution is tested for change in wavelength or radioactivity.

Sequencing DNA One of the most important methods, Sanger's method, works by arbitrarily cleaving parts of the DNA off only one end. One is then presumably left with all possible prefixes of the analyzed strand. This is achieved with a fluoresce labeled di-deoxynucleotidetriphosphates that attach to a DNA end and can not be continued. All what is left is running the mixture through gel electrophoresis and reading off the tags at the end in reverse order. This method has been supplanted by novel "Next-Gen" sequencing methods for large-scale automated genome analysis. However, it remains in wide usage, especially for smaller-scale projects and for obtaining long contiguous sequences of DNA reads (500 - 1000 bases).

2.7 Mutation and species diversity

Genetic makeup manifests itself in traits such as eye color or susceptibility to Huntington's disease. The human genome, for instance, has over 3 billion base pairs, and even though there is only a 0.1% variation between any two individuals, no two individuals are quite the same. This can be easily explained on a formal mathematical level as $0.1\% \times 3 \cdot 10^9 = 3 \cdot 10^6$, so the number of theoretically different genomes is a vast $4^{3000000}$.

Another surprising fact is that when analyzing the genome of two unrelated species, their genomes appear amazingly similar. For instance, as many as 99% of the human genome is conserved across all mammals. Some human genes show string homological across worms, plants and (deadly) bacteria.

The DNA of a species changes between generations by mutation. Errors in self-replication, ionizing radiation or sexual recombination of two different strands result in a variety of possibilities. When looking at a genome of a individual, one can identify couple of most common mutation.

Substitution The most common mutation is the change of a single nucleotide base with another. For its transcription properties, a substitution can either have no impact on the protein if the old and new codons code for the same amino acid; it can change one amino acid to another; or worse, it can prematurely signal the protein to stop building, causing the product to be of limited, if any, function.

Insertion/deletion An insertion/deletion of a number of DNA bases into the organism's DNA. Smaller ones can affect only a single gene, while larger ones may delete or insert complete genes.

Duplication A fragment of DNA can spontaneously duplicate itself during recombination. Even though it is a rare occurrence, it has been postulated that this mechanism has been the main driving force for increasing DNA lengths that the organisms developed over time.

Inversion A fragment of DNA reverses itself during recombination. Slightly more common than duplication, although much rarer than substitution, insertion or deletion.

This concludes the minimum biological back-pinnings needed to understand this thesis. The next section will present the main algorithm tool-set that is most helpful with when dealing with biological problems and which will be used when presenting the results.

3 Algorithmic background

3.1 Suffix arrays

Suffix arrays are a fast and simple way to solve the following problem:

Definition 1 (Multi-pattern substring query problem)**.** *Given a string S, implement a data structure that can answer the following query:*

i Query(T): for a string T, find all occurrences of T as a substring of S.

Suffix arrays were introduced by Manber and Myers in [9] as a way to simplify the formerly known suffix trees. Formally, a suffix array $\pi(i)$ of a string S is an array of indices such that the suffixes

$$S_{\pi(1)..|S|},$$
$$S_{\pi(2)..|S|},$$
$$...$$
$$S_{\pi(|S|)..|S|}$$

are sorted. Here I denote the substring $S_a S_{a+1} ... S_{b-1} S_b$ as $S_{a..b}$. When the array π is constructed, one can answer queries for some T in a simple manner with the following algorithm.

Algorithm 1 Suffix array query when π is constructed

```
1:  function QUERY(T)
2:      l ← 1, r ← |S|
3:      for i in {1, 2, ..., |T|} do
4:          binary search for the first occurrence k of the character T_i in
    S_{π(k)+i−1} such that l ≤ k ≤ r
5:          l ← k
6:          binary search for the last occurrence k of the character T_i in S_{π(k)+i−1}
    such that l ≤ k ≤ r
7:          r ← k
8:      end for
9:      if l > r then
10:         return empty
11:     end if
12:     return (π_l, π_{l+1}, ..., π_r)
13: end function
```

The problem of constructing the array π itself is a much harder one. The naive approach of lexicographically sorting the suffixes achieves an $O(|S|^2 \log |S|)$ running time. However, an improvement can be made by optimizing the comparator in that sorting function and achieve a running time of $O(|S| \log^2 |S|)$. The problem in the comparator function is the following: given two indicies $1 \leq a \neq b \leq |S|$, is $S_{a..|S|}$ lexicographically before or after $S_{b..|S|}$? This answer can be solved via hashing with the following algorithm.

The only parts that was left out is the comparation between $S_{a..a+m} == S_{c..c+m}$ by comparing $hash(S_{a..a+m}) == hash(S_{c..c+m})$. This problem is, however, widely studies and can be most easily solved using hashing functions. See [10].

Algorithm 2 Optimized suffix comparator

```
1: function COMPARE((a, b))
2:     p ← LongestCommonPrefix(a, b)
3:     return compara S_{a+p} with S_{b+p}
4: end function
5: function LONGESTCOMMONPREFIX(a, b)
6:     l ← 0
7:     r ← |S| − max(a, b) + 1
8:     while l < r do
9:         m ← ⌈(l+r)/2⌉
10:        if hash(S_{a..a+m−1}) == hash(S_{b..b+m−1}) then
11:            l ← m
12:        else
13:            r ← m − 1
14:        end if
15:    end while
16: end function
```

More sophisticated algorithms for the construction of suffix arrays are available. The method that was described above shows a $(n \log^2 n)$ running time, which is often fast enough in practice, and is easy to implement. Better algorithms include the one described in [11] which features an minimal $\Theta(n)$ time complexity, near linear space complexity and competitive practical value. Recent work proposes an algorithm for updating a suffix array when new text has been inserted that performs well in practice (although with a worst case still $O(n \log n)$), see [12]. For a more complete taxonomy of suffix array algorithms, see [13].

3.2 Fenwick trees

A Fenwick tree is a classical data structure proposed in [14] that can improve the running time of numerous algorithms by optimizing its inner-most loops. Suppose one is working on an array A of integers and you need to find the sum $\sum_{i=a}^{b}$ multiple times. The naive approach would be to sum up the number one-by-one and it would yield a $\Theta(n)$ approach. Using a data structure like the Fenwick tree, one can implement this operation in $\Theta(\log n)$ and drastically improve the running time. We first need to state the problem formally:

Definition 2 (Summable array)**.** *Implement a data structure that can support these two operations over an (virtual) array A of size n:*

i Update(i, y): change the value of A_i to $A_i + y$ ($1 \leq n$; y arbitrary)

ii Query(b): report the sum $\sum_{i=1}^{i=b} A_i$ ($1 \leq b \leq n$).

Notice that the query operation reports only the prefix sums, but one can easily convert then into general contiguous sums by the identity

$$\sum_{i=a}^{b} A_i = \sum_{i=1}^{b} A_i - \sum_{i=1}^{a-1} A_i, \quad \text{for } 1 < a \leq b \leq n$$

The Fenwick tree data structure will support both operations in $\Theta(\log n)$ per operation. The structures maintains an array of S "running sums" such that

$$S_k = \sum_{k-\text{hipot}_2(k)<n\leq k} A_n$$

where $\text{hipot}_2(k)$ is defined as the largest power of 2 that evenly divides k, or more formally $\max\{2^n | k \text{ is divisible by } 2^n, n \in \mathbb{Z}\}$.

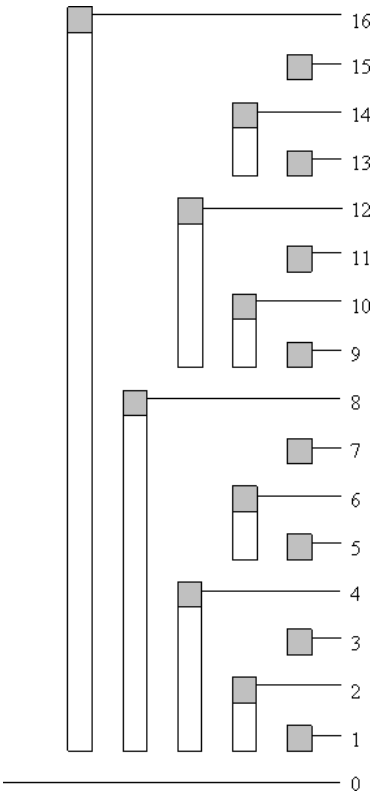


Figure 1: Diagram of responsibility for each element of S

The algorithm can be succinctly presented via recursion as follows, assuming one can use the hipot_2 function in $O(1)$.

The performance guarantee of the *Query* function can be seen as every step removes one active bit from the binary representation of i, while *Update* can be argued in a similar way.

A final thing to note is that the summable Fenwick tree can be trivially generalized to other operations obeying commutative and associative laws. For instance, one can have the maximum Fenwick tree where the *Query* operation reports the prefix maximum. In other words, a Fenwick tree can be used to solve the following problem:

Definition 3 (Prefix-max array)**.** *Implement a data structure that can support these two operations over an (virtual) array A of size n:*

i Update(i, y): change the value of A_i to y ($1 \leq n$; y arbitrary)

Algorithm 3 Fenwick tree operations for the summable array problem

```
1: function UPDATE(i, y)
2:     if i is beyond the end of the array then
3:         return
4:     end if
5:     S_i ← S_i + y
6:     Update(i + hipot_2(i), y)
7: end function
8: function QUERY(b)
9:     if b == 0 then
10:        return 0
11:    end if
12:    return S_b+ Query(b − hipot_2(b))
13: end function
```

ii Query(b): report the value of $\max_{i=1}^{i=b} A_i$ *(*$1 \leq b \leq n$*).*

Note however, that this operations can't be used to report maximums in an interval because the max function isn't invertible.

3.3 Longest common subsequence (LCS)

Historically, the problem of aligning two DNA sequences was solved by (dense) dynamic programming with high sensitivity and $\Theta(|X| \cdot |Y|)$ running time, recent proliferation of genomic data have rendered these approaches less useful. It does not make much sense to run such an algorithm when $|X| \approx |Y| \approx 10^8$. However, homologous regions in such sequences can often be recognized by less sensitive, but faster approaches like the k-longest common subsequence that will be introduced and used later in this thesis.

Formally, the problem is defined as finding the maximum n such that the there exists a common subsequence defined as below.

Definition 4 (Common subsequence)**.** *Given two strings X and Y consider two sets of distinct indices $I = i_1, i_2, ..., i_n$ and $J = j_1, j_2, ..., j_n$ such that $i_1 < i_2 < ... < i_n; j_1 < j_2 < ... < j_n$ and $X_{i(k)} = Y_{j(k)}$ for $k = 1..n$. Sets I and J determine a* common subsequence *of X and Y whose length is equal to n.*

The most famous method for calculating the LCS is an application of dynamic programming. Let $dp(i, j)$ be the longest increasing subsequence for the prefixes $X_{1..i}$ and $Y_{1..j}$. Then we have the following recursive relation.

$$dp(i,j) = \max \begin{cases} 0 & \\ dp(i-1,j) & i \geq 1 \\ dp(i,j-1) & j \geq 1 \\ dp(i-1,j-1) & X_i = Y_j \end{cases}$$

The longest common subsequence length is, of course, stored in $dp(|X|, |Y|)$. The time and memory complexity are of course $\Theta(|X| \cdot |Y|)$.

Research working towards a faster or more memory efficient approach for solving the LCS problem has been intensive. I will single out two most significant improvements that have been found:

Hirschberg's linear memory reconstruction algorithm While it is an easy fact that the LCS can be calculated in $O(|X| + |Y|)$ memory if the result doesn't have to be reconstructed, the existance of such memory efficient reconstructive algorithm is a quite difficult problem. Nevertheless, Hirschberg proved in [15] that such an algorithm exists and is currently widely used.

"Method of Four Russians" The only significant time-reducing algorithm found today is the "Four Russian Method" of block matrices with a time complexity of $O(\frac{n^2}{\log n})$ for two strings of length n[16].

A survey of more recent longest common subsequence approaches can be seen in [17].

3.4 LCSk++

$LCS++$ is a recent generalization of the LCS algorithm proposed in [1]. It can be viewed as a measure of similarity between the strings X and Y. It is defined as the longest common subsequence where each contiguous part is of length at least k. A more formal definition follows.

Definition 5 (k++ common subsequence)**.** *Consider a common subsequence of strings X and Y. Such subsequence uniquely determines two sets of indices I and J, $I = i_1, i_2, ..., i_n$ and $J = j_1, j_2, ..., j_n$ such that $i_1 < i_2 < ... < i_n; j_1 < j_2 < ... < j_n$ and $X_{i(k)} = Y_{j(k)}$ for $k = 1..n$. If both I and J can be partitioned into families of sets of consecutive indices such that every set has a size of at least k, this subsequence is called a* ***k++ common subsequence****.*

Let's denote the longest $k++$ common subsequence between the strings X and Y as $LCSk++(X, Y)$. While the general problem of LCS in unfeasable on genome-sized data, $LCSk++$ can be a reasonable choice if one chooses k wisely. The naive method of computation of $LCSk++(X, Y)$ can be furnished via the following recursive relation similar to the computation of LCS.

$$dp(i,j) \quad = \quad \max \begin{cases} 0 & \\ dp(i-1,j) & i \geq 1 \\ dp(i,j-1) & j \geq 1 \\ dp(i-q,j-q)+q & \text{for all } q \geq k \text{ s.t. } X_{i-q...i-1} = Y_{j-q...j-1} \end{cases} \tag{1}$$

The 2^{nd} and 3^{rd} terms in the above formula correspond to inheriting the $LCSk++$ value from previously computed values while the last term tries to extend the $LCSk++(X_{0...i-q-1}, Y_{0...j-q-1})$ with $X_{i-q...i-1}$ and $Y_{j-q...j-1}$ if they are equal. Those terms contribute $|X_{i-q...i-1}| = |Y_{j-q...j-1}| = q$ to the resulting length. A direct implementation of the above idea leads to an algorithm with time complexity $O(nm \cdot \min(n, m))$ where $|X| = n$ and $|Y| = m$.

3.4.1 An efficient algorithm for the $LCSk++$ computation

Definition 6 (Match pair[5]). *For a given strings X, Y and integer $k \geq 1$ we define:*

$$kMatch(i,j) = \begin{cases} 1 & if X_{i+f} = Y_{j+f}, for\ every\ 0 \leq f \leq k-1 \\ 0 & otherwise \end{cases} \quad (2)$$

If kMatch(i,j)=1, we call (i,j) *a* ***match pair****. In other words, kMatch(i,j)=1 when the substring of A starting at i and having an length of* ***exactly*** *k is equal to the substring of B starting at j with the same length. (i,j) is also called the* ***start*** *and $(i+k, j+k)$ is called the* ***end*** *of the match pair.*

For every match pair $P = (i_P, j_P)$ we use dynamic programming to compute $dp(P) = LCSk{+}{+}(X_{0\ldots i_P+k-1}, Y_{0\ldots j_P+k-1})$, which represents the value of $LCSk{+}{+}$ ending with P. The following definitions will be useful:

Definition 7 (Precedence of match pairs). *Let P=(i_P, j_P) and G=(i_G, j_G) be k-match pairs. Then G* ***precedes*** *P if $i_G + k \leq i_P$ and $j_G + k \leq j_P$. In other words, G precedes P if the end of G is on the upper left side of the start of P in the dynamic programming table (see Figure 2).*

Definition 8 (Continuation of match pairs). *Let P=(i_P, j_P) and G=(i_G, j_G) be k-match pairs. Then P* ***continues*** *G if $i_P - j_P = i_G - j_G$ (i.e. they are on the same primary diagonal) and $i_P - i_G = 1$ (P is only one down-right position from G, see Figure 2).*

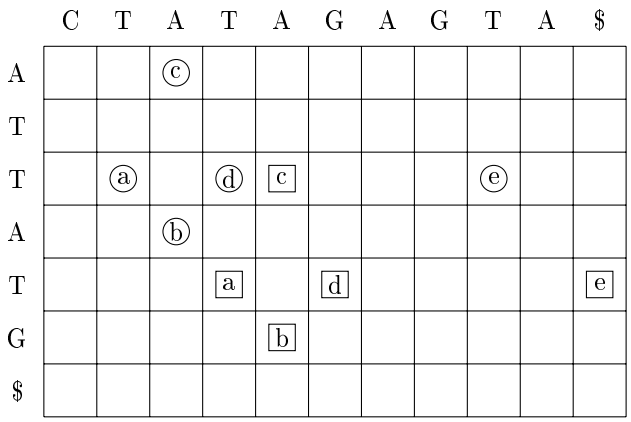


Figure 2: $k = 2$; strings $X = ATTATG$ and $Y = CTATAGAGTA$ construct exactly five 2-match pairs denoted a to e. Starts are represented by circles, ends are represented by squares. The following holds: "b continues a", "c precedes e", while the following does not hold: "a precedes b", "c precedes d", "a continues b".

We can express the $dp(P) = LCSk{+}{+}(X_{0\ldots i_P+k-1}, Y_{0\ldots j_P+k-1})$ via the fol-

[5] Original definition given in [18].

lowing formula which will be the basis for the efficient algorithm:

$$dp(P) \quad = \quad \max \begin{cases} k & \\ \max_G dp(G) + k & \text{over all G preceding P} \\ dp(G) + 1 & \text{if P continues G} \end{cases} \tag{3}$$

In other words, a k-match pair P can either start its own k-common subsequence, extend a k-common subsequence ending with a match G such that G precedes P or extend a k-common subsequence ending with a match pair G such that P continues G. In the second case the k-common sequence is enlarged by k (e.g. $c \to e$ in figure 2), while in the latter it's enlarged by 1 (e.g. $a \to b$ in figure 2).

Algorithm 4 Efficient $LCSk{+}{+}$ computation

```
1:  MaxColDp ← 1D array filled with n zeros
2:  MatchPairs ← find all k-match pairs between X and Y
3:  events ← all starts and ends of MatchPairs sorted in row-major order, if
    some start S = (i_S, j_S) and some end E = (i_E, j_E) share the same indices,
    E should come first
4:  for all event ∈ events do
5:      if event is a start P = (i_P, j_P) then
6:          dp(P) ← k + max_{x∈0...j_P} MaxColDp(x)
7:      else if event is an end P = (i_P + k, j_P + k) then
8:          if ∃G s.t. P continues G then
9:              dp(P) ← max{dp(P), dp(G) + 1}
10:         end if
11:         MaxColDp(j_P + k) ← max{MaxColDp(j_P + k), dp(P)}
12:     end if
13: end for
14: return max_P dp(P)
```

Algorithm 4 starts by extracting all of the r match pairs on line 2. This can be done in two ways: one can employ a suffix array in the exact same manner as in [19] to get the time complexity of $O(n + m + r)$. However, as k is small in practice[6] we can find all match pairs using a simple hash table in $O(n + m + kr) = O(n + m + r)$.

Line 3 creates events and sorts them, which ensures the correctness of the sweeping algorithm on lines 4-14. This can be accomplished in $O(r \log r)$ using a standard comparison based sorting algorithm. Line 8 can be implemented as a binary search over the *events* array. If $MaxColDp$ is implemented as a Fenwick tree [14], the operations on lines 6 and 11 have a cost of $O(\log n)$ which implies that the sweep algorithm runs in $O(r \log n)$. Overall complexity is $O(m + n + r \log r)$[7]. The memory complexity is $O(n + m + r)$ because we only need the space to save the match pairs and the $MaxColDp$ structure. If we would like to reconstruct the sequence, the dp array has to store $O(1)$ additional information per match pair: a pointer to the previous match pair in case some other match pair G preceded P or P continued some G in the optimal solution.

[6] Usually k will be small enough such that every k-length substring can be perfectly hashed using 64 bits.

[7] This is assuming that r is at least as big as n. In the other case the correct complexity is $O(m + n + r \log r + r \log n)$.

3.5 Smith-Waterman algorithm

The problem of (local) sequence alignment was first posed and solved by T. F. Smith and M. S. Waterman in 1981 (see [20]). The problem can be formally stated as follows:

Definition 9 (Alignment)**.** *Given two DNA sequences X, Y over an alphabet Σ, their alignment is a pair of equal-length strings (A, B) over an alphabet $\Sigma \cup \{'-'\}$ such that X is a subsequence of A and Y is a subsequence of B.*

Definition 10 (Global sequence alignment problem)**.** *Given two DNA sequences X, Y over an alphabet Σ and a scoring function $\delta : \Sigma \cup \{'-'\} \times \Sigma \cup \{'-'\} \to \mathbb{R}$, find the alignment of (A, B) that maximizes $\sum_i \delta(A_i, B_i)$.*

Definition 11 (Local sequence alignment problem)**.** *Given two DNA sequences X and Y, find a subsequence A of X and B of Y to maximize the global sequence alignment value between them.*

The algorithm Smith and Waterman proposed for this problem relied on a clever dynamic programming application not unlike the longest common subsequence problem. Let $dp(i, j)$ be the maximum local sequence alignment of the prefixes $X_{1..i}$ and $Y_{1..j}$. Then we can calculate the optimal local alignment of the entire sequence by the following recursion.

$$dp(i,j) = \max \begin{cases} 0 & \\ dp(i-1,j) + \delta(X_i, -) & i \geq 1 \\ dp(i,j-1) + \delta(-, Y_j) & j \geq 1 \\ dp(i-1,j-1) + \delta(X_i, Y_j) & i,j \geq 1 \end{cases}$$

The Smith-Waterman algorithm for local sequence alignment has often been described as the most sensitive of all the alignment algorithms. Its drawback, however, is its unavoidable $\Theta(|X| \cdot |Y|)$ running time. This led to the development of GPU aware alignment algorithms to try to mitigate the running time (see [21]), albeit the algorithm will probably never be able to align chromosome-sized sequences.

4 A new global sequence alignment tool

4.1 Problem statement

The problem of global sequence alignment traditionally included running expensive $\Theta(n^2)$ algorithms that are infeasible on genome size data. The approach to global alignment taken in this thesis is akin to the one taken by the popular tool MUMmer (see [22]): instead of calculating the whole genome alignment that will make sense on the homologous regions and won't make any sense on the regions that aren't related by function or lineage, we will try to identify only the homologous regions.

To put this into perspective, we can look at figure 3 which shows the famous alignment between the first human (Homo Sapiens) chromosome and the entire genome of a mouse (Mus Musculus). While there is certainly many important homologous regions present, the area between the regions seem not to carry any significant information.

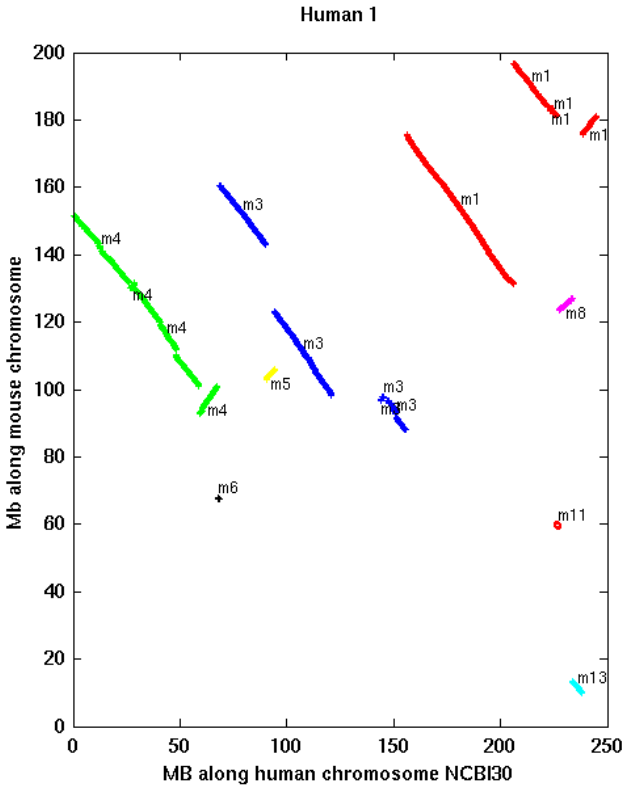


Figure 3: Homologous regions between the first human chromosome and the entire mouse genome [23]

Definition 12 (Whole sequence alignment)**.** *Given two DNA strings X and Y, find all the significant homologous regions between those two strings.*

As the "homologous regions" themselves are a fuzzy term that is difficult to quantify, I will measure my success ratio in correspondence to the regions found by other researchers such as the one on figure 3.

4.2 Alignment algorithm

The main algorithmic overview can be relatively easily stated once the groundwork of section 3 have been laid.

The first requirement on our algorithm is to detect both forward and reverse matches. This will be dealt in a straightforward manner by running the algorithm in two separate instances: once for the forward run, another time for the reverse. This is arguably the approach taken by both MUMmer [22] and BLASR [24]. Assuming we are given two DNA strings X and Y, the algorithm proceeds as follows:

1. Construct a suffix array of the string Y.
2. For each i in $1, 2, ..., |X|$ calculate the longest common prefix of $X_{i..|X|}$ that occurs within Y, let it's length be L. Discard this match if $L < k$, where k is an fixed parameter, otherwise find all occurrences of the string $X_{i...i+L-2}$ within Y.
3. Between all the matches found in the last step, calculate the best sparse Smith-Waterman path. This step will be described in detail in the next section.
4. Find the highest scoring value in the sparse Smith-Waterman matrix, reconstruct its path, remove it and repeat this step. Repeat until there is a sharp drop in the quality of the paths.

Notice that in step 2 we are looking for all occurrences of the prefix of length $L - 1$ and not L. The reason for this is that the last character can sometimes be the result of pure chance, so this enhances the sensitivity of the algorithm. This move has been inspired by BLASR that uses a very similar technique.

Another thing to keep in mind is the coy parameter k that determines the minimum length of a valid match. This parameter is paramount to the speed and sensitivity of the tool and will be analyzed in one of the following sections.

4.3 Sparse Smith-Waterman approximation

The centerpiece of the tool is a modified sparse Smith-Waterman dynamic programming algorithm. Its input is an array of exact match pairs from the suffix array query stage of the algorithm and its output is an approximation of the Smith-Waterman score for every match pair in the input.

Its formal definition can somewhat similar as for the $LCSk++$ problem, but is given here for completeness.

Definition 13 (Match pair)**.** *Given a string X, Y, a match pair is a triplet (a, b, l) such that $X_{a..a+l-1} == Y_{b..b+l-1}$.*

Definition 14 (Match pair sequence value)**.** *Given an ordered list of match pairs $\{(a_1, b_1, l_1), (a_2, b_2, l_2), ..., (a_n, b_n, l_n)\}$ such that $a_i + l_i \leq a_{i+1}$ and $b_i + l_i \leq$*

b_{i+1} whenever i and $i+1$ are defined, one can assign a value to the list by computing

$$E \cdot \sum_{i=1}^{n} l_i + N \cdot \sum_{i=1}^{n-1} (b_{i+1} - b_i - l_i) + (a_{i+1} - a_i - l_i).$$

This value is the match pair sequence value. *The values $N < 0$ and $E > 0$ are algorithm parameters.*

The crux of the upper definition lies in the fact that the match pair sequence value approximates the Smith-Waterman value for sparse matrices. To be more precise, suppose that we are given a special Smith-Waterman scoring function $\delta : \Sigma \cup \{'-'\} \times \Sigma \cup \{'-'\} \to \mathbb{R}$ function such that

$$\delta(a,b) = \begin{cases} E & a = b \\ N & a = - \vee b = - \\ 2N & \text{otherwise} \end{cases}$$

and that all matches between X and Y have been registered in the match pairs list. Then the biggest possible match pair sequence value is the same as the Smith-Waterman optimal local sequence alignment value. While this comparison may seem overly overly optimistic as the proposed δ function is Very special, one can still make a claim that this function will still be at most constant times worse than any other, "practical", functions, hence preserving the optimality ratio for arbitrary long sequences. The other objection to this result is that usually not all of the possible matches reside in the acquired list, but only the ones that are longer than a parameter k. This, however, is a minor setback as short matches are more often noise than signal. On the other hand, setting k can greatly reduce the running time.

We can finally define our approximation of the Smith-Waterman algorithm with the following problem.

Definition 15 (Optimal match pair sequence problem)**.** *For each match pair P in the input list, compute the largest match pair sequence value one can get with any sequence ending with P.*

The question how to efficiently solve this problem remains. Similarly as in the $LCSk++$, we turn to sparse dynamic programming. The following definition will make the description a bit more legible.

Definition 16 (Precedence of match pairs)**.** *Let $P = (i_P, j_P, l_P)$ and $G = (i_G, j_G, l_G)$ be match pairs. Then P precedes G if $i_P + l_P \leq i_G$ and $j_P + l_P \leq j_G$. In other words, if we sketch each match pair with the two dimensional line $(i + \tau, j + \tau)$ where $0 \leq \tau < l$), then P will be strictly up and to the left of G.*

Let's define $dp(P)$ as the optimal match pair sequence value ending with P. We can then state a recursive relation that will lay the groundwork for an efficient solution.

$$dp(P) = \max \begin{cases} E \cdot l_P \\ N \cdot ((i_P - i_G - l_G) + (j_P - j_G - l_G)) + E \cdot l_P & G \text{ precedes } P \end{cases}$$

The former equation can, however, be rewritten in the following manner:

$$dp(P) = \max \begin{cases} E \cdot l_P \\ (Ni_P + Nj_P + El_P) - N(i_G + j_G + 2l_G) & G \text{ precedes } P. \end{cases}$$

Notice that the left hand side part of the last equation depends on only P, while the right hand side depends on only G. Therefore we can uncouple the entire equation into if we divide the match pairs into "starts" (i_P, j_P) and ends $(i_P + l_P, j_P + l_P)$. The full algorithm follows.

Algorithm 5 Efficient optimal match pair sequence problem

```
1: function SparseLongestMatchSequence((MatchPairs))
2:     MaxColDp ← Fenwick max-array initialized with n −∞
3:     events ← all starts and ends of MatchPairs sorted in row-major order,
   if some start S = (i_S, j_S) and some end E = (i_E, j_E) share the same indices,
   E should come first
4:     for all event ∈ events do
5:         if event is a start P = (i_P, j_P) then
6:             bestG ← MaxColDp.Query(j_P)
7:             dp(P) ← bestG + Ni_P + Nj_P + El_P
8:         else if event is an end P = (i_P + k, j_P + k) then
9:             MaxColDp.Update(j_P + l_P, dp(P) − N(i_P + j_P + 2l_P))
10:        end if
11:    end for
12:    return dp(P), ∀P
13: end function
```

Before I get to the results, the next section will detail the repercussions involving the choice of the critical parameter k.

4.4 Theoretical background

As was said, all match pairs shorter than a fixed k will be discarded. If one chooses k too small, a proliferation of unimportant match pairs will occur and running times will deteriorate exponentially. On the other hand, if one chooses k too large, no matches will be found. This section will try to answer the question of picking k just large enough for the algorithms to be feasible on genome-scale data.

We model a pair of *non-homologous strings* $X^{(n)}$ and $Y^{(n)}$ of length n as random strings over the alphabet $\{A, C, G, T\}$ where each character has a known and mutually independent probability of appearing.Specifically, let X and Y be either non-homologous or homologous pair of strings of length m, n constructed over the alphabet $\{A, C, G, T\}$ with a priori distributions p_A, p_C, p_G, p_T. Then $S := E[X_i = Y_j] = (p_A^2 + p_C^2 + p_G^2 + p_T^2)$ for all $i \neq j$ (this restriction is needed to cover both models with this proof). Expected number of match pairs then equals:

$$
\begin{aligned}
E[r] &= E[\#\text{ of pairs (i,j) such that } X_{i..i+k-1} = Y_{j..j+k-1}] \\
&= E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} 1[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} 1[i=j]1[X_{i..i+k-1} = Y_{j..j+k-1}]\right] + \\
&E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} 1[i\neq j]1[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= O(n+m) + \sum_{i=0}^{n-k}\sum_{j=0}^{m-k} E\left[1[i\neq j]1[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= O(n+m+nmS^k).
\end{aligned}
$$

Corollary 1. *By choosing* $k_{fast} = \log_{1/S}\frac{nm}{n+m}$ *it follows that* $E[r] = O(n+m)$, *so the expected complexity of the whole algorithm is* $O((n+m)\log(n+m))$.

Corollary 2. *For uniformly distributed alphabets, the expected number of match pairs drops as the size of the alphabet increases. That implies that the bigger the alphabet is, the smaller* k *is needed for the LCSk++ computation to run efficiently.*

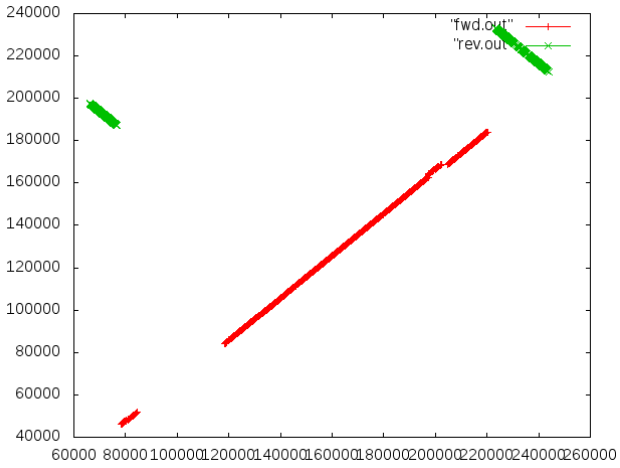
This results proves the efficiency of the method provided one chooses k carefully.

5 Results

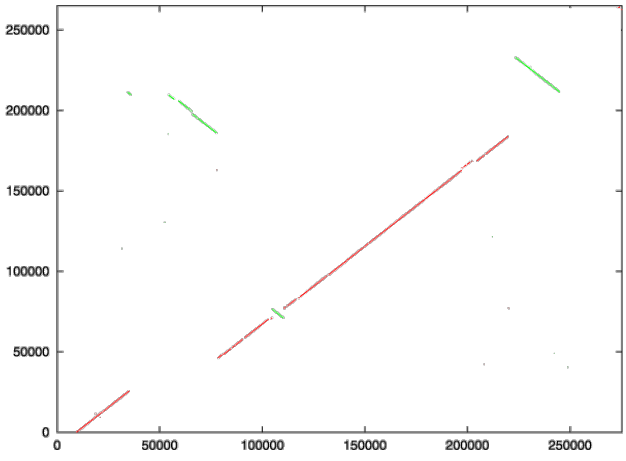
The results are obtained by testing the tool on several problem instances such as

- Helicobacter pylori (two strains)
- Human chromosome 1 vs. mouse chromosome 4
- Randomly generated data of various sizes.

As a quality check, I will measure the similarity between the new tool and MUMmer generated plots. The following figure shows a global alignment between H. pylori, strain 26695, and H. pylori, strain J99.



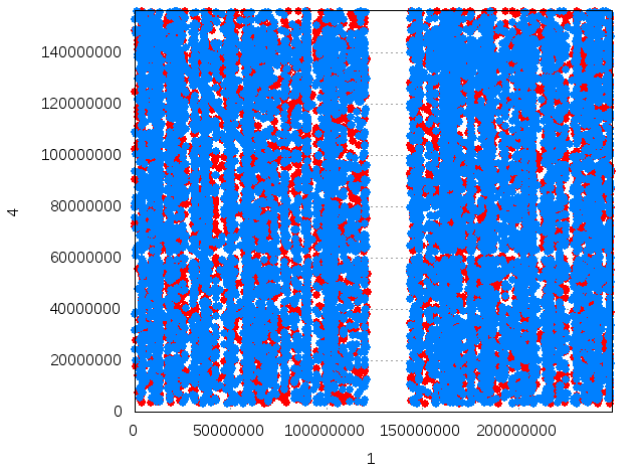
(a) H. pylori 26695 vs J99: sparse match sequences



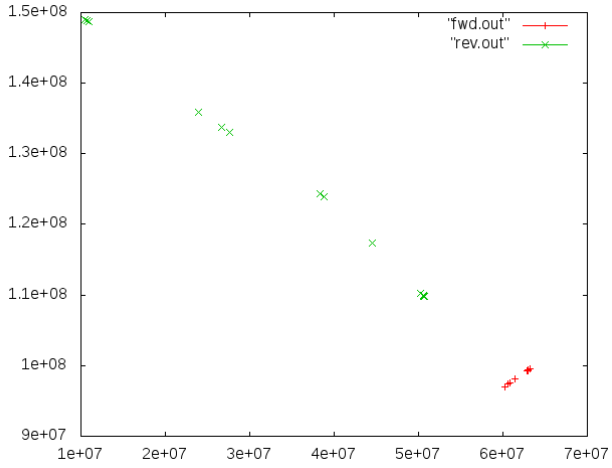
(b) H. pylori 26695 vs J99: MUMmer

As seen from the figures, the quality of the uncovered pictures is quite similar. This examples was a moderately small $250KB \times 250KB$ alignment, a scale feasible even for full Smith-Waterman iterations. The next example, however,

features the entire human chromosome 1 vs. the entire mouse chromosome 4. This corresponds to aligning $250MB \times 150MB$, a scale unfeasible by ordinary methods.



(a) Human chromosome 1 vs. mouse chromosome 4: sparse match sequences



(b) Human chromosome 1 vs. mouse chromosome 4: MUMmer

This example shows a different story the the previous one. The MUMmer output is cluttered by random noise in the signal, while the new algorithm is much more resilient to spurious data. A thing to note is that setting MUMmer parameters, even manually, helped little in mitigating the clutter that can be seen.

Another critical factor that has been neglected so far is the time complexity of the algorithm. Tested together were MUMmer version 3; BLASR; and the new algorithm. Table 2 shows the corresponding running times of these algorithms on various input sequences.

Input	Size	new tool time	MUMmer time	BLASR time
H. pylori 26997 vs J99	$250KB \times 250KB$	$<$ 1s	$<$ 1s	8s
Randomly generated	$1MB \times 1MB$	$<$ 1s	$<$ 1s	28s
Randomly generated	$10MB \times 10MB$	6s	9s	6m 14s
Homo S. 1 vs. Mus M. 4	$250MB \times 150KB$	3m 7s	7m 4s	didn't finish
Randomly generated	$1GB \times 1GB$	11m 50s	13m 41s	N/A

Table 2: Running times of the new algorithm, BLASR and MUMmer on various input sequences

6 Conclusion

To sum up, I have developed a new whole genome sequence alignment tool that is blazing fast, can handle huge amounts of data, and has the sensitivity to detect large homologous regions across different species.

In particular, the tool has shown far better efficiency performance than BLASR and comparable efficiency as MUMmer. On the other hand, it lacks the sensitivity of other bioinformatics tools.

For future work, I would recommend improving the sensitivity of tool via nonexact match searching and seed extension techniques similar to BLAST.

References

[1] F. Pavetić, G. Žužić, and M. Šikić, "*lcsk++*: Practical similarity metric for long strings," *arXiv preprint arXiv:1407.2407*, 2014.

[2] N. C. Jones and P. Pevzner, *An introduction to bioinformatics algorithms*. MIT press, 2004.

[3] F. Rosner, "Hemophilia in the talmud and rabbinic writings," *Annals of Internal Medicine*, vol. 70, no. 4, pp. 833–837, 1969.

[4] O. V. G. Mendel, *the first Geneticist*. Oxford: Oxford University Press, 1996.

[5] O. T. Avery, C. M. MacLeod, and M. McCarty, "Studies on the chemical nature of the substance inducing transformation of pneumococcal types induction of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type iii," *The Journal of experimental medicine*, vol. 79, no. 2, pp. 137–158, 1944.

[6] R. Mur, O. M. Skulberg, and H. Utkilen, "Cyanobacteria in the environment," 1999.

[7] "Prokaryotic cell." `http://en.wikipedia.org/wiki/File:Average_prokaryote_cell-_en.svg`. Accessed: 2015-1-10.

[8] "Eukaryotic cell." `http://en.wikipedia.org/wiki/File:Plant_cell_structure_svg.svg`. Accessed: 2015-1-10.

[9] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[10] M. C. Harrison, "Implementation of the substring test by hashing," *Communications of the ACM*, vol. 14, no. 12, pp. 777–779, 1971.

[11] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Data Compression Conference, 2009. DCC'09.*, pp. 193–202, IEEE, 2009.

[12] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard, "Dynamic extended suffix arrays," *Journal of Discrete Algorithms*, vol. 8, no. 2, pp. 241–257, 2010.

[13] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *acm Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.

[14] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.

[15] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[16] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664–675, 1977.

[17] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pp. 39–48, IEEE, 2000.

[18] G. Benson, A. Levy, and B. Shalom, "Longest common subsequence in k length substrings," in *Similarity Search and Applications* (N. Brisaboa, O. Pedreira, and P. Zezula, eds.), vol. 8199 of *Lecture Notes in Computer Science*, pp. 257–265, Springer Berlin Heidelberg, 2013.

[19] S. Deorowicz and S. Grabowski, "Efficient algorithms for the longest common subsequence in k-length substrings," *Information Processing Letters*, vol. 114, no. 11, pp. 634 – 638, 2014.

[20] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[21] M. Korpar and M. Šikić, "Sw#–gpu-enabled exact alignments on genome scale," *Bioinformatics*, p. btt410, 2013.

[22] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic acids research*, vol. 27, no. 11, pp. 2369–2376, 1999.

[23] "Human chromosome 1 and mouse genome comparison." `http://en.wikipedia.org/wiki/File:Plant_cell_structure_svg.svg`. Accessed: 2015-1-20.

[24] M. J. Chaisson and G. Tesler, "Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory," *BMC bioinformatics*, vol. 13, no. 1, p. 238, 2012.

Alat za globalno poravnanje sekvenci

Sažetak

Ovaj rad predlaže bioinformatički alat za poravnavanje cijelog genoma. Dano je detaljno objašnjenje algoritma te usporedba s drugim renomiranim alatima koji rješavaju slični problem poput MUMmer-a i BLASR-a. Dobiveni algoritam se pokazao kao vrlo efikasan, usporediv s alatom MUMmer, iako uz manju mjeru osjetljivosti.

Ključne riječi: bioinformatika, poravnanje sekvenci, sufiksni nizovi, LCSk++, rijetko dinamičko programiranje

Global sequence alignment tool

Abstract

This thesis proposes a bioinformatics tool that can be used for whole genome alignment. A detailed description of the algorithm is provided and compared to well-established tools like MUMmer and BLASR. The algorithm proves to be highly efficient, comparable to MUMmer, although with lower sensitivity.

Keywords: bioinformatics, sequence alignment, suffix arrays, LCSk++, sparse dynamic programming