Comparison of texts using string kernels
Frederik Brinck Truelsen, 20116293

Master’s Thesis, Computer Science
January 2017
Advisor: Christian Nørgaard Storm Pedersen
Abstract

This thesis is concerned with the distance between texts of strings, mainly molecular sequences but not limited to. We achieve this through a vectorization, using string kernels, representing the strings as frequency vectors indexed by their substrings counting the occurrence of that given index.

This thesis contributes by defining two algorithms for calculating the distance given this vectorization, with upper bounds of $O(n^3)$ and $O(n^2)$, respectively. The algorithms have been implemented, tested and evaluated. Experiments have been carried out verifying the time complexities are satisfied in practice.

We aimed for a linear time complexity, instead we achieved a $O(n^2)$ running time. We carried out experiments, to benchmark the implementations against each other, as well a small application of the distance measure.
Acknowledgements

A special thanks goes to my thesis advisor Christian Nørgaard Storm Pedersen for advice and help with the problems that appeared along the road. I would like to thank Lars Løbner Hansen, Jan Brinck and Mathias Vorreiter Pedersen for dedicating a portion of their time to proof-reading and to making suggestions to my thesis. Thanks to Marni Tausen for pleasant conversation and sparing on various subjects and for his help. Heartfelt thanks goes to Mai Løbner Andersen for keeping me fed and motivated during this process, and proof-reading my thesis.

Frederik Brinck Truelsen,
Contents

Abstract iii

Acknowledgments v

1 Introduction 1
   1.1 Thesis Structure ................................. 1

2 Fundamentals 3
   2.1 Definitions and Notation ............................ 3
   2.2 Suffixes ........................................... 4
   2.3 String Kernel ...................................... 4
   2.4 Measures .......................................... 5
   2.5 Neighbor Joining ................................... 6
   2.6 Robinson–Foulds Metric ......................... 6

3 Algorithms 9
   3.1 Naive ...................................................... 9
   3.2 Suffix Tree ........................................... 10

4 Implementation 13
   4.1 Naive ...................................................... 13
   4.2 Suffix Tree ........................................... 15
   4.3 Testing of Correctness .............................. 17

5 Experiments 19
   5.1 Setup ..................................................... 19
   5.2 Naive ..................................................... 20
   5.3 Suffix Tree ........................................... 21
   5.4 Construction Time ................................... 22
   5.5 Benchmarks ............................................ 24
   5.6 Application ............................................ 28

6 Conclusion 29
   6.1 Future Work ........................................... 29
Chapter 1

Introduction

The comparison of strings has been a field of interest of many in the past few decades, and especially for molecular sequences more recently. In the past few decades, a rather large volume of molecular sequences have been generated, and therefore lots of information on living organisms, their traits and evolution has been discovered.

As these sequences only contains a limited alphabet size, they solely give us a limited amount of information. In order to extract more information out of these sequences a way of comparing these sequences are essential. The sequence comparison methods can be divided into two main categories: alignment-free [1, 2] and alignment-based [3, 4].

In this thesis we propose an algorithm for alignment-free comparison of strings, including two algorithms a naive and a linear time algorithm, respectively.

The focus of this thesis is mainly two-folded. The first objective of this thesis is to implement alignment-free comparison of strings. This being based on string kernels and the corresponding vectorization of strings and to calculate the distance between these vectors, based on inner products.

The second objective is do experiments both to verify, that the running time of the two implementations compares to those of the theoretical time complexity of the algorithms, and to benchmark the implementations against each other.

An application of the proposed alternative was partially carried out to compare the alternative to a known alignment-based distance.

1.1 Thesis Structure

This section will briefly describe how the remaining part of the thesis will be structured.
Chapter 2. Fundamentals  Introduces the fundamentals used in the thesis. This consists of the basic background theory and notation for understanding the algorithms described in chapter 3.

Chapter 3. Algorithms  Describes two variants of the naive implementation, as well as a linear time algorithm (based on suffix tree). In the thesis, one of the naive and linear time algorithms have been implemented and will be compared.

Chapter 4. Implementation  The implementation details of the two algorithms are described. Lastly the correctness of the implementations will be argued.

Chapter 5. Experiments  Describes the experiments carried out throughout the thesis. The main experiments will be investigating the running time of the algorithms in comparison to the time complexity in theory. Benchmark experiments will also be carried out to compare the performance of the algorithms against each other. A small application of the novelty distance measure compared to known alignment-based was partial carried out.

Chapter 6. Conclusion  Summarizes all the most important aspects of the thesis in a final conclusion, and describes the future work, as well as possible extensions.
Chapter 2

Fundamentals

This chapter will cover the fundamentals and preliminary theory for calculating an alignment-free distance between strings, as well as clustering based on some distance measure.

2.1 Definitions and Notation

This sections covers the basic definitions and notation used throughout the thesis.

**Strings** Let $\Sigma$ be a non-empty finite set of symbols (Alternately called characters), called the alphabet. Let an array $x : [0...n]$ of $\Sigma$ with length $n = |x|$. The letter at position $i$ is $x[i]$, i.e $x = x[1]x[2]...x[n]$.

**Substrings** $x[i...j] = x[i]x[i+1]...x[j]$ is a substring of $x$ with length $j - i + 1$, a proper substring if $j - i + 1 < n$. The substring occurs at position $i$ in $x$.

**Uniformly random strings** This type of string is generated by choosing every character in the string from a uniform distribution over some alphabet, for instance $\Sigma = \{A, G, C, T\}$.

**Taxa** A group of one or more populations of an organism or organisms to form a unit.

**k-mer** The set of all possible substrings of length $k$ that is contained in a string.

**∞-mer** A simple extension of the above definition with a none fixed $k$, and therefore the infinite set of all possible substrings over the alphabet.
2.2 Suffixes

This section will present the suffix tree data structure. The suffix tree will be used directly in the implementation of the linear time algorithm for finding distances between strings. I will very briefly touch on construction time, as it is essential for the overall running time of the algorithm.

**Suffix Tree.** Let $S[0..n]$ be a string $S$, that has a special ending character $S[n] \notin \Sigma$. The suffix tree of $S$ is then denoted $T_S$ and is a tree where the edges are denoted by strings, such that every suffix of $S$ is represented by a path of a leaf in $T_S$. For notation we denote the path label between node $u$ and $v$ as $label(u, v)$. There exists a generalization of the tree called a generalized suffix tree, in which the tree contains the suffixes of multiple strings. In the thesis we will only consider trees containing exactly two strings. You construct these trees by simply concatenating the strings, all of these ending on their unique ending character, or by building the tree one string at a time.

![Suffix Tree Diagram]

A suffix tree $T_S$ of a string $S$ can be constructed in $O(n)$ using several different construction algorithms. However very little focus will be placed on construction throughout the thesis. We only consider Ukkonen’s algorithm defined by Ukkonen [5], since this is the construction algorithm used when a suffix tree is needed for the implementation.

2.3 String Kernel

A string kernel is a kernel function [6] that operates on strings, i.e. finite DNA sequences of symbols, that does not need to be of the same length. String kernels is intuitively understood as a function measuring the similarity of a pair of strings. The more similar the two strings $s_1$ and $s_2$ are, the higher
the value the string kernel $K(s_1, s_2)$ will be. The string kernels are used in domains where sequences are to be clustered or classified, e.g. in text mining and gene analysis, which we are concerned with in this thesis.

A kernel on a domain $D$ is a function $K : D \times D \rightarrow \mathbb{R}$. The function $K$ can be expressed as $K(s_1, s_2) = \phi(s_1) \cdot \phi(s_2)$ with $\phi$ being the mapping $\phi : \Sigma^n \rightarrow \mathbb{R}^k$ of finite strings of length $n$ into a vector of $k$ dimension. We call this an inner product space, as we require this vector space to have a well defined construct of inner product between its elements.

2.4 Measures

Given the definition above, we can define our measures based on this vectorization of strings into an inner product space. The precise derivation of the inner product will be carried out in chapter 3, which leads to the formalization of the algorithms, as these follow directly from the definition of the known euclidean dot product.

**Cosine Similarity**  The cosine of two non-zero vectors can be derived from the standard definition of euclidean inner products.

$$v_1 \cdot v_2 = ||v_1|| \cdot ||v_2|| \cdot \cos(\theta)$$

Given two vectors of attributes, $v_1$ and $v_2$, the cosine similarity, $\cos(\theta)$, is defined using the dot product and magnitude as:

$$\cos(\theta) = \frac{v_1 \cdot v_2}{||v_1|| \cdot ||v_2||}$$

In general the resulting similarity ranges from $-1$ meaning exactly opposite, to 1 meaning exactly the same, with 0 indicating orthogonality and in-between values indicating the intermediate similarity or dissimilarity.

In the thesis the vectors only contain strictly positive numbers, and therefore range from 0 meaning exactly opposite, to 1 meaning the same.

**Angular distance**  Based on the cosine similarity, we are able to define an angular distance based on arc degree, which for the remainder of the thesis is the distance measure referred to unless mentioned otherwise:

$$distance = \frac{\cos^{-1}(\text{similarity}) \times 180}{\pi}$$
2.5 Neighbor Joining

Neighbor joining is a bottom-up (agglomerative) clustering method for the creation of phylogenetic trees, made by Saitou and Nei [7]. The algorithm is used for tree reconstruction, based on either DNA or protein sequence data. The algorithm requires some distance knowledge between each pair of taxa to form the tree.

The Neighbor joining algorithm requires a distance matrix specifying the distance between each pair of taxa. Given this matrix it calculates a matrix $Q$, that have entries $Q(i,j) = (n - 2)d(i, j) - \sum_{i=1}^{n} d(i, k) - \sum_{i=1}^{n} d(j, k)$, where $d(i, j)$ is the distance between taxa $i$ and $j$.

**Neighbor Joining**

1: Calculate the Q matrix based on the distance matrix
2: Find the taxa $i$ and $j$, where $i \neq j$ and $Q(i, j)$ have the lowest values. These taxa are joined to create a new node.
3: Calculate the distance from each of the taxa in the pair to this new node.
4: Calculate the distance from each of the taxa outside of this pair to the new node.
5: Repeat, and replace the pair of joined neighbors with the new node using the distance calculated in the previous step.

The RapidNJ library [8] is used for tree reconstruction in the application of the novelty distance measure, that in chapter 5 is compared to a known alignment-based distance measure.

2.6 Robinson–Foulds Metric

The Robinson-Foulds distance or metric, is a measure of the distance between unrooted phylodetic trees. It is defined as $S_{T1} + S_{T2}$, where $S_{T1}$ is the number of splits implied in the $T_1$ but not in $T_2$ and $S_{T2}$ is the number of splits implied by the $T_2$ but in not in $T_1$. It is also known as a symmetric difference metric. The algorithm sketched below, made by William H. E. Day [9] is a linear time algorithm for calculating the distance between unrooted phylodetic trees, and the implementation of the Robinson–Foulds distance used in the thesis. The implementation yields a $O(n \ast log(n))$ time complexity as a standard sorting algorithm, that has been used in the implementation.
Days Algorithm
1: Root the two trees $T_1$ and $T_2$ at the same leaf
2: Annotate leaves of $T_1$ with a Depth-First numbering
3: Annotate the leaves in $T_2$ cf. DF-numbering of leaves in $T_1$
4: Annotate internal nodes in $T_1$ with their DF-intervals (The DF-numbering of the leaves).
5: For each internal node in $T_2$ find the min and max DF-leaves and size of the sub tree, if max - min + 1 = size, then the sub tree is an interval of the DF-leaves, and is a potential shared split. Annotate the $T_2$ node with this interval.
6: Every DF-interval in $T_1$ which also occurs in $T_2$ is a shared split. Sort the intervals and identify doublets.

Normalized Robinson–Foulds The normalized Robinson–Foulds metric is simply the above definition, normalized with respect to the number of internal nodes in the two trees you are calculating the distance between. This gives a similarity measure between 0 and 1, with respect to how many nodes the two trees do not share in relation to how many internal nodes they have overall. We define it as follows:

$$1 - \frac{S_{T_1} + S_{T_2}}{T_1^i + T_2^i}$$

Where $T_1^i$ and $T_2^i$ is the number of internal nodes in the two trees, respectively.
Chapter 3

Algorithms

This chapter will cover the two algorithms for finding the dot product between the strings given their vectorization. The algorithms will both be presented by starting from the mathematics that form the foundation and leading to a formalization of an algorithmic definition, that summarizes the steps in broad terms to actually make these algorithms.

Later in later chapter 4 we look into further detail and based on these algorithmic definitions will show the actual implementation work.

The time complexity of the algorithms will be explored in detail in chapter 5, where we dive into comparing the algorithms time complexity stated in this chapter, to those of the implementation in the next chapter.

3.1 Naive

Let $T_1, T_2 \subseteq \Sigma^*$ be texts of strings over some alphabet $\Sigma$. It follows directly from the definition of euclidean space inner product, given the vectorization of strings is defined as:

$$\vec{t}_1 \times \vec{t}_2 = \sum_{s \in \Sigma^*} \#occ_{T_1}(s) \cdot \#occ_{T_2}(s)$$

A problem that stems from this definition from being formalized into algorithm is the fact that it is defined as an infinite sum over the $\Sigma^*$. To solve this problem let us take a closer look at our definition, and what happens when $|s| > |T_1|$. This entails that $\#occt_1 = 0$. There is infinitely many of these empty terms which add nothing. We make a small modification our to sum and get the following definition:

$$\vec{t}_1 \times \vec{t}_2 = \sum_{s \in \Sigma^* \text{such that } |s| \leq \min(|T_1|, |T_2|)} \#occ_{T_2}(s) \cdot \#occ_{T_2}(s)$$
We could take this definition and derive an algorithmic definition, but the problems with this definition is, that there is exponentially many terms, since we sum over all strings \( s \in \Sigma^{\min(|T_1|,|T_2|)} \), many of these being zero terms.

We make a modification to our definition which allows for speeding up the algorithm. We use the known folklore that \( T_1 \) can have at most \( |T_1|^2 \) terms and that a term only contributes, when it occurs in both \( T_1 \) and \( T_2 \).

Instead of taking the sum over all the terms with right length in the alphabet, we sum over all the substrings that occur in our strings.

\[
\vec{T}_1 \times \vec{T}_2 = \sum_{s \subseteq T_1 \land s \subseteq T_2} \#occ_{T_2}(s) \cdot \#occ_{T_2}(s)
\]

From this mathematical definition we derived an algorithmic definition, as seen below.

**Algorithm 3 Naive Algorithm**

1: function GENERATE(T)
2: Let \( v \) be vector of frequencies indexed by strings
3: for all \( s_i \in \text{subsTr}(T) \) do
4: if \( s_i \in v.\text{index} \) then
5: \( v.freq \leftarrow v.freq + 1 \)
6: else
7: \( v.freq \leftarrow 1 \)
8: return \( v \)
9: function DOTPRODUCT(\( T_1, T_2 \))
10: Generate \( v_1 \) and \( v_2 \)
11: \( \text{sum} \leftarrow 0 \)
12: for all index \( i \) in \( v_1 \) do
13: for all index \( j \) in \( v_2 \) do
14: if \( v_1.\text{index}(i) = v_2.\text{index}(j) \) then
15: \( \text{sum} \leftarrow \text{sum} + v_1.freq \cdot v_2.freq \)
16: return \( \text{sum} \)

We know from previous, that there is at most \( O(n^2) \) entries, and that it takes linear time to read the indexes to compare them, thus yielding a time complexity of \( O(n^3) \) for the above algorithmic definition.

### 3.2 Suffix Tree

Let \( T \) be a generalized suffix tree (GST) build from \( T_1 \) and \( T_2 \) with two internal nodes \( u \) and its predecessor \( v \) in the tree. Let \( \alpha \) be the \( \text{label}(v, u) \). Let \( T_u^1 \) and \( T_u^2 \) denoted the number of leafs/suffixes from \( u \)'s sub tree for \( T_1 \) and \( T_2 \), respectively.
In the figure above we look at the suffix $s$ which spells a word, given that a path from the root of the tree to a position on the edge of $v \rightarrow u$ (incl. $u$) it follows that $u$ contributes with the following:

$$\text{occ}_{T_1}(s) \cdot \text{occ}_{T_2}(s) = T_u^1 \cdot T_u^2$$

There is $|\alpha|$ of these locations that contributes to the sum that ends on the $v \rightarrow u$ edge, which means $u$ overall contribution is given by:

$$|\alpha| \cdot T_u^1 \cdot T_u^2$$

In the end we have to sum these contributes from all the internal nodes $v_i$ in the GST, where $|\alpha|$ is the length of the label between $v_i$ and its predecessor. This leads to the following definition:

$$\overrightarrow{T_1} \times \overrightarrow{T_2} = \sum_{\text{nodes } v_i} |\alpha| \cdot T_u^1 \cdot T_u^2$$

In the following algorithmic definition We summarize the necessary steps to make an algorithm, that follows the mathematical definition stated above.

Figure 3.1: Generalized suffix tree for the suffix $s$ annotated with $T_u^1$ and $T_u^2$
Algorithm 4 Suffix Tree Algorithm

1: Build the GST over $T_1$ and $T_2$
2: Annotate all internal $v_i$ nodes with $T_{v_i}^1$ and $T_{v_i}^2$ by doing a DFS-traversal of the GST
3: for all internal nodes $v_i$ do
4: \[ \text{sum} \leftarrow \text{sum} + |\alpha| \cdot T_{v_i}^1 \cdot T_{v_i}^2, \] where $|\alpha|$ is the length of the edge label between $v_i$ and its predecessor

We know from the fundamentals in chapter 2, that we can build the generalized suffix tree in $\mathcal{O}(n)$ and that a traversal of a tree takes linear time proportional to the number of internal nodes in the tree. Therefore yielding a $\mathcal{O}(n)$ time complexity of the above given algorithmic definition.
Chapter 4

Implementation

We have successfully implemented the algorithms described in the previous chapter. The implementations were done in Java 8. For the generalized suffix tree we used a library available at https://github.com/abahgat/suffixtree. The implementation is available at http://bit.ly/2jiIokJ. The code is accompanied with a data folder that contains the real data sets and the generated trees made by running a multiple sequence alignment on the data set, giving it to the library code used for tree reconstruction. Note that a recent JVM installation is required to run the program.

4.1 Naive

In this section we briefly describe the most important pieces of the implementation, and show only the very essence of the implementation and therefore left pieces of the code out and shortened the code example by removing the none essential aspects of the code samples. The naive implementation is composed of two main parts, that we will explain separately below. We ran into some memory issues during implementation. We will address the ideas and the chosen solution in further details below.

4.1.1 Memory Issues

The memory issues lead the implementation to throw a memory exception on DNA sequences of only 1500 characters, with the Java VM running out of memory causing the exception. We realized that this happened due to the sheer amount indexes with a length close to the original string, we stored in our data structure. We devised two possible solutions to the problem, which we describe in further detail below and do a very short analyses of the chosen solution.

Instead of indexing by strings, we thought of using a pair \((i, j)\) of integers as our index. The pair contains the start and end position of the occurrence...
of the index string in the original complete string. This leads to a much more effective memory usage and keeps the linear complexity of comparing if two indexes are identical.

The other idea is to hash the index strings, using these value as our index. This leads to a smaller memory usage than the previous as we are using an integer number instead of a representation of a pair, as well as leading to a constant time complexity for comparing of the indexes. This do however lead to an increase in the time complexity of the construction from quadratic to cubic time.

We chose to hashing due to simplicity of adding it to our implementation and the idea to of potentially reusing the data structures we favored having the heavier computation on the construction.

### 4.1.2 Construction

The first step of the implementation is to generate the frequency vector. Concretely we have chosen the hash map data structure as our representation of the vector. The hash map fits nicely with our abstraction of the vector from earlier, as you read an index and get the corresponding frequency for that given index.

The construction simply consists of generating the hashes of all the possible substrings and update accordingly based on whether we have seen this index before or not. If we have seen the index before, then we retrieve the value and increase it by one, otherwise we make a new entry with one as the initial value.

```java
private Map<Integer, Integer> generate(String str) {
    int length = str.length();
    Map<Integer, Integer> mapVec = new HashMap<>();

    for (int i = 0; i < length; i++) {
        for (int j = i + 1; j <= length; j++) {
            int subStrHash = str.substring(i, j).hashCode();
            if (mapVec.containsKey(subStrHash)) {
                mapVec.replace(subStrHash, mapVec.get(subStrHash) + 1);
            } else {
                mapVec.put(subStrHash, 1);
            }
        }
    }
    return mapVec;
}
```
4.1.3 Inner product

We can simply calculate the inner product between the frequency vectors by iterating over either of them, while checking for equal indexes in the other frequency vector. We know from previous chapter that a term in the sum is exactly the product of the indexes that are equal.

```java
private int dotProduct(Map<Integer, Integer> vecMap1, Map<Integer, Integer> vecMap2) {
    int sum = 0;
    for (Integer key : vecMap1.keySet()) {
        if (vecMap2.keySet().contains(key)) {
            sum += vecMap1.get(key) * vecMap2.get(key);
        }
    }
    return sum;
}
```

4.2 Suffix Tree

In this section we briefly describe the most important pieces of the implementation for the suffix tree implementation. We only showing the very essence of the implementation and therefore leaving pieces of the code out and shortening the code examples, by removing none essential aspects of the code. The suffix tree implementation is composed of four main components, and we leave two of them out. The two components we leave out are the first and the last. We do this as the one part being only library code, which we have not written and the other is the simple similarity calculation.

4.2.1 Annotation

In this section we are annotating the internal nodes of the generalized suffix tree with the frequency occurrence counts. To shorten the code example we have removed the initialization of the two data structures. We use one to build the other one, that is the stack that guide our path trough the tree. We use the stack to go trough the tree in a reverse level order traversal, this ensures that we visit the children of the node beforehand, and that the immediate values for the node is updated before updating the node. The rest of the code checks whether its a leaf or an internal node. In case it is a leaf, we check which special character seen on the edge and update accordingly.
```java
public void updateFreq() {
    ....
    while (!stack.empty()) {
        Node current = stack.pop();
        for (Edge e : current.getEdges().values()) {
            Node child = e.getDest();
            if (child.getCount(0) == 0 && child.getCount(1) == 0) {
                String label = e.getLabel();
                if (label.charAt(label.length() - 1) == '$') {
                    current.updateFreq(0);
                } else {
                    current.updateFreq(1);
                }
            } else {
                current.updateFreq(0, child.getCount(0));
                current.updateFreq(1, child.getCount(1));
            }
        }
    }
}
```

### 4.2.2 Inner product

We simply calculate the inner product between the frequency vectors represented by the generalized suffix tree, by traversing the tree, beside the root as it counts for the empty substring. We completely follows the algorithmic definition seen previous, taking the sum over all the contributes of the internal nodes.
```java
public int dotProduct() {
    Queue<Edge> queue = new LinkedList<>();
    int result = 0;

    updateFreq();
    queue.addAll(root.getEdges().values());

    while (!queue.isEmpty()) {
        Edge currentEdge = queue.poll();
        Node currentNode = currentEdge.getDest();
        String CurrentLabel = currentEdge.getLabel();

        result += CurrentLabel.length() * 
                  currentNode.getCount(0) * currentNode.getCount(1);

        queue.addAll(currentNode.getEdges().values());
    }

    return result;
}
```

### 4.3 Testing of Correctness

Having implemented the algorithms for calculating the distance between strings by representing them as frequency vectors, we wanted to test the correctness of each of them. Since the naive algorithm simply generates all of the possible substrings for each of the given strings and saves the substring hash as index and its frequency in some data structure.

The essential component that make up this distance measure is the inner product, and a term is simply the multiplication of the frequencies if the two substrings occur in both their respective data structures.

We verified the algorithm, by running it on small examples we checked by hand as well. We convinced ourselves that the implementation does not contain any errors. The suffix tree based implementation was tested in comparison to the naive implementation on multiple uniformly random strings of various lengths, yielding the same results. This gives us no cause to believe that the suffix tree based implementation has any errors.
Chapter 5

Experiments

5.1 Setup

All the experiments were performed on a Lenovo T530 laptop with the following specification:

- Running Ubuntu 16.04
- CPU: Intel i5-3320M @ 2.60 GHz
- 8 GB RAM

The experiments were all performed from IntelliJ with a maximum memory heap size of 2048 MB. We ran all the timing experiments on the algorithms ten times and the plot consist of the median value of these runs.

The use of median values was chosen to avoid plotting outliers affecting the result. Such outliers could be caused by the operating system prioritization of other processes. By removing these we hope to achieve a more representative picture.

5.1.1 Data

In order to test the algorithms, solid test data is vital. For the two main types of experiments we chose synthetic data, created by sampling from uniformly random strings, consisting of characters for DNA sequences. For the application some real protein sequences were chosen, to give a picture of the performance of the novelty measure compared to a known alignment-based measure on real data.

5.1.2 Timing data

For the timing and benchmark experiments we used uniformly random strings with length variations from fifty up to a few thousands characters, with an
alphabet consistent with that of DNA sequences. We chose this particular data due to two main criteria. Namely that are extremely easy to construct and the implementations do not depend on the appearance of the data.

5.1.3 Real data

For the real data we have chosen 395 protein sequences with various lengths from the P-type ATPase database, given in fasta-format. We originally chose this data in particular because of a known biological grouping, into five groups indicated by the first character in its name in the fasta-file. We wanted to use this to carry out a clustering application using our distance measure in a k-meoid to check how well this replicates the known partition. However we kept this data in our small application experiments to test how well it compares to an alignment-based measure.

5.2 Naive

In this section the theoretical time complexity of the naive algorithm will be compared to the actual running time of the naive implementation. The algorithm has a time complexity of $O(n^3)$.

Figure 5.1 shows the running time of the algorithm calculating the distance between uniformly random strings. The algorithm has a theoretical time complexity of cubic time, and therefore a cubic correlation would be expected. The figure shows as expected a cubic correlation.
Figure 5.1: Running time of the naive implementation with a cubic regression showing its coefficient of determination. Lengths from 0 to 2500, with length intervals of 50.

5.3 Suffix Tree

In this section the theoretical time complexity of the the suffix tree algorithm will be compared to the actual running time of the suffix tree implementation. The algorithm has a linear time complexity of $O(n)$.

Figure 5.2 shows the running time of the algorithm calculating the distance between uniformly random strings. The algorithm has a theoretical time complexity of linear time, and therefore a linear correlation would be expected. The figure shows unexpectedly a quadratic correlation. We have a hunch, that it stems from the library code building the generalized suffix tree and will investigate this further.
5.4 Construction Time

In this section we will investigate the construction time of the two proposed algorithm. We closely investigate the unexpected behavior of the suffix tree implementation, that we had a hunch stems from the construction of the generalized suffix tree, as well as investigating the construction time of the naive algorithm.

5.4.1 Naive

In this experiment we expect a construction time of $O(n^3)$ for the frequency vector in the naive implementation. Since we implemented our frequency vector as a hash map containing all possible $O(n^2)$ substrings, as well as using linear time to hash the substrings into the integer index, this leads to a cubic time complexity of the construction. Figure 5.3 shows as expected a cubic correlation.

Figure 5.2: Running time of the suffix tree implementation with a quadratic regression showing its coefficient of determination. Lengths from 0 to 2500, with length intervals of 50.
Figure 5.3: Construction time of the frequency vector in the naive implementation, based on the hash map data structure, with a cubic regression showing its coefficient of determination. Lengths from 0 to 2500, with lengths intervals of 50.

5.4.2 Suffix Tree

In this experiment we investigate the unexpected quadratic time behavior noticed previously. We use library code supposedly promising a linear time construction algorithm of generalized suffix trees. We build the tree in our implementation by placing one string at a time into the tree. We realized afterwards that this not effective, if we wish to achieve linear time construction.

In this experiment we investigate whether it is our flawed use of the library or the library itself that does not deliver a linear running time.

The figure 5.4 shows the running time of the generalized suffix tree, given concatenation of the two strings. The figure shows a quadratic correlation regardless of this change, therefore is it the library code that does not deliver a linear time construction.
5.5 Benchmarks

In this section, the running time of the naive and suffix tree algorithm will be compared in order to investigate how they compete against each other. The benchmark experiments was done with two different setup where we both include and exclude construction time for both the algorithms, respectively.

5.5.1 Benchmark including construction

Figure 5.5 shows the running time of naive and the suffix based implementation. The running time of the naive implementation is clearly much slower than that of the suffix tree implementation, as it looks almost as an constant function with only a small increase of the slope at the very end.

Figure 5.6 shows the running time of the suffix tree implementation such that we closer can compare the actual difference of the running time for the two algorithms. The figure shows there is a approximately difference of a factor of one hundred between our naive implementation and our suffix tree implementation.
Figure 5.5: Benchmark of the naive implementation compared to the suffix tree implementation. Lengths from 0 to 2500, with length intervals of 50.

Figure 5.6: Running time of the suffix tree implementation on data with lengths from 50 to 2500 with length intervals of 50
5.5.2 Benchmark excluding construction

Figure 5.7 shows the running time of naive and the suffix based implementation, without the construction time of their respective data structures. The running time of the naive implementation is clearly much slower than that of the suffix tree implementation, as it looks almost as an constant function with only a small increase of the slope at the very end.

Figure 5.8 shows the running time of the suffix tree implementation, without construction, such that we closer can compare the actual difference of the running time for the two algorithms. The figure shows there is a approximately difference of a factor of sixty between the naive implementation and our suffix tree implementation.

![Benchmark of the Naive vs. Suffix without construction](image)

Figure 5.7: Benchmark of the naive implementation compared to the suffix based implementation, with no construction time of the data structure for either implementations. Lenghts from 0 to 2500, with length intervals of 50.
Figure 5.8: Running time of the suffix tree implementation, excluding the construction time. Lengths from 0 to 2500, with length intervals of 50.
5.6 Application

In this section a small application of the novelty distance measure was carried out where we compare the distance measure with the following known multiple sequence alignment tools: Clustal Omega, Kalign, MAFFT and MUSCLE [10]. We constructed alignments and a distance matrix based on the multiple sequence alignment tools and the distance taken between all the sequences using the novelty distance measure, respectively.

From these we did a tree reconstruction using neighbor joining, and using the RapidNJ library. The alignments and the distance matrix as well as the trees from the reconstruction can be found in the data folder along side the implementation.

The following tables summarizes the distances of the known-alignment based implementations only. We unfortunately had problems with the implementation of the Robinson-Foulds metric on the generated trees from the rapidNJ library, based on the distance matrix generated by our program. We were unable to calculate the distance between these trees. The plan was to take the novelty distance measure and investigate the trade offs between using our implementation or the known alignment-based, to look into if using our implementation causes any reduction in precision and how much compared to the known alignment-based algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Clustal</th>
<th>Kalign</th>
<th>MAFFT</th>
<th>MUSCLE</th>
<th>Novelty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustal Omega</td>
<td>1</td>
<td>0.72</td>
<td>0.79</td>
<td>0.69</td>
<td>NaN</td>
</tr>
<tr>
<td>Kalign</td>
<td>0.72</td>
<td>1</td>
<td>0.27</td>
<td>0.27</td>
<td>NaN</td>
</tr>
<tr>
<td>MAFFT</td>
<td>0.79</td>
<td>0.27</td>
<td>1</td>
<td>0.33</td>
<td>NaN</td>
</tr>
<tr>
<td>MUSCLE</td>
<td>0.69</td>
<td>0.27</td>
<td>0.33</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>Novelty</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1: Table over the Normalized Robinson-Foulds of the trees based on the 395 sequences, using the three multiple sequence alignment tools of Clustal Omega, Kalign, MAFFT and MUSCLE
Chapter 6

Conclusion

The main focus of this thesis has been to create an alternative alignment-free based linear time and naive algorithm for calculating the distance between strings based on a vectorization using string kernels.

We have proposed and implemented the two algorithmic definitions, derived from the standard mathematical definition of inner products.

We have done running time experiments verifying the running times of the implementations against the theoretical running time of their algorithmic definition. This lead to a verification of the running time of the naive implementation and discovery of the running time of the suffix tree implementation to be $O(n^2)$, due to the library code used for the generalized suffix tree not being linear.

Furthermore we did construction time experiments verifying the cubic time complexity of the naive, as well as the quadratic time complexity of the generalized suffix tree.

Benchmark experiments where carried out verifying the speed of the suffix tree implementation to be much faster than that of the naive both in- and excluding the construction time, achieving a factor of one hundred and a factor of sixty, respectively.

These experiments showed us the complexity to be quadratic for the naive implementation and linear for suffix tree implementation without their construction.

We succeeded with a fast suffix tree implementation, but not a linear running time as we had hoped, because the libery code could not build the generalized suffix tree in linear time.

6.1 Future Work

In this section we describe some of the work originally planned that we did not manage to get done due to time constraints, as well as some logical extensions and improvements of the work.
Construction of suffix tree  A natural extension and optimization of the thesis work, would be to fix the library code or extend the code with an implementation for the construction of the generalized suffix tree, that have a linear time complexity.

Comparison of multiple strings  In the application section in the previous chapter we deployed a strategy of comparing one DNA sequence against all other sequences. This leads to building the generalized suffix tree multiple times changing only one of the sequences contained in the tree. A natural extension of the work done in the thesis would be to instead build the annotated generalized suffix tree once for all the DNA sequences you are comparing, having some indexes into the tree you use to pick which two sequences you compare.

Applications  We did only a small application, though there is a wide range of applications we can think of similar to the one we carried out, but to a less limited extent, such as carrying the application on a wider range of sample DNA sequences, as well as a range of other applications comparing the distance measure with other known both alignment-free and alignment-based measures. We had originally a clustering application planned, using k-medoid on some sequences with a known partition to investigate how well the novelty distance measure would reproduce this known partition. The partition is produced by neighbor joining sequences based on their distance from an alignment-based distance measure. The above applications would also be interesting to apply to strings in general and not solely on sequences.
Bibliography


