RNA Sequences Similarity-based Search using Set-based Vector-based Models

Technical Report

Rayan Saade
Michel Mansour
Edwin Odeimi

Abstract — The advent of rapid RNA sequencing methods has greatly accelerated biological and medical research and discovery. Knowledge of RNA sequences has become indispensable for basic biological research, and in numerous applied fields such as medical diagnosis, biotechnology, virology, and biological systematics. Specifically, RNA sequencing facilitates the ability to look at alternative gene spliced transcripts, post transcriptional modifications, gene fusion, mutations, and changes in gene expression over time, or differences in gene expression in different groups or treatments. The rapid speed of RNA sequence processing coined with modern RNA sequencing technology has been instrumental in the identification of complete RNA sequences of numerous types of genes. This document serves to describe the techniques employed in designing RNA differencing and similarity searching tools.

Index Terms — Edit Distance, RNA, Sequencing, Jaccard, String, Mults, PCC, TF, IDF

- - - - - - - - - - - - - ◆ - - - - - - - - - - - -

CONTENTS

1 Introduction..... 1
2 Background..... 1
3 Design and implementation…..... 2
4 Experimental evaluation..... 5
5 Personal Experience.....5
6 References 6

1 Introduction

Sequence comparison is a common practice in different fields, mainly in bioinformatics when comparing DNA and RNA sequences. Multiple tools and algorithms have been implemented along the years in hope to provide the most precise and efficient model. The goal of this project is to implement different set-based and vector-based models used in RNA sequence comparison. Our implementation will then be compared to find the best performing model when it comes to both time and quality. Finally, we will provide atoll capable of taking an RNA sequence as input and list the most similar RNA sequences from a provided dataset.

2 Background

The comparison between two or more RNA sequences is the most important aspect of our RNA similarity search component. Thus, we need to have a well functioning tool to differentiate the sequences before proceeding to the dataset searching and ranking phase. To provide a well functioning tool, we need to accurately compute the Edit Distance. We then need to implement well renown algorithm like the Jaccard set-

based model or the Cosine vector-based model. These algorithms will provide us with the needed similarity coefficient that will be essential for the enactment of a dataset similarity searching component.

3 Design and implementation

3.1 Introduction

In this project we decided to implement all the set-based and four vector-based algorithms that we have learned in class. The set-based algorithms were relatively easy to implement since they are all quite similar after managing to calculate the multiplicity of the elements in the RNA sequence. The more challenging part were the vector-based models, specifically the fact that the PCC measure slightly differs from the others. What we struggled most with, was implementing a searching component that would be using all the previous models on multiple sequences which required us to write variations of our previous methods to make them compatible with a dataset.

3.2 Concepts and Building Blocks

3.2.1 RNA Sequence pre-processing

The pre-processing phase is divided into two crucial classes. The “Literal” class, same as the one used in the first project serves for tokenization and dissecting the sequence into nucleotide symbols and more importantly provides options to what a certain symbol can be. For instance, a Literal “R” can be both the Literal “A” and “G”.

What is new here, is the “Frequency” class, used to calculate the multiplicity of the 4 basic nucleotide symbols in a sequence. This is done by assigning a weight corresponding to the probability of a nucleotide. A literal “U” would increment the multiplicity of “U” by 1 while a literal “N” would only increment it by 0.25 since we have a probability of 25% of it corresponding to a “U” nucleotide. All multiplicities are then return in an array with indexes in the following order, A, C, G, U.

Before we are able create our search engine, we need a dataset to search from we do so by creating an xml file and we preprocess all the sequences that are randomly generated in order to calculate the TF and IDF of each and store them in the xml file along with the sequence. XML file would look like this:

```
<RNAdb>
    <NVMGCUU id="0">
        <A>6.730420570591233</A>
        <C>12.945028669013427</C>
        <G>9.837724619802328</G>
        <U>13.98286822144993</U>
    </NVMGCUU>
    ……
</RNAdb>
```

3.2.2 RNA set-based representations

Three set-based similarity coefficients were computed. All three of them are rather similar with a few alterations to the formula. We started off with the intersection which was just a simple for loop to calculating the sum of the smaller multiplicity between the two sequences for each nucleotide symbol. The coefficient is the normalized though a simple division by the minimum among the two multiplicity sums.

The Jaccard representation is then just a small modification to the final formula's denominator, using the already calculated sums of multiplicity for each sequence, making it look like this:

$$Sim_{Jaccard} = \frac{Intersection}{mults(Sequence1) + mults(Sequence2) - Intersection}$$

With mults being the summation of all multiplicities of a given sequence.

Lastly, we compute the Dice Similarity Measure which simply removes the intersection from the denominator of the previous formula and multiplies the whole thing by 2.

3.2.3 RNA vector-based representations

Vector-based model are a bit more complicated to implement. Before we start implementing the four chosen algorithms, we need to properly identify the Term Frequency TF and the Inverse Document Frequency IDF which will be used in all of our models. The TF of each sequence is none other than the multiplicities provided by our "Frequency" class in the pre-processing phase. To be able to calculate the IDF we the most essential constant in the formula DF. Computing the DF was as simple as checking for every non-zero value in the mults array of each sequence signifying that least one occurrence of the element and incrementing DF by 1 to a maximum of 2. We then apply the following formula for IDF using the constant 2 as the number of documents in the collection as we only have 2 sequences.

$$IDF = \log\left(\frac{N}{DF}\right)$$

We already know that the weights needed in our computation are the product of TF and IDF for each element which are all stored in a dedicated array for each sequence.

After managing to compute the weights of each element in for both sequences we can now measure the similarity using each model. We started off with the simplest model, the cosine measure. Measuring the numerator and denominator of this model is quite simple and only requires one for loop to compute the summations needed. The formula is then applied to return the similarity. The Euclidean and Manhattan measure are a bit similar as they both require us to compute a distance. This also only requires one for loop to get the sum of either the difference of weights squared or their absolute value, we then get the square root of both. The last and slightly more complicated model is the Pearson Correlation Coefficient which required us to get the average values of the weights before using them in the formula's numerator and denominator which are then computed similarly to the cosine model.

3.2.4 Symbol weighting

The next step in our implementation is allowing the user to fine-tune TF-IDF weights. To allow that to happen we only had to add an if statement for the computation of the TF or the IDF and adding two Boolean variables to the parameters of each vector-based method. The Boolean variable will depend on whether the user ticks the corresponding box or not.

3.2.5 RNA set and vector similarity evaluation

So far, we managed to compute all the different similarities and can now compare them to see their accuracy. But a model must not only be accurate but should also have optimized processing and computation times. The evaluation of these two times is done using Java's System nanoTime() method which starts a timer in nanoseconds. We added the method at the beginning of each of our algorithms and saved the timer at the end of processing the weights and variables needed before calculating the processing by simple subtraction of the start and stop time, we later do it again for the computation time.

These constants needed to be returned along with the computed similarity which is why we created an object called SimilarityResult having as parameters the similarity computed and the processing and computation time. The SimilarityResult is thus returned by every method implemented.

3.2.6 Comparison with ED measure

To provide the user with the ability to compare all the models with the ED measure, we utilized the ED computing tool implemented in project 1. The method was then configured to get the processing and computing time and return a SimilarityResult object. We also added the option for the user to alter the weights of the Insert, Delete and Update by adding those as parameter to the method.

3.2.7 RNA similarity-based search function

First of all, to be able to provide the user with a search engine we rewrote a variation of each similarity retrieval method in order to accept the multiplicity of elements through its parameters, since those will be provided by the dataset. We also created arrays for each model in order to store all the similarities of a given model for multiple RNA sequences from the dataset.

Two new essential methods are implemented here, the first one to check whether the sequence compared is relevant or not. This is done by first checking whether the input length is larger or smaller than the sequence the sequence retrieved by from the database. If it is larger the two sequences are flipped so that the retrieved sequence acts as the input. Next, we compare characters that could've been or the character itself with the character with the same index in the other sequence.

E.g : input= ACM…. result=ACR….

Temp1= "M" + M.canBe() = "MAC"

Temp2= "R" + R.canBe() = "RAG"

Does temp1 contain any character in temp2?

If yes, then continue to the next iteration.

If not, return false.

The method will check if the input is part of any of the results or if the result is part of the input?

The other method to process the results in a list and rank them accordingly.

A for loop is then created to make sure models go through all the sequences in the dataset.

On a given model, we first get the similarity using the previously mentioned method and store it in an array dedicated for the model itself. We then check whether the sequence is relevant in order to increment a variable responsible of True Positives and False Negatives. The sequence, the similarity and whether it is relevant are all stored in an array respectively.

The results of each model are then added onto a dedicated arraylist in the form of a SearchResult object while checking if the result is a true positive or false positive and incrementing the variables accordingly. The result should also have a similarity higher than a certain threshold set by the user. These arraylists are then printed on the screen for the user to see.

Now that we have the values of TP, FN and FP we can now calculate F-value and the mean average precision following their formulas. Then using FN, and the result sequence we calculate the precision recall of each result sequence and store them in an array for them to be printed as a graph to the user. These values are all stored in a static way in order to be accessed by the controller eventually

4. Experimental evaluation

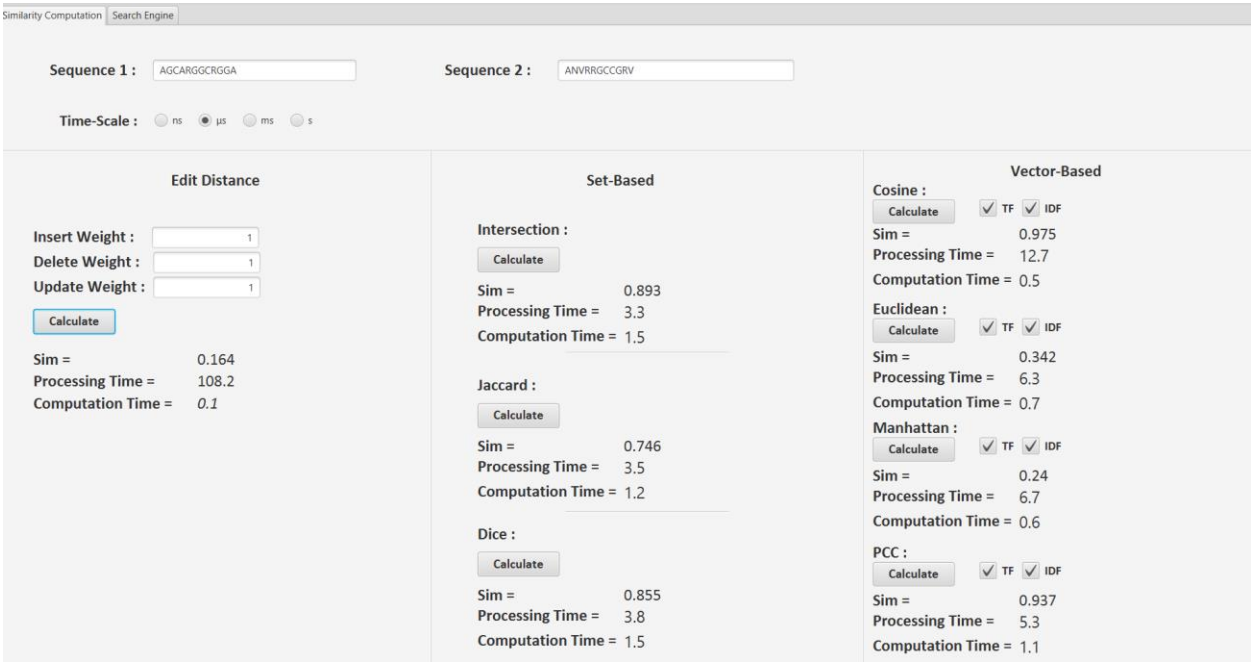
Two pillars are essential to prove the dependability of our program: on the one hand, efficiency is vital for data retrieval; it is essential for our program to perform its computation in a speedy manner.
On the other hand, accuracy is key to obtain precise results considering that RNA sequence comparison requires a great deal of attention.

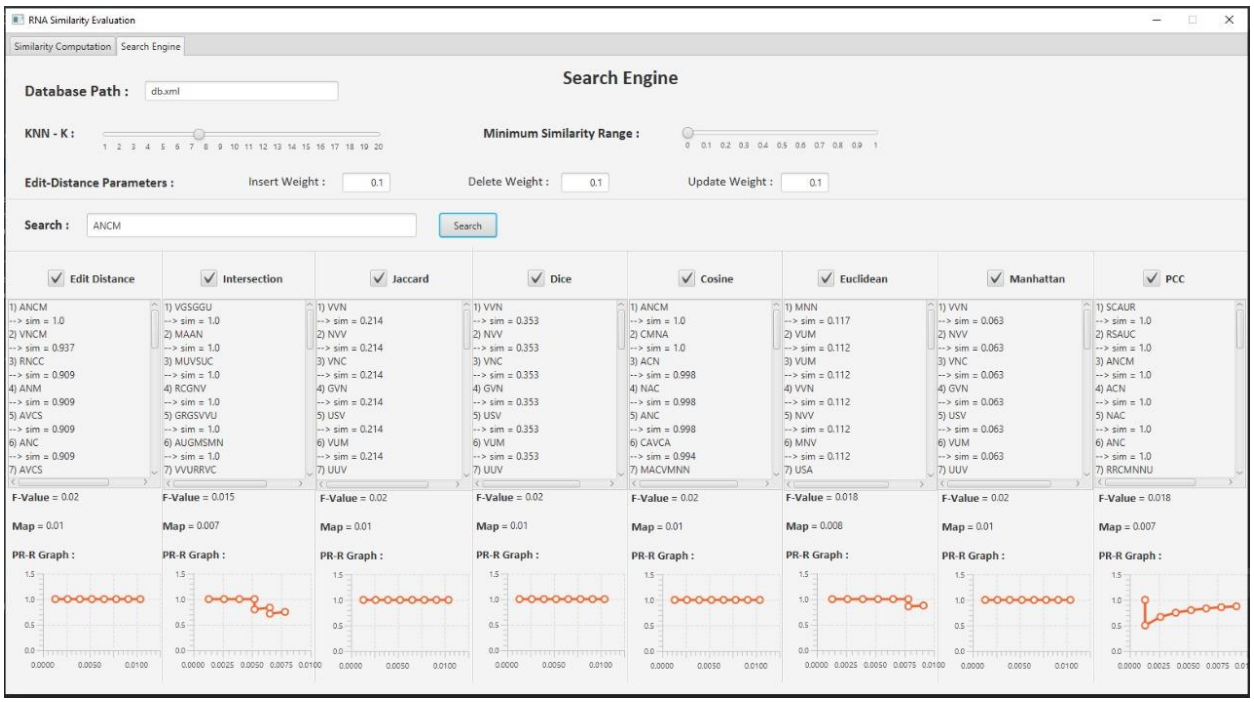
By checking the computation and processing time of all the implemented similarity calculators we notice a very fast working program with a worst-case scenario of 30us compared to the previously computed ED that has a processing time of at least 90us.

Our program also proves to be very accurate as we can notice from multiple tests using different sequences.

5. Personal Experience

We tried to run this code several times on several machines and it worked fine with minimal running and computing time, we tried several outputs, and the similarity features were giving totally good results. Overall, this project served us well in getting a hands on experience of the similarity models we encountered in class. It also proved challenging at times and provided us with the necessary tools to work with Intelligent Data Processing and Algorithms.





6. References

Lectures provided by Dr. Joe Tekli