MINIMUM SPANNING TREES AS AN ALTERNATIVE TO GUSFIELD'S GUIDE TREE

LUIS RODRÍGUEZ RUBIO, AU703347
SAM MARTIN VARGAS GIAGNOCAVO, AU703393

PROJECT IN BIOINFORMATICS - 10 ECTS
January 2023
Advisor: Christian Storm Pederson

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

MINIMUM SPANNING TREES AS AN ALTERNATIVE TO GUSFIELD'S GUIDE TREE

LUIS, SAM

Project in Bioinformatics - 10 ECTS

Bioinformatics Research Centre
Faculty of Natural Sciences
Aarhus University

January 2023

1 INTRODUCTION

Multiple sequence alignment remains one of the most cost intensive tasks in bio-informatics. Finding the optimal score for multiple sequences can prove challenging even in scenarios involving a reduced number of short sequences.

In order to find the optimal solution, as in any naïve implementation, the algorithm has to check for all of the possible combinations at a certain position in the alignment, thus considerably increasing the complexity of the problem for each evaluated sequence in the final alignment.

In addition, depending on how the algorithm defines the cost (or score) to be, the complexity of the algorithm will build up to an even greater extent. Different strategies exist, such as the *sum-of-pairs* score, which computes the score of the entire alignment by computing the score for each of the columns (see *Equation 1*). Unfortunately, it can prove problematic in certain situations where one sequence - or a small number sequences - skews the score in an otherwise perfect alignment. Other scoring functions like the *consensus score* or *tree score* exist but only as an alternative to the widely used *sum-of-pairs* score used in most of the multiple-sequence-alignment workloads [**fu-berlin**].

$$\text{SP Score} = \sum_{c=0}^{n} \sum_{i=0}^{k} \sum_{\substack{j=0 \\ j>i}}^{k} g_c(i,j) \tag{1}$$

where:

n = Length of the alignment (number of columns)

k = Number of sequences in the alignment

g_c = Cost function at specific column with sequence i and j

As for what concerns the project, *sum-of-pairs* will be the chosen cost function since the goal is to improve and build upon the *Gusfields* algorithm, which uses the aforementioned function. As a result, all references to scoring functions or optimal scores throughout the document will be referring or resulting from the *sum-of-pairs* score.

Having specified the scoring methodology that will be used, a comprehensive definition of the naïve algorithm can be made.

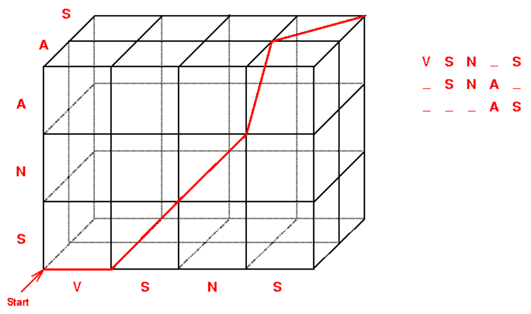


Figure 1: Optimal path of three sequences in a hyperlattice representation. Image extracted from bioinfo3d.cs.tau.ac.il.

Say the optimal alignment for k sequences of n length each needs to be found, since the *sum-of-pairs* score is a column-based scoring function, all of the different possibilities need to be evaluated (which in this case is 2, one for a new gap and another for the current character).

The *sum-of-pairs* score is additive, so the algorithm needs to keep track of whatever was scored previously - the dynamic programming approach (see *Figure 1* for a sketch of the process). This results on an algorithm that needs to fill out n^k cells each representing a column, trying $2^k - 1$ combinations[1] for each column and evaluating each combination with $k^2 - 1$ comparisons[2]. The total running time amounts to a massive $O(n^k * (2^k - 1) * (k^2 - 1))$ with a space consumption of $O(n^k)$ in order to store the values.

Consequently, methods that use heuristics and offer near-optimal results receive the utmost attention in multiple sequencing workloads. Multiple sequence alignment is an NP-complete problem after all, to which no polynomial time solution exists [**cornell-msa**].

Amongst powerful solutions such as ClustalW - a progressive algorithm - we find the Gusfields 2-approximation algorithm: A simple algorithm which guarantees an alignment whose score is, at most, two times that of the optimal one [**fu-berlin**] [**approx2**].

The algorithm achieves this by choosing an order in which paired alignments of sequences are merged together to form the complete alignment. The order depends on finding a “center sequence", the sequence the hast the least aggregate distance amongst all of the sequences we wish to align.

The $2 * \text{OPTIMAL}$ bound for the approximation algorithm can be demonstrated as follows.

1 A column can’t be completely made up of gaps, therefore one combination has to be subtracted from the total.

2 This assumes $\text{cost}(A, B) = \text{cost}(B, A)$.

First we need to find an upper bound of the score for the computed alignment. In order to get the score we will use *sum-of-pairs*, as we mentioned earlier. Then we will combine this upper bound (see *Equation* 2) with the lower bound of the score for the optimal alignment (see *Equation* 3). The factor resulting of dividing them will be our approximation.

$$
\begin{aligned}
SP(M) = {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} Score(M(S_i, S_j)) \\
= {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} d(i,j) \\
\leqslant {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} (d(i,1) + d(1,j)) \\
= {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} (d(1,i) + d(1,j)) \\
= {} & \frac{1}{2}\sum_{l=2}^{k} 2(k-1)d(1,l) \\
= {} & (k-1)\sum_{l=2}^{k} Score(M(S_1, S_l)) \\
= {} & (k-1)\sum_{l=2}^{k} D(S_1, S_l)
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
SP(M^*) = {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} Score(M^*(S_i, S_j)) \\
= {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} d^*(i,j) \\
\geqslant {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} D(S_i, S_j) \\
= {} & \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} D(S_1, S_j) \\
= {} & \frac{1}{2}k\sum_{j=1}^{k} D(S_1, S_j) \\
= {} & \frac{1}{2}k\sum_{l=2}^{k} D(S_1, S_l)
\end{aligned} \tag{3}
$$

where:

$$
\begin{aligned}
\mathrm{SP}(M) &= \text{Sum of Pairs score for MSA in matrix M.}\\
M(S_i, S_j) &= \text{Induced alignment of sequences i and j present in a MSA matrix M.}\\
\mathrm{Score}() &= \text{Score of a pairwise alignment.}\\
d(i,j) &= \text{Score(M(}S_i, S_j) rewritten.\\
D(S_i, S_j) &= \text{Optimal alignment between sequences i and j.}
\end{aligned}
$$

And by diving them between each other we get the inequality which amounts to 2 as seen in *Equation 4* [**fu-berlin**].

$$\frac{\mathrm{SP}(M)}{\mathrm{SP}(M^*)} \leqslant \frac{(k-1)\sum_{l=2}^{k} D(S_1, S_l)}{\frac{1}{2}k\sum_{l=2}^{k} D(S_1, S_l)} = \frac{2(k-1)}{k} < 2 \tag{4}$$

In this project, we will explore improvements to Gusfield's merging order by using a minimum spanning tree algorithm. From the bio-informatics standpoint, the goal is to see if the most optimal graph of sequences - which is generated by the minimum spanning tree algorithm - is suitable and offers an improvement over the “stock" merging order. And from the Computer Science standpoint, the goal will be to develop a solution that works both locally and in a website.

2 ARCHITECTURE & IMPLEMENTATION

2.1 THE CORE IMPLEMENTATION

As mentioned before, our implementation tries to explore the result of using Minimal Spanning Trees as "guide trees" when calculating an approximation for a Multiple Sequence Alignment. More precisely we will test if Kruskal's algorithm [**Kruskal1956**] can be used in different ways to improve the results from Gusfield's approach.

2.1.1 *Gusfield's algorithm & Minimal Spanning Trees*

Both Gusfield's approach and our Minimal Spanning Trees implementation start from a common initial procedure which consist of calculating all of the pairwise alignments between every sequence in our set.

This can be understood as calculating every edge of an adjacency matrix of a fully connected graph where every node is a sequence we want to align as we can observe in *Figure 2*.

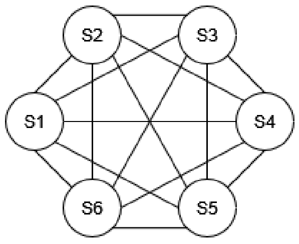


Figure 2: Fully connected graph were each node represent a unique sequence

This is the most computationally intensive section of the execution with a time complexity of $O(k^2 \cdot n^2)$ being n the number of sequences we want to ailing and k their length.

After this part what both Gusfield and our Minimal Spaning Trees implementation do is find a tree within this graph to guide the Multiple Sequence Alignment. And the shape and methodology of creating this tree is the difference that we are going to explore.

In Gusfield's algorithm this guide tree is obtained first by finding the “center string" of our adjacency matrix as mentioned in the 2-factor proof from the introduction. *Equation 5* shows the process.

$$\begin{gathered}\text{Find } S_1 \text{ such that } \sum_{S \in F - S_1} \mathrm{Score}(S_1, S) \text{ is minimized} \\ \text{Call the remaining strings } S_2, S_3, ..., S_k\end{gathered} \tag{5}$$

Once the center string has been found, we will connect every other sequence with this center string as we can observe in *Figure 3*.

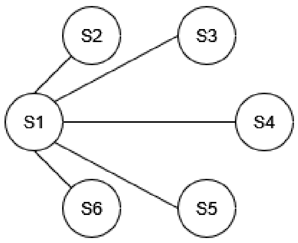


Figure 3: Tree obtained from joining every sequence to the Center String S1

On the other hand, in our Minimal Spanning Trees approach we will build the tree following Kruskal's algorithm. The use of a MST as a guide tree should provide more information about the relationships between all the sequences in the alignment, hopefully allowing for a more accurate alignment.

In comparison, a star-shaped tree, where the center string is the root and all other sequences are connected directly to the root, only considers the relationship between the center string and the other sequences and does not take into account the relationship between the other sequences.

A MST is a tree that connects all the sequences in the alignment and has the minimum possible total edge weight. In this case our MST will be constructed using a pairwise distance matrix of the sequences, where the distance between two sequences is a measure of their similarity.

It's important to note that the use of a MST as a guide tree is not commonly used for MSA, as it doesn't have a theoretical guarantee for the alignment quality. Other alternatives such as progressive methods are more widely used.

A summary of Kruskal's algorithm to build a MST can be seen in *pseudocode 1* [**cormen2022introduction**]. One possible tree shape that we can obtain from Kruskal can be the one observed in *Figure 4*. Any resulting tree from the algorithm will not contain cycles.

From this point on, both algorithms follow their respective "guide tree" and add sequences one by one to the final alignment. In this case, the merging order in Kruskal would be determined by the the same merging order followed in *pseudocode 1*.

Algorithm 1 Kruskal's algorithm

```
1: Order every edge on the fully connected graph by their weights.
2: Edges are represented by the nodes they are connected to (u,v).
3: for $(u, v) inordered_e dges$ do
4:     if $(u, v) doesnotmakeacyclewhenaddingittothetree$ then
5:         Add (u,v) to T.
6:     end if
7: end for
```

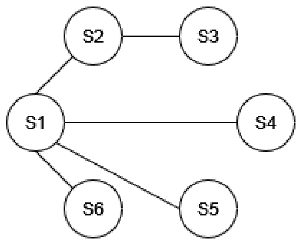


Figure 4: Possible tree shape that can be obtained from Kruskal's algorithm

This can be very easy to implement on the Gusfield's approach as every node is connected to the same one [**Gusfield_1997**]. This detail makes that can add each sequence one by one in any order starting with the "center string". We can do this because the tree shape is going to follow this same characteristics every time, but we can not say the same about the Minimal Spanning Trees alternative.

With Minimal Spanning trees we wont always have the same tree shape. To deal with this problem of trying to find which node should we add at each step we modified Gusfield's implementation.

The matrix representing the multiple sequence alignment will be extended by merging matrix instead of only a single sequence to it.

Even though it is not part of our focus to optimize the execution, we implemented the Disjoint Set Union algorithm [**disjointsetunion**] to build Kruskal's MST. This apart from being one of the fastest implementations is also very helpful with our goal of mixing sets of sequences together. Its main concept can be seen in *pseudocode* 2.

Here, u & v are representing sequences, each one of them can be already in a bigger set which will be merged together.

Although computational complexity is not the primary focus of this implementation, it is worth noting that it does not significantly impact the running time. The Kruskal alternative may take slightly longer than Gusfield's method ($O(e \log V)$, V being the number of vertices), but this difference is negligible. This is because the initial step of calculating the pairwise distance matrix is the most computationally intensive part of the process. Therefore, the time complexity remains the same for both Gusfield and our Kruskal alternatives,

Algorithm 2 Kruskal's Disjoint Set Union

```
1: Initially make |V| single node sets
2: Edges are represented by the nodes they are connected to (u,v).
3: for $(u, v) in ordered_e dges$ do
4:     $S_u$ = set containing u
5:     $S_v$ = set containing v
6:     if $S_u$ is not the same as $S_v$ then
7:         add (u,v) to T
8:     end if
9: end for
```

which is $O(k^2 \cdot n^2)$ where n is the number of sequences to be aligned and k is their length.

One key thing to note is the non-deterministic nature of our implementation in scenarios where sequences are reordered within the FASTA file. Currently, the adjacency matrix used for our Kruskal implementation computes all alignments as found in the FASTA file. This means that if a FASTA file contains sequences S1, S2 and S3 we will compare the sequences in the following order: S1 with S2, S1 with S3 and S2 with S3. However, if a variation of the FASTA file rearranges the sequences to S3, S2 and S1, the comparison will be the following: S3 with S2, S3 with S1 and S2 with S1. While distance is symmetrical, two sequences might have multiple alignments that share the same score. In our implementation there might be instances where depending on the arrangement of sequences in the FASTA files, different alignments will be used to merge the different clusters, resulting in a different final cluster and, consequently, score.

2.1.2 *Rust*

We decided to implement this algorithm using the Rust programming language. We did this because even though performance was not the primary focus we are interested in fast calculations and processing of the data to make the tool as usable as possible.

We also chose Rust instead other fast-performing languages because the reliability and memory safety as well as provide a WebAssembly package which we used to create our visualization web application.

This has implied that along the duration of this project we learnt a new programming language with which we did not had any prior experience, and also learn how Web Assembly worked and how to include it in a web application.

During the development, our focus was on reusing as much code as possible for our two implementations, one as a CLI using Rust exclusively, and the other one calling Rust functions directly from JavaScript via WebAssembly.

We also focused on robustness and reliability working with error handling and edge cases along the way. We used GitHub's CD/CI capabilities to keep these principles in mind for every change committed to the repository (more insight into the configuration can be checked in *section A.1*). This required the project to be able to build and pass every unit test with each addition we made. Apart for unit testing we also added a benchmark framework to monitor and evaluate our running sequence. More information regarding the unit test and benchmarking features can be found on *Chapter 3*.

2.2 WEBASSEMBLY, OUR VESSEL TO THE WEB

As previously mentioned, we needed a programming language that would be flexible enough to allow for both a CLI and Web execution of the algorithm. Some options that were initially discussed included building an API and rewriting key sections of the CLI code in JavaScript. These were quickly discarded as WebAssembly would allow us to reuse a great part of the codebase.

WebAssembly acts as an intermediary between a number of supported programming languages (e.g. Python, C++, Rust, Go, etc.) and JavaScript by converting the original code into byte-code that can be run by most browsers used at the time of writing [**mdn-wasm**].

Converting the original code into the byte-code that needs to be called from JavaScript requires tools that are language-dependant. In the case of Rust, *wasm-pack* is the tool to use.

Not all of the code is compiled into WebAssembly. The user has to be specify which functions have to be compiled by listing them in the *lib.rs*[1] file and using a macro. An example can be seen in the following code listing:

```
#[wasm_bindgen]
pub fn wasm_serialize_fasta_string(fasta_string: String) ->
    JsValue {
    let sequences = parse_fasta_string(fasta_string);
    serde_wasm_bindgen::to_value(&sequences).unwrap()
}
```

In it's current state, WebAssembly in Rust does not offer a “smooth" transition in most cases. Notice the use of the `serde_wasm_bindgen` function.

1 In our project we have renamed the file to *wasm.rs* in order to reflect the main use of the file.

One of the limitations that is still present at the time of writing the report is the lack of any native option to return vectors storing custom *structs* to JavaScript. Some workarounds, such as using a *&[u8]* type intermediary exist but are only valid for “simpler" *structures* that can be easily converted and parsed from an array of bytes. In our case, we make use of multiple nested structures so an alternative was needed.

As listed in the code above, Serde was the solution we decided to use. In this case, the workaround implemented from the library is using a JSON string as an intermediary. The problem in this case would be the need for the bare JavaScript code to parse the incoming arguments, adding an additional overhead to the web client. However, after brief tests we considered this a minor issue. If results from the WASM functions were too big to be parsed, other aspects of the website would also be harmed, regardless of the methodology for passing data between the WebAssembly and JavaScript code.

One of the other interesting alternatives that were explored included sharing a memory buffer between the Rust code - which ends up being converted to WebAssembly - and JavaScript code at all times. In the end, this option was discarded due to its complexity.

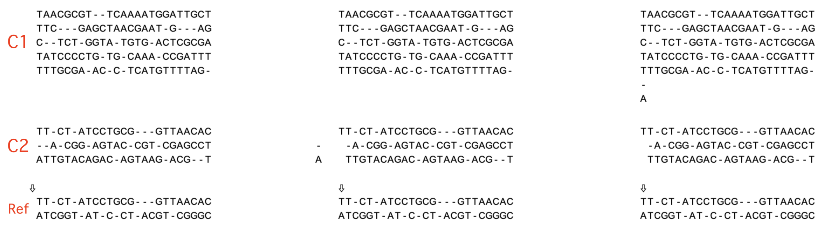
For the final version of the website all of the alignments were done in WebAssembly and parsed and displayed in JavaScript.

2.3 VISUALIZING ALIGNMENTS

With the ever-expanding state of the web we thought it would be interesting to take advantage of the flexibility and accessibility of the available programming tools and frameworks to display the process and steps of the alignment, something that is not easily done with popular languages in the desktop space such as C++, C# and others.

Since we would be taking on a space - canvas visualization on the web - that we had no prior experience with, this part of development ended up being prone to changes.

Initial designs and sketches for the website included the visualization of the guide tree with a live step by step display of the cluster alignments. After an initial review of the available JavaScript visualization libraries we decided to prioritize the step by step visualization of the alignments over displaying the graph. Unless we opted for a bare bones implementation there would not be a general framework or library that would easily allow us to implement both features, thus including both would require two different JavaScript libraries which would increase the complexity and maintainability of our solution. The chosen library for the implementation would end up being CreateJS, a suite of inter-compatible JavaScript libraries.



```
   TAACGCGT--TCAAAATGGATTGCT
   TTC---GAGCTAACGAAT-G---AG
C1 C--TCT-GGTA-TGTG-ACTCGCGA
   TATCCCCTG-TG-CAAA-CCGATTT
   TTTGCGA-AC-C-TCATGTTTTAG-

   TT-CT-ATCCTGCG---GTTAACAC
C2 --A-CGG-AGTAC-CGT-CGAGCCT
   ATTGTACAGAC-AGTAAG-ACG--T

  ⇩
    TT-CT-ATCCTGCG---GTTAACAC
Ref ATCGGT-AT-C-CT-ACGT-CGGGC
```

```
   TAACGCGT--TCAAAATGGATTGCT
   TTC---GAGCTAACGAAT-G---AG
   C--TCT-GGTA-TGTG-ACTCGCGA
   TATCCCCTG-TG-CAAA-CCGATTT
   TTTGCGA-AC-C-TCATGTTTTAG-

   TT-CT-ATCCTGCG---GTTAACAC
-   -A-CGG-AGTAC-CGT-CGAGCCT
A   TTGTACAGAC-AGTAAG-ACG--T

   ⇩
   TT-CT-ATCCTGCG---GTTAACAC
   ATCGGT-AT-C-CT-ACGT-CGGGC
```

```
TAACGCGT--TCAAAATGGATTGCT
TTC---GAGCTAACGAAT-G---AG
C--TCT-GGTA-TGTG-ACTCGCGA
TATCCCCTG-TG-CAAA-CCGATTT
TTTGCGA-AC-C-TCATGTTTTAG-
-
A

TT-CT-ATCCTGCG---GTTAACAC
 -A-CGG-AGTAC-CGT-CGAGCCT
 TTGTACAGAC-AGTAAG-ACG--T

⇩
TT-CT-ATCCTGCG---GTTAACAC
ATCGGT-AT-C-CT-ACGT-CGGGC
```

Figure 5: Character-based steps animated in our first prototypes.

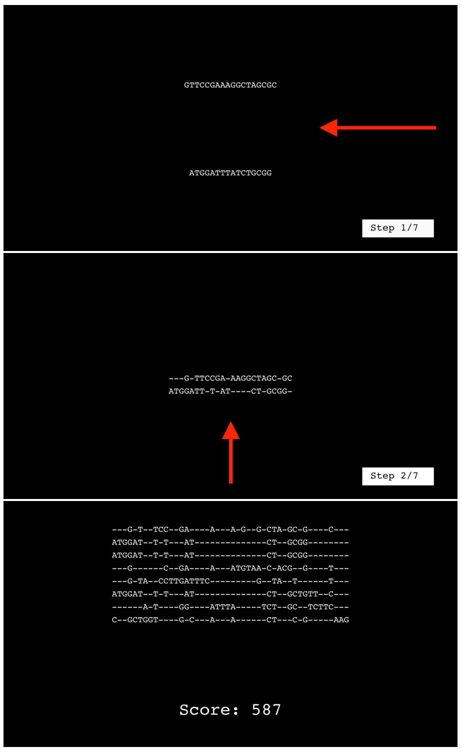
Due to the design choices that were made at the initial stages of the project - e.g. working with the merge-order vectors - development of the user interface did not require having the command-line client finished. Early prototypes of the visualization can be seen in *Figure 5*. Here we can see the three main stages the visualization goes through in order to complete a character-based alignment visualization. From left to right, the visualization displays two clusters that are joining together and a reference alignment used to guide the merging process (left-most image), an animation that starts moving parts of one cluster to another (center image) and finally the end result of the operation for that specific position. Note that *Figure 5* shows an early prototype so the reference alignment used to guide the merging process is completely random and serves as a placeholder.

As development of the Rust back-end continued, problems concerning the performance of the visualization were a concern. The initial approach that was taken, as suggested in *Figure 5* was to have each character as a unit. The issue with this approach lies on the capabilities of the library to manage thousands of different "sprites"[2] at the same time, which in this case was not possible. Some optimizations, such as the removal of off-screen visualizations were attempted but even then, performance was sub-optimal.

Consequently, a simpler approach was taken which involved simply displaying and animating the three clusters from each step. At the beginning of each step the two clusters that are yet to be merged swipe from the right and after that the result swipes from the bottom. This process repeats itself until the visualization reaches the end and displays the resulting cluster with the sum-of-pairs score. *Figure 6* shows the three main events in order.

The website allows the user to configure all of the alignment parameters available in the command line program. Users can both, paste and select files containing sequences and cost matrices as seen in *Figure 7*. Once all of the data has been loaded and parameters have been set, users can click on the "PLAY" button to start the visualization. A slower alternative to the continuous visualization can be triggered by using the "STEP" button, which requires an interaction from the user

2 Term used to describe images that represent an asset.



```
GTTCCGAAAGGCTAGCGC

ATGGATTTATCTGCGG
```

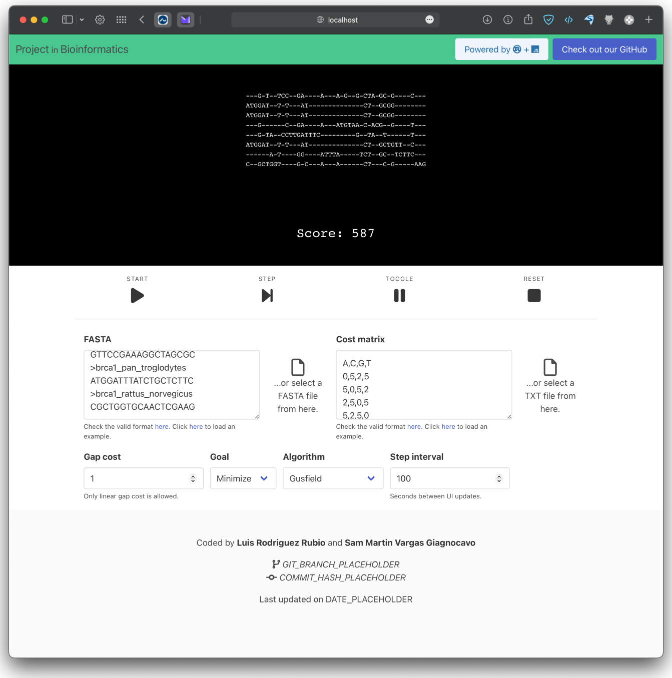
```
---G-TTCCGA-AAGGCTAGC-GC
ATGGATT-T-AT----CT-GCGG-
```

```
---G-T--TCC--GA----A---A-G--G-CTA-GC-G----C---
ATGGAT--T-T---AT--------------CT--GCGG--------
ATGGAT--T-T---AT--------------CT--GCGG--------
---G------C--GA----A---ATGTAA-C-ACG--G----T---
---G-TA--CCTTGATTTC---------G--TA--T------T---
ATGGAT--T-T---AT--------------CT--GCTGTT--C---
------A-T----GG----ATTTA-----TCT--GC--TCTTC---
C--GCTGGT----G-C---A---A------CT---C-G-----AAG

Score: 587
```

Figure 6: Final visualization. As opposed to the initial prototypes, which manipulated individual characters, this only animates and moves clusters.

per step. "TOGGLE" and "RESET" will toggle the on-going animation and remove all images from the canvas respectively.



```
---G-T--TCC--GA----A---A-G--G-CTA-GC-G----C---
ATGGAT--T-T---AT--------------CT--GCGG--------
ATGGAT--T-T---AT--------------CT--GCGG--------
---G------C--GA----A---ATGTAA-C-ACG--G----T---
---G-TA--CCTTGATTTC---------G--TA--T------T---
ATGGAT--T-T---AT--------------CT--GCTGTT--C---
------A-T----GG----ATTTA-----TCT--GC--TCTTC---
C--GCTGGT----G-C---A---A------CT---C-G-----AAG
```

Score: 587

FASTA

```
GTTCCGAAAGGCTAGCGC
>brca1_pan_troglodytes
ATGGATTTATCTGCTCTTC
>brca1_rattus_norvegicus
CGCTGGTGCAACTCGAAG
```

...or select a FASTA file from here.

Check the valid format here. Click here to load an example.

Cost matrix

```
A,C,G,T
0,5,2,5
5,0,5,2
2,5,0,5
5,2,5,0
```

...or select a TXT file from here.

Check the valid format here. Click here to load an example.

Gap cost: 1

Only linear gap cost is allowed.

Goal: Minimize

Algorithm: Gusfield

Step interval: 100

Seconds between UI updates.

Coded by **Luis Rodriguez Rubio** and **Sam Martin Vargas Giagnocavo**

GIT_BRANCH_PLACEHOLDER
COMMIT_HASH_PLACEHOLDER

Last updated on DATE_PLACEHOLDER

Figure 7: Overview of the final website.

3 EVALUATION & CONCLUSION

Evaluation and testing was thought of from the initial stages of development. *cargo*, Rust's package manager, allows for benchmarking[1] and unit testing[2] to be included with ease in any Rust project. In our project, two folders, *benchmarks* and *tests*, have been added to the *src* folder.

To test the implementation, two "types" of test data were used: Sequenced data and completely randomized data. Amongst the sequenced data two sets were tested, a collection of chromosomes from *mus musculus'* reference genome and sequences from different SARS-CoV-2 variants. In addition to the completely randomized scenario, we wanted to test the algorithm's capability in two "realistic" scenarios where we can expect considerably different rates of divergence between sequences.

For each iteration of our tests we decided to use a fairly simple score matrix. The reasoning for not using other popular score matrices such as BLOSUM or PAM is mostly due to our lack of experience with most of the well-established matrices and resources such as [**pearson2013selecting**], which argue that DNA sequence comparison is much less sensitive than protein sequence comparison.

As for variations regarding the algorithm, two additions have been added specifically for the testing phase. In one, instead of using the default merge-order vector from our Kruskal implementation we invert the steps, this time starting from the end. In the other variation, we randomize the merging steps. The goal is to further explore the effect of the merging phase has in our score.

Before going through the final results of our comparison, we briefly explain how to compile, set up and test our solution. The code is hosted at `https://github.com/Lewis-11/PiB`. Users can either clone the git repository locally or download a ZIP file containing the source code. Within the root of the project, *cargo build --release* - which has to be run from the command line prompt - will generate an executable binary inside the *target/release* folder. As a requirement, users should have both, the Rust compiler and Cargo installed. An execution of the program looks as follows: *./target/release/msa mst –submat mat.txt sequences.fa*. For more info regarding the different options that can be specified, run *./target/release/msa help*.

1 https://doc.rust-lang.org/cargo/commands/cargo-bench.html
2 https://doc.rust-lang.org/cargo/commands/cargo-test.html

In order to verify that all unit tests are working as expected, users can run *cargo test* from the root of the project. The tool will output in the command line the status of each of the tests.

While performance was not a priority throughout development since we had to adapt our code for two different platforms, the *cargo bench* command is available and will generate a detailed report comparing running times between the different algorithms. See *section A.2* for more information.

Users that in addition want to deploy the web interface, need to install and run the *wasm-pack build* command from the command prompt. Once the WASM files are generated, the user can go inside the *www* folder and run *npm start* to start a local instance of the website at *localhost:8080*. However, in the repository's *README.md* file there is already a link to an external website that hosts all of the active git branches[3]. Now that we have introduced the setup we proceed to talk more extensively about our results.

3.1 SCORING COMPARISON

In accordance with the previously discussed methodology, we recorded the score of utilizing four distinct guide tree configurations. These configurations included a star-shaped tree generated via Gusfield's “center string" algorithm, a minimum spanning tree (MST) constructed utilizing the Disjoint Set Union implementation of Kruskal's algorithm, and two variations of the aforementioned MST. The variations comprised of reversing the order of merges, beginning with the most dissimilar sequences within the MST, and the other variation involved randomly shuffling the order of the merges.

We have evaluated each of the four implementations with the three data sets mentioned in this chapter's introduction. The “real-world" FASTAS range from sequences of relatively small length (100-600 characters) to larger sequences of 30,000 and 23,000 characters. With regards to the artificially generated sequences, we have examined groups of 5 and 10 sequences of lengths 1000, 5000, and 10,000.

We first compared the score ratio of our implementation of Kruskal's algorithm with Gusfield's score. A lower ratio indicates an improvement in Gusfield's score. We calculated the average score ratio for each test by combining the lengths of the sequences. As shown in *Figure 8*, the results were mostly better, with an average ratio of 0.67, which translates to a 33% improvement. It's important to keep in mind that this comparison is based on our observations and there's yet to be proof that MSTs lead to a higher alignment quality.

3 A personal server is used so there is no guarantee that the website will be operational at all times.

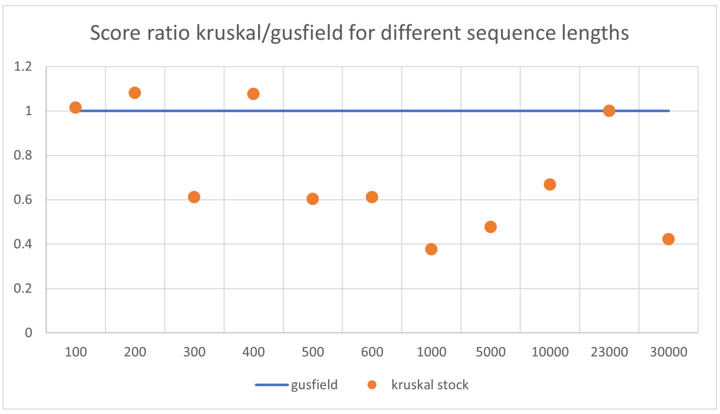


Figure 8: Score ratio between Gusfield's "center string" & Kruskal's MST as guide tree for different sequence lengths.

We also evaluated various MST alternatives that modify the order in which sequences are merged following the MST. The standard approach merges sequences with smaller scores (more similar sequences) first, while the reversed version starts with sequences with higher scores (more dissimilar sequences), and the last one is a random order. To determine if these alternatives show an improvement over the standard MST (lower scores first), we calculated a score ratio between the alternatives and the standard MST, as we did before. The results can be seen in *Figure* 9.

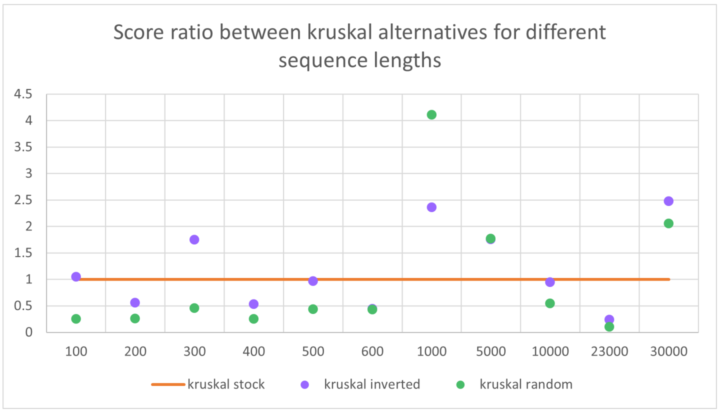


Figure 9: Score ratio between Kruskal's MST alternatives for different sequence lengths.

The average ratio for the inverted order is 1.25 and for the random order is 1.06, which represents a 25% and 6% decrease in performance (since we aim for a low ratio to minimize scores). Thus, based on our observations, the standard MST order is the one that yields the best results on average.

Based on our observations, it is notable that the differences between using Gusfield's star-tree and Kruskal's MST become more apparent when the number of sequences (k) is larger. This can be observed by comparing *Figure 10*, which shows results for 5 sequences, with *Figure 11*, which shows results for different sequence lengths. This trend can also be observed in real-world data, such as the comparison of different mouse sequences in *Figure 13*, which have 5 sequences, versus the results obtained from aligning 9 sequences of COVID variants in *Figure 12*, where we can see a more significant improvement from Kruskal's alternative when compared to Gusfield's method.

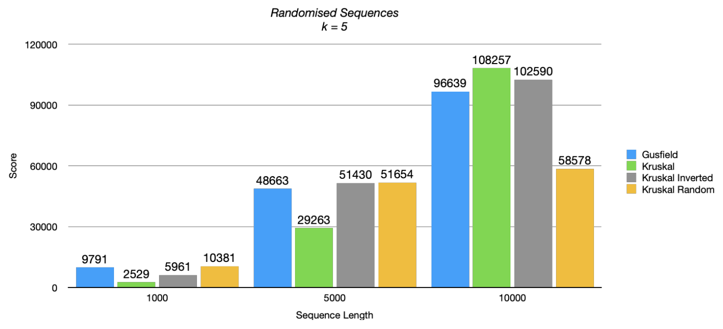


Figure 10: Scores from aligning 5 randomized sequences of multiple sizes.

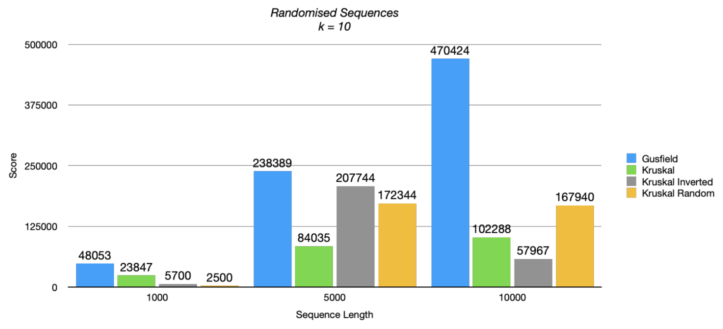


Figure 11: Scores from aligning 10 randomized sequences of multiple sizes.

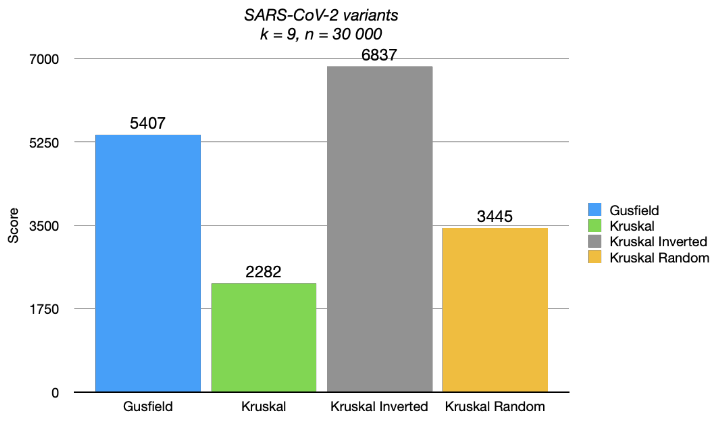


Figure 12: Scores from aligning 9 SARS-CoV-2 variants.

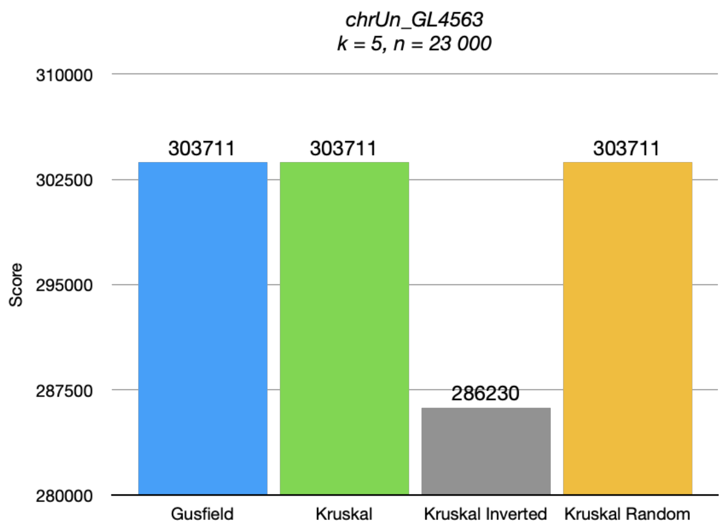


Figure 13: Scores from aligning 5 sequences from the *mus musculus* reference genome.

This can be attributed to the fact that an increase in the number of sequences leads to an increase in the number of nodes in the adjacency matrix of alignments. This also means that there are more potential trees. This has been demonstrated in *Figure 13*, with the mouse dataset, Gusfield, standard Kruskal, and random Kruskal all obtained the same score. This was due to the fact that the guide tree was similar in every case. In cases where there are very few sequences, random order of kruskals MST yields the best results as can be seen in *Figure 14*.

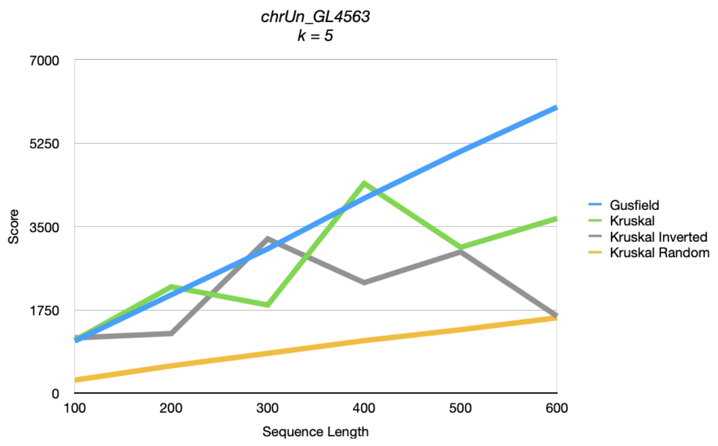


Figure 14: Scores from aligning 5 differently-sized sequences from the *mus musculus* reference genome.

As a final note, in *chapter 2* we mentioned the increased running time of adding Kruskal's algorithm on top of Gusfield's. *Figure 15* clearly shows that in theory there is some overhead most of the processing is done aligning both sequences and clusters. The base Gusfield implementation and Gusfield + Kruskal implementation offer near identical running times.

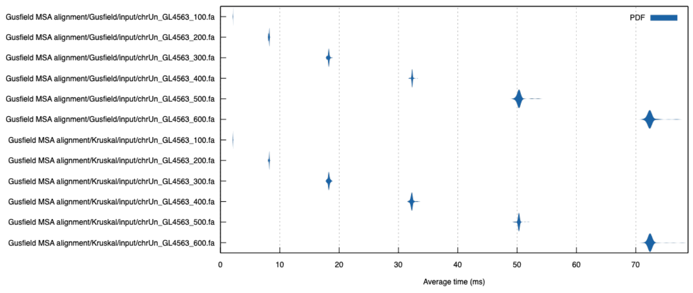


Figure 15: Running times from a series of tests comparing the base Gusfield implementation with the improved one.

3.2 Conclusion and Future Work

As described in the introduction, we were interested in the possibilities this project had to offer both from a bio-informatics and computer science standpoint. While the diverging results don't offer the clear conclusion we were hoping for, we still found the evaluation phase interesting as it showed how small changes can greatly change the end result. Further testing could shed some light on the effects not only the different variations of the Kruskal have but also what type of algorithm should be used depending on the type of sequences we are working with (e.g. extremely similar sequences might achieve a better alignment with a certain algorithm).

Regarding possible improvements if the project were to be continued, optimization and visualization could be the next major areas to upgrade. In multiple instances our Rust implementation stores redundant data, hence reducing the memory footprint would be ideal. Visualization-wise, the initial prototypes that were ideated in the early stages proved too ambitious for the scope of the project but would make the website that was developed for the project significantly more functional.

Another limitation, which is also present in the visualization but cascaded from our Rust implementation, is the lack of sequence identifiers in the resulting cluster.

Despite the limitations, we consider the project a great learning experience since we expanded our knowledge in areas that we didn't have an opportunity to expand on during courses like *Algorithms in Bioinformatics* and *Genome Scale Algorithms* (e.g. trying to optimize Gusfield's 2-approximation algorithm and the alignment of clusters as opposed to single sequences). In addition, from a more practical point of view, we learned to work with two relatively new tools such as Rust and WebAssembly. Both of which, we consider, will be useful in future projects.

A APPENDIX

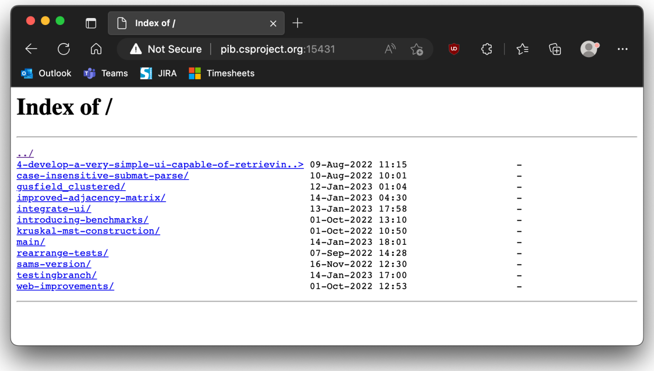
A.1 OUR GITHUB'S CD/CI CONFIGURATION

In order to quickly and automatically build, test and deploy our code we defined two *GitHub Action* jobs. Part of one can be seen in the code listing below. Here we can see how we install *wasm-pack*, build and convert our Rust code into WebAssembly and do some minor modifications to the website before sending the compiled version to a web server.

```
...

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Install wasm-pack
      run: curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf | sh
    - name: Generate pkg files
      run: wasm-pack build
    - name: Replace GIT_BRANCH_PLACEHOLDER, COMMIT_HASH_PLACEHOLDER and DATE_
         PLACEHOLDER
      run: |
        cd www
        sed -i 's/GIT_BRANCH_PLACEHOLDER/'"${GITHUB_REF##*/}"'/g' index.html
        sed -i 's/COMMIT_HASH_PLACEHOLDER/'"${GITHUB_SHA}"'/g' index.html
        sed -i 's/DATE_PLACEHOLDER/'"$(TZ=Europe/Copenhagen date)"'/g' index.html
    - name: Compile Webpack
      run: cd www && npm install && npm run build
    - name: Copy to server
      uses: easingthemes/ssh-deploy@main
      env:
        SSH_PRIVATE_KEY: ${{ secrets.RSA_KEY }}
        REMOTE_HOST: ${{ secrets.HOST }}
        SOURCE: "msa/www/dist/"
        REMOTE_USER: ${{ secrets.REMOTE_USER }}
        REMOTE_PORT: ${{ secrets.PORT }}
        TARGET: "builds/${GITHUB_REF##*/}"
```

The web server is set up in a way that allows the viewing of different branches. The idea behind this setup was to make testing as easy as possible. *Figure 16* shows the index page of the server.



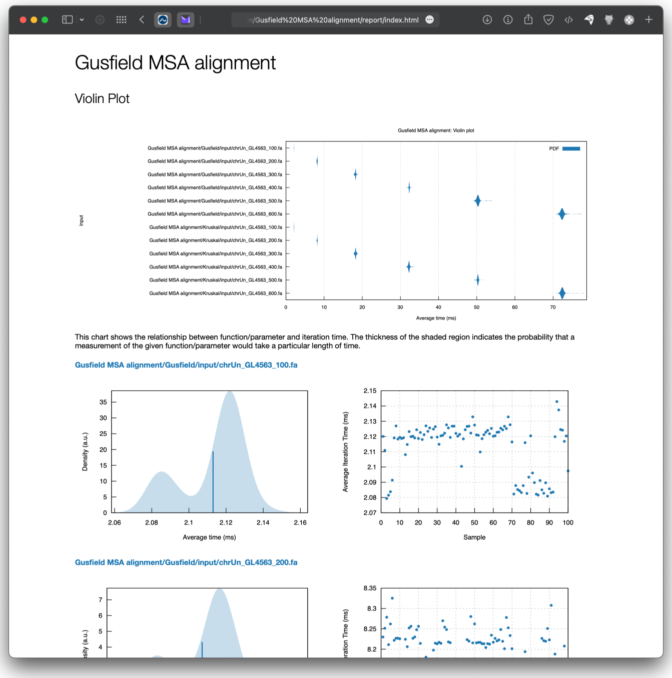
Index of /

```
../
4-develop-a-very-simple-ui-capable-of-retrievin..> 09-Aug-2022 11:15                   -
case-insensitive-submat-parse/                     10-Aug-2022 10:01                   -
gusfield_clustered/                                12-Jan-2023 01:04                   -
improved-adjacency-matrix/                         14-Jan-2023 04:30                   -
integrate-ui/                                      13-Jan-2023 17:58                   -
introducing-benchmarks/                            01-Oct-2022 13:10                   -
kruskal-mst-construction/                          01-Oct-2022 10:50                   -
main/                                              14-Jan-2023 18:01                   -
rearrange-tests/                                   07-Sep-2022 14:28                   -
sams-version/                                      16-Nov-2022 12:30                   -
testingbranch/                                     14-Jan-2023 17:00                   -
web-improvements/                                  01-Oct-2022 12:53                   -
```

Figure 16: Root of the custom server used for the development of the solution.

A.2 RUNNING TIMES GENERATED BY THE *criterion* LIBRARY

Running the *cargo bench* command will run all of the benchmarks defined within the *Cargo.toml* file. The *criterion* library has been used in order to automate most of the prior (e.g. a test warm up) and subsequent (e.g. computing the standard deviation) steps of the benchmarking process. An example of the generated HTML file can be seen in *Figure 17*. Locally, this file is located inside the *./target/criterion/Gusfield MSA alignment/report* folder.



n/Gusfield%20MSA%20alignment/report/index.html
Gusfield MSA alignment: Violin plot
Input
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_100.fa
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_200.fa
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_300.fa
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_400.fa
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_500.fa
Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_600.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_100.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_200.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_300.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_400.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_500.fa
Gusfield MSA alignment/Kruskal/input/chrUn_GL4563_600.fa
PDF
0 10 20 30 40 50 60 70
Average time (ms)
Density (a.u.)
35 30 25 20 15 10 5 0
2.06 2.08 2.1 2.12 2.14 2.16
Average time (ms)
Average Iteration Time (ms)
2.15 2.14 2.13 2.12 2.11 2.1 2.09 2.08 2.07
0 10 20 30 40 50 60 70 80 90 100
Sample
7 6 5 4
8.35 8.3 8.25 8.2

Gusfield MSA alignment

Violin Plot

This chart shows the relationship between function/parameter and iteration time. The thickness of the shaded region indicates the probability that a measurement of the given function/parameter would take a particular length of time.

Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_100.fa

Gusfield MSA alignment/Gusfield/input/chrUn_GL4563_200.fa

Figure 17: Generated output from *criterion*.