Comparison of multiple biological sequences

Projects in Bioinformatics, 10 ECTS
Astrid Christiansen, 201404423
Aarhus University
Supervisor: Christian Nørgaard Storm Pedersen

June 2022

Contents

1 Introduction

This project is about constructing *multiple sequence alignments* (*MSAs*) of biological sequences. A sequence alignment is a way of arranging sequences of biological data, such as DNA, RNA, or protein, in order to identify regions of similarity, which can help infer relationships between the sequences. Figure 1 is an example of a MSA of three DNA sequences. The gaps, "-", denote insertions and deletions. Different symbols in the same column denotes substitution.

$$M = \begin{bmatrix} A & C & C & A & C & C & A & G & - & G & T \\ G & C & - & A & - & C & T & - & A & - & T \\ A & C & C & - & - & A & T & G & A & G & G \end{bmatrix}$$

Figure 1: A multiple sequence alignment, M, of sequences $s_1 = ACCACCAGGT$, $s_2 = GCACTAT$, and $s_3 = ACCATGAGG$.

Given a set of sequences, there are many ways of arranging them, i.e., there are many possible alignments. However, not all of these alignments are equally relevant, as some represent more probable biological scenarios than others; so we want to be able to rank them. In this project, I use the so-called *sum-of-pairs score (SP score)* to achieve this. There might be multiple different alignments with the same optimal, i.e., minimum, SP score; we call these *optimal alignments*. An optimal alignment can be found using dynamic programming in time $O(n^k)$ for k sequences of length $O(n)$; i.e., for two sequences, this takes $O(n^2)$ time, for three sequences, it takes $O(n^3)$ time, etc. This quickly becomes computationally infeasible, even for short sequences; e.g., for 10 sequences of length 100, it would take many years to construct the MSA in this way. Typically, we are also interested in much longer sequences than this.

Therefore, we instead use approximation algorithms and heuristics that are not guaranteed to find an *optimal* alignment, but have better time complexities. One such approximation algorithm is *Gusfield's approximation algorithm*, which achieves a running time of $O(k^2n^2)$ and is guaranteed a score that is less that twice the score of an optimal alignment. Gusfield's algorithm uses a so-called *star tree* to construct an alignment with a score below this limit. In this project, I will explore a modified version of this algorithm, that uses a *minimum spanning tree* (*MST*) instead of the star tree used by Gusfield's algorithm. The modified algorithm is structurally similar to Gusfield's algorithm, and they have the same time complexity, but due to the different trees that they use, they will often result in different MSAs. The modified algorithm will be a heuristic for which we do not have a score bound. Therefore, it would be interesting to explore experimentally which algorithm constructs the best MSAs on different types of data. I have already implemented Gusfield's algorithm in a previous course, and in this project, I will implement the MST algorithm for constructing MSAs. I will conduct experiments in order to compare the two implemented algorithms, both with regard to running time and SP score. For these experiments, I will implement a simple model for simulating DNA sequence data to use as input to the two algorithms; thereby being able to run the algorithms on DNA data with different properties.

2 Sum-of-pairs score

As shown in Figure 1, we can represent a multiple alignment of k sequences, $s_1, ..., s_k \in \Sigma^*$, for a given alphabet, Σ, as a $(k \times \ell)$ matrix, M, where $M_{i,j} \in \Sigma \cup \{\text{-}\}$ [2]. M will thus consist of k rows, and the number of columns, ℓ, depends on the number of gaps in the alignment; there cannot be less than n or more than kn columns, however, since columns consisting only of gaps are not allowed. As explained, we want to be able to score alignments, such that we can rank them. To calculate such alignment scores, we use a so-called *gap cost* and *substitution matrix*. The gap cost is the cost of an insertion or deletion, i.e., the cost of a gap in the alignment. The substitution matrix, S, is a $(|\Sigma| \times |\Sigma|)$ matrix, where $S_{i,j}$ is the cost of substituting letter i with letter j. In the case of DNA data, we are interested in the alphabet $\Sigma = \{A, C, G, T\}$, so the substitution matrix is a (4×4) matrix.

If the alignment consists of only two sequences, then we calculate the score using the gap cost for insertions and deletions, and the substitution matrix for substitutions, directly on each column in the alignment; we then sum the score of each column to get the total score [1]. We call this the pairwise alignment score of M. In summary, we calculate the score of M for $k = 2$ as

$$SCORE(M) = \sum_{0 \le c < \ell} d(M_{0,c}, M_{1,c})$$

where ℓ is the number of columns in M, and d is given from the gap cost and substitution matrix.

We now want to extend this to $k > 2$. In this case, we can use d on every pair of entries in each column of M, sum these values, and then sum the column scores; we call this the *sum-of-pairs score* (*SP score*) of M. So we can calculate the SP score of M as

$$SP(M) = \sum_{0 \le c < \ell} \sum_{0 \le i < j < k} d(M_{i,c}, M_{j,c}).$$

Note, that for $k = 2$, $SP(M) = SCORE(M)$, but in this case, we do not have to sum over pairs of entries, since we only have two entries in each column.

The above is one definition of the SP score. However, we can also choose to think of and calculate the SP score in a different way; namely as the sum of the scores of the induced pairwise alignments of M; see Figure 2.

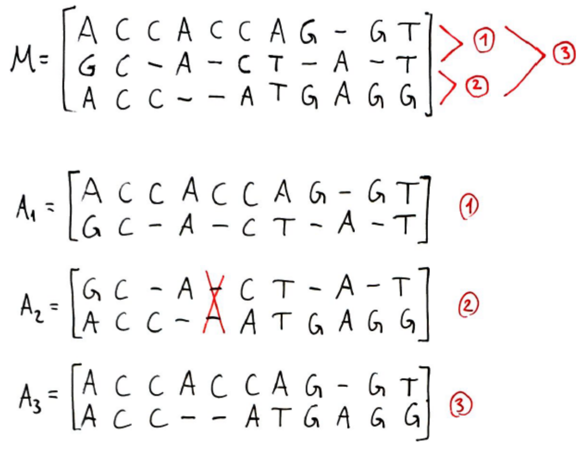

Figure 2: Example of multiple alignment, M, and the induced pairwise alignments, A_1, A_2 and A_3. The SP score of M is $SP(M) = SCORE(A_1) + SCORE(A_2) + SCORE(A_3)$.

We call each pair of rows in M an *induced* pairwise alignment (removing potential all-gap columns), and then reuse the scoring function for $k = 2$ on each such alignment. We then sum these scores to get the total SP score of M. To see that this is the same as the SP score defined previously, remember that addition is commutative, so it does not matter whether we sum over the rows or columns first, i.e.,

$$\begin{aligned} SP(M) &= \sum_{0 \le c < \ell} \sum_{0 \le i < j < k} d(M_{i,c}, M_{j,c}) \\ &= \sum_{0 \le i < j < k} \sum_{0 \le c < \ell} d(M_{i,c}, M_{j,c}). \end{aligned}$$

I use SP scores as the score measurement in my experiments in section 6 to quantify and compare how good the two different alignment algorithms are.

3 Gusfield's approximation algorithm

One approximation algorithm for computing MSAs is *Gusfield's approximation algorithm* [3]. The algorithm can be described in two steps:

1. Find the *center string*, and create a *star tree* from it.
2. Construct an alignment, M, using the star tree as a *guide tree.*

I will now describe each of these steps in more detail.

3.1 Finding the center string

First, we want to find the so-called *center string*. This is the sequence that has the lowest summed optimal alignment score to all the other sequences. Let S be the set of input sequences, and given two sequences, $s_i, s_j \in S$, let $OPT(s_i, s_j)$ be the score of an optimal alignment of s_i and s_j. Then, we define the center string, S_1, as

$$\arg\min_{S_1} \sum_{s \in S} OPT(S_1, s).$$

Let us then denote the remaining sequences $S_2, S_3, ..., S_k$, at random. Then, we construct a *star tree* with the edges (S_1, S_i) for $i \in [2, k]$; see Figure 3.

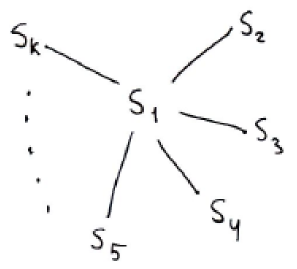

Figure 3: The star tree used in Gusfield's approximation algorithm.

Each edge in the star tree corresponds to an optimal pairwise alignment, and we can represent the SP score of these alignments as edge weights. We can construct an optimal pairwise alignment in time $O(n^2)$, using dynamic programming; so finding the center string takes time $O(k^2n^2)$, since we have $O(k^2)$ pairs to align.

3.2 Constructing the alignment

After constructing the star tree, we can use this as a *guide tree* for constructing a MSA, in the following way. We build the alignment matrix, M, row by row, until all the input sequences have been added. We use the guide tree to determine the order in which to add the sequences to M. The structure of the algorithm is summarised in Algorithm 1.

Algorithm 1 GUSFIELD-MSA(S)

```
1: $S_1 \leftarrow center(S)$
2: $M \leftarrow [\,]$
3: for $i = 2$ to $k$ do
4:     $A \leftarrow align(S_1, S_i)$
5:     $M \leftarrow extend(M, A)$
6: end for
7: return $M$
```

In Algorithm 1, we first find the center string, S_1, in line 1. Then, we initiate the MSA, M, as an empty list in line 2; M should be a matrix, so we can think of it as a list of rows. In lines 3-6, we iteratively extend M using pairwise alignments between S_1 and every other sequence. So we start by aligning S_1 and S_2, which results in an alignment, A. Since M is empty, extending M using A simply means appending the two rows of A to M. Now, M contains two rows, corresponding to S_1 and S_2. In the next iteration, A is now an optimal alignment of S_1 and S_3, and we extend M using this alignment; now, M is not empty, so extending M using A means adding a row to M corresponding to S_3, using A. We continue doing this, until we have extended M with the alignment of S_1 and S_k, after which we have a row in M for each of the k sequences. In section 3.2.1, I will explain how to extend M using A in more detail.

Let us consider the running time of Algorithm 1. Finding the center string in line 1 is $O(k^2n^2)$. The for loop in line 3-6 is executed $O(k)$ times, and constructing A is $O(n^2)$. Extending M means running through a row in A and a row in M once each, as I will explain in section 3.2.1; this is $O(kn)$, though usually the running time is much lower that this. So the total running time is $O(k^2n^2 + kn^2 + k^2n) = O(k^2n^2)$, and is thus dominated by the time it takes to find the center string.

3.2.1 Extending M using A

Extending M using A means adding a new, bottom row to M, corresponding to the bottom row of A. The top row of A corresponds to the top row of M, i.e., the center string. We extend M by iterating the columns of M using a pointer, i, and the columns of A, using a pointer, j. Given such a column pair, i, j, if $M_{0,i} = A_{0,j}$, i.e., the characters in the two center string rows match, then we simply insert $A_{1,j}$ in the new row of M in column i. We then move on to $(i + 1, j + 1)$. See Figure 4 for a visualisation of this.

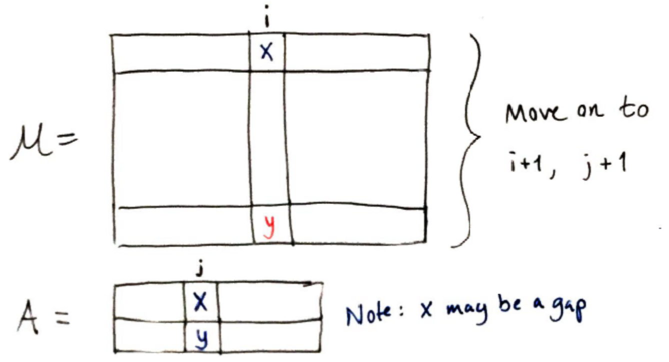

Figure 4: Case where $M_{0,i} = A_{0,j}$, i.e., the symbols (x, which may be a gap) match in the center string rows. Then we simply insert $A_{1,j}$ (y) in the new row of M in column i (red), and move on to $(i + 1, j + 1)$.

If the characters do not match, we have two cases:

1. $M_{0,i}$ is a gap, but $A_{0,j}$ is not.
2. $A_{0,j}$ is a gap, but $M_{0,i}$ is not.

If $M_{0,i}$ is a gap, but $A_{0,j}$ is not, we will insert a gap in the new row of M in column i. We then move on to $(i + 1, j)$; see Figure 5.

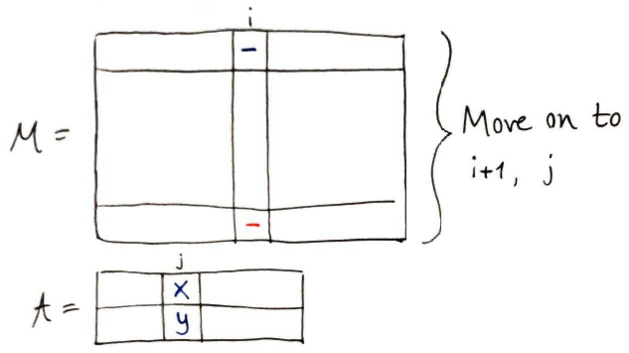

Figure 5: Case where the symbols do not match; $M_{0,i}$ is a gap, but $A_{0,j}$ is not. We insert a gap (red) in the new row of M in column i, and then move on to $(i + 1, j)$.

If $A_{0,j}$ is a gap, but $M_{0,i}$ is not, we insert a new column in M to the left of column i with $A_{1,j}$ in the new row, and gaps in all rows above. Once we have done this, we move on to $(i, j + 1)$; see Figure 6.

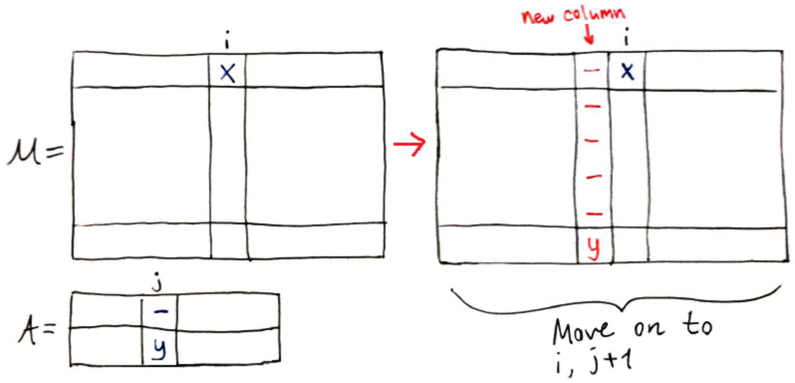

Figure 6: Case where symbols do not match; $A_{0,j}$ is a gap, but $M_{0,i}$ is not. We insert a new column to the left of column i; the new column (red) consists of $A_{1,j}$ (y) in the new row, and gaps in all rows above. We then move on to $(i, j + 1)$.

Note, from the way we extend M (Figure 4, 5, and 6), that if we extract the rows from M corresponding to a given A, then this induced alignment (removing potential all-gap columns) will have the same score as A, i.e., it will be optimal. Also note, that this is only necessarily true for the induced pairwise alignments that correspond to edges in the star tree, and not all possible induced pairwise alignments of M. If it were true for all possible induced pairwise alignments, then M would be optimal, and the algorithm would be an *exact* algorithm, and not an approximation algorithm. It can be proven that the algorithm creates alignments with a score that is less than twice the score of an optimal alignment, i.e., $SP(M) < 2 \cdot SP(M^*)$ if M^* is an optimal alignment. I will not go into the proof here, as the primary focus of this project is the MST algorithm, as well as the experiments comparing the two algorithms.

4 MST heuristic algorithm

The guide tree in Gusfield's approximation algorithm is a star tree, as explained in section 3. The guide tree *guides* us on which pairwise alignment we should use to extend M next. As stated previously, an edge weight in the guide tree represents the score of an optimal alignment of the two endpoint sequences. These scores are part of the total SP score of the alignment. As such, it is relevant to consider which guide tree would result in the best MSA; a star tree or some other tree? In Gusfield's algorithm, since we choose the center string in the way that we do, the best possible star tree is used, with regard to edge weight. But is it necessarily a good idea to restrict the algorithm to only consider star trees? What would happen if we chose the *overall* best tree, and not just the best star tree? The overall best tree would be a *minimum spanning tree* (*MST*), and this is what I will explore in this project. I will experiment with how using a MST as a guide tree, will affect the resulting MSAs and their scores. I will measure and compare the performance of the two algorithms; Gusfield's algorithm using a star tree and the MST algorithm for constructing MSAs. First, I will introduce the concept of MSTs and how to construct them; then, how to use them as guide trees for constructing MSAs.

4.1 Minimum spanning trees

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices in the graph and E is a set of edges connecting (some of) these vertices [4]. Vertices are also often called nodes, and I will use these terms interchangeably. For each edge, $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost of connecting u to v. Creating a MST of G means finding a non-cyclical subset of edges that connect all the vertices in V, with minimum summed cost. There are multiple methods of doing this; in this project, I will focus on *Prim's algorithm.*

4.2 Prim's algorithm

Prim's algorithm is a greedy algorithm for constructing MSTs; it starts from a given root node, and grows the MST one edge at a time. The edges chosen by Prim's algorithm forms a single tree in every iteration of the algorithm. As stated, the algorithm is greedy, meaning that in every iteration, it always chooses to add the lightest possible edge to the tree. How the algorithm determines which edge is the lightest, depends on the implementation. Typically, Prim's algorithm is implemented using a min-priority queue that can achieve the above with an extract-min function. Pseudocode for such an algorithm can be seen in Algorithm 2.

Algorithm 2 MST-PRIM($G, w, root$)

```
1: for each $u \in V[G]$ do
2:     $key[u] \leftarrow \infty$
3:     $\pi[u] \leftarrow NIL$
4: end for
5: $key[root] \leftarrow 0$
6: $Q \leftarrow V[G]$
7: while $Q \neq \emptyset$ do
8:     $u \leftarrow$ EXTRACT-MIN($Q$)
9:     report $(\pi[u], u)$
10:    for each $v \in Adj[u]$ do
11:        if $v \in Q$ and $w(u, v) < key[v]$ then
12:            $\pi[v] \leftarrow u$
13:            $key[v] \leftarrow w(u, v)$
14:        end if
15:    end for
16: end while
```

In Algorithm 2, the inputs are the connected graph, G, the weight function, w, and the *root*, i.e., the node from which we will start growing the MST. This is typically just a random node, as the algorithm will construct a MST, regardless of which node is chosen as the root. In each iteration, the key value of any node not currently in the MST, is the weight of the lightest edge connecting the node to the MST. The parent of the node is the node at the other endpoint of this lightest edge. As such, the key values are what we are interested in, when deciding which edge to add to the tree next. We start by initializing the key value of every node to ∞ and the parent of every node to NIL, in lines 1-4 in Algorithm 2. Then, we set the key value of the root to 0 in line 5, since we want the root to be the first node in the tree. In this version of Prim's algorithm, we use a min-priority queue, Q, which we initialize to be the entire set of nodes, in line 6. The queue has an EXTRACT-MIN function, that lets us extract the node with minimum key value from Q. In lines 7-16, while Q is not empty, we extract the minimum node, u, add it to the MST by reporting the edge $(\pi[u], u)$, and then update the key and parent values of its adjacent nodes that are not already in the MST.

So what is the time complexity of Algorithm 2? It depends on the implementation of the min-priority queue, Q. We could implement Q as a binary min-heap. Then it would take time $O(|V|)$ to build Q. The while loop is executed $O(|V|)$ times, and EXTRACT-MIN is $O(\log|V|)$ for a binary min-heap. The for loop in lines 10-15 is executed $O(|E|)$ times in total, since the sum of the lengths of all the adjacency lists is $2|E|$. We can test for membership in Q (line 11) in constant time if we use a bit for each node, which we update when we remove the node from Q. Line 13 is an implicit decrease-key operation on Q, which can be implemented on a binary min-heap in time $O(\log|V|)$. So the total time is $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$.

In our case, the set of nodes is the k sequences, so $|V| = k$, and we want to consider all possible edges between these nodes. Since every edge consist of two nodes, we have

$$\binom{k}{2} = \frac{k!}{2!(k-2)!} = \frac{k\cdot(k-1)\cdot(k-2)\cdot\ldots\cdot 1}{2\cdot(k-2)\cdot(k-3)\cdot\ldots\cdot 1} = \frac{k\cdot(k-1)}{2} = O(k^2)$$

edges in our graph.

So we want to construct a MST for a complete graph with $|E| = O(k^2)$ edges. This means that we could get a running time of $O(|E|\log|V|) = O(k^2\log k)$, using a binary min-heap. It turns out, however, that there is a simpler version of Prim's algorithm, that does not require a queue, and that runs in time $O(|V|^2) = O(k^2)$, making it a faster option for our particulate purpose. Note, that since the preprocessing of w is $O(k^2n^2)$, and typically $n^2 >> \log k$, then which version of Prim's algorithm we use will not be of great importance. However, since the faster version is also simpler, it makes sense to use this.

4.3 Prim's algorithm in time $O(|V|^2)$

In this section, I will explain a simple version of Prim's algorithm, that does not require a priority queue, and that runs in time $O(|V|^2)$. Since this corresponds to $O(k^2)$ in our case, it is actually faster than the version using a queue. Pseudocode for the modified algorithm is shown in Algorithm 3.

Algorithm 3 MST-PRIM-SIMPLE$(G, w, root)$

```
1: for each $u \in V[G]$ do
2:     $key[u] \leftarrow \infty$
3:     $\pi[u] \leftarrow NIL$
4: end for
5: $key[root] \leftarrow 0$
6: for $i = 1$ to $|V[G]|$ do
7:     $u \leftarrow$ FIND-MIN$(V[G], key)$
8:     report $(\pi[u], u)$
9:     for each $v \in Adj[u]$ do
10:        if $v \notin MST$ and $w(u, v) < key[v]$ then
11:            $\pi[v] \leftarrow u$
12:            $key[v] \leftarrow w(u, v)$
13:        end if
14:    end for
15: end for
```

The algorithm is structurally very similar to Algorithm 2, but here, we do not require a queue to find the minimum node. Instead, we use a simple FIND-MIN function which is described in Algorithm 4. FIND-MIN runs through all nodes in V that are not already in the MST and keeps track of the node with the minimum key value. This minimum node is then returned. In Algorithm 3, once we have found the minimum node, u, we report $(\pi[u], u)$. Then, we again update the key and parent of all the adjacent nodes.

Algorithm 4 FIND-MIN(V, key)

```
1: $min\_key \leftarrow \infty$
2: $min\_node \leftarrow NIL$
3: for $v \in V$ do
4:     if $v \notin MST$ and $key[v] < min\_key$ then
5:         $min\_node \leftarrow v$
6:         $min\_key \leftarrow key[v]$
7:     end if
8: end for
9: return $min\_node$
```

Let us consider the running time of Algorithm 3. The for loop in lines 6-15 is executed $|V|$ times. FIND-MIN is $O(|V|)$, assuming that we can check for membership in the MST in constant time; we can of course achieve this by using a bit for each node, as we did in Algorithm 2. The for loop in lines 9-14 is still executed a total of $O(|E|)$ times, but now the operations inside the loop are all constant time. So the running time is $O(|V|^2 + |E|)$. For a complete graph, $|E| = |V|^2$, so the running time is $O(|V|^2)$.

Now we have seen two versions of Prim's algorithm; Algorithm 2 which has a running time of $O(|E| \log |V|) = O(k^2 \log k)$, and Algorithm 3 with a running time of $O(|V|^2) = O(k^2)$. I have chosen to implement the latter version in this project, due to its simplicity and time complexity. Now that we know how to construct MSTs, I will explain how to use them to construct MSAs.

4.4 MST alignment algorithm

In this section, I will explain an algorithm for constructing MSAs that is based on Gusfield's approximation algorithm, but uses a MST as a guide tree, instead of the star tree explained in section 3.1. This means that we add the sequences to the alignment matrix, M, in the order that Prim's algorithm adds them to the MST. Let MST be a list of edges sorted in the order that Prim's algorithm would add them to the MST. Then, the basic MST algorithm for constructing MSAs is outlined in Algorithm 5.

Algorithm 5 MST-MSA(MST)

```
1: $M \leftarrow [\,]$
2: for $(S_i, S_j) \in MST$ do
3:     $A \leftarrow align(S_i, S_j)$
4:     $M \leftarrow extend(M, A)$
5: end for
6: return $M$
```

The basic idea in Algorithm 5 is the same as in Gusfield's algorithm. The differences lie in which sequences we construct pairwise alignments for, and the order in which we add rows to M. In Gusfield's algorithm, we made pairwise alignments of the center string, S_1, and every other sequence. Here, we do not have a center string, but instead we create pairwise alignments between the sequences that correspond to edges in the MST, (S_i, S_j), in the order that they are added to the MST by Prim's algorithm. The way that we extend M is very similar to the method used in Gusfield's algorithm; only here, we do not always look at the first row of M, i.e., the center string row. Instead, in an iteration where we want to extend M with the MST edge, (S_i, S_j), we look at the row of M that corresponds to the parent, S_i, of node S_j; then, we extend M with a new, bottom row corresponding to S_j. Since we add the rows to M in the same order that we build the MST, the parent will always already be a row in M, so the row we need to look at in M will always be there. Other than looking at different rows of M, the extend function is the same as in Gusfield's algorithm. Therefore, the same property holds, that the induced pairwise alignments of rows corresponding to the edges in the guide tree, are optimal. In this case, these are simply given by the edges in the MST, and not the star tree, as in Gusfield's algorithm.

Let us now consider the running time. In Gusfield's algorithm, the dominating term was finding the center string, which was $O(k^2n^2)$. Here, we still have to construct optimal pairwise alignments of all pairs of sequences; we need the scores of these alignments as input to Prim's algorithm (as the weight function, w). The for loop in lines 2-5 of Algorithm 5 is executed $O(k)$ times, and pairwise aligning is still $O(n^2)$, and extending is still $O(kn)$. So the total running time is again dominated by the time it takes to construct all the optimal pairwise alignments, $O(k^2n^2)$, so the running time is the same as for Gusfield's algorithm. Now, we have seen the idea behind constructing MSAs using MSTs; in the next section, I will show part of my implementation of such an algorithm.

5 Implementations

I will now show part of my python implementation of the MST algorithm for constructing MSAs. The below function first creates a MST, using Prim's algorithm; then constructs the alignment matrix, M. The rest of my implementation, including Prim's algorithm, the *extend_M* function, and the code used for the experiments in section 6, can be seen at https://github.com/Allegris/MSA-project-2022.

```python
1  def MST_MSA_approx(seq_indices, seqs, sub_matrix, gap_cost):
2      M = []
3      # Keep track of seq index vs row index in M
4      seq_idx_to_row = [None] * len(seq_indices)
5      # Iterate MST edges
6      MST = MST_prim(seq_indices, seqs, sub_matrix, gap_cost)
7      for i in range(len(MST)):
8          # Edge (parent, node)
9          parent = MST[i][0]
10         node = MST[i][1]
11         # Fill out dyn. prog. table for pairwise alignment
12         table = fill_table(seqs[parent], seqs[node],
13                 sub_matrix, gap_cost)
14         # Construct opt. pairwise align. from table
15         A = construct_alignment(table, seqs[parent], seqs[node],
16             sub_matrix, gap_cost)
17         if i == 0:
18             seq_idx_to_row[parent] = 0
19             seq_idx_to_row[node] = 1
20             M = A
21         else:
22             seq_idx_to_row[node] = len(M)
23             M = extend_M(M, A, seq_idx_to_row[parent])
24     return M
```

The code above has a structure very similar to that of Algorithm 5. In line 2 above, I initiate the alignment matrix, M, as an empty list. I then make a list, *seq_idx_to_row* that keeps track of which input sequence corresponds to which row of M. We need to know this, since when extending M using A, we need to look at the row of M corresponding to the first row of A; this could be any row of M. This information is not needed in Gusfield's algorithm, in which we always look at the first row of M, i.e., the center string row, but here, it is necessary. Then, in line 6, I create a MST for the sequences, using my implementation of Prim's algorithm. I have implemented the version of Prim's algorithm described in section 4.3. I then iterate the edges of the MST, in the order that they are added to the MST by Prim's algorithm. For each edge, $(parent, node)$, I create an optimal pairwise alignment, A, of the *parent* and *node* sequences. In the first iteration, I set $M = A$ and record that the row index of *parent* and *node* is 0 and 1, respectively. For all other iterations, I extend M using A by calling an *extend_M* function, very similar to the one described in section 3.2.1; it takes the row index of the parent as input, such that it knows which row of M to look in. Then it extends M with a new row corresponding to *node*. This iterative process continues until there are no more edges left in the MST; by then, M contains exactly one row for each input sequence. The function terminates by returning M in line 24.

As mentioned, I had already implemented Gusfield's algorithm during a previous course, so this new MST implementation made it possible to compare the two algorithms experimentally. In section 6, I will show the results of some of my experiments.

6 Experiments

I have run a series of experiments with short, simulated DNA sequences of lengths $n \in \{10, 50, 100, 150\}$. For each of these lengths, I generated a random DNA sequence that I used as a so-called *common ancestor*, and from this, I produced a series 19 simulated data sets, one for each $k \in [2, 20]$, each data set consisting of k sequences. I will explain in more detail, how I created these data sets in section 6.1. For each data set, I ran both Gusfield's algorithm and the MST algorithm, and measured the score of the resulting alignments, as well as the running time. Every plot in this section is the result of running the code on the same data five times and taking the average measurement. In all my experiments, I used a gap cost of 5 and the following substitution matrix:

$$\begin{array}{c c} & \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{array}{c} A \\ C \\ G \\ T \end{array} & \begin{pmatrix} 0 & 5 & 2 & 5 \\ 5 & 0 & 5 & 2 \\ 2 & 5 & 0 & 5 \\ 5 & 2 & 5 & 0 \end{pmatrix} \end{array}.$$

6.1 Simulating DNA sequence data

In order to test the two algorithms, I needed multiple data sets, each consisting of somewhat similar sequences to align. It was not straightforward to find a simple, already existing tool for creating DNA sequences, e.g., from a common ancestor, that would take biological properties into account. Therefore, I created my own, very simple, model for simulating DNA sequences. I did this by first creating a common ancestor, which for simplicity, I chose to be a random string, $a \in \{A, C, G, T\}^n$ for a given n. Then, from this common ancestor string, I simulated k sequences, for a given k, in the following way.

For each character, c, in the ancestor string, there is a probability that a new character will be inserted before c (insertion). There also is a probability that c will be deleted from the string (deletion). If c is not deleted, there is a probability that c will be replaced with another character (substitution). These probabilities are given as input to the data simulation function. For simplicity, when an insertion or substitution occurs, the new symbol is chosen at random. Of course, these choices are not necessarily biologically accurate, but in lieu of a better model, I have chosen to use a very simple one. The k simulated sequences are then stored in a FASTA file, from which they can be read and aligned by the two algorithms. In my experiments, I have used different substitutions rates, and always set the insertion-deletion rate to 1/5 of the substitution rate.

6.2 Correctness

First, I wanted to test the correctness of the MST algorithm for constructing MSAs. To do this, I created random sequences for $n \in \{10, 50, 100, 150\}$ and $k = 2, ..., 20$, resulting in $4 \cdot 19 = 76$ sets of sequences. I then ran the algorithm on each of these data sets. For each run, I calculated the score of an optimal pairwise alignment corresponding to each edge in the constructed MST, and checked that this score was equal to the score of the corresponding induced alignment in the constructed MSA. This experiment showed that this was indeed the case, so I believe that the algorithm works as expected. Since I had already implemented Gusfield's algorithm during a previous course, I had already tested the correctness of this implementation, in a similar way. In conclusion, both algorithms seem to work as expected. I will now show the experiments that I conducted in order to compare the algorithms with regard to running time and SP score.

6.3 Running time

First, I measured the running time of Prim's algorithm in isolation, which we expect to be $O(k^2)$, cf. section 4.3. In Figure 7, we can see the measured running times for random data with $n \in \{10, 50, 100, 150\}$ and 8 values of k between 5 and 40 (again averaged over five runs). In Figure 8, we see that if we divide the running time with k^2, the results seem to lie beneath a constant (horizontal) line for all four values of n, indicating that the running time of the implementation of Prim's algorithm is indeed $O(k^2)$, as expected.

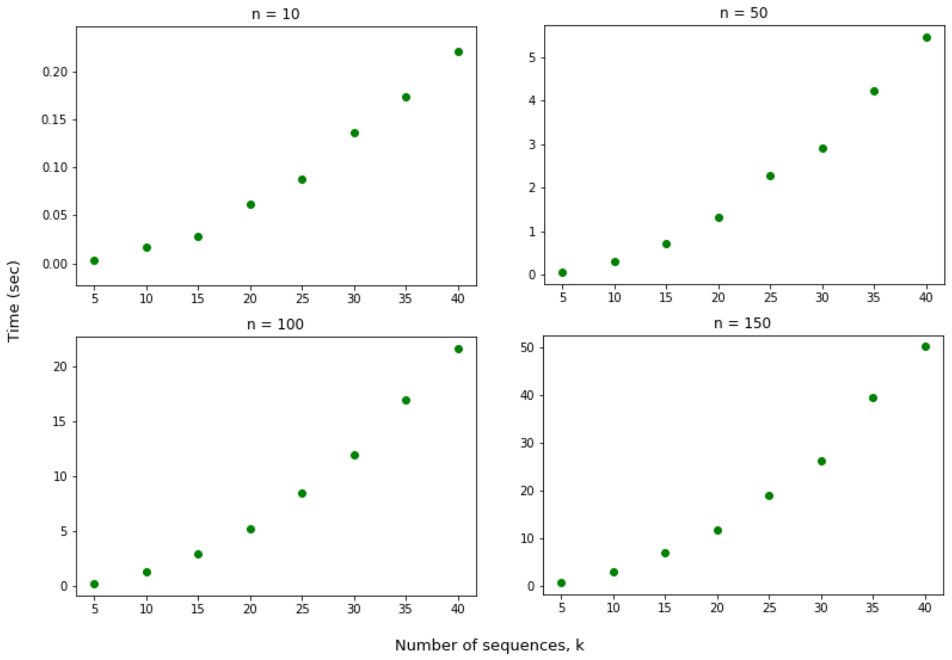

Figure 7: Running time of Prim's algorithm on random data as a function of k for four different values of n.

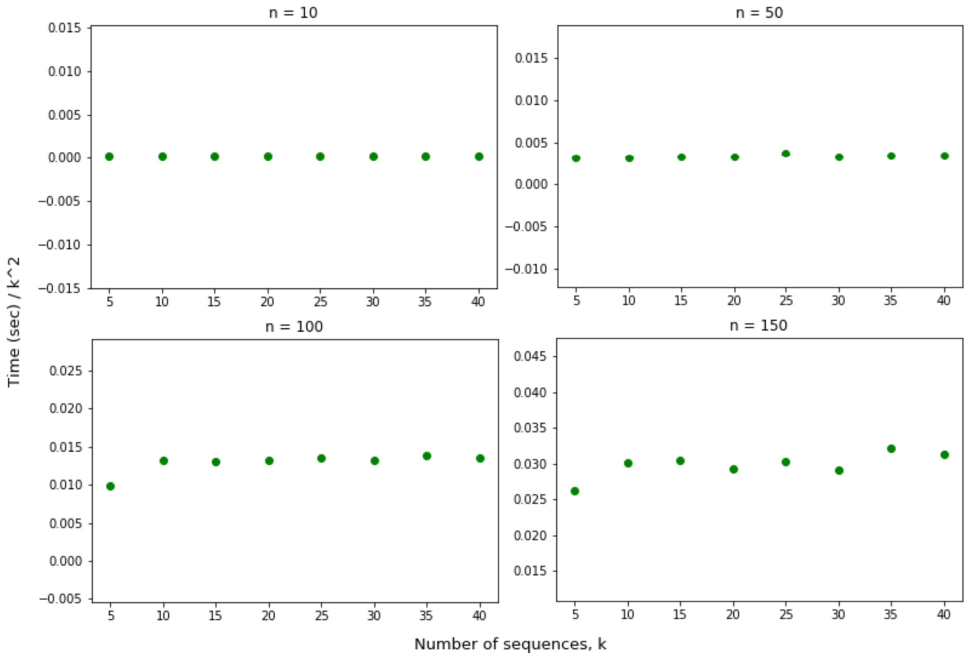

Figure 8: Running time of Prim's algorithm divided by the expected worst case running time, k^2.

Then, I measured the running time of each of the two MSA algorithms, as a whole. In Figure 9, the measured running times for random data with $n \in \{10, 50, 100, 150\}$ and $k = 2, ..., 20$ can be seen. As expected, it seems that the running times of the two algorithms are very similar.

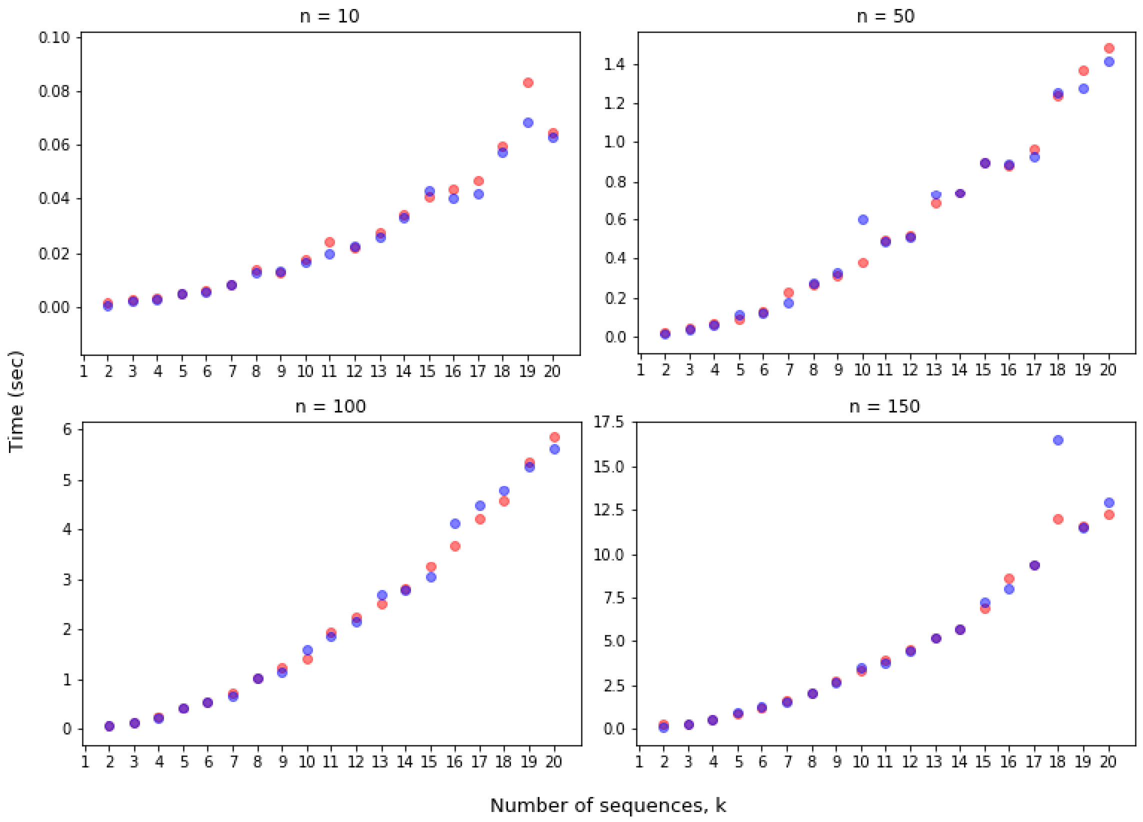

Figure 9: Running time of the two algorithms on random data as a function of k for four different values of n. Gusfield is red, MST is blue.

We expected the running time to be $O(k^2n^2)$ for both algorithms, and if we plot the time measurements divided by k^2n^2, we see that the results indeed seem to lie beneath a constant line; see Figure 10.

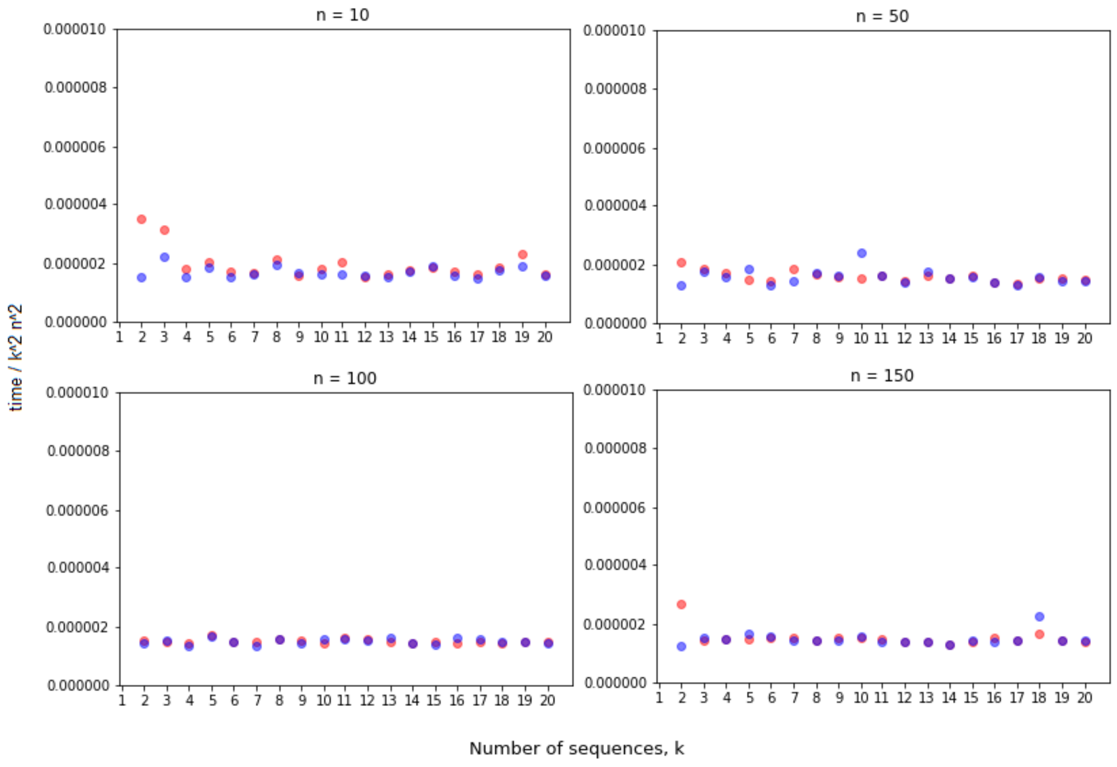

Figure 10: Running time of the two algorithms divided by expected worst case running time, k^2n^2, on random data. Gusfield is red, MST is blue.

6.4 SP score

I also wanted to compare the two algorithms with regard to the SP score of their constructed MSAs for different types of data; first, I used randomly generated data, i.e., the same 76 data sets used previously. The resulting scores can be seen in Figure 11. From the figure, we see that the MST algorithm (blue) never gets a better score than Gusfield's algorithm (red) for this type of data.

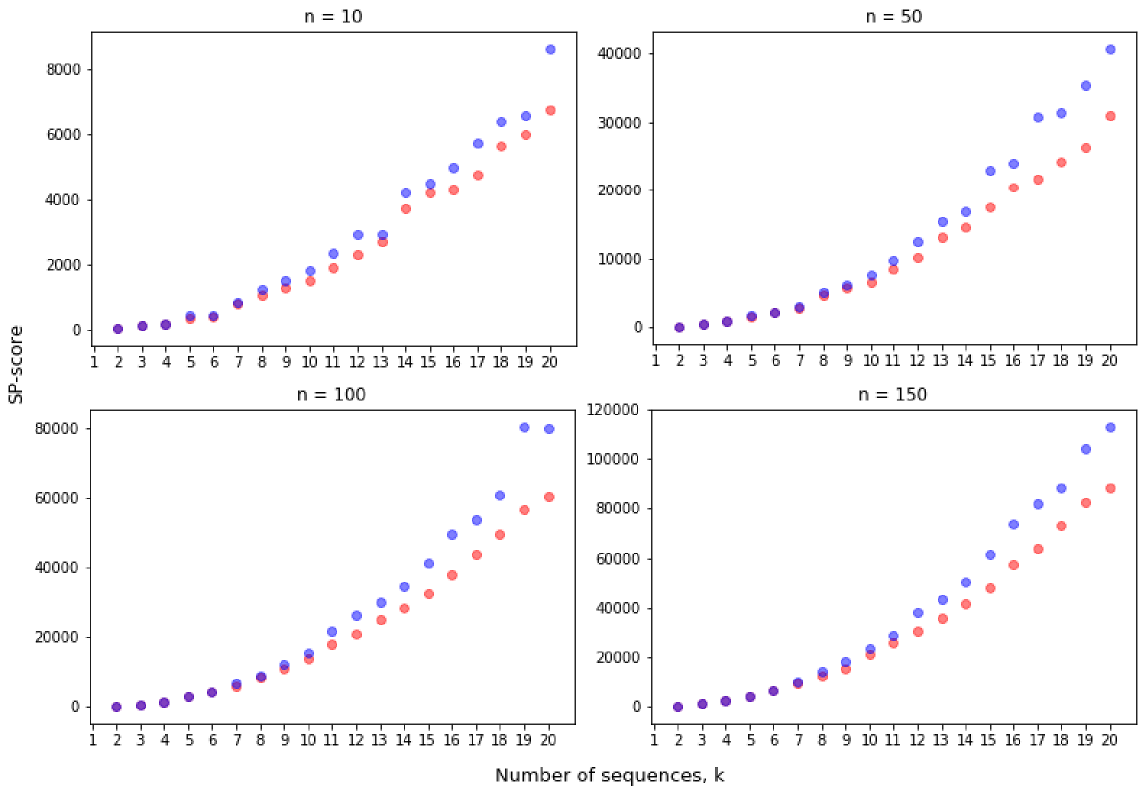

Figure 11: SP scores for the two algorithms as a function of k for four different values of n on random data. Gusfield is red, MST is blue.

I then explored if this was also the case for other types of data, by doing the same experiment, but with lower substitution rates, $\mu \in \{0.05, 0.2, 0.5\}$. A general result of these experiments were that the two algorithms seem to achieve very similar scores for small values of k, but that Gusfield's algorithm almost always outperforms the MST algorithm for larger values of k. Furthermore, their scores seem to differ more, the higher the substitution rate is, i.e., the closer the data is to random.

In Figure 12, the score plot for a low substitution rate of 0.05 can be seen. Here, we see that the scores are very similar, even for larger values of k. It makes sense that the algorithms yield more similar results for smaller substitutions rates, as this implies more similar input sequences. For this type of data, there are not as many "decisions to be made" by the algorithms, as there are for random data, which makes it more probable that the algorithms yield similar results.

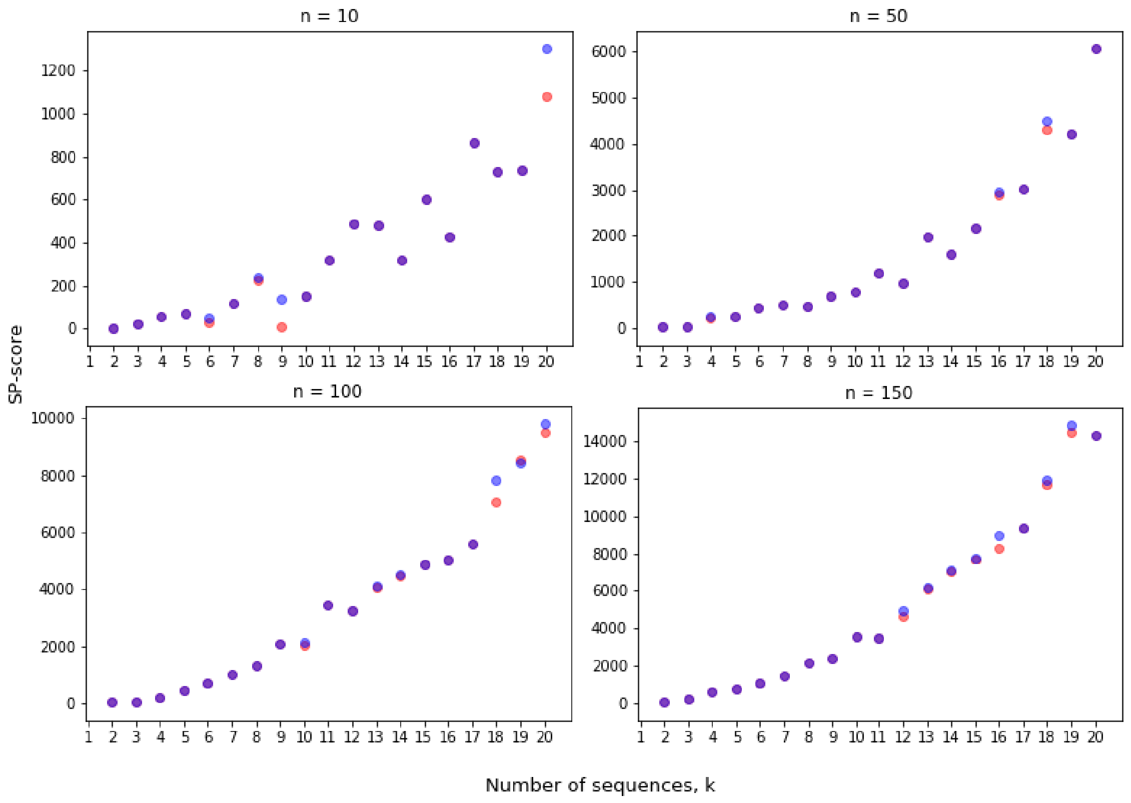

Figure 12: SP scores for the two algorithms as a function of k for four different values of n for simulated data with a low substitution rate of 0.05. Gusfield is red, MST is blue.

6.4.1 Clustered data

In the algorithm, the idea behind choosing the *overall best tree* (the MST), and not just the best star tree, as a guide tree, is that the restriction of only being able to use a star tree might be bad in some cases. Imagine a scenario in which we have two clusters that are well separated from each other. In this case, Gusfield's algorithm would have to choose a center string in one of the clusters, making the distances from the center string to the other cluster relatively long. This would result in a much heavier tree than the corresponding MST, as visualized in Figure 13.

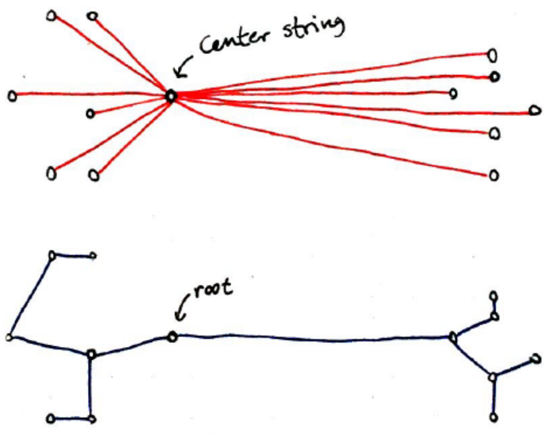

Figure 13: Gusfield's star tree (top) and a MST (bottom) for the same set of nodes. The total weight of the tree edges (indicated by edge length) is clearly much larger for the star tree than for the MST.

Remember, that the weight of a tree edge corresponds to the score of an optimal pairwise alignment between the endpoint nodes. So the sum of the pairwise alignment scores for the star tree in Figure 13 would be much higher than that of the MST. Therefore, intuitively, one would think that the MST algorithm would perform better on this type of data. Of course, the pairwise alignments corresponding to the edges in each tree are just part of the total SP score, so this is not necessarily the case. The score of the MSA also greatly depends on the order in which the sequences are added to the MSA. Nevertheless, this would be an interesting experiment to do. I therefore created two different ancestors, and created a set of descendants from each of them, with a low substitution rate of 0.05, such that the sequences should be much more similar within the two clusters than between them. The scores from this experiment can be seen in Figure 14.

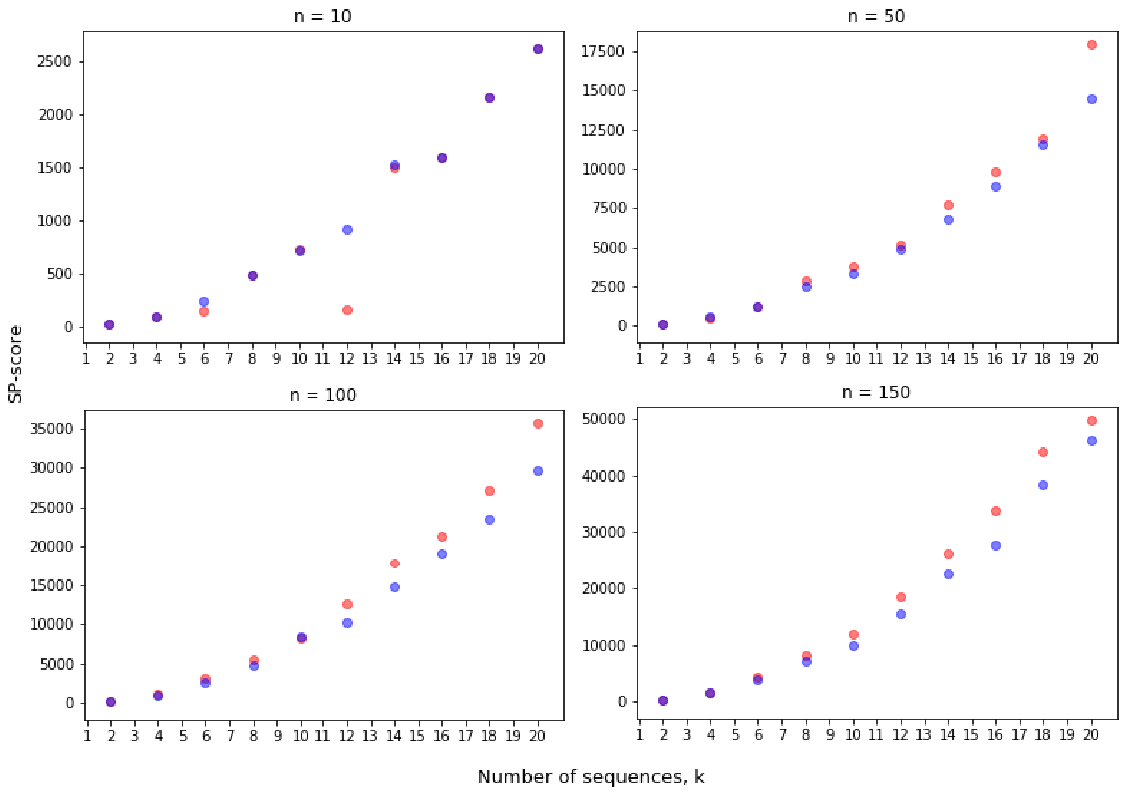

Figure 14: SP scores for the two algorithms as a function of k for four different values of n for simulated data consisting of two clusters (each made with a low substitution rate of 0.05). Gusfield is red, MST is blue.

In Figure 14, we see that now the MST algorithm outperforms Gusfield's algorithm more often than not (except for the smallest $n = 10$, for which the clusters are most likely to randomly be very similar). I repeated the experiment above a few more times with newly generated data to see that this was not just a coincidental result; this showed that the results were indeed very consistent. This is an interesting result, that also opens up the question of whether we can improve the MST algorithm further. Maybe using a different algorithm than Prim's algorithm for creating the MST, would yield even better results. Using a different algorithm would probably result in the same MST edges, but the order in which we create and merge alignments could be very different. I will discuss one way of doing this in section 7.1.

7 Discussion

7.1 An alternative MST approach

Intuitively, it seems that aligning the most similar sequences first should yield good results; once we insert gaps into the alignment, we do not remove the gaps again, so by aligning the most similar sequences first, we minimize the number of gaps that we insert from the start. This is, in a way, what we try to achieve using Prim's algorithm. However, e.g., in the case of two separate clusters, as in Figure 13, the MST algorithm using Prim's algorithm, would start by iteratively extending M to contain rows for all the nodes in one cluster, and then add a row for the closest node from the other cluster, and continue to add rows for the other cluster, until it is done. Maybe it would yield better results to create a MSA for each cluster and then merge them subsequently. Then, our guide tree would have to be different; a MST algorithm that could achieve this is Kruskal's algorithm. The primary difference between Prim's and Kruskal's algorithm is that Kruskal's algorithm always chooses the lightest possible edge left in the *entire graph* (still without creating any cycles), and not just the edges that are connected to the current set of MST nodes. So in each iteration, the set of chosen MST edges do not necessarily form a single tree. For a visualisation of how Prim's and Kruskal's algorithm add edges to the MST in different orders, see Figure 15.

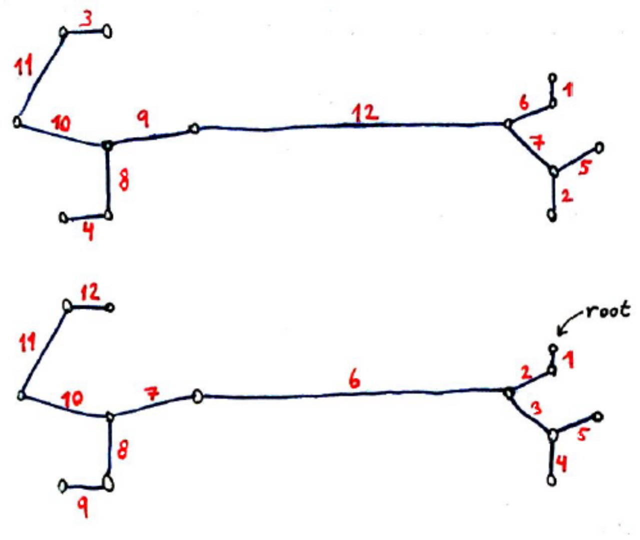

Figure 15: The same MST constructed by Kruskal's algorithm (top) and Prim's algorithm (bottom), respectively. The red numbers indicate the order in which the edges are added to the MST; edge 1 is added first, then 2, etc. Note that the longest edge in both trees (the one between the two clusters) is added to the MST last (as edge number 12) by Kruskal's algorithm, while it is added in-between the two clusters by Prim's algorithm (as edge number 6).

If we were to use Kruskal's algorithm to construct a MSA, we would align the two closest sequences first, then the two next closest sequences, etc. So it would not be possible to have just one single M, and then add one row to it in each iteration. This would, of course, make the merging process different from the one we have used so far, and so it might result in some very different MSAs. It would be interesting to implement and experiment with such an algorithm as well, but this is out of scope for this project.

7.2 An alternative star tree approach

I also experimented with another version of Gusfield's algorithm, by modifying the star tree in the following way. After finding the center string, S_1, the remaining strings are usually denoted S_2 to S_k, at random. As an alternative approach, I tried denoting the string closest to the center string S_2, the second closest string to the center string S_3, etc. This changes the order in which the pairwise alignments are created and sequences are added to the MSA. The idea behind this is, again, that aligning more similar sequences first, might reduce the number of gaps in the MSA. In this case, however, the strings are simply ordered by closeness to the center string, so, e.g., S_2 and S_3 could both be close to S_1, while not being close to each other. The experiment showed that this modification of Gusfield's algorithm resulted in very similar scores to the original Gusfield algorithm.

In summary, the experiments in this section show that Gusfield's algorithm achieves better SP scores than the MST algorithm for data sets consisting of sequences generated from one common ancestor, especially if the substitution rate is high. If the substitution rate is low, the two algorithms produce alignments of very similar scores. For data sets consisting of sequences simulated from two different ancestors with a low substitution rate, i.e., data in two separate clusters, the MST algorithm generally achieves better SP scores than Gusfield's algorithm. Furthermore, it may be possible to improve the MST algorithm by using, e.g., Kruskal's algorithm instead of Prim's algorithm. Ordering the star tree edges in Gusfield's algorithm by increasing weight did not seem to have any significant effect on the resulting SP scores.

7.3 Conclusion

In this project, I have compared two different algorithms for creating multiple sequence alignments. First, I described Gusfield's approximation algorithm that uses a star tree as a guide tree. Then, I explained how to modify the algorithm to use a minimum spanning tree instead of a star tree, and I explained how to construct MSTs using Prim's algorithm. I have implemented the MST algorithm for constructing MSAs using Prim's algorithm, and I have compared this experimentally to an implementation of Gusfield's algorithm that I had made during a previous course. The results show that with regard to running time, both algorithms seem to be $O(k^2n^2)$, as expected. With regard to SP score, Gusfield's algorithm outperformed the MST algorithm for data created from one common ancestor with a high mutation rate; for data with low a mutation rate, the algorithms yielded very similar SP scores. For clustered data, i.e., data simulated from two different ancestors, the MST algorithm generally outperformed Gusfield's algorithm. I also discussed how using Kruskal's algorithm instead of Prim's algorithm might improve the MST algorithm. Implementing and experimenting with such an algorithm is out of scope for this project, but would be interesting future work.

8 References

[1] - Chapter 8 (on sequence alignment) of Bioinformatics Algorithms, by Enno Ohlebusch, 2013, ISBN: 978-3000413162.

[2] - Slides about SP score and MSAs by Christian Storm at BiRC, Aarhus University, 2021.

[3] - Slides about approximate MSAs by Christian Storm at BiRC, Aarhus University, 2021.

[4] - Chapter 23 (on MSTs) of Introduction to Algorithms, Third Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, 2009