Algorithms in Bioinformatics - Project 3 Multiple Alignment

Group 6
Anastasia Kratschmer – 201907407
Noa Van de Velde – 202209940
Stinna Danger – 201907211
Anna Kathrine Skov – 201905355

March 20th 2023

1 Introduction

In this project, we are asked to implement and algorithm that can perform exact multiple alignment on 3 sequences, and an algorithm that can make an approximation for a multiple alignment with k sequences of which the cost is at most twice the optimal cost (when minimizing). Our algorithms yield both the optimal alignment cost and the alignment itself. Both programs work as intended, though could of course be optimised for efficiency and, in the case of the approximation algorithm, accuracy.

At the time of writing, the second part of the project (to be presented in class) is not yet implemented, therefore code pertaining to those tasks is unfinished.

2 Methods

In this section we give a overview of how our programs are constructed and run, and comment briefly on testing.

2.1 Exact optimal alignment for 3 sequences

The script `sp_exact.py` computes the exact optimal alignment of 3 given sequences. The algorithm is implemented quite analogously to the global linear pairwise alignment that we made in project 2, the main difference being the cases for the last column.

Instead of 3 possible cases for the last column for 2 sequences, this time there are 7 possible cases for the last column. Generally the number of last columns is (2^k) -1 for k sequences. The columns are all the possible combinations of gaps and letters from the columns of which we need to find the combination with the minimal cost.

Other differences compared with the exact linear global alignment algorithm for 2 sequences are the following:

- The matrix that we fill out has 3 dimensions now, instead of 2, and therefore the indexing is now also done with 3 ”coordinates” instead of 2, which also means that we have 7 possible last columns.
- Instead of initializing all the border cases with a separate loop from the one filling out the 3-d matrix, like we did in project 2, we just make sure that these cases are covered in the if-statements of the main loop.

2.1.1 Backtracking for 3 sequences

To retrieve the alignment itself, we backtrack the generated matrix from computing the score. This entails “reversing” the algorithm, working our way back through the 3D table and along the way testing the entry to see which combination of predecessors it came from. If this is possible, these predecessors are added into the alignments of the three sequences. In the end, the alignments are reversed, because backtracking builds the sequences from behind.

2.1.2 Running the algorithm

The command needed to run the algorithm in the terminal is the following - a more detailed description of how to run the program can be read in the project's `README.md`.

```
python sp_exact_3.py [sequence 1] [sequence 2] [sequence 3] [gap cost] [score
    matrix]
```

2.2 SP-approximation algorithm for k sequences

We also implemented the approximation algorithm for k sequences. This is in the script `sp_approx.py`. It first compute the (approximate) optimal alignment of the given sequences and then the cost of the found alignment.

2.2.1 Computing the (approximate) optimal alignment

The steps for the algorithm are:

1. Find the middle string, which is the string with the smallest sum of distance to all the other strings. We do this by running our linear global alignment algorithm for 2 sequences for each pair of two sequences, filling out a matrix of the cost of all the combinations. Then we can sum the rows or the columns and find the index of the column or row that has the lowest sum. This string is the middle string.
2. Use the middle string as a reference: Make a pairwise alignment of all the sequences to the middle string (using our linear gap cost global alignment algorithm from project 2).
3. Merge the alignments together stepwise, using the alignment of each sequence with the middle string as a guide. This is done column-wise by dividing the possible combinations of previous alignments and a new alignment into four cases, that are handled differently.

Case 1 Here, the next character of the sequence is that is being added to the alignment needs to be added in the same column as a gap in the guide sequence. There is a gap at the current column in the guide sequence, so the character from the new sequence is just added into the column.

Case 2 The next character of the sequence that is being added to the alignment needs to be added to a column where there is a certain letter in the guide sequence. But instead, the guide sequence's current column contains a gap. Here, a gap is added to the column, and no letter from the new sequence is added into this column.

Case 3 The next character of the sequence that is being added to the alignment needs to be added to a column where there is a gap in the guide sequence, and the column we are standing in contains something else for the guide sequence. Here we put in our character and fill the rest of the column with gaps.

Case 4 The sequence that is being added to the alignment needs to be added to a column where there is a certain character in the guide sequence, and the column we are standing in contains this character. Then the character from the new sequence is just inserted into the column.

2.2.2 Computing the cost

The optimal alignment cost (which is of course only an approximation) is retrieved by looping through all the columns in the alignment and adding their cost to the total score. This is possible, because our algorithm is column based because of the linear gap cost.

2.2.3 Running the algorithm

The command needed to run the algorithm in the terminal is the following - a more detailed description of how to run the program can be read in the project's `README.md`.

```
python sp_exact_3.py [FASTA file of sequences] [gap cost] [score matrix]
```

2.3 Testing

the bulk of our testing for this terminal was done via the terminal incrementally as we implemented the algorithms. There is a small automated test suite in the file, `tests.py`, which tests the correctness of our exact algorithm on both shorter and longer sequences.

Both programs make use of helper functions that are not all tested in this project, since they are functions from project 2 and as such have been tested previously.

3 Experiments

The experiments for the report can be run with the `experiments.py` script, which prompts the user on which experiment they would like to run, repeating this until the user exits. Again, a more detailed description of how to run the program can be read in the project's `README.md`.

3.1 What is the score of an optimal alignment of the first 3 sequences in ”brca1-testseqs.fasta” as computed by your program ”sp_exact_3”? How does an optimal alignment look like?

The optimal score of the alignment of the three first sequences in `brca2-testseqs.fasta` using our exact alignment algorithm is **790**.
The alignment is the following:

```
brca1_bos_taurus: atggatttatctgcggatcatgttgaagaagtacaaaatgtcctcaatgctatgca-
    gaaaatcttag--agtgtccaat-atgtctggagttgatcaaagag-cct-gtctctacaaagtgtga-cca-ca
    -tattttgcaaattttg-tatgctgaa-ac-ttctcaacca-gaagaaagggccttcacaatgtcc--
    tttgtgtaagaatga-
```

```
brca1_canis_lupus: atggatttatctgcggatcgtgttgaagaagtacaaaatgttcttaatgctatgca-
    gaaaatcttag--agtgtccaat-atgtctggagttgatcaaagag-cct-gtttctacaaagtgtga-tca-ca
    -tattttgcaaattttg-tatgctgaa-ac-ttctcaacca-gaggaaggggccttcacagtgtcc--
    tttgtgtaagaacga-
```

```
brca1_gallus_gallus: gcgaa---atgta-aca-cg-gtagaggtgat-cggggtg-cgtt-atac-
    gtgcgtggtgacctcggtcggtgt-tgacggtgcctggggttcctcagagtgttttggggtctgaaggatg-
    gacttgtcagtg-attgccattggagacgtgcaaaatgtgctttcagccatgcagaa-gaa-ctt-
    ggagtgtccagtctgtttagatgtgat
```

3.2 What is the score of the alignment of the first 5 sequences in ”brca1-testseqs.fasta” as computed by your program sp_approx? Which of the 5 sequences is choosen as the ’center string’?

The cost of the alignment of the 5 first sequences in the brca-file is **4103**. Our center string was the first string (index 0), which is the string from **Bos Taurus** (cattle).

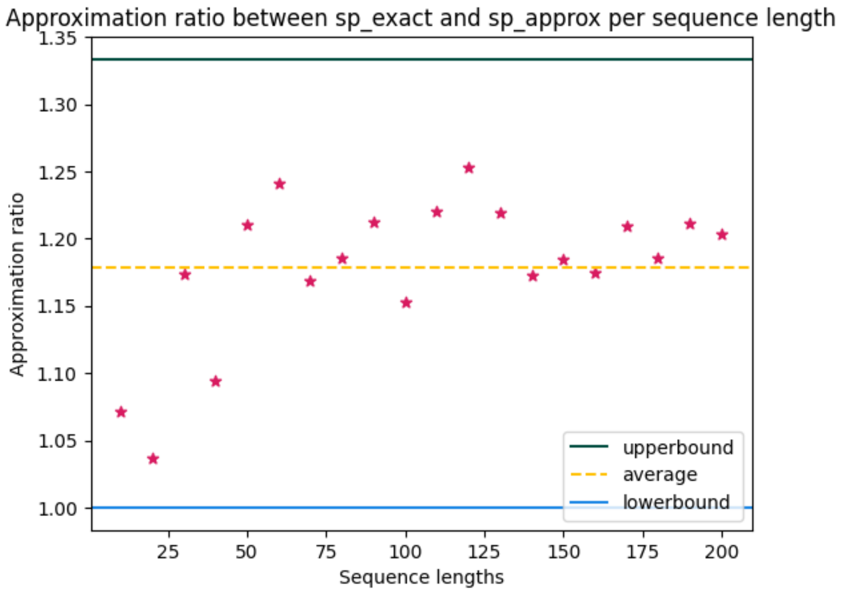
3.3 Make an experiment comparing the scores of the alignments computed by sp_exact_3 and sp_approx that validates that the approximation ratio of sp_approx is 2(k-1)/k for k sequences. i.e 4/3 for three sequences.

We test our algorithms and see how the differences are between the exact cost of alignment using our exact algorithms for aligning 2 and 3 strings compared to using our approximation algorithm.

k	seq	exact cost	appr. cost	ratio	exp ratio
2	seq 1+2	18	18	1	1
3	seq 1+2+3	790	879	1.11	1.33
3	seq 1+2+4	934	1021	1.09	1.33
3	seq 1+2+5	100	100	1	1.33
3	seq 1+3+4	1369	1680	1.23	1.33

We see that the ratio between the two costs is always lower than 2 and actually for our tested cases quite a lot lower than 2(k-1)/k, which is the maximum expected for k sequences. For one of the cases (seq 1+2+5) we actually got the same number using both algorithms.

The above is run using our terminal commands. Using the experiment script as described previously. The third experiment in the script loops through and logs the computed alignment cost of the triples of sequences provided in the `testseqs`-folder. The program yields an **average** ratio between our exact and approximate algorithms, which is ≈ 1.17. Below we have (colourfully ;-)) plotted the approximation ratios as a function of the sequence lengths.



We observe that for the shorter sequences, the ratio is closer to 1, i.e. more correct. Then, as the sequences get longer, the ratio sits around the average value as marked by the yellow dotted line.

4 Aligning the 8 brca1-sequences

In the second part of the project, we are asked to align the 8 sequences of length 200 in brca1-testseqs.fasta as close to optimum as possible (the sequences corresponds to first 200 nucleotides in the mRNA from the BRCA1 gene in cow, wolf, chicken, human, macaque, mouse, chimpanzee, and rat). We do this in the `brca1-alignment.py` script, which outputs the alignment to the `brca1-alignment.fasta`-folder.