Results of Using Different Heuristics for Carrillo-Lipman Algorithm
CS 466, Fall 2023
Professor Mohammed El-Kebir

Group members:

- Shaurya Singh (NetID: shaurya8, Email: shaurya8@illnois.edu)
- Rachanon Petchoo (NetID: petchoo2, Email: petchoo2@illinois.edu
- Nick Winkler (NetID: ngw4, Email: ngw4@illinois.edu)

*Our group decided to do a mix of implementation of different heuristics and benchmarking their performance which is why our report will be a mix of both project types.

*One of our group members dropped the course mid-project (after the proposal), so we decided not to implement ClustalW (as proposed). To compensate for this, we decided to implement the Greedy Progressive algorithm **and** Inorder Alignment algorithm instead.

Introduction:

Consider the following problem statement:

"Given strings $v_1, v_2 \ldots, v_k \in \Sigma^*$ and a scoring function δ : (Σ ∪ {−}) × (Σ ∪ {−}) → R, find a multiple sequence alignment A such that the SP score of A is the minimum, where SP score is the sum-of-pairs score"

This problem can be solved using forward dynamic programming in $O(2^k k^2 n^k)$, in which the optimal alignment is given by traversing from the source (0,...,0) to the sink ($|v_1|$,...,$|v_k|$) of the dynamic programming table using the following recurrence (taken from Lecture 6, UIUC CS466 Fall 2023):

$$d[i_1, i_2, \ldots, i_{k-1}, i_k] = \min \begin{cases} s[i_1 - 1, i_2 - 1, \ldots, i_{k-1} - 1, i_k - 1] + \sum_{p=1}^{k} \sum_{q=p+1}^{k} \delta(\mathbf{v_p}[i_p], \mathbf{v_q}[i_q]) & \text{no gaps} \\ s[i_1 - 1, i_2 - 1, \ldots, i_{k-1} - 1, i_k] + (k-1)\sigma + \sum_{p=1}^{k-1} \sum_{q=p+1}^{k-1} \delta(\mathbf{v_p}[i_p], \mathbf{v_q}[i_q]) & \\ \quad \vdots & \text{one gap} \\ s[i_1 - 1, i_2 - 1, \ldots, i_{k-1} - 1, i_k - 1] + (k-1)\sigma + \sum_{p=2}^{k} \sum_{q=p+1}^{k} \delta(\mathbf{v_p}[i_p], \mathbf{v_q}[i_q]) & \\ \quad \vdots & \\ s[i_1 - 1, i_2, \ldots, i_{k-1}, i_k] + (k-1)\sigma & \\ \quad \vdots & k-1 \text{ gaps} \\ s[i_1, i_2, \ldots, i_{k-1}, i_k - 1] + (k-1)\sigma & \end{cases}$$

Question: How many cases have 2 gaps?

In practice this can be very slow, and this is where the Carrillo-Lipman Algorithm is useful. We can use Carrillo-Lipman to determine in advance whether a cell in our dynamic programming table is going to be part of the optimal solution using the following "pruning mechanism" by pruning $(i_1, i_2, \ldots, i_n)$ if:

$$z < D(i_1, i_2, \ldots, i_n) + \sum_{p=1}^{n} \sum_{q=p+1}^{n} D^+_{p,q}(i_p, i_q)$$

where

- z is the score of a known solution (typically suboptimal)
- $D(i_1, i_2, \ldots, i_n)$ be the minimum SP-cost of aligning prefixes $v_1[1, \ldots, i_1], v_2[1, \ldots, i_2], \ldots, v_n[1, \ldots, i_n]$
- $D^+_{p,q}(i_p, i_q)$ be the minimum cost of aligning $v_p[i_p, .., n], v_q[i_q, \ldots, n]$

By comparing the cell's (potential) optimal score to the score of a solution (z) that we computed efficiently, we know whether the cell can lead to the optimal solution or not. If it cannot, then we can stop exploring the cell's neighbors and subsequent cells as it will only lead to suboptimal solutions, which is not what we wanted. This is in effect pruning the search space, so we can deliver powerful speed ups over the standard dynamic programming method of searching all possible paths.

The z-score essentially is the score of an suboptimal MSA computed efficiently by some heuristics. There are multiple heuristics that have been created, and we implement the following:

1. Greedy Progressive Alignment
2. Feng and Doolittle Alignment
3. ClustalW Alignment
4. Inorder Alignment (a made up algorithm to see if its performance match up with the popular heuristics)

The implementations of all of the above alignments, along with Carillo-Lipman algorithm and visualizations code (generating plots) can be found on the following Github page: https://github.com/Rachanon-Andrew-Petchoo/CarilloLipman-466

We also benchmark the performance of solving the stated problem with different inputs using Carillo-Lipman with different heuristics, charting their performance in plots that are elaborated more in subsequent sections.

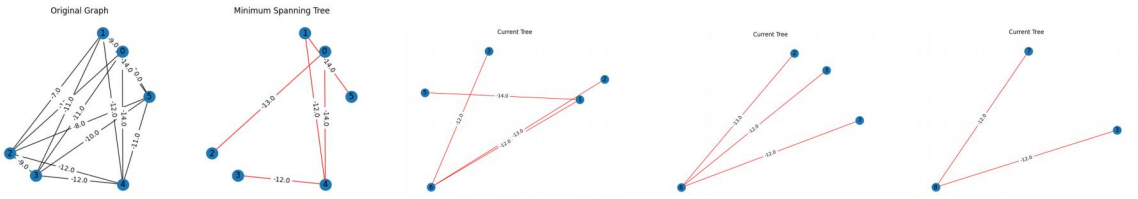
Methods:

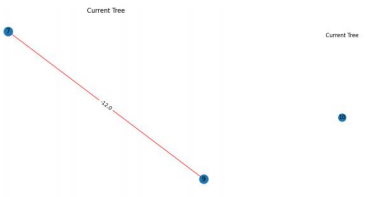
We run our implementation of Carrillo-Lipman repeatedly against self-generated DNA inputs of varying lengths and varying numbers of sequences. In each run we use each heuristic as the tool to find a suboptimal solution to be used in Carillo-Lipman pruning, and compare its performance regarding time and space to other heuristics. In particular we compare:

1. Relationship between Carillo-Lipman's running time and its search space's size after pruning
2. Determine which heuristic algorithm prune more search spaces (in general cases)
3. Determine which heuristic algorithm makes Carillo-Lipman run fastest (in general cases)
4. Relationship between length of sequence in the inputs and size of Carrillo-Lipman's search space
5. Relationship between length of sequence in the inputs and runtime of Carrillo-Lipman

The heuristics we chose to implement are:

1. Greedy Progressive Alignment Heuristic:
    a. We choose the most similar pairs of strings and align into a profile, so we have k-1 sequences/profiles left.
    b. We repeat the process (considering the most sequence-to-sequence, sequence-to-profile, and profile-to-profile alignment) until we have only one resulting profile, with all the sequences aligned.
2. Feng and Doolittle Alignment:
    a. Compute pairwise sequence alignments of sequences.
    b. Generate a complete graph G = (V, E) with vertices representing each sequence, and edge weights representing the score of aligning the sequences of the two vertices connecting
    c. Compute a (rooted) minimum spanning tree T of G
    d. Perform sequence- sequence, sequence-alignment and alignment-alignment alignment to construct MSA according to guide tree T. Graphs below show the complete graph, the original guide tree (MST), and the guide tree as it merges its vertices. More examples can be found by running our code (the code for visualization is provided, but is commented out):





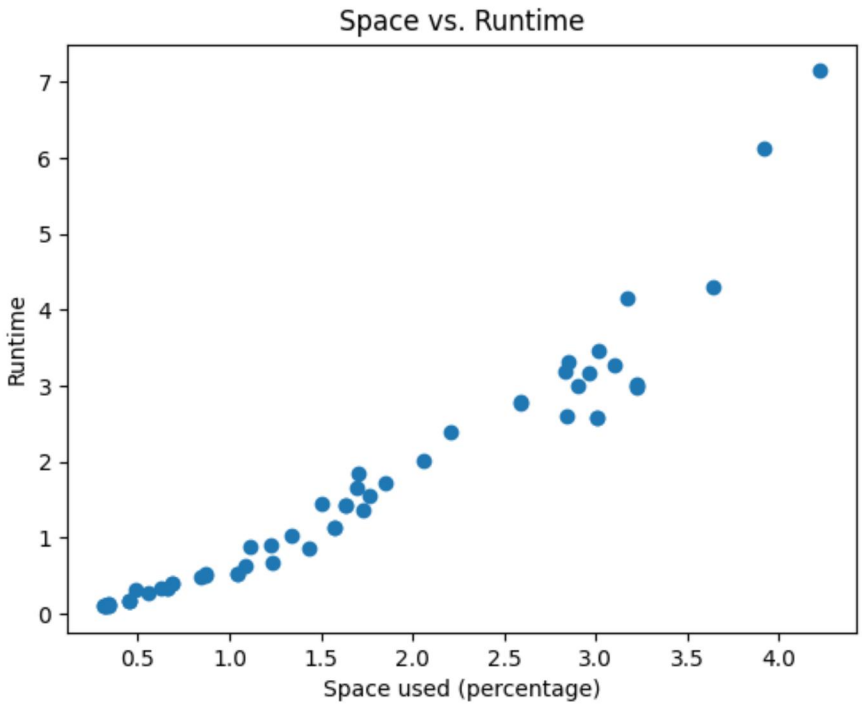
3. Inorder Alignment:
    a. Align the first and the second sequence of the sequences from the input.
    b. Combine the MSA from the previous step with the next sequence in the input.
    c. Repeat step b. until we have no more sequences to align with.
    d. Result in an MSA that contains all sequences from the input.

For our testing dataset, we created a function that randomly generated DNA strings to a desired number (of sequences) and length. This will be different every time we run the code, meaning when we call our heuristics they will run on different enough data to not bias the results.

Validation Results & Benchmarking Results:

Relationship between Carillo-Lipman's running time and its search space's size after pruning:

- Method:
  - We fixed the number of sequences to be 5 and the sequence length to be 9, and used about 20 different sets of DNA sequences
    - We have to fix these parameters so that the runtime only depends on the size of the search space, and not affected by varying number of sequences and sequence lengths
    - This number is chosen because it is large enough to differentiate the performance of the 3 heuristics
  - We run these inputs through each heuristic algorithm and subsequently passed into Carillo-Lipman, the results are then aggregated together across heuristic
- Plot:

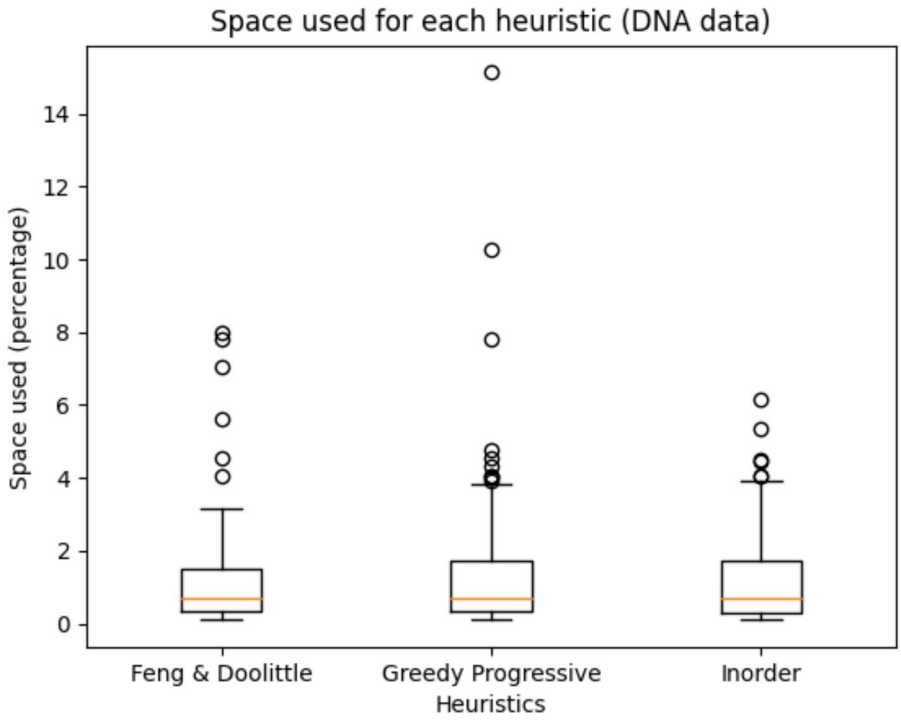


- Analysis:
  - As the space grows larger, the runtime is longer as well
  - The space size and runtime has linear relationship

Determine which heuristic algorithm prune more search spaces (in general cases):

- Method:
  - We used about 150 different sets of DNA sequences, with varying number of sequences and varying sequence length
    - This is chosen because we want to analyze each heuristic's performance in general cases, independent of the magnitude of number of sequences or sequence length

    - We run these inputs through each heuristic algorithm and subsequently passed into Carillo-Lipman, the results are then recorded separately between heuristics
- Plot:

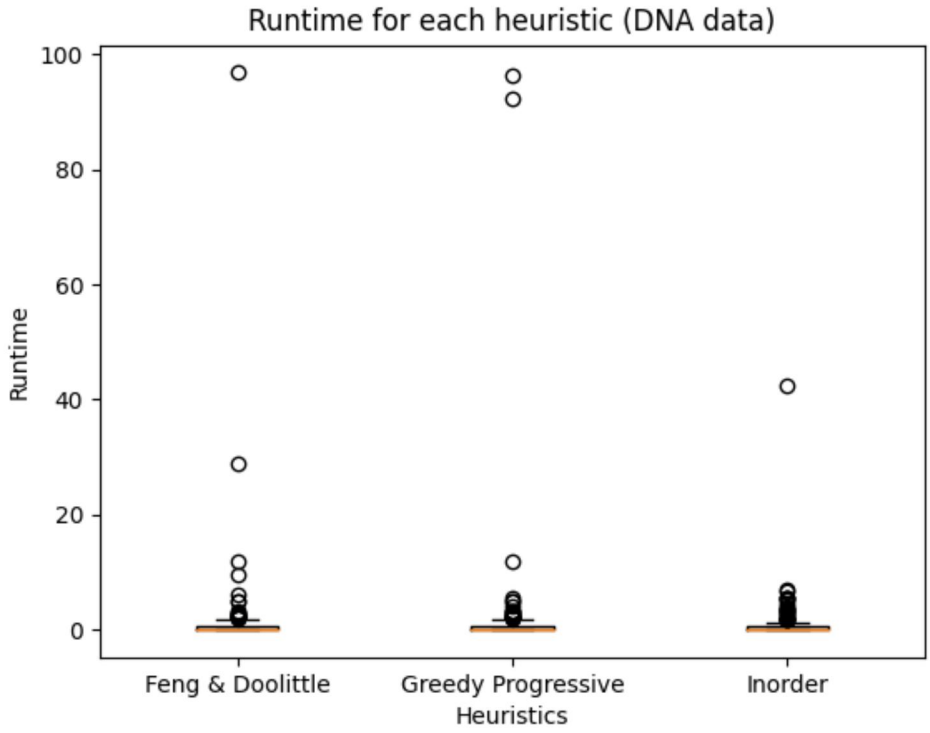


    -
- Analysis:
    - In general, Feng and Doolittle and Greedy algorithm works really well, pruning down more than 95% of the search space
    - Inorder algorithm works well with a small number of sequences and sequence length, matching the performance of other heuristics. However, as the parameters grow large, the inorder algorithm's performance deteriorates.
    - Comparing the Feng and Doolittle and Greedy algorithms, Feng and Doolittle generally prunes more search space. The difference is more evident when the number of sequences and sequence length grow large

<u>Determine which heuristic algorithm makes Carillo-Lipman run fastest (in general cases):</u>

- Method:
    - We used about 150 different sets of DNA sequences, with <u>varying</u> number of sequences and <u>varying</u> sequence length
        - This is chosen because we want to analyze each heuristic's performance in general cases, independent of the magnitude of number of sequences or sequence length
    - The recorded runtime only accounts for Carillo-Lipman's running time, not including the running time of each heuristic algorithm
        - This is because we just want to compare how much each heuristic speeds up the Carillo-Lipman's algorithm

    - We run these inputs through each heuristic algorithm and subsequently passed into Carillo-Lipman, the results are then recorded separately between heuristics
- Plot:

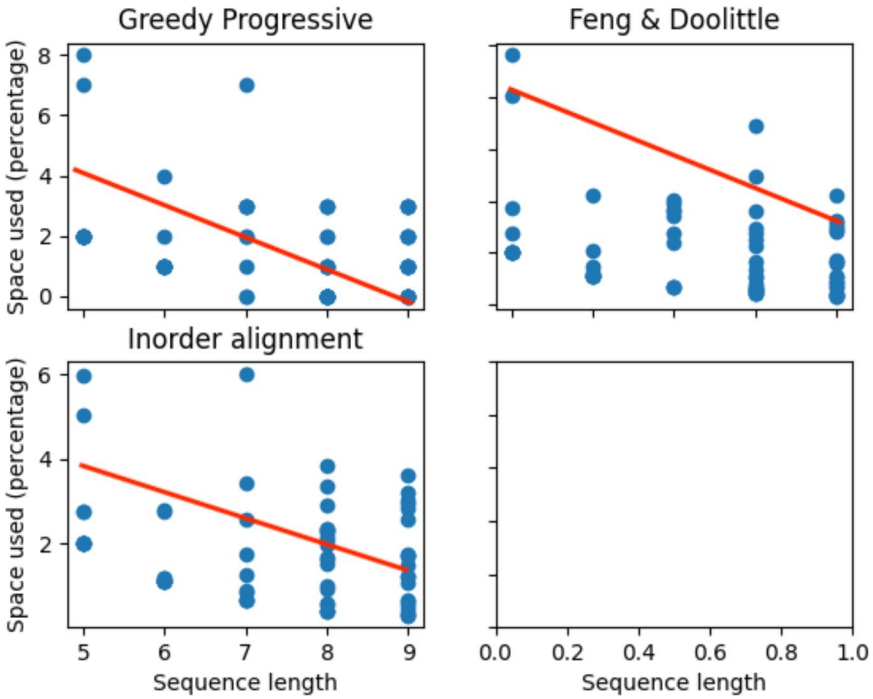


    -
- Analysis:
    - The runtime reflects the same result as the previous ‘Determine which heuristic prune more search spaces’ analysis
    - This is expected because the runtime of Carillo-Lipman should be linearly correlated with the number of search space it has to go through, as suggested by the first analysis

<u>Relationship between length of sequence in the inputs and size of Carrillo-Lipman’s search space:</u>

- Method:
    - We <u>fixed</u> the number of sequences to be 5 and used <u>varying</u> number of sequence length, and used about 70 different sets of DNA sequences
        - We have to fix the number of sequences so that the size of seach space only depends on the sequence length, and not affected by varying number of sequences
        - This number is chosen because it is large enough to differentiate the performance of the 3 heuristics
    - We run these inputs through each heuristic algorithm and subsequently passed into Carillo-Lipman, the results are then recorded separately between heuristics

        - This is because we only want to see the relationship of seach space size and length of sequence, so we must disregard the differences in effectiveness of each heuristic
- Plot:

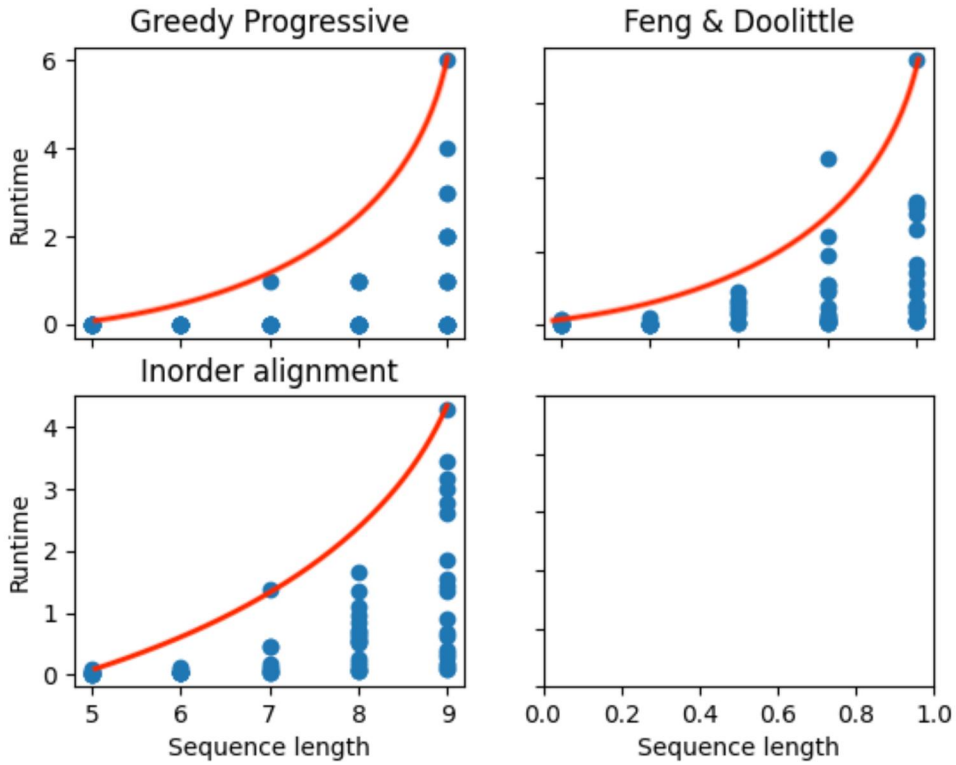


- Analysis:
    - As the number of sequences gets larger, the size of search space gets smaller <u>in percentage.</u>
    - Our analysis is:
        - The number of sequences get larger, the total search space (without pruning) grow exponentially (n^k, n^{k+1},...)
        - However, the heuristic is still working well, giving z-value close to the score of the optimal solution, so the Carillo-Lipman's search space (after pruning) is still small <u>in number.</u>
        - The total search space keeps outgrowing the Carillo-Lipman's search space <u>in percentage</u> as the number of sequences get larger

<u>Relationship between length of sequence in the inputs and runtime of Carrillo-Lipman:</u>

- Method:
    - We <u>fixed</u> the number of sequences to be 5 and used <u>varying</u> number of sequence length, and used about 70 different sets of DNA sequences
        - We have to fix the number of sequences so that the runtime only depends on the sequence length, and not affected by varying number of sequences

        - This number is chosen because it is large enough to differentiate the performance of the 3 heuristics
    - We run these inputs through each heuristic algorithm and subsequently passed into Carillo-Lipman, the results are then recorded separately between heuristics
        - This is because we only want to see the relationship of runtime and length of sequence, so we must disregard the differences in effectiveness of each heuristic
- Plot:



    -
- Analysis:
    - As the number of sequences get larger, the runtime gets larger <u>in numbers</u>
        - Note that this doesn't contradict the previous observation about 'sequence length vs. search space size' as the previous observation is measure <u>in percentage</u>, but this observation is measured <u>in number</u>
        - Meaning that the search space might actually be smaller <u>in percentage</u> as the previous observation suggested, but the total search space is larger, so the search space is larger <u>in number</u>
    - This make sense our search space is getting larger <u>in number</u>, and running time should be linearly correlated with search space size as suggested in the first analysis question

Conclusion:

In conclusion, we see that different heuristics lead to different performance results — the closer the heuristic is to the optimal solution, the more space we pruned, meaning faster runtime. In general, we see that for a large number of sequences, Feng and Doolittle and Greedy algorithms give better heuristics than the naive Inorder algorithm (which is expected), with no substantial differences between the two.

Future Work:

We would like to extend our alignment algorithms and heuristics to work with different alphabets, e.g. the protein alphabet to align sequences of amino acids and other structures.

Another area of improvement could be our runtime. As the sequence length increased or the number of sequences increased we saw an exponential increase in runtime (as expected). A solution to this could be through programming techniques like parallel computing to run multiple heuristics in parallel and get results faster, as the algorithms for each heuristic is independent.

Finally, after we optimize our program to work with a larger number of sequences and sequence length, we could implement file reading functionality that allows us to use real DNA data from FASTA files. This would allow us to run our heuristics to see performance on DNA of real organisms which is more useful than the randomly generated data we use here.