CS238: Homework 3

Pratyay Dutta, SID: 862465552

May 29, 2025

1 Divide and Conquer Multiple Sequence Alignment

1.1 Description of the algorithm:

The content below summarizes the main parts of the algorithm that I have impemented.

- **Objective:** Given three biological sequences A, B, and C, the algorithm computes an optimal global alignment of all three, maximizing the sum-of-pairs alignment score as defined by a user-supplied scoring function. In our case, we use the Blast matrix for scoring [1].
- **3D Dynamic Programming (DP) Recursion:**
  - We define a 3D DP array $S[i, j, k]$, where $S[i, j, k]$ is the optimal alignment score for prefixes $A_{1:i}$, $B_{1:j}$, and $C_{1:k}$.
  - The DP recursion at each cell is given by:

$$S[i,j,k] = \max \Bigg\{ \begin{aligned} & S[i-1,j-1,k-1] + \sigma(A_i, B_j, C_k), \\ & S[i-1,j-1,k] + \sigma(A_i, B_j, -), \\ & S[i-1,j,k-1] + \sigma(A_i, -, C_k), \\ & S[i,j-1,k-1] + \sigma(-, B_j, C_k), \\ & S[i-1,j,k] + \sigma(A_i, -, -), \\ & S[i,j-1,k] + \sigma(-, B_j, -), \\ & S[i,j,k-1] + \sigma(-, -, C_k) \end{aligned} \Bigg\}$$

    where $\sigma(\cdot, \cdot, \cdot)$ is the scoring function for any triplet of residues (or gaps).
- **Base Cases:**
  - The boundary conditions are initialized according to the number of gaps:

$$\begin{aligned} S[0,0,0] &= 0 \\ S[i,0,0] &= S[i-1,0,0] + \sigma(A_i, -, -) \\ S[0,j,0] &= S[0,j-1,0] + \sigma(-, B_j, -) \\ S[0,0,k] &= S[0,0,k-1] + \sigma(-, -, C_k) \end{aligned}$$

  - Similar logic applies for edges with two zero indices (e.g., $S[0, j, k]$).
- **Divide-and-Conquer (DC) Step:** [2]
  - When at least one sequence is long, the alignment is split to conserve memory.
  - Suppose A is split at index $i = \lfloor |A|/2 \rfloor$. Define

$$\begin{aligned} F[j,k] &= \text{Optimal DP score for } (A_{1:i}, B_{1:j}, C_{1:k}) \\ R[j,k] &= \text{Optimal DP score for } (A_{i+1:|A|}, B_{j+1:|B|}, C_{k+1:|C|}) \end{aligned}$$

– The optimal split (j^*, k^*) is found by:

$$(j^*, k^*) = \arg\max_{j,k} [F[j,k] + R[j,k]]$$

– The full alignment is assembled recursively:

$$\text{Align}(A, B, C) = \text{Align}(A_{1:i}, B_{1:j^*}, C_{1:k^*}) \parallel \text{Align}(A_{i+1:|A|}, B_{j^*+1:|B|}, C_{k^*+1:|C|})$$

where $\parallel$ denotes concatenation of the left and right sub-alignments.

- **Threshold for Fallback to Direct DP:**
  - If $\min(|A|, |B|, |C|) \leq \tau$ for some small threshold τ (in our case 15), compute alignment directly using full 3D DP and traceback.
- **Subroutines:**
  - `BASE3DALIGN` (Full 3D DP and Traceback):
    - Implements the standard dynamic programming algorithm for global three-way alignment.
    - Constructs the entire 3D DP table $S[i, j, k]$ as described in the recursion equations above.
    - After filling the table, performs a traceback procedure starting from $S[|A|, |B|, |C|]$ back to $S[0, 0, 0]$ to reconstruct the optimal alignment paths for all three sequences.
    - The traceback at each cell decides which predecessor was used to achieve the current score (e.g., aligning three residues, introducing gaps, etc.), and appends the appropriate character or gap to the alignment strings.
    - Used for small subproblems (where all sequences are below the threshold), as it is space- and time-intensive for long sequences.
  - `FORWARD3DSLICE` (Forward DP Slice for Divide-and-Conquer):
    - Computes the optimal alignment scores for all prefixes of B and C, given a fixed prefix of A (up to the chosen split point i).
    - Rather than storing the full 3D DP array, the algorithm only maintains two 2D arrays of size $|B| \times |C|$: one for the current i-th layer ($S[i, :, :]$) and one for the previous $(i-1)$-th layer ($S[i-1, :, :]$). [6]
    - At each step, the algorithm updates the current layer $S[i, :, :]$ using values from the previous layer $S[i-1, :, :]$ and the DP recurrence. After completing the updates for all j and k, the previous layer can be overwritten or reused for the next iteration.
    - This approach reduces the space complexity from $O(|A||B||C|)$ to $O(|B||C|)$, as only the current and previous slices along A are ever kept in memory.
    - After processing the split index i, the final 2D array $S[i, :, :]$ contains the alignment scores needed for all combinations of $B_{1:j}$ and $C_{1:k}$ against $A_{1:i}$, which are used in the divide-and-conquer step.
  - `BACKWARD3DSLICE` (Backward DP Slice for Divide-and-Conquer):
    - Works analogously to `FORWARD3DSLICE`, but proceeds from the end of A towards the split point.
    - The algorithm again maintains only two 2D arrays of size $|B| \times |C|$ at any time: one for the current layer and one for the next layer along the A dimension.
    - At each iteration, the current layer $S[i, :, :]$ is updated using values from the next layer $S[i+1, :, :]$ according to the DP recurrence.
    - This in-place update strategy ensures that the space complexity remains $O(|B||C|)$, as only two layers are stored, irrespective of the length of A.
    - When the split point is reached, the resulting 2D array provides the necessary suffix alignment scores for all combinations of $B_{j:}$ and $C_{k:}$ against $A_{i+1:}$.

1.2 Helper Functions

We define a few helper functions to preprocess, clean and modify the dna sequence data in a fasta format for our algorithm. My Aligner class takes in a fasta file containing three sequences and reads from it to get the optimal alignment using the previously defined algorithm. We also define a function for the scoring matrix. The functions are defined and explained as below.

- `dnagen(raw_text: str, output_file: str = "sequence.txt")`
  - Converts a block of raw DNA text (with or without line numbers) into a continuous DNA sequence and saves it as a text file.
  - **Steps:**
    - Splits the input raw text into individual lines.
    - For each line, removes any leading line number and concatenates the DNA letters.
    - Converts the complete sequence to lowercase (can be changed to uppercase as needed).
    - Writes the resulting sequence into the specified output text file.
- `fastagen(seq_files, output_fasta="example.fasta")`
  - Combines three individual DNA sequence text files into a single FASTA file suitable for multiple sequence alignment.
  - **Steps:**
    - Checks that exactly three input files are provided; raises an error otherwise.
    - For each input file:
      - Reads the DNA sequence, removing any whitespace and newlines.
      - Writes the sequence to the output FASTA file in standard FASTA format with headers `>seq1`, `>seq2`, and `>seq3`.
- `blast_sigma(a, b, c)` [1]
  - Defines the scoring function for triplet sequence alignment based on a simple pairwise DNA scoring scheme.
  - **Steps:**
    - For every pair in (a, b, c), assigns a score:
      - 0 for a pair of gaps.
      - -8 for a gap and a nucleotide.
      - $+5$ for a match (same nucleotide).
      - -4 for a mismatch (different nucleotides).
    - Returns the sum of scores for all three pairs: (a, b), (a, c), and (b, c).
- **Numba Acceleration for Computational Efficiency** [7]
  - The core dynamic programming routines for three-way alignment are implemented using the `Numba` JIT (Just-In-Time) compiler to significantly speed up computation.
  - Key functions accelerated with Numba include:
    - `numba_base3d_align`: Performs the full three-dimensional DP table filling and traceback for small subproblems.
    - `numba_forward3dslice`: Efficiently computes a 2D DP slice for the divide-and-conquer approach, maintaining only two 2D arrays at any time.
    - `numba_backward3dslice`: Computes the reverse DP slice, implemented by running the forward slice algorithm on reversed input sequences and flipping the output accordingly.
  - By decorating these functions with `@njit`, the Python code is compiled to fast machine code at runtime, eliminating the interpreter overhead and greatly reducing execution time.
  - Numba-accelerated routines work directly with NumPy arrays, ensuring efficient memory access and computation within tight loops, which is critical for large-scale DP table computations.
  - This acceleration allows both full dynamic programming and divide-and-conquer computations to efficiently handle longer sequences and larger DP tables, making the algorithm practical for real biological data even in pure Python.

1.3 Analysis

1.3.1 Time Complexity

The three-way sequence alignment algorithm constructs a 3-dimensional dynamic programming (DP) table for sequences of lengths m, n, and k. The recurrence requires evaluating every possible combination of indices $i \in [0, m]$, $j \in [0, n]$, $k \in [0, k]$. At each cell, the score is computed by considering up to 7 possible transitions.

- The naive dynamic programming approach (`Base3DAlign`) fills the entire DP table, leading to a time complexity of

  $$O(mnk)$$

  since there are $m \times n \times k$ entries, each filled in constant time.

- The divide-and-conquer version (`ThreeWayAlign`) recursively divides the problem along the longest sequence (say, A of length m), splitting it into two subproblems of roughly $m/2$ and using 2D DP slices over n and k at each level.

- At each recursion level, two 2D DP slices (`Forward3DSlice` and `Backward3DSlice`) are computed, each with time complexity $O(mnk)$, but these are only over the smaller split range in A.

- The overall recurrence for time complexity $T(m, n, k)$ can be written as:

  $$T(m, n, k) = O(nkm) + T(m_1, n_1, k_1) + T(m_2, n_2, k_2)$$

  where $m_1 + m_2 = m$, $n_1 + n_2 = n$, $k_1 + k_2 = k$ (depending on split points).

- In the worst case, the divide-and-conquer recursion produces a logarithmic number of levels with respect to the longest sequence, so the **overall time complexity** remains:

  $$O(mnk)$$

  Thus, while the divide-and-conquer approach improves space usage, the asymptotic time complexity is the same as the full 3D DP.

1.3.2 Space Complexity

- The standard DP approach requires storing the entire 3D table, which incurs a space complexity of:

  $$O(mnk)$$

  This becomes quickly prohibitive for long sequences.

- The divide-and-conquer implementation improves space efficiency by only retaining two 2D DP slices ($n \times k$ arrays) at each recursion level for both forward and backward passes, and by constructing alignments recursively.

- At any given recursion level, the memory usage is dominated by the two $n \times k$ arrays (one for the current slice and one for the previous/next), and the memory for storing the partial alignments (which is $O(m + n + k)$ for the final result).

- The **overall space complexity** at any given time is therefore:

  $$O(nk + m + n + k)$$

  where nk comes from the DP slices and $m + n + k$ from the alignment output.

- In practice, this is a significant reduction compared to the naive $O(mnk)$ requirement and enables alignment of much longer sequences.

1.4 Results

The results of the three-way multiple sequence alignment algorithm were evaluated on five benchmark DNA datasets. For each dataset, the following metrics were recorded:

- **Optimal Alignment Score:** The total alignment score is computed using the defined scoring function, summing over all columns of the final aligned sequences.
- **Alignment Length:** The length of the output alignment, defined as the number of columns in the aligned sequences (including any gap symbols). This represents the total length after introducing all necessary gaps to achieve the optimal alignment.
- **Number of Perfect Matches:** The number of alignment columns in which all three nucleotides are identical and are not gaps. This is a direct measure of conserved positions across the three sequences. Formally, this counts columns where $A_i = B_i = C_i$ and $A_i \neq$ -.
- **Running Time:** The total wall-clock time required to compute the optimal alignment, measured in seconds. This parameter was recorded using system time before and after the alignment process.
- **Memory Utilization:** The peak memory usage during the alignment process, measured in megabytes (MB). To obtain this value, a separate monitoring thread periodically queries the process's resident set size (RSS) at short intervals throughout the computation [3], recording the highest value observed. This approach captures the true maximum memory consumed at any point during alignment, including transient peaks that may not be visible by measuring only before and after execution.
- **Conserved Regions**: Inspection of the alignment reveals that several consecutive columns are perfectly conserved, indicating a highly conserved region among the sequences.

Each of these parameters was extracted directly from the output of the alignment program:

- The alignment score, alignment length, and number of perfect matches are calculated from the alignment strings produced by the algorithm.
- Running time is measured by recording timestamps before and after the alignment routine.
- Memory usage is measured by querying the process memory before and after execution, and computing the difference.

Table 1 summarizes these results for the five datasets.

Table 1: Results of Three-Way Multiple Sequence Alignment on Five Datasets

Dataset	Alignment Score	Alignment Length	Perfect Matches	Running Time (s)	Peak Memory Usage (MB)
Haemoglobin Alpha	5284	592	425	4.68	144.14
Homo sapiens hepatocyte	18992	1785	1505	32.39	173.08
DAPK2	3427	2793	1417	59.07	204.49
Hepatic nuclear factor 4	39908	4895	3371	809.94 $\approx$ 13 mins	408.45
Cystic fibrosis	55591	6457	4523	2015.78 $\approx$ 34 mins	633.11

1.5 Device Specifications

All experiments were carried out on a personal computer with the following specifications:

- **Processor:** AMD Ryzen Threadripper PRO 3955WX
- **RAM:** 48GB
- **Operating System:** Ubuntu 22.04.5 LTS
- **Python Version:** Python 3.10.12
- **Processor Architecture:** x86_64
- **Number of Logical Cores:** 16

2 Analysis and Speedup of the Spliced Alignment Algorithm

2.1 Original Spliced Alignment Algorithm

- **Input:**
  - Genomic sequence v of length n
  - cDNA sequence w of length m
  - Scoring parameters for match, mismatch, gap penalties, and special penalties for splice junctions
- **Dynamic Programming Table:**
  Let $S(i, j)$ denote the best alignment score for the first i letters of v and the first j letters of w.
- **Recurrence Relation:**

$$S(i,j) = \max \begin{cases} S(i-1, j-1) + \text{score}(v_i, w_j), \\ S(i-1, j) + \text{gap penalty}, \\ S(i, j-1) + \text{gap penalty}, \\ \max\limits_{k<i} S(k, j) + \text{splice penalty} \end{cases}$$

  Here, the last term corresponds to spliced alignment: introducing a long gap (putative intron) between positions k and i in the genome, aligned to a contiguous segment in cDNA [5, 4].
- **Drawback:**
  - For each (i, j), all previous exon boundaries $k < i$ must be considered, leading to $O(n)$ operations per DP cell.
  - The DP graph is dense, with $O(n^2)$ edges per row and $O(n^2 m)$ total edges.
  - For every cell (i, j) in the dynamic programming table, **all previous positions** $k < i$ in the genome must be considered as potential exon boundaries.
  - **Biological motivation:** An exon can end at any previous position, and an intron (splice) may start from there.
  - To compute $S(i, j)$, you must check:

$$\max_{k<i} S(k, j) + \text{splice penalty}$$

  - For each i, there are up to $i - 1$ possible choices for k. Therefore we have to compute $O(n)$ computations for every cell.

2.2 Motivation for Improvement

- The major inefficiency arises from the need to check all possible exon boundaries for every DP cell.
- Most possible boundaries are biologically implausible, and a sparse representation of the alignment graph suffices [8].

2.3 Improved Algorithm: Sparse Spliced Alignment Graph

- **Key Idea:**
  - Precompute the best scores for all plausible exon block boundaries using an auxiliary function $P(i, j)$.
  - Only consider biologically likely boundaries (e.g., based on donor/acceptor site motifs or annotated blocks).
- **Definition:**

$$P(i,j) = \max_{\text{all blocks } B \text{ preceding } i} S(\text{end}(B), j, B)$$

  where $S(\text{end}(B), j, B)$ is the alignment score up to the end of block B. This auxiliary function, $P(i, j)$:

- collects, for every plausible exon block B ending before i, the alignment score up to the end of B at position j in cDNA, and takes the maximum of all these scores.
- Thus, $P(i, j)$ tells you: *"What's the best score I can have at (i, j) if I have just started a new exon at position i?"*
- This is kept track of on the fly during the computation of each cell.

- **Improved Recurrence:**

$$S(i,j) = \max \begin{cases} S(i-1, j-1) + \text{score}(v_i, w_j), \\ S(i-1, j) + \text{gap penalty}, \\ S(i, j-1) + \text{gap penalty}, \\ P(i,j) + \text{splice penalty} \end{cases}$$

  $P(i, j)$ can be computed efficiently if the list of plausible block boundaries is precomputed or bounded in size.

- **Efficiency:**
  - The value of k is the maximum number of plausible exon block boundaries considered at any position i.
  - **Upper bound:** In the worst case, if every position is considered plausible, $k \leq n$, where n is the length of the genomic sequence. However, in practice, k is much smaller and often treated as a constant, $k = O(1)$, due to biological constraints (e.g., known splice sites or annotated boundaries).

2.4 Complexity Analysis

- **Original Algorithm:**
  - Time complexity: $O(n^2m)$, since each of the nm cells may examine up to n previous exon boundaries.
  - Space complexity: $O(nm)$, to store the DP table.

- **Improved Algorithm:**
  - Time complexity: $O(knm)$, where k is the maximum number of plausible boundaries per position (often $O(1)$). For practical purposes, this is $O(nm)$.
  - Space complexity: $O(nm)$.

2.5 Comparison and Conclusion

- The original spliced alignment algorithm requires $O(n^2m)$ time due to dense connections in the DP graph.
- By transforming the alignment graph into a sparse structure that only tracks plausible block boundaries, the improved algorithm achieves $O(nm)$ time.
- This transformation yields a dramatic speedup, making it feasible to align cDNA sequences to large genomic regions efficiently [8, 5, 4].

3 Evolutionary Tree reconstruction from Distance Matrix

Given distance matrix:

	A	B	C	D	E
A	0	5	7	6	9
B	5	0	8	5	8
C	7	8	0	9	12
D	6	5	9	0	7
E	9	8	12	7	0

Step 1: First Integer Hanging Edge Reduction

Step 1: Since we find no degenerate triplets in this matrix, we find minimum nonzero off-diagonal: 5.
So, largest integer δ_1 such that $2\delta_1 \leq 5$ is $\delta_1 = 2$ (since $2 \times 2 = 4$).

Step 2: Subtract $2\delta_1 = 4$ from all off-diagonal entries:

	A	B	C	D	E
A	0	1	3	2	5
B	1	0	4	1	4
C	3	4	0	5	8
D	2	1	5	0	3
E	5	4	8	3	0

Step 2: Removing degenerate triplets

Step 2.1: A triplet (i, j, k) is said to be *degenerate* if the triangle inequality holds as an equality for all permutations:

$$D(i, j) + D(i, k) = D(j, k)$$

or equivalently, the distances among (i, j, k) satisfy:

$$D(i, j) + D(i, k) = D(j, k), \quad D(i, k) + D(j, k) = D(i, j), \quad D(i, j) + D(j, k) = D(i, k)$$

whenever any of these equalities hold exactly.
For example, after reduction, consider triplets such as (A, C, E), (A, B, D), and (A, E, D). For (A, C, E):

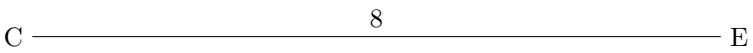
$$D'(C, A) + D'(A, E) = D'(C, E) \implies 3 + 5 = 8$$

thus (C, A, E) is degenerate with central element A. Similarly, we find (A, B, D) and (A, D, E) are degenerate with central elements B and D respectively.

Step 2.2: Remove A, B, D from the matrix. Remaining submatrix is on (C, E):

	C	E
C	0	8
E	8	0

Step 3: Forming C and E connection

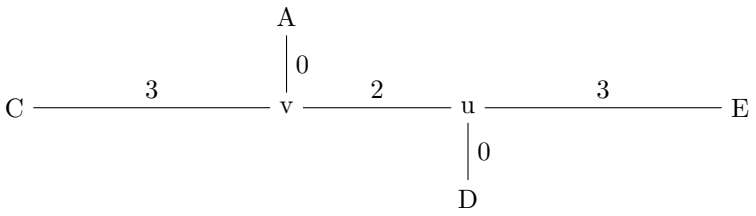


Step 4: Inserting D

From reduced matrix, $DE = 3$ and $CD = 5$. Therefore, node u inserted like so:

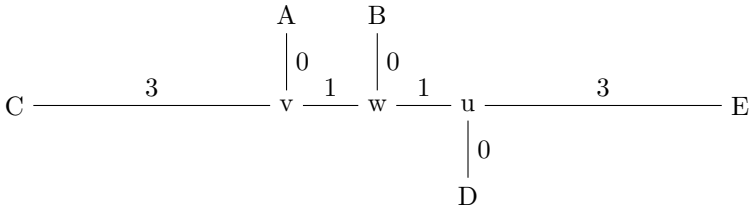
Step 5: Inserting A

From reduced matrix, $AC = 3$ and $AE = 5$. Therefore, node v inserted like so:



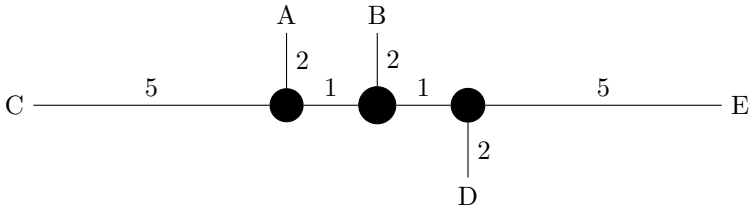
Step 6: Inserting B

From reduced matrix, $AC = 3$ and $AE = 5$. Therefore, node w inserted like so:



Step 7: Adding $\delta = 2$ to all edges

Since we reduce the original matrix by 2δ, we add δ to all edges to complete the phylogenetic tree like so:



References

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] Alberto Apostolico and Daniel S. Hirschberg. Optimal alignment of two strings under space constraints. *Journal of the ACM*, 32(3):646–662, 1985.

[3] Giacomo Barbieri, Jay Loden, and contributors. psutil: Cross-platform process and system utilities. https://psutil.readthedocs.io/, 2024. Accessed: 2024-05-27.

[4] C.B. Burge and S. Karlin. Prediction of complete gene structures in human genomic dna. *Journal of Molecular Biology*, 268(1):78–94, 1997.

[5] M.S. Gelfand, A.A. Mironov, and P.A. Pevzner. Spliced alignment: A new approach to gene recognition in genomic sequences. *Nucleic Acids Research*, 24(17):3623–3630, 1996.

[6] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[7] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. https://numba.pydata.org/, 2015. Accessed: 2024-05-27.

[8] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, Cambridge, MA, 2000.