Parallelization of Needlemam-Wunsch Sequence Alignment Algorithm

Phillip Le (pnl2111) and Emily Lo (ejl2192)

Coms 4995 Parallel Functional Programming

Final Project Proposal

Introduction

In bioinformatics, sequences of nucleotides and proteins are compared to better understand differences in DNA or protein sequences and their effects. One algorithm used to compare genetic sequences is the Needlemam-Wunsch (NW) algorithm also referred to as the optimal matching algorithm or the global alignment technique. The objective is to align two sequences by accepting matches, misalignments and inserting gaps into the sequence. Accepting misalignments and inserting gaps have associated penalties whereas accepting matches increases the total score, which the algorithm tries to maximize. Genetic sequencing is often utilized to gain insight into the role played by sequences within an organism's variability and to discern the mutations responsible for forming these specific sequences.

Needlemam-Wunsch (NW) algorithm

This is a dynamic programming algorithm that begins by initializing an (m+1) x (n+1) 2D array called A, m being the length of one string and n being the length of the next. The (0,0) tile is set to 0. We also have a scoring scheme that allows us to standardize penalties and points for gaps and misalignments in the strings. Then we begin populating A inside a double for loop, i tracking the row index and j tracking the column index, each tile contains the maximum of three cases.

- Case 1: A[i-1][j-1]+ penalty for a mismatch
- Case 2: A[i-1][j] + penalty for a gap
- Case 3: A[i][j-1] + penalty for a gap

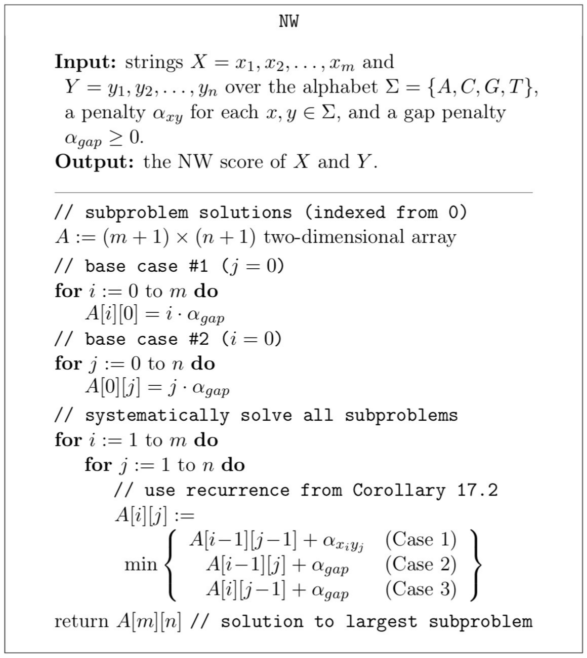
Once A is fully populated, we start from the bottom-right corner of A, and follow the greatest path to the top-left corner. By examining the path between these tiles we can determine the optimal sequences.

- A diagonal arrow indicates either a match or a mismatch, so no change should be made to either sequence.
- Horizontal or vertical arrows indicate gaps should be added. Vertical arrows indicate a gap with the letter of the sequence represented on the side of the A whereas horizontal arrows indicate a gap with the letter of the sequence represented on the top of A.
- In cases with multiple arrow options, it implies a branching of alignments. If these branches contribute to paths from the bottom right to the top left cell, each path is a valid alignment. Each of these paths can be treated as distinct alignment candidates.

Here's the pseudocode + graph!

Scoring scheme:

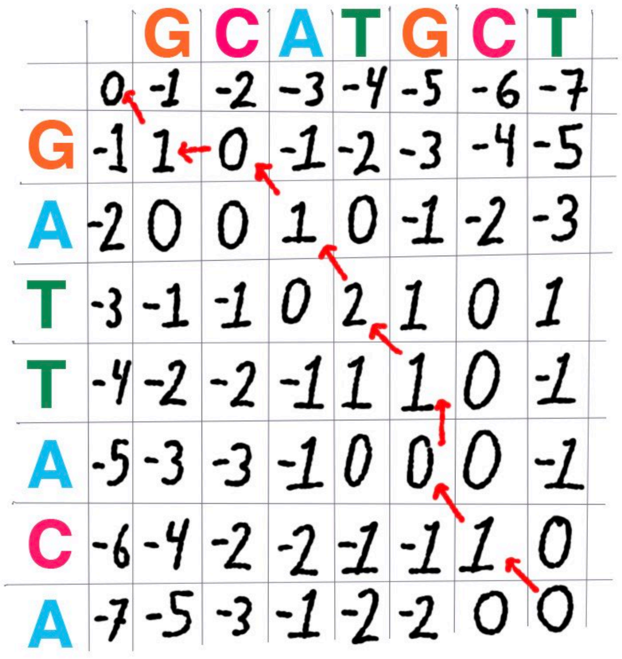
- $+1$ same letter
- $-1 = \alpha_{gap}$
- 0 for mismatch $= \alpha_{x_i y_i}$



```
NW

Input: strings $X = x_1, x_2, \ldots, x_m$ and
 $Y = y_1, y_2, \ldots, y_n$ over the alphabet $\Sigma = \{A, C, G, T\}$,
 a penalty $\alpha_{xy}$ for each $x, y \in \Sigma$, and a gap penalty
 $\alpha_{gap} \geq 0$.
Output: the NW score of $X$ and $Y$.

// subproblem solutions (indexed from 0)
$A := (m+1) \times (n+1)$ two-dimensional array
// base case #1 ($j = 0$)
for $i := 0$ to $m$ do
    $A[i][0] = i \cdot \alpha_{gap}$
// base case #2 ($i = 0$)
for $j := 0$ to $n$ do
    $A[0][j] = j \cdot \alpha_{gap}$
// systematically solve all subproblems
for $i := 1$ to $m$ do
    for $j := 1$ to $n$ do
        // use recurrence from Corollary 17.2
        $A[i][j] :=$
            $\min \left\{ \begin{array}{ll} A[i-1][j-1] + \alpha_{x_i y_j} & \text{(Case 1)} \\ A[i-1][j] + \alpha_{gap} & \text{(Case 2)} \\ A[i][j-1] + \alpha_{gap} & \text{(Case 3)} \end{array} \right\}$
return $A[m][n]$ // solution to largest subproblem
```



		G	C	A	T	G	C	T
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Parallelization

The parallelization of this algorithm is rather simple. We break the dynamic programming matrix, A in our previous example, into blocks or chunks. Each block can be solved independently. For example, the matrix can be split into four quadrants (top-left, top-right, bottom-left, and bottom-right). First, the top-left would have to be computed, then the top-right and bottom-left at the same time, and finally, the bottom-right. The order of computations important because the algorithm compares the A[i-1][j-1] score – thus this order allows us to maintain that.

References

Analysis of Algorithms class at Columbia. Professor Tim Roughgarden
T. Roughgarden. 2021.

Algorithms Illuminated (Part 3): Greedy Algorithms and Dynamic Programming.