CS 2427 - Algorithms in Molecular Biology

Lecture #9: 8 February 2006

Lecturer: Michael Brudno

Scribe Notes by: Philipp Hertel

1 Today's Topics:

- Reduction from LCS to LIS
- BLAST
- LAGAN

2 Longest Increasing Subsequence and Sparse LCS

Today we consider a modification of the Needleman/Wunsch LCS algorithm which can be analysed in terms of the number of matches between two sequences A and B, rather than their lengths. In order to define an LCS algorithm whose complexity can be analysed in terms of matches, we first define the Longest Increasing Subsequence (LIS) problem and define an algorithm for solving it.

Longest Increasing Subsequence (LIS)
Input: A list L of numbers.
Output: The longest monotonically increasing subsequence S of L.

For example, given $L = (1, 4, 2, 3, 7, 5, 9)$, we want to output either $S = (1, 2, 3, 7, 9)$ or $S = (1, 2, 3, 5, 9)$. This problem can easily be solved in quadratic time using the array A where $A(i)$ is the length of the longest increasing subsequence of L that ends with $L(i)$. We can compute A as follows:

1. **For** $1 \leq i \leq n$ **Do**
2. $\quad A(i) = 1 + max\{A(j) | 1 \leq j \leq i \text{ and } L(j) \leq L(i)\}$
3. **End For**

We can easily keep track of the actual list S by keeping a separate array of the same cardinality as A, in which we can store the index j returned by the max operation for each cell of A. This time can be improved to $O(n \lg n)$ by using a totally different dynamic program. It uses an array M where $M(i)$ will contain the last element of the longest increasing subsequence of length i which has the smallest last element. M's maximum size will be $|L|$ if L starts as a sorted array of unique elements.

Consider $L = (1, 4, 2, 3, 7, 5, 9, 4)$, we fill in M as follows:

- $t = 0$

  $M =$ | · | · | · | · | · | · | · | · |

- $t = 1$, $L(1) = 1$

  $M =$ | 1 | · | · | · | · | · | · | · |

  Since M is empty 1 must be the smallest element of any choice for S so far.

- $t = 2$, $L(2) = 4$

  $M =$ | 1 | 4 | · | · | · | · | · | · |

  Since $4 \geq 1$, 4 is made the current successor of 1, reflecting that so far $(1, 4)$ is the ordered sequence of length 2 with the smallest last element (since it is the only such sequence).

- $t = 3$, $L(4) = 2$

  $M =$ | 1 | 2 | · | · | · | · | · | · |

  Since $2 < 4$, 2 replaces 4 in M, relecting that $(1, 2)$ is the ordered sequence of length 2 with the smallest last element since the only other such sequence so far has been $(1, 4)$.

- $t = 4$, $L(4) = 3$

  $M =$ | 1 | 2 | 3 | · | · | · | · | · |

  Since $2 < 3$, we append 3 to the end of M.

- $t = 5$, $L(5) = 7$

  $M =$ | 1 | 2 | 3 | 7 | · | · | · | · |

  Since $3 < 7$, we append 7 to the end of M.

- $t = 6$, $L(6) = 5$

  $M =$ | 1 | 2 | 3 | 5 | · | · | · | · |

  Since $3 < 5$ and $5 < 7$, we replace 7 with 5 in M.

- $t = 7$, $L(7) = 9$

  $M =$ | 1 | 2 | 3 | 5 | 9 | · | · | · |

  Since $5 < 9$, we append 9 to the end of the array.

- $t = 8$, $L(8) = 4$

  $M =$ | 1 | 2 | 3 | 4 | 9 | · | · | · |

  Since $3 < 4$ and $4 < 5$, we replace 5 with 4 in M.

Inductively, we can see that M will always be a sorted array. The basis is trivial because an array containing a single element is sorted. For the induction step, if the current element is larger than every other element in the array so far, we add it to the end of the array. Clearly this keeps the array in sort order. If the current element is not larger than every other element, we replace

another element x in the array with it. The element x is larger than the current element, and x's predecessor is smaller than the current element. Inductively we can assume that M was in sort order before we put the current element in, so all of x's successors must be larger than x, so replacing x with the smaller current element will keep M sorted.

At each step we need to find where in M the current element belongs. Since the current element can be much smaller than the last one, we need to search through M to find this location. Since M is always sorted, we can use binary search to find the location. This is where the $\lg n$ comes into the algorithm's running time of $O(n \lg n)$.

At the end of the process the last element of the LIS can be found in the last non-empty slot of M. This tells us that the LIS of L must have length $|M|$. If we want to return the actual LIS rather than just its length, we can keep another table of backpointers which will point to the index of the current elements predecessor in L.

We can now reduce the LCS problem to LIS so that we can solve LCS in time $O(k \lg k)$ where k is the number of matches in the LCS instance . This reduction is discussed in Section 12.4 and 12.5 of [Gus97]. Consider Figure 1 which shows the matches between two DNA strings, $X = ATGCTA$ and $Y = ATAGC$. The numbers in the left-hand column represent the positions of symbols in the vertical string Y. We can use these indices to rank where matches occur in the Figure 1. We want to know which positions in Y match with $X(1)$. From the first column of the table we can see that $Y(1)$ and $Y(3)$ both match with $X(1)$. We therefore include the indices 1 and 3 in our LIS string, but since $X(1)$ can only match either to $Y(1)$ or $Y(3)$, we add the indices into the LIS string in descending order, that way choosing 1 will rule out 3 since it means we skipped it, and choosing 3 rules out 1 since $1 < 3$. We do this for each successive position in X. Essentially, for each column we include the Y indices of the matches into the LIS string starting from the bottom of the column. The LIS string corresponding to the matches of Figure 1 is therefore $(3, 1, 2, 4, 5, 2, 3, 1)$. It is not hard to see that the LIS of this sequence corresponds to the LCS of the two orignal DNA sequences.

		A	T	G	C	T	A
1	A	×					×
2	T		×			×	
3	A	×					×
4	G			×			
5	C				×		

Figure 1: *Table of matches between two DNA strings.*

Since the expected number of matches between DNA strings is $O(nm)$, this algorithm will not perform as quickly as the standard Needleman/Wunsch algorithm for LCS. This algorithm is more suited to solving Sparse LCS, *i.e.* LCS in strings between which we can expect few matches.

3 BLAST

BLAST is a fast heuristic for finding the optimal local alignment between two sequences X and Y. BLAST is an acronym for *Basic Local Alignment Search Tool*. Though there are no theoretical guarantees about BLAST's accuracy, it seems to work well in practice. We will use the following notation. X will have length n and Y will have length m. Also X_j^i, where $1 \leq i \leq j \leq n$ will refer to the substring of X from index i to index j. BLAST proceeds in three phases. Before describing the three phases we introduce the concept of a dictionary which BLAST uses as a look-up table.

Let Σ be a fixed alphabet and let k be a constant. Using only the symbols in Σ we can create exactly $|\Sigma|^k$ different strings of length k. Each of these strings can be viewed as a natural number encoded in base $|\Sigma|$. For DNA Σ is $\{A, T, C, G\}$ and strings of length k correspond to base 4 numbers of length k. For example, if $A = 0$, $T = 1$, $C = 2$, and $G = 3$, then the string $GGTAC$ corresponds uniquely to the decimal number:

$$(3 \times 4^4) + (3 \times 4^3) + (1 \times 4^2) + (0 \times 4^1) + (2 \times 4^0) = 768 + 192 + 16 + 0 + 2 = 978$$

If we are interested in identifying all the substrings of length k in a given string X, we can create a hash table D with exactly $|\Sigma|^k$ slots. Length k substrings of X will then map uniquely to the slot in D with the address which corresponds to their base 4 numbers. This is an $O(1)$ operation. D can be thought of as a dictionary for the pair (Σ, k). We can use it to count all the occurrences of substrings in X.

BLAST is given two sequence X and Y over an alphabet Σ as well as a parameter k and a dictionary for (Σ, k). It then proceeds through the following three phases.

1. In phase 1, BLAST linearly traverses X. At index i, BLAST adds X_{i+k}^i to the dictionary D and labels the entry as occurring in X at index i. If more than one index of X maps to the same slot of D, then they are both remembered using chaining. See Figure 2 which shows an example of the first phase of BLAST for $X = AAGGGTAGG$ with $k = 3$.

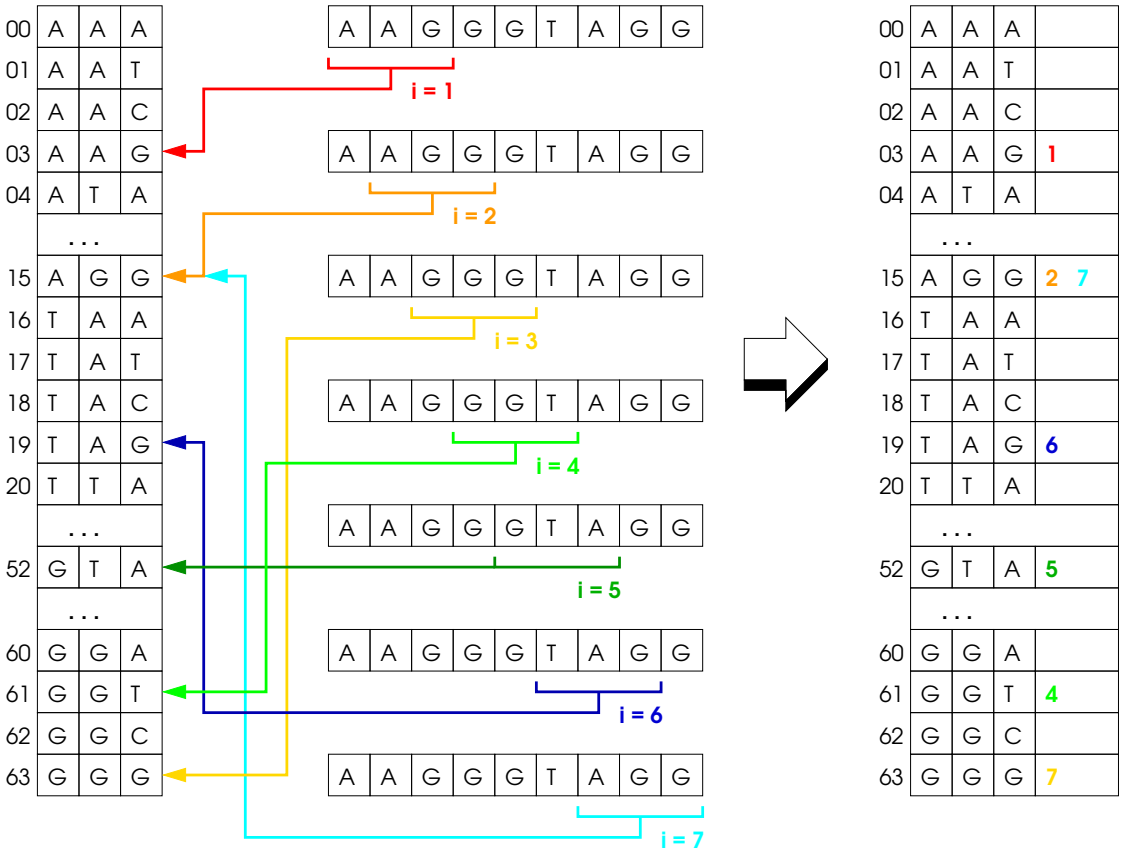

Figure 2: Phase 1 of BLAST, building a dictionary for $X = AAGGGTAGG$ with $k = 3$.

2. When phase 1 completes, BLAST begins to traverse Y similarly, except that at index j, Y_{j+k}^{j} is not entered into the dictionary. Rather, it is checked against the dictionary to see if the same substring is already in D. If so D returns a list of locations in X where Y_{j+k}^{j} occurred. After we finish with Y, we'll have a list of all the locations where X and Y align perfectly on any substrings of length k.

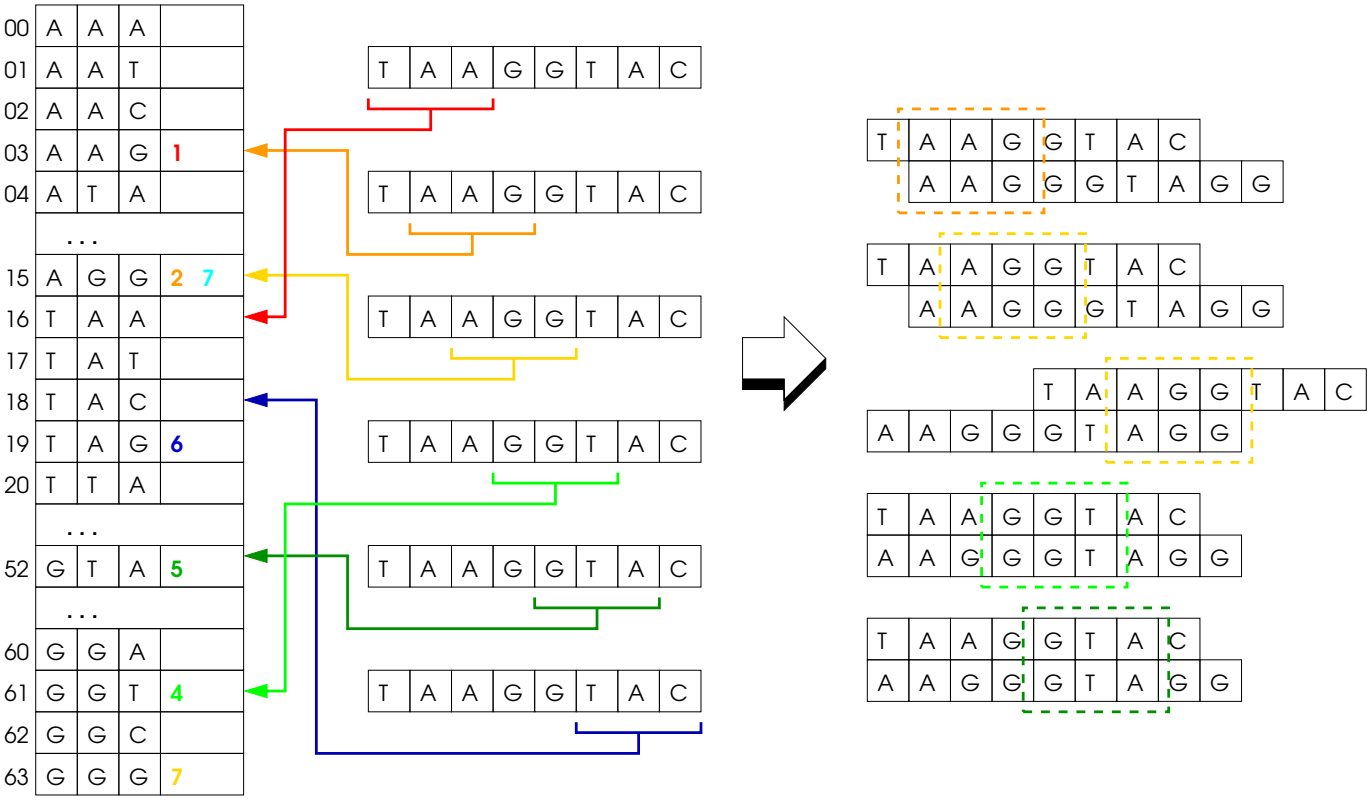

Figure 3: Phase 2 of BLAST, finding length 3 matches between $Y = TAAGGTAC$ and $X = AAGGGTAGG$.

3. In phase 3, BLAST tries to extend each of the matches found in phase 2 using the Smith Waterman recurrence. Each of these extensions is returned as a potential local alignment. Different implementations of BLAST differ on how they extend the matches. Some extend them linearly as in Figure 4a, in which case every position is considered either a match or a mismatch and no gaps are considered. Others extend the matches using the Smith Waterman recurrence as depicted in Figure 4b.

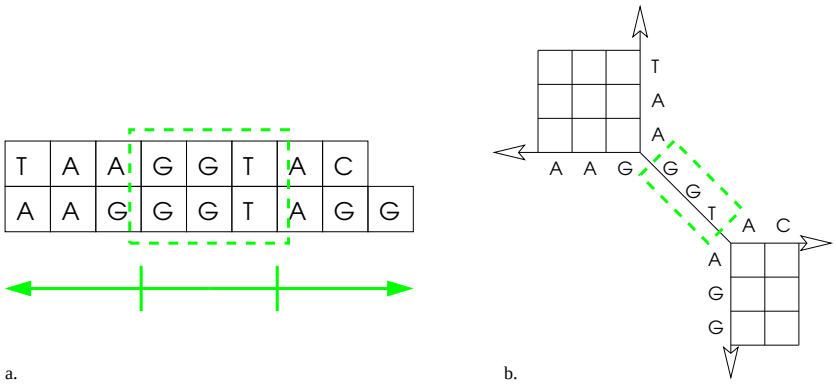

Figure 4: Phase 3 of BLAST, **a)** extend each match linearly or **b)** extend each match using Smith Waterman.

BLAST was conceived by Altschul, Gish, Miller, Myers, and Lipman. It is one of the most popular computational tools used in by biologists. The original BLAST paper [AGM+90] is the most cited bioinformatics paper of the 1990s.

4 LAGAN

The LAGAN heuristic uses a combination of local alignment, sparse LCS and limited area dynamic programming to find a global alignment between two strings X and Y. Like BLAST, LAGAN provides no guarantees on the accuracy of its output but it seems to work well in practice. LAGAN proceeds in three phases. LAGAN was introduced in the paper [BDC$^+$03].

1. In phase 1, LAGAN finds the local alignments between X and Y. This is done using fast techniques such as BLAST. Since we are trying to solve global alignment which can be computed in $O(mn)$ time, it would not make sense to use Smith Waterman for this stage, since it also has $O(mn)$ time.

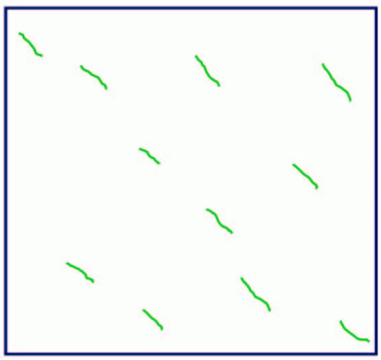

Figure 5: Phase 1 of LAGAN, find local alignments using fast heuristics.

2. In phase 2, LAGAN decides which of the local alignments should be pieced together in order to form a global alignment between X and Y. This step is based on the sparse Longest Common Subsequence algorithm, as each local alignment is treated as a "match" between the sequences (there will be a homework problem on how exactly this is done).

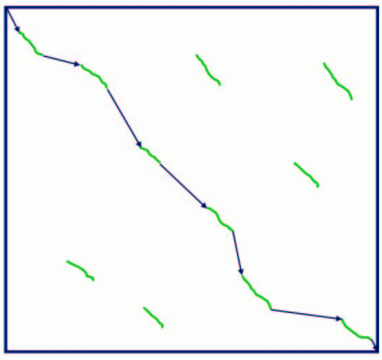

Figure 6: Phase 2 of LAGAN, chain local alignments.

3. In phase 3, once LAGAN has decided which local alignments will be chained together, dynamic programming is used between the endpoints of the local alignments and around them to create a single global alignment. LAGAN uses the standard Needleman-Wunsch global alignment algorithm, but only calculates the valuesof cells within the limited area specified in Figure 7.

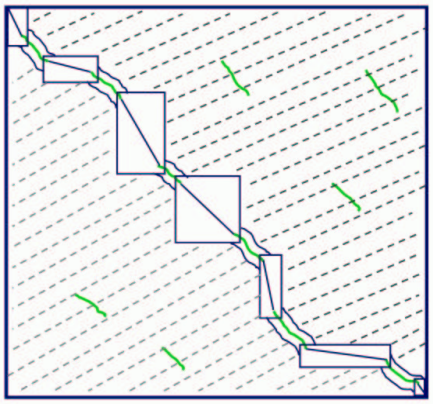

Figure 7: Phase 3 of LAGAN, compute the global alignment piecewise.

References

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[BDC+03] M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, E. D. Green, A. Sidow, and S. and Batzoglou. Lagan and multi-lagan: efficient tools for large-scale multiple alignment of genomic dna. *Genome Res*, 13(4):721–731, April 2003.

[Gus97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, 1997.