CS 395T: Algorithms for Computational Biology

Lecture 12: February 21, 2008 Lecturer: Tandy Warnow

Scribe: Rahul Suri

12.1 Course Logistics

Paper Presentations. A list of publications on phylogenetic networks has been posted to the course website at `http://www.cs.utexas.edu/~tandy/395T-2008.html`. Each student should select a paper to present to the class and notify Professor Warnow of this selection by e-mail prior to next Tuesday's lecture.

Scribe Notes. Beginning with today's lecture, a designated scribe will be responsible for taking notes on each lecture, typesetting a polished and thorough version of the notes in LaTeX, e-mailing the notes in both `.tex` and `.pdf` formats to Professor Warnow, and revising the notes to incorporate the professor's feedback.

12.2 Applications of Dynamic Programming (HW)

Both problems on Homework #2 were concerned with applications of dynamic programming to bioinformatics tasks. This section briefly outlines solutions to these problems:

Problem 1. Give a dynamic programming algorithm for computing the longest leaf-to-leaf path in a tree. (Here we define the length of a path to be the number of edges in the path.) This is also called the "topological diameter" of the tree. Analyze the running time.

Problem 2. Give a dynamic programming algorithm for the longest common subsequence between two strings. This is the same problem as finding the minimum number of deletions from the two strings so that the result is two identical strings. Thus, the longest common subsequence of AAAAAAAAAA and CATTAGAA is AAAA. Analyze the running time.

12.2.1 Computing the topological diameter of a tree

To find the topological diameter of a tree, we first define $\mathcal{H}(n)$, the *height* of a node $n \in \mathcal{N}$ to represent the maximum number of edges from a leaf node it is:

$$\mathcal{H}(n) = \begin{cases} 0 & \text{if } n \in \mathcal{L}, \\ 1 + \max_{c \in \mathcal{C}(n)} \mathcal{H}(c) & \text{otherwise} \end{cases}$$

where $\mathcal{L}$ denotes the set of leaf nodes, $\mathcal{N}$ the set of all nodes in the tree, and $\mathcal{C}(n)$ the set of node n's child nodes; Algorithm 1 shows pseudocode for calculating $\mathcal{H}(n)$.

Algorithm 1: $\mathcal{H}$, Tree height of a node

Input: node $n \in \mathcal{N}$
Output: associative array $\mathcal{H} : n \in \mathcal{N} \mapsto \mathbb{R}$

```
   /* leaf nodes have height 0 */
 1 if C(n) empty then
 2     H(n) ⟵ 0
 3 else
       /* internal nodes have height 1 greater than maximum child
          node height  */
 4     max ⟵ 0
 5     foreach c ∈ C(n) do
 6         if H(c) > max then
 7             max ⟵ H(c)
 8         end
 9     end
10     H(n) ⟵ 1 + max
11 end
```

We now define $\mathcal{D}(n)$ the *topological diameter* of the subtree rooted at node $n \in \mathcal{N}$ as follows:

$$\mathcal{D}(n) = \begin{cases} 0 & \text{if } n \in \mathcal{L}, \\ max \begin{cases} \max_{c \in \mathcal{C}(n)} \mathcal{D}(c), \\ \max_{c,c' \in \mathcal{C}(n)} \mathcal{H}(c) + \mathcal{H}(c') + 2. \end{cases} & \text{otherwise.} \end{cases}$$

Algorithm 2 shows pseudocode for calculating $\mathcal{D}(n)$.

The running time to calculate the height of all nodes in a tree is $O(b|\mathcal{N}|)$, where $b = \max_{n \in \mathcal{N}} |\mathcal{C}(n)|$ is the tree's branching factor; calculating the topological diameter of all nodes given all nodes' heights requires $O(b^2|\mathcal{N}|)$ time. The algorithm for calculating the topological diameter of a tree is therefore linear in the number of nodes in the tree.

12.2.2 Computing the longest common subsequence of a pair of strings [1]

Given strings s_1 and s_2, we define $\mathcal{LCS}(i, j)$ for $i, j \in \mathbb{R}$ to be the length of a longest common subsequence of the two strings defined by the first i letters of

Algorithm 2: $\mathcal{D}$, Topological diameter of the subtree rooted at a node

Input: node $n \in \mathcal{N}$
Output: associative array $\mathcal{D} : n \in \mathcal{N} \mapsto \mathbb{R}$

```
   /* subtree rooted at leaf node has topological diameter 0 */
 1 if C(n) empty then
 2     D(n) ⟵ 0
 3 else
 4     max ⟵ 0
       /* search diameters restricted to subtrees rooted at child
          nodes */
 5     foreach c ∈ C(n) do
 6         if D(c) > max then
 7             max ⟵ D(c)
 8         end
 9     end
       /* search diameters spanning subtrees rooted at all pairs
          of child nodes */
10     foreach c, c' ∈ C(n) do
11         if H(c) + H(c') + 2 > max then
12             max ⟵ H(c) + H(c') + 2
13         end
14     end
15     D(n) ⟵ max
16 end
```

s_1 and the first j letters of s_2, respectively. A mathematical formulation of the longest common subsequence problem is thus:

$$\mathcal{LCS}(i,j) = \begin{cases} 0 & \text{if } i = 0, \\ 0 & \text{if } j = 0, \\ max \begin{cases} 1 + \mathcal{LCS}(i-1, j-1) & \text{if } s_1[i] = s_2[j], \\ \mathcal{LCS}(i-1, j), & \text{otherwise.} \\ \mathcal{LCS}(i, j-1). \end{cases} \end{cases}$$

where $s_1[i]$ and $s_2[j]$ denote the i^{th} letter of s_1 and the j^{th} letter of s_2, respectively. Algorithm 3 shows pseudocode for calculating a matrix M representing $\mathcal{LCS}(i,j)$ for all pairs (i,j) of characters from s_1 and s_2 and a path matrix P indicating the paths through the matrix yielding each of the entries in M; the answer to the longest common subsequence problem is given by the entry in $M[n_1, n_2]$. The running time of the algorithm is $O(n_1 n_2)$.

Figure 12.1 gives a pictorial representation of the matrices M and P following execution of the $\mathcal{LCS}$ algorithm on input strings $s_1 = ABCA$ and $s_2 = BAD$, where the entries in the table represent the entries of M and the arrows represent the entries of P.

	–	B	A	D
–	0	0	0	0
A	0	0	1	1
B	0	1	1	1
C	0	1	1	1
A	0	1	2	2

Figure 12.1: Output of $\mathcal{LCS}$ algorithm on example input strings $s_1 = ABCA$ and $s_2 = BAD$

12.3 Pairwise Sequence Alignment

Recall the formulation for the pairwise sequence alignment problem presented in a previous lecture. Given strings s_1 and s_2 of length n_1 and n_2, respectively,

Algorithm 3: $\mathcal{LCS}$, Longest common subsequence of a pair of strings

Input: strings $s_1 = s_1[1 \ldots n_1], s_2 = s_2[1 \ldots n_2]$
Output: $(n_1 + 1) \times (n_2 + 1)$ matrices M and P

```
   /* initialize entries in first column */
 1 foreach i = 0...n1 do
 2     M[i,0] ⟵ 0
 3     P[i,0] ⟵ {}
 4 end
   /* initialize entries in first row */
 5 foreach j = 0...n2 do
 6     M[0,j] ⟵ 0
 7     P[0,j] ⟵ {}
 8 end
   /* calculate remaining matrix entries */
 9 foreach i = 1...n1 do
10     foreach j = 1...n2 do
           /* if letters match */
11         if s1[i] = s2[j] then
12             M[i,j] ⟵ M[i-1,j-1] + 1
13             P[i,j] ⟵ {(i-1,j-1)}
14         else
15             if M[i,j-1] > M[i-1,j] then
16                 M[i,j] ⟵ M[i,j-1]
17                 P[i,j] ⟵ {(i,j-1)}
18             else if M[i,j-1] < M[i-1,j] then
19                 M[i,j] ⟵ M[i-1,j]
20                 P[i,j] ⟵ {(i-1,j)}
               /* if there's a tie */
21             else
22                 M[i,j] ⟵ M[i,j-1]
23                 P[i,j] ⟵ {(i,j-1),(i-1,j)}
24             end
25         end
26     end
27 end
```

indel cost 1, and substitution cost c, we define $M[i,j]$ to be the cost of the best (i.e. minimal-cost) transformation of the first i letters of s_1 to the first j letters of s_2:

$$M[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ min \begin{cases} M[i-1,j-1] + c\mathsf{H}(s_1[i], s_2[j]), \\ M[i-1,j] + 1, \\ M[i,j-1] + 1. \end{cases} & \text{otherwise} \end{cases}$$

where $\mathsf{H}(s_1[i], s_2[j])$ denotes the Hamming distance from the i^{th} letter of s_1 to the j^{th} letter of s_2. It is straightforward to formulate an $O(n_1 n_2)$ dynamic programming algorithm for calculating $M[n_1, n_2]$ by analogy from Algorithm 3.

We can consider reformulations of the pairwise alignment problem in which the substitution cost is 1 and the cost of a gap of length l is $\mathsf{cost}_g(l) = c_0 + c_1 l$ or, more generally, simply any function $\mathsf{cost}_g(l)$. Note that in these formulations, the cost of a pairwise alignment is not simply the sum of the cost of the columns of letters. Figure 12.2 shows an example pair of pairwise alignments which would have equal costs under the earlier problem formulation, but unequal costs given even general affine gap costs. It has been shown [2] that the pairwise alignment problem with affine gap costs can be solved in $O(n_1 n_2)$ time.

A	C	T	A	G	A		
A	$-$	$-$	$-$	$-$	A		
A	C	A	T	A	A	G	A
A	$-$	A	$-$	A	$-$	$-$	A

Figure 12.2: A pair of pairwise alignments with equal costs under traditional problem formulation and unequal costs given gap penalties

12.4 Multiple Sequence Alignment

Multiple sequence alignment (MSA) is the extension of the pairwise alignment problem from Section 12.3 to 3 or more sequences. This section details two optimization approaches to the MSA problem.

12.4.1 Computing MSA by sum-of-pairs optimization

Given a set of sequences $\mathcal{S} = \{s_1 \dots s_n\}$ and an edit cost function for a pairwise alignment on sequences s and s' $\mathsf{cost} : \mathcal{A}(s, s') \mapsto \mathbb{R}$, our goal is to compute a

multiple sequence alignment $\mathcal{A}^*_{\mathcal{S}}$ on set $\mathcal{S}$ of minimum cost

$$\mathcal{A}^*_{\mathcal{S}} = \arg\min_{\mathcal{A}_{\mathcal{S}}} \sum_{s_i, s_j \in \mathcal{S}} \mathsf{cost}(\mathcal{A}_{\mathcal{S}}(s_i, s_j)),$$

where $\mathcal{A}_{\mathcal{S}}$ is a multiple sequence alignment on set $\mathcal{S}$ and $\mathcal{A}_{\mathcal{S}}(s_i, s_j)$ is the pairwise alignment of sequences s_i and s_j induced by the multiple sequence alignment $\mathcal{A}_{\mathcal{S}}$.

This formulation of the MSA problem is known as the "sum-of-pairs" optimization problem and is known to be NP-hard [3], with running time $O(n^2k)$ to calculate the cost of a given multiple alignment, where k is the length of the longest sequence in $\mathcal{S}$. Figure 12.3 shows an example MSA on 3 sequences and Figure 12.4 shows the corresponding induced pairwise alignments.

$$\begin{array}{ccccc} - & A & C & A & - \\ C & - & - & A & T \\ T & A & - & - & T \end{array}$$

Figure 12.3: A multiple sequence alignment of 3 sequences

$$\begin{array}{ccccc} - & A & C & A & - \\ C & - & - & A & T \end{array}$$

$$\begin{array}{ccccc} - & A & C & A & - \\ T & A & - & - & T \end{array}$$

$$\begin{array}{cccc} C & - & A & T \\ T & A & - & T \end{array}$$

Figure 12.4: Pairwise alignments induced by the MSA in Figure 12.3

12.4.2 Computing MSA by generalized tree alignment optimization

Another approach to computing an optimal MSA is referred to as the "generalized tree alignment" optimization problem; the inputs are the same as those to the sum-of-pairs optimization problem outlined in Section 12.4.1. The output is a tree $\mathcal{T}$ with the sequences in $\mathcal{S}$ at the leaves and a set $\mathcal{I}$ of new sequences at the tree's internal nodes. Our goal is to compute a tree $\mathcal{T}^*$ with leaf nodes $\mathcal{T}^*_{\mathcal{L}} = \mathcal{S}$ and internal nodes $\mathcal{T}^*_{\mathcal{I}}$ of minimum cost

$$\mathcal{T}^* = \arg\min_{\mathcal{T}} \sum_{(n_1,n_2)\in\mathcal{T}_\mathcal{E}} \mathsf{cost}(\mathcal{A}(s[n_1], s[n_2]))$$

where $(n_1, n_2) \in \mathcal{T}_\mathcal{E}$ denotes the edge delimited by nodes n_1 and n_2 from tree $\mathcal{T}$'s edge set $\mathcal{T}_\mathcal{E}$ and $s[n]$ gives the sequence associated with node n. Figure 12.5 shows an example tree on a set of 7 sequences $S_1 \ldots S_7$ with five internal sequence nodes circled and lettered $a \ldots e$. The problem of finding the minimum cost tree is known to be NP-hard [3].

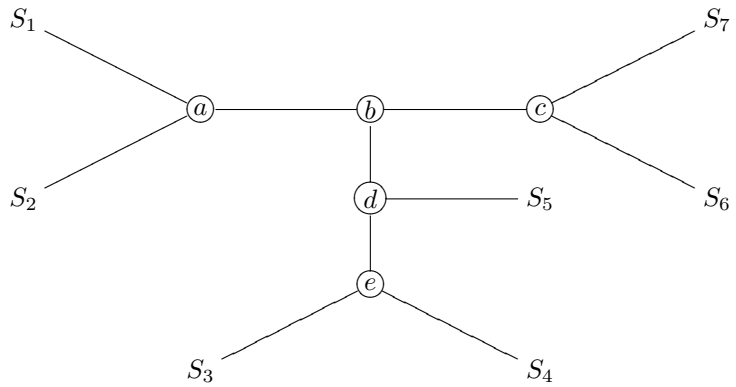

Figure 12.5: A tree with sequences at the leaves and internal sequence nodes

We can consider a relaxation of the generalized tree alignment problem in which the tree $\mathcal{T}$ is given and we need only assign sequences to the internal nodes to minimize the overall cost. This simplified problem is known as the "tree alignment" optimization problem and is *still* NP-hard [3].

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition,* 2001.

[2] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.,* 162:705–708, 1982.

[3] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *J. Comput. Biol.,* 1:337-348, 1994.