Lecture 12

Maximum Subsequence Problem

February 15, 2000
Notes: Mathieu Blanchette

12.1. Scoring Regions of Sequences

We have studied a variety of methods to score a DNA sequence so that regions of interest obtain a high score. For example, Section 11.4.2 suggested codon bias as a means for finding coding regions. If C is a codon, the score associated with C is $\log_2 \frac{C_R}{C_B}$, where C_R is the frequency of C in known coding regions (in the correct reading frame), and C_B is the frequency of C in noncoding regions (usually taken as the background distribution).

Notice that our goal is to identify *new* coding regions, but the method requires that we already know some coding regions in order to estimate C_R. There are a few easy ways one can identify a subset of likely coding regions. First, one could look for long *open reading frames* (*ORFs*), that is, long contiguous reading frames without STOP codons. Since 3 of the 64 codons are STOP codons (see Table 1.1), in random sequences one would expect a STOP codons every 64/3 triplets, i.e., every 64 bases, if codons are distributed uniformly. Since most genes are at least hundreds of bases long, very long ORFs are likely to be coding regions. This method will work well if we assume that the new genome contains no introns (or at least many very long exons), and if the codon distribution in long genes is similar to that in all genes, so that we can use it to estimate C_R. Another easy way to find a training set of coding sequences is by sequence similarity: compare the sequence of interest with a genome in which many genes are known, and extract the regions with high sequence similarity to known genes.

Then, if we assume that different triplets in the sequence are independent, we would like to find contiguous stretches of triplets with high total score (and thus with high log likelihood ratio). These regions would be good candidates for coding regions, to be subjected to further testing.

Another relevant question is in which reading frame to look for codons. There are 6 possible reading frames: 3 on each of the 2 strands of DNA. When looking for coding region, one would search for high scoring regions in each of these 6 reading frames.

12.2. Maximum Subsequence Problem

We can distill the following general computational problem from the preceding discussion. We are given a sequence $X_1, X_2, \ldots, X_n$ of real numbers, where X_i corresponds to the score of the ith element of the sequence. The problem is to find a contiguous subsequence $X_i, X_{i+1}, \ldots, X_j$ that maximizes $X_i + X_{i+1} + \cdots + X_j$. We will call this a *maximum subsequence*. Note that, if all X_i's are nonnegative, the problem is

not interesting, since the maximum subsequence will always be $X_1, X_2, \ldots, X_n$, so the interesting case is when some of the scores are negative.

The following algorithm for finding a maximum subsequence was given by Bates and Constable [1] and Bentley [2, Column 7].

Suppose we already knew that the maximum subsequence B of $X_1, X_2, \ldots, X_k$ has score b. How can we find the maximum subsequence of $X_1, X_2, \ldots, X_k, X_{k+1}$? If X_k is included in B, then it is easy: if $X_{k+1} > 0$, we will add X_k to B, and if not, we will leave B unchanged. But what if X_k is not included in B? In that case, in addition to B we will have to keep track of the score of the *maximum suffix* F of $X_1, X_2, \ldots, X_k$: F is the suffix $X_s, X_{s+1}, \ldots, X_k$ that maximizes $f = X_s + X_{s+1} + \cdots + X_k$. Let us assume that F is also known for $X_1, X_2, \ldots, X_k$. We are now given X_{k+1}, and we want to update B and F accordingly:

if $f + X_{k+1} > b$
  then add X_{k+1} to F and replace B by F
  else if $f + X_{k+1} > 0$
    then add X_{k+1} to F
    else reset F to be empty.

The complexity of the algorithm is $O(n)$, since a constant amount of work is done for every new element X_{k+1}, and there are n such elements.

12.3. Finding *All* High Scoring Subsequences

The algorithm described in Section 12.2 works very well if we are interested in finding *one* maximum subsequence. However, we are generally looking for *all* high scoring regions, for instance, all good candidates for coding regions. We could repeatedly use the previous algorithm to find them all: find the maximum subsequence, remove it, and repeat on the two remaining parts of the sequence. We will call the problem of finding exactly these disjoint maximum subsequences the *all maximum subsequences* problem. (In practice, one would only want to retain those reported maximum subsequences with scores sufficiently high to be interesting.)

The problem with repeatedly running the previous algorithm is that it will take $O(n)$ operations per subsequence reported, and thus possibly $O(n^2)$ operations to identify all high scoring regions. Intuitively, one might hope to do better, since much of the work done to find the first maximum subsequence could be reused to find the second one, and so on. We now present an algorithm that solves the all maximum subsequences problem in time $O(n)$, the same as the time to find just one maximum subsequence. This algorithm is due to Ruzzo and Tompa [3]. We first describe the algorithm, and then discuss its performance.

Algorithm. The algorithm reads the scores from left to right, and maintains the cumulative total of the scores read so far. Additionally, it maintains a certain ordered list $I_1, I_2, \ldots, I_{k-1}$ of disjoint subsequences. For each such subsequence I_j, it records the cumulative total L_j of all scores up to but not including the leftmost score of I_j, and the total R_j up to and including the rightmost score of I_j.

The list is initially empty. Input scores are processed as follows. A nonpositive score requires no special processing when read. A positive score is incorporated into a new subsequence I_k of length one[1] that is then

[1] In practice, one could optimize this slightly by processing a consecutive series of positive scores as I_k.

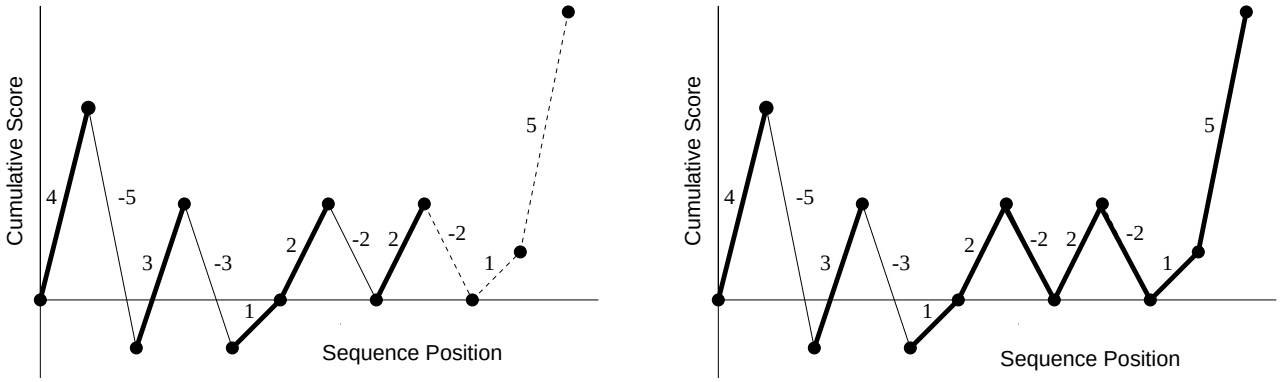

Figure 12.1: An example of the algorithm. Bold segments indicate score sequences currently in the algorithm's list. The left figure shows the state prior to adding the last three scores, and the right figure shows the state after.

integrated into the list by the following process.

1. The list is searched from right to left for the maximum value of j satisfying $L_j < L_k$.
2. If there is no such j, then add I_k to the end of the list.
3. If there is such a j, and $R_j \geq R_k$, then add I_k to the end of the list.
4. Otherwise (i.e., there is such a j, but $R_j < R_k$), extend the subsequence I_k to the left to encompass everything up to and including the leftmost score in I_j. Delete subsequences $I_j, I_{j+1}, \ldots, I_{k-1}$ from the list (none of them is maximum) and reconsider the newly extended subsequence I_k (now renumbered I_j) as in step 1.

After the end of the input is reached, all subsequences remaining on the list are maximum; output them.

As an example of the execution of the algorithm, consider the input sequence $(4, -5, 3, -3, 1, 2, -2, 2, -2, 1, 5)$. After reading the scores $(4, -5, 3, -3, 1, 2, -2, 2)$, suppose the list of disjoint subsequences is $I_1 = (4), I_2 = (3), I_3 = (1, 2), I_4 = (2)$, with $(L_1, R_1) = (0, 4)$, $(L_2, R_2) = (-1, 2)$, $(L_3, R_3) = (-1, 2)$, and $(L_4, R_4) = (0, 2)$. (See Figure 12.1.) At this point, the cumulative score is 2. If the ninth input is -2, the list of subsequences is unchanged, but the cumulative score becomes 0. If the tenth input is 1, Step 1 produces $j = 3$, because I_3 is the rightmost subsequence with $L_3 < 0$. Now Step 3 applies, since $R_3 \geq 1$. Thus $I_5 = (1)$ is added to the list with $(L_5, R_5) = (0, 1)$, and the cumulative score becomes 1. If the eleventh input is 5, Step 1 produces $j = 5$, and Step 4 applies, replacing I_5 by $(1, 5)$ with $(L_5, R_5) = (0, 6)$. The algorithm returns to Step 1 without reading further input, this time producing $j = 3$. Step 4 again applies, this time merging I_3, I_4, and I_5 into a new $I_3 = (1, 2, -2, 2, -2, 1, 5)$ with $(L_3, R_3) = (-1, 6)$. The algorithm again returns to Step 1, but this time Step 2 applies. If there are no further input scores, the complete list of maximum subsequences is then $I_1 = (4), I_2 = (3), I_3 = (1, 2, -2, 2, -2, 1, 5)$.

The fact that this algorithm correctly finds all maximum subsequences is not obvious; see Ruzzo and Tompa [3] for the details.

Analysis. There is an important optimization that may be made to the algorithm. In the case that Step 2 applies, $I_1, \ldots, I_{k-1}$ are maximum subsequences, and so may be output before reading any more of the

input. Thus, Step 2 of the algorithm may be replaced by the following, which substantially reduces the memory requirements of the algorithm.

2.′ If there is no such j, all subsequences $I_1, I_2, \ldots, I_{k-1}$ are maximum. Output them, delete them from the list, and reinitialize the list to contain only I_k (now renumbered I_1).

The algorithm as given does not run in linear time, because several successive executions of Step 1 might re-examine a number of list items. This problem is avoided by storing with each subsequence I_k added during Step 3 a pointer to the subsequence I_j that was discovered in Step 1. The resulting linked list of subsequences will have monotonically decreasing L_j values, and can be searched in Step 1 in lieu of searching the full list. Once a list element has been bypassed by this chain, it will be examined again only if it is being deleted from the list, either in Step 2′ or Step 4. The work done in the "reconsider" loop of Step 4 can be amortized over the list item(s) being deleted. Hence, in effect, each list item is examined a bounded number of times, and the total running time is linear.

The worst case memory complexity is also linear, although one would expect on average that the subsequence list would remain fairly short in the optimized version incorporating Step 2′. Empirically, a few hundred stack entries suffice for processing sequences of a few million residues, for either synthetic or real genomic data.

References

[1] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, Jan. 1985.

[2] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[3] W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241, Heidelberg, Germany, Aug. 1999. AAAI Press.