Applied Algorithms

Finding All Maximal Scoring Subsequences

3/12/2007

Mary Ann Joy

Introduction: In molecular biology while analyzing large sequences of DNA or protein it is important to identify unusual subsequences. An often used technique is to assign a score to each nucleotide or amino acid and then search for contiguous sequences with high scores.

Problem Definition: Given an input sequence ($x_1, x_2, \ldots\ldots\ldots\ldots x_n$) of positive and negative real numbers. We define the score of a consecutive subsequence ($x_i, x_{i+1}, \ldots\ldots\ldots\ldots x_j$) as $S_{ij} = \sum_{i<=k<=j} x_k$. The goal is to identify all disjoint, contiguous subsequences having positive scores. We refer to these sequences as maximal scoring subsequences. In order to avoid overlapping sequences with tied scores we do not allow sequences with non empty zero prefixes or suffixes.

Quadratic algorithm:

We first define an algorithm that finds a subsequence with the maximum score. This algorithm was given by Bates and Constable [1]. Let $\mathbf{M_n}$ be the maximum subsequence with score **m** for a subsequence of **($\mathbf{x_1, x_2, \ldots\ldots\ldots\ldots x_n}$).**

Let us consider the scores of all subsequences ending in $\mathbf{x_n}$. Let $\mathbf{L_n}$ be the subsequence with maximum score **l** among all such subsequences.

By induction when $\mathbf{n = 1, L_1 = x_1}$; Assume for a sequence of length **n**, $\mathbf{L_n}$ is the maximum value subsequence ending in $\mathbf{x_n}$. When we add $\mathbf{x_{n+1}}$; $\mathbf{L_{n+1}}$ is either $\mathbf{x_{n+1}}$ or $\mathbf{L_n + x_{n+1}}$ whichever is larger.

$\mathbf{L_{n+1} = Max\ (x_{n+1}, L_n + x_{n+1})}$ **------------------------1**

Let us calculate $\mathbf{M_{n+1}}$ when we add $\mathbf{x_{n+1}}$ to our input sequence:

Case 1: $\mathbf{x_{n+1}}$ is not part of the maximum scoring subsequence; i.e $\mathbf{M_{n+1} = M_n}$.

Case2: The new maximum subsequence does include $\mathbf{x_{n+1}}$.

Combining the two cases we get:

$\mathbf{M_{n+1}}$ = Max ($\mathbf{M_n}$, $\mathbf{L_{n+1}}$)

$\mathbf{M_{n+1}}$ = Max ($\mathbf{M_n}$, **Max** ($\mathbf{x_{n+1}}$, $\mathbf{L_n + x_{n+1}}$)) (Substituting for $\mathbf{L_{n+1}}$ from 1)

The following algorithm can be used to solve this problem:

```
If $x_{n+1} > 0$ then
    If $x_n \varepsilon M$ then
        Add $x_{n+1}$ to M
    Else
        Add $x_{n+1}$ to L
        If l > m then
            Replace M by L
        Endif
    Endif
Else
    Reset L.
```

Example: If the input sequence (3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5), then **M** = (5, 7, 2, -3, 10) and **m** = 21. **L** = (5, 7, 2, -3, 10, -7, 5) with **l** = 19. Now if the next element we want to add is 40. Then **M** = **S** = (5, 7, 2, -3, 10, -7, 5) with the new maximum score = 59.

Analysis: This is a linear time algorithm. Given this algorithm we can use a divide and conquer approach to calculate all maximal subsequences. We first find the highest scoring subsequence, remove it and then recursively apply the algorithm to the remaining sequence on the left and then on the right. If the input sequence is random the running time of this algorithm could be **O(nlogn)**. However if the maximal subsequence always falls at one end of the sequence then the worst case complexity could be $\mathbf{O(n^2)}$.

Linear time Algorithm

We now look at an algorithm that finds all maximum subsequences in linear time. This algorithm was introduced in a paper by Ruzzo and Tompa [2]. Given a sequence: ($\mathbf{x_1}$, $\mathbf{x_2}$,............ $\mathbf{x_n}$) of length **n**; let $\mathbf{I_1}$, $\mathbf{I_2}$,............ $\mathbf{I_{k-1}}$ be an ordered list of disjoint maximal score subsequences. For n = 0 the list is empty. For each $\mathbf{I_j}$ in our list we keep track of two scores, $\mathbf{L_j}$: is the total cumulative score of all scores up to but not including the left most score in $\mathbf{I_j}$. And $\mathbf{R_j}$ is the cumulative total of all the scores in the sequence up to and including the rightmost score in $\mathbf{I_j}$**.** The score of the subsequence represented by the list $\mathbf{I_j}$ is $\mathbf{R_j}$ - $\mathbf{L_j}$.

Initially our list is empty.

Let us consider the point when we add the $\mathbf{x_{n+1}}^{\mathbf{th}}$ element in the sequence:

Case 1:

$\mathbf{x_{n+1}}$ is negative; then we do nothing.

Case 2:

$\mathbf{x_{n+1}}$ is positive. Then our new ordered list will be $\mathbf{I_1, I_2, \ldots\ldots\ldots\ldots I_{k-1}, I_k}$. Where = $\mathbf{I_k = \{x_{n+1}\}}$**.**

Case 2.a: There exists a list $\mathbf{I_j}$ (1<=j<=k-1) such that by adding $\mathbf{I_k}$ as a suffix to $\mathbf{I_j}$ we create a new list $\mathbf{I'_j}$ with score greater than that of both $\mathbf{I_k}$ and $\mathbf{I_j}$. In this case we will grow $\mathbf{I_j}$ to include all the scores in the sequence up to and including $\mathbf{x_{n+1}}$**.** And we remove all the lists from $\mathbf{I_{j+1}}$ to $\mathbf{I_k}$. The total score for $\mathbf{I'_j = R_k - L_j}$. Our condition for merging the two lists was:

$\mathbf{R_k - L_j > R_k - L_k}$ and

$\mathbf{R_k - L_j > R_j - L_j}$

$\Rightarrow$ $\mathbf{L_j < L_k}$ and $\mathbf{R_j < R_k}$.

Formally the algorithm as given by [2] is:

Initially the list is empty. Input scores are processed as follows. If the score is negative do nothing. If it is positive then create a new subsequence $\mathbf{I_k}$ of length 1. This list is added to the existing list of subsequences by the following steps.

1. The list is searched from right to left for the maximum value of **j** satisfying $\mathbf{L_j < L_k}$.
2. If there is no such **j**, then add $\mathbf{I_k}$ to the end of the list.
3. If there is such a **j** and $\mathbf{R_j >= R_k}$, then add $\mathbf{I_k}$ to the end of the list.
4. Otherwise, extend the subsequence $\mathbf{I_k}$ to the left to encompass everything up to and including the leftmost score in $\mathbf{I_j}$. Delete subsequences $\mathbf{I_j}$, $\mathbf{I_{j+1}} \ldots \mathbf{I_{k-1}}$. Now reconsider the newly extended subsequence $\mathbf{I_k}$ (now renamed) as $\mathbf{I_j}$ in step 1.

When we get to the end of our input all subsequences remaining in our list are maximal.

Example: Consider the input sequence (3, -5, 2, 3, -1, 1, -1, 6). After reading the first 7 scores the list of disjoint subsequences is:

I_1= (3), I_2= (2, 3), I_3= (1) with (L_1, R_1) = (0,3), (L_2, R_2) = (-2, 3) and (L_3, R_3) = (2, 3). The 9th input is 6 which is added as I_4= (6) and (L_4, R_4) = (2 ,8). When we scan through the list from right to left we find I_2 to be the list with $L_2 < L_4$ and $R_2 < R_4$. So we expand L_2 to include (2, 3, -1, 1, -1, 6) and delete I_3 and I_4. The maximal subsequences at the end of the algorithm are: (3) and (2, 3, -1, 1, -1, 6).

Analysis: We have to apply some optimizations to the above algorithm to make it a linear time algorithm. In step 2 if we cannot find any **j** then all the lists up until that point are maximal and we can output those subsequences and reset our list with $\mathbf{I_k} = \mathbf{I_1}$. Similarly in step 3 when we add $\mathbf{I_k}$ we keep a pointer to the subsequence $\mathbf{I_j}$. Now in step 1 instead of scanning through the entire list we just search this linked list of subsequences. The resulting optimized algorithm has **O(n)** complexity.

Further Areas of Research: Alves, Caceres and Song [3] have presented a parallel algorithm to compute the basic maximum subsequence problem. Given **p** processors they divide the input into **p** sequences each of size **n/p**. On each of the processors the subsequence is partitioned into 5 subsequences. These 5 subsequences are then reduced to 5 numbers. Processor 1 receives these 5 numbers from each of the p processors and then combines them in linear time to find the maximum subsequence in O(n/p) time. It would be interesting to research extending this algorithm to solve the problem of finding all maximal subsequences.

References:

1. Bates, J. L., and Constable, R. L. 1985. Proofs as programs.
2. Ruzzo W. L. and Tompa M. 1999. A Linear Time Algorithm for Finding All Maximal Scoring Subsequences.
3. Alves C. E. R., Caceres E. N. and Song S. W. 2003. Computing Maximum Subsequence in Parallel.