Introduction to Bioinformatics Algorithms
Homework 4 Solution

Saad Mneimneh, Computer Science, Hunter College of CUNY

Problem 1: Spliced alignments

(a) Consider the Exon Chaining problem in the case where all intervals have the same weight. For this setting, the best chain is obviously the one that has the maximum number of non-overlapping intervals. Describe a **greedy** algorithm that finds the optimal solution.

Solution: Each exon has a start and an end, and can be represented as an interval $[s_i, e_i]$. We make the following observation. The exon j with the smallest e_j can be part of the optimal solution. We can show this by contradiction: Assume we are given an optimal solution that does not contain $[s_j, e_j]$. Every exon in the solution must have $e_i \geq e_j$. This means that for this exon i to overlap with exon j, $e_j \in [s_i, e_i]$. Since the optimal solution cannot include overlaps, there can be at most one such exon i. We can then replace i with j. Therefore, we can choose the exon with the smallest e_i, eliminate all exons that overlap with it, and do this repeatedly. That's the greedy algorithm.

(b) For the Spliced Alignment problem presented in the book, if m is the length of the target t, n is the length of the genome g, and l is the sum of all block lengths, then the algorithm runs in $O(ml|V| + |V|^2)$, where $O(|V|^2)$ is needed to determine the order of blocks using topological sort ($|V|$ is the number of blocks). If $l = O(n|V|)$, this algorithm runs in $O(mn|V|^2)$ time.

Suggest a way to improve this running time by scanning all blocks from left to right. In particular, if $S(i, B, j)$ represents the score of the optimal spliced alignment that ends in g_i and t_j where $i \in B$, compute S like this:

for each (i, B) in lexicographic order
  for each $j \leftarrow 0$ to m
    do ...

When $(i, B, j) = (right(B), B, j)$, we are "ending" block B. So after $S(right(B), B, j)$ is computed, update a quantity $W(j)$ if that's the highest seen for j among all block endings so far. This quantity will be used to avoid going through the list of all blocks that precede a given one (see recurrences in the book).

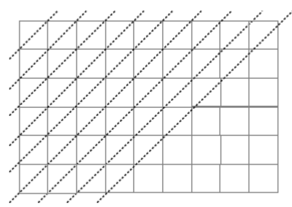
What is the running time of your algorithm?

Solution

for $j \leftarrow 0$ to m
 do $W(j) \leftarrow 0$
for each (i, B) in lexicographic order
 for $j \leftarrow 0$ to m
  do if $i \neq left(B)$
   then $S(i, B, j) \leftarrow \max[S(i-1, B, j) - \gamma, S(i, B, j-1) - \gamma, S(i-1, B, j-1) + s(g_i, t_j)]$
   else $S(i, B, j) \leftarrow \max[S(i, B, j-1) - \gamma, W(j) - \gamma, W(j-1) + s(g_i, t_j)]$
  if $i = right(B)$ and $S(i, B, j) > W(j)$
   then $W(j) \leftarrow S(i, B, j)$

To sort the (i, B), we need $O(l \log l)$ time. Then we spend $O(ml)$ time in the main loop. Alternatively, if $l = O(n|V|)$, then we can go over every $i = 1 \dots n$ and every block B and check if $i \in B$. Then we have $O(n|V| + mn|V|) = O(mn|V|)$ time.

Problem 2: Space efficient balanced alignment
In parallel implementations of alignment, most algorithms fill the table diagonal-wise because the entries required for filling the diagonal are all contained in the previous two diagonals, and there are no intra-dependencies within a diagonal, so each diagonal can be computed in parallel.



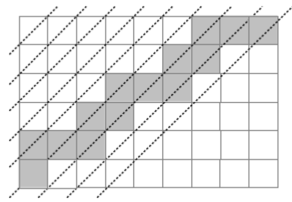
(a) Implement a linear-space (no backtracking) global alignment algorithm that computes the optimal score and works diagonally.

Solution: Assume $n < m$, we need three vectors of size at most $n+1$ (three diagonals), both index from 0 to n. Assume the vertical direction has length n. Then the j^{th} element of the i^{th} diagonal represents entry $(i-j, j)$ when $i \leq n$, and entry $(n-j, i+j-n)$ when $i > n$.

$diag_0[0] \leftarrow 0$
for $i \leftarrow 1$ to n
 for $j \leftarrow 0$ to i
  $diag_i[j] = \max\{diag_{i-1}[j-1] - \gamma, diag_{i-1}[j] - \gamma, diag_{i-2}[j-1] + s(x_{i-j}, y_j)\}$
for $i \leftarrow n+1$ to m
 for $j \leftarrow 0$ to n
  $diag_i[j] = \max\{diag_{i-1}[j] - \gamma, diag_{i-1}[j+1] - \gamma, diag_{i-2}[j+1] + s(x_{n-j}, y_{i+j-n})\}$
for $i \leftarrow m+1$ to $m+n$
 for $j \leftarrow 0$ to $m+n-i$
  $diag_i[j] = \max\{diag_{i-1}[j] - \gamma, diag_{i-1}[j+1] - \gamma, diag_{i-2}[j+1] + s(x_{n-j}, y_{i+j-n})\}$

We assume $diag_i[j] = -\infty$ if $j > i$ and $diag_i[-1] = diag_i[n+1] = -\infty$. At the end, we compute $diag_{m+n}[0]$. Since we need at most 3 diagonals at any point in time, we can replace $diag_i$ by $diag_{i \bmod 3}$ (conditions above without mod).

(b) Modify your algorithm to save a set of entries (i, j) such that $ij = (m - i)(n - j)$, roughly, so that every global alignment must cross this set.



(c) In the linear-space approach to sequence alignment, the original problem of size $m \times n$ is reduced to two subproblems of sizes $\frac{m}{2}j$ and $\frac{m}{2}(n - j)$. In a fast parallel implementation of sequence alignment, it is desirable to have a *balanced partition* that breaks the original problem into subproblems of equal sizes. Design a space-efficient version with balanced partitioning (but you don't have to implement it).

Hint: The optimal alignment must pass through one of the entries you saved.

Problem 3: Partial digest

Rewrite the pseudocode for the "practical" partial digest algorithm using fewer lines, by moving any change to X and L into the recursive call. In addition, make it stop when a solution is found. Implement the algorithm using the vector library in C++.

Solution:

First, here's the modification to simplify the pseudicode:

PartialDigest(L)
  $width \leftarrow \max L$
  $X \leftarrow \{0, width\}$
  $L \leftarrow L - \{width\}$
  Place($L, X, width$)

Place($L, X, width$)
  If $L = \emptyset$
    then output X
      return
  $y \leftarrow \max L$
  if $\Delta(y, X) \subset L$
    Place($L - \Delta(y, X), X \cup \{y\}, width$)
  if $\Delta(width - y, X) \subset L$
    Place($L - \Delta(width - y, X), X \cup \{width - y\}, width$)

Second, we add the feature to stop when a solution is found, this now returns true if a solution is found, and false otherwise:

```
PartialDigest(L)
  width ← max L
  X ← {0, width}
  L ← L − {width}
  Place(L, X, width)

Place(L, X, width)
  If L = ∅
    then output X
         return true
  y ← max L
  if Δ(y, X) ⊂ L
    then if Place(L − Δ(y, X), X ∪ {y}, width)
           then return true
  if Δ(width − y, X) ⊂ L
    then if Place(L − Δ(width − y, X), X ∪ {width − y}, width)
           then return true
  return false
```