ECS 222A: Assignment #3

Due on Tusday, January 20, 2015

Daniel Gusfield TR 4:40pm-6:00pm

Wenhao Wu

Contents

Problem 1 **3**
Problem 1(a) . . . 3
Problem 1(b) . . . 4

Problem 2 **4**
Problem 2(a) . . . 5
Problem 2(b) . . . 5
Problem 2(c) . . . 5

Problem 3 **6**

Problem 4 **8**
Problem 4(a) . . . 8
Problem 4(b) . . . 9
Problem 4(c) . . . 9
Problem 4(d) . . . 9

Problem 1

The secondary structure problem discussed in section 6.5 in the book seeks to find the secondary structure that maximizes the number of base pairs it contains, i.e. the number of pairs that are in the matching (or pairing). Review the definitions of "secondary structure" on page 274 in the book. In class, we used the term "non-crossing pairing" for what the book calls a "secondary structure".

Now suppose that instead of wanting to find the the secondary structure maximizing the number of pairs it contains, we want to *count* the exact *number* of distinct secondary structures possible in a given RNA sequence. Some of these secondary structures will not contain the largest number of pairs possible.

For example, in the RNA molecule ACGGGUGU there are five secondary structures. One contains no pairs (hey, its a legal secondary structure according to the definition); one pairs the A to the farthest U; one that pairs the A to closest U; one pairs the C to the farthest G; and one pairs the A to the farthest U, and the C to the farthest G.

This counting problem can be solved by DP.

Problem 1(a)

Write recurrence relations that give the solution to the counting problem.

Hint: You may be tempted to just do a simple conversion of the recurrences we used in class to find the maximum number of pairs in a non-crossing pairing (secondary structure), but you need to be careful. The reason, is that the cases in the recurrences we used in class were not disjoint. That is, the same secondary structure might arise by more than one case in the recurrences. Since we were taking the Max over all the cases, it did not matter if the same secondary structure arose different ways. But now that we want to count the number of distinct secondary structures, we have to be more careful. Define $N(i, j)$ as the number of secondary structures involving the positions from i to j inclusive. It includes the empty matching as one of the matchings. For technical reasons, you may want to define define $N(j, j) = N$ and $N(j + 1, j) = 1$. The problem asks you to write recurrences for $N(i, j)$.

Be sure to explain why your recurrences give a correct recursive solution for the problem of counting the number of secondary structures.

Answer: Consider all the secondary structures on the inclusive interval (i, j):

- The number of secondary structures in which element j is not paired with any element in $(i, j-1)$ is the same as the number of secondary structures on $(i, j-1)$, i.e.

$$N_1(i, j) = N(i, j-1)$$

- If element j can be paired with some element $k \in (i, j-1)$, indicated by $\beta(k, j) = 1$, then any secondary structure on $(i, k-1)$ and any secondary structure on $(k+1, j-1)$, together with the pair formed by element k and j, forms a distinct secondary structure on (i, j) that is not included in the previous case. Consequently, the number of these secondary structures is

$$N_{2,k}(i, j) = \begin{cases} N(i, k-1)N(k+1, j-1), & \text{if } \beta(k, j) = 1 \\ 0, & \text{else} \end{cases}$$

These 2 cases exhaust all distinct secondary structures, therefore the recurrence of counting is

$$\begin{aligned} N(i,j) &= N_1(i,j) + \sum_{k=i}^{j-1} N_{2,k}(i,j) \\ &= N(i,j-1) + \sum_{\substack{k=i \\ \beta(k,j)=1}}^{j-1} N(i,k-1)N(k+1,j-1) \end{aligned}$$

And the base cases are (according to the definition on the textbook and the example given here, no sharp turns)

$$N(j, j+m) = 1,\ m = -1, 0, 1, 2, 3, 4.$$

In these cases the only secondary structure is no pairing at all.

Problem 1(b)

As before, instead of using the recurrences in a top-down recursive algorithm, we want to use them in a DP solution to the problem.

Write out the pseudo-code for a DP solution to the counting problem, and analyze the worst-case running time of the DP solution.

Answer: The pseudo-code for a DP solution is given in Algorithm 1. There are $O(n^2)$ $N(i,j)$ terms in the

Algorithm 1 Count the number of secondary structures in a RNA string.

1: Initialize $N(j, j+m) = 1$ for $m = -1, 0, 1, 2, 3, 4$, $j = 1, \ldots, n$.
2: **for** $m = 1$ **to** $n-1$ **do**
3: **for** $i = 1$ **to** n **do**
4: Compute and save

$$N(i, i+m) = N(i, i+m-1) + \sum_{\substack{k=i \\ \beta(k,i+m)=1}}^{i+m-1} N(i,k-1)N(k+1,i+m-1)$$

5: **end for**
6: **end for**
7: **return** $N(1,m)$.

DP table to fill in. To fill in each term using the recurrence equation requires $O(n)$ multiplication, addition and table lookup (RAM model). Consequently, the worst-case running time of the DP solution is $O(n^3)$.

Problem 2

Suppose we are given a rooted tree T with n leaves and m non-leaf nodes. Each leaf is colored with one of $k < n$ given colors, so several leaves can have the same color. We need to color each interior node of T with one of the k given colors to *maximize* the number of edges whose (two) endpoints are colored the same color.

We can solve this with a DP algorithm that runs in $O(mk)$ time. Let $V(v,i)$ denote the optimal solution value when the problem is applied to the subtree rooted at node v, and v is required to be given color i. Let $V(v)$ denote the optimal solution value when the problem is applied to the subtree rooted at node v, and there is no restriction on which of the k colors v can be.

Problem 2 continued on next page...

Problem 2(a)

Using that notation, develop recurrences for this problem, and explain the correctness of your recurrences.

Answer: Denote the set of leaf children and non-leaf children of node v as $v.C_l$ and $v.C_{nl}$, respectively. Also denote the set of colors i for which $V(v,i) = V(v)$ as $v.Color_{max}$. Denote indicator function $\delta(x \in X) = 1$ if $x \in X$ and 0 otherwise, and indicator function $\delta(x,y) = 1$ if $x = y$ and 0 otherwise. Denote the color of a leaf node l as $l.color$. Then we have

$$V(v,i) = \sum_{l \in v.C_l} \delta(i, l.color) + \sum_{n \in v.C_{nl}} (V(n) + \delta(i \in n.Color_{max}))$$
$$V(v) = \max_{i=1,\ldots,k} V(v,i)$$

Here is an explanation for this recurrence:

- For the 1st term in the RHS of the first equation, node v forms a same-color edge with any leaf nodes with color i.
- For the 2nd term in the RHS the first equation, given a non-leaf child n of v
  - If n with color i can achieve $V(n)$, then apart from the $V(n)$ same-color edges that the tree from n contains, n and v can also form another same-color edge, therefore the maximum number of same-color edges introduced by n is $V(n) + 1$.
  - If n with color i can not achieve $V(n)$:
    - By setting the color of n to some color $j \in n.Color_{max}$, the maximum number of same-color edges introduced by n is $V(n)$;
    - By setting the color of n to some color $j \notin n.Color_{max}$, the maximum number of same-color edges in the tree from n is at most $V(n) - 1$, and there is at most one additional same-color edge (v, n), consequently the maximum number of same-color edges introduced by n is at most $V(n)$.

  In conclusion, $V(n) + \delta(i \in n.Color_{max})$ is an achievable upper bound for the maximum number of same-color edges introduced by n if $n.parent$ is colored i. Also, the above analysis indicate that, if $n.parent$ is colored i, then the optimal coloring for n is i if $i \in n.Color_{max}$ or any $j \in n.Color_{max}$ if $i \notin n.Color_{max}$.

Note that with this recurrence we don't need to explicitly define a base case. The solution to the original problem, is simply $V(T.root)$. To determine an optimal coloring scheme, each non-leaf, non-root node v maintains the set $v.Color_{max}$ and a traceback from $T.root$ will determine the optimal color for v.

Problem 2(b)

Expalin how the recurrences are evaluated (solved) in an efficient DP way.

Answer: The pseudo-code for a DP solution is given in Algorithm 2.

Problem 2(c)

Show that the time bound for your DP is $O(mk)$.

Answer: The essential computation happens in Step 3 of Algorithm 2. The first δ function can be evaluated in constant times, V(n) can be accessed in constant times and the second δ function can also be evaluated in constant time (e.g. using hash). Therefore, the total amount of time to evaluate a single

Algorithm 2 Count the number of maximum same-color edges.

1: Order the non-lear nodes in tree T into a heap H, i.e. put each node into a queue in the same order as in a BFS but perform no dequeueing. This can be done in $O(m)$.
2: **for** $l = m : l >= 1: l--$ **do**
3: Compute and save

$$V(h[m], i) = \sum_{l \in h[m].C_l} \delta(i, l.color) + \sum_{n \in h[m].C_{nl}} (V(n) + \delta(i \in n.Color_{max})),\ i = 1, \ldots, k$$

and determine $V(h[m]) = \max_{i=1,\ldots,k} V(h[m], i)$ and $h[m].Color_{max} = \{j | V(h[m]) = V(h[m], j)\}$.
4: **end for**
5: To determine an optimal coloring scheme, color $T.root$ with any $j \in T.root.Color_{max}$, then starting from $T.root$, perform a BFS or DFS during which the color of v is set to $v.parent.color$ if it is in $v.Color_{max}$, or any $j \in v.Color_{max}$ if it is not.
6: **return** the maximal number of edges $V(T.root)$.

$V(h[m], i)$ is $O(h[m].degree)$, i.e. linear w.r.t to the number of children that $h[m]$ has. Then the evaluation of all k $V(h[m], i)$, the $V(h[m])$ and $h[m].Color_{max}$ takes $O(k \cdot h[m].degree)$ time. The traceback in Step 5 takes $O(m)$ time. Consequently, the time bound for this DP is

$$t = \sum_{v \in T} O(k \cdot v.degree) + O(m) = O(mk).$$

Problem 3

In the sequence alignment problem, suppose now we want to compute the number of optimal alignments that align character i with character j, or align character i with a space, for each pair (i, j). Show how to compute this via DP in $O(nm)$ time.

Hint: You might be able to extend your answer to Problem 5 in HW 2.

Answer: Firstly we introduce a few notations:

- $P(i, j)$: the minimum number of edit operations to transform S_1 to S_2, in which $S_1[i]$ is aligned to $S_2[j]$.
- $Q(i)$: the minimum number of edit operations to transform S_1 to S_2, in which $S_1[i]$ is aligned to $S_2[j]$.
- $R(i, j)$: the minimum number of edit operations to transform S_1 to S_2, in which $S_1[i]$ is aligned to $S_2[j]$ or deleted, i.e.

$$R(i, j) = \min\{P(i, j), Q(i)\} \tag{1}$$

- $D(i, j)$: the minimum number of edit operations to transform $S_1[1 : i]$ to $S_2[1 : j]$.
- $E(i, j)$: the minimum number of edit operations to transform $S_1[i : m]$ to $S_2[j : n]$.
- $M(i, j)$: the number of optimal alignments between $S_1[1 : i]$ to $S_2[1 : j]$. This is the variable we defined in Probelm 5, HW2.
- $N(i, j)$: the number of optimal alignments between $S_1[i : m]$ to $S_2[j : n]$.

Problem 3 continued on next page...

- $L(i)$: the number of optimal alignments between S_1 and S_2, in which $S_1[i]$ is deleted.
- $K(i,j)$: the number of optimal alignments that align $S_1[i]$ to $S_2[j]$, or align $S_1[i]$ with a space (i.e. $S_1[i]$ is deleted).

To compute all $K(i,j)$, we would like to make use of the following facts:

- For an optimal alignments where $S_1[i]$ is aligned to $S_2[j]$, $S_1[1:i-1]$ must be optimally aligned to $S_2[1:j-1]$, and $S_1[i+1:m]$ must be optimally aligned to $S_2[j+1:n]$. Consequently,

$$P(i,j) = D(i-1,j-1) + E(i+1,j+1) + t(i,j). \tag{2}$$

  And the total number of this kind of alignments, i.e. the total number of combination of an optimal alignment between $S_1[1:i-1]$ and $S_2[1:j-1]$ and an optimal alignment between $S_1[i+1:m]$ and $S_2[j+1:n]$, is simply $M(i-1,j-1)N(i+1,j+1)$.

- For an optimal alignments where $S_1[i]$ is deleted, there must exist some $k \in \{0,\dots,n\}$ such that $S_1[1:i-1]$ is optimally aligned to $S_2[1:k]$ and $S_1[i+1:m]$ is optimally aligned to $S_2[k+1:n]$. Such k's are determined so that the total number of operation is minimized. Consequently,

$$Q(i) = \min_{k=0,\dots,n} (D(i-1,k) + E(i+1,k+1) + 1). \tag{3}$$

  And the total number of this kind of alignments, i.e. the total number of combination of an optimal alignment between $S_1[1:i-1]$ and $S_2[1:k]$ and an optimal alignment between $S_1[i+1:m]$ and $S_2[k+1:n]$ for all feasible k, is simply

$$L(i) = \sum_{\substack{k=0,\\ Q(i)=D(i-1,k)+E(i+1,k+1)+1}}^{n} M(i-1,k)N(i+1,k+1). \tag{4}$$

- $D(i,j)$ and $E(i,j)$ have very similar recurrent structure, which is given in the class:

$$D(i,j) = \min\{D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)\} \tag{5}$$
$$D(i,0) = i, D(0,j) = j \tag{6}$$

$$E(i,j) = \min\{E(i+1,j)+1, E(i,j+1)+1, E(i+1,j+1)+t(i,j)\} \tag{7}$$
$$E(i,n+1) = m-i+1, E(m+1,j) = n-j+1 \tag{8}$$

- Similarly, $M(i,j)$ and $N(i,j)$ have very similar recurrent structure, which we have analyzed in Problem 5 HW2:

$$\begin{aligned} M(i,j) = {} & M(i,j-1)\delta(D(i,j), D(i,j-1)+1) + M(i-1,j)\delta(D(i,j), D(i-1,j)+1) \\ & + M(i-1,j-1)\delta(D(i,j), D(i-1,j-1)+t(i,j)) \end{aligned} \tag{9}$$
$$M(i,0) = M(0,j) = 1 \tag{10}$$

$$\begin{aligned} N(i,j) = {} & N(i,j+1)\delta(E(i,j), E(i,j+1)+1) + N(i+1,j)\delta(E(i,j), E(i+1,j)+1) \\ & + N(i+1,j+1)\delta(E(i,j), E(i+1,j+1)+t(i,j)) \end{aligned} \tag{11}$$
$$N(i,n+1) = N(m+1,j) = 1 \tag{12}$$

  where $\delta(x,y) = 1$ if $x = y$ and $\delta(x,y) = 0$ otherwise.

Problem 3 continued on next page...

Algorithm 3 Compute $K(i,j)$: the number of optimal alignments that align $S_1[i]$ to $S_1[j]$ or delete $S_1[i]$ for $i = 1, \ldots, m$ and $j = 1 \ldots, n$.

1: Compute all $D(i,j)$ and $M(i,j)$ with DP following the recurrence Eq. (5)(6)(9)(10). This algorithm has already been defined in the class and in HW2.
2: Compute all $E(i,j)$ and $N(i,j)$ with DP following the recurrence Eq. (7)(8)(11)(12). This algorithm is very similar to the one that computes all $D(i,j)$ and $M(i,j)$ and a description is thus omitted.
3: Given all $D(i,j)$ and $E(i,j)$, compute all $P(i,j)$, $Q(i)$ and then $R(i,j)$ with Eq. (2)(3)(1), respectively.
4: Given all $M(i,j)$, $N(i,j)$, $D(i,j)$, $E(i,j)$ and $Q(i)$, compute all $L(i)$ with Eq. (4).
5: **return** all $K(i,j)$ computed with Eq. (13).

- Once $P(i,j)$, $Q(i)$, $R(i,j)$, $M(i,j)$, $N(i,j)$ and $L(i)$ are available, $K(i,j)$ can be simply computed as

$$K(i,j) = L(i)\delta(R(i,j), Q(i)) + M(i-1, j-1)N(i+1, j+1)\delta(R(i,j), P(i,j)) \tag{13}$$

  Namely, if deleting $S_1[i]$ results in the optimal alignments, we count all this kind of optimal alignments; if aligning $S_1[i]$ with $S_2[j]$ results in the optimal alignments, we count all this kind of optimal alignments.

The overall DP algorithm to compute all $K(i,j)$ is given in Algorithm 3. The 2 DP algorithm in line 1 and line 2 has $O(nm)$ complexity. The computation of nm $P(i,j)$, $R(i,j)$ and m $L(i)$, $Q(i)$ each have $O(nm)$ complexity. Finally, the computation of all $K(i,j)$ also have $O(nm)$ complexity. Consequently, Algorithm 3 has $O(nm)$ complexity.

Problem 4

Here is another Four-Russians approach to doing bit-matrix multiplication of $A \times B = C$. Assume that A and B are both of dimension n n. Instead of doing the preprocessing of B (which is how the Four Russians for bit-matrix mult. was done in class), we will preprocess A. Suppose A is dimension $n \times n$, and n is a multiple of q (to be set later). We partition A into squares of dimension $q \times q$. So there are n^2/q^2 of these squares, and they are indexed by (s,t) where s and t both range from 1 to n/q. For each pair (s,t), build a table of size 2^q, one cell for each possible binary number of length q. The cells in that table are indexed by the binary numbers 0 through $2^q - 1$. For the cell in the (s,t) table, indexed by binary number b, compute the bit-matrix multiplication of square (s,t) times b. The result is a vector of length q. That is the preprocessing for A.

Problem 4(a)

In the RAM model, we can follow a pointer, or use an index, or lookup a value, or take the OR of two vectors in constant time, provided that each pointer, index, value or vector only uses $\log m$ bits, where m is the number of bits used to represent the input. In our case, m is n^2. However, the bit-matrix multiplication of two vectors of size q, takes time q. How much time does the preprocessing of A take in the RAM model?

Answer: There are n^2/q^2 tables to build. To process the (s,t)-th table $T_{s,t}$ alone, we need to compute 2^q multiplications between a q-by-q matrix and a q-by-1 vector. Each matrix-vector multiplication compose of q vector-vector multiplication which takes time q. In summary, the preprocessing of A takes

$$t_A = \frac{n^2}{q^2} \cdot 2^q \cdot q \cdot q = n^2 2^q.$$

However, there are smarter ways to bulid each $T_{s,t}$. Since entries in $T_{s,t}$ represent all bit-wise OR of all subset of columns of the (s,t)-th block of A, if computed in the right order, each entry in $T_{s,t}$ can be computed

using a single bit-wise OR of 2 q-bit vector. With this method, the preprocessing of A takes

$$t_A = O\left(\frac{n^2}{q^2} \cdot 2^q \cdot q\right) = O\left(\frac{n^2 2^q}{q}\right).$$

Problem 4(b)

After preprocessing of A, we want to compute $A \times B$. We will view that as n multiplication $A \times B_j$, where B_j is the j column of B, and where j ranges from 1 to n. That is, the j column of C is A times the j column of B.

To do multiplication $A \times B_j = C_j$, we divide B_j and C_j into n/q groups of size q each.

Explain how to use the preprocessed tables to compute the first group of size q of C_j.

Answer: Denote the (s,t)-th q-by-q block in A as $A(s,t)$. Denote the t-th group of size q in B_j, C_j as $B_j(t)$, $C_j(t)$, respectively. We have

$$C_j(1) = \sum_{t=1}^{n/q} A(1,t)B_j(t)$$

in which $A(1,t)B_j(t)$ is the $B_j(t)$-th element in the $(1,t)$-th table $T_{1,t}$. In summary, we only need to look up $T_{1,t}[B_j(t)]$ for $t = 1, \ldots, n/q$, and then perform $n/q - 1$ vector summation of size q to compute the first group of size q of C_j.

Problem 4(c)

Estimate the total time needed to compute C_j, using the preprocessed tables, under the RAM model.

Answer: As analyzed in the previous problem, to compute each group of size q in $C_j(s)$, $s = 1, \ldots, n/q$, we need to perform n/q table look-up and $n/q - 1$ q-bit vector summation. In RAM model, the time to compute C_j is

$$t_{C_j} = O\left(\frac{n}{q} \cdot \frac{n}{q} \cdot q\right) = O\left(\frac{n^2}{q}\right).$$

Problem 4(d)

Show that this approach can compute the bit-matrix multiplication in $O(n^3/\log n)$, by picking q to be $\epsilon \log n$, for any $\epsilon > 1$.

Answer: Since all C_j where $j = 1, \ldots, n$ needs to be computed, taking into account of the cost to preprocess A, the total amount of time to compute C is:

$$\begin{aligned} t_C &= t_A + n t_{C_j} = O\left(n^2 2^q + \frac{n^3}{q}\right) \\ &= O\left(n^{2+\epsilon} + \frac{n^3}{\epsilon \log n}\right) \\ &= O\left(\frac{n^3}{\log n}\right) \end{aligned}$$

Problem 4 [Problem 4(d)] continued on next page...

for all $0 < \epsilon < 1$. If we use the other approach to build each $T_{s,t}$, then

$$\begin{aligned} t_C &= t_A + nt_{C_j} = O\left(\frac{n^2 2^q}{q} + \frac{n^3}{q}\right) \\ &= O\left(\frac{n^{2+\epsilon}}{\epsilon \log n} + \frac{n^3}{\epsilon \log n}\right) \\ &= O\left(\frac{n^3}{\log n}\right) \end{aligned}$$

for all $0 < \epsilon \leq 1$.