CSE 4/531
Solution 03

1. (20 points) Let M be an $n \times n$ matrix with each entry having a real number. Design a dynamic programming algorithm to find a longest sequence $S = (m_{i_1,j_1}, m_{i_2,j_2}, \cdots m_{i_k,j_k})$ such that $i_r < i_{r+1}, j_r < j_{r+1}$, and $m_{i_r,j_r} > m_{i_{r+1},j_{r+1}}$ for all $1 \leq r \leq k$. You should make your algorithm run as fast as possible.

ANS: Without loss generality, we assume that the last number m_{i_k,j_k} of S is $M[i,j]$ for some $1 \leq i,j \leq n$. It means that $i_k = i$, $j_k = j$ and $m_{i_k,j_k} = M[i,j]$. Because of decreasing property for indexes $(i_r, j_r), 1 \leq r \leq k$, we have $i_{k-1} \leq i-1$ and $j_{k-1} \leq j-1$. It also implies that $(m_{i_1,j_1}, m_{i_2,j_2}, \cdots, m_{i_{k-1},j_{k-1}})$ is a longest decreasing sequence of a sub-matrix $M[1 \sim i', 1 \sim j']$ for some $1 \leq i' < i$ and $1 \leq j' < j$.

From the above discussion, we immediately have the following recursive formula: Let $L[i,j]$ denote the length of a longest decreasing sequence $S = (m_{i_1,j_1}, m_{i_2,j_2}, \cdots, m_{i_k,j_k})$ with (1): $m_{i_k,j_k} = M[i,j]$, (2): $i_k = i$ and (3): $j_k = j$ in the matrix $M[1 \sim i, 1 \sim j]$.

$$L[i,j] = \max_{\substack{i'<i, j'<j and \\ M[i',j']>M[i,j]}} \{1 + L[i',j']\}. \tag{1}$$

Next, how to complete the table L? The following two pseudocodes are approaches to find the length of a longest sequence of M and a longest sequence S. Also, we provide the time complexity analysis for this question.

Algorithm 1 Compute the length of a longest sequence of M

for $1 \leq i \leq n$ **do**
    $L[i,1] := 1$
end for
for $1 \leq i \leq n$ **do**
    $L[1,j] := 1$
end for
for $1 \leq i \leq n$ **do**
    for $1 \leq j \leq n$ **do**
        Find a pair of indexes (i',j') satisfying Equation (1).
        $L[i,j] := 1 + L[i',j']$
    end for
end for

Since we have n^2 entries in L and each entry $L[i,j], 1 \leq i,j \leq n$ takes $(i-1) \times (j-1)$ time to find the best i' and j', the total time complexity is

$$\sum_{1 \leq i,j \leq n} (i-1) \times (j-1) = O(n^4).$$

Algorithm 2 FindLongestSequence(L, M)

Find a maximum $L[i, j]$ in L.
$i_r := i$ and $j_r := j$.
Assign $M[i_r, j_r]$ to S.
while $i_r > 1$ && $j_r > 1$ **do**
  search the sub-matrix $L[1 \sim i_r - 1, 1 \sim j_r - 1]$ to find a pair (i, j) with $L[i_r, j_r] = L[i, j] + 1$.
  $i_r := i$.
  $j_r := j$.
  Add $M[i_r, j_r]$ to the head of S.
end while
  return S;

2. (20 points) Let P be a convex polygon with n vertices. A triangulation of P is an addition of a set of non-crossing diagonals (which connect non-neighboring vertices of P) such that the interior of P is partitioned by the set of diagonals into a set of triangles. The weight of each diagonal is the Euclidean distance of the two vertices it connects. The weight of a triangulation is the total weight of its added diagonals. Design a dynamic programming algorithm to find a minimum weighted triangulation of P. You should make your running time as short as possible.

ANS: This question is similar to Minimum Weight of Matrix Multiplication Problem. Hence we have a similar recursive formula:

Assume the polygon $P = (1, 2, \cdots, n)$. Then, let $T[i, j]$ be the weight of a minimum triangulation for the polygon $(i, i+1, \cdots, j-1, j)$ and $w(a, b)$ be the Euclidean distance between a and b. We can observe that given a minimum weighted triangulation for P with cost $T(1, n)$, the edge $(1, n)$ will company with some point k to form the triangle $\triangle_{1,k,n}$ and $T[1, n] = T[1, k] + w(\triangle_{1,k,n}) + T[k, n]$ where $w(\triangle_{i,k,j})$ is $w(i, j) + w(j, k) + w(i, k)$. Then we have

$$T[i,j] = \begin{cases} \min_{i<k<j}\{T[i,k] + T[k,j] + w(\triangle_{i,k,j})\} & \text{if } i < j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Next, how to complete the table T? The following two pseudocodes are approaches to find the weight of a minimum triangulation of P and a minimum triangulation of P. Also, we provide the time complexity analysis for this question.

Algorithm 3 Compute Minimum Triangulation of P

Initialize two $n \times n$ matrics T and K.
for $1 \leq i \leq n$ **do**
  for $i + 2 \leq j \leq n$ **do**
    find a best k for the Equation (2).
    $T[i, j] = T[i, k] + T[k, j] + w(\triangle_{i,k,j})$.
    $K[i, j] = k$.
  end for
end for

Each entry $T[i, j]$ has to take $(j - i - 1)$ time to find the best k and we have $O(n^2)$ entries for T. So, the total time complexity is

$$\sum_{1 \leq i < j \leq n} (j - i - 1) = O(n^3).$$

Algorithm 4 FindMinTriangulation($1, n, K$)

Set $i := 1$, $j := n$ and $k = K[i, j]$.
Draw two diagonals (i, k) and (k, n) in P.
FindMinTriangulation(i, k, K).
FindMinTriangulation(k, j, K).

3. (20 points) Let $G = (V, E)$ be a directed graph modeling a communication network. Each link e in E is associated with two parameters, $w(e)$ and $d(e)$, where $w(e)$ is a non-negative number representing

the cost of sending a unit-sized packet through e, and $d(e)$ is an integer between 1 and D representing the time (or delay) needed for transmitting a packet through e. Design an algorithm to find a route for sending a packet between a given pair of nodes in G such that the total delay is no more than k and the total cost is minimized. Your algorithm should run in $O(k(|E|+|V|))$ time and $O(k|V|)$ space (additional to space for storing the graph).

ANS: Suppose the given starting point is v_s, the ending point is v_e. If there is an edge $e(u,v)$ which is an edge from u to v, let $d(u,v)=d(e)$, $w(u,v)=w(e)$. Here suppose we use Adjacency List representation of graph. Let C be a $|V|\times k$ matrix. Each $C[v,j]$ represents the minimum cost from v_s to v with delay j. Let M be a $|V|\times k$ matrix. Each $M[v,j]$ records the last element before v in the path from v_s to v with the minimum cost $C[v,j]$. Then we have

$$C[v,j]=\begin{cases} 0 & \text{if } v=v_s \\ \min_{u:v\in Adj[u]}\{C[u,j-d(u,v)]+w(u,v)\} & \text{if } \exists u, \text{ s.t. } d(u,v)<j \\ \infty & \text{if } \forall u,\ d(u,v)\geq j \end{cases}$$

So we use BFS to traverse the vertices of the graph, each time we come up with a vertex u, we update matrix C for all the vertices it directs into. The problem can be solved follow these steps:

- Initialization.

$$\begin{aligned} C[v_s,j] &= 0, M[v_s,j]=v_s, j=1,\ldots,k \\ C[v,j] &= \infty, v\neq v_s, j=1,\ldots,k \end{aligned}$$

- BFS travel the vertices of the graph.
- Each time we come up with a vertex u, $\forall v\in Adj[u]$,
  - If $u=v_s$, $d(u,v)\leq k$, then $C[v,j]=w(u,v)$, $M[v,j]=v_s$, $j=d(u,v),\ldots,k$.
  - If $u=v_s$, $d(u,v)>k$, then do nothing.
  - If $u\neq v_s$, $d(u,v)<k$, $C[u,j-d(u,v)]+w(u,v)<C[v,j]$, then $C[v,j]=C[u,j-d(u,v)]+w(u,v)$, $M[v,j]=u$, $j=d(u,v)+1,\ldots,k$.
  - If $u\neq v_s$, $d(u,v)\geq k$, then do nothing.

The pesudocode of the algorithm is shown in Algorithm 5.

The initialization precess need $O(k|V|)$ time. BFS takes $O(|V|+|E|)$ time to visit each vertex. But when we visit each vertex, we compute up to k entries of C and k entries of M. So the total running time would be $O(k(|V|+|E|))$. The extra space needed is for storing C and M. They are $|V|\times k$ matrices, so the space need is $O(k|V|)$.

Algorithm 5 Minimum Cost within k delay.

Input: A directed graph $G = (V, E)$, each e in E has two parameters $w(e)$ and $d(e)$, integer k, two vertices v_s and v_e.
Output: A path from v_s to v_e with minimum cost and delay within k.

Let C be a $|V| \times k$ matrix.
Let M be a $|V| \times k$ matrix.
Let Q be an empty queue.
Let *color* be a vector with length $|V|$.
for $j = 1$ **to** k **do**
    $C[v_s, j] = 0;$
    $M[v_s, j] = v_s;$
end for
for $v \in V, v \neq v_s$ **do**
    $color[v] = white;$
    for $j = 1$ **to** k **do**
        $C[v, j] = \infty;$
    end for
end for
$color[v_s] = grey;$
for each $v \in Adj[v_s]$ **do**
    $color[v] = grey;$
    $Enqueue(Q, v);$
    if $d(v_s, v) \leq k$ **then**
        for $j = d(v_s, v), \ldots, k$ **do**
            $C[v, j] = w(v_s, v);$
            $M[v, j] = v_s;$
        end for
    end if
end for
$color[v_s] = black;$
while $Q \neq \emptyset$ **do**
    $u = Dequeue(Q);$
    for each $v \in Adj[u]$ **do**
        if $color[v] = white$ **then**
            $color[v] = grey;$
            $Enqueue(Q, v);$
        end if
        if $d(u, v) < k$ **then**
            for $j = d(u, v) + 1$ **to** k **do**
                if $C[u, j - d(u, v)] + w(u, v) < C[v, j]$ **then**
                    $C[v, j] = C[u, j - d(u, v)] + w(u, v);$
                    $M[v, j] = u;$
                end if
            end for
        end if
    end for
    $color[u] = black;$
end while
OutputPath(M);

function OutputPath(M)
    $mincost = \min\{C[v_e, j], j = 1, \ldots, k\};$
    if $mincost = \infty$ **then**
        return No such path!
    end if
    Let j be the index s.t. $C[v_e, j] = mincost;$
    $u = v_e;$
    Let S be an empty stack.
    while $u \neq v_s$ **do**

$Push(S, u)$;
$j = j - d(u, v)$;
$u = M[u, j]$;
end while
$Push(S, v_s)$;
while $S \neq \emptyset$ **do**
Print $Pop(S)$;
end while
end function

4. (20 points) Let T be a rooted tree with n nodes and each node v associated with a weight $w(v)$. A subset of nodes S is an independent set of T if no node in S is the child or parent of another node in S. Design a dynamic programming algorithm to find an independent set of T with maximum weight, where the weight of an independent set is the total weight of its nodes. You should make your algorithm run as fast as possible.

ANS: Let (1): r be the root of T, (2): T_v denote the subtree rooted at v, (3): $N^1_T(v)$ denote the children of v in T and (4): $N^2_T(v)$ denote the descendants of v which is two edges away from v in T.

Observation 1 *Given a maximum independent set S containing r, S can be partitioned into $\{S_v | v \in N^2_T(r)\}$ and each S_v is also a maximum independent set of T_v.*

Observation 2 *Given a maximum independent set S not containing r, S can be partitioned into $\{S_v | v \in N^1_T(r)\}$ and each S_v is also a maximum independent set of T_v.*

From the above two observations, we immediately have the following recursive formula:

$$M(T_r) = \max \left\{ w(r) + \sum_{v \in N^2_T(r)} M(T_v), \sum_{v \in N^1_T(r)} M(T_v) \right\}$$

Next, how to complete the table M? The following two pseudocodes are approaches to find the weight of a maximum independent set of T and a maximum independent set of T. First, let $l_T(v)$ be the level of v in T, we can recursively define $l_T(v)$ as follows:

1. for the root r of T, $l_T(r) = 1$ and
2. $\{l_T(v) = l_T(v) + 1 | v \in N^1_T(v)\}$.

Algorithm 6 Compute the Weight of a Maximum Independent Set of T

$L := \max\{l_T(v) | v \in T\}$
for $i = L$ to 1 **do**
for each v with $l_T(v) = i$ **do**
if v is a leaf **then**
$M(T_v) = 1$.
else
$M(T_v) = \max \left\{ w(v) + \sum_{v' \in N^2_T(v)} M(T_{v'}), \sum_{v' \in N^1_T(v)} M(T_{v'}) \right\}$.
end if
end for
end for

Because we have $O(n)$ rooted trees in this dynamic programming formula and each tree rooted at v takes $O(|N^1_T(v)| + |N^2_T(v)|)$ time to execute summations in the recursive formula. Also, we have $\sum_{v \in T} |N^1_T(v)| = O(n)$ and $\sum_{v \in T} |N^2_T(v)| = O(n)$. So, the total time complexity is

$$\sum_{v \in T} |N^1_T(v)| + |N^2_T(v)| = O(n).$$

Algorithm 7 FindMaxIndependentSet(T)

Let r be the root of T and X be a empty set.
if $M(T_r) == w(r) + \sum_{v \in N_T^2(r)} M(T_v)$ **then**
    $X := X \cup \{r\}$
    for each $v \in N_T^2(r)$ **do**
        $X := X \cup$ FindMaxIndependentSet(T_v).
    end for
else
    for each $v \in N_T^1(r)$ **do**
        $X := X \cup$ FindMaxIndependentSet(T_v).
    end for
end if
    return X;

5. (20 points) Let $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$ be two strings with length m and n respectively. Design a dynamic programming based algorithm to convert A into B with minimum cost using the following rules. For a cost of 1, one can delete any letter from a string. For a cost of 2, one can insert a letter in any position. For a cost of 3, one can replace any letter by any other letter. For example, you can convert $A = abcabc$ to $B = abacab$ via the following sequence of operations: *abcabc* with a cost of 3 can be converted to *abaabc*, which with a cost of 1 can be converted to *ababc*, which with a cost of 1 can be converted to *abac*, which with a cost of 2 can be converted to *abacb*, which with a cost of 2 can be converted to *abacab*. Thus the total cost for this conversion is 9 (may not be the cheapest one).

ANS: Let C be a $(m+1) \times (n+1)$ matrix recording the minimum cost of converting $A[1..i]$ to $B[1..j]$. Let M be a $(m+1) \times (n+1)$ matrix recording the last operation. Then

- If A is empty, then to convert A to B, at least we need to insert every element in B into A with cost $5 \times B.length$. And if we want to get the minimum cost, that is enough. So $C[0, j] = 5j$.
- If B is empty, at least we need to delete every element in A with cost $3 \times A.length$. And if we want to get the minimum cost, that is enough. So $C[i, 0] = 3i$.
- If $a_i = b_j$, all we need to do is to convert $A[1..i-1]$ to $B[1..j-1]$. So, $C[i, j] = C[i-1, j-1]$.
- If $a_i \neq b_j$, there are three choices:
  (a) delete $A[i]$, convert $A[1..i-1]$ to $B[1..j]$ with total cost $C[i-1, j] + 3$;
  (b) convert $A[1..i]$ to $B[1..j-1]$, then insert $B[j]$ with total cost $C[i, j-1] + 5$;
  (c) convert $A[1..i-1]$ to $B[1..j-1]$, then replace $A[i]$ with $B[j]$ with total cost $C[i-1, j-1] + 7$.

  We'll choose the one with minimum cost.

$$C[i,j] = \begin{cases} 5j, & \text{if } i = 0; \\ 3i, & \text{if } j = 0; \\ C[i-1, j-1], & \text{if } i > 0,\ j > 0,\ a_i = b_j; \\ \min\{C[i-1, j] + 3, C[i, j-1] + 5, C[i-1, j-1] + 7\}, & \text{if } i > 0,\ j > 0,\ a_i \neq b_j. \end{cases}$$

To compute $C[i, j]$ we need $C[i-1, j]$, $C[i, j-1]$ and $C[i-1, j-1]$. So the algorithm will compute C from left to right, form up to bottom.

To output the solution, use $(m+1) \times (n+1)$ matrix to record the last operation. If $A[i] = B[j]$, we do nothing in this step, so $M[i, j] = N$. In case of $A[i] \neq B[j]$, if the delete operation leads to the minimum cost, let $M[i, j] = D$; if the insert operation leads to the minimum cost, let $M[i, j] = I$; if the replace operation leads to the minimum cost, let $M[i, j] = R$. The pseudocode of the algorithm is show in Algorithm 8.

In the algorithm, we compute each entry of matrix C and M. It takes constant time to compute each entry. So the total running time would be $\Theta(mn)$.

Algorithm 8 Convert String.

Input: String $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$.
Output: Steps of converting A into B with minimum cost.

Let C be a $(m+1) \times (n+1)$ matrix.
Let M be a $(m+1) \times (n+1)$ matrix.
for $i = 0$ **to** m **do**
  $C[i, 0] = 3i$;
end for
for $j = 0$ **to** n **do**
  $C[0, j] = 5j$;
end for
for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    Update(L,M,i,j);
  end for
end for
Output(L,M);

---

function UPDATE(L,M,i,j)
  if $a_i = b_j$ **then**
    $C[i, j] = C[i-1, j-1]$;
    $M[i, j] = N$; ▷ No operation is made.
  else
    if $C[i-1, j] + 3 = \min\{C[i-1, j] + 3, C[i-1, j-1] + 7, C[i, j-1] + 5\}$ **then**
      $C[i, j] = C[i-1, j] + 1$;
      $M[i, j] = D$; ▷ Delete.
    else if $C[i-1, j-1] + 7 = \min\{C[i-1, j] + 3, C[i-1, j-1] + 7, C[i, j-1] + 5\}$ **then**
      $C[i, j] = C[i-1, j-1] + 7$;
      $M[i, j] = R$; ▷ Replace.
    else
      $C[i, j] = C[i, j-1] + 5$;
      $M[i, j] = I$; ▷ Insert.
    end if
  end if
end function
function OUTPUT(L,M) ▷ Print in reverse order.
  $i = m$, $j = n$;
  while $j > 0$ **do**
    Switch($M(i, j)$):
    Case N: $i--$; $j--$; break;
    Case R: $A[i] = b_j$; Print A; $i--$; $j--$; break;
    Case I: $A = A[1..i]b_jA[i+1..A.length]$; Print A; $j--$; break;
    Case D: $A = A[1..i-1]A[i+1..A.length]$; Print A; $i--$; break;
    end switch
  end while
end function