Homework #4  Due: March 7–10, 2017

(25 pts) 1. (Eliminating a Subsequence.)

You’re given a string $S$ of length $n$, and a string $F$ of length $m$. $F$ is called the forbidden word. Both of these strings are from some finite alphabet $\Sigma$.

In the following parts you are to give an algorithm to compute the minimum number of letters that must be deleted from $S$ to form a new string $S'$ such that there is no subsequence of $S'$ that equals $F$. In other words $|\text{LCS}(S', F)| < |F|$.

(a) Assume that $F$ contains no repeated letter. For full points give an algorithm that runs in $O(n + m)$ time. (A weaker time bound of $O(nm)$ will get you partial credit.)

(b) Removing the assumption of part (a), give an algorithm that runs in time $O(nm)$.

Solution: Let’s start with part (b). Let $F = f_1, f_2, \ldots, f_m$ and $S = s_1, s_2, \ldots, s_n$. Denote $s[1..i]$ to be the prefix of a string $s$ from $s_1$ to $s_i$. We define $c[i][j]$ to be the minimum number of letters we need to delete from $s[1..i]$ to avoid subsequence $f[1..j]$. We then compute all the elements in the table of $c[i][j]$ from top to bottom and from left to right.

To calculate $c[i][j]$, there are two cases:

(a) $s_i \neq f_j$. In this case $s_i$ will not affect the existence of $f[1..j]$. “$s[1..i]$ does not contain $f[1..j]$” is equivalent to “$s[1..i-1]$ does not contain $f[1..j]$”. Hence we copy the value $c[i-1][j]$ to $c[i][j]$. We call it a “copy step”.

(b) $s_i = f_j$. We call it an “update step”. We have two choices:

i. Keep $s_i$. In this case we need to make sure that the rest part of $s$ ($s[1..i-1]$) does not contain $f[i..j-1]$. The minimum deletion is $c[i-1][j-1]$.

ii. Delete $s_i$. In this case we only need to worry about $s[1..i-1]$, which means the minimum deletions is $c[i-1][j] + 1$ (plus one because of the deletion of $s_i$).

Then $c[i][j]$ is the better one of the above two choices: $c[i][j] = \min(c[i-1][j-1], c[i-1][j] + 1)$.

Initially all $c[i][0] = +\infty$ for all $i$, which means that you need to use $+\infty$ deletions to make sure a string does not contain an empty subsequence. In other words, when you encounter $s_i = f_1$, you cannot choose to keep it. The only choice you have is to delete it. The final answer is $c[n][m]$.

To compute $c[i][j]$ only costs constant time as long as $c[i-1][j]$ and $c[i-1][j-1]$ have been computed. Hence the algorithm runs in time $O(nm)$. The correctness can be proved by induction.

For part (a), we know that $F$ have no repeated letter. This means that for each $s_i$, there is at most one value of $j$ that $s_i = f_j$. First we establish a lookup table $t[x](x \in \Sigma)$ to
locate a letter \( x \) in \( f \): \( t[x] = j \) if \( x = f_j \), and \( t[x] = -1 \) otherwise. This costs \( O(m) \) time. The “update step” will be done at most once for each \( s_i \) at the point \( c[i][j] \) where \( s_i = f_j \), so there will be only \( O(n) \) update steps. If we can save the cost of all copy steps, the complexity of the algorithm is \( O(m + n) \). Notice that we only need to use \( c[i-1][j] \) to compute \( c[i][j] \), we compress the table \( c[i][j] \) into a list \( c[j] \), s.t.:

\[
c[j] = \min\{c[j-1], c[j]+1\}
\]

We process \( s \) from left \( (s[1]) \) to right \( (s[n]) \). We call it step \( i \) when we are processing \( s[i] \). In each step \( i \) we first locate \( s[i] \) in \( f \) by looking at \( j = t[s[i]] \). If \( j = -1 \) we go to the next step, otherwise we update \( c[j] \) using Equation 1. In this part the value \( c[j] \) at step \( i \) is the exact value \( c[i][j] \) in part (b). The final value of \( c[m] \) is the answer we’re looking for. The copy steps are now saved because \( c[j] \) at step \( i \) is exact \( c[j] \) at step \( i-1 \) if no update occurs. The total cost is \( O(m + n) \).

Another way to interpret Equation 1 is as follows: we keep a set of variables \( c_1, c_2, \ldots, c_m \). Initially they’re all 0. Say we’ve finished processing \( s_i \). \( c_j \) represents the number of deletions needed to make it so that the string \( s_1, s_2, \ldots, s_i \) does not contain \( f_1, f_2, \ldots, f_j \) as a subsequence. Now we start to process \( s_{i+1} \). To eliminate the subsequence \( f_1, \ldots, f_k \) from \( s_1, \ldots, s_{i+1} \) we can either eliminate the subsequence \( f_1, \ldots, f_{k-1} \) from \( s_1, \ldots, s_i \) then keep \( s_{i+1} \), or we eliminate the subsequence \( f_1, \ldots, f_k \) from \( s_1, \ldots, s_i \) and then delete \( s_{i+1} \). This update takes constant time for each character of \( S \). The final value of \( c_m \) is the answer we’re looking for.

(25 pts) 2. (Locating stores in DEN.) Having graduated, you work for a company that is charged with locating \( s \geq 1 \) food shops in Denver airport. This terminal is narrow and long, and you model it as a straight line. There are \( n \geq s \) gates, which are located at positions \( 0 = a_1 < a_2 < \ldots < a_n \) on the line, with distances on the line modeling walking distances between the points. You are allowed to locate the shops anywhere on the line. If you decide to locate the \( s \) shops at positions \( b_1, b_2, \ldots, b_s \), the “pain” of this solution is the distance from an average gate to its closest shop, i.e.,

\[
\frac{1}{n} \sum_{i=1}^{n} \left( \min_{j=1}^{s} |a_i - b_j| \right).
\]

Give a dynamic programming algorithm to find the solution (i.e., the location of shops) that incurs the least pain, and that runs in time \( O(n^3) \). It should not depend on the values of \( a_i \).

Hint: consider subproblems that correspond to a prefix \( \{a_1, \ldots, a_t\} \) of the gates and a number of shops \( s' \leq s \). One more useful fact is that we need only consider solutions that place shops at a subset of the gate locations \( \{a_i\} \) (i.e., \( \{b_j\} \subseteq \{a_i\} \)); this can be seen most easily in the case of \( s = 1 \) where if the shop is between \( a_i \) and \( a_{i+1} \), we can simply move it in whichever direction has more gates, or to either one if there is a tie. You may use this fact without proof, but convince yourself that it is true.

**Solution:** First of all, let’s try to minimize the sum of the distances rather than the average, since this is just scaling by a factor of \( n \) that does not depend on the solution. We will call this the “total cost”.


Let \( T(i, s') \) be the total cost of the best solution just for the interval \( \{a_1, \ldots, a_i\} \) when we open \( s' \) shops within these \( i \) locations, and allocate all these \( i \) locations to these shops. If we compute this for all \( 1 \leq i \leq n, 1 \leq s' \leq s \), we will be done, since we need to return \( T(n, s) \).

To compute \( T(i, s') \): since each gate will go to their nearest shop, the first few gates go to the first shop, the next few to the next shop, and the last few to the last shop. So consider at the “rightmost” of these \( s' \) locations to be opened. The gates which go to this shop will be \( \{j^* + 1, \ldots, i\} \) for some value of \( j^* \), and the other gates \( \{1, \ldots, j^*\} \) will be served (optimally) by the other \( s' - 1 \) shops. Hence,

\[
T(i, s') = \left( T(j^*, s' - 1) + C(j^* + 1, i) \right),
\]

where \( C(j + 1, i) \) is the cost of opening one shop in the interval \( \{j + 1, \ldots, i\} \) and serving the gates optimally. (Of course we don’t know this unknown “breakpoint” \( j^* \).) Moreover, if we choose some other “breakpoint” \( j \), open some solution of with \( s' - 1 \) locations in \( \{1, \ldots, j\} \) and a single shop in \( \{j + 1, \ldots, i\} \) on the right, this will cost more than the optimal solution (and also cost at most \( T(j, s' - 1) + C(j + 1, i) \)). So we

\[
T(i, s') = \min_{j < i} \left( T(j, s' - 1) + C(\{j + 1, i\}) \right),
\]

And

\[
T(i, 1) = C(1, i),
\]

and also

\[
T(i, j) = 0 \text{ for } j \geq i.
\]

For each \( p \leq q \), the quantity \( C(p, q) \) can be computed in time \( O(q - p + 1)^2 \) easily: try all \( q - p + 1 \) locations, compute their costs in time \( O(q - p + 1) \), and take the best one. However, it can also be done in linear time \( O(q - p + 1) \). Approach #1: we start off by computing the sum of distances from \( a_p \) to all \( a_p, \ldots, a_q \). Then for each integer \( r \in [p + 1, q] \), we can compute the sum of distances of \( a_r \) to all \( a_p, \ldots, a_q \) by taking this same quantity for \( a_{r-1} \), adding \( (r - p)|a_r - a_{r-1}| \) and subtract \( (q - r + 1)|a_r - a_{r-1}| \). Approach #2: the optimal single location will be at the median of \( a_p, \ldots, a_q \). (Why?) Hence the computation of all the \( C(\cdot, \cdot) \) values can be done in \( O(n^3) \) time.

Now for the \( T(\cdot, \cdot) \) values: there are \( O(ns) \) problems (one for each \( i, s' \)). Each considers at most \( n \) subproblems within it, and hence takes \( O(n) \) time to compute the minimum (assuming the C’s are precomputed). That takes a total of \( O(n^2 s) \) time more. Since \( s \leq n \) the total time is \( O(n^3) \).

Exercise: since \( C(p, q) \) is the total distance from the median of \( a_p, \ldots, a_q \) to these values, how can you compute the entire matrix \( C(p, q) \) faster than \( O(n^3) \)?

(25 pts) 3. (The 61C Bus.) The eternal dilemma is: wait for the bus, or just walk to school? And if indeed you’re to wait, how long before you give up? This is what we’ll solve.

We model this process as a zero-sum game. Walking to work takes \( A \) units of time. The bus takes \( B \) units of time. The bus arrives at time 1 or 2 or 3, etc. If the bus
arrives at time \( t \geq 1 \) and you take it, you reach school at time \( t + B \). However, if you have waited \( w \geq 0 \) units of time, and if the bus has not yet arrived, you may decide to walk, which takes \( A \) units of time, so you reach at time \( w + A \). (Once you decide to start walking you cannot catch the bus any more.)

Your actions are “wait \( w \) time; if no bus yet, walk”: one action for every integer \( w \geq 0 \). (So \( w = 0 \) means you don’t wait at all.) The adversary’s actions are: “bus arrives at time \( t \)”, one for every positive integer \( t \geq 1 \).

For instance if \( w = 2 \) and Will \( t \leq 2 \) then you catch the bus and arrive at time \( t + B \), but if \( w = 2 \), \( t = 3 \) then you wait 2 time steps, (leave just before the bus arrives) and reaching at time \( 2 + A \).

Given the pair of actions \((t, w)\), your pain (aka. the payoff to the adversary) is the ratio: (the time it took you) divided by (the optimum time it would take you if you knew when the bus were to arrive).

(a) Let \( R \) be the matrix of payoffs to the adversary whose rows \( t \) are the adversary’s actions and columns \( w \) are your actions, then for any \( t \geq 1 \) and \( w \geq 0 \), write down the mathematical expression for the payoff to the adversary.

\[
R_{tw} = \begin{cases} 
  \frac{t + B}{\min(t + B, A)} & \text{if } t \leq w \\
  \frac{w + A}{\min(t + B, A)} & \text{else}
\end{cases}
\]

For the rest of the parts, assume that \( B = 1 \) and \( A = 3 \).

(b) This game has an infinite number of columns and an infinite number of rows. Let’s add one more row, called “row \( \infty \)” for the scenario that the bus never arrives. Argue that without loss of generality, we can assume the adversary chooses only rows \( t = 1 \) or \( t = \infty \). Formally, argue that for any \( t \geq 2 \) we have \( R_{tw} \leq R_{\infty w} \) for all \( t \). This means that your strategy achieving expected payoff \( V \) in a world with only those two scenarios possible (bus arrives at time \( t = 1 \), or it never arrives) will achieve also payoff \( V \) over the whole range of scenarios.

\textbf{Solution:} First, for \( t \geq 2 \), the denominator \( \min(t + B, A) = \min(t + 1, 3) = 3 \). We consider three cases:

\textbf{Case 1}(\( t \leq w \)): In this case, \( R_{tw} = \frac{t + 1}{3} \) and \( R_{\infty w} = \frac{w + 3}{3} \). We know \( \frac{t + 1}{3} < \frac{w + 3}{3} \) because \( t \leq w \). Thus, \( R_{tw} < R_{\infty w} \).

\textbf{Case 2}(\( w < t \)): In this case, \( R_{tw} = \frac{w + 3}{3} \) and \( R_{\infty w} = \frac{w + 3}{3} \), so \( R_{tw} = R_{\infty w} \).

Since for any \( t \geq 2 \), we have \( R_{tw} \leq R_{\infty w} \), we can now assume the adversary chooses only rows \( t = 1 \) or \( t = \infty \).

(c) Now that the game has just two rows, argue that the minimax-optimal strategy can safely put probability 0 on all columns except for \( w \in \{0, 1\} \). Formally, argue that for any \( w > 1 \) we have \( R_{tw} \geq R_{t1} \) for \( t \in \{1, \infty \} \).

\textbf{Solution:} We will consider two cases based on the value of \( t \) (remember \( w > 1 \)):
Case 1 ($t = 1$): In this case, $R_{tw} = \frac{2}{2} = 1$ and $R_{t1} = \frac{1}{1} = 1$. Thus, $R_{tw} = R_{t1}$.

Case 2 ($t = \infty$): In this case, $R_{tw} = \frac{w+3}{3}$ and $R_{t1} = \frac{1+3}{3} = \frac{4}{3}$. We know $\frac{w+3}{3} > \frac{4}{3}$ because $w > 1$. Thus, $R_{tw} > R_{t1}$.

Thus, we have proven that for any $w > 1$, we have $R_{tw} \geq R_{t1}$ for $t \in \{1, \infty\}$ and thus the minimax-optimal strategy can safely put probability 0 on all columns except for $t \in \{0, 1\}$.

(d) Now write down the 2-by-2 matrix that results. Write down the LP for the minimax-optimal strategy for you, the column player, and solve it to find the value of the game.

Solution: The 2-by-2 matrix is as follows:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{2}{3}$</td>
<td>$\frac{2}{3} = 1$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\frac{2}{3} = 1$</td>
<td>$\frac{4}{3}$</td>
</tr>
</tbody>
</table>

Now if $q$ and $1 - q$ are the probabilities that we pick columns $w = 0$ and $w = 1$, then we want to solve

$$\min_{0 \leq q \leq 1} \max(\frac{3}{2}q + (1-q), q + \frac{4}{3}(1-q))$$

This is not an LP, but we can rewrite this as:

minimize $v$

subject to $(3/2)q + 1 - q \leq v$

$q + (4/3)(1-q) \leq v$

$0 \leq q \leq 1$

We can solve this by plotting this: we get $q = 2/5$ and the optimal value is $6/5$.

(e) Finally, suppose there is an bus driver (row player) who controls the arrival time of the bus, and is trying to cause you the most pain (get the highest payoff for herself). Solve for the row player’s strategy.

Solution: Now if $p$ and $1 - p$ are the probabilities that we pick rows $w = 1$ and $w = \infty$, then we want to solve

$$\max_{0 \leq p \leq 1} \min(\frac{3}{2}p + (1-p), p + \frac{4}{3}(1-p))$$

The LP looks very similar.

maximize $v$

subject to $v \leq (3/2)p + 1 - p$

$v \leq p + (4/3)(1-p)$

$0 \leq p \leq 1$

Again the solution is $p = 2/5$ and the optimal value is $6/5$. 
(25 pts) 4. **Travelling Salesperson Problem**

In this problem you will implement an algorithm that can solve the Travelling Salesperson Problem (TSP). We discussed this problem in Lecture #11. The input to your program will be a connected graph $G$ with $n$ vertices and $m$ edges. Each edge $\{u, v\}$ has a non-negative length in each direction, and $\text{len}(u, v)$ may be different from $\text{len}(v, u)$. Your program will compute the length of the shortest TSP tour for the given graph, and also output the tour. The time limit is 10 seconds. For Python we’ll give 1 minute.

**Input**

First line consists of two positive integers $n$ and $m$. $n \leq 20$ and $n − 1 \leq m \leq n(n − 1)/2$. The following $m$ lines each contain four integers that represent an edge. These numbers are $i \ j \ d_{i,j} \ d_{j,i}$. This means that there is an edge $\{i, j\}$ in the graph and the directed length from $i$ to $j$ is $d_{i,j}$ and the directed length from $j$ to $i$ is $d_{j,i}$. Each edge will occur at most once in the list in either direction. There are no self loops, and the graph is connected. The vertex numbers are in $[0, n − 1]$, and the lengths of the edges are in $[0, 10^6]$.

**Output**

The first line of output contains the cost of the shortest travelling salesperson tour for the given graph.

The second line contains list of the cities in the order visited, starting at city 0 and ending at city 0. Each pair of neighboring cities on this list must be an edge in $G$. Any tour which is of minimum cost is acceptable.

See the samples below for formatting details.

**Samples**

For example, if the input is:

```
5 5
0 1 1 1
1 2 1 1
2 3 1 2
3 4 1 1
4 0 1 1
```

Then the output (which is unique in this case) will be:

```
optimal tour cost = 5
tour: 0 1 2 3 4 0
```
Samples

For example, if the input is:

```
6 6
0 1 1 1
1 2 1 1
2 3 1 1
3 1 1 2
2 4 1 1
3 5 1 1
```

Then the output might be:

```
optimal tour cost = 9
tour: 0 1 2 4 2 3 5 3 1 0
```