Homework #9

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **You do not need to turn in these problems. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Instructions: For all the following problems, give the recurrence relation, prove your optimal substructure is correct and calculate the runtime.

Problem 1: Longest Paths

Given an undirected graph $G = (V, E)$ with positive edge weights w_e for each edge $e \in E$, give a dynamic programming algorithm to compute the longest path in G from a given source s that contains at most n edges.
(Hint: Let $A[v, k]$ denote the weight of the longest path from s to node v of at most k edges).

Solution

Initialize $A[v, 1] = w_{sv}$ if $(s, v) \in E$, $A[v, 1] = 0$ otherwise.
$A[v, k] = \max\{A[v, k-1], \max_{u:(u,v)\in E}\{w_{uv} + A[u, k-1]\}\}$
Correctness: The problem has optimal substructure. Take a maximum weight path from s to t in atmost k steps. If this path is $\{s, u_1, u_2, \cdots, u_j, t\}$, then the maximum weight path from u_1 to t in atmost $k-1$ steps must be $\{u_1, u_2, \cdots, u_j, t\}$.
Perform induction on k. The base case $k = 1$ is trivial. Assume it is optimal for k. To prove optimality for $k+1$ using contradiction, assume that the algorithm is not optimal. This implies that there exists a $u \in V$ such that the maximum weight path to u in k steps is greater than $A[u, k]$. But since the algorithm is optimal for k, this is impossible. Therefore the algorithm is optimal for $k+1$.
Running time: Any path can have atmost $|V|$ steps. For any k, the cost is $O(|V| + |E|)$ since for each node v the cost of computing $A[v, k]$ is proportional to $1 + degree(v)$. Thus the total runtime is $O(|V|(|E| + |V|))$.

Problem 2: Moving in a Grid

Imagine that you are placed on a grid with n spaces in every row and n spaces in every column. You can start anywhere along the bottom row of the grid, and you must move to the top row of the grid. Each time you move, you can either move directly up (staying in the same column, but moving up a row), up and to the left (moving over one column and up one row), or up and to the right (moving over one column and up one row). You cannot move up and to the left if you are in the leftmost row, and you cannot move up and to the right if you are in the right most row.

Each time you move, you are either paid or pay; that is, every legal move from square x to square y is assigned a real value $p(x, y)$. Sure, $p(x, y)$ can also be 0.

Give a dynamic programming algorithm to compute your sequence of moves to receive the maximum payoff to move from the bottom of the grid to the top of the grid. (Your maximum payoff may be negative.) You must calculate the *value* of the optimal solution (i.e., the payoff) *and*

the solution itself (i.e., the sequence of moves). Again, you can start at any square in the bottom row and end in any square in the top row.

Solution

I'm going to assume that the rows of the grid are numbered starting at 1 (at the top) down to n (at the bottom) and that the columns are numbered similarly left to right. Therefore, the maximum payoff is the max value of any grid square in row 1 (the top row). We start at the bottom (in the n^{th} row to fill in values for our grid. I will use the following recurrence to fill in values for each of the n^2 grid squares on the board, where we refer to the elements in a two dimensional array P (for payoff) by a single index a; $B(a)$ refers to the grid square just below a, $BL(a)$ refers to the grid square below and to the left of a, and $BR(a)$ refers to the grid square below and to the right of a. For all grid squares a in the bottom row, $P[a] = 0$

$$P[a] = \begin{cases} \max\{(P[B(a)] + p(B(a), a)), (P[BR(a)] + p(BR(a), a))\} \\ \qquad \text{if } a \text{ is in the leftmost column} \\ \max\{(P[B(a)] + p(B(a), a)), (P[BL(a)] + p(BL(a), a))\} \\ \qquad \text{if } a \text{ is in the right most column} \\ \max\{(P[B(a)] + p(B(a), a)), (P[BL(a)] + p(BL(a), a)), (P[BR(a)] + p(BR(a), a))\} \\ \qquad \text{otherwise} \end{cases}$$

If we use i and j to index the row and column of the grid square, we can rewrite the above as:

$$P[i, j] = \begin{cases} \max\{(P[i+1, j] + p(B(a), a)), (P[i+1, j+1] + p(BR(a), a))\} \\ \qquad \text{if } a \text{ is in the leftmost column} \\ \max\{(P[i+1, j] + p(B(a), a)), (P[i+1, j-1] + p(BL(a), a))\} \\ \qquad \text{if } a \text{ is in the right most column} \\ \max\{(P[i+1, j] + p(B(a), a)), (P[i+1, j-1] + p(BL(a), a)), \\ \qquad\qquad (P[i+1, j+1] + p(BR(a), a))\} \\ \qquad \text{otherwise} \end{cases}$$

We can fill in the array then with the following procedure:

```
1  for j = 1 to n
2        do P[n, j] = 0
3  for i = n − 1 downto 1
4        do P[i, 1] = max((P[i + 1, j] + p(B(a), a)), (P[i + 1, j + 1] + p(BR(a), a)))
5           for j = 2 to n − 1
6                 do P[i, j] = max((P[i + 1, j] + p(B(a), a)), (P[i + 1, j − 1] + p(BL(a), a)),
                                   (P[i + 1, j + 1] + p(BR(a), a)))
7           P[i, n] = max((P[i + 1, j] + p(B(a), a)), (P[i + 1, j − 1] + p(BL(a), a)))
8  return max_{j:1≤n}(P[1, j])
```

This gets you the *value* of the optimal solution. To recreate the actual solution (i.e., the set of moves), I start at the grid square in the top row with the highest value (call it a). I recreate the solution to this square from the three possibilities below to figure out which grid square I came from (call this b). I output the move (b, a) as the final move in the optimal solution, the recurse, following the same procedure from b. I do this $n - 1$ times until I hit the bottom row. There are $O(n^2)$ values $P[i, j]$ to build up, and each takes constant time to fill in from the results of previous subproblems. So the total running time is $O(n^2)$.

Problem 3: Dynamic Programming and Subsequences

You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, the $s = 100010101$ is an interleaving of x and y since characters 1, 2, 5, 7, 8, 9 form 101101 (a repetition of x), and the remaining characters 3, 4, 6 form 000 (a repetition of y).

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to. We want to make sure that s "unravels" into simple repetitions of x and y. Given an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Solution

Let's suppose that s has n characters. To make things easier to think about, let's consider the repetition of x' of x consisting of exactly n characters and the repetition of y' of y consisting of exactly n characters. Our problem can then be phrases as: is s an interleaving of x' and y'? The advantage of working with these elongated strings is that we don't need to "wrap around" and consider multiple periods of x' and y' since each is already as long as s.

Let $s[j]$ denote the j^{th} character of s and let $s[1:j]$ denote the first j characters of s. We define the analogous notation for x' and y'. We know that if s is an interleaving of x' and y', then its last character comes from either x' or y'. Removing this character (wherever it is), we get a smaller recursive problem on $s[1:n-1]$ and prefixes of x' and y'.

Thus, we consider subproblems defined by prefixes of x' and y'. Let $M[i,j] = yes$ if $s[1:i+j]$ is an interleaving of $x'[1:i]$ and $y'[1:j]$. If there exists such an interleaving, then the final character is either $x'[i]$ or $y'[j]$, and so we have the following basic recurrence:

$M[i,j] = yes$ if and only if $M[i-1,j] = yes$ and $s[i+j] = x'[i]$ or $M[i,j-1] = yes$ and $s[i+j] = y'[j]$.

We can build the following program:

```
1   M[0,0] = yes
2   for k = 1 to n
3         do for all pairs (i,j) such that i + j = k
4               do if M[i-1,j] = yes and s[i+j] = x'[i]
5                     M[i,j] = yes
6               else  if M[i,j-1] = yes and s[i+j] = y'[j]
7                     M[i,j] = yes
8               else
9                     M[i,j] = no
10  return yes if and only if there is some pair (i,j) with i + j = n such that M[i,j] = yes
```

There are $O(n^2)$ values $M[i,j]$ to build up, and each takes constant time to fill in from the results of previous subproblems. So the total running time is $O(n^2)$.