Recitation Problems - Com S 311

Week of Apr 22^{nd} - Apr 27^{th}

1. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are low-stress (e.g., setting up a Web site for a class at the local elementary school) and those that are high-stress (e.g., protecting nation's most valuable secrets).

   If you select a low-stress job for your team in week i, then you get a revenue of $l_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

   Your objective is to schedule a sequence of jobs such that the in n weeks, the revenue earned is maximal.

   For example,

	Week 1	**Week 2**	**Week 3**	**Week 4**
l	10	1	10	10
h	5	50	5	1

   Best schedule will yield the revenue: 70 (no jobs in week 1 and high stress in week 2 and low stress jobs in weeks 3 and 4).

   Use a Dynamic Programming strategy. First, give a recursive formulation for the maximum revenue. Then, write pseudocode for an iterative DP algorithm using your recursive formulation and a dictionary to store the values of the maximum revenue for each week. Finally, give a runtime analysis of your algorithm.

   As we have done with all problems that applies DP strategy, first we will find the maximum revenue possible. Let the recursive function that captures the solution for weeks 1 to n be `maxRev(n)`

$$\texttt{maxRev(n)} = \begin{cases} 0 & \text{if } n == 0 \\ l_n & \text{if } n == 1 \\ \texttt{max}\{h_n + \texttt{maxRev}(n-2),\ l_n + \texttt{maxRev}(n-1)\} & \text{otherwise} \end{cases}$$

   The iterative algorithm for implementing the above recursion (assume that the h_i and l_i are given as an array):

```
// dict[i]: holds maxRev(i)
for i=0 to n
    if i==0 then dict[i] = 0
    if i==1 then dict[i] = l[i]
    else
        dict[i] = max{ h[i] + dict[i-2], l[i] + dict[i-1]}
```

`dict[n]` contains the solution.

Finally, find the number of schedules that realizes the maximum revenue. Let us write the iterative version.

```
\\ dict[i]: already computed
\\ numSched[i]: number of schedules that realizes the maximum revenue
\\              for weeks 1 to i
for i=0 to n
    if i == 0 or i==1
        numSched[i] = 1
    else
        if h[i] + dict[i-2] == l[i] + dict[i-1] // both choices are optimal
            numSched[i] = numSched[i-2] + numSched[i-1]
        else
            if h[i] + dict[i-2] > l[i] + dict[i-1] // h[i] is optimal for week i
                numSched[i] = numSched[i-2]
            else
                numSched[i] = numSched[i-1] // l[i] is optimal for week i
```

The solution is at `numSched(n)`.

Both constructing the dictionary and finding the number of schedules take $O(n)$ time, where n is the number of weeks. As a result, the runtime for this algorithm is $O(n)$.

2. Write pseudocode for an algorithm to find the number of shortest paths from a given source vertex s to a given destination vertex t in a graph where the weights of the edges can be positive or negative integers. You can assume that the graph does not contain any negative weight cycles.

Consider the Bellman-Ford algorithm. In that algorithm, add the parent relations such that all parents for each vertex are identified. These parents are the ones via which shortest path to the vertex can be computed.

```
for i=0 to n-1: dict[i][s] = 0;
for all v in V: if v!=s then dict[0][v] = infty;

for i=1 to n-1:
    for all v in V:
        dict[i][v] = dict[i-1][v]
        for all (u, v) in E:
            if dict[i][v] > dict[i-1][u] + wt(u, v):
                dict[i][v] = dict[i-1][u] + wt(u, v)
                parent[v] = {u}
            else:
                if dict[i][v] == dict[i-1][u] + wt(u, v):
                    parent[v] = parent[v] union {u}
```

The parent relations induce a directed acyclic graph (DAG) including the source vertex s. This happens when there is only positively weighted cycles. Using the parent relations, construct the DAG, $G' = (V, E')$ as follows:

```
Initialize a new graph G' with the same V vertices
for all v in V:
    add a directed edge (u, v) for each u in parent[v]
```

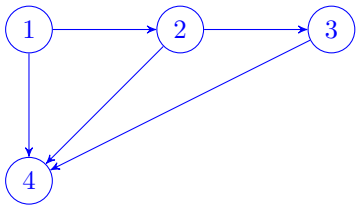
Then, to find the number of shortest paths from s to t, we then construct a topological ordering of the vertices in G'. If t appears before s in the ordering, then there is no path from s to t, and we just return 0. Otherwise, we loop over the vertices starting from t in reverse topological ordering as follows:

- The number of paths from x to t
  (a) is 1 if x is t
  (b) is the sum of number of paths to t already set for all neighbors of x

Then, we can just return the value associated with s.

```
order = topologicalSort(G')
if t appears before s in order:
    return 0
rOrder = reverse of order
for x=t in rOrder to each vertices after t in rOrder:
    if x is t:
        numOfPaths[x] = 1
    else:
        numOfPaths[x] = sum(numOfPaths[v] for each (x, v) in E')
return numOfPaths[s]
```

Use the following example with $s = 1$ and $t = 4$ to get some idea of why this should work.



Consider the reverse topological order: $4, 3, 2, 1$.

- Number of paths to 4 from 4: 1
- Number of paths to 4 from 3: sum of number of paths to 4 from 3's neighbors; this is equal to 1.
- Number of paths to 4 from 2: sum of the number of paths to 4 from 2's neighbors; this is $1 + 1 = 2$.
- Number of paths to 4 from 1: $1 + 2 = 3$

Justification: number of paths from s to t is the sum of the number of paths from s' to t where s' is a neighbor of s. Any neighbor of a vertex will appear after the vertex in topological ordering.

Runtime: The runtime is dominated by the construction of the parent relationship using the modified Bellman-Ford algorithm, which is $O(VE)$.