**CS5800 Algorithms**

# Recitations from Nov 4, 2019 to Nov 8 2019

**Ravi Sundaram**

## Problem 1 (Network Flow)

1. Suppose you are given some graph $G = (V, E)$ with integer capacities and a max flow algorithm has been run on it previously (so you know the max flow and how much flow is going through each edge). Now suppose that the capacity of some edge $e \in E$ has been decreased by 1. Describe an algorithm that recomputes the max flow in this modified graph in linear time.

2. A hospital is trying to schedule their doctors for night shifts over the next $n$ nights. There are $d$ doctors. Each doctor $i$ can only work certain nights $P_i$ due to personal constraints. For night $k$, the hospital needs exactly $s_k$ doctors to do the night shift. To be fair, the hospital does not want any doctor to do more than $m$ night shifts.

   Give a polynomial-time algorithm to determine whether the hospital can schedule their doctors so that all the constraints are met. If the hospital can, your algorithm should also output for each doctor the list of their night shifts.

### Solution:

1. Let us denote the amount flow going through edge $e = (a, b)$ in the max flow by $f(e)$ and let us denote the capacity of edge $e$ by $c(e)$. Let us denote the max flow of the original graph by $f$ and the max flow of the new graph we get from decrementing the capacity of $e$ by $f'$.

   There are two cases. One case is when $f(e) \leq c(e) - 1$. Then decrementing the capacity of edge $e$ by 1 still allows the original amount of flow to go through, so the max flow is unchanged.

   Now the second case is when $f(e) = c(e)$. In this case, decrementing the capacity of edge $e$ results in a capacity that's less than the original amount of flow going through it, so the max flow might potentially change. There are two options. One option is when there's another way for one unit of flow to get from $s$ to $t$ that doesn't use edge $e$. The other option is that was the only way.

   In order to test this in linear time we can do the following. In the original graph there's a path in the residual graph going from $s \to a$, then through edge $e = (a, b)$, and then from $b \to t$ carrying one unit of flow. Since we decreased the capacity of $e$ by 1, we need to 'return' one unit of flow starting from $t$ to $s$ (through the edge $e$). To do this, we can simply run a BFS in the residual flow graph from $t$ to $b$, finding an arbitrary path and sending one unit of flow back. And then we do the same from $a$ to $s$, sending one unit of flow back.

   Now, you should think of this residual graph as almost the exact same residual graph you'd get before you run the last iteration of Ford-Fulkerson. We know that when the residual graph is at this stage, that there is a path that includes $e$ through which we are able to push one more unit of flow from $s$ to $t$. The question now is are there any other paths in this residual graph from $s$ to $t$ that don't include $e$, since we decremented the capacity of $e$ and now $e$ is filled to capacity.

To answer this, we can simply run one iteration of Edmonds-Karp in the modified graph (with the decremented capacity) from $s$ and check to see if there's a path. Note that if there is, it won't use the edge $e$ (since $e$ is already filled to capacity). If there is, then the max flow $f' = f$. Otherwise, $f' = f - 1$.

2. We create a graph that we then run max flow on. First we create the vertices $s, t$. For each doctor $i$, we create the vertex $d_i$ and the edge $(s, d_i)$ with capacity $m$. For each night $j$ we create the vertex $n_j$ and the edge $(d_i, n_j)$ with capacity 1 if $j \in P_i$. Finally, we create the edge $(n_j, t)$ with capacity $s_j$.

   Then we run max flow on this and if its equal to $\sum_{j=1}^{n} s_j$ then its possible. Given that its possible, for each night we look at the incoming edges that have capacity 1 and those correspond to the doctors assigned to that night.

...

■

## Problem 2 (Dynamic Programming )

1. Suppose Alice and Bob are given an array $A[1...n]$ of integers. They are playing a game where they alternate turns, and at each turn a player chooses one of the two integers at the end of the array. After they choose that number, the number gets deleted from the array and its the next person's turn. Give a dynamic programming solution for Alice to maximize the sum of the integers she chooses, assuming that Bob plays optimally.

   For example, suppose we have the array $A = [4, 7, 2, 9]$. Alice can choose 9 in the first turn. Then we have the array $[4, 7, 2]$ and Bob can choose 2. Then we have the array $[4, 7]$. Alice can choose 4, and then Bob is left with 7. So Alice's sum at the end is 13 for this sequence of moves.

2. Suppose you are given an $n \times n$ matrix $A$ with integer entries and some parameter $c$. You would like to count the number of paths starting from the top left entry $A[1][1]$ and ending at the bottom right entry $A[n][n]$ with cost $c$. A cost of a path is the sum of the entries that the path goes through. You can only move down one entry or right one entry at each step of your path.

   For example, if we are given the matrix

$$\begin{bmatrix} 4 & 2 & 1 \\ 5 & 9 & 3 \\ 12 & 2 & 12 \end{bmatrix}$$

   One possible path is the sequence $4 \to 2 \to 9 \to 2 \to 12$ and this path has cost 29.

### Solution:

1. • English Description of problem: $OPT[i, j]$ will contain the maximal sum of the integers Alice can choose when starting the game with the array $A[i, i+1, ...., j]$, assuming that Bob is playing optimally.

- English Description of logic: When its Alice's move and she is given the array $A[i, i+1, ...., j]$ to work with, she can either choose the element $A[i]$ or the element $A[j]$. Let's say she chooses $A[i]$. Then its Bob's move and he has two choices. He can either choose $A[i+1]$ or $A[j]$. But remember he's playing optimally, so he's trying to minimize the value of the $OPT$ array. So on his turn he will look at the values of both $OPT[i+2, j]$ (the case if he chooses $A[i+1]$) and $OPT[i+1, j-1]$ (the case if he chooses $A[j]$) and choose whichever one does Alice the most harm.

  The expression on the other side is similar, and Alice on her move wants to choose the max among the 2 choices.
- Recurrence Relation (and Base Case)

$$OPT[l] = \begin{cases} A[i] & \text{if } i = j \\ \max(A[i], A[j]) & \text{if } i+1 = j \\ \max(A[i] + \min(OPT[i+2, j], OPT[i+1, j-1], & \\ \qquad A[j] + \min(OPT[i, j-2], OPT[i+1, j-1]) & \end{cases}$$

- Iterative Algorithm - The outer loop will be over $j$, starting from 1 and up to $n$. The inner loop will be over $i$, starting at $n$ and ending at 1.
- Number of subproblems - $n^2$
- Time per subproblem - Constant, just array lookup
- Total runtime - $O(n^2)$
- Final Return Value - $OPT[1, n]$

2.

- English Description of problem: $P[i, j, t]$ will contain the number of paths that start at the entry $(i, j)$ of the array and end at entry $(n, n)$ that have cost $t$.
- English Description of logic: When we are entry $(i, j)$, we have two choices. We can either go down or to the right. If we go down one step, then we are at index $(i+1, j)$. Now **suppose** there is some path of cost $t$ starting at index $(i, j)$ that goes through $(i+1, j)$. Then there **must be** a path of cost $t - A[i, j]$ starting at $(i+1, j)$. This is the key observation.

  Note this is similar to the knapsack problem we had on a previous recitation. There is an extra parameter in our memoization array that keeps track of intermediate values that the cost of a partial path can take, and only at the end do we plug in the given value for $c$.

  The base case is weird, but you should think of it as the following situation. Suppose we are at entry $(n, n)$ and $t = A[n, n]$. Then we get $P[n, n, t] = P[n, n+1, 0] + P[n+1, n, 0]$. We want $P[n, n, t]$ to be 1, so we just set the first case to be equal to 1. The second case will be equal to 0 by the next line of the base case, and all other instances when $i > n$ or $j > n$ will be 0.
- Recurrence Relation (and Base Case)

$$P[i, j, t] = \begin{cases} 1 & \text{if } i = n \wedge j = n+1 \wedge t = 0 \\ 0 & \text{if } i > n \vee j > n \\ P[i+1, j, t - A[i, j]] + P[i, j+1, t - A[i, j]] & \end{cases}$$

- Iterative Algorithm - The outer loop will be over $t$, starting from 0 to $c$. This is valid if we are working under the assumption that all the entries are positive. If there can be negative entries, then you need to be a little more careful (this is good to think about). The middle loop will be over $j$, starting from $n$ and down to 1. The inner loop will be over $i$, starting at $n$ and ending at 1.

- Number of subproblems - $cn^2$
- Time per subproblem - Constant, just array lookup
- Total runtime - $O(cn^2)$
- Final Return Value - $P[1, 1, c]$. Note that at the end we can plug in any value $d < c$ and get the number of paths starting at $(1, 1)$ that have cost $d$.

...

■

## Problem 3 (Extras...)

1. Suppose you are given a directed graph $G = (V, E)$ and some vertices $s, t \in V$. The capacity of each edge is 1. You are also given some parameter $k$. You would like to find a way to delete $k$ edges from $G$ so that the maximum s-t flow is reduced by as much as possible. Give a polynomial time algorithm and a proof of correctness.