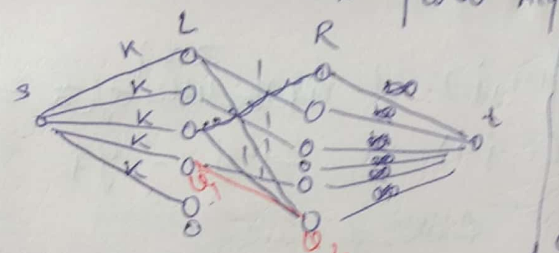
[1] Given an undirected graph G and a nonnegative integer k, we want to choose a direction for each edge of G so that in the resulting directed graph G', each node v is the tail of at most k arcs. Give an efficient algorithm for finding such an orientation of the edges and prove its correctness. (3+2)

Check total number of edges. Num. of edges should be atmost nk where n is the no. of vertices else there would atleast be a vertex with more than k outgoing edges.

We model this as a max flow Algorithm



In L Set we place all vertices and number them 1 to n. In R set we make a copy of L i.e 1 is infront of 1.

Connect all the left vertices by source with edge capacity K.

Join all R set vertices with sink with edge capacities ∞. Now if there is a ~~vertex~~ edge from v_i to v_j in undirected graph then draw a directed edge from v_i in L to v_j in R if $i \leq j$. of edge capacity 1.

Now, find the max flow.

If there is water in all the edges from v_i in L to v_j in R

Draw a directed edge from v_i to v_j in graph G.

This gives the orientation. Running time O(mnk) by Ford-Fulkerson.

1½

Correctness: ① Every Flow gives an orientation assignment satisfying the constraints.

Each vertex in L is supplied with atmost k units of water, so it can't send water to more than k edges. Now if we make directed edge from v_i in L to v_j in R, v_i can have atmost outdegree of K. There may be some vertices with 0 or less than k out degree but that's not a problem w.r.t constraint. Also if there are more than nk edges some edge won't have water so such orientation won't be possible.

② In directed graph all edges go from lower vertex to higher vertex in numbering. So there is always a flow possible, we just sent water from that v_i in L to v_j in R. Directed graph of desired orientation won't have any vertex with more than k outdegree which is also satisfied in flow construction.

Thus, when we find max flow, all edges get assigned a direction corresponding to desired directed graph and if an edge b/w L & R remains empty, such orientation ...

Scanned with CamScanner

[2] We are given three strings of characters X, Y, Z. Z is said to be the shuffle of X and Y if Z can be formed by interspersing the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string. For example, *cchocohilaptes* is a shuffle of *chocolate* and *chips* but *chocochilatspe* is not. Devise an algorithm that takes as input X, Y, Z and determines whether or not Z is a shuffle of X and Y. Analyze the worst-case running time and space requirement of your algorithm. (3.5+1+0.5)

~~Define M(i,j) = True if~~

Define X as $x_1 x_2 \cdots x_m$, $Y = y_1 y_2 \cdots y_n$ & $Z = z_1 z_2 \cdots z_{m+n}$

Now define $M(i,j)$ = True if $z_1 z_2 \cdots z_{i+j}$ is a shuffle of $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.

Now $$M(i,j) = \text{OR} \begin{cases} M(i-1,j) & \text{if } z_{i+j} = x_i \\ M(i,j-1) & \text{if } z_{i+j} = y_j \end{cases}$$

We defined OR as any one can be true and it will suffice ~~(OR of these~~

$M(i,j)$ is a matrix of size $m \times n$ (m = size of x, n = size of y)

Time: Filling each entry is $O(1)$ and to fill mn entries

Time $= O(mn)$

Space: We have a matrix of size mn where each entry is a boolean (1 byte), m sized string x, n sized string y and $(m+n)$ sized string Z.

Total space $= mn + 2(m+n)$

$\sim O(mn)$

5

We fill the matrix row-wise as we would need $M(i-1,j)$ & $M(i,j-1)$ at each step.

This algorithm says that if last character of subproblem Z ~~is~~ ~~the last~~ is last character of subproblem x or y, then we just need a small subproblem to be true that is $M(i-1,j)$ or $M(i,j-1)$ either should be true for correctness because $M(i-1,j)$ will denote shuffle of $x_1 x_2 \cdots x_{i-1}$ & $y_1 y_2 \cdots y_j$ is possible or not.

Similarly for $M(i,j-1)$.

Scanned with CamScanner

Tue 3-5

[3] Recall the maximum flow algorithm in which we routed flow along the shortest path in each step. A phase was defined as a sequence of steps in which the length of the shortest path remains the same. Show how to implement a phase in $O(n^2)$ time, where n is the number of vertices. (5)

Consider a phase i. In general phase i was implemented in O(mn) as the invariant was to remove atleast 1 edge in every iteration. We change the invariant to remove a vertex in every iteration. For that we define Bottleneck(v_i) = min (capacity of incoming edges, capacity of outgoing edges).

We define this for all vertices and this takes O(n) time as there are n vertices.

Now, we route min bottleneck flow in a specific way such that all that flow passes through minimum bottleneck vertex for its removal.

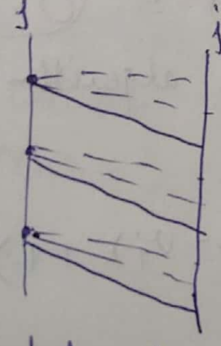
We follow the following procedure: Say Bottleneck flow is B.

- Start from bottleneck vertex
- Route the maximum flow out of B to its outgoing edges such that you choose the next edge only one previous is saturated in ascending order of capacities.
- Do the same in reverse direction that is from vertex to source.

½

Time reqd. to do so.

½



3½

Between any 2 layers, then dashed edges represent saturated edges by our flow and solid edges denote unsaturated edge.

½

Time to route flow is

O(num of unsaturated edges + num of saturated edges)

(num of unsaturated edges = no. of vertices in layer j as atmost 1 per vertex by construction)

½ $\Rightarrow O$(num. vertices in j + num of saturated edges)

Now taking this sigma over all layers over all iterations

½ $\sum_{\text{over all iteration}} \left(\sum_{\text{over layer}} \text{num of vertices in } j \right) + \sum_{\text{over all iteration}} \text{num of saturated edges}$

$\Rightarrow n \times (n) + m + \underbrace{(n \times n)}_{\text{Finding bottleneck}} = O(2n^2 + m)$

$\sim O(n^2)$

Note: Over all iterations num of saturated edges is $\leq m$. So overall sum of them is O(m). We got $2n^2$ as O(n) for finding bottleneck + O(n) for ~~routing flow~~ sum of vertices.

[4] You are given n types of coin denominations of values $v_1 < v_2 < \cdots < v_n$ (all integers). Assume $v_1 = 1$, so you can always make change for any amount of money C. Give an algorithm which makes change for an amount of money C with as few coins as possible. Analyse the running time of your algorithm (4+1)

We will solve the problem using dynamic programming.
Define opt(i) to be the optimum amount of coins required for making money i.

Base Case: opt(0) = 0

Then define $opt(i) = \min_{\substack{j \in [1,n] \\ v_j \le i}} (opt(i - v_j) + 1)$

↑ explain

i.e at ith step we check which denomination is best to be chosen and keep the store of array 'opt'.
We fill the array in ascending order. So, accessing opt(i-v_j) is ~~order~~ O(1).
Taking minimum is O(n).
So, calculating opt(i) = O(n)
We need to calculate n such values as Opt(n) is the final answer

~~so Calculating~~

Running time = O(n×n) = O(n²) ×

2

Proof:
~~Opt~~ Define base case Opt(1) = 1 for this algorithm. which is optimum
say opt(i) is optimum for i ≤ k
Now $opt(k) = \min_{\substack{j \in [1,n] \\ v_j < i}} (opt(k - v_j) + 1)$
We will get opt[k] as optimum as we consider all possible subcases and choose the minimum answer
Thus opt(C) gives the answer