The basic preflow-push algorithm

We showed in the last lecture that this algorithm always terminates. This means it always returns a max-flow.

- we proved the termination of this algorithm by bounding the number of while-loop iterations.

Every iteration of the while-loop performs either a push operation or a relabel operation.

- we saw that the total number of relabel operations is $O(n^2)$.
- we classified each push into one of 2 types:
  * saturating push: $O(mn)$ such pushes can happen in the algo.
  * non-saturating push: $O(mn^2)$ such pushes can happen in the algo.

Thus the number of while-loop iterations is $O(mn^2)$.

- Does this mean the running time of the algorithm is $O(mn^2)$?
  * no - because we need to check if there is an eligible edge (u,v) out of u

Since the number of relabels is $O(n^2)$, we can try to balance costs as follows:

- whenever a vertex gets relabelled, spend $O(n)$ time in traversing its list of out-neighbours and all out-neighbours whose level is $k-1$ (where k is the new level of our vertex) will be in a list of eligible out-neighbours.

However these out-neighbours may get relabelled in later iterations. So how do we deal with this?

Exercise. Show that the basic preflow-push algorithm can be implemented to run in $O(mn^2)$ time.

An improved preflow-push algorithm

Date

- Put all active vertices in a queue Q.
  (active vertices: those with positive excess)
- Consider vertices in Q one by one.
  * let u be the current vertex under consideration — persist with u till u is deactivated or u is relabelled.

deactivated means its excess is 0.

So the improved algorithm is the same as the previous algorithm except for the above changes. More formally, the pseudocode is given below.

1. Start with $f(e) = c(e)$ for all edges e outgoing from s
   $f(e) = 0$ for other edges
   Initialize Q = queue containing all out-neighbours of s
2. While $Q \neq \phi$ do:
   {
   - u = first vertex in Q; delete u from Q.
   - persist with u till either $excess(u) = 0$ or there is no eligible edge out of u
     (in this case u is relabelled and entered into Q again)
3. Return f.

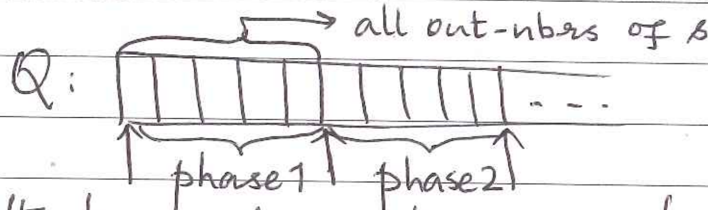
As before, the function f returned is a max-flow. Recall that we showed for the previous algo. that
- the number of relabels is $O(n^2)$.
- the number of saturating pushes is $O(mn)$.

Those bounds hold for the above algorithm as well. We want to show an improved bound for non-sat. pushes.

Bounding the number of non-saturating pushes

Let us divide the execution of the algorithm into phases.

- a phase ends when all the vertices that were active at the beginning of the phase have been selected from Q.



- note that each vertex is selected at most once from Q in a phase.

The number of non-sat. pushes
$\leq$ (number of phases) $\cdot n$ (why?)

How many phases are there?

To answer the above question, we will use the following potential function.

$$\phi = \max\{\ell(u) : u \text{ is active}\}.$$

Let ϕ_i denote the ~~function~~ value taken by ϕ at the end of the i-th phase.

$\phi_0 = 0$ and $\phi_i \geq 0$.

So $\phi_t = (\phi_t - \phi_{t-1}) + (\phi_{t-1} - \phi_{t-2}) + \cdots + (\phi_1 - \phi_0)$
$= \sum_{i=1}^{t} \Delta\phi_i$ where $\Delta\phi_i = \phi_i - \phi_{i-1}$

As before, call $\Delta\phi_i$ blue if it is non-negative and call it red otherwise.

So sum of blue terms + sum of red terms ≥ 0.

Thus sum of blue terms $\geq$ − sum of red terms

(sum of blue terms:) this is the total increase in ϕ in the first t iterations

(− sum of red terms:) this is the total decrease in ϕ in the first t iterations

Suppose phase i is a phase without relabel operation. This means every vertex selected in phase i got deactivated.

- this vertex may have got reactivated again in this phase.
- however the vertices in the highest level remains inactive at the end of such a phase.

So $\phi_i - \phi_{i-1} \leq -1$.

Suppose phase i is a phase with relabel operation. Then $\phi_i - \phi_{i-1} \leq 1$.

- this is because $\phi_i = \phi_{i-1} + 1$ if a vertex at the highest level gets relabelled; else $\phi_i \leq \phi_{i-1}$

Thus we have: number of phases without relabel in the first t itns. $\leq$ total decrease in ϕ in the first t iterations.

Also number of phases with relabel in the first t itns. $\geq$ total increase in ϕ in the first t iterations.

Since total decrease in ϕ in the first t itns. $\leq$ total increase in ϕ in the first t itns., we get number of phases without relabel in the first t itns. $\leq$ number of phases with relabel in the first t itns.

$\leq$ total number of relabel operations

$\leq \underbrace{2n^2}$.
this is independent of t.

Hence the total number of phases without relabel $\leq 2n^2$.
Every phase is either a phase with relabel or phase without relabel.
Thus the total number of phases $\leq 2n^2 + 2n^2 = 4n^2$.

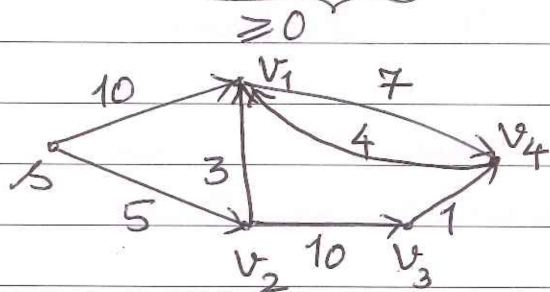
By our earlier observation that the number of non-saturating pushes $\leq$ (number of phases) $\cdot n$, we get the total number of non-sat. pushes is $O(n^3)$.

- As done for the previous algorithm, show that the improved algorithm can be implemented to run in $O(n^3)$ time.

We will move to a new topic now:

Single Source Shortest Paths

Input: a directed graph $G = (V, E)$ with weights or "distances" on its edges. (distances ≥ 0)



We have a special vertex s (the source) and our goal is to compute shortest paths from s to all vertices.

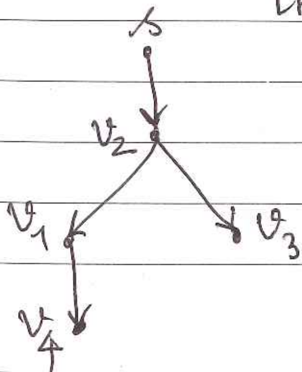
Ex: In the above example, a shortest path from s to v_4 is

$$s \to v_2 \to v_1 \to v_4$$

Is a subpath of a shortest path again a shortest path? That is, is $s \to v_2 \to v_1$ a shortest path from s to v_1? The answer is yes, please justify this.

How do we store shortest paths?

- what we will compute is a shortest path tree rooted at s.



We will use a function π:

$\pi(u)$ = parent of u in this tree

For each vertex u, our algorithm maintains a "distance estimate" discovered so far from s to u.

- right at the beginning, we have not examined any edge and so:

$$d[s] = 0, \quad d[u] = \infty \quad \forall\, u \in V - \{s\}$$
$$\pi(u) = nil \quad \forall\, u \in V.$$

At any point in time, there will be a set $S \subseteq V$ which is the set of those vertices whose final distance estimates have been determined.

- initially $S = \phi$.

Let $Q = V - S$

- so Q consists of vertices whose final distance estimates have not yet been found.

A high-level view of the algorithm:

while $Q \neq \phi$ do

- extract the vertex u with the least d-value from Q and move it from Q to S.
- relax all edges leaving u.

Relax (u, v)

- if $d[v] > d[u] + w(u,v)$ (→ this is the weight of edge (u,v)) then

$$d[v] = d[u] + w(u,v);$$
$$\pi(u) = v$$

This is a greedy algorithm since it selects the vertex with "least" d-value from Q and says this is its final distance estimate and moves it to S.

- Is the algorithm correct? We will prove this next.