Network flows (Cont.)

Let Δ be some threshold.

$G(\Delta)$ is the graph which doesn't have (deleted) edges with capacities less than Δ. Keep the rest. (Keep edges with capacities $\geq \Delta$).

→ Algo 2:

- Find s-t path in $G(\Delta)$
- Augment the flow by bottle neck
- Construct residual graph and ensure edges have cap. $\geq \Delta$.
- Repeat until there's no s-t path

Suppose we have L iterations. So the flow increments by atleast $L \cdot \Delta$.

If there's an edge with cap. 1.5Δ, then once Δ flow is augmented, then we have 0.5Δ. So in the next iter, we don't consider 0.5Δ in this iter. coz it's lesser than Δ.

In each phase; running time:

$$\log|C| \cdot \max\{\#\text{ of iterations in each } \tfrac{\Delta}{2^i} \text{ phase}\} \cdot O(m+n)$$

$< 2m$

$O(\log|C| \cdot m \cdot O(m+n))$

where $|C|$: $\min\left\{\sum_{s\to u \in E} c(s\to u),\ \sum_{w\to t \in E} c(w\to t)\right.$

Previous algo: $O(|C| \cdot (m+n))$

When $\underbrace{m \cdot \log|C|}_{m \cdot n} \leq \underbrace{|C|}_{2^n \cdot e}$, then algo 2 is better.

In prev. lec. example, algo. 2 is better.

Now if $|C|$ is 100, then algo 1. is better

(Get max. flow as fast as possible by picking up higher valued bottle-necks first).

Init:
→ How do we pick a Δ?

Initial Δ. $\Delta \leftarrow$ Largest power of 2 s.t it's smaller than $|C|$.

$\Rightarrow \quad \Delta : \max \{ 2^k \mid 2^k < |C| \}$.

Also gives guarantee than Δ is strictly less than $|C|$.

Bounds on Δ are : $\frac{|C|}{2} \leq \Delta < |C|$

- Construct the graph $G(\Delta)$. New graph

Algo.

1. Find a $s \rightsquigarrow t$ path in $G(\Delta)$ if it exists.

NO | YES

- Update the flow.

$$F \leftarrow F + \text{bottleneck}. \quad (F_{\text{updated}} \geq F + \Delta).$$

This is the only difference from prev. algo

- Compute residual graph by removing edges of cap. $< \Delta$.

Only edges on the path would be changed. so we only need to check for those edges (if less than or greater than Δ).

$O(m+n)$ book keeping to maintain flow & capacity w.r.t original graph.

- Go to step 1.

- Update $\Delta \leftarrow \frac{\Delta}{2}$
- Update the graph (residual) by including edges of cap. $\geq \Delta$.

If $\Delta = 1$ and there are no s-t paths, then return F.

→ Need to bound the no. of iterations in each Δ phase.

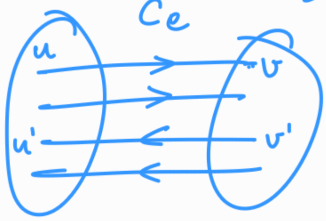
Lemma : No. of augmentations in each $\frac{\Delta}{2}$-phase $\leq 2m$.

Claim : If F be the flow at the end of Δ-phase, then the cap. of the cut obtained at the end of Δ-phase in $G_F(\Delta)$ is atmost $F + m \cdot \Delta$.

$u \to v \in E(G_{ori.})$

$v' \to u' \in E(G_{ori.})$

C_e → Going forward.



s-T cut.

1. $c_e < f_e + \Delta$

If not, then $c_e \geq f_e + \Delta$ $\Rightarrow$ There is a residual cap. of $\geq \Delta$

$\Downarrow$

The algo. would not have terminated as this would lead to s-t path

(or)

v would be part of S itself.

$\Downarrow$

Contradiction that $v \in T$.

($\because$ v is reachable from s in $G_F(\Delta)$ res.)

2. Back edge.

$f_e < \Delta$

$$\text{Flow} = \sum_{\substack{\text{fwd. edges} \\ S \to T}} f_e - \sum_{\substack{e': \\ \text{Back edges.}}} f_{e'}$$

$$> \sum_{\substack{e \\ \text{fwd.}}} (c_e - \Delta) - \sum_{\substack{e': \\ \text{back}}} \Delta$$

$$> \underbrace{\sum_{\substack{e \\ \text{fwd.}}} c_e}_{\text{Cap. of cut.}} - \underbrace{\sum_{e, e'} \Delta}_{\leq m \cdot \Delta}$$

$$\Rightarrow \underbrace{\sum_{e} c_e}_{\text{Capacity of cut.}} < F + m\Delta$$

Hence, claim proved.

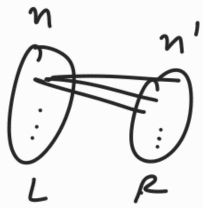
Let F' be the augmented flow at the end of $\frac{\Delta}{2}$ phase.

F be the flow at the end of Δ-phase.

$$\underbrace{F + m\cdot\Delta}_{\text{Any feasible flow is atmost capacity of any cut.}} > F' \geq F + L\cdot\frac{\Delta}{2} \implies L < 2m.$$

No odd cycles.

- **Bipartite Matching.** (Edmund augmented).



Matching : Subset of edges s.t $v \in$ only one edge in the set M.

Perfect matching if every vertex has an incident edge in M.

To find: Maximal matching for a given bipartite graph. | Min. vertex cover

If we add any more edges, it'll no longer be a matching.

(Can apply network flow concepts).

→ Add s, t and add edges from s to every vertex in L and from every vertex in R to t.

→ Assign cap. of 1 to each edge

Obs. : Max. flow gives maximal matching.

→ A maximal matching can be a perfect matching when # of vertices in R = # of vertices in L.

→ Matching → general graph → Polynomial time