Oct 11 — CSE 14

Max Flow Problem

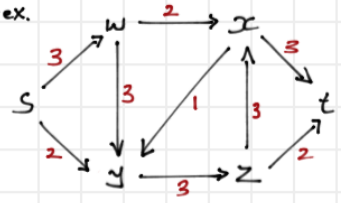
INPUT: A weighted directed graph G
Two vertices — s (source) and t (target)

OUTPUT: Edge assignment of max flow

a flow is a $f: E \to R$, such that

1. $0 \leq f(edge) \leq edge.weight$
2. $f_{in}(v) = f_{out}(v) \quad \forall v \neq u$ such that
$f_{in}(v) = \sum_{uv \in E} f(uv) \qquad f_{out}(v) = \sum_{vu \in E} f(uv)$

-ex.



GOAL: Move from s to t
Edges are max capacity
⇒ Move max possible flow

METHODS:

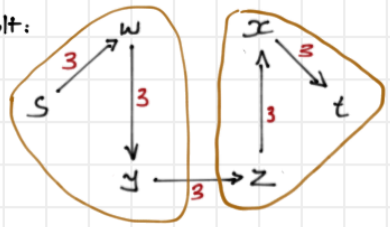
Brute force — inefficient, find all possible configurations for a valid flow → $\prod_{e \in E} e.weight$

Greedy — no way of selecting any part of flow greedily

Dynamic Programming — not possible to break problem into independent recursive problems

-ex. Maxflow(G, s, t): if we push flow f_1 from w to t, need to solve Maxflow(G − f_1, y, t)

result:



$V_s \to V_t$ flow = 3 (not optimal)

IDEA: local search algorithm — start with initial assignment of flow guaranteed to be correct but not necessarily maximum, then try to make incremental improvements — stop when no improvements possible

Intuition: Max flow from s to t is also the min-cut from s to t (same problem)

Oct 16 CSE 15

DEFINITIONS

network — a directed graph $G = (V, E)$ with
- source vertex $s \in V$ with no incoming edges
- target vertex $t \in V$ with no outgoing edges
- capacity $c(e)$ for each edge $e \in E$

network flow problem: assign a flow $f(e)$ to each edge $e \in E$ such that we have the maximum flow in the network, subject to these constraints:

1. capacity — for each edge $e \in E$, $0 \le f(e) \le c(e)$
   ie. no negative flow, can't exceed capacity
2. conservation — for every $v \in V$, $v \neq s, t$: $f_{in}(v) = f_{out}(v)$

where $f_{in}(v) = \sum_{uv \in E} f(uv)$

$f_{out}(v) = \sum_{vu \in E} f(vu)$

total flow in the network $|f| = f_{out}(s) = f_{in}(t)$
$= f_{in}(v) = f_{out}(v)$ $v \in V, v \neq s, t$

cut — a partition of V into V_s, V_t such that
1. $V_s \cap V_t \neq \emptyset$ and $V_s \cup V_t = V$
2. $s \in V_s$ and $t \in V_t$

forward edge — an edge (u, v) with $u \in V_s$ and $v \in V_t$

backward edge — an edge (u, v) with $v \in V_s$ and $u \in V_t$

for any cut $X = (V_s, V_t)$

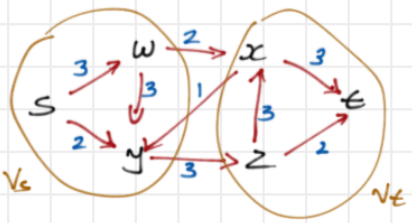
$cap(X)$ — the capacity of cut X = sum of forward edges

$$cap(X) = \sum_{v \in V_s, u \in V_t} c(vu)$$

$f(X)$ — the flow across cut X = total flow forward minus the total flow backwards across X

$$f(X) = \sum_{v \in V_s, u \in V_t} f(vu) - \sum_{v \in V_s, u \in V_t} f(uv)$$

ex. $X = (V_s, V_t)$



$cap(X) = 5$

$f(X) = 4$

for any cut $X = (V_s, V_t)$ and any flow f

lemma 1. — $f(X) \le cap(X)$

proof: $f(X) = \sum_{v \in V_s, u \in V_t} f(vu) - \sum_{v \in V_s, u \in V_t} f(uv)$

$\le \sum_{v \in V_s, u \in V_t} f(vu) \le \sum_{v \in V_s, u \in V_t} c(vu) = cap(X)$

lemma 2. — $f(X) = |f|$

proof: $|f| = f_{out}(S) = \sum_{sv \in E} f(sv)$

$= \sum_{sv \in E} f(sv) + \sum_{v \in V_s - s} \overbrace{f_{out}(v) - f_{in}(v)}^{=0}$

$= \sum_{sv \in E} f(sv) + \sum_{v \in V_s - s} \left(\sum_{vu \in E} f(vu) - \sum_{uv \in E} f(uv) \right)$

notice that for any edge $e = (v, u)$
- if $v, u \in V_t$ then neither $f(uv)$ nor $f(vu)$ influence the equation
- if $v, u \in V_s$ then $\sum f(uv)$ negates $\sum f(vu)$
- if $v \in V_s$ and $u \in V_t$ then $f(vu)$ has a positive influence on the equation
- if $v \in V_t$ and $u \in V_s$ then $f(vu)$ has a negative influence on the equation

$= \sum_{\substack{v \in V_s, \\ u \in V_t}} f(vu) - \sum_{\substack{v \in V_s, \\ u \in V_t}} f(uv) = f(X)$, $X = cut$

$\Rightarrow$ for any cut $X = (V_s, V_t)$ and any flow f $|f| \le cap(X)$

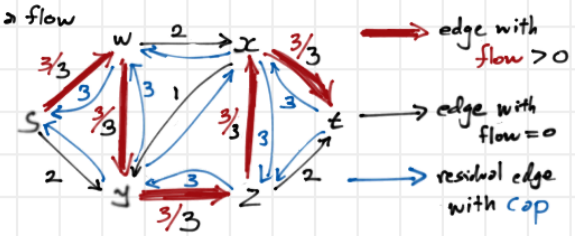
$\Rightarrow |f| \le \min_X \{cap(X)\}$

$\Rightarrow \max_f \{|f|\} \le \min_X \{cap(X)\}$

reverse directions: graph must take into account that we may want to flow in reverse direction of the current flow on an edge

for any edge e a **reverse edge** e^r exists in the reverse direction such that $cap(e^r) = f(e)$

ex. graph with a flow



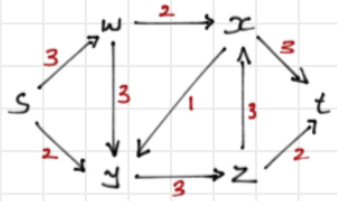
CSC 148 Lec 16

INPUT: a network graph $G=(V,E)$

OUTPUT: maxflow edge assignments

Want ——— 1. maximize $|f| = f_{out}(s)$ ⟺

2. find the minimize $cap(X)$, X = a cut



SOLUTIONS

Algorithm overview ——

start with any valid flow f (can be $\forall e \in E, f(e)=0$)

while there is an augmenting path P: do

augment f using P

return f

IDEA 1 ——— Since all flow must start at s and end at t find s-t paths along which flow can be increased

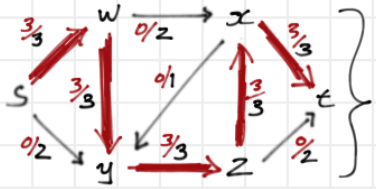
ALGO: path $P = s \to \cdots \to t$ where $f(e) < c(e)$ for each e.

Let residual capacity $\Delta_f(e) = c(e) - f(e)$ and

$\Delta_f(P) = \min\{e \in P \mid \Delta_f(e)\}$

Then augment path P by adding $\Delta_f(P)$ to each $e \in P$

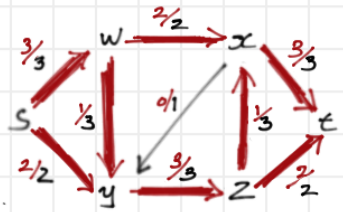
PROBLEM: notation too narrow, might get stuck with sub optimal flow; ex.



$P_1 = s \to w \to y \to x \to z \to t$

$P_2 = s \to w \to x \to t$ $\quad |f| = 3 = f(P_1)$

optimal sol. ↳



$|f| = 5$

explanation:

P_1 was the first path found in G with flow > 0

by maximizing P_1, flow through other paths are restricted $\Rightarrow$ P_2 cannot get any flow

IDEA 2 ——— define reverse edges such that:

original edge has unused capacity that can be used to push more flow from $s \to t$

reverse edge has surplus capacity that can be redirected to push more flow from $s \to t$

ALGO: Let residual capacity $\Delta_f(e) = c(e) - f(e)$ for original edges

$= f(e)$ for reverse edges

Let augmenting path π be a path $s \to t$ where each edge $e \in \pi$ has a positive residual capacity:

$c(e) - f(e) > 0$, for original edges

$f(e) > 0$, for reverse edges

Then add $\Delta_f(\pi)$ to original edges, and reverse edges

Resultant f for each edge is the net flow between its original edge and reverse edge

FORD-FULKERSON:

Let $G_f = (V_f, E_f)$ the graph G with a flow f where

$V_f = V$

$E_f = \{e \in E \mid f(e) < c(e)\} \cup \{e^r \mid f(e) > 0\}$

$$\Delta_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e \notin E \end{cases}$$

ALGORITH: $G_f \leftarrow G = (V, E$ where $\forall e \in E, f(e)=0)$

while there's a path $\pi \in G_f$ from s to t

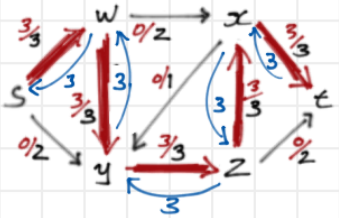
add π to G_f

- ex. have a G_f, see if it can be increased

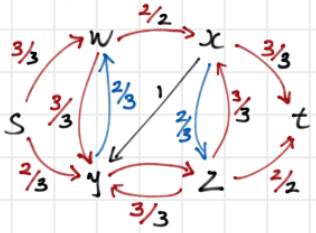
1. found an augmenting path

$\pi = s \to y \to w \to x \to z \to t$

$\Delta_f(\pi) = 2$



2. add $\Delta_f(\pi)$ to all $e \in \pi$



3. find net flow between normal & reverse edges:

Proof of Optimality

note: everytime an augmenting path is added to G_f the amount of flow is increased by at least 1

let max possible flow be f_{max}, then after at most f_{max} iterations G_f will reach f_{max} $\Rightarrow$ no more augmenting paths exists in G_f

Show that for any network G and flow f, $|f|$ is max $\Rightarrow$ there's no augmenting path (Ford Fulkerson thm)

IDEA: the algorithm will improve flow until min cut is reached when that cut is reach eventually it'll be completely utilized $\Rightarrow$ no augmenting path $s \to t$ then can exist

Show that there's no augmenting path $\Rightarrow$ $|f|$ is max

Construct a cut X such that

- Let V_s be all nodes reachable from s in G_f
- Let $V_t = V - V_s$ (all nodes not reachable from s in G_f)

$t \in V_t$, since there's no augmenting path

By definition of X, every edge crossing X has property that $f(e) = c(e)$ for forward edges and $f(e) = 0$ for backward edges, else an augmenting path exists

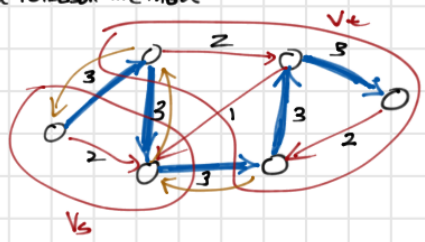
Hence $|f| = f(X) = c(X)$

Then $|f| \iff \nexists \pi \in G_f$

Oct 21 LEC 18

max-flow – min cut

Ford Fulkerson method



flow(V_s, V_t) = 3

Cap(V_s, V_t) = 6

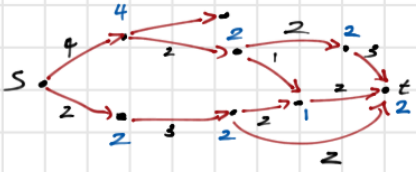
Karp - Edmonds

BFS — $O(|E||V|)$

※ iterations

$O(|E|^2|V|+|E||V|^2) \in O(|V|^5)$ since $|E| \leq |V|^2$

use BFS but use more than 1 path



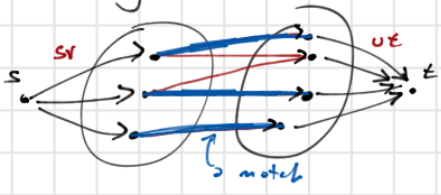
push-relabel idea $O(|V|)$

push as much as possible over edges, then fix it to become a flow

bipartite matching —

bipartite graph G is a graph where there is a partition of its vertices such that all edges have one endpoint in each

matching: a list of edges such that no two edges share endpoints



input: a bipartite graph G

output: a maximum matching

algorithm: $V_H = V_G \cup \{s,t\}$ $s,t \notin V_G$

$O(|V|+|E|)$ $E_H = \{(u,v) \mid \begin{matrix} u \in V_1 \\ v \in V_2 \end{matrix}\} \cup \{(s,v) \mid v \in V_1\} \cup \{(u,t) \mid u \in V_2\}$

$O(|V|^3)$ $f = \text{max-flow}(H,s,t)$

$O(|E|)$ $M = \{e \in E_G \mid f(e) \neq 0\}$

return M

running time: $O(|V|^3)$

correctness: A finds the maximum match

M is a max matching in G ⟵ f is a max flow in H

1. Construct H from G
2. M is constructed from f, G

reduction: let P and Q be two computational problems

$P \leq Q$ P reduces Q

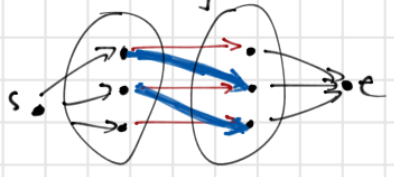
if there is an algorithm R such that using Q solves P

1. if f is a flow in H, then M is a matching in G
2. if M is a matching, then there is some f that gives it

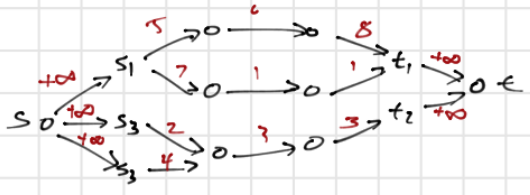
$|f| = |M|$ ⟵ 1 to 1 correspondence between flow and matching

find max in one will find the max in the other

- ex. none max matching



Suppose multiple starting node / ending nodes



- agnet (reduce) by adding s, t so that the problem can be solved.

Run time of Ford-Fulkerson: In worst-case, Ford-Fulkerson takes time $\Theta(mC)$, where C is sum of the capacity of all edges leaving the source s – this is not polytime.

Choosing augmenting paths efficiently:

- Edmonds-Karp algorithm: use BFS (modified to consider only augmenting edges) to find augmenting paths; guaranteed to find an augmenting path with smallest number of edges in time $O(V)$. Possible to prove that no more than $O(VE)$ augmentations are required to find max. flow. Total time: $O(VE^2) = O(V^5)$.
- Dinitz's algorithm: perform complete BFS, use all augmenting paths w.r.t. BFS tree, then repeat. Worst-case time down to $O(V^2E) = O(V^4)$.
- "Preflow-Push" algorithm: don't use augmenting paths; instead, push as much as possible along individual edges then go back to fix conservation. More complicated to explain and write down correctly, but cuts down time to $O(V^3)$.
- Why are we not concerned about details? Applications of network flow do not involve writing new algorithms: you always solve the same problem, so you can use an existing implementation that's already been debugged and optimized. Applications involve taking a problem, casting it in the guise of a network flow, solving the network flow problem, then casting the answer back to your problem.

Applications of network flows:

Multi-source, multi-sink network: Add "super-source" with edges of capacity ∞ to each source and "super-sink" with edges of capacity ∞ from each sink (instead of using ∞, we can set capacity to sum of outgoing/incoming capacities).
Max flow in resulting network = max flow in original network because:

- any flow in original network can be extended to a flow in resulting network (for new edges from super-source to source, set flow equal to total flow out of source; for new edges from sink to super-sink, set flow equal to total flow into sink) – hence, max flow in new network $\geqslant$ max flow in original network;
- any flow in resulting network induces flow in original network (flow out of every source and into every sink limited only by edges in original network because of "infinite" capacities on new edges) – hence, max flow in original network $\geqslant$ max flow in new network.

Maximum bipartite matching: We are given an undirected bipartite graph $G = (V_1, V_2, E)$ – one where every edge is between V_1 and V_2 (i.e., no edge has both endpoints in the same "side").
The goal is to identify a disjoint subset of edges of maximum size (i.e., no edge in a matching shares an endpoint with any other edge in the matching).

Given input graph, create network by turning every original edge into a directed edge (from V_1 to V_2) with capacity 1; add source with edges of capacity 1 to each vertex in V_1, sink with edges of capacity 1 from each vertex in V_2.

- Any matching in graph yields flow in network: set flow = 1 for graph edges in matching, 0 for graph edges not in matching; set flow equals 1 for new edges to/from matched vertices, 0 for new edges to/from unmatched vertices.
- Any integer flow in network yields matching in graph: pick edges with flow = 1 (leave out edges with flow = 0).

For this correspondence, size of matching = value of flow. Hence, any max flow in network yields a maximum matching in graph (because a larger matching would give a larger flow).

Difference between Network Flow and other techniques:
When solving a problem by "using network flows", what we are doing is actually a *reduction* or *transformation*: we take the input to our problem and create a network from it, then use standard algorithms to solve the maximum flow problem on that network (it's always the same problem and always the same algorithm, only the input differs). Then, we use the solution to the network flow problem to reconstruct a solution to our problem.

There are two ways that this could go wrong:

- The network we construct could fail to represent certain solutions to our original problem. We show this is *not* the case by arguing that every solution to the original problem yields a valid flow (or cut) in the network.
- The network we construct could have solutions that don't correspond to anything in our original problem. We show this is *not* the case by arguing that every flow (or, depending on the problem, every *integer* flow or every cut) in the network corresponds to a solution to the original problem.

Sometimes those arguments can be very short, because the correspondence is obvious between both problems. But both arguments are important and must always be included.

One last example:

Project selection: There exist a set of projects P each with revenue p_i (integer, can be positive or negative), and prerequisites between projects (given as directed graph on vertices P with edges (i, j) meaning project i depends on j). The goal is to find a *feasible* subset A of P with maximum total revenue (set A is feasible means that for each $i \in A$, A contains all i's prerequisites).

Unlike other examples, this can be solved by network flow techniques where we are interested in minimum cuts (flow values are irrelevant).

Construct network N from G as follows:

- Add source s, sink t,
- Add edges (s, i) with capacity p_i for each i such that $p_i \geqslant 0$,
- Add edges (i, t) with capacity $-p_i$ for each i such that $p_i < 0$,
- Set capacity of all other edges (i, j) to $C + 1$, where $C = \sum_{p_i >= 0} p_i$.

Find a minimum cut (V_s, V_t) in N. Then $V_s - \{s\}$ is a feasible subset of projects with maximum profit! This requires proof...

Claim 1: for any cut (V_s, V_t), the set $V_s - \{s\}$ is feasible iff the capacity of the cut is at most C.
Proof: $c(V_s, V_t) \leqslant C$ implies no edge (i, j) (with capacity $C+1$) crosses the cut forward, i.e., for all projects $i \in V_s$, V_s contains all of i's prerequisites.

Claim 2: $c(A \cup \{s\}, \bar{A} \cup \{t\}) = C - \sum_{i \in A} p_i = C - profit(A)$ for all feasible sets of projects A (where $\bar{A} = P - A$).
Proof: Since A is feasible, no edge (i, j) with capacity $C + 1$ crosses the cut forward (some may cross backward). Forward edges crossing the cut fall into two groups:

- entering sink contribute $\sum_{i \in A, p_i < 0} -p_i$
- leaving source contribute $\sum_{i \in \bar{A}, p_i \geqslant 0} p_i = C - \sum_{i \in A, p_i \geqslant 0} p_i$ (because $C = \sum_{p_i \geqslant 0} p_i = \sum_{i \in A, p_i \geqslant 0} p_i + \sum_{i \in \bar{A}, p_i \geqslant 0} p_i$)

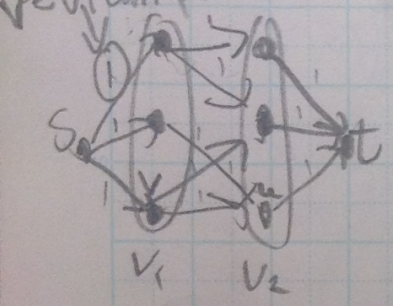
So total capacity of cut is $\sum_{i \in A, p_i < 0} -p_i + C - \sum_{i \in A, p_i \geqslant 0} p_i = C - \sum_{i \in A} p_i = C - profit(A)$.

These two facts imply that feasible sets of projects and cuts with capacity at most C correspond to each other. Since for any such set, $c(A \cup \{s\}, \bar{A} \cup \{t\}) = C - profit(A)$, and C is constant, any minimum-capacity cut (V_s, V_t) yields a maximum-profit set $V_s - \{s\}$.

* BFS search for max or min.

like → Theorem | f | < min cut → there is an augmenting path in G_f
LI.

Max size matching in a bipartite graph. Oct. 23, 2013



$V_H = V_G \cup \{s, t\}$

$E_H = \{sv \mid v \in V_1\} \cup \{vu \mid v \in V_1, u \in V_2, vu \in E_G\} \cup \{ut \mid u \in V_2\}$

Let f be a max flow in H from s to t.

$M = \{vu \mid f(vu) \neq 0, vu \in E_G\}$

Running Time: { construct H from G $O(n^2)$
Find a max flow for H from s to t $O(n^3)$
construct M from f, G. $O(n^2)$
$O(n^3)$

M = max matching G $\Leftrightarrow$ max flow in $H^G = f$.

Correctness:
1) M^f is a matching in G
2) M^f is optimal.

Pf
1) For every flow f in H, M^f is a matching
2) For every matching M in G, there is a flow f in H.
$M^f = M$.

Pf of 1): We have to show that no two edges in M share endpoints

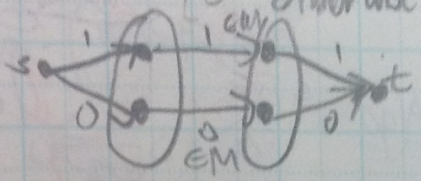
We show that no vertex appears in more than one edge in M.
If $v \in V_1$, the flow to v is at most 1: $f_{in}(v) \leq 1$
$\Rightarrow$ $f_{out}(v) = f_{in}(v) \leq 1$.
$\Rightarrow$ only use only one edge going out from v.
If $u \in V_2$, the argument is similar [illegible]

Pf of 2) L

* |f| = |M
* {|M^f| ≥

Pf of 2) Let M be a matching in G
Define flow f in H s.t. $M^f = M$.

$$f(vw) = \begin{cases} 1 & vw \in M, v \in V_1 \\ 0 & vw \notin M, v \in V_1 \\ 1 & uv \text{ is in } M \text{ for some } u, v \in V_2 \\ 0 & \text{otherwise} \\ 1 & wv \text{ is in } M \text{ for some } w, u = t \\ 0 & \text{otherwise} \end{cases}$$



Need to show f is a flow, i.e.
1) $f(vw) \geq 0 \quad \forall vw \in E_H$ ✓
2) $f^{in}(v) = f^{out}(v) \quad \forall v \in V_H - \{s,t\}$

Pf: $\forall v \in V_1$, either:
1. v is in some edge vu in M:
   $f(vw) = 1$
   $f(sv) = 1$
   for any other w, $f(xw) = 0$ bec. not in M.
   So $f^{in}(v) = f^{out}(v) = 1 = f(vw) = f(sv)$ ✓
2. v is not in any edge in M:
   $f(sv) = 0$
   for any other w, $f(vw) = 0$.
   So $f^{in}(v) = 0 = f^{out}(v)$ ✓ □

$M^f = M$:

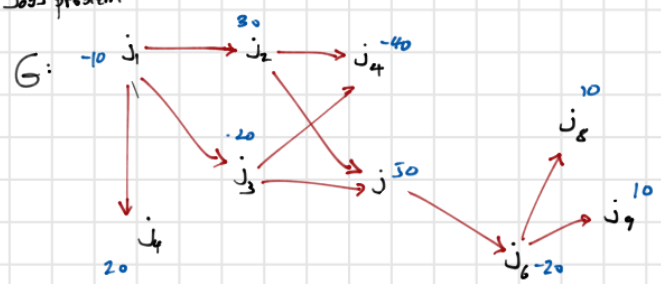
$M^f = \{vu \mid f(vu) \neq 0\} = \{vu \mid vu \in M\} = M$

☆ $|f| = |M^f|$

* $|M^f| \geq |M^{f'}| \iff |f| \geq |f'|$

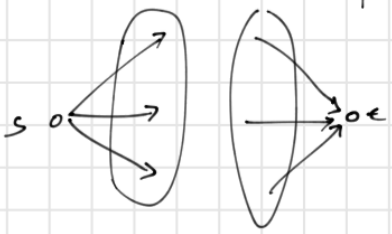
Oct 25 — Lec 20

Jobs problem



Input: a list of jobs with their prices & dependencies
output: select a list of jobs which maximizes profit

use network flow — reduce the problem to a min-cut problem

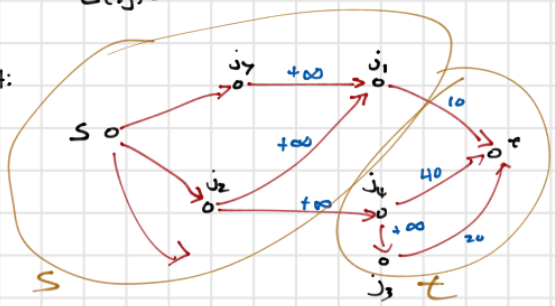


for jobs i where $P_i \geq 0$
put an edge s_i with
$C(s_i) = P_i$

for jobs i where $P_i < 0$
put an edge it with
$C(it) = -P_i$

if job i depends on job j,
add an edge ij with
$C(ij) = +\infty$

H:



min cut in H $\longleftrightarrow$ max profit selection in G
(S, T) is a cut in H $\longleftrightarrow$ a selection of jobs A in G
$A = S - \{s\}$

max profit in G

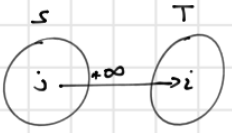
Problem 1: Input 1 → Output 2 $A = S - \{s\}$

problem 2: min cut — Input 2: H → output 2: (S, T)

we have to show A is a valid selection of jobs
— i.e if $j \in A$ and $ij \in G$ then $i \in A$

for contradiction say: if $ij \in G$, $j \in A$, $i \notin A$
then $ij \in H$, $C(ij) = +\infty$

$j \in S$, $i \in T$
$cap(S,T) = +\infty$



if $cap(S,T) \neq +\infty$
then A is a valid selection of jobs

If A is a valid selection of jobs
then $S = A \cup \{s\}$ $T = V - S$ is a cut
in H where $cap(S,T) \neq +\infty$
(S,T) is a cut in H
$cap(S,T)$ because we don't have any edge $i \in S, j \in T$ $\Leftrightarrow i \in A$ $\Leftrightarrow j \notin A$
$C(ij) = +\infty$ if i depends on j which contradicts
our assumption that A was a valid selection

$$cap(S,T) = \underbrace{\sum_{\substack{s_i \in E_H \\ i \notin A}}^{C(s_i)} P_i + \sum_{\substack{it \in E_H \\ i \in A}}^{C(it)} -P_i}$$

$$profit(A) = \sum_{i \in A} P_i$$

$$= \overbrace{\sum_{s_i \in E_H} P_i - \sum_{\substack{s_i \in E_H \\ i \in A}} P_i - \sum_{\substack{it \in E_H \\ i \in A}} P_i}^{profit(A)}$$

Constant ⇒ change min cut

$$\Leftrightarrow \sum_{P_i \geq 0} P_i$$

Oct 24 Lecture

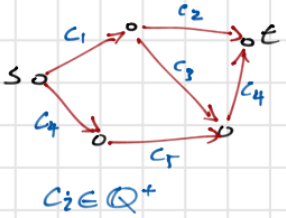
Network Flow

input: directed graph $G=(V,E)$

all the edges have a capacity

flow defined as a function

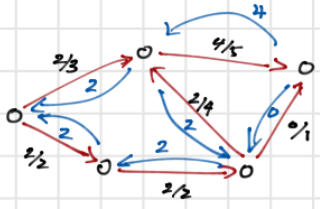
$f: E \to \mathbb{Q}^+$



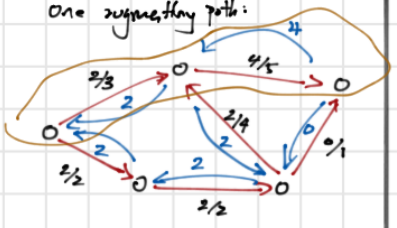
1. $\forall e \in E: f(e) \geq 0$
2. $\forall u \in E: \sum_{(w,u)\in E} f(w,u) = \sum_{(u,v)\in E} f(u,v)$
3. $f(e) \leq c_e \quad \forall e \in E$

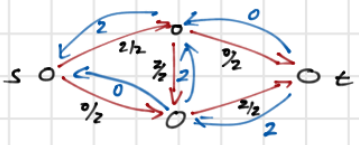
Ford-Fulkerson Augmenting Path Algorithm

residual edge indicates how much flow remains at each vertex?



one augmenting path:





$\Longrightarrow$ BFS only cross edges w/ none zero flow or none max capacity

an augmenting path

- decrement all possible edges not at capacity and $\longrightarrow$?

prob 1 $\longrightarrow$ set of teachers $t_1, t_2, \dots, t_n \to (T_n, h_n)$

set of classes $c_1, c_2, \dots, c_n \to (T_1, h_1)$

$\#$ of classes teacher i is allowed to teach $\hookrightarrow h_i \in \mathbb{Z}^+$

classes teacher 1 is comfortable with teaching $T_i \subseteq \{c_1, c_2, \dots, c_n\}$

problem: Give an assignment of teachers

1. Teacher only teach the classes they're comfortable w/
2. Every Teacher t_i is assigned to exactly h_i classes
3. At most one teacher is assigned to each class

ex. constraints:

$t_1 \to T_1 = \{c_1, c_2\}$ $\quad h_1 = 1$

$t_2 \to T_2 = \{c_3, c_4\}$ $\quad h_2 = 2$

$t_3 \to T_3 = \{c_1, c_2, c_3, c_4, c_5\}$ $\quad h_3 = 2$

classes: $c_1, c_2, \dots, c_5$

solution: $t_1 \mapsto c_1$

$t_2 \mapsto c_2$

$t_3 \mapsto c_3$

given an instance of Prob 1:

$P = \{t_1, t_2, \dots, t_n\}$ where $t_i = \{T_i, h_i\}$

$C = \{c_1, c_2, \dots, c_m\}$ $\quad T_i \subseteq C, h_i \in \mathbb{Z}^+$

full flow $\Rightarrow$ all hours satisfied

P C

Consider the $\#$ of classes 1 amount of flow each $\longrightarrow$ from P to C cost 1

maps only to classes t_i are comfortable with

every flow from P to C represents a class assignment

need to prove a lemma:

if a max flow $\Rightarrow$?

```
# Generate the sub-list, working backwards.
S := [b]
while N[b] != b:
    S := N[b] + S
    b := N[b]
```

Problem 2: Consider the problem of creating a weekly schedule of TA office hours. You are given a list of TA's $t_1, t_2, \ldots, t_n$ and a list of time slots $s_1, s_2, \ldots, s_m$ for office hours. Each TA is available for some of the time slots and unavailable for others. Each time slot s_j must be assigned at most one TA, and every week, each TA t_i is responsible for some positive integer number of office hours h_i.
We want to know if there is a feasible schedule of office hours, *i.e.*, if it is possible to assign time slots to TA's to satisfy all of the problem constraints (each TA gets exactly h_i time slots and each time slot gets at most one TA — some time slots may remain unfilled).

(a) Describe precisely how to model this problem as a network flow problem. (Don't forget to specify all edge directions and capacities in your network.)

Solution: Create a network N with

- vertices $V = \{s, s_1, \ldots, s_m, t_1, \ldots, t_n, t\}$,
- edges $E = \{(s, s_i) : 1 \leqslant i \leqslant m\} \cup \{(s_i, t_j) : 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$, and TA t_j is available at time $s_i\} \cup \{(t_j, t) : 1 \leqslant j \leqslant n\}$, where $c(s, s_i) = 1$ and $c(s_i, t_j) = 1$ and $c(t_j, t) = h_j$ for $1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$.
  (Note: It is also correct to do this with all edges directed in the opposite direction.)

(b) Explain clearly the correspondence between valid assignments of TAs to office hour time slots and valid integer flows in your network above.

Solution:

- Every valid assignment of TAs to time slots generates a valid flow in N by setting $f(s_i, t_j) = 1$ iff t_j is assigned to s_i, $f(s, s_i) = 1$ iff someone is assigned to time s_i, $f(t_j, t) =$ the number of hours assigned to t_j.
- Every valid integer flow in N corresponds to a valid assignment of TAs to time slots by assigning t_j to s_i for all edges with $f(s_i, t_j) = 1$, because no time can have more than one TA assigned and no TA t_i can be assigned to more than h_i times, by the capacity and conservation constraints.

Problem 3 [If you have time]:
Consider the following "teaching assignment" problem: We are given a set of profs $p_1, ..., p_n$ with teaching loads $L_1, ..., L_n$, and a set of courses $c_1, ..., c_m$ with number of sections $S_1, ..., S_m$, along with subsets of courses that each prof is available to teach. The goal is to assign profs to courses so that: (1) each prof p_i assigned exactly L_i courses, and (2) each course c_j assigned exactly S_j profs.

Show how to represent this problem as a network flow, and how to solve it using network flow algorithms. Justify carefully that your solution is correct and can be obtained in polytime.

Solution: Given input, create network with vertices $p_1, ..., p_n, c_1, ..., c_m$, source s, sink t, and edges (s, p_i) of capacity L_i for each p_i, edges (c_j, t) of capacity S_j for each c_j, edges (p_i, c_j) of capacity 1 for each p_i, c_j such that p_i is available to teach c_j.

- Any assignment of profs to courses yields flow in network: set $f(p_i, c_j) = 1$ if p_i assigned c_j, 0 otherwise; set $f(s, p_i) =$ number of courses assigned to p_i; set $f(c_j, t) =$ number of profs assigned to c_j. Value of this flow

= number of course sections assigned. This implies maximum flow in network at least as large as maximum number of course sections that can be assigned.

- Any integer flow in network yields assignment of profs to courses: assign p_i to c_j iff $f(p_i, c_j) = 1$. By capacity constraints, no prof can be assigned more than L_i courses, no course can be assigned more than S_j profs, and no prof will be assigned to a course they are unavailable to teach. This means the maximum number of courses sections that can be assigned is at least as large as the maximum flow value for the network.

In other words, maximum flow value = maximum number of course sections that can be assigned. So, find max flow f (in polytime). If $|f| = L_1 + ... + L_n = S_1 + ... + S_m$, then it is possible to assign profs to courses to satisfy all constraints (as indicated above); otherwise it isn't. If it is not possible, max flow yields max assignment possible. This could be used to determine set of courses that can be offered, or maximum teaching load for profs, for example.