Lec25-flow-applications

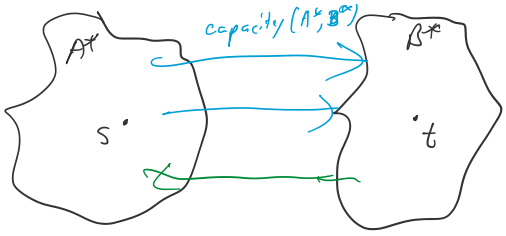
Wednesday, November 6, 2024 4:24 AM

KT 7.5 – 7.7

Recall: Last time we showed that min-cut & max-flow problems are *dual*

different ways of seeing the same problem.

Equivalence: Ford-Fulkerson keeps augmenting paths until no s–t path exists in G_{f^*}, so there is a cut (A^*, B^*).



$v(f^*) = \text{capacity}(A^*, B^*)$

Also, can't have higher $v(f^*)$

(need saturation or else still s–t path in G_{f^*})

$\Rightarrow$ f^* is a max-flow & (A^*, B^*) is a min-cut.

Note: Ford-Fulkerson only works with integer weights for $O(mC)$ time.

For irrational weights, need to deal with cycles explicitly, but a variation Edmunds-Karp works in $O(VE^2)$ time.

Reductions

A reduction rewrites one problem A as a different problem A', so that if we can solve A', then we can get a solution of A.

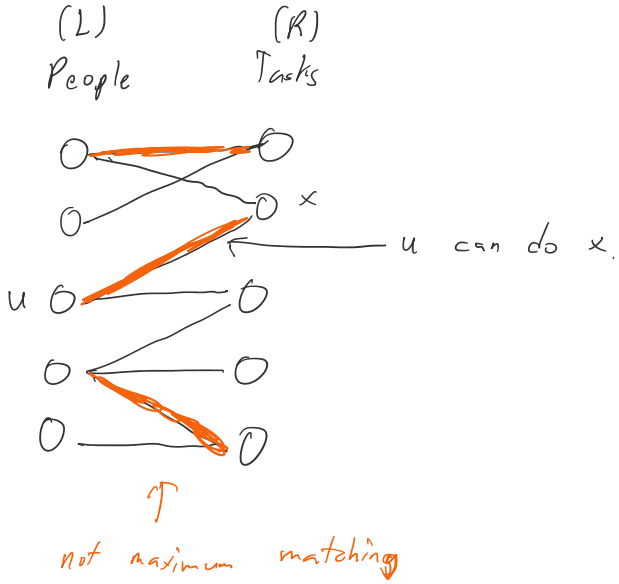
Ex. We rewrote problems as shortest path to use things like Dijkstra on A^*.

Maximum Bipartite Matching

- Set of people L & tasks R
- Each person can only do certain jobs
- Bipartite graph

A matching assigns people to tasks

A maximum matching contains as many edges as possible

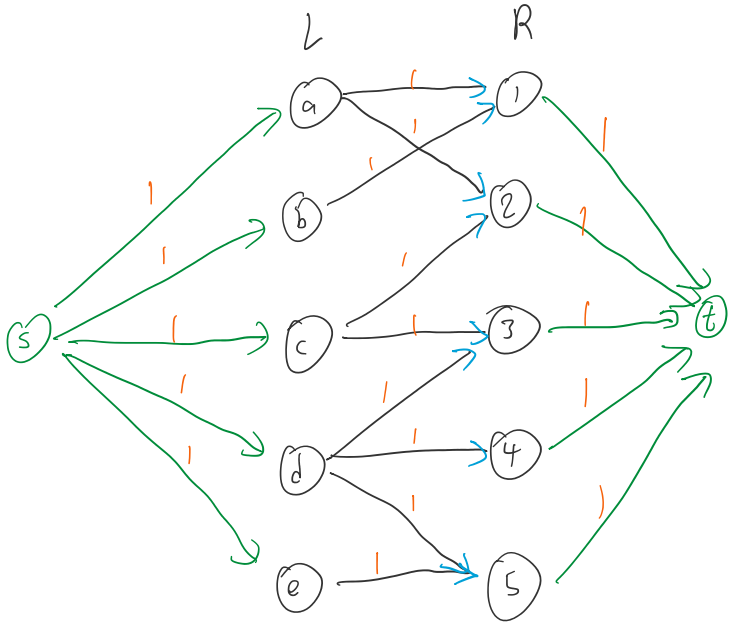


not maximum matching

Formally: Given bipartite graph $G = (L \cup R, E)$, find $S \subseteq E \subseteq L \times R$ that is a matching and as large as possible

Note:
- Given L & R, so don't need to find them, though could using BFS
- S is a perfect matching if every vertex is matched.
- Maximum (↑ as big as possible) vs. maximal (↑ can't make any bigger w/o undoing a choice)

Reduction: Convert problem into network flow problem to solve, where the solution to the flow problem is easily converted back into original matching problem.



turn into directed graph by pointing all arrows from L to R.

Add vertex s pointing to all $v \in L$
Add vertex t pointing from all $v \in R$

Assign all edges capacity 1

Solve max-flow on new graph G' using Ford-Fulkerson

Analysis: Ford-Fulkerson augments with s-t paths, each one having value 1.

Flows ⇒ matchings
Let M be the set of $L \to R$ edges we use in flow f.

M must be a matching because if two edges are used touching a node, the flow f will be unbalanced, since flow through a node is ≤ 1 by construction.
Also, $|M| = v(f)$ because each edge implies a different unique node in L used, which must have 1 flow going through it from s.

⇒ max-flows found by Ford-Fulkerson correspond to matchings, of same value.

All matchings correspond to an integral flow
Given a matching simply push flow into all nodes of L with a selected out-edge, and out of all nodes of R with a selected in-edge.

$\Rightarrow$ A max-flow is also a maximum matching

Runtime:

Ford-Fulkerson takes $O(m'C)$ time, where m' is number of edges in flow network

$= m + 2n$ ← edges in G; ← new edges

$$C = \sum_{e \in Out(s)} c_e = |Out(s)| = |L| = n.$$

$$= O((m+2n)n) = O(mn + n^2) = O(mn) \text{ time.}$$

for maximum bipartite matching.

Even though maximum bipartite matching doesn't look like a flow problem, we can reduce it to a flow problem.

Max-flow extensions

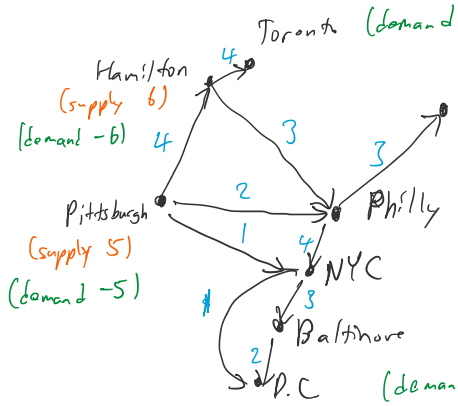
The max-flow problem had just a single source s and target t. (sink)

What if multiple sources & targets.

Circulation with demands has multiple sources S (supply) and multiple sinks T (demand).

Supply = negative demand. $\forall s \in S$, let $d_s = -\text{supply}$

$\forall t \in T$, let $d_t = \text{demand}$



Feasible flow must have

$$\sum_{s \in S} d_s + \sum_{t \in T} d_t = 0.$$

Let $D = \sum_{t \in T} d_t$.

Also usual constraints:

$\forall e \in E, \quad 0 \le f(e) \le c_e$

$\forall v \in V, \quad f^{in}(v) - f^{out}(v) = d_v$

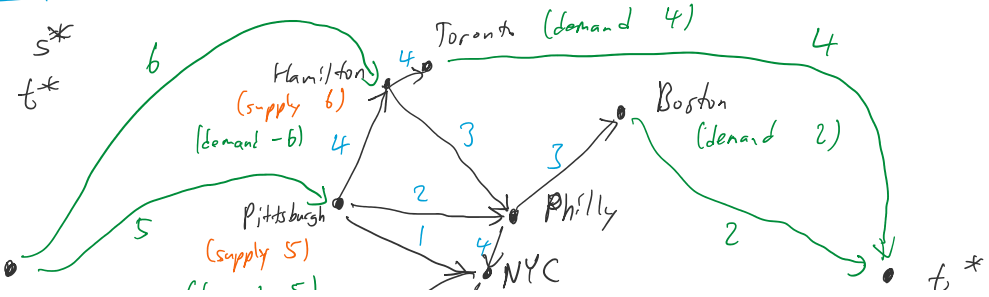
($d_v = 0$ for nodes not in S or T)

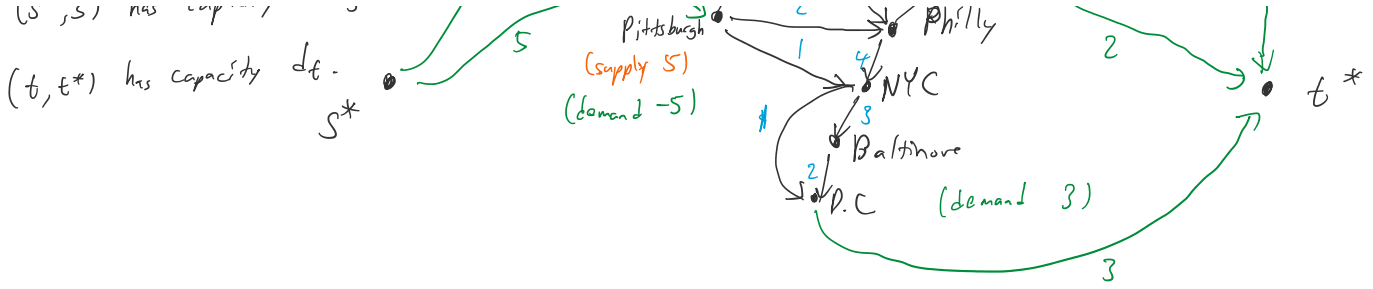
Reduction to max-flow

Add new super-source s^* & super-sink t^*

(s^*, s) has capacity $-d_s$.

(t, t^*) has capacity d_t.





Feasible circulation if and only if there is a flow of value D, from s^* to t^*.

Circulation with lower bounds

Suppose Pittsburg made a commitment to always ship at least 1 ton of steel on the Pittsburgh – Philly line. Can we modify our algorithm?

New constraints: $\forall e \in E, \quad \ell_e \le f(e) \le c_e$

$\forall v \in V, \quad f^{in}(v) - f^{out}(v) = d_v.$

We want to convert into ordinary circulation w/ demands problem.

Idea: Start by preallocating the flow demanded by ℓ_e as part of the demands by the nodes.

Define flow f_0 by setting $f_0(e) = \ell_e \quad \forall e \in E$.

– meets capacity constraint but not demand constraint.

Define $L_v = f_0^{in}(v) - f_0^{out}(v)$.

(normally, we need $f^{in}(v) - f^{out}(v) = d_v$ for demand constraint.)

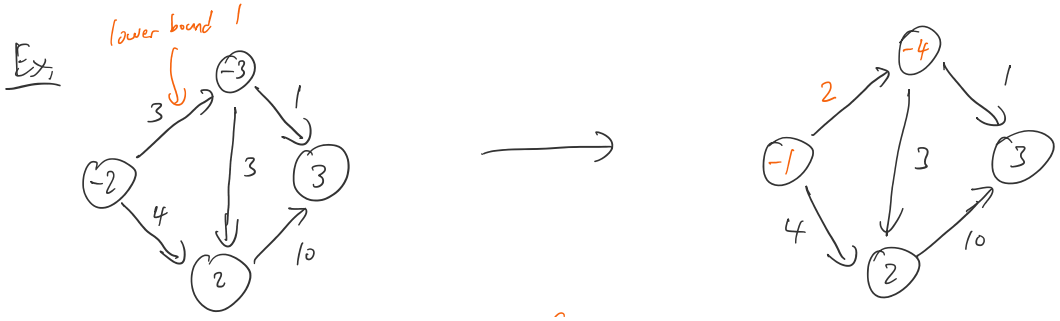
New demand constraint: $f^{in}(v) - f^{out}(v) = d_v - L_v$.

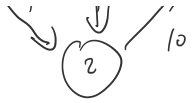
and new capacity constraint: $0 \le f(e) \le c_e - \ell_e$.

(since we used some of the flow in f_0 already)

$\Rightarrow$ new standard circulation problem.

Ex.





lower bound messes up obvious solution

(Reduction) Transformation to equivalent problem w/o lower bounds

Serial reductions: circulation with lower bounds

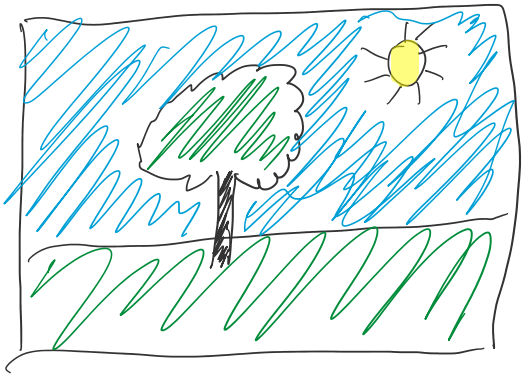
↓

circulation w/o lower bounds

↓

max flow

Image segmentation:



Given an image, what is foreground & what is background?

e.g. a tree against a blue sky.

Idea: Use color. If p_i is blue, likely to be background.

If p_i is green, likely to be foreground.

More formally, have likelihoods

a_i = likelihood pixel is in foreground

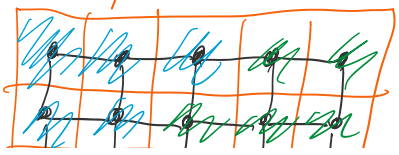
b_i = likelihood pixel is in background

Caveat: If p_i is a foreground pixel, then nearby pixels are also likely foreground.

Idea: Encode image by undirected graph $G(V, E)$

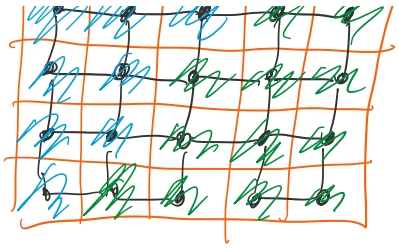
V: pixels E: edges between all neighbors

excerpt from image as graph



penalty p_{ij} for putting i in foreground

sky



tree

penalty p_{ij} for putting i in foreground and j in background or vice versa.

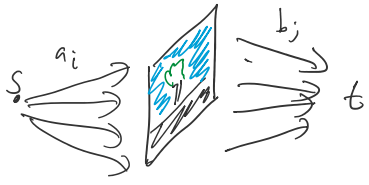
Image segmentation problem: Partition the set of pixels into two sets A & B to maximize

$$q(A,B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i,j \text{ sep.}}} p_{ij}.$$

Minimum Cut problem: Partition the vertices of a directed graph into 2 sets A & B with $s \in A$, $t \in B$ to minimize weight of edges crossing from A to B.

Differences:
- max vs. min
- source & sink
- objective functn $q(A,B)$
- undirected vs. directed.

Reduction: Make all edges into two directed edges with capacity p_{ij}.
Add source s with edges pointing to every pixel with capacity a_i
Add sink t with edges pointing from every pixel with capacity b_j.



Want to maximize $q(A,B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i,j \text{ sep}}} p_{ij}$

Notice: Let $Q = \sum_{i \in V} (a_i + b_i) = \left(\sum_{i \in A} a_i + \sum_{j \in B} b_j\right) + \left(\sum_{i \in A} b_i + \sum_{j \in B} a_j\right)$.

maxing this = minning this

Same as minimizing $q'(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_i + \sum_{(i,j) \in E} p_{ij}$

Same as minimizing $q'(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i,j) \in E,\ i,j \text{ sep}} p_{ij}$

Capacity of cut (A,B):

- $\sum_{i \in A} b_i$: cutting edges $(i,t)\ \forall i \in A$ (cost of putting i in foreground)
- $\sum_{j \in B} a_j$: cutting edges $(s,i)\ \forall i \in B$ (cost of putting i in background)
- $\sum_{(i,j) \in E,\ i,j \text{ sep}} p_{ij}$: cuts b/t edges in image

Thus finding the minimum cut in this network flow is exactly optimizing the foreground-background segmentation.