CSC 373 Lecture Summary for Week 5 Winter 2019

Network Flow

Definition: "network" = directed graph N = (V,E) with

- a single "source" s (- V with no incoming edge,
- a single "sink" t (- V with no outgoing edge,
- nonnegative integer "capacity" c(e) for each edge e (- E.
- Networks can be used to model, e.g., computer networks (capacity = bandwidth), electrical networks, etc.
- (Example: V = {s,a,b,c,d,t}, E = { (s,a):16, (s,b):13, (a,c):12, (b,a):4, (b,d):14, (c,b):9, (c,t):20, (d,c):7, (d,t):4 }.)

Network flow problem: Assign flow f(e) (- R to each edge e such that we have maximum flow "in the network" (to be defined), subject to:

- capacity constraint: for each edge e, 0 <= f(e) <= c(e) (flow does not exceed capacity);
- conservation constraint: for each vertex v != s,t, f^in(v) = f^out(v), where f^in(v) = total flow into v = sum_{(u,v) (- E} f(u,v) and f^out(v) = total flow out of v = sum_{(v,u) (- E} f(v,u);
- total flow in network is denoted |f| and defined as |f| = f^out(s) (by conservation, |f| = f^in(t); this will be proved later).

Brute force? \Omega(\prod_{e (- E} c(e)) for integer flows -- each edge e can get a flow of 0,1,2,...,c(e), and we consider all possibilities independently of other edges -- much worse than simple exponential!

Greedy? No way to select any part of flow greedily.

Dynamic programming? No way to break down problem into independent recursive sub-problems.

Idea: Local search strategy: start with initial assignment of flow guaranteed to be correct but not necessarily maximum, then try to make incremental improvements -- stop when no improvement possible.

- Ford-Fulkerson algorithm:

```
start with any valid flow f (e.g., f(e) = 0 for all e (- E)
while there is an "augmenting path" P:
    "augment" f using P
output f
```

Augmenting paths? Augment a flow?

- Intuition: Since all flow must "start" at s and "end" at t, find s-t paths along which flow can be increased. Instead of adding flow to edges in haphazard manner, this preserves conservation.

- First idea: path P = s -> ... -> t where f(e) < c(e) for each e. Define "residual capacity" for edges: \Delta_f(e) = c(e) - f(e), and residual capacity for paths: \Delta_f(P) = MIN_{e (- P} \Delta_f(e). Augment path by adding \Delta_f(P) to all edge flows.

- Problem: notion too narrow, can get stuck with sub-optimal solution. (Example: f(s,a) = 8, f(s,b) = 13, f(a,c) = 12, f(b,a) = 4, f(b,d) = 9, f(d,c) = 5, f(d,t) = 4, f(c,t) = 17 on earlier network.)

- Second idea: allow flow to decrease along some edges (instead of only using edges on which flow increases).

Residual network.

- Network N, flow f => residual network N_f:
  . same vertices as N;
  . for each edge (u,v) in N with f(u,v) < c(u,v), N_f contains "forward" edge (u,v) with capacity c_f(u,v) = c(u,v) - f(u,v);
  . for each edge (u,v) in N with f(u,v) > 0, N_f contains "backward" edge (v,u) with capacity c_f(v,u) = f(u,v).
  Intuition: "forward edge" has unused capacity that can be used to push more flow from s to t; "backward edge" has surplus flow that can be redirected to push more flow from s to t.
  Note: this is a form of backtracking -- changing our mind about previously assigned flow -- but in a specific, controlled fashion.

- Augmenting path = any s-t path in N_f.

- Augmentation: add \Delta_f(P) (defined as before) to forward edges, subtract it from backward edges.

  Example: s -8-> a -4-> b -5-> d -2-> c -3-> t in residual network for earlier example. \Delta_f(P) = 2 (minimum c_f for edges on P).
  Add 2 to each edge traversed forward (s,a), (b,d), (d,c), (c,t) and subtracting 2 from each edge traversed backward (b,a).
  New flow: f(s,a) = 10, f(b,a) = 2, f(b,d) = 11, f(d,c) = 7, f(c,t) = 19.

Correctness of Ford-Fulkerson Algorithm:

- A "cut" is a partition of V into V_s, V_t (i.e., V = V_s u V_t and V_s n V_t = {}) such that s (- V_s, t (- V_t;
  . an edge (u,v) with u (- V_s, v (- V_t is a "forward" edge;
  . an edge (u,v) with u (- V_t, v (- V_s is a "backward" edge.
  Careful! Two different notions of "forward/backward": with respect to augmenting paths and with respect to cuts.

- For any cut X=(V_s,V_t),
  . The "capacity" of cut X is the sum of the capacities of the forward edges: c(X) = sum_{e forward} c(e).
  . The "flow across X" is the total flow forward minus the total flow backward across the cut:
        f(X) = sum_{e forward} f(e) - sum_{e backward} f(e).

- Example: X_0 = (V_x = {s,a}; V_t = {b,c,d,t}) on ongoing example network. c(X_0) = c(a,c) + c(s,b) = 12 + 13 = 25 -- don't count backward edge (b,a).

- Lemma: For any cut X and any flow f, f(X) <= c(X).
  Proof:
      f(X) = sum_{e forward} f(e) - sum_{e backward} f(e)
           <= sum_{e forward} f(e)
           <= sum_{e forward} c(e) = c(X).

- Lemma: For any cut X and any flow f, f(X) = |f|.
  Proof: Omitted for length. Intuition: all flow "generated" at s and "consumed" at t so value across any cut remains constant.

- Corollary: For any cut X and any flow f, |f| <= c(X). (From two facts above). In particular, max flow in network <= min capacity of any cut.

- Theorem (Ford-Fulkerson): For any network N and flow f, |f| is maximum (and equal to c(X) for some cut X) iff there is no augmenting path.

  Proof: (=>) augment
  (<=) Construct cut X as follows:
    . start with V_s = {s}, V_t = V - {s};
    . if (u,v) (- E with u (- V_s, v (- V_t, f(u,v) < c(u,v), then move v from V_t to V_s;
    . if (u,v) (- E with v (- V_s, u (- V_t, f(u,v) > 0, then move u from V_t to V_s;
    . repeat until no further change possible.
  Since there is no augmenting path, this must stop with t (- V_t (otherwise, there is some 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. Hence, |f| = f(X) = c(X).

  Corollary (max-flow/min-cut theorem): For any network, the maximum flow value equals the minimum cut capacity.

- Additional property: because of nature of augmentation, can prove by induction that max flow can always be achieved with integer flow values (as long as all capacities are integer).

For Next Week
* Readings: Sections 7.3, 7.1
* Self-Test: Exercises 7.17(a), 7.17(b), 7.2