CSE 347 Lecture 10

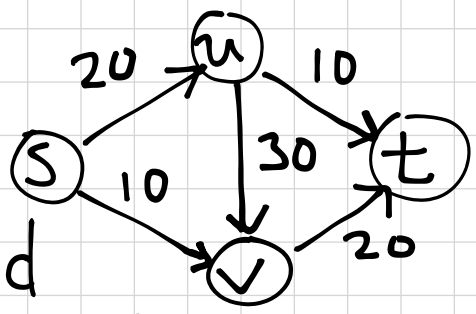
Max-Flow I: Ford-Fulkerson

1. The Max-Flow Problem
2. Residual graphs & augmenting paths
3. The Ford-Fulkerson Algorithm

Up to now, we've studied families of algorithms characterized in terms of their basic structure. In the next part of the course, we'll study algorithms based on "reductions" – that is, we will identify a core computational problem that turns out to capture the "difficult part" of many other problems. Then, given an algorithm for this "difficult part" as a subroutine, algorithms for these many other problems often consist of no more than "translating" the new problem back to the one we can already solve.

The specific "difficult" problem we'll consider today (which is one of several possible such problems) is called "Maximum Flow." It originates in the study of transportation networks: let's consider a weighted directed graph $G = (V, E)$ in which the weight of an edge, $C(e)$ for $e \in E$, is the number of vehicles that can simultaneously traverse e (per unit of time) – e.g., the number of lanes. We

may refer to this as the capacity of e. The basic question we consider is the following. We wish to route vehicles from some specified starting point $s \in V$ to a specified destination $t \in V$ as quickly as possible. Specifically, we want an assignment of some average number of vehicles to send along each road per unit of time — a flow in G. Naturally, this assignment $f: E \to \mathbb{R}^+$ must satisfy two constraints: (1) f cannot exceed the capacity of any $e \in E$, i.e., $0 \leq f(e) \leq c(e)$ for all $e \in E$.



(2) the amount of traffic arriving at each node matches the amount leaving for all nodes except the start and destination: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e) \quad \forall v \notin \{s, t\}$.
We don't constrain the amount leaving the start and arriving at the destination — rather, maximizing this quantity is the objective. That is, precisely, we wish to maximize $v(f) = \sum_{e \text{ out of } s} f(e)$.

For example, what is the maximum rate of flow in our example graph? 30 — we can route 20 along (s,u), of which 10 continue along (u,t) and 10 traverse (u,v). We also route 10 along (s,v); so, the 10+10=20 vehicles arriving at v per unit time can be routed along (v,t). Notice, no more than 20+10=30 vehicles can possibly exit s/enter t per unit time,

so this must be optimal.

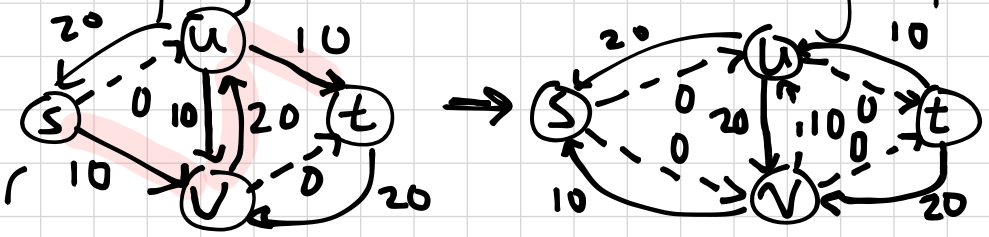
For convenience, we'll let $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$ and $f^{in}(v) = \sum_{e \text{ into } v} f(e)$. (So, our objective is to maximize $f^{out}(s)$.) We can extend this notation to sets of vertices, $S \subseteq V$: we define $f^{out}(S) = \sum_{v \in S} f^{out}(v)$ and $f^{in}(S) = \sum_{v \in S} f^{in}(v)$.

There's no clear way to use dynamic programming to solve this problem, but an interesting variant of a greedy algorithm will turn out to work. Let's consider a naive greedy algorithm first, and see why it fails: suppose we start with zero flow ($f(e) = 0 \; \forall e$) and repeatedly try to add (somehow) an additional batch of flow along a path with the maximum remaining capacity. (Nevermind, for now, how we do this.) What happens in the example graph? The highest capacity path pushes 20 vehicles along both 20-capacity edges by following the 30-capacity route from u to v—but now we're stuck, since any vehicles routed along the 10-capacity route from s to v have nowhere to go, since the u-to-v route is one-way and the v-to-t route is at full capacity.

The problem is that our 30 vehicle routing wisely diverts 10 vehicles from the s-to-u route along u-to-t to make room for the extra 10 from s-to-v along v-to-t

Residual Graphs & Augmenting Paths

We can fix the greedy algorithm by enabling it to (partially) undo its assignments and divert traffic along other routes. We represent this with a residual graph in which (1) we reduce the capacity of each edge in G by the amount traversing it in f and (2) we add (or increase the capacity of) a backwards edge, from v-to-u, with a capacity $f((u,v))$ for each edge (u,v) with nonzero flow. This represents the amount of f that could be diverted from the u-to-v route to take some other path from u, to make room for additional flow entering v (that subsequently obeys the same assignment for traffic entering v we used previously.) We then only need to find a new path for the diverted flow from u — e.g., in our example, the residual graph after our greedy choice is:



Now, we'd like to route another 10 vehicles by "undoing" 10 vehicles worth of assignment along the u-to-v route and diverting them along u-to-t instead. Observe that this is simply a 10-capacity path in the residual graph. We call this an "augmenting path." Indeed, it always suffices to simply find such paths: let P be a simple path from s-to-t (i.e., that visits a node at most once) in the residual graph G_f, and let bottleneck(P,f) be the minimum capacity of any edge traversed by P in G_f. Then the operation

augment(f,P) increases f(e) for each forward edge e of G_f traversed by P by bottleneck(P,f), and decreases f(e) for each edge edge e of G for which P traverses the reverse edge in G_f (also by bottleneck(P,f)).

Lemma augment(f,P) satisfies the flow constraints of G.

Proof? (1) The capacity constraints for $e \notin P$ are satisfied since these assignments are untouched, as are the conservation constraints for any v not visited in P. (2) for $e \in P$, if e was a forward edge, bottleneck$(P,f) \le c(e) - f(e)$, so after augment(f,P) we obtain f' with $f'(e) \le f(e) + c(e) - f(e)$. If e was a backwards edge, bottleneck$(P,f) \le f(e)$, so $f'(e) \ge f(e) - f(e) = 0$. Thus, f' satisfies the capacity constraint. (3) At each v traversed, since P is a simple path, we take one edge (u,v) in to v, and one edge (v,w) out. Either both have their flow increased/decreased by $b = \text{bottleneck}(P,f)$, so $f'^{in}(v) = \sum_{e \text{ in to } v} f(e) \pm b = \sum_{e \text{ out of } v} f(e) \pm b = f'^{out}(v)$, or else flow is redirected from one {in-/out-going} edge to another, so $f'^{out}(v) = \sum_{e \text{ out of } v} f(e) + b - b = f'^{in}(v)$ (or $= \sum_{e \text{ in to } v} f(e) + b - b = \dots$)

The Ford-Fulkerson Algorithm

We have thus arrived at the following extension of the natural greedy algorithm:

Ford-Fulkerson(G):
- Initialize $f(e) \leftarrow 0$ for all $e \in E$, $G_f \leftarrow G$
- While there is a simple s-t path P in residual graph G_f
  - $f \leftarrow$ augment(f,P)
  - $G_f \leftarrow$ the residual graph of G with f

Return f.

We'll assume, without much loss in generality, that no edges in G enter s or leave t, and that the capacities are integers. We'll first observe that the capacities remain integers:

Lemma: For all e, $f(e)$ and the capacity of e in G_f are integers

Proof by induction on # of iterations: Base, 0 iterations, is immediate. For the induction step, observe bottleneck(P,f) is the capacity of some edge of G_f, which by IH is an integer, so augment(f,P) simply changes the integer-valued f by some integer amount on each edge. Likewise, each capacity in G_f is now either this integer $f(e)$, or else $c(e)-f(e)$, which is again an integer ▨

Next, we observe the value increases on each iteration:

Lemma Each iteration computes a flow f' with value $v(f') = v(f) + \text{bottleneck}(P,f) > v(f)$ (since bottleneck$(P,f) > 0$)

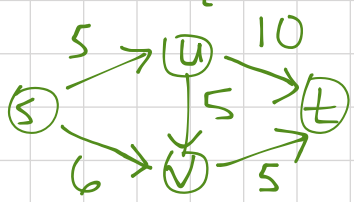
Proof: The first edge of the s-t path P leaves s, and since P is simple, it does not enter s again. Moreover, since we assumed no edges enter s in G, the edge we traverse from s cannot be one of the backwards edges added in G_f. Thus we increase the flow on this edge leaving s by bottleneck(P,f), and do not alter the flow on any other edge incident to s, and we see $v(f') = f'^{out}(s) = \sum_{e \text{ out of } s} f(e) + \text{bottleneck}(P,f) = v(f) + \text{bottleneck}(P,f)$ ▨

So, finally, we observe that the amount of flow leaving s is at most $C = \sum_{e \text{ out of } s} c(e)$. Since $v(f)$ increases by an integer amount – at least 1 – on each iteration,

we find that the while loop terminates within C iterations. Next time, we'll leverage this fact to prove that Ford-Fulkerson can be implemented to run in time $O(|E| \cdot C)$. We will also prove it indeed computes a maximum flow.

Counterexample to greedy algorithm

"take [locally] smallest-capacity path"



Achieves 5

OPT = 10