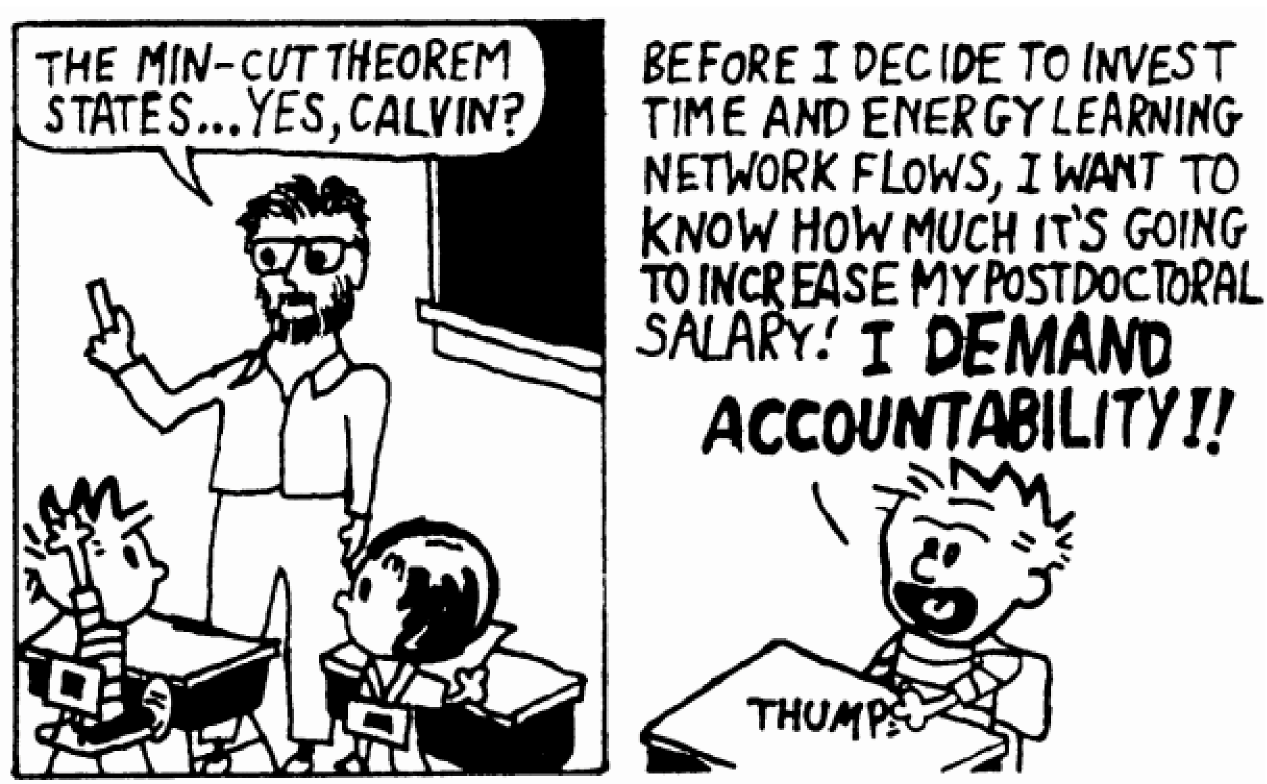
Chapter 12

Network Flow II

CS 573: Algorithms, Fall 2014
October 2, 2014

12.0.1 Accountability

12.0.1.1 Accountability



`http://www.cs.berkeley.edu/~jrs/`Calvin

12.0.1.2 Accountability

(A) People that do not know maximum flows: essentially everybody.
(B) Average salary on earth ¡ $5,000
(C) People that know maximum flow – most of them work in programming related jobs and make at least $10,000 a year.

(D) Salary of people that learned maximum flows: $> \$10,000$
(E) Salary of people that did not learn maximum flows: $< \$5,000$.
(F) Salary of people that know Latin: 0 (unemployed).
Conclusion ***Thus, by just learning maximum flows (and not knowing Latin) you can double your future salary!***

12.0.2 The Ford-Fulkerson Method

12.0.2.1 Ford Fulkerson

```
algFordFulkerson(G,s,t)
    Initialize flow f to zero
    while ∃ path π from s to t in G_f do
        c_f(π) ← min { c_f(u,v) | (u → v) ∈ π }
        for ∀ (u → v) ∈ π do
            f(u,v) ← f(u,v) + c_f(π)
            f(v,u) ← f(v,u) − c_f(π)
```

Lemma 12.0.1. *If the capacities on the edges of* G *are integers, then* **algFordFulkerson** *runs in* $O(m\,|f^*|)$ *time, where* $|f^*|$ *is the amount of flow in the maximum flow and* $m = |E(\mathsf{G})|$.

12.0.2.2 Proof of Lemma...

Proof: Observe that the **algFordFulkerson** method performs only subtraction, addition and min operations. Thus, if it finds an augmenting path π, then $c_f(\pi)$ must be a *positive* integer number. Namely, $c_f(\pi) \geq 1$. Thus, $|f^*|$ must be an integer number (by induction), and each iteration of the algorithm improves the flow by at least 1. It follows that after $|f^*|$ iterations the algorithm stops. Each iteration takes $O(m+n) = O(m)$ time, as can be easily verified. ∎

12.0.2.3 Integrality theorem

Observation 12.0.2 (Integrality theorem). *If the capacity function* c *takes on only integral values, then the maximum flow* f *produced by the* **algFordFulkerson** *method has the property that* $|f|$ *is integer-valued. Moreover, for all vertices* u *and* v*, the value of* $f(u,v)$ *is also an integer.*

12.0.3 The Edmonds-Karp algorithm

12.0.3.1 Edmonds-Karp algorithm

Edmonds-Karp: modify **algFordFulkerson** so it always returns the shortest augmenting path in G_f.

Definition 12.0.3. For a flow f, let $\delta_f(v)$ be the length of the shortest path from the source s to v in the residual graph G_f. Each edge is considered to be of length 1.

Assume the following key lemma:

Lemma 12.0.4. $\forall v \in V \setminus \{s,t\}$ *the function* $\delta_f(v)$ *increases.*

12.0.3.2 The disappearing/reappearing lemma

Lemma 12.0.5. *During execution* **Edmonds-Karp***, edge* $(u \to v)$ *might disappear/reappear from* G_f *at most* $n/2$ *times,* $n = |V(\mathsf{G})|$.

Proof: (A) iteration when edge $(u \to v)$ disappears.
(B) $(u \to v)$ appeared in augmenting path π.
(C) Fully utilized: $c_f(\pi) = c_f(uv)$. f flow in beginning of iter.
(D) till $(u \to v)$ "magically" reappears.
(E) ... augmenting path σ that contained the edge $(v \to u)$.
(F) g: flow used to compute σ.
(G) We have: $\delta_g(u) = \delta_g(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$
(H) distance of s to u had increased by 2. QED.

12.0.3.3 Comments...

(A) $\delta_?(u)$ might become infinity.
(B) u is no longer reachable from s.
(C) By monotonicity, the edge $(u \to v)$ would never appear again.

Observation 12.0.6. *For every iteration/augmenting path of* **Edmonds-Karp** *algorithm, at least one edge disappears from the residual graph* $\mathsf{G}_?$.

12.0.3.4 Edmonds-Karp # of iterations

Lemma 12.0.7. **Edmonds-Karp** *handles* $O(nm)$ *augmenting paths before it stops.*
Its running time is $O(nm^2)$*, where* $n = |V(\mathsf{G})|$ *and* $m = |E(\mathsf{G})|$.

Proof: (A) Every edge might disappear at most $n/2$ times.
(B) At most $nm/2$ edge disappearances during execution **Edmonds-Karp**.
(C) In each iteration, by path augmentation, at least one edge disappears.
(D) **Edmonds-Karp** algorithm perform at most $O(mn)$ iterations.
(E) Computing augmenting path takes $O(m)$ time.
(F) Overall running time is $O(nm^2)$.

12.0.3.5 Shortest distance increases during Edmonds-Karp execution

Lemma 12.0.8. **Edmonds-Karp** *run on* $\mathsf{G} = (V, E)$*,* s*,* t*, then* $\forall v \in V \setminus \{s, t\}$*, the distance* $\delta_f(v)$ *in* G_f *increases monotonically.*

Proof
(A) By Contradiction. f: flow before (first fatal) iteration.
(B) g: flow after.
(C) v: vertex s.t. $\delta_g(v)$ is minimal, among all counter example vertices.
(D) v: $\delta_g(v)$ is minimal and $\delta_g(v) < \delta_f(v)$.

12.0.3.6 Proof continued...

(A) $\pi = s \to \cdots \to u \to v$: shortest path in G_g from s to v.
(B) $(u \to v) \in \mathsf{E}(\mathsf{G}_g)$, and thus $\delta_g(u) = \delta_g(v) - 1$.
(C) By choice of v: $\delta_g(u) \geq \delta_f(u)$.
  (i) If $(u \to v) \in \mathsf{E}(\mathsf{G}_f)$ then

$$\delta_f(v) \leq \delta_f(u) + 1 \leq \delta_g(u) + 1 = \delta_g(v) - 1 + 1 = \delta_g(v).$$

  This contradicts our assumptions that $\delta_f(v) > \delta_g(v)$.

12.0.3.7 Proof continued II

(ii) f $(u \to v) \notin \mathsf{E}(\mathsf{G}_f)$:
(A) π used in computing g from f contains $(v \to u)$.
(B) $(u \to v)$ reappeared in the residual graph G_g (while not being present in G_f).
(C) $\implies$ π pushed a flow in the other direction on the edge $(u \to v)$. Namely, $(v \to u) \in \pi$.
(D) Algorithm always augment along the shortest path. By assumption $\delta_g(v) < \delta_f(v)$, and definition of u:

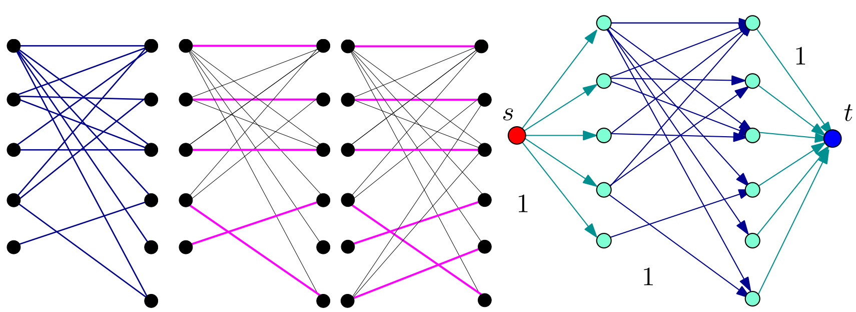
$$\delta_f(u) = \delta_f(v) + 1 > \delta_g(v) = \delta_g(u) + 1,$$

(E) $\implies$ $\delta_f(u) > \delta_g(u)$
  $\implies$ monotonicity property fails for u.
  But: $\delta_g(u) < \delta_g(v)$. A contradiction. ■

12.1 Applications and extensions for Network Flow

12.1.1 Maximum Bipartite Matching

12.1.1.1 Bipartite Matching



12.1.1.2 Bipartite matching

Definition 12.1.1. $\mathsf{G} = (V, E)$: undirected graph.

$M \subseteq E$: ***matching*** if all vertices $v \in V$, at most one edge of M is incident on v.

M is ***maximum matching*** if for any matching M': $|M| \geq |M'|$.

M is ***perfect*** if it involves all vertices.

12.1.1.3 Computing bipartite matching

Theorem 12.1.2. *Compute maximum bipartite matching in $O(nm)$ time.*

Proof: (A) G: bipartite graph G. (n vertices and m edges)
(B) Create new graph H with source on left and sink right.
(C) Direct all edges from left to right. Set all capacities to one.
(D) By Integrality theorem, flow in H is 0/1 on edges.
(E) A flow of value k in $H \implies$ a collection of k vertex disjoint $s-t$ paths $\implies$ matching in G of size k.
(F) M: matching of k edge in G, $\implies$ flow of value k in H.
(G) Running time of the algorithm is $O(nm)$. Max flow is n, and as such, at most n augmenting paths. ■

12.1.1.4 Extension: Multiple Sources and Sinks

Question Given a flow network with several sources and sinks, how can we compute maximum flow on such a network?

Solution The idea is to create a super source, that send all its flow to the old sources and similarly create a super sink that receives all the flow.

Clearly, computing flow in both networks in equivalent.

12.1.1.5 Proof by figures