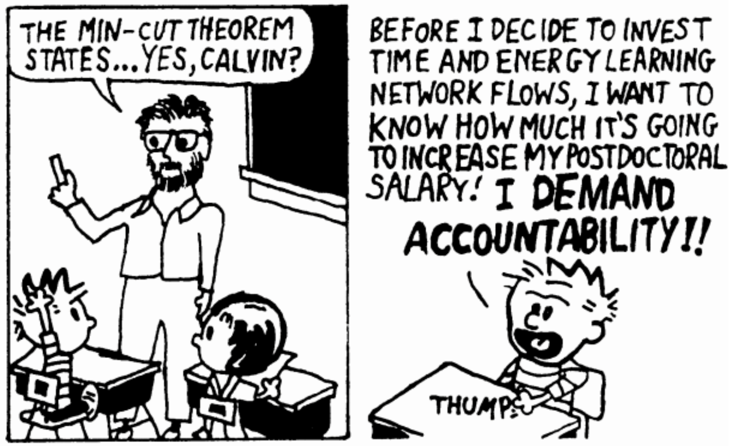
CS 573: Algorithms, Fall 2014

Network Flow II

Lecture 12
October 2, 2014

Accountability



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

Accountability

1. People that do not know maximum flows: essentially everybody.
2. Average salary on earth ¡ **$5,000**
3. People that know maximum flow – most of them work in programming related jobs and make at least **$10,000** a year.
4. Salary of people that learned maximum flows: $>$ **$10,000**
5. Salary of people that did not learn maximum flows: $<$ **$5,000**.
6. 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!

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

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* $m = |E(\mathsf{G})|$.

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. □

Integrality theorem

Observation (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.*

Edmonds-Karp algorithm

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

Definition

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

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

The disappearing/reappearing lemma

Lemma

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

Proof.

1. iteration when edge $(u \to v)$ disappears.
2. $(u \to v)$ appeared in augmenting path π.
3. Fully utilized: $c_f(\pi) = c_f(uv)$. f flow in beginning of iter.
4. till $(u \to v)$ "magically" reappears.
5. ... augmenting path σ that contained the edge $(v \to u)$.
6. g: flow used to compute σ.
7. We have: $\delta_g(u) = \delta_g(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$
8. distance of s to u had increased by 2. QED. □

Comments...

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

Observation

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

Edmonds-Karp # of iterations

Lemma

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

Proof.

1. Every edge might disappear at most $n/2$ times.
2. At most $nm/2$ edge disappearances during execution **Edmonds-Karp**.
3. In each iteration, by path augmentation, at least one edge disappears.
4. **Edmonds-Karp** algorithm perform at most $O(mn)$ iterations.
5. Computing augmenting path takes $O(m)$ time.
6. Overall running time is $O\left(nm^2\right)$.

Shortest distance increases during Edmonds-Karp execution

Lemma

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

1. By Contradiction. f: flow before (first fatal) iteration.
2. g: flow after.
3. v: vertex s.t. $\delta_g(v)$ is minimal, among all counter example vertices.
4. v: $\delta_g(v)$ is minimal and $\delta_g(v) < \delta_f(v)$.

Proof continued...

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

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

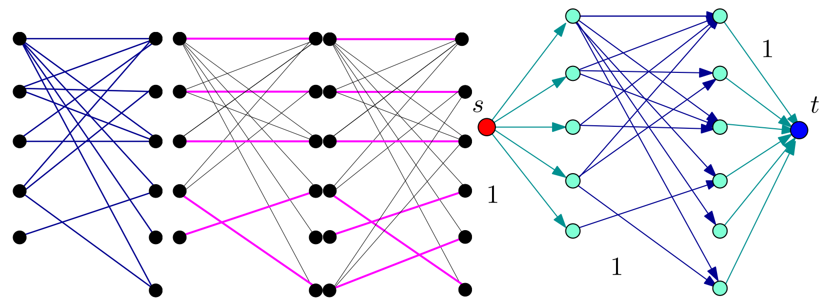
This contradicts our assumptions that $\delta_f(v) > \delta_g(v)$.

Proof continued II

(ii) f $(u \to v) \notin E(G_f)$:

1. π used in computing g from f contains $(v \to u)$.
2. $(u \to v)$ reappeared in the residual graph G_g (while not being present in G_f).
3. $\implies$ π pushed a flow in the other direction on the edge $(u \to v)$. Namely, $(v \to u) \in \pi$.
4. Algorithm always augment along the shortest path. By assumption $\delta_g(v) < \delta_f(v)$, and definition of u:
$$\delta_f(u) = \delta_f(v) + 1 > \delta_g(v) = \delta_g(u) + 1,$$
5. $\implies$ $\delta_f(u) > \delta_g(u)$
$\implies$ monotonicity property fails for u.
But: $\delta_g(u) < \delta_g(v)$. A contradiction. ■

Bipartite Matching



Bipartite matching

Definition

$G = (V, E)$: 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.

Computing bipartite matching

Theorem

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

Proof.

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

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.

Proof by figures