Chapter 13

Network Flow II - The Vengeance

By Sariel Har-Peled, December 7, 2009[1] Version: 0.11

13.1 Accountability

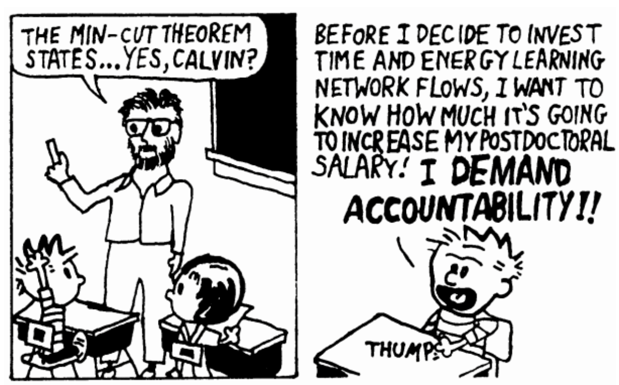


Figure 13.1: `http://www.cs.berkeley.edu/~jrs/`

The comic in Figure 13.1 is by Jonathan Shewchuk and is referring to the Calvin and Hobbes comics.

People that do not know maximum flows: essentially everybody.

Average salary on earth $< \$5,000$

People that know maximum flow - most of them work in programming related jobs and make at least $\$10,000$ a year.

Salary of people that learned maximum flows: $> \$10,000$

Salary of people that did not learn maximum flows: $< \$5,000$

Salary of people that know Latin: 0 (unemployed).

Thus, by just learning maximum flows (and not knowing Latin) you can double your future salary!

13.2 Ford-Fulkerson Method

The **Ford-Fulkerson** method is depicted on the right.

```
Ford-Fulkerson(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 13.2.1 *If the capacities on the edges of* G *are integers, then* **Ford-Fulkerson** *runs in* $O(m|f^*|)$ *time, where* $|f^*|$ *is the amount of flow in the maximum flow and* $m = |E(\mathsf{G})|$.

Proof: Observe that the **Ford-Fulkerson** 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. ■

The following observation is an easy consequence of our discussion.

Observation 13.2.2 (Integrality theorem.) *If the capacity function c takes on only integral values, then the maximum flow f produced by the* **Ford-Fulkerson** *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.*

13.3 The Edmonds-Karp algorithm

The **Edmonds-Karp** algorithm works by modifying the **Ford-Fulkerson** method so that it always returns the shortest augmenting path in G_f (i.e., path with smallest number of edges). This is implemented by finding π using **BFS** in G_f.

Definition 13.3.1 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.

We will shortly prove that for any vertex $v \in V \setminus \{s,t\}$ the function $\delta_f(v)$, in the residual network G_f, increases monotonically with each flow augmentation. We delay proving this (key) technical fact (see Lemma 13.3.5 below), and first show its implications.

Lemma 13.3.2 *During the execution of the* **Edmonds-Karp** *algorithm, an edge* $(u \to v)$ *might disappear (and thus reappear) from* G_f *at most* $n/2$ *times throughout the execution of the algorithm, where* $n = |V(\mathsf{G})|$.

Proof: Consider an iteration when the edge $(u \to v)$ disappears. Clearly, in this iteration the edge $(u \to v)$ appeared in the augmenting path π. In fact, this edge was fully utilized; namely, $c_f(\pi) = c_f(uv)$, where f is the flow in the beginning of the iteration when it disappeared. We continue running **Edmonds-Karp** till $(u \to v)$ "magically" reappears. This means that in the iteration before $(u \to v)$ reappeared in the residual graph, the algorithm handled an augmenting path σ that contained the edge $(v \to u)$. Let g be the flow used to compute σ. We have, by the monotonicity of $\delta(\cdot)$ [i.e., Lemma 13.3.5 below], that

$$\delta_g(u) = \delta_g(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$$

as **Edmonds-Karp** is always augmenting along the shortest path. Namely, the distance of s to u had increased by 2 between its disappearance and its (magical?) reappearance. Since $\delta_0(u) \geq 0$ and the maximum value of $\delta_?(u)$ is n, it follows that $(u \to v)$ can disappear and reappear at most $n/2$ times during the execution of the **Edmonds-Karp** algorithm. ■

(The careful reader would observe that in fact $\delta_?(u)$ might become infinity at some point during the algorithm execution (i.e., u is no longer reachable from s). If so, by monotonicity, the edge $(u \to v)$ would never appear again, in the residual graph, in any future iteration of the algorithm.

Observation 13.3.3 *Every time we add an augmenting path during the execution of the* **Edmonds-Karp** *algorithm, at least one edge disappears from the residual graph* $\mathsf{G}_?$. *Indeed, every edge that realizes the residual capacity of the augmenting path will disappear once we push the maximum possible flow along this path.*

Lemma 13.3.4 *The* **Edmonds-Karp** *algorithm handles at most* $O(nm)$ *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: Every edge might disappear at most $n/2$ times during **Edmonds-Karp** execution, by Lemma 13.3.2. Thus, there are at most $nm/2$ edge disappearances during the execution of the **Edmonds-Karp** algorithm. At each iteration, we perform path augmentation, and at least one edge disappears along it from the residual graph. Thus, the **Edmonds-Karp** algorithm perform at most $O(mn)$ iterations.

Performing a single iteration of the algorithm boils down to computing an Augmenting path. Computing such a path takes $O(m)$ time as we have to perform BFS to find the augmenting path. It follows, that the overall running time of the algorithm is $O\left(nm^2\right)$. ■

We still need to prove the aforementioned monotonicity property. (This is the only part in our discussion of network flow where the argument gets a bit tedious. So bear with us, after all, you are going to double your salary here.)

Lemma 13.3.5 *If the* **Edmonds-Karp** *algorithm is run on a flow network* $\mathsf{G} = (V, E)$ *with source* s *and sink* t, *then for all vertices* $v \in V \setminus \{s, t\}$, *the shortest path distance* $\delta_f(v)$ *in the residual network* G_f *increases monotonically with each flow augmentation.*

Proof: Assume, for the sake of contradiction, that this is false. Consider the flow just after the first iteration when this claim failed. Let f denote the flow before this (fatal) iteration was performed, and let g be the flow after.

Let v be the vertex such that $\delta_g(v)$ is minimal, among all vertices for which the monotonicity fails. Formally, this is the vertex v where $\delta_g(v)$ is minimal and $\delta_g(v) < \delta_f(v)$.

Let $\pi = s \to \cdots \to u \to v$ be the shortest path in G_g from s to v. Clearly, $(u \to v) \in E\left(G_g\right)$, and thus $\delta_g(u) = \delta_g(v) - 1$.

By the choice of v it must be that $\delta_g(u) \geq \delta_f(u)$, since otherwise the monotonicity property fails for u, and u is closer to s than v in G_g. This contradicts our choice of v as being the closest vertex to s that fails the monotonicity property. There are now two possibilities:

(i) If $(u \to v) \in E\left(G_f\right)$ then

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

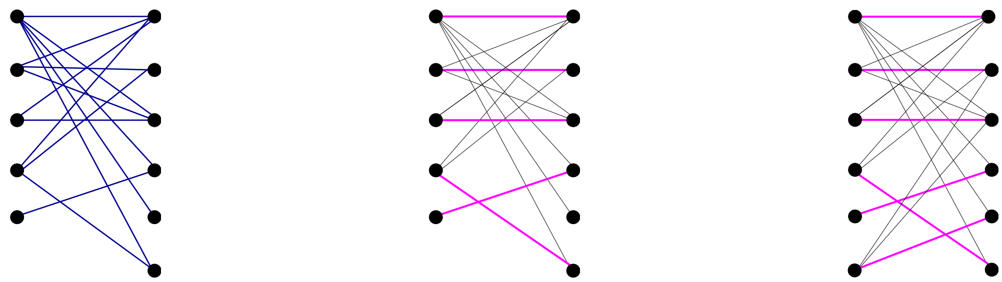


Figure 13.2: (i) A bipartite graph. (ii) A maximum matching in this graph. (iii) A perfect matching (in a different graph).

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

(ii) If $(u \to v)$ is not in $E\left(G_f\right)$ then the augmenting path π used in computing g from f contains the edge $(v \to u)$. Indeed, the edge $(u \to v)$ reappeared in the residual graph G_g (while not being present in G_f). The only way this can happens is if the augmenting path π pushed a flow in the other direction on the edge $(u \to v)$. Namely, $(v \to u) \in \pi$. However, the algorithm always augment along the shortest path. Thus, since by assumption $\delta_g(v) < \delta_f(v)$, we have

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

by the definition of u.

Thus, $\delta_f(u) > \delta_g(u)$ (i.e., the monotonicity property fails for u) and $\delta_g(u) < \delta_g(v)$. A contradiction to the choice of v. ■

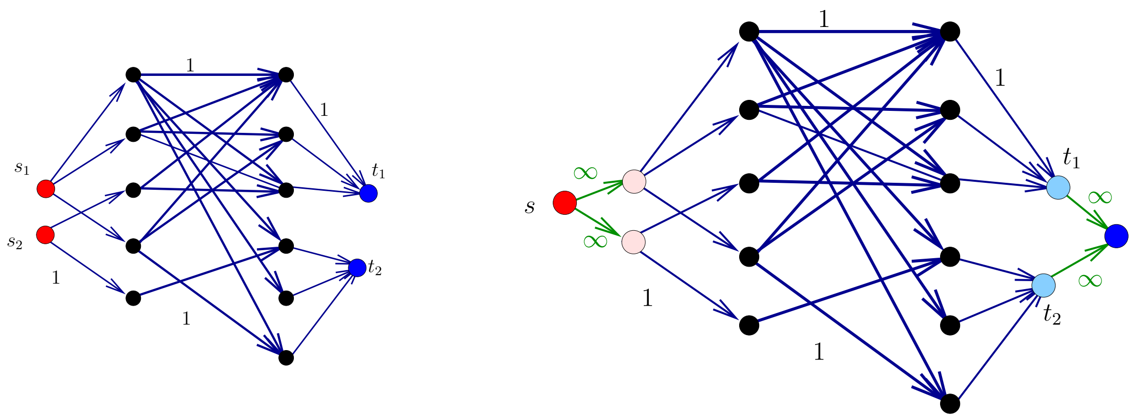
13.4 Applications and extensions for Network Flow

13.4.1 Maximum Bipartite Matching

Definition 13.4.1 For an undirected graph $\mathsf{G} = (V, E)$ a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. A *maximum matching* is a matching M such that for any matching M' we have $|M| \geq |M'|$.

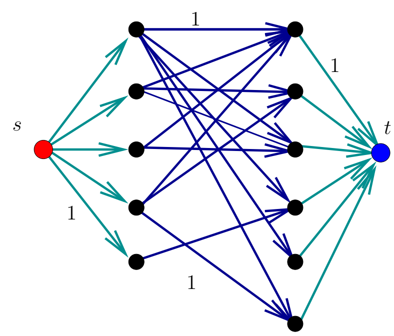
A matching is *perfect* if it involves all vertices. See Figure 13.2 for examples of these definitions.

Theorem 13.4.2 *One can compute maximum bipartite matching using network flow in $O(nm^2)$ time, for a bipartite graph with n vertices and m edges.*



(i) (ii)

Figure 13.3: (i) A flow network with several sources and sinks, and (ii) an equivalent flow network with a single source and sink.



Proof: Given a bipartite graph G, we create a new graph with a new source on the left side and sink on the right, as depicted. Direct all edges from left to right and set the capacity of all edges to 1. Let H be the resulting flow network. It is now easy to verify that by the Integrality theorem, a flow in H is either 0 or one on every edge, and thus a flow of value k in H is just a collection of k vertex disjoint paths between s and t in G, which corresponds to a matching in G of size k.

Similarly, given a matching of size k in G, it can be easily interpreted as realizing a flow in H of size k. Thus, computing a maximum flow in H results in computing a maximum matching in G. The running time of the algorithm is $O(nm^2)$. ■

13.4.2 Extension: Multiple Sources and Sinks

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

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. See Figure 13.3. Clearly, computing flow in both networks in equivalent.