CS 6901 (Applied Algorithms) – Lecture 14

Antonina Kolokolova[*]

November 4, 2014

1 Network flows

1.1 Motivating examples

A matching in a bipartite (undirected) graph G is a set of edges such that each vertex has in at most one edge in the matching. A matching is *maximal* if it has at least as many edges as any other matching in the graph; it is *perfect*, on a graph with n vertices on each side, if every vertex is an endpoint of exactly one edge included in the matching. This is the same meaning of the word "matching" as in the "stable matching"; however, there are no rankings, but also no edges between some pairs of vertices on different sides.

The first example we consider will be the following problem: given a bipartite graph G, find a maximal matching. Alternatively, we can ask, given G, whether there exists a perfect matching, and if not, what is the maximal matching in this graph. For example, in a graph with vertices a, b, c on one side and x, y, z on the other, with edges $(a, y), (b, z), (c, y)(b, x)$ the maximal matchings are of size 2: for example, $(b, z), (c, y)$ is such a matching. However, if we change (b, x) to (a, x), we can obtain a perfect matching $(a, x), (b, z), (c, y)$.

The second problem sounds very different, but we will approach it using some of the same tools.

Imagine a mining company that is constructing an open-pit mine for some mineral deposit. Say they mapped the deposit, and they know, for each cubic meter of soil, what it's value would be, and also what is the cost of taking out this cubic meter provided by then there is nothing above it (assuming they cannot dig horizontally; there could be different ways to

[*]This set of notes uses a variety of sources, in particular some material from Kleinberg-Tardos book and notes from University of Toronto CSC 364

describe "above", too: for simplicity, let's just say the cubic meter of earth directly above needs to be taken out). They need to decide what to dig out, and what to leave to maximize profits.

We will approach both of these problems using an algorithm design paradigm called *network flow.*

1.2 Flow networks and their properties

Consider a weighted directed graphs, with all weights (here called capacities) all non-negative real numbers, and two vertices marked s and t. Such a graph (G, c, s, t) is called a *flow network* (though a name "capacity network" or "capacitated network" could have been more correct). An intuition for such definition is that in many real-world problems we are dealing with a system of channels (roads, pipes, etc), where each channel has a capacity, and the goal is to get as much "stuff" through the system from the source s to the target t as possible. In real life applications, of course, there can be multiple sources and targets, but here we will so far simplify to have only one s and one t. For example, the goal could be to route traffic through the network where every link has a fixed bandwidth, or get a fleet of trucks from one location to another, where different roads could have different capacities.

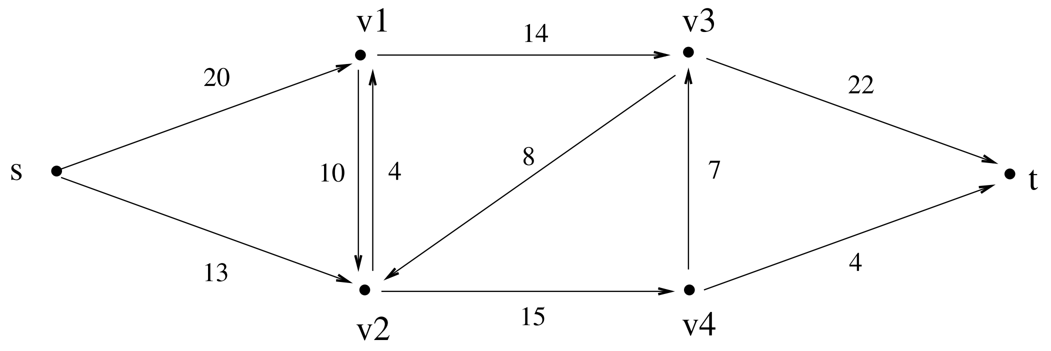
With this intuition in mind, define a flow with respect to this graph as a function $f\colon E \to \mathbb{R}^+$ satisfying several properties:

1) (Capacity constraints) $\forall e \in E, f(e) \leq c(e)$. That is, an edge can only have flow up to its capacity.
2) (Flow conservation) $\forall v \in V - \{s,t\}, \Sigma_{u \in V} f((u,v)) = \Sigma_{u \in V} f((v,u))$. That is, for every intermediate (not a source and not the target) vertex in the graph, as much stuff that flows in will flow out.

We will use the notation $f(G, c, s, t)$ (or simply $f(G)$) to mean the total flow on the flow network (G, c, s, t), and define it as $f(G, c, s, t) = \Sigma_{u \in V} f((s,v))$, that is, sum of the flows on all edges exiting the source s. You will see later that it is the same as defining it as sum of flows on all edges into t

Example 1 *The following is an example of a flow network. Only edges with positive capacities are shown; edges with capacity 0 are omitted from the diagram. Each edge is labelled with its capacity. For each edge (u,v) that is shown, the edge (v,u) is also assumed to be present; if the edge (v,u) is not shown, then it has capacity* 0.

FLOW NETWORK $\mathcal{F}$:

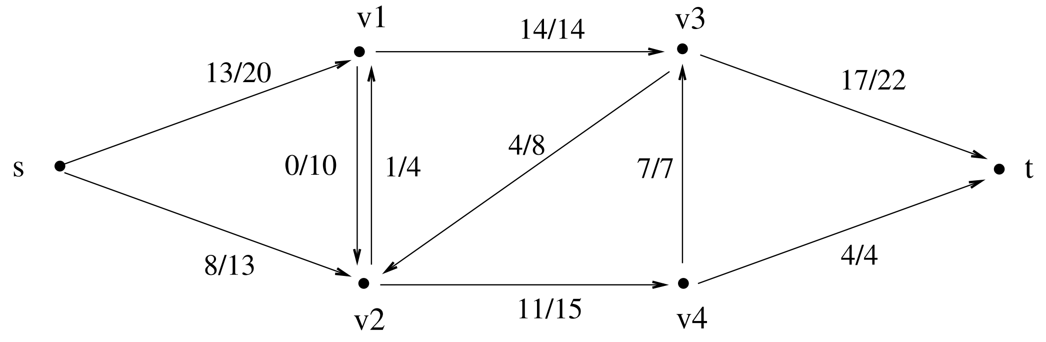


The following shows the above example of a flow network, together with a flow. The notation x/y on an edge (u,v) means

x is the flow ($x = f(u,v)$)
y is the capacity ($y = c(u,v)$)

Only flows on edges of positive capacity are shown. In this example we have $|f| = 13{+}8 = 21$.

FLOW NETWORK $\mathcal{F}$ WITH A FLOW f:



1.3 Residual Networks

Let $\mathcal{F}$ be a flow network, f a flow. For any $(u,v) \in E$, the *residual capacity* of (u,v) induced by f is

$$c_f(u,v) = c(u,v) - f(u,v) + f(v,u) \geq 0.$$

The *residual graph* of $\mathcal{F}$ induced by f is

$$G_f = (V, E_f)$$

where

$$E_f = \{(u,v) \in E \mid c_f(u,v) > 0\}.$$

The flow f also gives rise to the residual flow network $\mathcal{F}_f = (G, c_f, s, t)$.

The residual network $\mathcal{F}_f$ is itself a flow network with capacities c_f, and any flow in $\mathcal{F}_f$ is also a flow in $\mathcal{F}$.

Note that if we have a flow f in a network and a flow f_0 in the residual network, then f_0 can be added to f to obtain an improved flow in the original network. This will be a technique we will use to continuously improve flows until we have a maximum possible flow.

1.4 Augmenting Paths

Given a flow network $\mathcal{F} = (G, c, s, t)$ and a flow f, an *augmenting path* π is a simple path (that is, a path where no vertex repeats) from s to t in the residual graph, G_f; note that every edge in G_f has positive capacity. Equivalently, an augmenting path is a simple path from s to t in G consisting only of edges of positive residual capacity. We will use an augmenting path to create a flow f_0 of positive value in $\mathcal{F}_f$, and then add this to f as in the above lemma, in order to create the flow $f' = f + f_0$ of value bigger than f.

The maximum amount of net flow we can ship along the edges of an augmenting path π is called the *residual capacity* of π. We denote it by $c_f(\pi)$; because π is augmenting, $c_f(\pi)$ is guaranteed to be positive.

$$c_f(\pi) = \min\{c_f(u,v) \mid (u,v) \text{ is on } \pi\} > 0.$$

Lemma 1 *Fix flow network $\mathcal{F} = (G, c, s, t)$, flow f, augmenting path π, and define $f_\pi : E \to \mathbb{R}^+$:*

$$f_\pi(u,v) = \begin{cases} c_f(\pi) & \textit{if } (u,v) \textit{ is on } \pi \\ 0 & \textit{otherwise} \end{cases}$$

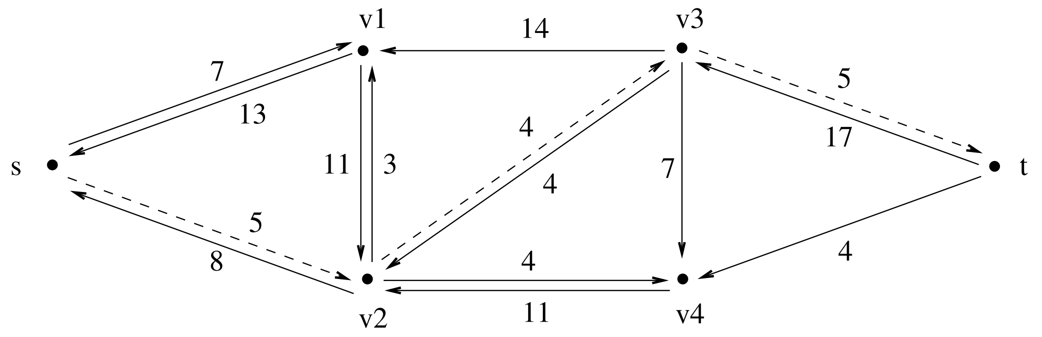
Then f_π is a flow in $\mathcal{F}_f$, and $|f_\pi| = c_f(\pi) > 0$.

Corollary 1 *Fix flow network $\mathcal{F} = (G, c, s, t)$, flow f, augmenting path π, and let f_π be defined as above. Let $f' = f + f_\pi$. Then f' is a flow in $\mathcal{F}$, and*

$$|f'| = |f| + |f_\pi| > |f|.$$

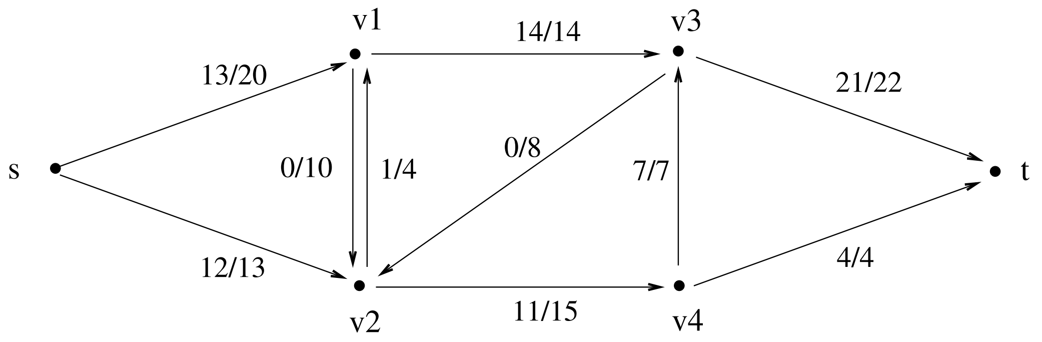
Example: Continuing the previous example, the following diagram shows the residual graph G_f consisting of edges with positive residual capacity. The residual capacity of each edge is also shown. An augmenting path π is indicated by $- - -$. We have $c_f(\pi) = 4$.

THE RESIDUAL GRAPH G_f WITH AUGMENTING PATH π:



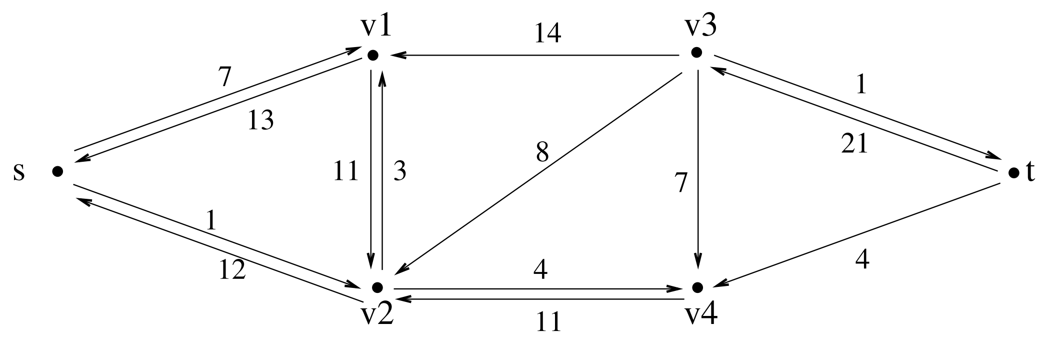
The following diagram shows the network $\mathcal{F}$ with flow $f' = f + f_\pi$. We have $|f'| = |f| + |f_\pi| = 21 + c_f(\pi) = 21 + 4 = 25$.

FLOW NETWORK $\mathcal{F}$ WITH FLOW f':



After creating the improved flow f', it is natural to try the same trick again and look for an augmenting path with respect to f'. That is, we consider the new residual graph $G_{f'}$ and look for a path from s to t. We see however, that no such path exists.

THE RESIDUAL GRAPH $G_{f'}$:



All of the above suggests the famous *Ford-Fulkerson* algorithm for network flow. The algorithm begins by initializing the flow f to the all-0 flow, that is, the flow that is 0 along every edge. The algorithm then continually improves f by searching for an augmenting path π, and using this path to improve f, as in the previous lemma. The algorithm halts when there is no longer any augmenting path.

```
FORD-FULKERSON($G, c, s, t$)

Initialize flow $f$ to the all-0 flow
WHILE there exists an augmenting path in $G_f$ DO
    choose an augmenting path $\pi$
    $f \leftarrow f + f_\pi$
end WHILE
```

There are a number of obvious questions to ask about this algorithm. Firstly, how are we supposed to search for augmenting paths? That is, how do we look for a path from s to t in the graph G_f? There are many algorithms we could use. However, since all we want to do is find a path between two points in an unweighted, directed graph, two of the simplest and fastest algorithms we can use are "depth-first search" or "breadth-first" search. Each of these algorithms runs in time linear in the size of the graph, that is, linear in the number of edges in the graph.

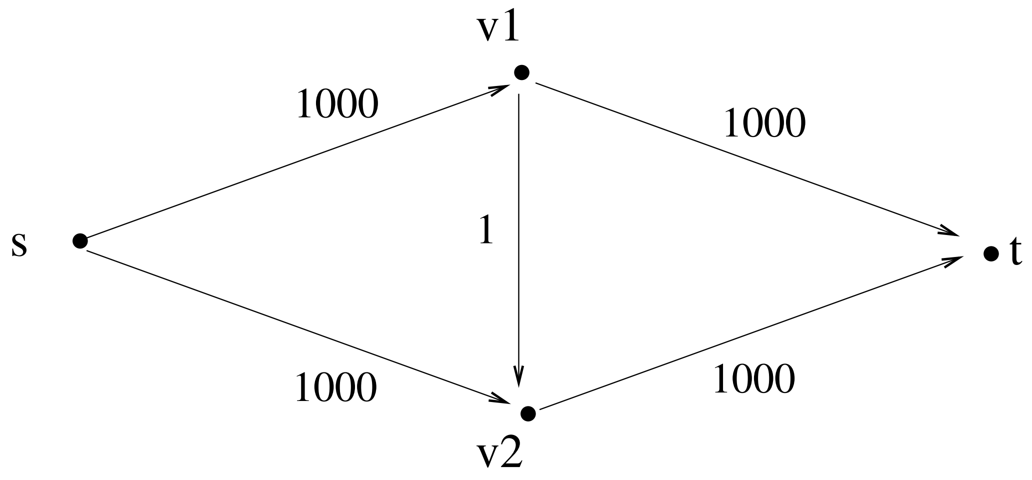
The next question is, is the algorithm guaranteed to halt? The answer to this, remarkably, is *NO*. If we do not constrain how the algorithm searches for augmenting paths, then there are examples where it can run forever. These examples are complicated and use irrational capacities, and we will not show one here.

What if the capacities are all integers? Then it is clear that the algorithm increases the flow by at least 1 each time through the loop, and so it will eventually halt. (Exercise: fill in the details of this argument; give a similar argument in case the capacities are only guaranteed to be rational numbers.) However, this may still take a very long time.

As an example, consider the following flow network with only 4 vertices. If we are lucky (or careful), the algorithm will choose $[s, v_1, t]$ for the first augmenting path, creating a flow with value 1000; it will then have no choice but to choose $[s, v_2, t]$ for the next augmenting path, and it will be halt, having created a flow with value 2000. However, the algorithm *may* choose $[s, v_1, v_2, t]$ as its first augmenting path, creating a flow with value 1; it *may* then choose $[s, v_2, v_1, t]$ as the next augmenting path, creating a flow with value 2; continuing in this way, it *may* go 2000 times through the loop before it eventually halts.

Thus, we can bound the running time by $O(mC)$, where $C = \Sigma_v c(s, v)$.

EXAMPLE OF A BAD FLOW NETWORK FOR FORD-FULKERSON:



The Edmonds-Karp version of this algorithm looks for a path in G_f using *breadth-first search*. This finds a path that contains as few edges as possible. We call this algorithm FF-EK:

```
FF-EK($G, c, s, t$)

Initialize flow $f$ to the all-0 flow
WHILE there exists an augmenting path in $G_f$ DO
    choose an augmenting path $\pi$ using breadth-first search in $G_f$
    $f \leftarrow f + f_\pi$
end WHILE
```

Let us assume (without loss of generality) that $|E| \geq |V|$. Then breadth-first search finds an augmenting path (if there is one) in time $O(|E|)$. Using an augmenting path to improve the flow takes time $O(|E|)$, so each execution of the main loop runs in time $O(|E|)$.

It is a difficult theorem (that we will not prove here) that the main loop of FF-EK will be executed at most $O(|V||E|)$ times. Thus, FF-EK halts in time $O(|V||E|^2)$. Hence, this is a polynomial time algorithm. A huge amount of research has been done in this area, and even better algorithms have been found. One of the fastest has running time $O(|V|^3)$.

Example: Consider the previous example of flow network $\mathcal{F}$ with flow f'. We know that $|f'| = 25$, and so the flow across every cut will be 25, and the capacity of every cut will be greater than or equal to 25. We have seen that there is no augmenting path, so the above theorem tells us that there must be a cut of capacity 25. It even tells us how to find such a cut: let S be the set of nodes reachable from s in $G_{f'}$. In fact, it is easy to check that if we choose $S = \{s, v_1, v_2, v_4\}$ and $T = \{v_3, t\}$, then $c(S, T) = 14 + 7 + 4 = 25$.

1.5 Solving bipartite matching problem

Now recall our first motivating example: the bipartite matching problem.

It turns out that there is a way to convert such a matching problem into a flow problem. First we add two vertices to create $V' = V \cup \{s, t\}$. We add edges from s to each vertex in L, and edges from each vertex in R to t; all of these edges (including the original edges in E) are assigned capacity 1. Lastly, we add the the reverse of all these edges, with capacity 0. In this way we form the flow network $\mathcal{F} = (G', c, s, t)$, $G' = (V', E')$. Consider integer flows in $\mathcal{F}$, that is, flows that take on integer values on every edge; for each edge of capacity 1, the flow on it must be either 1 or 0 (since its reverse edge has capacity 0). It is easy to see that Ford-Fulkerson, when applied to a network with only integer capacities, will always yield an integer (maximum) flow. The following two lemmas show that an integer flow f can be used to construct a matching of size $|f|$, and that a matching M can be used to construct a flow of value $|M|$. This will allow us to use Ford-Fulkerson to compute a maximum flow in polynomial time.

Lemma 2 *Let $G = (V, E)$ and $\mathcal{F} = (G', c, s, t)$ be as above, and let f be an integer flow in $\mathcal{F}$. Then there is a matching M in G such that $|M| = |f|$.*

Proof:
Let f be an integer flow. Let $M = \{(u, v) \in L \times R \,|\, f(u, v) = 1.\}$.

To see why M is a matching, imagine that $(u, v_1), (u, v_2) \in M$ for some $v_1 \neq v_2$; since u has only one edge of positive capacity (namely 1) coming into it, we would have $\sum_{v \in N(u)} f(u, v) \geq 1$, contradicting flow conservation. (A similar argument shows that no two edges in M can share a right endpoint.)

We now show that $|M| = |f|$. Recall that $|f|$ is equal to the total flow coming out of s. So we must have $|f|$ distinct vertices $u_1, u_2, \ldots, u_{|f|}$ such that
$f(s, u_1) = 1, f(s, u_2) = 1, \ldots, f(s, u_{|f|}) = 1$. So in order for flow conservation to hold, for each u_i we must have some $v_i \in R$ such that $f(u_i, v_i) = 1$. So $(u_i, v_i) \in M$ for each i, and we have $|M| = |f|$. $\square$

Lemma 3 *Let $G = (V, E)$ and $\mathcal{F} = (G', c, s, t)$ be as above, and let M be a matching in G. Then there exists a flow f in $\mathcal{F}$ with $|f| = |M|$.*

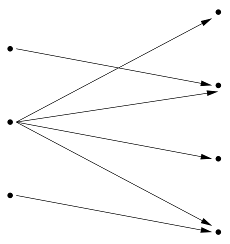
Proof:
Let $M = \{(u_1, v_1), (u_2, v_2), \ldots, (u_{|M|}, v_{|M|}) \subseteq L \times R\}$ be a matching. Define f by $f(s, u_i) = 1$, $f(u_i, v_i) = 1$, $f(v_i, t) = 1$ for each i, and $f(e) = 0$ for every other edge $e \in E'$. It is easy to check that f is a flow, and that $|f| = |M|$ (exercise). $\square$

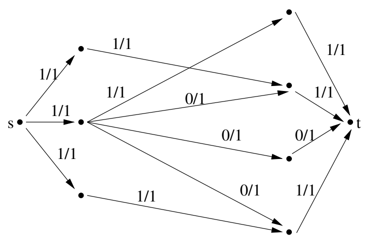
We now see how to use Ford-Fulkerson to find a maximum matching in $G = (V, E)$. Assume, without loss of generality, that $|V| \leq |E|$. Note that $|V'| \in O(|V|)$ and $|E'| \in O(|E|)$.

We first construct $\mathcal{F} = (G', c, s, t)$ as above; this takes time $O(|E|)$. We then perform the Ford-Fulkerson algorithm to create a maximum flow f. We observe that this algorithm, no matter how we find augmenting paths, will increase the flow value by exactly 1 each time, and hence will execute its main loop at most $|V|$ times. If we use an $O(|E|)$ time algorithm to search for augmenting paths, then each execution of the loop will take time $O(|E|)$. So the total time of the Ford-Fulkerson algorithm here is $O(|V||E|)$. Lastly, we use the integer flow f to create a matching M in G such that $|M| = |f|$; this takes time $O(|E|)$. The last lemma above tells us that since f is a maximum flow, M must be a maximum matching.

So the entire maximum matching algorithm runs in time $O(|V||E|)$. This is a polynomial time algorithm. Faster algorithms have also been found. For example, there is an algorithm for this problem that runs in time $O(\sqrt{|V|}|E|)$.

Example: The following is an example of a (directed) bipartite graph G. The next figure shows the network $\mathcal{F}$ derived from G, together with a maximum flow f in $\mathcal{F}$. The last figure shows the maximum matching M obtained from f.




s
t
1/1
1/1
1/1
1/1
1/1
1/1
0/1
0/1
0/1
0/1
1/1
1/1
1/1