Scribe: Chuanqi Shen (2015), Virginia Williams, Luke Johnston (2016), G. Valiant (2017), M. Wootters (2017)
Date: November 29, 2017
Adapted from Virginia Williams' Lecture notes.

1 History of Flows and Cuts

Today we will continue the theme of studying cuts in graphs. In particular we will be studying a very interesting problem called the max flow problem. Before that, a short history lesson. During the Cold War, the US Air Force at that time was very interested in the Soviet train networks. In reports that were declassified in 1999, it was revealed that the Air Force collected enough information about the train network that they were able to determine how resources were shipped from the Soviet Union to Europe. The Air Force was very interested in determining how much resources can be transported from the Soviet Union to Europe, and what needed to be done to destroy this movement of resources. What this translates to is the min cut problem, i.e., cut the minimum number of train tracks so that nothing goes to Europe. Here, cutting an edge means dropping a bomb. Nowadays, however, there are much milder (but still important!) applications of this problem, for instance, understanding the flow of information through the Internet.

2 Formulation of the Maximum Flow Problem

You are given an input graph $G = (V, E)$, where the edges are directed. There is a function $c : E \rightarrow \mathbb{R}^+$ that defines the capacity of each edge. We also label two nodes, s and t in G, as the source and destination, respectively. The task is to output a *flow* of *maximum value*. We will shortly define what a *flow* is and what a flow of *maximum value* means.

A flow f is a function $f : E \rightarrow \mathbb{R}_0^+$ such that

1. (Capacity Constraint)
$$\forall (u, v) \in E, 0 \leq f(u, v) \leq c(u, v)$$
2. (Flow Conservation Constraint)
$$\forall v \in V \backslash \{s, t\}, \sum_{x \in N_{in}(v)} f(x, v) = \sum_{y \in N_{out}(v)} f(v, y)$$
Here $N_{in}(v)$ denotes the set of nodes with an edge that points to v and $N_{out}(v)$ denotes the set of nodes that v points to.

Suppose that there are no edges going into s and no edges coming out of t. From the above, you can verify yourself that $\sum_{x \in N_{out}(s)} f(s, x) = \sum_{y \in N_{in}(t)} f(y, t)$. We define the value $\sum_{x \in N_{out}(s)} f(s, x)$ to be the value of the flow f. We usually denote the value of a flow f as $|f|$. If there are edges going into s and out of t, then the value of f is
$$|f| = \sum_{x \in N_{out}(s)} f(s, x) - \sum_{y \in N_{in}(s)} f(y, s).$$
The max flow problem is to find some flow f such that $|f|$ is maximized.

Remark 1. In the analysis below we consider graphs with a single source s and a single sink t. However, if we need to work with a graph with multiple sources, we can do so by adding a new source node, and then adding edges with capacity infinity from it to each of the multiple sources. Similarly, if we want to have multiple sinks, we add a new sink node and add edges from the multiple sinks to that sink with capacity infinity.

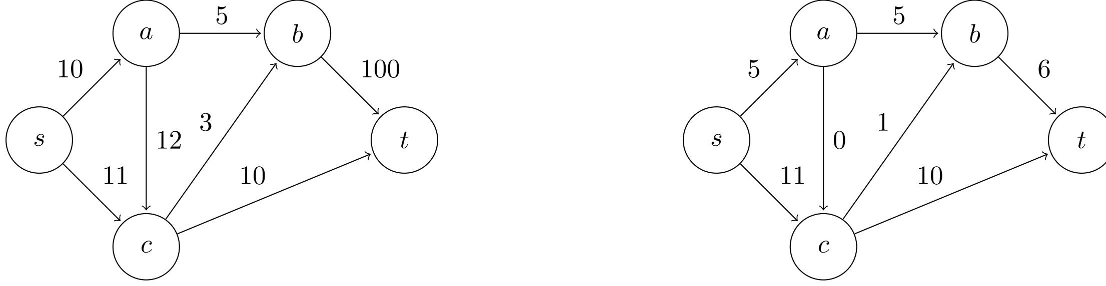

Figure 1: (Left) Graph G with edge capacities (Right) Graph G with a sample flow

3 Example

In Figure 1, we have a graph G and a sample flow f. Observe that the two constraints for a flow are satisfied. There can be multiple other flows possible that can satisfy the constraints. For our given flow, $|f| = 16$. The max flow for this graph is actually 18, as we will see shortly.

4 Formulation of the Minimum Cut Problem

Now, we give a formulation of the min cut problem defined for directed graphs with source and destination nodes s and t. (There is also a version of the Min-Cut problem without a source and sink node, though we won't discuss that now.)

An $s-t$ cut is a partition $V = S \bigcup T$ where S and T are disjoint and $s \in S, t \in T$, and the size/cost of an $s-t$ cut is

$$\|S, T\| = \sum_{x \in S, y \in T} c(x, y)$$

For our graph G shown above, if we set $S = \{s, a, c\}$ and $T = \{b, t\}$, then the cost of the cut is $c(a, b) + c(c, b) + c(c, t) = 5 + 3 + 10 = 18$. If we take another cut $S' = \{s, c\}, T' = \{a, b, t\}$, then $\|S', T'\| = c(s, a) + c(c, b) + c(c, t) = 10 + 3 + 10 = 23$. Note that we **do not** consider the edge $\{a, c\}$ as it is in the wrong direction (we only consider edges from S' to T').

5 The Max-Flow Min-Cut Theorem

Lemma 5.1. For any flow f and any $s-t$ cut (S, T) of G, we have $|f| \leq \|S, T\|$. In particular, the value of the max flow is at most the value of the min cut.

Proof.

$$\begin{aligned}
|f| &= \sum_{x \in N_{out}(s)} f(s, x) - \sum_{y \in N_{in}(s)} f(y, s) \\
&= \sum_{v \in S} \left(\sum_{x \in N_{out}(v)} f(v, x) - \sum_{y \in N_{in}(v)} f(y, v) \right) \text{ [by the Flow Conservation Constraint all added terms sum to 0]} \\
&= \sum_{v \in S} \left(\sum_{x \in N_{out}(v) \cap S} f(v, x) - \sum_{y \in N_{in}(v) \cap S} f(y, v) \right) + \sum_{v \in S} \left(\sum_{x \in N_{out}(v) \cap T} f(v, x) - \sum_{y \in N_{in}(v) \cap T} f(y, v) \right)
\end{aligned}$$

$$= \sum_{v \in S} \left(\sum_{x \in N_{out}(v) \cap T} f(v,x) - \sum_{y \in N_{in}(v) \cap T} f(y,v) \right) \text{ [first term sums to 0]}$$

$$\leq \sum_{v \in S, x \in T, x \in N_{out}(v)} f(v,x) \leq \sum_{v \in S, x \in T, x \in N_{out}(v)} c(v,x) = \|S,T\|$$

In the proof, $\sum_{v \in S} \left(\sum_{x \in N_{out}(v) \cap S} f(v,x) - \sum_{y \in N_{in}(v) \cap S} f(y,v) \right) = 0$ since we add and subtract the flow $f(u,v)$ for every $u, v \in S$ such that $(u,v) \in E$. □

We get the following consequence.

Corollary 5.1. *If we can find f and (S,T) such that $|f| = \|S,T\|$, then f is a max flow and (S,T) is a min cut.*

It turns out that we can always find such f and (S,T) for any graph.

Theorem 5.1 (Max-flow min-cut theorem). *For any graph G, source s and destination t, the value of the max flow is equal to the cost of the min cut.*

We will show this by coming up with an algorithm. The algorithm will take the graph G and some flow f that has already been constructed, and create a new graph that is called the residual graph. In this new graph, the algorithm will try to find a path from s to t. If no such path exists, we will show that the value of the flow we started with is the value of the maximum flow. If not, we show how to increase the value of our flow by pushing some flow on that path.

6 The Ford-Fulkerson Max-Flow Algorithm

We will make an assumption on our graph. The assumption can be removed, but it will make our lives easier. We will assume that for all $u, v \in V$, G does not have both edges (u,v) and (v,u) in E. We can make this condition hold by modifying the original graph in the following way. If $(u,v), (v,u) \in E$, we split the edge (u,v) to two edges (u,x) and (x,v), where x is a new node we introduce into the graph. This makes the number of nodes at most $m+n$.

Now, let f be a flow given to us. We will try to see if we can improve this flow. We will define the *residual capacity* $c_f : V \times V \to \mathbb{R}_0^+$ as follows.

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Basically, what this does is that, if there is any flow through the edge, you remove the flow from the capacity and add an edge in the opposite direction with the value of the flow. The reason we do this is because the flow we picked thus far might not be the correct flow, and this formulation allows us to undo changes that we have done. The residual capacity $c_f(u,v)$ represents how much flow we can send from u to v in addition to the flow f.

We define G_f to be a residual network defined with respect to f, where $V(G_f) = V(G)$ and $(u,v) \in E(G_f)$ if $c_f(u,v) > 0$. Figure 2 shows G with the residual edges.

We will show that, if there is a path from s to t in G_f, then f is not a max flow. If no such path exists, that f is a max flow.

Lemma 6.1. *If t is not reachable from s is G_f, then f is a maximum flow.*

Proof. Let S be the set of nodes reachable from s in G_f and $T = V \setminus S$. There are no edges in G_f from S to T since the nodes in T are not reachable from s. Note that (S,T) defines an $s-t$ cut. Now consider any $v \in S, w \in T$. We have $c_f(v,w) = 0$ since (v,w) is not an edge in G_f. There are three cases:

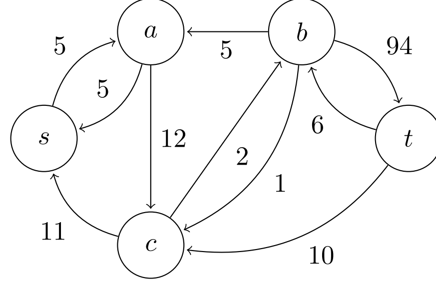

Figure 2: The residual network given the flow presented in Figure 1

1. If $(v, w) \in E$, then by definition $c_f(v, w) = c(v, w) - f(v, w) = 0 \implies c(v, w) = f(v, w)$.
2. If $(w, v) \in E$, then $c_f(v, w) = f(w, v) = 0$.
3. If $(v, w) \notin E$ and $(w, v) \notin E$, we can disregard (v, w) and (w, v) since they do not appear in any flow or cut.

Using this, and the proof in Lemma 5.1, we have

$$\begin{aligned}
|f| &= \sum_{v \in S} \left(\sum_{x \in N_{out}(v) \cap T} f(v, x) - \sum_{y \in N_{in}(v) \cap T} f(y, v) \right) \\
&= \sum_{v \in S} \sum_{x \in N_{out}(v) \cap T} f(v, x) \text{ [from case 2 the second term is 0]} \\
&= \sum_{v \in S} \sum_{x \in N_{out}(v) \cap T} c(v, x) \text{ [from case 1]} \\
&= \|S, T\|
\end{aligned}$$

Thus, we show that the flow is equal to the cut. From Corollary 5.1 we know that f is a maximum flow, and $\|S, T\|$ is a min cut. □

Lemma 6.2. *If G_f has a path from s to t, we can modify f to f' such that $|f| < |f'|$.*

Proof. Pick a path P from s to t in G_f, and consider the edge of minimum capacity on the path. Let that capacity be F. Then we can increase our flow by F. For each edge in P, if $c_f(v, w)$ is the right direction (i.e there is an edge $(v, w) \in E(G)$), then we can increase our flow on this edge by F. If $c_f(v, w)$ is in the opposite direction (i.e. $(w, v) \in E(G)$), then we can decrease the flow on this edge by F. In effect, we are "undoing" the flow on this edge. By doing so, we have increased our flow by F.

As an example, consider Figure 2 again. The path $s \to a \to c \to b \to t$ is a path with minimum capacity 2. Therefore, we can update our flow and push additional 2 units of flow, resulting in a flow of 18.

Formally, Let $s = x_0 \to x_1 \to \ldots \to x_k = t$ be a simple path P in G_f, and let $F = \min_i c_f(x_i, x_{i+1})$. Define a new flow f' where

$$f'(u, v) = \begin{cases} f(u, v) + F & \text{if } (u, v) \in P \\ f(u, v) - F & \text{if } (v, u) \in P \\ f(u, v) & \text{otherwise} \end{cases}$$

We now need to show that f' is a flow. The capacity constraints are satisfied because for every $(u, v) \in E$,

1. If $(u, v) \in P$, then $0 \le f(u, v) + F \le f(u, v) + c_f(u, v) = f(u, v) + c(u, v) - f(u, v) = c(u, v)$

2. If $(v, u) \in P$, then $f(u, v) - F \leq f(u, v) \leq c(u, v)$ and $f(u, v) - F \geq f(u, v) - c_f(v, u) = 0$.
3. Otherwise, $f(u, v)$ is from the original flow f.

The conservation constraints are also satisfied: Recall that P is a simple path. Thus, for every $v \in V \setminus \{s, t\}$, P uses 0 or two edges incident on v. If P uses 0 edges on v, then flow values of the edges incident on v have not changed when going from f to f'. Thus, suppose that P uses two edges (x, v) and (v, y) incident on v. Because in G_f some edges appear in the opposite direction compared to G, we need to consider a few cases.

1. (x, v) and (v, y) are both in the same direction (an edge into v and an edge out of v); the flow into v increases by F and the flow out of it also increases by F.
2. (x, v) and (v, y) are both in the opposite direction ($(v, x), (y, v) \in E$); the flow into v decreases by F and the flow out of it also decreases by F.
3. (x, v) is in the correct direction and (v, y) is in the opposite direction. Then the flow into V changes by $F - F = 0$.
4. (x, v) is in the opposite direction and (v, y) is in the correct direction. Then the flow out of V changes by $F - F = 0$.

Finally, note that we increase our flow by F. Consider the edge (s, x_1) in P. If $(s, x_1) \in E$, the flow out of s increases by F. If $(x_1, s) \in E$, the flow into s decreases by F. By our definition of G_f, it must be that $F > 0$, and we get that $|f'| > |f|$. □

From this, we can construct an algorithm to find the maximum flow. Starting with some arbitrary flow of the graph, construct the residual network, and check if there is a path from s to t. If there is a path, update the flow, construct the new residual graph and repeat. Otherwise, we have found the max flow.

A path from s to t in the residual graph is called an *augmenting* path, and pushing flow through it to modify the current flow is referred to as *augmenting* along the path.

Algorithm 1: maxflow(G, s, t)

```
f ← all zeroes flow;
G_f ← G;
while t is reachable from s in G_f (check using DFS) do
    P ← path in G_f from s to t;
    F ← min capacity on P;
    f ← f' as defined in Lemma 6.2;
    Update G_f to correspond to new flow;
return f;
```

The run time of this algorithm is bounded by the number of times we update our flow. If the edge capacities are all integers, we can increase the flow by at least 1 each time we update our flow. Therefore, the runtime is $O(|f|m)$ where $|f|$ is the value of the max flow. If we have rational edge capacities, then we can multiply all edge capacities by a factor to make them all integers. However, the runtime blows up by that factor as well. If we have irrational edge capacities, then the algorithm is no longer guaranteed to terminate. So we have a problem.

We will save the day in the next sections. Algorithm 1 is called the Ford-Fulkerson method. It is actually part of a family of algorithms that depend on how the path P between s and t in G_f is selected. One can obtain P via DFS, BFS, or any other method for selecting paths. It turns out that two methods work particularly well: the shortest path method and the fattest path method. The shortest path method is known as the Edmonds-Karp algorithm or Dinic's algorithm. In this class you are only required to know the above implementation and the running time of the Edmonds-Karp algorithm, but we have included the details of these two improvements for the interested student.

The fattest path method This method finds a path between s and t that maximizes $\min_{e\in P} c_f(e)$ among all $s-t$ paths P. Finding such a path can be done in $O(m+n)$ time by a clever mix of linear time median-finding and DFS.

The shortest path method (the Edmonds-Karp algorithm/Dinic's algorithm) This method picks the path between s and t using BFS, thus picking a path that minimizes the number of edges. Finding such a path also runs in $O(m)$ time: BFS takes $O(m+n)$ to explore the whole graph, but since we only care about the vertices reachable from s this is $O(m)$ time. The total run time is $O(nm^2)$.

Since both methods of selecting a path run in linear time, the main question becomes, how many iterations does Ford-Fulkerson perform? We will answer these questions below in the next section.

7 Running time of various implementations of Ford-Fulkerson

NOTE: we did not discuss the details of this section in class, but it's in the notes for the interested reader.

7.1 The fattest path version of Ford-Fulkerson

In this section we will show that the fattest path method results in a runtime of $O(m(m+n)\log|f|)$ when run on a graph with integer capacities. Thus, when rational capacities are converted to integers by multiplying by N, we get a runtime of $O(m(m+n)(\log|f|+\log N))$ for rational capacities. Thus the effect of large N is mitigated. This method does not solve the issues when the capacities can be irrational.

To show the runtime, we prove a main claim that states that after each iteration of the algorithm, the maximum flow value in G_f goes down by a factor of $(1-1/m)$. This max flow value starts as $|f|$ since $G_f = G$ in the beginning of the algorithm, and ends at 0 as in the end s and t are disconnected.

Claim 1 (Main). *Let f' be the max flow in G_f. Then after one iteration of Ford-Fulkerson on G_f, the max flow value becomes $\leq |f'|(1-1/m)$..*

Proof. Let P be the fattest path from s to t in G_f. Let $F = \min_{e\in P} c_f(e)$. Let S be the nodes reachable from s in G_f via paths composed of edges with residual capacities $> F$. Thus, any edge (x,y) of G_f with $x\in S$, $y\notin S$ must have $c_f(x,y)\leq F$. In particular, this means that the size of the cut between S and $V\S$ is $\sum_{x\in S, y\in V\setminus S} c_f(x,y)\leq mF$. Thus, the size of the min s-t cut in G_f is at most mF.

By the max-flow-min-cut theorem from last lecture, the size of the min $s-t$ cut is at least the size of the max-flow $|f'|$ in G_f, and so $|f'|\leq mF$. Thus $F\geq |f'|/m$.

Now, when we augment (push flow) along P, the flow in G increases by F, while the flow in G_f decreases by F. Thus, the new flow in G_f after augmenting along P becomes $|f'|-F\leq |f'|(1-1/m)$. □

Now that the main claim has been proven, we can conclude with a discussion of the runtime. Consider how the max flow value in G_f evolves after t iterations. It starts as $|f|$ (where f is the max flow in G) and then after t iterations is

$$\leq |f|\,(1-1/m)^t\,.$$

If $t = m\ln|f|$, we het that the max flow value in G_f is

$$\leq |f|\,((1-1/m)^m)^{\ln|f|} < |f|(1/e)^{\ln|f|} = 1.$$

Since all the capacities are integers, all the residual capacities are also integers, and so the max flow value in G_f is an integer. Since it is < 1, it must be 0. Hence after $m\ln|f|$ iterations, the max flow value in G_f is zero, s and t are disconnected and the computed flow in G is maximum. The runtime is $O((m+n)m\log|f|)$.

7.2 The shortest path version of Ford-Fulkerson

Here we analyze running Ford-Fulkerson using BFS to find a path between s and t in G_f.

With each augmentation along a path P in G_f, at least one edge is removed from G_f, namely the edge with residual capacity $F = \min_{e \in P} c_f(e)$. The main claim that we need to prove the runtime is that the number of times an edge can be removed from G_f is small. Since each iteration of the algorithm causes at least one removal, the main lemma will show that the number of iterations is small and hence the runtime is small as well.

Claim 2 (Main). *Fix any (u, v) that is ever an edge in G_f. Then the number of times that (u, v) can disappear from G_f is at most $n/2$.*

Once this claim is proven, we would get that the total number of edge disappearances is at most $mn/2$ and hence the number of iterations of the algorithm is also $\leq mn/2$. Because of this, the algorithm runtime is $O((m + n)mn)$.

To prove the claim, we will need a useful lemma (see below) that shows that as G_f evolves through the iterations, for any v, the (unweighted) distance from s to v in G_f cannot go down. Let's begin with some notation. Let G_f^i be the residual network after the ith iteration of the algorithm; $G_f^0 = G$. For a vertex v, let $d_i(v)$ be the (unweighted) distance from s to v in G_f^i.

Lemma 7.1. For all $i \geq 1$, and all $v \in V$, $d_{i-1}(v) \leq d_i(v)$.

Proof. Fix i. We will prove the statement for i by induction on $d = d_i(v)$.

The inductive hypothesis is that for all d and all v with $d_i(v) = d$, $d_{i-1}(v) \leq d_i(v)$. The base case is $d = 0$. We note that if $d_i(v) = 0$, then $v = s$ since we view G_f^i as an unweighted graph. But then we also have $d_{i-1}(s) = 0 \leq d_i(s)$.

For the induction, let's assume that the inductive hypothesis holds for $d - 1$, i.e. that for all x with $d_i(x) = d - 1$, $d_{i-1}(x) \leq d_i(x)$. We want to show that for all v with $d_i(v) = d$, we also have $d_{i-1}(v) \leq d_i(v)$.

Consider some v with $d_i(v) = d$. Let u be the node just before v on a shortest $s - v$ path in G_f^i. Then, $d_i(u) = d_i(v) - 1 = d - 1$ and the inductive hypothesis applies to it so that $d_{i-1}(u) \leq d_i(u)$.

We consider two cases.

Case 1: $(u, v) \in G_f^{i-1}$. Then, by the triangle inequality in G_f^{i-1}, we have that $d_{i-1}(v) \leq d_{i-1}(u) + 1$. Since $d_{i-1}(u) \leq d_i(u)$, we get that

$$d_{i-1}(v) \leq d_i(u) + 1 = (d_i(v) - 1) + 1 = d_i(v).$$

Case 2: $(u, v) \notin G_f^{i-1}$. Then, since $(u, v) \in G_f^i$, we must have that (v, u) was on the $(i-1)$st augmenting path. Hence $d_{i-1}(u) = d_{i-1}(v) + 1$. Hence:

$$d_{i-1}(v) = d_{i-1}(u) - 1 \leq d_i(u) - 1 = d_i(v) - 2 \leq d_i(v).$$

In both cases $d_{i-1}(v) \leq d_i(v)$ and the induction is complete. □

Now we are ready to prove the main claim.

Fix some (u, v) that is an edge in G_f at some point. Let's consider two consecutive disappearences of (u, v). Suppose that $(u, v) \in G_i$ but $(u, v) \notin G_{i+1}$. If after this disappearance (u, v) had another one later on, then at some point (u, v) must have appeared in G_f again. Let j be the first iteration after i so that the jth augmenting path made (u, v) appear in G_f^{j+1}.

Because $(u, v) \in G_f^i$ but $(u, v) \notin G_f^{i+1}$, (u, v) must have been in the ith augmenting path P_i.
Because $(u, v) \notin G_f^j$ but $(u, v) \in G_f^{j+1}$, (v, u) must have been in the jth augmenting path P_j.

From this we obtain that $d_i(v) = d_i(u) + 1$ and $d_j(u) = d_j(v) + 1$. Using the fact that $j > i$ and the key lemma from above we obtain

$$d_j(u) = d_j(v) + 1 \geq d_i(v) + 1 = d_i(u) + 2.$$

Thus, between (u, v)'s disappearance and its next reappearance, the distance from s to u increased by $+2$. Hence between any two consective disappearances the distance to u increases by ≥ 2. The distance starts as ≥ 0 and can be $\leq n-1$ before becoming ∞. Thus the total number of disappearances of (u, v) is $\leq n/2$. This completes the proof of the main claim and the proof of the runtime.

8 Applications

We wrap up by talking about some applications of the Ford-Fulkerson algorithm.

8.1 Bipartite perfect matching

Let $G = (V, E)$ be an undirected, unweighted *bipartite graph*: the set of vertices is partitioned into V_1 and V_2 so that there are no edges with two endpoints entirely in V_1 or entirely in V_2. A matching in G is a collection of edges, no two of which share an end point. A perfect matching is a matching M such that every node in V has exactly one incident edge in M. In order for G to have a perfect matching, we need that $|V_1| = |V_2|$. The perfect matching problem is, given a bipartite graph G with $|V_1| = |V_2| = n$ and on m edges, determine whether G has a perfect matching.

We will solve the bipartite perfect matching problem by creating an instance of max flow and using Ford-Fulkerson's algorithm.

Given $G = (V_1 \cup V_2, E)$, direct all the edges in E from V_1 to V_2. Add two extra nodes s and t. Add (directed) edges from s to every node in V_1 and from every node of V_2 to t. In this new graph H, let all the edge capacities be 1 and then run Ford-Fulkerson's algorithm to compute the max flow.

Suppose that G has a perfect matching M. Then, H has max flow value $n = |V_1| = |V_2|$. This is because we can set $f(e) = 1$ for every $e \in M$, ell the edges out of s and all the edges out of t. All other flow values are 0. The capacity constraints are trivially satisfied. The flow conservation constraints are satisfied since for every $x \in V_1$ there is exactly one edge (s, x) into x that has flow 1, and exactly one edge $(x, y) \in M$ with flow 1; similarly for every $x \in V_2$ there is exactly one edge (x, t) out of x that has flow 1, and exactly one edge $(y, x) \in M$ with flow 1.

Suppose now that Ford-Fulkerson returns a flow f of value n. Hence $f(s, x) = f(y, t) = 1$ for all $x \in V_1, y \in V_2$. Because Ford-Fulkerson causes all flow values on the edges to be integers, the flow values on all edges are either 1 or 0. Because of this, every node $x \in V_1$ gets flow of 1 going into it and a flow of 1 needs to come out so that there is a single edge (x, y) that has flow value 1 and all other edges out of x have flow value 0. Similarly, for every $y \in V_2$ there is a unique edge into y with positive flow value 1. The edges in $V_1 \cup V_2$ with positive flow through them must hence form a perfect matching.

8.2 More applications

There are many applications of max-flow and min-cut! We may talk about a few more in class if time (check the slides), and also check out Section 7.7 of Kleinberg and Tardos.