A Course in Graduate Algorithms **Lecture 5**

Max Flow and Min Cut

Designed by Prof. Bo Waggoner for the University of Colorado-Boulder *Updated: 2023*

The maximum-flow and minimum-cut problems are gems in the field of combinatorial algorithms, as well as preparation for advanced topics like linear programming.

Objectives:

- Understand the max flow problem, the Ford-Fulkerson algorithmic framework, and the Edmonds-Karp algorithm.
- Understand the min cut problem and how to solve it via max flow; know the max-flow min-cut theorem.
- Be able to use graph reductions to apply max flow to related problems such as (weighted) bipartite matching.

1 The Max Flow Problem

The *max s to t flow problem* captures routing large amounts of traffic on a graph along multiple paths from a source vertex s to a sink vertex t. Different edges in the graph have different capacities of how much flow they can handle.

Example: We have an Internet network where each wire can support a certain bandwidth in bits/second. We wish to send as many bits/second from s to t as possible, and we can split the routing of these bits over many different paths.

Input: A graph $G = (V, E)$ with a capacity function[1] $c : E \to \mathbb{R}_{\geq 0}$. Also, two vertices s (the source) and t (the sink).

If we have an edge $e = (u, v)$, then we may write either $c(e)$ or $c(u, v)$ for the capacity of that edge. We also extend c to a function on $V \times V$ by letting $c(u, v) = 0$ if $(u, v) \notin E$.

Output: the maximum flow from s to t, where we use the following definitions:

An **s-t flow** is a function $f : V \times V \to \mathbb{R}$ satisfying:

- (skew-symmetric) For all $u, v \in V$, we have $f(u, v) = -f(v, u)$.
- (net flow constraint) For each $v \notin \{s, t\}$, we have $\sum_{u \in V} f(u, v) = 0$.
- (capacity constraint) For all $u, v \in V$, we have $f(u, v) \leq c(u, v)$.

Notice that $f(u, v)$ is the "net" flow from u to v. If it is negative, this implies that the flow is actually from v to u. We note that if neither edge (u, v) nor (v, u) is in the graph, then $f(u, v) = f(v, u) = 0 = c(u, v) = c(v, u)$.

Given f, the **amount** of the flow is written $|f|$ and equals $\sum_{v \in V} f(s, v)$, the net flow out of s. A **maximum flow** from s to t for input (G, c) is any flow f with largest $|f|$. We may also call the amount $|f|$ the max flow.

[1] Note that c can be represented in the input as edge weights.

Internet routing example: The capacity constraint says that we can't send more bits/second over a wire than its capacity allows. The flow constraint says that the net total number of bits flowing into a router has to equal the net total bits flowing out.

Exercise 1. Draw a small instance graph G (say 4 to 5 nodes) with capacity constraints c and draw a flow f. Make sure it satisfies the flow, capacity, and skew-symmetric constraints.

(a) What is the amount $|f|$?

(b) What is the maximum flow on this instance? Is f a maximum flow?

1.1 An Attempted Solution

A natural idea is to start with zero flow: $f(e) = 0$ everywhere. Then, find a path from s to t where we can add a little bit of flow everywhere along the path. For example, maybe the smallest capacity on any edge in this path is 3. Then we can route 3 units of flow along each of the edges of the path. Now, we still have a flow (show that the constraints are still satisfied!), and the amount of flow is larger, i.e. 3. So if we find another path where there is room to add, and keep repeating this, hopefully we will stop at the maximum possible flow.

This idea is pretty good, but it won't actually quite work. How can we derive the actual correct version? Answer: math! We'll prove some facts about flows. This will tell us when the flow we've found is optimal, as well as the correct method for "augmenting" a submaximal flow.

Exercise 2. Draw a small instance graph where the naive algorithm above fails, and explain what goes wrong. *Hint: try a square with s in the upper left and t the lower right, and add an edge connecting the other two vertices. Now have the algorithm make a bad choice of first path...*

Exercise 3. Recall and write down the definitions of: *(a)* an s-t flow (including the three constraints it must satisfy), *(b)* $|f|$, *(c)* maximum flow.

2 Residual Graph, Augmenting Paths, and Ford-Fulkerson

Let's see how to improve or *augment* a given flow by sending additional net flow along a given path. Given an instance (G, c), we'll need some definitions:

Given a flow f, define the **residual capacity** function $r_f : V \times V \to \mathbb{R}_{\geq 0}$ as follows:

$$r_f(u, v) = c(u, v) - f(u, v).$$

In other words, $r_f(u, v)$ is the maximum amount of additional net flow we could send from u to v. Given r_f, we define the **residual graph** $G_f = (V, E')$, a directed graph where the vertices are the same as in G and $(u, v) \in E'$ if $r_f(u, v) > 0$.

In other words, an edge (u, v) is in the residual graph if we could send more flow from u to v without violating capacity constraints. (This includes cases where we're currently sending flow from v to u.)

Now, given a flow f, an **augmenting path in** G_f is a simple s-t path in G_f. We **augment** f **along the path** as follows:

1. Let $\alpha = \min_{u,v} r_f(u, v)$ where the minimum is over edges (u, v) in the path.
2. For each edge (u, v) in the path, we increase the net flow from u to v by α, i.e. set $f(u, v) += \alpha$. We update $f(v, u) = -f(u, v)$.

Example. Suppose we currently have $f(u, v) = -2$, i.e. 2 units of flow in the opposite direction. Suppose we augment f by 7 along a path that includes (u, v). This results in $f(u, v) = 5$. Similarly, $f(v, u)$, which was 2, is now -5.

We now prove two intuitive facts about augmentations.

Lemma 1. *Augmenting a flow along an augmenting path results in a flow (i.e. all three constraints are satisfied).*

Proof. The augmentation step ensures that the skew-symmetry constraints are still satisfied. We must show that the flow and capacity constraints are satisfied as well. Let α be the amount of augmentation.

Flow constraints: Let $v \notin \{s, t\}$. If v is not in the path, there is no change in flow in or out. Otherwise, say the path is $s, \ldots, u, v, w, \ldots, t$. We have $f(u, v)$ increased by α, while $f(w, v)$ is decreased by α. No other edges of v change, so the total net flow into v is still zero. So all flow constraints are satisfied.

For capacity constriants: if an edge and its reverse are not in the path, its flow does not change. Consider forward edges (u, v) in the path and recall $\alpha \leq r_f(u, v) = c(u, v) - f(u, v)$. So $f(u, v) + \alpha \leq c(u, v)$, as required. For reverse edges, we previously had $f(v, u) \leq c(v, u)$ and $f(v, u)$ only decreases by α, so its capacity constraint is still satisfied as well. So all capacity constraints are satisfied. □

Lemma 2. *Augmenting a flow by α increases the flow amount by α.*

Proof. Only one edge incident to s can be in the simple path, and net flow along all edges in the path increases by α. □

Exercise 4. Why can't $r_f(u, v)$ ever be negative?

Exercise 5. After drawing an instance (G, c) and a flow f (perhaps one you drew earlier), compute and draw r_f and G_f.

2.1 Ford-Fulkerson

The **Ford-Fulkerson framework** for solving max flow is as follows. Given an instance (G, c), begin with flow $f(u, v) = 0$ for all u, v. Then, repeat:

- Build the residual capacity r_f and residual graph G_f.
- Find a simple path from s to t in the residual graph. If no such path can be found, stop the algorithm and return f.
- Augment the flow f along the path.

Notice this is a general framework, not a specific algorithm, because step 2 (how to find a path from s to t) is not fully specified. Implementing step 2 in different ways results in different algorithms.

Next, we will develop some mathematics to prove that any Ford-Fulkerson algorithm correctly outputs a maximum flow. After that, we will see a specific example and consider its efficiency.

Exercise 6. After drawing an instance (G, c), non-maximal flow f, and the residual capacities r_f and residual network G_f (perhaps all reused from a previous exercise): Simulate at least one step of the Ford-Fulkerson framework, implementing step 2 however you choose.

3 Max Flows and Min Cuts

How can we actually prove that a given flow is maximum? The key idea is to formalize *bottlenecks* in the graph: places that all of the flow has to squeeze through simultaneously to get to the destination. For example, if there are only two doors to a building, then no matter how people move around on the inside and on the outside, the rate of entrants will max out at the total rate people can fit through these two doors.

Definitions. Formally, the min s-t cut problem gives as input a directed, weighted graph $G = (V, E)$ with nonnegative edge weights $c : E \to \mathbb{R}_{\geq 0}$, along with vertices s, t. Again, we define $c(u, v) = 0$ if $(u, v) \notin E$. The output is the minimum s-t cut, defined as follows.

A **cut** is a partition of the vertices of the graph into two sets, S and T. It is called an s-t cut if $s \in S$ and $t \in T$. [2]

Define the **value** of an s-t cut to be $K(S, T) := \sum_{u \in S} \sum_{v \in T} c(u, v)$, i.e. the sum of all capacities of all edges that start in S and end in T.

The **minimum s-t cut** is the value of the cut S, T minimizing $K(S, T)$. We also refer to the cut S, T itself (rather than the value) as the min cut.

In other words, you are allowed to split the vertices in any way as long as $s \in S$ and $t \in T$, and your goal is to find the split with the smallest amount of total capacity crossing from S over to T.

Exercise 7. Consider any weighted directed graph G with vertices s, t. I claim that, if I create an edge (t, s) and/or add any amount of weight (capacity) to that edge, the value of every s-t cut stays the same. Why is this true? *Hint: the definition of value of a cut only considers edges going what direction?*

Exercise 8. Draw a directed, weighted graph on a few vertices including s and t. What is the minimum s-t cut (S, T), and what is its value?

3.1 The max-flow min-cut theorem

Given a flow $f(u, v)$ and a cut (S, T), define the *net flow across the cut*

$$F(S, T) := \sum_{u \in S, v \in T} f(u, v).$$

Extend the same notation to any two subsets of vertices S, T.

Lemma 3. *Let an instance (G, c) be given. For any s-t cut S, T and any flow f, we have $F(S, T) \leq K(S, T)$.*

Proof.

$$\begin{aligned} F(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) && \text{(capacity constraints)} \\ &= K(S, T). \end{aligned}$$

□

This is very powerful, because we can pick *any* s-t flow and *any* s-t cut and we know that $F(S, T) \leq K(S, T)$. It becomes even more useful combined with the following fact: the flow across any s-t cut is the same regardless of the cut, and equals the amount of the flow.

Lemma 4. *Let an s-t flow f be given. Then $F(S, T) = |f|$ for all s-t cuts (S, T).*

Proof. Let $S = \{s\}, T = V \setminus S$. Then $F(S, T) = |f|$. Now let $S' = S \cup \{v\}$ and $T' = T \setminus \{v\}$, for any $v \neq t$. Then we can write $F(S, T) = F(S, T') + F(S, \{v\})$ while $F(S', T') = F(S, T') + F(\{v\}, T')$. The the difference net flow is

$$F(S, T) - F(S', T') = F(S, \{v\}) - F(\{v\}, T') = \sum_{u \neq v} f(u, v) = 0.$$

[2] Please do not confuse a cut (S, T) with the set of edges that cross the cut. The set of edges is not a cut (in this class); it is a different object. In this class, if you are asked for a cut in a graph, your answer should not involve edges, it should be some partition (S, T).

This follows from the net flow constraint[3] on f. So we have shown that moving one vertex from T to S does not change the net flow. Repeating the argument as many times as necessary proves the result. □

In particular, notice this proves that the net flow into t equals the net flow out of s.

Corollary 1. *For any instance (G, c), the amount of a max s-t flow is at most the value of a min s-t cut.*

Proof. Let f be a max flow and (S, T) a min cut. By Lemma 4, $|f| = F(S, T)$. And by Lemma 3, $F(S, T) \leq K(S, T)$. □

Armed with these facts, we can begin to prove Ford-Fulkerson correct and, along the way, prove the max-flow min-cut theorem.

Lemma 5. *If there exists an augmenting path in the residual network G_f, then f is not a maximum flow.*

Proof. Suppose there exists an augmenting path in G_f; it has some amount $\alpha > 0$. Then by Lemma 1, we can augment f along this path to obtain a new flow, and by Lemma 2, that flow has α higher amount. So f is not maximum. □

Lemma 6. *If there does not exist any augmenting path in G_f, then f is a max flow. Furthermore, in this case, a min cut is (S, T) where S is the vertices reachable from s in G_f, and $T = V \setminus S$.*

Proof. Suppose there does not exist an augmenting path in G_f. Let S be the set of all vertices reachable from s in G_f, and let $T = V \setminus S$. We must have $t \in T$, otherwise there would be an augmenting path. So (S, T) is an s-t cut. Furthermore, for every edge (u, v) crossing the cut (i.e. $u \in S, v \in T$), we have $r_f(u, v) = 0$ by definition. Otherwise, v would be reachable.

This implies $f(u, v) = c(u, v)$ for all $u \in S, v \in T$. So $F(S, T) = K(S, T)$. But by Corollary 1, all flows have amount at most $K(S, T)$. So f is a max flow. Similarly, all s-t cuts have value at least $F(S, T)$. So (S, T) is a min cut. □

Theorem 1 (Max-flow min-cut theorem)**.** *Any max s-t flow in a graph is equal to any min s-t cut.*

Proof. Let f be a flow, and suppose it is not equal to the min cut. We show it is not a max flow. By Corollary 1, $|f|$ cannot be greater than the min cut, so it is strictly less.

We claim there exists an augmenting path in G_f, as follows. Let $S' = \{s\}$ and $T' = V \setminus S'$. Then $|f| < K(S', T')$, since the $|f|$ is less than the min cut. Then $\sum_{u \in S', v \in T'} f(u, v) < \sum_{u \in S', v \in T'} c(u, v)$, so there must exist some such pair (u, v) with $f(u, v) < c(u, v)$. So v is reachable from s in G_f. So we can add v to S'. Now we repeat the argument until we add t to S'. At each point, all of S' is reachable from s in G_f, so in particular we have a simple path from s to t in G_f.

Since there is an augmenting path in G_f, by Lemma 5, f is not a maximum flow. This shows that any max flow must have $|f| = K(S, T)$. □

Exercise 9. Find a graph with nonnegative edge weights, perhaps one that you drew earlier. Check that an arbitrary s-t flow is at most an arbitrary s-t cut. Check that the max flow equals the min cut.

4 Analyzing Ford-Fulkerson; Edmonds-Karp

First, we prove that algorithms in the Ford-Fulkerson framework, if they manage to terminate, are correct. Then we analyze their efficiency per iteration. Finally, we analyze the Edmonds-Karp algorithm, which uses breadth-first-search to find s-t paths.

[3]Note that we technically also use $f(v, v) = 0$; this follows from the skew-symmetry constraint.

4.1 Ford-Fulkerson and min cut

Proposition 1. *If an algorithm in the Ford-Fulkerson framework terminates, then it outputs a maximum flow.*

Proof. A Ford-Fulkerson algorithm halts and returns f when there is no s-t path in the residual graph G_f. In this case, by Lemma 6, f is a maximum flow. □

Regarding efficiency, we can observe that computing the residual capacities and residual graph is essentially free, in a big-O sense. The trick is to not bother constructing these objects at all! Instead, we wait until the algorithm queries the object, and compute the answer just-in-time. We can show this only takes constant time:

Lemma 7. *Given an instance (G, c) and flow f, with c and f accessed in constant time, for any u, v we can compute $r_f(u, v)$ and check whether edge (u, v) is in G_f in constant time.*

Proof. Recall that $r_f(u, v) = c(u, v) - f(u, v)$, which can be computed in constant time by looking up these three values. Then, edge (u, v) is in the residual graph if $r_f(u, v) > 0$, which takes constant time to check. □

Obtaining a min cut. If we go back to the facts we proved about flows and cuts, we observe that Lemma 6 gives us a **reachability algorithm for min-cut given max flow**:

1. Run a max flow algorithm to obtain flow f.
2. Let S be the set of vertices reachable from s in the residual network G_f.
3. Let $T = V \setminus S$. Return (S, T).

Proposition 2. *The reachability algorithm for min cut is correct and, given a max flow f, runs in linear time, i.e. $O(|V| + |E|)$.*

Proof. If f is a maximum flow, then by Lemma 6, we get a min cut by letting S be the set of vertices reachable from s in G_f. Breadth-first search can find S in linear time $O(|V| + |E|)$. We then let T be the remaining vertices. □

4.2 Edmonds-Karp

The **Edmonds-Karp** algorithm is a case of the Ford-Fulkerson framework. It finds an augmenting path via breadth-first search in the residual graph G_f from s to t. If we plug this into the Ford-Fulkerson framework, we get:

Edmonds-Karp - full definition. Given an instance (G, c), begin with flow $f(u, v) = 0$ for all u, v. Then, repeat:

- Build the residual capacity r_f and residual graph G_f.
- Run Breadth-First Search from s to find a simple path from s to t in G_f. If no such path can be found, stop the algorithm and return f.
- Augment the flow f along the path.
- Repeat.

Correctness has already been argued for all algorithms in the Ford-Fulkerson framework, if they halt. So we need to argue that Edmonds-Karp does indeed halt, and analyze its efficiency.

Halting and running time: This proof is a bit involved, so we prove it in steps.

Given a residual graph G_f and an augmenting path p chosen by the algorithm, call an edge in the path *critical* if it achieves the minimum value of $r_f(u, v)$ along that path. Note that, in each iteration, the residual graph G_f changes by removing some forward edges (u, v) of the augmenting path (the critical ones), because their residual capacity becomes zero after the augmentation; and may change by adding reverse edges (v, u), because the augmentation creates residual capacity in the reverse direction.

Lemma 8. *In each iteration of the Edmonds-Karp algorithm, any vertex's distance from s (i.e. number of hops) in the residual graph cannot decrease.*

Proof. Let (u, v) be an edge in an augmenting path. This is a shortest path from s, so in general u is, say, distance k from s while v is distance $k+1$. Removing (u, v) can only increase the distance of v (and any other vertex). The edge (v, u) is added, but this cannot shorten any paths in the graph because the shortest path to u has distance k, whereas going through v would be strictly farther. Apply this argument repeatedly to all critical edges if there are more than one. □

Lemma 9. *In the Edmonds-Karp algorithm, each edge that may be present in the residual graph can be critical in at most $|V|/2$ iterations.*

Proof. Consider a potential edge $e = (u, v)$ and suppose it is critical in some iteration of the algorithm. Let the shortest-path distance from s to u at this time be k, and the distance to v is therefore $k + 1$. After the augmentation, e is removed from the residual graph.

Now, e can only be added back to the residual graph if the reverse edge (v, u) becomes critical. But in this case, it must be that a shortest path from s to u goes through v. The distance to v cannot have decreased, by Lemma 8. So u has distance at least $k + 2$

Repeating this, we see that each time (u, v) becomes critical, the distance of u from s has increased by 2, and this can occur at most $n/2$ times because the distance never exceeds n. □

Theorem 2. *The Edmonds-Karp algorithm on graph $G = (V, E)$ has running time $O(|V||E|^2)$.*

Proof. For each edge (u, v) of the original graph, we may have an edge (u, v) or (v, u) in the residual graph. By Lemma 9, each of these $2E$ edges is critical at most $|V|/2$ times, so there can be no more than $|E| \cdot |V|$ iterations. Each iteration can be accomplished via a call to breadth-first search on a graph with $|V|$ vertices and at most $2|E|$ edges, taking $O(|V| + |E|)$ time. (One may check that constructing the residual graph also takes linear time; one can also construct it "lazily" just-in-time.) Now, $|E| \geq |V| - 1$ (because we assume every vertex is on a path from s to t), so each iteration takes $O(|E|)$ time, giving a total of $O\left(|V| \cdot |E|^2\right)$ time. □

Exercise 10. Simulate Edmonds-Karp on a small instance graph (say 4 or 5 nodes) with small integer edge weights.

Exercise 11. What is the time complexity of using Edmonds-Karp to find a min cut?

5 Bipartite Matching

5.1 Definitions

A graph G is **bipartite** if we can divide the vertices into two sets, call them U_1 and U_2, such that every edge has exactly one endpoint in U_1 and one endpoint in U_2. We will often write such graphs $G = (U_1, U_2, E)$. Both directed and undirected graphs can be bipartite, although it is somewhat more common to consider undirected graphs.

Examples of bipartite graphs:

- A *star graph* has a central vertex u and a bunch of outer vertices $v_1, \ldots, v_{n-1}$. There is an edge between u and v_i for each i, and these are all the edges.
- A graph consisting of a pair (u_1, v_1) and the edge between them; a separate pair (u_2, v_2) and the edge between them; and so on.
- Any tree is bipartite (can you see why?).

Examples of non-bipartite graphs:

- A triangle graph (three vertices, with an edge between each pair).
- Any complete graph (all possible edges present) on $n \geq 3$ vertices, because it contains a triangle.
- Any graph that contains an odd-length cycle.

In any graph, a **matching** is a set of edges M that have no vertices in common. Given a matching M, a vertex is **matched** if it is the endpoint of one of the edges in the matching. Otherwise, it is **unmatched**. A matching is **perfect** if every vertex in the graph is matched.

Examples:

- The empty set is a matching (nobody is matched to anybody).
- A set containing one edge is a matching.
- In a star graph, the largest possible matching has size just one. (Do you see why?)

Motivating example: We have a set of people U_1 and a set of jobs U_2. If person u is qualified to do job v, we have an edge (u, v). A matching is an assignment of each person to at most one job and vice versa.

Exercise 12. Draw an undirected graph. Is it bipartite? If so, give the partition (U_1, U_2). If not, explain why not.

Exercise 13. Find a matching in your graph (what is the definition of a matching?).

5.2 Maximum matchings

In the **maximum bipartite matching problem**, the input is an unweighted, undirected bipartite graph $G = (U_1, U_2, E)$ and the output is a matching M of largest possible size.

To solve this problem, we will simply reduce to the max flow problem. (Note there are more efficient, specialized combinatorial algorithms, but the "augmenting paths" ideas they use are quite similar to max flow.)

Algorithm. Given the bipartite graph, we construct a weighted, directed graph G', which we call the *flow graph* of G. In G', we have all vertices in U_1 and U_2, and for each $(u, v) \in E$, we create a directed edge (u, v) with capacity $c(u, v) = 1$. We then create a "source vertex" s and create an edge (s, u) for each $u \in U_1$ with capacaity $c(s, u) = 1$. Finally, we create a "sink vertex" t with an edge (v, t) for each $v \in U_2$ with capacity $c(v, t) = 1$.

Then, we simply run a max-flow algorithm on G' with capacities c and source s, sink t. We return the value of the maximum flow as the size of the maximum matching, and the edges of the form (u, v) used in that flow as the edges of the matching.

Correctness - wait a minute! Counterexample: one can construct a non-integral max flow. For example, $U_1 = \{a, b\}$ and $U_2 = \{c, d\}$ with all edges $E = \{(a, c), (a, d), (b, c), (b, d)\}$. We can have a flow $f(e) = 0.5$ along each of these edges $e \in E$, which is a max flow, but doesn't correspond to any matching.

Luckily, we can show that in these cases, our algorithms find an *integral* max flow.

Theorem 3 (Integrality theorem)**.** *In a max-flow instance where each capacity $c(u, v)$ is an integer, any algorithm in the Ford-Fulkerson framework assigns an integral amount of flow $f(u, v)$ to every edge. In particular, the max flow amount is an integer.*

Proof. We simply need to argue that the flow f is an integer for all edges throughout the entire algorithm. At the beginning, the flow is zero everywhere. Now inductively: At each iteration, the amount of augmenting flow that is added to f along a path is $\alpha := r_f(u, v)$ for some critical edge (u, v). And $r_f(u, v) = c(u, v) - f(u, v)$. By assumption, each of these are integers, so the augmentation amount is an integer. So after augmenting, f remains integral everywhere. □

Correctness - revisited.

Proposition 3. *Given a bipartite undirected graph $G = (U_1, U_2, E)$, the max flow (and min cut) of the flow graph G' is equal to the size of the maximum matching in G. Furthermore, the edges $(u, v) \in E$ assigned positive flow by a Ford-Fulkerson algorithm comprise a maximum matching.*

Proof. By the integrality theorem, the max flow f found a Ford-Fulkerson max flow algorithm is integral. Each edge's capacity constraint in G' is 1, so in an integral flow, each edge has flow either zero or one. We claim that every valid integral flow in G' corresponds to a matching, i.e. the set of edges in E with positive flow 1.

For all $u \in U_1$, because $c(s, u) = 1$, it can only be incident to one edge (u, v) with flow 1; the same holds for all $v \in V_1$ because $c(v, t) = 1$. So a valid integral flow satisfies the matching constraints. On the other hand, any matching corresponds to an integral flow by the same reasoning. Furthermore, the size of the matching equals the amount of flow. So the max flow gives the maximum matching. □

Efficiency. Constructing the input to the max flow problem takes linear time, so the big-O complexity is equal to the complexity of solving max flow.

Exercise 14. Recall the precise definitions of a *cut* and of a *matching*. How does the maximum matching in G relate to the min cut in G' arising from the same max flow? (Draw a picture!)

5.3 Hall's Theorem

An important result in mathematics is Hall's "marriage" theorem on when a perfect matching exists in a bipartite graph. (Recall that a matching is *perfect* if all vertices participate.) It says this can occur if and only if any subset of c vertices on each side have at least c choices total on the other side.

Given a set of vertices X, define $N(X)$ to be the set of *neighbors* of X, i.e. vertices that share an edge with some vertex in X.

Theorem 4 (Hall's Theorem)**.** *An undirected bipartite graph $G = (U_1, U_2, E)$ with $|U_1| = |U_2|$ has a perfect matching if and only if, for every $X \subseteq U_1$, $|N(X)| \geq |X|$.*

In other words, each left vertex must have at least one neighbor on the right; each set of two left vertices must have at least two neighbors on the right total; etc.

Proof. The reverse direction is easy: if there is a perfect matching, then every set $X \subseteq U_1$ is matched to $|X|$ different vertices in U_2, so $|N(X)| \geq |X|$.

For the forward direction, suppose that $|N(X)| \geq |X|$ for all $X \subseteq U_1$. Consider the flow graph G' of G. We make one change to the flow graph as defined in the previous section: for each $(u, v) \in E$,

let $c(u, v) = \infty$ instead of 1. We leave it to the reader to check that Proposition 3 is still true, i.e. the max flow in G' is equal to the size of the maximum matching in G. The reason intuitively is that the capacity-one edges incident to s and t still provide all necessary constraints.

Now, we prove that the min cut S, T has value at least $|U_1|$. This implies the max flow is at least $|U_1|$, which implies by Proposition 3 that G has a perfect matching.

So given a min cut S, T, let $k = |T \cap U_1|$. Each of these vertices $v \in T \cap U_1$ contributes 1 to the value of the cut, since the edge (s, v) crosses the cut.

Of the $n-k$ vertices in $U_1 \cap S$, by assumption, they have at least $n-k$ neighbors in U_2. Furthermore, because this is a min cut, all of these neighbors are in S (otherwise an ∞ edge would be included, and this would not be a min cut). Their edges to t add at least $n-k$ to the value of the cut. So the cut has total value at least n. □