Cornell University, Fall 2016	CS 6820: Algorithms
Lecture notes: Flows and Cuts	7 Sep–12 Sep

Network flows are a structure with many nice applications in algorithms and combinatorics. A famous result called the *max-flow min-cut theorem* exposes a tight relationship between network flows and graph cuts; the latter is also a fundamental topic in combinatorics and combinatorial optimization, with many important applications.

These notes introduce the topic of network flows, present and analyze some algorithms for computing a maximum flow, prove the max-flow min-cut theorem, and present some applications in combinatorics. There are also numerous applications of these topics elsewhere in computer science. For example, network flow has obvious applications to routing in communication networks. Algorithms for computing minimum cuts in graphs have important but less obvious applications in computer vision. Those applications (along with many other practical applications of maximum flows and minimum cuts) are beyond the scope of these notes.

1 Basic Definitions

We begin by defining flows in directed multigraphs. (A multigraph is a graph that is allowed to have parallel edges, i.e. two or more edges having the same endpoints.)

Definition 1. In a directed multigraph $G = (V, E)$, a *flow* with source s and sink t (where s and t are vertices of G) is an assignment of a non-negative value f_e to each edge e, called the "flow on e", such that for every $v \neq s, t$, the total flow on edges leaving v equals the total flow on edges entering v. This equation is called "flow conservation at v". The *value of the flow*, denoted by $|f|$, is the total amount of flow on edges leaving the source, s.

One can formulate a clean notation for re-expressing this definition using the *incidence matrix* of G, which is the matrix B with rows indexed by vertices, and columns indexed by edges, whose entries are defined as follows.

$$B_{we} = \begin{cases} 1 & \text{if } w \text{ is the head of } e, \text{ i.e. } e = (u, v) \text{ and } w = v \\ -1 & \text{if } w \text{ is the tail of } e, \text{ i.e. } e = (u, v) \text{ and } w = u \\ 0 & \text{otherwise} \end{cases}$$

For any vertex v let $\mathbf{1}_v$ denote the *indicator vector* of v, i.e. the column vector (with rows indexed by V) whose entries are defined as follows.

$$(\mathbf{1}_v)_w = \begin{cases} 1 & \text{if } v = w \\ 0 & \text{otherwise} \end{cases}$$

In this notation, if we interpret a flow f as a column vector whose rows are indexed by E, then a vector of non-negative numbers, f, is a flow from s to t if and only if $Bf = \lambda(\mathbf{1}_t - \mathbf{1}_s)$ for some scalar $\lambda \in \mathbb{R}$, in which case the value of f is given by $|f| = \lambda$.

A useful interpretation of flows is that "a flow is a weighted sum of source-sink paths and cycles".

Lemma 1. *For an edge set S let its* characteristic vector $\mathbf{1}_S$ *be the vector in $\mathbb{R}^E$ whose e^{th} entry equals 1 if $e \in S$, 0 if $e \notin S$. A vector $f \in \mathbb{R}^E$ is a flow from s to t if and only if f is equal to a weighted sum (with non-negative weights) of vectors $\mathbf{1}_S$ as S ranges over s-t paths, t-s paths, and directed cycles. The value of f is the combined weight of s-t paths minus the combined weight of t-s paths in any such weighted-sum decomposition.*

Proof. Let $\mathbb{R}_+$ denote the set of non-negative real numbers. When S is the edge set of (i) an s-t path, (ii) a t-s path, or (iii) a directed cycle, we have $\mathbf{1}_S \in \mathbb{R}_+$ and $B\mathbf{1}_S = \lambda_S(\mathbf{1}_t - \mathbf{1}_s)$ where λ_S equals 1 in case (i), -1 in case (ii), and 0 in case (iii), respectively. Taking weighted sums of these identities, this verifies that any non-negative weighted sum of source-sink paths and cycles is a flow with the stated value.

Conversely, if f is a flow we must prove that it is a non-negative weighted sum of source-sink paths and cycles. Let $E_+(f) = \{e \mid f_e > 0\}$. The proof will be by induction on the number of edges in $E_+(f)$. When this number is zero, the lemma holds vacuously, so assume $|E_+(f)| > 0$. If $E_+(f)$ contains an s-t path, a t-s path, or a directed cycle, then let S denote the edge set of this path or cycle, and let $w = \min\{f_e \mid e \in S\}$. The vector $g = f - w\mathbf{1}_S$ is a flow of value $|g| = |f| - w\lambda_S$, and $|E_+(g)| < |E_+(f)|$, so by the induction hypothesis we can decompose g as a weighted sum of s-t paths, t-s paths, and cycles, and $|g|$ is the combined weight of s-t paths minus the combined weight of t-s paths. The induction step then follows because $f = g + w\mathbf{1}_S$.

To complete the proof we need to show that when $|E_+(f)| > 0$ there is an s-t path, a t-s path, or a directed cycle contained in $E_+(f)$. If $E_+(f)$ does not contain a directed cycle then $(V, E_+(f))$ is a directed acyclic graph with non-empty edge set. As such, it must have a source vertex, i.e. a vertex u_0 with at least one outgoing edge, but no incoming edges. Construct a path $P = u_0, u_1, \ldots, u_k$ starting from u_0 and choosing u_i, for $i > 1$, by following an edge $(u_{i-1}, u_i) \in E_+(f)$. Since $E_+(f)$ contains no cycles this greedy path construction process must terminate at a vertex with no outgoing edges. Flow conservation implies that every vertex other than s and t which belongs to an edge in $E_+(f)$ has both incoming and outgoing edges. Therefore, the endpoints of P are s and t (in some order) which completes the proof that $E_+(f)$ has either a path joining the source to the sink (in some order) or a directed cycle. □

Definition 2. A *flow network* is a directed multigraph $G = (V, E)$ together with a non-negative *capacity* $c(e)$ for each edge e. A *valid flow* in a flow network is a flow f in G that satisfies the *capacity constraints* $f_e \leq c(e)$ for all edges e. A *maximum flow* is a valid flow of maximum value.

Maximum flow turns out to be a versatile problem that encodes many other algorithmic problems. For example, the maximum bipartite matching in a graph $G = (U, V, E)$ can be encoded by a flow network with vertex set $U \cup V \cup \{s, t\}$ and with edge set $(\{s\} \times U) \cup E \cup (V \times \{t\})$, all edges having capacity 1. For each edge $(u, v) \in E$, the flow network contains a three-hop path $P_e = \langle s, u, v, t\rangle$, and for any matching M in G one can sum up the characteristic vectors of the paths P_e $(e \in M)$ to obtain a valid flow f such that $|f| = |M|$.

Conversely, any valid flow f satisfying $f_e \in \mathbb{Z}$ for all e is obtained from a matching M via this construction. As we will see shortly, in any flow network with integer edge capacities, there always exists an integer-valued maximum flow. Thus, the bipartite maximum matching problem reduces to maximum flow via the simple reduction given in this paragraph.

The similarity between maximum flow and bipartite maximum matching also extends to the algorithms for solving them. The most basic algorithms for solving maximum flow revolve around a graph called the *residual graph* which is analogous to the directed graph $D(G, M)$ that we defined when presenting algorithms for the bipartite maximum matching problem.

Definition 3. Let $G = (V, E, c)$ is a flow network and f a valid flow in G. Let $\bar{E}$ denote a set containing a directed edge $\bar{e}$ for every $e \in E$, whose endpoints are the same as the endpoints of e but in the opposite order. If $e \in E$ and $\bar{e} \in \bar{E}$, the *residual capacities* $c_f(e), c_f(\bar{e})$ are defined by

$$\begin{aligned} c_f(e) &= c(e) - f_e \\ c_f(\bar{e}) &= f_e. \end{aligned}$$

The *residual graph* G_f is the flow network $G_f = (V, E_f, c_f)$, where E_f is the set of all edges in $E \cup \bar{E}$ with positive residual capacity. An *augmenting path* is a path from s to t in G_f.

To any valid flow h in G_f one can associate the vector $\pi(h) \in \mathbb{R}^E$ defined by

$$\pi(h)_e = h_e - h_{\bar{e}}.$$

The vectors $\pi(h)$ encode all the ways of modifying f to another valid flow in G.

Lemma 2. *If f is a valid flow in G and h is a valid flow in the residual graph G_f then $f + \pi(h)$ is a valid flow in G. Conversely, every valid flow in G can be expressed as $f + \pi(h)$ for some valid flow h in G_f.*

Proof. The equation $Bh = B\pi(h)$ follows from the definition of $\pi(h)$, and implies that $\pi(h)$ satisfies the flow conservation equations, hence $f + \pi(h)$ does as well. The residual capacity constraints in G_f are designed precisely to guarantee that the value of $f + \pi(h)$ on each edge e lies between 0 and $c(e)$, hence $f + \pi(h)$ is a valid flow in G. Conversely, suppose $\tilde{f}$ is any valid flow in G. Using the notation x^+ to denote $\max\{x, 0\}$ for any real number x, we may define

$$\begin{aligned} h_e &= (\tilde{f}_e - f_e)^+ \quad \text{for all } e \in E \\ h_{\bar{e}} &= (f_e - \tilde{f}_e)^+ \quad \text{for all } \bar{e} \in \bar{E} \end{aligned}$$

and verify that h is a valid flow in G_f satisfying $\tilde{f} = f + \pi(h)$. □

Lemma 3. *If f is a valid flow in G, then f is a maximum flow if and only if G_f does not contain an augmenting path.*

Proof. If f is not a maximum flow then let f^* be any maximum flow and write $f^* = f + \pi(h)$. Since $|f^*| = |f| + |h| > |f|$, we must have $|h| > 0$. According to Lemma 1, the flow h decomposes as a weighted sum of vectors $\mathbf{1}_S$ where S ranges over s-t paths, t-s paths, and directed cycles in G_f, and at least one s-t path must have a positive coefficient in this decomposition because $|h| > 0$. In particular, this implies that G_f contains an s-t path, i.e. an augmenting path. Conversely, if G_f contains an augmenting path P, let $\delta(P)$ be the minimum residual capacity of an edge of P. The flow $f + \delta(P)\pi(\mathbf{1}_P)$ is a valid flow with value $|f| + \delta(P)$, so f is not a maximum flow. □

1.1 Comparison with Other Definitions

The Kleinberg-Tardos textbook assumes that s has no incoming edges and t has no outgoing edges. In these notes we do not impose any such assumption. Consequently G may contain a path from t to s, leading to the somewhat counter-intuitive convention that a flow on a path from t to s is considered to be an s-t flow of negative value. This convention is useful, for example, in Lemma 3 where it allows us to simply say that if f and $\tilde{f}$ are two s-t flows in G, then their difference $\tilde{f} - f$ can always be represented by an s-t flow in the residual graph G_f.

The Kozen textbook represents a flow using a skew-symmetric matrix, whose (u, v) entry represents the difference $f_{uv} - f_{vu}$, i.e. the net flow from u to v along edges that join the two vertices directly. This allows for a beautifully simple formulation of the flow conservation equations and of the lemma that the difference between any two flows is represented by a flow in the residual graph. However, when the graph contains a two-cycle comprising edges (u, v) and (v, u), the representation of a flow as a skew-symmetric matrix eliminates the distinction between sending zero flow on (u, v) and (v, u) and sending an equal (but non-zero) amount in both directions. Philosophically, I believe these should be treated as distinct flows so I have opted for a definition that enables such a distinction. The cost of making this choice is that some definitions become messier and more opaque, especially those involving the residual graph and the function π that maps flows in G_f to flows in G.

2 The Max-Flow Min-Cut Theorem

One important corollary of Lemma 3 is the *max-flow min-cut theorem*, which establishes a tight relationship between maximum flows and cuts separating the source from the sink. We first present some definitions involving cuts, and then we present and prove the theorem.

Definition 4 (s-t cut)**.** An *s-t cut* in a directed graph $G = (V, E)$ with vertices s and t is a partition of the vertex set V into two subsets S, T such that $s \in S$ and $t \in T$. An edge $e = (u, v)$ *crosses* the cut (S, T) if $u \in S$ and $v \in T$. (Note that edges from T to S do not cross the cut (S, T), under this definition.) The capacity of cut (S, T), denoted by $c(S, T)$, is the sum of the capacities of all edges that cross the cut.

Theorem 4 (Max-flow min-cut)**.** *For any flow network, the value of any maximum flow is equal to the capacity of any minimum s-t cut.*

Proof. Let (S, T) be any s-t cut, and let $\mathbf{1}_T$ be the vector in $\mathbb{R}^V$ whose v^{th} component is 1 if $v \in T$, 0 if $v \notin T$. The row vector $x = \mathbf{1}_T^\intercal B$ satisfies $x_e = 1$ if e goes from S to T, $x_e = -1$ if e goes from T to S, and $x_e = 0$ otherwise.

For a flow f and two disjoint vertex sets Q, R, let $f(Q, R)$ denote the sum of f_e over all edges e going from Q to R. We have

$$\mathbf{1}_T^\intercal B f = xf = f(S, T) - f(T, S) \leq c(S, T) \tag{1}$$

where the last inequality is justified because $f_e \leq c(e)$ for all e from S to T, and $f_e \geq 0$ for all e from T to S. Since f is a flow, we have

$$\mathbf{1}_T^\intercal B f = \mathbf{1}_T^\intercal (\mathbf{1}_t - \mathbf{1}_s)|f| = |f|. \tag{2}$$

Combining equations (1) and (2) yields the conclusion that the value of any flow is bounded above by the capacity of any s-t cut; in particular, the min-cut capacity is an upper bound on the maximum flow value.

To prove that this upper bound is tight, first note that in our derivation of the inequality $|f| \leq c(S, T)$, the only step that was an inequality (rather than an equation) was the inequality at the end of line (1). Reviewing our justification for that inequality, one can see that the two sides are equal if $f_e = c(e)$ for all edges e from S to T and $f_e = 0$ for all edges e from T to S. When f is a maximum flow, we can find a cut (S, T) that satisfies these properties by applying Lemma 3, which says that there is no s-t path in the residual graph G_f. Define S to be the set of all vertices reachable from s via a directed path in G_f, and T to be the complement of S; note that $s \in S$ and $t \in T$, so (S, T) is a valid cut. Since G_f contains no edges from S to T, it must be the case that for each edge e from S to T, the residual capacity $c_f(e)$ is zero (hence $f_e = c(e)$) and for each edge e from T to S, the residual capacity $c_f(\bar{e})$ is zero (hence $f_e = 0$). This confirms that S, T satisfies the conditions for the left and right sides of (1) to equal one another. □

3 Combinatorial Applications

In combinatorics, there are many examples of "min-max theorems" asserting that the minimum of `XXX` equals that maximum of `YYY`, where `XXX` and `YYY` are two different combinatorially-defined parameters related to some object such as a graph. Often these min-max theorems have two other salient properties.

1. It's straightforward to see that the maximum of `YYY` is no greater than the minimum of `XXX`, but the fact that they are equal is usually far from obvious, and in some cases quite surprising.

2. The theorem is accompanied by a polynomial-time algorithm to compute the minimum of `XXX` or the maximum of `YYY`.

Most often, these min-max relations can be derived as consequences of the max-flow min-cut theorem. (Which is, of course, one example of such a relation.) This also explains where the accompanying polynomial-time algorithm comes from.

There is a related phenomenon that applies to decision problems, where the question is whether or not an object has some property `P`, rather than a question about the maximum or minimum of some parameter. Once again, we find many theorems in combinatorics asserting that `P` holds if and only if `Q` holds, where:

1. It's straightforward to see that `Q` is necessary in order for `P` to hold, but the fact that `Q` is also sufficient is far from obvious.

2. The theorem is accompanied by a polynomial-time algorithm to decide whether property `P` holds.

Once again, these necessary and sufficient conditions can often be derived from the max-flow min-cut theorem

The main purpose of this section is to illustrate five examples of this phenomenon. Before getting to these applications, it's worth making a few other remarks.

1. The max-flow min-cut theorem is far from being the only source of such min-max relations. For example, many of the more sophisticated ones are derived from the Matroid Intersection Theorem, which is a topic that we will not be discussing this semester.

2. Another prolific source of min-max relations, namely LP Duality, has already been discussed informally this semester, and we will be coming to a proof later on. LP duality by itself yields statements about continuous optimization problems, but one can often derive consequences for discrete problems by applying additional special-purpose arguments tailored to the problem at hand.

3. The "applications" in these notes belong to mathematics (specifically, combinatorics) but there are many real-world applications of maximum flow algorithms. See Chapter 7 of Kleinberg & Tardos for applications to airline routing, image segmentation, determining which baseball teams are still capable of getting into the playoffs, and many more.

3.1 Preliminaries

The combinatorial applications of max-flow frequently rely on an easy observation about flow algorithms. The following theorem asserts that essentially everything we've said about network flow problems remains valid if some edges of the graph are allowed to have infinite capacity. Thus, in the following theorem, we define the term *flow network* to be a directed graph $G = (V, E)$ with source and sink vertices s, t and edge capacities $(c_e)_{e \in E}$ as before — including the stipulation that the vertex set V is finite — but we allow edge capacities $c(u, v)$ to be any non-negative real number *or infinity*. A flow is defined as before, except that when $c(u, v) = \infty$ it means that there is no capacity constraint for edge (u, v).

Theorem 5. *If G is a flow network containing an s-t path made up of infinite-capacity edges, then there is no upper bound on the maximum flow value. Otherwise, the maximum flow value*

and the minimum cut capacity are finite, and they are equal. Furthermore, any maximum flow algorithm that specializes the Ford-Fulkerson algorithm (e.g. Edmonds-Karp or Dinic) remains correct in the presence of infinite-capacity edges, and its worst-case running time remains the same.

Proof. If P is an s-t path made up of infinite capacity edges, then we can send an unbounded amount of flow from s to t by simply routing all of the flow along the edges of P. Otherwise, if S denotes the set of all vertices reachable from s by following a directed path made up of infinite-capacity edges, then by hypothesis $t \notin S$. So if we set $T = V \setminus S$, then (S, T) is an s-t cut and every edge from S to T has finite capacity. It follows that $c(S, T)$ is finite, and the maximum flow value is finite.

We now proceed by constructing a different flow problem $\hat{G}$ with the same directed graph structure finite edge capacities $\hat{c}_e$, and arguing that the outcome of running Ford-Fulkerson doesn't change when its input is modified from G to $\hat{G}$. The modified edge capacities in $\hat{G}$ are defined by

$$\hat{c}(u, v) = \begin{cases} c(u, v) & \text{if } c(u, v) < \infty \\ c(S, T) + 1 & \text{if } c(u, v) = \infty. \end{cases}$$

If (S', T') is any cut in $\hat{G}$ then either $\hat{c}(S', T') > \hat{c}(S, T) = c(S, T)$, or else $\hat{c}(S', T') = c(S', T')$; in particular, the latter case holds if (S', T') is a minimum cut in $\hat{G}$. To see this, observe that if $\hat{c}(S', T') \leq \hat{c}(S, T) = c(S, T)$, then for any $u \in S', v \in T'$, we have $\hat{c}(u, v) \leq c(S, T)$ and this in turn implies that $\hat{c}(u, v) = c(u, v)$ for all $u \in S', v \in T'$, and consequently $\hat{c}(S', T') = c(S', T')$.

Since $\hat{G}$ has finite edge capacities, we already know that any execution of the Ford-Fulkerson algorithm on input $\hat{G}$ will terminate with a flow f whose value is equal to the minimum cut capacity in $\hat{G}$. As we've seen, this is also equal to the minimum cut capacity in G itself, so the flow must be a maximum flow in G itself. Every execution of Ford-Fulkerson on $\hat{G}$ is also a valid execution on G and vice-versa, which substantiates the final claim about running times. □

3.2 Menger's Theorem

As a first application, we consider the problem of maximizing the number of disjoint paths between two vertices s, t in a graph. Menger's Theorem equates the maximum number of such paths with the minimum number of edges or vertices that must be deleted from G in order to separate s from t.

Definition 5. Let G be a graph, either directed or undirected, with distinguished vertices s, t. Two $s - t$ paths P, P' are *edge-disjoint* if there is no edge that belongs to both paths. They are *vertex-disjoint* if there is no vertex that belongs to both paths, other than s and t. (This notion is sometimes called *internally-disjoint*.)

Definition 6. Let G be a graph, either directed or undirected, with distinguished vertices s, t. An $s - t$ edge cut is a set of edges C such that every $s - t$ path contains an edge of C. An $s - t$ vertex cut is a set of vertices U, disjoint from $\{s, t\}$, such that every $s - t$ path contains a vertex of U.

Theorem 6 (Menger's Theorem)**.** *Let G be a (directed or undirected) graph and let s, t be two distinct vertices of G. The maximum number of edge-disjoint $s-t$ paths equals the minimum cardinality of an $s-t$ edge cut, and the maximum number of vertex-disjoint $s-t$ paths equals the minimum cardinality of an $s-t$ vertex cut. Furthermore the maximum number of disjoint paths can be computed in polynomial time.*

Proof. The theorem actually asserts four min-max relations, depending on whether we work with directed or undirected graphs and whether we work with edge-disjointness or vertex-disjointness. In all four cases, it is easy to see that the minimum cut constitutes an upper bound on the maximum number of disjoint paths, since each path must intersect the cut in a distinct edge/vertex. In all four cases, we will prove the reverse inequality using the max-flow min-cut theorem.

To prove the results about edge-disjoint paths, we simply make G into a flow network by defining $c(u, v) = 1$ for all directed edges $(u, v) \in E(G)$; if G is undirected then we simply set $c(u, v) = c(v, u) = 1$ for all $(u, v) \in E(G)$. The theorem now follows from two claims: **(A)** an integer $s-t$ flow of value k implies the existence of k edge-disjoint $s-t$ paths and vice versa; **(B)** a cut of capacity k implies the existence of an $s-t$ edge cut of cardinality k and vice-versa. To prove (A), we can decompose an integer flow f of value k into a set of edge-disjoint paths by finding one $s-t$ path consisting of edges (u, v) such that $f(u, v) = 1$, setting the flow on those edges to zero, and iterating on the remaining flow; the transformation from k disjoint paths to a flow of value k is even more straightforward. To prove (B), from an $s-t$ edge cut C of cardinality k we get an $s-t$ cut of capacity k by defining S to be all the vertices reachable from s without crossing C; the reverse transformation is even more straightforward.

To prove the results about vertex-disjoint paths, the transformation uses some small "gadgets". Every vertex v in G is transformed into a pair of vertices $v_{\text{in}}, v_{\text{out}}$, with $c(v_{\text{in}}, v_{\text{out}}) = 1$ and $c(v_{\text{out}}, v_{\text{in}}) = 0$. Every edge (u, v) in G is transformed into an edge from u_{out} to v_{in} with infinite capacity. In the undirected case we also create an edge of infinite capacity from v_{out} to u_{in}. Now we solve max-flow with source s_{out} and sink t_{in}. As before, we need to establish two claims: **(A)** an integer $s_{\text{out}} - t_{\text{in}}$ flow of value k implies the existence of k vertex-disjoint $s-t$ paths and vice versa; **(B)** a cut of capacity k implies the existence of an $s_{\text{out}} - t_{\text{in}}$ vertex cut of cardinality k and vice-versa. Claim (A) is established exactly as above. Claim (B) is established by first noticing that in any finite-capacity cut, the only edges crossing the cut must be of the form $(v_{\text{in}}, v_{\text{out}})$; the set of all such v then constitutes the $s-t$ vertex cut. □

3.3 The König-Egervary Theorem

Recall that a matching in a graph is a collection of edges such that each vertex belongs to at most one edge. A *vertex cover* of a graph is a vertex set A such that every edge has at least one endpoint in A. Clearly the cardinality of a maximum matching cannot be greater than the cardinality of a minimum vertex cover. (Every edge of the matching contains a distinct element of the vertex cover.) The König-Egervary Theorem asserts that in bipartite graphs, these two parameters are always equal.

Theorem 7 (König-Egervary)**.** *If G is a bipartite graph, the cardinality of a maximum matching in G equals the cardinality of a minimum vertex cover in G.*

Proof. The proof technique illustrates a very typical way of using network flow algorithms: we make a bipartite graph into a flow network by attaching a "super-source" to one side and a "super-sink" to the other side. Specifically, if G is our bipartite graph, with two vertex sets X, Y, and edge set E, then we define a flow network $\hat{G} = (X \cup Y \cup \{s, t\}, c, s, t)$ where the following edge capacities are nonzero, and all other edge capacities are zero:

$$\begin{aligned} c(s, x) &= 1 \quad \text{for all } x \in X \\ c(y, t) &= 1 \quad \text{for all } y \in Y \\ c(x, y) &= \infty \quad \text{for all } (x, y) \in E \end{aligned}$$

For any integer flow in this network, the amount of flow on any edge is either 0 or 1. The set of edges (x, y) such that $x \in X, y \in Y, f(x, y) = 1$ constitutes a matching in G whose cardinality is equal to $|f|$. Conversely, any matching in G gives rise to a flow in the obvious way. Thus the maximum flow value equals the maximum matching cardinality.

If (S, T) is any finite-capacity $s - t$ cut in this network, let $A = (X \cap T) \cup (Y \cap S)$. The set A is a vertex cover in G, since an edge $(x, y) \in E$ with no endpoint in A would imply that $x \in S, y \in T, c(x, y) = \infty$ contradicting the finiteness of $c(S, T)$. The capacity of the cut is equal to the number of edges from s to T plus the number of edges from S to t (no other edges from S to T exist, since they would have infinite capacity), and this sum is clearly equal to $|A|$. Conversely, a vertex cover A gives rise to an $s - t$ cut via the reverse transformation, and the cut capacity is $|A|$. □

3.4 Hall's Theorem

Theorem 8. *Let G be a bipartite graph with vertex sets X, Y and edge set E. Assume $|X| = |Y|$. For any $W \subseteq X$, let $\Gamma(W)$ denote the set of all $y \in Y$ such that $(w, y) \in E$ for at least one $w \in W$. In order for G to contain a perfect matching, it is necessary and sufficient that each $W \subseteq X$ satisfies $|\Gamma(W)| \geq |W|$.*

Proof. The stated condition is clearly necessary. To prove it is sufficient, assume that $|\Gamma(W)| \geq |W|$ for all W. Transform G into a flow network $\hat{G}$ as in the proof of the König-Egervary Theorem. If there is a integer flow of value $|X|$ in $\hat{G}$, then the edges (x, y) such that $x \in X, y \in Y, f(x, y) = 1$ constitute a perfect matching in G and we are done. Otherwise, there is a cut (S, T) of capacity $k < n$. We know that

$$|X \cap T| + |Y \cap S| = k < n = |X \cap T| + |X \cap S|$$

from which it follows that $|Y \cap S| < |X \cap S|$. Let $W = X \cap S$. The set $\Gamma(W)$ is contained in $Y \cap S$, as otherwise there would be an infinite-capacity edge crossing from S to T. Thus, $|\Gamma(W)| \leq |Y \cap S| < |W|$, and we verified that when a perfect matching *does not* exist, there is a set W violating Hall's criterion. □

3.5 Dilworth's Theorem

In a directed acyclic graph G, let us say that a pair of vertices v, w are *incomparable* if there is no path passing through both v and w, and define an *antichain* to be a set of pairwise incomparable vertices.

Theorem 9. *In any finite directed acyclic graph G, the maximum cardinality of an antichain equals the minimum number of paths required to cover the vertex set of G.*

The proof is much trickier than the others. Before presenting it, it is helpful to introduce a directed graph G^* called the *transitive closure* of G. This has same vertex set V, and its edge set E^* consists of all ordered pairs (v, w) such that $v \neq w$ and there exists a path in G from v to w. Some basic facts about the transitive closure are detailed in the following lemma.

Lemma 10. *If G is a directed acyclic graph, then its transitive closure G^* is also acyclic. A vertex set A constitutes an independent set in G^* (i.e. no edge in E^* has both endpoints in S) if and only if A is an antichain in G. A sequence of vertices $v_0, v_1, \ldots, v_k$ constitutes a path in G^* if and only if it is a subsequence of a path in G. For all k, G^* can be partitioned into k or fewer paths if and only if G can be covered by k or fewer paths.*

Proof. The equivalence of antichains in G and independent sets in G^* is a direct consequence of the definitions. If $v_0, \ldots, v_k$ is a directed walk in G^* — i.e., a sequence of vertices such that (v_{i-1}, v_i) is an edge for each $i = 1, \ldots, k$ — then there exist paths P_i from v_{i-1} to v_i in G, for each i. The concatenation of these paths is a directed walk in G, which must be a simple path (no repeated vertices) since G is acyclic. This establishes that $v_0, \ldots, v_k$ is a subsequence of a path in G, as claimed, and it also establishes that $v_0 \neq v_k$, hence G^* contains no directed cycles, as claimed. Finally, if G^* is partitioned into k paths then we may apply this construction to each of them, obtaining k paths that cover G. Conversely, given k paths $P_1, \ldots, P_k$ that cover G, then G^* can be partitioned into paths $P_1^*, \ldots, P_k^*$ where P_i^* is the subsequence of P_i consisting of all vertices that do not belong to the union of $P_1, \ldots, P_{i-1}$. □

Using these facts about the transitive closure, we may now prove Dilworth's Theorem.

Proof of Theorem 9. Define a flow network $\hat{G} = (W, c, s, t)$ as follows. The vertex set W contains two special vertices s, t as well as two vertices x_v, y_v for every vertex $v \in V(G)$. The following edge capacities are nonzero, and all other edge capacities are zero.

$$\begin{aligned} c(s, x_v) &= 1 \quad \text{for all } v \in V \\ c(x_v, y_w) &= \infty \quad \text{for all } (v, w) \in E^* \\ c(y_w, t) &= 1 \quad \text{for all } w \in V \end{aligned}$$

For any integer flow in the network, the amount of flow on any edge is either 0 or 1. Let F denote the set of edges $(v, w) \in E^*$ such that $f(x_v, y_w) = 1$. The capacity and flow conservation constraints enforce some degree constraints on F: every vertex of G^* has at

most one incoming edge and at most one outgoing edge in F. In other words, F is a union of disjoint paths and cycles. However, since G^* is acyclic, F is simply a union of disjoint paths in G^*. In fact, if a vertex doesn't belong to any edge in F, we will describe it as a path of length 0 and in this way we can regard F as a partition of the vertices of G^* into paths. Conversely, every partition of the vertices of G^* into paths translates into a flow in $\hat{G}$ in the obvious way: for every edge (v, w) belonging to one of the paths in the partition, send one unit of flow on each of the edges $(s, x_v), (x_v, y_w), (y_w, t)$.

The value of f equals the number of edges in F. Since F is a disjoint union of paths, and the number of vertices in a path always exceeds the number of edges by 1, we know that $n = |F| + p(F)$. Thus, if the maximum flow value in $\hat{G}$ equals k, then the minimum number of paths in a path-partition of G^* equals $n - k$, and Lemma 10 shows that this is also the minimum number of paths in a path-covering of G. By max-flow min-cut, we also know that the minimum cut capacity in $\hat{G}$ equals k, so to finish the proof, we must show that an $s - t$ cut of capacity k in $\hat{G}$ implies an antichain in G — or equivalently (again using Lemma 10) an independent set in G^* — of cardinality $n - k$.

Let S, T be an $s - t$ cut of capacity k in $\hat{G}$. Define a set of vertices A in G^* by specifying that $v \in A$ if $x_v \in S$ and $y_v in T$. If a vertex v does not belong to A then at least one of the edges (s, x_v) or (y_v, t) crosses from S to T, and hence there are at most k such vertices. Thus $|A| \geq n - k$. Furthermore, there is no edge in G^* between elements of A: if (v, w) were any such edge, then (v, w') would be an infinite-capacity edge of $\hat{G}$ crossing from S to T. Hence there is no path in G between any two elements of A, i.e. A is an antichain. □

4 The Ford-Fulkerson Algorithm

Lemma 3 constitutes the basis for the Ford-Fulkerson algorithm, which computes a maximum flow iteratively, by initializing $f = 0$ and repeatedly replacing f with $f + \delta(P)\pi(\mathbf{1}_P)$ where P is an augmenting path in G_f, and $\delta(P)$ is the minimum residual capacity of an edge in P. The algorithm terminates when G_f no longer contains an augmenting path, at which point Lemma 3 guarantees that f is a maximum flow.

Algorithm 1 FordFulkerson(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s-t path P do
3:     Let P be one such path.
4:     Let δ(P) = min{c_f(e) | e ∈ P}.
5:     f ← f + δ(P)π(1_P)              // Augment f using P.
6:     Update G_f.
7: end while
8: return f
```

Theorem 11. *In any flow network with integer edge capacities, any execution of the Ford-Fulkerson algorithm terminates and outputs an integer-valued maximum flow, f^*, after at most $|f^*|$ iterations of the main loop.*

Proof. At any time during the algorithm's execution, the residual capacities c_f are all integers; this can easily be seen by induction on the number of iterations of the main loop, the key observation being that the quantity $\delta(P)$ computed during each loop iteration must always be an integer.

It follows that $|f|$ increases by at least 1 during each loop iteration, so the algorithm terminates after at most $|f^*|$ loop iterations, where f^* denotes the output of the algorithm. Finally, Lemma 3 ensures that f^* must be a maximum flow because, by the algorithm's termination condition, its residual graph has no augmenting path. □

Each iteration of the Ford-Fulkerson main loop can be implemented in linear time, i.e. the time required to search for the augmenting path P in G_f (using BFS or DFS) and to construct the new residual graph after updating f. For the sake of simplicity, we will express the running time of each loop iteration as $O(m)$ rather than $O(m+n)$, which can be justified by making a standing assumption that each vertex of the graph is incident to at least one edge, hence $m \geq n/2$. (If the standing assumption is violated, isolated vertices can be removed using a trivial $O(n)$ preprocessing step, which adds $O(n)$ to the running time of every algorithm considered in these notes.) The benefit of the standing assumption is that it leads to simpler and more readable running time bounds for the maximum-flow algorithms we are analyzing. In integer-capacitated graphs, we have seen that the Ford-Fulkerson algorithm runs in at most $|f^*|$ linear-time iterations, where $|f^*|$ is the value of a maximum flow, hence the algorithm's running time is $O(m|f^*|)$.

5 The Edmonds-Karp and Dinitz Algorithms

The Ford-Fulkerson algorithm's running time is pseudopolynomial, but not polynomial. In other words, its running time is polynomial in the *magnitudes* of the numbers constituting the input (i.e., the edge capacities) but not polynomial in the *number of bits* needed to describe those numbers. To illustrate the difference, consider a flow network with vertex set $\{s, t, u, v\}$ and edge set $\{(s, u), (u, t), (s, v), (v, t), (u, v)\}$. The capacities of the edges are

$$c(s, u) = c(u, t) = c(s, v) = c(v, t) = 2^n, \quad c(u, v) = 1.$$

The maximum flow in this network sends 2^n units on each of the paths $\langle s, u, t\rangle$ and $\langle s, v, t\rangle$, and if the Ford-Fulkerson algorithm chooses these as its first two augmenting paths, it terminates after only two iterations. However, it could alternatively choose $\langle s, u, v, t\rangle$ as its first augmenting path, sending only one unit of flow on the path. This results in adding the edge (v, u) to the residual graph, at which point it becomes possible to send one unit of flow on the augmenting path $\langle s, u, v, t\rangle$. This process iterates 2^n times.

A more sophisticated example shows that in a flow network whose edge capacities are irrational numbers, the Ford-Fulkerson algorithm may run through its main loop an infinite number of times without terminating.

In this section we will present two maximum flow algorithms with *strongly polynomial* running times. This means that if we count each arithmetic operation is consuming only one unit of running time (regardless of the number of bits of precision of the numbers involved)

then the running time is bounded by a polynomial function of the number of vertices and edges of the network.

5.1 The Edmonds-Karp Algorithm

The Edmonds-Karp algorithm refines the Ford-Fulkerson algorithm by always choosing the augmenting path with the smallest number of edges.

Algorithm 2 EdmondsKarp(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s − t path P do
3:     Let P be an s − t path in G_f with the minimum number of edges.
4:     f ← f + δ(P)π(1_P)              // Augment f using P.
5:     Update G_f
6: end while
7: return f
```

To begin our analysis of the Edmonds-Karp algorithm, note that the s-t path in G_f with the minimum number of edges can be found in $O(m)$ time using breadth-first search. Once path P is discovered, it takes only $O(n)$ time to augment f using P and $O(n)$ time to update G_f, so we see that one iteration of the **while** loop in EdmondsKarp(G) requires only $O(m)$ time. However, we still need to figure out how many iterations of the **while** loop could take place, in the worst case.

To reason about the maximum number of **while** loop iterations, we will assign a distance label $d(v)$ to each vertex v, representing the length of the shortest path from s to v in G_f. We will show that $d(v)$ never decreases during an execution of EdmondsKarp(G). Recall that the same method of reasoning was instrumental in the running-time analysis of the Hopcroft-Karp algorithm.

Any edge (u, v) in G_f must satisfy $d(v) \leq d(u) + 1$, since a path of length $d(u) + 1$ can be formed by appending (u, v) to a shortest s-u path in G_f. Call the edge *advancing* if $d(v) = d(u) + 1$ and *retreating* if $d(v) \leq d(u)$. Any shortest augmenting path P in G_f is composed exclusively of advancing edges. Let G_f and $\tilde{G}_f$ denote the residual graph before and after augmenting f using P, respectively, and let $d(v), \tilde{d}(v)$ denote the distance labels of vertex v in the two residual graphs. Every edge (u, v) in $\tilde{G}_f$ is either an edge of G_f or the reverse of an edge of P; in both cases the inequality $d(v) \leq d(u) + 1$ is satisfied. Therefore, on any path in $\tilde{G}_f$ the value of d increases by at most one on each hop of the path, and consequently $\tilde{d}(v) \geq d(v)$ for every v. This proves that the distance labels never decrease, as claimed earlier.

When we choose augmenting path P in G_f, let us say that edge $e \in E(G_f)$ is a bottleneck edge for P if it has the minimum residual capacity of any edge of P. Notice that when $e = (u, v)$ is a bottleneck edge for P, then it is eliminated from G_f after augmenting f using P. Suppose that $d(u) = i$ and $d(v) = i + 1$ when this happens. In order for e to be added back into G_f later on, edge (v, u) must belong to a shortest augmenting path, implying

$d(u) = d(v) + 1 \geq i + 2$ at that time. Thus, the total number of times that e can occur as a bottleneck edge during the Edmonds-Karp algorithm is at most $n/2$. There are $2m$ edges that can potentially appear in the residual graph, and each of them serves as a bottleneck edge at most $n/2$ times, so there are at most mn bottleneck edges in total. In every iteration of the **while** loop the augmenting path has at least one bottleneck edge, so there are at most mn **while** loop iterations in total. Earlier, we saw that every iteration of the loop takes $O(m)$ time, so the running time of the Edmonds-Karp algorithm is $O(m^2n)$.

5.2 The Dinitz Algorithm

Similar to the way that the Hopcroft-Karp algorithm improves the running time for finding a maximum matching in a graph by finding a *maximal* set of shortest augmenting paths all at once, there is a maximum-flow algorithm due to Dinitz that improves the running time of the Edmonds-Karp algorithm by finding a so-called *blocking flow* in the residual graph.

Definition 7. If G is a flow network, f is a flow, and h is a flow in the residual graph G_f, then h is called a *blocking flow* if every shortest augmenting path in G_f contains at least one edge that is saturated by h, and every edge e with $h_e > 0$ belongs to a shortest augmenting path.

Algorithm 3 EdmondsKarp(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s − t path P do
3:     Let h be a blocking flow in G_f.
4:     f ← f + π(h)
5:     Update G_f
6: end while
7: return f
```

Later we will specify how to compute a blocking flow. For now, let us focus on bounding the number of iterations of the main loop. As in the analysis of the Edmonds-Karp algorithm, the distance $d(v)$ of any vertex v from the source s can never decrease during an execution of Dinitz's algorithm. Furthermore, the length of the shortest path from s to t in G_f must *strictly* increase after each loop iteration: the edges (u, v) which are added to G_f at the end of the loop iteration satisfy $d(v) \leq d(u)$ (where $d(\cdot)$ refers to the distance labels at the *start* of the iteration) so any s-t path of length $d(t)$ in the *new* residual graph would have to be composed exclusively of advancing edges which existed in the *old* residual graph. However, any such path must contain at least one edge which was saturated by the blocking flow, hence deleted from the residual graph. Therefore, each loop iteration strictly increases $d(t)$ and the number of loop iterations is bounded above by n.

The algorithm to compute a blocking flow explores the subgraph composed of advancing edges in a depth-first manner, repeatedly finding augmenting paths.

Algorithm 4 BLOCKINGFLOW(G_f)

$h \leftarrow 0$
Let G' be the subgraph composed of advancing edges in G_f.
Initialize $c'(e) = c_f(e)$ for each edge e in G'.
Initialize stack with $\langle s \rangle$.
repeat
    Let u be the top vertex on the stack.
    if $u = t$ **then**
        Let P be the path defined by the current stack. *// Now augment h using P.*
        Let $\delta(P) = \min\{c'(e) \mid e \in P\}$.
        $h \leftarrow h + \delta(P)\mathbf{1}_P$.
        $c'(e) \leftarrow c'(e) - \delta(P)$ for all $e \in P$.
        Delete edges with $c'(e) = 0$ from G'.
        Let (u, v) be the newly deleted edge that occurs earliest in P.
        Truncate the stack by popping all vertices above u.
    else if G' contains an edge (u, v) **then**
        Push v onto the stack.
    else
        Delete u and all of its incoming edges from G'.
        Pop u off of the stack.
    end if
until stack is empty
return h

The block of code that augments h using P is called at most m times (each time results in the deletion of at least one edge) and takes $O(n)$ steps each time, so it contributes $O(mn)$ to the running time of BLOCKINGFLOW(G_f). At most n vertices are pushed onto the stack before either a path is augmented or a vertex is deleted, so $O(mn)$ time is spent pushing vertices onto the stack. The total work done initializing G', as well as the total work done deleting vertices and their incoming edges, is bounded by $O(m)$. Thus, the total running time of BLOCKINGFLOW(G_f) is bounded by $O(mn)$, and the running time over Dinitz's algorithm overall is bounded by $O(mn^2)$.

A modification of Dinitz's algorithm using fancy data structures achieves running time $O(mn \log n)$. The preflow-push algorithm, presented in Section 7.4 of Kleinberg-Tardos, has a running time of $O(n^3)$. The fastest known strongly-polynomial algorithm, due to Orlin, has a running time of $O(mn)$. There are also weakly polynomial algorithms for maximum flow in integer-capacitated networks, i.e. algorithms whose running time is polynomial in the number of vertices and edges, and the logarithm of the largest edge capacity, U. The fastest such algorithm, due to Lee and Sidford, has a running time of $O(m\sqrt{n}\,\mathrm{poly}(\log n, \log U))$.