Cornell University, Fall 2024	CS 6820: Algorithms
Lecture notes: Flows and Cuts	7 Oct–14 Oct

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 with the following definition of flow.

Definition 1. Let V be a finite set with two distinguished elements s, t, called the source and sink. An $s - t$ *flow* in G is a two-variable function $f : V \times V \to \mathbb{R}$ that satisfies:

- **skew-symmetry:** $f(u, v) + f(v, u) = 0$ for all $u, v \in V$.
- **flow conservation:** $\sum_{v \in V} f(u, v) = 0$ for all $u \in V$.

A *flow network* consists of a vertex set V with source and sink s, t, and a capacity function $c : V \times V \to [0, \infty]$. A flow is *feasible* with respect to c if it obeys

- **capacity constraints:** $f(u, v) \leq c(u, v)$ for all $u, v \in V$.

The *value* of a flow f is the total amount of flow leaving the source: $\text{val}(f) = \sum_{v \in V} f(u, v)$. A *maximum flow* in flow network $G = (V, s, t, c)$ is a feasible flow of maximum value.

A useful interpretation of flows is that "a flow is a weighted sum of source-sink paths and cycles". To flesh out this interpretation, we have the following definition and lemma.

Definition 2. Suppose $P = v_0, v_1, \ldots, v_k$ is a sequence of vertices forming either:

- a simple path connecting s and t, i.e. $\{v_0, v_k\} = \{s, t\}$ and $v_0, \ldots, v_k$ are all distinct; or
- a simple cycle, i.e. $v_0 = v_k$ and $v_1, \ldots, v_k$ are all distinct.

The *elementary flow* f^P is the flow defined by

$$f(u,v) = \begin{cases} 1 & \text{if } \exists i \text{ s.t. } u = v_i, v = v_{i+1} \\ -1 & \text{if } \exists i \text{ s.t. } u = v_{i+1}, v = v_i \\ 0 & \text{otherwise.} \end{cases}$$

An *elementary decomposition* of a flow f is an expression $f = \sum_P w_P f^p$, where each of the flows f^P in the sum on the right side is an elementary flow and each of the weights w_P is non-negative.

Lemma 1. *Every flow f has an elementary decomposition. The value* $\mathrm{val}(f)$ *is the weight of s-t paths minus the weight of t-s paths in any elementary decomposition.*

Proof. If f is a flow, let $E_+(f) = \{(u,v) \mid f(u,v) > 0\}$. To prove that f has an elementary decomposition we use induction on the number of elements of $E_+(f)$. When this number is zero, the lemma holds vacuously (i.e., the decomposition is an empty sum), so assume $|E_+(f)| > 0$. We claim this assumption implies there exists an s-t path, t-s path, or 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 $(u_{i-1}, u_i) \in E_+(f)$ if there is at least one such. 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. Now let P denote the sequence of vertices constituting this path or cycle, and let $w = \min\{f(u,v) \mid u, v \text{ consecutive in } P\}$. The flow $g = f - wf^P$ satisfies $|E_+(g)| < |E_+(f)|$, so by the induction hypothesis g has an elementary decomposition. Now $f = g + wf^P$ shows that f has an elementary decomposition as well.

To finish proving the lemma, we need to show that the value of f equals the weight of s-t paths minus the weight of t-s paths in an elementary decomposition of f. This equation is true when f itself is an elementary flow and the decomposition is the trivial decomposition consisting of f itself. By linearity, it holds true for every elementary decomposition of every flow. □

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

$$c(a,b) = \begin{cases} 1 & \text{if } a = s,\ b \in U \text{ or } a \in V,\ b = t \\ 1 & \text{if } a \in U,\ b \in V,\ (a,b) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For each edge $(u,v) \in E$, the flow network contains a three-hop path $P(u,v) = s, u, v, t$, and for any matching M in G one can sum up the elementary flows $f^{P(u,v)}$ of the edges of M to

obtain a valid flow f such that $\text{val}(f) = |M|$. Conversely, any feasible 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 generalizes the directed graph G_M that we defined when presenting algorithms for the bipartite maximum matching problem.

Definition 3. If $G = (V, s, t, c)$ is a flow network and f is a feasible flow in G, the residual network G_f is the flow network $G_f = (V, s, t, c - f)$. The capacity of edge (u, v) in G_f, namely $c_f(u, v) = c(u, v) - f(u, v)$, is called the *residual capacity of* (u, v) *with respect to* f. If $c(u, v) = \infty$ and $f(u, v) \in \mathbb{R}$ we adopt the convention that $c(u, v) - f(u, v) = \infty$.

Lemma 2. *If f is a feasible flow in G then there is a bijection between feasible flows in G and in G_f, defined by mapping a feasible flow f' in G to the flow $h = f' - f$ in G_f. This bijection restricts to a bijection on the sets of maximum flows in G and G_f.*

Proof. The functions $f' \mapsto f' - f$ and $h \mapsto h + f$ are mutually inverse bijections, so we just need to verify that they map feasible flows in G to feasible flows in G_f or vice-versa. Suppose f' and h are flows satisfying $h = f' - f$. We have the following chain of equivalences.

$$\begin{aligned} h \text{ is feasible in } G_f &\Leftrightarrow \forall u, v \;\; h(u, v) \leq c(u, v) - f(u, v) \\ &\Leftrightarrow \forall u, v \;\; h(u, v) + f(u, v) \leq c(u, v) \;\Leftrightarrow\; f' \text{ is feasible in } G \end{aligned}$$

where the last equivalence holds because $f' = h + f$. This completes the proof that flows in G and G_f are in bijective correspondence. The bijection preserves the difference in value between two flows, i.e. $\text{val}(h_1) - \text{val}(h_0) = \text{val}(h_1 + f) - \text{val}(h_0 + f)$, so in particular the bijection preserves the property of being a maximum flow. □

2 The Max-Flow Min-Cut Theorem

The *max-flow min-cut theorem* is an important result that 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 flow network $G = (V, s, t, c)$ 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:

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

A *minimum s-t cut* is an s-t cut of minimum capacity.

For any flow f and pair of vertex sets Q, R let $f(Q, R)$ denote the net flow from Q to R:

$$f(Q, R) = \sum_{u \in Q} \sum_{v \in R} f(u, v).$$

Note that the expression $f(Q, R)$ includes positive contributions from edges (u, v) with $u \in Q$, $v \in R$, $f(u, v) > 0$ and negative contributions from edges (v, u) with $u \in Q, v \in R$, $f(v, u) > 0$, since, in the latter case, $f(u, v) = -f(v, u)$. That is why it is best to describe $f(Q, R)$ as the *net* flow from Q to R: if 100 units of flow, in total, are flowing from vertices in Q to vertices in R, while 40 units of flow, in total, are flowing from vertices in R to vertices in Q, then $f(Q, R) = 100 - 40 = 60$.

Lemma 3. *If $G = (V, s, t, c)$ is a flow network, f is any flow in G, and S, T is any s-t cut, then* $\text{val}(f) = f(S, T)$. *If f is a feasible flow, then* $\text{val}(f) \leq c(S, T)$ *with equality if and only if every edge from S to T has zero residual capacity.*

Proof. It is clear from the definition of $f(Q, R)$ that it is additive under disjoint unions: if R is partitioned into disjoint sets R_1, R_2 then $f(Q, R) = f(Q, R_1) + f(Q, R_2)$. Also, it follows easily from skew-symmetry then $f(Q, Q) = 0$ for all Q:

$$f(Q, Q) = \sum_{u \in Q} \sum_{v \in Q} f(u, v) = \frac{1}{2} \sum_{u \in Q} \sum_{v \in Q} [f(u, v) + f(v, u)] = 0.$$

These two properties of the function $f(Q, R)$ justify the first line of the following calculation, where S, T is an arbitrary s-t cut.

$$\begin{aligned} f(S, T) &= f(S, T) + f(S, S) = f(S, V) \\ &= \sum_{u \in S} \sum_{v \in V} f(u, v) \\ &= \sum_{v \in V} f(s, v) = \text{val}(f). \end{aligned}$$

The last line follows from the one above by applying the flow conservation equation at every $u \in S \setminus \{s\}$.

Now supposing f is feasible, we have $f(u, v) \leq c(u, v)$ for every u, v. Summing these inequalities over all pairs $u \in S$, $v \in T$ we find that $f(S, T) \leq c(S, T)$, with equality if and only if $f(u, v) = c(u, v)$ for all $u \in S$, $v \in T$. Recalling that $f(S, T) = \text{val}(f)$, this proves the second part of the lemma. □

Definition 5. If $G = (V, s, t, c)$ is a flow network and f is a feasible flow in G, an augmenting path with respect to f is a path from s to t whose edges all have strictly positive residual capacity.

As an example of residual graphs and augmenting paths, suppose we have a bipartite graph $G_0 = (L, R, E)$ and consider the flow network $G = (V, s, t, c)$ with vertices and capacities defined as follows. (When referring to edges of G_0, we will adopt the convention that

we always represent such edges as ordered pairs (u, v) with $u \in L$, $v \in R$.)

$$V = \{s, t\} \cup L \cup Rc(u, v) \qquad = \begin{cases} 1 & \text{if } u = s, v \in L \text{ or } (u, v) \in E \text{ or } u \in R, v = t \\ -1 & \text{if } u \in L, v = s \text{ or } (v, u) \in E \text{ or } u = t, v \in R \\ 0 & \text{otherwise.} \end{cases}$$

For every edge $e = (u, v) \in E$ there is a corresponding s-t path $P(e)$ in G that visits the vertices s, u, v, t in sequence. For any matching M in G_0 there is a corresponding feasible flow $f = \sum_{e \in M} f^{P(e)}$ in G. Let F denote the set of vertices of G_0 that are free in M. In the residual graph G_f, the following edges have positive residual capacity.

- edges (u, v) such that $(u, v) \in E \setminus M$
- edges (v, u) such that $(u, v) \in M$
- edges (s, u) such that $u \in L \cap F$
- edges (v, t) such that $v \in R \cap F$
- edges (u, s) such that $u \in L \setminus F$
- edges (t, v) such that $v \in R \setminus F$

Hence, the structure of G_f encodes the edge set of the residual graph G_M (as the set of directed edges with positive residual capacity having endpoints in $L \cup R$) as well as information about which vertices are free in M (in the form of the orientations of edges with positive residual capacity incident to s and t). In particular, every augmenting path in G_f is formed from an M-augmenting path P_0 in G_0 by sequencing the vertices of P_0 to start in $L \cap F$ and end in $R \cap F$, then prepending s to the start of the sequence and appending t to the end of the sequence. Conversely, if we take any M-augmnting path and applying this construction it always yields an augmenting path in G_f.

The following theorem asserts four equivalent conditions characterizing maximum flows in a network. The equivalence of the first and fourth conditions is usually called the max-flow min-cut theorem.

Theorem 4 (Max-flow Min-cut)**.** *For any flow network $G = (V, s, t, c)$ and feasible flow f, the following are equivalent.*

1. *f is a maximum flow in G.*
2. *There is no augmenting path with respect to f.*
3. *There exists an s-t cut S, T such that $c(S, T) = \text{val}(f)$.*
4. *The value of f equals the capacity of any minimum s-t cut.*

Proof. To prove (1) implies (2) it is convenient to establish the contrapositive. Let P be an augmenting path with respect to f, and let $\delta(P) > 0$ be the minimum residual capacity among the edges of P. The flow $f + \delta(P) \cdot f^P$ is feasible and has value $\text{val}(f + \delta(P) \cdot f^P) = \text{val}(f) + \delta(P)$, which is strictly greater than $\text{val}(f)$, so f is not a maximum flow.

The proof that (2) implies (3) is constructive. Define an *augmenting walk* to be any sequence of vertices $s = u_0, u_1, \ldots, u_k$ such that (u_i, u_{i+1}) has positive residual capacity for all $i < k$. Assuming f has no augmenting path, let S be the set of all vertices $u \in V$ such that there is a augmenting walk ending at u, and let T be the complement of S. By assumption, $t \notin S$, whereas $s \in S$ because the one-element sequence (u_0) is trivially an augmenting walk. Hence, S, T indeed forms an s-t cut. Now, consider any $u \in S$, $v \in T$. Since $u \in S$ there is an augmenting walk $s = u_0, u_1, \ldots, u_k = u$. Since $v \in T$, the sequence $u_0, u_1, \ldots, u_k, v$ is *not* an augmenting walk, hence edge $(u, v) = (u_k, v)$ has zero residual capacity. We have proven that $f(u, v) = c(u, v)$ for all $u \in S$, $v \in T$. Summing this equation over all such pairs (u, v) implies $\text{val}(f) = c(S, T)$.

To prove that (3) implies (4), suppose f is a feasible flow and S, T is an s-t cut such that $\text{val}(f) = c(S, T)$. Now let S', T' denote any minimum s-t cut. By the definition of a minimum cut, $c(S', T') \leq c(S, T)$. On the other hand, by Lemma 3, $c(S', T') \geq \text{val}(f)$. Since $\text{val}(f) = c(S, T)$, we have established that $c(S', T') - c(S, T)$ is both non-negative and non-positive, i.e. it must equal zero.

To prove that (4) implies (1), suppose that f is a feasible flow, S, T is a minimum s-t cut, and $\text{val}(f) = c(S, T)$. If f' is any other feasible flow, we have $\text{val}(f') \leq c(S, T)$ by Lemma 3. Hence $\text{val}(f') \leq \text{val}(f)$ for every feasible flow f', i.e. f is a maximum flow. □

3 The Ford-Fulkerson Algorithm

The first paragraph of the proof of Theorem 4 constitutes the basis for the Ford-Fulkerson algorithm, which computes a maximum flow iteratively, by initializing $f = 0$ and repeatedly replacing f with $f + \delta(P) \cdot f^P$ where P is an augmenting path with respect to 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 Theorem 4 guarantees that f is a maximum flow.

In all of the algorithms presented in this section, we adopt the convention that the edge set of G_f is considered to be the set pairs (u, v) with strictly positive residual capacity. In particular, under this interpretation of the edge set of G_f, the term "augmenting path" becomes synonymous with "s-t path in G_f".

Algorithm 1 FordFulkerson(G)

```
f ← 0; G_f ← G
while G_f contains an s-t path P do
    Let P be one such path.
    Let δ(P) = min{c(u,v) − f(u,v) | (u,v) an edge of P}.
    f ← f + δ(P) · f^P                    // Augment f using P.
    Update G_f.
end while
return f
```

Theorem 5. *In any flow network with edge capacities in $\mathbb{N} \cup \{\infty\}$ and with finite minimum cut capacity, any execution of the Ford-Fulkerson algorithm terminates and outputs an integer-valued maximum flow, f^*, after at most $\mathrm{val}(f^*)$ iterations of the main loop.*

Proof. At any time during the algorithm's execution, the residual capacities $c(u, v) - f(u, v)$ 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 $\mathrm{val}(f)$ increases by at least 1 during each loop iteration, so the algorithm terminates after at most $\mathrm{val}(f^*)$ loop iterations, where f^* denotes the output of the algorithm. Finally, Theorem 4 ensures that f^* must be a maximum flow because, by the algorithm's termination condition, its residual graph has no augmenting path. □

Regarding the running time of the Ford-Fulkerson algorithm, we will assume the flow network $G = (V, s, t, c)$ has n vertices and $m \geq n - 1$ edges with non-zero capacity. Flows f and residual graphs G_f will be represented in $O(m)$ space by storing the flow values $f(u, v)$ and residual capacities $c_f(u, v) = c(u, v) - f(u, v)$ only for those pairs (u, v) such that at least one of $c(u, v)$, $c(v, u)$ is strictly positive. Each iteration of the Ford-Fulkerson main loop can be implemented in $O(m)$ time, i.e. the time required to search for the augmenting path P in G_f (using breadth-first or depth-first search) and to construct the new residual graph after updating f. In integer-capacitated graphs, we have seen that the Ford-Fulkerson algorithm runs in at most $\mathrm{val}(f^*)$ linear-time iterations, where f^* is a maximum flow, hence the algorithm's running time is $O(m \,\mathrm{val}(f^*))$.

4 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.

Later 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.

4.1 Digression: Rational vs. Irrational Capacities

If one runs the Ford-Fulkerson algorithm in a network $G = (V, s, t, c)$ whose capacities are rational numbers, and the minimum cut capacity in G is finite, then the algorithm always terminates. This is because if the capacities are all multiples of $1/k$, then there is a related flow network $k \cdot G = (V, s, t, k \cdot c)$ with integer-valued capacities, and the operation of scaling by k defines a bijection between executions of Ford-Fulkerson in G and in $k \cdot G$. Since we have proven in Section 3 that every execution in network $k \cdot G$ terminates, the same must be the case for network G.

However, 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.

Example 1. Let G be a flow network with vertex set $V = \{s, t\} \cup \{u_i, v_i \mid i = 0, 1, 2\}$ and with capacities defined as follows.

- For $i \in \{0, 1, 2\}$ there are infinite capacity edges (s, u_i), (v_i, u_i), (v_i, t).
- For all pairs of distinct indices $i, j \in \{0, 1, 2\}$ there are infinite capacity edges (u_i, u_j) and (v_i, v_j).
- Let r denote the positive root of the quadratic equation $r^2 + r = 1$, i.e. $r = \frac{1}{2}(\sqrt{5} - 1)$. The edges (u_0, v_0) and (u_1, v_1) have capacities r and 1, respectively.
- All other capacities are zero.

If P_0, P_1 denote the paths s, u_0, v_0, t and s, u_1, v_1, t, respectively, the maximum flow in G is $rf^{P_0} + f^{P_1}$, with value $r + 1$, the golden ratio. An execution of the Ford-Fulkerson algorithm that chooses augmenting paths P_0, P_1, in either order, will discover this maximum flow in just two iterations of the main loop. However, it is also possible to construct a non-terminating execution of the Ford-Fulkerson algorithm using the circuitous augmenting paths

$$P_{012} = s, u_0, v_0, v_1, u_1, u_2, v_2, t$$
$$P_{120} = s, u_1, v_1, v_2, u_2, u_0, v_0, t$$
$$P_{201} = s, u_2, v_2, v_0, u_0, u_1, v_1, t$$

as we now demonstrate.

First, observe that every edge of P_{120} has positive capacity in G. In fact, the only two finite-capacity edges of P_{120} are (u_1, v_1) an (u_0, v_0) with capacities 1 and r, respectively. Hence, $\delta(P_{120}) = r$. If we augment path P_{120}, we will obtain a residual graph in which the infinite-capacity edges still have infinite residual capacity, and the only two edges with non-zero finite residual capacity are edges (u_1, v_1) and (u_2, v_2). Their residual capacities are

$$c_f(u_1, v_1) = 1 - r = r^2$$
$$c_f(u_2, v_2) = 0 + r = r.$$

In other words, in one iteration we have gone from a network whose three middle edges (u_0, v_0), (u_1, v_1), (u_2, v_2) have residual capacities $r, 1, 0$, respectively, to a network where the three middle edges have residual capacities $0, r^2, r$. This residual network is the same as the original flow network, up to rescaling capacities by r and permuting the index set $\{0, 1, 2\}$ cyclically. By induction, we may continue cycling through augmenting paths $P_{201}, P_{012}, P_{120}$ ad infinitum.

It is interesting that the existence of the non-terminating execution of the Ford-Fulkerson algorithm presented in Example 1 implies that $r = \frac{1}{2}(\sqrt{5} - 1)$ is irrational. This can be interpreted as an algorithmic proof of the irrationality of $\sqrt{5}$, in contrast to the usual number-theoretic proof that involves positing that $\sqrt{5} = \frac{p}{q}$ for relatively prime integers p, q and deriving a contradiction by reasoning about the divisibility of p and q by 5.

4.2 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) · f^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 $O(m^2 n)$.

4.3 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 6. 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 DINITZ(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s − t path P do
3:     f ← f + BLOCKINGFLOW(G_f)
4:     Update G_f
5: end while
6: return f

7: function BLOCKINGFLOW(G_f)
8:     h ← 0
9:     Let G' be the subgraph composed of advancing edges in G_f.
10:    Initialize c'(e) = c_f(e) for each edge e in G'.
11:    Initialize stack with ⟨s⟩.
12:    repeat
13:        Let u be the top vertex on the stack.
14:        if u = t then
15:            Let P be the path defined by the current stack. // Now augment h using P.
16:            Let δ(P) = min{c'(e) | e ∈ P}.
17:            h ← h + δ(P) · f^P.
18:            c'(e) ← c'(e) − δ(P) for all e ∈ P.
19:            Delete edges with c'(e) = 0 from G'.
20:            Let (u, v) be the newly deleted edge that occurs earliest in P.
21:            Truncate the stack by popping all vertices above u.
22:        else if G' contains an edge (u, v) then
23:            Push v onto the stack.
24:        else
25:            Delete u and all of its incoming edges from G'.
26:            Pop u off of the stack.
27:        end if
28:    until stack is empty
29:    return h
30: end function
```

Dinitz's algorithm initializes $f = 0$ and repeatedly updates f by adding a blocking flow, until no augmenting paths remain. Later we will discuss and analyze the algorithm for computing 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

Algorithm 4 BLOCKINGFLOW(G_f)

```
1: h ← 0
2: Let G′ be the subgraph composed of advancing edges in G_f.
3: Initialize c′(e) = c_f(e) for each edge e in G′.
4: Initialize stack with ⟨s⟩.
5: repeat
6:     Let u be the top vertex on the stack.
7:     if u = t then
8:         Let P be the path defined by the current stack.      // Now augment h using P.
9:         Let δ(P) = min{c′(e) | e ∈ P}.
10:        h ← h + δ(P) · f^P.
11:        c′(e) ← c′(e) − δ(P) for all e ∈ P.
12:        Delete edges with c′(e) = 0 from G′.
13:        Let (u, v) be the newly deleted edge that occurs earliest in P.
14:        Truncate the stack by popping all vertices above u.
15:    else if G′ contains an edge (u, v) then
16:        Push v onto the stack.
17:    else
18:        Delete u and all of its incoming edges from G′.
19:        Pop u off of the stack.
20:    end if
21: until stack is empty
22: return h
```

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.

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 $O(mn^2)$.

5 The Push-Relabel Algorithm

In this section we present an algorithm to compute a maximum flow in $O(n^3)$ time. Unlike the algorithms presented in earlier lectures, this one is *not based on augmenting paths.* Augmenting-path algorithms maintain a feasible flow at all times and terminate when the

residual graph has no $s - t$ path. The push-relabel algorithm maintains the invariant that the residual graph contains no $s - t$ path, and it terminates when it has found a feasible flow. The state of the algorithm before terminating is described by a more general structure called a *preflow*.

Definition 7. A *preflow* in a flow network $G = (V, E, c, s, t)$ is a function $f : V^2 \to \mathbb{R}$ that satisfies

1. **skew-symmetry:** $f(u, v) = -f(v, u)$ for all $u, v \in V$
2. **semi-conservation:** $\sum_{u \in V} f(u, v) \geq 0$ for all $v \neq s$
3. **capacity:** $f(u, v) \leq c(u, v)$ for all $u, v \in V$.

The non-negative quantity $x(v) = \sum_{u \in V} f(u, v)$ is called the *excess* of v with respect to f.

Note that a preflow is a flow if and only if every vertex except s and t has zero excess. The preflow-push algorithm works by always pushing flow away from vertices with positive excess. This is done using an operation PUSH(v, w) that pushes enough flow on edge (v, w) to either saturate the edge or remove all of the excess at v. The former case is called a *saturating push*, the latter is a *push*.

PUSH(v, w):
  $\delta \leftarrow \min\{x(v), r(v, w)\}$
  $f(v, w) \leftarrow f(v, w) + \delta$
  $f(w, v) \leftarrow f(w, v) - \delta$

Note that the quantity δ in the PUSH operation is carefully chosen to ensure that if f is a preflow before performing PUSH(v, w) then it remains a preflow afterward. This is because $x(v)$ decreases by δ, hence it cannot become negative, and $f(v, w)$ increases by δ, hence it cannot exceed $f(v, w) + r(v, w) = c(v, w)$.

To keep track of where and when to push flow in the network, and to ensure that flow is going toward the sink, the algorithm makes use a *height function* taking non-negative integer values. The height function will satisfy the following invariants.

1. **boundary conditions:** $h(s) = n$, $h(t) = 0$;
2. **steepness condition:** for all edges (v, w) in the residual graph G_f, $h(v) \leq h(w) + 1$.

The following two lemmas underscore the importance of the height function invariants.

Lemma 6. *If f is a flow, h is a height function satisfying the steepness condition, and $v_0, v_1, \ldots, v_k$ is a path in the residual graph G_f, then $h(v_0) \leq h(v_k) + k$.*

Proof. The proof is by induction on k. When $k = 0$ the lemma holds vacuously. For $k > 0$, the induction hypothesis and the steepness condition imply $h(v_0) \leq h(v_1) + 1 \leq h(v_k) + (k - 1) + 1$, and the lemma follows. □

Lemma 7. *If f is a flow and h is a height function satisfying the boundary and steepness conditions, then f is a maximum flow.*

Proof. To prove that f is a maximum flow it suffices to prove that G_f has no path from s to t. Since G_f has only n vertices, every simple path $v_0, \ldots, v_k$ in G_f satisfies $k \leq n-1$ and hence, by Lemma 6, $h(v_0) \leq h(v_k) + n - 1$. The boundary condition now implies that the endpoints of the path cannot by s and t. □

The following algorithm, known as the push-relabel algorithm, computes a maximum flow by maintaining a preflow f and height function h satisfying the boundary and steepness conditions. The flow f is modified by a sequence of PUSH operations, and the height function h is modified by a sequence of RELABEL operations, each of which increments the height of a vertex to enable future push operations without risking a violation of the steepness condition. (To see why PUSH(v, w) may risk violating the steepness condition, note that it may introduce a new edge (w, v) into the residual graph. Hence, PUSH(v, w) should only be applied when $h(v) \geq h(w) - 1$.)

Algorithm 5 Push-Relabel Algorithm

Initialize $h(s) = n$ and $h(v) = 0$ for all $v \neq s$.
Initialize $f(u, v) = \begin{cases} c(u, v) & \text{if } u = s \\ -c(v, u) & \text{if } v = s \\ 0 & \text{otherwise.} \end{cases}$
Initialize $x(s) = 0$ and $x(v) = c(s, v)$ for all $v \neq s$.
while there exists v such that $x(v) > 0$ **do**
    Pick v of maximum height among the vertices with $x(v) > 0$.
    if there exists w such that $r(v, w) > 0$ and $h(v) > h(w)$ **then**
        PUSH(v, w)
    else
        $h(v) \leftarrow h(v) + 1$
    end if
end while
return f

By design, the algorithm maintains the invariants that f is a preflow and h satisfies the boundary and steepness conditions. Hence, if it terminates, by Lemma 7 it must return a maximum flow. The remainder of the analysis is devoted to proving termination and bounding the running time. Our first task will be to bound the heights of vertices with positive excess.

Lemma 8. *If f is a preflow and v is a vertex with $x(v) > 0$, then G_f contains a path from v to s.*

Proof. Let A denote the set of all u such that G_f contains a path from u to s, and let

$B = V \setminus A$. Note that G_f contains no edges from B to A. We have

$$\begin{aligned}
\sum_{v \in B} x(v) &= \sum_{v \in B} \sum_{u \in V} f(u, v) \\
&= \sum_{v \in B} \sum_{u \in A} f(u, v) && \textit{(All other terms cancel, by skew-symmetry.)} \\
&= \sum_{v \in B} \sum_{u \in A} -f(v, u) \\
&\leq \sum_{v \in B} \sum_{u \in A} r(v, u) = 0,
\end{aligned}$$

which shows that the sum of excesses of the vertices in B is non-positive. Since $s \notin B$ and s is the only vertex that has negative excess, it follows that every vertex in B has zero excess. In other words, all of the vertices with positive excess belong to A, QED. □

Lemma 9. *If f is a preflow and h is a height function satisfying the boundary and steepness conditions, then $h(v) \leq 2n - 1$ for all v such that $x(v) > 0$.*

Proof. This follows directly from Lemmas 6 and 8 and the fact that $h(s) = n$. □

It's time to start bounding the number of operations the algorithm performs.

Relabelings. Since the graph has n vertices and the height of each one never exceeds $2n$, the number of relabel operations is bounded by $2n^2$.

Saturating pushes. Each time a saturating push occurs on edge (v, w), it is removed from G_f. Also, note that $\textsc{Push}(v, w)$ is only executed if $h(v) > h(w)$. In order for (v, w) to reappear as an edge of G_f, it must regain positive residual capacity through application of the operation $\textsc{Push}(w, v)$. However, in order for $\textsc{Push}(w, v)$ to take place, it must be the case that the height of w increased to exceed that of v, meaning that w was relabeled at least twice. Since w is relabeled at most $2n$ times in total, we conclude that edge (v, w) experiences at most n saturating pushes. Summing over all m edges of the graph and their reversals, the algorithm performs at most $2mn$ saturating pushes.

Non-saturating pushes. This is the hardest part of the analysis. To bound non-saturating pushes we define

$$H = \max\{h(v) \mid x(v) > 0\}$$

and divide the algorithm's execution into phases during which H is constant. In other words, each time the value of H changes, a phase ends and the next phase begins. Now, since H can only increase when a relabel operation takes place, the total amount by which H increases is bounded by $2n^2$. The H starts at 0 and is always non-negative, the total amount by which H decreases is also at most $2n^2$. Hence, the number of phases is bounded by $4n^2$. During a phase, we claim that each vertex experiences at most one non-saturating push. Indeed, during a phase we only perform $\textsc{Push}(v, w)$ if $h(v) = H$ and $x(v) > 0$. If the operation is

a non-saturating push then $x(v) = 0$ afterward, and the only way for v to acquire positive excess is if some other operation $\textsc{Push}(u, v)$ is later performed. However, for $\textsc{Push}(u, v)$ to be performed we would need to have $h(u) = H + 1$, implying that the next phase has already begun. Thus, during a phase there can be at most one non-saturating push per node, or n non-saturating pushes in total. As there are at most $4n^2$ phases, there can be at most $4n^3$ non-saturating pushes.

5.1 Epilogue: Faster Algorithms for Maximum Flow

Earlier we presented a simple implementation of Dinitz's algorithm running in time $O(mn^2)$. Using sophisticated data structures, Sleator and Tarjan discovered a way to implement Dinitz's algorithm to run in time $O(mn \log n)$. The push-relabel algorithm, presented above, 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, discovered in 2022 by Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva, is a randomized algorithm running in time $O(m^{1+o(1)} \log U)$ with high probability. That algorithm combines methods from continuous optimization (*interior point methods*) with novel randomized data structures. The ideas behind its design and analysis lie well beyond the scope of this course.

6 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.

6.1 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 8. 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 9. 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 10 (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. □

6.2 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 11 (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 $|A|$. □

6.3 Hall's Theorem

Theorem 12. *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. □

6.4 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 13. *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 14. *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 13. 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 $n = |F| + p(F)$, where $p(F)$ denotes the number of paths in 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 14 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 14) 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 (x_v, y_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. $\square$