Cornell University, Fall 2019	**CS 6820: Algorithms**
The Dinitz, Hopcroft-Karp, and Push-Relabel Algorithms	**30 Sep 2020**

These lecture notes present two closely related algorithms: Dinitz's blocking-flow algorithm for the maximum flow problem, and the Hopcroft-Karp bipartite maximum matching algorithm. Both algorithms are based on the same insight: a potentially wasteful aspect of earlier maximum flow and maximum matching algorithms is that they only use one augmenting path in each iteration, even if many augmenting paths are discovered in the process of searching the residual graph. As long as the augmenting paths discovered are *compatible*, meaning that one can augmenting all of them simultaneously without violating the flow or matching constraints, then it is efficient to augment a large set of compatible paths before rebuilding the residual graph and searching once again for additional augmenting paths.

Finally, in Section 3 we present a maximum flow algorithm that is not based on augmenting paths at all: the push-relabel algorithm, based on Karzanov's notion of a *preflow*.

1 Dinitz's Algorithm

Dinitz's Algorithm improves the Edmonds-Karp Algorithm by discovering a *blocking flow*, which is in some sense a *maximal* set of shortest augmenting paths that can be used simultaneously to update the current flow without violating capacity constraints.

Definition 1. 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 1 DINITZ(G)

```
f ← 0; G_f ← G
while G_f contains an s − t path P do
    Let h be a blocking flow in G_f.
    f ← f + h
    Update G_f
end while
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 2 BlockingFlow(G_f)

```
h ← 0
Let G′ be the subgraph composed of advancing edges in G_f.
Initialize c′(e) = r(e) (residual capacity of e) for each edge e in G′.
Initialize stack with ⟨s⟩.
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 δ(P) = min{c′(e) | e ∈ P}.
        h ← h + δ(P)1_P.
        c′(e) ← c′(e) − δ(P) for all e ∈ 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 push-relabel algorithm, presented in Section 3 below, 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.

2 The Hopcroft-Karp Algorithm

Recall that one can reduce the bipartite maximum matching problem to the maximum flow problem by transforming an undirected bipartite graph $G = (L, R, E)$ into a directed flow network with vertex set $V = \{s, t\} \cup L \cup R$ and unit-capacity edges (s, u) for all $u \in L$, (v, t) for all $v \in R$, and (u, v) for all (u, v) in the edge set of the original bipartite graph.

The Hopcroft-Karp Algorithm for bipartite maximum matching simply runs Dinitz's maximum flow algorithm on the flow network produced by this reduction. As we shall soon see, it is possible to prove a much better running time bound for this specialization of Dinitz's Algorithm. (Hopcroft and Karp discovered their algorithm independently of Dinitz's work, which was published in a Soviet math journal in 1970 and was not known in the West until its popularization by Shimon Even and Alon Itai in 1974. The Hopcroft-Karp Algorithm was independently discovered and analyzed by Alexander Karzanov in the Soviet Union; both papers were published in 1973.)

The improved running time bound for the Hopcroft-Karp stems from two observations, encapsulated in the following two lemmas.

Lemma 1. *The Hopcroft-Karp algorithm terminates after fewer than $2\sqrt{n}$ iterations of its outer loop.*

Proof. After the first $\sqrt{n}$ iterations of the outer loop are complete, the minimum length of an M-augmenting path is greater than $\sqrt{n}$. If M^* denotes a maximum matching, and the cardinality of M^* satisfies $|M^*| = |M| + k$, then the symmetric difference $M^* \oplus M$ contains at least k vertex-disjoint M-augmenting paths; if each of these paths has at least $\sqrt{n}$ vertices then it must be the case that $k \leq \sqrt{n}$. Each remaining iteration of the outer loop strictly increases $|M|$, hence there are fewer than $\sqrt{n}$ outer loop iterations remaining. □

Lemma 2. *When algorithm* BlockingFlow(G_f) *is called on a residual graph G_f whose edge capacities are $\{0, 1\}$-valued, it runs in $O(m)$ time.*

Proof. The algorithm BlockingFlow(G_f) performs three types of work: augmenting a path P, pushing vertices onto the stack, and popping vertices off the stack. The number of operations performed in augmenting a path P is proportional to the length of P; we charge these operations to the edges of the path, assigning a charge of $O(1)$ to each edge of P when the path is augmented. The number of operations performed when pushing a vertex v onto the stack or popping it off the stack is $O(1)$; we charge these operations to the edge (u, v), where u is the vertex directly beneath v in the stack when the operation is performed. In this way, each operation is charged to an edge. We will show that each edge is charged for only $O(1)$ operations. Summing over edges, this yields the $O(m)$ bound stated in the lemma.

The key observation is that the residual capacity of each edge in G_f is equal to 1. Hence, whenever BlockingFlow(G) augments a path P, it sets $\delta(P) = 1$, reduces the residual

capacity of each edge of P to zero, and deletes the edge from G'. Thus, each edge of G belongs to at most one augmenting path, and is charged only $O(1)$ for augmentation operations. The only other reason BlockingFlow(G) might pop v from the stack (other than using v in an augmenting path) is if it discovers that v has no remaining outgoing edges; when that happens, v and its incoming edges, including (u, v), are deleted from G'. Since we have argued that (u, v) is deleted the first time v is popped from the stack, it follows also that (u, v) is only pushed onto the stack at most once. We have shown that, in total, (u, v) participates in at most one push, at most one pop, and at most one path augmentation, resulting in $O(1)$ operations being charged to (u, v) as claimed. □

Theorem 3. *The Hopcroft-Karp Algorithm computes a maximum matching in a bipartite graph in $O(m\sqrt{n})$ time.*

Proof. The theorem is a direct consequence of Lemma 1, which bounds the number of loop iterations by $2\sqrt{n}$, and Lemma 2, which bounds the amount of time per iteration by $O(m)$. □

3 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 2. 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 4. *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 5. *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 4, $h(v_0) \leq h(v_k) + n - 1$. 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 3 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**

    $\textsc{Push}(v,w)$

  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 5 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 6. *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 7. *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 4 and 6 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.

4 Some combinatorial applications of maximum flow

This section illustrates some combinatorial applications of maximum flow algorithms and of the two main structural results we've seen about flows: the max-flow min-cut theorem and the flow integrality theorem.

4.1 An application of flow integrality to discrepancy minimization

Combinatorial discrepancy theory is concerned with the existence (or non-existence) of finite sets F such that the uniform distribution over F approximates a given probability distri-

bution as closely as possible. In this section we illustrate a typical application of the flow integrality theorem to discrepancy theory.

Suppose we are given a set $A = \{a_1, a_2, \ldots, a_m\}$ and two different partitions of A into nonempty sets: $A = B_1 \sqcup B_2 \sqcup \cdots \sqcup B_r$ and $A = C_1 \sqcup C_2 \sqcup \cdots \sqcup C_s$. We are also given numbers $k_1, k_2, \ldots, k_r$. Our goal is to select a set F that contains k_i elements from each set B_i, so that F appears "as random as possible" from the standpoint of its intersection with each C_j. To interpret the phrase "as random as possible", observe that a random k_i-element subset of B_i contains $k_i \cdot |B_i \cap C_j|/|B_i|$ elements of C_j in expectation. If we define

$$\phi_{ij} = k_i \cdot |B_i \cap C_j|/|B_i|$$
$$\Phi_j = \sum_{i=1}^{r} \phi_{ij}$$

then our goal will be to ensure that the selected set F satisfies

$$\forall j \quad \lfloor \Phi_j \rfloor \leq |F \cap C_j| \leq \lceil \Phi_j \rceil. \tag{1}$$

Two applications that motivate this problem are the following.

symposium speakers: Imagine you're organizing a symposium to celebrate your department's 50th anniversary. To construct the list of speakers, you plan to invite two Ph.D. graduates from each of the first five decades of the department's history, and you'd like the number of speakers in each research area to match, as closely as possible, the expected number that would be produced if you randomly sampled two Ph.D. graduates from each decade. This corresponds to letting A be the set of all Ph.D. graduates from the first five decades, partitioning these potential speakers according to decade (B_i) and research area (C_j), and setting $k_i = 2$ for all i.

carpool planning: *This example is drawn from Chapter 7, Exercise 27, of Kleinberg and Tardos's textbook "Algorithm Design."* A set of n people share a car over a period of d days. Let A denote the set of pairs (i, j) such that person j will ride in the car on day i. We wish to choose one driver for each day; this corresponds to letting B_i be the set of all ordered pairs in A whose first component is i, and setting $k_i = 1$ for all i. The goal is satisfy a fairness constraint that the number of times each person is required to drive the car is as close as possible to their "driving obligation," where each day is deemed to generate one unit of driving obligation divided equally among the occupants of the car that day. The fairness constraint is modeled by letting C_j be the set of ordered pairs in A whose second component is j, and noting that ϕ_{ij} equals $1/p$ if p people are scheduled to drive in the car on day i and person j is one of them, otherwise $\phi_{ij} = 0$. Hence the quantity Φ_j defined above matches the definition of "driving obligation" given in this paragraph.

Using the flow integrality theorem, we can prove that a set F satisfying the discrepancy constraints (1) always exists. We use a flow network G with the following vertices:

- source s, sink t

- vertex b_i representing the set B_i for $i = 1, \ldots, r$
- vertex c_j representing the set C_j for $j = 1, \ldots, s$
- additional "gadget vertices" $d_0, d_1, \ldots, d_s$.

The edges of G and their capacities are as follows:

- an edge (s, b_i) of capacity k_i for each i
- an edge (b_i, c_j) of capacity $\lceil \phi_{ij} \rceil$ for each i, j
- edge (c_j, d_j) and (d_j, t), both of capacity $\lfloor \Phi_j \rfloor$, for each j
- an edge (c_j, d_0) of capacity 1 for each j such that Φ_j is not an integer
- an edge (d_0, t) of capacity $\sum_{i=1}^{r} k_i - \sum_{j=1}^{s} \lfloor \Phi_j \rfloor$.

Note that the cuts $(\{s\}, V \setminus \{s\})$ and $(V \setminus \{t\}, \{t\})$ both have capacity $K = \sum_{i=1}^{r} k_i$, so the maximum flow value in the network is at most K. Furthermore, the network is designed to have a fractional flow of value K; the edges with positive flow values are as follows.

- $f(e) = c(e)$ for $e \in \{(s, b_i) : 1 \leq i \leq r\} \cup \{(d_j, t) : 0 \leq j \leq s\}$
- $f(b_i, c_j) = \phi_{ij}$
- $f(c_j, d_j) = \lfloor \Phi_j \rfloor$
- $f(c_j, d_0) = \Phi_j - \lfloor \Phi_j \rfloor$

This satisfies flow conservation at b_i because

$$\sum_{j=1}^{s} \phi_{ij} = \sum_{j=1}^{s} \frac{k_i \cdot |B_i \cap C_j|}{|B_i|} = \frac{k_i}{|B_i|} \sum_{j=1}^{s} |B_i \cap C_j| = \frac{k_i}{|B_i|} \cdot |B_i| = k_i$$

and it satisfies flow conservation at c_j because $\Phi_j = \sum_{i=1}^{r} \phi_{ij}$ by definition.

Since G has integer-valued capacities, the flow integrality theorem ensures that there exists an integer-valued maximum flow. We have seen that the maximum flow value in G equals K, so there exists an integer-valued flow, f^*, whose value is K. If we select a set F by choosing $f(b_i, c_j)$ elements from each set $B_i \cap C_j$, then F will have exactly k_i elements from each set B_i, and it will have at least $\lfloor \Phi_j \rfloor$ and at most $\lceil \Phi \rceil$ elements of C_j for each j, because the flow coming out of vertex c_j must saturate the edge (c_j, d_j) and must fit within the combined capacity of edges (c_j, d_j) and (c_j, d_0).

4.2 Combinatorial applications of the max-flow min-cut theorem

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 this section 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.

4.3 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 8. *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. $\square$

4.4 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 3. 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 4. 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 9 (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. □

4.5 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 10 (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 $|A|$. □

4.6 Hall's Theorem

Theorem 11. *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. □

4.7 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 12. *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 13. *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 12. 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 13 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 13) 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. $\square$