1. **The Basics**  
1.1. Graphs  2  
1.2. Isomorphisms and subgraphs  2  
1.3. Some applications of graph theory  3  
1.4. Some important graphs and basic constructions  4  
1.5. Vertex degrees and some counting formulas  5  
1.6. Paths, trails, walks and cycles (Diestel §§1.3–1.4)  6  
1.7. Trees and Forests (Diestel §1.5)  8  
1.8. Bipartite graphs (Diestel §1.6)  10  
1.9. Eulerian Graphs  11  
1.10. Matrices associated with graphs  12  
2. **Counting spanning trees (Not in Diestel)**  
2.1. Deletion and contraction  14  
2.2. The Matrix-Tree Theorem  15  
2.3. The Prüfer Code  18  
2.4. MSTs and Kruskal’s algorithm  20  
3. **Matchings and covers**  
3.1. Basic definitions  24  
3.2. Equalities among matching and cover invariants  25  
3.3. Augmenting paths  26  
3.4. Hall’s Theorem and consequences  30  
3.5. Weighted bipartite matching  31  
3.6. Stable matchings  34  
3.7. Nonbipartite matching  37  
4. **Connectivity, Cuts, and Flows**  
4.1. Vertex connectivity  41  
4.2. Edge connectivity  41  
4.3. The structure of 2-connected and 2-edge-connected graphs  45  
4.4. Counting strong orientations  47  
4.5. Menger implies edge-Menger  47  
4.6. Network flows  49  
4.7. The Ford-Fulkerson algorithm  53  
4.8. Corollaries of the Max-Flow/Min-Cut theorem  57
1. The Basics

1.1. Graphs.

**Definition 1.1.** A graph $G$ is a pair $(V,E)$, where $V$ is a (finite), nonempty set of **vertices** and $E$ is a (finite) set of **edges**. Each edge $e$ is given by an unordered pair of two (possibly equal) vertices $v,w$, called its **endpoints**.

Equivalent statements:

- $v,w$ are the **endpoints** of $e$; or
- $v,w$ are **joined** by the edge $e$; or
- $e = vw$.

Technically, this last notation should only be used when $e$ is the only edge joining $v$ and $w$. Note that $e = vw$ is equivalent.

Sometimes, we don’t want to bother to give the edge $e$ a name; it is enough to know that there exists some edge joining $v$ and $w$. Then we might say that $v,w$ are **adjacent** or are **neighbors**. (It’s tempting to say “connected” instead, but you should try to make a habit of resisting temptation, because that term properly means something else.)

Graphs can have **loops** (edges that join a vertex to itself) and **parallel edges** (edges with the same pairs of endpoints). Sometimes we want to exclude these possibilities, often because they are irrelevant. A graph with no loops and no parallel edges is called **simple**.

When studying graph theory, one quickly learns to be flexible about notation. For instance, when working with a single graph we want to use the concise symbols $V$ and $E$ for its vertex and edge sets; but if there are several different graphs around then it is clearer to write $V(G)$, $E(H)$, etc.

1.2. Isomorphisms and subgraphs. As in man fields of mathematics, one of our first orders of business is to say when two of the things we want to study are the same, and when one is a subthing of another thing.

**Definition 1.2.** Let $G,H$ be graphs. An **isomorphism** is a bijection $f : V(G) \rightarrow V(H)$ such that for every $v,w \in V(G)$,

$$\#\{\text{edges of } G \text{ joining } v,w\} = \#\{\text{edges of } H \text{ joining } f(v),f(w)\}.$$  

“$G \cong H$” means $G,H$ are isomorphic.

Notice that this has nothing to do with what the graph looks like on the paper. A **drawing of a graph is not the same as the graph itself**! These three graphs are all isomorphic to each other; the red numbers indicate the isomorphism.
Think of an isomorphism as a relabeling, which doesn’t really change the underlying structure of the graph.

**Definition 1.3.** An **isomorphism invariant** is a function $\psi$ on graphs such that $\psi(G) = \psi(H)$ whenever $G \cong H$. (Equivalently, a function on equivalence classes of graphs.)

For example, the number of vertices is an invariant, as is the number of edges — but not the number of crossings when you draw the graph (although the minimum number of crossings among all possible drawings is indeed an invariant). Nor is a property like “Every edge has one vertex labeled with an odd number and one vertex labeled with an even number,” since there’s nothing to prevent me from shuffling the numbers to make this false. On the other hand, “The graph can be labeled so that every edge has one odd and one even label” is an invariant.

It is always possible to draw a given graph in lots of different ways, many geometrically inequivalent. It is crucial to remember that a graph is a combinatorial object, not a geometric one. That is, the structure of a graph really is given by its list of vertices, edges and incidences, not by a drawing composed of points and lines.

**Definition 1.4.** Let $G$ be a graph. A **subgraph** of $G$ is a graph $H$ such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. For short we write $H \subseteq G$.

Note that it is not true that for every $X \subseteq V(G)$ and $F \subseteq E(G)$, the pair $(X, F)$ is a subgraph of $G$, because it might not even be a graph—it needs to satisfy the condition that every endpoint of an edge in $F$ belongs to $X$. (You can’t have an edge dangling in the breeze — there needs to be a vertex on each end of it.)

Every subset $F \subseteq E(G)$ determines a subgraph whose vertex set is the set of all vertices that are endpoints of at least one edge in $F$. Also, every subset $X \subseteq V(G)$ determines a subgraph $G[X]$, the **induced subgraph**, whose edge set is the set of all edges of $G$ with both endpoints in $X$. Being an induced subgraph is a stronger property than being a subgraph.

1.3. **Some applications of graph theory.** Graph theory has about a zillion applications. Here are a few.

*Discrete optimization:* a lot of discrete optimization problems can be modeled using graphs. For example, the TSP (traveling salesperson problem); the knapsack problem; matchings; cuts and flows.

*Discrete geometry and linear optimization:* the vertices and edges of a polytope $P$ form a graph called its **1-skeleton;** when using the simplex method to solve a linear programming problem whose feasible region (i.e., the set of legal, although perhaps not optimal, solutions) is $P$, the 1-skeleton of $P$ describes exactly the steps of the algorithm.
Algebra: the Cayley graph of a group $G$ is a graph whose edges correspond to multiplication by one of a given set of generators; basic group-theoretic notions such as relations, conjugation, etc. now have natural descriptions in terms of the Cayley graph.

Topology: you can study an infinite and therefore complicated topological space by replacing it with a finite simplicial complex (a generalized kind of graph) from which you can calculate properties of the original space; also, deep graph-theoretic concepts such as deletion/contraction often have topological analogues.

Theoretical computer science: many fundamental constructions such as finite automata are essentially glorified graphs, as are data structures such as binary search trees.

Chemistry: A molecule can be regarded as a graph in which vertices are atoms and edges are bonds. Amazingly, the chemical properties of a substance, such as its boiling point, can sometimes be predicted with great accuracy from the purely mathematical properties of the graph of the molecule!

Biology: More complicated structures like proteins can be modeled as graphs. The theory of rigidity of graphs has been used to understand how proteins fold and unfold.

Not to mention the wonderful applicability of graphs to all manner of subjects including forestry, communications networks, efficient garbage collection, and evolutionary biology.

1.4. Some important graphs and basic constructions. The path $P_n$ (Diestel: $P^n$) has $n$ vertices and $n - 1$ edges, connected sequentially. The cycle $C_n$ (Diestel: $C^n$) has $n$ vertices and $n$ edges and can be drawn as a polygon.

The complete graph $K_n$ (Diestel: $K^n$) has $n$ vertices and one edge between each pair of vertices. Thus there are \( \binom{n}{2} = \frac{n(n-1)}{2} \) edges in total. Often we assume that the vertex set is $[n] = \{1, 2, \ldots, n\}$. (This notation is standard in combinatorics.) A complete graph is also called a clique, particularly when it occurs inside another graph.

The complete bipartite graph $K_{p,q}$ has $p + q$ vertices, with $p$ of them painted red and $q$ painted blue, and an edge between each pair of differently colored vertices, for a total of $pq$ edges.
The empty graph or $\bar{K}_n$ consists of $n$ vertices and no edges. A copy of $\bar{K}_n$ appearing as an induced subgraph of a graph $G$ is the same as a set of vertices of $G$ of which no two are adjacent. Such a set is called a coclique (or independent set or stable set).

A few operations on graphs.

- If $G$ is a simple graph, its complement $\bar{G}$ is the graph obtained by toggling adjacency and non-adjacency.
- The underlying simple graph $G^u$ of any graph $G$ is obtained by deleting all loops and all but one element of each parallel class of edges. Note that the connectivity relation on $G^u$ is the same as that on $G$.
- The disjoint union $G + H$ is the union of $G$ and $H$, assuming that the vertex sets are disjoint. For example, $K_n + K_m = \overline{K}_{m+n}$ and $\overline{K}_{m,n} = K_m + K_n$.
- The join $G \ast H$ also has vertex set $V(G) \cup V(H)$, but this time we add every possible edge between a vertex of $G$ and a vertex of $H$.

1.5. **Vertex degrees and some counting formulas.** The number of vertices of a graph $G$ is its order, often written $n(G)$. The number of edges is its size, written $e(G)$. Often when we are talking about a single graph $G$, we will just write $n$ and $e$. Diestel uses $|G|$ for the order and $\|G\|$ for the size.

**Definition 1.5.** Let $G = (V,E)$ be a graph. The degree of a vertex $v$ in $G$, written $d(v)$ or $d_G(v)$, is the number of edges of $G$ having $v$ as an endpoint (counting loops twice). The minimum and maximum degrees of a vertex in $G$ are written $\delta(G)$ and $\Delta(G)$ (or $\delta$ and $\Delta$).

**Proposition 1.6 (Degree-Sum Formula / Handshaking Theorem).** For every graph $G$,

\[ \sum_{v \in V(G)} d(v) = 2e(G). \]

**Proof.** Each edge contributes 2 to each side of the equation. \qed

**Corollary 1.7.** Every graph has an even number of vertices of odd degree.

**Corollary 1.8.** For every vertex $v$, $\delta(G) \leq d(v) \leq \Delta(G)$, so

\[ \delta \leq \frac{2e}{n} \leq \Delta. \]

**Definition 1.9.** A graph $G$ is $d$-regular if every vertex has degree $d$.

In this case equality holds in Corollary 1.8.

**Corollary 1.10.** There are no regular graphs of odd degree and odd order.

**Example 1.11.** The cycle $C_n$ is 2-regular and the clique $K_n$ is $(n - 1)$-regular. An icosahedron has 12 vertices and is 5-regular, so $e = dn/2 = 5 \cdot 12/2 = 30$.

**Example 1.12.** The $n$-dimensional cube or hypercube $Q_n$ is defined as follows. Let $V = 2^n$ be the power set of $[n]$ (so in particular $|V| = 2^n$), and let $E = \{ST \mid |S \Delta T| = 1\}$, where $\Delta$ denotes symmetric difference. This graph is called the $n$-dimensional cube or hypercube $Q_n$. 

\[
\begin{align*}
Q_0 & \quad Q_1 \\
Q_2 & \quad Q_3
\end{align*}
\]
Note that $|V(Q_n)| = 2^n$ and it is regular of degree $n$ (why?). Therefore, $|E(Q_n)| = n2^{n-1}$.

Equivalently, you can regard the vertices of $Q_n$ as bit strings of length $n$, with two vertices adjacent if they agree in $n-1$ places. These two descriptions are isomorphic via associating a bit string $(b_1, \ldots, b_n)$ with the set $\{i \in [n] \mid b_i = 1\} \subseteq [n]$.

1.6. Paths, trails, walks and cycles (Diestel §§1.3–1.4).

Definition 1.13. Let $x, y \in V(G)$. A $x, y$-walk in $G$ is an alternating sequence of vertices and edges

$$x = v_0, e_0, v_1, e_1, \ldots, v_{n-1}, e_{n-1}, y = v_n$$

where $v_i, v_{i+1}$ are the endpoints of $e_i$ for all $i$. The length of the walk is the number of edges, namely $n$. The vertices $x, y$ are the endpoints; the other vertices are internal to the walk. The walk is trivial if $n = 0$.

It’s not always necessary to specify all this data; e.g., we could just give the starting vertex and a sequence of edges. Or, if $G$ has no parallel edges, we could just give the sequence of vertices.

Often we don’t care about what the internal vertices are — in this case we can write just $xWy$ (where technically $W$ stands for $e_0, v_1, \ldots, v_{n-1}, e_{n-1}$). This makes it easy to concatenate walks: if $xWy$ and $yW'z$ are walks, then so is $xWyW'z$. We’ll write $\ell(W)$ for the length of $W$.

Definition 1.14. A walk is closed if $v_0 = v_n$. A trail is a walk with no repeated edges. A path is a walk with no repeated vertices. A cycle is a closed path.

These definitions of “path” and ‘cycle” are consistent with the previous ones. A path in $G$ of length $n$ is the same thing as a subgraph of $G$ isomorphic to $P_{n+1}$, and a cycle of length $n$ is just a subgraph of $G$ isomorphic to $C_n$.

Paths are the nicest kind of walks. Frequently, we are in a situation where we know how to walk from $u$ to $v$, but what we really want is a $u, v$-path. Fortunately, if a walk is not a path, then it must contain some redundancy which can be eliminated, and repeating this process will eventually yield a path. To be precise:

Proposition 1.15. If $G$ has an $x, y$-walk, then it has an $x, y$-path.

Proof. Let $xWy$ be a walk. If some vertex $z$ occurs more than once, then $xWy$ has the form $xW'zW''y$, where $W'$ and $W''$ may be trivial, but $W''$ is not. But then $xW'zW''y$ is a strictly shorter $x, y$-walk (since its length is $\ell(W) - \ell(W'')$). Keep repeating this process until no further shortening is possible, which means that the walk is a path. □

Technically, the proof of Lemma 1.2.5 is an inductive argument, but I have phrased it instead as a recursive algorithm (which is really the same thing). The proof implies that every minimal-length walk is in fact a path.

Definition 1.16. Two vertices of $G$ are connected if there is a path in $G$ between them (equivalently, a walk). The graph $G$ is connected if every pair of vertices $u, v$ is connected. The (connected) components of $G$ are its maximal connected subgraphs. The number of components is denoted $c(G)$.

Note that any two adjacent vertices are connected, but not every two connected vertices are adjacent.

Proposition 1.17. The relation “$u$ is connected to $v$” is an equivalence relation on $V(G)$, whose equivalence classes are the vertex sets of the connected components of $G$.

Proof. Connectedness is reflexive (consider the trivial walk), symmetric (walks can be reversed), and transitive (walks can be concatenated). □
Proposition 1.18. Let $G$ be connected on $n$ vertices. Then the vertices can be labeled $v_1, \ldots, v_n$ so that every induced subgraph $G_j := G[v_1, \ldots, v_j]$ is connected, for $1 \leq j \leq n$. In addition, $v_1$ can be chosen arbitrarily.

Proof. Choose $v_1$ arbitrarily. Clearly $G_1 \cong K_1$ is connected. To construct $G_{j+1}$ from $G_j$, choose any vertex $x \notin \{v_1, \ldots, v_j\}$ and find a path from $v_1$ to $x$. Take $v_{j+1}$ to be the first vertex on this path not in $G_j$. □

Again, I have chosen to express the proof as an algorithm rather than a formal proof by induction.

Corollary 1.19. If $G$ is connected, then $e(G) \geq n(G) - 1$. More generally, $c(G) \geq n(G) - e(G)$ for all $G$.

Proposition 1.20. Let $a \in E(G)$, and let $G - a$ denote the graph obtained by removing $a$. If $a$ belongs to a cycle in $G$, then $c(G - a) = c(G)$. Otherwise, $c(G - a) = c(G) + 1$. In the latter case, $a$ is called a cut-edge or bridge or isthmus or coloop of $G$.

Proof. First, it is clear that every two vertices connected in $G - a$ are connected in $G$, so $c(G) \leq c(G - a)$.

Suppose that $a$ belongs to a cycle, and let $P$ be the path that constitutes the rest of the cycle. Then any two vertices that are connected in $G$ are connected in $G - a$, because $a$ can be replaced with $P$ in any walk. Therefore the connectivity relations on $G$ and $G - a$ are the same, and $c(G) = c(G - a)$.

Now suppose that $a$ does not belong to any cycle. Then its two endpoints cannot be connected by any path $P \subseteq G - a$, for then $P \cup a$ would be a cycle in $G$ containing $a$. So $c(G) > c(G - a)$. On the other hand, adding $a$ to $G - a$ can only join two components into one. So $c(G) = c(G - a) + 1$. □

By the way, a cut-vertex is a vertex $v$ such that $c(G - v) > c(G)$. (Synonyms: cutpoint, articulation point.) Here $G - v$ means the graph obtained by deleting all $v$ and all its incident edges; equivalently, $G - v = G[V(G) \setminus v]$.

Example 1.21. In the connected graph $G$ on the left below, $q$, $r$ and $s$ are cut-vertices; the others aren’t. Note that $c(G - q) = c(G - s) = 2$ but $c(G - r) = 3$. The cut-edges are $pq, qr, rv$. We have $c(G - a) = 2$ for each cut-edge $a$.

Note that a loop cannot be a cut-edge, nor can any edge that has another parallel edge.

Example 1.22. A cycle has no cut-vertices or cut-edges. On the other hand, every internal vertex of a path (but not either of the endpoints) is a cut-vertex, and every edge is a cut-edge.
1.7. Trees and Forests (Diestel §1.5).

**Definition 1.23.** A graph is **acyclic**, or a **forest**, if it has no cycles. By Proposition 1.20, this is equivalent to the condition that every edge is a cut-edge. A connected forest is a **tree**.

**Proposition 1.24.** A graph \(G\) is acyclic if and only if \(c(G) = n(G) - e(G)\). In particular, every tree \(T\) has \(n(T) - e(T) = 1\).

*Proof.* Start with the vertex set \(V(G)\) and no edges. This is certainly acyclic and \(c = n\) and \(e = 0\). Now add edges one by one. Each time you do so, \(e\) increases by 1 and \(c\) might or might not decrease by 1. If \(c\) ever stays constant, you just created a cycle. Otherwise, every edge is a cut-edge, which means that you didn’t create a cycle. \(\Box\)

The following corollary will be very useful (although not immediately).

**Corollary 1.25.** Every tree \(T\) with \(n \geq 2\) vertices has exactly at least two **leaves** (vertices of degree 1).

*Proof.* Handshaking says that \[\sum_{v \in V(T)} d_T(v) = 2e(T) = 2n - 2.\]

If a sum of \(n\) positive integers equals \(2n - 2\), then at least two of the summands must equal 1. \(\Box\)

Here are the three isomorphism classes of trees on 5 vertices:

![Three isomorphism classes of trees on 5 vertices](image)

**Theorem 1.26.** (Characterizations of trees; Diestel Thm. 1.5.1) Let \(G = (V, E)\) with \(n = |V|, e = |E|\). TFAE:

1. \(G\) is a tree (i.e., connected and acyclic).
2. \(G\) is connected and \(e = n - 1\).
3. \(G\) is acyclic and \(e = n - 1\).
4. \(G\) has no loops, and for every \(v, w \in V(G)\), there is exactly one \(v, w\)-path in \(G\).

*Proof.* We’ve already proved that \(G\) is acyclic if and only if \(e = n - 1\). It’s actually easy to prove from this that (1), (2), (3) are equivalent:

- If \(G\) is acyclic and connected, then \(e = 1 = n - 1\).
- If \(G\) is acyclic and \(e = n - 1\), then \(e = 1\), i.e., \(G\) is connected.
- If \(G\) is connected and \(e = n - 1\), then in fact \(c = 1 = n - e\), which means that \(G\) is acyclic.

The proof of \(4 \iff 1\) is left as an exercise. \(\Box\)

**Definition 1.27.** Let \(G\) be connected. A **spanning tree** is a tree \(T \subset G\) with \(V(T) = V(G)\). (More generally, a **spanning subgraph** of \(G\) is a subgraph with the same vertex set, i.e., a subgraph obtained by deleting edges but not deleting any vertices.)
Every connected graph has at least one spanning tree. For example, you can find one by labeling the vertices as in Prop. 1.18 and keeping only the \( n-1 \) edges that join \( v_{j+1} \) to a previous vertex, for each \( j \in [n-1] \). Or, you can repeatedly delete non-bridge edges until only a tree is left.

Some natural questions:

1. How many spanning trees does a given graph have? This number \( \tau(G) \) is an interesting measure of the complexity of the graph, and for many graphs there are amazing formulas for \( \tau(G) \).

2. How can you find the best spanning tree? Suppose each edge has a particular cost and you want to find the spanning tree that minimizes total cost.

We will come back to these things.

Frequently we want to think of one of the vertices \( r \) of a tree \( T \) as the root. In this case there is a partial order on vertices of the tree: \( x \preceq y \) if \( y \) lies on the unique \( xPr \) in \( T \) (i.e., \( xPr \) factors as \( xP'P'yP''r \)). For every \( x \neq r \), the vertex adjacent to \( x \) in \( xPr \) is called its parent, denoted \( p(x) \).

**Theorem 1.28.** Let \( G \) be a connected simple graph and let \( r \in V(G) \). There exists a spanning tree \( T \) with the property that for every \( x \in V(G) \), the \( rPx \) in \( T \) is of minimum length over all \( r,x \)-paths in \( G \). (Such a tree is called normal with respect to \( r \), or a breadth-first search tree.)

**Proof.** Here is some notation that will be useful. For each \( x \in V(G) \), let \( N[x] = N(x) \cup \{x\} \). (The letter \( N \) stands for “neighborhood”; the parentheses and square brackets are intended to suggest open and closed neighborhoods respectively.) In addition, define

\[
N^0[x] = \{x\}, N^2[x] = \bigcup_{y \in N[x]} N[y]
\]

\[
N^3[x] = \bigcup_{y \in N^2[x]} N[y]
\]

\[...
\]

\[
N^k[x] = \bigcup_{y \in N^{k-1}[x]} N[y]
\]

\[...
\]

Equivalently, \( N^k[x] \) is the set of vertices that are at distance at most \( k \) from \( r \) (i.e., are connected to \( r \) by a path of length at most \( k \)). Since \( G \) is connected and finite, we have \( N^k[x] = V(G) \) for sufficiently large \( k \).

Now, construct a spanning tree \( T \) with root \( r \) by the following algorithm.

0. Start by putting \( r \) in \( T \).
1. For every \( x \in N[r] \backslash \{r\} \), add the edge \( rx \). (So \( p(x) = r \) for all such \( x \).)
2. Each \( x \in N^2[r] \backslash N[r] \) has a neighbor \( y \) in \( N[r] \). Add the edge \( xy \), so that \( p(x) = y \).
3. Each \( x \in N^3[r] \backslash N^2[r] \) has a neighbor \( y \) in \( N^2[r] \). Add the edge \( xy \), so that \( p(x) = y \).

\[...
\]

By induction on \( k \), the vertices added at step \( k \) are exactly those at distance \( k \) from \( r \). In other words, \( T \) is normal with respect to \( r \).

Some remarks:

1. This definition of distance in fact makes \( G \) into a metric space.
(2) This algorithm can be souped up by assigning every edge $e$ a positive real number $\ell(e)$ (think of this as “length” in a metric sense), and then defining the distance between two vertices to be the shortest possible total length of a path between them. In this form it is known as Dijkstra’s algorithm, and is fundamental in computer science and discrete optimization. It is a theoretically efficient algorithm in the sense that its run time is polynomial in the numbers of vertices and edges.

1.8. Bipartite graphs (Diestel §1.6).

Definition 1.29. A graph $G$ is bipartite if $V(G) = X \cup Y$, where $X, Y$ are cocliques. That is, every edge has one endpoint in each of $X, Y$. The pair $X, Y$ is called a bipartition and the sets $X, Y$ themselves are partite sets or color classes. Also, we might say for short that $G$ is an $X, Y$-bigraph.

More generally, a graph $G$ is $k$-partite if its vertex set is the disjoint union of $k$ cocliques (also called partite sets).

- A graph is bipartite if and only if every one of its components is bipartite.
- A bipartite graph can’t contain any loops (parallel edges are OK).
- Any subgraph of a bipartite graph is bipartite.
- Even cycles are bipartite but odd cycles are not.
- $Q_n$ is bipartite. Remember that the edges of $Q_n$ are pairs $S, T \in 2^[[n]]$ with $|S \triangle T| = 1$. For this to happen, one of $S, T$ must have even cardinality and the other odd, so parity gives a bipartition.

Proposition 1.30 (Bipartite Handshaking). Let $G$ be an $X, Y$-bigraph. Then
\[ \sum_{v \in X} d(v) = \sum_{v \in Y} d(v) = e(G). \]

Corollary 1.31. If $G$ is a regular $X, Y$-bigraph, then $|X| = |Y|$ (and in particular $|V(G)|$ is even).

Bipartite graphs arise in lots of real-world applications, notably matching problems:

$X = \{\text{workers } w\}$, $Y = \{\text{shifts } s\}$,
$E = \{(w, s) : \text{worker } w \text{ is able to work shift } s\}$

$X = \{\text{job applicants } a\}$, $Y = \{\text{available jobs } p\}$,
$E = \{(a, p) : a \text{ is qualified for job } p\}$.

$V = \{\text{people}\}$, $E = \{\text{brother-sister pairs}\}$

$X = \text{women}$, $Y = \text{men}$

Lemma 1.32. Every closed odd walk contains an odd cycle.

Proof. Suppose that we have a closed odd walk that is not itself an odd cycle. Then it has some repeated vertex, so it has the form $xWxW\cdots$. But $\ell(W) + \ell(W')$ is odd, so exactly one of $W$ or $W'$ is odd (say $W$), which means that $xWx$ is a shorter closed odd walk. Repeating this, we eventually obtain an odd cycle. \(\square\)

Proposition 1.33. A graph is bipartite if and only if it contains no odd cycle.

Proof. Odd cycles are non-bipartite, so no bipartite graph can contain an odd cycle.

Now suppose that $G$ contains no odd cycle. We may as well assume that $G$ is connected. Fix a vertex $v$ and define

$X = \{x \in V(G) \mid G \text{ has an even } vP_\text{even}\}$,
$Y = \{y \in V(G) \mid G \text{ has an odd } vP_\text{odd}\}$. 

10
Then $X \cup Y = V(G)$ because $G$ is connected. If $x \in X \cap Y$, then we have a closed walk $vPxP'v$ with $\ell(P)$ even and $\ell(P')$ odd, but by the Lemma, this means that $G$ has an odd cycle, which is impossible. Hence $V(G) = X \cup Y$.

Suppose that two vertices $x, x' \in X$ are adjacent via an edge $a$. Then again we have a closed walk $vPax'P'v$, of odd length $\ell(P) + \ell(P') + 1$ (since $P, P'$ are even), which again is a contradiction. Hence $X$ is a coclique. The same argument implies that $Y$ is a coclique (here $P, P'$ are both odd again so again $\ell(P) + \ell(P') + 1$ is odd). □

Note that this proof is essentially constructive: if $G$ is bipartite, you can construct it by picking a starting vertex, coloring it blue, and walking around the graph, toggling your color between blue and red at every step. (For that matter, you can test bipartiteness easily by doing exactly this and seeing if it works.)

One way to think about this: Odd cycles are the minimal obstructions to being bipartite.

**Corollary 1.34.** Acyclic graphs are bipartite.

**Proof.** If you have no cycles, you certainly have no odd cycles! □

1.9. **Eulerian Graphs.** Königsberg Bridge Problem (Euler, 1737)

---

**Definition 1.35.** A circuit (or tour) in a graph is a closed trail, i.e., a walk that ends where it started and does not repeat every edge. An Euler circuit of a graph is a circuit using every edge. A graph is Eulerian if it has an Euler circuit.

**Example:** $K_4$ is not Eulerian. $K_5$ is.

— Removing or adding loops does not affect whether or not a graph is Eulerian.
— If $G$ is Eulerian and disconnected, then it has at most one nontrivial component.
— So from now on, suppose that $G$ is loopless and connected.

**Theorem 1.36.** A connected graph $G$ is Eulerian if and only if it is an even graph, i.e., every vertex has even degree.

---

11
Proof. ( \( \implies \) ) Let \( W \) be an Euler tour. Then \( W \) leaves and enters each vertex the same number of times, and since it traverses each edge exactly once, every vertex must therefore have even degree.

( \( \impliedby \) ) Let \( W = x \cdots y \) be a trail of greatest possible length. I claim that \( W \) is in fact a circuit, i.e., \( x = y \). Indeed, if \( x \neq y \), then the number of edges of \( W \) incident to \( y \) is odd, but by the assumption that \( G \) is even, there must be at least one edge of \( G \setminus W \) incident to \( y \), which contradicts the assumption that \( W \) is of maximum length.

Now, suppose that \( W \) is not an Euler tour. Then there is some vertex \( v \) that has at least one edge in \( W \) and at least one edge \( e = vu \) not in \( W \). Say \( W = xW'W''x \); then \( uexW''xW'v \) is a trail — but it is longer than \( W \) which is a contradiction. \( \square \)

Another method of proof is a little more constructive. By induction on the number of edges, every even graph decomposes as a (edge-)disjoint union of cycles (since erasing the edges in a cycle preserves evenness), and the cycles can be glued together to produce an Euler tour.

There is a simple method, called Fleury’s algorithm, for constructing an Euler tour in an even connected graph. Start at any vertex and start taking a walk, erasing each edge after you traverse it. There is only one rule: cross a bridge only if it is the only option open to you.

1.10. Matrices associated with graphs. [One note: Prove that every tree has at least two leaves. I think I forgot this last time.]

Let \( G \) be loopless, \( V(G) = \{v_1, \ldots, v_n\} \), and \( E(G) = \{e_1, \ldots, e_r\} \).

Definition 1.37. The adjacency matrix is the \( n \times n \) matrix \( A = A(G) = [a_{ij}] \), where \( a_{ij} \) is the number of edges joining vertices \( i \) and \( j \). Note that \( A^T = A \).

Fix an orientation on \( E(G) \). That is, for each edge, call one of its vertices the head and the other the tail. What we have is now a directed graph, or digraph, which can be drawn by replacing each edge with an arrow pointing from the tail to the head.

Definition 1.38. The incidence matrix is the \( n \times r \) matrix \( B = B(G) = [b_{ve}] \), where

\[
b_{ve} = \begin{cases} 
1 & \text{if } v = \text{head}(e), \\
-1 & \text{if } v = \text{tail}(e), \\
0 & \text{otherwise.}
\end{cases}
\]

Example 1.39. Let \( G \) be the graph as follows (actually, it's the Königsberg bridge graph):

Then

\[
A(G) = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \quad B(G) = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 \end{bmatrix},
\]

12
Warning: Diestel defines these matrices $A$ and $B$ as living over $\mathbb{Z}_2$ instead of $\mathbb{R}$. This doesn’t affect the behavior of $A$ or $B$ appreciably, and it has the advantage of making the orientation irrelevant (since $1 = -1 \mod 2$). However, in order to work with $L$ you really have to work over $\mathbb{R}$.

**Theorem 1.40.** Let $G = (V, E)$ be a connected graph, $H = (V, S)$ a spanning subgraph, and let $B(H) = \{B_e \mid e \in H\}$ be the corresponding set of columns of $B = B(G)$.

- $H$ acyclic $\iff$ $B(H)$ linearly independent;
- $H$ connected $\iff$ $B(H)$ spans the column space of $B$;
- $H$ is a spanning tree $\iff$ $B(H)$ is a column basis for $B$;

**Proof.** First, notice that all of these properties are independent of the choice of orientation (since reorienting simply multiplies one or more columns by $-1$ without changing which sets of columns are linearly (in)dependent).

Suppose that $C$ is a cycle. Traverse the cycle starting at any point, and keep track of whether you walk forward or backward (i.e., with or against the arrows). Then

$$\sum_{\text{forward } e \in C} B_e - \sum_{\text{backward } e \in C} B_e = 0.$$ 

For example, consider the 6-cycle shown below, traversed clockwise.

```
   /\  \\
  /   \     \\
(+) a----b----(+) c
  \   /     \\
   \(+) /      \\
   /\  \\
  /   \     \\
(+) e----d----(+) f
```

Now, suppose that $H$ has only one edge $e$ incident to some vertex $x$. Then $B_e$ is the only column in $B(H)$ to have a nonzero entry in the $x$ row. Therefore, it is linearly independent in $B(H)$. By induction, it follows that if if $H$ is acyclic, then $B(H)$ is a linearly independent set (remove leaves one by one).

In particular, if $H$ is a spanning tree then the rank of $B(H)$ is $n - 1$. On the other hand, the rank of the entire incidence matrix $B$ is no more than $n - 1$, since it has $n$ rows and they are not linearly independent — their sum is zero. Hence every spanning tree corresponds to a column basis, and any edge set containing a spanning tree spans the column space.

Suppose that $H$ has $c$ components $H_1, \ldots, H_c$. Then the column spaces of $B(H_1), \ldots, B(H_c)$ are disjoint, so

$$\text{rank } B(H) = \sum_{i=1}^{c} \text{rank } B(H_i) = \sum_{i=1}^{c} n(H_i) - 1 = n - c$$

(since each $B(H_i)$ is connected). We have seen that rank $B = n - 1$, so $B(H)$ is a spanning set if and only if $H$ is connected. $\square$
2. Counting spanning trees (Not in Diestel)

2.1. Deletion and contraction. Let \( \mathcal{T}(G) \) denote the set of spanning trees of \( G \), and let \( \tau(G) = |\mathcal{T}(G)| \) be the number of spanning trees.

\[
\begin{array}{|c|c|}
\hline
G & \tau(G) \\
\hline
\text{any tree} & 1 \\
C_n & n \\
K_3 & 3 \\
K_4 & 16 \\
K_{2,3} & 12 \\
K_{3,3} & 81 \\
Q_3 & 384 \\
Pete & 2000 \\
\hline
\end{array}
\]

Definition 2.1. Let \( G \) be a graph and \( e \in E(G) \) an edge. The deletion \( G - e \) is the graph obtained by erasing \( e \), leaving its endpoints (and everything else) intact. The contraction \( G/e \) is obtained by erasing \( e \) and merging its endpoints into a single vertex. (Contraction is not defined if \( e \) is a loop.) Note:

\[
\begin{align*}
n(G - e) &= n(G), & n(G/e) &= n(G) - 1, \\
e(G - e) &= e(G) - 1, & e(G/e) &= e(G) - 1.
\end{align*}
\]

Two kinds of edges are special:

- If \( e \) is a loop, then it can’t belong to any spanning tree. So \( \mathcal{T}(G) = \mathcal{T}(G - e) \) and \( \tau(G) = \tau(G - e) \).
- If \( e \) is a bridge, then it belongs to every spanning tree (since you can’t have a connected spanning subgraph without it). In fact \( \tau(G) = \tau(G/e) \).

By contrast, each “ordinary” edge (one that is neither a loop nor a bridge) belongs to at least one spanning tree, but not to all spanning trees. More specifically:

Theorem 2.2. If \( e \in E(G) \) is not a loop, then \( \tau(G) = \tau(G - e) + \tau(G/e) \).

Proof. We will find bijections

\[
\begin{align*}
\{T \in \mathcal{T}(G) : e \notin E(T)\} & \rightarrow \mathcal{T}(G - e) \quad \text{and} \\
\{T \in \mathcal{T}(G) : e \in E(T)\} & \rightarrow \mathcal{T}(G/e).
\end{align*}
\]

The first bijection is the easy one: a spanning tree of \( G \) not containing \( e \) is the same thing as a spanning tree of \( G - e \).
For the second bijection, if $T$ is a spanning tree of $G$ containing $e$, then $T' = T/e$ is a spanning tree of $G/e$. Indeed, $T'$ is connected because $T$ is, and
\[ e(T') = e(T) - 1 = (n(G) - 1) - 1 = n(G/e) - 1. \]

On the other hand, given any spanning tree $T' \in \mathcal{T}(G/e)$, the corresponding edges of $T$, together with $e$ itself, form a spanning tree of $G$.

Remark 2.3. The recurrence even works when $e$ is a bridge (because $G - e$ is disconnected, hence has zero spanning trees) or even if $e$ is a loop (well, in a silly way: $G/e$ is undefined, so it isn’t even a graph and then doesn’t have any spanning trees).

Example: By repeatedly applying deletion/contraction, we can calculate $\tau(G)$ of any graph. Here’s the calculation for the “diamond graph” obtained by removing an edge from $K_4$.

![Diagram of a diamond graph](image)

We start by applying deletion/contraction to edge $a$. On the left side, $G - a$ has a bridge $b$, and contracting it gives a 3-cycle, which we know has $\tau = 3$. On the right side, $G/a$ has neither a loop nor a bridge, so we recurse again, deleting and contracting edge $b$. The deletion is another 3-cycle, and contracting gives a 2-cycle plus a loop. So $\tau(G/a) = \tau(G/a-b) + \tau(G/a/b) = 3 + 2 = 5$, and then $\tau(G) = \tau(G-a) + \tau(G/a) = 3 + 5 = 8$.

The bad news is that computing $\tau(G)$ by deletion/contraction takes exponential time, essentially $O(2^{e(G)})$, because each instance of the recursion contributes a factor of 2. So this is not a good way to compute $\tau(G)$ in practice, although there are some families of graphs for which you can find interesting recurrences for $\tau(G)$ (see problem set). In the next section we will see a computationally efficient way of calculating $\tau(G)$: the Matrix-Tree Theorem, which exploits linear algebra.

2.2. The Matrix-Tree Theorem.

Definition 2.4. The Laplacian matrix is the $n \times n$ matrix
\[ L = L(G) = BB^T \]
where $B$ is the incidence matrix. Note that $L = D - A$, where $D$ is the diagonal matrix of vertex degrees. Also, the choice of orientation does not affect the Laplacian.
Example: The graph $G = K_4 - e$ has

$$A = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix}.$$  

For the Königsberg bridge graph, $A, B, L$ are as follows:

$$A = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 & 1 & 0 \\ -1 & -1 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad L(G) = \begin{bmatrix} 3 & -2 & 0 & -1 \\ -2 & 5 & -2 & -1 \\ 0 & -2 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}.$$  

In general, for any loopless graph $G$, $L(G) = BB^T$ is a symmetric, positive-semi-definite matrix of rank $n(G) - e(G)$, with entries as follows:

$$\ell_{ij} = \text{dot product of } i^{th} \text{ and } j^{th} \text{ rows of } B = \begin{cases} d_G(i) & \text{if } i = j, \\ -m_{ij} & \text{if } i, j \text{ share } m_{ij} \text{ edges,} \\ 0 & \text{otherwise.} \end{cases}$$  

**Theorem 2.5** (Matrix-Tree Theorem, Kirchhoff 1845). Let $n \geq 2$, let $G$ be a loopless graph on vertex set $[n]$, let $i, j \in V(G)$, and let $L_{i,j}$ be the “reduced Laplacian” matrix obtained by deleting the $i^{th}$ row and $j^{th}$ column of $L(G)$. Then:

1. $\tau(G) = (-1)^{i+j} \det L_{i,j}$. (In particular, if $i = j$, then the sign is $+1$.)
2. Let the nonzero eigenvalues of $G$ be $\lambda_1, \ldots, \lambda_{n-1}$. Then

$$\tau(G) = \frac{\lambda_1 \cdots \lambda_{n-1}}{n}.$$  

Example: For $G = K_4 - e$, we have

$$L^{1,1}(G) = \begin{bmatrix} 3 & -1 & -1 \\ -1 & 2 & 0 \\ -1 & 0 & 2 \end{bmatrix}$$

and $\det L^{1,1} = 8 = \tau(G)$, which is the answer we had gotten by deletion-contraction.

OTOH, if you go ahead and diagonalize $L$ (which you always can, because it’s symmetric), you get

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

i.e., the eigenvalues are 4,4,2,0, which says that the number of spanning trees is $4 \times 4 \times 2/4 = 8$.

Note: The number of nonzero eigenvalues, counting multiplicities, is always $n - 1$ (provided $G$ is connected), and they are always positive real numbers (because $L(G)$ is symmetric). But they don’t have to be integers.
Example 2.6. Cayley’s formula says that \( \tau(K_n) = n^{n-2} \). This can be proven by the Matrix-Tree Theorem.

\[
L = L(K_n) = \begin{bmatrix}
n - 1 & -1 & \cdots & -1 \\
-1 & n - 1 & \cdots & -1 \\
\vdots & \vdots & \ddots & \vdots \\
-1 & -1 & \cdots & n - 1
\end{bmatrix}
\]

and \( L^{n,n} \) looks the same, only it’s an \((n-1) \times (n-1)\) matrix. What is \( \det L^{n,n} \)? Don’t expand the determinant! Instead, think about what the eigenvectors might look like. The all-1’s vector is an eigenvector, with eigenvalue 1. And any vector with a 1 in one place and a -1 in another place is an eigenvector with eigenvalue \((n-1) - (-1) = n\). These span an eigenspace of dimension \( n - 2 \). Hence, by the MTT,

\[
\tau(K_n) = \det L^{n,n} = n^{n-2}.
\]

There are other graphs for which all eigenvalues are integers, such as \( Q_n \) and \( K_{m,n} \). But it’s a rare property.

Proof #1 of Matrix-Tree Theorem (i). Proceed by double induction on \( n = n(G) \) and \( r = e(G) \).

If \( n = 2 \), then \( G \) consists of \( r \) parallel edges, any one of which is a spanning tree. So \( \tau(G) = r \). Meanwhile,

\[
L = \begin{bmatrix} r & -r \\ -r & r \end{bmatrix}, \quad L^{1,1} = L^{2,2} = [r], \quad L^{1,2} = L^{2,1} = [-r].
\]

If \( r = 0 \), then \( G \) is not connected, so \( \tau(G) = 0 \), and meanwhile \( L(G) \) is the zero matrix.

Now suppose that \( n > 2 \) and \( r > 0 \), and that the MTT holds for all graphs with either fewer vertices, or \( n \) vertices and fewer edges. Let \( e \in E(G) \); assume WLOG that its endpoints are 1, \( n \). Note that \( L(G) \) and \( L(G - e) \) are almost the same:

\[
\ell_{i,j}^{G-e} = \left[L(G - e)\right]_{ij} = \begin{cases} 
\ell_{i,j}^G - 1 & \text{if } i = j = 1 \text{ or } i = j = n, \\
\ell_{i,j}^G + 1 & \text{if } \{i,j\} = \{1,n\}, \\
\ell_{i,j}^G & \text{otherwise.}
\end{cases}
\]

When we delete the \( n^{\text{th}} \) row and column, we obtain reduced Laplacians that differ only in one entry, namely

\[
\ell_{1,1}^G = \ell_{1,1}^{G-e} + 1.
\]

Therefore, if we evaluate each of \( \det \tilde{L}(G) \) and \( \det \tilde{L}(G-e) \) by expanding on the top row, then the calculations are almost the same; the difference is

\[
\det \tilde{L}(G) - \det \tilde{L}(G-e) = \begin{vmatrix}
\ell_{2,2}^G & \cdots & \ell_{2,n-1}^G \\
\vdots & \ddots & \vdots \\
\ell_{n-1,2}^G & \cdots & \ell_{n-1,n-1}^G
\end{vmatrix}.
\]

But that matrix is precisely the reduced Laplacian of \( G/e \) obtained by deleting the row and column indexed by the merged vertex. (The degrees of nonmerged vertices aren’t affected by the contraction, nor are edges between two nonmerged vertices.) Thus

\[
\det \tilde{L}(G) = \tau(G - e) + \tau(G/e) = \tau(G)
\]

by induction and the deletion-contraction recurrence (Theorem 2.2). \( \square \)

Proof #2 of Matrix-Tree Theorem (ii). This proof uses the Binet-Cauchy Theorem, a linear algebra fact that we will use as a black box.
Binet-Cauchy Formula:
Let \( m \geq p \), \( A \in \mathbb{R}^{p \times m} \), \( B \in \mathbb{R}^{m \times p} \), so \( AB \in \mathbb{R}^{p \times p} \).
For \( S \subset [m] \), \( |S| = p \), let
\( A_S = p \times p \) submatrix of \( A \) with columns \( S \)
\( B_S = p \times p \) submatrix of \( B \) with rows \( S \)
Then:
\[
\det AB = \sum_S (\det A_S)(\det B_S).
\]

Let \( N \) be the “reduced incidence matrix” formed by deleting a row from the signed incidence matrix \( M \). Observe that \( NN^T = L_{1,1} \).

Let \( S \) be a set of \( n-1 \) edges of \( G \), and consider the corresponding columns of \( N \). Note that \( S \) either contains a cycle or is a spanning tree. As noted before:

- If \( S \) contains a cycle, then the columns are linearly dependent.
- If \( S \) is acyclic (hence is a spanning tree), then every \( (n-1) \times (n-1) \) submatrix of \( N \) with columns \( S \) has determinant \( \pm 1 \).

Now we can apply Binet-Cauchy with \( p = n-1 \), \( m = e \), \( A = N \), \( B = N^T \).

\[
\det L_{1,1}^{11}(G) = \det NN^T = \sum_{S \subseteq E(G): |S| = n-1} (\det N_S)(\det N_S^T) \quad \text{(Binet-Cauchy)}
\]
\[
= \sum_S (\det N_S)^2
\]
\[
= \sum_S \begin{cases} 
1 & \text{if } S \text{ is a tree} \\
0 & \text{if it isn’t} 
\end{cases} = \tau(G).
\]

\[\square\]

2.3. The Prüfer Code.

**Theorem 2.7.** There is a bijection
\[ P : \mathcal{F}_n \to [n]^{n-2} = \{(p_1, \ldots, p_{n-2}) : p_i \in [n]\}, \]
called the Prüfer code, such that for every vertex \( v \),
\[ \deg_T(v) = 1 + \#\{i \in [n-2] : p_i = v\}. \]

Cayley’s formula \( \tau(K_n) = n^{n-2} \) is an immediate corollary, as is an even more refined count of trees called the Cayley-Prüfer formula. Here’s the idea:

- Peel off leaves, one by one, choosing the smallest available leaf each time.
- Keeping track of which leaf is deleted is not enough information to recover the tree; we need to keep track of the stem (unique neighbor of the deleted leaf)
- The list of stems is enough information to recover the tree.

Here is a pseudocode algorithm for computing \( P(T) \):

**Input:** \( T \in \mathcal{F}_n \)

**Output:** \( P(T) \in [n]^{n-2} \)
\[ T_0 := T \]
for \( i \) from 1 to \( n - 2 \) do
\{
    \( y_i := \) smallest leaf of \( T_{i-1} \)
    \( p_i := \) unique neighbor ("stem") of \( y_i \)
    \( T_i := T_{i-1} - p_i \)
\}
\( P(T) = (p_1, \ldots, p_{n-2}) \)

Here it is in Sage:

```python
def PruferCode(T): ## assume that T is a tree on 2 or more vertices
    U = deepcopy(T)
    P = []
    while U.num_verts() > 2:
        Leaves = [v for v in U.vertices() if U.degree(v) == 1]
        y = min(Leaves)
        p = U.neighbors(y)[0]
        P.append(p)
        U.delete_vertex(y)
    return P
```

Before proving that this algorithm gives a bijection, let’s do an example. Let \( n = 8 \) and let \( T \) be the tree shown.

![Tree Diagram](image)

**Step 1:** Leaves: 2, 5, 6, 7. Delete \( y_1 = 2 \), write down \( \ell_1 = 8 \).
**Step 2:** Leaves: 5, 6, 7. Delete \( y_2 = 5 \), write down \( \ell_2 = 4 \).
**Step 3:** Leaves: 4, 6, 7. Delete \( y_3 = 4 \), write down \( \ell_3 = 1 \).
**Step 4:** Leaves: 1, 6, 7. Delete \( y_4 = 1 \), write down \( \ell_4 = 8 \).
**Step 5:** Leaves: 6, 7. Delete \( y_5 = 6 \), write down \( \ell_5 = 3 \).
**Step 6:** Leaves: 3, 7. Delete \( y_6 = 3 \), write down \( \ell_6 = 8 \).

There’s only one edge left, so we are done; the Prüfer code is \((8, 4, 1, 8, 3, 8)\).

**Lemma 2.8.** For all \( v \in V(T) \), we have \( \deg_T(v) = 1 + \# \{ i : p_i = v \} \).

**Proof.** Every vertex \( v \) eventually becomes a leaf (either it is deleted, or it is one of the two remaining ones). To make \( v \) into a leaf, we need to remove \( \deg_T(v) - 1 \) of its neighbors, so \( v \) will occur exactly that many times in \( P(T) \).

**Proof of Theorem 2.7.** Now suppose you are given the Prüfer code of a tree \( T \in \mathcal{T}_8 \). I claim that you can reconstruct the leaves \( \ell_1, \ldots, \ell_n \), hence \( T \). We’ll do this with the running example \( P(T) = (p_1, \ldots, p_6) = (8, 4, 1, 8, 3, 8) \).
The first leaf deleted must have been \( \ell_1 = 2 \), because it is the smallest vertex not in \( P(T) \). I.e.,
\[
\ell_1 = \min \left( \{n\} \setminus \{p_1, p_2, \ldots, p_{n-2}\} \right).
\]

The second leaf deleted must have been \( \ell_2 = 5 \), because it is the smallest vertex that is not \( \ell_1 \) (hence is a vertex of \( T - \ell_1 p_1 \)) and does not appear in \( \{p_2, \ldots, p_{n-2}\} \) (hence is a leaf of \( T - \ell_1 p_1 \)). That is,
\[
\ell_2 = \min \left( \{n\} \setminus \{\ell_1, p_2, \ldots, p_{n-2}\} \right).
\]

By the same reasoning, the third leaf deleted must have been
\[
\ell_3 = \min \left( \{n\} \setminus \{\ell_1, \ell_2, p_3, \ldots, p_{n-2}\} \right).
\]
and in general, for all \( i \in [n-2] \), we have
\[
\ell_i = \min \left( \{n\} \setminus \{\ell_1, \ldots, \ell_i, p_{i+1}, \ldots, p_{n-2}\} \right).
\]

Thus \( \ell_1, \ldots, \ell_{n-2} \) are all distinct vertices, and the edges \( \ell_i p_i \) include all but one of the edges of \( T \). The other edge is the one left when the algorithm finishes; its endpoints are the two vertices that were never deleted, i.e., the elements of \( [n] \setminus \{\ell_1, \ldots, \ell_{n-2}\} \). So we can recover \( T \) from \( P(T) \). On the other hand, starting with an arbitrary sequence \( (p_i) \in [n]^{n-2} \) and constructing the sequence \( (\ell_i) \) yields a tree \( T \) such that \( P(T) = (p_i) \), so we have a bijection. \( \square \)

**Corollary 2.9** (Cayley-Prüfer Formula).
\[
\sum_{T \in \mathcal{F}_n} \prod_{j \in [n]} x_j^{d_T(j)} = x_1 \cdots x_n (x_1 + \cdots + x_n)^{n-2}.
\]

**Proof.** This is a straight calculation using Lemma 2.8.
\[
\sum_{T \in \mathcal{F}_n} \prod_{j \in [n]} x_j^{d_T(j)} = x_1 \cdots x_n \sum_{p=(p_1, \ldots, p_{n-2}) \in [n]^{n-2}} \prod_{i=1}^{n-2} x_{p_i} \\
= x_1 \cdots x_n \sum_{p_1 \in [n]} \sum_{p_2 \in [n]} \cdots \sum_{p_{n-2} \in [n]} \prod_{i=1}^{n-2} x_{p_i} \\
= x_1 \cdots x_n \left( \sum_{p_1 \in [n]} x_{p_1} \right) \left( \sum_{p_2 \in [n]} x_{p_1} \right) \cdots \left( \sum_{p_{n-2} \in [n]} x_{p_{n-2}} \right) \\
= x_1 \cdots x_n (x_1 + \cdots + x_n)^{n-2}.
\]

The idea of the Prüfer code can be extended to many other general kinds of graphs: complete bipartite graphs, complete multipartite graphs, and more. For a very general construction, see A. Kelmans, “Spanning trees of extended graphs”, *Combinatorica* 12 (1992), 45–51.

### 2.4. MSTs and Kruskal’s algorithm

Let \( G = (V, E) \) be a loopless graph equipped with a weight function \( \text{wt} : E \to \mathbb{R}_{\geq 0} \). For a subset \( A \subseteq E \), define
\[
\text{wt}(A) = \sum_{e \in A} \text{wt}(e).
\]

How do we find a spanning tree of minimum total weight?

The most naive algorithm is as follows. Find the cheapest edge and color it green. Then find the next cheapest edge and color it green (provided it isn’t parallel to the first edge). Then find the next cheapest edge and color it green (provided it isn’t parallel to either of the first two edges, or complete a \( C_3 \)). Keep coloring the cheapest edge available green, provided you never complete a cycle.
This procedure, called Kruskal’s algorithm, is very easy to understand and implement, but it is not clear that it works. But, amazingly, it does. The key to the proof is understanding the structure of the family of spanning trees of a graph $G$. We already know that all spanning trees have the same number of edges, but not just any family of sets of the same size can be $\mathcal{F}(G)$ for some $G$. In fact, any two trees interact in a very specific way: his procedure, called Kruskal’s algorithm, is very easy to understand and implement, but it is not clear that it works. But, amazingly, it does.

**Proposition 2.10** (Exchange rules for spanning trees). Let $G$ be connected on $n$ vertices, and let $T,T'$ be spanning trees of $G$. Then:

1. For each $e \in E(T) - E(T')$, there exists $e' \in E(T') - E(T)$ such that $T - e + e'$ is a spanning tree.
2. For each $e \in E(T) - E(T')$, there exists $e' \in E(T') - E(T)$ such that $T' + e - e'$ is a spanning tree.

(Note: I am using $+$ for the operation of adding an edge. This is different from $G + H$ which would denote the disjoint union of two graphs.)

**Proof:**

(1): $T - e$ has exactly two components (shown in green and blue in the figure below). It suffices to choose $e'$ to be an edge of $T'$ with one endpoint in each component of $T - e$. Such an edge must exist because $T'$ is connected.

(2): $T' + e$ has a cycle (since trees are maximally acyclic graphs); call it $C$. It is shown in yellow below. Then $C \subseteq T$ (because $T$ is acyclic), so pick $e' \in C \setminus T$. Then $T' + e - e'$ is still connected and has $n - 1$ edges, hence is a spanning tree. □
You’ll actually prove a stronger fact on HW #2: given $e \in E(T) - E(T')$, there exists $e' \in E(T') - E(T)$ such that $T - e + e'$ and $T' + e - e'$ are both spanning trees of $G$.

Here is a precise statement of Kruskal’s algorithm.

```
Input: connected graph $G$ with weight function $w$
Output: a MST $T$

$T_0 := 0$
$i = 0$
$A := E$  # available edges
while $(V, T_i)$ is disconnected and $A \neq \emptyset$ do
    Choose $e \in A$ of minimum weight
    $A := A - e$
    if $T_i + e$ is acyclic:
        Set $T_{i+1} := T_i + e$
        Set $i := i + 1$
Output $T = T_i$
```

**Theorem 2.11.** The output $T$ of Kruskal’s algorithm is a MST of $G$.

**Proof.** First, we check that the output is really a spanning tree. By construction, $T$ is acyclic. If it is disconnected, then the algorithm made a mistake — pick any edge $e$ that is a bridge of $T + e$; that was still true at whatever stage in the algorithm $e$ was considered (since $T_i \subset T$) so $e$ should have been added, but wasn’t.

Now comes the clever part. Let $T^*$ be some MST (certainly one must exist since $\mathcal{T}(G)$ is finite). If $T^* = T$, then we’re done.

Otherwise, let $e$ be the first edge chosen for $T$ that is not in $T^*$. Let

$$F := \{ f \in E(T) : f \text{ was added earlier than } e \}.$$  

- By the choice of $e$ we have $F \subseteq E(T^*)$.
- By Prop. 2.1.7, we can choose $e' \in E(T^*) - E(T)$ so that $T^{**} = T^* - e' + e$ is a spanning tree.
- Note that $e'$ could not have been available at the stage of the algorithm when $e$ was added to $T$. Otherwise it would have been added to $T$ (since $F + e' \subset E(T^*)$ is acyclic) and then would be in $F$ (by the choice of $e$), hence in $T^*$, but it isn’t.
- Therefore, $e'$ was considered after $e$, so $\text{wt}(e') \geq \text{wt}(e)$.
- In particular, $\text{wt}(T^{**}) \leq \text{wt}(T^*)$.
- But $T^*$ is a MST, so equality must hold, which means that in fact $T^{**}$ is a minimum spanning tree.

We have shown: if $G$ has a MST $T^*$ with $|T \cap T^*| = k < |T|$, then it has a MST $T^{**}$ with $|T \cap T^{**}| = k + 1$.

By induction (or iteration, if you prefer), $T$ itself must be a MST. \qed

Some notes on Kruskal’s Algorithm:

1. **Computational issues.** If you are going to implement Kruskal’s Algorithm, it is best to first sort the edges in increasing order by weight. Probably the best way to check whether an edge can be added is to check that its endpoints are in different components. This means keeping track of which vertices lie in a common component, and updating that data whenever the algorithm successfully adds an edge.
2. Matroids. Note that the exchange rules (Proposition 2.10) hold in a more general context. Let
\[ E = \text{set of edges} \quad S = \text{set of vectors spanning a vector space} \]
\[ T, T' = \text{spanning trees} \quad B = \text{subsets of } E \text{ that are bases for } S \]
Then Kruskal’s Algorithm can be used to find a minimum-weight basis. More generally, let
\[ E = \text{any finite set}, \]
\[ w : E \to \mathbb{R}, \]
\[ B = \text{collection of subsets of } E \text{ of the same size satisfying the exchange properties}. \]
In this case \( B \) is called a matroid basis system, and Kruskal’s Algorithm can be used to find an element of \( B \) of minimum weight. In fact, matroids can be characterized as set systems for which Kruskal’s algorithm works. Matroids are of high importance both in combinatorial optimization and in algebraic combinatorics.

3. Prim’s Algorithm is another way to efficiently compute an MST. It works like this:

```
Pick an arbitrary vertex \( v \)
\[ X := \{ v \}, \quad T := \emptyset \]
while \( X \neq V \) do
    Choose \( e = xy \) of minimum weight such that \( x \in X, y \notin X \)
    \[ T := T + e, \quad X := X + y \]
Output \( T \)
```

This method also produces an MST (proof omitted). It has the advantage of being somewhat easier to implement than Kruskal’s algorithm, because it is easier to keep track of the single vertex set \( X \) than the component structure of a graph. On the other hand, Prim’s algorithm is specifically about graphs and cannot be extended to matroids (the concept “cycle” has an analogue for matroids, but “vertex” doesn’t).
3. Matchings and covers

Throughout this section, $G = (V,E)$ will be a connected simple graph. We will not generally distinguish between an edge set $A \subseteq E(G)$ and the corresponding spanning subgraph $(V(G), A)$. Doing so is usually more trouble than it’s worth. Sometimes we’ll need to say whether we mean to include all vertices, or just the set of vertices $V(A)$ incident to at least one edge of $A$. But the meaning of terms such as “acyclic” and “component” should be clear when they are applied to edge sets. Maybe this warning should be put in the first section of the notes next time.

3.1. Basic definitions.

**Definition 3.1.** A **vertex cover** of $G$ is a set $Q \subseteq V(G)$ that contains at least one endpoint of every edge. An **edge cover** of $G$ is a set $L \subseteq E(G)$ that contains at least one edge incident to every vertex.

Here are some pictures of vertex covers.

![Vertex Cover Pictures]

**Warning:** “Minimal” and “minimum” don’t mean the same thing. In the first cover shown above ($G = C_6$, $Q = \{1, 2, 4, 5\}$), the cover is minimal because no proper subset is a cover, but it is not minimum because $G$ has a cover of strictly smaller cardinality.

Of course, $V$ itself is always a vertex cover, and $E$ is always an edge cover (provided that $G$ has no isolated vertices). The interesting problem is to try to find covers that are as small as possible.

**Definition 3.2.** A **matching** on $G$ is an edge set $M \subseteq E$ that includes at most one edge incident to each vertex. A vertex is **matched** (or **saturated**) by $M$ if it is incident to an edge in $M$. The set of matched vertices is denoted $V(M)$; note that $|V(M)| = 2|M|$. The matching $M$ is **maximal** if it is not contained in any strictly larger matching; **maximum** if it has the largest possible size among all matchings on $G$; and **perfect** if $V(M) = V(G)$. More generally, a **$k$-factor** in $G$ is defined to be a $k$-regular spanning subgraph; thus a perfect matching is a 1-factor.

**Warning:** Again, “maximum” is a stronger condition than “maximal.” For example, in the figure below, the blue matching $M$ is maximum; the red matching $M'$ is maximal but not maximum.
The vertex analogue of a matching is a **coclique** (or *independent set* or *stable set*): a set of vertices such that no two are adjacent. In general we want to know how large a coclique or matching can be in a given graph.

So we have four related notions:

- **coclique**: a set of vertices touching each edge at most once
- **vertex cover**: a set of vertices touching each edge at least once
- **matching**: a set of edges touching each vertex at most once
- **edge cover**: a set of edges touching each vertex at least once

Define

\[ \alpha = \text{maximum size of a coclique}, \quad \beta = \text{minimum size of a vertex cover}, \]
\[ \alpha' = \text{maximum size of a matching}, \quad \beta' = \text{minimum size of an edge cover}. \]

(This is West’s notation, which may or may not be standard. The mnemonic is that invariants without primes involve sets of vertices; the primed versions involve edges. The symbol \( \alpha \), the first letter of the Greek alphabet, is actually fairly standard for the size of the largest coclique in \( G \). The last letter, \( \omega \), is the size of the largest clique.)

The matching and edge cover problems are equivalent and can be solved in polynomial time. The coclique and vertex cover problems are equivalent and are NP-complete in general. However, all four problems are equivalent for bipartite graphs. Fortunately, many matching problems are naturally bipartite (e.g., matching job applicants with positions, students with advisors, or workers with tasks, or columns with rows).

By the way, **counting** the maximum matchings of \( G \) is hard in general (although it can be done for, say, \( K_{2n} \) and \( K_{n,n} \)). It is unknown even for such nice graphs as \( Q_n \).

**Example 3.3.** For the cycle \( C_n \), it should be clear that

\[ \alpha = \alpha' = \left\lfloor \frac{n}{2} \right\rfloor \quad \text{and} \quad \beta = \beta' = \left\lceil \frac{n}{2} \right\rceil. \]

In particular \( \alpha + \beta = \alpha' + \beta' = n \), and equality holds throughout if and only if the cycle is even. The first observation holds for all graphs, as we will see; the second one suggests that bipartiteness may be important.

**Example 3.4.** If \( G \) is bipartite, then each partite set is a vertex cover. On the other hand, a bipartite graph can have covers that are smaller than either partite set. (Example on the left below.) It seems plausible to try to build a minimum vertex cover by using vertices of large degree — but this doesn’t always work either. (For example, in the graph on the right below, the unique vertex of largest degree is \( x \), but the unique minimum vertex cover is \( \{a, b, c\} \).

\[ \begin{align*}
\text{graph 1} & \quad \text{graph 2}
\end{align*} \]

3.2. **Equalities among matching and cover invariants.**

**Proposition 3.5.** \( \alpha + \beta = n \).

**Proof.** A vertex set is a coclique iff its complement is a cover. In particular, the complement of a maximum coclique is a minimum cover. \( \square \)
Proposition 3.6 (Gallai’s identity). \( \alpha' + \beta' = n \).

Proof. First, we show that \( \alpha' + \beta' \leq n \). Let \( M \) be a maximum matching, so that \( |M| = \alpha' \). Let \( A \) be a collection of edges, each incident to one \( M \)-unmatched vertex. These edges are all distinct (otherwise \( M \) would not even be maximal), so \( |A| = n - |V(M)| = n - 2\alpha' \). On the other hand, \( A \cup M \) is an edge cover, so \( |A \cup M| \geq \beta' \) and

\[
\alpha' + \beta' \leq |M| + |A \cup M| = 2|M| + |A| = n.
\] (1)

Second, we show the opposite inequality. Let \( L \) be a minimum edge cover, so \( |L| = \beta' \). For every edge \( e = xy \in L \), we must have either \( \text{deg}_L(x) = 1 \) or \( \text{deg}_L(y) = 1 \), otherwise \( e \) could be removed from \( L \) to yield a smaller edge cover. This implies that every component of \( L \) is a star; in particular it is acyclic, so \( |L| = n - c(L) \). Now construct a matching \( M \) by choosing one edge from every component of \( L \). Then \( |M| = c(L) \leq \alpha' \) and

\[
\alpha' + \beta' \geq |M| + |L| = c(L) + n - c(L) = n.
\] (2)

Now combining (1) and (2) finishes the proof. \( \square \)

Note that once the proof is complete, it follows that equality had to hold in both (1) and (2). That means that the proof gives us an easy way of constructing either a matching from an edge cover, or vice versa.

Proposition 3.7. \( \alpha' \leq \beta \).

Proof. If \( M \) is a matching in \( G \), then no vertex can cover more than one of the edges in \( M \), so every vertex cover has to have at least \( |M| \) vertices. \( \square \)

This kind of result is called a weak duality: every matching has size less than or equal to the size of every vertex cover. It implies that if we can find a matching and a vertex cover of the same size, then the matching must be maximum and the cover must be minimum — but does not guarantee that such a pair exists. In fact equality does not always hold, e.g., for an odd cycle. However, there is good news:

Theorem 3.8 (König-Egerváry Theorem). If \( G \) is bipartite then \( \alpha' = \beta \).

This is going to take some proof. We are actually going to construct an algorithm to enlarge a matching, one edge at a time. If the matching is maximum then the algorithm will actually certify that by producing a vertex cover of the same size as the matching.

3.3. Augmenting paths.

Definition 3.9. Let \( M \) be a matching in \( G \) and \( P \subset G \) a path. A path \( P \) is \( M \)-alternating if its edges alternate between edges in \( M \) and edges not in \( M \). It is \( M \)-augmenting if it is \( M \)-alternating and both endpoints are unmatched by \( M \).
Note that every $M$-augmenting path has an even number of vertices (two unmatched endpoints and an even number of matched interior vertices), hence odd length.

If $P$ is an $M$-augmenting $p,q$-path, then $M\triangle P$ is a matching with one more edge than $M$, where $\triangle$ means symmetric difference. (The operation of passing from $M$ to $M\triangle P$ might be called “toggling $P$ with respect to $M$”: take the edges in $P \cap M$ out, and put the edges of $P \setminus M$ in.)

One fact about the symmetric difference operation: if $C = A \triangle B$ then $A = B \triangle C$ and $B = A \triangle C$. (Each of these equations says exactly the same thing: every element is contained in an even number of the sets $A, B, C$.)

The following lemma will be useful both immediately in proving Berge’s theorem, and also later in proving Tutte’s 1-factor theorem for nonbipartite matching.

**Lemma 3.10.** Let $M, N$ be matchings on $G$. Then every nontrivial component of $M \triangle N$ is either a path or an even cycle.

**Proof.** Each vertex of $G$ can have degree at most 2 in $M \triangle N$. Therefore, every nontrivial component $H$ is either a path or a cycle. If $H$ is a cycle, then its edges alternate between edges of $M$ and edges of $N$, so $H$ is even.

**Theorem 3.11.** (Berge, 1957) Let $M$ be a matching on $G$. Then $M$ is maximum if and only if $G$ contains no $M$-augmenting path.

**Proof.** ( $\Rightarrow$ ) If $M$ has an augmenting path $P$ then $M \triangle P$ is a bigger matching.

( $\Leftarrow$ ) Suppose that $M$ is a non-maximum matching. Let $N$ be a matching with $|N| > |M|$, and let $F = M \triangle N$. Then $|F \cap N| > |F \cap M|$, because

$$|F \cap N| = |N| - |M \cap N| > |F \cap M| = |M| - |M \cap N|.$$ 

In particular, some component $H$ of $F$ contains more edges of $N$ than of $M$. By Lemma 3.10 $H$ must be a path of odd length $\ell$, say $H = v_0, v_1, \ldots, v_{\ell}$ with

$$v_0v_1 \in N, \ v_1v_2 \in M, \ v_2v_3 \in N, \ldots, \ v_{\ell-2}v_{\ell-1} \in M, \ v_{\ell-1}v_{\ell} \in N.$$ 

Now, both $F$ and $N$ contain exactly one edge incident to $v_1$, namely $v_1v_2$. But $F = M \triangle N$, so $M = F \triangle N$, and so $M = F \triangle N$ does not contain any edge incident to $v_1$. By the same logic, $v_2 \notin V(M)$ as well. But this says precisely that $H$ is an $M$-augmenting path.
Berge’s theorem reduces the problem of finding a maximum matching to the problem of determining whether a given matching has an augmenting path. This problem is easier when $G$ is bipartite.

Notation: $N(x) = \{\text{neighbors of } x\}, \quad N(X) = \bigcup_{x \in X} N(x)$

Here is the algorithm. It is really a form of breadth-first search.

1. Start at an unmatched vertex $x_0 \in X$. Let $S_0 = \{x_0\}$.
2. Put in the edges $x_0y$ for all $y \in N(x_0) \subset Y$.
3. If any vertex added in step (2) is unmatched, then we have an augmenting path. Else, put in the edges that match every $y \in N(x_0)$ to its spouse (which must lie in $x$). Call this set of spouses $S_1$.
4. Put in the vertices $y$ and edges $sy$ for all $s \in S_1$ and $y \in N(s)$.
5. If any vertex added in step (4) is unmatched, then we have an augmenting path. Else, put in the edges that match every $y \in N(x_0)$ to its spouse (which must lie in $x$). Call this set of spouses $S_2$.

(*) Iterate until either we find an augmenting path, or no new vertices are found in an even-numbered step.

If the algorithm does not find an augmenting path, repeat for every possible unmatched starting vertex $v_0 \in X \setminus V(M)$.

**Example.** Let us run the algorithm on the following graph and matching:

![Graph and Matching](example_graph.png)

Here are the search trees we get by using $x_3$ and $x_7$, respectively, as the starting vertices:

![Search Trees](search_trees.png)
If we start at $x_3$, the search peters out quickly, and no augmenting path is found. Starting at $x_7$, however, finds an augmenting path $P$ Here is what $P$ looks like in the original graph, and the larger matching $M' = M \triangle P$:

At this point, the search will find no augmenting path — we start at $x_3$, construct the two-edge search tree $x_3 - y_1 - x_1$, and terminate. Note that the search has told us that

$$N\{\{x_1, x_3\}\} \subseteq \{y_1\}.$$  

But that means that $Q = \{y_1\} \cup (X \setminus \{x_1, x_3\})$ is a vertex cover, and $|Q| = 7 = |M'|$, which verifies (by weak duality) that $M'$ is a maximum matching and $Q$ is a minimum vertex cover.

**Proposition 3.12.** Let $G$ be a bipartite graph and $M$ a matching with no augmenting path. Let $U = U_X \cup U_Y$ be the set of vertices visited in the (unsuccessful) call to the Augmenting Path Algorithm. Then

$$Q = U_Y \cup (V(M) \setminus U_X)$$  

is a vertex cover of cardinality equal to $|M|$.

**Proof.** First, we show that $Q$ is a vertex cover. Note that $N(U_X) \subseteq U_Y$ by the construction of the APA. So if $e = xy$ with $x \in U_X$, then $y \in N(U_X) \subseteq U_Y \subseteq Q$, while if $x \notin U_X$ then $x \in Q$.

What about its cardinality? Every vertex in $Q$ is matched. On the other hand, the vertices in $U_Y$ are matched to vertices in $U_X$, which means that no edge of $M$ meets $Q$ more than once. Therefore, $|Q| \leq |M|$, and the reverse equality holds by weak duality.  

Here is another example. The matching on the left is maximum, with the unsuccessful search trees shown on the right. We have therefore

$$U_X = \{x_1, x_2, x_3, x_5, x_6\}, \quad U_Y = \{y_2, y_3, y_4\}$$

which says that

$$Q = U_Y \cup (V(M) \setminus U_X) = \{y_2, y_3, y_4, x_4, x_7, x_8\}$$

is a vertex cover, as shown.
The punchline: Berge’s Theorem together with Prop. 3.12 proves the König-Egerváry Theorem: $\beta = \alpha'$ for all bipartite graphs.

3.4. Hall’s Theorem and consequences. Hall’s Matching Theorem is a classical theorem about matchings with lots of proofs; the one I like uses the Augmenting Path Algorithm. (This is not Hall’s original proof.)

**Theorem 3.13** (Hall’s Matching Theorem, 1935). Let $G$ be an $X,Y$-bipartite. Then $G$ has a matching saturating $X$ if and only if $|N(S)| \geq |S|$ for every $S \subseteq X$.

*Proof*. Necessity of Hall’s condition ($\implies$): Let $M$ be a matching saturating $X$. Then every $S \subseteq X$ is matched to a set of equal cardinality, which is a subset of $N(S)$.

Sufficiency of Hall’s condition ($\impliedby$): Let $M$ be a maximum matching, and suppose that $x \in X$ is unsaturated. Consider the unsuccessful search tree computed by the Augmenting Path Algorithm starting at $x$. Call its vertex set $S \cup T$, with $S \subseteq X$ and $T \subseteq Y$. Observe that:

- $|S| > |T|$, since every vertex in $T$ is matched to a vertex in $S$, and $S$ contains at least one unmatched vertex, namely $x$.
- $N(S) = T$, since that is precisely how the algorithm works.

Therefore, $S$ violates Hall’s condition.

Hall’s Theorem is not useful as an algorithm because actually computing $|N(S)|$ for every $S \subseteq X$ would require looking at all $2^{|X|}$ subsets. On the other hand, it is a great theoretical tool. Here are some consequences:

**Corollary 3.14.** Every regular bipartite simple graph has a perfect matching.

*Proof*. Let $G$ be a $k$-regular $X,Y$-bipartite. By bipartite handshaking, $e(G) = k|X| = k|Y|$, so in particular $|X| = |Y|$.

Let $S \subseteq X$ and consider the induced subgraph $H = G|_{S \cup N(S)}$, which is bipartite with partite sets $S$ and $N(S)$. Each vertex of $S$ has degree exactly $k$ in $H$, and each vertex of $N(S)$ has degree at most $k$ in $H$. By the bipartite handshaking formula, $|S| \leq N(S)$. Since $S$ was arbitrary, Hall’s Theorem implies that $G$ has a perfect matching.
Corollary 3.15. Every \( k \)-regular bipartite simple graph decomposes into the union of \( k \) perfect matchings. (Here “decomposes” refers to the edge set.)

This corollary can be rephrased in terms of matrices. A simple bipartite graph can be recorded by its bipartite adjacency matrix, with a row for each vertex in \( X \) and a column for each vertex \( Y \), with edges indicated by 1’s and non-edges by 0’s. The graph is \( k \)-regular iff every column and row sum is \( k \) (which requires the numbers of columns and rows to be the same). A matching corresponds to a transversal: a collection of 1’s including exactly one entry in every row and column (this is essentially the same thing as a permutation matrix). The corollary then says that every \( n \times n \) 0,1-matrix with all row and column sums equal to \( k \) can be written as the sum of \( k \) permutation matrices.

3.5. Weighted bipartite matching. Let \( G \) be a bipartite graph with partite sets \( X, Y \), and let \( w : E(G) \to \mathbb{R}_{\geq 0} \) be a weight function.

Problem: Find a matching \( M \) of maximum total weight, i.e., maximizing

\[
  w(M) = \sum_{e \in M} w(e).
\]

WLOG, we may assume that \( |X| = |Y| = n \) (adding isolated vertices if necessary), and that \( G \cong K_{n,n} \) (adding edges of weight 0 if necessary). Then the maximum cardinality matchings are the \( n! \) perfect matchings, and we may as well look for one of them.

Represent the pair \((G, w)\) by an \( n \times n \) matrix

\[
  W = (w_{ij})_{i,j \in [n]}\]

where \( w_{ij} = w(x_i y_j) \).

Definition 3.16. A transversal of \( W \) is a set of \( n \) matrix entries, one in each row and column. (Equivalent to a perfect matching on \( K_{n,n} \); can be described by a permutation \( \sigma : [n] \to [n] \).)

Definition 3.17. A (weighted cover) \( C \) of \( W \) is a list of row labels \( a_1, \ldots, a_n \) and column labels \( b_1, \ldots, b_n \) such that

\[
  \forall (i, j) \in [n] \times [n]: \quad a_i + b_j \geq w(x_i y_j) .
\]

The cost of the cover is \( |C| = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \).

Example:

\[
\begin{bmatrix}
  3 & 1 & 4 & 5 \\
  0 & 2 & 6 & 2 \\
  0 & 4 & 0 & 4 \\
  1 & 2 & 5 & 0 \\
\end{bmatrix} \quad \begin{bmatrix}
  3 & 1 & 4 & 5 \\
  0 & 2 & 6 & 2 \\
  0 & \text{boxed} & 0 & 4 \\
  1 & 2 & \text{boxed} & 0 \\
\end{bmatrix} \quad \begin{bmatrix}
  0 & 0 & 1 & 2 \\
  3 & 1 & 4 & 5 \\
  5 & 0 & 2 & 6 \\
  4 & 0 & 4 & 0 \\
\end{bmatrix}
\]

weight matrix \( W \) transversal of weight 14 cover of cost 19

Problem: Given a square nonnegative integer matrix, find a cover of minimum-cost.
Lemma 3.18. The maximum weighted matching and minimum weighted cover problems are weakly dual. That is, for every matching $M$ and cover $C$,

$$w(M) \leq |C|.$$

Moreover, equality holds if and only if $w(x_iy_j) = a_i + b_j$ for every edge $x_iy_j \in M$. In that case, $M$ and $C$ are optimal.

Proof. Represent $M$ by a transversal $\sigma$ of the weight matrix $W$. The cover condition (3) says that $w_{i,\sigma(i)} \leq a_i + b_{\sigma(i)}$ for all $i$, so

$$w(M) = \sum_{i=1}^{n} w_{i,\sigma(i)} \leq \sum_{i=1}^{n} (a_i + b_{\sigma(i)}) = |C|.$$

and equality holds in (4) if and only if it holds for each $i$, since the inequality is term-by-term. □

The cover shown above has cost 19. Can this be improved? If we could find a column or a row in which every entry was overcovered (i.e., for which the inequality (4) was strict), then we could decrease the label of that column. But there is not always such a column or row. The good news is that we can do something even more general in the spirit of the Augmenting Path Algorithm. The key is to increase the cover on some columns by some amount $\epsilon$ and decrease it on some rows by the same $\epsilon$, amount, making sure that we decrease more labels than we increase.

To find this, circle the matrix entries that are covered exactly, i.e., those $w_{ij}$ such that $w_{ij} = a_i + b_j$. The corresponding edges form a spanning subgraph $H \subseteq K_{n,n}$ called the equality subgraph $H = Eq(W,C)$.

Now run the Augmenting Path Algorithm to find a maximum matching $M$ on $H$, together with a minimum vertex cover $Q$. Recall that $|M| = |Q|$ by the Konig-Egervary Theorem (which we proved using the APA), and that $Q = (X \setminus U_X) \cup U_Y$, where $U$ is the set of vertices reached during the last (unsuccessful) search for an augmenting path. (Note that there is no guarantee of uniqueness for $M$ and $Q$, because $H$ may have several different maximum matchings and minimum vertex covers, but the APA will certainly produce one of each.) A possible output is shown below.
If $|M| = |Q| = n$, then $w(M) = |C|$ and we are done by weak duality (Lemma 3.18). Otherwise, we can use $Q$ to find a less expensive cover, as follows. In terms of the matrix, $Q = Q_X \cup Q_Y$ corresponds to a collection of rows and columns (which we’ll also call $Q_X$ and $Q_Y$), of total cardinality $< n$, containing every circled matrix entry.

Construct the **excess matrix**, whose $(i,j)$ entry is $w_{ij} - a_i - a_j$. (So the zeros in this matrix correspond precisely to edges of $H$.) Then paint blue the rows and columns corresponding to the vertices in $Q$. The numbers not painted blue in the excess matrix must all be positive; their minimum is the **tolerance**, denoted $\varepsilon$. Here $\varepsilon = 2$.

Now decrease the labels on $X \setminus Q_X$ by $\varepsilon$, and increase the labels on all columns in $Q_Y$ by $\varepsilon$. This is shown in the third matrix below, with the red arrows indicating which labels have been increased or decreased.

This operation maintains the cover conditions, since the only matrix entries that decrease are those with rows in $X \setminus Q$ and columns in $Y \setminus Q_Y$, but all those entries were already over-covered by at least $\varepsilon$. Moreover, the cost of the cover has dropped by $\varepsilon(n - |Q|)$.

Repeat this procedure until the equality subgraph has a perfect matching. In this case, it just takes one more step.

Now we’re done — we have a perfect matching whose weight equals the cost of the cover.\footnote{Thanks to Lawrence Chen for catching a mistake in an earlier example of the procedure.}

This procedure is called the **Hungarian Algorithm**. Here is a summary of the algorithm.
The Hungarian Algorithm

**Input:** weight function \( w : E(K_{n,n}) \to \mathbb{R} \)

**Output:** a maximum weighted matching \( M \) and minimum cover \( C = (a_1, \ldots, a_n, b_1, \ldots, b_n) \)

1. Initialize \( a_i = \max\{w_{ij} : i \neq j\} \) and \( b_j = 0 \) for all \( i, j \in [n] \)
2. Use the APA to compute a maximum matching \( M \) and a minimum cover \( Q \)
3. While \( |Q| < n \) do
   4. Let \( \varepsilon := \min\{w_{ij} - a_i - b_j : x_i, y_j \notin Q\} \)
   5. Set \( a_i := a_i - \varepsilon \) for all \( x_i \in X\setminus Q_X \)
   6. Set \( b_j := b_j + \varepsilon \) for all \( y_j \in Q_Y \)
   7. Recompute \( H \).
   8. Use the APA (starting with \( M \)) to recompute the pair \( M, Q \)
9. Return \( (M, C) \)

One not-totally-obvious fact is that after the cover is adjusted in steps 5 and 6, the new equality subgraph computed in step 7 will still have \( M \) as a matching. This is left as an exercise.

Observe that if all the weights were nonnegative integers to begin with, then the procedure will definitely terminate (in at most a number of steps equal to the original cost of the cover). We have proved:

**Theorem 3.19.** For any bipartite graph \( G \) and weight function \( w : E(G) \to \mathbb{N}_{\geq 0} \), the Hungarian Algorithm calculates a minimum cover \( C = (a, b) \) and a maximum matching \( M \), with \( |C| = w(M) \).

We don’t need to assume positive weights, since adding the same constant to all \( n^2 \) edges does not change which matchings have maximum weight. Also, \( \mathbb{N} \) can be replaced with \( \mathbb{Q} \) — there are finitely many weights, so just multiply them all weights by some common denominator to convert them to integers. Again, this will not change which matchings have maximum weight.

What about real weights? The potential problem is that the sequence of cover values produced by the Hungarian Algorithm might be something like

\[ 2, 1.1, 1.01, 1.001, 1.0001, \ldots \]

and the algorithm might never terminate, even though the minimum cover value is actually 1. Fortunately, this doesn’t happen! The proof is a homework problem.

### 3.6. Stable matchings.

(Note: This material appears in the 3rd and subsequent editions of Diestel (2005); it is not in the 2nd edition (2000).)

You are the chair of the mathematics department, and you have to assign \( n \) students \( X = \{x_1, \ldots, x_n\} \) to \( n \) advisors \( Y = \{y_1, \ldots, y_n\} \). That is, you need to choose a perfect matching in \( K_{n,n} \). (Let’s suppose \( n = 3 \), so that the problem is a manageable size.) You’d like to do this in some way that reflects the preferences of each student and advisor, and you’ve asked each person to submit a form listing his or her three preferences in descending order. The data you have is as follows:

<table>
<thead>
<tr>
<th>Student</th>
<th>Preference order</th>
<th>Advisor</th>
<th>Preference order</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>( y_1, y_2, y_3 )</td>
<td>( y_1 )</td>
<td>( x_1, x_2, x_3 )</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>( y_1, y_3, y_2 )</td>
<td>( y_2 )</td>
<td>( x_1, x_3, x_2 )</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>( y_1, y_3, y_2 )</td>
<td>( y_3 )</td>
<td>( x_3, x_2, x_1 )</td>
</tr>
</tbody>
</table>

Unfortunately, it’s not so clear what “optimal” means. Should we be trying to maximize overall happiness? Are the students’ preferences more important than the advisors’, or vice versa?
Since we know how to solve the weighted bipartite matching, we could try the following approach. Each student assigns two points to his top-choice advisor and one point to his second choice. Each advisor assigns two points to her top-choice student and one point to her second choice. We then get a weighted copy of $K_{3,3}$, with weight matrix as follows:

$$
\begin{array}{ccc}
  x_1 & y_1 & y_2 & y_3 \\
  2 + 2 = 4 & 1 + 2 = 3 & 0 + 0 = 0 \\
  2 + 1 = 3 & 0 + 0 = 0 & 1 + 1 = 2 \\
  2 + 0 = 2 & 0 + 1 = 1 & 1 + 2 = 3 \\
\end{array}
$$

We now select the matching of maximum total weight. Since $n$ is so small, we can just do this by brute force instead of applying the Hungarian Algorithm. It’s not hard to verify that the unique maximum-weight matching is $M = \{x_1y_2, x_2y_1, x_3y_3\}$.

There is a problem with this solution, though. $x_1$ and $y_1$ are each other’s first choice, yet they are not paired in $M$. So what’s to prevent them from deciding to work with each other, leaving $x_2$ and $y_2$ high and dry? Those two could pair up, it’s true, but neither of them would be at all happy about it—each is the other’s last choice. They’d certainly have cause to complain about the system. Meanwhile, if you use your power as chair to prevent $x_1$ and $y_1$ from working together, neither of them is going to thank you for it.

Maybe looking for a maximum-weight matching is not the way to go. What you’d really like is a system that will produce a matching in which no advisor-swapping will take place: that is, a stable matching. Not everybody will necessarily be happy, but at least no one (we hope) will have reason to complain about the system’s fairness, and no one will have an incentive to ignore the rules of the game.

**Definition 3.20.** Let $M$ be a perfect matching on $K_{n,n}$, and write $M(z)$ for the vertex matched to $z$. An **unstable pair** in a matching $M$ is a pair $(x,y)$ such that $x$ prefers $y$ over $M(x)$ and $y$ prefers $x$ over $M(y)$. A matching is **stable** if it has no unstable pair.

It is by no means obvious that a stable matching always exists for any given list of preferences—but it does. It can be found by the following algorithm, due to Gale and Shapley. We designate one partite set $X$ as the **proposers** and the other, $Y$, as the **responders**. For clarity in pronouns, I will refer to the elements of $X$ as male and the elements of $Y$ as female, but this could just as easily be switched

---

**The Gale-Shapley Stable Matching Algorithm — Deterministic Version**

- Each proposer proposes to the top-choice responder who has not already rejected him.
- If the set of proposals forms a perfect matching, that matching is the output.
- Otherwise, each responder rejects all but her top choices of the proposals she receives.
- Repeat until the proposals form a perfect matching.

---

**Theorem 3.21.** The Gale-Shapley algorithm terminates and produces a stable matching.

**Proof.** First, we show that the algorithm does not get stuck — in another words, we never reach a situation in which some proposer has been rejected by every responder. Here is why this can’t happen.

1. If $y$ ever issues a rejection, then she will receive at least one proposal at every subsequent stage of the algorithm. (Whoever survived $y$’s previous cut will keep proposing to $y$ until replaced by someone else.)
2. Suppose that $x$ has already been rejected by $n - 1$ responders. In the iteration in which $x$ does so, every responder other than $y$ must receive at least one proposal (since they have each previously
issued at least one rejection). But by the pigeonhole principle, that means that every responder receives exactly one proposal, and so the algorithm terminates with a perfect matching \( M \).

By the way, each iteration that does not terminate removes at least one edge from the graph of potential matched pairs. Therefore, the algorithm terminates in at most \( n^2 - n \) iterations.

Now, suppose that \((x, y)\) is an unstable pair. That is,

- \( x \) prefers \( y \) to \( M(x) \),
- \( y \) prefers \( x \) to \( M(y) \),
- \( x \) and \( y \) aren’t matched.

Since \( y \) prefers \( x \) to \( M(y) \), it follows that \( x \) never proposed to \( y \). But that must mean that \( x \) prefers \( M(x) \) to \( y \), which contradicts the definition of unstable pair. We conclude that \( M \) must be a stable matching. □

The algorithm would run equally well if the roles of advisors and students were switched. It turns out that the side doing the proposing is uniformly better off, and the side being proposed to is uniformly worse off. That is, if \( M \) is the matching produced by the Gale-Shapley algorithm and \( N \) is any other stable matching, then every \( x \in X \) prefers \( M(x) \) to \( N(x) \) (unless \( M(x) = N(x) \)) and every \( y \) prefers \( N(y) \) to \( M(y) \) (unless \( M(y) = N(y) \)). The moral of the story is that it is better to propose than to be proposed to.

When I talked about this in class on 2/16/16, the Math 725 students asked several questions, which I list here with my attempts to answer them.

1. Does the algorithm still work if the partite sets do not have the same size? (Brandon Caudell) Yes, I think so. If there are more proposers than responders, the algorithm terminates when every responder receives at least one proposal (and rejects all but her top choice). If there are more responders, the algorithm terminates when every responder receives at most one proposal. The first part of the proof can be modified to show that the termination state is indeed reached, and the second part (i.e., that the output matching is stable) makes no reference to cardinality and thus still goes through.

2. What if some proposer or responder has ties among his or her preferences? (Joe Cummings) This is a subtle problem; see [this Wikipedia article]. Now you might have a unmatched pair \( x, y \) in which \( x \) prefers \( y \) over \( M(x) \), but \( y \) is indifferent between \( x \) and \( M(y) \); we might call such a pair weakly unstable. In this case, we’d have to do some more work to produce a matching without any pair that is even weakly unstable — it is not clear whether such a matching always exists.

3. Can there be a preference list where no proposer gets her first choice under Gale-Shapley? Can there be a preference list where no proposer or responder gets her first choice under Gale-Shapley? Yes to both. Joseph Doolittle came up with an example in class for \( n = 4 \).

4. Does Gale-Shapley produce a universally optimal result for every proposer (i.e., if \( M \) is the Gale-Shapley output and \( M' \) is any stable matching, then each proposer is at least as happy under \( M \) than under \( M' \))? Likewise, does Gale-Shapley produce a universally pessimal result for every responder? Yes. See below.

5. Can there be a preference list where no proposer gets his first choice under any stable matching? If you believe that the answers to (3) and (4) are yes, then the answer to this one is yes as well.

6. Can there be a preference list where no proposer or responder gets his/her first choice under any stable matching? Probably, but I haven’t constructed one.

7. Can there be a preference list where the algorithm takes the maximum number of iterations to finish, i.e., \( n^2 - n \)? I am pretty sure the answer to this one is yes.

8. What about \( K_n \) instead of \( K_{n,n} \)? In other words, suppose that each vertex in \( K_n \) has a preference order on the \( n - 1 \) other vertices, and we still want to find a stable matching (let’s assume \( n \) is even).

\(^2\)“Pessimal” is the antonym of “optimal.”
This is called the stable roommates problem, and it does not always have a solution, but it is possible to check in polynomial time whether a solution exists and, if so, to find it.

The “rounds” in the Gale-Shapley algorithm are in fact unnecessary. Consider the following “non-deterministic” version of the algorithm:

**The Gale-Shapley Stable Matching Algorithm — Nondeterministic Version**

- Let \( x \) be some proposer who is not currently on a wait list.
- \( x \) proposes to the top-choice responder \( y \) who has not already rejected him.
- If \( y \)'s wait list was empty, then she puts \( x \) on it. Otherwise, \( y \) chooses between \( x \) and the proposer currently on her wait list, and rejects the one she likes less.
- The algorithm terminates when all wait lists are full and we have a perfect matching.

It turns out that no matter how \( x \) is chosen in each iteration, the algorithm produces the same matching. The proof is an exercise, and is a key step in proving that Gale-Shapley is universally proposer-optimal and responder-pessimal.

A little more generally, one can also prove that no stable matching is universally better than any other, in the following sense:

**Proposition 3.22.** Let \( M, M' \) be stable matchings for the same preference list. If at least one proposer \( x \) is happier in \( M \) than in \( M' \), than at least one responder \( y \) is happier in \( M' \) than in \( M \).

The proof is left as an exercise.

### 3.7. Nonbipartite matching

Matching is harder without the assumption of bipartiteness. The König–Egerváry Theorem \((\alpha' = \beta)\) need not hold: for instance, the odd cycle \(C_{2n+1}\) has matching number \(\alpha' = n\) and vertex cover number \(\beta = n + 1\). (In fact the gap between the matching and cover numbers can be arbitrarily large.) In particular, the Augmenting Path and Hungarian Algorithms don’t work, although there do exist polynomial-time algorithms to compute maximum-cardinality and maximum-weight matchings. One maximum-cardinality matching is Edmonds’ Blossom Algorithm; this might make a good end-of-semester project. We are not going to look at algorithms, but we will prove two fundamental results in this area, namely Tutte’s 1-Factor Theorem and the Berge-Tutte Formula. (Recall that “1-factor” is a synonym for “perfect matching” and more generally that “\( k \)-factor” means (the edge set of a) \( k \)-regular subgraph.)

For a simple graph \( G \), not necessarily connected, define

\[
o(G) = \text{number of odd(-order) components of } G.
\]

**Lemma 3.23.** Let \( H \) be a spanning subgraph of \( G \). Then \( o(H) = o(G) + 2k \equiv n \pmod 2 \).

**Proof.** Construct \( G \) from \( H \) by starting with \( n \) isolated vertices (hence \( n \) odd components) adding edges, one at a time. Each addition either doesn’t change the component structure; makes an even component out of two even components; makes an odd component out of an even and an odd component; or makes an even component out of two odd components. The first three cases don’t change the number of odd components; the last case decreases it by 2. \( \Box \)

**Theorem 3.24** (Tutte’s 1-Factor Theorem). A simple graph \( G = (V, E) \) has a perfect matching if and only if it satisfies Tutte’s condition:

\[
o(G - S) \leq |S| \quad \forall S \subseteq V.
\]
Proof. \( \implies \) Let \( M \) be a perfect matching of \( G \), let \( S \subseteq V \), and let \( \bar{S} = V \setminus S \). Consider the graph \( H \) with vertex set \( \bar{S} \) and edges \( M|_{\bar{S}} \). This is a spanning subgraph of \( G - S \), so
\[
o(G - S) \leq o(H) = \# \text{ isolated vertices in } H = \# \text{ vertices in } \bar{S} \text{ matched to a vertex in } S = \# \text{ vertices in } S \text{ matched to a vertex in } \bar{S} \quad \text{(using } M \text{ as a bijection)} \leq |S|.
\]

\( \iff \) First, observe that by the argument of Lemma 3.23 adding one or more edges to \( G \) can only decrease the LHS of (5), hence preserves Tutte’s condition. Accordingly, if the \( \iff \) direction is false, we can choose a maximal graph for which it fails — i.e., a graph \( G \) such that

- \( G \) satisfies Tutte’s condition;
- \( G \) has no perfect matching; and
- adding any single missing edge to \( G \) produces a graph with a perfect matching.

We will show that these conditions imply a contradiction.

First, since \( G \) satisfies Tutte’s condition, in particular \( o(G - \emptyset) = o(G) \leq |S| = 0 \), so \( n(G) \) is even by Lemma 3.23.

Define \( U = \{v \in V \mid vw \in E \forall w \in V \setminus \{v\} \}. \) We will examine the graph \( G - U \).

**Case 1:** \( G - U \) is a disjoint union of cliques. We can construct a perfect matching on \( G \) as follows. Start with a maximal matching on \( G - U \); the number of leftover vertices is \( o(G - U) \). But by Tutte’s condition, there are at least that many vertices in \( U \), so all of those leftovers can be matched to vertices of \( U \) (in lots of ways). We have now matched all vertices outside \( U \), and \( G[U] \) is a clique, so the remaining vertices of \( U \) can be matched to each other (in lots of ways).

**Case 2:** \( G - U \) has some component \( H \) that is not a clique. Then \( H \) must contain an induced \( P_3 \), i.e., \( x, y, z \in V(H) \) and \( xy, yz \in E(H), xz \notin E(H) \). Also, since \( y \notin U \), there is a vertex \( w \in V(G - U) \) such that \( wy \notin E(G) \). Note that \( w \) may or may not belong to \( H \).

Here’s the picture. The vertices of \( U \) are colored gray, and all edges with one or both endpoints in \( U \) are omitted (otherwise the figure would be unreadable; remember, every gray vertex is adjacent to every other vertex.)

![Diagram](image)

By the choice of \( G \), adding any single edge to \( G \) produces a graph with a perfect matching (which must contain that added edge). Accordingly, let \( M_1 \) and \( M_2 \) be perfect matchings of \( G + xz \) and \( G + wy \) respectively.

38
The dashed edges $wy$ and $xz$ do not belong to $G$; all other edges do. Let $F = M_1 \triangle M_2$; then $xz, wy \notin F$.

Let $C$ be the component of $F$ containing $xz$ (highlighted above). By Lemma 3.10 $C$ is either a path or an even cycle. In fact, it’s a cycle, because $M_1$ and $M_2$ are both perfect matchings.

**Case 2a:** $yw \notin C$ (not the case of the example). Then

$$M_1 \triangle C = (M_2 \cap E(C)) \cup (M_1 - E(C))$$

is a perfect matching that contains neither $xz$ nor $wy$, so it is a perfect matching of $G$. That’s a contradiction—we had assumed that $G$ contained no perfect matching.

**Case 2b:** $yw \in C$. Label the vertices in cyclic order as

$$w, y, a_1, \ldots, a_p, z, x, b_1, b_q.$$ (It is possible that $x$ and $z$ are switched, but that case is equivalent because we have made no distinction between these vertices—they can be interchanged.) Note also that the numbers $p$ and $q$ are both odd (in the example, $p = 7$ and $q = 3$). This is because the path $y, a_1, \ldots, a_p, z$ has the same number of edges in $M_1$ and $M_2$, hence has an even number of edges and an odd number of vertices. Meanwhile, $|V(C)| = 4 + p + q$ is even, so $p$ and $q$ have the same parity.

Now, the edge set

$$M^* = \{a_1a_2, \ldots, a_{p-2}a_{p-1}, a_pz, yx, b_1b_2, \ldots, b_{q-2}b_{q-1}, b_qw\} \subseteq E(G)$$

(shown in green below) is a perfect matching on $V(C)$. Since $M_1 - E(C)$ (shown in yellow) is a perfect matching on $V(G) - V(C)$, it follows that $(M_1 - E(C)) \cup M^*$ is a perfect matching of $G$, as desired. □

---

3 In more detail: If $C$ were a path, then both of its endpoints would have to be in $V(M_1 \triangle M_2)$. OTOH, $V(M_1 \triangle M_2) \subseteq V(M_1) \triangle V(M_2)$ for any two matchings $M_1, M_2$ (this isn’t hard to see). In this case, they’re both perfect matchings, so $V(M_1) \triangle V(M_2) = \emptyset$.  

39
Corollary 3.25 (Berge–Tutte Formula). Let $G = (V, E)$, $n = n(G)$, and for $S \subseteq V$ define

$$u(S) = o(G - S) - |S|.$$ 

Let $m = \max\{u(S) \mid S \subseteq V(G)\}$. Then

$$\alpha'(G) = \frac{n - m}{2}. \tag{6}$$

The number $u(S)$ measures the extent to which $S$ forms an obstruction to the existence of a perfect matching. Note that $u(\emptyset) = 0$, so the formula does say that $\alpha' \leq n/2$. Also, Tutte’s condition says precisely that $\alpha' = n/2$ iff $u(S) \leq 0$ for all $S$, so formula (6) generalizes Tutte’s theorem.

**Proof.** **Step 1:** Prove that $\alpha' \leq (n - m)/2$.

This is the easier step. Let $M$ be a matching and $S \subseteq V(G)$. For each odd component $H$ of $G - S$, either $H$ has a vertex whose spouse is in $S$, or else $H$ has an $M$-unmatched vertex. Therefore, there are at least $u(S)$ vertices which are not matched by $M$, and at most $n - u(S)$ matched vertices. So $|V(M)| \leq n - u(S)$. This is true for all $S$, so $|V(M)| \leq n - m$, and dividing by 2 gives the desired inequality.

**Step 2:** Prove that $\alpha' \geq (n - m)/2$. First note that

$$m \geq u(\emptyset) = o(G) \geq 0 \quad \text{and} \quad m \equiv n \pmod{2}$$

by Lemma 3.23. Let $\tilde{G} = G \vee K_m$, where $\vee$ denotes the join. That is, $\tilde{G}$ is obtained from the disjoint union $G + K_m$ by adding an edge between every vertex of $\tilde{G}$ and every vertex of $K_m$. (If you like, $\tilde{G} = G + K_m$.)

We claim that $\tilde{G}$ satisfies Tutte’s Condition (5). (Details left to reader; it’s routine using the definition of join.)

By Tutte’s 1-Factor Theorem, $\tilde{G}$ has a perfect matching $M$.

- At most $m$ edges of $M$ have endpoints in $K_m$.
- Deleting all such edges yields a matching of $G$ that matches at least $n(\tilde{G}) - 2m = n - m$ vertices.
- So $\alpha' \geq (n - m)/2$. \hfill \Box

Like Hall’s Theorem 3.13 (which it implies), the Tutte-Berge Theorem does not yield an efficient algorithm for calculating $\alpha'$ — you would have to calculate $u(S)$ for every $S \subseteq V(G)$ — but it is a useful theoretical tool. Here is a famous corollary.

**Theorem 3.26** (Petersen’s theorem). Let $G$ be a 3-regular simple graph with no cut-edge. Then $G$ has a perfect matching.

**Proof.** Fix $S \subseteq V(G)$ and let $k = o(G - S)$. If $k = 0$ then there is nothing to prove. Otherwise, let $H$ be some odd component of $G_S$. Then $a = \sum_{x \in H} d_G(x) = 3|H|$ is odd, but on the other hand $b = \sum_{x \in H} d_H(x)$ is even. So $a - b$, which is the number of edges from $H$ to $S$, is a positive odd number, but it cannot be 1 because then the edge it counts would be a cut-edge, so it is at least 3. This is true for every component $H$, so the number of edges from $G - S$ to $S$ is at least $3o(G - S)$, but on the other hand it is also at most $3|S|$, again because $G$ is 3-regular. It follows that $|S| \geq o(G - S)$, and we have proved that $G$ satisfies Tutte’s condition. \hfill \Box

Hall’s Marriage Theorem can be proved from Tutte’s 1-Factor Theorem; this is left as an exercise.
4. Connectivity, Cuts, and Flows

4.1. Vertex connectivity. Let \( G = (V, E) \) be a simple connected graph, and let \( n = n(G) \geq 2 \).

**Definition 4.1.** A **separator** (vertex cut, separating set) of \( G = (V, E) \) is a vertex set \( S \subseteq V \) such that \( G \setminus S \) is disconnected or has only one vertex. Two vertices \( x, y \) are **separated** by \( S \) if they are in different components of \( G \setminus S \) (equivalently, every \( x, y \)-path has an internal vertex in \( S \)).

In the following figure, the vertex sets circled in red are separators.

For example, \( N(v) \) is a separator for any \( v \in V \), and a cut-vertex is just a separator of cardinality 1. If \( G \cong C_n \), then any two nonadjacent vertices of \( G \) form a separator.

**Definition 4.2.** Let \( x \neq y \in V(G) \). The **local (vertex) connectivity** of the pair \( x, y \) is

\[
\kappa(x, y) = \kappa_G(x, y) = \min\{|S| : S \text{ is an } x, y\text{-separator}\}
\]

and the **(vertex) connectivity** of \( G \) is

\[
\kappa = \kappa(G) = \min_{x, y} \kappa_G(x, y).
\]

The graph \( G \) is **\( k \)-connected** iff \( \kappa(G) \geq k \).

Some easy observations:

- \( \kappa(G) = 0 \iff G \) is not connected, or \( G = K_1 \).
- \( \kappa(G) = 1 \iff G \) is connected but has a cut-vertex.
- \( \kappa(G) = n(G) - 1 \iff G \cong K_n \).
- If \( H \) is a spanning subgraph of \( G \), then every separator of \( G \) is also a separator of \( H \), so \( \kappa(H) \leq \kappa(G) \).
- Loops and parallel edges do not affect connectivity.
- \( \kappa(G) \leq \delta(G) \), because for every \( v \in V(G) \), the set \( N(v) \) is a separator. Note that \( v \) is an isolated vertex in \( G - N(v) \). This even works if \( G = K_n \).

As Diestel points out, it seems a little unnatural to define connectivity in this way. Since “connected” means that there is a path joining any two vertices, it seems as though “\( k \)-connected” ought to mean that there are \( k \) different paths between any two vertices. In fact “different” is not the right notion here: “disjoint” is.

**Definition 4.3.** Let \( x, y \in V(G) \). A **disjoint path family**\(^4\) for \( x, y \) (for short, “\( x, y \)-DPF”) is a family of distinct \( x, y \)-paths \( P = \{P_1, \ldots, P_k\} \) such that no two of the \( P_i \) have a common vertex other than \( x \) and \( y \). We will temporarily denote the maximum size of an \( x, y \)-DPF by \( \lambda(x, y) \).

\(^4\)This terminology is not standard. West calls such a family “internally disjoint.” Diestel just says “disjoint.” Other sources use terms such as “independent.”
The left-hand graph in Figure 4.1(a) shows an \( x,y \)-DPF, and the three red vertices form an \( x,y \)-separator. In the right-hand graph (b), the path family shown is not a DPF, because some vertices belong to more than one path in the family. (It is true that no vertex other than \( x,y \) belongs to all of the paths, but that does not matter — in order to qualify as a DPF, no other vertex can belong to even two of the paths.)

In fact, the graph in Figure 4.1(b) does have an \( x,y \)-DPF of size 3, shown in Figure 4.1(a). The red circled vertices form a \( x,y \)-separator; note that there is a bijection between vertices in the separator and paths in the DPF. If the dashed edge were removed, then the largest \( x,y \)-DPF would have size 2 and there would be a separator of size 2, as in Figure 4.1(b).

In general, every \( x,y \)-separator must have cardinality at least \( \lambda(x,y) \). In other words, for each pair \( x,y \) there is a weak duality relation

\[
\max\{|P| : \text{\( P \) is an \( x,y \)-DPF}\} \leq \min\{|S| : \text{\( S \) is an \( x,y \)-separator}\}.
\]

**Theorem 4.4 (Menger’s Theorem).** Equality holds in (7) for all \( x,y \); that is, \( \kappa(x,y) = \lambda(x,y) \). In particular, \( G \) is \( k \)-connected if and only if every pair of vertices admits a DPF of size at least \( k \).

These two assertions are sometimes called the local and global formulations of Menger’s theorem, respectively.

One direction is clear: if \( G \) has a separator \( S \) of size \( k - 1 \), then no pair \( x,y \) of vertices separated by \( S \) has a DPF of size \( k \), since every \( x,y \)-path has to use a vertex of \( S \). The other direction is the interesting one. There are three proofs in Diestel, none of which we will do — we are going to derive it as a corollary of the more powerful Max-Flow/Min-Cut Theorem.

**Warning! Warning! Warning!** Disjointness is a property of families of paths, not of individual paths. There is no such thing as a “disjoint path.” If you ever find yourself saying, “Let \( \mathcal{P} \) be a disjoint path family; add more disjoint paths to \( \mathcal{P} \) until we have a total of \( \lambda(u,v) \) paths,” you have made a (very common) mistake.
In particular, you cannot necessarily construct a maximal DPF greedily — that is, not every maximal DPF is maximum. For example, if $P_1$ and $P_2$ are the yellow and green paths in Figure 4.1(b), then every other $x, y$-path has an internal vertex in common with at least one of $P_1$ or $P_2$ in an internal vertex, so $P = \{P_1, P_2\}$ is maximal, but it is not maximum because there is a larger $x, y$-DPF (Figure 4.1(a)).

4.2. Edge connectivity.

**Definition 4.5.** Let $G$ be a connected simple graph. The edge-connectivity is

$$\kappa' = \kappa'(G) = \min\{|F| : F \subseteq E, G - F \text{ not connected}\}.$$ 

Such a set $F$ is called a disconnecting set. (This terminology is in West but not Diestel.) The graph $G$ is $k$-edge-connected iff $\kappa'(G) \geq k$.

Some observations:

- $\kappa'(G) > 0 \iff G$ is connected $\iff \kappa(G) > 0$.
- $\kappa'(G) \geq 2 \iff G$ is connected and has no cut-edge $\iff G$ is connected and is the union of cycles.
- $\kappa'(G)$ is not affected by loops, but can be affected by parallel edges. E.g., if $H$ is formed from $G$ by cloning every edge, then $\kappa'(H) = 2\kappa'(G)$.
- For every $v \in V$, deleting the set $E(v)$ of edges incident to $v$ isolates $v$. Therefore, $\kappa'(G) \leq \delta(G)$.
- $\kappa(G)$ and $\kappa'(G)$ need not be equal. For example, the bowtie graph has $\kappa = 1$ and $\kappa' = 2$.

![Bowtie Graph](image)

**Theorem 4.6.** Every simple graph $G$ satisfies $\kappa \leq \kappa' \leq \delta < n$. Moreover, given any integers $\kappa \leq \kappa' \leq \delta < n$, there exists a simple graph with those parameters.

The proof is left as an exercise.

The following notation will be useful. If $X$ and $Y$ are disjoint subsets of $V(G)$, then we can write

$$[X, Y] = [X, Y]_G = \{e \in V(G) \mid e \text{ has one endpoint in } X \text{ and its other endpoint in } Y\}.$$ 

Also, if $A$ is an edge set and $x$ is a vertex, we write $A(x)$ for the set of edges in $A$ incident to $x$. E.g., $E(x) = \{\{x\}, V\setminus\{x\}\}_G$.

**Definition 4.7.** A cut or edge cut is a set of the form $[S, \bar{S}]$, where $\emptyset \neq S \subseteq V$ and $\bar{S} = V \setminus S$.

**Proposition 4.8.** Let $F \subseteq E(G)$. Then $G - F$ is disconnected iff $F$ contains a cut.

*Proof.* If $F \supseteq [S, \bar{S}]$ then $G - F$ contains no path from $S$ to $\bar{S}$. OTOH, if $G - F$ is disconnected then we can take $S$ to be the vertex set of any component of $G - F$. □

This proposition seems trivial but is very useful. Given an edge set $F$ that you want to show is a disconnecting set, it is typically easier and more natural to construct a vertex set $S$ such that $F \supseteq [S, \bar{S}]$ than it is to show directly that $G - F'$ is not connected.

**Remark 4.9.** A cut can strictly contain another cut. For example, consider a 4-cycle with vertices labeled 1,2,3,4 in cyclic order. Let $S = \{1, 2\}$ and $T = \{1, 3\}$. Then $[T, T]$ is the entire edge set and $[S, \bar{S}]$ is not. More generally, if $G$ is $x, y$-bipartite, then $[X, Y] = E(G)$, which is certainly not a minimal cut unless $G = K_2$.  

43
Definition 4.10. A bond is a minimal cut.

Proposition 4.11. Let $G$ be connected and let $F = [S, \bar{S}]$ be a cut. Then $F$ is a bond if and only if $c(G - F) = 2$ (i.e., $G - F$ has exactly two components).

Proof. If $F$ is a bond then by definition $G - F + e$ is connected for every $e \in F$, so $c(G - F)$ must equal 2.

On the other hand, if $G - F$ has exactly two components, then they must be $G|_S$ and $G|_{\bar{S}}$ for some $\emptyset \neq S \subseteq V$. So every $e \in F$ has one endpoint in each of $S$ and $\bar{S}$, so $G - F + e$ is connected. Therefore no proper subset of $F$ is a disconnecting set, hence $F$ is a bond. □

Bonds behave like cycles. A spanning tree is an edge set that contains no cycle and meets every bond (since if it were disjoint from some bond then it would be disconnected). This is equivalent to saying that the complement of a spanning tree contains no bond and meets every cycle.

Proposition 4.12. Let $C, C'$ be cycles and let $B, B'$ be bonds.

1. $C \triangle C'$ is the (edge-)disjoint union of cycles.
2. If $C \neq C'$ are cycles in $G$ and $e \in C \cap C'$, then $(C \cup C') - \{e\}$ contains a cycle.
3. $B \triangle B'$ is a cut.
4. If $B, B'$ are bonds in $G$ and $e \in B \cap B'$, then $(B \cup B') - \{e\}$ contains a bond.

The proofs are left to the reader. They are both closely related to the exchange rules for spanning trees (if $T, T'$ are spanning trees and $e \in T - T'$, then there is some edge $e' \in T' - T$ so that $T - e + e'$ is a spanning tree).

Sneak preview: if $G$ is planar, then there is a graph $G^*$ called the planar dual of $G$, such that cycles of $G$ correspond to bonds of $G^*$, and vice versa.

Definition 4.13. Let $x, y \in V(G)$. An edge-disjoint path family for $x, y$ (for short, “$x, y$-EPF”) is a family of distinct $x, y$-paths $P = \{P_1, \ldots, P_k\}$ such that no two of the $P_i$ have a common edge. We will temporarily denote the maximum size of an $x, y$-EPF by $\lambda(x, y)$.

Every DPF is an EPF, but not vice versa. Therefore, $\lambda'(x, y) \geq \lambda(x, y)$ for all $x, y$.

Note that every bond separating $x$ and $y$ must have cardinality at least $\lambda'(x, y)$. In other words, we have a weak duality relation

$$\max\{|P| : P \text{ is an } x, y\text{-EPF}\} \leq \min\{|[S, \bar{S}]| : x \in S, y \notin S\}.$$  

(8)

Theorem 4.14 (Menger’s Theorem, edge version). Equality holds in (8) for all $x, y$, that is, $\kappa'(x, y) = \lambda(x, y)$. In particular, $G$ is $k$-edge-connected if and only if every pair of vertices admits an EPF of size at least $k$.

Again, the $\Leftarrow$ direction is easy, and the other direction is the hard one. We will soon show (in §4.5) that the edge version of Menger’s theorem is a consequence of vertex version.
4.3. The structure of 2-connected and 2-edge-connected graphs. There are several useful equivalent conditions for 2-connectivity.

**Definition 4.15.** Let $G$ be a connected graph. Let $v,w \in V(G)$, possibly equal. Construct a new graph $G'$ from $G$ by adding a new path

$$v = x_0, e_0, x_1, \ldots, x_{n-1}, e_{n-1}, x_n = w.$$ 

This path is called an ear. It is a closed ear if $v = w$ (in which case the trail is a cycle) and an open ear if $v \neq w$ (in which case the trail is a path).

**Theorem 4.16 (Characterization of 2-Connectivity).** Let $G$ be a simple connected graph with $n(G) \geq 3$. The following are equivalent:

(A) $G$ is 2-connected.
(B) Every two vertices of $G$ lie on a common cycle.
(C) Every two edges lie on a common cycle.
(D) $G$ has an open ear decomposition $G = C \cup P_1 \cup \cdots \cup P_k$, where $C$ is a cycle and each $P_i$ is an open ear.

**Sketch of proof.** (A $\iff$ B): This is a special case of Menger’s Theorem, which we will prove later.

(D $\implies$ A): The original cycle is 2-connected, and so is any graph obtained by adjoining ears to a 2-connected graph.

(A/B $\implies$ C): Let $e, e'$ be edges. If $e, e'$ share an endpoint, say $e = wx$ and $e' = yz$, then by 2-connectivity we can find an $x, y$-path $P$ in $G - y$, and then $e + P + e'$ is a cycle. Otherwise, let $e = wx$ and $e' = yz$. Construct a new graph $G'$ by adding vertices $s, t$ and edges $sw, sx, ty, tz$. This is equivalent to adding two open ears, so $G'$ is 2-connected. In particular $G'$ has two disjoint $s, t$-paths $P, P'$. WLOG, $P = sw \cdots yt$ and $P' = sx \cdots zt$. Then $P + P' - sw - sx - ty - tz + e + e'$ is a cycle containing $e$ and $e'$.
(C $\implies$ D): If $G$ is a graph satisfying (C), then the following algorithm produces an ear decomposition of $G$:

Let $G_0 \subseteq G$ be any cycle. (So $G_0$ is 2-connected.)
Initialize $i = 1$.
During the $i^{th}$ step:
- Choose any edges $e \in E(G) - E(G_i)$ and $f \in E(G_i)$.
- Let $C$ be a cycle containing $e$ and $f$. (Such a cycle must exist by condition (C).)
- Let $P_i$ be the smallest path in $C$ that contains no edges of $G_{i-1}$.
- This is the new ear; set $G_i := G_{i-1} \cup P_i$.
- Increment $i$.
Repeat until $E(G_i) = E(G)$.

There is an analogous theorem for 2-edge-connected graphs. For the purpose of this statement, define a circuit to be a closed walk without repeated edges (but allowing repeated vertices).

**Theorem 4.17 (Characterization of 2-Edge-Connectivity).** Let $G$ be a connected graph with $n(G) \geq 2$. The following are equivalent:

(A) $G$ is 2-edge-connected.
(B) Every two vertices $u, v \in V(G)$ lie on a common circuit.
(C) $G$ has an ear decomposition $G = C \cup P_1 \cup \cdots \cup P_s$, where $C$ is a cycle and each $P_i$ is an ear (either open or closed).
(D) $G$ has a strong orientation, i.e., it is possible to orient all edges so that every edge belongs to a directed cycle.

The figure on the left is an example of a closed-ear decomposition. Note that $P_3$ is a closed ear—it can be regarded as a closed path from $v$ to $v$, where $v$ is the indicated vertex of $C \cup P_1 \cup P_2$. Indeed, the graph is 2-edge-connected but not 2-connected.

The figure on the right shows how to translate a closed-ear decomposition into a strong orientation (an example of the implication (C) $\implies$ (D) of the theorem): just orient the original cycle consistently (which can be done in one of two ways), and whenever you add an ear, orient it consistently (which again can be done in one of two ways).
4.4. **Counting strong orientations.** Let $G$ be a graph (with loops and multiple edges allowed). Let $S(G)$ be the set of strong orientations of $G$, and let $s(G) = |S(G)|$. What can we say about this number?

- $s(G) > 0$ iff $G$ is 2-edge-connected, by Thm. 4.17.
- $s(G)$ is always even (unless $G = K_1$), because reversing the direction of every edge preserves strongness.
- Cycles have two strong orientations (“clockwise” and “counterclockwise”).
- For consistency, this should still be true for the loop $C_1$. In fact, adding a loop to $G$ should double the value of $s(G)$, since the loop itself can be oriented in one of two ways.
- The value of $s(K_n)$ is not so obvious. Starting with $n = 1$, the sequence begins 1, 0, 2, 24, 544, 22320, 1677488, ... 

**Proposition 4.18.** The invariant $s$ satisfies a deletion/contraction recurrence: $s(G) = s(G/e) + s(G - e)$.

**Proof.** Let $D$ be a strong orientation of $G$, and suppose that there exists $e \in E(G)$ such that reversing $e$ in $D$ also gives a strong orientation. Then $D - e$ is a strong orientation of $G - e$. On the other hand, each strong orientation of $G - e$ certainly gives rise to a pair of strong orientations of $G$. So

$$s(G - e) = \frac{\# \{ D \in S(G) \mid e \text{ is reversible} \}}{2}.$$ 

Now let $D'$ be a strong orientation of $G/e$. Such a thing could come from a pair of strong orientations of $G/e$ that are identical except for the orientation of $e$ (so $e$ is reversible), or from a single strong orientation in which $e$ is not reversible. Therefore,

$$s(G/e) = \frac{\# \{ D \in S(G) \mid e \text{ is reversible} \}}{2} + \# \{ D \in S(G) \mid e \text{ is not reversible} \}.$$ 

Adding these two equations gives the desired recurrence. \[ \square \]

This should remind you of the deletion-contraction recurrence for $\tau(G)$. Stay tuned!

4.5. **Menger implies edge-Menger.**

**Definition 4.19.** Let $G$ be a simple graph. The **line graph** $L(G)$ is defined by

- $V(L(G)) = E(G)$,
- $E(L(G)) = \{ ef : e, f \text{ have a common endpoint} \}$.

Note that the edge set of a (closed) trail in $G$ corresponds to the vertex set of a path in $L(G)$, as in the below figure.
Theorem 4.20. The vertex version of Menger’s Theorem implies the edge version.

Proof. Assume that the vertex version of Menger’s Theorem holds. Let \( G \) be a graph, \( x, y \in V(G), xy \notin E(G) \). Let \( G' \) be the graph formed from \( G \) by adding vertices \( s, t \) and edges \( sx, yt \). Construct the line graph \( L(G') \).

We are now going to apply vertex-Menger to \( L(G') \) in order to deduce edge-Menger for \( G \).

First, observe that a set \( A \subseteq E(G) \) disconnects \( x \) from \( y \) in \( G \) if and only if the corresponding vertices separate \( sx \) from \( yt \) in \( L(G') \). Therefore,

\[
\kappa'_G(x, y) = \kappa_{L(G')}(sx, yt).
\]  

Second, the vertex version of Menger’s theorem implies that

\[
\kappa_{L(G')}(sx, yt) = \lambda_{L(G')}(sx, yt).
\]  

Third, observe that for each \( x, y \)-path \( P \) in \( G \), there is a corresponding \( s, t \)-path \( P' = sx + P + yt \) in \( G' \) and an \( sx, yt \)-path \( \hat{P} \) in \( L(G') \), and that \( P, Q \) are edge-disjoint if and only if \( \hat{P}, \hat{Q} \) are internally disjoint. Therefore,

\[
\lambda_{L(G')}(sx, yt) = \lambda'_G(x, y)
\]  

and chaining [9], [10], and [11] together completes the proof. \( \square \)
4.6. Network flows.

**Definition 4.21.** A network or *s, t*-network \( N = (G,s,t,c) \) consists of a simple digraph \( G = (V,E) \) with two distinguished vertices \( s, t \), called the **source** and **sink** respectively, and a **capacity function** \( c : E \rightarrow \mathbb{N}_{>0} \). We may assume that \( G \) is a simple digraph: it has no loops, and that for every \( x,y \in V \) there is at most one edge of the form \( \overrightarrow{xy} \) and at most one edge of the form \( \overrightarrow{yx} \).

We want to think of a network as modeling a situation where stuff (data, traffic, liquid, electrical current, etc.) is flowing from source \( s \) to sink \( t \). The capacity of an edge is the amount of stuff that can flow through it (or perhaps the amount of stuff per unit time). This is a very general model that can be specialized to describe cuts, connectivity, matchings and other things in directed and undirected graphs.

A flow on \( N \) is a function \( f : E \rightarrow \mathbb{R} \) that satisfies the constraints

\[
\begin{align*}
0 \leq f(e) \leq c(e) & \quad \forall e \in E \quad \text{(the capacity constraints)}, \\
\text{For} \quad \forall v \in V \setminus \{s,t\}, \quad f_{\text{in}}(v) = f_{\text{out}}(v) & \quad \text{(the conservation constraints)},
\end{align*}
\]

where for \( v \in V \) we define

\[
\begin{align*}
f_{\text{in}}(v) = \sum_{e=v\overrightarrow{v}} f(e), \quad f_{\text{out}}(v) = \sum_{e=v\overleftarrow{v}} f(e). \quad \text{(14)}
\end{align*}
\]

The function \( f(e) = 0 \) is of course a flow. Here is a nontrivial example. I will consistently use blue for capacities and red for flows.
Note that the conservation constraints say that flow cannot accumulate at any internal vertex.

The value \(|f|\) of a flow \(f\) is the net flow into the sink:

\[
|f| := f_{in}(t) - f_{out}(t) = f_{out}(s) - f_{in}(s).
\]

Note that the second equality follows from the conservation constraints. The flow \(f\) shown above has \(|f| = 3\).

**Max-Flow Problem:** Given a source-sink network \((G, s, t, c)\), find a flow of maximum value.

We would like to have a way of increasing the value of a given flow \(f\), or showing that no such way exists. This ought to remind you of the Augmenting Path Algorithm, and in fact we can start by looking for an “\(f\)-augmenting path”—an \(s,t\)-path \(P \subseteq N\) in which no edge of \(P\) is being used to its full capacity, that is, such that

\[
f(e) < c(e) \quad \forall e \in P.
\]

In this case, we can define \(z = \min\{c(e) - f(e) : e \in P\}\) and define a new flow \(\tilde{f}\) by

\[
\tilde{f}(e) = \begin{cases} f(e) + z & \text{for } e \in P, \\ f(e) & \text{for } e \notin P. \end{cases}
\]

This will preserve the conservation and capacity constraints, and will increase the value of the flow by \(z\).

However, there can be nonmaximum flows where no such path \(P\) exists. Consider the network shown in Figure 4.6. Continuing the analogy with matchings and the APA, the flow \(f\) on the left is “maximal”, in the sense that there does not exist any flow \(f'\) such that \(|f'| > |f|\) and \(f'(e) \geq f(e)\) for every \(e \in E\). However, it is not maximum: \(|f| = 1\), while the flow \(g\) on the left has \(|g| = 2\).

**Figure 3.** Two “maximal” flows, one of which \((f)\) is not maximum.

There is a more general way to increase flow: Allow the augmenting path \(P\) to contain edges that point in the wrong direction, but along which the flow is nonzero. As far as the conservation constraints and the value of the flow is concerned, decreasing “backward” flow is equivalent to increasing “forward” flow. The “forward” edges \(a,c\) are not being used to full capacity, and the “backward” edge \(b\) contains flow “in the wrong direction”. So we can define a new flow \(\hat{f}\) by

\[
\begin{align*}
\hat{f}(a) &= f(a) + 1, \\
\hat{f}(b) &= f(b) - 1, \\
\hat{f}(c) &= f(c) + 1, \\
\hat{f}(e) &= f(e) \text{ for all other edges } e.
\end{align*}
\]

Then \(\hat{f}\) satisfies the capacity and conservation constraints, and \(|\hat{f}| = |f| + 1\). (In fact \(\hat{f} = g\).)
**Definition 4.22.** Let \( f \) be a flow in an \( s,t \)-network \( N = (G, s, t, c) \). Let \( P \) be an \( s,t \)-path in \( G \), which may include backward edges \( \overleftrightarrow{e} \) as well as forward edges \( \overrightarrow{e} \in P \). The tolerance of an edge \( e \in P \) is defined as
\[
\epsilon(e) = \begin{cases} 
 c(e) - f(e) & \text{if } \overleftrightarrow{e} \in P, \\
 f(e) & \text{if } \overrightarrow{e} \in P,
\end{cases}
\]
and the tolerance of the path \( P \) is
\[
\epsilon(P) = \min_{e \in P} \epsilon(e).
\]
The path \( P \) is **augmenting** for \( f \) if \( \epsilon(P) > 0 \).

The proof of the following proposition is then completely routine.

**Proposition 4.23.** If \( P \) is an \( f \)-augmenting path, then the function \( \tilde{f} \) defined by
\[
\tilde{f}(e) = \begin{cases} 
 f(e) + \epsilon & \text{if } \overleftrightarrow{e} \in P, \\
 f(e) - \epsilon & \text{if } \overrightarrow{e} \in P, \\
 f(e) & \text{otherwise},
\end{cases}
\]
is a flow (i.e., it satisfies the capacity and conservation constraints), and \( |f'| = \epsilon + |f| \).

The dual problem to the Max-Flow problem is the **Min-Cut problem**.

**Definition 4.24.** Let \( N = (G, s, t, c) \) be an \( s,t \)-network. A **source-sink cut** is a cut of the form \([S, T]\), where \( V = S \cup T, s \in S, \) and \( t \in T \). The **capacity** of the cut is
\[
c(S, T) = \sum_{e \in E} c(e).
\]
In the figure above, $S = \{s, a, b, p, q\}$ (gold) and $T = \overline{S} = \{t, r, y, z\}$ (pink). The resulting source-sink cut is $[S, T] = \{br, py, qz\}$ (shown with green hash marks), so $c(S, T) = 2 + 1 + 2 = 5$.

**Min-Cut Problem: Find a source-sink cut of minimum capacity.**

A source-sink cut can be thought of as a bottleneck: a channel through which all flow must pass. Therefore, the capacity of any cut should be an upper bound for the maximum value of a flow — this is the “weak duality” inequality, analogous to the easy directions of results such as the König-Egerváry theorem and the various versions of Menger’s theorem.

For a flow $f$ and a vertex set $A \subseteq V$, define

\begin{align}
    f_{\text{in}}(A) &= \sum_{e \in [A, \overline{A}]} f(e), \\
    f_{\text{out}}(A) &= \sum_{e \in [A, \overline{A}]} f(e).
\end{align}

**Proposition 4.25.** Let $f$ be a flow, and let $A \subseteq V$. Then:

\begin{align}
    f_{\text{out}}(A) - f_{\text{in}}(A) &= \sum_{w \in A} (f_{\text{out}}(w) - f_{\text{in}}(w)).
\end{align}

In particular, if $[S, T]$ is a source-sink cut, then

\begin{align}
    f_{\text{out}}(S) - f_{\text{in}}(S) &= f_{\text{in}}(T) - f_{\text{out}}(T) = |f| \leq c(S, T).
\end{align}

That is, the Max-Flow and Min-Cut problems are weak dual.
Proof. We just need some careful bookkeeping. First, by the definition of $f_{\text{in}}$ and $f_{\text{out}}$, we have

$$f_{\text{out}}(A) - f_{\text{in}}(A) = \left( \sum_{e \in [A, A]} f(e) + \sum_{e \notin [A, A]} f(e) \right) - \left( \sum_{e \in [A, A]} f(e) + \sum_{e \notin [A, A]} f(e) \right)$$

$$= \sum_{e : \text{head}(e) \in A} f(e) - \sum_{e : \text{tail}(e) \in A} f(e)$$

$$= \sum_{w \in A} \left( \sum_{e : \text{head}(e) = w} f(e) - \sum_{e : \text{tail}(e) = w} f(e) \right)$$

$$= \sum_{w \in A} (f_{\text{out}}(w) - f_{\text{in}}(w)).$$

establishing (16).

Now let $[S, T]$ be a source-sink cut. The conservation constraints (13) together with (16) give

$$f_{\text{out}}(S) - f_{\text{in}}(S) = \sum_{w \in S} f_{\text{out}}(w) - f_{\text{in}}(w)$$

$$= f_{\text{out}}(s) - f_{\text{in}}(s)$$

$$= |f|$$

$$= f_{\text{in}}(t) - f_{\text{out}}(t) = f_{\text{in}}(T) - f_{\text{out}}(T).$$

and then the capacity constraints (12) give

$$|f| = f_{\text{out}}(S) - f_{\text{in}}(S) \leq f_{\text{out}}(S) \leq \sum_{e \in [S, T]} c(e) = c(S, T).$$

□

4.7. The Ford-Fulkerson algorithm. In fact, the Max-Flow and Min-Cut problems are strongly dual. They can be solved simultaneously in finite time by the following simple but very powerful algorithm, due to Ford and Fulkerson.

### The Ford-Fulkerson Algorithm

**Input:** a network $N = (G, s, t, c)$  
**Output:** a maximum flow $f$ and minimum $s,t$-cut $[S,T]$

*Initialize* $f(e) = 0 \ \forall e.$

*Repeat:*
  - Let $S$ be the set of all vertices reachable from $s$ by an $f$-augmenting path  
    - If $t \in S$ ("breakthrough"), then increase flow along some augmenting path until breakthrough does not occur.  
    - Return the flow $f$ and the cut $[S, S].$

**Theorem 4.26** (The Max-Flow/Min-Cut Theorem — “MFMC”). *The Ford-Fulkerson algorithm finishes in finite time and computes a maximum flow and a minimum cut.*
Proof. Since everything in sight is an integer, each instance of breakthrough increases $|f|$ by at least 1. Therefore, the algorithm will terminate in a number of steps equal to or less than the minimum capacity of an $s,t$-cut.

Let $f$ and $[S,T]$ be the output of the FFA. The fact that breakthrough did not occur means that every forward edge of $[S,T]$ is being used to full capacity. That is,

$$f(e) = c(e) \quad \forall e \in [S,T] \quad \text{and} \quad f(e) = 0 \forall e \notin [S,T].$$

Therefore

$$|f| = f_{\text{out}}(S) - f_{\text{in}}(S) \quad \text{(by (17))}$$

$$= \sum_{e \in [S,T]} f(e) - \sum_{e \in [T,S]} f(e) \quad \text{(by definition)}$$

$$= \sum_{e \in [S,T]} c(e) \quad \text{(by the last remark)}$$

$$= c(S,T)$$

and by weak duality, $f$ is a maximum flow $f$ and $[S,T]$ is a minimum source-sink cut. □

**Example 4.27.** Let $N$ be the network shown.

![Network Diagram](image)

Initialize $f$ to be the zero flow and work through the algorithm. Note that there may be several possible augmenting paths in each iteration, so in that sense the algorithm is not deterministic.

**Step 1:**

Augmenting path: $P = s,a,d,c,t$

Edge tolerances: $\epsilon(sa) = 64, \epsilon(ad) = 30, \epsilon(dc) = 24, \epsilon(ct) = 25$

Path tolerance: $\min\{64,30,24,25\} = 24$
Step 2:
Augmenting path:  $P = s, b, d, t$
Tolerance:  $\epsilon(P) = \min\{27, 20, 57\} = 20$

Step 3:
Augmenting path:  $P = s, a, c, d, t$. Note that $\overrightarrow{dc} \in E$, so we have a backward edge.
Tolerance:  $\epsilon(P) = \min\{40, 49, 24, 37\} = 24$. Note that $\epsilon(\overrightarrow{cd}) = f(\overrightarrow{dc}) = 24$.
New flow:  Add $24$ to $f(s\overrightarrow{a})$, $f(\overrightarrow{ac})$, $f(\overrightarrow{dt})$; subtract $24$ from $f(\overrightarrow{dc})$.

Step 4:
Augmenting path:  $P = s, a, d, t$
Tolerance:  $\epsilon(P) = \min\{16, 6, 13\} = 6$
Step 5:
Augmenting path: \( P = s, a, c, t \)
Tolerance: \( \epsilon(P) = \min\{64 - 54, 49 - 24, 25 - 24\} = 1 \)

\[
\begin{array}{llllll}
\text{Step 6:} & \text{At this point in the algorithm, breakthrough fails, since every edge in the cut } [S = \{s, a, b, c\}, \ T = \{d, t\}] & \text{is either a forward edge being used at capacity (yellow), or a backward edge with flow 0 (blue).}
\end{array}
\]

Moreover, \( c(S, T) = c(\overrightarrow{ad}) + c(\overrightarrow{bd}) + c(\overrightarrow{ct}) = 30 + 20 + 25 = 75 = |f| \). So \( f \) is a maximum flow and \([S, T]\) is a minimum cut. The algorithm has succeeded!!

Note that the algorithm still works if the capacities are required only to be rational, rather than integers. It is really the same problem — since the network has finitely many edges, simply multiply every capacity by the greatest common denominator to convert the problem to an integer one. If the capacities are required to be
real, then the Max-Flow and Min-Cut problems are still strongly dual (in the sense of linear programming), but the Ford-Fulkerson algorithm may not terminate.

Henceforth, we continue to assume that all capacity functions and flows are integers.


Proposition 4.28 (Acyclic flows). Every network $N$ has a maximum flow $f$ that is acyclic in the sense that for every directed cycle $C \subseteq N$, there is at least one edge of $C$ with $f(e) = 0$.

Proof. More generally, any flow $f$ can be “acyclized” as follows. If $C$ is a directed cycle with $f(e) > 0$ for every $e \in C$, then $z = \min\{f(e) : e \in C\} > 0$, and we can define a new flow $\tilde{f}$ by

$$\tilde{f}(e) = \begin{cases} f(e) - z, & e \in C, \\ f(e), & e \notin C. \end{cases}$$

Then $\tilde{f}$ satisfies the capacity and conservation constraints and $|\tilde{f}| = |f|$. Moreover, the flow $\tilde{f}$ has at least one more edge with zero flow than $f$ did, so if we repeat this construction we will eventually produce an acyclic flow. □

Proposition 4.29 (Partitionability of flows). Every acyclic integer flow $f$ can be “partitioned into paths.” That is, there is a family of directed $s,t$-paths $P = \{P_1, \ldots, P_k\}$ such that $k = |f|$ and

$$f(e) = \#\{i \mid e \in P_i\}.$$

Warning: There is certainly no guarantee that $P$ is a DPF or EPF as defined above in §4.1 and §. In fact, typically it will be neither.

Proof. The following algorithm constructs such a family:

- Initialize $P := \emptyset$.
- Start walking from $s$ along edges of positive flow until we reach $t$.
- Put $P$ into $\mathcal{P}$, and reduce flow by 1 along every edge of $P$.
- Repeat until $|f| = 0$.

Since $f$ is acyclic, every walk $P$ is a path. Also, the conservation constraints imply that we never get “stuck” when forming $P$ — after entering an internal vertex of the network, it is always possible to leave. Finally, each time we put a new path into $\mathcal{P}$, the value $|f|$ decreases by 1. □

The warning above applies to networks in general. On the other hand, if we are clever about choosing the capacity function, we will automatically obtain additional constraints on the path family obtained by partitioning.

For example, let $D$ be a digraph with $s,t \in V(D)$. Treat $D$ as a unit-capacity network, i.e., $c(e) = 1$ for every $e \in E$. Then

- $P$ is a (directed) $s,t$-EPF, and
- $c(S,T) = |[S,T]|$ for any source-sink cut $[S,T]$.

Therefore, applying the MFMC in this context, we obtain the directed edge version of Menger’s theorem:

$$\max\{|P| : P \text{ is a directed } s,t\text{-EPF}\} = \min\{|[S,T]| : [S,T] \text{ an } s,t\text{-cut}\} = \kappa'(s,t).$$
This is a common technique to prove min/max theorems in graph theory: transform an arbitrary graph into a source/sink network in some clever way, then apply MFMC to the network to recover the desired result for your original graph.

The undirected vertex version of Menger’s theorem requires a slightly more elaborate transformation.

**Theorem 4.30 (Menger’s Theorem, finally!).** Let $G$ be a simple graph, and let $s,t \in V(G)$ be nonadjacent. Then

$$\max \{|P| : P \text{ is an } x,y\text{-DPF} \} \leq \min \{|S| : S \text{ is an } x,y\text{-separator} \}.$$  

**Proof.** Construct a $s,t$-network $N$ by splitting every internal vertex $v$ into an in-vertex $v^-$ and an out-vertex $v^+$. That is,

- $V(N) := \{s,t\} \cup \{v^-, v^+ : v \in V(G)\setminus\{s,t\}\}$,
- $E(N) := \{v^- v^+ : v \in V(G)\setminus\{s,t\}\}$ ("private"; capacity = 1) \cup \{v^+ w^- : vw \in E(G)\}$ ("public"; capacity = $\infty$).

For every $v \in V(N)\setminus\{s,t\}$, we have $f_{\text{out}}(v^-) = f_{\text{in}}(v^+) \in \{0,1\}$.

Let $f$ be a feasible integer acyclic flow of $N$, partitioned into paths $P'_1, \ldots, P'_k$, where $k = |f|$. Each $P'_i$ has the form

$$s, v^-_1, v^+_1, \ldots, v^-_k, v^+_k, t.$$  

The path $P'_i$ corresponds to the $s,t$-path $P_i$ in $G$ given by $s, v_1, \ldots, v_n, t$, and this correspondence is bijective. Moreover, each pair $\{v^-, v^+\}$ can occur in at most one $P'_i$ (because $c(v^-, v^+) = 1$), so each $v \in V(G)\setminus\{s,t\}$ belongs to at most one $P_i$ — that is, the family $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a DPF! So integer acyclic flows in $N$ correspond exactly to $s,t$-DPF’s in $G$, and

$$\max |f| = \lambda_G(s,t).$$  

Meanwhile, the set of private edges is a $s,t$-cut of capacity $n(G) - 2$. On the other hand, any edge set containing at least one public edge has infinite capacity. Therefore, every minimum source-sink cut consists only of private edges, and the corresponding vertices of $G$ form an $s,t$-cut.

Therefore $\lambda_G(s,t) = \max |f| = \min c(S,T) = \kappa_G(s,t)$.  

Other applications of MFMC, many of which would make good projects, include:

---

5If this bothers you, just define the capacity of the public edges large enough, say, oh, I don’t know, $(73n + 86)!$, so that the sentence following the footnote is true.
• The König-Egerváry theorem can be derived from MFMC. This would make a good homework problem.
• Gale-Ryser Theorem (characterizing degree sequences of a bipartite graph)
• Score vectors of tournaments (i.e., what in-/out- degree sequences can arise in an orientation of $K_n$?)
• Dilworth’s Theorem (chain covers and antichains in posets)

One can also generalize flows and study, say...

• Real-valued networks
• Networks with multiple sinks and sources
• Supplies and demands
• Cost networks (where each edge has a cost per unit flow, and the problem is to find a feasible flow of fixed value and minimum cost)

This is the revision frontier.