Graph Theory

Discrete Math, Spring 2026

Konstantin Chukharev

Graph Theory

- Graphs & digraphs
- Paths & connectivity
- Trees & spanning trees
- Bipartite graphs
- Matchings & Hall's theorem
- Planarity & coloring
- Network flows

Languages & Computation

- Alphabets & formal languages
- Regular expressions
- Finite automata (DFA, NFA)
- Pumping lemma
- Context-free grammars
- Pushdown automata
- Turing machines
- Decidability & complexity

Combinatorics & Recurrences

- Counting principles
- Permutations & combinations
- Inclusion–exclusion
- Partitions & Stirling numbers
- Generating functions
- Recurrence relations
- Asymptotic analysis

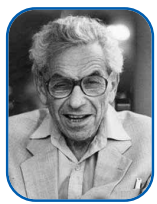
Graph Theory

"The origins of graph theory are humble, even frivolous."

— Norman L. Biggs


Leonhard Euler


William Rowan Hamilton


Paul Erdős


Paul Turán


Frank Ramsey


Frank Harary


Béla Bollobás

Why Graph Theory?

Graphs are *everywhere* — they model relationships, connections, and structures.

Real-world applications:

- Social networks (friendships)
- Computer networks (routers)
- Transportation (roads, flights)
- Biology (protein interactions)
- Chemistry (molecular bonds)

Computer science applications:

- Data structures (linked lists, trees)
- Algorithms (shortest paths, flows)
- Compilers (dependency graphs)
- Databases (query optimization)
- AI (neural networks, knowledge graphs)

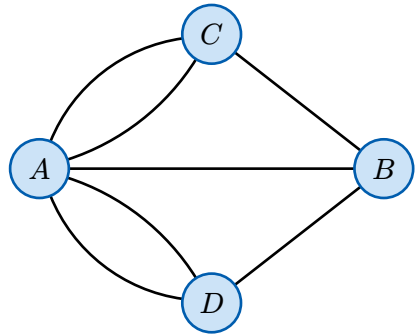
The power of abstraction: By stripping away irrelevant details, graphs let us see the *structure* of a problem. The same algorithm that finds the shortest route between cities also finds the fastest path in a game tree or the most efficient way to schedule tasks.

The Seven Bridges of Königsberg

In 1736, Leonhard Euler solved a famous puzzle:

Can one walk through the city of Königsberg, crossing each of its seven bridges exactly once?

Euler proved this is *impossible* — and in doing so, invented graph theory.



Historical note: This problem marks the birth of *topology* and *graph theory* as mathematical disciplines.

Basic Definitions

What is a Graph?

Graphs as models: Graphs are *mathematical abstractions* for modeling relationships, connections, and structures. Different kinds of relationships lead to different types of graphs.

Definition 1 (Abstract Approach): A *graph* is fundamentally a triple $G = (V, E, F)$, where:

- $V = \{v_1, v_2, \ldots\}$ is a finite set of *abstract vertices* (unique objects)
- $E = \{e_1, e_2, \ldots\}$ is a finite set of *abstract edges* (connections)
- F is a collection of *functions* that capture the graph's structure and semantics

The power of abstraction: Vertices and edges are just *labels* — the functions F define *all* the meaning:

- For *undirected* graphs: $F = \left\{\text{ends} : E \to \binom{V}{2}\right\}$ maps each edge to its two endpoints
- For *directed* graphs: $F = \{\text{begin} : E \to V, \text{end} : E \to V\}$ specify source and target
- For *weighted* graphs: add $\text{weight} : E \to \mathbb{R}$

What is a Graph? [2]

- For *hypergraphs:* incidence : $E \to 2^V$ maps edges to *subsets* of vertices
- For *vertex-labeled* graphs: add label : $V \to \Sigma$ for some alphabet Σ

Notation:

- $V(G)$ denotes the vertex set of graph G
- $E(G)$ denotes the edge set of graph G
- $|V(G)|$ is the *order* of G (number of vertices)
- $|E(G)|$ is the *size* of G (number of edges)

Bonus: This abstract approach handles *multigraphs* (parallel edges) and *loops* naturally — multiple edges in E can map to the same endpoint pair, and a loop edge maps to a singleton set $\{v\}$ or has $\text{begin}(e) = \text{end}(e) = v$.

Structural Representation (Alternative Approach)

Definition 2 (Structural Approach): Instead of abstract edges + functions, we can *encode structure directly* into the edge definition:

- *Undirected:* $E \subseteq \binom{V}{2}$ (unordered pairs $\{u, v\}$)
- *Directed:* $E \subseteq V \times V$ (ordered pairs (u, v))
- *Weighted:* $E \subseteq V \times V \times \mathbb{R}$ (triples (u, v, w))
- *Loops:* Include singletons $\{v\}$ in E or allow (v, v)

Trade-offs:

- *Pros:* Simpler for basic graphs; closer to programming impl (edge lists, adjacency matrices)
- *Cons:* Less flexible; need ad-hoc extensions for weighted graphs, hypergraphs, attributes; mixing structure with semantics

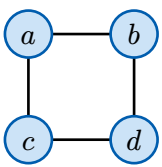
In practice: For this course, we'll mostly use the *structural representation* for simplicity, but keep the *abstract view* in mind — it explains why we can freely add weights, directions, labels, *etc.*

Undirected vs Directed Graphs

Definition 3 (Undirected Graph): In an *undirected graph*, edges are *unordered pairs*:

$$E \subseteq \binom{V}{2} = \{\{u, v\} \mid u, v \in V, u \neq v\}$$

The edge $\{u, v\}$ connects u and v symmetrically.



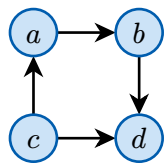
Undirected

Models: Mutual relationships (friendships, two-way roads, chemical bonds)

Definition 4 (Directed Graph): In a *directed graph* (digraph), edges are *ordered pairs*:

$$E \subseteq V \times V$$

The edge (u, v) goes *from* u *to* v.



Directed

Models: One-way relationships (follows, one-way streets, dependencies, function calls)

Simple Graphs, Multigraphs, and Pseudographs

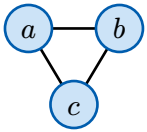
Definition 5:

- A *simple graph* has no *loops* (edges from a vertex to itself) and no *multi-edges* (multiple edges between the same pair of vertices).
- A *multigraph* allows *multi-edges* but no loops.
- A *pseudograph* allows both loops and multi-edges.

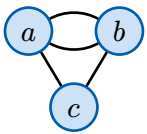
Abstract view: In the function-based approach, these distinctions are natural:

- *Simple:* the "ends" function is *injective* (different edges ⟶ different endpoint pairs)
- *Multigraph:* "ends" can be non-injective; multiple edges map to the same $\{u, v\}$
- *Loops:* "ends" can map an edge to a singleton $\{v\}$ (or $\text{begin}(e) = \text{end}(e)$)

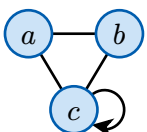
Note: Unless otherwise stated, "graph" means *simple undirected graph* in this course.



Simple



Multigraph



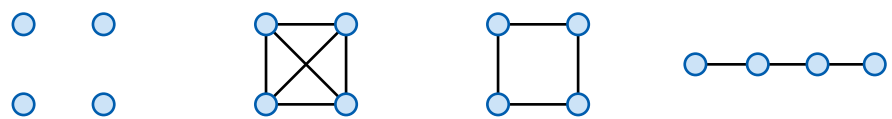
Pseudograph

Special Graphs

Definition 6:

- *Null graph*: no vertices ($V = \emptyset$)
- *Trivial graph*: single vertex, no edges ($|V| = 1$, $E = \emptyset$)
- *Empty graph* $\overline{K}_n$: n vertices, no edges
- *Complete graph* K_n: n vertices, all pairs connected
- *Cycle* C_n: n vertices in a cycle
- *Path* P_n: n vertices in a line

Example:



$\overline{K}_4$ (empty) K_4 (complete) C_4 (cycle) P_4 (path)

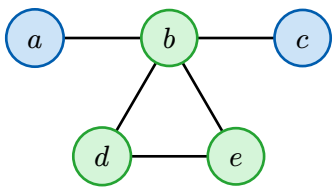
Theorem 1: The complete graph K_n has exactly $\binom{n}{2} = \frac{n(n-1)}{2}$ edges.

Adjacency and Incidence

Definition 7:

- Two vertices u and v are *adjacent* if there is an edge between them: $\{u, v\} \in E$.
- An edge e is *incident* to vertex v if v is an endpoint of e.
- The *neighborhood* of v is $N(v) = \{u \in V \mid \{u, v\} \in E\}$.

Example:



- a and b are *adjacent*
- a and c are *not adjacent*
- Edge $\{a, b\}$ is *incident* to a and b
- $N(b) = \{a, c, d, e\}$

Degree of a Vertex

Definition 8: The *degree* of a vertex v, denoted $\deg(v)$, is the number of edges incident to v.

- $\delta(G) = \min_{v \in V} \deg(v)$ is the *minimum degree*
- $\Delta(G) = \max_{v \in V} \deg(v)$ is the *maximum degree*

Theorem 2 (Handshaking Lemma): For any graph $G = \langle V, E \rangle$:

$$\sum_{v \in V} \deg(v) = 2 \ |E|$$

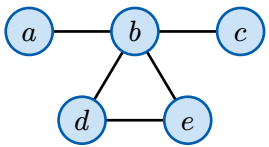
Proof: Each edge contributes exactly 2 to the sum of degrees (once for each endpoint). □

Corollary: The number of vertices with odd degree is always *even*.

Degree Sequences

Definition 9: The *degree sequence* of a graph is the list of vertex degrees in non-increasing order.

Example:



Degrees: $\deg(a) = 1$, $\deg(b) = 4$, $\deg(c) = 1$, $\deg(d) = 2$, $\deg(e) = 2$

Degree sequence: $(4, 2, 2, 1, 1)$

Question: Given a sequence of integers, can we determine if it's the degree sequence of some graph?

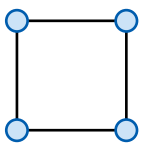
This is the *graph realization problem*.

Regular Graphs

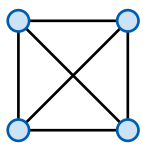
Definition 10: A graph is *r-regular* if every vertex has degree r:

$$\forall v \in V : \deg(v) = r$$

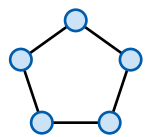
Example:



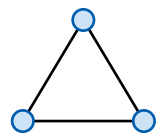
2-regular
(cycle C_4)



3-regular
(complete K_4)



2-regular
(cycle C_5)



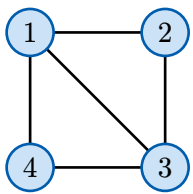
2-regular
(complete K_3)

Graph Representations: Adjacency Matrix

Definition 11: The *adjacency matrix* A of a graph G with n vertices is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 \text{ if } \{v_i, v_j\} \in E \\ 0 \text{ otherwise} \end{cases}$$

Example:



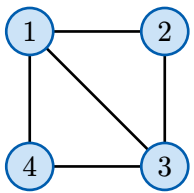
$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Properties: For undirected graphs, A is *symmetric*. The diagonal is all zeros for simple graphs.

Graph Representations: Adjacency List

Definition 12: The *adjacency list* representation stores, for each vertex v, a list of its neighbors $N(v)$.

Example:



Vertex	Neighbors
1	2, 3, 4
2	1, 3
3	1, 2, 4
4	1, 3

Space complexity: Adjacency matrix uses $O(n^2)$, adjacency list uses $O(n + m)$ where $m = |E|$.

Subgraphs

Definition 13: A graph $H = \langle V', E' \rangle$ is a *subgraph* of $G = \langle V, E \rangle$, denoted $H \subseteq G$, if

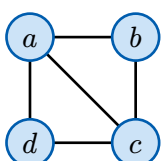
$$V' \subseteq V \quad \text{and} \quad E' \subseteq E$$

Definition 14:

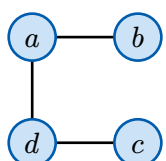
- A *spanning subgraph* includes all vertices: $V' = V$.
- An *induced subgraph* $G[S]$ on vertex set $S \subseteq V$ includes all edges between vertices in S:

$$E' = \{\{u, v\} \in E \mid u, v \in S\}$$

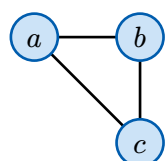
Example:



Original G



Spanning subgraph



Induced $G[\{a, b, c\}]$

Graph Isomorphism

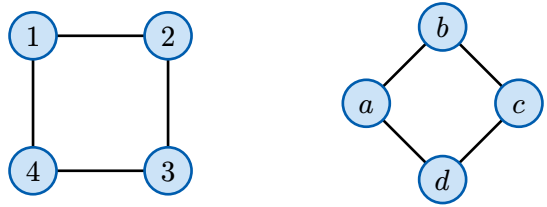
Definition 15: Graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are *isomorphic*, written $G_1 \simeq G_2$, if there exists a bijection $\varphi : V_1 \to V_2$ that *preserves adjacency*:

$$\{u, v\} \in E_1 \iff \{\varphi(u), \varphi(v)\} \in E_2$$

Intuition: Isomorphic graphs are "the same graph" with different vertex labels. They have identical structure.

Graph Isomorphism [2]

Example:



Both graphs are isomorphic to C_4. The bijection $\varphi : 1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d$ preserves adjacency.

Computational mystery: Graph isomorphism is in NP but *not known* to be NP-complete or in P.

In 2015, Babai showed it's in *quasipolynomial time* — a major breakthrough, but the exact complexity remains open.

Summary: Graph Basics

Core concepts:

- A *graph* $G = (V, E)$ is a pair of vertices and edges connecting them
- *Directed* vs *undirected*; *simple* graphs vs *multigraphs* vs *pseudographs*
- *Degree* $\deg(v)$ counts edges incident to v; Handshaking Lemma: $\sum \deg(v) = 2|E|$
- *Special graphs:* Complete K_n, cycle C_n, path P_n, bipartite $K_{m,n}$, hypercube Q_n

Coming up: Paths, connectivity, trees, bipartite graphs, matchings, Eulerian and Hamiltonian cycles, planarity, and coloring.

Graph representations:

- *Adjacency matrix:* $n \times n$ matrix, good for dense graphs, $O(n^2)$ space
- *Adjacency list:* list of neighbors per vertex, good for sparse graphs, $O(n + m)$ space

Structural concepts:

- *Subgraph:* subset of vertices/edges; *induced subgraph:* includes all edges between chosen vertices
- *Graph isomorphism:* bijection preserving adjacency — graphs are "the same" up to relabeling

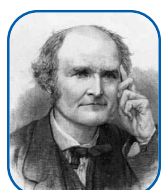
Paths and Connectivity

"All paths are not equal; if they were, they wouldn't be paths but rather the points at each end."

— H.E. Huntley


Edsger Dijkstra


Robert Tarjan


Arthur Cayley


Heinz Prüfer


Øystein Ore


Karl Menger

Walks, Trails, and Paths

Definition 16: A *walk* in a graph is an alternating sequence of vertices and edges:

$$v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$$

where each edge $e_i = \{v_{i-1}, v_i\}$.

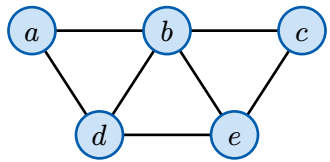
- A *trail* is a walk with *distinct edges*.
- A *path* is a walk with *distinct vertices* (hence distinct edges).

Type	Vertices repeat?	Edges repeat?	Closed version
Walk	Yes ✓	Yes ✓	Closed walk
Trail	Yes ✓	No ✗	Circuit
Path	No ✗	No ✗	Cycle

Note: A walk/trail/path is *closed* if it starts and ends at the same vertex.

Walks, Trails, and Paths: Visual Example

Example: Consider the graph:



- **Walk:** a, b, d, b, c — vertex b is repeated ✓
- **Trail:** a, d, b, e, c, b — all edges distinct, but vertex b repeats ✓
- **Path:** a, d, e, c, b — all vertices distinct ✓
- **Cycle:** b, d, e, b — closed path of length 3

Why distinguish these? The distinction is essential in algorithms:

- DFS/BFS naturally find *paths*.
- Euler's algorithm finds *trails* (visiting every edge).
- Many proofs require transforming a walk into a path.

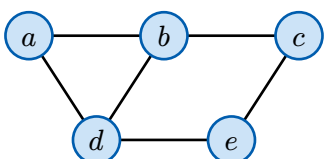
Length and Distance

Definition 17: The *length* of a walk (trail, path) is the number of edges in it.

Definition 18: The *distance* $\mathrm{dist}(u, v)$ between vertices u and v is the length of the shortest path from u to v.

If no path exists, we write $\mathrm{dist}(u, v) = \infty$.

Example:



- $\mathrm{dist}(a, b) = 1$
- $\mathrm{dist}(a, c) = 2$
- $\mathrm{dist}(a, e) = 2$
- Path a-b-c has length 2
- Trail a-d-b-c-e-d has length 5

Length and Distance [2]

Theorem 3 (Distance is a Metric): For a connected graph G, the distance function satisfies:

1. *Non-negativity:* $\text{dist}(u, v) \geq 0$, and $\text{dist}(u, v) = 0$ iff $u = v$
2. *Symmetry:* $\text{dist}(u, v) = \text{dist}(v, u)$
3. *Triangle inequality:* $\text{dist}(u, w) \leq \text{dist}(u, v) + \text{dist}(v, w)$

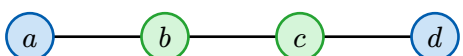
Algorithm: Breadth-first search (BFS) computes $\text{dist}(s, v)$ for all v from a source s in $O(n + m)$ time.

Eccentricity, Radius, and Diameter

Definition 19:

- *Eccentricity* of vertex v: $\mathrm{ecc}(v) = \max_{u \in V} \mathrm{dist}(v, u)$
- *Radius* of graph: $\mathrm{rad}(G) = \min_{v \in V} \mathrm{ecc}(v)$
- *Diameter* of graph: $\mathrm{diam}(G) = \max_{v \in V} \mathrm{ecc}(v)$
- *Center* of graph: $\mathrm{center}(G) = \{v \in V \mid \mathrm{ecc}(v) = \mathrm{rad}(G)\}$

Example:

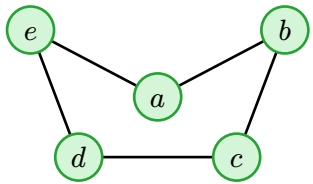


Path graph P_4:

- $\mathrm{ecc}(a) = \mathrm{ecc}(d) = 3$
- $\mathrm{ecc}(b) = \mathrm{ecc}(c) = 2$
- $\mathrm{rad}(G) = 2, \mathrm{diam}(G) = 3$
- $\mathrm{center}(G) = \{b, c\}$

Eccentricity, Radius, and Diameter [2]

Example:



Cycle graph C_5:

- $\mathrm{ecc}(v) = 2$ for all v
- $\mathrm{rad}(G) = \mathrm{diam}(G) = 2$
- All vertices are central!

Radius and Diameter: Bounds

Theorem 4: For any connected graph G: $\text{rad}(G) \leq \text{diam}(G) \leq 2 \cdot \text{rad}(G)$

Proof: Left inequality: min $\leq$ max (by definition).

Right inequality: Let u, v achieve $\text{diam}(G) = \text{dist}(u, v)$, and let $w \in \text{center}(G)$. By triangle inequality:

$$\text{diam}(G) = \text{dist}(u, v) \leq \text{dist}(u, w) + \text{dist}(w, v) \leq 2 \cdot \text{ecc}(w) = 2 \cdot \text{rad}(G)$$

□

Application: The center minimizes the *worst-case* distance to any vertex — optimal for facility placement (servers, hospitals, emergency services).

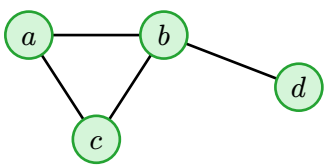
Connectivity

Definition 20: Two vertices u and v in an undirected graph G are *connected* if G contains a path from u to v. Otherwise, they are *disconnected*.

Definition 21: A graph G is *connected* if every pair of vertices in G is connected (*i.e.*, there exists a path between any two vertices).

A graph that is not connected is called *disconnected*.

Example:



Connected: path exists between every pair.

Disconnected: no path a to c.

Connectivity [2]

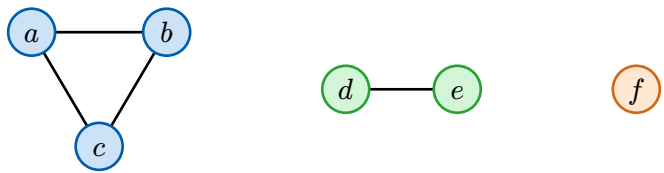
Theorem 5 (Walk implies Path): If there exists a walk from u to v, then there exists a *path* from u to v.

Proof: Take a shortest walk. If any vertex repeats, shortcut between occurrences to get a shorter walk — contradiction. □

Connected Components

Definition 22: A *connected component* of G is a maximal connected subgraph.

Example:



This graph has 3 connected components: $\{a, b, c\}$, $\{d, e\}$, and $\{f\}$.

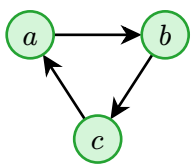
Key insight: "Being in the same connected component" is an *equivalence relation* on vertices.

Connectivity in Directed Graphs

Definition 23: A directed graph G is:

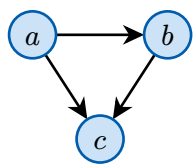
- **Weakly connected** if replacing all directed edges with undirected produces a connected graph.
- **Unilaterally connected** (or *semiconnected*) if for every pair of vertices u, v, there is a directed path from u to v *or* from v to u (or both).
- **Strongly connected** if for every pair of vertices u, v, there is a directed path from u to v *and* from v to u.

Example:



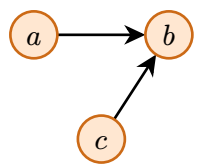
Strongly connected

$a \to b \to c \to a$



Unilaterally connected

$a \to b, a \to c, b \to c$



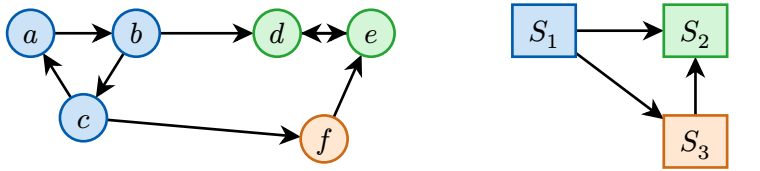
Weakly connected

No path $a \nrightarrow c$

Strongly Connected Components

Definition 24: A *strongly connected component* (SCC) of a digraph is a maximal strongly connected subgraph.

Example:



Digraph G **Condensation (DAG)**

Three SCCs: $\{a, b, c\}$, $\{d, e\}$, $\{f\}$. Contracting each gives a DAG.

Condensation: Contracting each SCC to a vertex always gives a DAG.

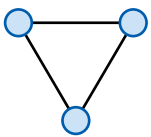
Algorithms: SCCs in $O(n + m)$ via Kosaraju's or Tarjan's algorithm.

Girth

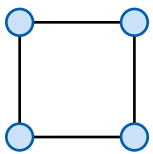
Definition 25: The *girth* of a graph G is the length of the shortest cycle in G.

If G has no cycles (is acyclic), we say $\text{girth}(G) = \infty$.

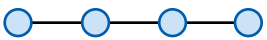
Example:



$\mathbf{girth}(K_3) = 3$



$\mathbf{girth}(C_4) = 4$



$\mathbf{girth}(P_4) = \infty$

Why girth matters: Graphs with large girth are "locally tree-like" — they behave like trees in a neighborhood of each vertex. This property is exploited in coding theory (LDPC codes) and extremal graph theory.

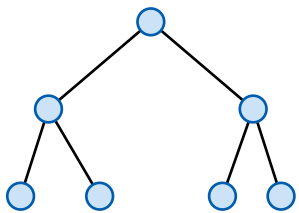
Trees and Forests

Trees: Definition

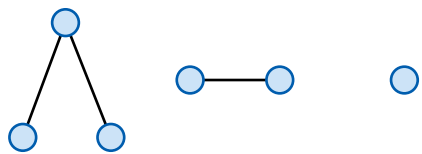
Definition 26: A *tree* is a connected acyclic graph.

A *forest* is an acyclic graph (a disjoint union of trees).

Example:



A tree



A forest (3 trees)

Characterizations of Trees

Theorem 6: For a graph G with n vertices, the following are equivalent:

1. G is a tree (connected and acyclic)
2. G is connected with exactly $n - 1$ edges
3. G is acyclic with exactly $n - 1$ edges
4. Any two vertices are connected by a *unique path*
5. G is *minimally connected*: removing any edge disconnects it
6. G is *maximally acyclic*: adding any edge creates a cycle

Trees are *ubiquitous* structures with simple recursive algorithms:

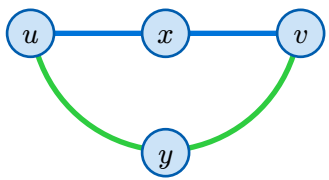
- file systems
- parse trees
- network design

Characterizations of Trees: Unique Path

Theorem 7: A connected graph is acyclic if and only if every pair of vertices is joined by a unique path.

Proof: *(⇒)* Suppose G is connected and acyclic. If there were two distinct u-v paths, their union would contain a cycle, contradiction.

(⇐) Suppose every pair of vertices has a unique path. If a cycle existed, choose distinct vertices u, v on that cycle. Traversing the cycle in opposite directions gives two distinct u-v paths, contradiction. □



Cycle gives two distinct u-v paths:

- u-x-v
- u-y-v

Therefore uniqueness fails whenever a cycle exists.

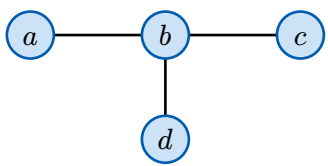
Characterizations of Trees: Maximally Acyclic

Theorem 8: Let T be a tree and let $u, v \in V(T)$ be non-adjacent. Then the graph $T + \{u, v\}$ contains exactly one cycle.

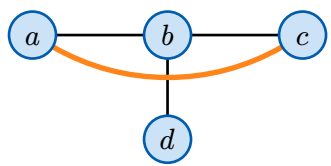
Proof: Since T is a tree, there is a unique u-v path P in T (see Theorem 7).

After adding the edge $\{u, v\}$, we obtain a cycle by going from u to v along P and returning via $\{u, v\}$.

Why is it the *only* cycle? Any cycle in $T + \{u, v\}$ must use the new edge $\{u, v\}$. Otherwise that cycle would already exist in T, contradicting acyclicity of T. Removing $\{u, v\}$ from such a cycle leaves a u-v path in T, and uniqueness of P forces that path to be exactly P. Therefore the cycle is unique. □



Before: tree T



After: $T + \{a, c\}$ has exactly one cycle a-b-c-a

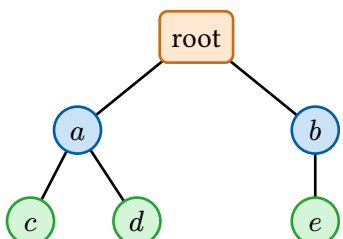
Rooted Trees

Definition 27: A *rooted tree* is a tree with one designated vertex called the *root*.

In a rooted tree:

- The *parent* of v is the neighbor of v on the path to the root
- The *children* of v are the other neighbors of v
- A *leaf* is a vertex with no children
- An *internal vertex* has at least one child

Example:



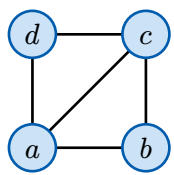
- Root has children a, b
- Leaves: c, d, e
- Internal vertices: root, a, b

Spanning Trees

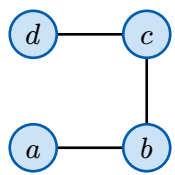
Definition 28: A *spanning tree* of a connected graph G is a spanning subgraph that is a tree.

Theorem 9: Every connected graph has at least one spanning tree.

Example:



Original graph



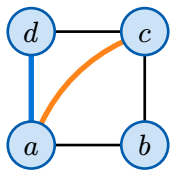
A spanning tree

Application: Finding minimum spanning trees (MST) is fundamental in network design — Kruskal's and Prim's algorithms solve this in $O(m \log n)$.

Fundamental Cycles

Definition 29: Let T be a spanning tree of G. For each edge $e \in E(G) \setminus E(T)$ (a *non-tree edge*), the graph $T + e$ contains exactly one cycle, called the *fundamental cycle* of e with respect to T.

Example:



Spanning tree: a-b-c-d (black edges).

- Adding $\{a, d\}$ creates fundamental cycle a-b-c-d-a.
- Adding $\{a, c\}$ creates fundamental cycle a-b-c-a.

Key insight: Each non-tree edge generates exactly one cycle. For a graph with n vertices and m edges: $m - n + 1$ fundamental cycles (the *cycle rank*).

Cayley's Formula

Theorem 10 (Cayley's Formula): The number of *labeled* trees on n vertices is exactly n^{n-2}.

Example:

- $n = 2$: $2^0 = 1$ tree (just one edge)
- $n = 3$: $3^1 = 3$ trees (three ways to pick the center)
- $n = 4$: $4^2 = 16$ trees
- $n = 5$: $5^3 = 125$ trees

Historical note: Cayley proved this in 1889. The formula counts *labeled* trees, *i.e.*, trees on the vertex set $\{1, 2, \ldots, n\}$. The number of *unlabeled* (non-isomorphic) trees grows much slower.

Why n^{n-2}? Prüfer sequences give a bijection between labeled trees on $[n]$ and sequences in $[n]^{n-2}$.

Prüfer Sequences

Definition 30: A *Prüfer sequence* is a unique encoding of a labeled tree on n vertices as a sequence of $n-2$ labels.

Encoding algorithm:

1. Find the leaf with the smallest label
2. Add its neighbor's label to the sequence
3. Remove the leaf from the tree
4. Repeat until 2 vertices remain

Example: Tree: 1-3-4-2, 3-5

- Encoding: Remove 1 (neighbor 3), remove 2 (neighbor 4), remove 5 (neighbor 3).
- Prüfer sequence: $(3, 4, 3)$

Observation: A vertex appears in the Prüfer sequence exactly $\deg(v) - 1$ times. In particular, *leaves never appear* in the sequence (they have degree 1).

Connectivity Theory

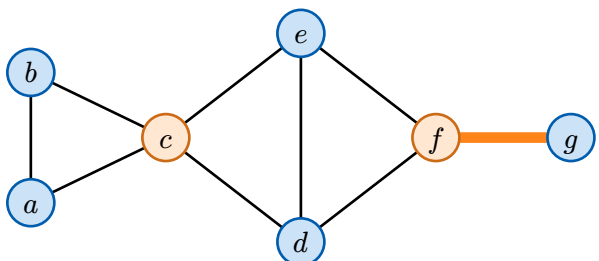
"The question is not whether a graph is connected,
but how well *it is connected."*

Cut Vertices and Bridges

Definition 31: A *cut vertex* (or *articulation point*) is a vertex whose removal increases the number of connected components.

Definition 32: A *bridge* (or *cut edge*) is an edge whose removal increases the number of connected components.

Example:



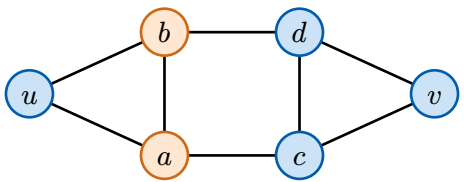
- Cut vertices: c, f
- Bridge: edge $\{f, g\}$

Separators and Cuts

Definition 33: For vertices $u, v \in V$, a *u-v separator* (or *u-v vertex cut*) is a set $S \subseteq V \setminus \{u, v\}$ such that u and v are in different components of $G - S$.

Definition 34: A *u-v edge cut* is a set $F \subseteq E$ such that u and v are in different components of $G - F$.

Example:



$S = \{a, b\}$ is a minimum u-v separator (size 2). $S' = \{c, d\}$ is another minimum separator (same size).

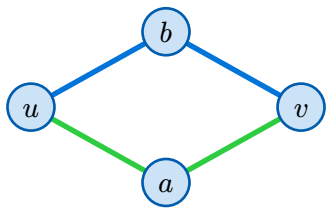
Note: A *minimum* u-v separator has the smallest possible cardinality. Its size is the local vertex-connectivity between u and v.

Internally Disjoint Paths

Definition 35: Two u-v paths are *internally vertex-disjoint* if they share no vertices other than u and v.

Two u-v paths are *edge-disjoint* if they share no edges.

Example:



Path 1 (u-a-v) and Path 2 (u-b-v) are both *internally vertex-disjoint* and *edge-disjoint*.

Key duality: Separators *block* paths; disjoint paths *survive* removal of a few vertices/edges.

Menger's theorem quantifies this duality precisely.

Vertex and Edge Connectivity

Definition 36: The *vertex connectivity* $\kappa(G)$ is the minimum size of a vertex set whose removal disconnects G (or reduces it to one vertex).

Definition 37: The *edge connectivity* $\lambda(G)$ is the minimum size of an edge set whose removal disconnects G.

Example:

Graph	κ	λ	δ
K_5	4	4	4
C_6	2	2	2
Tree	1	1	1
Petersen	3	3	3

For K_n, we define $\kappa(K_n) = n - 1$ (need to remove all but one vertex).

k-Connectivity

Definition 38: A graph G is ***k*-vertex-connected** (or simply *k-connected*) if $\kappa(G) \geq k$.

Equivalently: G has more than k vertices, and $G - S$ is connected for every set S with $|S| < k$.

Definition 39: A graph G is ***k*-edge-connected** if $\lambda(G) \geq k$.

Equivalently: $G - F$ is connected for every edge set F with $|F| < k$.

Example:

- K_n is $(n-1)$-connected (both vertex and edge).
- C_n (cycle) is 2-connected and 2-edge-connected.
- A tree with $n \geq 2$ vertices has $\kappa = \lambda = 1$ (every edge is a bridge).

Fault tolerance: A k-connected network survives any $k - 1$ node failures.

Whitney's Inequality

Theorem 11 (Whitney's Inequality): For any graph G:

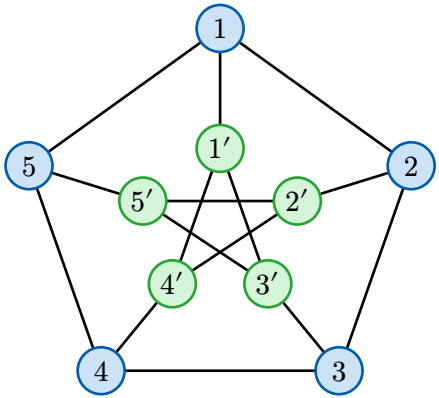
$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

where $\delta(G)$ is the minimum degree.

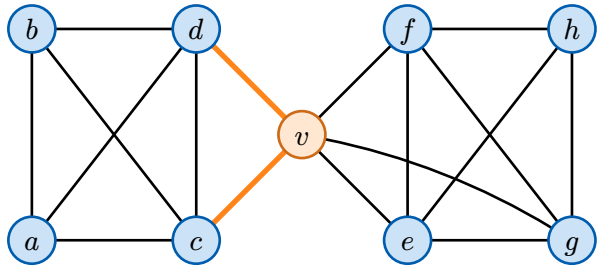
Proof:

- $\lambda(G) \leq \delta(G)$: Let v be a vertex of minimum degree. The set of all edges incident to v is an edge cut of size $\delta(G)$ (removing them isolates v).
- $\kappa(G) \leq \lambda(G)$: Let F be a minimum edge cut separating G into parts A and B. We construct a vertex separator S of size $\leq |F|$: for each edge $e = \{a, b\}$ in F with $a \in A$ and $b \in B$, include b in S (choosing the endpoint on one fixed side). Then S separates any remaining vertex in A from any remaining vertex in B, and $|S| \leq |F|$. □

Whitney's Inequality [2]



Petersen graph:
$\kappa = \lambda = \delta = 3$



Strict inequality example:
$\kappa = 1, \lambda = 2, \delta = 3$

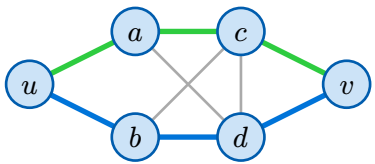
Menger's Theorem (Vertex Form)

Theorem 12 (Menger): Let u, v be non-adjacent vertices in G. Then:

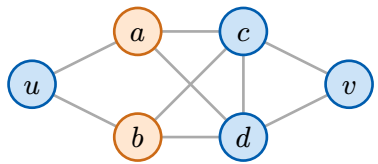
$$\max\{\text{number of internally vertex-disjoint } u\text{-}v \text{ paths}\} = \min\{|S| : S \text{ is a } u\text{-}v \text{ separator}\}$$

In other words: the maximum number of paths from u to v that share *no internal vertices* equals the minimum number of vertices needed to separate u from v.

Example:



2 internally vertex-disjoint paths (max)

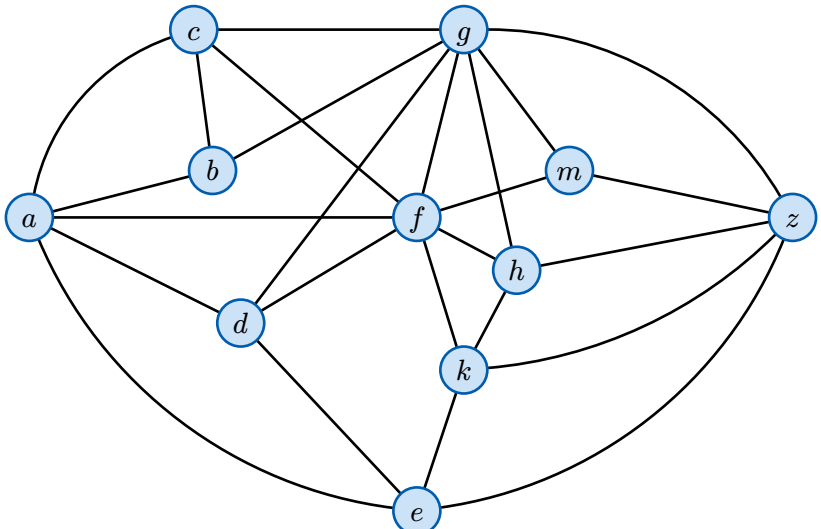


Min vertex separator: $\{a, b\}$ of size 2

Path 1 (u-a-c-v) and Path 2 (u-b-d-v) are *internally vertex-disjoint*.

Deleting the orange vertices a and b disconnects u from v, so a *minimum vertex separator* has size 2.

Menger's Theorem Visualized

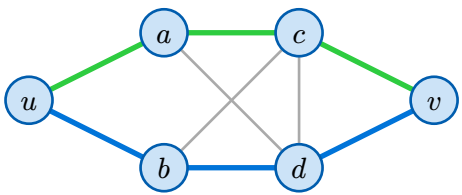


Menger's Theorem (Edge Form)

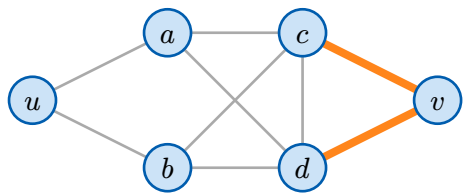
Theorem 13 (Menger): For any distinct vertices u, v in G:

$$\max\{\text{number of edge-disjoint } u\text{-}v \text{ paths}\} = \min\{|F| : F \text{ is a } u\text{-}v \text{ edge cut}\}$$

Example:



2 edge-disjoint paths (max)



Min edge cut: $\{(c, v), (d, v)\}$ of size 2

Highlighted paths (*u-a-c-v* and *u-b-d-v*) are *edge-disjoint*.

In the same graph, removing (c, v) and (d, v) disconnects u from v, so a *minimum edge cut* has size 2.

Menger's Theorem: Significance

Max-Flow Min-Cut duality: Menger's theorem (1927) is equivalent to the Max-Flow Min-Cut theorem (Ford–Fulkerson, 1956) with unit capacities.

Why this matters:

- *Network reliability:* Independent routes between nodes
- *Routing:* Parallel data streams
- *VLSI:* Non-overlapping wire paths
- *Algorithmic foundation:* Connectivity decomposition

Menger's Theorem: Corollaries

Theorem 14 (Global Vertex Connectivity): A graph G is k-connected if and only if every pair of distinct vertices is connected by at least k internally vertex-disjoint paths.

Theorem 15 (Global Edge Connectivity): A graph G is k-edge-connected if and only if every pair of distinct vertices is connected by at least k edge-disjoint paths.

Example: The cycle C_5 is 2-connected. For any two vertices, there are exactly 2 internally vertex-disjoint paths (the two arcs of the cycle). Removing any single vertex leaves a path — still connected.

Example: K_4 is 3-connected. Between any two vertices, there are 3 internally disjoint paths (each through one of the three remaining vertices). Removing any 2 vertices leaves at least one edge — still connected.

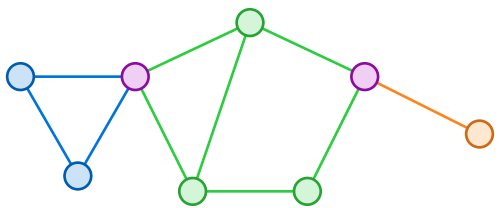
Intuition: High connectivity = many independent routes. Menger explains Whitney's inequality: $\kappa(G) \leq \lambda(G) \leq \delta(G)$.

Blocks (2-Connected Components)

Definition 40: A *block* of a graph G is a maximal connected subgraph with no cut vertex (*i.e.*, maximal 2-connected subgraph, or a bridge, or an isolated vertex).

Note: Every edge belongs to exactly one block. Blocks share at most one vertex (a cut vertex).

Example:



Three blocks: blue triangle, green pentagon, orange bridge. Purple cut vertices.

Whitney's Characterization of 2-Connectivity

Theorem 16 (Whitney, 1932): A graph G with at least 3 vertices is 2-connected if and only if every pair of vertices lies on a common cycle.

Proof: ($\Rightarrow$) If G is 2-connected, Menger gives 2 internally disjoint u-v paths forming a cycle.

($\Leftarrow$) If every pair lies on a cycle, then removing any vertex v leaves at least one path between any u, w (via the cycle not passing through v). Hence no cut vertex exists. □

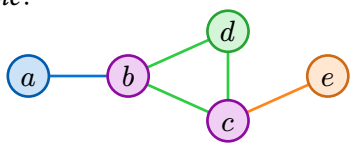
Insight: Blocks are the "robust cores" — any two vertices in a block lie on a common cycle.

Block-Cut Tree

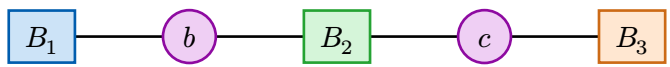
Definition 41: The *block-cut tree* (or *BC-tree*) of a connected graph G is a bipartite tree T where:

- One part contains a node for each *block* of G.
- The other part contains a node for each *cut vertex* of G.
- A block-node B is adjacent to a cut-vertex-node v iff $v \in B$.

Example:



Graph G



Block-Cut Tree

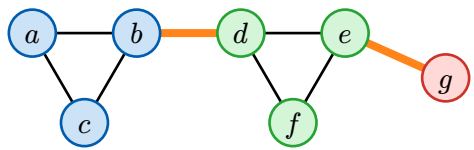
Applications: Dynamic programming on the block-cut tree solves many problems on general graphs.

Islands (2-Edge-Connected Components)

Definition 42: An *island* (or *2-edge-connected component*) is a maximal connected subgraph with no bridges.

Equivalently: vertices u and v are in the same island iff they lie on a common circuit.

Example:

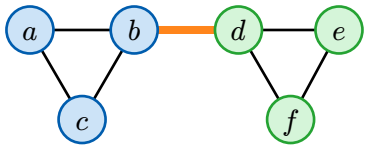


Three islands: $\{a, b, c\}$, $\{d, e, f\}$, $\{g\}$. The orange edges are bridges.

Note: Islands partition vertices (no sharing). Separated by bridges.

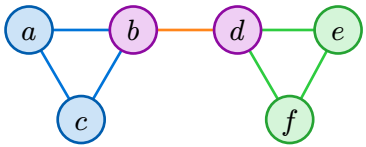
Blocks vs Islands

Example:



Islands

Blue and green are 2-edge-connected components.
Orange = bridge.



Blocks

Blue triangle, green triangle, orange bridge.
Purple = cut vertices b and d.

Key difference:

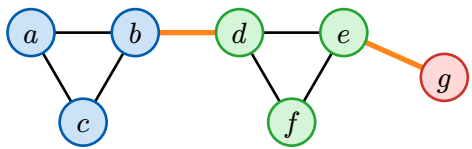
- **Blocks** = 2-vertex-connectivity: no cut vertices within a block.
- **Islands** = 2-edge-connectivity: no bridges within an island.

Blocks may share cut vertices; islands partition vertices.

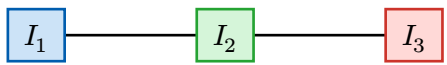
Bridge Tree

Definition 43: The *bridge tree* (or *island tree*) of a connected graph G is obtained by contracting each island to a single vertex. The edges of this tree are exactly the bridges of G.

Example:



Graph G



Bridge Tree

- Block-cut tree: decomposition by *cut vertices* into *blocks*.
- Bridge tree: decomposition by *bridges* into *islands*.

Theorem 17: A graph is 2-edge-connected iff its bridge tree is a single vertex (no bridges).

A graph is 2-vertex-connected iff its block-cut tree has a single block node.

Summary: Paths, Trees, and Connectivity

Paths & Distance

- Walk $\longrightarrow$ Trail $\longrightarrow$ Path
- $\text{dist}(u, v)$: shortest path (metric)
- Eccentricity, radius, diameter, center

Trees

- Connected + acyclic
- n vertices, $n - 1$ edges
- Spanning trees
- Fundamental cycles: $m - n + 1$

Connectivity

- Cut vertices, bridges
- $\kappa(G) \leq \lambda(G) \leq \delta(G)$
- **Menger:** max disjoint paths = min separator
- Blocks: 2-connected components
- Islands: 2-edge-connected

The big picture: Menger's theorem reveals the fundamental duality between *independent routes* and *bottlenecks*. This forms the foundation for network flow theory.

Eulerian and Hamiltonian Graphs

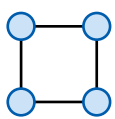
Eulerian Paths and Circuits

Definition 44:
- An *Eulerian trail* is a trail that visits every edge exactly once.
- An *Eulerian circuit* is a closed Eulerian trail.
- A graph is *Eulerian* if it has an Eulerian circuit.

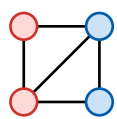
Theorem 18 (Euler's Theorem): A connected graph has an Eulerian circuit if and only if every vertex has *even degree*.

A connected graph has an Eulerian trail (but not circuit) if and only if exactly *two vertices* have odd degree.

Example:



Eulerian
(all degrees even)



Has Eulerian trail
(2 odd vertices)

Hamiltonian Paths and Cycles

Definition 45:

- A *Hamiltonian path* visits every vertex exactly once.
- A *Hamiltonian cycle* is a cycle that visits every vertex exactly once.
- A graph is *Hamiltonian* if it has a Hamiltonian cycle.

Warning: Unlike Eulerian graphs, there is *no simple characterization* of Hamiltonian graphs!

Determining if a graph is Hamiltonian is NP-complete.

Sufficient Conditions for Hamiltonicity

Theorem 19 (Ore's Theorem): If G has $n \geq 3$ vertices and for every pair of non-adjacent vertices u, v:

$$\deg(u) + \deg(v) \geq n$$

then G is Hamiltonian.

Theorem 20 (Dirac's Theorem): If G has $n \geq 3$ vertices and $\delta(G) \geq \frac{n}{2}$, then G is Hamiltonian.

Eulerian vs Hamiltonian: Summary

	Eulerian	**Hamiltonian**
Visits	Every *edge* once	Every *vertex* once
Characterization	Degree condition	NP-complete to decide
Algorithm	$O(m)$ — Hierholzer's	Exponential (backtracking)
Named after	Euler (1736)	Hamilton (1857)

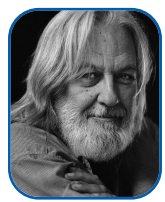
Historical note: Hamilton sold a puzzle ("Icosian game") based on finding Hamiltonian cycles on a dodecahedron graph.

The dodecahedral graph has exactly 30 distinct Hamiltonian cycles.

Bipartite Graphs and Matchings


Dénes Kőnig


Philip Hall


Jack Edmonds


Harold Kuhn

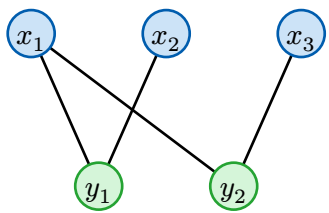

Leonid Mirsky

Definition of Bipartite Graphs

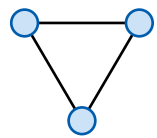
Definition 46: A graph $G = \langle V, E \rangle$ is *bipartite* if its vertices can be partitioned into two disjoint sets $V = X \sqcup Y$ such that every edge connects a vertex in X to a vertex in Y.

We write $G = \langle X \cup Y, E \rangle$ or $G = (X, Y, E)$.

Example:



Bipartite



Not bipartite
(contains triangle)

Characterization of Bipartite Graphs

Theorem 21: A graph is bipartite if and only if it contains no odd-length cycles.

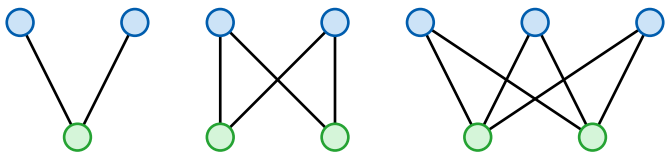
Proof *(sketch)*: *(⇒)* If $G = \langle X \sqcup Y, E \rangle$ is bipartite, every edge goes from X to Y. Hence, along any cycle, vertices must alternate between the two parts. Therefore every cycle has even length.

(⇐) Assume G has no odd cycle. In each connected component, pick a root r and partition the vertices by *parity* of distance from r: $X = \{v \mid \mathrm{dist}(r, v) \text{ is even}\}$ and $Y = \{v \mid \mathrm{dist}(r, v) \text{ is odd}\}$. Suppose an edge joined two vertices of the same parity. Together with shortest root-paths, this gives a closed walk of odd length, so G contains an odd cycle — contradiction. So all edges go between X and Y, and G is bipartite. □

Complete Bipartite Graphs

Definition 47: The *complete bipartite graph* $K_{m,n}$ has parts of sizes m and n, with every vertex in one part adjacent to every vertex in the other.

Example:



$K_{2,1}$ $K_{2,2}$ $K_{3,2}$

Note: $K_{m,n}$ has $m+n$ vertices and $m \cdot n$ edges.

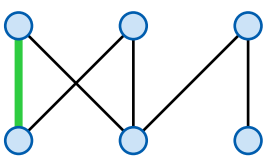
Matchings

Definition 48: A *matching* $M \subseteq E$ is a set of pairwise non-adjacent edges (no two edges share a vertex).

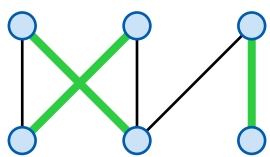
Definition 49:

- A matching is *maximal* if no edge can be added to it.
- A matching is *maximum* if it has the largest possible size.
- A *perfect matching* covers all vertices.

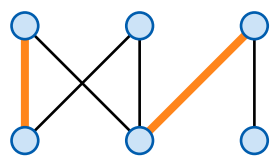
Example:



Matching
(not maximal)



Maximum
(perfect)



Maximal
(not maximum)

Hall's Marriage Theorem

Definition 50: Let $G = \langle X \cup Y, E \rangle$ be a bipartite graph. For a subset $S \subseteq X$, define its *neighborhood*:

$$N(S) = \{y \in Y \mid \exists x \in S : \{x, y\} \in E\}$$

Theorem 22 (Hall's Marriage Theorem (Hall, 1935)): A bipartite graph $G = \langle X \cup Y, E \rangle$ has a matching that *saturates* X (*i.e.*, every vertex in X is matched) if and only if:

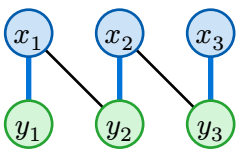
$$\forall S \subseteq X : |N(S)| \geq |S|$$

This is called **Hall's condition** or the *marriage condition*.

Examples: Hall's Condition

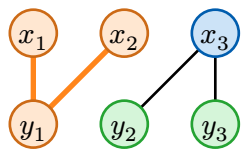
How to read Hall's condition: For every subset $S \subseteq X$, compare the demand $|S|$ with the available partners $|N(S)|$. A matching saturating X exists exactly when every group S has *enough* distinct choices: $|N(S)| \geq |S|$.

Example:



Hall holds

Matching: $\{x_1y_1, x_2y_2, x_3y_3\}$.
So X is saturated.



Hall fails

Here $|X| = |Y| = 3$, but for $S = \{x_1, x_2\}$ we have $N(S) = \{y_1\}$.
Hence $|N(S)| = 1 < 2 = |S|$, so no matching can saturate X.

Proof of Hall's Theorem

We prove both directions.

Direction ($\Rightarrow$): If a matching saturating X exists, then Hall's condition holds.

Proof: Let M be a matching that saturates X. For any $S \subseteq X$:

- Each vertex in S is matched to a distinct vertex in Y (by definition of matching).
- Let $M(S)$ be the set of partners of S under M. Then $|M(S)| = |S|$.
- Since every partner is a neighbor in G, $M(S) \subseteq N(S)$.
- Therefore: $|N(S)| \geq |M(S)| = |S|$. □

Direction ($\Leftarrow$): If Hall's condition holds, then a matching saturating X exists.

This is the interesting direction. We use **strong induction** on $n = |X|$.

Proof (Sufficiency): Base & Strategy

Base Case ($n = 1$): If $X = \{x\}$, Hall's condition gives $|N(\{x\})| \geq 1$, so x has a neighbor y. The edge $\{x, y\}$ is a matching.

Inductive Hypothesis: Assume the theorem holds for all bipartite graphs with $|X| < n$.

Inductive Step: Consider G with $|X| = n \geq 2$. We split into two cases:

- **Case 1:** Every proper subset S has *surplus* neighbors: $|N(S)| \geq |S| + 1$.
- **Case 2:** Some proper subset S is *tight*: $|N(S)| = |S|$.

Proof: Case 1 (Surplus)

Case 1: For all $\emptyset \neq S \subsetneq X$, we have $|N(S)| \geq |S| + 1$.

Strategy: Match an arbitrary edge, then use induction on the smaller graph.

1. Pick any $x \in X$ and choose any neighbor $y \in N(\{x\})$ (such y exists since $|N(\{x\})| \geq |\{x\}| = 1$).
2. Remove both endpoints: let $G' = G - \{x, y\}$ and $X' = X \setminus x$.
3. **Verify Hall's condition in G':** Let $S' \subseteq X'$ be arbitrary.
   - If $S' = \emptyset$, the inequality is trivial.
   - If $S' \neq \emptyset$, then $S' \subsetneq X$, so in G we have $|N_G(S')| \geq |S'| + 1$.
   - Removing y from Y reduces $|N_G(S')|$ by at most 1.
   - So $|N_{G'}(S')| \geq |N_G(S')| - 1 \geq (|S'| + 1) - 1 = |S'|$.
4. By induction, G' has a matching M' saturating X'.
5. Then $M = M' \cup \{\{x, y\}\}$ saturates X. $\square$

Proof: Case 2 (Tight Subset)

Case 2: There exists $\emptyset \neq S_0 \subsetneq X$ such that $|N(S_0)| = |S_0|$.

Strategy: Match S_0 independently, then match the rest.

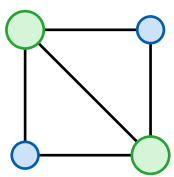
1. **Match S_0:** The induced subgraph $G[S_0 \cup N(S_0)]$ satisfies Hall's condition (inherited from G). Since $|S_0| < n$, by induction there exists a matching M_1 saturating S_0.
2. **Match the remainder:** Let $G' = G - S_0 - N(S_0)$ and $X' = X \setminus S_0$.
   We verify Hall's condition for G'. Let $A \subseteq X'$ be arbitrary.
   - In G: $|N(A \cup S_0)| \geq |A \cup S_0| = |A| + |S_0|$ (Hall's condition).
   - Also $N(A \cup S_0) = N(A) \cup N(S_0)$.
   - Therefore, $|N(A) \setminus N(S_0)| = |N(A) \cup N(S_0)| - |N(S_0)| \geq |A| + |S_0| - |S_0| = |A|$.
   - But $N_{G'}(A) = N_G(A) \setminus N_G(S_0)$, so $|N_{G'}(A)| \geq |A|$.
3. By induction, G' has a matching M_2 saturating X'.
4. Then $M = M_1 \cup M_2$ saturates X. □

Vertex and Edge Covers

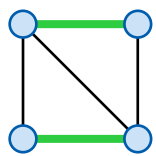
Definition 51: A *vertex cover* $R \subseteq V$ is a set of vertices such that every edge has at least one endpoint in R.

Definition 52: An *edge cover* $F \subseteq E$ is a set of edges such that every vertex is incident to at least one edge in F.

Example:



Vertex cover $\{a, c\}$



Edge cover $\{\{a, b\}, \{c, d\}\}$

König's Theorem

Theorem 23 (König's Theorem): In a bipartite graph:

$$\nu(G) = \tau(G)$$

where $\nu(G)$ is the size of a *maximum matching* and $\tau(G)$ is the size of a *minimum vertex cover*.

Key insight: This equality does *not* hold for general graphs! In a triangle K_3: $\nu = 1$ but $\tau = 2$.

König's theorem follows from the LP duality of matching and vertex cover.

It also follows from the Max-Flow Min-Cut theorem on the associated network.

König's Theorem [2]

Theorem 24: In any graph without isolated vertices:

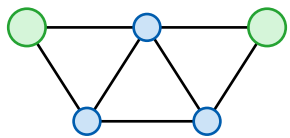
$$|\text{minimum vertex cover}| + |\text{maximum stable set}| = |V|$$

$$|\text{minimum edge cover}| + |\text{maximum matching}| = |V|$$

Stable Sets (Independent Sets)

Definition 53: A *stable set* (or *independent set*) $S \subseteq V$ is a set of pairwise non-adjacent vertices.

Example:



The green vertices $\{a, c\}$ form a stable set — no edges between them.

Complement relationship: S is a stable set in $G \iff S$ is a clique in $\overline{G}$.

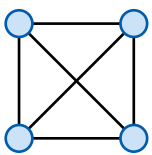
Planar Graphs

Planar Graphs: Definition

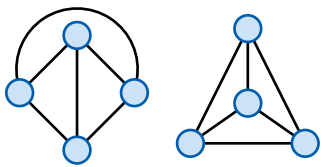
Definition 54: A graph is *planar* if it can be drawn in the plane without edge crossings.

A *plane graph* is a specific planar embedding (drawing) of a planar graph.

Example:



K_4 with crossings



K_4 planar embeddings

K_4 is planar — it can be redrawn without crossings.

Faces and Euler's Formula

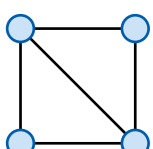
Definition 55: A *face* of a plane graph is a connected region bounded by edges. The unbounded region is the *outer face* (or *infinite face*).

Theorem 25 (Euler's Polyhedron Formula): For any connected plane graph with n vertices, m edges, and f faces:

$$n - m + f = 2$$

Deep insight: The quantity $n - m + f$ is called the *Euler characteristic*. It equals 2 for any surface homeomorphic to a sphere. For a torus, it equals 0. This connects graph theory to topology!

Example:



- Vertices: $n = 4$
- Edges: $m = 5$
- Faces: $f = 3$ (2 inner + 1 outer)

Check: $4 - 5 + 3 = 2$ ✓

Consequences of Euler's Formula

Theorem 26: For any simple planar graph with $n \geq 3$ vertices and m edges:

$$m \leq 3n - 6$$

Proof: Each face has ≥ 3 edges on its boundary, and each edge borders at most 2 faces. So $3f \leq 2m$, giving $f \leq \frac{2m}{3}$.

By Euler's formula: $2 = n - m + f \leq n - m + \frac{2m}{3} = n - \frac{m}{3}$. Therefore $m \leq 3n - 6$. □

Theorem 27: For any simple planar *bipartite* graph with $n \geq 3$ vertices:

$$m \leq 2n - 4$$

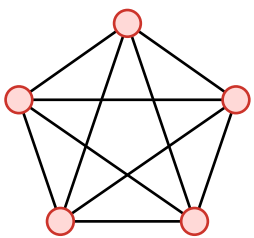
Corollary: K_5 (with 10 edges but $3 \cdot 5 - 6 = 9$) and $K_{3,3}$ (with 9 edges but $2 \cdot 6 - 4 = 8$) are *not* planar.

Kuratowski's Theorem

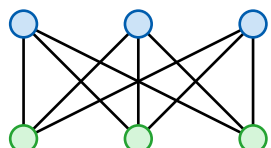
Theorem 28 (Kuratowski's Theorem): A graph is planar if and only if it contains no subdivision of K_5 or $K_{3,3}$ as a subgraph.

Definition 56: A *subdivision* of a graph G is obtained by replacing edges with paths.

Example:



K_5 (not planar)



$K_{3,3}$ (not planar)

Graph Coloring

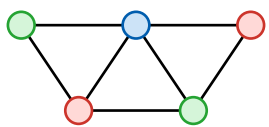
Vertex Coloring

Definition 57: A *(proper) vertex coloring* of a graph assigns colors to vertices such that adjacent vertices receive different colors.

Definition 58: A graph is *k-colorable* if it has a proper coloring using at most k colors.

The *chromatic number* $\chi(G)$ is the minimum k such that G is k-colorable.

Example:



This graph is 3-colorable. Is $\chi(G) = 3$?

Chromatic Number: Bounds

Theorem 29: For any graph G:

$$\omega(G) \leq \chi(G) \leq \Delta(G) + 1$$

where $\omega(G)$ is the *clique number* and $\Delta(G)$ is the maximum degree.

Proof *(Lower bound)*: A clique of size k needs k different colors. □

Theorem 30 (Brooks' Theorem): For any connected graph G that is not a complete graph or an odd cycle:

$$\chi(G) \leq \Delta(G)$$

Computing $\chi(G)$ is NP-hard, but checking 2-colorability is $\mathcal{O}(n + m)$.

The Four Color Theorem

Theorem 31 (Four Color Theorem): Every planar graph is 4-colorable: $\chi(G) \leq 4$ for all planar G.

A controversial proof:

- Conjectured in 1852 by Francis Guthrie
- Proved in 1976 by Appel and Haken *using a computer*
- First major theorem requiring computational verification
- Checked ~1,500 "unavoidable" configurations
- Sparked debates: Is a computer-assisted proof a "real" proof?

The dual view: Coloring vertices of a planar graph = coloring regions of a map so adjacent regions differ. Every map needs at most 4 colors!

Edge Coloring

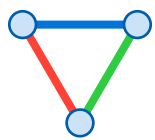
Definition 59: An *edge coloring* assigns colors to edges such that edges sharing a vertex receive different colors.

The *chromatic index* $\chi'(G)$ is the minimum number of colors needed.

Theorem 32 (Vizing's Theorem): For any simple graph G:

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$$

Example:



Triangle K_3 needs 3 colors: $\chi'(K_3) = 3 = \Delta + 1$.

Cliques and Stable Sets

Cliques

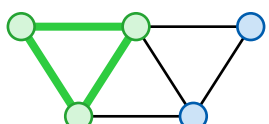
Definition 60: A *clique* is a subset of vertices $Q \subseteq V$ such that every pair of vertices in Q is adjacent.

Equivalently, Q induces a complete subgraph.

Definition 61:

- *Clique number* $\omega(G)$: size of the largest clique
- A clique is *maximal* if no vertex can be added
- A clique is *maximum* if it has the largest possible size

Example:



Maximum clique $\{a, b, c\}$ shown in green. $\omega(G) = 3$.

Ramsey Theory: A Taste

Theorem 33 (Ramsey's Theorem): For any positive integers r and s, there exists a number $R(r, s)$ such that any 2-coloring of the edges of K_n (with $n \geq R(r, s)$) contains either a red K_r or a blue K_s.

Example: $R(3, 3) = 6$: Among any 6 people, there are either 3 mutual friends or 3 mutual strangers.

Warning: Computing Ramsey numbers is extremely hard. We know $R(3, 3) = 6$, $R(4, 4) = 18$, but $R(5, 5)$ is unknown!

Famous quote by Erdős: "Suppose aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find $R(5, 5)$. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If they digit $R(6, 6)$, we would have no choice but to launch a preemptive attack."

Network Flows

"The whole is more than the sum of its parts —
but the flow is limited by the narrowest channel."

Motivation: Moving Things Through Networks

Many *real-world* problems ask: *"how much can move from A to B through a network?"*

Water networks
Pipes with limited diameter connect a reservoir to a city.

How much water per second can flow?

Internet routing
Routers connected by links with limited bandwidth.

How much data can be transferred?

Supply chains
Factories, warehouses, trucks with limited capacity.

How many goods can be shipped?

The abstraction: All three are instances of the same mathematical structure — a *flow network*. Once we solve the abstract problem, all concrete instances are solved.

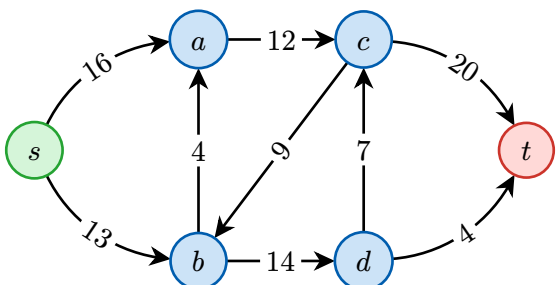
What we'll prove: The maximum amount of flow you can push equals the capacity of the "tightest bottleneck" — made precise by the **Max-Flow Min-Cut Theorem**.

Flow Network

Definition 62: A *flow network* is a tuple $N = \langle V, E, s, t, c \rangle$ where:

- $G = \langle V, E \rangle$ is a directed graph,
- $s \in V$ is the *source* and $t \in V$ is the *sink* (with $s \neq t$),
- $c : E \to \mathbb{R}_{\geq 0}$ is the *capacity* function assigning a non-negative real number to each edge.

We extend c to all pairs: $c(u, v) = 0$ whenever $(u, v) \notin E$.



Note: We assume every vertex lies on some s-t path (no isolated or useless vertices).

Flow

Definition 63: A *flow* in a network N is a function $f : E \to \mathbb{R}_{\geq 0}$ satisfying:

1. **Capacity constraint:** $0 \leq f(e) \leq c(e)$ for every edge $e \in E$.
2. **Flow conservation:** For every internal vertex $v \in V \setminus \{s, t\}$:

$$\underbrace{\sum_{e \in \text{ in}(v)} f(e)}_{\text{flow into } v} = \underbrace{\sum_{e \in \text{ out}(v)} f(e)}_{\text{flow out of } v}$$

where $\text{in}(v)$ and $\text{out}(v)$ denote the sets of incoming and outgoing edges of v.

Flow conservation embodies *"what goes in — must come out"*.

- No *internal* vertex stores or creates flow.
- Only s *produces* flow and t *absorbs* it.

Flow Value

Definition 64: The *value* of a flow f, denoted $|f|$, is the net flow out of the source:

$$|f| = \underbrace{\sum_{e \in \text{ out}(s)} f(e)}_{\text{out of } s} - \underbrace{\sum_{e \in \text{ in}(s)} f(e)}_{\text{into } s \ (= 0 \text{ if no back-edges})}$$

Definition 65 (Maximum Flow Problem): Given a flow network N, find a flow f that *maximizes* $|f|$.

Flow Conservation Theorem

Theorem 34: For any feasible flow f, the net flow out of s equals the net flow into t:

$$|f| = \sum_{e \in \text{ in}(t)} f(e) - \sum_{e \in \text{ out}(t)} f(e)$$

Proof: First, define the *net flow* at a vertex v:

$$\text{net}(v) = \sum_{e \in \text{ out}(v)} f(e) - \sum_{e \in \text{ in}(v)} f(e)$$

Summing over all vertices gives zero:

$$\sum_{v \in V} \text{net}(v) = \sum_{v \in V} \sum_{e \in \text{ out}(v)} f(e) - \sum_{v \in V} \sum_{e \in \text{ in}(v)} f(e) = 0,$$

because each edge $e = (u, w)$ appears once with $+f(e)$ (as outgoing from u) and once with $-f(e)$ (as incoming to w).

Flow Conservation Theorem [2]

For every internal vertex $v \in V \setminus \{s, t\}$, flow conservation implies $\text{net}(v) = 0$, hence

$$\sum_{v \in V} \text{net}(v) = \text{net}(s) + \text{net}(t).$$

Therefore $\text{net}(s) + \text{net}(t) = 0$, *i.e.*

$$\sum_{e \in \text{ out}(s)} f(e) - \sum_{e \in \text{ in}(s)} f(e) = \sum_{e \in \text{ in}(t)} f(e) - \sum_{e \in \text{ out}(t)} f(e).$$

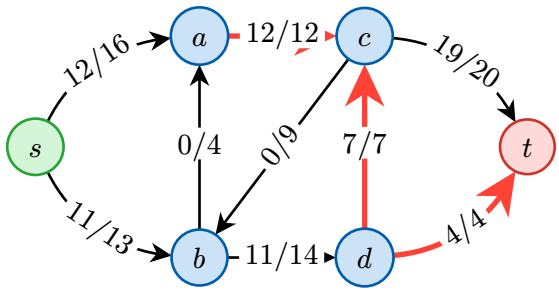
The left side is $|f|$ by definition, so

$$|f| = \sum_{e \in \text{ in}(t)} f(e) - \sum_{e \in \text{ out}(t)} f(e).$$

□

A Feasible Flow: Example

Each edge is labelled f/c (flow / capacity). Edges carrying strictly less than capacity are shown normally; *saturated* edges ($f = c$) are red.



Flow value: $|f| = 12 + 11 = 23$.

This flow turns out to be *maximum* — we will prove this shortly using the Min-Cut Theorem.

Cuts

Definition 66: Let $N = (V, E, s, t, c)$ be a flow network.

An *s-t cut* is an ordered pair (A, B) such that

$$A \cup B = V, \quad A \cap B = \emptyset, \quad s \in A, \quad t \in B.$$

Definition 67: For a cut (A, B), define the directed boundary edge sets:

$$\delta^+(A) := \{(u, v) \in E \mid u \in A, v \in B\},$$
$$\delta^-(A) := \{(u, v) \in E \mid u \in B, v \in A\}.$$

Cut Capacity

Definition 68: The *capacity* of an s-t cut (A, B) is

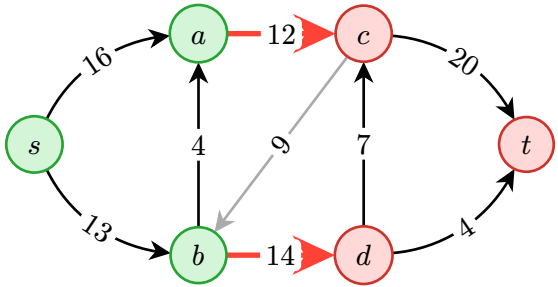
$$c(A, B) := \sum_{(u,v) \in \delta^+(A)} c(u, v).$$

Equivalently,

$$c(A, B) = \sum_{\substack{(u,v) \in E, \\ u \in A, \\ v \in B}} c(u, v).$$

Note: Only edges *from A to B* count — edges from B to A do not contribute to cut capacity.

Cut Capacity [2]



- Cut (A, B) with $A = \{s, a, b\}$, $B = \{c, d, t\}$.
- Red edges cross $A \to B$: capacity $12 + 14 = 26$.
- The gray back-edge $c \to b$ (from B to A) does *not* count.

Removing all $A \to B$ edges *severs* the network — no flow can reach t from s.

The *cut capacity* is the total pipe capacity you must cut to stop all flow.

Net Flow Across a Cut

Definition 69: Given a flow f and a cut (A, B), the *net flow across the cut* is

$$f(A, B) := \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e)$$

Theorem 35: For any feasible flow f and any s-t cut (A, B):

$$f(A, B) = |f|$$

Proof: Let $S := \sum_{v \in A} \left[\sum_{e \in \text{ out}(v)} f(e) - \sum_{e \in \text{ in}(v)} f(e) \right]$.

Expanding by edges, all terms from edges with both endpoints in A cancel, hence:

$$S = \sum_{(u,v) \in \delta^+(A)} f(u, v) - \sum_{(u,v) \in \delta^-(A)} f(u, v) = f(A, B).$$

Net Flow Across a Cut [2]

Also,

$$S = \left[\sum_{e \in \text{ out}(s)} f(e) - \sum_{e \in \text{ in}(s)} f(e) \right] + \sum_{v \in A \setminus \{s\}} \left[\sum_{e \in \text{ out}(v)} f(e) - \sum_{e \in \text{ in}(v)} f(e) \right].$$

Since $t \in B$, every $v \in A \setminus \{s\}$ is internal. By flow conservation:

$$\sum_{e \in \text{ out}(v)} f(e) - \sum_{e \in \text{ in}(v)} f(e) = 0$$

Therefore,

$$S = \sum_{e \in \text{ out}(s)} f(e) - \sum_{e \in \text{ in}(s)} f(e) = |f|.$$

Thus $f(A, B) = S = |f|$

$$f(A, B) = |f|$$

□

Net Flow Across a Cut [3]

Key corollary: For any flow f and any cut (A, B):

$$|f| = f(A, B) \leq c(A, B)$$

Every cut gives an *upper bound* on the maximum flow value.

Minimum Cut

Definition 70: The *minimum s-t cut* is the s-t cut with the smallest capacity:

$$c^* = \min_{(A,B):, s\in A,, t\in B} c(A, B)$$

Consequence of the upper bound: Every flow f satisfies $|f| \leq c(A, B)$ for every cut (A, B).

In particular, $|f| \leq c^*$ — no flow can exceed the minimum cut capacity.

The central question: can we always achieve equality?

— Yes! This is the **Max-Flow Min-Cut Theorem**.

Residual Network

To find maximum flows, we need to answer: *"where can we still push more flow?"*

Definition 71: Given a flow f in network N, the *residual capacity* of a pair (u, v) is:

$$c_f(u, v) = c(u, v) - f(u, v)$$

This covers both directions:

- if $(u, v) \in E$ then c_f is the *remaining* room (non-filled capacity);
- if $(v, u) \in E$ but $(u, v) \notin E$, then $c_f(u, v) = f(v, u)$ (the flow we can "*cancel*" or "*undo*").

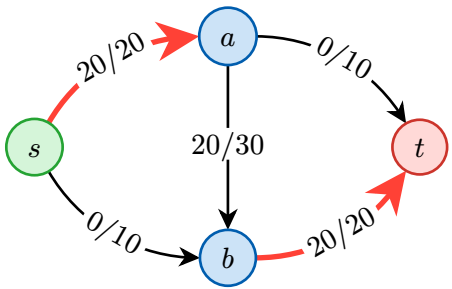
Definition 72: The *residual network* N_f consists of all pairs (u, v) with $c_f(u, v) > 0$:

- *Forward edge* (u, v): present when $f(u, v) < c(u, v)$, with capacity $c(u, v) - f(u, v)$.
- *Backward edge* (v, u): present when $f(u, v) > 0$, with capacity $f(u, v)$.

- A *forward* edge says "I can send more flow this way."
- A *backward* edge says "I can reduce the flow on this edge."

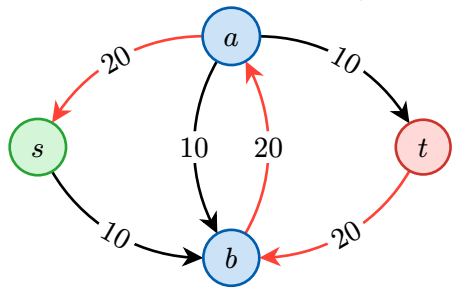
Residual Network: Example

Network with flow f



$|f| = 20$; saturated edges are red.

Residual network N_f



Red backward edges: "undo" capacity.

Path $s \to b \to a \to t$ exists in N_f:
an **augmenting path** with bottleneck 10!

Augmenting Paths

Definition 73: An *augmenting path* is an s-t path P in the residual network N_f.

Note: Every edge in P has positive residual capacity.

Definition 74: The *bottleneck* of P is $\Delta = \min_{e \in P} c_f(e) > 0$.

Augmenting Paths [2]

Theorem 36: If P is an augmenting path with *positive bottleneck* $\Delta > 0$, then the *augmented flow* f' (defined below) is a valid flow in N with $|f'| = |f| + \Delta$.

$$f'(u,v) = \begin{cases} f(u,v) + \Delta & \text{if } (u,v) \in P \\ f(u,v) - \Delta & \text{if } (v,u) \in P \\ f(u,v) & \text{otherwise} \end{cases}$$

Proof:

- **Capacity:** For a forward path edge (u,v): $f'(u,v) = f(u,v) + \Delta \leq f(u,v) + c_f(u,v) = c(u,v)$ and $f'(u,v) \geq 0$. For a backward path edge $(v,u) \in P$ (*i.e.*, (u,v) is a real edge used in reverse): $f'(u,v) = f(u,v) - \Delta \geq 0$ since $\Delta \leq c_f(v,u) = f(u,v)$.
- **Conservation:** Each internal vertex v of P has exactly one edge of P entering and one leaving. The $+\Delta$ and $-\Delta$ contributions cancel, so conservation is preserved.
- **Value:** The first edge of P out of s is a forward edge (since s has no incoming edges), so $|f'| = |f| + \Delta$.

Ford-Fulkerson Algorithm

Idea: Repeatedly find augmenting paths in the residual network and push flow along them until no augmenting path exists.

Input: Flow network $N = \langle V, E, s, t, c \rangle$

Output: Maximum flow f

Set $f(e) = 0$ for all $e \in E$

while there exists an s-t path P in residual network N_f **do**

  Let $\Delta = \min_{e \in P} c_f(e)$ *(bottleneck)*

  for each edge $(u, v) \in P$ **do**

    $f(u, v) := f(u, v) + \Delta$

    $f(v, u) := f(v, u) - \Delta$ *(skew-symmetry)*

return f

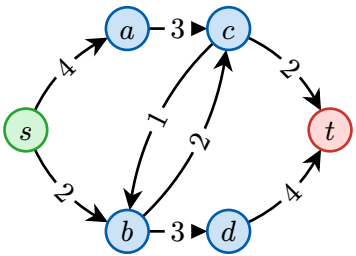
Ford-Fulkerson Algorithm [2]

Termination condition: For integer capacities, each iteration increases $|f|$ by at least 1, so the algorithm terminates in at most $|f^*|$ steps.

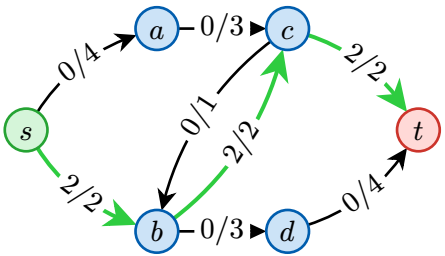
Warning: With irrational capacities, Ford-Fulkerson may not terminate — augmenting paths can reduce flow along one edge while increasing along another in an infinite cycle.
This motivates Edmonds-Karp (always use BFS).

Ford-Fulkerson: Worked Example

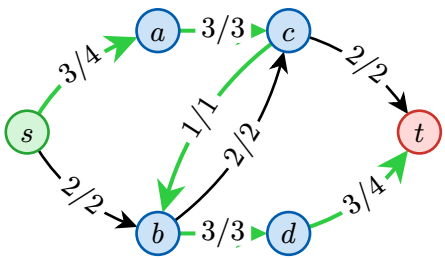
Step 0: $|f| = 0$



Step 1: $s \to b \to c \to t$, $\Delta = 2$



Step 2: $s \to a \to c \to b \to d \to t$, $\Delta = 3$



After step 2: $|f| = 5$. No augmenting path exists in the residual network — the algorithm terminates.

Step 2 uses the *backward* edge $c \to b$ in the residual network (cancelling 1 unit of the $b \to c$ flow from step 1) and then routes through d. Without backward edges, this path would not be possible.

Max-Flow Min-Cut Theorem

Theorem 37: In any flow network N, the following three conditions are *equivalent*:

1. f is a *maximum flow*.
2. There is *no augmenting path* in the residual network N_f.
3. There exists an s-t cut (A, B) with $|f| = c(A, B)$.

Moreover, when these hold, (A, B) is a *minimum cut*.

This is one of the deepest results in combinatorics — it equates two seemingly unrelated quantities: the maximum achievable flow and the minimum bottleneck capacity.

Proof of Max-Flow Min-Cut

Proof *(1 → 2) Contrapositive*: If an augmenting path P exists, its bottleneck $\Delta > 0$, so we can increase $|f|$. Hence f was not maximum. □

Proof *(3 → 1)*: For *any* flow f' and *any* cut (A, B) we showed $|f'| \leq c(A, B)$. Since $|f| = c(A, B)$ for this particular cut, no flow can exceed $|f|$. So f is maximum. □

Proof *(2 → 3)*: Define $A = \{v \in V \texttt{ mid } \exists \text{ path } s \rightsquigarrow v \text{ in } N_f\}$. Since no augmenting path exists, $t \notin A$. Set $B = V \setminus A$.

All $A \to B$ edges are saturated: if $(u, v) \in E$ with $u \in A$, $v \in B$ had $c_f(u, v) > 0$, then v would be reachable from s, contradicting $v \in B$. Hence $f(u, v) = c(u, v)$ for all such edges.

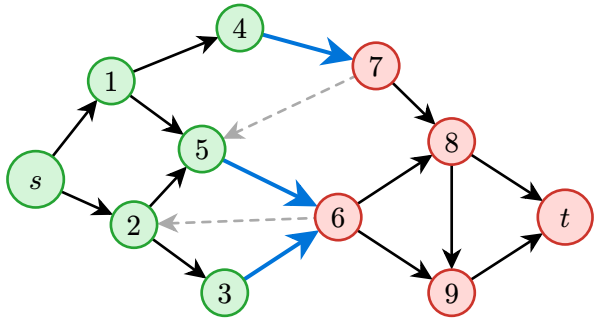
All $B \to A$ edges carry zero flow: if $(u, v) \in E$ with $u \in B$, $v \in A$ had $f(u, v) > 0$, then $c_f(v, u) = f(u, v) > 0$, so the backward edge $v \to u$ would be in N_f, making u reachable − contradicting $u \in B$.

Therefore:

$$|f| = f(A, B) = \sum_{\substack{u \in A, \\ v \in B}} \underbrace{f(u, v)}_{= c(u,v)} - \sum_{\substack{u \in B, \\ v \in A}} \underbrace{f(u, v)}_{= 0} = c(A, B)$$

□

Visualization of Max-Flow Min-Cut



- s, 1–5 = reachable set A in N_f.
- 6–9, t = $B = V \setminus A$.
- Blue edges ($A \to B$) are *saturated* ($f = c$).
- Gray back-edges ($B \to A$) are *empty* ($f = 0$).

Edmonds-Karp Algorithm

Ford-Fulkerson leaves the choice of augmenting path *unspecified*. A poor choice can lead to:

- Non-termination with irrational capacities.
- $O(|f^*| \cdot E)$ iterations with integer capacities — slow when $|f^*|$ is large.

Edmonds-Karp (1972): Always choose the *shortest* augmenting path (fewest edges), found by BFS.

Theorem 38: Edmonds-Karp runs in $O(VE^2)$ time — independent of the capacity values.

The key insight: with BFS shortest paths, the distance from s to every vertex is non-decreasing across iterations. Each edge can become a bottleneck at most $O(V)$ times. Hence the total number of augmentations is $O(VE)$, and each BFS costs $O(E)$.

Edmonds-Karp Algorithm [2]

Comparison of max-flow algorithms:

Algorithm	Time complexity	Key idea
Ford-Fulkerson	$O(\|f^*\| \cdot E)$	Any augmenting path
Edmonds-Karp	$O(VE^2)$	BFS shortest augmenting path
Dinic's	$O(V^2E)$	Blocking flows in level graph
Push-relabel	$O\left(V^2\sqrt{E}\right)$	Height-based local pushes

Integrality Theorem

Theorem 39 (Integrality of Flow): If all capacities in N are integers, then there exists a maximum flow that is *integer-valued* (every $f(e) \in \mathbb{Z}$).

Proof: Ford-Fulkerson starts with $f = 0$ (integer). At each step, the bottleneck Δ is the minimum of integer residual capacities — hence an integer. Augmenting adds Δ to each edge on the path. By induction, every flow value remains an integer throughout, so the final flow is integer-valued. □

Why this matters: Many combinatorial problems (matchings, disjoint paths, covers) naturally have integer optimal solutions. The Integrality Theorem guarantees this rigorously: run max-flow on the right network to get an integer solution.

Applications of Max-Flow

"A good algorithm is one that solves a hard problem by reducing it to an easy one."

Overview: Max-Flow as a Meta-Theorem

The power of reduction: Many combinatorial optimization problems can be formulated as max-flow instances. Solving them reduces to: *"construct the right network, then run Edmonds-Karp."* Correctness follows from Max-Flow Min-Cut; efficiency is $O(VE^2)$.

Matching & assignment:
- Maximum bipartite matching
- Weighted bipartite assignment
- Project-student assignment

Covering & duality:
- König's theorem
- Minimum vertex cover
- LP duality interpretation

Connectivity:
- Maximum edge-disjoint paths
- Maximum vertex-disjoint paths
- Menger's theorem (revisited)

Optimization with constraints:
- Project selection (closure)
- Survey design
- Image segmentation

Application 1: Maximum Bipartite Matching

Problem: Given bipartite graph $G = (X \cup Y, E)$, find the largest matching.

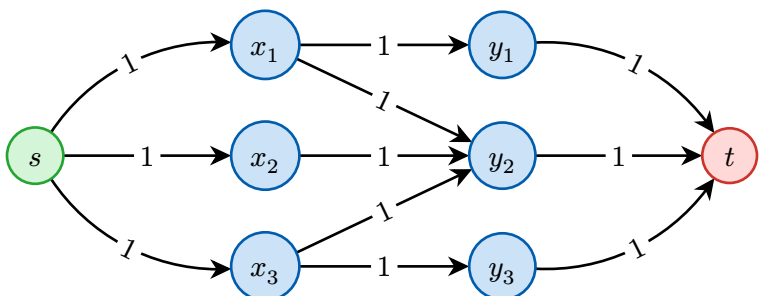
Reduction: Construct a flow network:

- Add super-source s with edge $s \to x$ of capacity 1 for all $x \in X$.
- Add super-sink t with edge $y \to t$ of capacity 1 for all $y \in Y$.
- Keep original edges $X \times Y$ with capacity 1.
- Run max-flow on this network.

Application 1: Maximum Bipartite Matching [2]

Theorem 40: The maximum flow in this network equals the size of the maximum matching in G.

Proof: Any integer flow of value k consists of k unit-flow paths $s \to x_i \to y_i \to t$. These correspond to k matching edges, pairwise distinct (each x_i and y_j has unit capacity toward s/t, so no two paths share an endpoint). Conversely, any matching of size k gives a feasible integer flow of value k. By the Integrality Theorem, a maximum integer flow exists and equals the maximum matching. □



Application 2: König's Theorem via Max-Flow

Theorem 41 (König): In a bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover.

Proof *(via max-flow)*: Let the max-flow (= max matching) have value k.

- By Max-Flow Min-Cut, there exists a cut (A, B) with $c(A, B) = k$.
- In this unit-capacity bipartite network, every finite-capacity cut edge corresponds to exactly one vertex: a s-side cut edge $s \to x$ (capacity 1) to some $x \in X$, or a t-side cut edge $y \to t$ (capacity 1) to some $y \in Y$.
- The set of these vertices forms a vertex cover of size k (every original edge $xy \in E$ must be covered, for otherwise the path $s \to x \to y \to t$ would cross the cut at zero cost, yielding $c(A, B) < k$).

Conversely, any vertex cover of size k defines a cut of capacity k.
Hence, $\text{max matching} = k = \text{min vertex cover}$. □

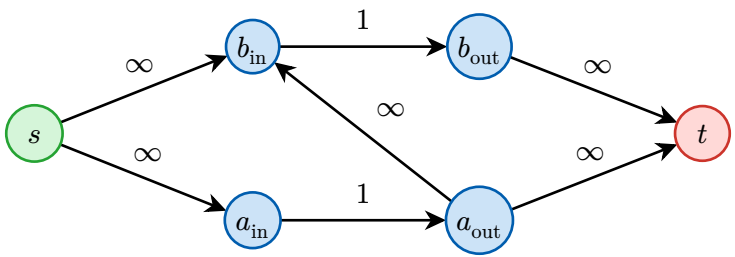
Application 2: König's Theorem via Max-Flow [2]

Deeper: König's theorem is an instance of *LP duality*. The LP relaxations of max matching and min vertex cover are dual programs. For bipartite graphs, both LPs have integer optimal solutions, so the LP optimum equals the integer optimum — and the two coincide.

Application 3: Menger's Theorem via Max-Flow

Edge-disjoint paths: Replace each undirected edge $\{u, v\}$ with two directed edges (u, v) and (v, u), each of capacity 1. The max flow from s to t equals the maximum number of edge-disjoint s-t paths. The min cut equals the minimum edge separator — this is precisely **Menger's theorem (edge form)**.

Vertex-disjoint paths: Replace each internal vertex v with a pair $v_{\text{in}}, v_{\text{out}}$ connected by an edge of capacity 1. All original edges become arcs from u_{out} to v_{in} with capacity ∞.



The max flow in this network equals the maximum number of internally vertex-disjoint s-t paths. The min cut equals the minimum vertex separator — this is **Menger's theorem (vertex form)**.

Application 4: Project Selection (Closure Problem)

Problem: You have a set of *projects* P, each with a profit p_i (which may be negative). Some projects *depend on* others: selecting i forces you to also select j for each dependency $i \to j$. Find a feasible set $S \subseteq P$ maximizing $\sum_{i \in S} p_i$.

Reduction to min-cut:

- Source s, sink t.
- For each profitable project i ($p_i > 0$): add edge $s \to i$ with capacity p_i.
- For each costly project i ($p_i < 0$): add edge $i \to t$ with capacity $|p_i|$.
- For each dependency $i \to j$: add edge $i \to j$ with capacity ∞.

Theorem 42: $\text{Max profit} = \left(\sum_{p_i > 0} p_i\right) - \min \text{ cut}(s, t)$.

Intuition: The min cut separates *selected* (A, containing s) from *not selected* (B, containing t). Cutting $s \to i$ means forgoing the profit of project i. Cutting $j \to t$ means project j is excluded at cost $|p_j|$. Infinite-capacity dependency edges cannot be cut, enforcing the selection constraints.

Real-World Applications

Direct flow problems:

- Pipeline throughput (oil, gas, water)
- Internet bandwidth allocation
- Airline scheduling (crew assignment)
- Traffic flow optimization

Combinatorial optimization:

- Job scheduling on machines
- Hospital-patient matching
- Image segmentation (min-cut)
- Baseball elimination (Schwartz 1966)
- Open-pit mining (project selection)

Historical note: The Max-Flow Min-Cut theorem was proved independently by Ford & Fulkerson (1956) and by Elias, Feinstein & Shannon (1956) — the latter motivated by information theory. It is one of the landmark results connecting graph theory, linear programming, and combinatorics.

Baseball elimination (Schwartz 1966): Team i is *eliminated* from winning if and only if the max flow in a specific network falls below a threshold. This surprised many sports analysts who assumed playoff standings were easy to compute by hand!

Edmonds-Karp: Implementation

The implementation follows directly from the algorithm: use BFS to find the shortest augmenting path, then update the residual graph by augmenting along it. Store the residual graph as an adjacency structure with explicit reverse edges.

```
from collections import deque

def bfs(cap, s, t, parent):
    visited = {s}
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v, c in cap[u].items():
            if v not in visited and c > 0:
                visited.add(v)
                parent[v] = u
                if v == t:
                    return True
                queue.append(v)
    return False
```

Edmonds-Karp: Implementation [2]

```
def max_flow(cap, s, t):
    flow = 0
    while True:
        parent = {}
        if not bfs(cap, s, t, parent):
            break  # No augmenting path
        # Find bottleneck along BFS path
        path_flow, v = float('inf'), t
        while v != s:
            u = parent[v]
            path_flow = min(path_flow, cap[u][v])
            v = u
        # Augment (update residual capacities)
        v = t
        while v != s:
            u = parent[v]
            cap[u][v] -= path_flow
```

Edmonds-Karp: Implementation [3]

```
            cap[v][u] += path_flow  # reverse edge
            v = u
        flow += path_flow
    return flow
```

Note: `cap` is a dictionary of dictionaries. Initialize `cap[v][u] = 0` for every reverse edge before running. The `cap[v][u] += path_flow` line automatically maintains the residual backward capacity.

Summary: Network Flow Landscape

Problem	Reduction	Complexity
Max bipartite matching	Unit network	$O\left(E\sqrt{V}\right)$
Max edge-disjoint s-t paths	Direct (unit caps)	$O(E^2)$
Max vertex-disjoint s-t paths	Vertex splitting	$O(V \cdot E)$
Menger's theorem	Unit network	Follows from above
Hall's theorem	Unit bipartite net	Follows from matching
König's theorem	Min-cut duality	Follows from matching
Project selection	Closure via min-cut	$O(V^2E)$

Meta-theorem: Max-Flow Min-Cut is a special case of *LP strong duality*. Many "maximum equals minimum" results in combinatorics — Hall, König, Menger, Dilworth — are all instances of this single unifying principle.

Network Flows: Exercises

1. Trace Ford-Fulkerson on the network from the "Worked Example" slide and verify $|f^*| = 5$.
2. Find the minimum s-t cut in the network and confirm its capacity equals the max flow.
3. Reduce the following bipartite matching to a max-flow problem and find the answer: $X = \{1, 2, 3\}$, $Y = \{a, b, c\}$, edges: 1-a, 1-b, 2-b, 3-b, 3-c.
4. Apply vertex splitting to find the maximum number of internally vertex-disjoint s-t paths in your favourite small graph.
5. Three projects $P = \{A, B, C\}$ with profits $p_A = 10$, $p_B = -5$, $p_C = 8$, dependencies $A \to B$. Set up the min-cut network and determine the optimal selection.
6. Prove that the minimum cut (A, B) found by the proof of Max-Flow Min-Cut satisfies $c(A, B) = |f|$ without using the theorem itself.

Summary and Connections

Graph Theory: Key Concepts

Structural concepts:

- Degree, adjacency, neighborhoods
- Paths, cycles, connectivity
- Trees, spanning trees, forests
- Bipartiteness (2-colorability)
- Planarity (Euler's formula)

Optimization problems:

- Matchings (Hall, König)
- Vertex/edge covers
- Graph coloring (χ, χ')
- Cliques and stable sets
- Connectivity (Menger)

Network flows:

- Flow networks (capacity, conservation)
- Max-flow min-cut theorem
- Ford-Fulkerson / Edmonds-Karp
- Applications (matching, paths)
- Integrality theorem

Foundational theorems:

- Handshaking: $\sum \deg(v) = 2m$
- Euler's formula: $n - m + f = 2$
- Hall's marriage theorem (matchings $\leftrightarrow$ neighborhoods)
- Menger's theorem (paths $\leftrightarrow$ cuts)
- Max-Flow Min-Cut theorem ($\max|f| = \min c(A, B)$)

Graph Theory & Flows: The Big Picture

Network flows unify graph theory:

- Maximum bipartite matching = max flow in a unit network
- Menger's theorem = max-flow min-cut with unit capacities
- Hall's condition = feasibility of a flow in a bipartite network
- König's theorem = strong LP duality for bipartite matching

One theorem rules them all: Max-Flow = Min-Cut is an instance of *LP strong duality*. The combinatorial "max = min" theorems of Hall, König, Menger, and Dilworth are all special cases of this single algebraic principle.

Graph theory provides the foundation for:

- Algorithms (BFS, DFS, shortest paths, MST, max-flow)
- Network design and optimization
- Formal language theory (automata are directed labeled graphs!)