Graph Theory - Comprehensive Topics

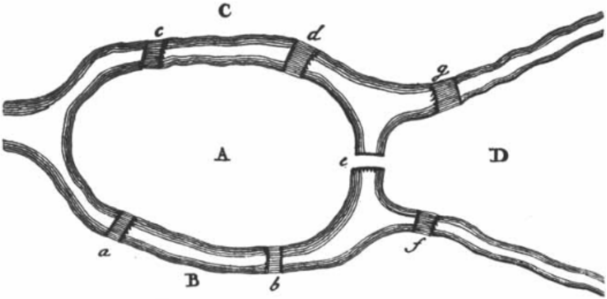
Detailed Concepts, Examples, and Exercises

SDB

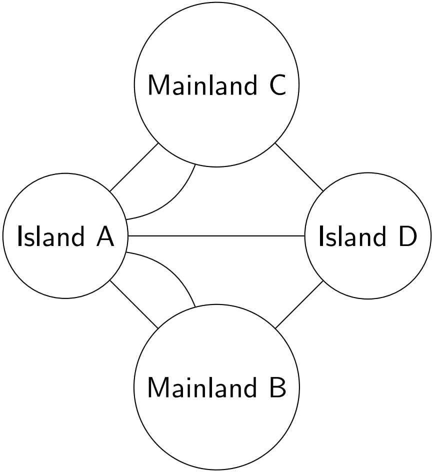
Spring 2025

Koenigsberg Bridges Problem - Original Scenario I

- The city of Koenigsberg (now Kaliningrad) was situated on both sides of the Pregel River.
- The river enclosed two islands connected to each other and the mainland by seven bridges.
- The challenge: Start at any point and traverse all bridges exactly once, returning to the starting point.



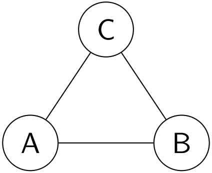
Koenigsberg Bridges Problem - Original Scenario II



Outline

Graph and Graph as Models

- A **Graph** $G = (V, E)$ consists of:
  - V: A set of vertices (nodes).
  - E: A set of edges (connections between vertices).
- Graphs are used to model relationships in various fields:
  - Social networks, transportation, communication, etc.



Exercise:

- Define a graph for your daily commute.
- List its vertices and edges.

Question to Ponder: Can all real-world problems be modeled using graphs?

Formal Definition of a Graph I

- A **graph** G is defined as an ordered pair:

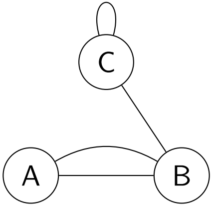
$$G = (V, E)$$

  where:
  - V is the set of vertices (nodes).
  - E is the set of edges (connections) represented as a subset of $V \times V$.
- Each edge is a relation (u, v), where $u, v \in V$.

Extensions:

- **Self-loop:** An edge of the form (v, v) where $v \in V$.
- **Parallel Edges:** Two or more edges connecting the same pair of vertices, represented as distinct elements in E.

Formal Definition of a Graph II



Exercise:

- Define the vertex set and the edge set for the above graph.

Question to Ponder: How do self-loops and parallel edges affect graph properties such as connectivity and cycles?

Basic Concepts in Graph Theory

- **Vertex (Node)**: A point in the graph, represented by a circle or a dot. The plural form of vertex is **vertices**, and it is also commonly referred to as **nodes** (with the plural form being **nodes**).
- **Edge (Arc)**: A line segment connecting two vertices, representing a relationship between them. The plural form of edge is **edges**, and it is also commonly referred to as **arcs** (with the plural form being **arcs**).

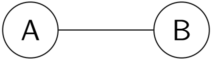


Figure: Vertex (Node) and Edge (Arc)

Note: Throughout this presentation, the terms vertex and node, as well as edge and arc, will be used interchangeably.

Degree of a Vertex

- **Degree of a Vertex**: The number of edges incident on a vertex; note that, it is also sometimes referred to as **valence** of a vertex.
- **Notation**: $d(v)$ or $\deg(v)$

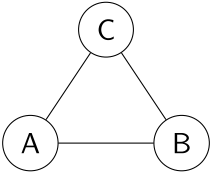


Figure: Degree of Vertices: $d(A) = 2$, $d(B) = 2$, $d(C) = 2$

Types of Degrees

- **Isolated Vertex**: A vertex with degree 0.
- **Pendant Vertex**: A vertex with degree 1.
- **Interior Vertex**: A vertex with degree greater than 1.

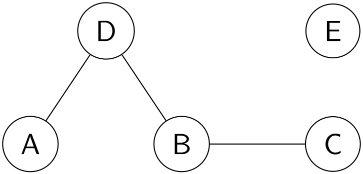


Figure: $d(A) = 1$, $d(B) = 2$, $d(C) = 1$, $d(D) = 2$, $d(E) = 0$

Adjacent Vertices

- **Adjacent Vertices**: Two vertices (nodes) are adjacent if they are connected by an edge (arc).
- **Notation**: $u \sim v$ or u is adjacent to v.

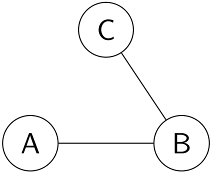


Figure: Adjacent Vertices: $A \sim B$, $B \sim C$, but $A \nsim C$

Neighbors of a Vertex

- **Neighbors of a Vertex**: The set of all vertices adjacent to a given vertex.
- **Notation**: $N(v)$ or $N(v) = \{u \mid u \sim v\}$.

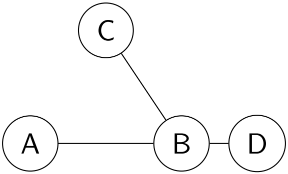


Figure: Neighbors of a Vertex: $N(B) = \{A, C, D\}$

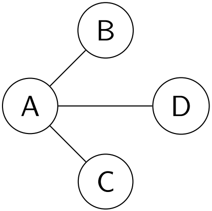
Outline

Types of Graphs I

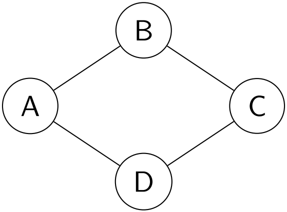
- Graphs can be categorized based on their structure and properties. Common types include:
  1. **Star Graph:** One central vertex connected to all others.
  2. **Cycle Graph:** A graph that forms a single closed loop.
  3. **Complete Graph:** Every pair of vertices is connected by an edge.
  4. **Bipartite Graph:** Vertices can be divided into two disjoint sets with edges only between the sets.
  5. **Tree:** A connected graph with no cycles.
  6. **Plannar Graph:** A graph that can be drawn on a plane without any edge overlapping.

Types of Graphs II

Examples:



Star Graph($K_{1,n}$)



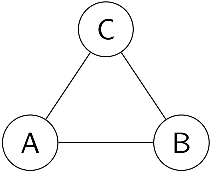
Cycle Graph(C_n)

Question to Ponder: How do different graph types model real-world problems?

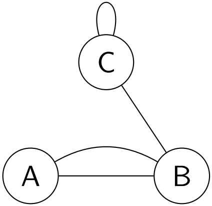
Simple and General Graphs

- **Simple Graph:**
  - Contains no self-loops (edges that connect a vertex to itself).
  - No multiple edges between any pair of vertices.
- **General Graph:**
  - May contain loops and multiple edges between pairs of vertices.
  - Represents more complex relationships compared to simple graphs.
  - Also referred to as **Multigraph** (no self-loops) and **Pseudograph** (no restrictions). *See Appendix.*

Examples:



Simple Graph

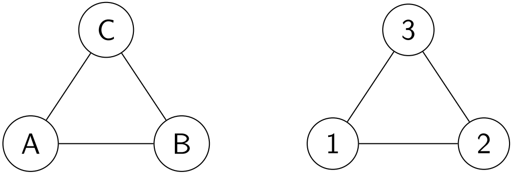


General Graph

Question to Ponder: How do loops and multiple edges change the properties of a graph?

Matrices and Isomorphism

- **Adjacency Matrix**: Represents edges between vertices as a matrix.
- **Isomorphism**: Two graphs are isomorphic if their structure can be mapped identically.



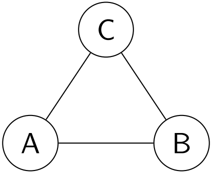
Exercise:

- Write the adjacency matrix for the above graphs.
- Prove their isomorphism.

Question to Ponder: Why is identifying isomorphism computationally challenging?

Complete Graphs

- A **complete graph** is a graph in which every pair of vertices is connected by an edge.
- Denoted as K_n, where n is the number of vertices.



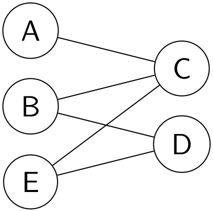
Exercise:

- Verify if a graph with n vertices and $n(n-1)/2$ edges is complete.

Question to Ponder: How do complete graphs relate to real-world applications?

Bipartite Graphs

- A graph is **bipartite** if its vertices can be divided into two disjoint sets such that every edge connects a vertex from one set to the other.
- No edges exist between vertices of the same set.

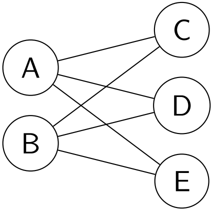


⋆ *Notice that cycle graphs with an even number of vertices are always bipartite.*

Question to Ponder: How can bipartite graphs be used in matching problems?

Complete Bipartite Graphs

- A **complete bipartite graph** is a bipartite graph where every vertex in one set is connected to every vertex in the other set.
- Denoted as $K_{m,n}$, where m and n are the sizes of the two sets.



⋆ *Notice that a* **Star Graph** *with n vertices is basically* $K_{1,n-1}$.

Exercise:

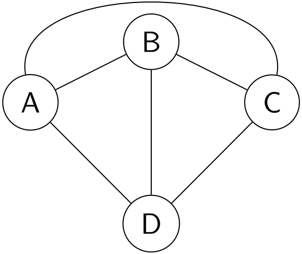
- Verify if the given graph is a complete bipartite graph.

Question to Ponder: How are complete bipartite graphs used in network design?

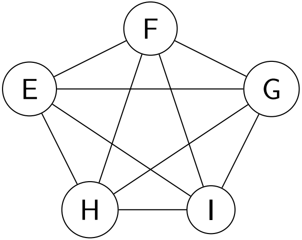
Planar Graphs I

- **Definition:** A graph is planar if it can be drawn on a plane without any edges crossing. - Such a drawing is called a plane embedding of the graph.
- Properties:
  - **Euler's Formula:** For a connected planar graph: $V - E + F = 2$, where V is the number of vertices, E the number of edges, and F the number of faces (including the outer face).
  - A **planar graph** must satisfy: $E \leq 3V - 6 \quad \text{for } V \geq 3$.
  - A **bipartite planar graph** satisfies: $E \leq 2V - 4$.
- Examples:
  - Planar: K_4, Cycle graphs (C_n).
  - Non-Planar: K_5, $K_{3,3}$.

Planar Graphs II



Planner K_4



Non-Planner K_5

Exercise:

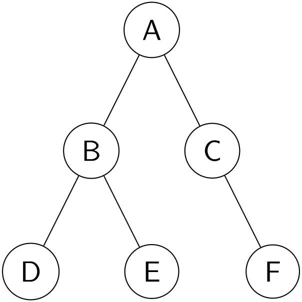
- Verify Euler's formula for the K_4 graph.
- Prove why K_5 is non-planar using edge and vertex counts.

Tree Graph I

- **Definition:** A tree is an undirected, connected, and acyclic graph. - Formally, a graph $T = (V, E)$ is a tree if:
  - T is connected (a path exists between any two vertices).
  - T contains no cycles.
- **Properties of Trees:**
  - A tree with n vertices has exactly $n - 1$ edges.
  - Adding one edge to a tree creates exactly one cycle.
  - Removing any edge from a tree disconnects it.
  - There is a unique path between any pair of vertices.
- **Examples of Trees:**
  - **Star Tree** ($K_{1,n}$): One central vertex connected to all others.
  - **Path Tree** (P_n): A tree where all vertices form a single path.
  - **Binary Tree**: A tree where each vertex has at most two children.

Tree Graph II

Visualization:



Exercise:

- Prove that a tree with n vertices has exactly $n - 1$ edges using induction.
- Identify the types of trees (binary, star, path) in the above visualization.

Outline

Vertex Degrees and Counting

- The **degree** of a vertex is the number of edges incident to it.
- Sum of all vertex degrees equals twice the number of edges.

Exercise:

- Prove the Handshaking Lemma for a graph with n vertices and m edges.

Question to Ponder: How can vertex degrees help in identifying graph properties?

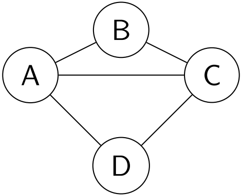
Graph Size and Order I

- **Definition:**
  - **Order** ($|V|$): The order of a graph is the number of vertices it contains.
  - **Size** ($|E|$): The size of a graph is the number of edges it contains.
- **Notation:** A graph G is represented as $G = (V, E)$, where:
  - V is the vertex set.
  - E is the edge set.
  - $|V|$ is the graph's order, and $|E|$ is its size.
- **Examples:**
  - For a complete graph K_n: Order: $|V| = n$, Size: $|E| = \binom{n}{2} = \frac{n(n-1)}{2}$.
  - For a cycle graph C_n: Order: $|V| = n$, Size: $|E| = n$.
  - For a path graph P_n: Order: $|V| = n$, Size: $|E| = n - 1$.
  - For any tree: Order: $|V| = n$, Size: $|E| = n - 1$.

Graph Size and Order II

- **Properties:**
  - The **size** of a graph is always bounded by: $0 \leq |E| \leq \binom{|V|}{2}$.
  - **Sparse graph:** $|E| \ll |V|^2$, **Dense graph:** $|E| \approx |V|^2$.

Visualization:



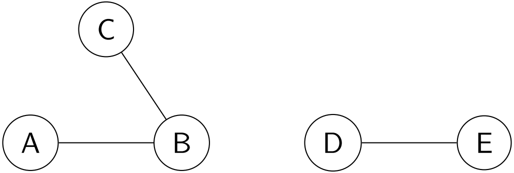
- Order: $|V| = 4$ (A, B, C, D).
- Size: $|E| = 5$ (five edges drawn).

Exercise:

- Determine the size and order of the complete bipartite graph $K_{3,2}$.
- Find the size of a tree with 10 vertices.

Connection in Graphs

- A graph is **connected** if there is a path between every pair of vertices.
- **Disconnected Graph**: Contains at least two components.



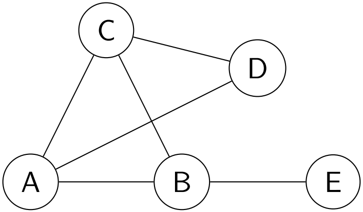
Exercise:

- Determine if the graph is connected.
- Identify its components.

Question to Ponder: How does connectivity impact graph algorithms?

Degree Sequence

- A **degree sequence** $(d_1, d_2, \ldots, d_n)$ is a list of vertex degrees in non-increasing order.
- Example: For the graph below, the degree sequence is (3, 3, 3, 2, 1).
- Not all degree sequences are valid for simple graphs. Conditions must be satisfied to determine if a sequence is **graphic**.



Exercise:

- Determine the degree sequence of a star graph with 6 vertices.
- Verify if the sequence (4, 3, 3, 2, 2, 1) is graphic.

Question to Ponder: How can the degree sequence help in identifying graph properties?

Graphic Sequences

- A sequence of numbers, $(d_1, d_2, \ldots, d_n)$, is **graphic** if it represents the degree sequence of a simple graph. Following conditions must be true.
  1. The sequence must be made up of non-negative integers
  2. The sum of all the degrees must be even
  3. d_i can be at most $n - 1$
- Example: (3, 3, 2, 2) is graphic (construct the graph to verify).

Exercise:

- Verify if (3, 2, 2, 1) is graphic.
- How about (5, 3, 2, 1, 1, 1) and (5, 3, 2, 1, 1)?

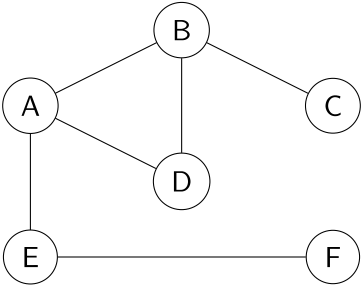
Question to Ponder: What conditions make a sequence graphic?

Degree Sequence to Graph Construction I

- **Problem Statement:** Given a degree sequence, construct a graph or determine if no graph exists.
- **Havel-Hakimi Algorithm:** A constructive method to check if a degree sequence is graphical and build the graph:
  1. Sort the degree sequence in non-increasing order.
  2. Remove the first vertex (highest degree) and decrease the degree of the next highest *d* vertices.
  3. Repeat until all degrees are 0 or a contradiction arises.

Degree Sequence to Graph Construction II

Example: Degree sequence: $\{3, 3, 2, 2, 1, 1\}$.



Exercise:

- Determine if the sequence $\{4, 3, 3, 2, 2, 2\}$ is graphical.
- Construct the graph if the sequence is valid.

Question to Ponder: Can there be more than one graph for a given degree sequence?

Walk, Trail, Circuit, Path and Cycle I

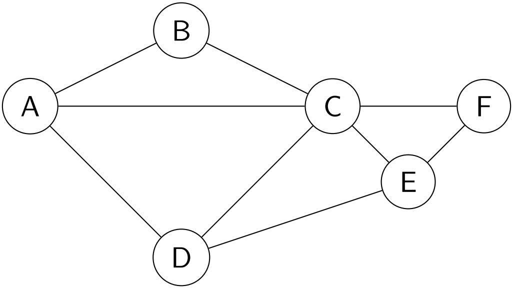
- **Walk:** A sequence of vertices and edges; vertices and edges may repeat: $W = v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$ where: $\forall 1 \leq i \leq k \quad e_i = \{v_{i-1}, v_i\}$.
- **Trail:** A walk in which all edges are distinct: $\forall i, j \quad i \neq j \quad e_i \neq e_j$.
- **Circuit:** A closed trail (first and last vertices are the same), where: $\forall i, j \quad i \neq j \quad v_0 = v_k, \quad e_i \neq e_j$. a
- **Path:** A trail in which all vertices are distinct: $\forall i, j \quad i \neq j \quad v_i \neq v_j$.
- **Cycle:** A closed path (first and last vertices a(e the same), where: $\forall i, j \quad i \neq j,\, i, j \in [1, k-1] \quad v_0 = v_k, \quad v_i \neq v_j$.

Notes:

- Paths and cycles are subsets of trails and walks.
- Circuits allow repeated vertices but not edges, while cycles restrict both.

Walk, Trail, Circuit, Path and Cycle II

Example:



- Example Walk: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ C $\rightarrow$ B $\rightarrow$ A $\rightarrow$ B.
- Example Trail: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ C $\rightarrow$ A $\rightarrow$ D.
- Example Circuit: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ C $\rightarrow$ A.
- Example Path: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ E $\rightarrow$ F.
- Example Cycle: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A.

Walk, Trail, Circuit, Path and Cycle III

Exercise:

- Identify whether the following sequences in the graph are walks, trails, paths, circuits, or cycles:
  1. A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A.
  2. F $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ C $\rightarrow$ E $\rightarrow$ C $\rightarrow$ B $\rightarrow$ A $\rightarrow$ C.
  3. A $\rightarrow$ D $\rightarrow$ C $\rightarrow$ E $\rightarrow$ F $\rightarrow$ C $\rightarrow$ B $\rightarrow$ A.
- Identify all cycles and circuits in the graph.
- Identify all paths between A and F in the graph.
- Identify all Walks between A and B in the graph.
- Identify all trails between A and D in the graph.

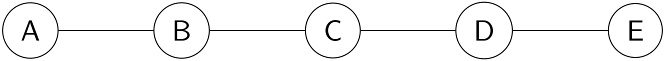
Diameter of a Graph I

- The **diameter** of a graph is the longest shortest path between any two vertices.
- It measures the graph's "largest reachability" in terms of path length.
- The **diameter** of a graph G is defined as:

$$\mathrm{diam}(G) = \max_{u,v \in V} d(u,v)$$

  where:
  - $d(u,v)$ is the shortest path distance between vertices u and v in G.
  - The diameter is the largest shortest path distance between any two vertices in the graph.
- If the graph is disconnected, the diameter is considered infinite.

Example:



Diameter of a Graph II

Exercise:

- Compute the diameter of a star graph with 5 vertices.
- Determine the diameter of a cycle graph with 6 vertices.

Question to Ponder:

- How does the diameter relate to the efficiency of communication in a network?
- What is the relationship between the diameter and the connectivity of a graph?

Eccentricity of a Vertex I

- The **eccentricity** of a vertex v in a graph G is the maximum shortest path distance from v to any other vertex in G.
- Formally,

$$e(v) = \max_{u \in V(G)} d(v, u)$$

where $d(v, u)$ is the shortest path distance between v and u.

Note: *Eccentricity helps in understanding the structure of a graph. It is used in network analysis, shortest path algorithms, and centrality measures.*

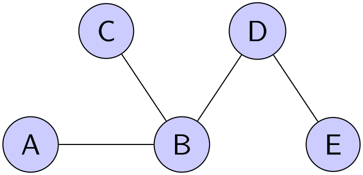
Properties of Eccentricity Definition:

- The vertex with the smallest eccentricity is called a **central vertex**.
- The largest eccentricity among all vertices is the **diameter** of the graph.
- The smallest eccentricity among all vertices is the **radius** of the graph.

Eccentricity of a Vertex II

Example:

- Consider the following graph:



- Compute $e(v)$ for each vertex and determine the radius and diameter.

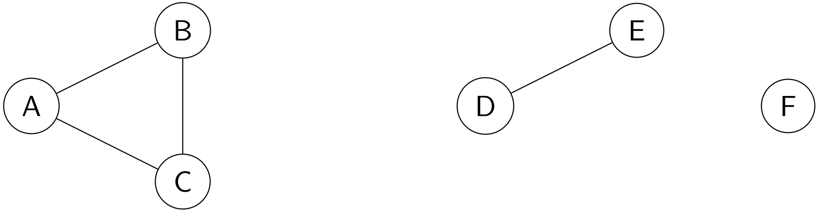
Exercises

- Compute the eccentricity of each vertex in the given graph.
- Determine the central vertex (or vertices).
- Find the diameter and radius of the graph.
- Modify the graph by adding an edge and observe how the eccentricities change.

Components of a Graph I

- **Definition:** A component of a graph is a maximal connected subgraph, meaning that:
  - Any two vertices in the same component are connected by a path.
  - No additional vertices or edges can be added without breaking connectivity.
- **Properties:**
  - A graph may consist of one or more components.
  - The components of a graph are disjoint.
  - Every vertex belongs to exactly one component.
- **Types of Components:**
  - **Connected Component:** A subgraph where all vertices are connected.
  - **Isolated Component:** A single vertex with no edges.

Components of a Graph II

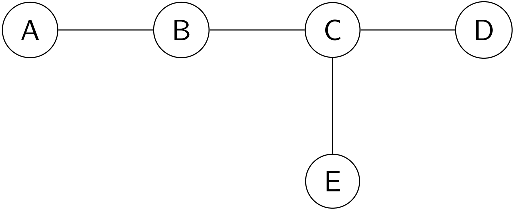


- In the above graph:
  - Component 1: $\{A, B, C\}$.
  - Component 2: $\{D, E\}$.
  - Component 3: $\{F\}$ (an isolated component).
- **Exercise:**
  - Identify all components in a complete bipartite graph $K_{2,3}$.
  - Prove: A graph with n vertices and no edges has n components.

Cut Edges and Cut Vertices I

- **Cut Edge (Bridge):**
  - A cut edge (or bridge) is an edge whose removal increases the number of connected components in the graph.
  - **Example:** In the graph below, edge $A \to B$ is a cut edge.
- **Cut Vertex (Articulation Point):**
  - A cut vertex is a vertex whose removal increases the number of connected components in the graph.
  - **Example:** In the graph below, vertex C is a cut vertex.
- **Properties:**
  - An edge is a cut edge if and only if it does not belong to any cycle in the graph.
  - In a complete graph, no vertex is a cut vertex.
  - In a tree, all vertices with degree greater than 1 are cut vertices, and all edges are cut edges because removing any edge disconnects the tree.
  - Removal of a cut vertex or cut edge isolates one or more subgraphs.

Cut Edges and Cut Vertices II



- In the above graph:
  - **Cut Edge:** Removing $B \rightarrow C$ disconnects A from the rest of the graph.
  - **Cut Vertex:** Removing C separates the graph into two disconnected components.
- **Exercise:**
  - Identify all cut edges and cut vertices in the graph above.
  - Prove that a tree with n vertices has at least $n - 1$ cut edges.

Complement Graph I

- **Definition:** The complement of a graph $G = (V, E)$ is a graph $\overline{G} = (V, \overline{E})$, where:

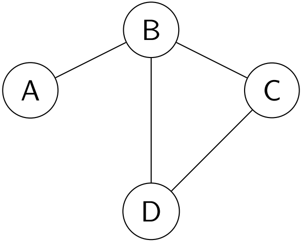
$$\overline{E} = \{(u, v) \mid u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}.$$

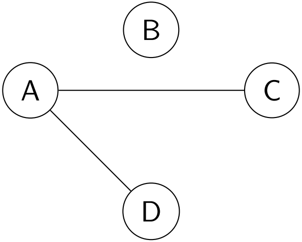
  - The complement graph $\overline{G}$ contains all edges not present in G, with the same vertex set V.

- **Properties:**
  - G and $\overline{G}$ together form a complete graph K_n.
  - If G is complete, then $\overline{G}$ is empty.
  - If G is disconnected, $\overline{G}$ is connected (for $|V| \geq 3$).

Complement Graph II

Example 1: Graph G:



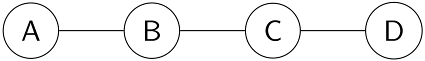


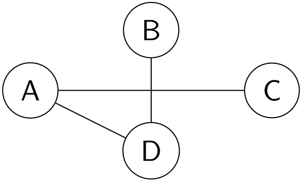
Explanation:

- Graph G:
  - Vertices: $\{A, B, C, D\}$.
  - Edges: $\{(A, B), (B, C), (C, D), (B, D)\}$.
- Complement Graph $\overline{G}$:
  - Vertices: $\{A, B, C, D\}$.
  - Edges: $\{(A, C), (A, D)\}$.

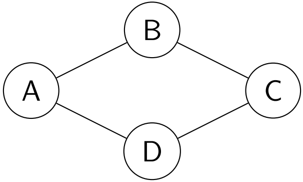
Complement Graph III

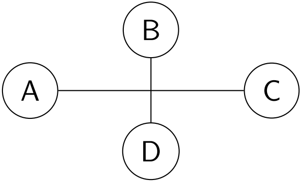
- **Example 2:** Complement of P_4 (Path graph with 4 vertices):





- **Example 3:** Complement of C_4 (Cycle graph with 4 vertices):





Complement Graph IV

- **Additional Properties:**
  - The complement of the complement graph is the original graph:

$$\overline{\overline{G}} = G.$$

  - If G is bipartite, $\overline{G}$ may or may not be bipartite.
  - If G is disconnected, $\overline{G}$ is connected for $|V| \geq 3$.
  - G and $\overline{G}$ cannot share any edges, but they share the same vertex set.
- **Examples:**
  - For K_3 (a complete graph with 3 vertices):

$$\overline{K_3} = K_3^c = \emptyset \quad \text{(no edges)}.$$

  - For a star graph $K_{1,n}$:

$$\overline{K_{1,n}} = K_n \setminus K_{1,n}.$$

Complement Graph V

- Applications of Complement Graphs:
  - Graph Algorithms: Some problems on a graph G can be simplified by studying $\overline{G}$.
  - Independent Sets: The complement graph helps find cliques in the original graph, as:

$$\text{A clique in } G \implies \text{an independent set in } \overline{G}.$$

  - Network Design: Designing complementary networks to optimize connectivity and minimize redundancy.

Exercise:

- Draw the complement graph of C_4 (a cycle with 4 vertices).
- Prove: The complement of a bipartite graph is not necessarily bipartite.
- Prove: If G has E edges, then $\overline{G}$ has $\binom{|V|}{2} - E$ edges.
- Find the complement graph of $K_{2,3}$ and determine its properties.

Graph Connectivity I

Definition: A graph $G = (V, E)$ is connected if there exists a path between every pair of vertices. Otherwise, it is disconnected.
Types of Connectivity:

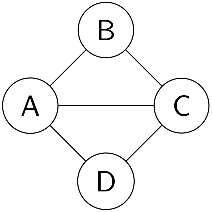
- **Vertex Connectivity ($\kappa(G)$):** The minimum number of vertices that must be removed to disconnect G.
  – If $\kappa(G) \geq k$, then G is k-connected.
- **Edge Connectivity ($\lambda(G)$):** The minimum number of edges that must be removed to disconnect G.
  – If $\lambda(G) \geq k$, then G is k-edge-connected.
- **Strong Connectivity (Directed Graphs):** A directed graph is strongly connected if there exists a directed path between every pair of vertices.
  – If for every $(u, v) \in V$, there is a path from u to v and a path from v to u, then the graph is strongly connected.

Graph Connectivity II

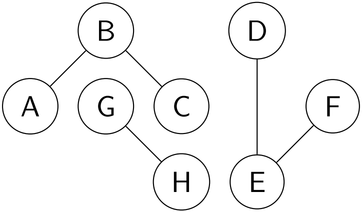
Key Theorems:

- **Menger's Theorem:** For any two non-adjacent vertices in a k-connected graph, there exist at least k vertex-disjoint paths between them.
- The minimum number of vertices needed to separate two vertices equals the maximum number of internally disjoint paths between them.

2-Connected Graph:



Disconnected Graph:



Graph Connectivity III

Graph Connectivity in Real-World Applications:

- Network Resilience: - A highly connected network is fault-tolerant since multiple paths exist between nodes.
- Transportation Systems: - Road and railway networks use connectivity analysis to prevent bottlenecks.
- Biological Networks: - Neuronal and protein interaction networks often rely on graph connectivity properties.

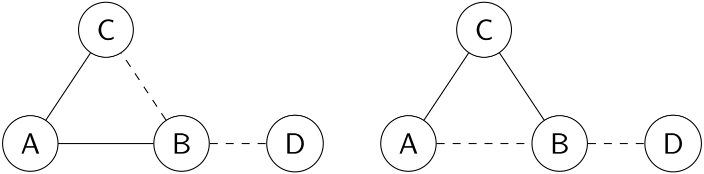
Exercise:

- Compute $\kappa(G)$ and $\lambda(G)$ for K_5.
- Prove that every graph with $\kappa(G) \geq 2$ is 2-connected.
- Find a real-world example where connectivity analysis is important.

Outline

Subgraphs

- A **subgraph** is a graph formed from a subset of the vertices and edges of a larger graph.
- The edges in the subgraph must exist in the original graph.



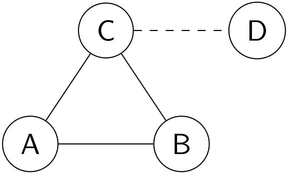
Exercise:

- Identify subgraphs in the given graph.

Question to Ponder: How do subgraphs help in graph decomposition?

Induced Subgraphs

- An **induced subgraph** is formed by a subset of the vertices of a graph and all edges between those vertices that are present in the original graph.
- Induced subgraphs are unique for a given vertex subset.



Exercise:

- Identify the induced subgraph for a given vertex subset.
- Compare it with non-induced subgraphs of the same vertex set.

Question to Ponder: How does the concept of induced subgraphs assist in graph theory proofs?

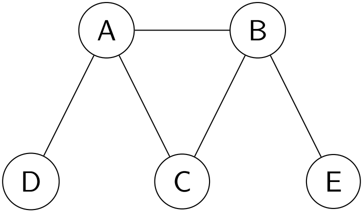
Graph Decomposition

A graph can be decomposed into smaller subgraphs to simplify its analysis and understanding. Decomposition techniques help in identifying the structure and properties of a graph.

- **Definition:** Graph decomposition involves partitioning a graph $G = (V, E)$ into subgraphs that satisfy specific properties.
- **Common Types of Decompositions:**
  - Vertex Decomposition: Partition the vertex set V into subsets.
  - Edge Decomposition: Partition the edge set E into subsets, forming edge-disjoint subgraphs.
  - Subgraph Decomposition: Divide G into subgraphs with specific structures.
- **Applications:**
  - Scheduling: Task assignments with dependencies.
  - Network Design: Subnetwork optimization.
  - Algorithm Design: Dynamic programming on decomposed structures.

Example - Vertex Decomposition

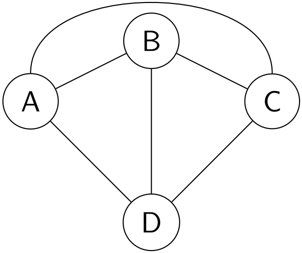
- **Vertex Decomposition:** Partition the vertex set into subsets such that subgraphs induced by these subsets meet certain criteria.
- **Example: Partition into Independent Sets**
  - A graph G can be decomposed into k independent sets $V_1, V_2, \ldots, V_k$.
- **Example Graph:**



- Partition: $V_1 = \{A, E\}, V_2 = \{B, D\}, V_3 = \{C\}$.
- Resulting Subgraphs: Each subset induces a subgraph with no edges.

Example - Edge Decomposition

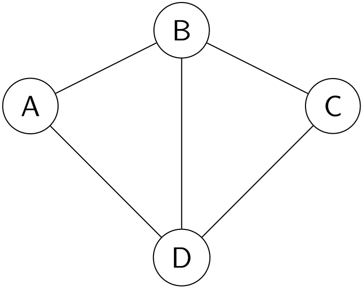
- **Edge Decomposition:** Partition the edge set into disjoint subsets $E_1, E_2, \ldots$ such that each subset forms a specific type of subgraph.
- **Example: Partition into Spanning Trees**
  - A connected graph can be decomposed into edge-disjoint spanning trees.
- **Example Graph:**



- Decompose edges into two spanning trees:
  - $E_1 = \{(A, B), (B, C), (C, D)\}$.
  - $E_2 = \{(D, A), (B, D), (A, C)\}$.

Subgraph Decomposition Example

- **Subgraph Decomposition:** Partition a graph into subgraphs, where each subgraph satisfies a specific property or structure.
- **Example: Partition into Cycles and Paths**
  - Given a graph G, decompose it into edge-disjoint subgraphs, where each subgraph is either a cycle or a path.
- **Example Graph:**



- Decompose G into the following subgraphs:
  1. Cycle: $\{A \to B \to D \to A\}$.
  2. Path: $\{B \to C\}$.

Applications of Graph Decomposition

- Scheduling:
  - Decompose a task dependency graph into levels for parallel processing.
- Network Design:
  - Divide a communication network into subgraphs for efficient routing.
- Algorithm Design:
  - Use tree decompositions to solve problems like vertex cover, maximum clique.
- Real-World Examples:
  - Internet backbone networks.
  - Transportation systems with zone-wise management.
- **Exercise:**
  - Decompose a graph into 2 edge-disjoint spanning trees.

Graph Blocks (Biconnected Components) I

Definition: A block of a graph is a maximal 2-connected subgraph, meaning:

- It has no cut vertices (removing any single vertex does not disconnect it).
- It is maximal (adding any more edges/vertices introduces a cut vertex).

Key Theorems:

- A connected graph can be decomposed uniquely into blocks.
- If a graph has no cut vertices, it is a single block.
- If a graph has at least one cut vertex, it consists of multiple blocks, each connected through a cut vertex.

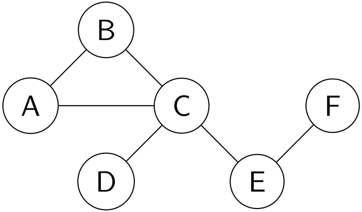
Block-Cut Tree Representation:

- A graph's block-cut tree represents its decomposition into blocks.
- Each block is a node in the tree.

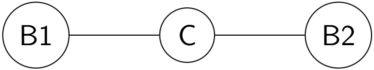
Graph Blocks (Biconnected Components) II

- Cut vertices form the links between these blocks.

Graph with Blocks:



Block-Cut Tree:



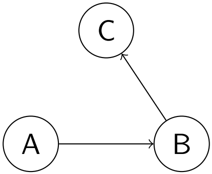
Exercise:

- Compute the block-cut tree for a cycle graph C_6.
- Use Tarjan's algorithm to find the biconnected components in the example graph.

Outline

Directed Graphs

- In a **directed graph** (also referred to as **digraph**), edges have a direction (e.g., $u \to v$).
- Applications: Representing tasks with dependencies.



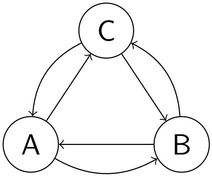
Exercise:

- Identify the in-degree and out-degree of vertices in the graph.

Question to Ponder: How does edge direction impact graph traversal algorithms?

Complete Directed Graph

- A **complete directed graph**, denoted as $\overrightarrow{K_n}$, is a directed graph where every pair of vertices has two directed edges, one in each direction.
- Total number of edges: $n(n-1)$ for n vertices.



Exercise:

- Calculate the number of edges in $\overrightarrow{K_4}$.

Question to Ponder: What real-world systems resemble complete directed graphs?

Directed Graphs: In-Degree and Out-Degree

- **In-Degree** ($d^-(v)$): The number of edges (arcs) entering a vertex.
- **Out-Degree** ($d^+(v)$): The number of edges (arcs) leaving a vertex.

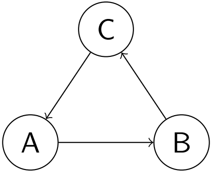


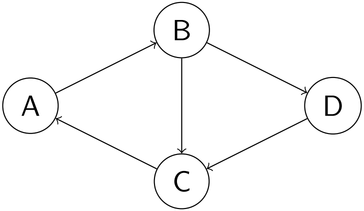
Figure: In-Degree and Out-Degree: $d^-(A) = 1$, $d^+(A) = 1$, $d^-(B) = 1$, $d^+(B) = 1$, $d^-(C) = 1$, $d^+(C) = 1$

Strongly Connected Digraphs I

- **Definition:** A directed graph (digraph) $G = (V, E)$ is strongly connected if, for every pair of vertices $u, v \in V$:

  There exists a directed path from u to v and from v to u.

- **Properties:**
  - Strong connectivity implies that every vertex can reach every other vertex in both directions.
  - The graph remains strongly connected if any strongly connected component is replaced by a single vertex.
- **Example:**

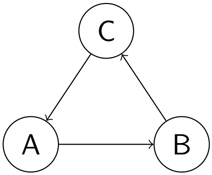


Strongly Connected Digraphs II

- In the above digraph:
  - There is a directed path from every vertex to every other vertex.
  - Therefore, the graph is strongly connected.
- **Exercise:**
  - Verify strong connectivity for the example above.
  - Provide an example of a digraph that is not strongly connected and explain why.

Orientation and Tournaments

- **Orientation**: Assigning a direction to edges in an undirected graph.
- **Tournament**: A directed graph where every pair of vertices is connected by a single directed edge.



Exercise:

- Verify if the given graph is a tournament.

Question to Ponder: What are the applications of tournament graphs?

Outline

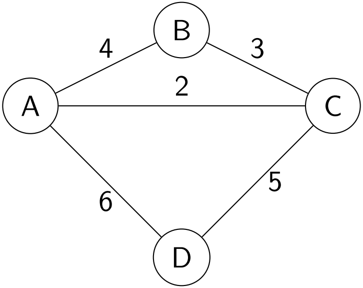
Weighted Graphs I

- **Definition:** A weighted graph is a graph where each edge has a numerical value (weight) associated with it. A **weighted graph** is a mathematical structure consisting of:
  - A set of **vertices**, denoted by $V = \{v_1, v_2, \ldots, v_n\}$.
  - A set of **edges**, denoted by $E = \{e_1, e_2, \ldots, e_m\}$, where each edge e_i is a pair of vertices (v_i, v_j).
  - A **weight function** $w : E \to \mathbb{R}$, which assigns a real number (called the **weight**) to each edge.

  The weighted graph is often denoted by the triple $G = (V, E, w)$.
- **Applications:**
  - **Transportation Networks**: Travel distances or times between locations.
  - **Communication Networks**: Data transmission costs.
  - **Logistics**: Supply chain optimization.

Weighted Graphs II

Example:



Exercise:

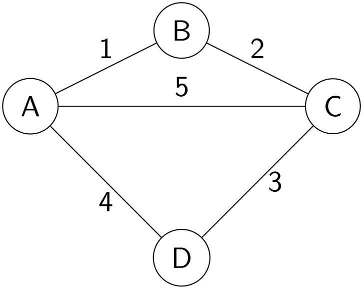
- Represent the above graph as an adjacency matrix with weights.
- Identify a practical scenario modeled by this graph.
- Create the mathematical definition equivalent for the above graph.

Dijkstra's Algorithm I

- **Goal:** Find the shortest path from a source vertex to all other vertices in a weighted graph.
- **Steps:**
  1. Initialize distances to infinity and the source distance to 0.
  2. Use a priority queue to repeatedly select the vertex with the smallest distance.
  3. Update distances to adjacent vertices.
  4. Repeat until all vertices are visited.

Dijkstra's Algorithm II

Example:



Exercise:

- Apply Dijkstra's algorithm to find the shortest paths from vertex A.

Bellman-Ford Algorithm I

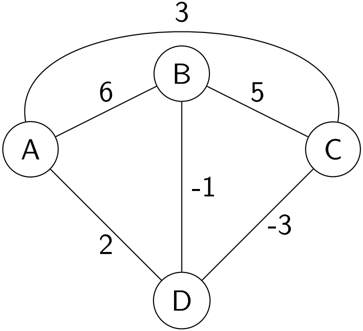
- Goal: Compute shortest paths from a source to all vertices, even with negative edge weights.
- Steps:
  1. Initialize distances to infinity, source to 0.
  2. Relax all edges $V - 1$ times:

$$\text{If } d[u] + w(u, v) < d[v], \text{ then } d[v] = d[u] + w(u, v).$$

  3. Check for negative cycles on the Vth iteration.
- Complexity:
  - Time: $O(VE)$.
  - Space: $O(V)$.

Bellman-Ford Algorithm II

- Exercise:
  - Apply Bellman-Ford to the weighted graph below:



Complexity Analysis of Shortest Path Algorithms

- Dijkstra's Algorithm:
  - Using Priority Queue: $O((V + E) \log V)$.
  - Without Priority Queue: $O(V^2)$.
  - Limitation: Cannot handle negative weights.
- Bellman-Ford Algorithm:
  - Time Complexity: $O(VE)$.
  - Use Case: Handles negative weights and detects negative cycles.
- Comparison:

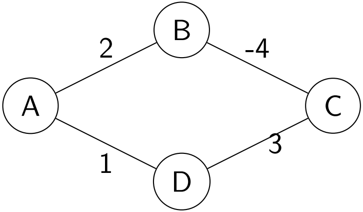
Algorithm	**Time Complexity**	**Handles Negative Weights**
Dijkstra's	$O((V + E) \log V)$	No
Bellman-Ford	$O(VE)$	Yes

Applications of Weighted Graphs

- Logistics:
  - Optimize delivery routes by minimizing transportation costs.
  - Example: Shortest paths in a road network for package delivery.
- Navigation Systems:
  - Google Maps and GPS use weighted graphs to find shortest routes.
  - Weights represent distances or travel times.
- Network Design:
  - Design cost-effective communication networks.
  - Example: Reducing data transmission costs in telecommunications.
- Exercise:
  - Model a real-world problem (e.g., warehouse logistics) as a weighted graph.

Negative Edge Weights

- Definition:
  - Negative edge weights represent costs, losses, or reductions.
- Challenges:
  - Dijkstra's algorithm fails with negative weights.
  - Negative cycles can result in infinite reductions.
- Solution: Bellman-Ford Algorithm
  - Handles graphs with negative weights.
  - Detects negative cycles.
- Example:



- Exercise:
  - Identify if the above graph contains a negative cycle.

Outline

Introduction to Counting and Bijections

Why is Counting Important in Graph Theory?

- Counting methods help analyze the **number of possible graphs**, **spanning trees**, **matchings**, and **paths** in a given structure.
- Many combinatorial proofs in graph theory use **bijections** to establish equivalences between different sets.
- Applications include **enumerative combinatorics**, **graph isomorphism**, **network design**, and **probabilistic graph theory**.

Bijections in Graph Theory:

- A bijection is a **one-to-one and onto mapping** between two sets.
- Establishing bijections helps in counting problems by reducing complexity.
- Example: Counting the number of spanning trees in a complete graph using **Cayley's Formula**.

Exercise:

- Find a bijection between the set of paths in a graph and the set of subgraphs of a given structure.

Basic Counting Principles in Graphs

Fundamental Counting Principles:

- **Addition Principle:** If event A can occur in m ways and event B in n ways (mutually exclusive), total ways $= m + n$.
- **Multiplication Principle:** If task A has m choices and task B has n choices (independent), total ways $= m \times n$.

Examples in Graph Theory:

- Number of labeled graphs with n vertices $= 2^{\binom{n}{2}}$.
- Number of different spanning trees in K_n (Cayley's Theorem) $= n^{n-2}$.
- Counting paths and cycles in different graph structures.

Exercise:

- How many simple graphs can be formed with 4 vertices?
- Prove that the number of labeled trees with 5 vertices is $5^3 = 125$.

Bijections and Counting Spanning Trees

Cayley's Theorem: Counting Trees

- The number of spanning trees of a complete graph K_n is given by:

$$T(K_n) = n^{n-2}$$

- Proof Idea: Use **Prüfer codes**, which establish a bijection between labeled trees and sequences of length $n-2$ over n.

Bijection Between Prüfer Sequences and Labeled Trees:

- Each labeled tree corresponds uniquely to an $(n-2)$-length Prüfer sequence.
- This allows a counting argument using the **multiplication principle**.

Exercise:

- Construct the Prüfer sequence for some tree.

Counting Paths and Cycles in Graphs

Path and Cycle Counting in Graphs:

- Number of paths of length k in a graph can be found using **matrix exponentiation**:

$$A^k(i,j) = \text{number of paths of length } k \text{ from } i \text{ to } j.$$

- Counting cycles in graphs is difficult, but known results include:
  - Number of Hamiltonian cycles in $K_n = (n-1)!/2$.
  - Counting cycles using **generating functions**.

Applications:

- Finding paths efficiently in **network routing**.
- Counting cycles in **circuit design and chemistry** (e.g., ring structures in molecules).

Exercise:

- Compute A^3 for the adjacency matrix of some graph.

Key Theorems and Proofs in Counting and Bijections

Theorem 1: Number of Spanning Trees in a Graph (Kirchhoff's Matrix-Tree Theorem)

- Let L be the Laplacian matrix of a graph.
- The number of spanning trees is given by:

$$T(G) = \text{any cofactor of } L.$$

Theorem 2: Counting Eulerian Circuits

- The number of Eulerian circuits in a graph is given by a determinant formula based on **BEST theorem**.
- **Proof Idea: Using Linear Algebra:** The determinant of a **reduced Laplacian matrix** gives the number of spanning trees.

Exercise:

- Compute the number of spanning trees for K_4 using Kirchhoff's theorem.

Introduction to Extremal Graph Theory

What is Extremal Graph Theory?

- Extremal graph theory studies the **maximum or minimum** values of graph properties under given constraints.
- It asks questions like:
  - What is the **largest** number of edges a graph can have while avoiding a specific subgraph?
  - What is the **smallest** number of edges required for a certain property to hold?

Famous Examples of Extremal Problems:

- **Turán's Theorem:** What is the maximum number of edges in a K_{r+1}-free graph?
- **Erdős-Stone Theorem:** Generalizing Turán's theorem for arbitrary forbidden subgraphs.
- **Mantel's Theorem:** Maximum edges in a triangle-free graph.

Exercise:

- Find the maximum number of edges in a bipartite graph with n vertices.
- Prove that any connected graph with n vertices must have at least $n - 1$ edges.

Turán's Theorem I

Statement: The maximum number of edges in an n-vertex graph that does not contain a complete subgraph K_{r+1} is:

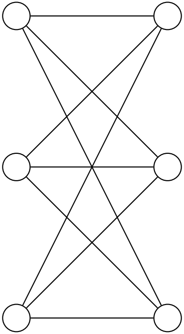
$$\text{ex}(n, K_{r+1}) = \left(1 - \frac{1}{r}\right)\frac{n^2}{2}.$$

Proof Idea:

- Construct the **Turán graph** $T_r(n)$, an r-**partite complete graph** with partitions of nearly equal size.
- Show that adding any extra edge introduces a K_{r+1}.
- Use combinatorial counting to verify the edge bound.

Turán's Theorem II

Visual Example:



Exercise:

- Construct $T_3(6)$ and count its edges.
- Prove that the Turán graph $T_r(n)$ minimizes the number of edges needed to contain a K_{r+1}.

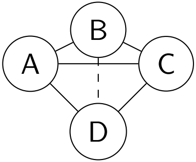
Mantel's Theorem (Triangle-Free Graphs)

Statement: A triangle-free graph with n vertices has at most $\lfloor n^2/4 \rfloor$ edges.

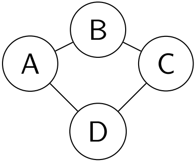
Proof (Extremal Argument):

- Consider a bipartite graph with equal partition sizes.
- Any triangle-free graph with maximum edges must be bipartite.
- The largest bipartite graph is **complete bipartite** $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$, which has $\lfloor n^2/4 \rfloor$ edges.

General Graph (with triangles)



Maximal Triangle-Free Graph



Exercise:

- Prove Mantel's Theorem using an adjacency matrix approach.
- Show that a maximal triangle-free graph must be bipartite.

Real-World Applications of Extremal Problems

Where Are Extremal Graphs Used?

- **Network Design:** Ensuring robustness while minimizing resource usage.
- **Biology:** Studying evolutionary relationships using extremal trees.
- **Coding Theory:** Help in error correction code design.
- **Social Networks:** Modeling influence networks with extremal properties.

Example: Internet Backbone Network

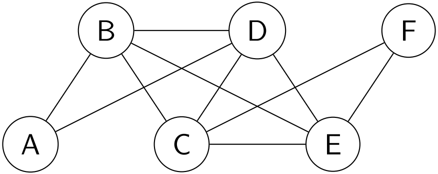
- Designing an internet routing graph with **maximum connectivity** while minimizing redundancy.
- Ensuring the **minimum number of links** required for a stable network.

Exercise:

- Find a real-world scenario where limiting triangle formation is beneficial.
- Research how extremal graph theory helps in wireless sensor network optimization.

Eulerian Graphs

- A graph is **Eulerian** if it contains an Eulerian cycle (a cycle visiting every edge exactly once).
- Necessary and Sufficient condition: All vertices have an even degree.



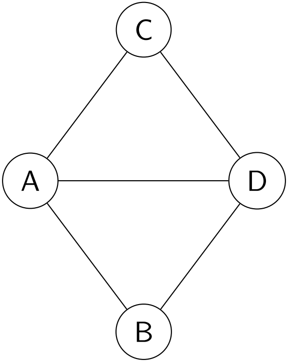
Exercise:

- Determine if a given graph is Eulerian.

Question to Ponder: Can a graph with an odd-degree vertex be Eulerian?

Koenigsberg Bridges Problem I

- The Koenigsberg Bridge Problem (posed in 1736) is a historical problem in graph theory.
- Goal: To determine if it is possible to traverse all seven bridges in the city of Koenigsberg exactly once and return to the starting point.



Koenigsberg Bridges Problem II

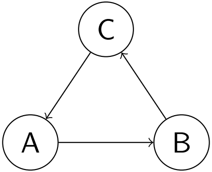
Key Insight: Euler proved it is impossible to traverse the graph in such a way, as all vertices in the graph have an odd degree.
Exercise:

- Analyze the graph to identify the vertex degrees.
- Prove why it is not Eulerian.

Question to Ponder: How did this problem shape the foundations of graph theory?

Eulerian Digraphs

- A **directed graph** is Eulerian if it has a directed Eulerian cycle.
- Necessary and Sufficient condition: In-degree equals out-degree for every vertex.



Exercise:

- Verify the Eulerian property for the above graph.

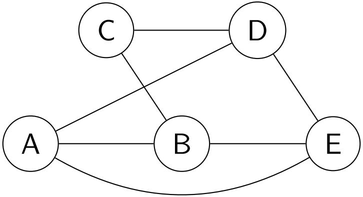
Question to Ponder: How does the Eulerian condition differ in directed graphs?

Hamiltonian Graphs I

- A **Hamiltonian graph** contains a Hamiltonian cycle, which is a cycle that visits each vertex exactly once and returns to the starting vertex.
- Not all graphs are Hamiltonian.
- Sufficient conditions for a graph to be Hamiltonian:
  - **Dirac's Theorem:** If a graph with n vertices ($n \geq 3$) has $\deg(v) \geq n/2$ for all vertices v, it is Hamiltonian.
  - **Ore's Theorem:** If $\deg(u) + \deg(v) \geq n$ for all non-adjacent vertices u and v, the graph is Hamiltonian.

Hamiltonian Graphs II

Example:



Exercise:

- Determine if the given graph satisfies Dirac's or Ore's condition.
- Verify if a complete graph K_n is Hamiltonian.

Question to Ponder: How do Hamiltonian graphs differ from Eulerian graphs in terms of edge and vertex traversal?

Comprehensive Summary I

- **Basic Concepts:**
  - Graph types: Simple, directed, undirected, complete, bipartite, etc.
  - Graph properties: Degrees, adjacency, connectivity.
  - Graph representations: Adjacency matrix, adjacency list.
- **Core Topics:**
  - Paths, cycles, walks: Definitions and examples.
  - Subgraphs, cut vertices, cut edges: Importance in connectivity.
  - Degree sequences: Understanding graph properties.
  - Graph operations: Union, intersection, and complement.
- **Advanced Topics:**
  - Weighted graphs:
    - ★ Definition and real-world applications (logistics, navigation, networks).
    - ★ Adjacency matrix for weighted graphs.
  - Shortest path algorithms:
    - ★ Dijkstra's algorithm: Fast and efficient for positive weights.
    - ★ Bellman-Ford algorithm: Handles negative weights and detects cycles.

Comprehensive Summary II

- **Additional Insights:**
  - Complexity analysis of algorithms:
    - ★ Dijkstra's: $O((V+E)\log V)$.
    - ★ Bellman-Ford: $O(VE)$.
  - Limitations of negative weights in shortest path problems.
- **Key Takeaways:**
  - Graph theory provides powerful tools to model and solve real-world problems.
  - Weighted graphs and their algorithms are foundational for optimization.
  - Algorithm selection depends on graph properties, such as the presence of negative weights.

Outline

Outline

8 Appendix
- Matrices
- DAG
- More on Graph Types
- More Properties
- More Graph Terminologies
- Shortest Paths
- Handy Proofs and Results
- More Problems to Explore

Adjacency Matrix and Incidence Matrix I

- **Adjacency Matrix:**
  - A square matrix A of size $n \times n$, where $n = |V|$ (number of vertices).
  - The entry $A[i][j]$ represents the number of edges between vertices v_i and v_j:

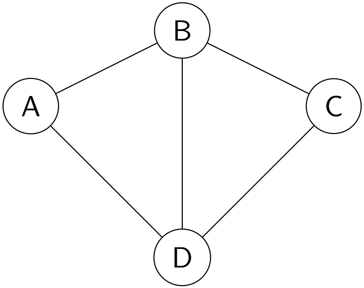
$$A[i][j] = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \text{ (undirected);} \\ 0, & \text{otherwise.} \end{cases}$$

  - **Properties:**
    - ★ For undirected graphs: A is symmetric.
    - ★ For simple graphs: Diagonal entries $A[i][i] = 0$.
    - ★ For weighted graphs: $A[i][j]$ stores the weight of edge (v_i, v_j).
- **Incidence Matrix:**
  - A matrix I of size $n \times m$, where $n = |V|$ and $m = |E|$.
  - The entry $I[i][j]$ indicates the relationship between vertex v_i and edge e_j:

$$I[i][j] = \begin{cases} 1, & \text{if } v_i \text{ is incident to } e_j \text{ (directed start);} \\ -1, & \text{if } v_i \text{ is incident to } e_j \text{ (directed end);} \\ 0, & \text{otherwise.} \end{cases}$$

Adjacency Matrix and Incidence Matrix II

- **Properties:**
  - ⋆ For undirected graphs: $I[i][j] = 1$ for all vertices incident to e_j.
  - ⋆ Sum of entries in each column equals 0 for directed graphs.



Adjacency Matrix:

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

Incidence Matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Adjacency Matrix and Incidence Matrix III

Exercise:

- Construct the adjacency and incidence matrices for a complete graph K_4.
- Prove: The sum of each row in the adjacency matrix equals the degree of the corresponding vertex.

Path Matrix I

- **Definition:** A path matrix P is an $n \times n$ matrix where $n = |V|$ (number of vertices).
  – The entry $P[i][j]$ represents whether there is a path from vertex v_i to vertex v_j:

$$P[i][j] = \begin{cases} 1, & \text{if there exists a path from } v_i \text{ to } v_j; \\ 0, & \text{otherwise.} \end{cases}$$

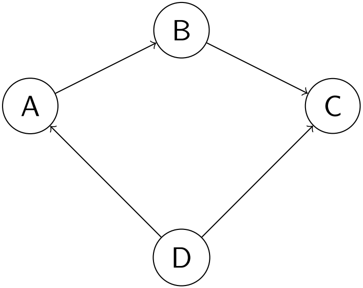
- **Construction:**
  - Compute the powers of the adjacency matrix A:

    $A^k[i][j]$ represents the number of paths of length k from v_i to v_j.

  - The path matrix P is obtained by summing the binary forms of $A, A^2, A^3, \ldots$:

$$P[i][j] = 1 \text{ if } (A + A^2 + \cdots + A^n)[i][j] > 0.$$

Path Matrix II



Path Matrix:

$$P = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Exercise:

- Compute the path matrix for a graph with $|V| = 3$ and edges: $\{(A, B), (B, C)\}$.
- Prove that for a strongly connected graph, $P[i][j] = 1$ for all i, j.

Adjacency Matrix for Directed and Weighted Graphs I

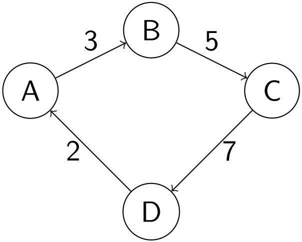
- **Directed Graphs:**
  - In a directed graph, $A[i][j] = 1$ if there is a directed edge from v_i to v_j, otherwise $A[i][j] = 0$.
  - The adjacency matrix is generally not symmetric.
- **Weighted Graphs:**
  - In a weighted graph, $A[i][j]$ represents the weight of the edge (v_i, v_j):

$$A[i][j] = \begin{cases} w, & \text{if edge } (v_i, v_j) \text{ exists with weight } w; \\ 0, & \text{otherwise.} \end{cases}$$

  - For directed weighted graphs, weights depend on edge direction.

Adjacency Matrix for Directed and Weighted Graphs II

Graph:



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 7 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

Exercise:

- Compute the adjacency matrix for a directed acyclic graph (DAG) with 4 vertices.
- Verify that the adjacency matrix for a directed graph is not symmetric unless all edges are bidirectional.

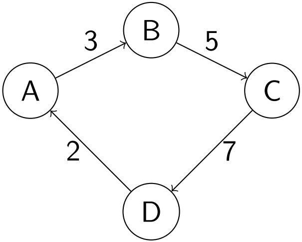
Incidence Matrix for Directed and Weighted Graphs I

- **Directed Graphs:**
  - In a directed graph, $I[i][j] = 1$ if vertex v_i is the start of edge e_j, and $I[i][j] = -1$ if v_i is the end of edge e_j.
  - All other entries are 0.
- **Weighted Graphs:**
  - For weighted graphs, the weight of edge e_j is included:

$$I[i][j] = \begin{cases} w, & \text{if } v_i \text{ is the start of } e_j; \\ -w, & \text{if } v_i \text{ is the end of } e_j; \\ 0, & \text{otherwise.} \end{cases}$$

Incidence Matrix for Directed and Weighted Graphs II

Graph:



Incidence Matrix:

$$I = \begin{bmatrix} 3 & 0 & 0 & -2 \\ -3 & 5 & 0 & 0 \\ 0 & -5 & 7 & 0 \\ 0 & 0 & -7 & 2 \end{bmatrix}$$

Exercise:

- Construct the incidence matrix for a directed cycle graph with 3 vertices.
- Prove that the sum of each column in the incidence matrix equals zero for directed graphs.

Path Matrix for Directed and Weighted Graphs

- **Directed Graphs:**
  - $P[i][j] = 1$ if there exists a directed path from vertex v_i to v_j; otherwise, $P[i][j] = 0$.
- **Weighted Graphs:**
  - $P[i][j]$ stores the weight of the shortest path between v_i and v_j:

$$P[i][j] = \begin{cases} \text{min weight of paths from } v_i \text{ to } v_j, & \text{if path exists;} \\ \infty, & \text{otherwise.} \end{cases}$$

- **Applications:**
  - Determining connectivity in directed graphs.
  - Computing shortest paths in weighted graphs (Floyd-Warshall Algorithm).

Exercise:

- Compute the path matrix for a weighted graph with weights $\{3, 5, 7, 2\}$ as shown earlier.
- Verify the Floyd-Warshall Algorithm to compute the shortest path matrix for a directed graph.

Other Graph Matrices

- **Distance Matrix:**
  - A matrix D where $D[i][j]$ represents the shortest distance (number of edges or weights) between vertices v_i and v_j.
  - For unconnected vertices, $D[i][j] = \infty$.
- **Laplacian Matrix:**
  - A matrix $L = D - A$, where D is the degree matrix (diagonal entries are vertex degrees) and A is the adjacency matrix.
  - Used in spectral graph theory to study properties like connectivity.
- **Transition Probability Matrix (for Random Walks):**
  - A matrix P where $P[i][j]$ represents the probability of transitioning from vertex v_i to v_j in a random walk.
  - Entries: $P[i][j] = \frac{1}{\deg(v_i)}$ if $(i, j) \in E$; otherwise, 0.

Exercise:

- Construct the Laplacian matrix for a star graph $K_{1,4}$.
- Compute the distance matrix for a path graph P_4.

Outline

Directed Acyclic Graph (DAG) I

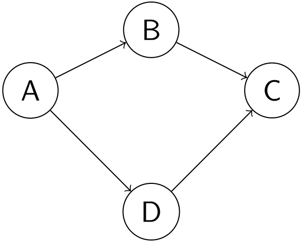
- **Definition:** A Directed Acyclic Graph (DAG) is a directed graph with no directed cycles.

$$G = (V, E) \quad \text{where no sequence of edges forms a cycle.}$$

- **Properties:**
  - DAGs have a topological ordering of vertices: For every edge (u, v), vertex u appears before v in the ordering.
  - Every DAG has at least one source (a vertex with in-degree 0) and one sink (a vertex with out-degree 0).
  - DAGs are used to model dependencies, where cycles are not allowed.
- **Applications:**
  - Task Scheduling: Representing tasks with dependencies (e.g., job scheduling, build systems).
  - Dataflow Analysis: Representing computation pipelines.
  - Shortest Path in Weighted Graphs: DAGs allow efficient shortest path computation using topological sorting.

Directed Acyclic Graph (DAG) II

DAG Visualization:



Key Observations:

- Sources: A (in-degree 0).
- Sinks: C (out-degree 0).
- Topological Order: $A \to B \to D \to C$.

Exercise:

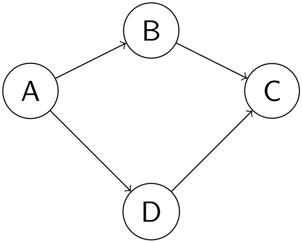
- Verify that the given graph is a DAG.
- Construct the topological ordering for the graph.
- Prove that a graph with a cycle cannot be a DAG.

Topological Sorting: DFS-Based Algorithm I

- **Purpose:** Produces a topological ordering of vertices in a DAG using Depth-First Search (DFS).
- **Algorithm Steps:**
  1. Initialize an empty stack S and mark all vertices as unvisited.
  2. For each vertex v: If v is unvisited, perform a DFS starting from v:
     1. Mark v as visited.
     2. For each neighbor u of v, recursively perform DFS if u is unvisited.
     3. Push v onto S after processing all its neighbors.
  3. Once all vertices are visited, the stack S contains the topological order in reverse.
- Time Complexity: $O(V + E)$.

Topological Sorting: DFS-Based Algorithm II

DAG:



Steps:

1. Start DFS at A: Process B, D, C recursively.
2. Push vertices to stack after processing: Stack $= [C, D, B, A]$.
3. Topological Order: $A \to B \to D \to C$.

Exercise:

- Implement the DFS-based algorithm for a DAG with 5 vertices.
- Prove that the algorithm outputs a valid topological order for all DAGs.

Outline

Multigraph and Pseudograph I

- Graph theory extends beyond simple graphs to include structures like **pseudographs** and **multigraphs**.
- These generalized graphs allow loops and multiple edges between the same pair of vertices.

Definition of a Multigraph

- A **multigraph** is a graph that allows multiple edges (parallel edges) between the same pair of vertices.
- However, it does not allow self-loops.

Definition of a Pseudograph

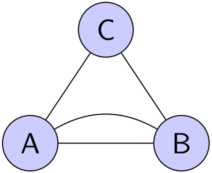
- A **pseudograph** is a generalization of a multigraph that allows both multiple edges and self-loops.
- Self-loops are edges that connect a vertex to itself.

Multigraph and Pseudograph II

Comparison: Multigraph vs. Pseudograph

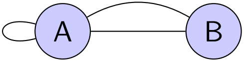
- **Multigraph:** Multiple edges allowed, but no self-loops.
- **Pseudograph:** Multiple edges and self-loops allowed.
- Both are useful in modeling real-world networks with redundant connections.

Multigraph Example:



The above diagram represents a multigraph with multiple edges between vertices.

Pseudograph Example:



The above diagram represents a pseudograph with a self-loop at vertex A.

Multigraph and Pseudograph III

Exercises

- Draw a multigraph with four vertices and at least one pair of parallel edges.
- Construct a pseudograph with three vertices, at least one self-loop, and one pair of parallel edges.
- Identify real-world scenarios where multigraphs and pseudographs are useful.

Conclusion

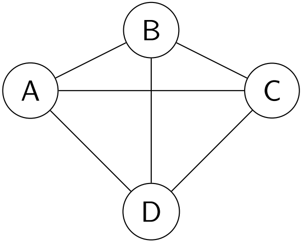
- Understanding these generalized graphs expands the applicability of graph theory.
- They provide more flexibility in modeling complex systems with repeated interactions.

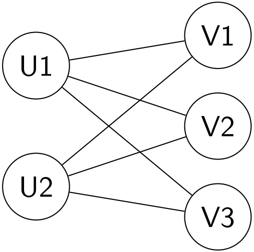
Types of Complete Graphs I

- **Definition:** A complete graph K_n is a simple graph in which every pair of distinct vertices is connected by a unique edge.
- **Types of Complete Graphs:**
  - **Undirected Complete Graph K_n:**
    - ★ Contains n vertices and $\binom{n}{2} = \frac{n(n-1)}{2}$ edges.
    - ★ All edges are bidirectional.
  - **Directed Complete Graph (Tournament Graph):**
    - ★ Every pair of distinct vertices is connected by two directed edges (one in each direction).
    - ★ Contains $n(n-1)$ edges.
  - **Complete Bipartite Graph $K_{m,n}$:**
    - ★ Bipartite graph where every vertex in set U is connected to every vertex in set V.
    - ★ Contains $m \cdot n$ edges.

Types of Complete Graphs II

- **Examples:**





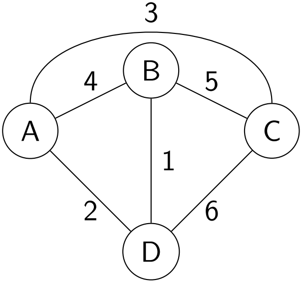
Undirected Complete Graph K_4 Complete Bipartite Graph $K_{2,3}$

- **Exercise:**
  - Calculate the number of edges in K_5 and $K_{3,4}$.
  - Prove: A complete bipartite graph $K_{m,n}$ is 2-colorable.

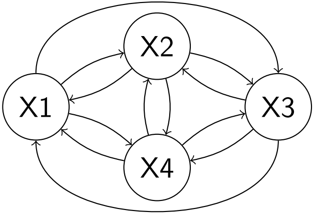
Extended Types of Complete Graphs I

- **Weighted Complete Graph:**
  - A complete graph where each edge is assigned a numerical weight.
  - Used in optimization problems like Traveling Salesman Problem (TSP) and network design.
- **Applications of Complete Graphs:**
  - Communication Networks:
    - ⋆ Fully connected networks where every device communicates with every other device.
  - Optimization Problems:
    - ⋆ Solve TSP to find the shortest route visiting all nodes exactly once.
  - Tournament Scheduling:
    - ⋆ Directed complete graphs model round-robin tournaments where each team plays every other team.

Extended Types of Complete Graphs II



Weighted Complete Graph K_4



Directed Complete Graph $\overrightarrow{K_4}$

- **Exercise:**
  - Compute the total weight of the minimum spanning tree in the weighted graph above.
  - Verify if the directed graph satisfies strong connectivity.

Solving the Traveling Salesman Problem (TSP) I

- **Problem Statement:** Given a weighted complete graph K_n, find the shortest possible route that visits each vertex exactly once and returns to the starting vertex.
- **Mathematical Formulation:** Minimize the total weight of the cycle:

$$\text{Minimize} \sum_{(u,v)\in E} w(u,v) \cdot x_{uv},$$

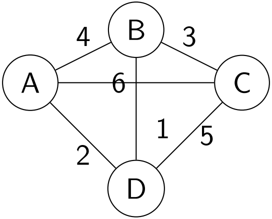
where $x_{uv} = 1$ if edge (u, v) is in the solution, and 0 otherwise.

- **Approximation Algorithms:**
  - Nearest Neighbor Algorithm: Start at a vertex and repeatedly visit the nearest unvisited vertex.
  - Christofides Algorithm: Produces a solution at most $1.5\times$ the optimal solution for metric graphs.

Solving the Traveling Salesman Problem (TSP) II

- **Applications:**
  - Logistics: Optimizing delivery routes.
  - Circuit Design: Minimizing wire lengths in chip design.

Example:



Exercise:

- Use the Nearest Neighbor Algorithm to find an approximate solution for the TSP.
- Verify the total weight of the solution.

Outline

Walks and Paths – Existence of a Path

Theorem: If there is a walk between two vertices u and v, then there is a path between them.

Proof:

- Suppose there exists a walk W from u to v.
- If W is already a path, we are done.
- Otherwise, W contains repeated vertices.
- Remove the segment between the first and second occurrence of any repeated vertex.
- Repeat this process until no vertex appears more than once.
- The resulting sequence is a path.

Corollary: Any two vertices in a connected graph are linked by at least one path.

Real-World Application:

- **Transportation Networks:** If a city map allows reaching one location from another, then a direct, non-redundant route can always be extracted.
- **Internet Routing:** Any redundant communication path can be reduced to a simpler direct route.

Closed Walks and Cycles

Theorem: If there is a closed walk in a graph, then the graph contains a cycle.

Proof:

- Suppose there is a closed walk W that starts and ends at vertex v.
- If W is already a cycle, we are done.
- Otherwise, W contains repeated vertices.
- The sub-walk from the first occurrence of a repeated vertex to its next occurrence forms a cycle.

Corollary: Any graph containing a closed walk also contains at least one cycle.

Real-World Application:

- **Electrical Circuits:** If a current follows a closed walk in a circuit, then some component forms a repeating cycle.
- **Airline Networks:** Any round-trip flight route implies the existence of a repeating flight cycle.

Trails and Eulerian Subgraphs

Theorem: If a graph has a closed trail, then it contains an Eulerian subgraph.

Proof:

- A closed trail visits vertices and edges without repeating edges.
- If a vertex appears more than once, its incident edges form cycles.
- The edge set of any closed trail can be decomposed into a union of cycles.
- Thus, the graph contains an Eulerian subgraph.

Corollary: Any graph with a closed trail must contain a cycle.

Real-World Application:

- **Postal Delivery Routes (Chinese Postman Problem):** If a mail carrier's route follows a closed trail, they must visit each location in a structured cycle.
- **Urban Planning:** Road networks must ensure Eulerian subgraphs exist for traffic flow optimization.

Eulerian Circuits and Degree Conditions

Theorem: A connected graph has an Eulerian circuit if and only if every vertex has even degree.

Proof:

- Suppose a graph has an Eulerian circuit.
- Each visit to a vertex must have a corresponding exit.
- Since edges must be used in pairs, each vertex must have an even degree.

Corollary: Any graph with an Eulerian circuit must be connected and have all even-degree vertices.

Real-World Application:

- **Network Packet Routing:** Ensuring all servers receive equal traffic distribution requires Eulerian circuits.
- **Manufacturing and Robotics:** Automated machines following Eulerian paths minimize repetitive movements.

Hamiltonian Cycles and Spanning Paths

Theorem 5: A Hamiltonian cycle implies multiple spanning paths.
Proof:

- Suppose a graph has a Hamiltonian cycle.
- Any removal of one edge from the cycle results in a spanning path.
- Multiple spanning paths can be constructed by considering different edge deletions.

Corollary: Any Hamiltonian graph has a spanning subgraph with at least one path covering all vertices.
Real-World Application:

- **Traveling Salesperson Problem:** Finding optimal routes that visit all cities efficiently.
- **Genome Sequencing:** Constructing DNA fragment sequences using Hamiltonian paths.

Key Takeaways

Key Results:

- **Walk to Path:** If there is a walk, there is a path.
- **Closed Walk to Cycle:** Every closed walk contains a cycle.
- **Trail to Eulerian Subgraph:** Every closed trail contains an Eulerian subgraph.
- **Eulerian Circuit Condition:** A connected graph has an Eulerian circuit if and only if all vertices have even degree.
- **Hamiltonian Cycle to Paths:** Every Hamiltonian cycle implies multiple spanning paths.

Real-World Applications:

- **Transportation & Routing:** Eulerian and Hamiltonian properties optimize delivery and travel.
- **Data Networks:** Packet routing, redundancy minimization, and internet traffic balancing.
- **Circuit Design:** Electrical networks use Eulerian paths for efficient wiring.
- **AI & Machine Learning:** Graph-based search optimization in AI applications.

Further Study:

- Research extremal graph properties for Hamiltonian and Eulerian graphs.
- Explore computational complexity of finding Eulerian and Hamiltonian paths.

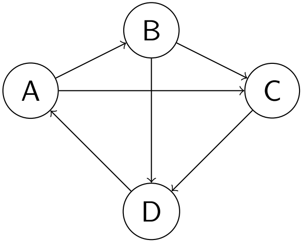
Properties of Tournaments I

- **Definition:** A tournament is a directed graph (digraph) where every pair of vertices is connected by exactly one directed edge. - For every pair of vertices u and v, either $(u \to v) \in E$ or $(v \to u) \in E$.
- **Properties of Tournaments:**
  - **Hamiltonian Path:** Every tournament has at least one Hamiltonian path (a directed path visiting all vertices exactly once).
  - **Strong Connectivity:** A tournament is strongly connected if and only if for every pair of vertices $u, v \in V$, there exists a directed path from u to v and vice versa.
  - **Score Sequence:** The out-degree sequence of vertices (also known as the score sequence) uniquely determines the tournament up to isomorphism.
  - **Transitivity:** A tournament is transitive if there exists an ordering of vertices such that all edges point in the same direction according to the ordering.
  - **Cycles:** Any non-transitive tournament contains directed cycles.

Properties of Tournaments II

- **Special Types of Tournaments:**
  - **Regular Tournament:** A tournament is regular if all vertices have the same in-degree and out-degree.
  - **Strong Tournament:** A tournament is strong if it is strongly connected.

Visualization:



- Hamiltonian Path: $A \to B \to C \to D$.

Properties of Tournaments III

Exercise:

- Prove that every tournament has at least one vertex with out-degree $\lfloor (n-1)/2 \rfloor$.
- Find the number of directed cycles in a tournament with 4 vertices.
- Verify if the example tournament graph is strongly connected.
- Find a Hamiltonian path for the tournament.

Algorithms for Tournaments I

- **Finding a Hamiltonian Path:**
  - Input: A tournament $T = (V, E)$ with $|V| = n$.
  - Output: A Hamiltonian path.
  - Algorithm (Greedy):
    1. Start with any vertex as the first in the path.
    2. Iteratively insert each remaining vertex into the current path:
    3. Place it after the last vertex u if there is a directed edge $u \to v$.
    4. Otherwise, place it before u.
    5. Continue until all vertices are included.
- **Strong Connectivity Check:**
  - Use DFS or BFS from any vertex v:
    - ★ If all vertices are reachable, the tournament is strongly connected.
    - ★ Otherwise, it is not strong.
- **Applications of Algorithms:**
  - Scheduling problems (e.g., round-robin tournaments).
  - Ranking systems (e.g., sports or voting results).

Algorithms for Tournaments II

Exercise:

- Write a Python program to find a Hamiltonian path in a tournament using the greedy algorithm.
- Prove the time complexity of the greedy algorithm for finding a Hamiltonian path is $O(n^2)$.

Applications of Tournaments I

- **Sports Scheduling:**
  - Representing a round-robin tournament where each team plays against every other team exactly once.
  - Directed edges indicate the winner of each match.
- **Ranking Systems:**
  - Modeling pairwise comparisons in voting or ranking systems.
  - Use the Hamiltonian path to infer a ranked order.
- **Decision-Making:**
  - Modeling preferences in decision-making processes.
  - Example: Comparing alternatives in a decision tree.

Applications of Tournaments II

- **Social Networks:**
  - Modeling dominance or influence between individuals.
- **Computational Problems:**
  - Solving problems like finding minimal feedback arc sets to convert tournaments into directed acyclic graphs (DAGs).

Exercise:

- Create a tournament graph for a 6-team round-robin sports league. Indicate the results of matches using directed edges.
- Describe how a Hamiltonian path could be used to rank the teams.

Connected, Disconnected, Strongly Connected Graphs I

- **Connected Graph:** A graph is connected if there is a path between every pair of vertices.

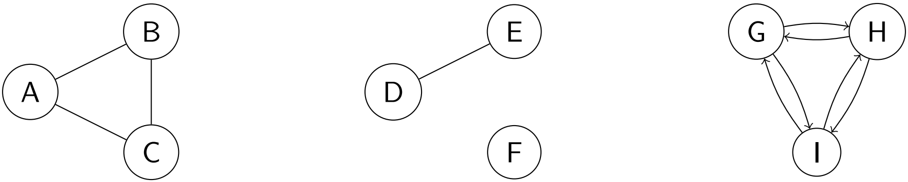
$$\forall u, v \in V, \text{ there exists a path from } u \text{ to } v.$$

- **Disconnected Graph:** A graph is disconnected if it has two or more components (i.e., not all vertices are reachable from each other).
- **Strongly Connected Graph (Directed Graphs):** A directed graph is strongly connected if there is a directed path between every pair of vertices:

$$\forall u, v \in V, \text{ there exists a path } u \to v \text{ and } v \to u.$$

Connected, Disconnected, Strongly Connected Graphs II

Examples:



Exercise:

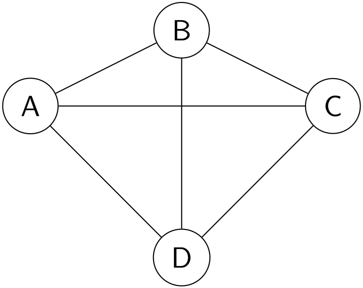
- Determine whether the graphs above are connected, disconnected, or strongly connected.
- Provide a real-world example of each type.

List of Special Graphs

- **Complete Graph (K_n):** Every pair of vertices is connected by an edge. - n vertices, $\binom{n}{2}$ edges.
- **Bipartite Graph:** Vertex set can be divided into two disjoint subsets U and V such that no edge connects vertices within the same subset.
- **Complete Bipartite Graph ($K_{m,n}$):** Every vertex in U is connected to every vertex in V. - $m \cdot n$ edges.
- **Star Graph ($K_{1,n}$):** A complete bipartite graph with one vertex in U and n vertices in V.
- **Cycle Graph (C_n):** A graph that forms a single cycle with n vertices and n edges.
- **Wheel Graph (W_n):** A cycle graph with an additional central vertex connected to all others.
- **Path Graph (P_n):** A graph consisting of a single path with n vertices.
- **Tree:** A connected acyclic graph.

Complete Graphs - Example and Properties

- **Properties of Complete Graphs (K_n):**
  - Number of edges: $\binom{n}{2} = \frac{n(n-1)}{2}$.
  - Chromatic number: n (each vertex requires a unique color).
  - Diameter: 1 (for $n \geq 2$) since every vertex is directly connected.
- **Example (K_4):**

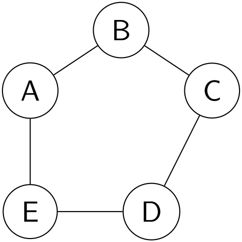


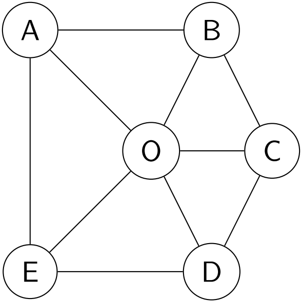
Exercise:

- Draw K_5. Verify the number of edges and chromatic number.

Cycle Graphs and Wheel Graphs

- **Cycle Graph (C_n):**
  - Consists of n vertices and n edges, forming a single cycle.
  - Even cycles are bipartite, odd cycles are not.
- **Wheel Graph (W_n):**
  - Formed by adding a central vertex to a cycle graph and connecting it to all other vertices.
  - Total edges: $2n - 2$.





Exercise:

- Prove that W_n is not bipartite for $n \geq 4$.

Real-World Applications of Special Graphs

- **Complete Graphs:**
  - Social Networks: Modeling complete interaction between individuals.
  - Network Design: Representing fully connected networks for communication.
- **Cycle Graphs:**
  - Traffic Systems: Modeling circular routes in cities.
  - Periodic Processes: Representing cyclic phenomena.
- **Wheel Graphs:**
  - Hub-and-Spoke Networks: Representing airline routes or logistics hubs.
  - Star Topology: Centralized communication networks.
- **Path Graphs:**
  - Linear Pipelines: Modeling linear workflows or transport routes.
- **Bipartite Graphs:**
  - Job Assignment Problems: Matching jobs with workers.
  - Recommendation Systems: Linking users with products.

Graph Isomorphism I

- **Graph Isomorphism**: A bijective function between the vertices of two graphs that preserves the adjacency of vertices.
- Two graphs are **isomorphic** if there exists an isomorphism between them.

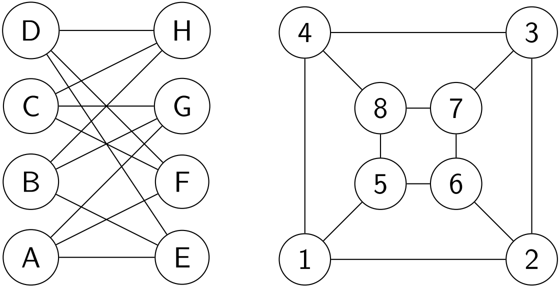


Figure: Isomorphic Graphs

Graph Isomorphism II

Properties of Graph Isomorphism:

- **Reflexive**: A graph is isomorphic to itself.
- **Symmetric**: If G_1 is isomorphic to G_2, then G_2 is isomorphic to G_1.
- **Transitive**: If G_1 is isomorphic to G_2 and G_2 is isomorphic to G_3, then G_1 is isomorphic to G_3.

Graph Isomorphism Theorem:

- Two graphs are isomorphic if and only if they have the same:
  - Number of vertices
  - Number of edges
  - Degree sequence
  - Adjacency matrix

Note: Examples of Graph Isomorphism

- **Same Graph, Different Drawings**: Two graphs that are drawn differently but have the same structure are isomorphic.
- **Renaming Vertices**: Two graphs that differ only in the names of their vertices are isomorphic.

Applications of Complement Graphs I

- **Graph Algorithms:**
  - Problems on cliques and independent sets can be interchanged using the complement graph:

$$\text{Clique in } G \implies \text{Independent Set in } \overline{G}.$$

  - Example: To find the maximum independent set in G, find the largest clique in $\overline{G}$.
- **Network Design:**
  - Use $\overline{G}$ to study alternative connectivity patterns in a network.
  - Example: Minimizing redundant connections by analyzing edges not in G.
- **Graph Coloring:**
  - The chromatic number of $\overline{G}$ can give insights into the graph coloring of G:

$$\chi(G) + \chi(\overline{G}) \geq |V|.$$

Applications of Complement Graphs II

- Real-World Applications:
  - **Social Networks:** $\overline{G}$ represents pairs of individuals who do not share a direct connection.
  - **Logistics:** Complement graphs help identify critical connections by studying the missing edges.

Exercise:

- Prove that for a complete graph K_n, $\overline{K_n} = \emptyset$.
- Show that G and $\overline{G}$ cannot both be disconnected.
- Verify that P_4 and its complement $\overline{P_4}$ together form K_4.
- Prove that for any cycle graph C_n, its complement $\overline{C_n}$ is disconnected when $n > 4$.

Proofs of Complement Graph Properties I

- Property 1: G and $\overline{G}$ together form a complete graph K_n.

Proof.

Let $G = (V, E)$ and $\overline{G} = (V, \overline{E})$. For every pair of vertices $u, v \in V$:

- If $(u, v) \in E$, it is an edge in G.
- If $(u, v) \notin E$, it is an edge in $\overline{G}$.

Since every pair of vertices is either connected in G or in $\overline{G}$:

$$E \cup \overline{E} = E(K_n).$$

Hence, $G \cup \overline{G} = K_n$. □

Proofs of Complement Graph Properties II

- Property 2: The number of edges in $\overline{G}$ is $\binom{|V|}{2} - |E|$.

Proof.

The total number of edges in a complete graph K_n is $\binom{|V|}{2}$. Since G and $\overline{G}$ share no edges:

$$|\overline{E}| = \binom{|V|}{2} - |E|.$$

□

Proofs of Complement Graph Properties III

- Property 3: $\overline{\overline{G}} = G$.

Proof.

By definition, $\overline{E} = \{(u, v) \mid (u, v) \notin E\}$. Taking the complement again gives:

$$\overline{\overline{E}} = \{(u, v) \mid (u, v) \notin \overline{E}\} = E.$$

Hence, $\overline{\overline{G}} = G$. □

Applications of Complement Graphs in Algorithms

- **Clique and Independent Set Problems:**
  - Finding a maximum independent set in G is equivalent to finding a maximum clique in $\overline{G}$.
  - Complement graphs simplify problems in computational graph theory.
- **Planarity Testing:**
  - Complement graphs are used to test if a graph is planar by analyzing edge density.
- **Network Analysis:**
  - Designing complementary networks to explore missing connectivity.
  - Example: Redundant link placement in communication networks.
- **Graph Coloring:**
  - Complement graphs help in studying chromatic properties:

$$\chi(G) + \chi(\overline{G}) \geq |V|.$$

Exercise:

- Prove: If G is a tree, $\overline{G}$ is a disconnected graph.
- Determine if the complement graph of a bipartite graph can have odd-length cycles.

Advanced Counting Techniques in Graph Theory

Why Advanced Counting?

- Traditional counting methods may become inefficient for large graphs.
- Advanced techniques like **generating functions** and **probabilistic counting** help in handling complex structures.
- Applications include **network reliability, molecular chemistry, social network analysis, and combinatorial optimization**.

Topics Covered:

- **Generating functions** for counting labeled and unlabeled graphs.
- **Probabilistic counting methods** in random graphs.
- **Real-world applications** in **network topology, data clustering, and statistical physics**.

Exercise:

- Why is generating function analysis useful in counting spanning trees?
- How does probability help in estimating large combinatorial counts?

Generating Functions for Graph Counting I

Definition: A generating function is a formal power series that encodes combinatorial structures:

$$G(x) = \sum_{n=0}^{\infty} a_n x^n$$

where a_n represents the number of structures of size n.

Application in Graph Theory:

- Counting the number of simple graphs, trees, or subgraphs using coefficient extraction.
- Example: Counting labeled graphs with n vertices:

$$G(x) = \sum_{n=0}^{\infty} \frac{2^{\binom{n}{2}}}{n!} x^n$$

Generating Functions for Graph Counting II

Example: Counting Paths Using Generating Functions

- Let A be the adjacency matrix of a graph.
- The number of paths of length k between vertices i and j is given by:

$$A^k(i,j) = \text{coefficient of } x^k \text{ in } (I - xA)^{-1}.$$

Exercise:

- Find the number of walks of length 3 in a given graph using matrix exponentiation.
- Compute the coefficient of x^3 in the expansion of $(1 - 3x)^{-1}$.

Probabilistic Counting in Graph Theory I

Why Use Probability in Counting?

- Many graphs are too large to count explicitly.
- Probabilistic methods provide **approximate counts** with high accuracy.
- Used in **random graph theory, network models, and large-scale data analysis**.

Example: Counting Large Graphs Using the Erdős–Rényi Model

- A random graph $G(n, p)$ is constructed by including each edge independently with probability p.
- Expected number of edges:

$$E[|E|] = p\binom{n}{2}.$$

- Expected number of spanning trees can be estimated using **Markov's inequality**.

Probabilistic Counting in Graph Theory II

Monte Carlo Estimation for Counting Subgraphs

- Instead of explicit enumeration, use random sampling.
- Example: Estimate the number of triangles in a graph by sampling vertex triples.

Exercise:

- Compute the expected number of Hamiltonian cycles in a random graph $G(n, 1/2)$.
- Design a Monte Carlo algorithm to estimate the number of 4-cycles in a large graph.

Real-World Applications of Counting and Bijections I

Where Do Counting and Bijections Matter?

- **Network Topology Analysis:** Counting the number of possible network configurations.
- **Molecular Chemistry:** Counting chemical isomers using graph enumeration.
- **Statistical Physics:** Modeling states in quantum and lattice systems.
- **Social Networks:** Counting possible connections in dynamic networks.

Example: Chemical Compound Enumeration

- Many molecules can be represented as **graph structures**.
- Counting distinct chemical structures is equivalent to counting **non-isomorphic graphs**.

Real-World Applications of Counting and Bijections II

Example: Counting Data Clustering Methods

- The number of ways to partition a dataset of size n into k clusters is given by **Stirling numbers**.

Exercise:

- Compute the number of different spanning trees in a network of 6 nodes.
- How many unique chemical structures exist for a given molecular formula using graph enumeration?

Erdős-Stone Theorem

Statement: If H is a non-bipartite graph with chromatic number $\chi(H)$, then the maximum number of edges in an H-free graph on n vertices is:

$$\mathrm{ex}(n, H) = \left(1 - \frac{1}{\chi(H) - 1}\right) \frac{n^2}{2} + o(n^2).$$

Significance:

- Generalizes **Turán's Theorem** for arbitrary forbidden subgraphs.
- Asymptotically determines extremal numbers for all non-bipartite graphs.

Proof Idea:

- Extends the idea of **Turán graphs** to any graph H.
- Uses probabilistic methods and the **Regularity Lemma**.
- Shows that large graphs avoiding H must resemble **Turán-type structures**.

Exercise:

- Show that Erdős-Stone theorem reduces to Turán's theorem when $H = K_{r+1}$.
- Explain why bipartite graphs do not follow Erdős-Stone bound.

Ramsey Theory – Finding Order in Chaos I

Statement (Ramsey's Theorem): For any integers r, s, there exists a number $R(r, s)$ such that any edge-coloring of the complete graph K_n with two colors contains:

- A red clique of size r, or
- A blue clique of size s.

Key Properties:

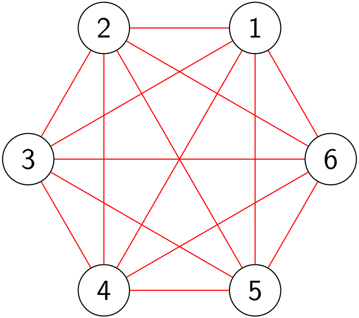
- Ensures that large enough graphs must contain ordered substructures.
- Fundamental to **graph colorings, combinatorics, and logic**.

Small Cases:

- $R(3, 3) = 6$: Every 2-coloring of K_6 has a monochromatic triangle.
- Bounds on $R(4, 4)$: Known to be between **18 and 25**.

Ramsey Theory – Finding Order in Chaos II

Example: Ramsey Number $R(3,3) = 6$



Exercise:

- Prove that $R(3,3) = 6$ using exhaustive cases.
- Research bounds for $R(5,5)$.

Real-World Applications of Extremal Graph Theory I

Applications in Different Fields:

- **Communication Networks:** Avoiding **network congestion** by limiting cliques.
- **Computational Biology:** Analyzing **protein interaction networks**.
- **Social Networks:** Ensuring efficient influence spread while avoiding redundant connections.
- **Data Storage & Compression:** Encoding optimal **error-correcting codes**.

Example: Avoiding Congestion in Wireless Networks

- In wireless communication, avoiding large **fully connected subgraphs** prevents interference.
- Extremal graph theory helps in designing **optimal network structures**.

Real-World Applications of Extremal Graph Theory II

Example: Ramsey Theory in Decision Problems

- In scheduling, extremal problems help determine the **minimum resources needed to avoid conflicts**.
- **Example:** Finding a clique-free graph ensures no subset of jobs requires the same resource.

Exercise:

- How does extremal graph theory help in **parallel computing**?
- What extremal properties are useful in designing **secure cryptographic networks**?

Introduction to Probabilistic Methods in Extremal Graph Theory I

Why Use Probability in Extremal Graph Theory?

- Some extremal problems are too complex for **constructive proofs**.
- The **probabilistic method** helps prove existence results without explicitly constructing an object.
- Often used to **find lower bounds** for extremal graph problems.

Key Idea:

- Construct a random graph and show that it has the desired properties **with positive probability**.
- If such a graph exists with nonzero probability, then at least one such graph must exist.

Introduction to Probabilistic Methods in Extremal Graph Theory II

Examples of Use:

- **Lower bounds on Ramsey numbers**.
- **Existence of graphs with large girth and high chromatic number**.
- **Random constructions for sparse graphs with high independence number**.

Exercise:

- Why is probability useful for proving existence rather than explicit construction?
- Research how randomness helps in designing efficient network topologies.

Erdős's Probabilistic Method I

Core Idea: Erdős introduced a **non-constructive proof technique** where:

- A randomly chosen structure is shown to have a desired property **with positive probability**.
- Since probability is positive, **such a structure must exist**.

Example: Lower Bounds on Ramsey Numbers

- Consider a random graph $G(n, 1/2)$ where each edge is included **independently** with probability $1/2$.
- Expected number of monochromatic K_r cliques in a **2-coloring** is:

$$E(X) \leq \binom{n}{r} 2^{1-\binom{r}{2}}.$$

- For large enough n, $E(X) < 1$, which means there exists at least one coloring without a monochromatic K_r.

Erdős's Probabilistic Method II

Why is This Important?

- Provides **lower bounds** for Ramsey numbers where exact values are unknown.
- Introduces randomness in **graph constructions**, leading to optimal network designs.

Exercise:

- Show that $R(4,4) > 17$ using Erdős's method.
- Research how random graphs help in machine learning and AI applications.

Lower Bounds on Ramsey Numbers Using Probability I

Ramsey's Theorem: Every edge-colored complete graph contains a **monochromatic clique** of a certain size.

Using Probability to Bound Ramsey Numbers:

- Erdős's method provides an **exponential lower bound** on Ramsey numbers:
$$R(r, r) \geq 2^{r/2}.$$
- This shows that Ramsey numbers grow **faster than polynomial functions**, proving why computing exact values is hard.

Proof Sketch:

- Consider a **random 2-coloring** of K_n.
- Compute the probability of a **monochromatic** K_r appearing.
- Show that for large n, such an event occurs with low probability.

Lower Bounds on Ramsey Numbers Using Probability II

Applications:

- **Network robustness** – ensuring redundancy while avoiding excessive connections.
- **Parallel processing** – scheduling large-scale computations to avoid bottlenecks.

Exercise:

- Prove that $R(5, 5) > 43$ using probability.
- Research why exact Ramsey numbers are difficult to compute.

Probabilistic Constructions in Graph Theory I

Key Idea:

- Instead of deterministic methods, construct graphs using **randomized rules**.
- Ensure the desired properties hold **with high probability**.

Example: Sparse Graphs with High Chromatic Number

- Erdős showed that there exist graphs with **large chromatic number** and **arbitrarily large girth**.
- Construction:
  - Start with an empty graph on n vertices.
  - Add edges **randomly** while ensuring no small cycles appear.
  - The resulting graph has a **high chromatic number**, proving its existence.

Probabilistic Constructions in Graph Theory II

Application: Random Graphs in Network Design

- Designing networks with **minimum edge density** while maximizing robustness.
- Using probabilistic models to **simulate real-world social networks**.

Exercise:

- Construct a random graph with $n = 10$ vertices and find its chromatic number.
- Explain why sparse graphs can have arbitrarily large chromatic numbers.

Real-World Applications of Probabilistic Methods

Why Use Random Graphs?

- Many real-world networks are **randomly evolving** (e.g., internet topology, social networks).
- Randomized algorithms **improve efficiency** in large-scale computations.

Applications:

- **Wireless Networks:** Random graphs help model **signal interference** in large-scale wireless networks.
- **Cryptography & Hashing:** Probabilistic methods are used in **randomized hashing techniques** for security.
- **Machine Learning & AI:** Randomized graph models are used in **deep learning architectures**.

Exercise:

- Research how random graphs model **brain neural networks**.
- Design a simple randomized **load-balancing algorithm** for networks.

Outline

Independent Set, Vertex Cover, and Clique I

- **Independent Set:**
  - A set of vertices $S \subseteq V$ in a graph $G = (V, E)$ is independent if no two vertices in S are adjacent.
  - *Maximum Independent Set*: The largest independent set in a graph.
  - The Maximum Independent Set of a graph G is referred to as $MIS(G)$. The independence number (aka adjacency number) of G is defined as $\alpha(G) = |MIS(G)|$.
- **Vertex Cover:**
  - A set of vertices $C \subseteq V$ such that every edge in the graph is incident to at least one vertex in C.
  - *Minimum Vertex Cover*: The smallest vertex cover in a graph.
  - The Minimum Vertex Cover of a graph G is referred to as $MVC(G)$. The size of $MVC(G)$ is denoted as $\beta(G)$, i.e., $\beta(G) = |MVC(G)|$.
- **Clique:**
  - A set of vertices $K \subseteq V$ that induces a complete subgraph.
  - *Maximum Clique*: The largest clique in a graph.
  - The size of the maximum clique of a graph G is denoted as $\omega(G)$.

Independent Set, Vertex Cover, and Clique II

Properties:

- *Independent Set* and *Vertex Cover* Relation:

$$S = V \setminus C,$$

  where S is an *independent set* and C is a *vertex cover*.

- *Independent Set* and *Clique* Relation:

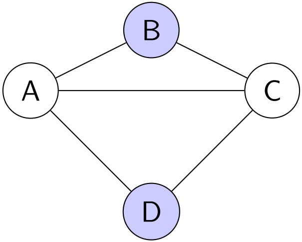
$$S \text{ is an } \textit{independent set} \text{ in } G \iff S \text{ is a } \textit{clique} \text{ in } \overline{G}.$$

Exercise:

- Identify all independent sets, cliques, and vertex covers in the graph above.
- Prove that finding a maximum independent set is NP-complete.

Example and Visualization - Independent Set

- **Definition Recap:** An independent set is a set of vertices in which no two vertices are adjacent.
- **Consider the graph G below:**



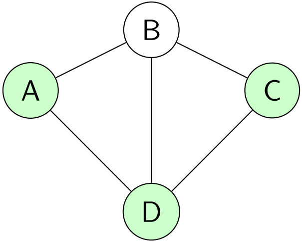
- **Independent Set:** $\mathbb{S} = \{B, D\}$ (highlighted in blue) is an independent set because no edges exist between B and D.

Exercise:

- Identify all independent sets in the graph.
- Prove that no independent set in the graph has more than 2 vertices.

Example and Visualization - Vertex Cover

- **Definition Recap:** A vertex cover is a set of vertices such that every edge in the graph is incident to at least one vertex in the set.
- **Consider the graph G below:**



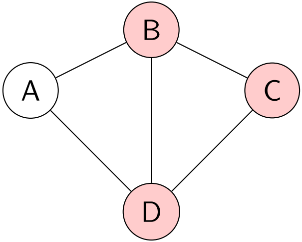
- **Vertex Cover:** $\mathbb{C} = \{A, C, D\}$ (highlighted in green) is a vertex cover because all edges are incident to at least one vertex in $\mathbb{C}$.

Exercise:

- Find the minimum vertex cover for the graph.
- Prove that the size of a vertex cover plus the size of a maximum independent set equals the total number of vertices.

Example and Visualization - Clique

- **Definition Recap:** A clique is a subset of vertices that forms a complete subgraph.
- **Consider the graph G below:**



- **Clique:** $\mathbb{K} = \{B, C, D\}$ (highlighted in red) is a clique because all pairs of vertices are connected.

Exercise:

- Identify all cliques in the graph and find the maximum clique.
- Prove that the size of a maximum clique in a graph is equal to the chromatic number of its complement graph.

Applications of Independent Set, Vertex Cover, and Clique

- **Independent Set Applications:**
  - Wireless Networks: Non-interfering stations in a frequency allocation graph.
  - Scheduling: Selecting non-conflicting tasks in a dependency graph.
- **Vertex Cover Applications:**
  - Network Monitoring: Ensuring every link in a network is monitored.
  - Power Distribution: Selecting substations to cover all transmission lines.
- **Clique Applications:**
  - Social Networks: Detecting tightly connected groups of individuals.
  - Bioinformatics: Identifying highly interacting protein complexes.

Exercise:

- Solve the vertex cover problem for a bipartite graph using the Hopcroft-Karp algorithm.
- Prove that a maximum independent set in a tree can be found in polynomial time.

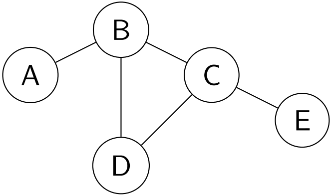
Maximal vs. Maximum & Minimal vs. Minimum I

Key Concepts:

- **Maximum/Minimum:** These are the absolute, global extremes in a set.
  - *Example:* A **maximum independent set** is an independent set of the largest possible size in the graph.
  - *Example:* A **minimum spanning tree** is the spanning tree with the least total weight.
- **Maximal/Minimal:** These are local extremes, meaning they cannot be extended (or reduced) further while preserving the property, though they may not be the best overall.
  - *Example:* A **maximal independent set** is an independent set that cannot have any additional vertex added to it without losing its independence, but it might not have the largest possible number of vertices.
  - *Example:* A **minimal vertex cover** is a vertex cover such that removing any vertex from it would cause it to cease being a vertex cover, yet it might not be the smallest possible vertex cover.

Maximal vs. Maximum & Minimal vs. Minimum II

Visual Illustration:



Independent Sets:

- **Maximal Independent Set:** For instance, $\{A, C\}$ is an independent set that cannot be extended (if adding any other vertex violates independence).
- **Maximum Independent Set:** The largest independent set in this graph is $\{A, D, E\}$.

Note: In this particular drawing, you would need to verify edge incidences; the idea is to illustrate that every maximum independent set is maximal, but a maximal one (like $\{A, C\}$) might be smaller than the absolute maximum.

Maximal vs. Maximum & Minimal vs. Minimum III

Key Observations:

- Every maximum (or minimum) solution is also maximal (or minimal), but not vice versa.
- In optimization problems, "optimal" solutions are those that achieve the global best, while "maximal" or "minimal" can refer to local, irreducible configurations.

Exercise:

- Given a cycle graph C_6, list all maximal independent sets and determine which is maximum.
- In a bipartite graph, distinguish between a maximal matching and a maximum matching.

Outline

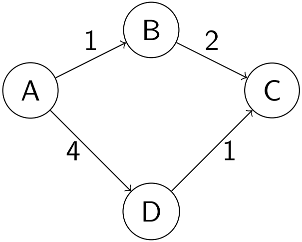
Dijkstra's Algorithm I

- **Purpose:** Computes the shortest path from a single source vertex to all other vertices in a weighted graph.
- **Input:** A graph $G = (V, E)$ with non-negative edge weights and source vertex s.
- **Output:** An array of shortest path distances $d[v]$ for all $v \in V$.
- **Algorithm:**
  1. **Initialize** $d[s] = 0$, and $d[v] = \infty$ for all other vertices.
  2. **Set** all vertices as unvisited. Use a priority queue for efficient edge relaxation.
  3. **While** there are unvisited vertices:
     - ★ **Extract** the vertex u with the smallest $d[u]$.
     - ★ **For each** neighbor v of u, update:

$$d[v] = \min(d[v], d[u] + \text{weight}(u, v)).$$

Dijkstra's Algorithm II

Graph:



Distances from A:

$$d[A] = 0, d[B] = 1, d[C] = 3, d[D] = 4.$$

Exercise:

- Implement Dijkstra's Algorithm for a weighted graph with 5 vertices.
- Explain why Dijkstra's Algorithm fails with negative edge weights.

Bellman-Ford Algorithm I

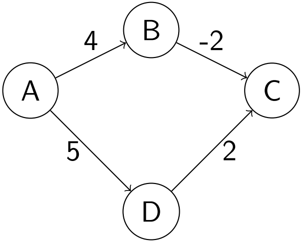
- **Purpose:** Computes shortest paths from a single source vertex to all others, handling negative edge weights.
- **Input:** A graph $G = (V, E)$ with edge weights (may include negative weights) and source vertex s.
- **Output:** An array of shortest path distances $d[v]$ for all $v \in V$. Detects negative weight cycles.
- **Algorithm:**
  1. **Initialize** $d[s] = 0$, and $d[v] = \infty$ for all other vertices.
  2. **Repeat** $|V| - 1$ times:
     - ⋆ For each edge $(u, v) \in E$, **update**:

$$d[v] = \min(d[v], d[u] + \text{weight}(u, v)).$$

  3. **Check for negative cycles:** For each edge $(u, v) \in E$, if $d[v] > d[u] + \text{weight}(u, v)$, report a negative weight cycle.

Bellman-Ford Algorithm II

Graph:



Distances from A:

$$d[A] = 0, d[B] = 4, d[C] = 2, d[D] = 4.$$

Exercise:

- Use Bellman-Ford to compute shortest paths in a graph with 5 vertices and negative weights.
- Prove that Bellman-Ford detects negative weight cycles in $O(|V| \cdot |E|)$ time.

Floyd-Warshall Algorithm I

- **Purpose:** Computes the shortest paths between all pairs of vertices in a weighted graph.
- **Input:** A graph $G = (V, E)$ with adjacency matrix A, where:

$$A[i][j] = \begin{cases} \text{Weight of edge } (i,j), & \text{if } (i,j) \in E; \\ \infty, & \text{if } i \neq j \text{ and } (i,j) \notin E; \\ 0, & \text{if } i = j. \end{cases}$$

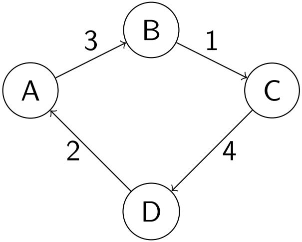
- **Output:** Matrix D, where $D[i][j]$ is the shortest path distance from i to j.
- **Algorithm:**
  1. **Initialize** $D^{(0)} = A$.
  2. **For each** vertex $k \in V$, update:

$$D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\right).$$

  3. **Repeat** for $k = 1, 2, \ldots, n$.

Floyd-Warshall Algorithm II

Graph:



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 3 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 4 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

After applying Floyd-Warshall:

$$D = \begin{bmatrix} 0 & 3 & 4 & 8 \\ 6 & 0 & 1 & 5 \\ 6 & 9 & 0 & 4 \\ 2 & 5 & 6 & 0 \end{bmatrix}$$

Exercise:

- Apply Floyd-Warshall to a triangle graph with weights $\{1, 5, 3\}$.
- Prove that Floyd-Warshall correctly handles negative edge weights (but no negative cycles).

Chinese Postman Problem I

Definition: The Chinese Postman Problem (CPP) seeks the shortest closed path that visits every edge at least once in an undirected graph.

Applications:

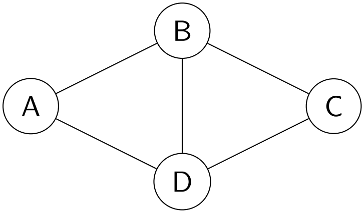
- Mail delivery and garbage collection.
- Road network optimization.
- Logistics and transportation planning.

Algorithm:

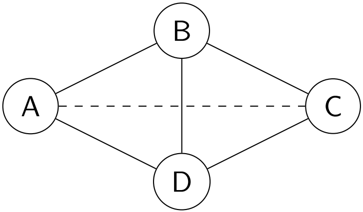
1. Identify vertices with odd degree.
2. Pair these vertices optimally using shortest paths.
3. Add the necessary edges to make all degrees even.
4. Find an Eulerian circuit in the modified graph.

Chinese Postman Problem II

Graph Before Modification:



After Adding Necessary Edges:



Exercise:

- Solve the Chinese Postman Problem for C_6.
- Find a real-world example where the CPP is useful.

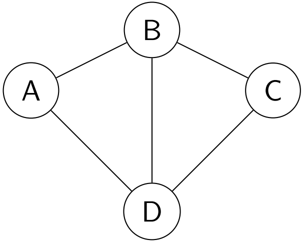
Solving the Chinese Postman Problem I

Definition Recap: The Chinese Postman Problem (CPP) seeks the shortest closed path covering all edges at least once.
Algorithm to Solve CPP:

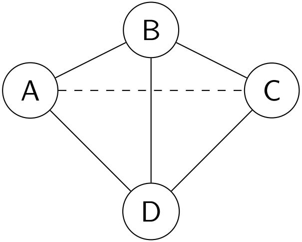
1. **Identify Odd-Degree Vertices:** If all vertices have even degree, an Eulerian circuit exists. Otherwise, find all odd-degree vertices (they must be paired).
2. **Pair Odd-Degree Vertices:** Find the shortest paths between all odd-degree vertices. Compute the optimal pairing to minimize added path length.
3. **Duplicate the Shortest Paths:** Add edges to make all degrees even.
4. **Find an Eulerian Circuit:** Use Fleury's algorithm or Hierholzer's algorithm.

Solving the Chinese Postman Problem II

Original Graph:



After Adding Extra Edges:



Key Observations:

- Odd-degree vertices: $\{A, C\}$ and $\{B, D\}$.
- Shortest paths added: (A, C) to make all degrees even.
- Eulerian circuit now exists.

Exercise:

- Solve the CPP for a small road network graph.
- Find an Eulerian circuit after adding the required edges.

Overview of Shortest Path Algorithms in Graphs I

Algorithm	**Best Suited For**	**Time Complexity**
Dijkstra's	Graphs with non-negative weights	$O((V+E)\log V)$
Bellman-Ford	Graphs with negative weights (but no negative cycles)	$O(VE)$
Floyd-Warshall	Dense graphs, all-pairs shortest paths	$O(V^3)$
Johnson's	Sparse graphs with negative weights (no cycles)	$O(V^2 \log V + VE)$
A* Search	Graphs with heuristic-based pathfinding (e.g., maps)	$O(E)$ (depends on heuristic)
Bidirectional Dijkstra	Large graphs with few goal vertices	$O((V+E)\log V)$ (faster in practice)
Yen's k-Shortest Paths	Finding multiple shortest paths	$O(k(V+E)\log V)$
Thorup's	Planar graphs with positive weights	$O(V)$
BFS (for Unweighted Graphs)	Unweighted graphs (unit weights)	$O(V+E)$
Dial's	Graphs with integer weights in a small range	$O(V+C)$ (where C is max edge weight)
Fringe Search	AI pathfinding (optimized for real-time systems)	$O(E)$ (depends on heuristic)
ALT (A*, Landmarks, Triangle)	Road networks (precomputed heuristics)	$O(E)$ (efficient in practice)
Hirschberg and Larmore's	Special cases in dynamic programming	$O(VE)$

Overview of Shortest Path Algorithms in Graphs II

Brief Reasoning for Suitability:

- **Dijkstra's Algorithm** – Efficient for graphs with non-negative weights due to priority queue optimization.
- **Bellman-Ford Algorithm** – Works even with negative weights by relaxing edges $V - 1$ times.
- **Floyd-Warshall Algorithm** – Best for small, dense graphs since it computes all-pairs shortest paths.
- **Johnson's Algorithm** – Handles negative weights efficiently by re-weighting edges using Bellman-Ford.
- **A* Search Algorithm** – Uses heuristics for directed graphs like road maps, reducing unnecessary searches.
- **Bidirectional Dijkstra** – Searches forward from the source and backward from the target, improving efficiency.
- **Yen's Algorithm** – Used when multiple shortest paths are needed, common in routing problems.

Overview of Shortest Path Algorithms in Graphs III

- **Thorup's Algorithm** – Fastest known algorithm for planar graphs with positive weights.
- **BFS (for Unweighted Graphs)** – Finds shortest path in unit weight graphs in linear time.
- **Dial's Algorithm** – Works well when weights are small integers (bucket-based).
- **Fringe Search** – Optimized version of A* for real-time AI and robotics.
- **ALT Algorithm** – Uses precomputed landmarks for fast route-finding in road networks.
- **Hirschberg and Larmore's Algorithm** – Applied in dynamic programming shortest path cases.

Overview of Shortest Path Algorithms in Graphs IV

Exercise:

- Compare Dijkstra's and Bellman-Ford algorithms on a weighted directed graph.
- Implement Floyd-Warshall for a 6-vertex graph and verify the all-pairs shortest paths.
- Research which algorithm is used in *Google Maps, GPS navigation, or AI game pathfinding*.

Outline

Complement of a Disconnected Graph is Connected

Statement: The complement of a simple disconnected graph must be connected.

Proof Sketch:

- Let G be a simple disconnected graph. Then, G has at least two components.
- For any two vertices u, v in G, there exists no edge uv in G if u and v are in different components.
- In $\overline{G}$, u and v are adjacent because uv is not an edge in G.
- Therefore, every pair of vertices from different components in G is connected in $\overline{G}$, making $\overline{G}$ connected.

Odd Edge Appearance Implies a Cycle

Statement: If edge e appears an odd number of times in a closed walk W, then W contains the edges of a cycle through e.

Proof Sketch:

- Let $e = uv$, and assume e appears $2k + 1$ times in W.
- Each traversal of e contributes to entering and exiting u and v.
- Focus on the first traversal of e and trace the path $u \to v$ without repeating e.
- This subpath forms a cycle containing e since W is closed.

Two Distinct u, v-Paths Contain a Cycle

Statement: If P and Q are two distinct u, v-paths in G, then G contains a cycle.

Proof Sketch:

- Let P and Q be two distinct u, v-paths.
- Combine P and Q to form a closed walk W.
- The closed walk W must include a cycle, as it contains redundant edges or vertices.
- Extract the cycle by tracing the paths until they overlap.

Connectivity via One Vertex

Statement: A graph is connected if and only if some vertex is connected to all other vertices.
Note: *'Connected' does not necessarily mean 'adjacent'.*
Proof Sketch:

- (*If direction*) If v is connected to all other vertices, every pair of vertices is connected via v.
- (*Only if direction*) If G is connected, there exists a spanning tree with v connected to all vertices.
- Thus, G is connected if and only if some vertex connects to all others.

Partitioning a Closed Trail into Cycles

Statement: The edge set of every closed trail can be partitioned into edge sets of cycles.

Proof Sketch:

- Let T be a closed trail in G.
- Identify the first repeated vertex v in T. Trace the subpath from v back to itself, forming a cycle C.
- Remove C from T, leaving a new closed trail.
- Repeat the process until T is empty, producing a partition of T into cycles.

Bipartite Graphs and Odd Cycles

Statement: Every graph G with no odd cycles is bipartite.
Proof Sketch:

- Assume G has no odd cycles.
- Assign vertices to two sets X and Y based on their distances (even/odd) from a starting vertex.
- Since there are no odd cycles, no edge connects vertices within the same set.
- Thus, G is bipartite.

Graph of Permutations is Connected

Statement: If G_n is the graph whose vertices are the permutations of $[n]$ and two permutations are adjacent if one results from switching two elements, then G_n is connected.

Proof Sketch:

- Any permutation can be transformed into another by a sequence of adjacent transpositions (swapping neighboring elements).
- Starting from any permutation, repeatedly swap adjacent elements to reach the identity permutation.
- Thus, any two permutations are connected via a series of transpositions, proving G_n is connected.

Biclique Characterization

Statement: A connected simple graph not having P_4 or C_3 as an induced subgraph is a biclique.

Proof Sketch:

- Assume G is connected and does not have P_4 or C_3 as induced subgraphs.
- Absence of P_4 implies that every pair of vertices has a common neighbor.
- Absence of C_3 ensures no three vertices form a triangle.
- These properties force G to be a complete bipartite graph (biclique).

Components of G_k

Statement: Show that the graph G_k whose vertices are the k-tuples of bits has at most two components.
Proof Sketch:

- Two k-tuples are adjacent if they differ in exactly one bit.
- All k-tuples with an even number of 1s form one component, and all k-tuples with an odd number of 1s form another component.
- If k is odd, flipping a single bit changes the parity, connecting the components.
- Thus, G_k has at most two components.

Connectivity and Reachability

Statement: A graph G is connected if, for any vertex x, the set of vertices reachable by paths from x is the set of all vertices.

Proof Sketch:

- (*If direction*) If G is connected, every vertex can be reached from x by definition.
- (*Only if direction*) If all vertices are reachable from x, there exists a path between any pair of vertices.
- Therefore, G is connected if and only if all vertices are reachable from any single vertex.

Path Between Odd-Degree Vertices

Statement: Let G be a graph with only two vertices of odd degree u and v. Then there exists a u, v-path.
Proof Sketch:

- By the Handshaking Lemma, the sum of degrees in G is even.
- If u and v are the only odd-degree vertices, all other vertices have even degree.
- Starting from u, construct an Eulerian trail, which must end at v.
- Thus, there exists a u, v-path.

Odd Edge Subgraph is Even

Statement: If C is a closed walk in a simple graph G, then the subgraph consisting of the edges appearing an odd number of times in C is an even graph.
Note: *A graph where every single vertex has an even degree, is called even graph.*
Proof Sketch:

- Let C be a closed walk in G.
- Construct a subgraph H consisting of edges appearing an odd number of times in C.
- Each vertex in H has even degree since every entry into a vertex in C is paired with an exit.
- Thus, H is an even graph.

Degree List of a Graph

Statement: Every list of nonnegative integers with an even sum is the degree list of some graph (*not necessarily a simple graph*).
Proof Sketch:

- Let $d_1, d_2, \ldots, d_n$ be a list of nonnegative integers with an even sum.
- Use the Havel-Hakimi algorithm to iteratively construct a graph:
  - Arrange the degrees in non-increasing order.
  - Remove the largest degree d_1 and reduce the next d_1 degrees by 1.
  - Repeat until all degrees are zero or invalid.
- Since the sum of the degrees is even, the process succeeds, yielding a graph.

Graphic n-Tuple Characterization

Statement: An n-tuple of nonnegative integers with largest entry k is graphic if the sum is even, $k < n$, and every entry is k or $k-1$.
Proof Sketch:

- Let $d_1, d_2, \ldots, d_n$ be the n-tuple.
- (*Sum Condition*) The sum of degrees must be even to ensure edge pairing.
- (*Largest Entry Condition*) $k < n$ ensures enough vertices for k edges.
- (*Value Range Condition*) Entries k or $k-1$ guarantee a realizable graph structure.
- Using these conditions, construct a graph using iterative degree reduction (e.g., Havel-Hakimi).

Independent Set in Loopless Digraph

Statement: Every loopless digraph has an independent set S such that every vertex not in S has a path of length at most 2 to S.

Proof Sketch:

- Use a greedy algorithm to construct S:
  - Start with $S = \emptyset$.
  - Iteratively add a vertex v to S if v has no incoming edges from S.
- Every vertex not in S is either adjacent to a vertex in S or has a neighbor adjacent to S (path of length at most 2).
- Thus, S satisfies the conditions.

Components After Removing an Edge

Statement: If e is an edge of G, then $G - e$ has at most one more component than G.

Proof Sketch:

- Removing edge e can only disconnect vertices that were connected by e.
- If e is a cut-edge, $G - e$ has one more component than G.
- If e is not a cut-edge, $G - e$ has the same number of components as G.
- Thus, $G - e$ has at most one more component than G.

Sum of Degrees Equals Twice the Edges

Statement: The number of edges in a graph is the sum of the degrees divided by 2.

Proof Sketch:

- Each edge contributes 1 to the degree of each of its endpoints.
- Summing over all vertices counts each edge twice.
- Let $d_1, d_2, \ldots, d_n$ be the degrees of the vertices.
- Total degree sum is $\sum_{i=1}^{n} d_i = 2e(G)$.
- Dividing by 2 gives $e(G) = \frac{1}{2} \sum_{i=1}^{n} d_i$.

Odd-Degree Vertices

Statement: The number of vertices of odd degree in a graph is even.
Proof Sketch:

- Total degree sum $\sum_{i=1}^{n} d_i$ is even since it equals $2e(G)$.
- Odd-degree vertices contribute an odd sum to $\sum_{i=1}^{n} d_i$.
- To ensure the total sum is even, the number of odd-degree vertices must be even.

Edge Bound in a Simple Graph

Statement: If a simple graph G has n vertices and k components, then $e(G) \leq \frac{(n-k)(n-k+1)}{2}$.

Proof Sketch:

- Each component of G has at most $\frac{v_i(v_i-1)}{2}$ edges, where v_i is the number of vertices in the i-th component.
- The function $\frac{x(x-1)}{2}$ is maximized when x is as large as possible.
- Distribute vertices evenly among components to maximize edges.
- Total edge count is bounded by $\frac{(n-k)(n-k+1)}{2}$.

Components in $G + H$

Statement: If G has k components and H has l components, then $G + H$ has $k + l$ components.

Proof Sketch:

- The union $G + H$ does not add edges between G and H.
- Components of G and H remain separate.
- Thus, the number of components in $G + H$ is $k + l$.

Maximum Degree in $G + H$

Statement: The maximum degree of $G + H$ is $\max\{\Delta(G), \Delta(H)\}$.
Proof Sketch:

- $G + H$ overlays the edges of G and H on the same vertex set.
- A vertex's degree in $G + H$ is the sum of its degrees in G and H.
- The maximum degree is therefore $\max\{\Delta(G), \Delta(H)\}$.

P_n is Bipartite

Statement: P_n, the path graph, is bipartite.
Proof Sketch:

- Partition vertices of P_n into two sets based on parity of their distance from an endpoint.
- No two vertices in the same set are adjacent.
- Thus, P_n is bipartite.

Largest Bipartite Subgraph of C_n

Statement: The largest bipartite subgraph of C_n has n edges if n is even, and $n - 1$ edges if n is odd.
Proof Sketch:

- A cycle C_n is bipartite if and only if n is even.
- For even n, the entire C_n is bipartite with n edges.
- For odd n, remove one edge to make the graph acyclic and bipartite.
- This results in a subgraph with $n - 1$ edges.

Largest Bipartite Subgraph of K_n

Statement: The largest bipartite subgraph of K_n has $\lfloor n^2/4 \rfloor$ edges.
Proof Sketch:

- Divide n vertices into two sets of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.
- Connect every vertex in one set to all vertices in the other set.
- Edge count is $\lfloor n^2/4 \rfloor$.

Outline

Extremal Problems in Graph Theory

Classic Extremal Graph Theory Problems

- **Turán's Theorem** - Maximum edges in a graph avoiding K_r.
- **Erdős-Stone Theorem** - Asymptotic version of Turán's theorem.
- **Mantel's Theorem** - Maximum edges in a triangle-free graph.
- **Zarankiewicz Problem** - Maximum edges in a bipartite graph avoiding $K_{s,t}$.
- **Kővári-Sós-Turán Theorem** - Bipartite analogue of Turán's theorem.

Ramsey-Type Problems

- **Ramsey's Theorem** - Existence of monochromatic subgraphs in edge-colored graphs.
- **Erdős-Rado Sunflower Theorem** - Combinatorial extremal result related to Ramsey theory.

More Extremal Graph Problems

Cycle & Path Extremal Problems

- **Erdős-Gallai Theorem** - Minimum edge count for containing a long path or cycle.
- **Dirac's Theorem** - Sufficient degree condition for a Hamiltonian cycle.
- **Bondy-Chvátal Theorem** - Closure concept for Hamiltonian graphs.
- **Ore's Theorem** - Hamiltonicity based on degree sum conditions.

Matching & Covering Extremal Theorems

- **Hall's Marriage Theorem** - Condition for perfect matchings in bipartite graphs.
- **König's Theorem** - Relation between maximum matching and minimum vertex cover in bipartite graphs.
- **Gallai-Edmonds Decomposition** - Structure of matchings in general graphs.
- **Tutte's Theorem** - Condition for a perfect matching in general graphs.

Advanced Graph Problems

Connectivity & Expansion Theorems

- **Menger's Theorem** - Maximum number of independent paths between two vertices.
- **Whitney's Theorem** - Characterization of 2-connected graphs.
- **Cheeger's Inequality** - Connection between expansion and eigenvalues of adjacency matrix.

Graph Coloring & Partition Theorems

- **Brook's Theorem** - Upper bound on chromatic number.
- **Hajnal-Szemerédi Theorem** - Equitable colorings of graphs.
- **Erdős-Ko-Rado Theorem** - Bounds on intersecting families of sets related to graphs.
- **Mycielski's Theorem** - Constructing triangle-free graphs with high chromatic number.

Problems Proven Impossible Using Graphs I

Certain problems that appear plausible at first glance can be shown to be impossible using graph theory:

1. **Koenigsberg Bridges Problem:**
   - Traversing all seven bridges exactly once is impossible due to odd-degree vertices.
2. **Three Utilities Problem:**
   - Connecting three houses to three utilities without crossing lines is impossible because $K_{3,3}$ is non-planar.
3. **Four Color Problem:**
   - Proves that no map can be colored with fewer than four colors without adjacent regions sharing the same color.
4. **Domino Tiling Problem:**
   - A $2 \times n$ chessboard with opposite corners removed cannot be tiled due to color imbalances.

Problems Proven Impossible Using Graphs II

5. **Handshake Problem:**
   - If the number of people is odd, it's impossible for everyone to pair up for handshakes.
6. **Traveling Salesman Problem (TSP):**
   - Finding the shortest path visiting all vertices is computationally infeasible for large graphs (NP-hard).

Question to Ponder: How does graph theory help simplify and formalize seemingly complex problems?

Famous Open Problems in Graph Theory I

1. **Hamiltonian Cycle Problem (HCP)**
   - NP-complete for general graphs.
   - Special cases like **Tait's Conjecture** remain unresolved.
2. **Longest Path Problem**
   - NP-hard: Finding the longest simple path in a graph is computationally difficult.
   - No known efficient algorithm for general graphs.
3. **Gallai's Path Decomposition Conjecture**
   - Every connected graph can be decomposed into at most $\lceil n/2 \rceil$ paths.
   - Proven for some special graph classes, but remains open in general.
4. **Lovász' Hamiltonicity Conjecture**
   - Are all connected vertex-transitive graphs Hamiltonian?
   - Still open for Cayley graphs and some families of graphs.

Famous Open Problems in Graph Theory II

5. **Erdős–Gyárfás Conjecture**
   - Every triangle-free graph of minimum degree d has a cycle of length $\leq 2^d$.
   - Partially solved but remains open for general graphs.
6. **Chvátal's Toughness Conjecture**
   - Is there a constant t such that every t-tough graph is Hamiltonian?
   - Open for general graphs.
7. **Barnette's Conjecture**
   - Every 3-connected cubic bipartite planar graph is Hamiltonian.
   - Open, but proven for some small cases.
8. **Graceful Tree Conjecture**
   - Every tree can be labeled gracefully (distinct edge differences).
   - Still an unsolved problem in combinatorial graph theory.

Famous Open Problems in Graph Theory III

9. Erdős-Pósa Property for Paths

- If a graph has many disjoint paths, does it always contain a bounded-size hitting set?
- Known for cycles, but open for general paths.

10. Alon-Saks-Seymour Conjecture

- The chromatic number of a graph is at most logarithmic in the number of edge-disjoint paths.
- Open in the general case.