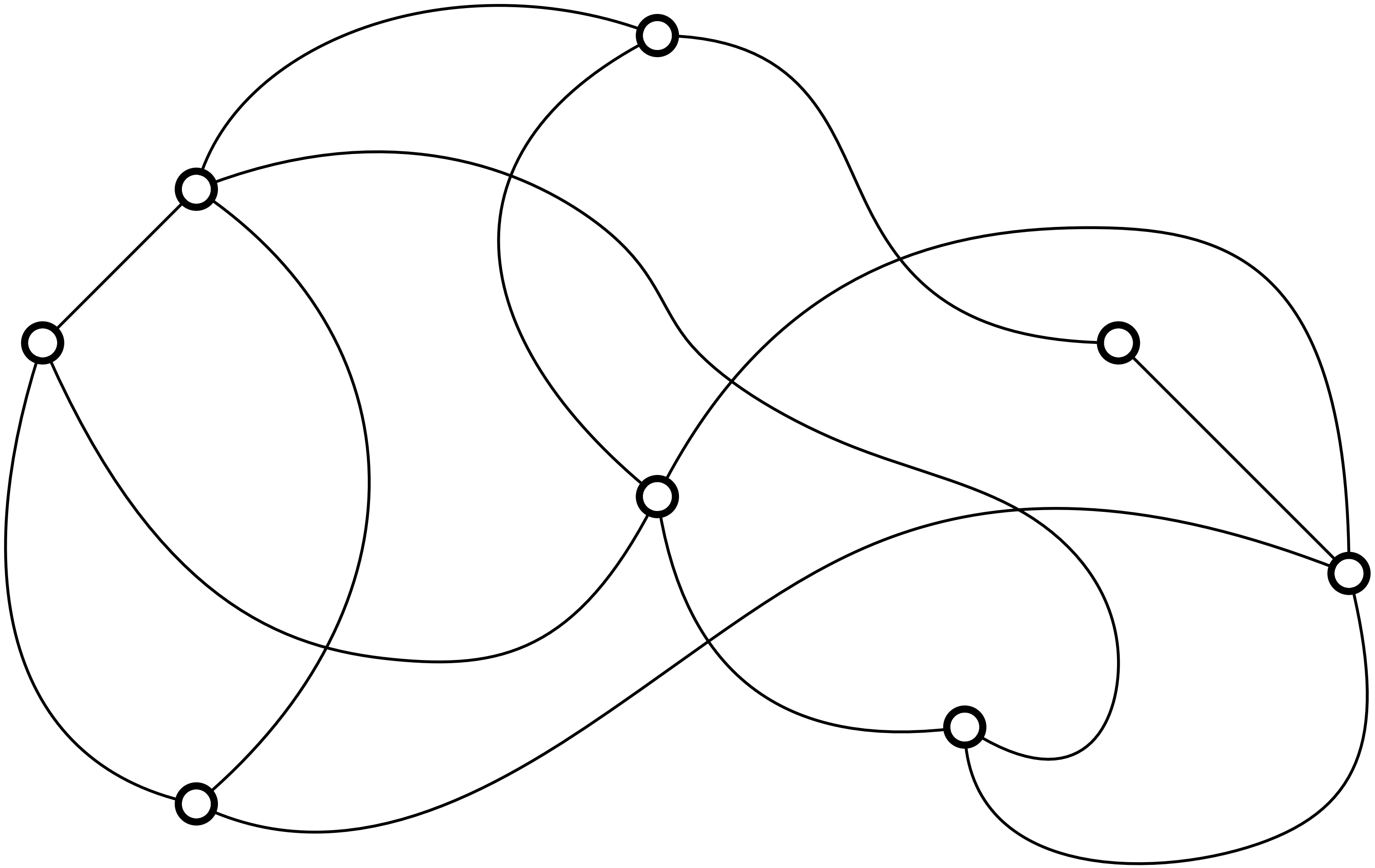
Basic Graph Theory

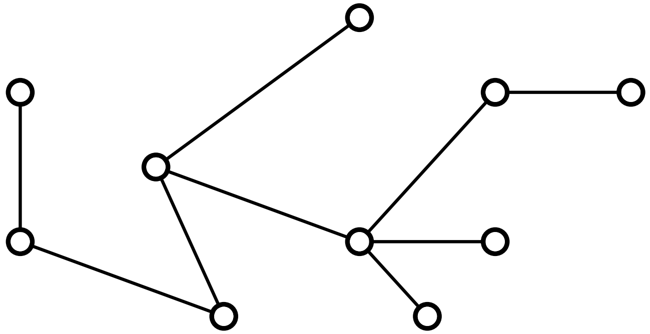
Birgit Vogtenhuber

Graphs?



Why Graphs?

- Graphs are a versatile data structure
- Graph-related algorithms are rather important in algorithm theory and applications
- Many computational problems can be formulated efficiently in terms of graphs, see for example optimization problems.



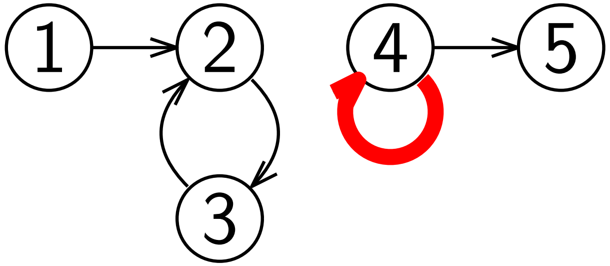
Overview

- Basic graph terminology
- Different ways to store graphs
- Some simple graph algorithms
- Planar graphs and plane drawings

Basic Terminology

- **Directed graph** $G = (V, E)$:
  V is the set of vertices and
  E is the set of edges: $E \subseteq V \times V$
  edge (u, v): ordered pair of vertices,
  $u =$ startpoint, $v =$ endpoint

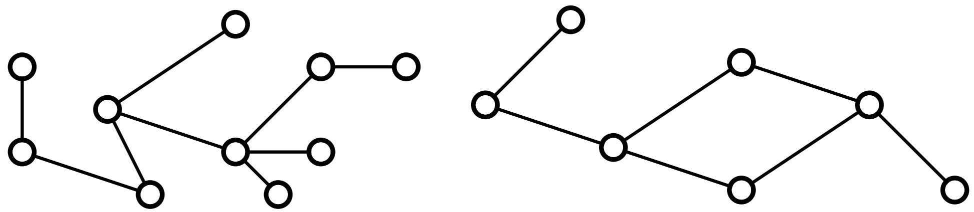
Example: $V = \{1, 2, 3, 4, 5\}$,
$E = \{(1, 2), (2, 3), (3, 2), (4, 5), (4, 4)\}$



- A **loop** at a vertex $v \in V$ is an edge (v, v)
- A directed Graph is called **simple** if it has no loops

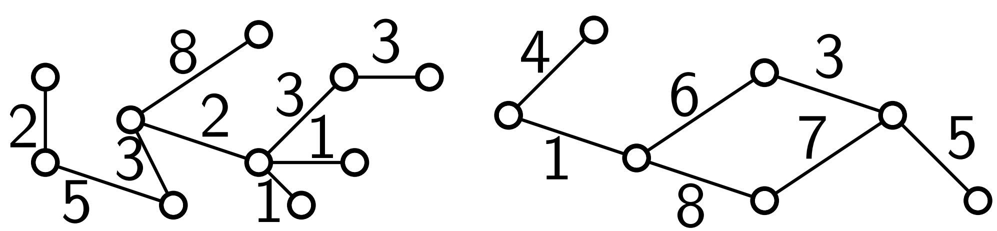
Basic Terminology

- **Undirected graph**: like a simple directed graph where E is symmetric, that is, $(v, w) \in E \Leftrightarrow (w, v) \in E$



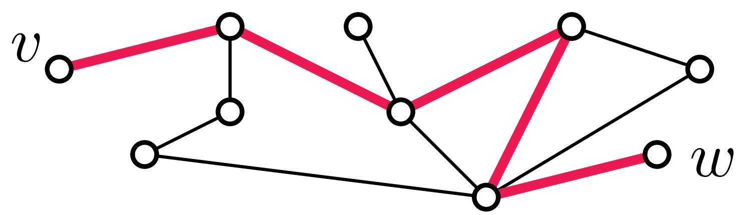
$\Leftrightarrow$ E is a set of unordered pairs (v, w) with $v \neq w \in V$

- A (directed or undirected) **weighted graph** is a triple $G = (V, E, g)$ where g assigns a weight to each edge



Basic Terminology

- A **path** in $G = (V, E)$ from v to w is a sequence of vertices $v = v_0, v_1, \ldots, v_k = w \in V$ with edges $(v_i, v_{i+1}) \in E$ for all $0 \leq i < k$



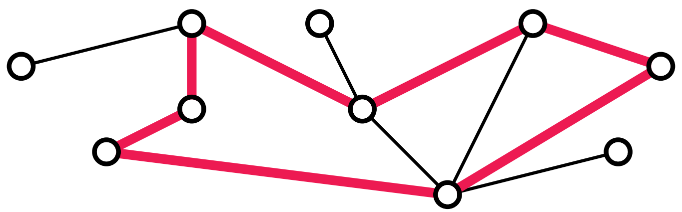
- **Simple** path: contains every vertex in V at most once

- **Length** of a path: number of its edges (v_i, v_{i+1})
  length in *weighted graphs*: sum of the edge weights

- **Distance** $d_G(v, w)$ from v to w: length of the *shortest path* from v to w in G (or ∞ if no such path exists)

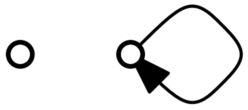
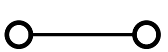
Basic Terminology

- A **cycle** in $G = (V, E)$ is a path $v_0, v_1, \ldots, v_k$ with $v_0 = v_k$



- **Simple** cycle: contains every vertex in V at most once, except for $v_0 = v_k$

- **Length** of a cycle: number of its edges (v_i, v_{i+1})
  length in *weighted graphs*: sum of the edge weights

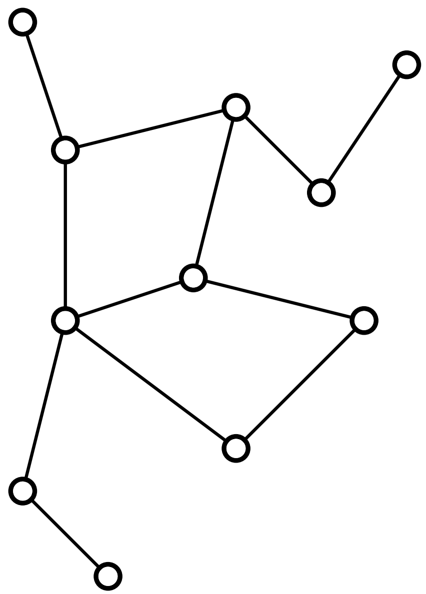
- **Trivial** cycle: contains only one vertex: 
  or two for undirected graphs: 

Basic Terminology

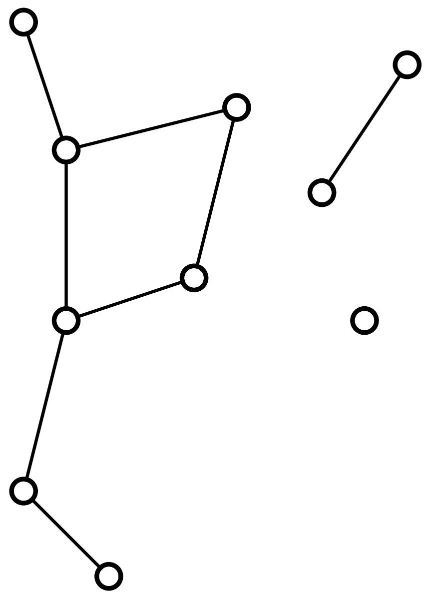
- A **subgraph** $G' = (E', V')$ of $G = (E, V)$ is a graph with $V' \subseteq V$ and $E' \subseteq E$. Note that for every edge $(u', v') \in E'$ we must have $u', v' \in V'$.

$Example:$

Graph G



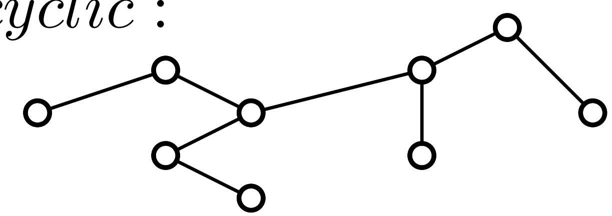
Subgraph G'



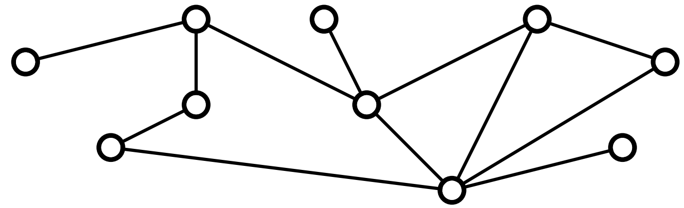
Basic Terminology

- A graph is **acyclic** if does not contain any subgraph that is a non-trivial simple cycle

acyclic :

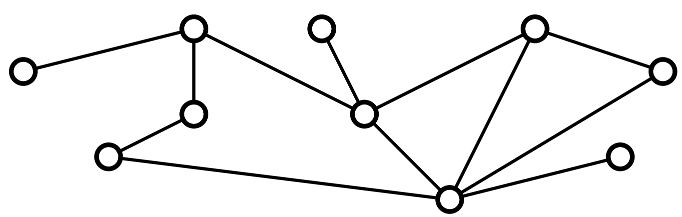


not acyclic :

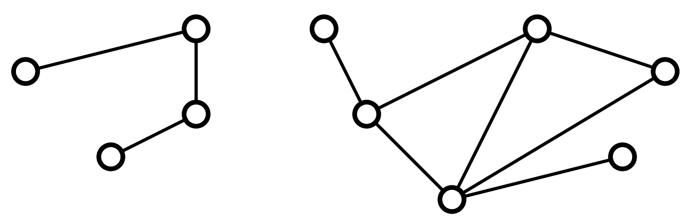


- A graph is **connected** if it contains a path from v to w for every pair $v, w \in V$ (not necessarily an edge (v, w))

connected :

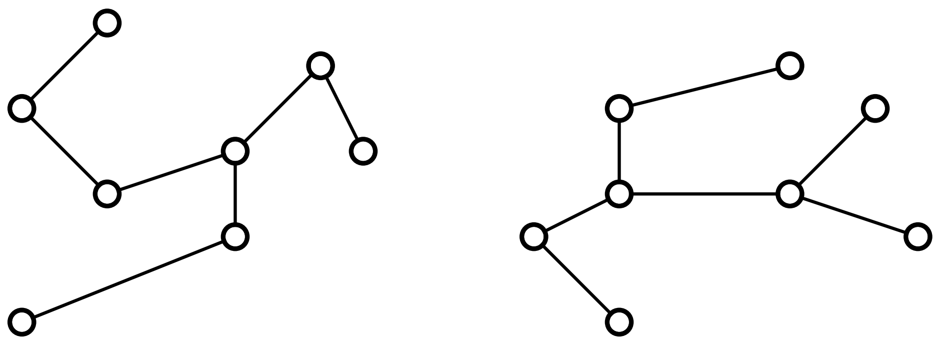


disconnected :

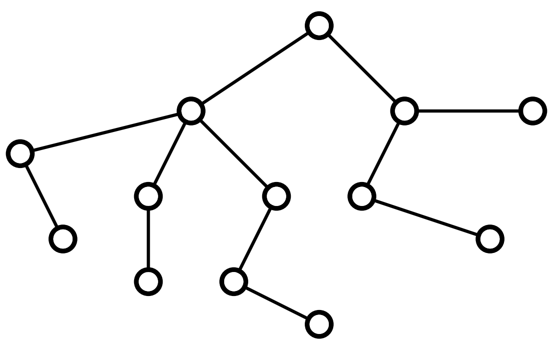


Basic Terminology

- An undirected graph is a **forest** if it is acyclic



- An undirected graph is a **tree** if it is acyclic and connected



Basic Terminology

Degrees of vertices in a graph $G = (V, E)$:

- The **in-degree (out-degree)** of a vertex $v \in V$ is the number of edges ending (starting) in v:

  in-degree $(v) := |\{w \mid (w, v) \in E\}|$

  out-degree $(v) := |\{w \mid (v, w) \in E\}|$

- For undirected graphs we have

  in-degree(v) = out-degree(v) $\quad \forall\, v \in V$.

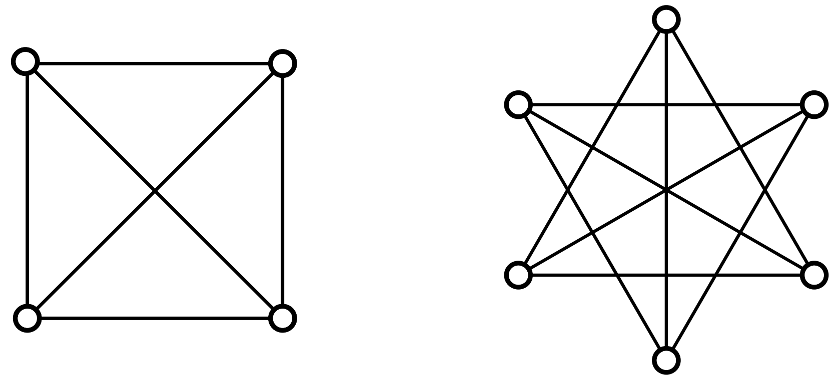
  In this case it is called the **degree** of the vertex v.

Basic Terminology

Degrees of vertices in an undirected graph $G = (V, E)$:

- degree $(v) = 0$: v is called an **isolated vertex** of G
- degree $(v) = 1$: v is called a **leaf** (end vertex) of G
- degree $(v) = r \quad \forall\, v \in V$: The graph G is called a **regular graph** of degree r

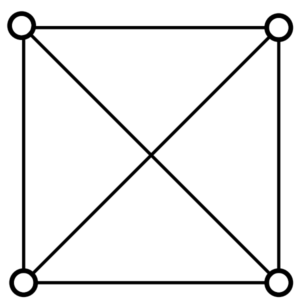
Example: regular graph(s) of degree 3



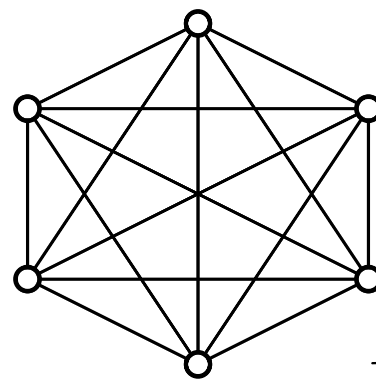
Basic Terminology

Degrees of vertices in an undirected graph $G = (V, E)$:

- An undirected **complete graph** on n vertices contains all $\binom{n}{2}$ possible edges (regular of degree $n-1$)

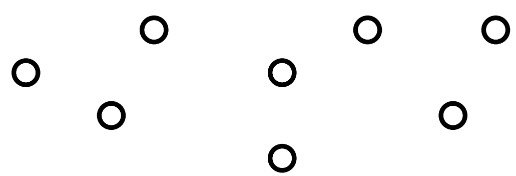


K_4



K_6

- An **empty graph** has no edges (regular of degree 0)



How to Store a Graph?

- **Size** of a graph $G = (V, E)$:
  - Number of vertices: $n = |V|$
  - Number of edges: $m = |E|,\ 0 \leq m \leq n^2$

$\Rightarrow$ In total: size $\Theta(n + m)$

$\Rightarrow$ **Analysis** needs two parameters !

- We distinguish between
  - **dense** graphs: $m \approx n^2$, for example complete graphs
  - **sparse** graphs: $m \ll n^2$, for ex. trees $(m = n - 1)$ or hypercubes $(m = \frac{d}{2} \cdot n = O(n \log n)$, where d is the dimension of the hypercube)

How to Store a Graph?

Adjacency-matrix: Matrix $A[1 \ldots n, 1 \ldots n]$ with

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{else} \end{cases}$$

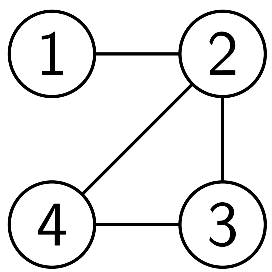
- Memory: $\Theta(n^2)$
- convenient for dense graphs
- test for existence of an edge in $\Theta(1)$ time

Adjacency-List: Array $F[1 \ldots n]$ with pointers,
$F[i]$ points to a linear list with all vertices that are incident to the i^{th} vertex

- Memory: $\Theta(n+m)$
- test for existence of an edge in $\Omega(1), O(n)$ time

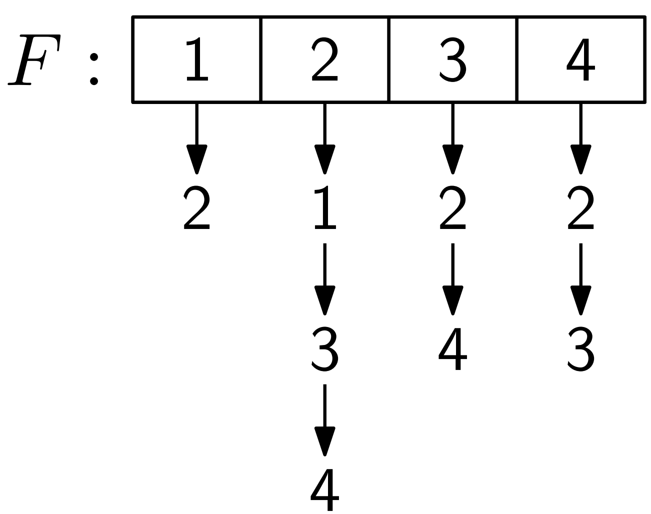
How to Store a Graph?

A small example:



$$A = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

$\uparrow$ symmetric if graph is undirected



$\leftarrow$ 1 Entry for every node,
$\Theta(n+m)$ memory for all edges.

The k neighbours of a node can be obtained in $\Theta(k)$ time.

Searching in Graphs

- Known from Data Structures and Algorithms 1:
  - Searching in binary trees
  - “Walking” through a binary tree:
    - in-order (symmetrische Reihenfolge)
    - pre-order (Hauptreihenfolge)
    - post-order (Nebenreihenfolge)
- Searching in general graphs:
  - Breadth-first search (BFS):
    search closeby vertices first
  - Depth-first search (DFS)
    search one branch first

Breadth First Search - BFS

Given: connected graph $G = (V, E)$, startnode $s \in V$

Idea: Starting from s, search in all directions uniformly

- visit vertices u with distance $d_G(s, u) = 1$, then with $d_G(s, u) = 2$, and so on.
- traverse a *BFS-tree* T with root s: branches of T consist of shortest paths to s.
- to build T, assign the following values to each vertex u:

$pre(u)$	predecessor of u in the BFS-Tree T
$state(u)$	new (unvisited), labelled (visited), saturated (all neighbours visited)

- store visited unsaturated vertices in a **queue** Q
- initially: Q empty, $pre(u)$ not set, $state(u) =$ new

BFS Algorithm: Pseudo-Code

BFS(G, s) /* G given as adjacency list F */
<u>for</u> all $u \in V$
    state(u)=new
state(s)=labelled
num(s)=1; $i = 2$; pre(s)=nil
PUT(Q, s)
<u>while</u> $Q \neq 0$
    GET(Q, u)
    <u>for</u> all $v \in F[u]$ /* testing all neighbors of u */
        <u>if</u> state(v)==new
            state(v)=labelled; num(v)=i
            pre(v)=u; PUT(Q, v); $i = i + 1$
    state(u)=saturated

BFS Algorithm: Analysis

Runtime:

- Each vertex is inserted into Q exactly once:
  $\Theta(1)$ time per vertex $\Rightarrow \Theta(n)$ for all vertices
- After removal of a vertex u from Q, the algorithm goes through the adjacency-list of u:
  $\Theta(\text{degree}(u))$ time for u $\Rightarrow$ How much for all vertices?

  Every edge contributes to exactly two lists
  $\Rightarrow \sum_{u \in V} \text{degree}(u) = 2m$ $\Rightarrow \Theta(m)$ for all vertices

$\Rightarrow$ The whole algorithm: $\Theta(n+m)$ time in total

Memory: $\Theta(n)$ for Q, +graph $\Rightarrow \Theta(n+m)$ space in total

$\Rightarrow$ Runtime and memory **linear in the size of** G

All Distances in an Unweighted Graph

Question: How can one compute all distances between pairs of vertices in an unweighted graph G ?

- All the shortest paths from some vertex u to s are coded in the BFS-tree T via the pre-pointers.
- distances to s can be easily computed during BFS:
  - $d_G(s, s) = 0$
  - $d_G(s, u) = d_G(s, pre(u)) + 1$

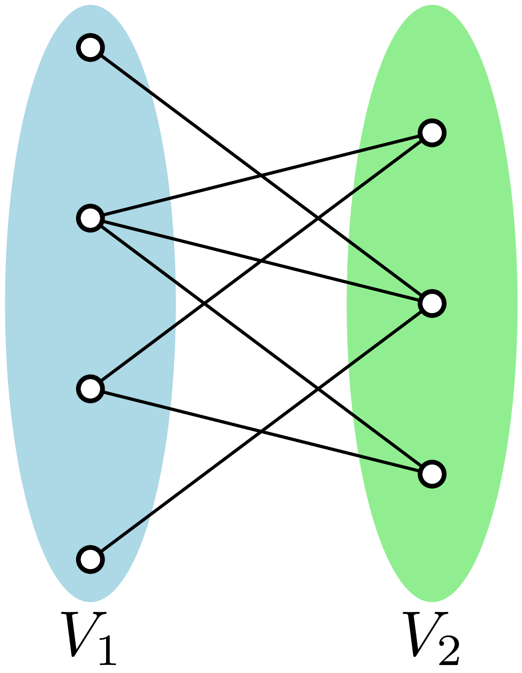
$\Rightarrow$ Running BFS for n times (once for each vertex), one can compute the distances between any $u, v \in V$

$\Rightarrow$ The *distance-matrix* of G can be computed in $\Theta(n \cdot m)$ time and $\Theta(n^2)$ space if G is connected.

? What if G is disconnected?

Bipartite Graphs

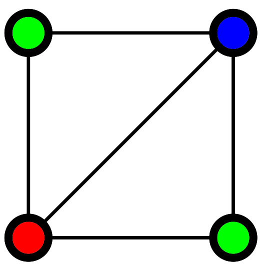
A graph $G(V, E)$ is called **bipartite**, if there exists a partition of V into V_1, V_2 such that all $(u, v) \in E$ have one endpoint in V_1 and the other one in V_2.



In other words:
G contains no edges within V_1 or V_2

k-Colorability of a Graph

A graph $G(V, E)$ is called **k-colorable** if its nodes can be colored with $\leq k$ colors, so that no nodes of the same color share an edge.

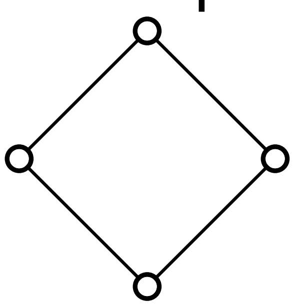


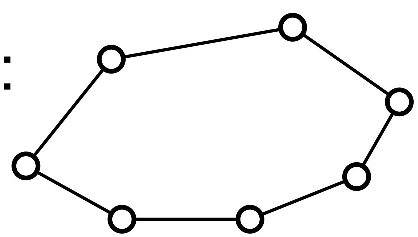
Example: How many colors are needed?

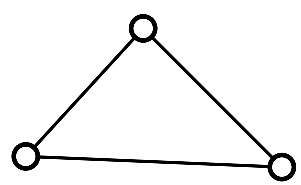
$k = 3$ colors are enough

The **chromatic number** $\chi(G)$ (spoken 'Chi of G') of a graph G is the minimum k such that G is k-colorable.

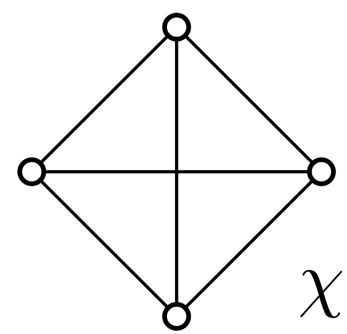
Examples:







$$\chi(\text{cycle}) = \begin{cases} 2, & n \text{ even} \\ 3, & n \text{ odd} \end{cases}$$



$$\chi(K_4) = 4$$

Recognizing Bipartite Graphs

Question: How to determine whether a graph is bipartite?

Observation: A graph G is 2-colorable $\Leftrightarrow$ G is bipartite

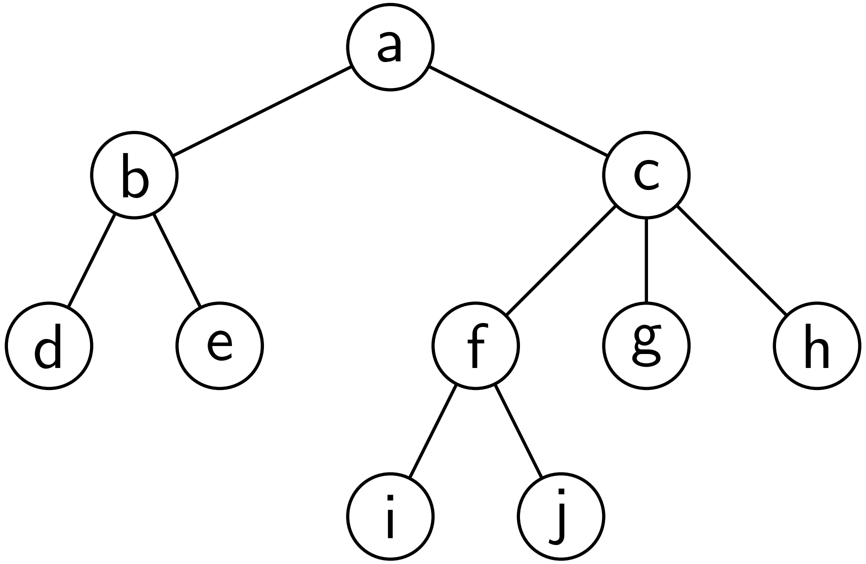
Algorithm:

- Choose arbitrary vertex s and color it blue
- Traverse G in BFS-Order, starting from s
- For each vertex u that is removed from the queue Q:
  - u is colored, w.l.o.g. say red
  - check for all colored neighbors of u that they are blue. If no: return false
  - color all uncolored neighbors of u in blue
- After processing all vertices: return true.

Questions: Correctness? Runtime & Memory?

Depth First Search - DFS

- **Idea**:
  DFS explores the graph, starting at the last visited vertex having unvisited neighbors.
- **Special case**: G is a tree $\Rightarrow$ DFS-Order $=$ pre-order



- Maintain Stack ST that contains all visited but not yet saturated nodes.
- Rest similar to BFS

Question: What is the pre-order for this tree?

a b d e c f i j g h

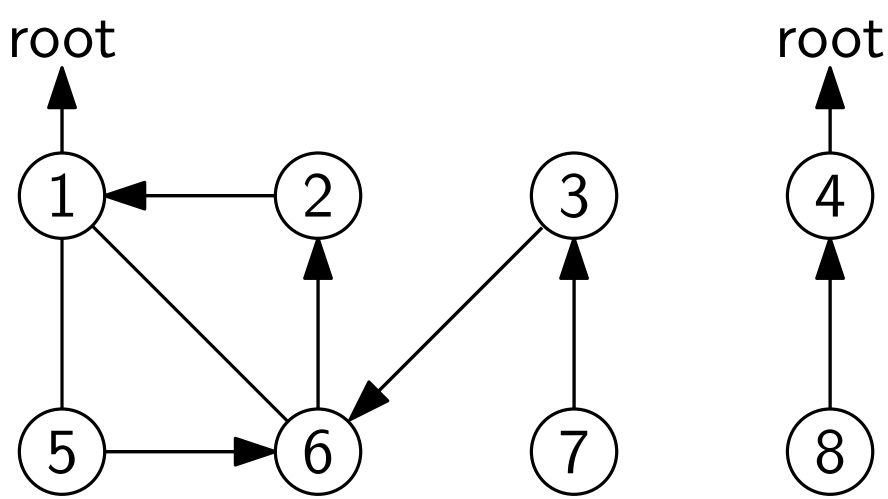
DFS Algorithm: Pseudo-Code

```
DFS(G)          /* G given as adjacency list F */
for all u ∈ V
    state(u)=new
    pre(u)=nil
for all u ∈ V       /* loop not necessary for connected graphs */
    if state(u)==new
        DEPTH(u)

DEPTH(u)
state(u)=visited; write(u)
for all v ∈ F[u]        /* test all neighbors of u */
    if state(v)==new
        pre(v)=u
        DEPTH(v)            ← recursion can be replaced by stack
state(u)=saturated
```

DFS Algorithm: Example

Example:



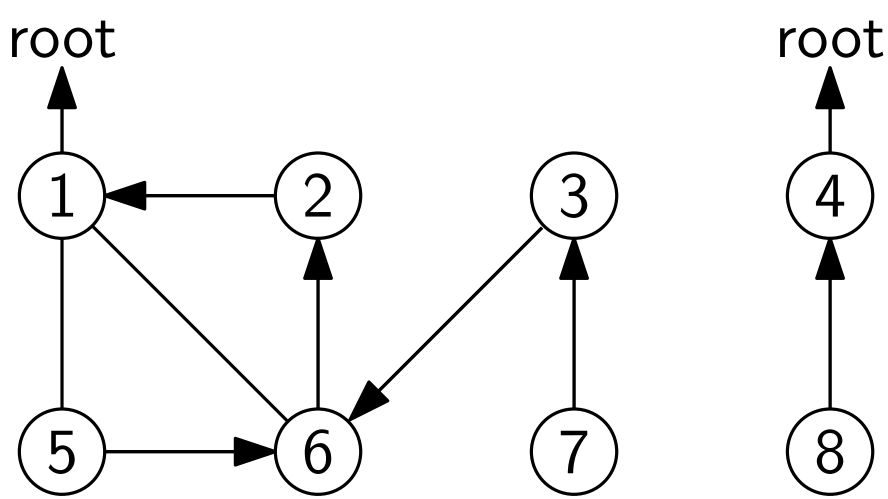
F :

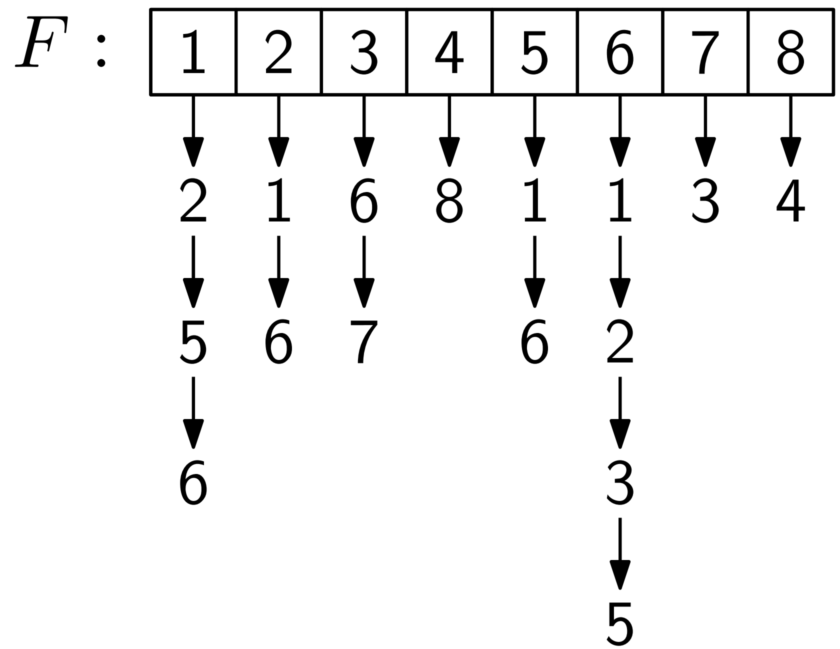
1	2	3	4	5	6	7	8
2	1	6	8	1	1	3	4
5	6	7		6	2		
6					3		
					5		

arrows point to predecessors

DFS Algorithm: Example

Example:





Further Observations:

- The pre-pointers form a set of trees (DFS-forest);
- every call of DEPTH in the main programm (not in the recursion) results in a new root and tree.
- For connected graphs there is only one root and tree.

DFS Algorithm: Analysis

Runtime Analysis:

- DEPTH ist called exactly once per node (only for new nodes, that are immediately marked as "visited").
- A call of DEPTH(v) takes $O(\text{degree}(v))$ time

$\Rightarrow$ $\Theta(n+m)$ time in total

Space: $\Theta(n+m)$ space in total

Correctness:

- vertex that is set to visited:
  - put on stack
  - when removed from stack, all neighbors are considered

$\Rightarrow$ every vertex set to visited exactly once

Next

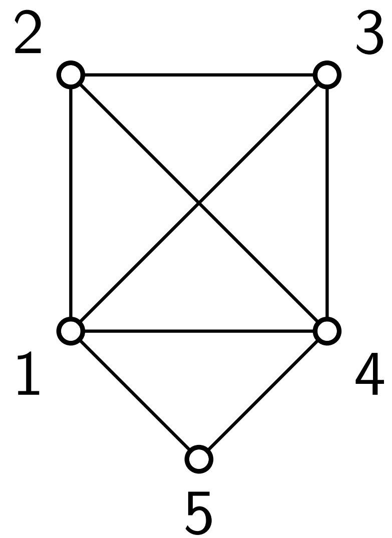
Planar and Plane Graphs:

Definition, Recognition, Properties

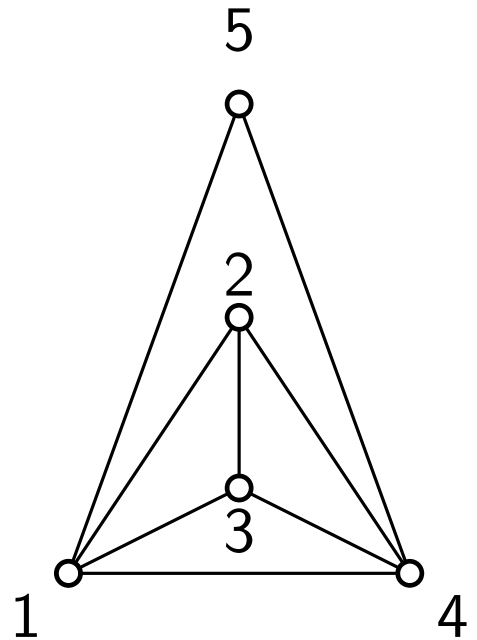
Isomorphic Graphs

Two graphs $G = (V, E)$ and $G' = (V', E')$ are called **isomorphic** if there exists a bijection $\phi : V \to V'$, so that $(v, w) \in E \Leftrightarrow (\phi(v), \phi(w)) \in E'$

Example:



$\Leftrightarrow$



Planar Graphs

Definition 1:
A graph $G = (V, E)$ is called **planar** if there exists an injective mapping of V to $n = |V|$ distinct points in the Euclidian plane and a mapping of E to pairwise disjoint open curves in the Euclidian plane, so that:

1. $e = (x_i, x_j) \Rightarrow$ curve (e) connects point (x_i) with point (x_j)
2. $e = (x_i, x_j) \Rightarrow$ curve (e) does not contain any other point $(x_k), k \neq i, j$

Definition 2:
A graph is called **planar** if it can be drawn in the plane without intersections between its edges (except in endpoints).

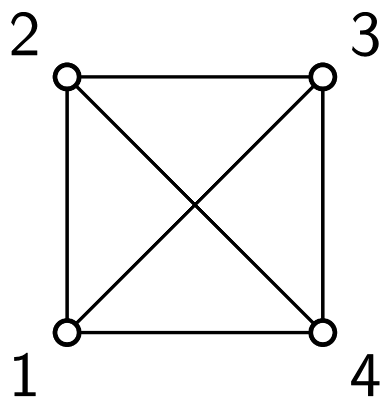
Plane & Embeddable Drawings

A drawing of a graph is called **plane** if it is free of intersections.

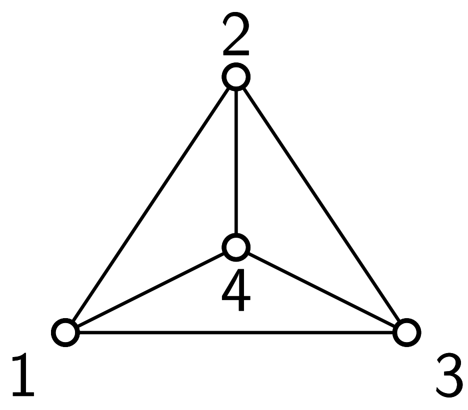
A non-plane drawing of a graph is called **embeddable** if it is isomorphic to a planar graph.

A **plane graph** is a plane drawing of a planar graph.

Example of an embeddable / plane drawing:

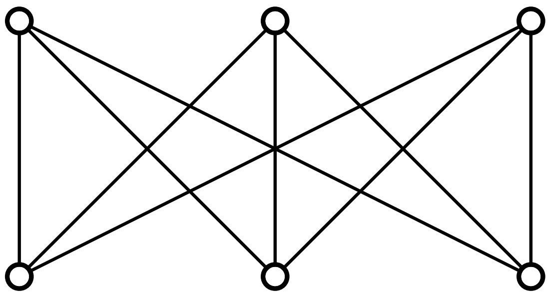


$\Leftrightarrow$

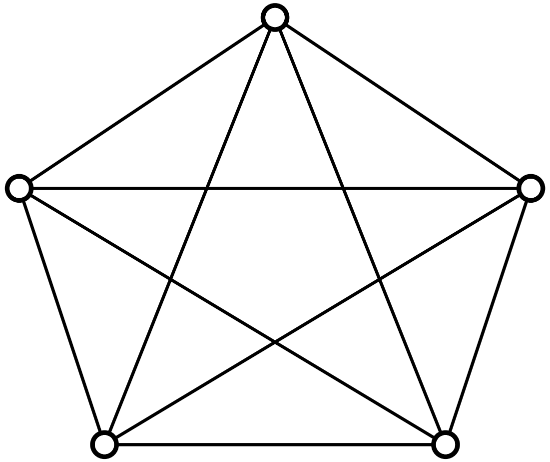


Recognizing Planar Graphs

Examples of non-embeddable drawings:

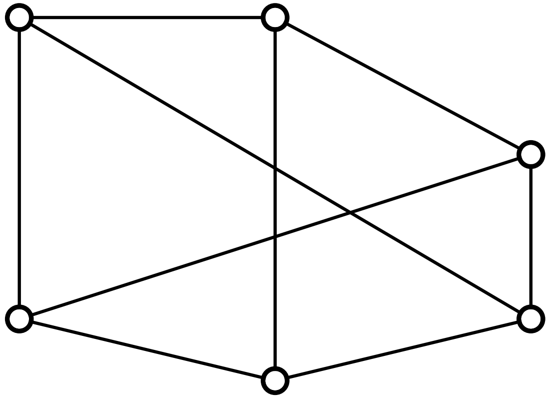


$K_{3,3}$

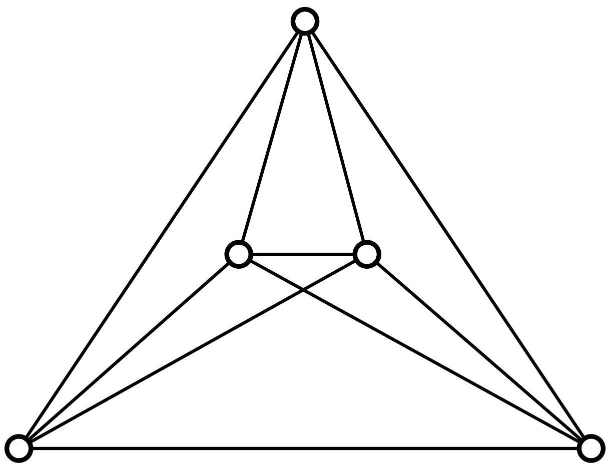


K_5

Non-embeddable drawing $\Leftrightarrow$ drawing of a non-planar graph



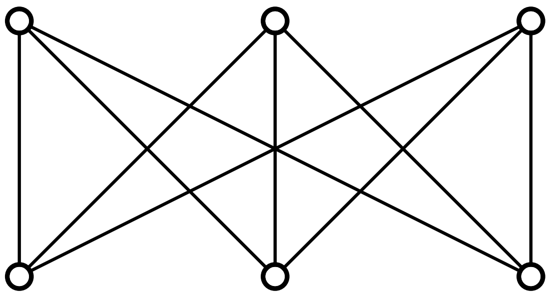
$K_{3,3}$



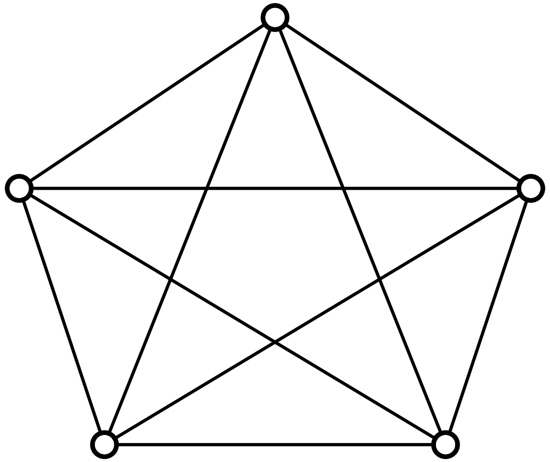
K_5

Recognizing Planar Graphs

Examples of non-embeddable drawings:

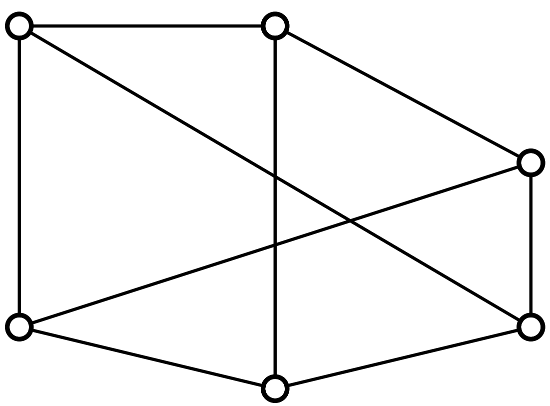


$K_{3,3}$

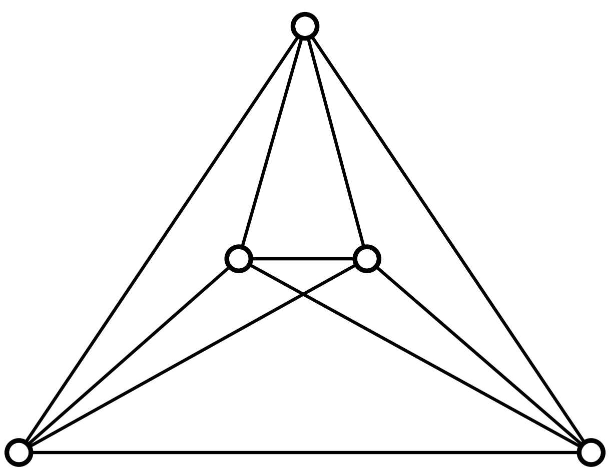


K_5

Embeddable drawing $\Leftrightarrow$ drawing of a planar graph



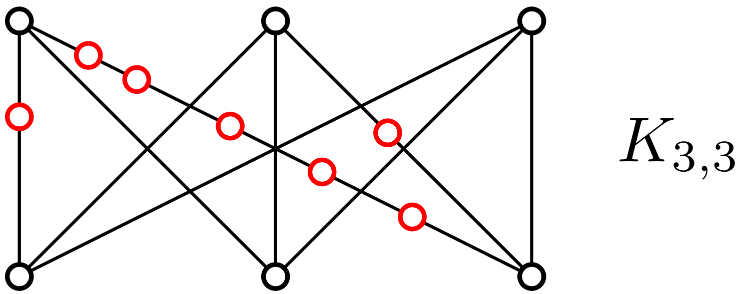
$K_{3,3}$



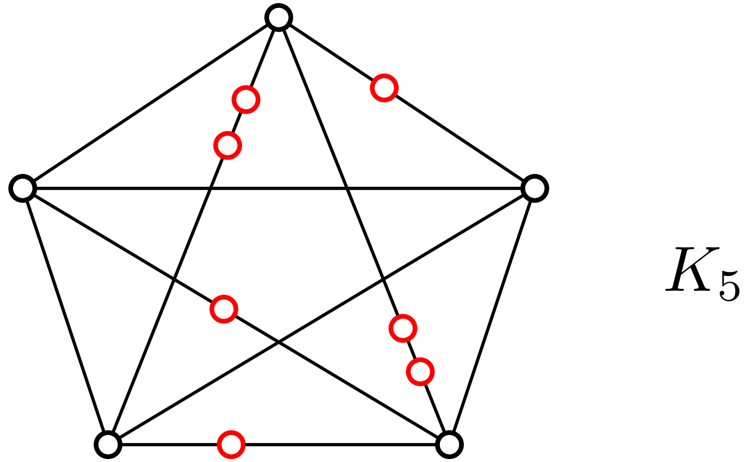
K_5

Recognizing Planar Graphs

embeddability is invariant under subdivision of edges



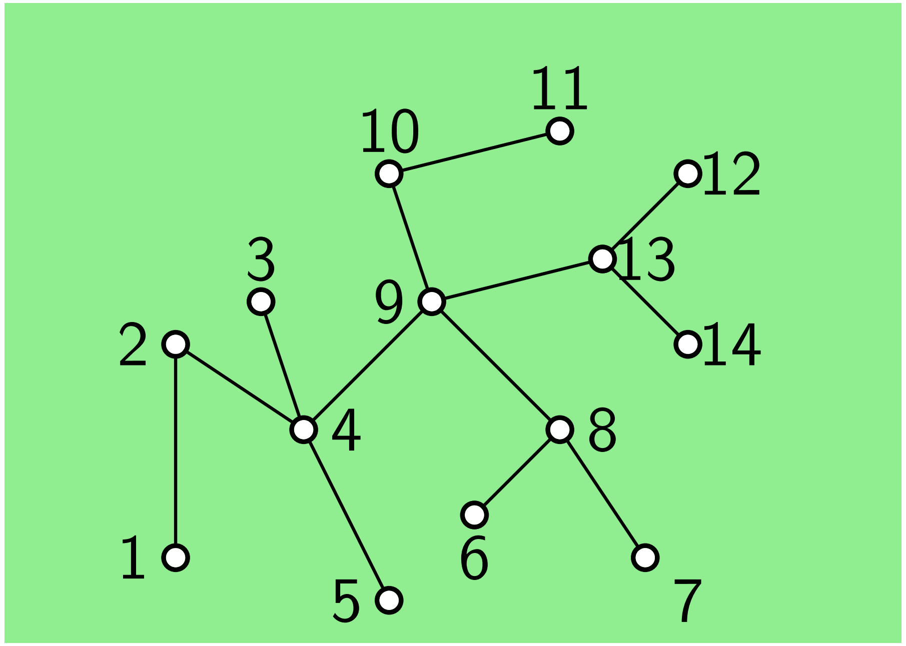
additional vertices: degree 2



Kuratowski's Theorem: A graph G is planar if and only if it does not contain a subgraph G' that is isomorphic to a *subdivision* of $K_{3,3}$ or K_5, respectively. [without proof]

Subdivision of a graph G: some vertices are added "on edges" of G (some edges of G replaced by paths)

Planar Graphs: Examples



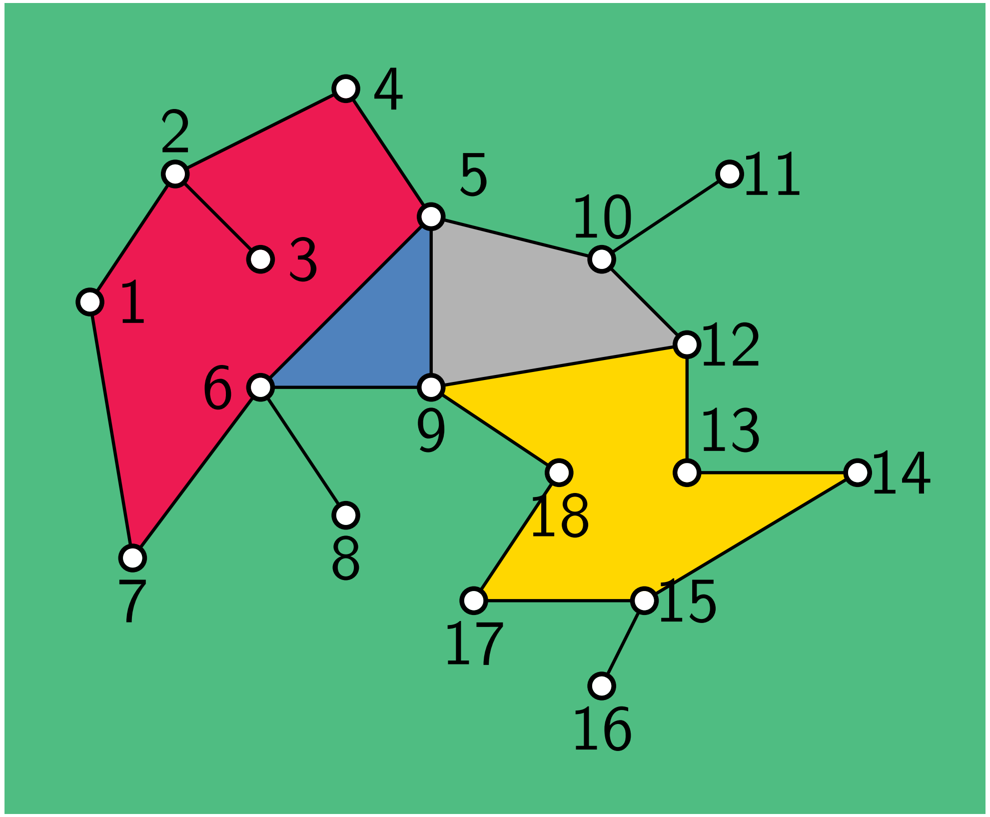
Vertices: $n = 14$

Edges: $m = 13$

Faces: $f = 1$

$14 + 1 = 13 + 2$ ✓

Planar Graphs: Examples



Vertices: $n = 18$

Edges: $m = 21$

Faces: $f = 5$

$18 + 5 = 21 + 2$ ✓

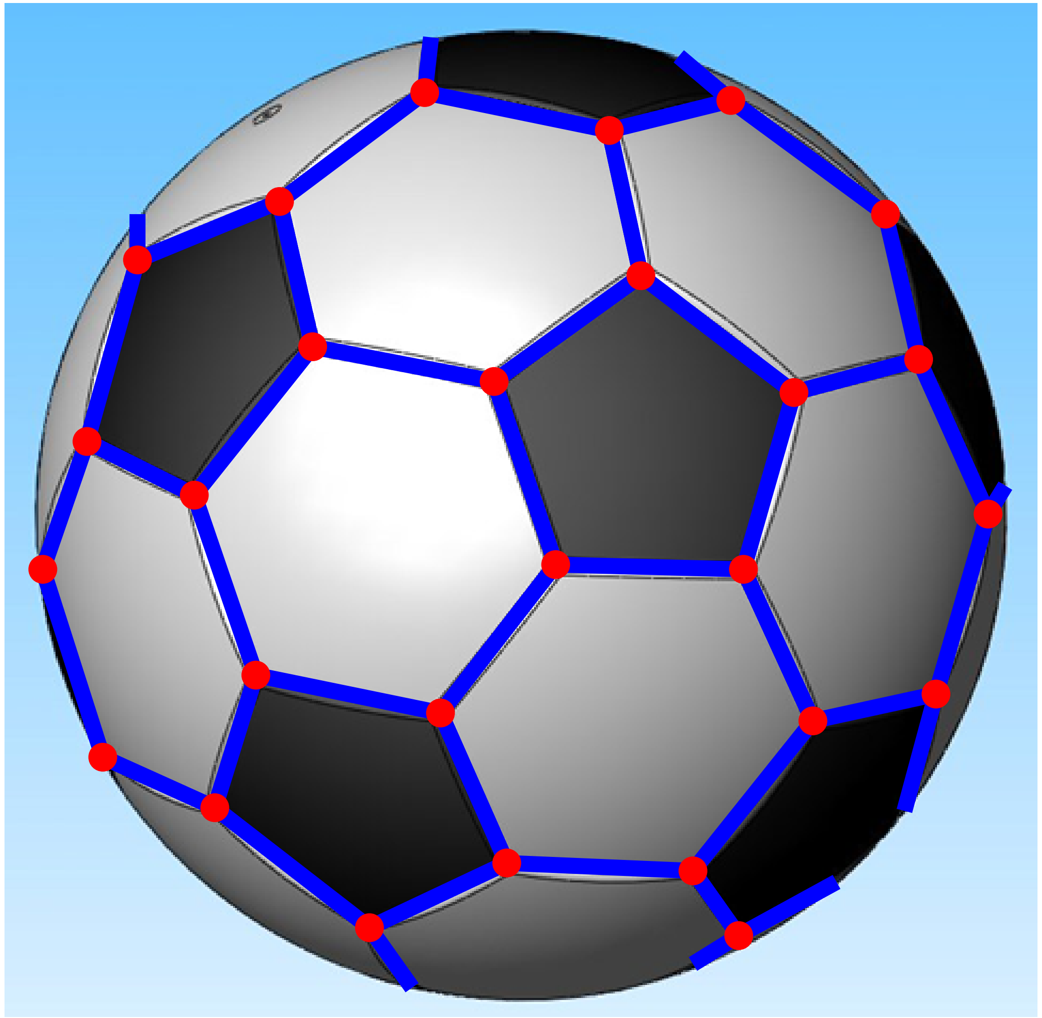
Note: All plane drawings of the same graph have the same number of faces

Plane Graphs: Examples

What is a **Truncated icosahedron** ?

Plane Graphs: Examples

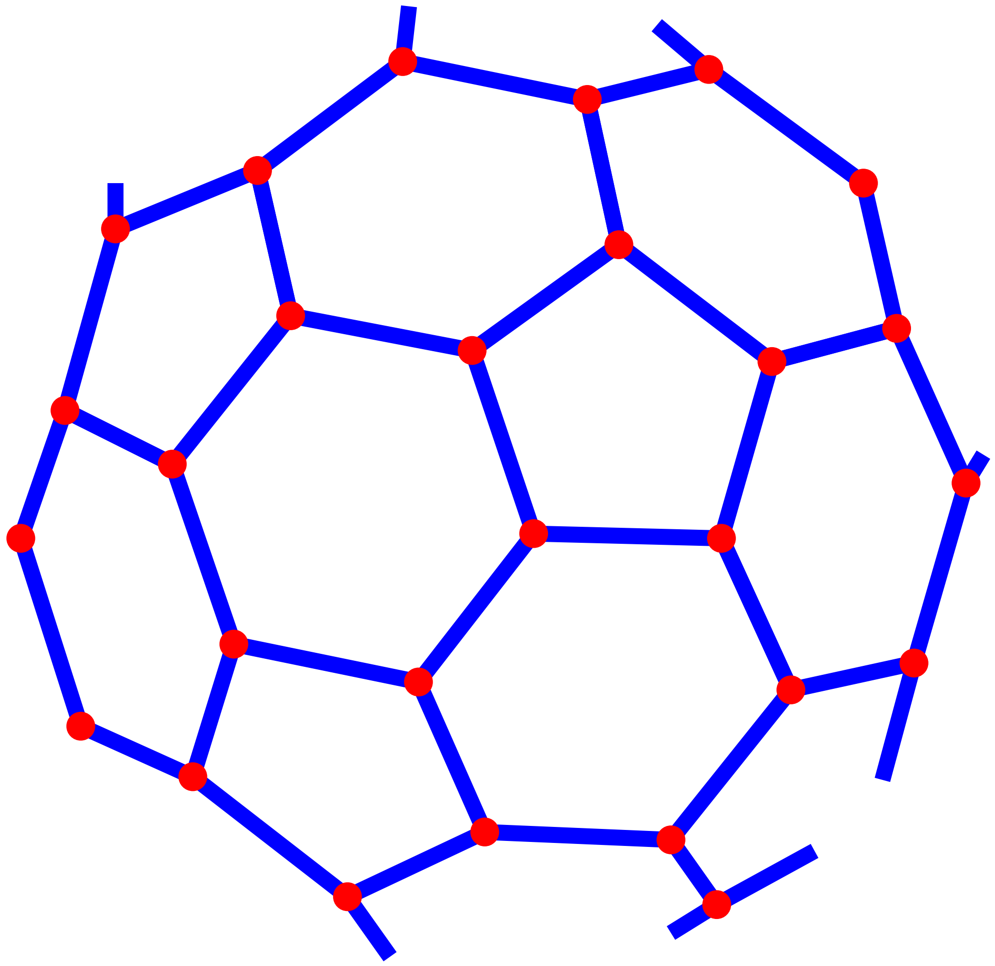
What is a **Truncated icosahedron** ?



vertices: n=60
edges: m=90
faces: f=32

Plane Graphs: Examples

What is a **Truncated icosahedron** ?



vertices: n=60
edges: m=90
faces: f=32

Euler's Formula:
$60 + 32 = 90 + 2$ ✓

Planar Graphs

Theorem: For a planar connected graph $G = (V, E)$ with $|V| = n \geq 3$ and $|E| = m$ it holds that $m \leq 3 \cdot n - 6$

Proof: Consider a plane drawing of G with f faces. Let $e_i,\ i = 1, \ldots, f$ be the number of edges which bound face i.

- $\sum_{i=1}^{f} e_i \geq \sum_{i=1}^{f} 3 = 3 \cdot f$ (since $e_i \geq 3$)
- $\sum_{i=1}^{f} e_i = 2 \cdot m$
  (since each edge is counted twice – once from each side)

$\Rightarrow$ Combine the two results: $3 \cdot f \leq 2 \cdot m$

$\Rightarrow$ Combine with Euler's Formula (multiplied by 3):

$$3 \cdot m + 6 = 3 \cdot n + 3 \cdot f \leq 3 \cdot n + 2 \cdot m$$

$$\Leftrightarrow \qquad m \leq 3 \cdot n - 6 \qquad \checkmark$$

Non-Planarity of K_5

Claim: K_5 is not planar.

Proof:

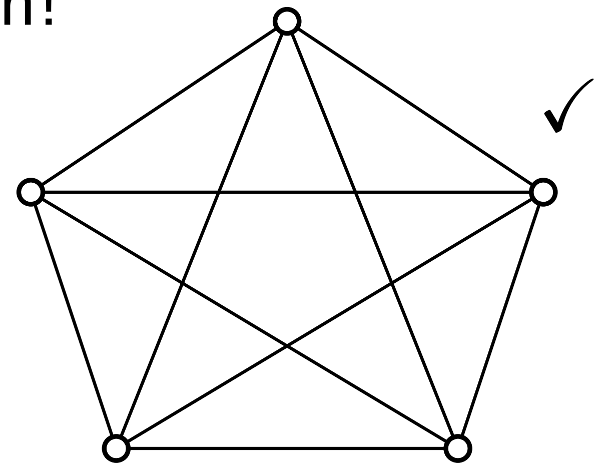
- K_5 has $n = 5$ vertices and $m = \binom{5}{2} = 10$ edges
- Assume K_5 is planar

$\Rightarrow$ We can apply the previous theorem

$$m \leq 3 \cdot n - 6, \quad \text{with } n = 5, m = 10$$

$$10 \leq 3 \cdot 5 - 6 = 9 \quad \text{contradiction!}$$

$\Rightarrow$ K_5 cannot be planar.



Non-Planarity of K_5

Claim: K_5 is not planar.

Proof:

- K_5 has $n = 5$ vertices and $m = \binom{5}{2} = 10$ edges
- Assume K_5 is planar

$\Rightarrow$ We can apply the previous theorem

$m \leq 3 \cdot n - 6, \quad$ with $n = 5, m = 10$

$10 \leq 3 \cdot 5 - 6 = 9 \quad$ contradiction!

$\Rightarrow$ K_5 cannot be planar. ✓

Question:

Can we show in the same way that $K_{3,3}$ is not planar?

Non-Planarity of $K_{3,3}$

Theorem: For a bipartite planar connected graph $G = (V, E)$ with $|V| = n \geq 3$ and $|E| = m$ it holds that $m \leq 2 \cdot n - 4$

Proof: Consider a plane drawing of G with f faces. Let $e_i,\ i = 1, \ldots, f$ be the number of edges which bound face i.

- G bipartite $\Rightarrow$ every face is 2-colorable $\Rightarrow$ $e_i \geq 4$
- $\sum_{i=1}^{f} e_i \geq \sum_{i=1}^{f} 4 = 4 \cdot f, \qquad \sum_{i=1}^{f} e_i = 2 \cdot m$

$\Rightarrow$ Combine the two results: $4 \cdot f \leq 2 \cdot m \Leftrightarrow 2 \cdot f \leq m$

$\Rightarrow$ Combine with Euler's Formula (multiplied by 2):

$$2 \cdot m + 4 = 2 \cdot n + 2 \cdot f \leq 2 \cdot n + m$$

$$\Leftrightarrow \qquad m \leq 2 \cdot n - 4 \qquad \checkmark$$

Non-Planarity of $K_{3,3}$

Claim: $K_{3,3}$ is not planar.

Proof:

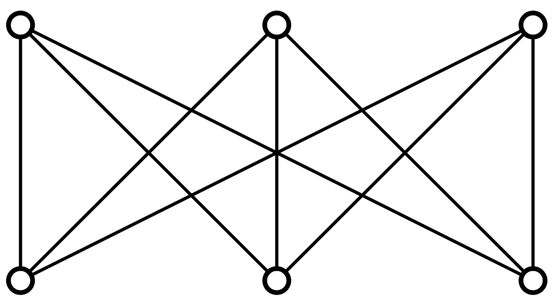
- $K_{3,3}$ has $n = 6$ vertices and $m = 3 \cdot 3 = 9$ edges
- Assume $K_{3,3}$ is planar

$\Rightarrow$ We can apply the previous theorem

$m \leq 2 \cdot n - 4,$ with $n = 6, m = 9$

$9 \leq 2 \cdot 6 - 4 = 8$ contradiction!

$\Rightarrow$ $K_{3,3}$ cannot be planar.



✓

Summary

- Basic graph terminology
- Different ways to store graphs:
  - Adjacency matrix
  - Adjacency list
- Some simple graph algorithms:
  - Breadth first search (BFS), depth first search (DFS)
  - Distances in (unweighted) graphs
  - Recognizing bipartite graphs
- Planar graphs and plane drawings:
  - Kuratowski's Theorem
  - Euler's Formula
  - Non-planarity of $K_{3,3}$ and K_5

Graphs - Final