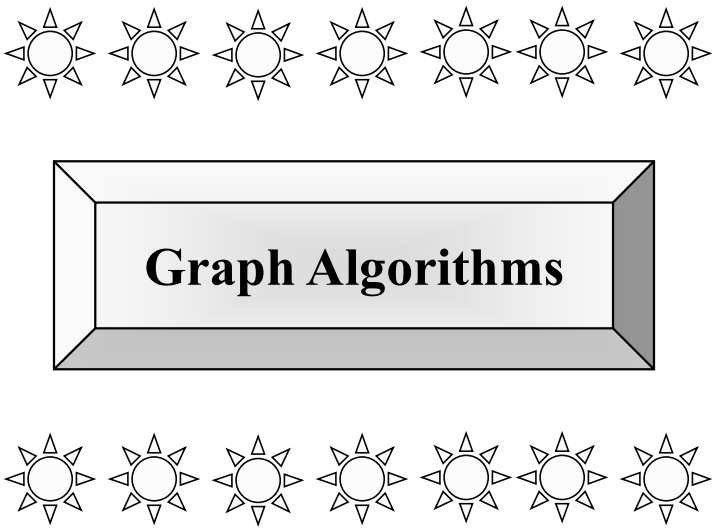


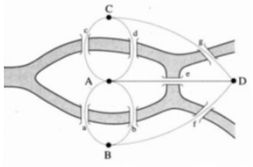
Graph Algorithms

2024 *M. Damrudi*

Introduction

Graph theory started with Euler who was asked to find a nice path across the seven Königsberg bridges

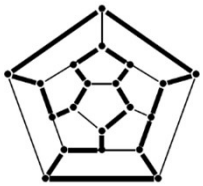
The (Eulerian) path should cross over each of the seven bridges exactly once



Introduction

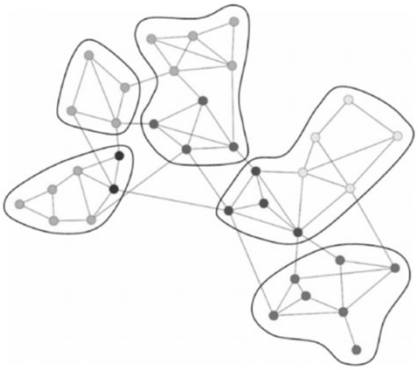
Another early bird was Sir William Rowan Hamilton (1805-1865)

In 1859 he developed a toy based on finding a path visiting all cities in a graph exactly once and sold it to a toy maker in Dublin. It never was a big success.



Introduction

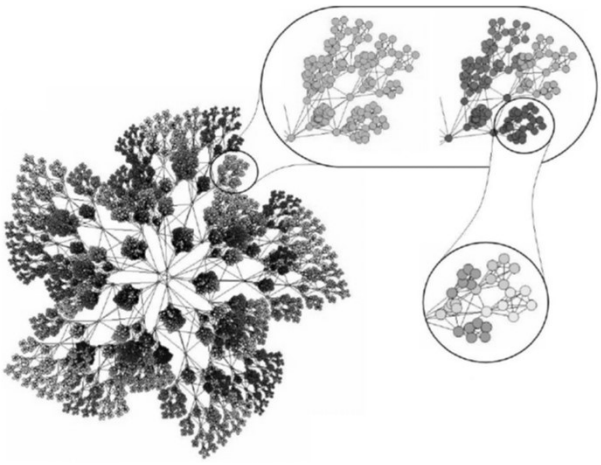
But now graph theory is used for finding communities in networks.



where we want to detect hierarchies of substructures.

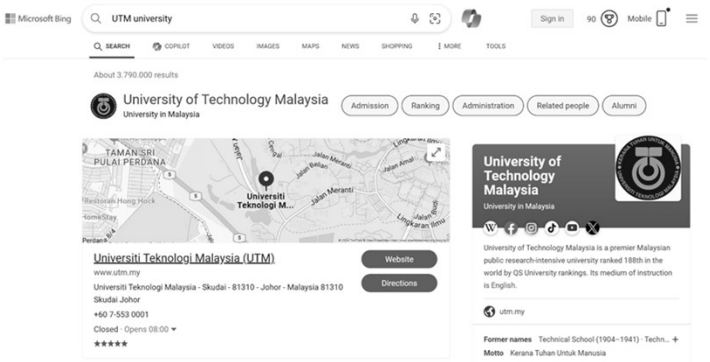
Introduction

and their sizes can become quite big ...



Introduction

It is also used for ranking (ordering) hyperlinks.

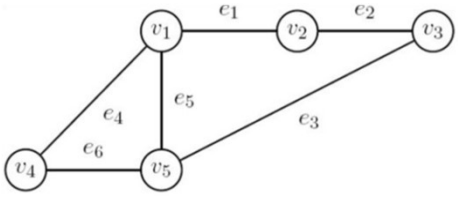


Introduction

or by your GPS to find the shortest path home ...

What are graphs?

A graph G = (V,E) is a pair of vertices (or nodes) V and a set of edges E, assumed finite i.e. |V|= n and |E| = m.
Here V(G) = {v_1, v_2, … ,v_5} and E(G) = {e_1, e_2, …,e_6}.



An edge $e_k = (v_i , v_j)$ is incident with the vertices v_i and v_j .
A simple graph has no self-loops or multiple edges like below

We usually forbid multiple edges between the same pair of vertices, or loop edges between a vertex and itself.

What are graphs?

- *Some properties*

The degree d(v) of a vertex V is its number of incident edges

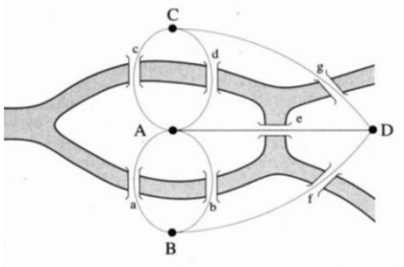
A self-loop counts for 2 in the degree function.

An isolated vertex has degree 0.

Proposition The sum of the degrees of a graph G = (V,E)

equals 2|E|= 2m (trivial)

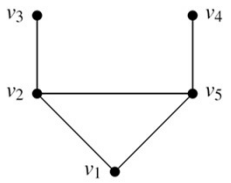
Corollary The number of vertices of odd degree is even (trivial)

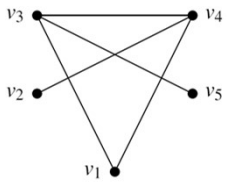


What are graphs?

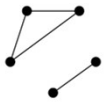
The complement G of a graph G is the graph with all the same vertices, but exactly the edges that don't appear in G.

(For every v,w V(G) with v ≠ w, vw E(G) if and only if vw E(G).)





Special case: the complement of the complete graph is called the empty graph. It has n vertices and no edges.

$K_{3,2}$:  $\overline{K_{3,2}} \cong K_3 \cup K_2$: 

What are graphs?

- *Subgraphs*

A graph H is a subgraph of a graph G if:

All vertices of H are also vertices of G ($V(H) \subseteq V(G)$).

All edges of H are also edges of G ($E(H) \subseteq E(G)$).

By this definition, G itself is a subgraph of G. If H is not all of G (it has fewer vertices, or fewer edges, or both) then we call it a proper subgraph.

A spanning subgraph of a graph G is a subgraph obtained by edge deletions only (so that a spanning subgraph is a subgraph of G with the same vertex set as G).

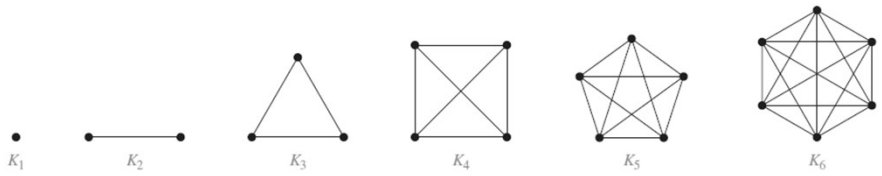
- *Special graphs*

The graph with no vertices (and hence no edges) is the null graph. Any graph with just one vertex is referred to as trivial. All other graphs are nontrivial.

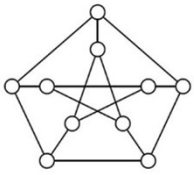
What are graphs?

- *Special graphs*

A complete graph K_n is a simple graph with all n(n-1)/2 possible edges, like the matrices below for n = 2,3, 4, 5.



A k-regular graph is a simple graph with vertices of equal degree k

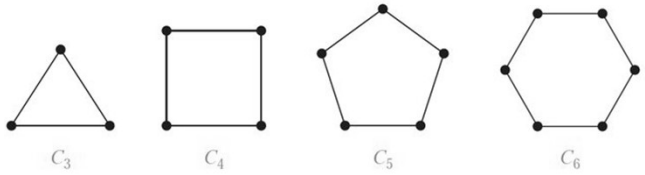


Corollary The complete graph K_n is (n - 1)-regular

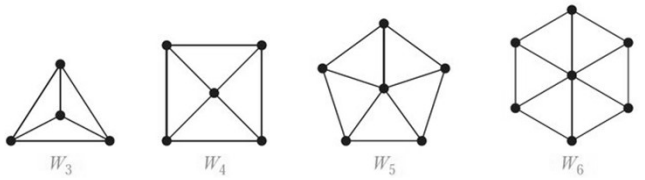
What are graphs?

➢ *Special graphs*

A Cycle graph C_n is a cycle for n ≥ 3 consists of n vertices v_1, v_2,…, v_n, and edges $\{v_1, v_2\}$, $\{v_2, v_3\}$,…,$\{v_{n-1}, v_n\}$, $\{v_n, v_1\}$.



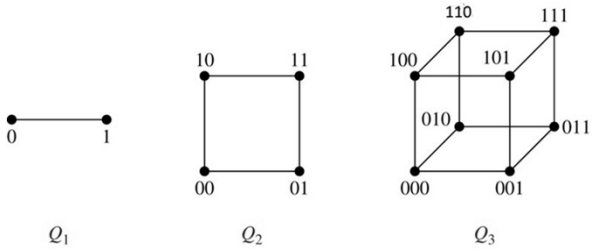
A Wheel graph W_n (n ≥ 3) is obtained from C_n by adding a vertex a inside C_n and connecting it to every vertex in C_n.



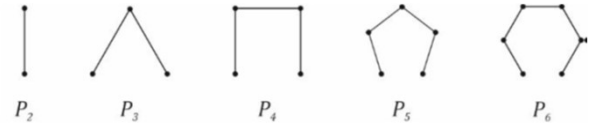
What are graphs?

➢ *Special graphs*

An n−dimensional hypercube, or n−cube, is a graph with 2n vertices representing all bit strings of length n, where there is an edge between two vertices if and only if they differ in exactly one bit position.



The path graph P_n on n vertices is defined for n>2 to be the graph which is obtained by deleting an edge of the circuit graph C_n and P_2=K_2.

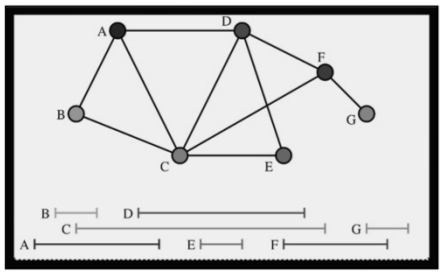


What are graphs?

- *Special graphs*

Interval graph: Given a finite number of intervals on a straight line, a graph associated with this set of intervals can be constructed in the following manner:

each interval corresponds to a vertex of the graph, and two vertices are connected by an edge if and only if the corresponding intervals overlap at least partially.



What are graphs?

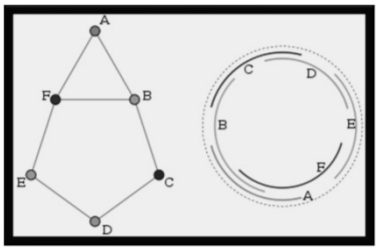
- *Special graphs*
  - Interval graphs are among the most useful mathematical structures for modeling real world problems. The line on which the intervals rest may represent anything that is normally regarded as one dimensional. The linearity may be due to physical restriction, such as blemishes on a microorganism, speed traps on a highway, or files in sequential storage in a computer. If a Hamiltonian circuit must be found, then there are no known efficient algorithms (unless the graph has more structure than just being an interval graph). Also, the speed with which such a problem can be solved will depend partially on whether we are given simply the interval graph G or, in addition, an interval representation of G.
  - Finding a set of intervals that represent an interval graph can also be used as a way of assembling contiguous subsequences in DNA mapping. Interval graphs also play an important role in temporal reasoning.
  - Suppose C_1, C_2, . . . , C_n are chemical compounds which must be refrigerated under closely monitored conditions. If compound C_j must be kept at a constant temperature between t_i and r_j degrees, how many refrigerators will be needed to store all the compounds?

What are graphs?

- *Special graphs*

The intersection graphs obtained from collections of arcs on a circle are called circular-arc graphs. A circular-arc representation of an undirected graph G which fails to cover some point p on the circle will be topologically the same as an interval representation of G. Specifically, we can cut the circle at p and straighten it out to a line, the arcs becoming intervals. It is easy to see, therefore, that every interval graph is a circular-arc graph. The converse, however, is false.

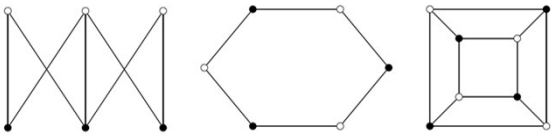
There are applications of circular-arc graphs in areas such as genetics, traffic control and many others.



What are graphs?

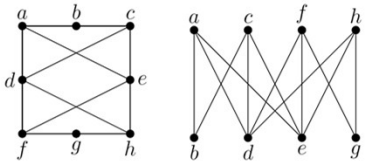
- *Special graphs*

A bipartite graph is one where $V = V_1 \cup V_2$ such that there are no edges between V1 and V2 (the black and white nodes below)



A graph G is bipartite there is a partition $V(G) = A \cup B$ (with $A \cap B = 0$) such that all edges of G have one endpoint in A and one endpoint in B. The pair (A,B) is called the bipartition.

Example: the graph below is a bipartite graph. . .



. . . because we can set A = {a, c, f, h} and B = {b, d, e, g}.

What are graphs?

➢ *Special graphs*

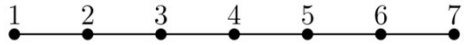
There are two ways a graph can end up being bipartite.

❖ Graphs that are about connections between two types of vertices.

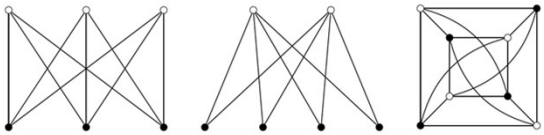
Example: the graph whose vertices are teachers and classes, with an edge between a each teacher and each class they're willing to teach.

❖ Graphs that just happen to be bipartite because of their structure.

Example: a path graph is bipartite because all edges join an even-numbered vertex to an odd-numbered vertex.



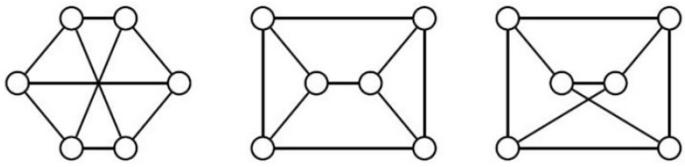
A complete bipartite graph is one where all edges between V_1 and V_2 are present (i.e. $|E| = |V_1|.|V_2|$). It is noted as $K_{n1;n2}$.



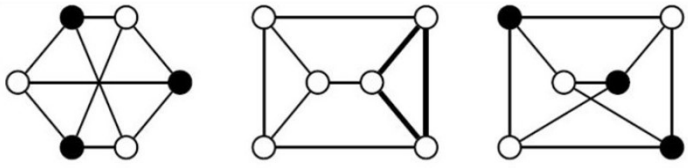
What are graphs?

➢ *When is G bipartite?*

Which graph is bipartite ?



It suffices to find 2 colors that separate the edges as below



The second example is not bipartite because it has a triangle

Cycles

- *Walking in a graph*

A walk of length k from node v_0 to node v_k is a non-empty graph

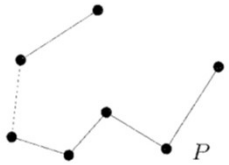
P = (V,E) of the form

$V = \{v_0, v_1, \dots, v_k\}$ $E = \{(v_0, v_1), \dots, (v_{k-1}, v_k)\}$

where edge j connects nodes j - 1 and j (i.e |V|= |E| + 1).

A trail is a walk with all different edges.

A path is a walk with all different nodes (and hence edges).



A walk or trail is closed when $v_0 = v_k$.

A cycle is a walk with different nodes except for $v_0 = v_k$.

Cycles

- *Walking in a graph*

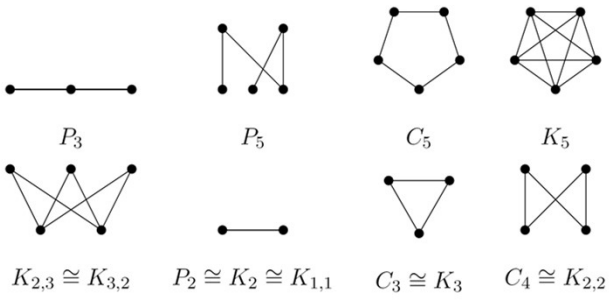
Try to prove the following useful lemmas

Proposition A walk from u to v ≠ u contains a path from u to v

Hint : eliminate sub cycles

Proposition A closed walk of odd length contains a cycle of odd length

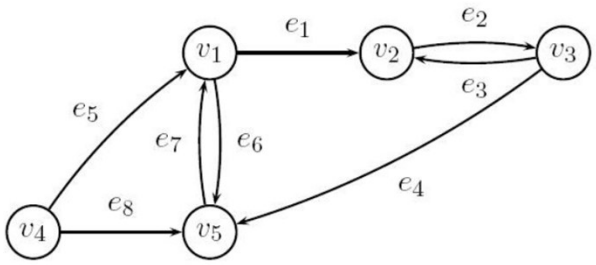
Hint : decompose recursively into distinct subgraphs and use induction



Cycles

- *Directed graphs*

In a directed graph or digraph, each edge has a direction.



For e = (v_s, v_t), v_s is the source node and v_t is the terminal node.

Each node v has an in-degree $d_{in}(v)$ and an out-degree d_{out} (v).

A graph is balanced if $d_{in}(v) = d_{out}$ (v) for all nodes.

Cycles

- *Topological order*

Let us now try to order the nodes in a digraph.

Define a bijection $f_{ord} : V \rightarrow \{1, 2, \ldots, n\}$, then f_{ord} (.) is a topological order for the graph G = (V,E)

$$\text{iff } f_{ord}(i) < f_{ord}(j);\ \forall (i, j) \quad E$$

This is apparently possible for the above graph.

It is easy to see that such a graph should have no cycles.

An acyclic graph is a graph without cycles.

Cycles

➢ *Topological order*

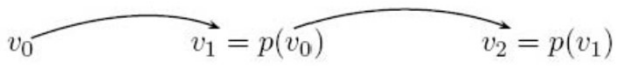
Proposition

Every acyclic graph contains at least one node with zero in-degree.

Proof By contradiction.

Assume $d_{in}(v) > 0$ for all nodes, then each node i has a predecessor p(i) such that $(v_{p(i)}, v_i) \in E$.

Start from an arbitrary v_0 to form a list of predecessors as below



Since |V| is bounded, one must eventually return to a node that was already visited, hence there is a cycle.

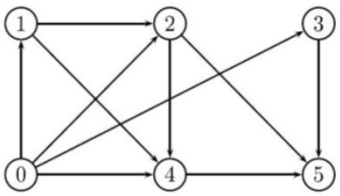
Cycles

➢ *Topological order*

Let us use this to find a topological order

Algorithm FindTopOrd(G)
t := 0;G^0 := G;
while ∃ v G^t : $d_{in}(v) = 0$ **do**
$G^{t+1} := G^t/\{v\}$; order(v) := t + 1; t := t + 1;
end while
if t = n **then** G is acyclic;
else if t < n **then** G has a cycle; **end if**
end if

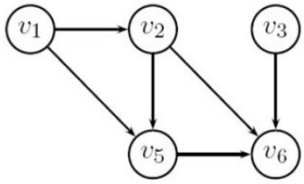
Let us verify this algorithm on this example.



Cycles

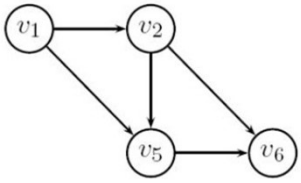
➢ *Topological order*

The only node of in-degree 0 is v_4. So for t = 1 we have



After removing v_4 there are two nodes of in-degree 0, v_1 and v_3.

If we pick v_3 then we have for t = 2



Further reductions yield the final order {v_4, v_3, v_1, v_2, v_5, v_6}.

The Havel Hakimi theorem

➢ *Havel Hakimi theorem*

Theorem (Havel 1957, Hakimi 1963). A sequence (d_1, d_2, ... , d_n)sorted in decreasing order is graphic if and only if the sequence

$$(\underbrace{d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1}_{d_1 \text{ terms}}, d_{d_1+2}, \ldots, d_n)$$

is graphic.

This lets us reduce the problem to a smaller problem, giving us a recursive algorithm to test if a sequence is graphic. (We'll see examples.)

Proof of (=. If the second sequence is graphic, take a graph on vertices v_2, v_3, . . . , v_n with that degree sequence.

Then add a vertex v_1 adjacent to v_2, v_3,. . . , v_{d1+1}.

The Havel Hakimi theorem

- *Havel Hakimi theorem*

Let's test the sequence (5, 5, 5, 4, 2, 1, 1, 1) for being graphic.

- Delete 5 and subtract 1 from the next 5 numbers, getting (5-1, 5-1, 4-1, 2-1, 1-1, 1, 1).
- Simplify to (4, 4, 3, 1, 0, 1, 1) and sort to (4, 4, 3, 1, 1, 1, 0).
- Delete 4 and subtract 1 from the next 4 numbers, getting (4-1, 3-1, 1-1, 1-1, 1, 0).
- Simplify to (3, 2, 0, 0, 1, 0) and sort to (3, 2, 1, 0, 0, 0).
- Delete 3 and subtract 1 from the next 3 numbers, getting (2-1, 1-1, 0-1, 0, 0).
- We got a negative number: the sequence isn't graphic.

The Havel Hakimi theorem

- *Havel Hakimi theorem*

Let's test the sequence (3, 3, 3, 3, 2) for being graphic.

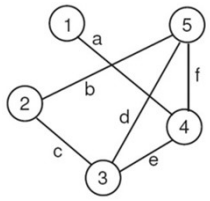
- Delete 3 and subtract 1 from the next 3 numbers, getting (3-1, 3-1, 3-1, 2).
- Simplify to (2, 2, 2, 2).
- Delete 2 and subtract 1 from the next 2 numbers, getting (2-1, 2-1, 2).
- Simplify to (1, 1, 2) and sort to (2, 1, 1).
- Delete 2 and subtract 1 from the next 2 numbers, getting (1-1, 1-1).
- Simplify to (0, 0) and realize that this is graphic.

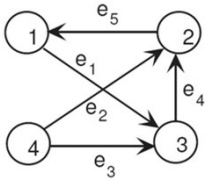
Representing graphs

➢*Representing graphs*

A graph G = (V,E) is often represented by its adjacency matrix.

It is an n×n matrix A with A(i, j) = 1 iff (i, j) E. For the graphs





the adjacency matrices are

$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Representing graphs

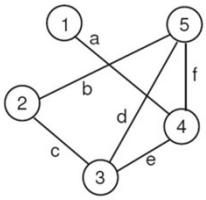
➢*Representing graphs*

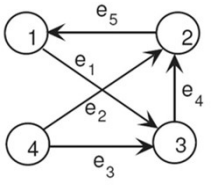
A graph can also be represented by its n×m incidence matrix T.

For an undirected graph T(i, k) = T(j, k) = 1 iff $e_k = (v_i , v_j)$.

For a directed graph T(i, k) = -1; T(j, k) = 1 iff $e_k = (v_i , v_j)$.

For the graphs





the incidence matrices are

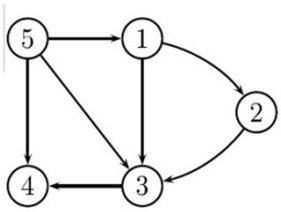
$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad T_2 = \begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & -1 \\ 1 & 0 & 1 & -1 & 0 \\ 0 & -1 & -1 & 0 & 0 \end{bmatrix}$$

Representing graphs

- *Representing graphs*

One can also use a sparse matrix representation of A and T.

This is in fact nothing but a list of edges, organized e.g. by nodes.



$$V(1) = \{2,3\}$$
$$V(2) = \{3\}$$
$$V(3) = \{4\}$$
$$V(4) = \emptyset$$
$$V(5) = \{1,3,4\}$$

Notice that the size of the representation of a graph is thus linear in the number of edges in the graph (i.e. in m = |E|).

To be more precise, one should count the number of bits needed to represent all entries :

$$L=(n+m)\log n$$

since one needs log n bits to represent the vertex pointers.

Representing graphs

- *Counting degrees*

Let 1 be the vector of all ones, then $d_{in} = A^T 1$ and $d_{out} = A1$ are the vectors of in-degrees and out-degrees of the nodes of A and $d_{out} = d_{in} = d$ for undirected graphs.

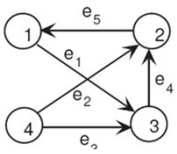
How should we then take self-loops into account ?

In an adjacency matrix of an undirected graph A(i, i) = 2

In an adjacency matrix of a directed graph A(i, i) = 1

For an undirected graph, we have d = T1.

For a directed graph one can define T_t and T_s as the matrices containing the terminal and source nodes : $T = T_t - T_s$ with



$$T_t := \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, T_s := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Then also we have $d_{in} = T_t 1$ and $d_{out} = T_s 1$.

Representing graphs

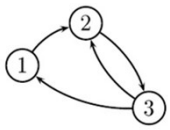
➢ *Powers of A*

Proposition $(A^k)_{ij}$ is the number of walks of length k from i to j

Proof Trivial for k=1; by induction for larger k.

The element (i, j) of $A^{k+1} = A^k .A$ is the sum of the walks of length k to nodes that are linked to node j via the adjacency matrix A.

One verifies this in the following little example



$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Corollary In a simple undirected graph one has the identities

tr=the trace of a matrix is he sum of the diagonal elements of the matrix.

tr (A) = 0; tr $(A^2)/2$ = |E| and

tr $(A^3)/6$ equals the number of triangles in G.

Isomorphism

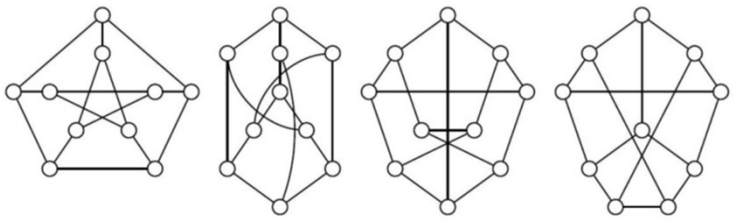
➢ *Isomorphic graphs*

Sometimes we want to say that two graphs have the same structure.

For instance, even though there are many possibly connected 2-regular graphs on n vertices, they all look like C_n.

Intuitively, we can "relabel" any of these graphs to get C_n.

Two graphs G_1 and G_2 are isomorphic iff there is a bijection between their respective nodes which make each edge of G_1 correspond to exactly one edge of G_2, and vice versa.



One must find a label numbering that makes the graphs identical.

This problem is still believed to be NP hard.

Isomorphism

➢ *Isomorphic graphs*

Definition: Isomorphism of Two Graphs. An isomorphism of two graphs G and H is a bijective function f : V(G)→V(H) such that any two vertices u and v of G are adjacent in G if and only if f(u) and f(v) are adjacent in H.

That is, two graphs G and H are said to be isomorphic if

(i) |V(G) | = |V(H)|,

(ii) |E(G)| = |E(H)|,

(iii) $v_i v_j$ E(G) => f (v_i) f (v_j) E(H).

This bijection is commonly described as edge-preserving bijection.

If an isomorphism exists between two graphs, then the graphs are called isomorphic graphs and denoted as G H or G H

Application:

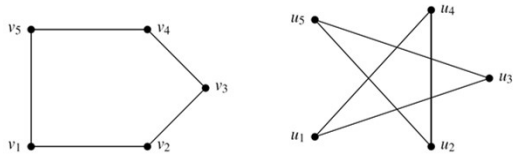
❖Identification of similar molecules

❖Pattern Recognition and Computer Vision.

Isomorphism

➢ *Isomorphic graphs*

For example, consider the graphs given in Figure



In the above graphs, we can define an isomorphism f from the first graph to the second graph such that f (v_1) = u_1, f (v_2) = u_3, f (v_3) = u_5, f (v_4) = u_2 and f (v_5) = u_4. Hence, these two graphs are isomorphic.

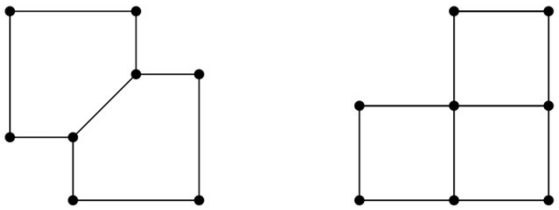
❖ Application: In chemistry, to find if two compounds have the same structure

One of the most important practical application of graph isomorphism lies in information retrieval from chemical databases, since every database of chemical compounds must offer some kind of chemical structure and substructure search system, let alone bibliographical searches in specialized databases or over the Internet. Several such chemical information retrieval systems based on graph isomorphism and subgraph isomorphism.

Isomorphism

- *Isomorphic graphs*

Are these two graphs isomorphic?



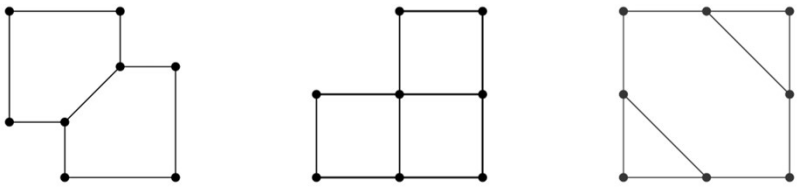
If the answer were "yes", we could try to prove it by finding an isomorphism. After we try it for a while and it doesn't work, we suspect the answer is "no", but that's not a proof!

We can prove the answer is isomorphic by finding a graph property that distinguishes them. In this case, the two graphs are distinguished by the number of edges: the left one has 9, and the right one has 10.

Isomorphism

- *Isomorphic graphs*

Is the third graph isomorphic to either of the first two?



It has 10 edges, so if it's isomorphic to any of them, it's isomorphic to the second one. But there are a few ways to distinguish it from the second one as well:

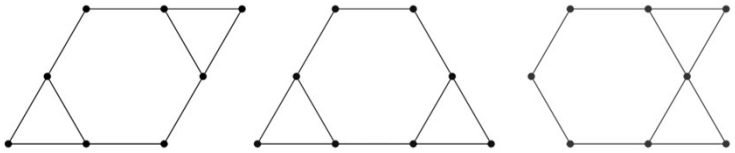
The second graph has a vertex of degree 4, but the third doesn't.

The second graph is bipartite, but the third graph isn't. (It contains an odd cycle!)

Isomorphism

- *Isomorphic graphs*

What about these three graphs?



The third graph is different because it has a vertex of degree 4. The first and second have the same degree sequence. . .

If the first two graphs are isomorphic, the degrees hint at how: degree-2 vertices in one graph should go to degree-2 vertices in the other.

This is impossible! In the blue graph, two degree-2 vertices are adjacent. No such vertices exist in the black graph.

Isomorphism

- *Subgraph Isomorphism*

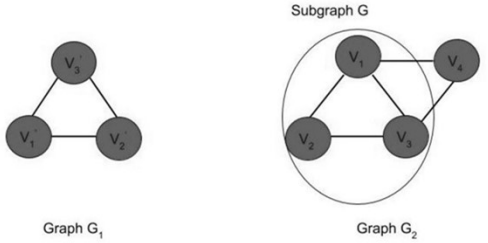
An important generalization of graph isomorphism is known as subgraph isomorphism. The subgraph isomorphism problem is to determine whether a graph is isomorphic to a sub graph of another graph, and is a fundamental problem with a variety of applications in engineering sciences, organic chemistry, biology, and pattern matching.

The problem of determining whether or not a graph is isomorphic to a subgraph of another graph belongs to the class of NP-complete problems, meaning that all known algorithms for solving the sub graph isomorphism problem upon general graphs (that is, graphs without any restriction on any graph parameter) take time exponential in the size of the graphs, and that it is highly unlikely that such an algorithm will be found which takes time polynomial in the size of the graphs.

Isomorphism

- *Subgraph Isomorphism*

Most practical applications of sub graph isomorphism require not only determining whether or not a given graph is isomorphic to a subgraph of another given graph, but finding all sub graphs of a given graph which are isomorphic to another given graph. Already the number of sub graph isomorphisms of a graph into another graph can be exponential in the size of the graphs, though.



Graph G, a subgraph of G2 is isomorphic to G1. f : (V_1, V_3') , (V_2, V_1') , (V_3, V_2') is a bijection.

Isomorphism

- *Subgraph Isomorphism*

Possible Approaches

- ❖ Brute Force
- ❖ Ulmanns backtracking algorithm (1976)
- ❖ Nauty by Brendan McKay (1981)
- ❖ VF algorithm by Cordella et all. (1998)
- ❖ VF2 by Cordella et al. (2001)

Applications

- ❖ Pattern Recognition in Bio-Informatics & BioComputing.
- ❖ Image Processing & Computer Vision
- ❖ Identification of sub-compound molecules of a given molecule.
- ❖ Recognition of distorted shapes.

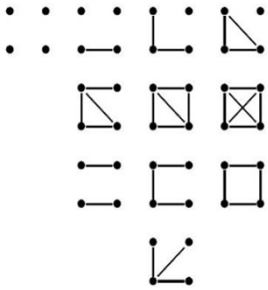
Isomorphism

- *Counting graphs*

How many different simple graphs are there with n nodes ?

A graph with n nodes can have n(n - 1)/2 different edges and each of them can be present or not.

Hence there can be at most $2^{n(n-1)/2}$ graphs with n nodes.



For n = 3 only 4 of the graphs are different (omitting the isomorphic ones)

With n = 4 one finds eventually 11 different graphs after collapsing the isomorphic ones

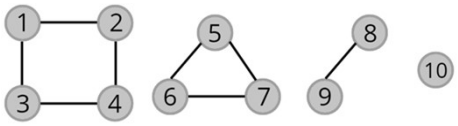
Connectivity

- *Connected components*

Definition*:* Connectedness in a Graph. Two vertices u and v are said to be connected if there exists a path between them. If there is a path between two vertices u and v, then u is said to be reachable from v and vice versa. A graph G is said to be connected if there exist paths between any two vertices in G.

Definition: Component of a Graph. A connected component or simply, a component of a graph G is a maximal connected subgraph of G.

Basic Idea: In a Graph Reachability among vertices by traversing the edges



4 connected component

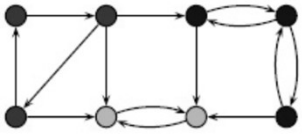
Application Example:

- In a city to city road-network, if one city can be reached from another city.
- Problems if determining whether a message can be sent between two computer using intermediate links.
- Efficiently planning routes for data delivery in the Internet

Connectivity

- *Connected components*

In a directed graph G = (V,E), u and v are strongly connected if there exists a walk from u to v and from v to u. This is an equivalence relation and hence leads to equivalence classes, which are called the connected components of the graph G. SCC: Strongly Connected Component



The graph reduced to its connected components is acyclic.

This shows up in many applications, e.g. in the dictionary graph.

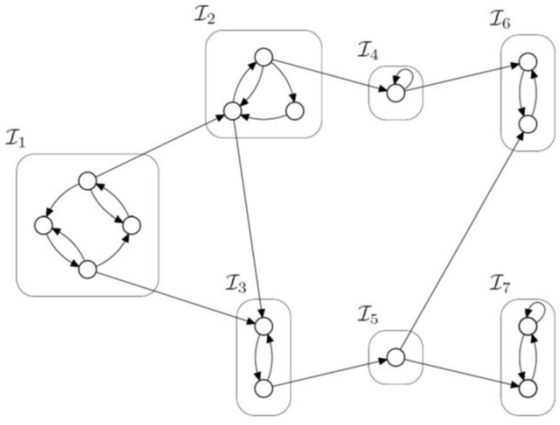
The connected components are the groups of words that use each other in their definition.

After the reduction one has an acyclic graph, which can be ordered topologically.

Connectivity

- *Connected components*

What do you obtain then ? Class orderings.



An initial class has $d_{in}(c) = 0$. A final class has $d_{out}(c) = 0$.

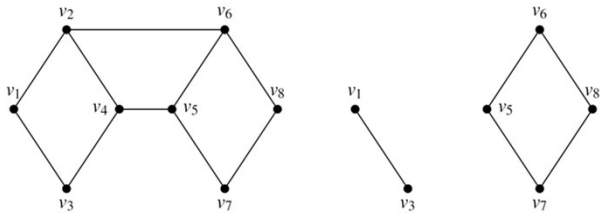
The other ones are intermediate.

Connectivity

- *Connected components*

First recall that a cut-vertex of a graph G is a vertex v in G such that G-v is disconnected. A cut-vertex is also called a cut-node or an articulation point.

Definition: Vertex-Cut. A subset W of the vertex set V of a graph G is said to be a vertex-cut or a Separating Set of G if G-W is disconnected.



Connected graph G Disconnected graph G-$\{v_2,v_4\}$

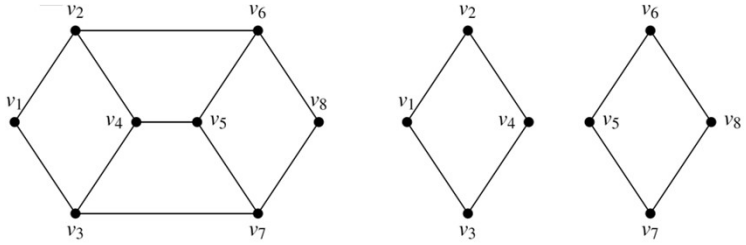
disconnected graph G-$\{v_4v_5,v_2v_6,v_3v_7\}$

Connectivity

- *Connected components*

Recall that an edge e of a graph G is a cut-edge of G if G-e is disconnected.

Definition: Edge-Cut. A subset F of the edge set E of a graph G is said to be an edge-cut of G if G-E is disconnected.



Disconnected graph G-$\{v_4v_5,v_2v_6,v_3v_7\}$

The edge set F =$\{v_4v_5,v_2v_6,v_3v_7\}$ is an edge-cut, since the removal of F makes G disconnected.

Connectivity

- *Connected components*

Definition: Cut-Set. A cut-set is a minimal edge-cut of G. That is, a cut-set of a graph G is a set of edges F of G whose removal makes the graph disconnected, provided the removal of no proper subset of F makes G disconnected.

A cut-set also called a *minimal cut-set*, a *proper cut-set*, a *simple cut-set*, or a *cocycle*.

Note that the edge-cut F in the above example is a cut-set of G.

Definition: Separable Graph. A connected graph G (or a connected component of a graph) is said to be a separable graph if it has a cut-vertex.

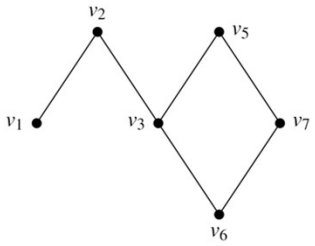
Definition: Non-separable Graph. A connected graph or a connected component of a graph, which is not separable, is called non-separable graph.

Definition: Block. A non-separable subgraph of a separable graph G is called a block of G.

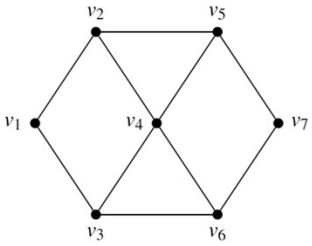
A connected graph that has no cut vertices is called a block. Every block with at least three vertices is 2-connected.

Connectivity

- *Connected components*

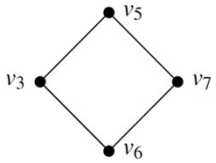


separable graph



A non-separable graph

v_2 and v_3 are cut-vertices. Hence, the three blocks of that graph are given in figure below:



Connectivity

- *Connected components*

Definition: Edge Connectivity of a Graph. Let G be a graph (may be disconnected) having k components. The minimum number of edges whose deletion from G increases the number of components of G is called the edge connectivity of G. The edge connectivity of G is denoted by (G).

Definition: Vertex Connectivity of a Graph. Let G be a graph (may be disconnected).The minimum number of vertices whose deletion from G increase the number of components of G is called the vertex connectivity of G. The vertex connectivity of G is denoted by (G).

Definition: A graph G is said to be *k-connected* if its vertex connectivity is k (That is, if (G) = k). A graph G is k-edge connected if its edge connectivity is k (That is, if (G) = k.)

THEOREM The edge connectivity of a graph G cannot exceed the degree of the vertex with the smallest degree in G.

THEOREM The vertex connectivity of any graph G can never exceed the edge connectivity of G.

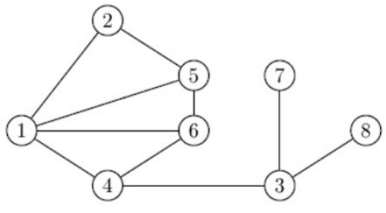
Connectivity

- *Connected components*

Verify (strong) connectivity of a graph based on its adjacency list

Idea : start from node s, explore the graph, mark what you visit

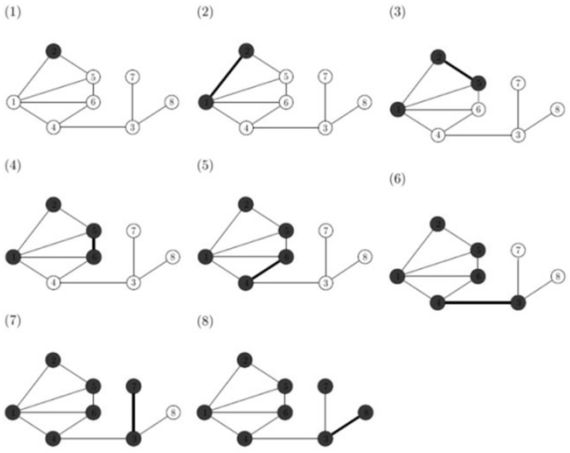
Algorithm GenericSearch(G,s)
mark(s); L := {s}
while L ≠ **do**
  choose u L;
  if ∃(u, v) such that v is unmarked **then**
    mark(v); L := L U {v};
  else
    L := L\{u};
  end if
end while



$V(1) = \{2, 4, 5, 6\}$
$V(2) = \{1, 5\}$
$V(3) = \{4, 7, 8\}$
$V(4) = \{1, 3, 6\}$
$V(5) = \{1, 2, 6\}$
$V(6) = \{1, 4, 5\}$
$V(7) = \{3\}$
$V(8) = \{3\}$

Connectivity

- *Connected components*



L	*mark*
{2}	2
{2, 1}	1
{2, 1, 5}	5
{2, 1, 5, 6}	6
{1, 5, 6}	
{1, 5, 6, 4}	4
{5, 6, 4}	
{5, 4}	
{5, 4, 3}	3
{5, 3}	
{5, 3, 7}	7
{5, 3}	
{3}	
{3, 8}	8
{3}	
{}	

The chosen nodes and the discovered nodes are marked.

This algorithm has 2n steps : each node is added once and removed once. Its complexity is therefore linear in n.

Connectivity

- *Connected components*

Because of the choices, this algorithm allows for different versions.

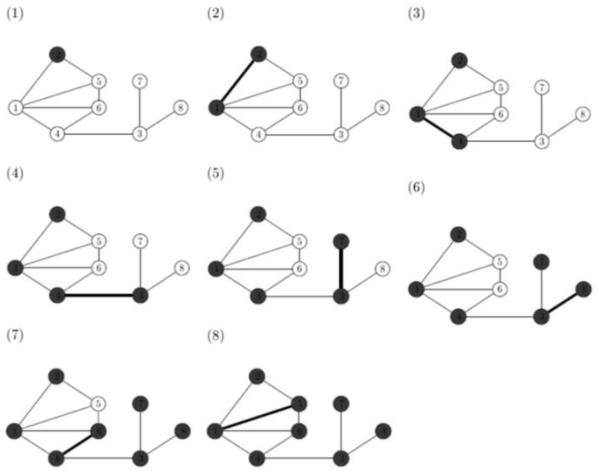
Let us use a LIFO list for L (Last In First Out) and choose for u the last element added to L. This is a depth first search (DFS).

```
Algorithm DeptFirstSearch(G,s)
mark(s); L := {s};
while L ≠  do
    u := last(L)
    if ∃(u, v) such that v is unmarked then
       choose (u, v) with v of smallest index;
       mark(v); L := L U {v};
   else
       L := L\{u}
   end if
end while
```

Connectivity

- *Connected components*

L	*mark*
{2}	2
{2, 1}	1
{2, 1, 4}	4
{2, 1, 4, 3}	3
{2, 1, 4, 3, 7}	7
{2, 1, 4, 3}	
{2, 1, 4, 3, 8}	8
{2, 1, 4, 3}	
{2, 1, 4}	
{2, 1, 4, 6}	6
{2, 1, 4, 6, 5}	5
{2, 1, 4, 6}	
{2, 1, 4}	
{2, 1}	
{2}	
{}	



The chosen nodes and the discovered nodes are marked.

The depth first algorithm builds longer paths than the generic one.

Connectivity

- *Connected components*

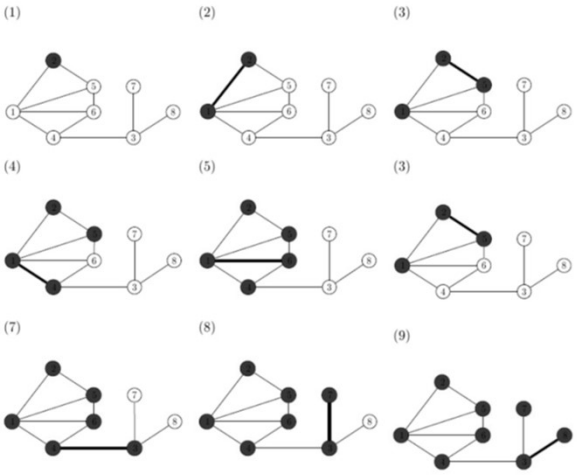
We now use a FIFO list for L (First In First Out) and choose for u the first element added to L. This is a breadth first search (BFS).

Algorithm BreadthFirstSearch(G,s)
mark(s); L := {s};
while L ≠ **do**
  u := first(L)
  if ∃(u, v) such that v is unmarked **then**
    choose (u, v) with v of smallest index;
    mark(v); L := L U {v};
  else
    L := L\{u}
  end if
end while

Connectivity

- *Connected components*

L	*mark*
{2}	2
{2,1}	1
{2,1,5}	5
{1,5}	
{1,5,4}	4
{1,5,4,6}	6
{5,4,6}	
{4,6}	
{4,6,3}	3
{6,3}	
{3}	
{3,7}	
{3,7,8}	8
{7,8}	
{8}	
{}	



The chosen nodes and the discovered nodes are marked.

The breadth first algorithm builds a wider tree.

Connectivity

- *Applications*

An Application: Suppose we are given n stations that are to be connected by means of e lines (telephone lines, bridges, railroads, tunnels, or highways) where $e \geq n - 1$.

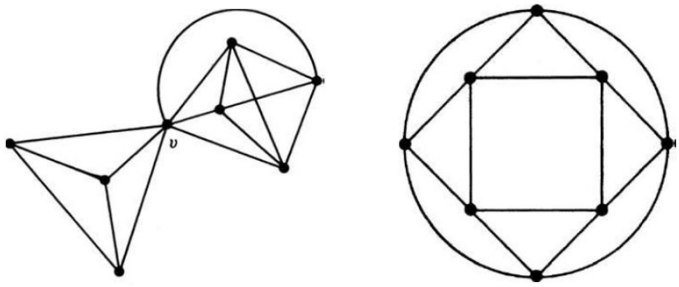
What is the best way of connecting? By "best" we mean that the network should be as invulnerable to destruction of individual stations and individual lines as possible. In other words, construct a graph with n vertices and e edges that has the maximum possible edge connectivity and vertex connectivity.

For example, the graph in Fig. A has n = 8, e = 16, and has vertex connectivity of one and edge connectivity of three.

Another graph with the same number of vertices and edges (8 and 16, respectively) can be drawn as shown in Fig. B.

Connectivity

- *Applications*



A B

It can easily be seen that the edge connectivity as well as the vertex connectivity of graph B is four. Consequently, even after any three stations are bombed, or any three lines destroyed, the remaining stations can still continue to "communicate" with each other. Thus the network of Fig. B is better connected than that of Fig. A.

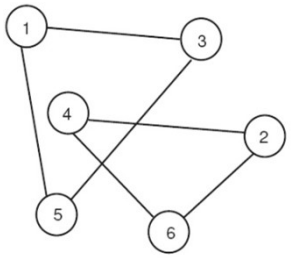
Connectivity

- *Testing connectivity*

The exploration algorithm finds the set of all nodes that can be reached by a path from a given node u b.

If the graph is undirected, each node in that set can follow a path back to u. They thus form the connected component C(u) of u.

To find all connected components, repeat this exploration on a node of V\C(u), etc.



Connectivity

➢ *Testing strong connectivity*

Proposition Let G = (V, E) be a digraph and let u V.

If ∀v V there exists a path from u to v and a path from v to u, then G is strongly connected.

The exploration algorithm finds the set of all nodes that can be reached by a path from a given node u V.

How can one find the nodes from which u can be reached?

Construct for that the inverse graph by reversing all arrows. Show that the adjacency matrix of this graph is just A^T .

Connectivity

➢ *Testing strong connectivity*

Proposition Let G = (V, E) be a digraph and let u V.

Let $R_+(u)$ be the nodes that can be reached from u and let $R_-(u)$ be the nodes that can reach u, then the strongly connected component of u is $C(u) = R_+(u) \cap R_-(u)$

The exploration algorithm applied to the inverse graph, starting from u finds the set $R_-(u)$

Here $R_+(v_6) = \{4, 6\}$ while $R_-(v_6) = V$ hence $C(v_6) = \{4, 6\}$

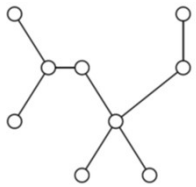
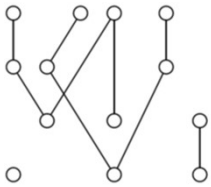
Find the other connected components.

Trees

- *Trees and forests*

A tree is an acyclic and connected graph.

A forest is an acyclic graph (and hence a union of trees)

tree   **forest**

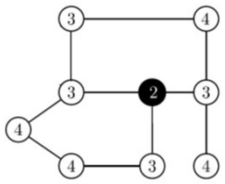
Proposition For a graph G = (V,E) of order n = |V|, the following are equivalent

1. G is connected and has n - 1 edges
2. G is acyclic and has n - 1 edges (|V|=|E|+1)
3. G is connected and acyclic
4. u, v V there is one and only one path from u to v
5. G is acyclic and adding an edge creates one and only one cycle
6. G is connected and removing an arbitrary edge disconnects it

Trees

- *Trees and forests*

eccentricity



radius

diameter

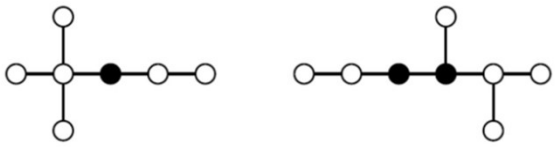
center

Trees

➢ *Trees and forests*

A leaf of a tree T is a node of degree 1

Proposition Let T be a tree and let T' be the tree obtained by removing all its leafs, then (T') = (T) - 1 for all nodes of T'.



Proposition The center of a tree is a single node or a pair of adjacent nodes.

Proof By induction using the previous proposition. Show that the center does not change.

Theorem Any tree with at least two vertices has at least two pendant vertices.

Theorem A vertex v in a tree is a cut-vertex of T if and only if d(v) ≥ 2.

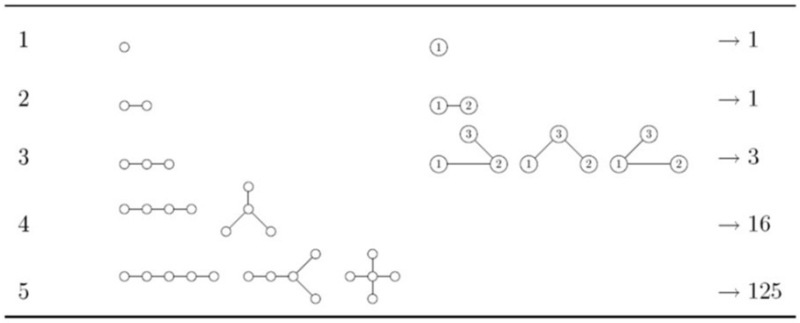
Trees

➢ *Counting trees*

How many different (labeled) trees are there with n nodes ?

The following table gives the count for small n.

The following theorem of Cayley gives the exact formula.



Proposition

The number of distinct labeled trees of order n equals n^{n-2}.

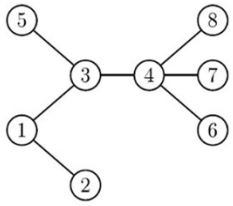
Trees

- *Counting trees*

We construct a bijection of T_n with a sequence via the algorithm

Algorithm PrüferSequence(T)
s := (); t := ();
while |E| > 1 **do**
  choose the leaf of smallest index i;
  T := T\{i}; s := (s, i); t := (t, neighbour(i));
end while

On the graph below, it yields the table next to it.



i	s_i	t_i
1	2	1
2	1	3
3	5	3
4	3	4
5	6	4
6	7	4

One shows that the graph can be reconstructed from the sequence t_i which are n - 2 numbers from {1,….,n} and there are exactly n^{n-2} such sequences.

Trees

- *Counting trees*

Given a Prüfersequence, it is possible to generate a finite labeled tree corresponding to that sequence.

Let $P=(a_1,a_2,\ldots,a_{n-2})$ be a Prüfersequence. This will be called **the sequence**.

It is assumed the sequence is not empty.
Step 1: Draw the nodes of the tree we are to generate, and label them from 1 to n. This will be called the tree.
Step 2: Make a list of all the integers (1,2,…,n). This will be called the list.
Step 3: If there are two numbers left in the list, connect them with an edge and then stop. Otherwise, continue on to step 4.
Step 4: Find the smallest number in the list which is not in the sequence, and also the first number in the sequence. Add an edge to the tree connecting the nodes whose labels correspond to those numbers.
Step 5: Delete the first of those numbers from the list and the second from the sequence. This leaves a smaller list and a shorter sequence. Then return to step 3.

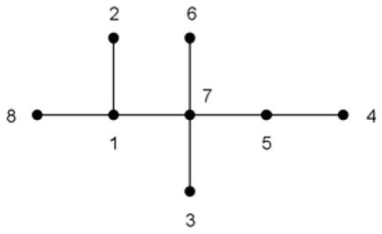
Trees

- *Counting trees*

Consider the following tree.

1- Construct the Prüfersequence.

2- Construct the labled tree from the Prüfersequence.

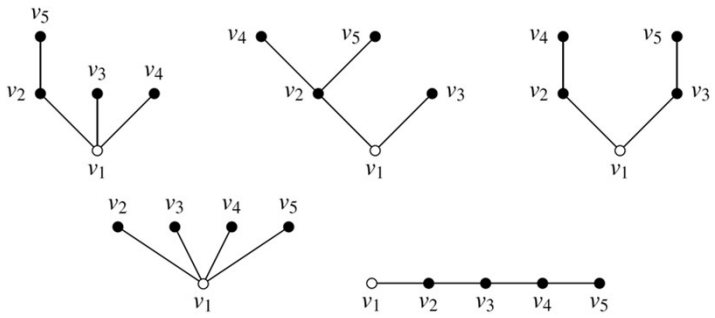


Trees

- *Rooted tree*

Definition Rooted Tree. A rooted tree is a tree T in which one vertex is distinguished from all other vertices. This particular vertex is called the root of T.

The figure illustrates some rooted trees on five vertices. In all these graphs, the white vertices represent the roots of the rooted trees concerned.

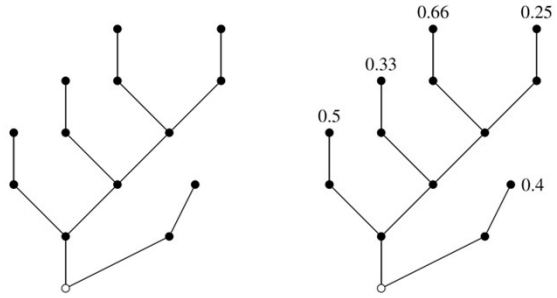


Trees

- *Rooted tree*

In certain practical or real-life problems, we may have to calculate the lengths or total lengths of vertices of rooted trees from the roots. In some cases, we may also need to assign weights to the vertices of a tree.

Consider the following binary tree whose pendant vertices are assigned some weights.



A rooted tree with and without weights to pendant vertices.

Trees

- *Rooted tree*

Definition Path Length of a Rooted Tree. The path length or (external path length) of a rooted tree T is the sum of the levels of all pendant vertices.

The path lengths of rooted trees are widely applied in the analysis of algorithms.

The path length of the first rooted tree in Figure is 2+3+4+5+5 = 19.

Definition Weighted Path Length of a Rooted Tree. If every pendant vertex v_i of a tree T is assigned some positive real number w_i, then the weighted path length of T is defined as

where l_i is the level of the vertex v_i from the root.

The weighted path length of the graph in Figure is 2×0.4+3×0.5+4×0.33+ 5(0.66+0.25) = 8.17.

Trees

- *Binary trees*

Binary trees are the most common type of tree used in computer science. A binary tree is a tree in which every node has at most two children, and can be defined "inductively" by the following rules:

Dfinition. A binary tree is either

(Rule 1) the empty tree EmptyTree, or

(Rule 2) it consists of a node and two binary trees, the left subtree and right subtree.

Again, Rule 1 is the "base case" and Rule 2 is the "induction step". This definition may appear circular, but actually it is not, because the subtrees are always simpler than the original one, and we eventually end up with an empty tree.

The maximum number of nodes in a binary tree of height d is equal to 2^{d+1}-1.

The maximum number of nodes in a binary tree with the number of levels l is 2^{l}-1.

A binary tree with n nodes has a height of $\log^{n+1}$-1 and the number of levels is equal to $\log^{n+1}$.

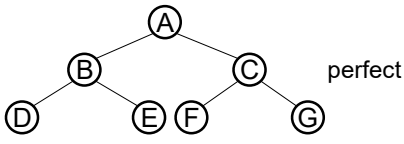
Trees

- *Binary trees*

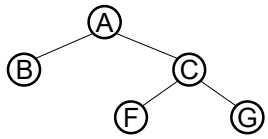
A full binary tree is a tree in which every node has either 0 or 2 children.

A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level (the level of a node defined as the number of edges or links from the root node to a node). A perfect binary tree is a full binary tree.

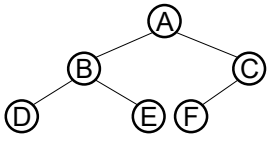
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. A perfect tree is therefore always complete but a complete tree is not always perfect.



perfect



full



complete

Trees

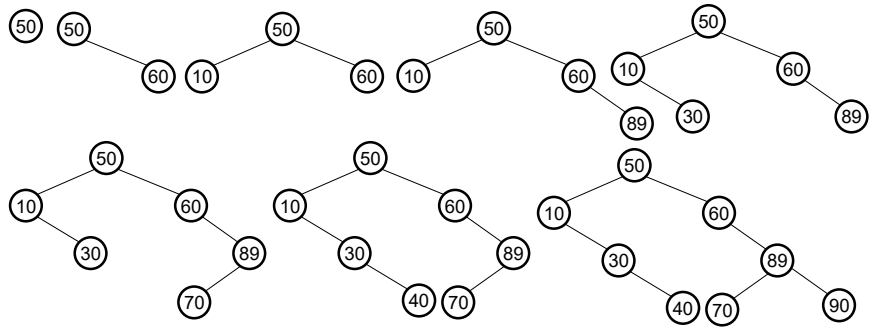
➢ *Binary trees*

The BST tree has the following properties:

The value in each node is greater than the left child and smaller than the right child value (if any).

The search and insertion time in the BST tree is O(log n). The time to create the tree is O(n log n).

50,60,10,89,,30,70,40,90



Trees

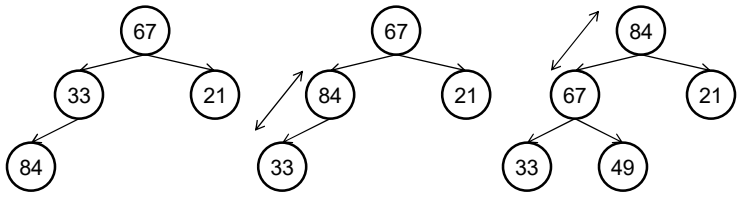
➢ *Heap trees*

Definition A binary heap tree is a complete binary tree which is either empty or satisfies the following conditions:

- ❖ The priority of the root is higher than (or equal to) that of its children.
- ❖ The left and right subtrees of the root are heap trees.

Alternatively, one could define a heap tree as a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendants. Or, as a complete binary tree for which the priorities become smaller along every path down through the tree.

67	33	21	84	49



Trees

➢ *B trees*

A B-tree is a generalization of a self-balancing binary search tree in which each node can hold more than one search key and have more than two children.

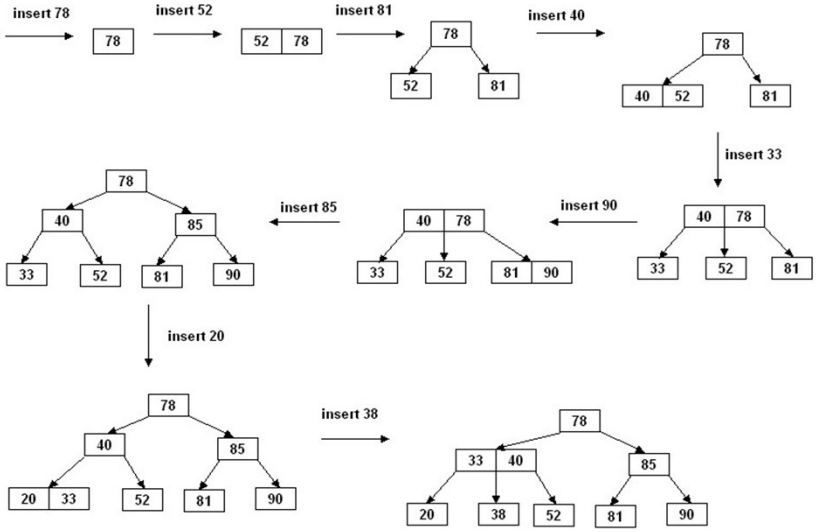
The structure is designed to allow more efficient self-balancing, and offers particular advantages when the node data needs to be kept in external storage such as disk drives. The standard (Knuth) definition is:

Definition A B-tree of order m is a tree which satisfies the following conditions:

- ❖ Every node has at most m children.
- ❖ Every non-leaf node (except the root node) has at least m/2 children.
- ❖ The root node, if it is not a leaf node, has at least two children.
- ❖ A non-leaf node with c children contains c-1 search keys which act as separation values to divide its sub-trees.
- ❖ All leaf nodes appear in the same level, and carry information.

Trees

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



Trees

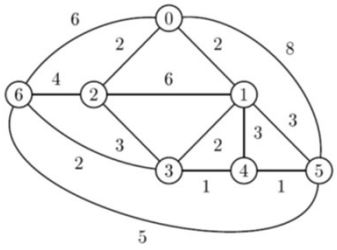
- *Spanning tree*

Remove from a connected graph as many edges as possible while remaining connected; this should yield a tree with n -1 edges.

This is the minimal spanning tree problem solved by the following algorithm (Kruskal), of time complexity O(mlogm).

Algorithm KruskalMST(G)
E_{ord} := sort(E);E' := ;E_{rest} := E_{ord} ;
while |E'| < n - 1 **do**
:= first(E_{rest}); E_{rest} := E_{rest}\{ };
if (V,E' U { }) is acyclic **then**
E' := E' U { })
end if
end while

The sorting is done efficiently in O(mlogm) time as well.
look at the example:

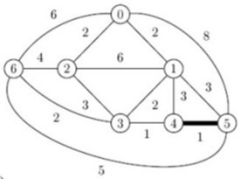


Trees

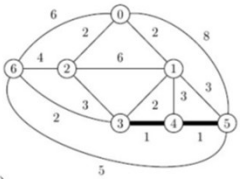
- *Spanning tree*

The different steps of the algorithm are

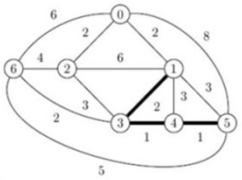
(1)



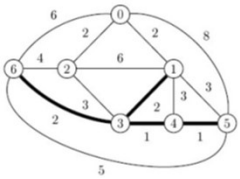
(2)



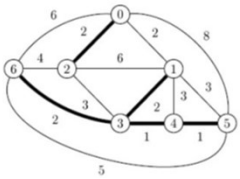
(3)



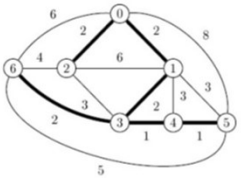
(4)



(5)



(6)



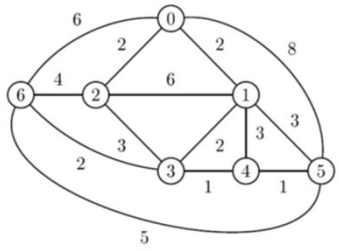
This constructs a tree which is a subgraph with n - 1 edges

Trees

- *Spanning tree*

Now we look at an alternative algorithm (Prim) of time complexity O((m + n) log n). The idea is to pick a random node and then grow a minimal tree from there.

Algorithm PrimMST(G)
Choose u V;V' :={u};E' := ;
for i = 1 : n - 1 **do**
E'' := edges linking V to V';
choose e = (u,v) E'' of minimal weight and such that
(V' U {v},E'{e}) is acyclic;
V' := V' U {v}; E' := E' U {e};
end for



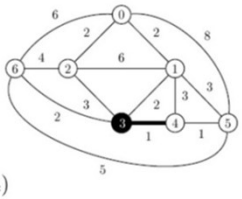
look at the same example:

Trees

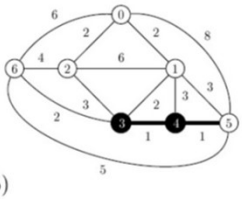
- *Spanning tree*

The different steps of the algorithm are

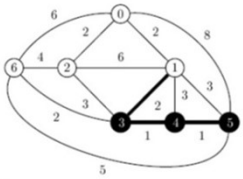
(1)



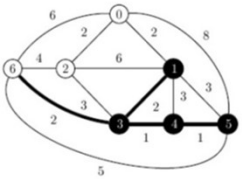
(2)



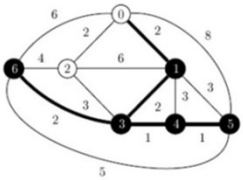
(3)



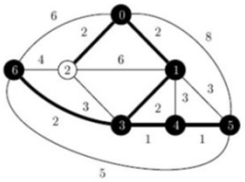
(4)



(5)



(6)



The graph (V,E') is a minimal spanning tree with n - 1 edges

Trees

- *Spanning tree*

One possible application of the shortest spanning tree is as follows: Suppose that we are to connect n cities $v_1, v_2, \ldots, v_n$ through a network of roads. The cost c_{ij} of building a direct road between v_i and v_j is given for all pairs of cities where roads can be built. (There may be pairs of cities between which no direct road can be built.) The problem is then to find the least expensive network that connects all n cities together.

Shortest path algorithms have a wide range of applications such as in-

- Google Maps
- Road Networks
- Logistics Research
- Social Networks

Shortest path

- *Shortest path problems*

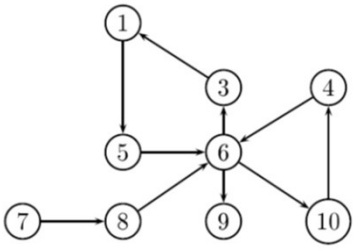
Proposition If there is a shortest walk from s to t, there is also a shortest path from s to t.

Proof

Assume the walk is not a path;

hence there is a recurring node.

Eliminate the cycle between the first and last occurrence of this node. Repeat this procedure.



In the above graph the path (7, 8, 6, 3, 1, 5, 6, 10, 4, 6, 9) has a cycle (6, 3, 1, 5, 6, 10, 4, 6). After its elimination we have a path (7, 8, 6, 9).

Corollary If G does not contain cycles of negative length, the resulting path is one of lower cost.

Proof Trivial

Shortest path

➢ *Dijkstra's algorithm*

This method is for a digraph G that has positive edge lengths.

For undirected graphs one can duplicate each edge as follows

Below, $V^+(u)$ denotes the set of children of u.

Algorithm Dijkstra(G,u)
S := {u}; d(u) := 0; d(v) := c(u, v) v ≠ u;
while S ≠ V **do**
choose v' S : d(v') ≤ d(v) v S;
S := S U {v'};
for each v V⁺(v') **do**
d(v) = min{d(v), d(v') + c(v', v)}
end for
end while

Shortest path

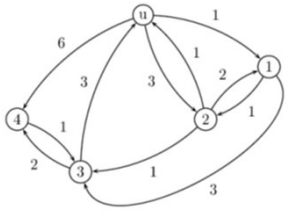
➢ *Dijkstra's algorithm*

Idea : Update a set S for which we know all shortest paths from u.

Let us see the behavior of this algorithm on an example.

The table indicates the steps and the distances computed for each node.

Iter	*S*	$d(u)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
0	$\{u\}$	0	1	3	∞	6
1	$\{u,1\}$	0	1	2	4	6
2	$\{u,1,2\}$	0	1	2	3	6
3	$\{u,1,2,3\}$	0	1	2	3	5
4	$\{u,1,2,3,4\}$	0	1	2	3	5

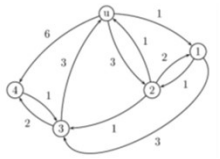


We indicate in more detail the exploration of the graph.

Shortest path

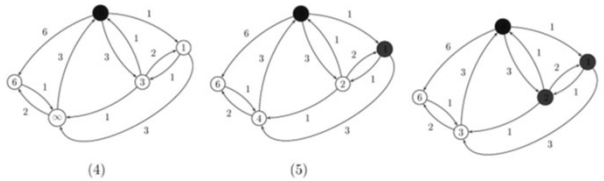
- *Dijkstra's algorithm*

S	$d(u)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
$\{u\}$	0	1	3	∞	6
$\{u,1\}$	0	1	2	4	6
$\{u,1,2\}$	0	1	2	3	6
$\{u,1,2,3\}$	0	1	2	3	5
$\{u,1,2,3,4\}$	0	1	2	3	5

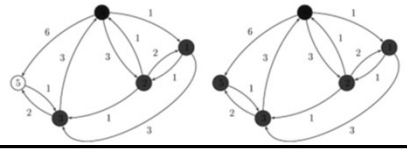


The node u is blue and the explored nodes are red.

(1) (2) (3)

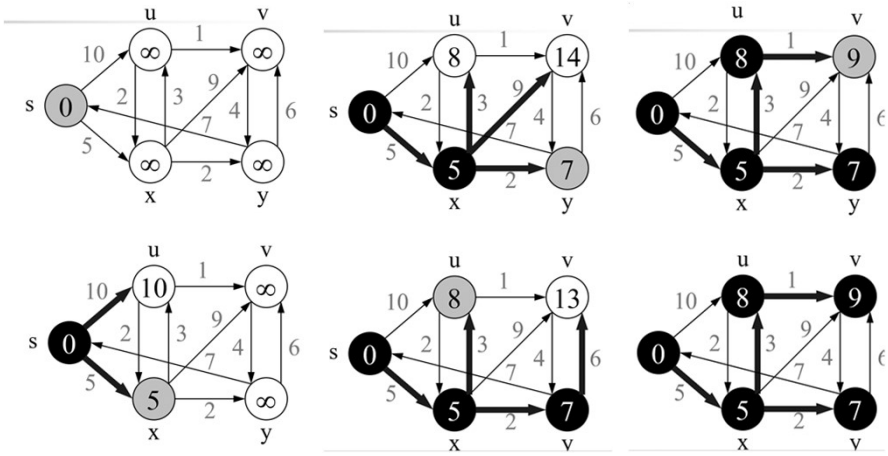


(4) (5)



Shortest path

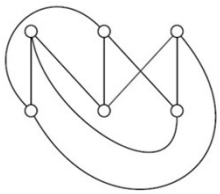
- *Dijkstra's algorithm*

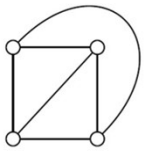


Planar graphs

- *Planar graphs*

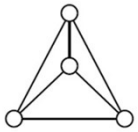
When drawing connected graphs one is naturally lead to the question of crossing edges. One says that a graph is planar if it can be drawn (or represented) without crossing edges The below graphs represent $K_{3,3}$(not planar) and K_4 (planar). K_5 is not planar.





Proposition (Fary, 1948)

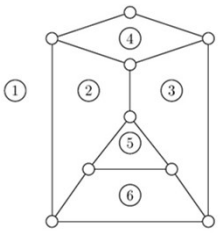
Every planar graph can be represented in the plane using straight edges only



Planar graphs

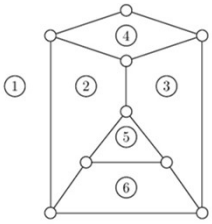
- *Planar graphs*

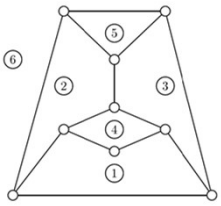
For such graphs, one can now define faces. These are the regions encircled by edges that form a cycle. One has to identify also an exterior face as shown in this figure with 6 faces.



The boundary of a face is a cycle in the graph. The length of the face (degree of face $\deg(f_i)$) is the length of that cycle.

Proposition A planar representation of a graph can be transformed to another one where any face becomes the exterior face.





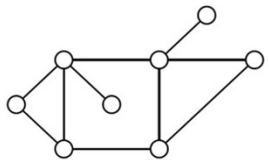
Planar graphs

- *Characterisation*

Proposition (Euler formula) Let G be connected planar, and let n(G) be its number of vertices, e(G) its number of edges, and f(G) its number of faces. Then f = e - n + 2.

In the example shown here

n = 8, e = 10, f = 4



Proof Use induction on the number of faces f .

For f = 1 there are no cycles and hence the connected graph is a tree, for which we know e = n - 1 and hence f = e - n + 2.

For f ≥ 2, remove an edge (u, v) between two faces to construct G' := G\(u, v). Then f (G') = f (G) – 1, e(G') = e(G) - 1 and n(G') = n(G). Use the result for smaller f to prove it for f .

Planar graphs

- *Characterisation*

Proposition Let G be planar, then $\sum_i deg(f_i) = 2e$.

Proposition Let G be planar with f > 1, then 3f ≤2e.

Proposition Let G be planar with f > 1 and G have no triangles then 2f ≤ e.

Proposition Let G be a planar (connected) graph. If n ≥3 then e ≤ 3n – 6.

Proposition Let G be a planar (connected) graph. If G has no triangles or is bipartite, then e ≤ 2n – 4.

Proposition K_5 and $K_{3,3}$ are not planar.

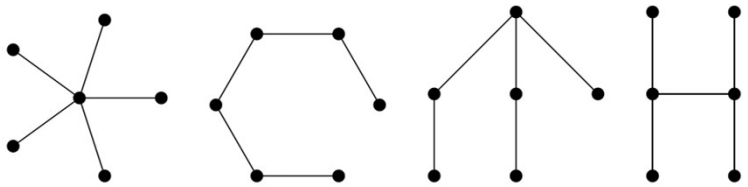
Proof?

Planar graphs

- *Planar graphs and trees*

Euler's formula starts from a familiar formula: if a tree has n vertices and m edges, then m = n - 1.

All trees are planar graphs! any drawing of a tree is a plane embedding unless you deliberately try to mess it up.
Trees have only one face, so we confirm that n - e + f = n - (n - 1) + 1 = 2.



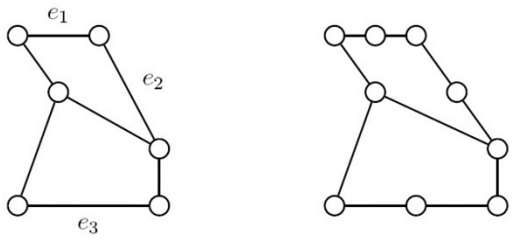
Planar graphs

- *Test for planar graphs*

We first need to introduce subdivisions and subgraphs.

Let us expand a graph G = (V,E) by a subdivision of one of its edges e = (u, v) E. We put a new node w on e and replace it by two new edges e_1 = (u,w) and e_2 = (w, v). The new graph is thus given by G' = (V U {w},E U {e_1, e_2} /{e}).

Two graphs are said to be homeomorphic to each other iff one can be derived from the other via a sequence of subdivisions.

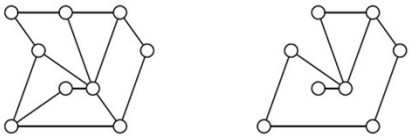


Corollary Homeomorphism is an equivalence relation.

Planar graphs

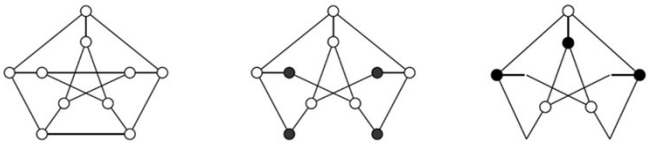
- *Test for planar graphs*

A graph G' = (V',E') is a subgraph of a graph G = (V,E) if V' V and E' E (edges must disappear along with nodes)



Proposition (Kuratowsky, 1930) A graph is planar iff it does not contain a subgraph homeomorphic to $K_{3,3}$ or K_5.

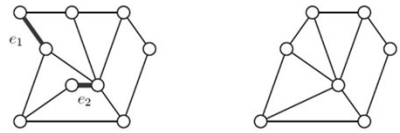
Example : the Petersen graph (subgraph + homeomorphism)



Planar graphs

- *Minors*

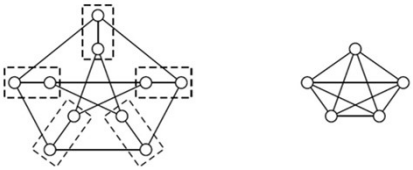
Let e = (u, v) be an edge of a graph G = (V,E). A contraction of the edge e consists of eliminating e and merging the nodes u and v into a new node w. The new graph G' is thus G' = (V\ {u, v}U {w},E\{e})



Proposition (Wagner, 1937) A graph is planar iff it does not have $K_{3,3}$ or K_5 as a minor.

Example :

the Petersen graph



Planar graphs

- *Minors*

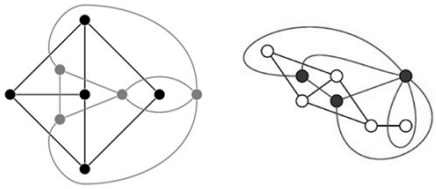
Proposition (Robertson-Seymour)

For a graph G, determining if a given graph H is a minor of G, can be solved in polynomial time (with respect to n(G) and m(G)).

A dual graph G*of a planar graph is obtained as follows:

1. G* has a vertex in each face of G
2. G* has an edge between two vertices if G has an edge

between the corresponding faces
This is again a planar graph
but it might be a multigraph
(with more than one edge
between two vertices)

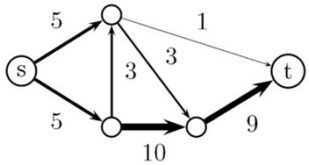


Flows

- *Networks and flows*

A network is a directed graph N = (V,E) with a source node s (with $d_{out}(s)>0$) and a terminal node t (with $d_{in}(t) > 0$).

Moreover each edge has a strictly positive capacity c(e) > 0.



A flow f : $V^2 \rightarrow R^+$ is associated with each edge e = (u, v) s.t.

1. for each edge e E we have $0 \le f(e) \le c(e)$
2. for each intermediate node v V\ {s,t} the in- and out-flow at that node match

$$\sum_{u \in V^-(v)} f(u,v) = \sum_{u \in V^+(v)} f(v,u)$$

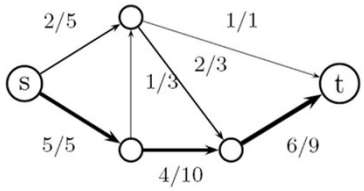
The total flow F of the network is then what leaves s or reaches t

$$F(N) := \sum_{u \in V} f(s,u) - \sum_{u \in V} f(u,s) = \sum_{u \in V} f(u,t) - \sum_{u \in V} f(t,u)$$

Flows

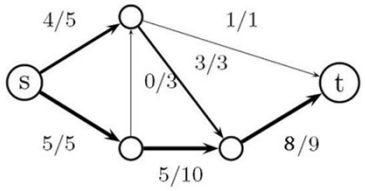
- *Networks and flows*

Here is an example of a flow



It has a value of F(N) = 7 and the conservation law is verified inside.

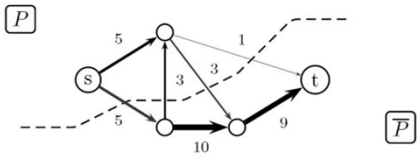
But the flow is not maximal, while the next one is (F(N) = 9) as we will show later. Notice that one edge is not being used (f = 0)



Flows

- *Cut of a network*

A cut of a network is a partition of the vertex set V = into two disjoint sets (containing s) and (containing t).



The capacity of a cut is the sum of the capacities of the edges (u,v) between and .

$$\kappa(P,\overline{P}) = \sum_{u\in P; v\in \overline{P}} c(u,v)$$

which in the above example equals 5 + 3 + 1 = 9.

We now derive important properties of this capacity.

Flows

- *Cut of a network*

Proposition Let ( ) be any cut of a network N = (V,E) then the associated flow is given by

$$F(N) = \sum_{u \in P; v \in \overline{P}} f(u,v) - \sum_{u \in P; v \in \overline{P}} f(v,u)$$

Proof First show that $F(N) = \sum_{u \in P} (\sum_v f(u,v) - \sum_v f(v,u))$

by summing all contributions in P and using conservation.

For all v P the term between brackets is zero (conservation).

Hence we only need to keep the edges across the partition.

Corollary A flow is bounded by the capacity of any cut F(N)≤ ( )

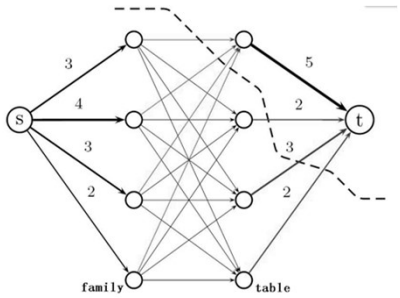
A minimal cut (with minimal capacity) also bounds F(N).

(we will construct one and will see it is in fact equal to F(N)).

Flows

- *Applications*
  - *The dining problem*

Can we seat 4 families with number of members (3,4,3,2) at 4 tables with number of seats (5,2,3,2) so that no two members of a same family sit at the same table?



The central edges are the table assignments (a capacity of 1).

The cut shown has a capacity 11 which upper bounds F(N).

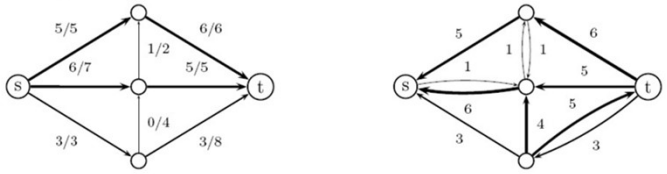
We can therefore not seat all 12 members of the four families.

Flows

- *Applications*

Given a network N(V,E) and a flow f then its residual network N_f is a network with the same nodes V but with new capacities

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E; \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases}$$



An augmenting path is a directed path $v_0,\ldots,v_k$ from $S = v_0$ to $t = v_k$ for which

$$\Delta_i = c(v_i, v_{i+1}) - f(v_i, v_{i+1}) > 0 \ \forall (v_i, v_{i+1}) \in E \text{ or}$$
$$\Delta_i = c(v_i, v_{i+1}) - f(v_{i+1}, v_i) > 0 \ \forall (v_{i+1}, v_i) \in E$$

This path is not optimal since the original flow can be increased.

Flows

- *Max-flow Min-cut*

Proposition

The flow is optimal if there exists no augmentation path from s to t.

Proof Construct a cut ( ) where u if there is an augmentation path from s to u and u otherwise.

Show that ( ) is a valid cut for which F(N) = ( ).

Proposition In a network N the following are equivalent

1. A flow is optimal
2. The residual graph does not contain an augmenting path
3. F(N) = ( ) for some cut ( )

The value of the optimal flow thus equals F(N) = min ( ).

Proof combine earlier results

$$\text{maximum flow} = \text{minimum cut}$$

Flows

- *Max-flow Min-cut*

The Ford-Fulkerson algorithm (1956) calculates this optimal flow using augmentation paths.

Algorithm MaxFlowFF(N,s,t)
f (u, v) := 0 ∀(u, v) E;
while N_f contains a path from s to t **do**
choose an augmentation path A_p from s to t
$\Delta := \min_{(u,v) \; A_p} \Delta_i$
Augment the flow by Δ along A_p
Update N_f
end while

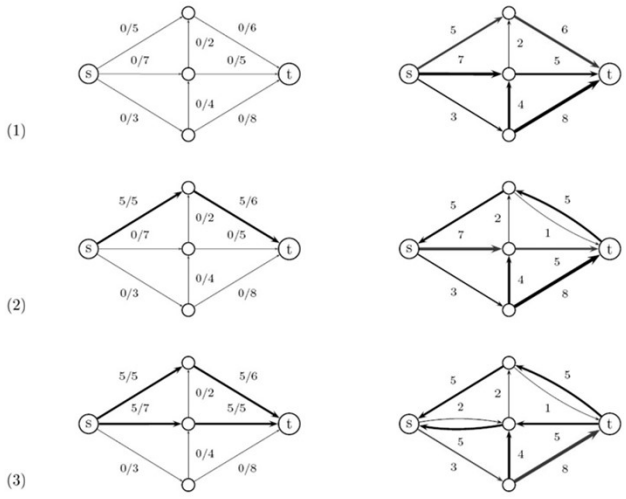
Finding a path in the residual graph can be implemented with a BFS or DFS exploration as shown below.

At each step we show the graph (left) and the residual graph (right)

Augmentation paths are in red. In 5 steps we find F(N) = 14

Flows

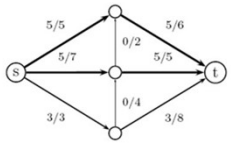
- *Max-flow Min-cut*

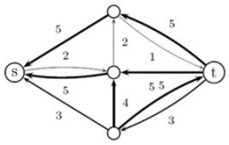


Flows

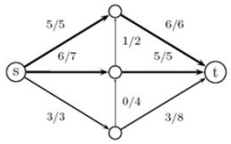
- *Max-flow Min-cut*

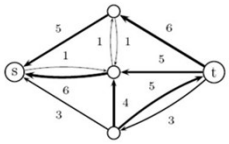
(4)



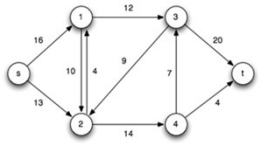


(5)





- *Max-flow Min-cut?*

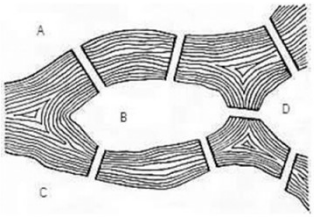


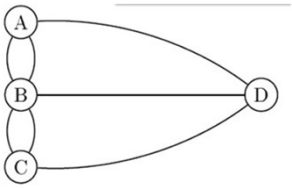
Euler

- *Eulerian tour (1756)*

An Eulerian cycle (path) is a subgraph $G_e = (V,E_e)$ of $G = (V,E)$ which passes exactly once through each edge of G.

G must thus be connected and all vertices V are visited (perhaps more than once). One then says that G is Eulerian.





Proposition A graph G has an Eulerian cycle iff it is connected and has no vertices of odd degree.

A graph G has an Eulerian path (i.e. not closed) iff it is connected and has 2 or no vertices of odd degree.

This would prove that the above graph is not Eulerian.

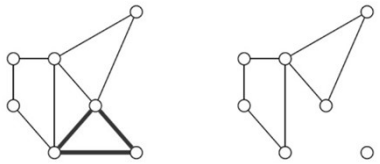
Euler

- *Eulerian tour (1756)*

Proof (of the first part regarding cycles)

Necessity Since G is Eulerian there is a cycle visiting all nodes.
Each time we visit v V, we leave it again, hence d(v) is even.

Sufficiency For a single isolated node, it is trivial. For |V| > 1.
there must be a cycle ϕ in the graph. Consider the subgraph H with the same nodes but with the edges of ϕ removed.



Each of its components H_i satisfy the even degree condition and again have an Eulerian cycle ϕ_i. By recurrence we then reduce G to its isolated vertices.
To reconstruct the Eulerian cycle, start from a basic cycle ϕ. Each time a node of another cycle ϕ_i is encountered, substitute that cycle to the node (and do this recursively).

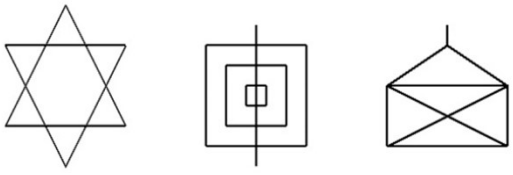
Euler

- *Eulerian tour (1756)*

The following algorithm of Fleury (1883) reconstructs a cycle C if it exists. E' is the set of edges already visited by the algorithm.

Algorithm FindEulerianCycle(G)
Choose v_0 V;E' := ;C := ;
for i = 1 : m **do**
choose e = (v_{i-1}, v_i) s.t. G' = (V,E\E') has 1 conn. comp.;
E':= E'U{e};C := ; $v_i := v_{i-1}$;
end for

The path problem says if you can draw a graph without lifting your pen.

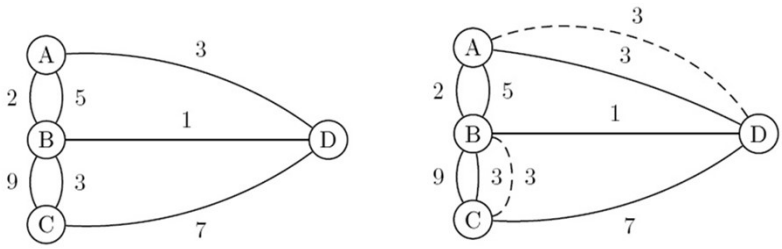


Euler

- *Chinese postman (1962)*

In this problem, discussed by the Chinese mathematician Mei-Ku Kwan, a postman wishes to deliver his letters, covering the least possible total distance and returning to his starting point. He must obviously traverse each road in his route at least once, but should avoid covering too many roads more than once.

We consider a minimum cost modification of the Eulerian cycle problem. A Chinese postman needs to find a tour passing along all edges of a graph and minimize the length of the path.

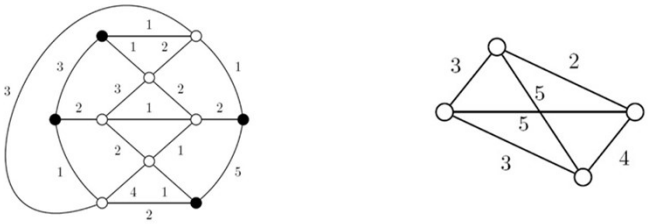


The edges have a cost and we need to make the graph Eulerian.

Euler

- *Chinese postman (1962)*

Solution: find all odd degree vertices and find the shortest paths between them.



- Which complete graphs K_n have an Euler circuit?
  K_n has an Euler circuit for n odd.

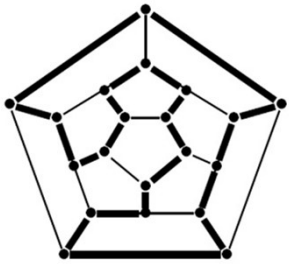
- When do bipartite complete graphs have an Euler circuit?
  $K_{m,n}$ when both m and n are even.

Hamilton

- *Hamiltonian cycle (1859)*

Was a game sold by Hamilton in 1859 to a toy maker in Dublin.

A Hamiltonian cycle is a cyclic subgraph $G_h = (V,E_h)$ of $G = (V,E)$ which passes exactly once through all nodes. It is a so-called hard problem and there is no general condition for its existence (in contrast with the Eulerian path problem).



It exists for Platonic solids and complete graphs, but not for the Petersen graph.

for all m,n≥2,

Hamilton

- *Hamiltonian cycle (1859)*

Proposition (Dirac, 1951) A graph G with n≥3 nodes and d(v)≥n/2, ∀ v V, is Hamiltonian.

Dirac's **theorem**: If (but not only if) G is connected, simple, has n≥3 vertices, and, ∀ v d(v) ≥n/2, then G has a Hamilton circuit.

Ore's **corollary**: If G is connected, simple, has n≥3 nodes, and d(u)+d(v)≥n for every pair u, v of non-adjacent nodes, then G has a Hamilton circuit.

corollary: A complete bipartite graph $K_{m,n}$ is Hamiltonian iff m = n, for all m, n ≥2.

A traveling salesman is supposed to visit a number of cities (nodes in a graph) and minimize the travel time (or total length).

This is NP-hard but can often be solved approximately in reasonable time.

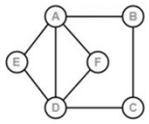
For n vertices in a complete graph, there will be (n−1)!=(n−1)(n−2)(n−3)⋯3·2·1 routes. Half of these are duplicates in reverse order, so there are (n−1)!/2 unique circuits.

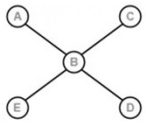
Hamilton

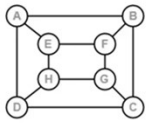
- *Hamilton/Euler*

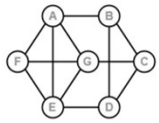
Which of the following is/are Hamiltonian graphs?

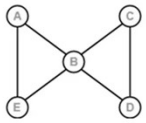
Which of the following is / are Euler Graphs?

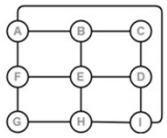
A) 

B) 

C) 

D) 

E) 

F) 

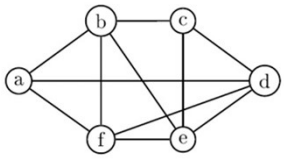
Hamilton

- *Test for planar graph*

There is a simple way to test if a Hamiltonian graph is planar.

1. Draw G with the Hamiltonian graph H at the outside

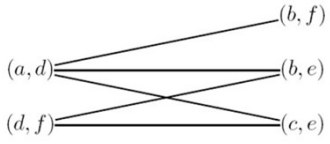
The following graph is already drawn with H = (a, b , c , d , e , f , a) outside



2. Define K as the graph whose nodes are the edges $e_1,\ldots,e_r$ not in H and with an edge between e_i and e_j if they cross in G.

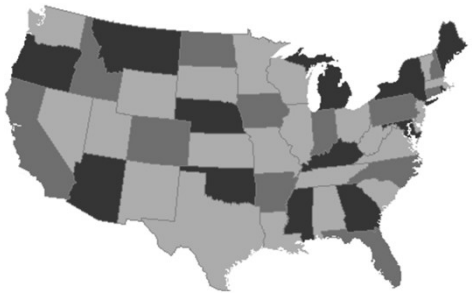
The following graph has the vertices (a, d),(b, f),(b, e), (c, e), (d, f) and five edges, corresponding to the crossings in G.

Then G is planar iff K is bipartite.



Coloring

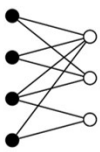
- *Four color problem*
- In 1852 it was conjectured that a country map (like the USA map) could always be colored with only four colors. the four color theorem, or the four color map theorem, states that no more than four colors are required to color the regions of any map so that no two adjacent regions have the same color. There is an underlying assumption for point borders.
- This was proven in 1976 by K. Appel and W. Haken but their proof used a computer search over 1200 so-called critical cases.



Coloring

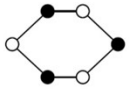
- *Coloring nodes*
- A k-coloring of a graph G = (V,E) is a mapping f : V → 1,…,k such that $f(v_i) \neq f(v_j)$ if (v_i, v_j) E.
- The chromatic number of a graph is the smallest number k for which there exists a k-coloring.
- Some examples of known chromatic numbers are:

Bipartite graph
$\chi(G) = 2$

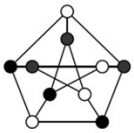


Clique
$\chi(K_n) = n$

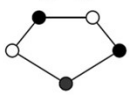
Even cycle (C_n)
$\chi(G) = 2$



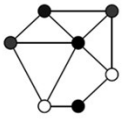
Petersen Graph
$\chi(G) = 3$



Odd cycle (C_n)
$\chi(G) = 3$



Planar graph
$\chi(G) = 4$



Coloring

➢ *Coloring nodes*

❖Let G be connected, if |E|≤|V|, then

❖Let G be Herschel graph,

❖**Proposition** Let Δ(G) = max{d(v)|v V}, then ∀G,

❖Let G be wheel graph,

(C_n)

❖**Proposition** (Brooks, 1941)

for any graph different from K_n or an odd cycle

Coloring

➢ *Triangle free*

❖ A graph is triangle-free if it does not contain a K_3 as an induced subgraph.

❖ Let G be a graph with V (G) ={v_1, v_2, ... ,v_n}. The Mycielskian of G is a graph M(G) constructed from G as follows:

1. Start with a copy of G.
2. For each vertex v_i V (G), add a "shadow vertex" u_i adjacent to all of v_i's neighbors in G. (There will never be edges between the shadow vertices.)
3. Finally, add a vertex w adjacent to all the shadow vertices u_1,u_2,... ,u_n.

If G is triangle-free, then so is M(G). $\chi(M(G))$ Is at least $\chi(G) + 1$.

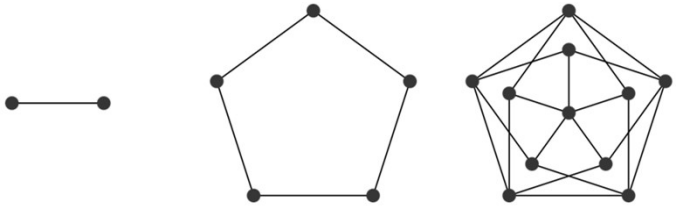
❖ Even though M(G) has some new triangles using a shadow vertex, those triangles are "shadows" of a triangle in G. If G were triangle-free, M(G) would be as well.

Coloring

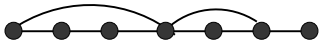
- *Mycielskian*

- The Mycielski construction is a method for turning a triangle-free graph with chromatic number k into a larger triangle-free graph with chromatic number k+ 1.

G M1 M2



- The Mycielskian of G is?



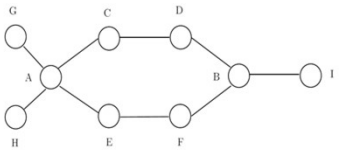
Coloring

- *Welsh-Powell Algorithm*

- vertex coloring is a way of labelling each individual vertex such that no two adjacent vertex have same color.
- Welsh-Powell algorithm gives the minimum colors. This algorithm is also used to find the chromatic number of a graph.
- This is an iterative greedy algorithm:
- Step 1: All vertices are sorted according to the decreasing value of their degree in a list V.
- Step 2: Colors are ordered in a list C.
- Step 3:The first non colored vertex v in V is colored with the first available color in C. available means a color that was not previously used by the algorithm.
- Step 4: The remaining part of the ordered list V is traversed and the same color is allocated to every vertex for which no adjacent vertex has the same color's all the vertices havn colored.

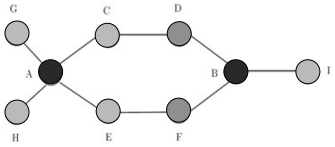
Coloring

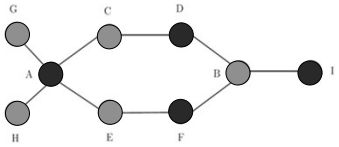
- *Welsh-Powell Algorithm*



main Graph

vertex color using Welsh-Powell

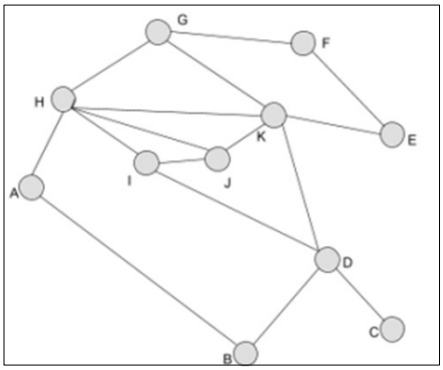




- However, we can color the graph with two colors. Let A, D, F, and I be red and C, E, G, H, and B, be blue. e vertices colored.

Coloring

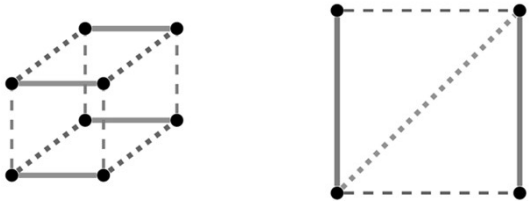
- *Welsh-Powell Algorithm*
- The chromatic number of the graph using Welsh-Powell is?e vertices colored.



Coloring

- *Edge coloring*

- In graph theory, edge coloring of a graph is an assignment of "colors" to the edges of the graph so that no two adjacent edges have the same color with an optimal number of colors. There is no known polynomial time algorithm for edge-coloring every graph with an optimal number of colors.
- proper vertex coloring of a graph G is an assignment of "colors" to each vertex, so that adjacent vertices get different colors.
- A proper edge coloring of a graph G is an assignment of colors to each edge, so that edges that share an endpoint get different colors.



vertices color

Coloring

- *Edge coloring*

- The edge chromatic number $\chi'(G)$ is the least number of colors needed to properly edge color G.
- **Proposition** All graphs G have $\chi'(G) \geq \Delta(G)$.
- **Proposition** For odd n : $\chi'(K_n) = n$.
- **Proposition** For even n : $\chi'(K_n) = n\text{-}1$.
- **Proposition** (Vizing (1964), Gupta (1966)). Let G be a simple graph. Then $\Delta(G) \leq \chi'(G) \leq \Delta(G)+1$.
- Let G be Herschel graph, $\chi'(G) = 4$.

- Graph Coloring applications:
  - Sudoku
  - Exam Timetable Scheduling.
  - Aircraft Scheduling

Coloring

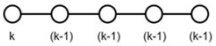
- *Chromatic polynomial (Birkhoff-Lewis 1918)*
- The chromatic polynomial of a graph $p_G(k)$ indicates how many different ways a graph can be colored with k colors. E.g.

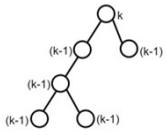
k	1	2	3	k
$p_G(k)$	0	2	24	$k(k-1)^3$

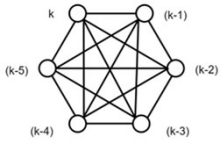
k	1	2	3	k
$p_G(k)$	0	0	6	$k(k-1)(k-2)$

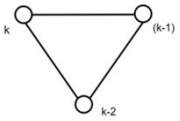
k	1	2	k
$p_G(k)$	1	16	k^4

k	1	2	k
$p_G(k)$	0	2	$k(k-1)^{n-1}$





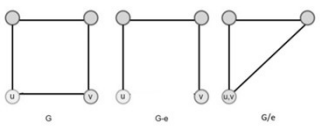




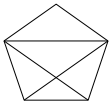
- $\chi(G) = \min\{P_G(k) > 0\}$

Coloring

- *Chromatic polynomial (Birkhoff-Lewis 1918)*
- There is a powerful induction theorem using the simpler graphs:
- G - (u,v) (remove an edge) and G○(u,v) (contract an edge)
- **Proposition** If (u,v) E then $p_G(k) = p_{G-(u,v)}(k) - p_{G\circ(u,v)}(k)$
- **Proof** u and v have different colors in G and the same in G○(u,v)
- This can be used to compute the chromatic polynomial of more complex networks.



- The chromatic polynomial for the graph is?

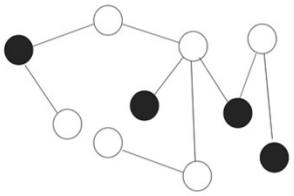


Independent set

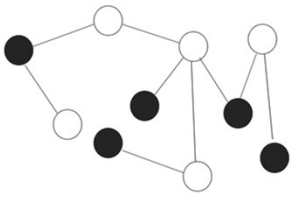
- *Independent set*
- An independent set S is a subset of V in G such that no two vertices in S are adjacent.
- Like other vertex sets in graph theory, independent set has maximal and maximum sets as follows:
- The independent set S is maximal if S is not a proper subset of any independent set of G.
- The independent set S is maximum if there is no other independent set has more vertices than S.
- That is, a largest maximal independent set is called a maximum independent set. The maximum independent set problem is an NP-hard optimization problem.
- All graphs has independent sets. For a graph G having a maximum independent set, the independence number $\alpha(G)$ is determined by the cardinality of a maximum independent set (independence number).ices

Independent set

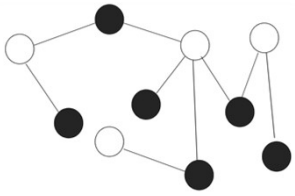
independent set (IS)



maximal IS (MIS)



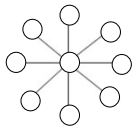
maximum IS (MaxIS)



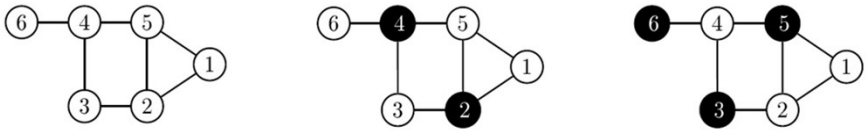
ices colored.

Independent set

- *Independent set*
  - Minimal and Maximal independent set?



- Independent set problem is related to coloring problem since vertices in an independent set can have the same color.
- In the following figures, the two sets of black nodes are stable sets (independent set) of the left graph. Such sets can clearly be colored with only one color.



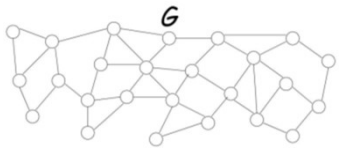
- **Proposition** If a graph is k-colorable then V can be partitioned as k stable sets.

Independent set

- *Finding a single maximal independent set*
  - Given a Graph G(V,E), it is easy to find a single MIS using the following algorithm with O(n+m) running time.

Algorithm Finding_single_maximal_independent_set
Initialize I to an empty set.
while V is not empty
  Choose a node v ∈ V
  Add v to the set I
  Remove from V the node v and all its neighbors
end while
Return Iices col

  - a single maximal independent set is?

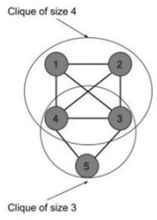


Independent set

- *Independent set*
- Applications in Distributed Systems: network topology
  - In a network graph consisting of nodes representing processors, a MIS defines a set of processors which can operate in parallel without interference.
  - For instance, in wireless ad hoc networks, to avoid interferences, a conflict graph is built, and a MIS on that defines a clustering of the nodes enabling efficient routing.
- Applications in Distributed Systems: network monitoring
  - In a network graph G consisting of nodes representing processors, a MIS defines a set of processors which can monitor the correct functioning of all the nodes in G (in such an application, one should find a MIS of minimum size, to minimize the number of sentinels, but as said before this is known to be NP-hard)
- The maximum independent set and its complement, is involved in proving the computational complexity of many theoretical problems. They also serve as useful models for real world optimization problems, for example maximum independent set is a useful model for discovering stable genetic components for designing engineered genetic systems.

Clique Problem

- *clique*
- Given an undirected graph G = (V, E). A subset of nodes S is a clique if every pair of nodes in S have an edge between them in G. A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph.
- A maximal clique is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique.
- A maximum clique of a graph, G, is a clique, such that there is no clique with more vertices. Moreover, the clique number ω(G) of a graph G is the number of vertices in a maximum clique in G.
- A k-clique is a clique of size k.
- The Clique Decision Problem is NP-Complete problem.



Clique Problem

- *clique*

- In greedy approach of finding a single maximal clique we start with any Starting with an arbitrary clique (for instance, any single vertex or even the empty set), grow the current clique one vertex at a time by looping through the graph's remaining vertices.
- For each vertex v that this loop examines, add v to the clique if it is adjacent to every vertex that is already in the clique, and discard v otherwise. The maximal clique can be different if we start with different vertex as a graph may have more than one maximal clique.

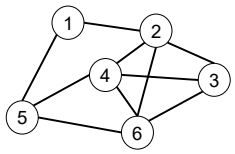
  Algorithm Finding_single_maximal_clique

  Start from an arbitrary vertex
  Given a clique of size, repeat:
  Add a vertex randomly from the common neighbors of the existing clique
  If there is no common neighbors, stop and return the clique
- The above algorithm runs in linear time in the size of the graph ($\Theta(n^2)$ edges)

Clique Problem

- *Applications*

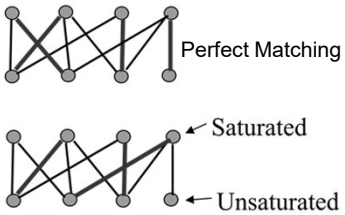
- Complete subgraphs are used to model social cliques, groups of people who all know each other. Luce & Perry (1949) used graphs to model social networks, and adapted the social science terminology to graph theory.
- Many different problems from bioinformatics have been modeled using cliques.
- Clique-finding algorithms have been used in electrical engineering
- Clique-finding algorithms have been used in chemistry.
- …
- The maximum clique is?

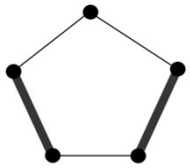


Matching

➢ *Bipartite Matching*

- A matching in a graph is a set of independent edges: edges that share no endpoints. We are often especially interested in finding a maximum matching which has the largest size possible.
- The vertices incident to the edges of a matching M are said to be saturated (covered) by M; the others are unsaturated(exposed).
- A perfect matching in a graph is a matching that saturates every vertex.



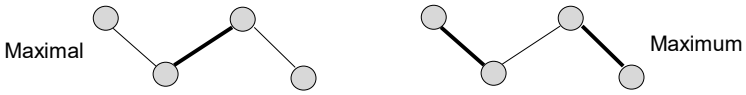


Maximum Matching
Not a perfect matching

Matching

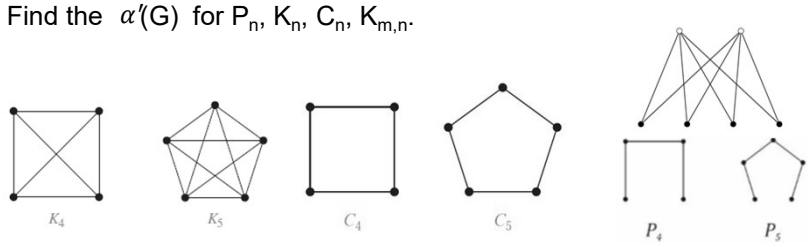
➢ *Bipartite Matching*

- We write α'(G) for the size of a maximum matching in G; the $'$ indicates that it's the edge version of a problem.
- We'll begin by looking at the maximum matching problem in bipartite graphs. Here, if G has bipartition (A,B), each edge in the matching has one endpoint in A and one in B, so the matching pairs up some vertices in A with adjacent vertices in B.
- A maximum matching in a graph is matching that can not be enlarged by adding an edge.
- A matching M is maximal if every edge not in M is incident to an edge already in M.
- The smallest graph having a maximal matching that is not a maximum matching is P4.

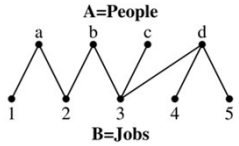


Matching

- *Bipartite Matching*
- Find the $\alpha'(G)$ for P_n, K_n, C_n, $K_{m,n}$.

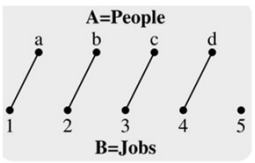


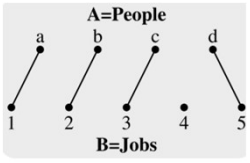
- This can model problems about pairing one type of object to another: workers to tasks, rooms to occupants, meetings to days, etc..
- Scenario: People applying for jobs



Matching

- *Bipartite Matching*
- Consider people applying for jobs:
  - A is the set of people
  - B is the set of jobs
  - Draw an edge uv when u ∈ A applies for v ∈ B.
  - This is a bipartite graph.
  - Each job can only go to one person, and each person can only get one job. It might not be possible to accommodate all people / jobs.

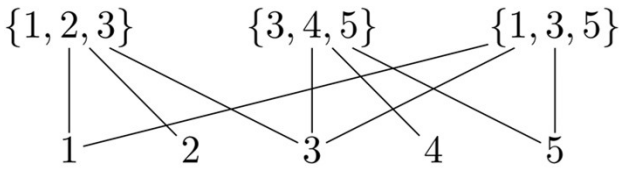




- On the left, a, b, c, d and 1, 2, 3, 4 are saturated and job 5 is exposed.
- A complete matching from A to B means every vertex of A is in an edge of the matching.
- The solutions give a complete matching from A to B, but not from B to A.

Matching

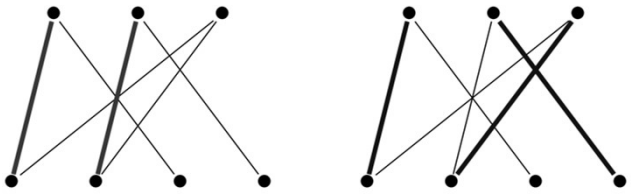
- *Bipartite Matching: Systems of distinct representatives*
- For some sets $S_1, S_2, \ldots, S_n$, a system of distinct representatives (also called a transversal) is a choice of elements $x_1 \in S_1, x_2 \in S_2, \ldots, x_n \in S_n$ that are all distinct: no two chosen elements are equal.
- Example. Take $S_1 = \{1, 2, 3\}$, $S_2 = \{3, 4, 5\}$, and $S_3 = \{1, 3, 5\}$.
- Then $x_1 = 1$, $x_2 = 3$, $x_3 = 5$ is a system of distinct representatives. This can be modeled as a matching problem in a bipartite graph! Put the sets $S_1, S_2, \ldots, S_n$ on one side, and their elements on the other.



- A system of distinct representatives is a matching of size n.

Matching

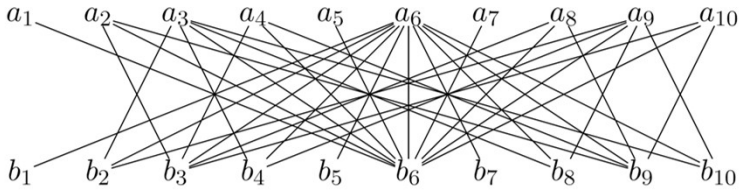
- *Bipartite Matching: Greedy algorithms*
- A greedy algorithm is, informally, any strategy for solving a problem that makes decisions without thinking ahead.
- The bipartite matching problem has a natural greedy strategy. Let $A \cup B$ be the bipartition of G. Go through the vertices of A one at a time; for each $a \in A$, let b be the first unmatched neighbor of a, and add edge ab to the matching.



- In red, we see the matching that the greedy algorithm finds. It is not always optimal! In blue, we see a maximum matching..

Matching

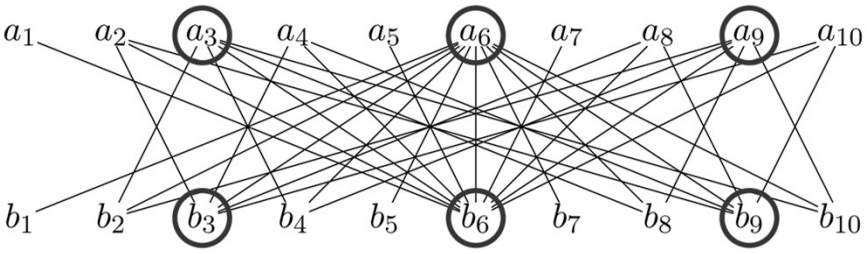
- *Bipartite Matching: Multiples of six*
- Consider the matching problem in the following bipartite graph: it has $A = \{a_1, \ldots, a_{10}\}$, $B = \{b_1, \ldots, b_{10}\}$, and an edge $a_i b_j$ whenever $i \cdot j$ is a multiple of 6.



- A maximum matching in this graph (there are many) has size 6.

Matching

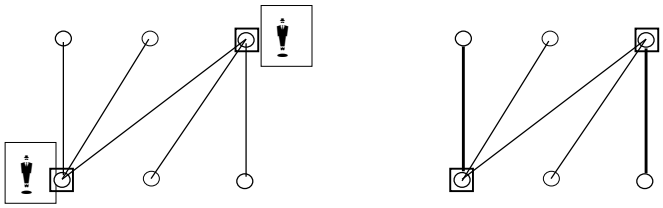
- *Bipartite Matching: vertex cover*
- If you play around with this problem for a while, you notice that multiples of 3 are the bottleneck. Every edge needs to include at least one of them; there are only 6, so once they run out, no other edge can be added to the matching.



- We call such a set of vertices a vertex cover. Formally, a vertex cover in a graph is a set of vertices that includes at least one endpoint of every edge.

Matching

- *Bipartite Matching: vertex cover*
- In a graph that represents a road network (with straight roads and no isolated vertices).
  - Finding a minimum vertex cover = Placing the minimum number of policemen to guard the entire road network (a).
  - We mark a vertex cover of size 2 and show a matching of size 2 in bold (b).

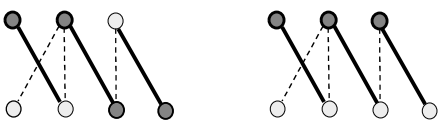


(a) (b)

Matching

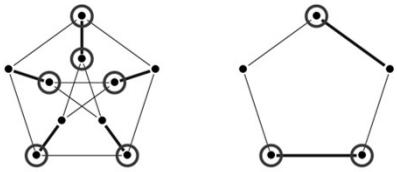
- *Bipartite Matching: vertex cover*
- We write $\beta(G)$ for the number of vertices in a minimum vertex cover of G: a vertex cover with as few vertices as possible.
- **Theorem** If G is any graph, $M \subseteq E(G)$ is any matching, and $U \subseteq V(G)$ is any vertex cover, then $|M| \leq |U|$. In particular, $\alpha'(G) \leq \beta(G)$.
- **Proof** Every edge of M contains at least one vertex of U, and they can't share: two edges of M can't have the same vertex of U. So for the edges $e_1, e_2, \ldots, e_k$ of M there must be just as many distinct vertices $v_1, v_2, \ldots, v_k$ in U such that v_i is an endpoint of e_i. There may also be other vertices in U, so all we know is that $|M| \leq |U|$.
- In particular, this holds when M is a maximum matching and U is a minimum vertex cover. In that case, $\alpha'(G) = |M| \leq |U| = \beta(G)$.

Green: Vertex cover
Red: Matching
$|U| \geq |M|$



Matching

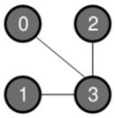
- *Bipartite Matching: vertex cover*
- α'(G) and β(G) are not equal for any graph G.



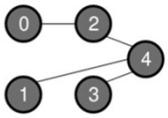
- They are equal for bipartite graphs. This makes the bipartite matching problem easier than the matching problem for general graphs.
- In all graphs G,α'(G) (the size of a maximum matching) is at most β(G) (the number of vertices in a minimum vertex cover).
- Example:

minimum vertex cover is empty

minimum vertex cover is {3}

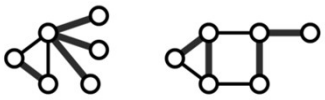


minimum vertex cover is {4,2} or {4,0}



Matching

- *Bipartite Matching: edge cover*
- An edge cover in a graph is a set of edges whose endpoints include every vertex. (Note: this is impossible when isolated vertices exist.)
- We write β'(G) for the size of a minimum edge cover in G..



edge covers

- Summary: Independent set, matching, vertex color edge color

Parameter	Min/max	Of what?	Condition	
$\alpha(G)$	max	vertices	no two are adjacent	Independent set
$\alpha'(G)$	max	edges	no two are incident	Matching
$\beta(G)$	min	vertices	that cover every edge	Vertex Cover
$\beta'(G)$	min	edges	that cover every vertex	Edge Cover