Research and Innovation Center of Excellence

Graph Theory

Panayiotis Kolios, PhD BEng AKC
Research Assistant Professor,
KIOS Research and Innovation Center of Excellence,
University of Cyprus
Web: www.kios.ucy.ac.cy/pkolios
Email: pkolios@ucy.ac.cy

University of Cyprus
Imperial College London

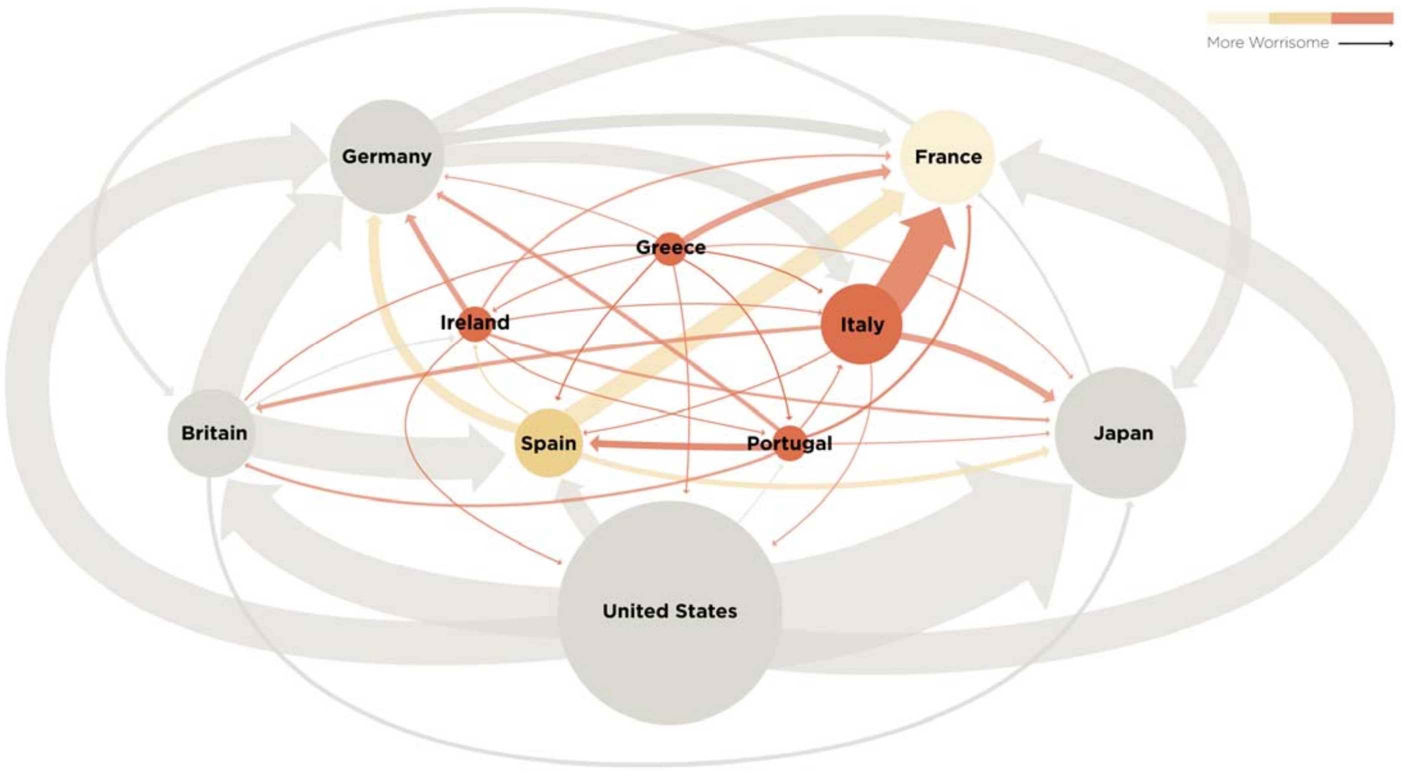
Funded by:

Role of networks

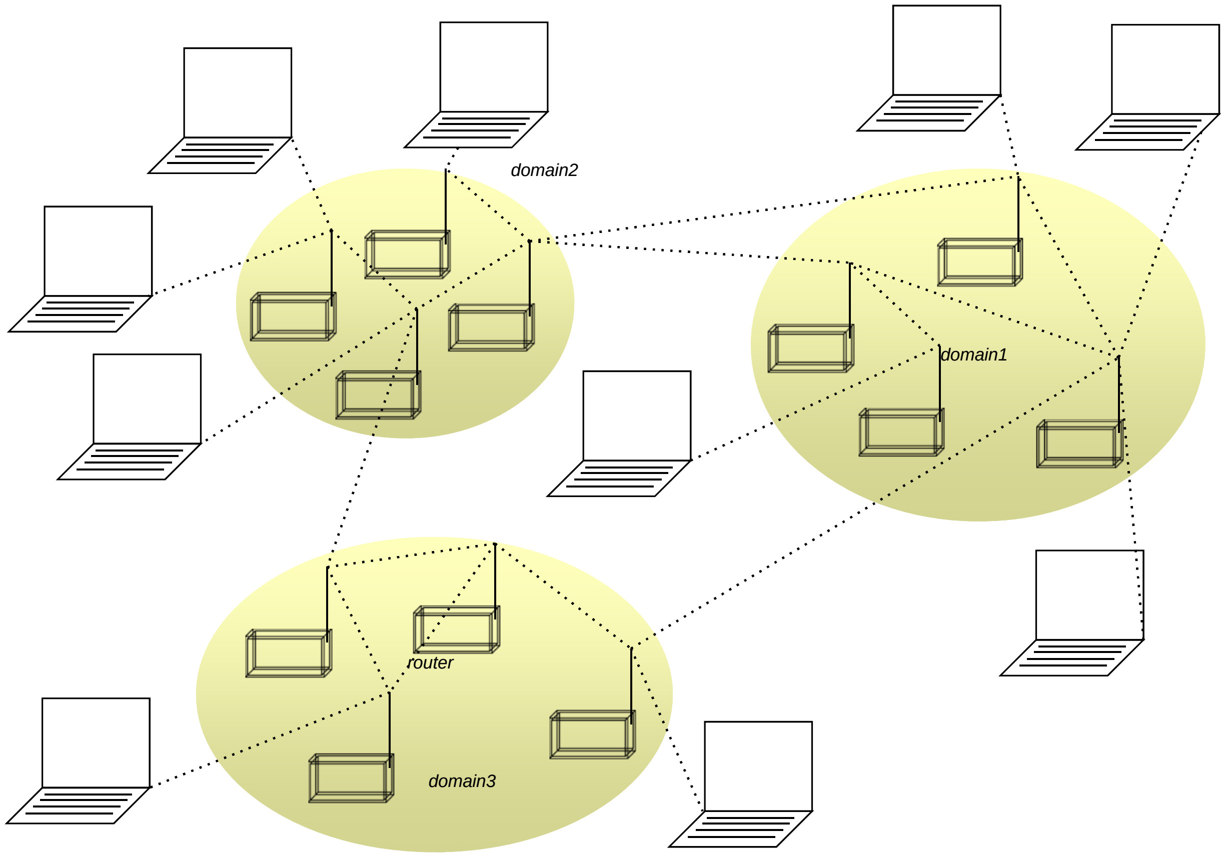
- Behind each complex system there is a network, that defines the interactions between the component.

Network	Nodes	Links	Directed / Undirected	N	L	‹K›
Internet	Routers	Internet connections	Undirected	192,244	609,066	6.34
WWW	Webpages	Links	Directed	325,729	1,497,134	4.60
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594	2.67
Mobile-Phone Calls	Subscribers	Calls	Directed	36,595	91,826	2.51
Email	Email addresses	Emails	Directed	57,194	103,731	1.81
Science Collaboration	Scientists	Co-authorships	Undirected	23,133	93,437	8.08
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908	83.71
Citation Network	Papers	Citations	Directed	449,673	4,689,479	10.43
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802	5.58
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930	2.90

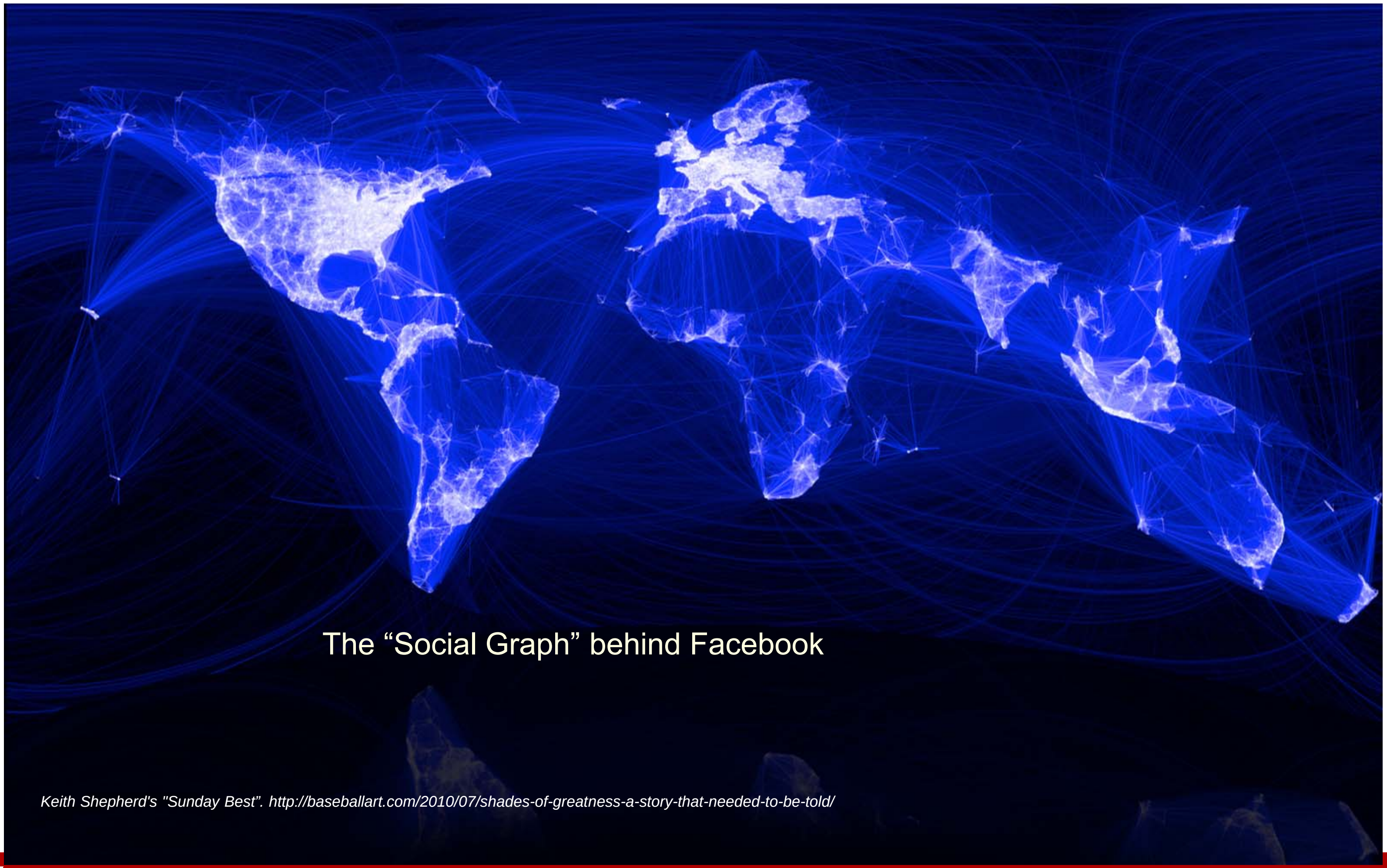
Financial Network



Internet



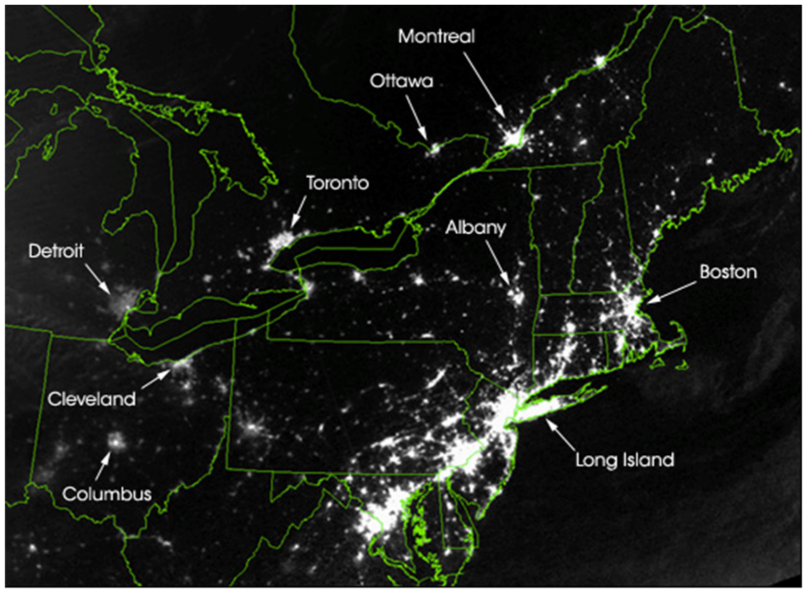
Facebook



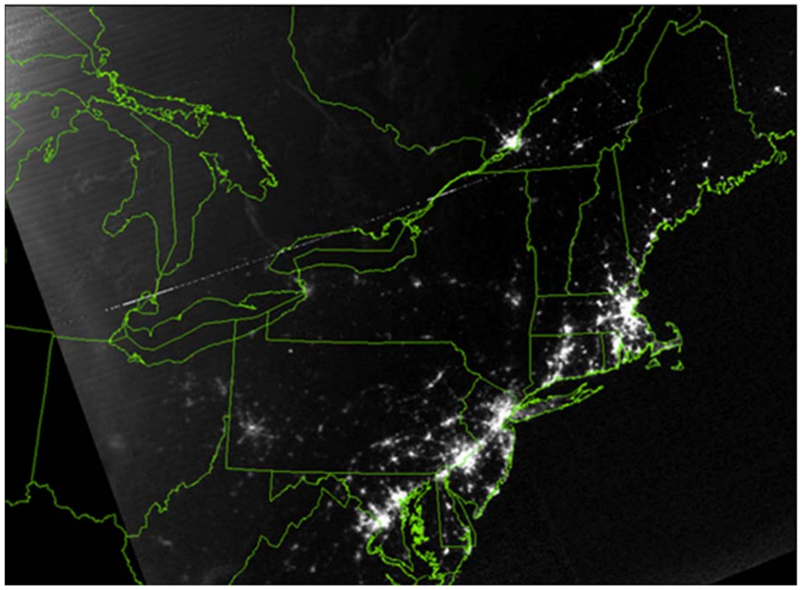
The “Social Graph” behind Facebook

Keith Shepherd's "Sunday Best”. http://baseballart.com/2010/07/shades-of-greatness-a-story-that-needed-to-be-told/

Power network and outages



August 14, 2003: 9:29pm EDT
20 hours before



August 15, 2003: 9:14pm EDT
7 hours after

Topics Covered

- Definitions
- Representation
- Sub-graphs
- Connectivity
- Trees and MST
- Hamilton and Euler definitions
- Shortest Path
- Planar Graphs
- Graph Coloring

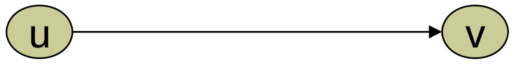
Definitions - Graph

A generalization of the simple concept of a set of dots, links, edges or arcs.

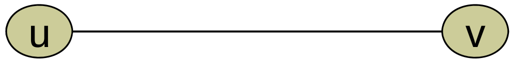
Representation: Graph G =(V, E) consists of a set of vertices denoted by V, or by V(G) and set of edges E, or E(G)

Definitions – Edge Type

Directed: Ordered pair of vertices. Represented as (u, v) directed from vertex u to v.

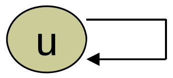


Undirected: Unordered pair of vertices. Represented as {u, v}. Disregards any sense of direction and treats both end vertices interchangeably.

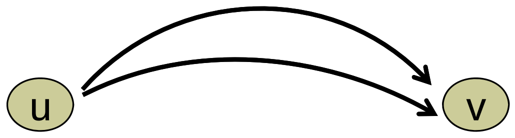


Definitions – Edge Type

- Loop: A loop is an edge whose endpoints are equal i.e., an edge joining a vertex to it self is called a loop. Represented as {u, u} = {u}



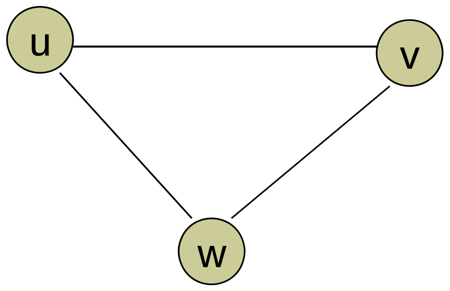
- Multiple Edges: Two or more edges joining the same pair of vertices.



Definitions – Graph Type

Simple (Undirected) Graph: consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges (undirected)

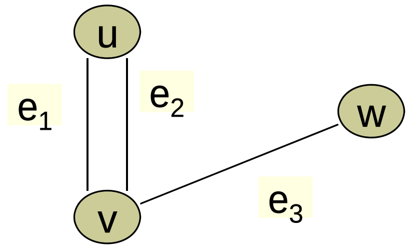
Representation Example: G(V, E), V = {u, v, w}, E = {{u, v}, {v, w}, {u, w}}



Definitions – Graph Type

Multigraph: G(V,E), consists of set of vertices V, set of Edges E and a function f from E to {{u, v}| u, v V, u ≠ v}. The edges e1 and e2 are called multiple or parallel edges if f (e1) = f (e2).

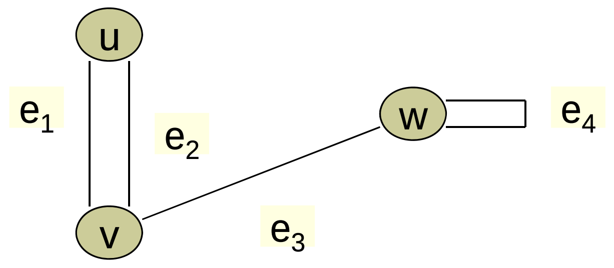
Representation Example: V = {u, v, w}, E = {e_1, e_2, e_3}



Definitions – Graph Type

Pseudograph: G(V,E), consists of set of vertices V, set of Edges E and a function F from E to {{u, v}| u, v Î V}. Loops allowed in such a graph.

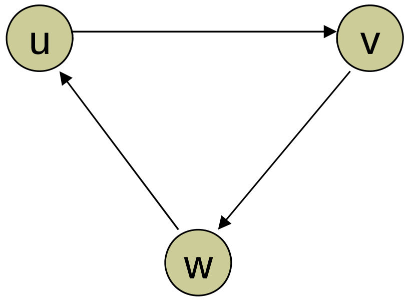
Representation Example: V = {u, v, w}, E = {e_1, e_2, e_3, e_4}



Definitions – Graph Type

Directed Graph: G(V, E), set of vertices V, and set of Edges E, that are ordered pair of elements of V (directed edges)

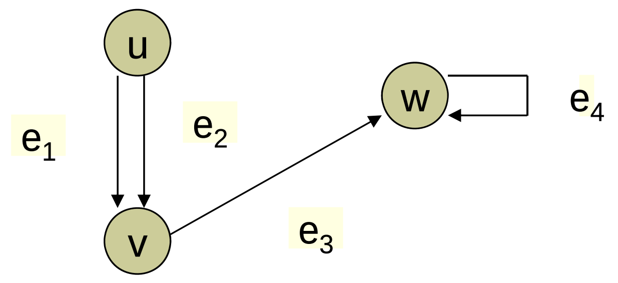
Representation Example: G(V, E), V = {u, v, w}, E = {(u, v), (v, w), (w, u)}



Definitions – Graph Type

Directed Multigraph: G(V,E), consists of set of vertices V, set of Edges E and a function f from E to {{u, v}| u, v V}. The edges e1 and e2 are multiple edges if f(e1) = f(e2)

Representation Example: V = {u, v, w}, E = {e_1, e_2, e_3, e_4}



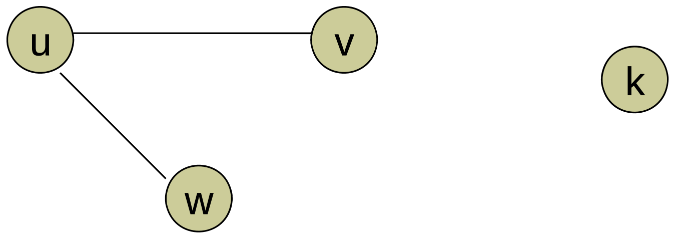
Definitions – Graph Type

Type	Edges	Multiple Edges Allowed?	Loops Allowed?	Example
Simple Graph	undirected	No	No	Mobile phone calls
Multigraph	undirected	Yes	No	Collaboration network
Pseudograph	undirected	Yes	Yes	Facebook Friendship links
Directed Graph	directed	No	Yes	Protein Interactions
Directed Multigraph	directed	Yes	Yes	WWW

Terminology – Undirected graphs

- u and v are adjacent if {u, v} is an edge, e is called incident with u and v. u and v are called endpoints of {u, v}
- Degree of Vertex (deg (v)): the number of edges incident on a vertex. A loop contributes twice to the degree (why?).
- Pendant Vertex: deg (v) =1
- Isolated Vertex: deg (k) = 0

Representation Example: For V = {u, v, w} , E = { {u, w}, {u, v} }, deg (u) = 2, deg (v) = 1, deg (w) = 1, deg (k) = 0, w and v are pendant , k is isolated



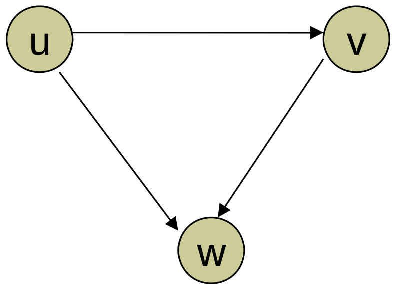
Terminology – Directed graphs

- For the edge (u, v), u is adjacent to v OR v is adjacent from u, u – Initial vertex, v – Terminal vertex
- In-degree ($deg^-(u)$): number of edges for which u is terminal vertex
- Out-degree ($deg^+(u)$): number of edges for which u is initial vertex

Note: A loop contributes 1 to both in-degree and out-degree (why?)

Representation Example: For V = {u, v, w} , E = { (u, w), (v, w), (u, v) }, **$deg^-(u) = 0$, $deg^+(u) = 2$, $deg^-(v) = 1$,**

$deg^+(v) = 1$, and $deg^-(w) = 2$, $deg^+(u) = 0$



Theorems: Undirected Graphs

Theorem 1

The Handshaking theorem:

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

Every edge connects 2 vertices

Theorems: Undirected Graphs

Theorem 2:

An undirected graph has even number of vertices with odd degree

Pr*oof* $V1$ is the set of even degree vertices and V2 refers to odd degree vertices

$$2e = \sum_{v \in V} \deg(v) = \sum_{u \in V_1} \deg(u) + \sum_{v \in V_2} \deg(v)$$

$\Rightarrow$ deg (v) is even for $v \in V_1$,

$\Rightarrow$ The first term in the right hand side of the last inequality is even.

$\Rightarrow$ The sum of the last two terms on the right hand side of

the last inequality is even since sum is 2e.

Hence second term is also even

$\Rightarrow$ second term $\sum_{v \in V_2} \deg(u) = \mathit{even}$

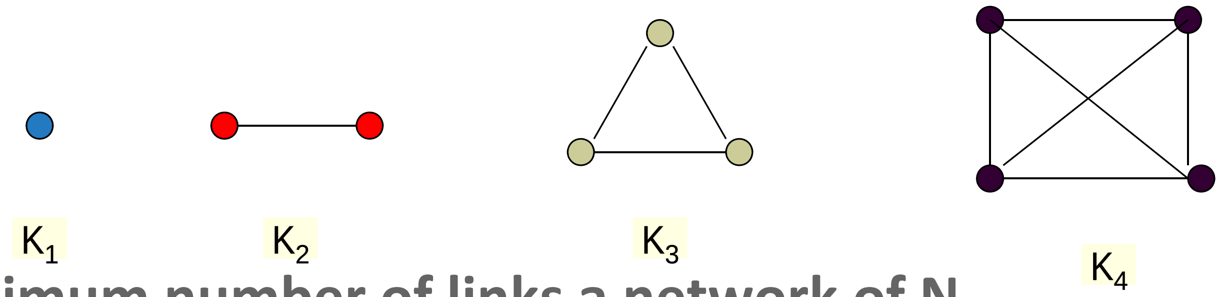
Theorems: directed Graphs

- **Theorem 3:** $\sum \deg^{+}(u) = \sum \deg^{-}(u) = |E|$

Simple graphs – special cases

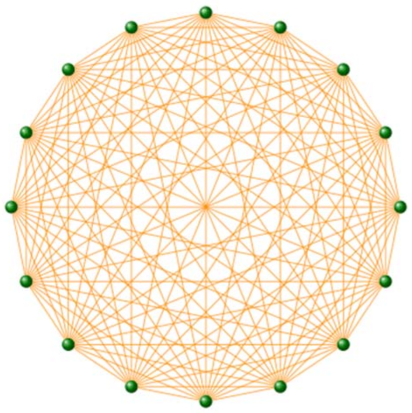
- Complete graph: K_n, is the simple graph that contains exactly one edge between each pair of distinct vertices.

Representation Example: K_1, K_2, K_3, K_4



- The maximum number of links a network of N nodes can have is:

$$L_{\max} = \binom{N}{2} = \frac{N(N-1)}{2}$$

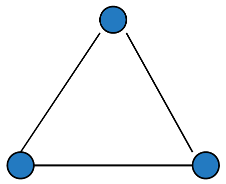


A graph with degree $L=L_{max}$ is called a complete graph, and its average degree is **$\langle k \rangle = N-1$**

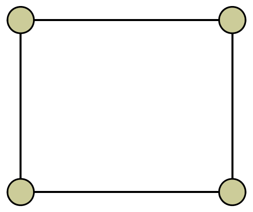
Simple graphs – special cases

- Cycle: C_n, $n \geq 3$ consists of n vertices $v_1, v_2, v_3 \dots v_n$ and edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_3, v_4\} \dots \{v_{n-1}, v_n\}$, $\{v_n, v_1\}$

Representation Example: C_3, C_4



C_3

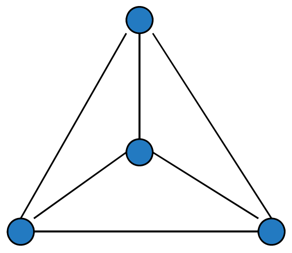


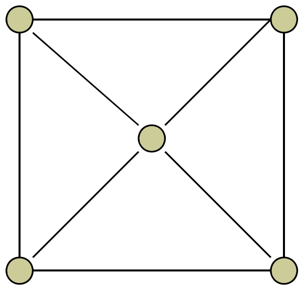
C_4

Simple graphs – special cases

- **Wheels: W_n, obtained by adding additional vertex to Cn and connecting all vertices to this new vertex by new edges.**

  Representation Example: W_3, W_4

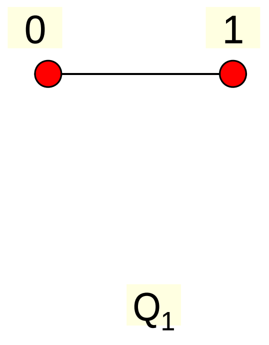

W_3

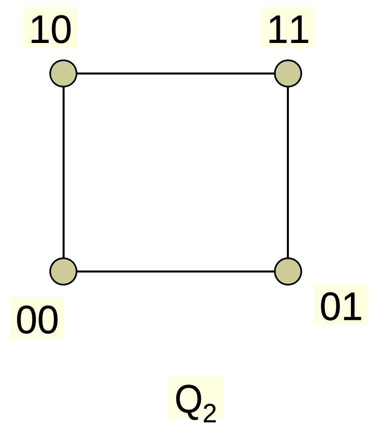

W_4

Simple graphs – special cases

- N-cubes: Q_n, vertices represented by 2n bit strings of length n. Two vertices are adjacent if and only if the bit strings that they represent differ by exactly one bit positions

Representation Example: Q_1, Q_2



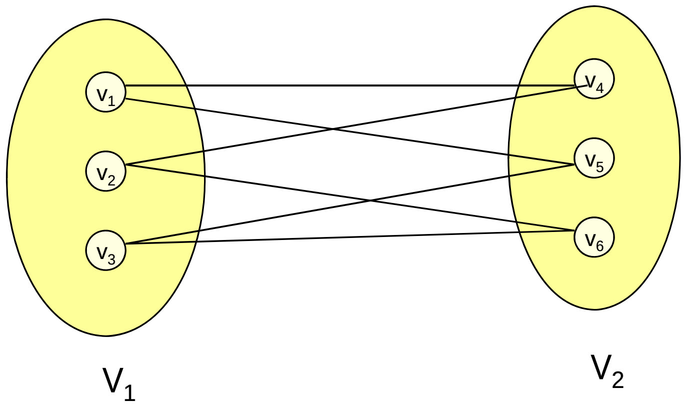


Bipartite graphs

- In a simple graph G, if V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex V_2 (so that no edge in G connects either two vertices in V_1 or two vertices in V_2)

  Application example: Representing Relations

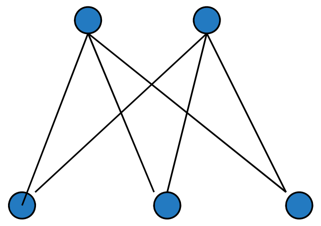
  Representation example: $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, v_5, v_6\}$,

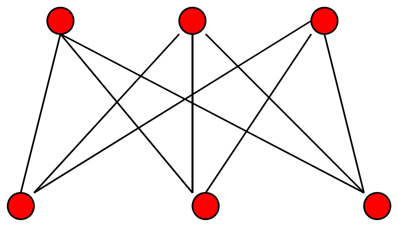


Complete Bipartite graphs

- $K_{m,n}$ is the graph that has its vertex set portioned into two subsets of m and n vertices, respectively. There is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.

Representation example: $K_{2,3}$, $K_{3,3}$


$K_{2,3}$

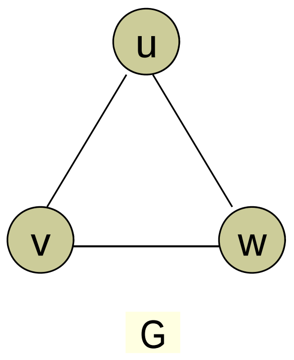

$K_{3,3}$

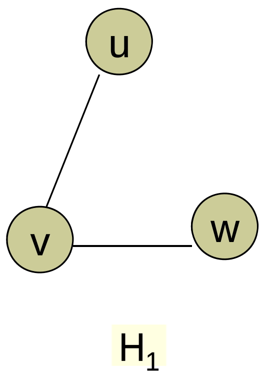
Subgraphs

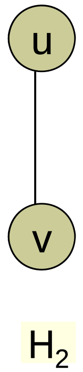
- A subgraph of a graph G = (V, E) is a graph H =(V’, E’) where V’ is a subset of V and E’ is a subset of E

  Application example: solving sub-problems within a graph

  Representation example: V = {u, v, w}, E = ({u, v}, {v, w}, {w, u}}, H_1 , H_2





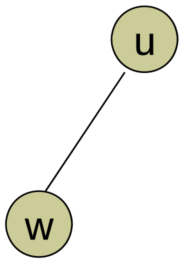


Subgraphs

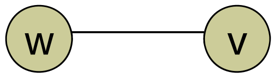
- G = G1 U G2 wherein E = E1 U E2 and V = V1 U V2, G, G1 and G2 are simple graphs of G

Representation example: V1 = {u, w}, E1 = {{u, w}}, V2 = {w, v},

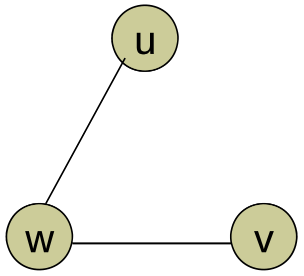
E1 = {{w, v}}, V = {u, v ,w}, E = {{{u, w}, {{w, v}}



G1



G2



G

Representation

- **Incidence** (Matrix): Most useful when information about edges is more desirable than information about vertices.

- **Adjacency** (Matrix/List): Most useful when information about the vertices is more desirable than information about the edges. These two representations are also most popular since information about the vertices is often more desirable than edges in most applications

Representation- Incidence Matrix

- G = (V, E) be an unditected graph. Suppose that $v_1, v_2, v_3, \ldots, v_n$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the n x m matrix M = [m_{ij}], where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise} \end{cases}$$

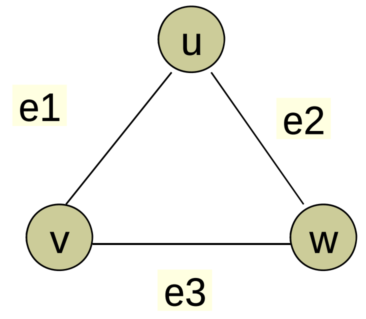
Can also be used to represent :

Multiple edges: by using columns with identical entries, since these edges are incident with the same pair of vertices

Loops: by using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with the loop

Representation- Incidence Matrix

- Representation Example: G = (V, E)



	e_1	e_2	e_3
v	1	0	1
u	1	1	0
w	0	1	1

Representation- Adjacency Matrix

- There is an N x N matrix, where |V| = N , the Adjacenct Matrix (NxN) A = [aij]

For undirected graph

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

For directed graph

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

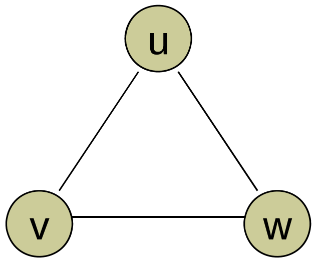
- This makes it easier to find subgraphs, and to reverse graphs if needed.

Representation- Adjacency Matrix

- Adjacency is chosen on the ordering of vertices. Hence, there are as many as n! such matrices.
- The adjacency matrix of simple graphs are symmetric (a_{ij} = a_{ji})
- When there are relatively few edges in the graph the adjacency matrix is a sparse matrix
- Directed Multigraphs can be represented by using aij = number of edges from v_i to v_j

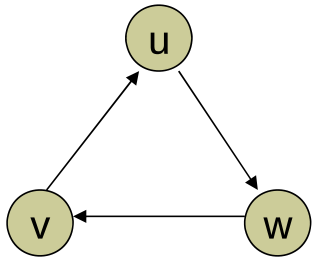
Representation- Adjacency Matrix

- Example: Undirected Graph G (V, E)



	v	u	w
v	0	1	1
u	1	0	1
w	1	1	0

- Example: Directed Graph G (V, E)

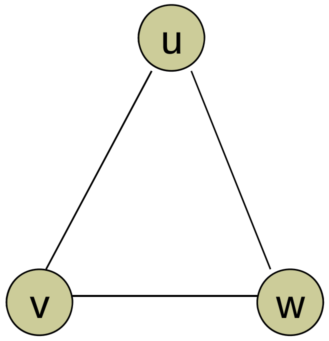


	v	u	w
v	0	1	0
u	0	0	1
w	1	0	0

Representation- Adjacency List

Each node (vertex) has a list of which nodes (vertex) it is adjacent

Example: undirectd graph G (V, E)



node	Adjacency List
u	v , w
v	w, u
w	u , v

Counting Walks

- Theorem: Let G be a graph with adjacency matrix A with respect to the ordering v_1, v_2, …, V_n (with directed on undirected edges, with multiple edges and loops allowed). The number of different paths of length r from Vi to Vj, where r is a positive integer, equals the (i, j)th entry of (adjacency matrix) $A^{r.}$

  Proof: By Mathematical Induction.

  Base Case: For the case N = 1, a_{ij} =1 implies that there is a path of length 1. This is true since this corresponds to an edge between two vertices.

  We assume that theorem is true for N = r and prove the same for N = r +1. Assume that the (i, j)th entry of A^r is the number of different paths of length r from v_i to v_j. By induction hypothesis, b_{ik} is the number of paths of length r from v_i to v_k.

Counting Walks

Case r +1: In $A^{r+1} = A^r . A$,

The (i, j)th entry in A^{r+1} , $b_{i1}a_{1j} + b_{i2}\, a_{2j} + \ldots + b_{in}\, a_{nj}$

where b_{ik} is the (i, j)th entry of A^r.

By induction hypothesis, b_{ik} is the number of paths of length r from v_i to v_k.

The (i, j)th entry in A^{r+1} corresponds to the length between i and j and the length is r+1. This path is made up of length r from v_i to v_k and of length from v_k to vj. By product rule for counting, the number of such paths is $b_{ik*}\, a_{kj}$ The result is $b_{i1}a_{1j} + b_{i2}\, a_{2j} + \ldots + b_{in}\, a_{nj}$, the desired result.

Counting Walks

```
a ------- b
 |         |
 |         |
 c --------d
```

$$A = \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix} \qquad A^4 = \begin{matrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{matrix}$$

Number of walks of length 4 from a to d is (1,4) th entry of $A^4 = 8$.

Graph - Isomorphism

- G1 = (V1, E2) and G2 = (V2, E2) are isomorphic if:
- There is a one-to-one function f from V1 to V2 with the property that
  - a and b are adjacent in G1 if and only if f (a) and f (b) are adjacent in G2, for all a and b in V1.
- Function f is called isomorphism

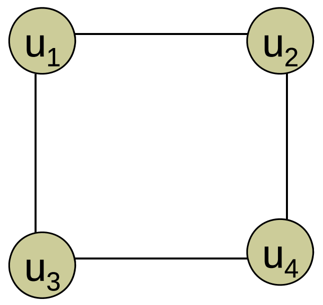
Application Example:

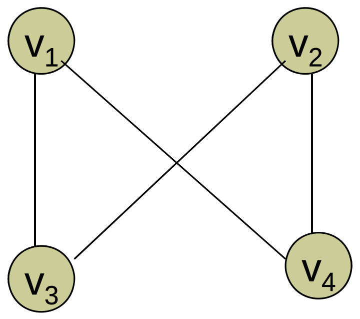
In chemistry, to find if two compounds have the same structure

Graph - Isomorphism

Representation example: G1 = (V1, E1) , G2 = (V2, E2)

$f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, $f(u_4) = v_2$,





Properties of the Adjacency Matrix

- **Isomorphic graphs**
  - There is a permutation matrix P such that $B = PAP^{-1}$
  - A permutation matrix is a square matrix whose entries are all 0 or 1, such that each row and each column contains exactly one 1.
- **Directed graph is k-regular if the degree of each vertex is k and eigenvalue is k.**
- **Perron–Frobenius Theorem**
  - Largest eigenvalue λ_{max}~ average degree : $\max\{d, \sqrt{d_{max}}\} \leq \lambda_{max} \leq d_{max}$
  - Bipartite graph iff $\lambda_{min}(G) = -\lambda_{max}(G)$

Connectivity

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

  Application Example:

  - In a city to city road-network, if one city can be reached from another city.

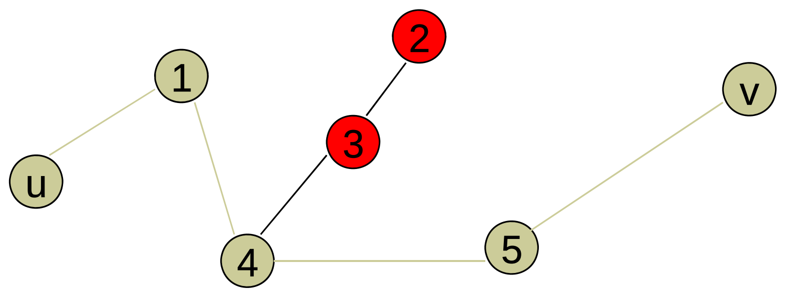
  - Problems if determining whether a message can be sent between two computers using intermediate links

  - Efficiently planning routes for data delivery in the Internet

Connectivity – Path

A Path is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices.

Representation example: G = (V, E), Path P represented, from u to v is {{u, 1}, {1, 4}, {4, 5}, {5, v}}



Connectivity – Path

Definition for Directed Graphs

A Path of length n (> 0) from u to v in G is a sequence of n edges e_1, e_2, e_3, …, e_n of G such that f (e_1) = (x_0, x_1), f (e_2) = (x_1, x_2), …, f (e_n) = (x_{n-1}, x_n) where x_0 = u and x_n = v. A path is said to pass through x_0, x_1, …, x_n or traverse e_1, e_2, e_3, …, e_n

For Simple Graphs, sequence is x_0, x_1, …, x_n

In directed multigraphs when it is not necessary to distinguish between their edges, we can use sequence of vertices to represent the path

Circuit/Cycle: u = v, length of path > 0

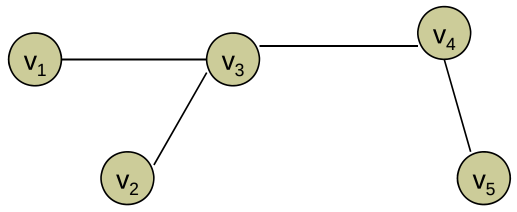
Simple Path: does not contain an edge more than once

Connectivity – Connectedness

Undirected Graph

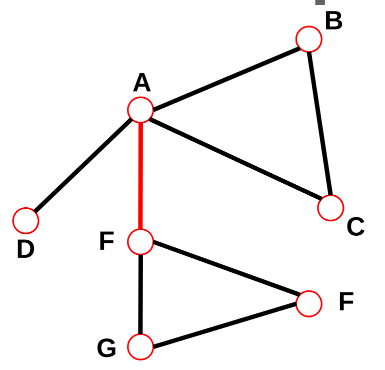
An undirected graph is connected if there exists a simple path between every pair of vertices

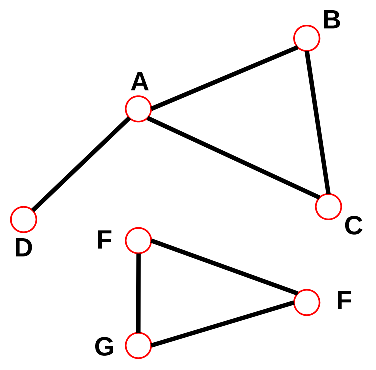
Representation Example: G (V, E) is connected since for V = {v_1, v_2, v_3, v_4, v_5}, there exists a path between {v_i, v_j}, 1 ≤ i, j≤ 5



Connectivity – Connectedness

- Connected (undirected) graph: any two vertices can be joined by a path.
- A disconnected graph is made up by two or more connected components.





Largest Component:
Giant Component

The rest: **Isolates**

Bridge: if we erase it, the graph becomes disconnected.

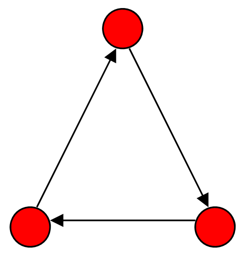
Connectivity – Connectedness

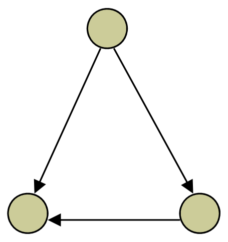
- **Strongly connected directed** graph: has a path from each node to every other node **and vice versa** (e.g. AB path and BA path).
- **Weakly connected** directed graph: it is connected if we disregard the edge directions.
- Strongly connected components can be identified, but not every node is part of a nontrivial strongly connected component.

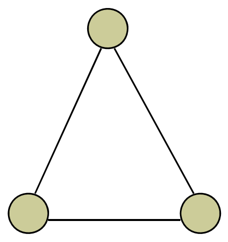
Connectivity – Connectedness

Directed Graph

Representation example: G1 (Strong component), G2 (Weak Component), G3 is undirected graph representation of G2 or G1


G1


G2

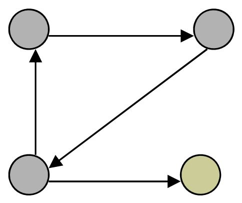

G3

Connectivity – Connectedness

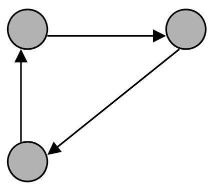
- Directed Graph

  Strongly connected Components: subgraphs of a Graph G that are strongly connected

  Representation example: G1 is the strongly connected component in G



G



G1

Paths and cycles

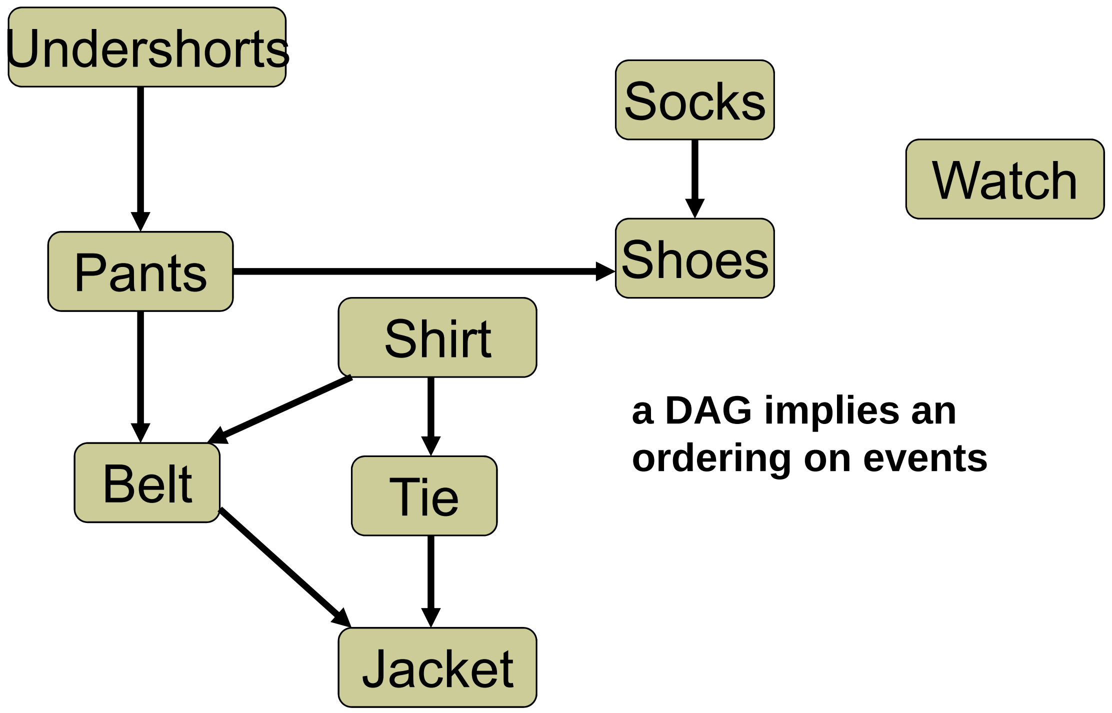
- **A *path* is a sequence of nodes $v_1, v_2, \ldots, v_N$ such that $(v_i, v_{i+1}) \in E$ for $0<i<N$**
  - The *length* of the path is N-1.
  - *Simple path*: all v_i are distinct, $0<i<N$

- **A *cycle* is a path such that $v_1=v_N$**
  - An *acyclic* graph has no cycles

Trees are graphs

- A *dag* is a **d**irected **a**cyclic **g**raph.
- A *tree* is a connected acyclic undirected graph.
- A *forest* is an acyclic undirected graph (not necessarily connected), i.e., each connected component is a tree.

Example DAG



a DAG implies an ordering on events

Topological Sort

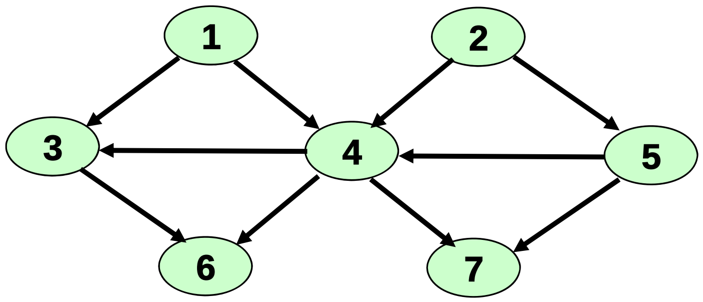
- For a directed acyclic graph G = (V,E)
- A topological sort is an ordering of all of G's vertices $v_1, v_2, \ldots, v_n$ such that...

Formally: for every edge (v_i,v_k) in *E*, i<k.

Visually: all arrows are pointing to the right

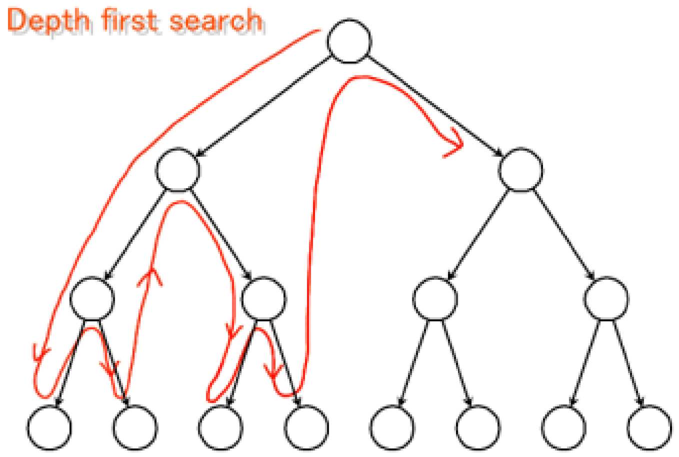
Topological Sort

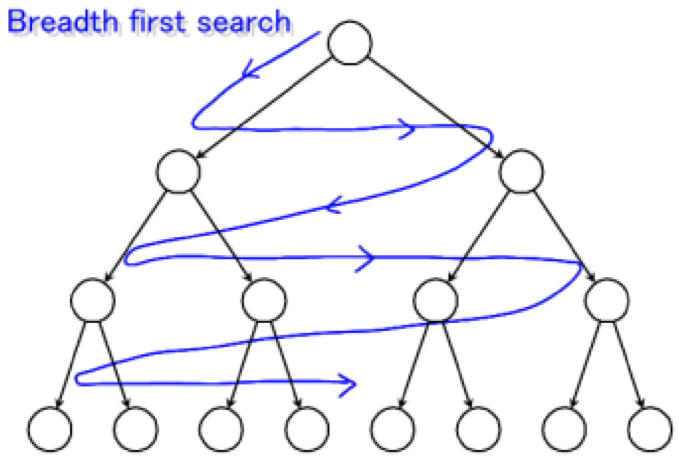
- There are often many possible topological sorts of a given DAG
- Topological orders for this DAG :
    - 1,2,5,4,3,6,7
    - 2,1,5,4,7,3,6
    - 2,5,1,4,7,3,6
    - *Etc.*


$$\begin{tikzcd}
& 1 \arrow[dl] \arrow[dr] & & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[ll] \arrow[dl] \arrow[dr] & & 5 \arrow[ll] \arrow[dl] \\
& 6 & & 7 &
\end{tikzcd}$$

- Each topological order is a ***feasible schedule***.

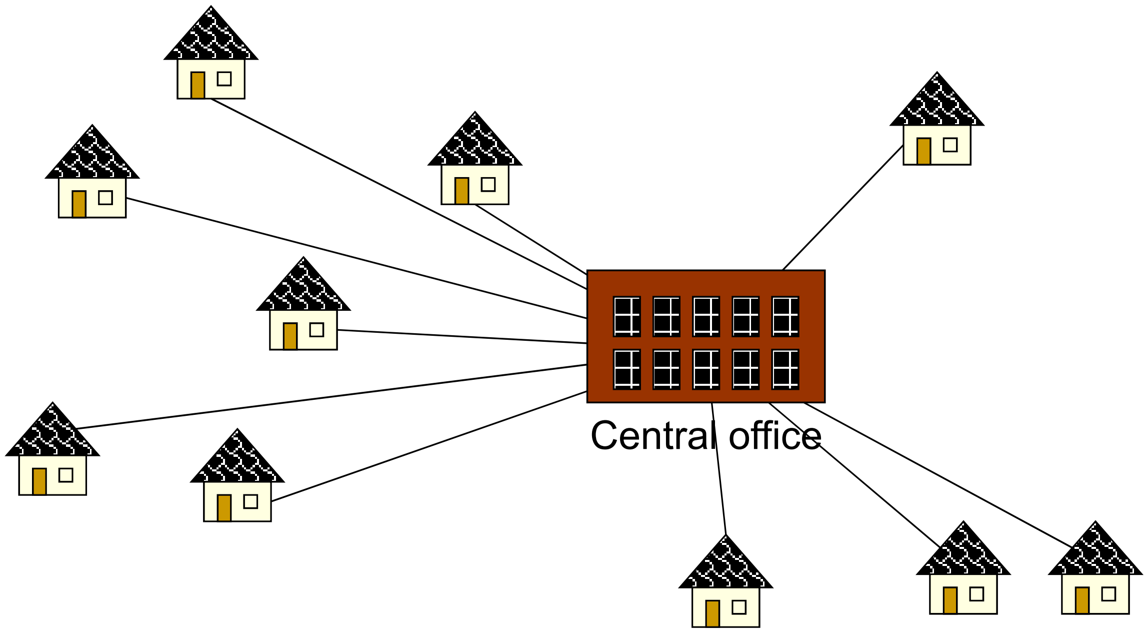
Graph Traversals





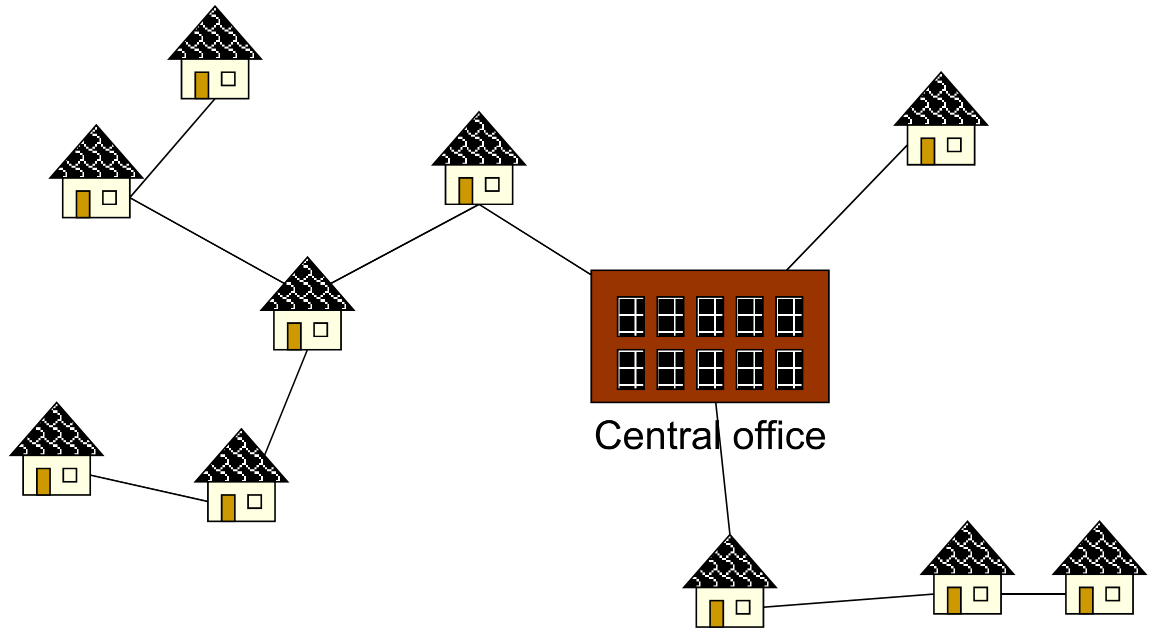
Both take time: O(V+E)

Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



Minimize the total length of wire connecting the customers

Minimum Spanning Tree (MST)

A **minimum spanning tree** is a subgraph of an undirected weighted graph ***G***, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices V
  - contains $|V|$ - 1 edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

Prim's Algorithm

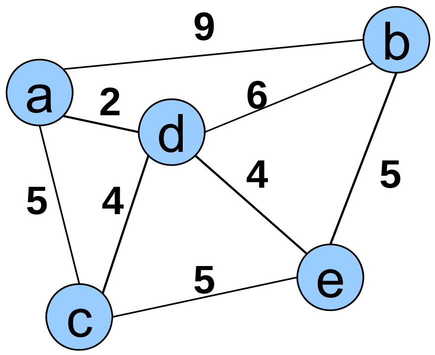
Initialization

a. Pick a vertex r to be the root

b. Set $D(r) = 0$, $parent(r) = null$

c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$

d. Insert all vertices into priority queue P, using distances as the keys



e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-

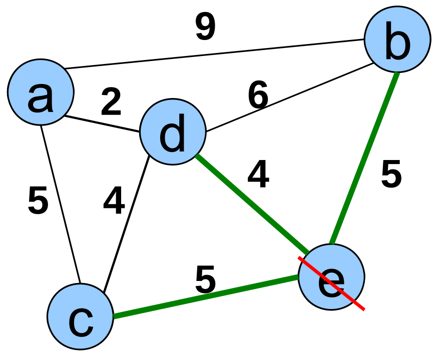
Prim's Algorithm

```
While $P$ is not empty:

  1. Select the next vertex $u$ to add to the tree
         $u = P.deleteMin()$

  2. Update the weight of each vertex $w$ adjacent to
     $u$ which is not in the tree (i.e., $w \in P$)
         If $weight(u,w) < D(w)$,
           a. $parent(w) = u$
           b. $D(w) = weight(u,w)$
           c. Update the priority queue to reflect
              new distance for $w$
```

Prim’s algorithm



e	d	b	c	a
0	∞	∞	∞	∞

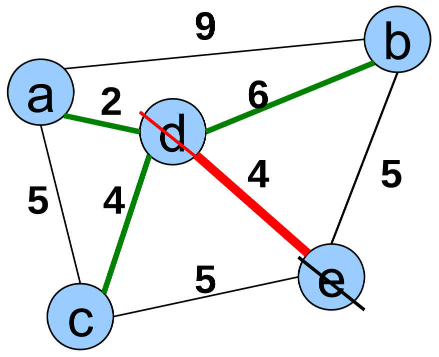
Vertex	Parent
e	-
b	-
c	-
d	-

d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex ***e,*** and we update the distances and parent for its adjacent vertices

Prim's algorithm



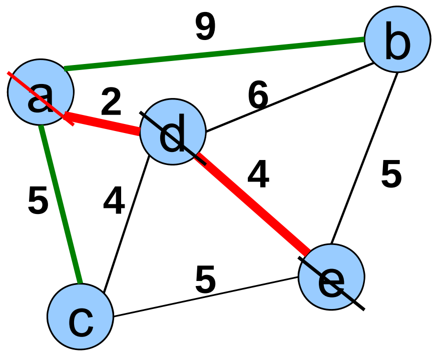
d	b	c	a
4	5	5	∞

Vertex	Parent
e	-
b	e
c	e
d	e

a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim’s algorithm



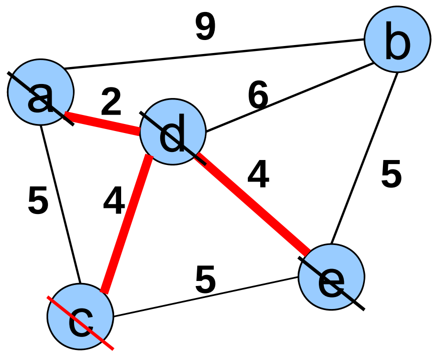
a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



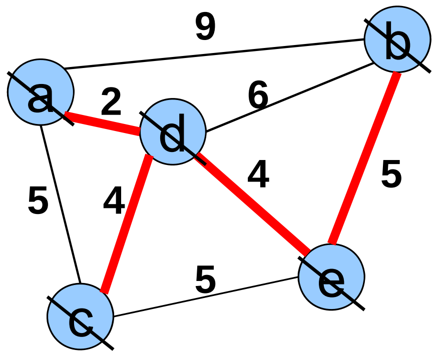
c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Running time of Prim's algorithm (without heaps)

Initialization of priority queue (array): O($|V|$)

Update loop: $|V|$ calls

- Choosing vertex with minimum cost edge: O($|V|$)
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O($|E|$) updates

Overall cost without heaps: $\boxed{O(|E| + |V|^2)}$

Prim’s Algorithm Invariant

- At each step, we add the edge (u,v) s.t. the weight of (u,v) is minimum among all edges where u is in the tree and v is not in the tree
- Each step maintains a minimum spanning tree of the vertices that have been included thus far
- When all vertices have been included, we have a MST for the graph!

Correctness of Prim’s

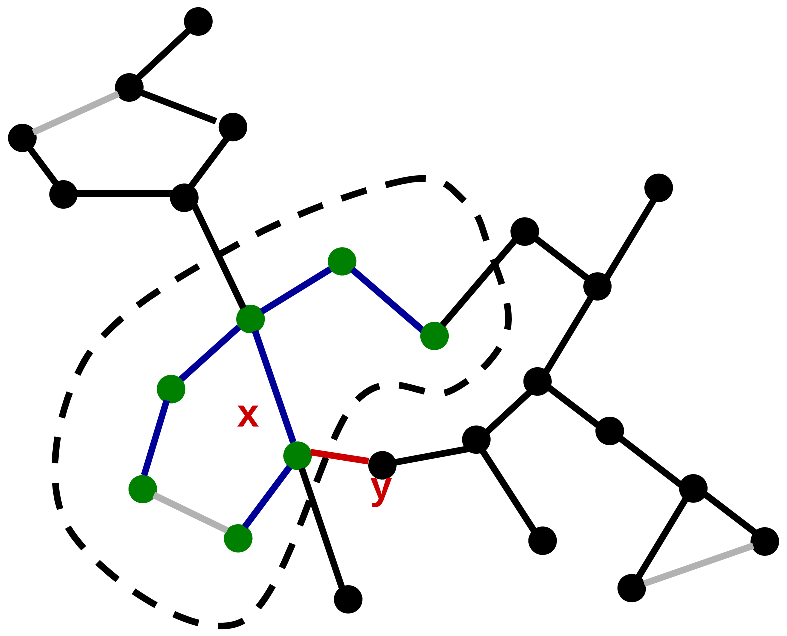
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (***you should be able to prove this***).

But is this a *minimum* spanning tree?

Suppose it wasn't.

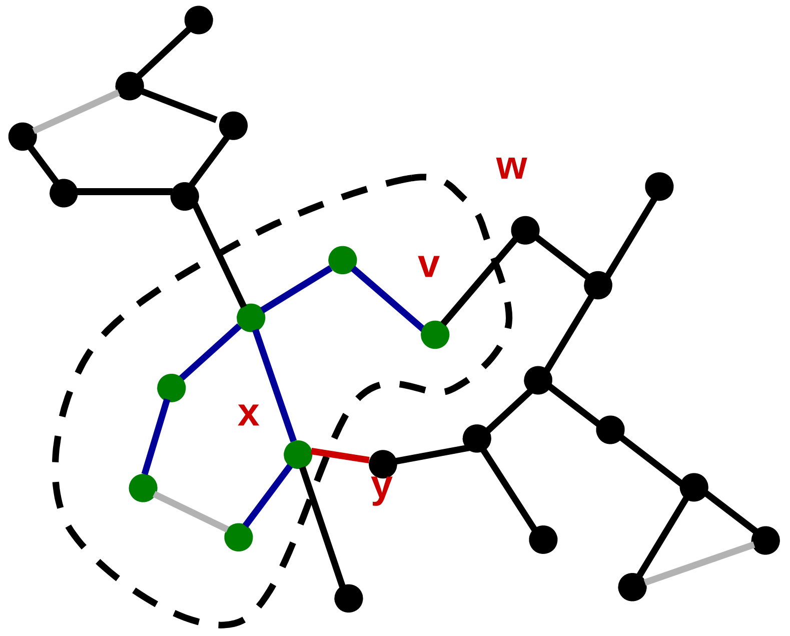
- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Prim's



- Let ***G*** be a connected, undirected graph
- Let **S** be the set of edges chosen by Prim's algorithm *before* choosing an errorful edge **(*x*,*y*)**
- Let **V'** be the vertices incident with edges in **S**
- Let **T** be a MST of ***G*** containing all edges in **S**, but not **(*x*,*y*)**.

Correctness of Prim's



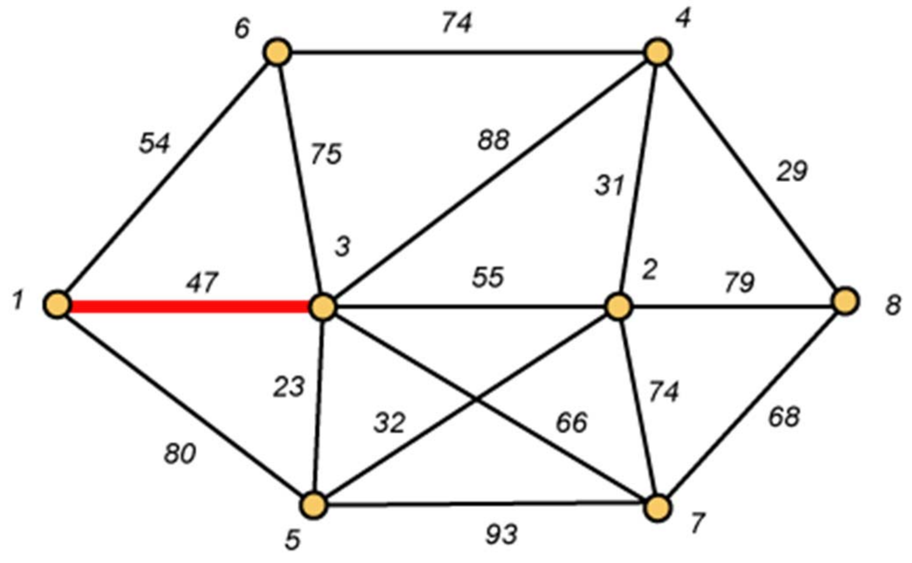
- Edge **(*x*,*y*)** is not in **T**, so there must be a path in **T** from ***x*** to ***y*** since **T** is connected.
- Inserting edge **(*x*,*y*)** into **T** will create a cycle
- There is exactly one edge on this cycle with exactly one vertex in **V'**, call this edge **(*v*,*w*)**

Correctness of Prim’s

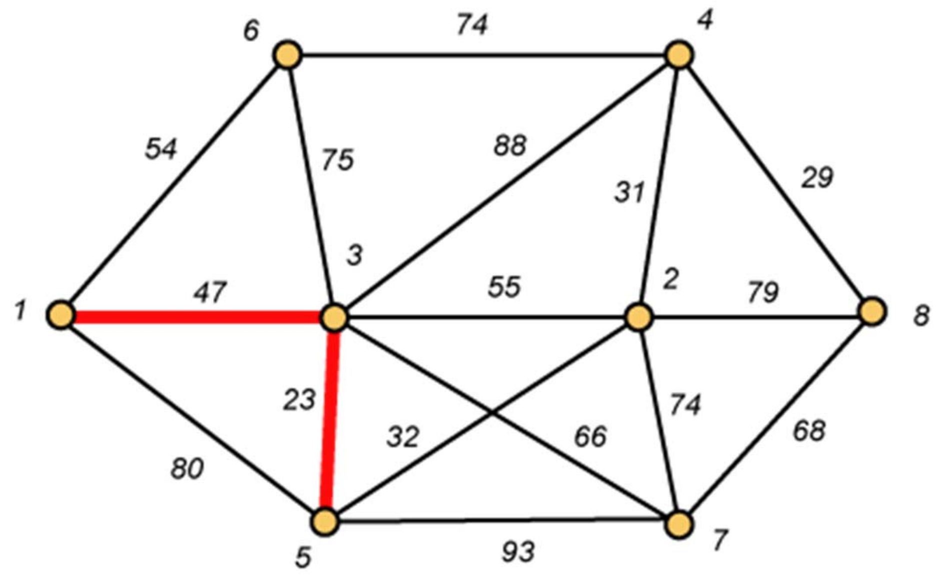
- Since Prim’s chose ***(x,y)*** over (*v*,*w*), w(v,w) >= w(**x,y**).
- We could form a new spanning tree T’ by swapping ***(x,y)*** for (*v*,*w*) in T (***prove this is a spanning tree***).
- w(T’) is clearly no greater than w(T)
- But that means T’ is a MST
- And yet it contains all the edges in **S**, and also ***(x,y)***

...Contradiction

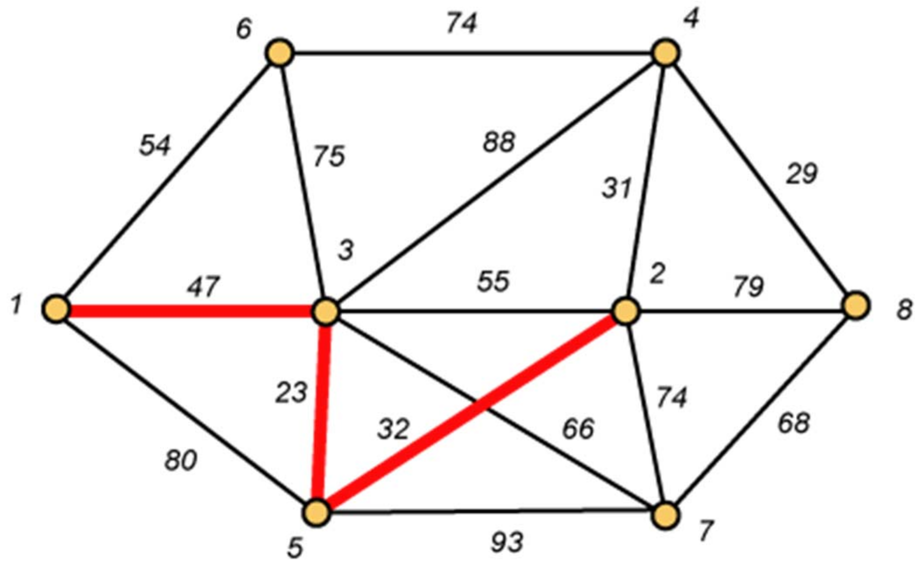
Prim – Step 1



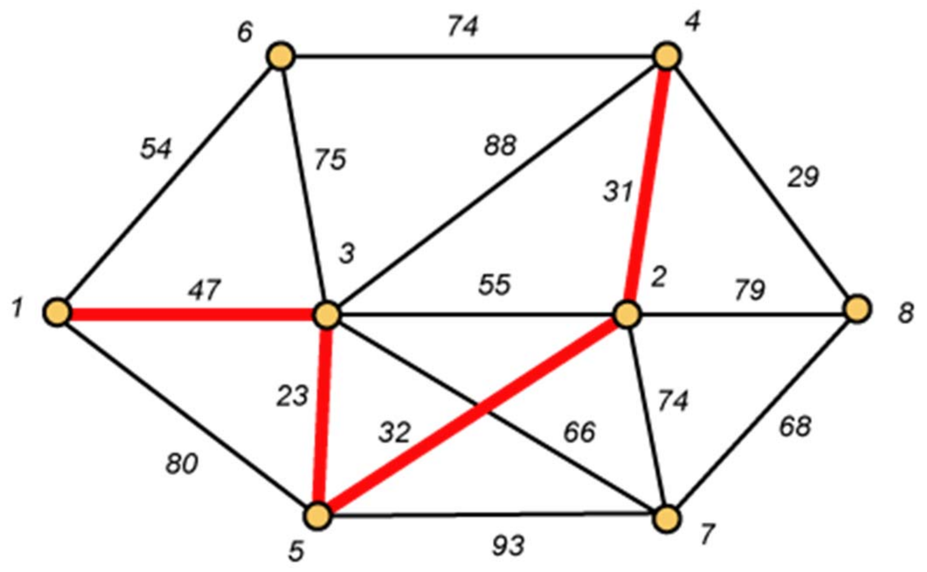
Prim – Step 2



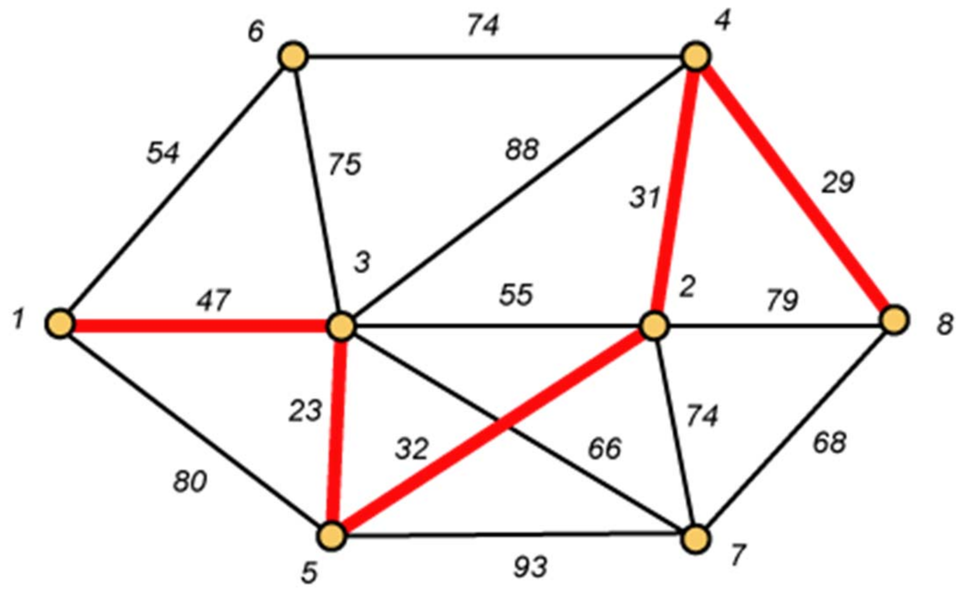
Prim – Step 3



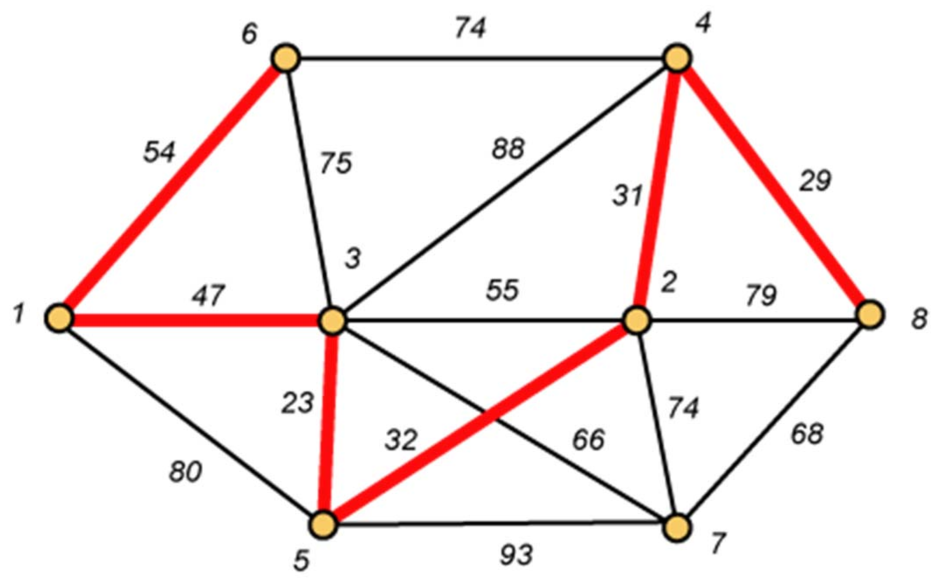
Prim – Step 4



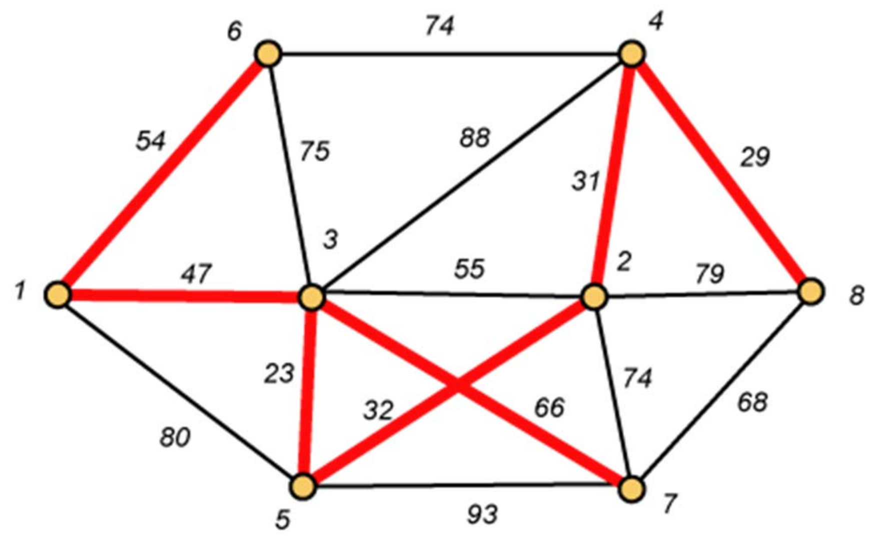
Prim – Step 5



Prim – Step 6



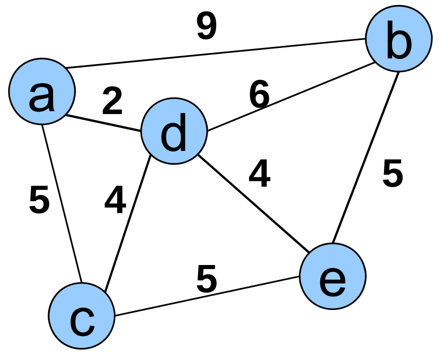
Prim – Step 7 *Done!!*



Weight (T) = 23 + 29 + 31 + 32 + 47 + 54 + 66 = **282**

Another Approach

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains
- A "safe edge" is an edge of minimum weight which does not create a cycle



forest: {a}, {b}, {c}, {d}, {e}

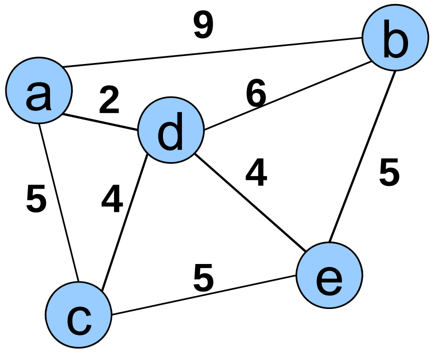
Kruskal’s algorithm

Initialization

a. Create a set for each vertex $v \in V$

b. Initialize the set of “safe edges” A comprising the MST to the empty set

c. Sort edges by increasing weight



F = {a}, {b}, {c}, {d}, {e}

$A = \varnothing$

E = {(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)}

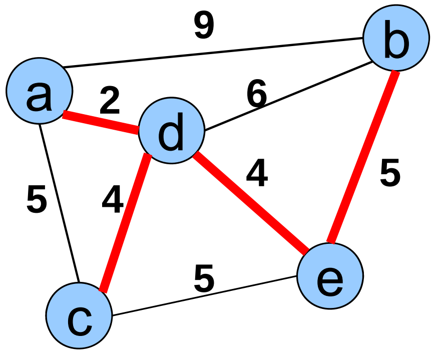
Kruskal’s algorithm

```
For each edge $(u,v) \in E$ in increasing order
while more than one set remains:
    If $u$ and $v$, belong to different sets $U$ and $V$
        a. add edge $(u,v)$ to the safe edge set
           $A = A \cup \{(u,v)\}$
        b. merge the sets $U$ and $V$
           $F = F - U - V + (U \cup V)$

Return $A$
```

- **Running time bounded by sorting (or findMin)**
- **O($|E|\log|E|$), or equivalently, O($|E|\log|V|$) (*why???*)**

Kruskal’s algorithm



E = {~~(a,d)~~, ~~(c,d)~~, ~~(d,e)~~, ~~(a,c)~~,
~~(b,e)~~, (c,e), (b,d), (a,b)}

Forest	***A***
{a}, {b}, {c}, {d}, {e}	$\varnothing$
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Kruskal’s Algorithm Invariant

- After each iteration, every tree in the forest is a MST of the vertices it connects

- Algorithm terminates when all vertices are connected into one tree

Correctness of Kruskal’s

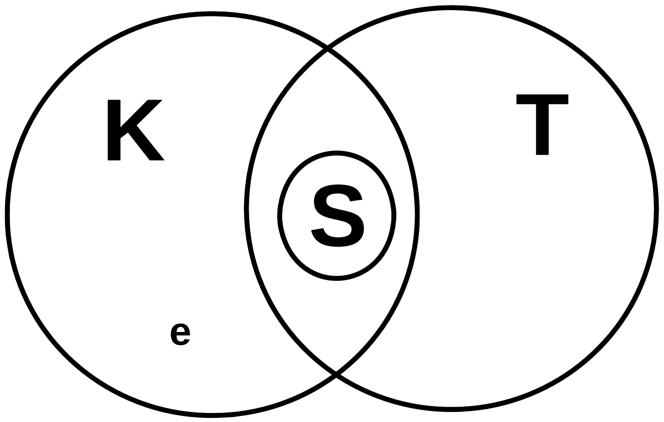
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Kruskal's



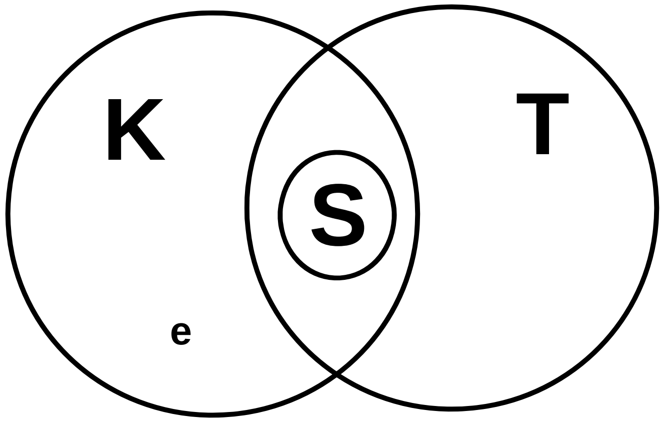
- Let e be this first errorful edge.
- Let K be the Kruskal spanning tree
- Let S be the set of edges chosen by Kruskal's algorithm *before* choosing e
- Let T be a MST containing all edges in S, but not e.

Correctness of Kruskal's

Lemma: w(**e'**) >= w(**e**) for all edges **e'** in **T - S**

Proof (*by contradiction*):

- Assume there exists some edge e' in T - S, w(e') < w(e)
- Kruskal's must have considered e' before e



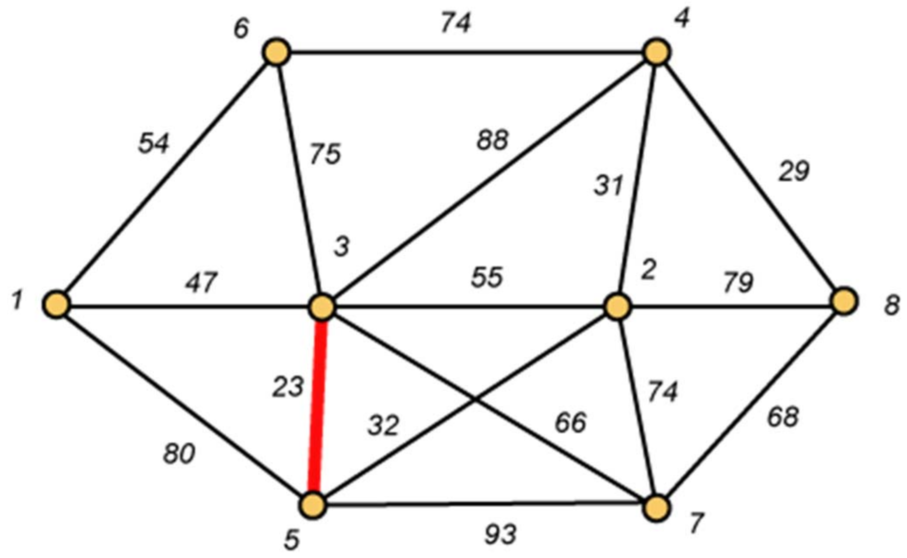
- However, since **e'** is not in **K** (*why??*), it must have been discarded because it caused a cycle with some of the other edges in **S**.
- But **e' + S** is a subgraph of **T**, which means it cannot form a cycle **...Contradiction**

Correctness of Kruskal's

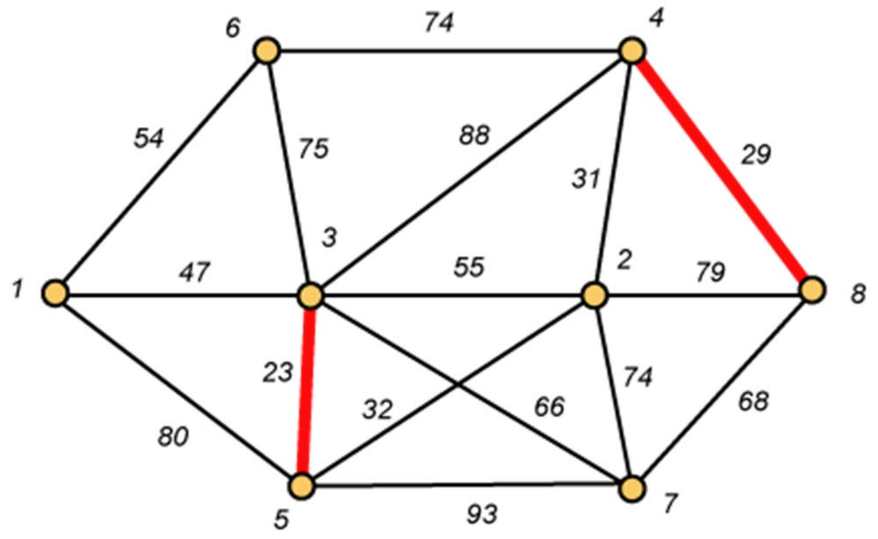
- Inserting edge e into T will create a cycle
- There must be an edge on this cycle which is not in K (***why??***). Call this edge e'
- e' must be in T - S, so (by our lemma) w(e') >= w(e)
- We could form a new spanning tree T' by swapping e for e' in T (***prove this is a spanning tree***).
- w(T') is clearly no greater than w(T)
- But that means T' is a MST
- And yet it contains all the edges in S, and also e

...Contradiction

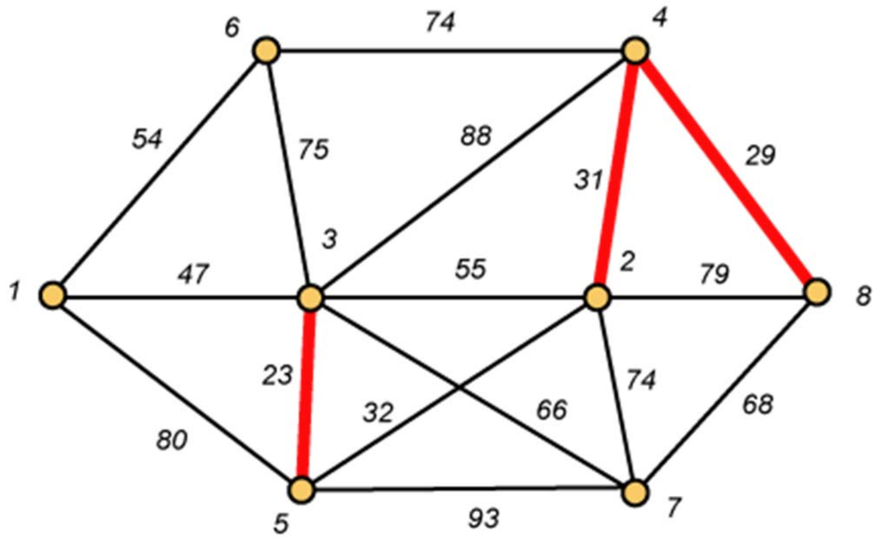
Kruskal – Step 1



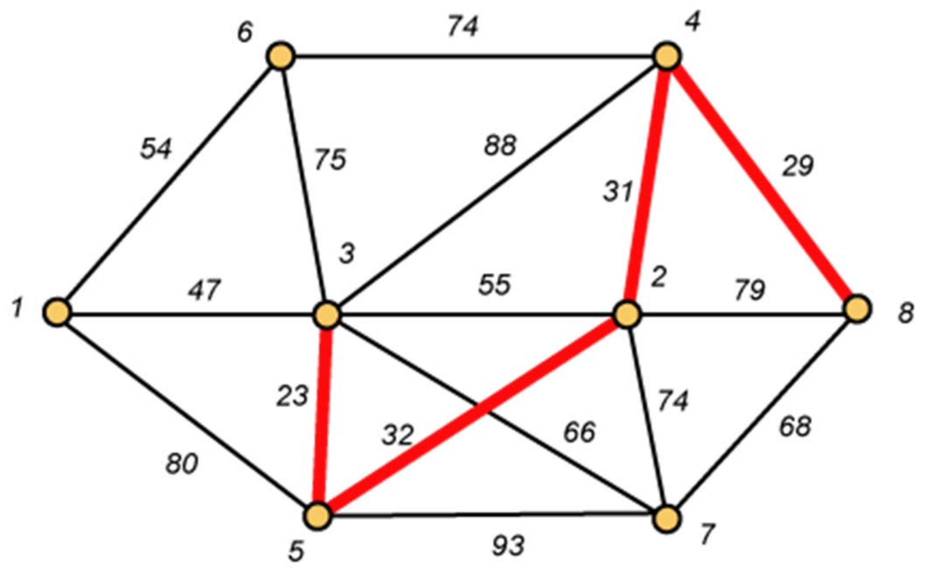
Kruskal – Step 2



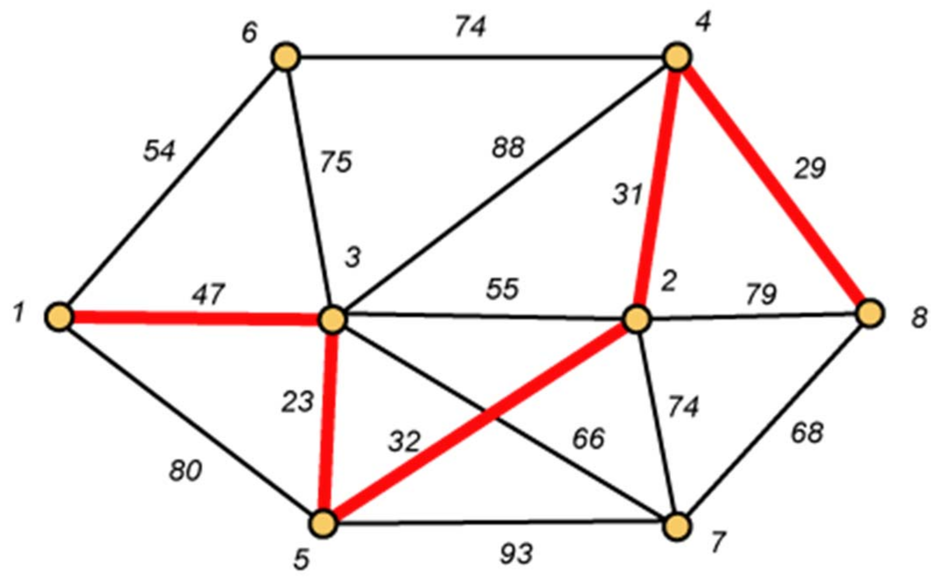
Kruskal – Step 3



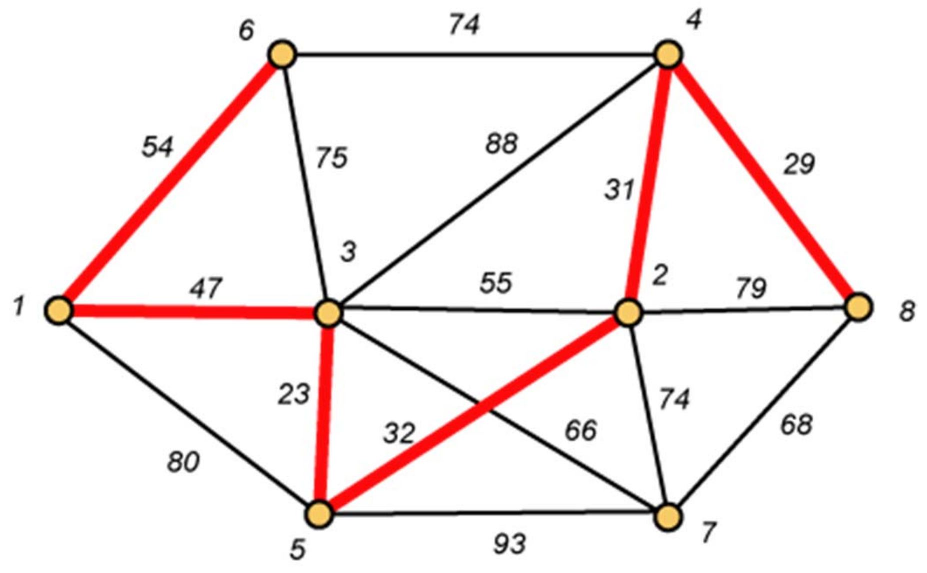
Kruskal – Step 4



Kruskal – Step 5

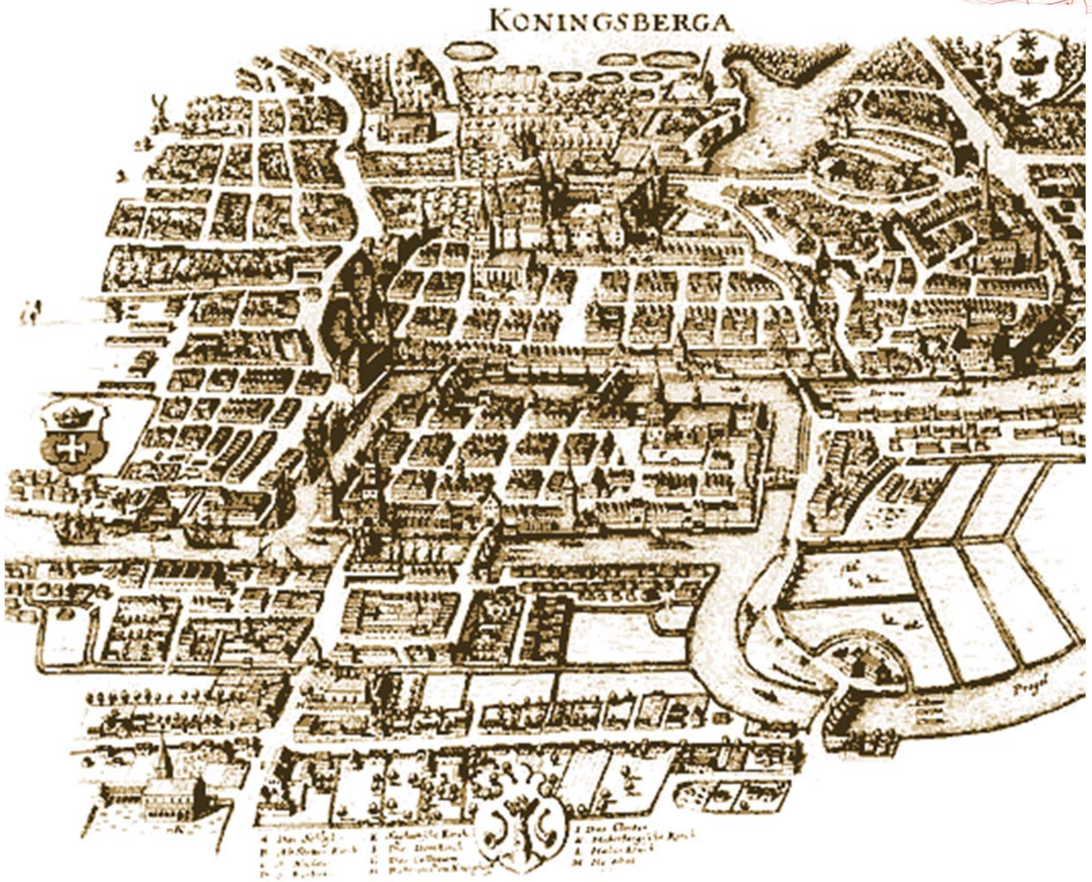


Kruskal – Step 6



The Seven Bridges of Königsberg

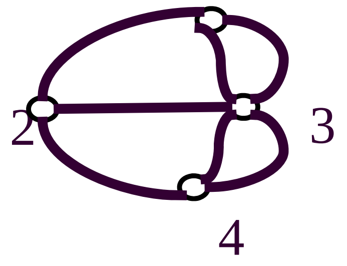
- The residents of Königsberg, Germany, wondered if it was possible to take a walking tour of the town that crossed each of the seven bridges over the Presel river exactly once. Is it possible to start at some node and take a walk that uses each edge exactly once, and ends at the starting node?



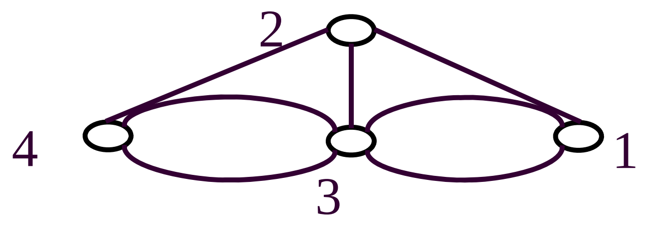
The Seven Bridges of Königsberg

- You can redraw the original picture as long as for every edge between nodes ***i*** and ***j*** in the original you put an edge between nodes ***i*** and ***j*** in the redrawn version (and you put no other edges in the redrawn version).

Original:



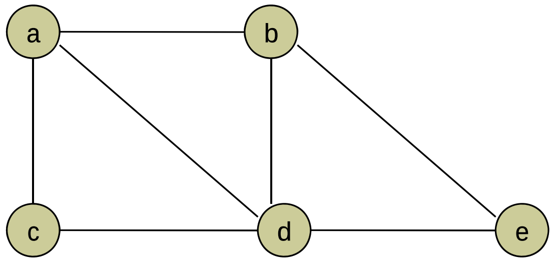
Redrawn:



- Has no tour that uses each edge exactly once

Euler - definitions

- An Eulerian path (Eulerian trail, Euler walk) in a graph is a path that uses each edge precisely once. If such a path exists, the graph is called traversable.
- An Eulerian cycle (Eulerian circuit, Euler tour) in a graph is a cycle that uses each edge precisely once. If such a cycle exists, the graph is called Eulerian (also unicursal).
- Representation example: G1 has Euler path a, c, d, e, b, d, a, b



Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree (each edge has even degree)
2. A connected graph G has a Euler trail from node *a* to some other node b if and only if G is connected and a $\neq$ b are the only two nodes of odd degree

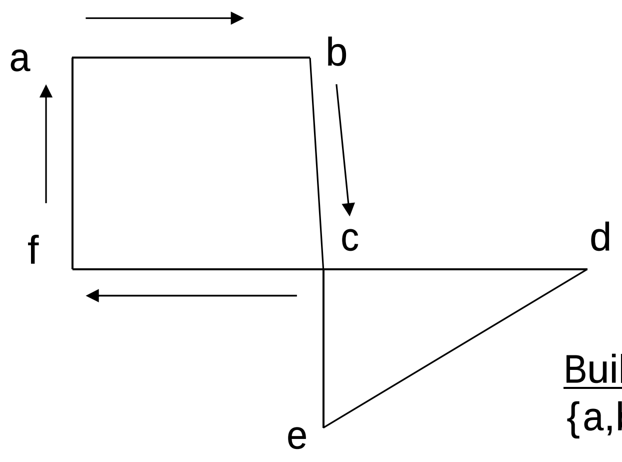
Assume ***G*** has an Euler trail ***T*** from node ***a*** to node ***b*** (***a*** and ***b*** not necessarily distinct).

For every node besides ***a*** and ***b***, ***T*** uses an edge to exit for each edge it uses to enter. Thus, the degree of the node is even.

1. If ***a*** = ***b***, then ***a*** also has even degree. → Euler circuit
2. If ***a*** $\neq$ ***b***, then ***a*** and ***b*** both have odd degree. → Euler path

Euler - examples

- A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree

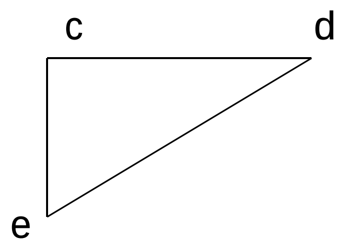


Building a simple path:
{a,b}, {b,c}, {c,f}, {f,a}

Euler circuit constructed if all edges are used. True here?

Euler - examples

A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree

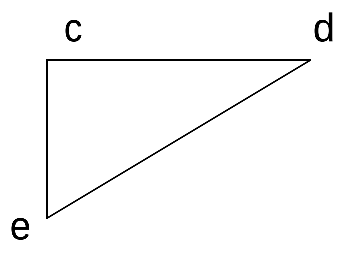


Delete the simple path:
{a,b}, {b,c}, {c,f}, {f,a}

C is the common vertex for this sub-graph with its “parent”.

Euler - examples

A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



Constructed subgraph may not be connected.

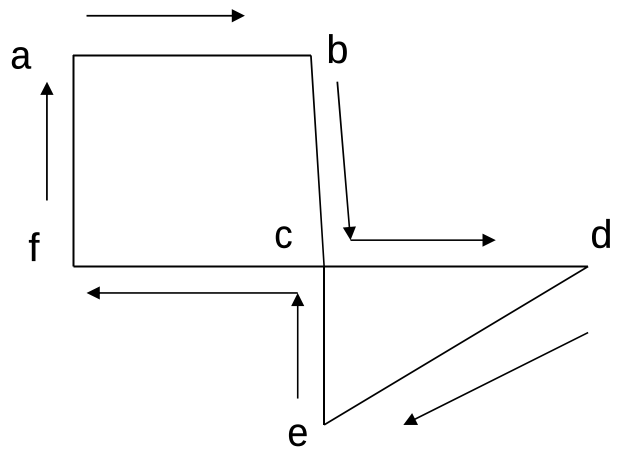
C is the common vertex for this sub-graph with its “parent”.

C has even degree.

Start at c and take a walk:
{c,d}, {d,e}, {e,c}

Euler - examples

A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



“Splice” the circuits in the 2 graphs:

{a,b}, {b,c}, {c,f}, {f,a}

“+”

{c,d}, {d,e}, {e,c}

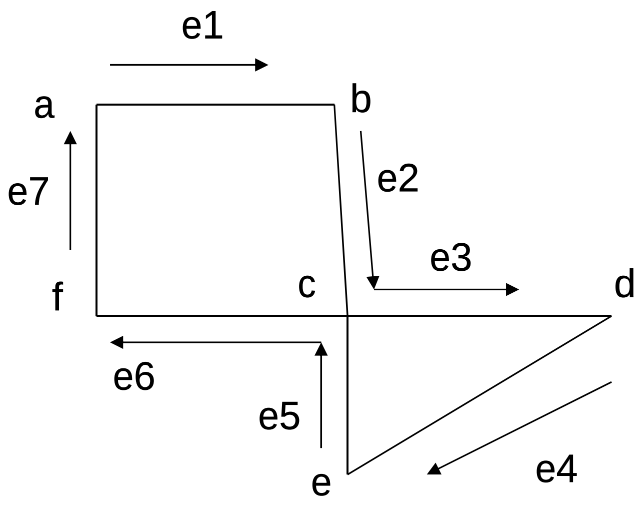
“=“

{a,b}, {b,c}, {c,d}, {d,e}, {e,c}, {c,f} {f,a}

Euler Circuit

1. **Circuit C := a circuit in G beginning at an arbitrary vertex v.**
    1. Add edges successively to form a path that returns to this vertex.
2. **H := G – above circuit C**
3. **While H has edges**
    1. Sub-circuit *sc* := a circuit that begins at a vertex in H that is also in C (e.g., vertex “c”)
    2. H := H – *sc* (- all isolated vertices)
    3. Circuit := circuit C “spliced” with sub-circuit *sc*
4. **Circuit C has the Euler circuit.**

Representation- Incidence Matrix



	e_1	e_2	e_3	e4	e_5	e_6	e_7
a	1	0	0	0	0	0	1
b	1	1	0	0	0	0	0
c	0	1	1	0	1	1	0
d	0	0	1	1	0	0	0
e	0	0	0	1	1	0	0
f	0	0	0	0	0	1	1

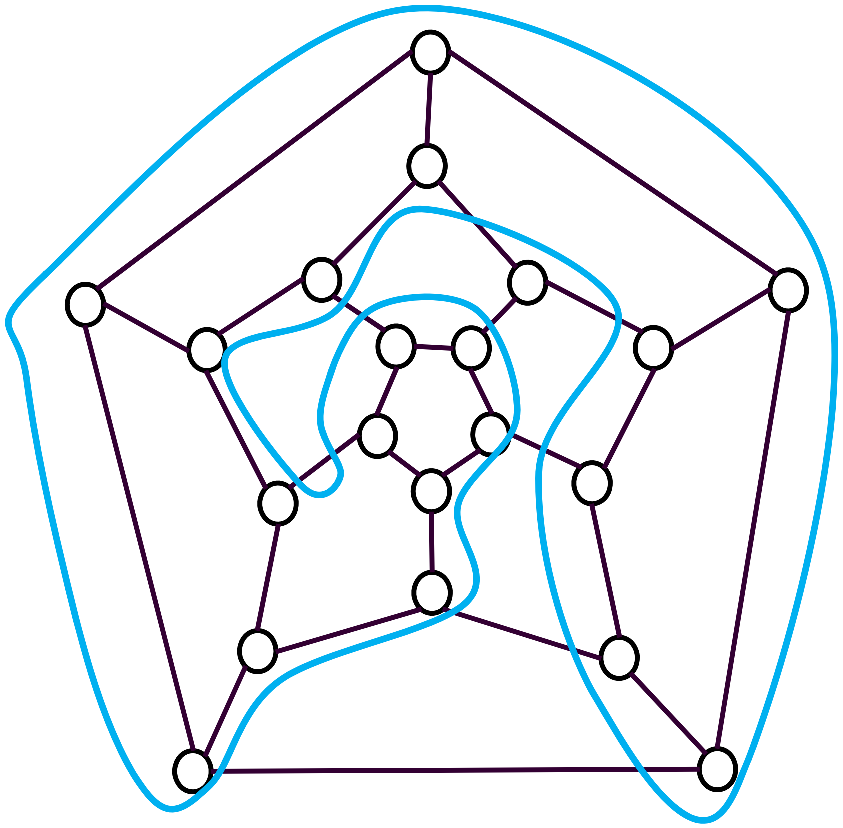
Hamiltonian Graph

- Hamiltonian path (also called *traceable path*) is a path that visits each vertex exactly once.

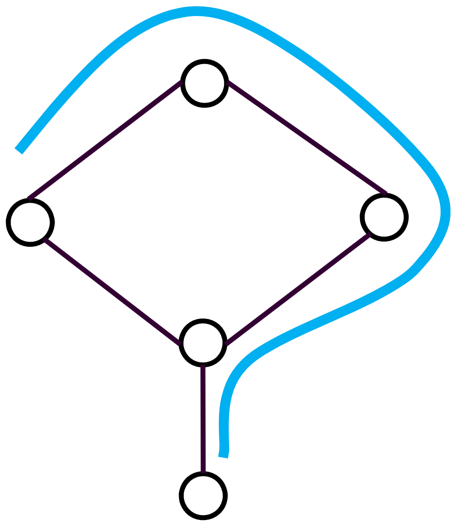
- A Hamiltonian cycle (also called *Hamiltonian circuit*, *vertex tour* or *graph cycle*) is a cycle that visits each vertex exactly once (except for the starting vertex, which is visited once at the start and once again at the end).

- A graph that contains a Hamiltonian path is called a traceable graph. A graph that contains a Hamiltonian cycle is called a Hamiltonian graph. Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges, but a Hamiltonian path can be extended to Hamiltonian cycle only if its endpoints are adjacent.

Dodecahedron example

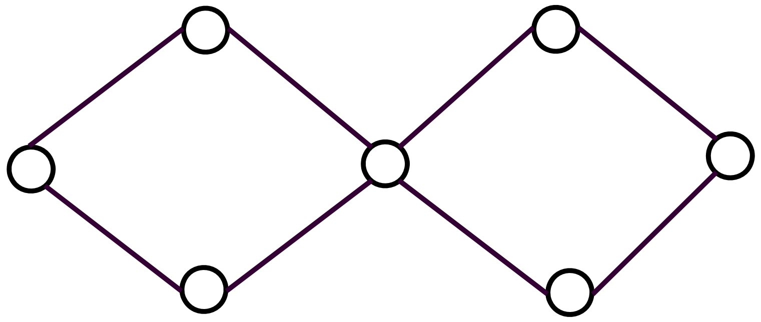


Hamiltonian Graph



- This one has a Hamiltonian path, but not a Hamiltonian tour.

Hamiltonian Graph



- This one has an Euler tour, but no Hamiltonian path

Hamiltonian Graph

- Similar notions may be defined for directed graphs, where edges (arcs) of a path or a cycle are required to point in the same direction, i.e., connected tail-to-head.

- The *Hamiltonian cycle problem* or *Hamiltonian circuit problem* in graph theory is to find a Hamiltonian cycle in a given graph. The *Hamiltonian path problem* is to find a Hamiltonian path in a given graph.

- There is a simple relation between the two problems. The Hamiltonian path problem for graph G is equivalent to the Hamiltonian cycle problem in a graph H obtained from G by adding a new vertex and connecting it to all vertices of G.

- Both problems are NP-complete. However, certain classes of graphs always contain Hamiltonian paths. For example, it is known that every tournament has an odd number of Hamiltonian paths.

Hamiltonian Graph

- DIRAC'S Theorem: if G is a simple graph with n vertices with n ≥ 3 such that the degree of every vertex in G is at least n/2 then G has a Hamilton circuit.

- ORE'S Theorem: if G is a simple graph with n vertices with n ≥ 3 such that deg (u) + deg (v) ≥ n for every pair of nonadjacent vertices u and v in G, then G has a Hamilton circuit.

Shortest Path

- Generalize distance to weighted setting
- Digraph *G* = (*V*,*E*) with weight function *W*: *E* → *R* (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

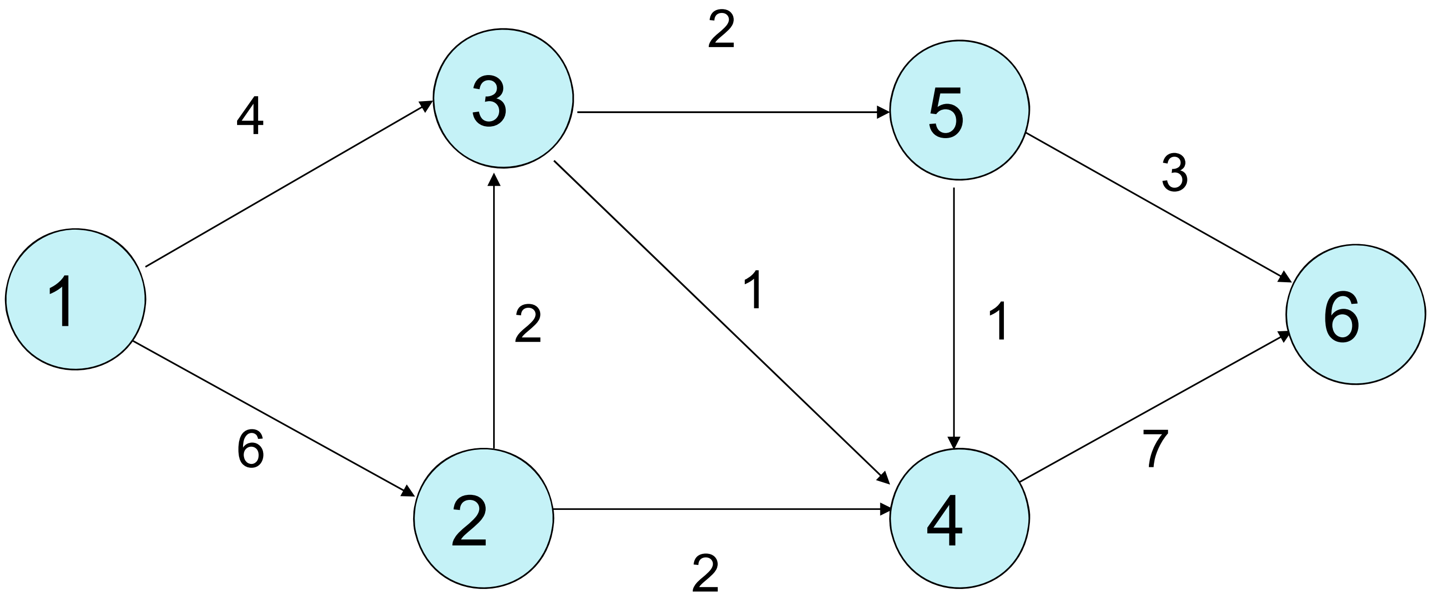
- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest-Path Problems

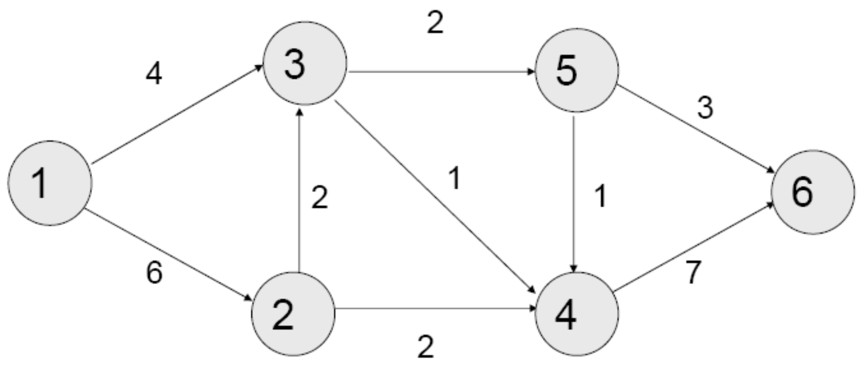
- **Shortest-Path problems**
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
  - Unweighted shortest-paths – BFS.

Algorithm 3 Shortest Path Algorithm on Acyclic Networks

```
begin
    $d(s) = 0$; $d(i) = \infty$ for all $i \in N \setminus \{s\}$;
    $k = 1$;
    while $k \leq n$ do
        pick $i$ such that $order(i) = k$;
        for all arcs $(i, j)$ that emanate from $i$ do
            if $d(j) > d(i) + c_{ij}$ then set $d(j) = d(i) + c_{ij}$ and $pred(j) = i$;        RELAX(u, v)
        end;
    end;
end;
```


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

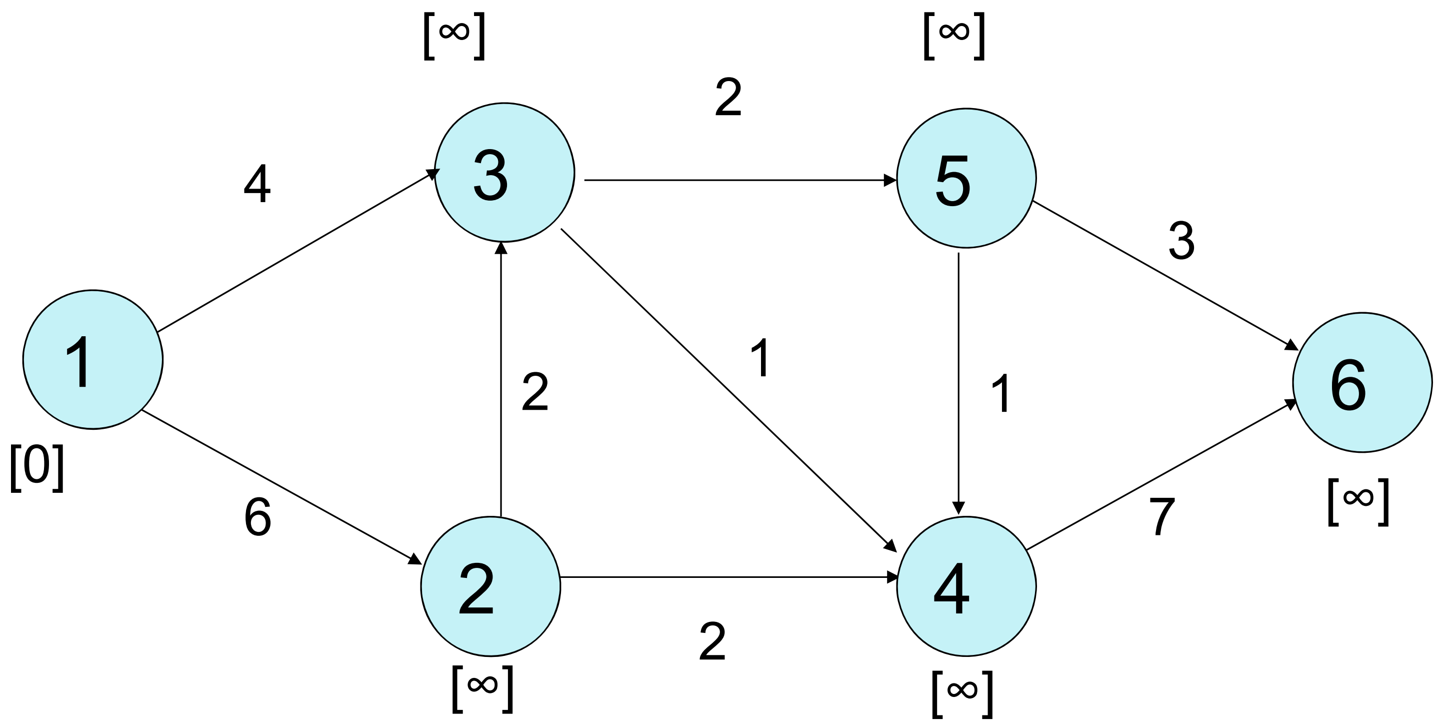
Label setting for acyclic Networks

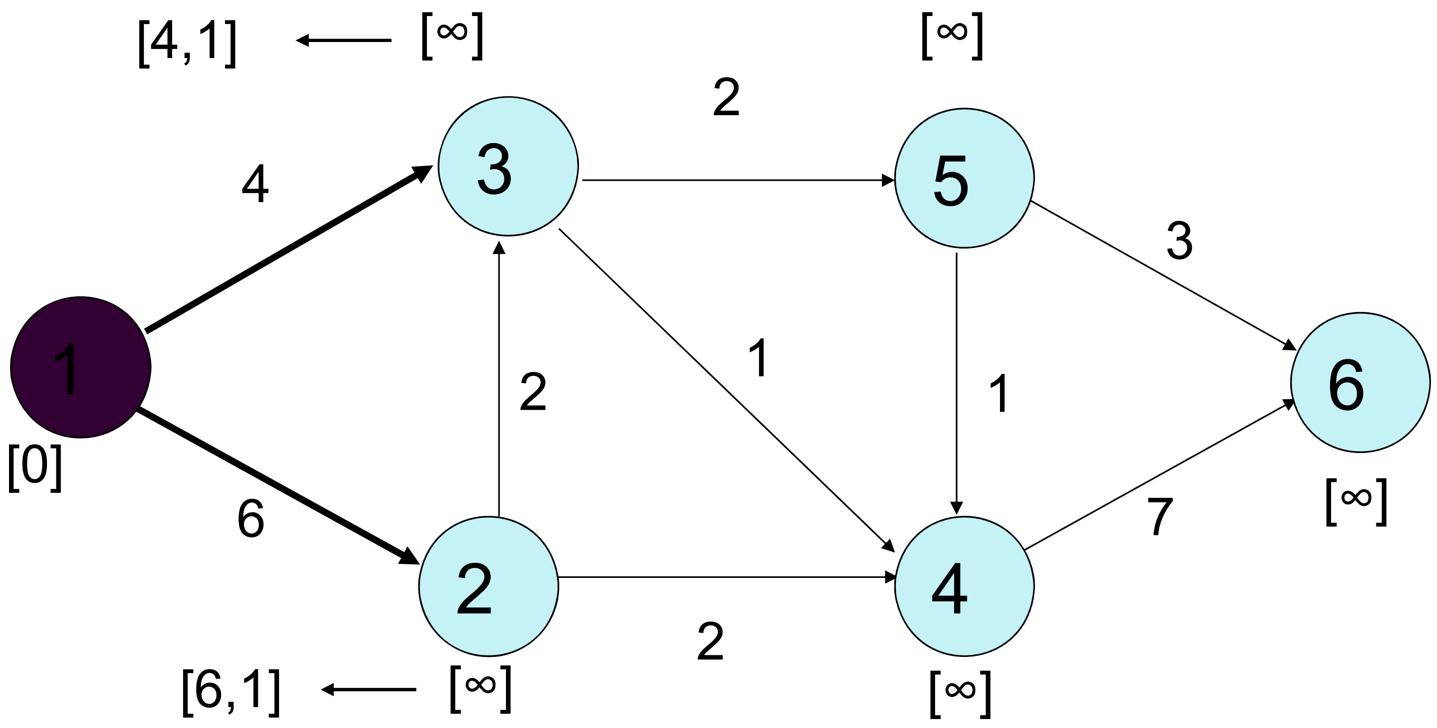


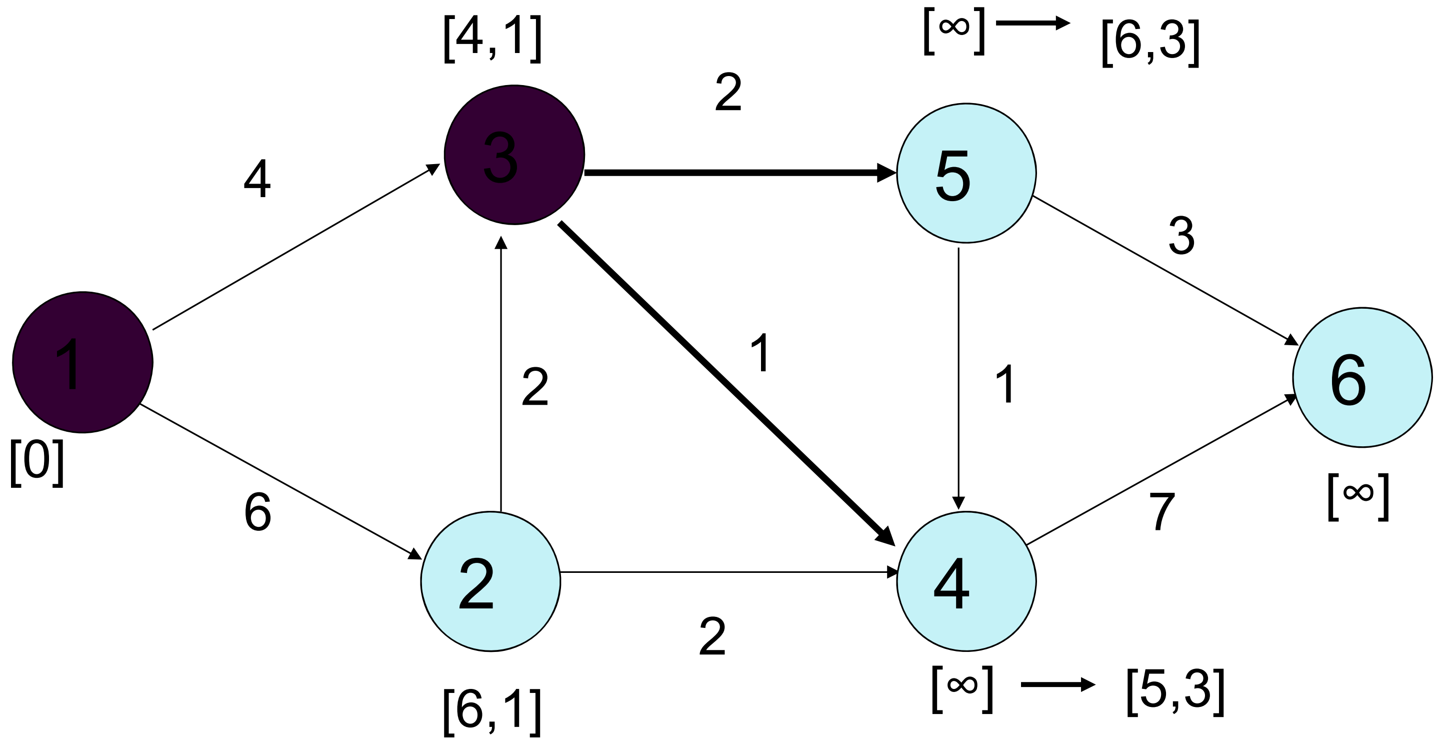
	Node i:	1	2	3	4	5	6	Next Selected Node
	Step							
$d(i)$	**1**	$\underline{0}$	∞	∞	∞	∞	∞	1
$pred(i)$		0						
	2		$\underline{6}$	4	∞	∞	∞	2
			1	1				
	3			$\underline{4}$	8	∞	∞	3
				1	2			
	4				5	$\underline{6}$	∞	5
					3	3		
	5				$\underline{5}$		9	4
					3		5	
	6						$\underline{9}$	6
							5	

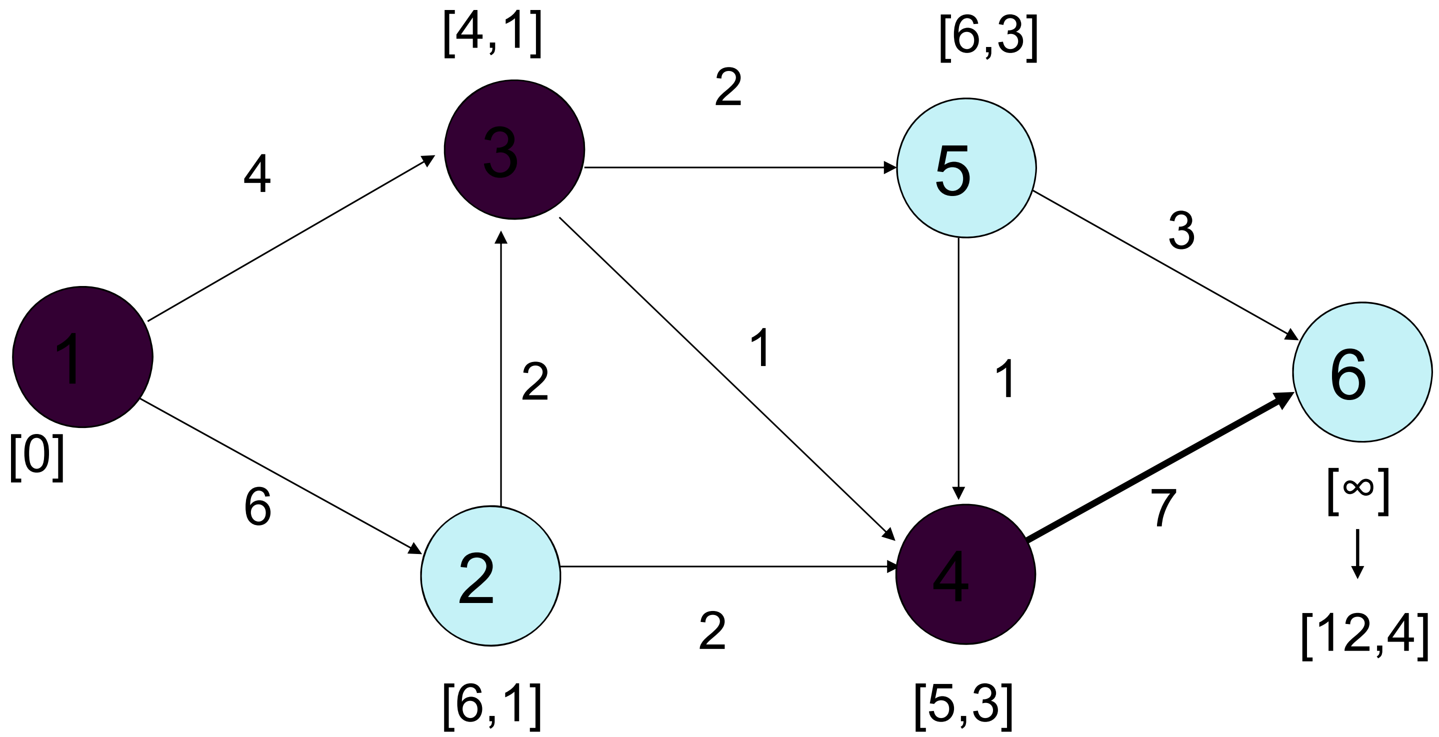
Algorithm 4 Dijkstra's Algorithm

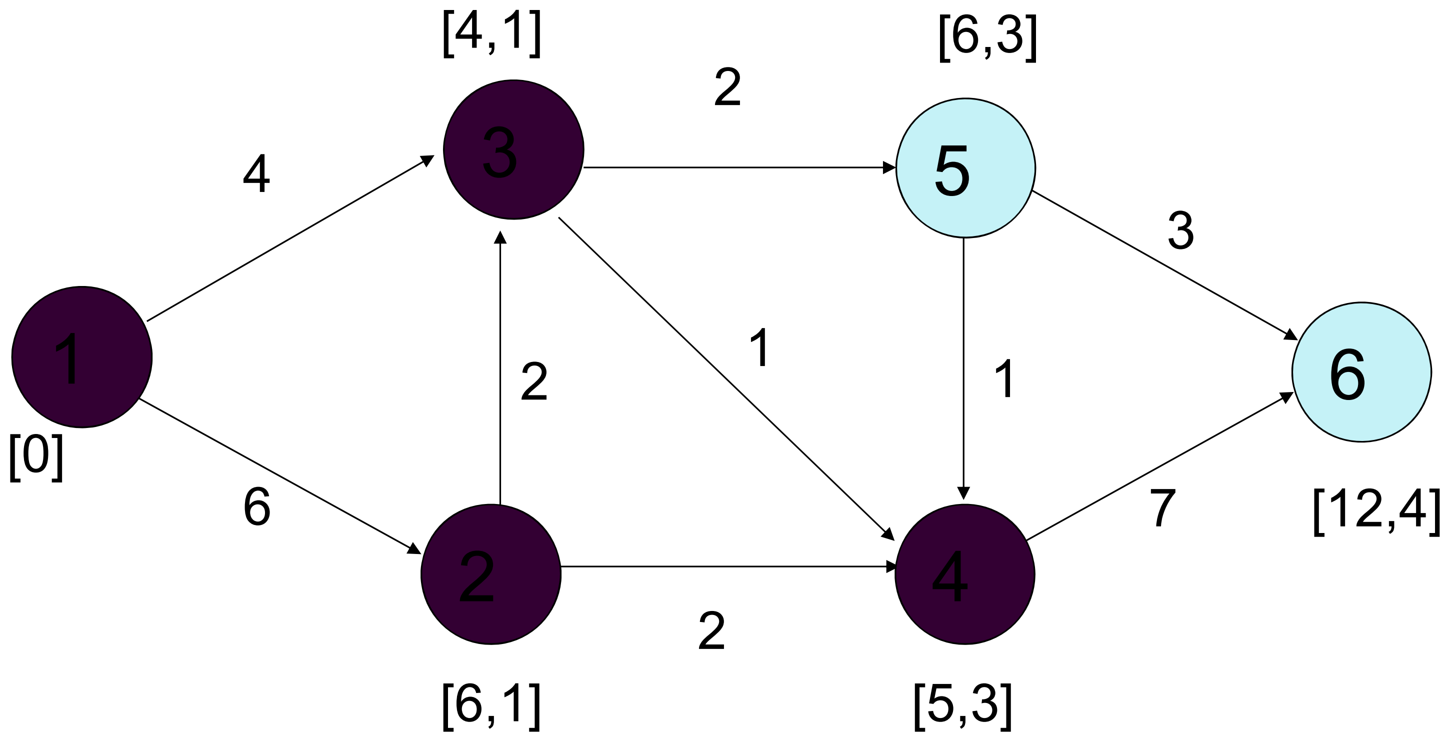
```
begin
    1) Set $PERM = \{\}$; $TEMP = \{1, 2, \ldots\}$; $d(1) = 0$; $d(i) = \infty$ for $i \in N \setminus \{1\}$.
    2) Pick node $i$ from $TEMP$ with minimum $d(i)$ and remove it from $TEMP$ and put it into
       $PERM$.
    3) Scan arcs $(i, j) \in A(i)$ such that $j \in TEMP$,
       if $d(j) > d(i) + c_{ij}$ then set $d(j) = d(i) + c_{ij}$ and $pred(j) = i$.
    4) If $TEMP = \{\}$ terminate, otherwise go to step 2.
end
```

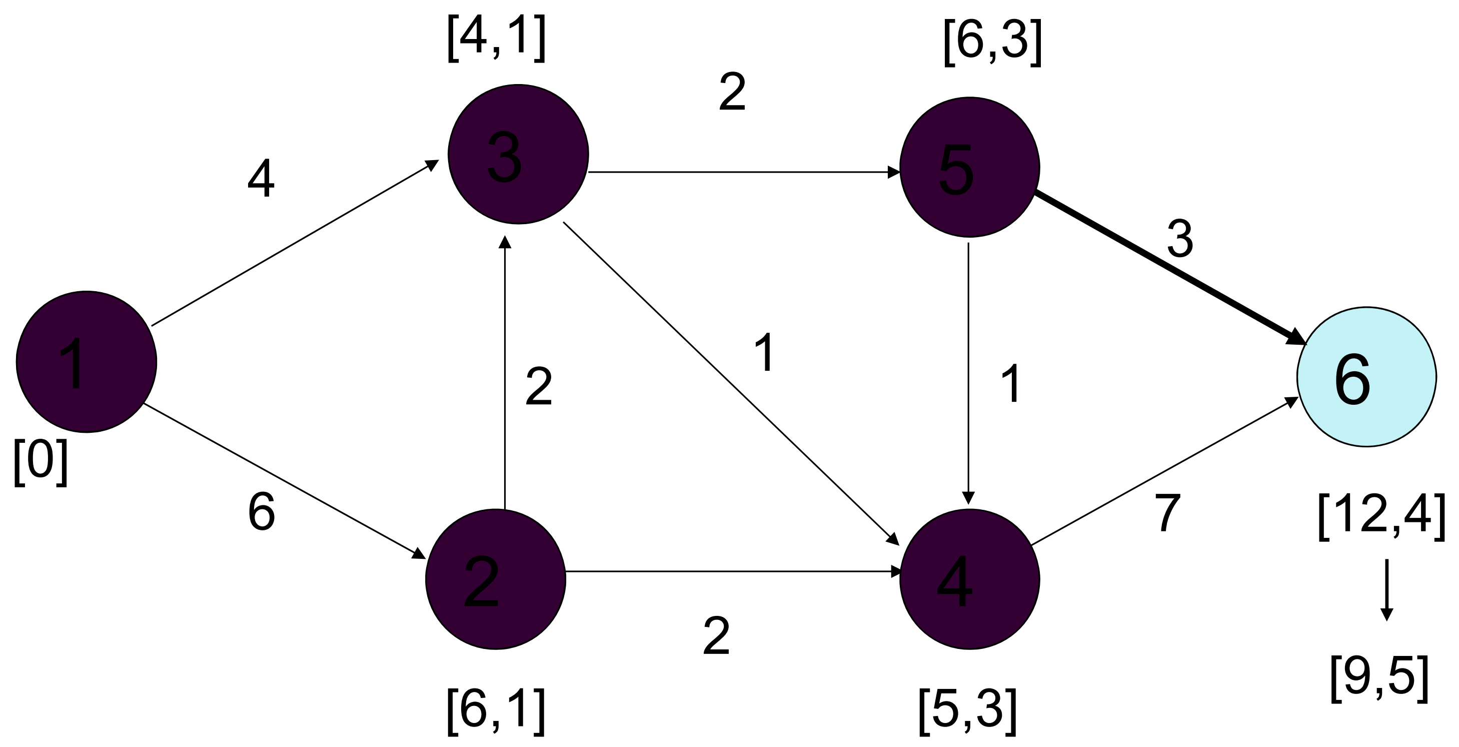

[∞]
[∞]
2
4
3
5
3
1
1
2
1
6
[0]
6
7
[∞]
2
4
2
[∞]
[∞]


[4,1]
[∞]
[∞]
2
3
5
4
3
1
1
2
1
1
[0]
2
6
6
[∞]
7
2
4
2
[6,1]
[∞]
[∞]


[4,1]
[∞] → [6,3]
2
4
3
5
3
1
1
1
2
1
6
[0]
6
7
[∞]
2
4
2
[6,1]
[∞] → [5,3]


[4,1]
[6,3]
2
4
3
1
1
2
1
[0]
6
7
[∞]
[12,4]
2
[6,1]
[5,3]
3
5
1
6
2
4


[4,1]
[6,3]
2
4
3
5
3
1
1
2
1
6
[0]
6
7
[12,4]
2
4
2
[6,1]
[5,3]


[4,1]
[6,3]
2
4
3
5
3
1
2
1
1
6
[0]
6
7
[12,4]
2
4
2
[9,5]
[6,1]
[5,3]

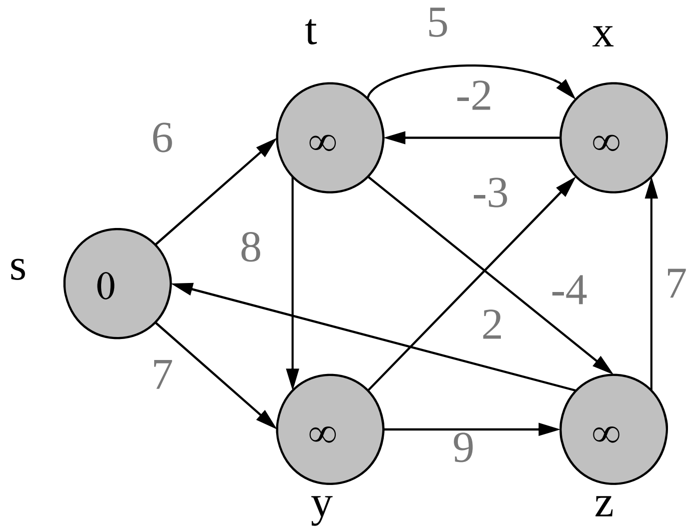
	Node i:	**1**	**2**	**3**	**4**	**5**	**6**	**Next Selected Node**
	Step							
$d(i)$	**1**	$\underline{0}$	∞	∞	∞	∞	∞	1
$pred(i)$		0						
	2		6	$\underline{4}$	∞	∞	∞	3
			1	1				
	3		6		$\underline{5}$	6	∞	4
			1		3	3		
	4		$\underline{6}$			6	12	2
			1			3	4	
	5					$\underline{6}$	12	5
						3	4	
	6						$\underline{9}$	6
							5	

Bellman-Ford Algorithm

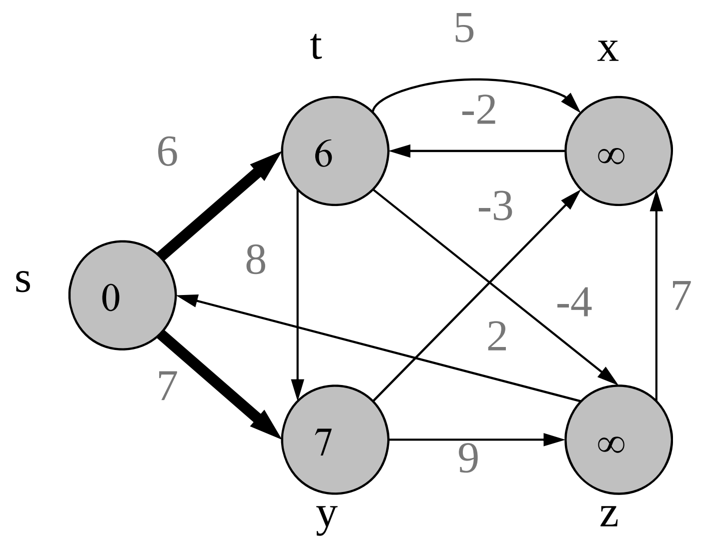
- More general than Dijkstra's algorithm:
  - Edge-weights can be negative
- Detects the existence of negative-weight cycle(s) reachable from s

```
BELMAN-FORD( G, s )
    INIT( G, s )
    for i ← 1 to |V|-1 do                                   O(VxE)
        for each edge (u, v) ∈ E do
            RELAX( u, v )
    for each edge ( u, v ) ∈ E do
        if d[v] > d[u]+w(u,v) then
            return FALSE      > neg-weight cycle
    return TRUE
```

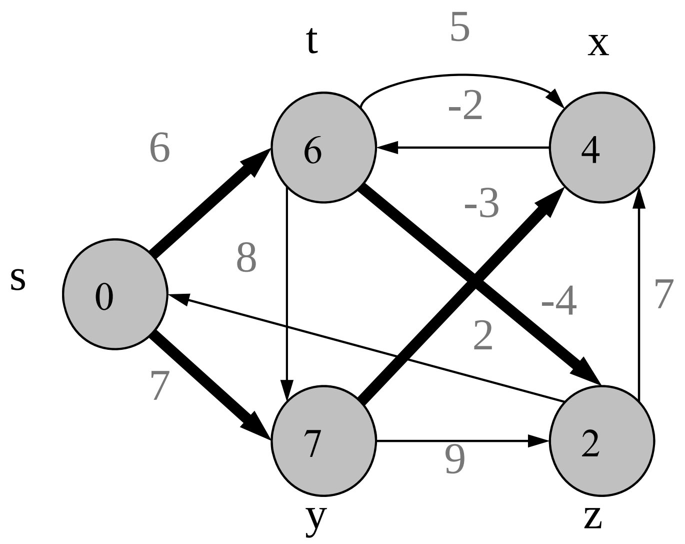
Bellman-Ford Algorithm Example



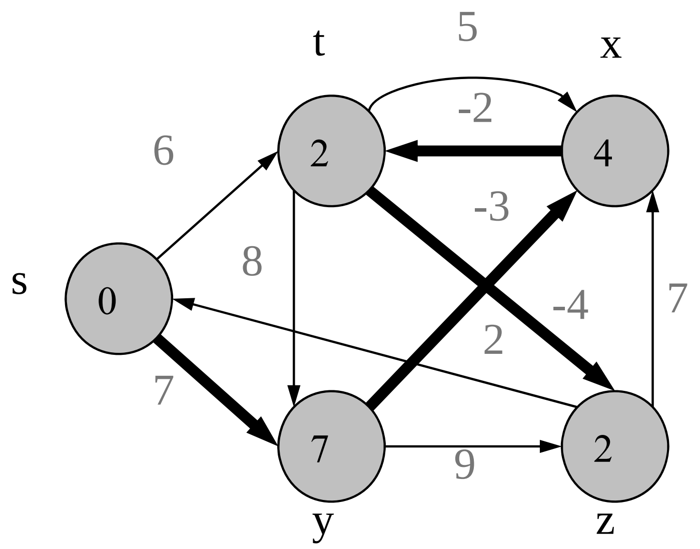
Bellman-Ford Algorithm Example



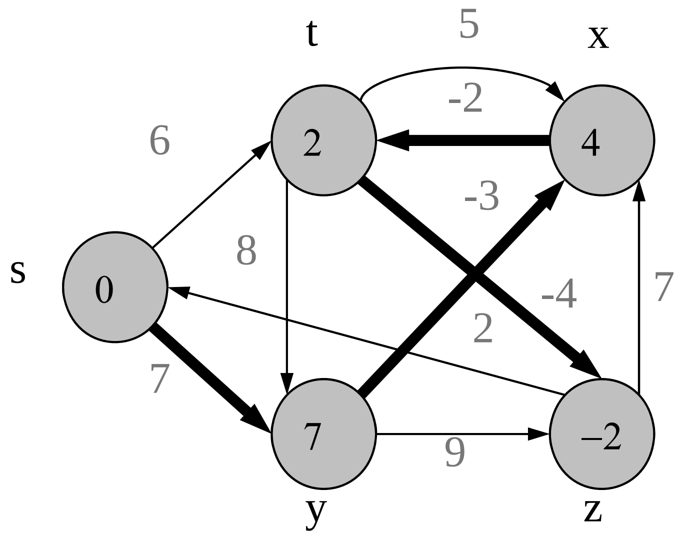
Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Traveling Salesman Problem

- Given a number of cities and the costs of traveling from one to the other, what is the cheapest roundtrip route that visits each city once and then returns to the starting city?
- An equivalent formulation in terms of graph theory is: Find the Hamiltonian cycle with the least weight in a weighted graph.
- It can be shown that the requirement of returning to the starting city does not change the computational complexity of the problem.
- A related problem is the (bottleneck TSP): Find the Hamiltonian cycle in a weighted graph with the minimal length of the longest edge.

TSP

Travelling Salesman Problem

Traveling Salesman Problem (TSP) - Optimization Problem:

Input: A graph $G = (V, E)$ and weights c_{ij} for each edge $(i, j) \in E$.

Output: A *tour*, a cycle that visits all the nodes exactly once

(also called a *Hamiltonian cycle*), of minimum weight.

Consider the Travelling Salesman Problem on a complete graph G = (V,E)

Hamiltonian Cycle Problem

Given an undirected graph $G = (V, E)$, decide whether it has a *Hamiltonian cycle*.

The Hamiltonian cycle problem is NP-complete

a-approximation to the TSP

Given a graph $G = (V, E)$, form an input to the TSP by setting, for each pair i, j, the cost c_{ij} equal to 1 if $(i, j) \in E$, and equal to $n + 2$ otherwise.

If there is a Hamiltonian cycle in G, then there is a tour of cost n, and otherwise each tour costs at least $2n + 1$.

If there were to exist a 2-approximation algorithm for the TSP, then we could use this algorithm to distinguish graphs with Hamiltonian cycles from those without any: run the approximation algorithm on the new TSP input, and if the tour computed has cost at most $2n$, then there exists a Hamiltonian cycle in G, and otherwise there does not.

Theorem 2.9: *For any $\alpha > 1$, there does not exist an α-approximation algorithm for the traveling salesman problem on n cities, provided $\mathrm{P} \neq \mathrm{NP}$. In fact, the existence of an $O(2^n)$-approximation algorithm for the TSP would similarly imply that $\mathrm{P} = \mathrm{NP}$.*

a-approximation to the TSP

We assume that the triangle inequality holds: $c_{ik} \leq c_{ij} + c_{jk}$.

Nearest Addition Algorithm for the Metric TSP

First iteration: find the two closest cities, say i and j, and build a tour that consists of going from i to j and then back to i again. Let $S = \{i, j\}$.

In each subsequent iteration, we find a pair of cities $i \in S$ and $j \notin S$ for which the cost c_{ij} is minimum; let k be the city that follows i in the current tour on S. We add j to S, and insert j into the current tour between i and k.

Nearest Addition Algorithm for the Metric

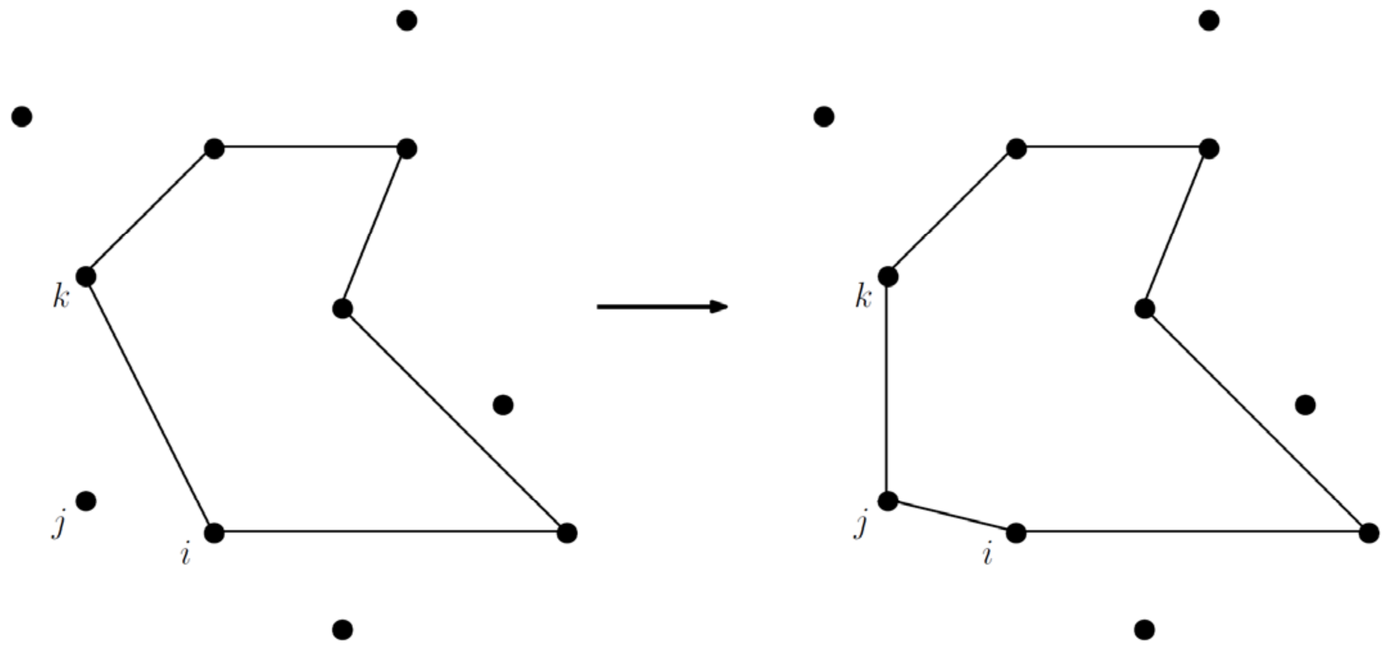


Figure 2.4: Illustration of a greedy step of the nearest addition algorithm.

Christofides Algorithm for the Metric TSP

Find a minimum spanning tree T.

Identify the set O of odd-degree nodes with even cardinality.

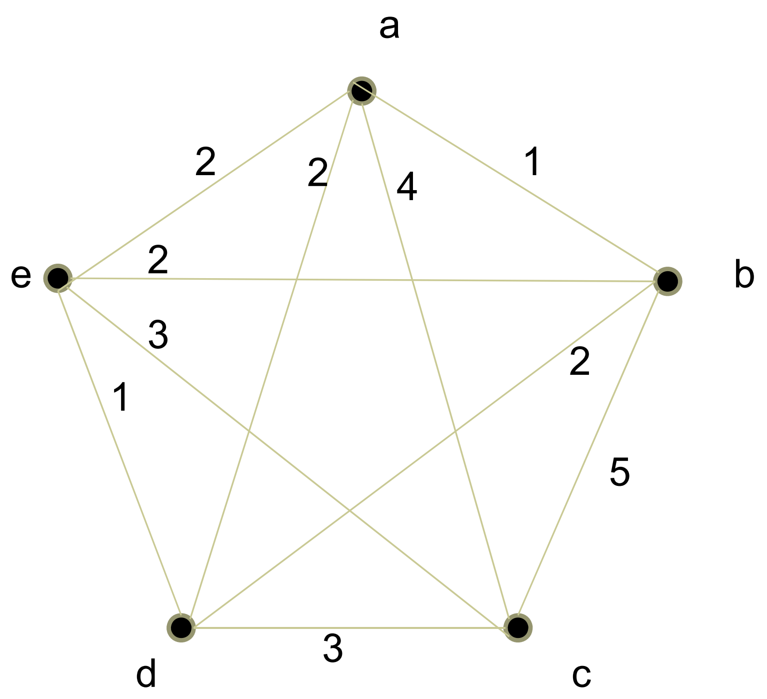
Compute a minimum-cost perfect matching M on O.

Add M to T to construct a Eulerian graph on the original set of cities.

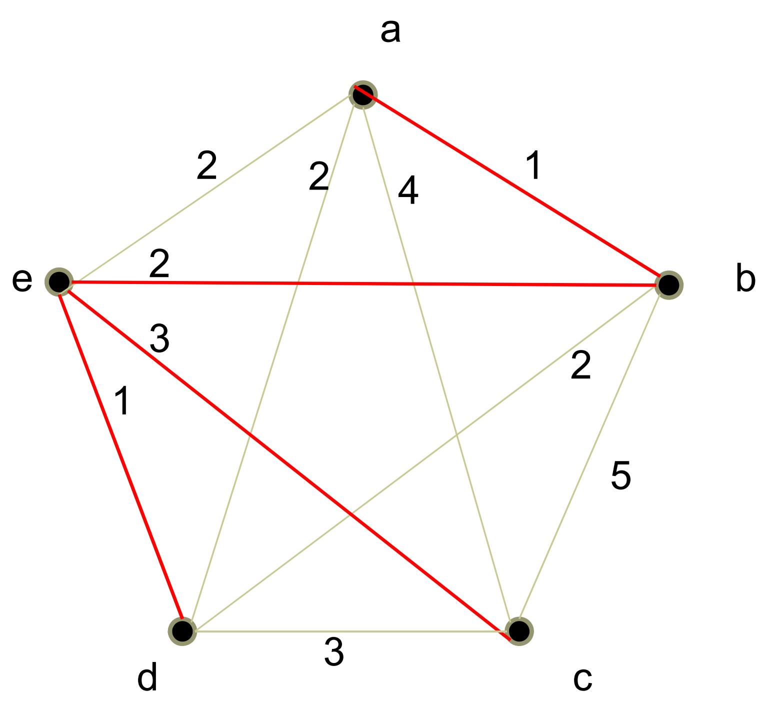
As in the double-tree algorithm, shortcut this graph to produce a tour.

Theorem 2.13: *Christofides' algorithm for the metric traveling salesman problem is a 3/2-approximation algorithm.*

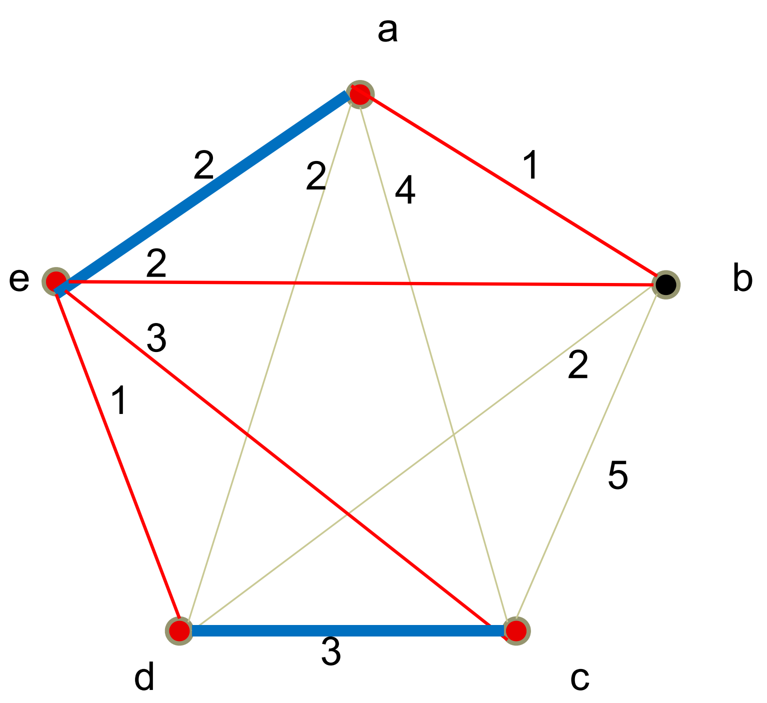
Christofides Algorithm for the Metric TSP



Christofides Algorithm for the Metric TSP

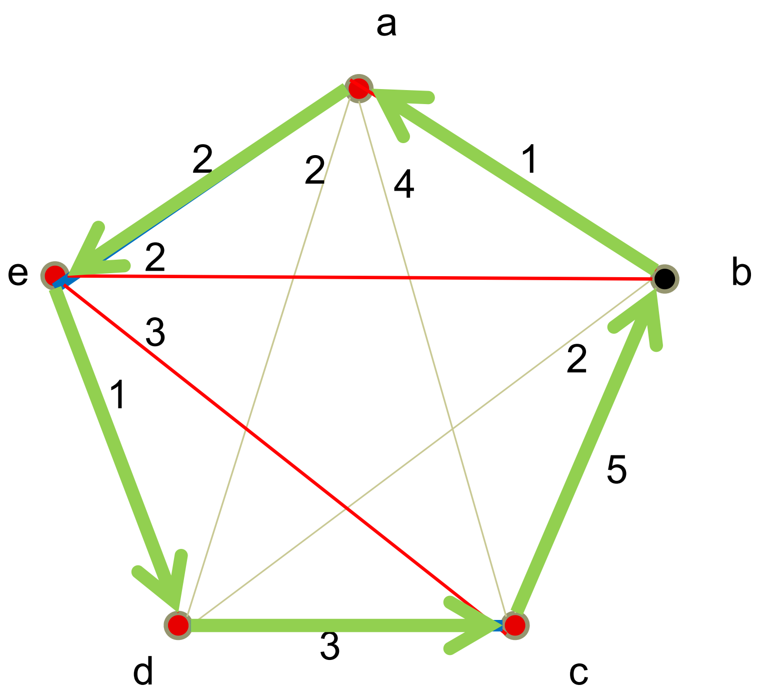


Christofides Algorithm for the Metric TSP



a,e,d,c,e,b,a

Christofides Algorithm for the Metric TSP



a,e,d,c,e,b,a

Approximation Algorithms for the Metric TSP

Remarkably, no better approximation algorithm for the metric traveling salesman problem is known. However, substantially better algorithms might yet be found, since the strongest negative result is as follows.

Theorem 2.14: *Unless* P = NP, *for any constant* $\alpha < \frac{220}{219} \approx 1.0045$, *no* α*-approximation algorithm for the metric TSP exists.*

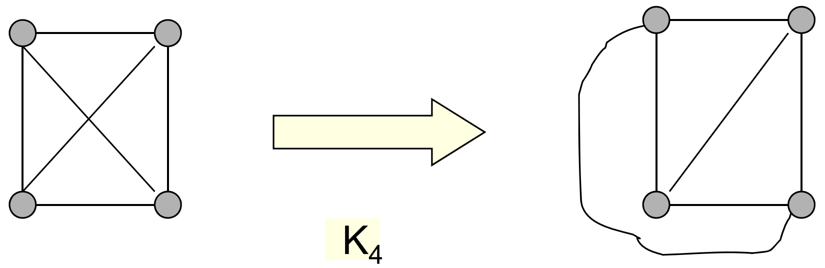
It is possible to obtain a polynomial-time approximation scheme in the case that cities correspond to points in the Euclidean plane and the cost of traveling between two cities is equal to the Euclidean distance between the corresponding two points.

Planar Graphs

- A graph (or multigraph) *G* is called *planar* if *G* can be drawn in the plane with its edges intersecting only at vertices of *G*, such a drawing of *G* is called an *embedding* of *G* in the plane.

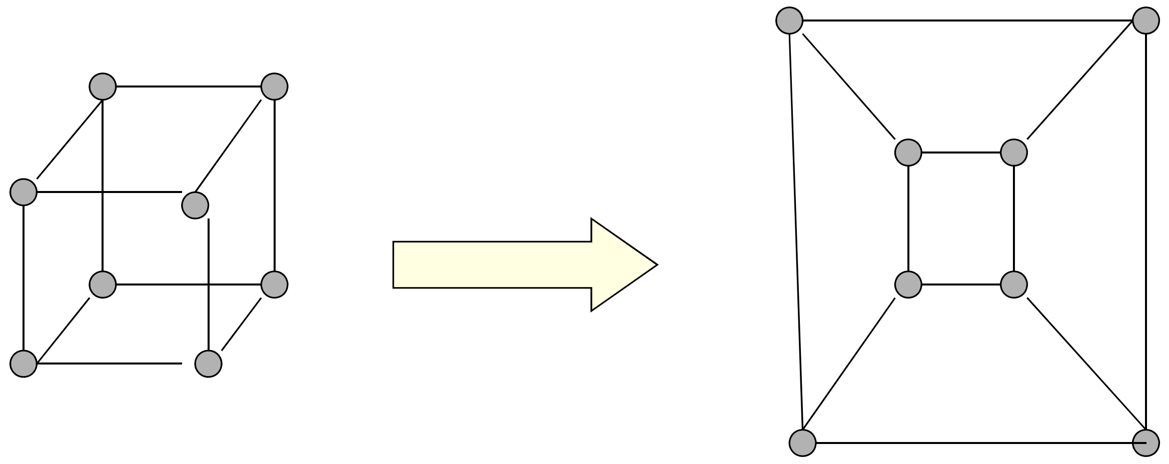
  Application Example: VLSI design (overlapping edges requires extra layers), Circuit design (cannot overlap wires on board)

  Representation examples: *K*1,*K*2,*K*3,*K*4 are planar, *Kn* for *n*>4 are non-planar



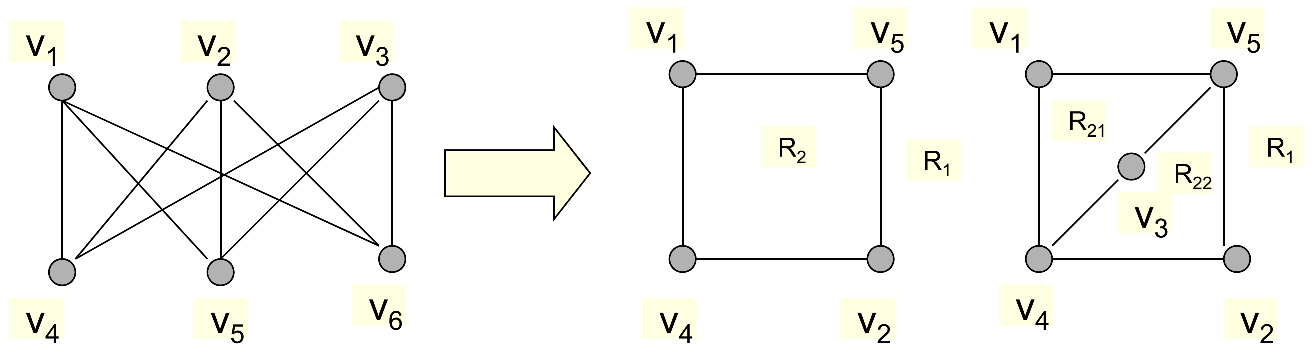
Planar Graphs

- Representation examples: Q_3



Planar graphs

- Representation examples: $K_{3,3}$ is Nonplanar



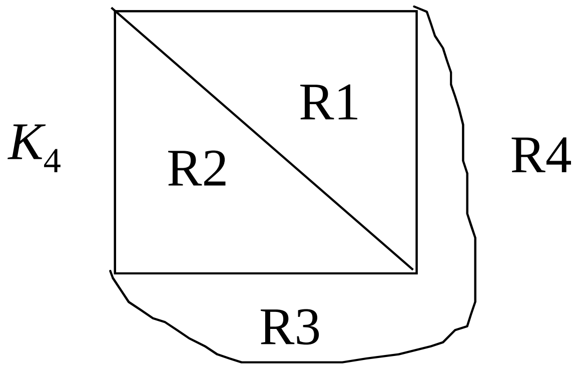
Planar graphs

- Theorem : *Euler's planar graph theorem*
- For a connected planar graph or multigraph:

$$\underbrace{v}_{\text{number of vertices}} - \underbrace{e}_{\text{number of edges}} + \underbrace{r}_{\text{number of regions}} = 2$$

Planar graphs

Example of Euler’s theorem



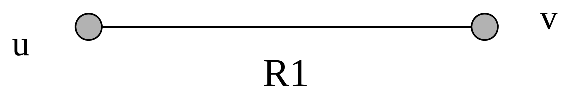
A planar graph divides the plane into several regions (faces), one of them is the infinite region.

v=4,e=6,r=4, v-e+r=2

Planar graphs

- Proof of Euler's formula: By Induction

<u>Base Case:</u> for G1 , $e_1 = 1$, $v_1 = 2$ and $r_1 = 1$



<u>n+1 Case:</u> Assume, $r_n = e_n - v_n + 2$ is true. Let {an+1, bn+1} be the edge that is added to Gn to obtain Gn+1 and we prove that $r_n = e_n - v_n + 2$ is true. Can be proved using two cases.

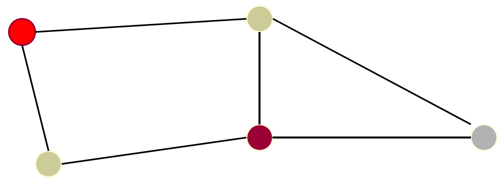
Graph Coloring Problem

- Graph coloring is an assignment of *"colors"*, almost always taken to be consecutive integers starting from 1 without loss of generality, to certain objects in a graph. Such objects can be vertices, edges, faces, or a mixture of the above.

- Application examples: scheduling, register allocation in a microprocessor, frequency assignment in mobile radios, and pattern matching

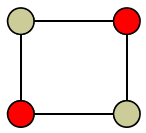
Vertex coloring problem

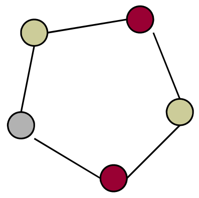
- Assignment of colors to the vertices of the graph such that proper coloring takes place (no two adjacent vertices are assigned the same color)
- Chromatic number: least number of colors needed to color the graph
- A graph that can be assigned a (proper) k-coloring is k-colorable, and it is k-chromatic if its chromatic number is exactly k

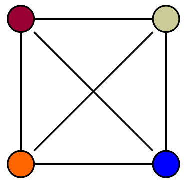


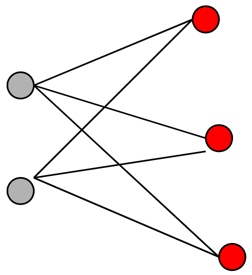
Vertex coloring problem

- The problem of finding a minimum coloring of a graph is NP-Hard
- The corresponding decision problem (Is there a coloring which uses at most *k* colors?) is NP-complete
- The chromatic number for C_n = 3 (n is odd) or 2 (n is even), K_n = n, $K_{m,n}$ = 2
- **Cn: cycle with n vertices; Kn: fully connected graph with n vertices; Km,n: complete bipartite graph**


C_4

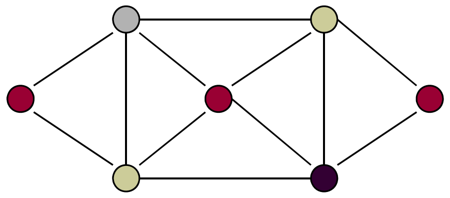

C_5

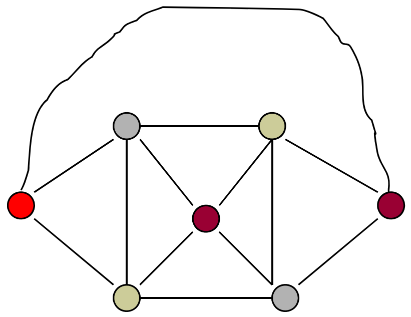

K_4


$K_{2,\,3}$

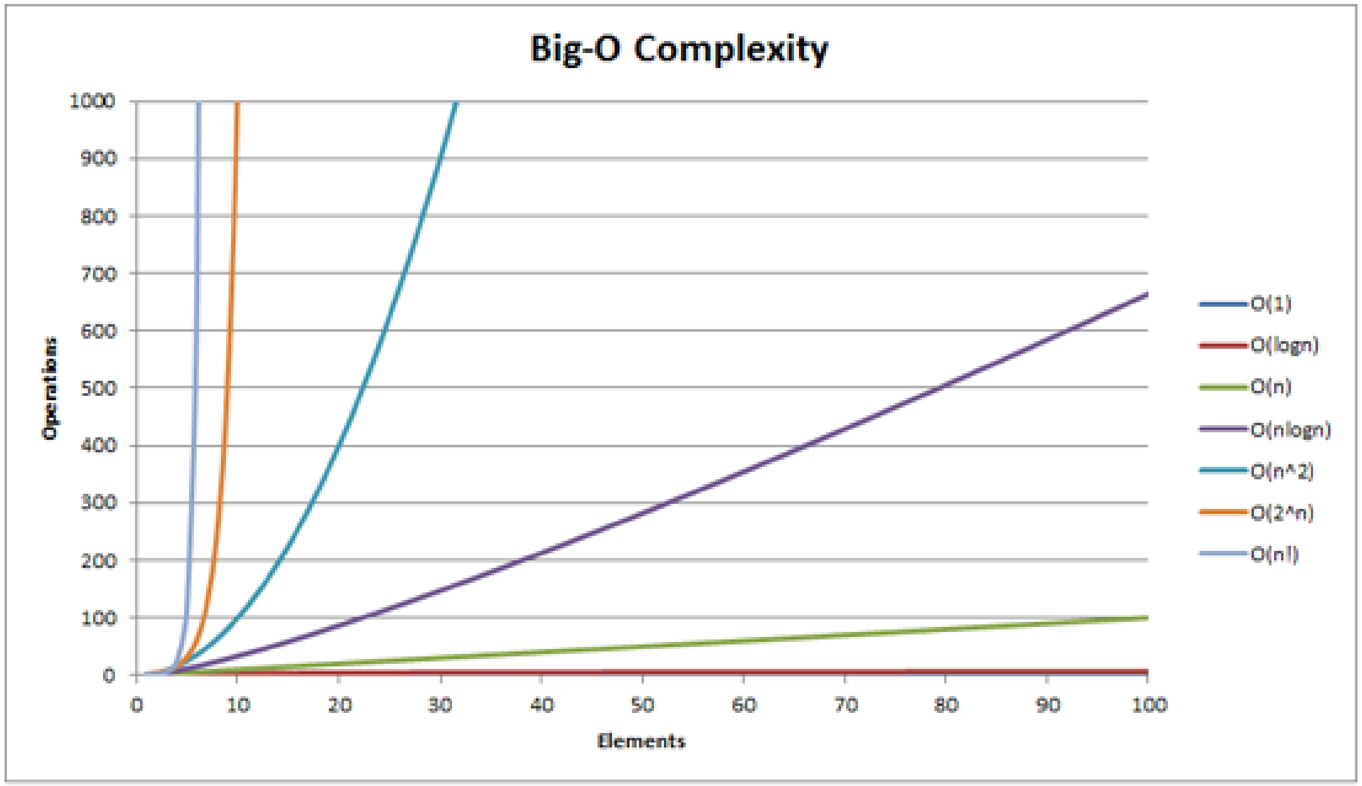
Vertex covering problem

- The Four color theorem: the chromatic number of a planar graph is no greater than 4
- Example: G1 chromatic number = 3, G2 chromatic number = 4
- (Most proofs rely on case by case analysis).


G1


G2

How bad is exponential complexity



The P class

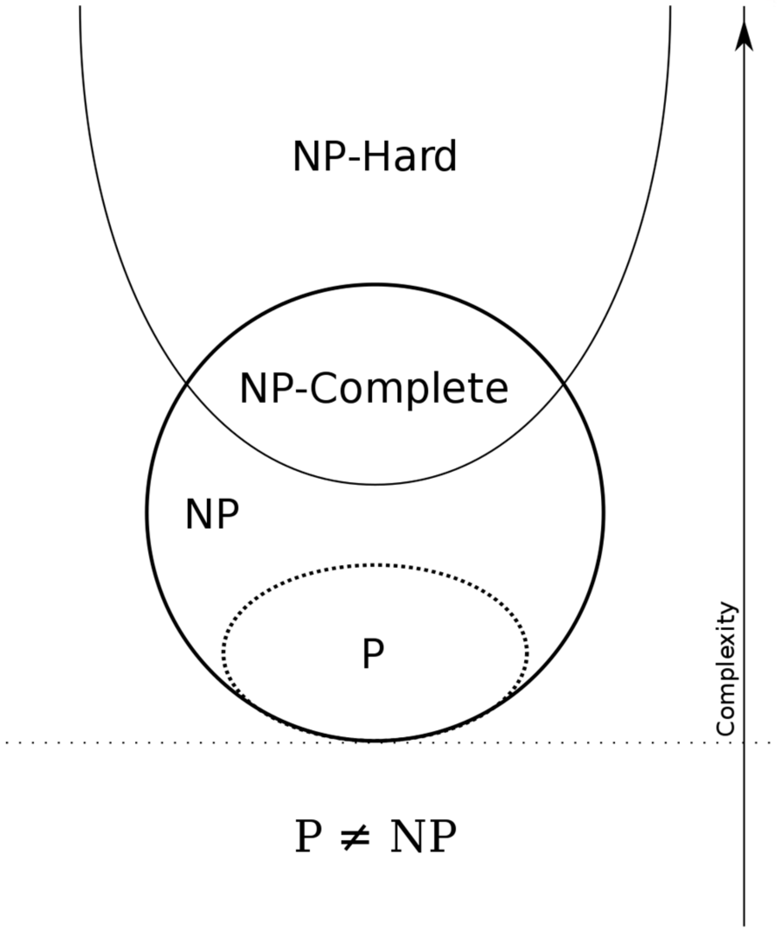
- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem
- The key is that n is the size of input

The NP class

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- **NP means verifiable in polynomial time**
- **Verifiable?**
  - If we are somehow given a ‘certificate’ of a solution we can verify the legitimacy in polynomial time

P / NP graph theory problems

- Shortest path algorithms **solvable in pseudo-polynomial time**
- Longest path **is NP complete**
- Eulerian tours **solvable in polynomial time**
- Hamiltonian tours **is NP complete**
- Vertex covering problem **is NP-hard**


NP-Hard
NP-Complete
NP
P
Complexity
P ≠ NP