Analysis and Design of Algorithms

Graphs

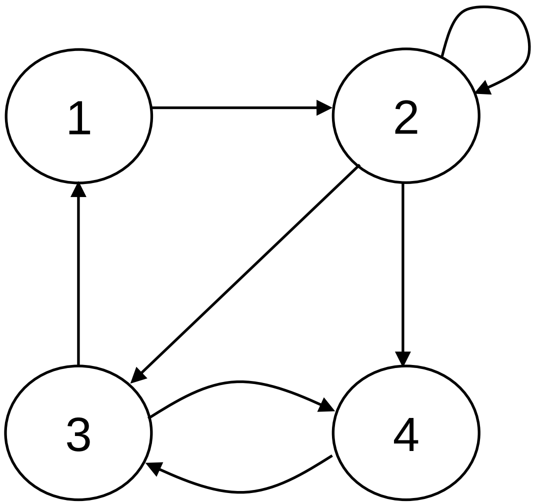
Part 1: Definition and Representations

Instructor: **Morteza Zakeri**

What is a Graphs?

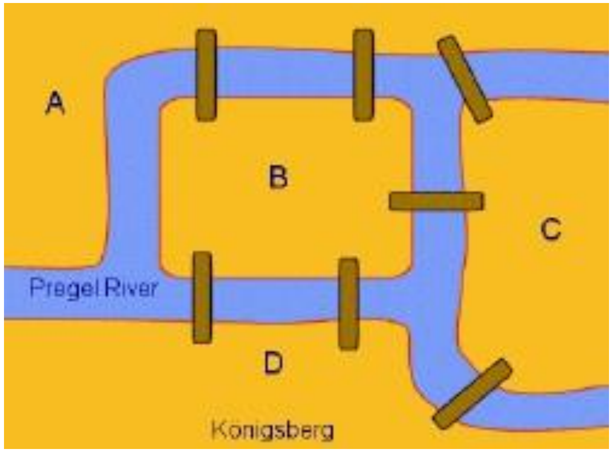
Definition = a set of nodes (vertices) with edges (links) between them.

- $G = (V, E)$ - graph
- V = set of vertices $|V| = n$
- E = set of edges $|E| = m$
  – Binary relation on V
  – Subset of V x V $=\{(u,v): u \in V, v \in V\}$



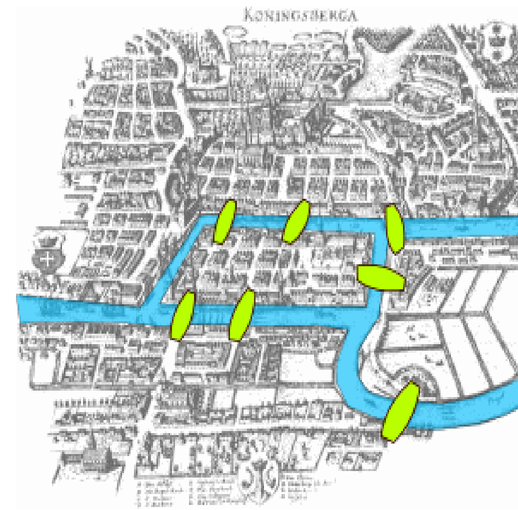
Graphs

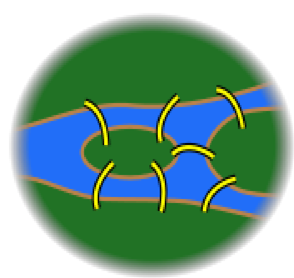
- Graph theory starts in 18th century with **Leonhard Euler**
  - Euler wrote a paper about the Königsberg bridges problem
  - Since then graphs have been studied extensively.

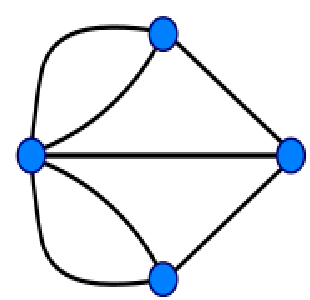


Königsberg Bridges Problem

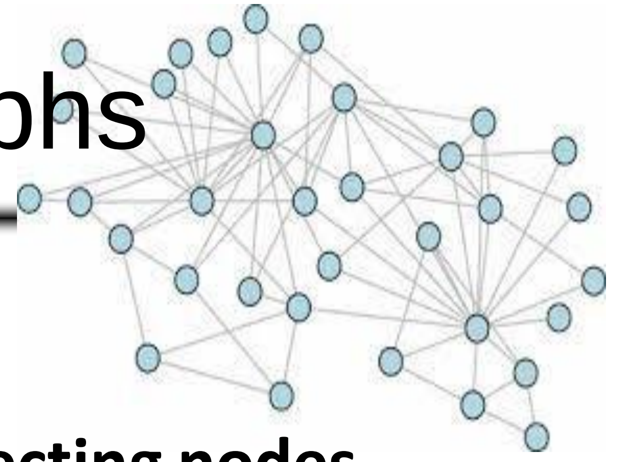
- Königsberg (now Kaliningrad, Russia)
  - A river divided the city into 4 regions
  - Seven bridges connected the regions
- Problem: is there a path in which:
  - Cross each bridge once (and only once)
  - the trip ends in the same place it began
- Euler proved: It is not possible
  - Such a cycle exists if: 1-connected graph 2-all node-degrees even







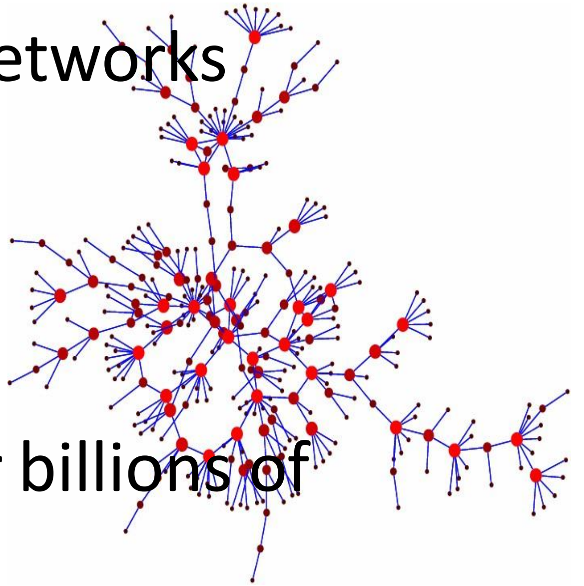
Networks: Real-world graphs



- A collection of **nodes (vertices)**
- And a collection of **edges (links) connecting nodes**
- A network model treats all nodes and links the same
  - But there are heterogeneous networks. Name a few?
- The <u>spatial location</u> of nodes is arbitrary (in visualization)
- Networks are abstractions of connection and relation
- A vast array of phenomena have been modeled by networks
- In <u>mathematics</u>, networks are called **graphs**
- The entities are nodes, and the links are edges

Networks in the Past and Now

- Graphs have been used in the past to model existing networks
  - e.g., networks of highways, social networks
  - Usually, these networks were small
- Networks Now:
  - More and larger networks appear.
  - Networks of thousands, millions, or billions of nodes
  - impossible to visualize



Why Networks? Why Now?

- Products of **technological advancement**
  - e.g., Internet, Web
- Result of our **ability to collect** more, better, and more complex data
  - e.g., gene regulatory networks
- **Data availability**
  - Rise of the Web 2.0 and Social media

Why Networks? Why Now? (cont'd)

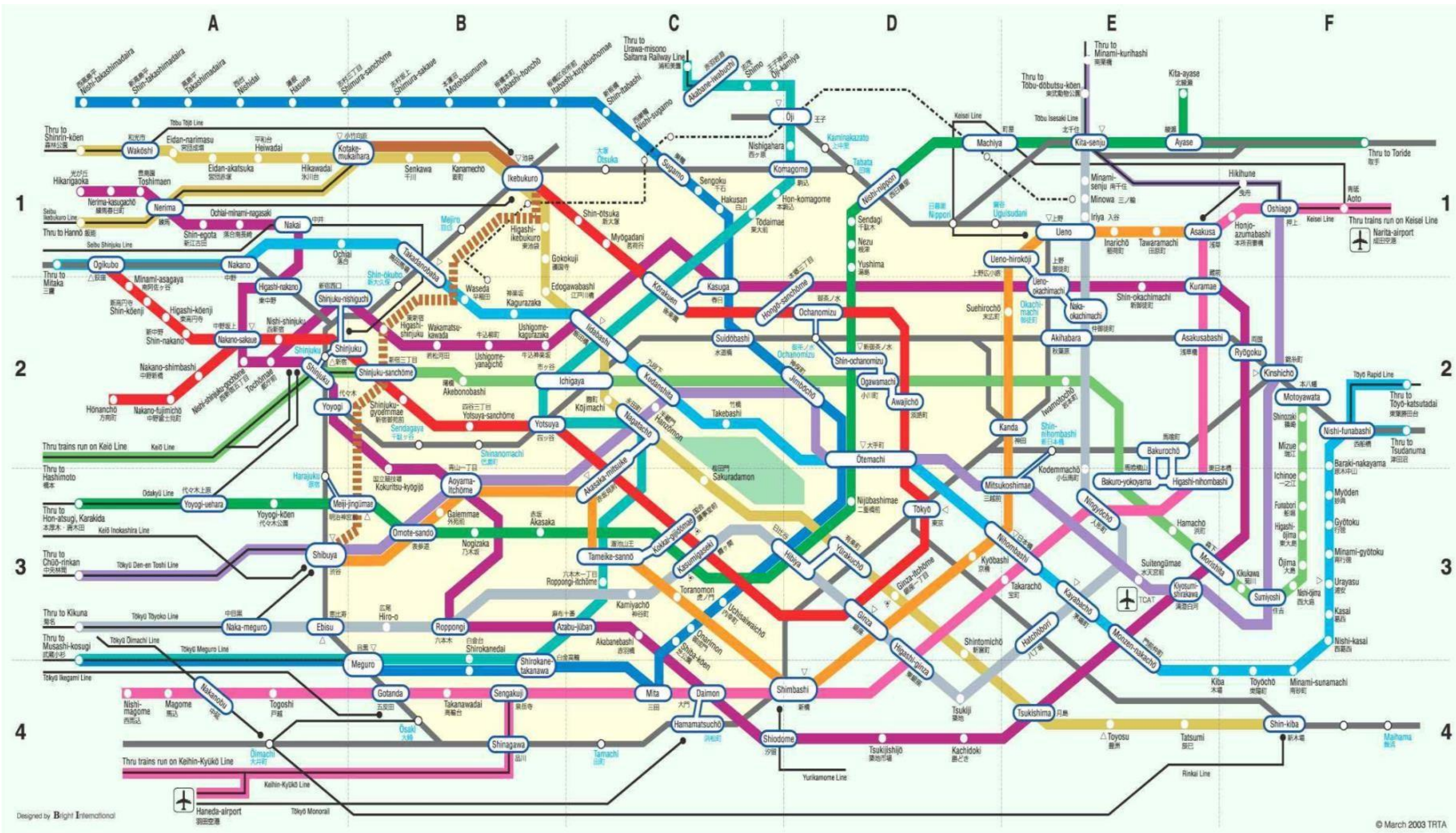
- **Universality**
  - Networks from various domains of science, nature, and technology are more similar than one would expect
- **Shared vocabulary between fields**
  - Computer Science, Social Sciences, Physics, Economics, Statistics, Biology, Political Science…
- Advances in **computational power**
  - Better hardware, cloud, clusters, distributed computing, …

Many examples of networks

- Technological Networks
- Social Networks
- Networks of Information
- Biological Networks
- And …

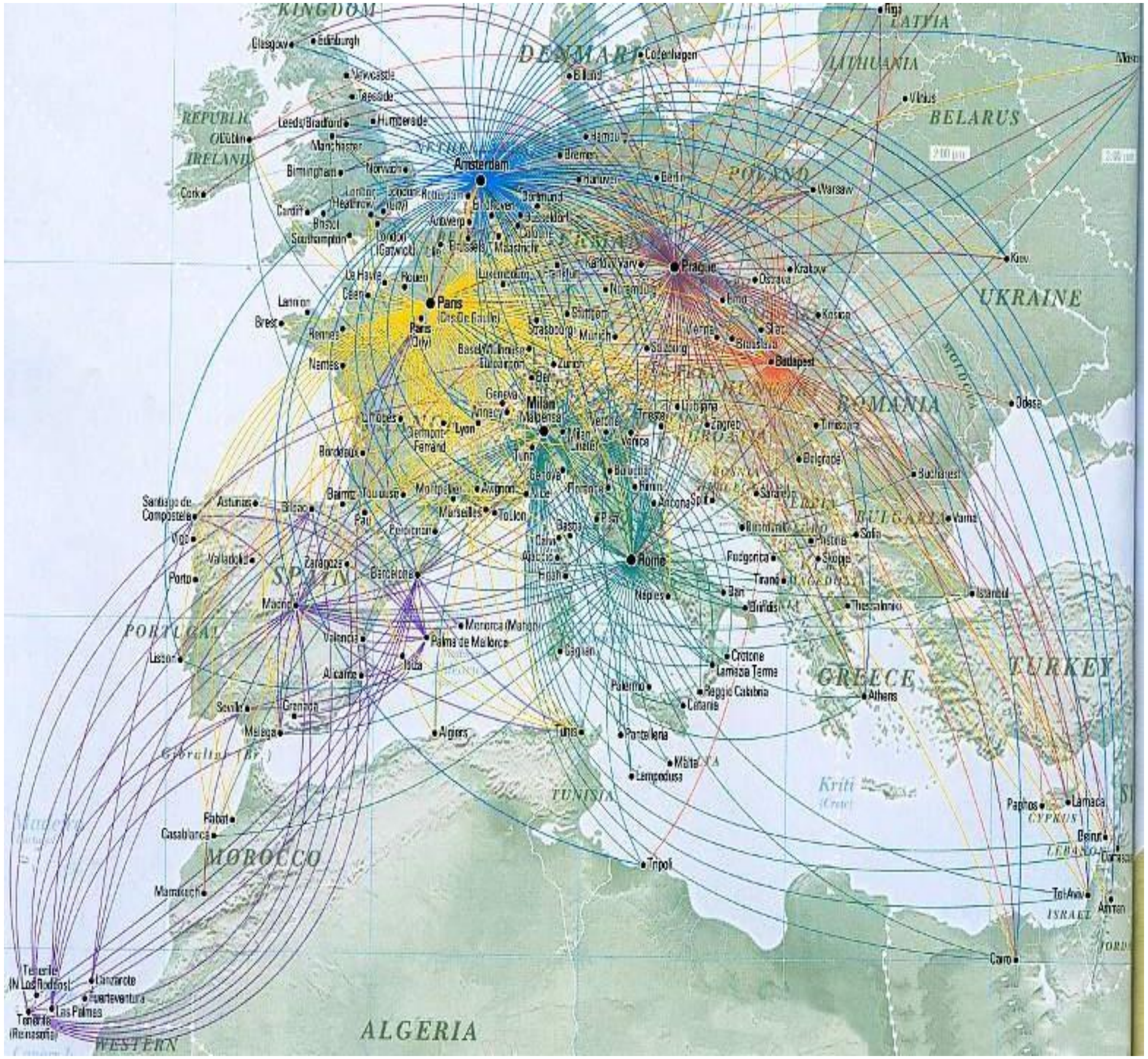
TECHNOLOGICAL NETWORKS

Railway Networks

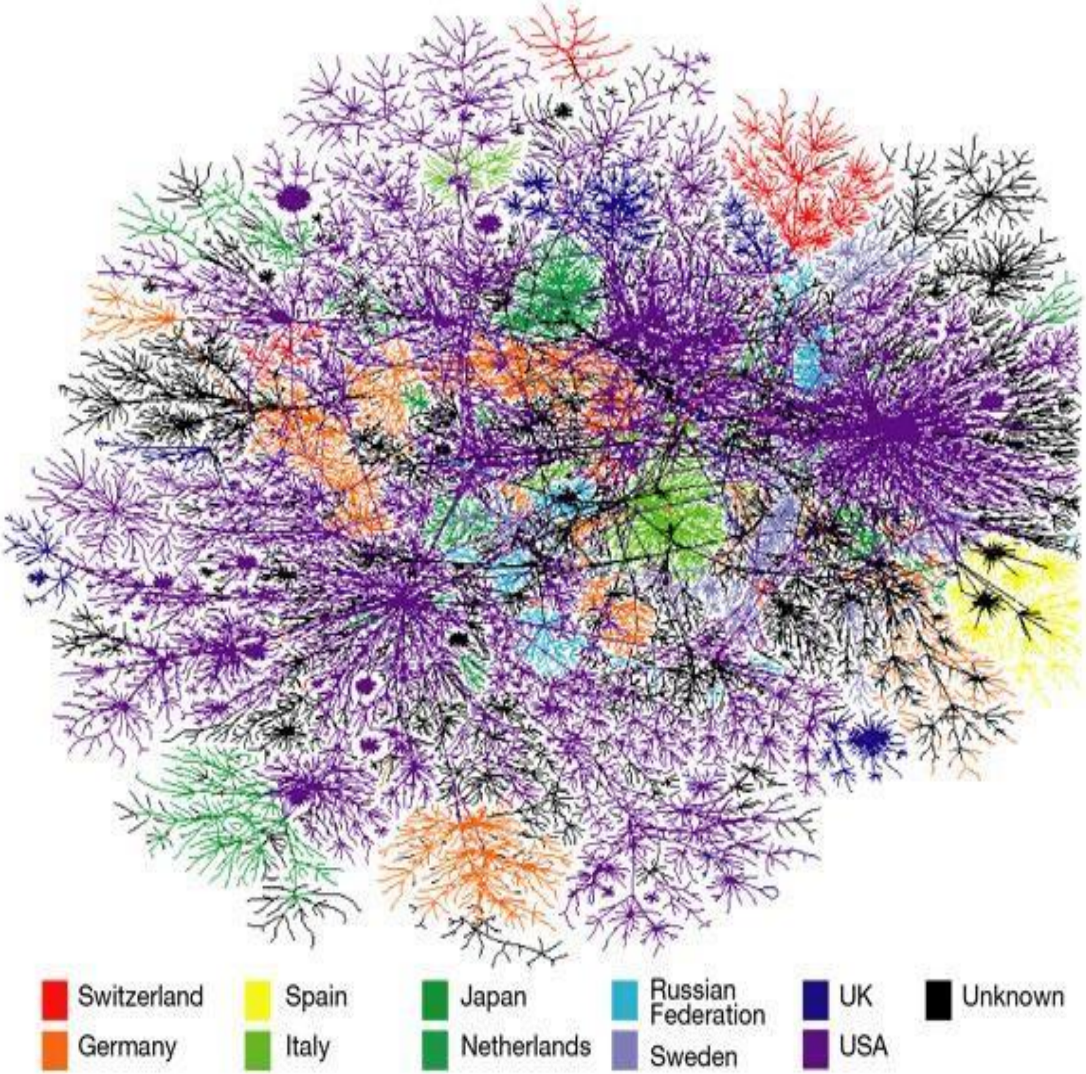


Source: TRTA, March 2003 - Tokyo rail map

The Airline Networks



The Internet Map

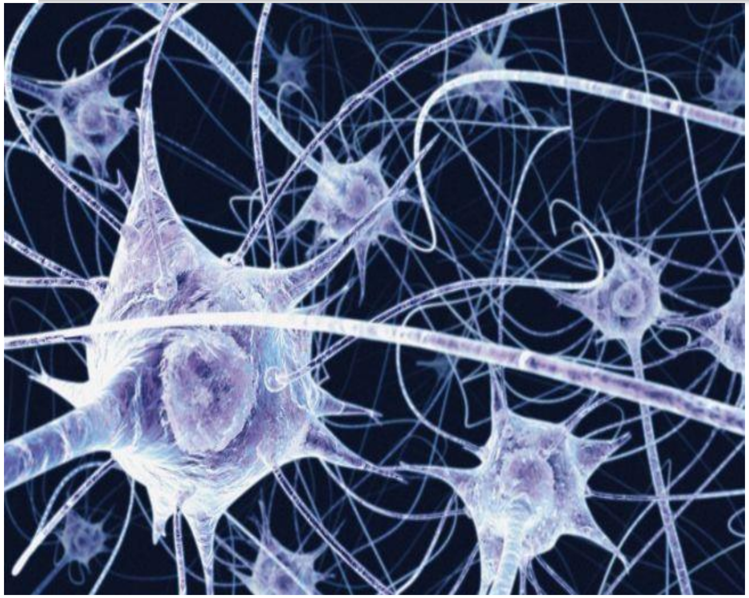


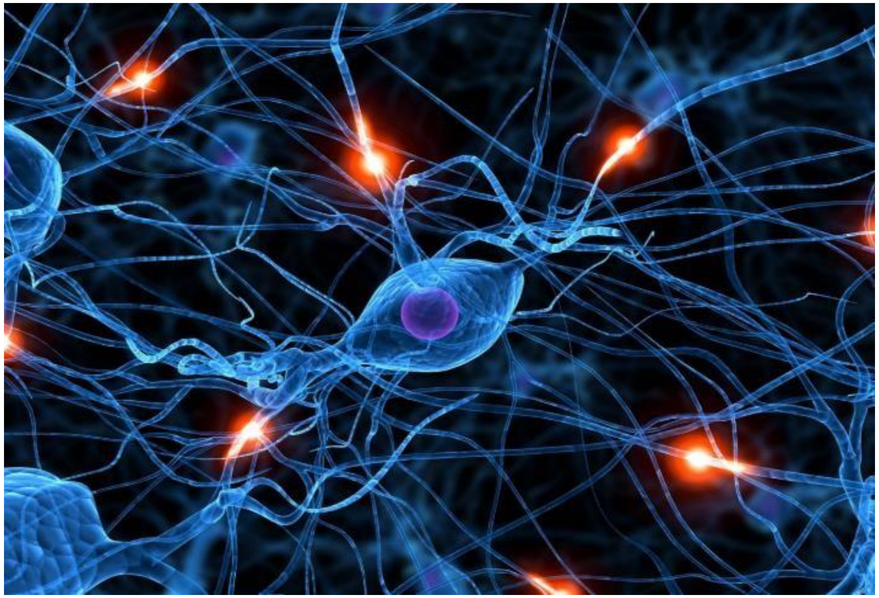
Other Technological Networks?

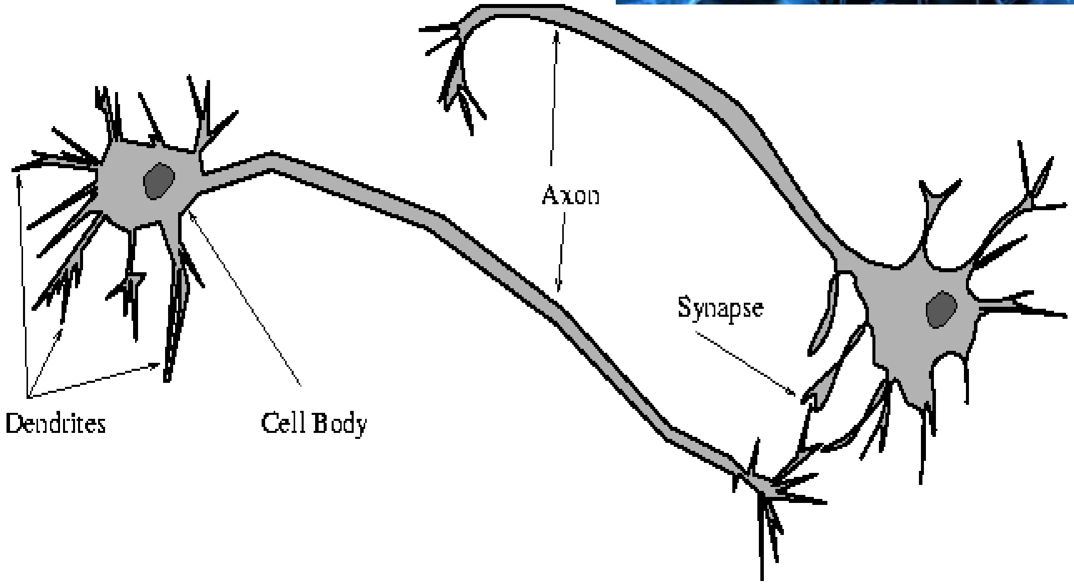
- Internet
- Telecommunication Networks, *e.g.*, telephone network
- Power Grid
  - The network of high-voltage transmission lines
  - that provide long-distance transport of electric power
- Transportation networks
  - Airlines, Railway, …
- Delivery and distribution networks
  - Gas, oil, water, Post, …

BIOLOGICAL NETWORKS

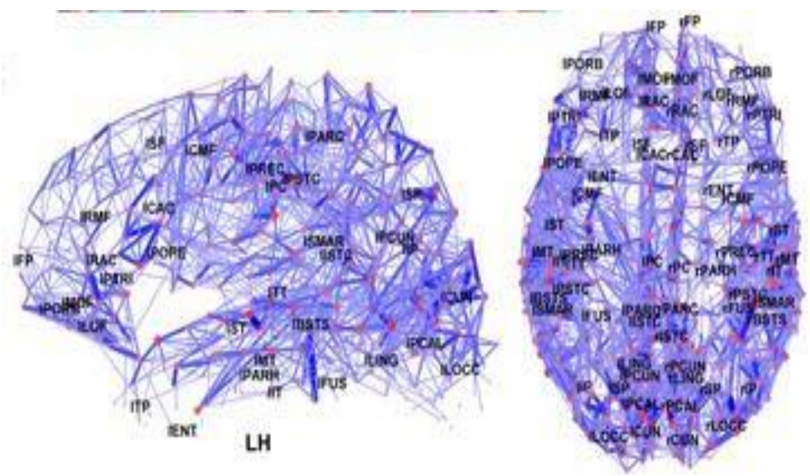
Neural Networks

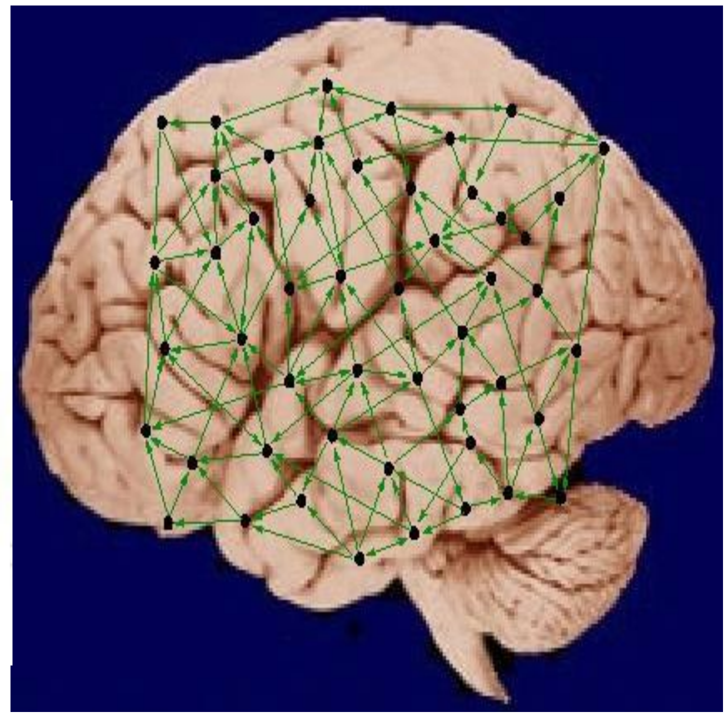




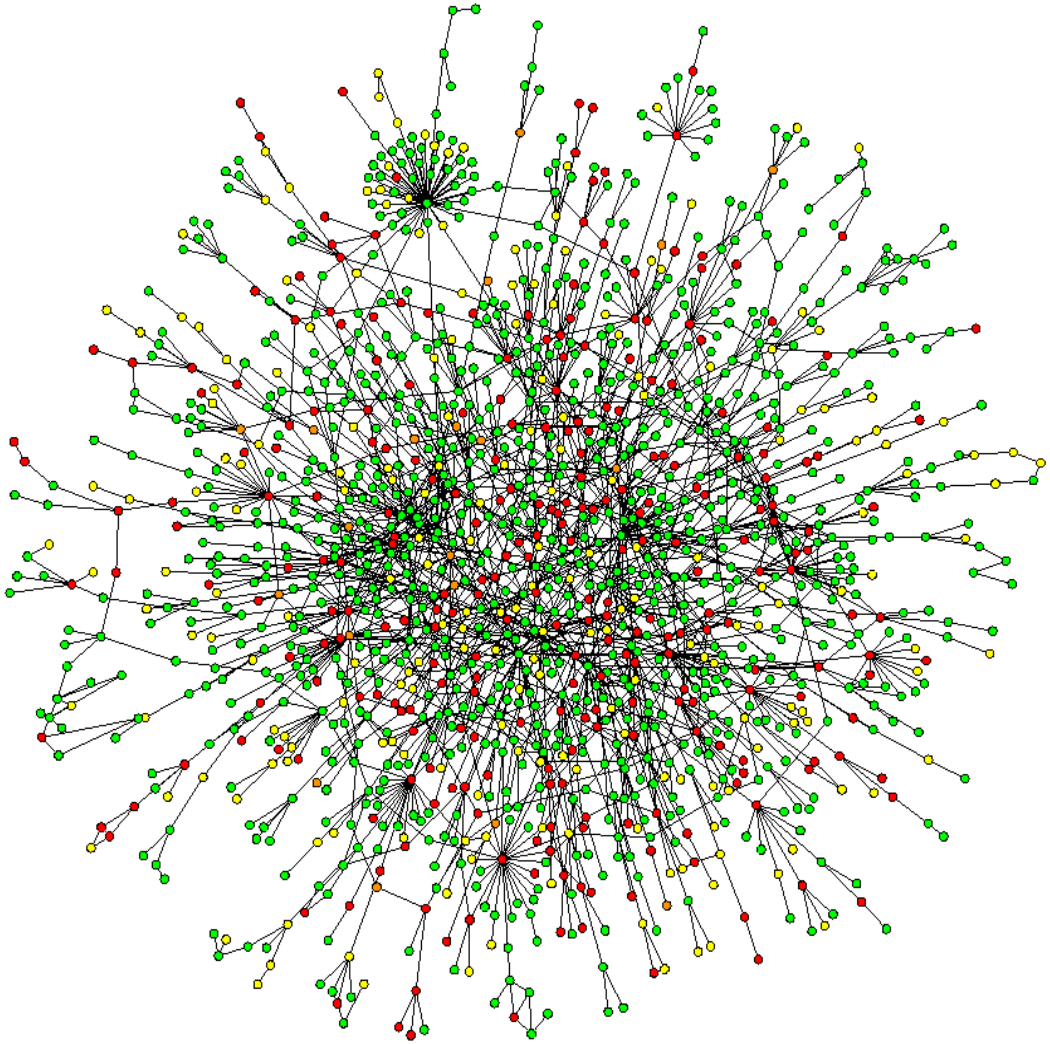


Brain Networks

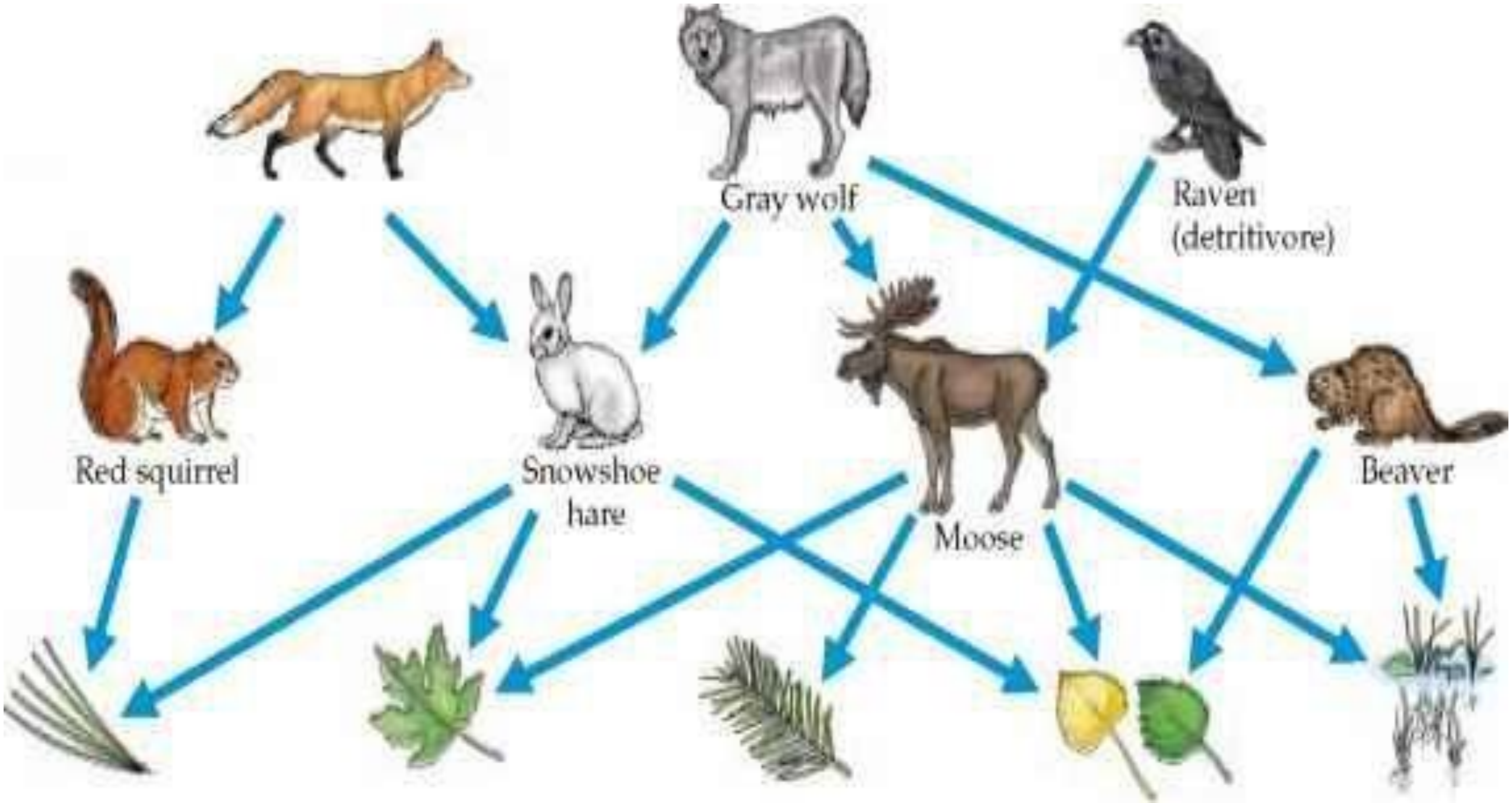




Protein-Protein Interaction



Food Webs (Ecology)

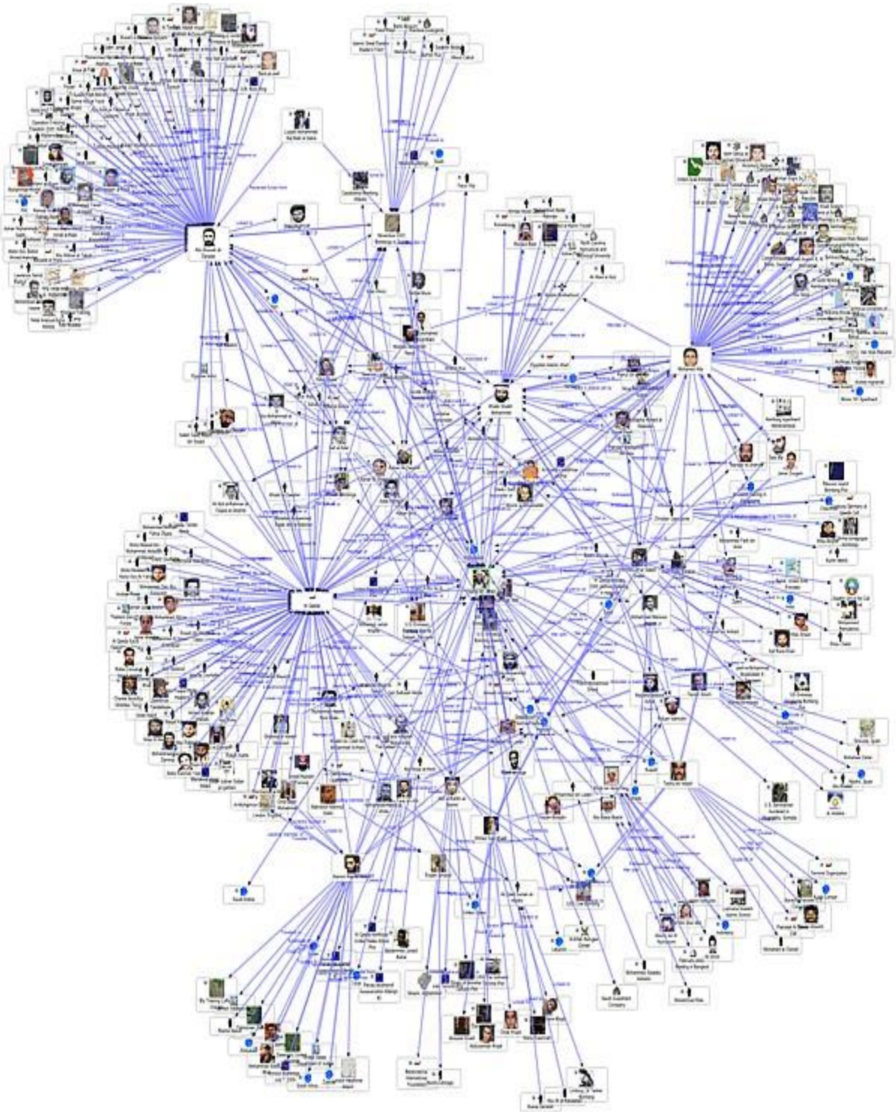


Other Biological Networks

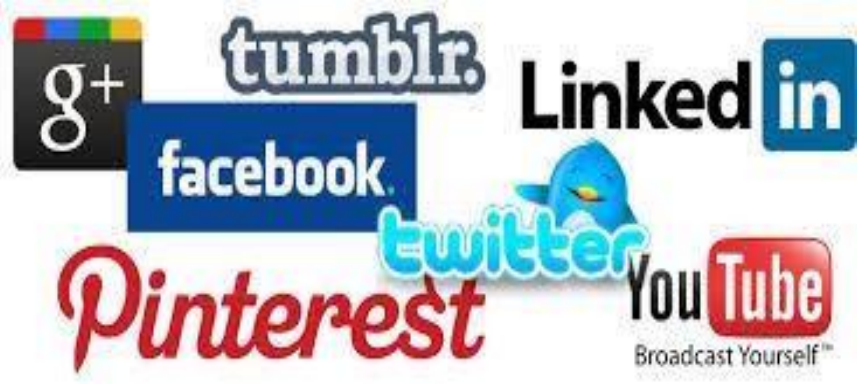
- Metabolic Networks
- Gene Regulatory Network
- Phylogenetic Trees
- Metabolic Pathways

SOCIAL NETWORKS

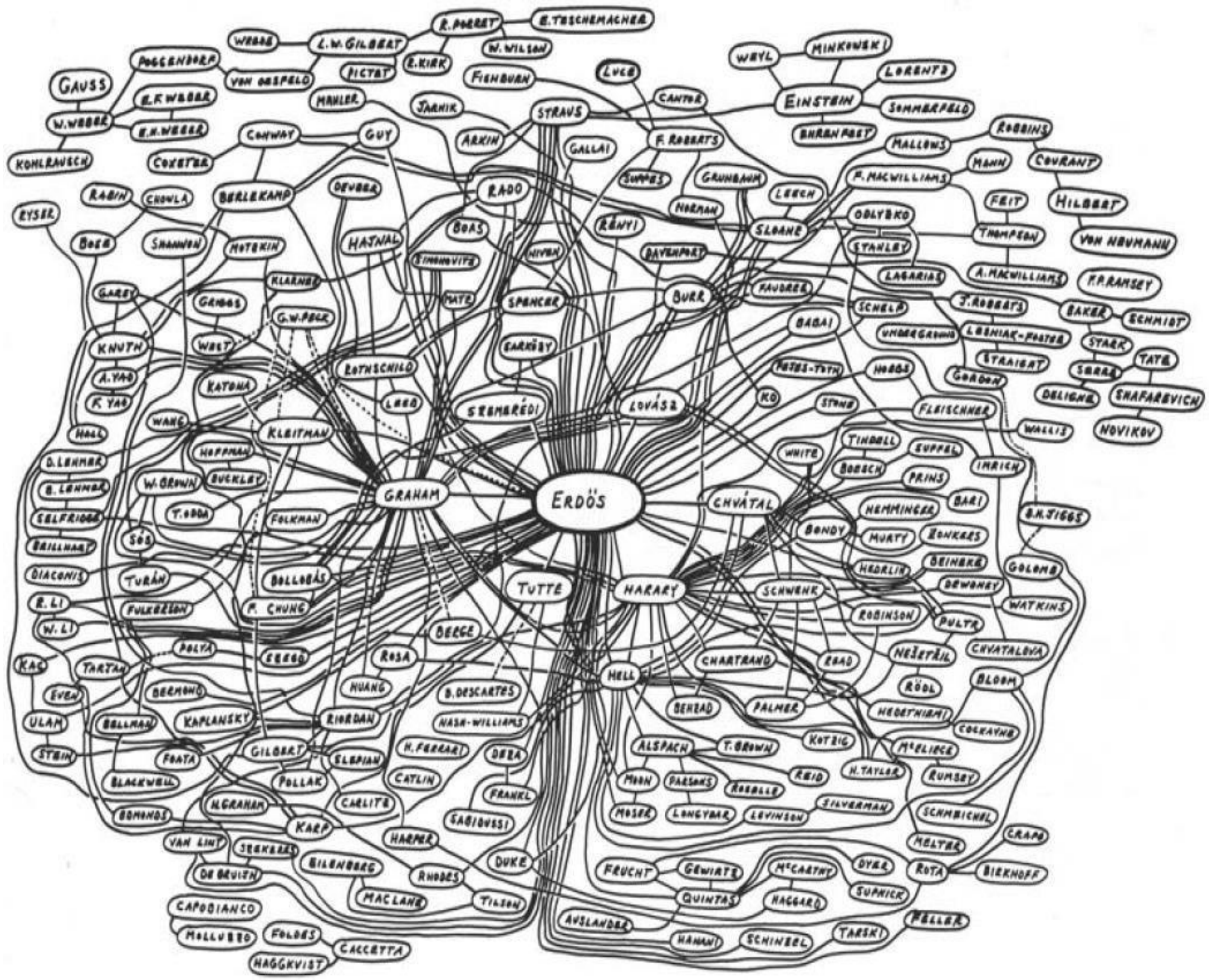
Friendship Networks



Online Social Networks

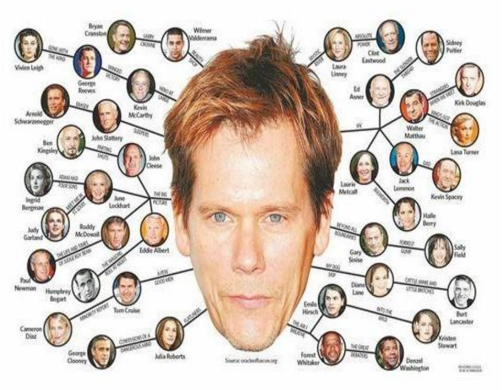


Co-authorship Network



Co-stardom Networks

- The collaboration graph of film actors



- Who is the co-star hub of Iranian films?!

Other Social Networks?

- Affiliation Networks
- Messaging Networks
  - Emails, phone calls, instant messaging, …
- Trust Network
- Non-human relations
  - Dolphins, …

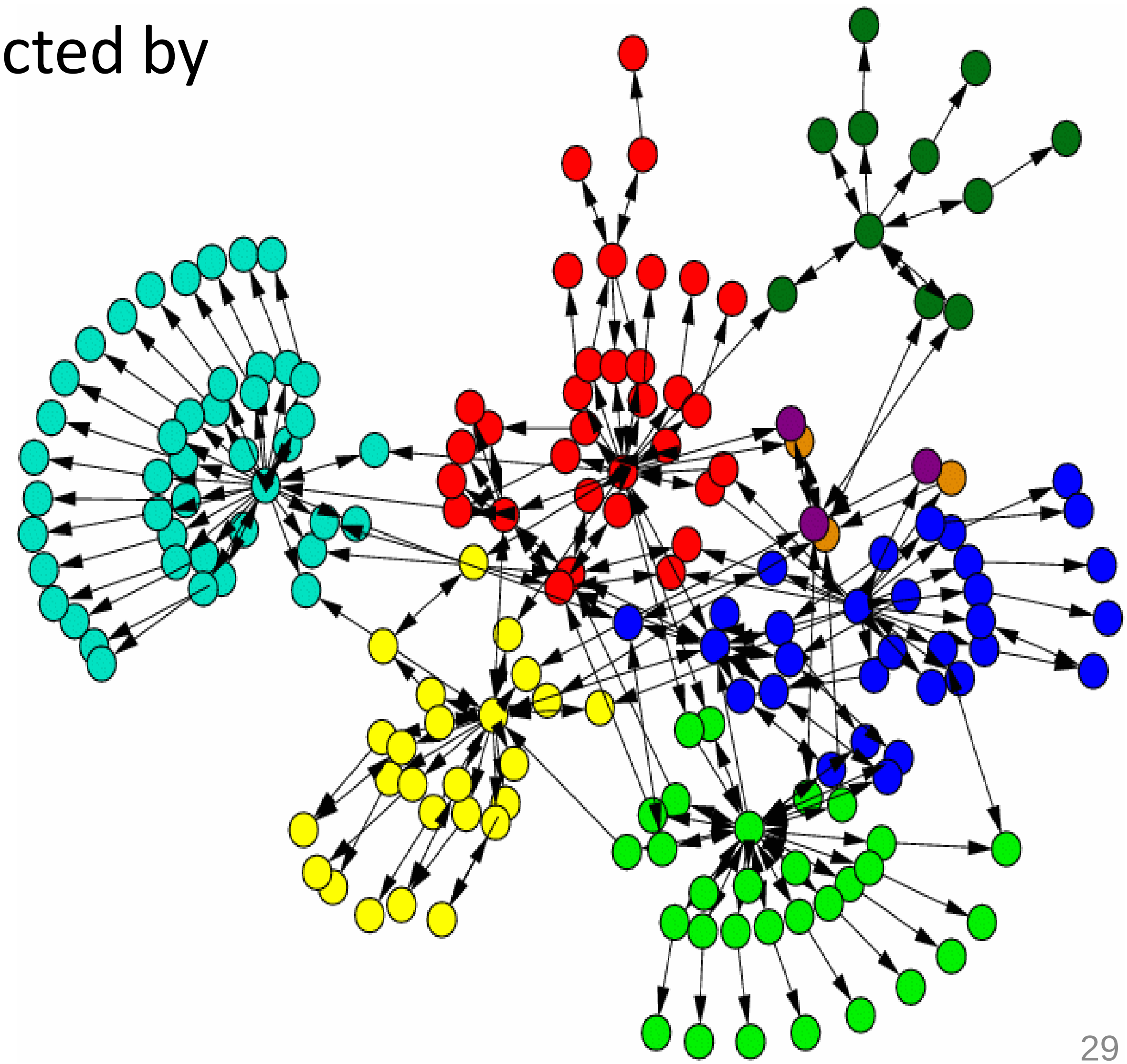
NETWORKS OF INFORMATION

Information Networks

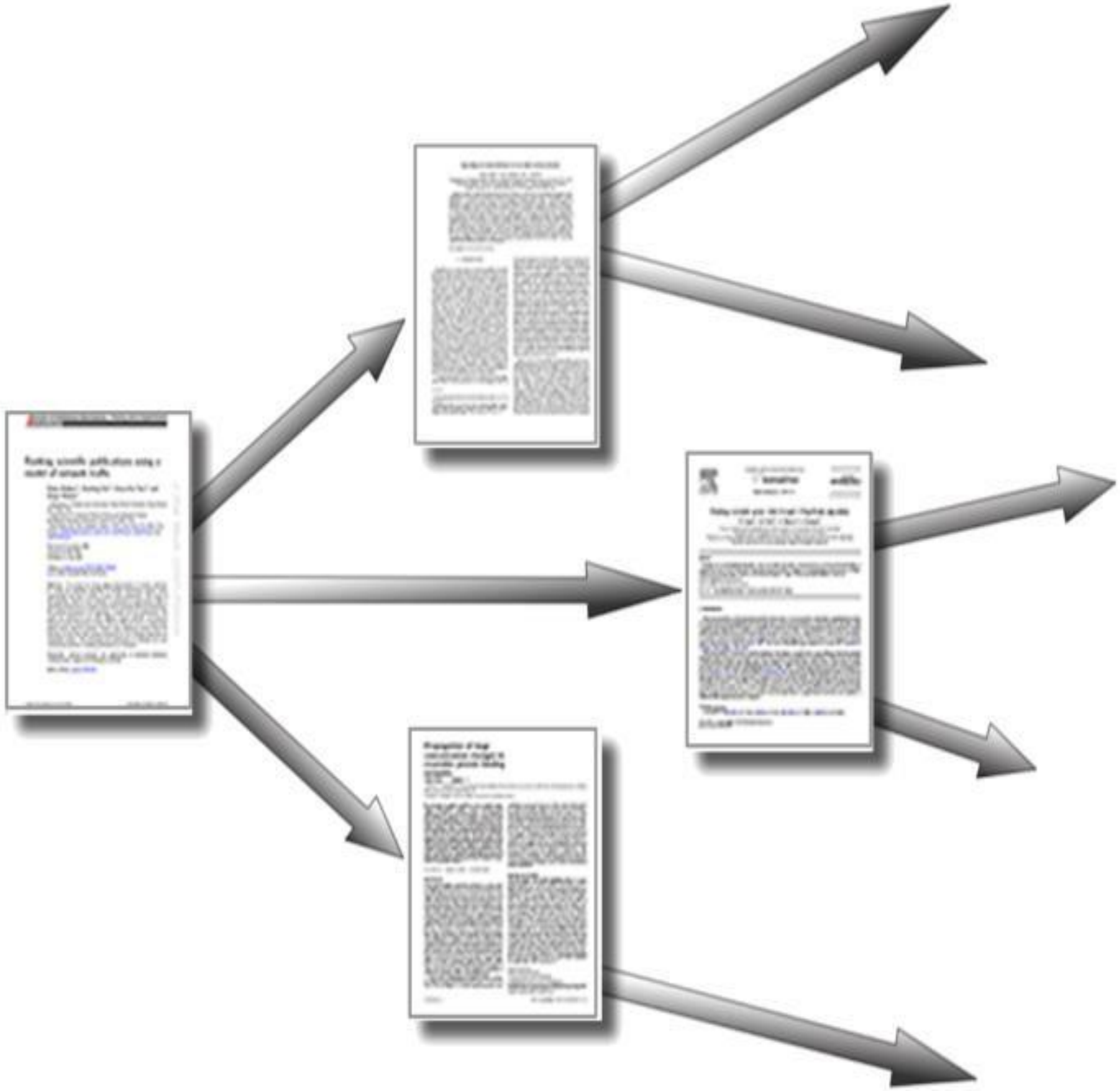
- World-wide web (WWW)
- Citation Networks
  - Papers, patents, …
  - Usually acyclic
  - Authors Citations: a social network extracted from papers
- P2P
- …

Webgages

- Webpages connected by **hyperlinks**



Citation Networks



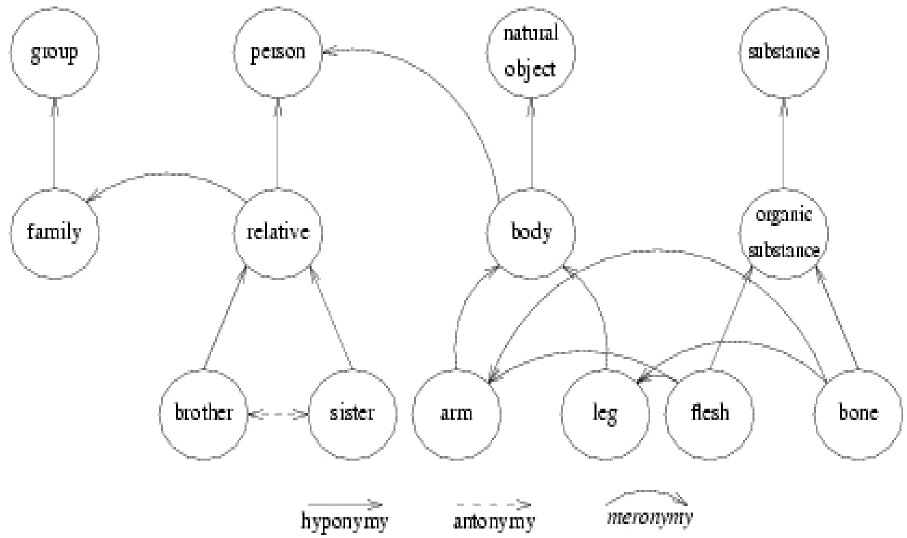
OTHER KINDS OF NETWORKS

Other Networks

- NLP: Words networks
- Economical Networks
  - **Money Transactions**
  - Trade Networks
  - Industry
  - Financial Networks
- Tourism
- …

Words Network

- E.g., wordnet



- Other words network?
  - Co-occurrences (in sentences, poems, …)
  - …

NETWORK QUESTIONS

Network Questions

- Structural
- Communities
- Dynamics of
- Dynamics on
- Algorithms
- Outlook

Network Questions: Structural

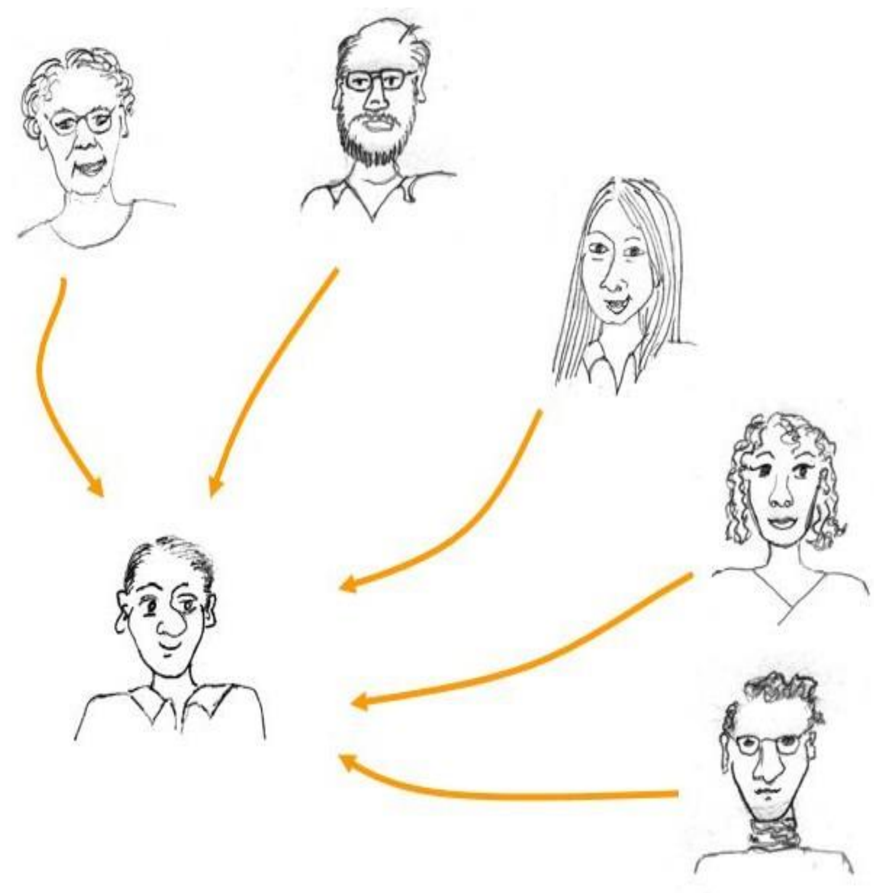
- How many connections does the average node have?
- Are some nodes more connected than others?
- Is the entire network connected?
- How many links are there between nodes?
  - Average distance, network diameter, …
- Are there clusters or groupings within which the connections are particularly strong?

Network Questions: Structural (cont’d)

- Is there any hierarchal structure?
- What is the best way to characterize a complex network?
- How can we tell if two networks are “different” or “similar”?
- Are there useful ways of classifying/categorizing nodes?
- Are there useful ways of classifying/categorizing networks?
- What are the important nodes and links?

Example

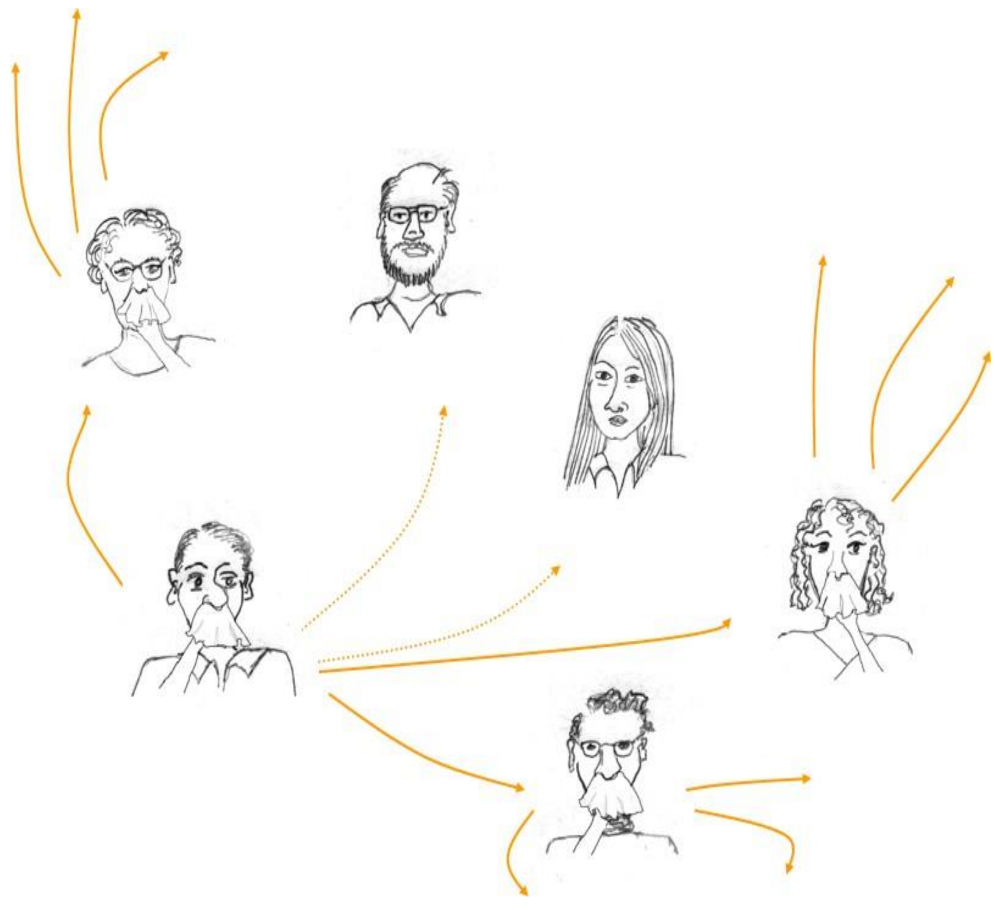
- In social networks, it's nice to be a hub



Example

- In social networks, it’s nice to be a hub

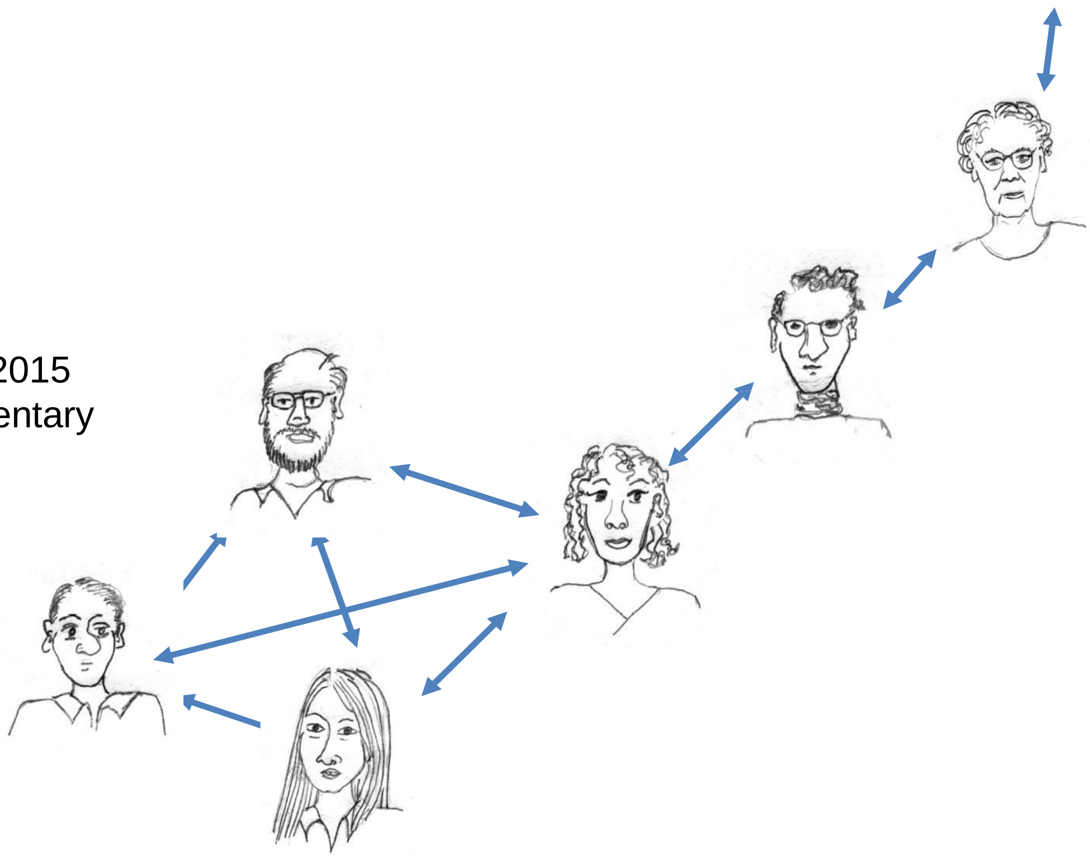
- But it Depends on What You’re Sharing!



Example: Small Worlds

- A friend of a friend is also frequently a friend
- Only **six hops** separate any two people in the world

Experimenter, 2015
Thriller/Documentary



Stanley Milgram

Born	August 15, 1933 The Bronx, New York City, U.S.
Died	December 20, 1984 (aged 51) Manhattan, New York City, U.S.
Education	Queens College, New York (B.A., Political Science, 1954) Harvard University (Ph.D., Social Psychology, 1960)
Known for	Milgram experiment Small-world experiment Familiar stranger

Network Questions: Communities

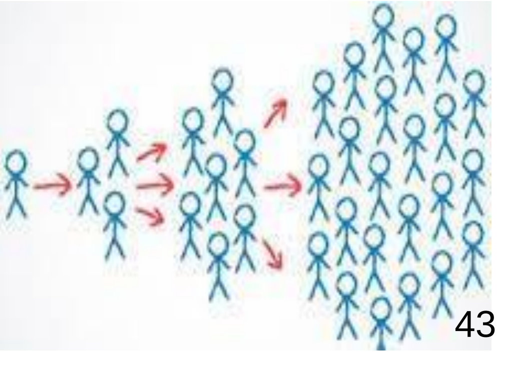
- Are there clusters in which the connections are particularly strong?
- How to discover communities, especially in large networks?
- How can we tell if these communities are statistically significant?
- What do these clusters tell us in specific applications?
- How we can optimize the number of communities?

Network Questions: Dynamics Of

- How can we model the growth of networks?
- What are the important features of networks that our models should capture?
- Are there “universal” models of network growth?
  - What details matter and what details don’t?
- How is the time-evolution of a network?
  - How about the reverse-time?! (e.g., sampling)
- How the network properties affected by its dynamical
- evolution?

Network Questions: Dynamics On

- How do diseases, computer viruses, innovations, rumors, revolutions, and opinions propagate on networks?
- What properties of networks are relevant to the answer of the above question?
- If you wanted to prevent (or encourage) spread of something on a network, what should you do?



Network Questions: Dynamics On

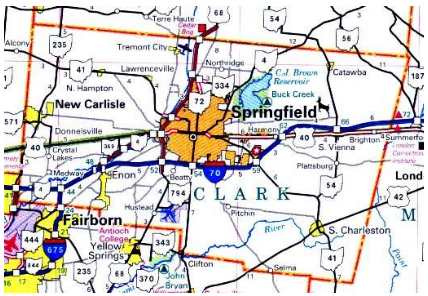
- What types of networks are **robust** to **random attack** or failure?
- What types of networks are robust to **intentional** and cascading attack?

Network Questions: Algorithms

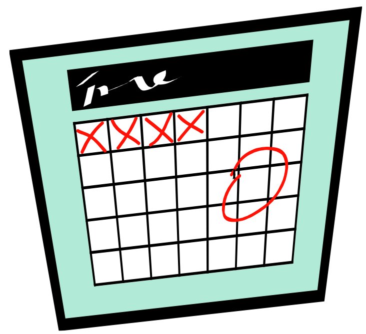
- What types of networks are **searchable** or navigable?
- What are good ways to **visualize** complex networks?
- What are the optimal algorithms for **computing network metrics**?
- How does google **page rank** work?
- If the internet were to double in size, would it still work?

Applications

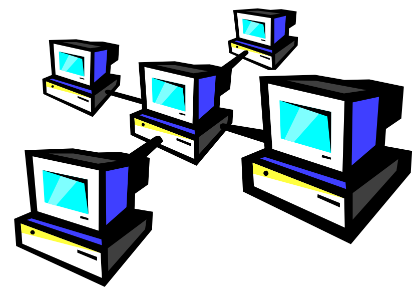
- Applications that involve not only a set of items, but also the connections between them



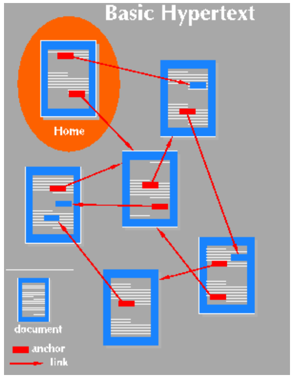
Maps



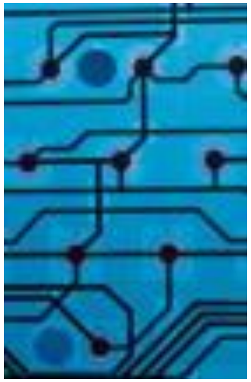
Schedules



Computer networks



Hypertext



Circuits

Graph Samples

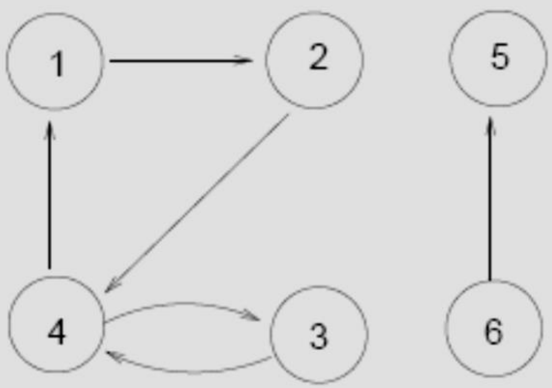
- Geography:
  - Cities and roads
  - Airports and flights (diameter $\approx$ 20 !!)
- Publications:
  - The co-authorship graph
    - E.g. the Erdos distance
  - The reference graph
- Phone calls: who calls whom
- Almost <u>everything</u> can be modeled as a graph !

Graph Terminology

Terminology

- Directed vs Undirected graphs

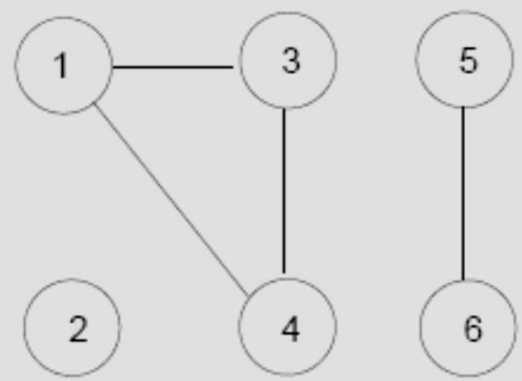
Directed graphs (digraphs)
(ordered pairs of vertices)



in-degree of v: # edges enetring v
out-degree of v: # edges leaving v

v is **adjacent** to u if there is an edge (u,v)

Undirected graphs
(unordered pairs of vertices)

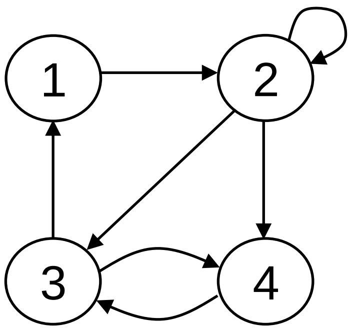


degree of v: # edges incident on v

v is **adjacent** to u and u is **adjacent** to v
if there is and edge (u,v)

Terminology (cont'd)

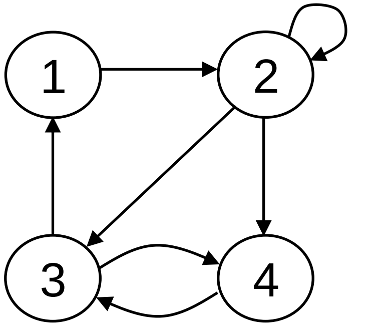
- Complete graph
  - A graph with an edge between each pair of vertices
- Subgraph
  - A graph (V', E') such that V'$\subseteq$V and E'$\subseteq$E
- Path from v to w
  - A sequence of vertices <v_0, v_1, …, v_k> such that v_0=v and v_k=w
- Length of a path
  - Number of edges in the path



path from v_1 to v_4
<v_1, v_2, v_4>

Terminology (cont'd)

- w is **reachable** from v
  - If there is a path from v to w
- **Simple** path
  - All the vertices in the path are distinct
- Cycles
  - A path $<v_0, v_1, \ldots, v_k>$ forms a cycle if $v_0=v_k$ and $k\geq 2$
- Acyclic graph
  - A graph without any cycles



cycle from v_1 to v_1
$<v_1, v_2, v_3, v_1>$

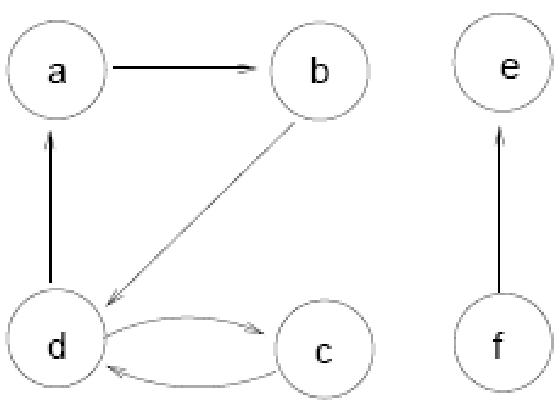
Terminology (cont'd)

Connected and Strongly Connected

directed graphs

strongly connected: every two vertices are reachable from each other

strongly connected components : all possible strongly connected subgraphs

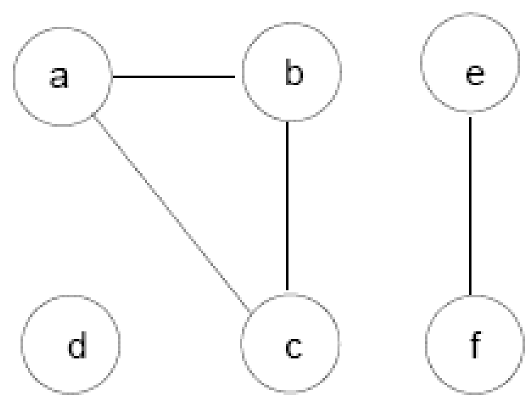


strongly connected components: {a,b,c,d} { e} {f}

undirected graphs

connected: every pair of vertices is connected by a path

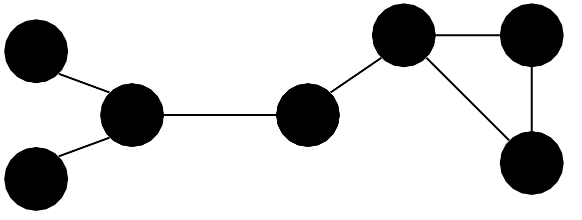
connected components: all possible connected subgraphs



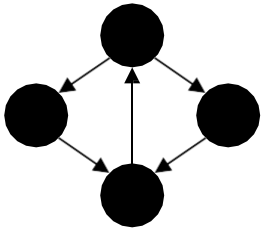
connected components: {a,b,c} {d} {e,f}

Connectivity

Undirected graphs are *connected* if there is a path between any two vertices

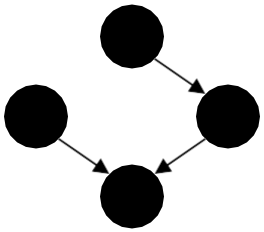


- Directed graphs are *strongly connected* if there is a path from any one vertex to any other

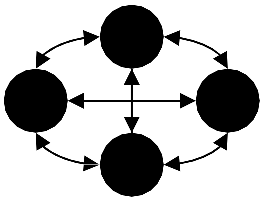


Connectivity

Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



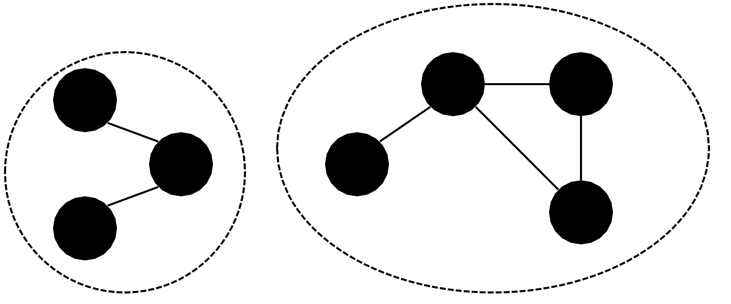
- A *complete* graph has an edge between every pair of vertices



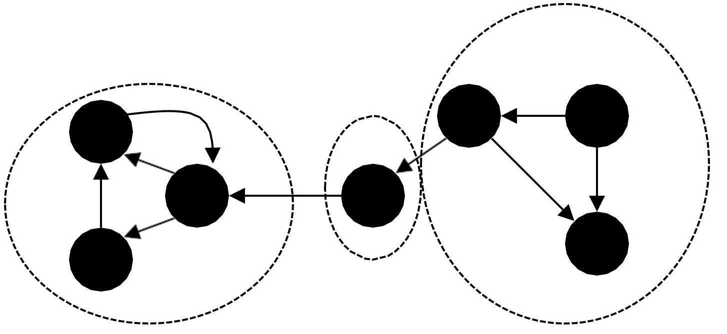
Connectivity

A *(strongly) connected component* is a subgraph which is (strongly) connected

CC in an undirected graph:

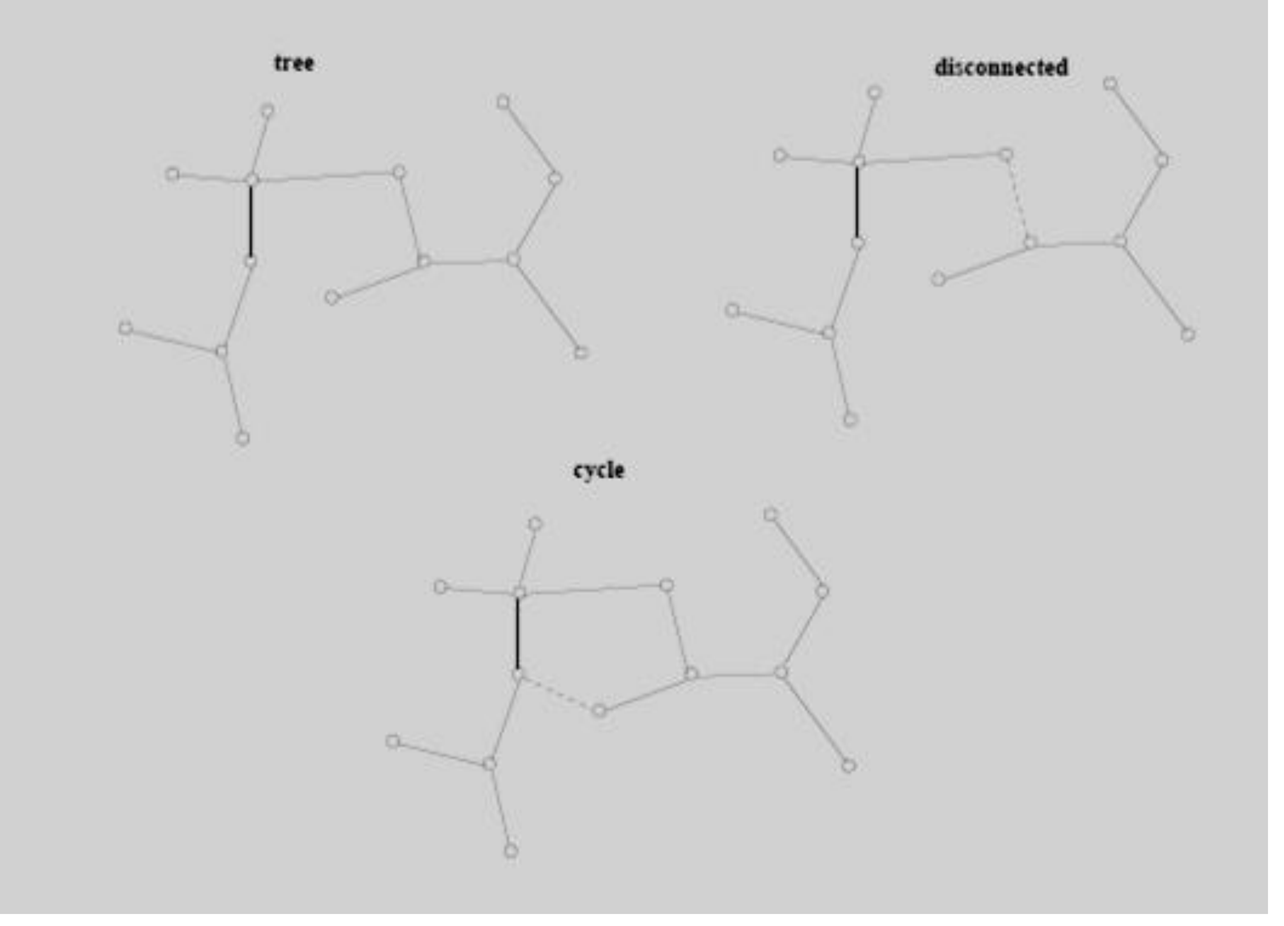


SCC in a directed graph:



Terminology (cont'd)

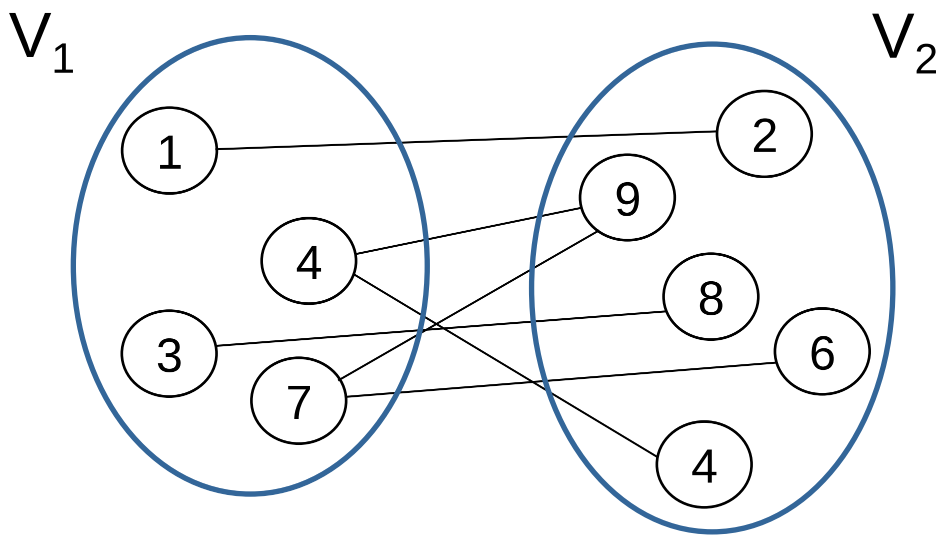
- A **tree** is a connected, acyclic undirected graph



Terminology (cont'd)

- A **bipartite graph** is an undirected graph

  G = (V, E) in which $V = V_1 + V_2$ and there are edges only between vertices in V_1 and V_2



Bipartiteness

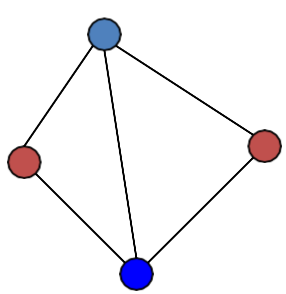
Graph G = (V,E) is **bipartite** iff it can be partitioned into two sets of nodes A and B such that each edge has one end in A and the other end in B

Alternatively:

- Graph G = (V,E) is bipartite iff all its cycles have even length
- Graph G = (V,E) is bipartite iff nodes can be coloured using two colours

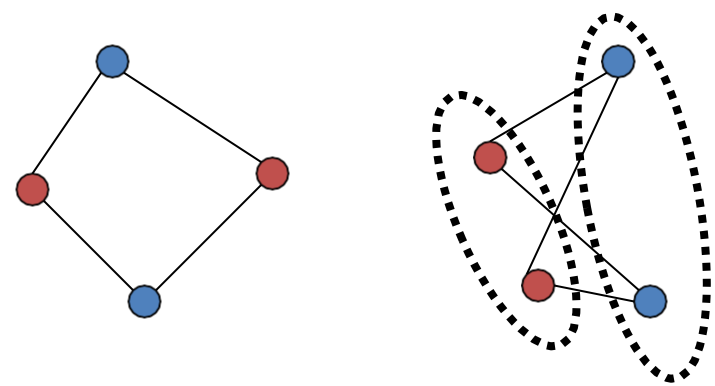
Question: given a graph G, how to test if the graph is bipartite?

Note: graphs without cycles (trees) are bipartite



non bipartite

bipartite:



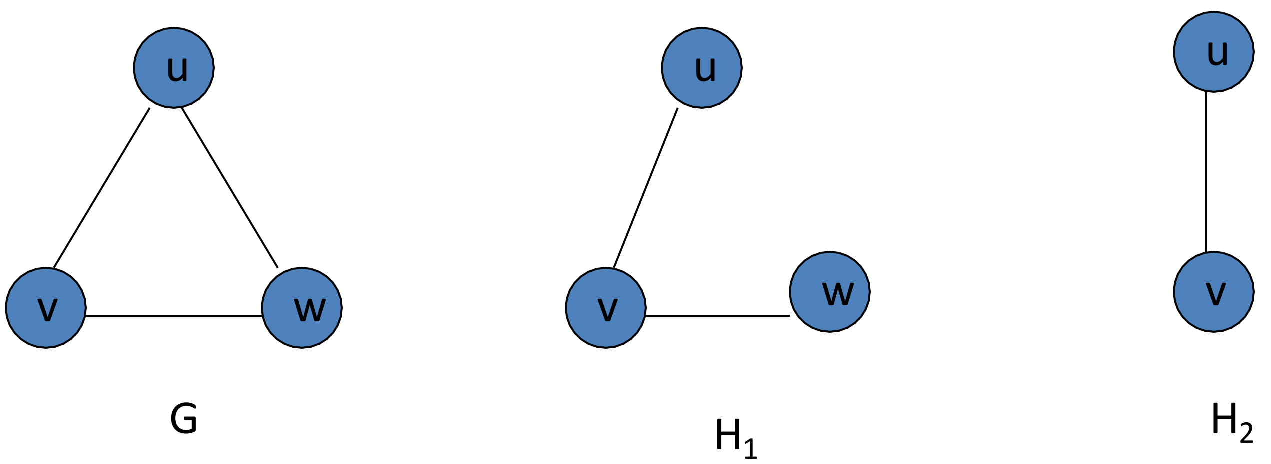
Distance and Diameter

- The distance between two nodes, d(u,v), is the length of **the shortest paths**, or ∞ if there is no path
- The diameter of a graph is the largest distance between any two nodes
- Graph is strongly connected iff diameter < ∞

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

- Example applications: solving sub-problems within a graph
  Representation example: V = {u, v, w}, E = ({u, v}, {v, w}, {w, u}}, H_1 , H_2



Graph - Isomorphism

- G1 = (V1, E2) and G2 = (V2, E2) are isomorphic if:
- There is a one-to-one and onto 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**

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

Graph - Isomorphism

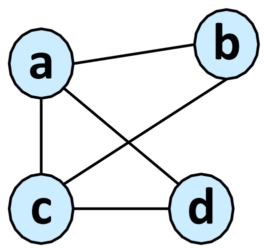
Representation example: G1 = (V1, E1) , G2 = (V2, E2)

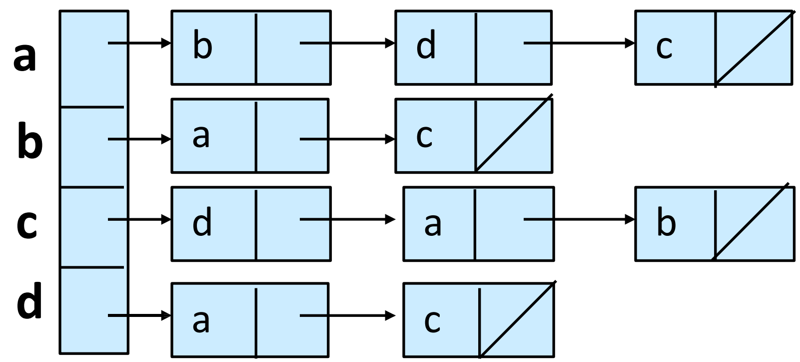
$f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, $f(u_4) = v_2$,

Representation of Graphs

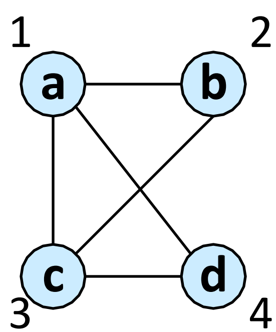
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





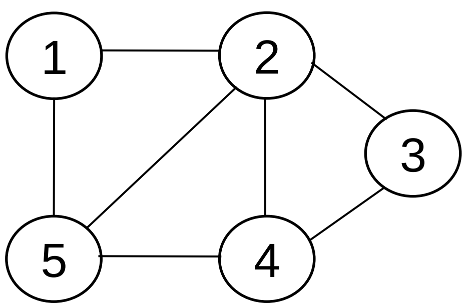
  - Adjacency Matrix.



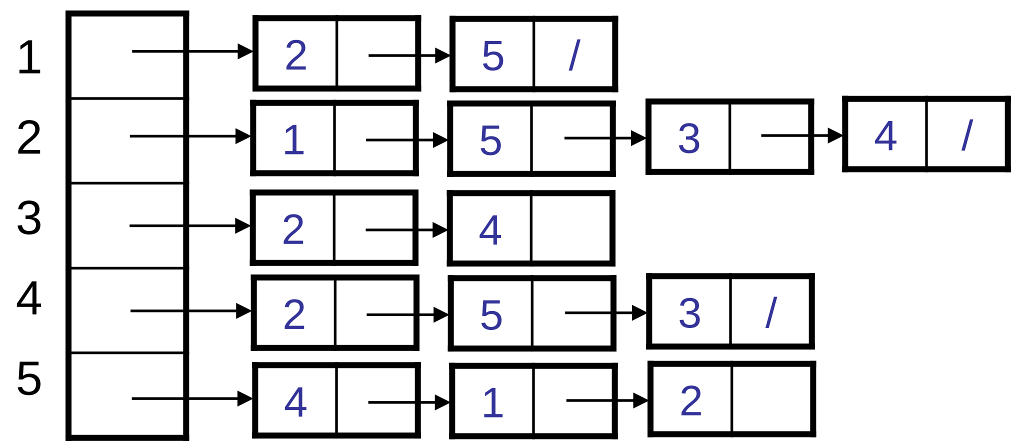
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Graph Representation

- **Adjacency list representation** of G = (V, E)
  - An array of |V| lists, one for each vertex in V
  - Each list `Adj[u]` contains all the vertices v that are adjacent to u (i.e., there is an edge from `u` to `v`)
  - Can be used for both directed and undirected graphs

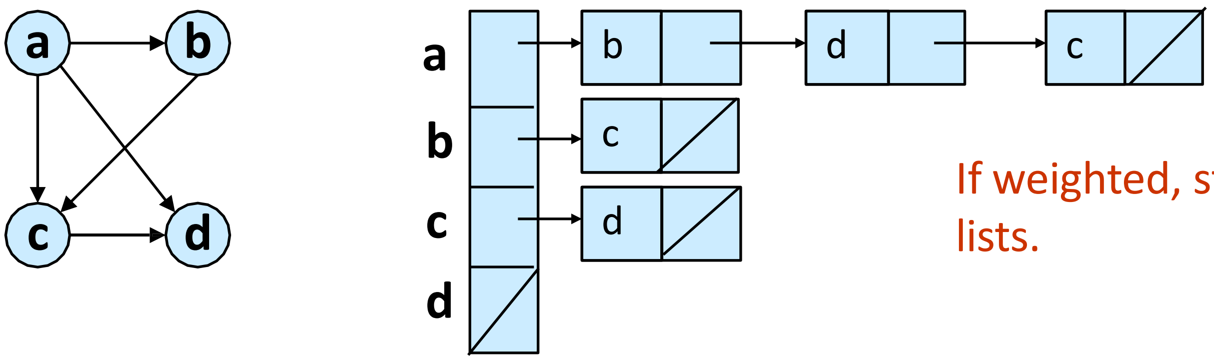


Undirected graph

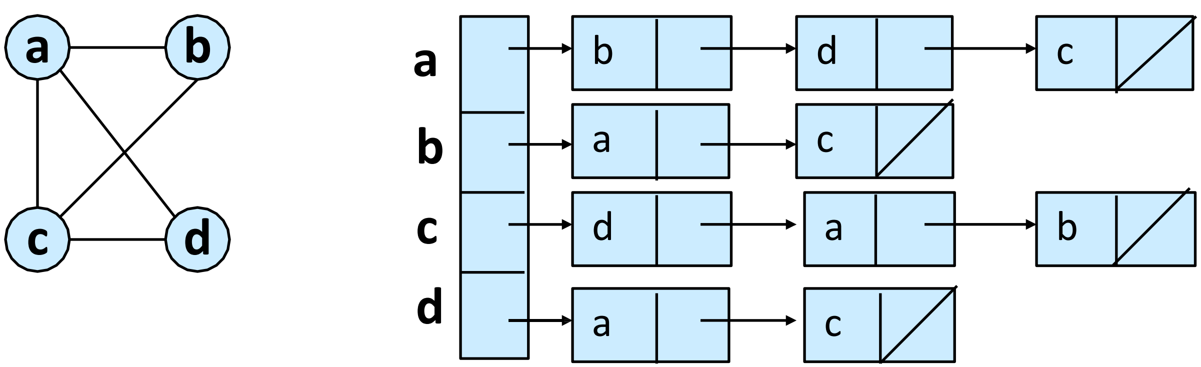


Adjacency Lists

- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.

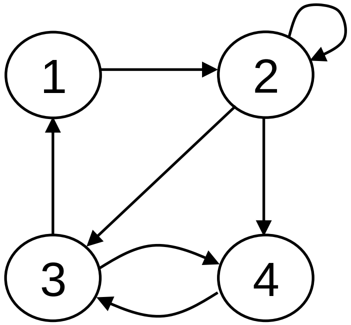


If weighted, store weights also in adjacency lists.

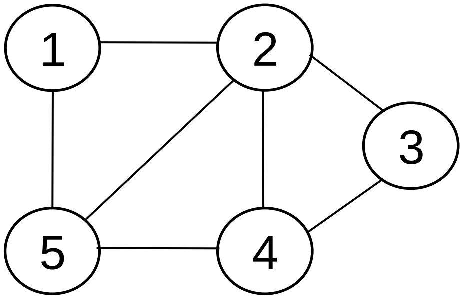


Properties of Adjacency-List Representation

- Sum of "lengths" of all adjacency lists
  - Directed graph: $|E|$
    - edge (u, v) appears only once (i.e., in the list of **u**)
  - Undirected graph: $2|E|$
    - edge (u, v) appears twice (i.e., in the lists of both **u** and **v**)



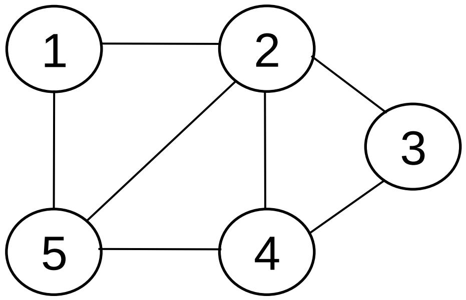
Directed graph



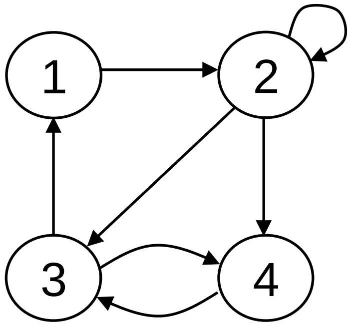
Undirected graph

Properties of Adjacency-List Representation

- Memory required
  - $\Theta(V + E)$
- Preferred when
  - The graph is **sparse**: $|E| \ll |V|^2$
  - We need to quickly determine the nodes adjacent to a given node.
- Disadvantage
  - No quick way to determine whether there is an edge between node u and v
- Time to determine if $(u, v) \in E$:
  - O(degree(u))
- Time to list all vertices adjacent to u:
  - Θ(degree(u))



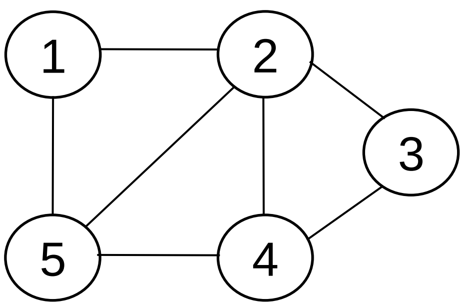
Undirected graph



Directed graph

Graph Representation

- **Adjacency matrix representation** of G = (V, E)
  - Assume vertices are numbered 1, 2, … |V|
  - The representation consists of a matrix $A_{|V| x |V|}$:
  - $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

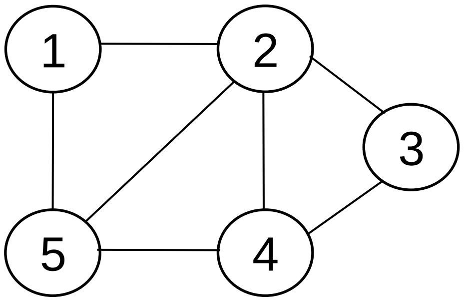
For undirected graphs, matrix A is symmetric:

$a_{ij} = a_{ji}$

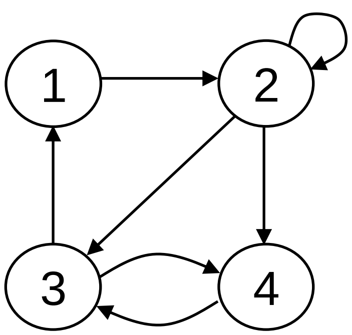
$A = A^T$

Properties of Adjacency Matrix Representation

- Memory required
  - $\Theta(V^2)$, independent on the number of edges in G
- Preferred when
  - The graph is **dense:** $|E|$ is close to $|V|^2$
  - We need to quickly determine if there is an edge between two vertices
- Time to determine if $(u, v) \in E$:
  - $\Theta(1)$
- Disadvantage
  - No quick way to determine the vertices adjacent to another vertex
- Time to list all vertices adjacent to u:
  - $\Theta(V)$



Undirected graph



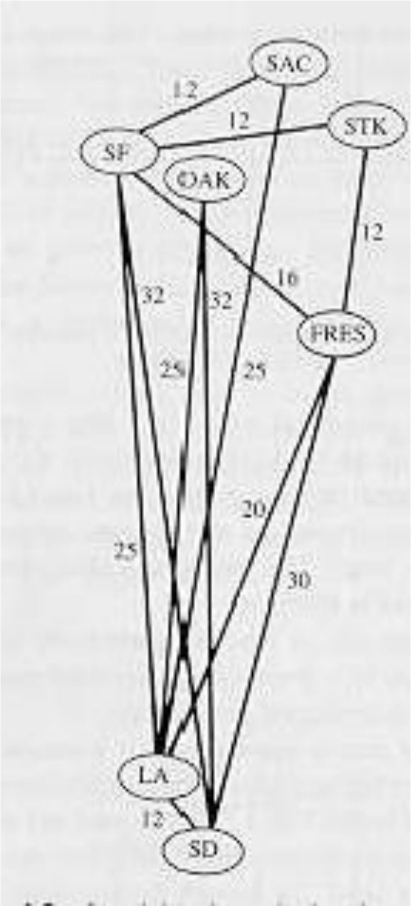
Directed graph

Weighted Graphs

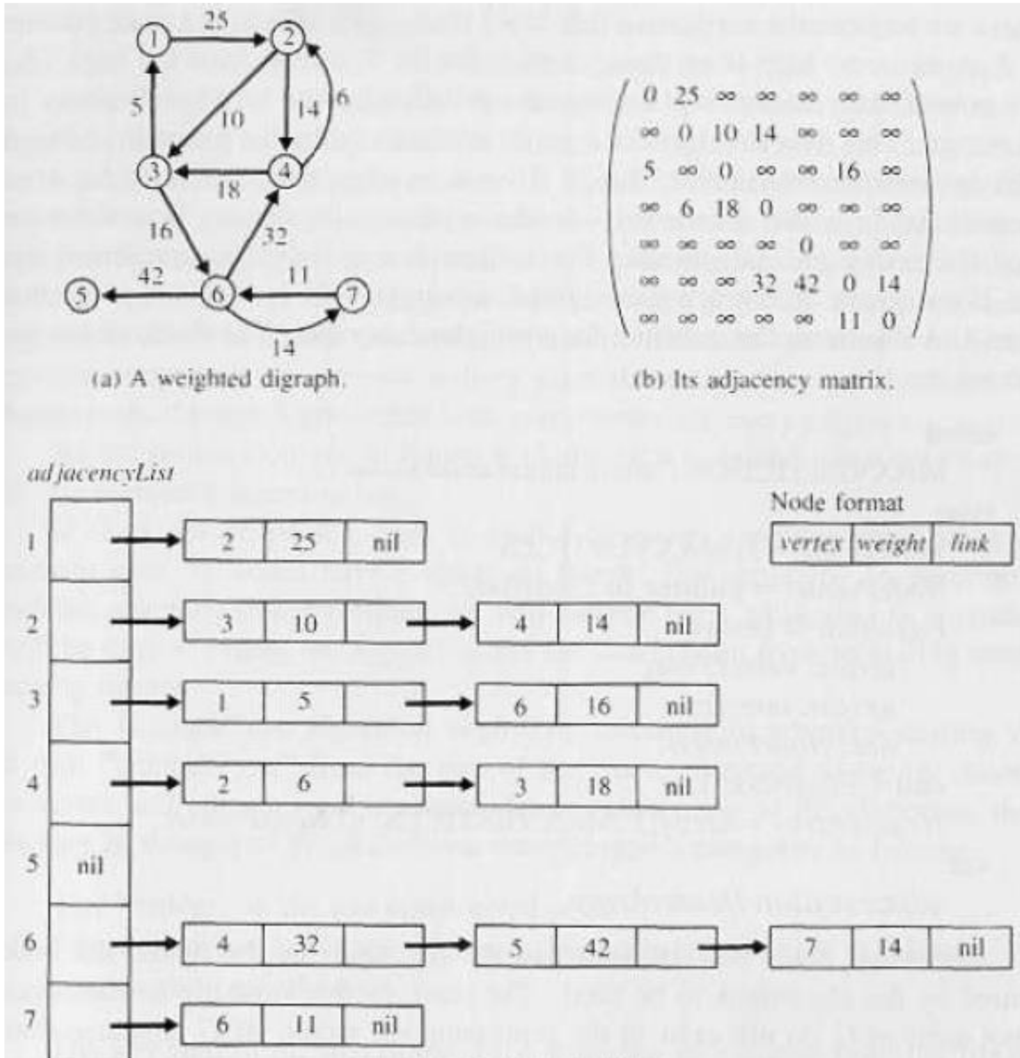
- Graphs for which each edge has an associated weight $w(u, v)$

  $w: E \rightarrow R$, weight function

- Storing the weights of a graph
  - Adjacency list:
    - Store $w(u,v)$ along with vertex v in u's adjacency list
  - Adjacency matrix:
    - Store $w(u, v)$ at location (u, v) in the matrix



Weighted Graphs



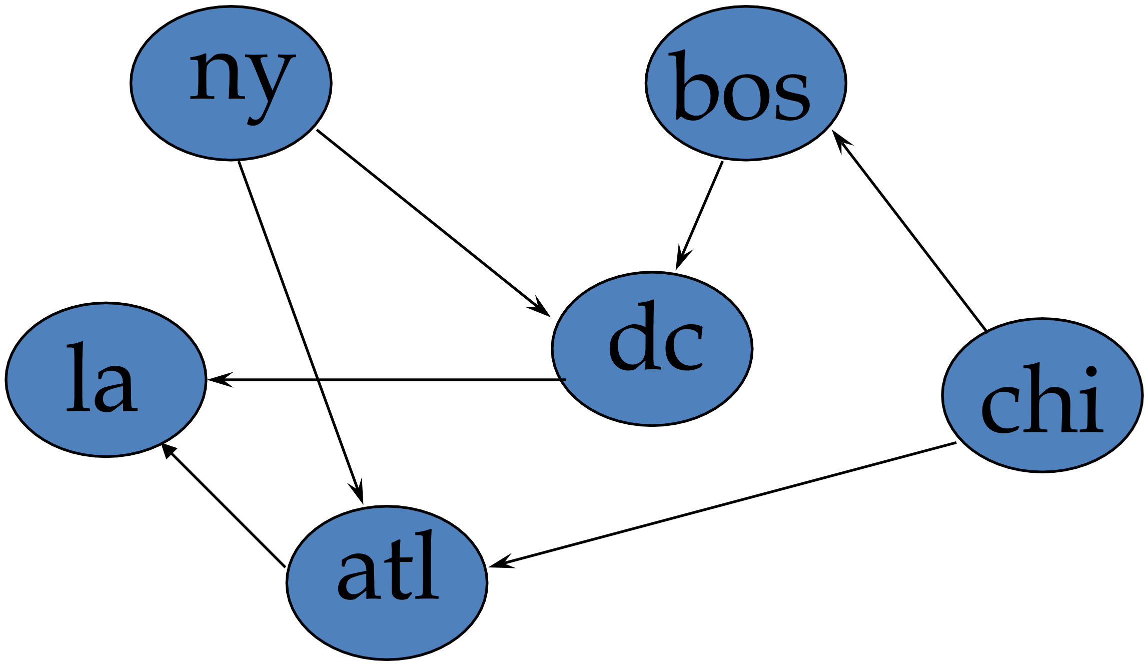
(a) A weighted digraph.
(b) Its adjacency matrix.

Some graph operations

	adjacency matrix	adjacency lists
insertEdge	O(1)	O(e)
isEdge	O(1)	O(e)
#successors?	O(V)	O(e)
#predecessors?	O(V)	O(E)

Traversing a graph

Traversing a graph



- Where to start?
- Will all vertices be visited?
- How to prevent multiple visits?

Graph-searching Algorithms

- **Searching a graph**
  - **Systematically** follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

Breadth-first Search (BFS)

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree <u>with root *s*</u> that contains all <u>reachable</u> vertices.

Breadth-first Search (BFS)

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - **White** – Undiscovered.
  - **Gray** – Discovered but not finished.
  - **Black** – Finished.

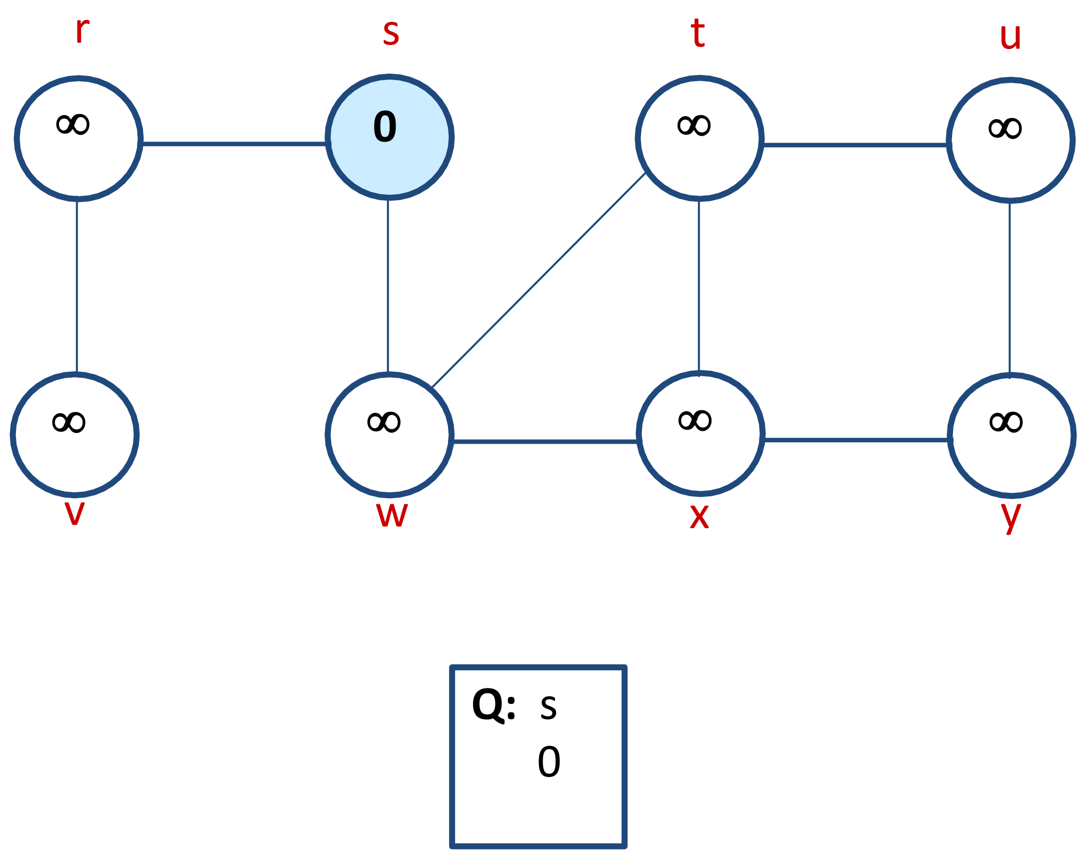
BFS(G,s)

```
for each vertex u in V[G] – {s}
        do color[u] ← white                initialization
           d[u] ← ∞
           π[u] ← nil
color[s] ← gray
d[s] ← 0                                   access source s
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
                for each v in Adj[u]
                        do if color[v] = white
                                then color[v] ← gray
                                     d[v] ← d[u] + 1
                                     π[v] ← u
                                     enqueue(Q,v)
                color[u] ← black
```

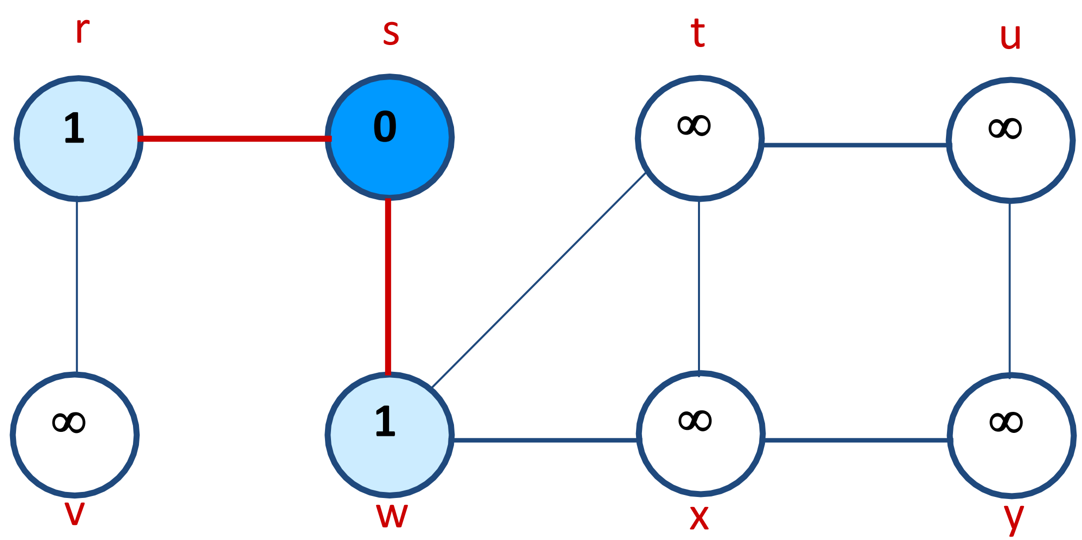
white: undiscovered
gray: discovered black:
finished

Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)

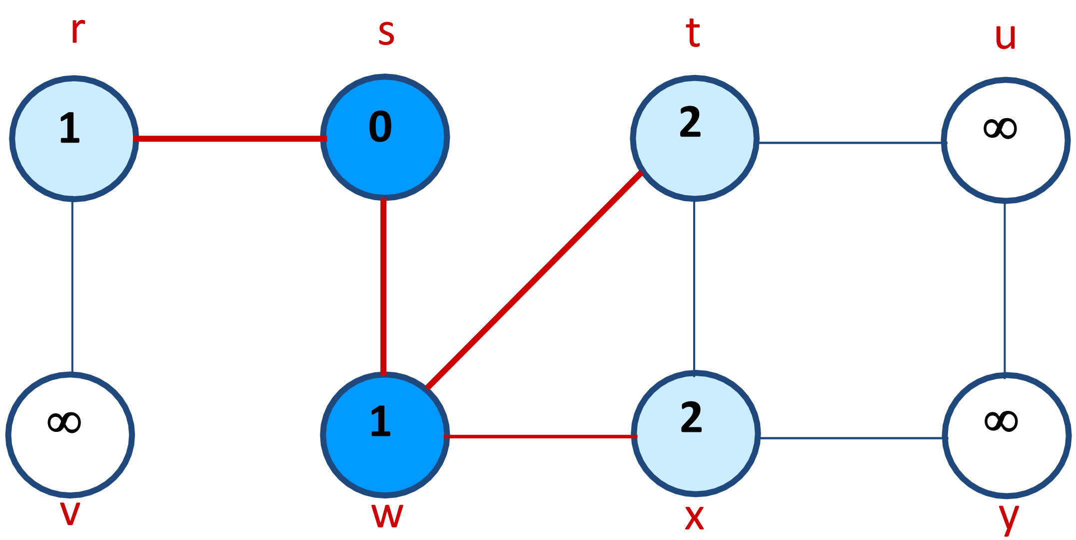


Example (BFS)



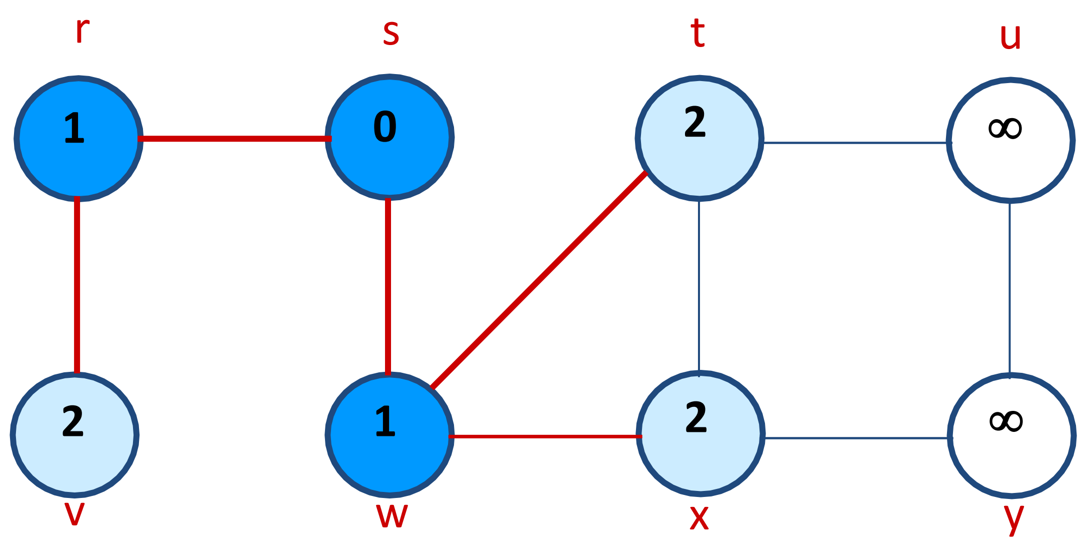
Q: w r
1 1

Example (BFS)



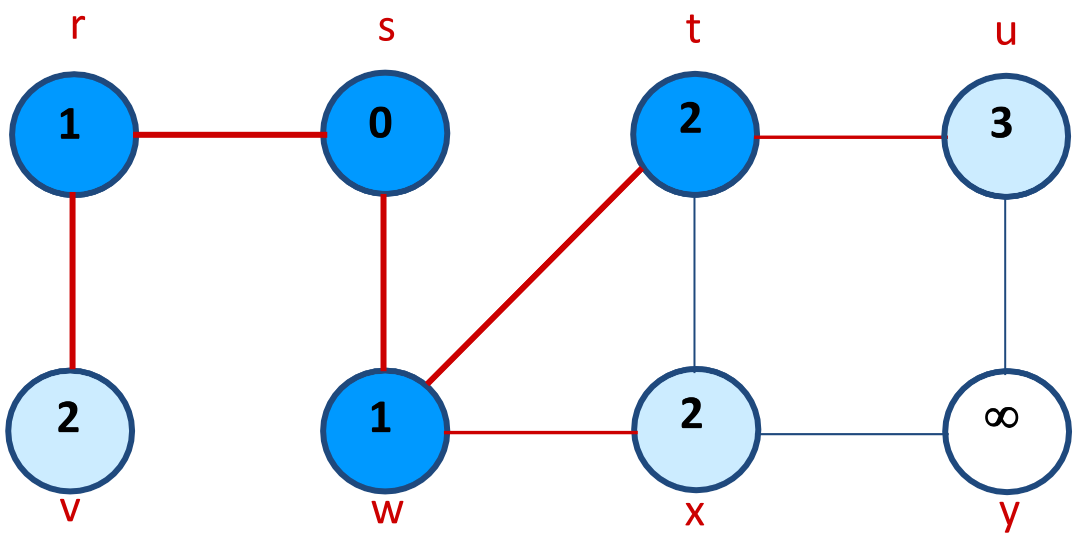
Q: r t x
1 2 2

Example (BFS)



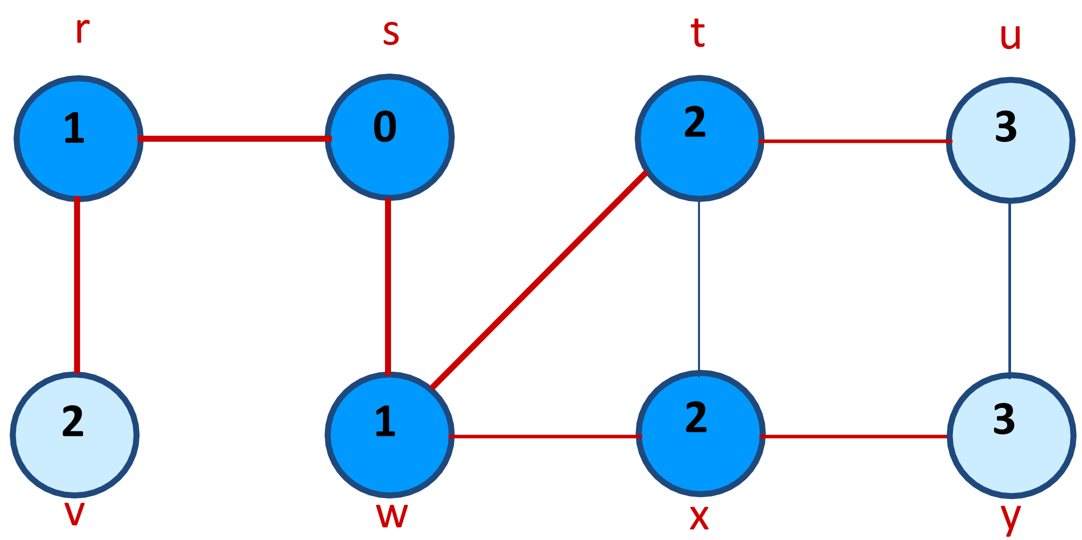
Q: t x v
2 2 2

Example (BFS)



Q: x v u
2 2 3

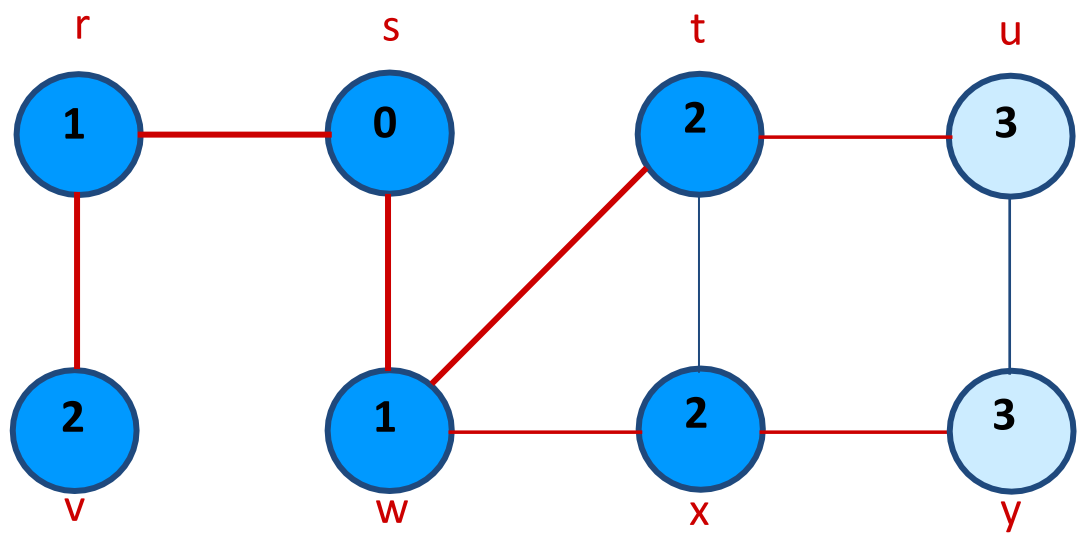
Example (BFS)



Q: v u y

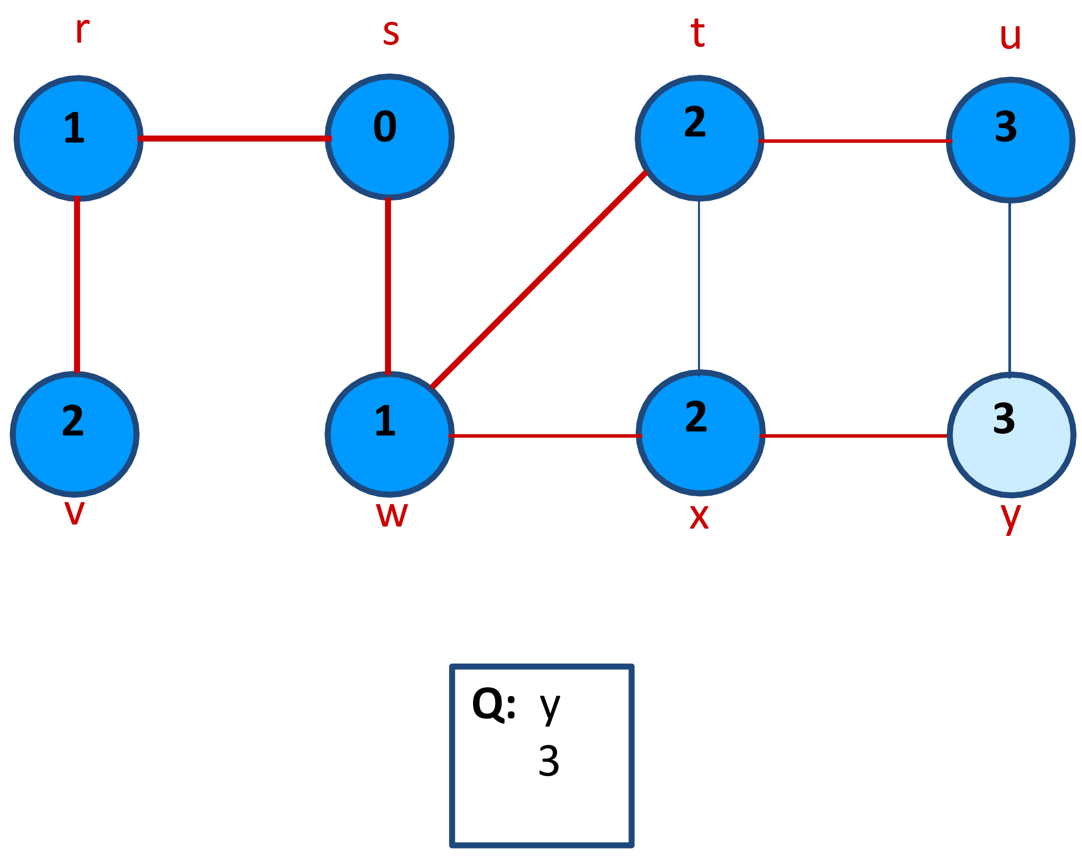
2 3 3

Example (BFS)

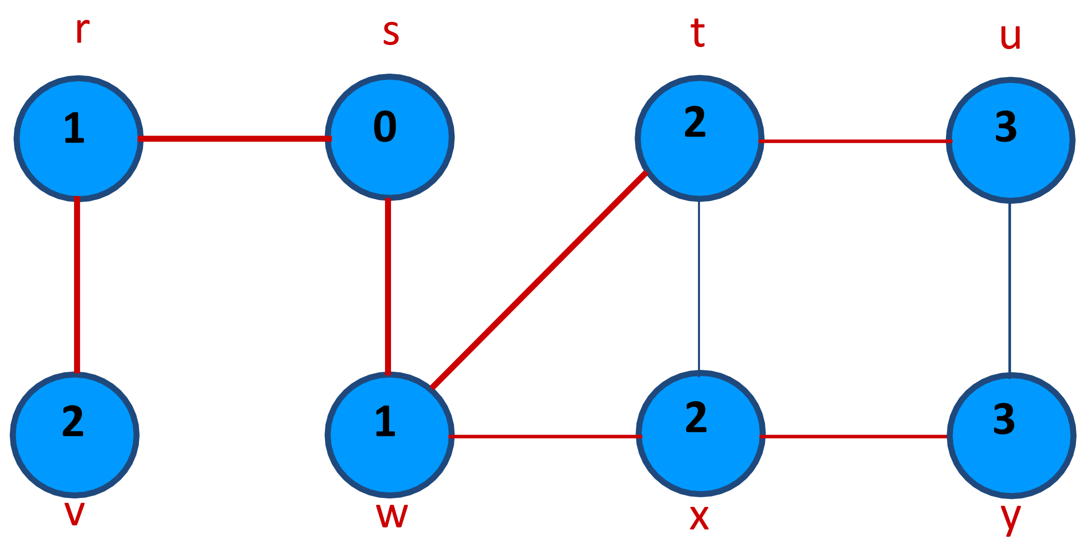


Q: u y
3 3

Example (BFS)

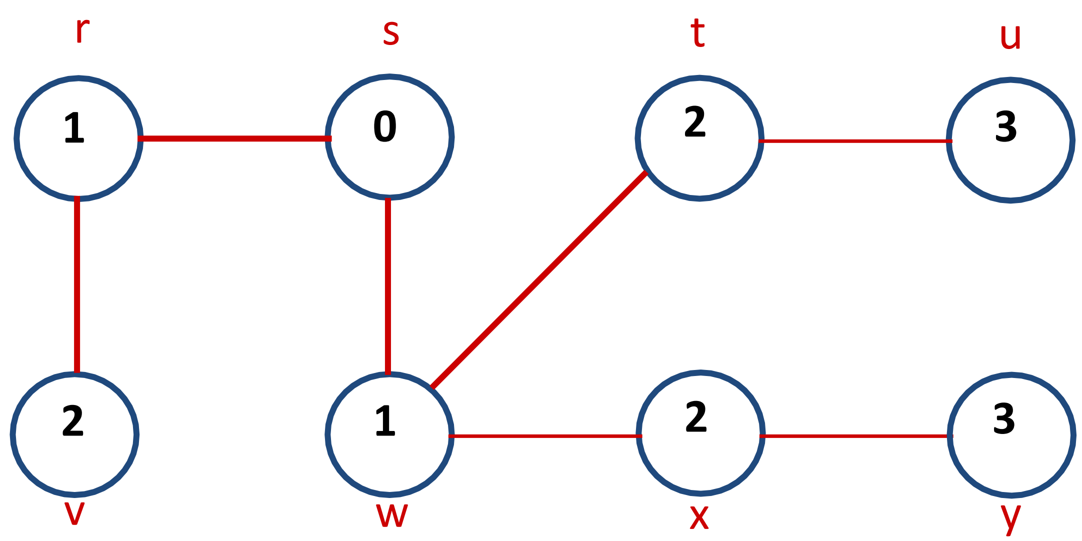


Example (BFS)



Q: $\varnothing$

Example (BFS)



BF Tree

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered.
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Coloring scheme for vertices as BFS. A vertex is
  - "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if it is a leaf node or all vertices adjacent to it have been finished.

Pseudo-code

DFS(*G*)

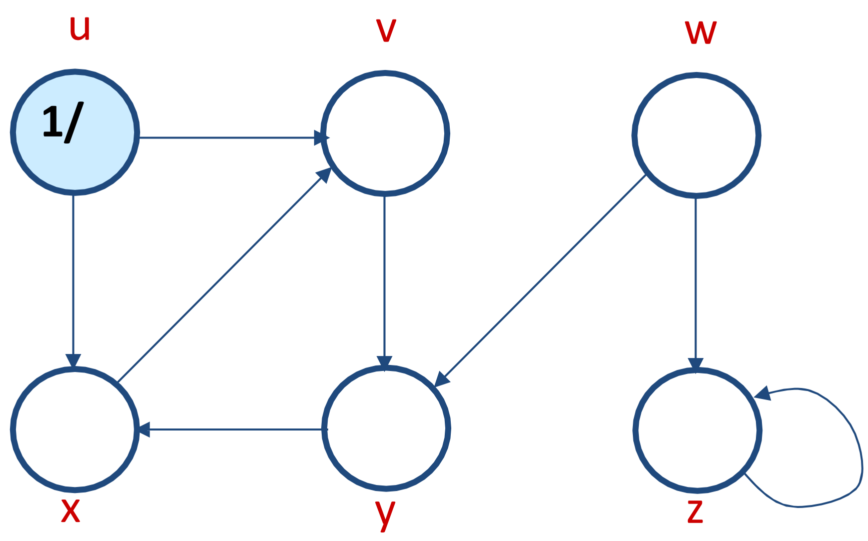
```
for each vertex u ∈ V[G]
    do color[u] ← white
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = white
          then DFS-Visit(u)
```

Uses a global timestamp ***time***.

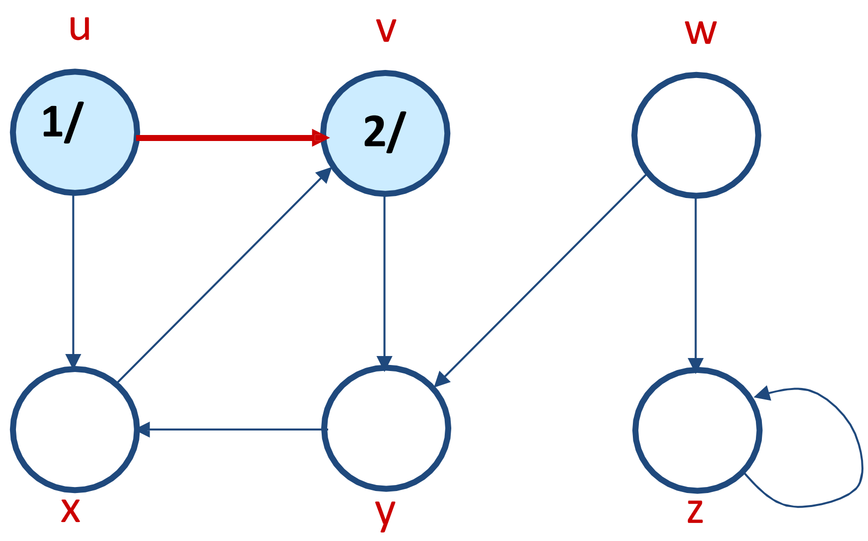
DFS-Visit(*u*)

```
color[u] ← GRAY  // White vertex u
                    has been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-Visit(v)
color[u] ← BLACK     // Blacken u;
                        it is finished.
f[u] ← time ← time + 1
```

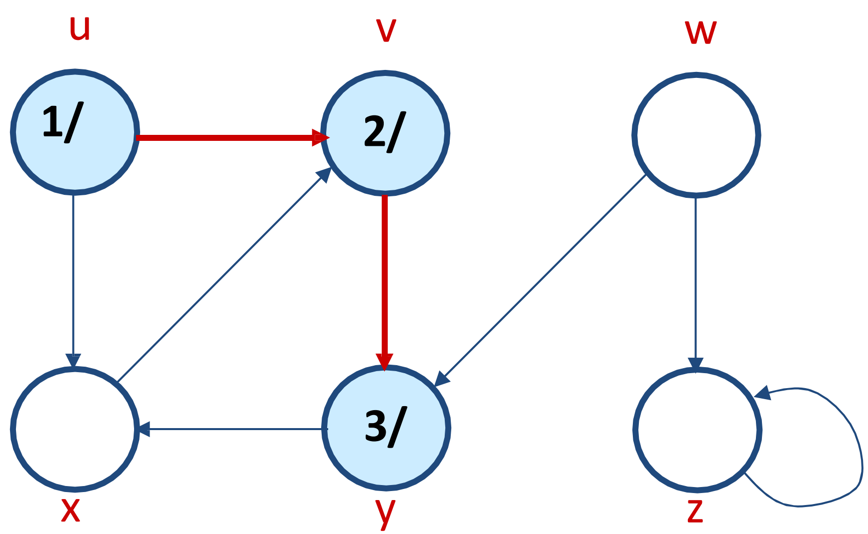
Example (DFS)



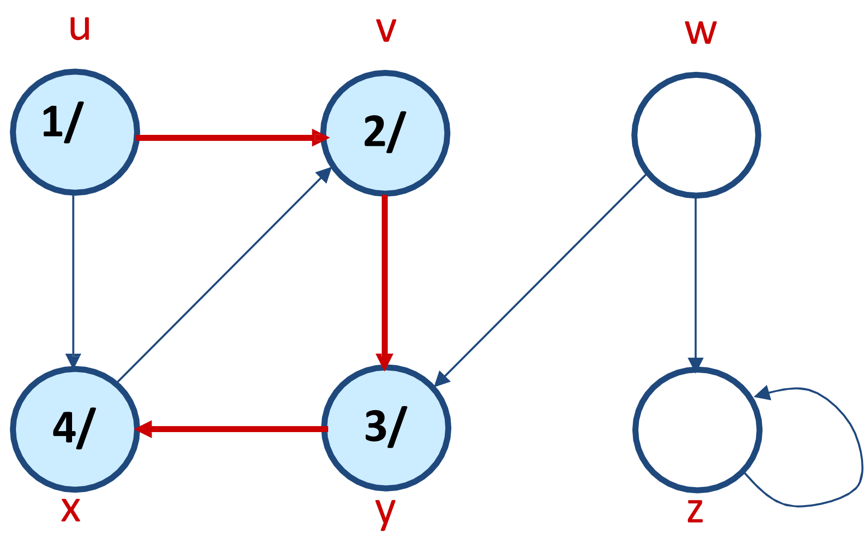
Example (DFS)



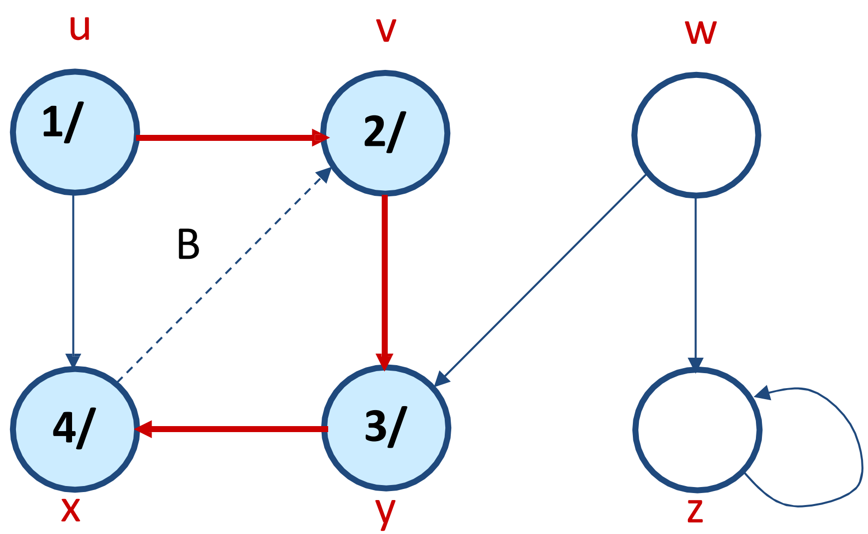
Example (DFS)



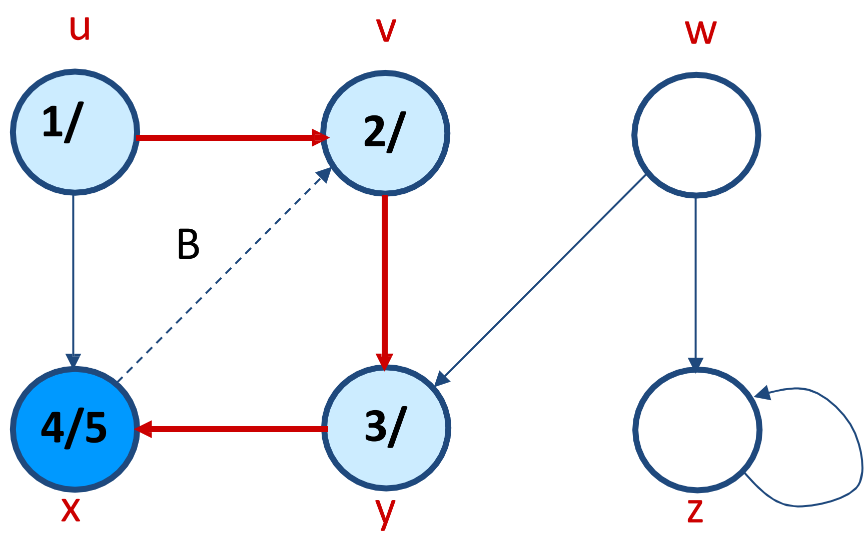
Example (DFS)



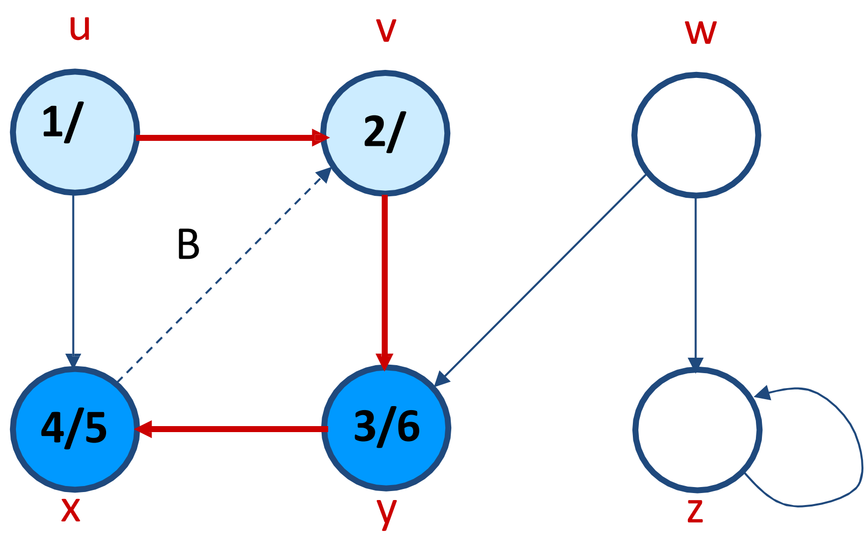
Example (DFS)



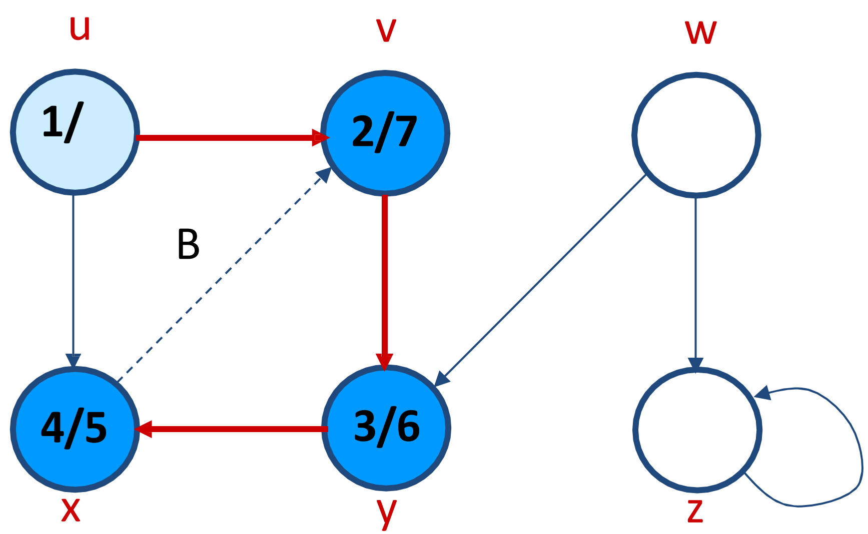
Example (DFS)



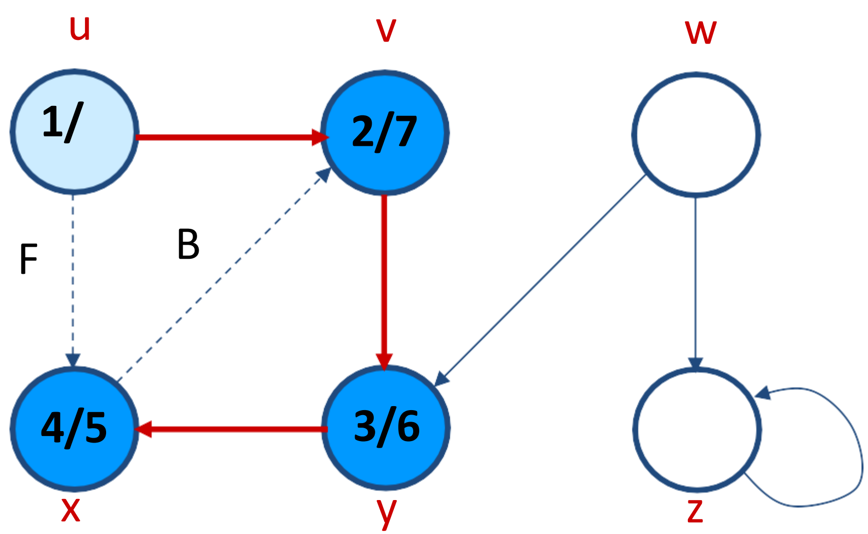
Example (DFS)



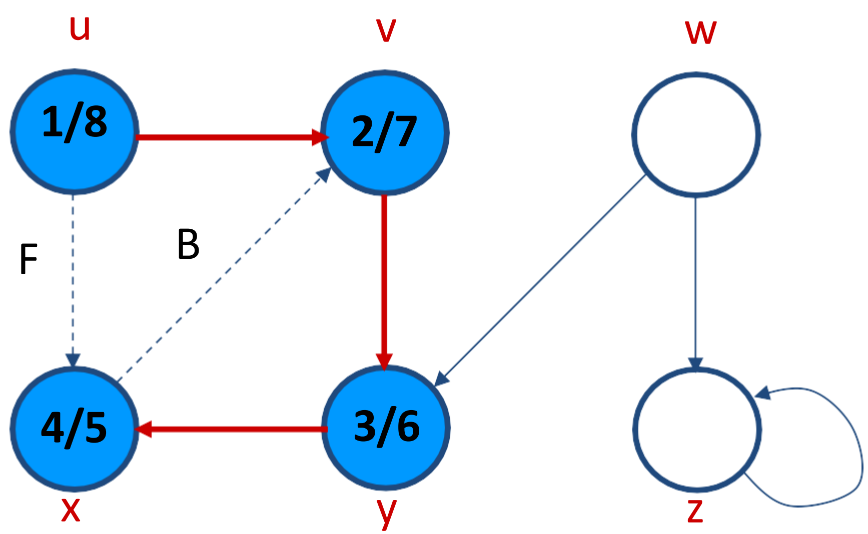
Example (DFS)



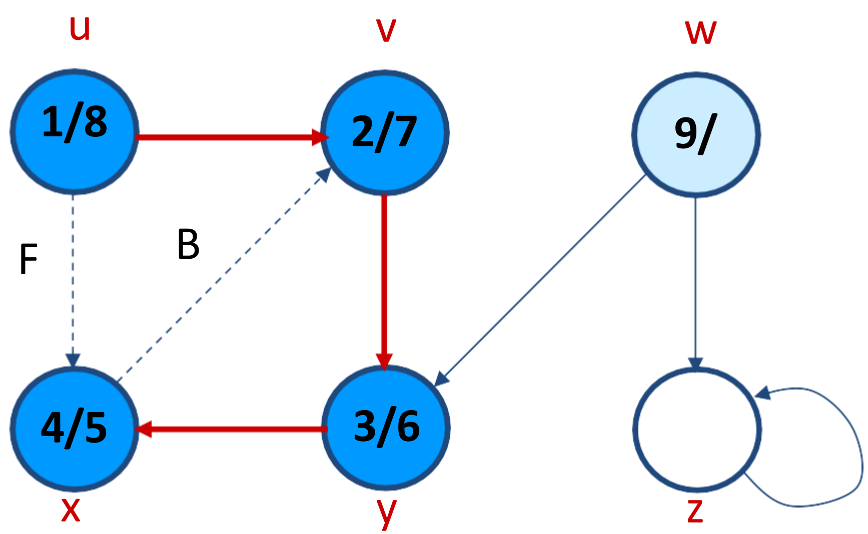
Example (DFS)



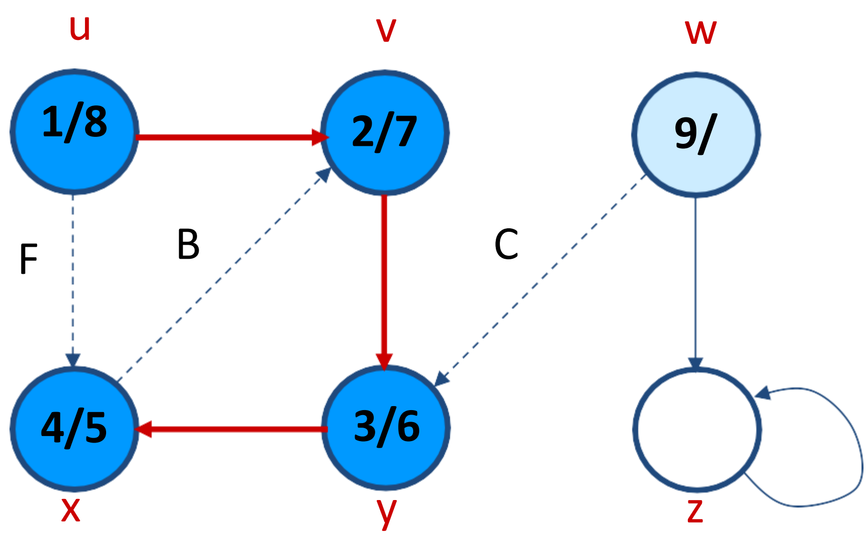
Example (DFS)



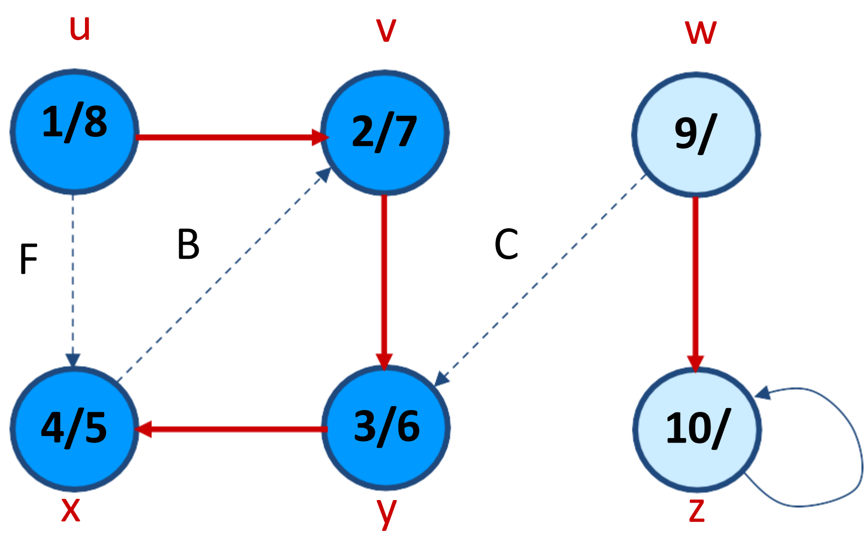
Example (DFS)



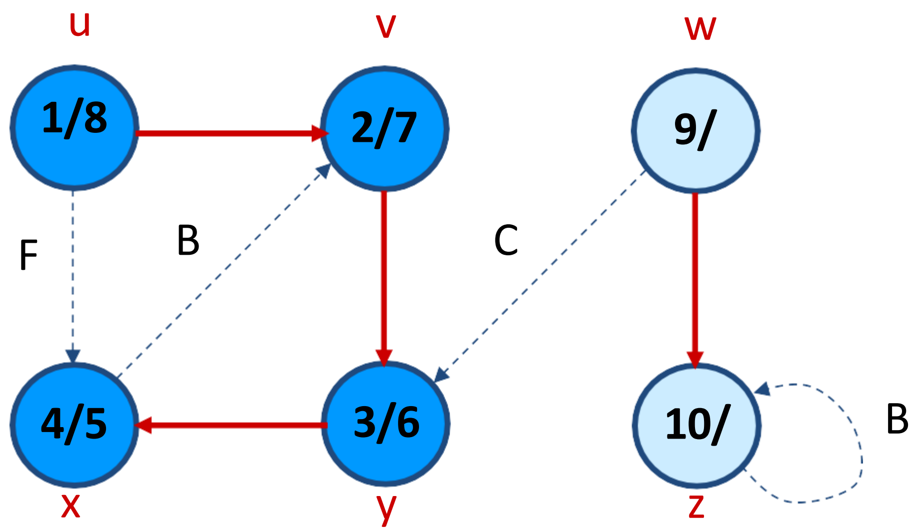
Example (DFS)



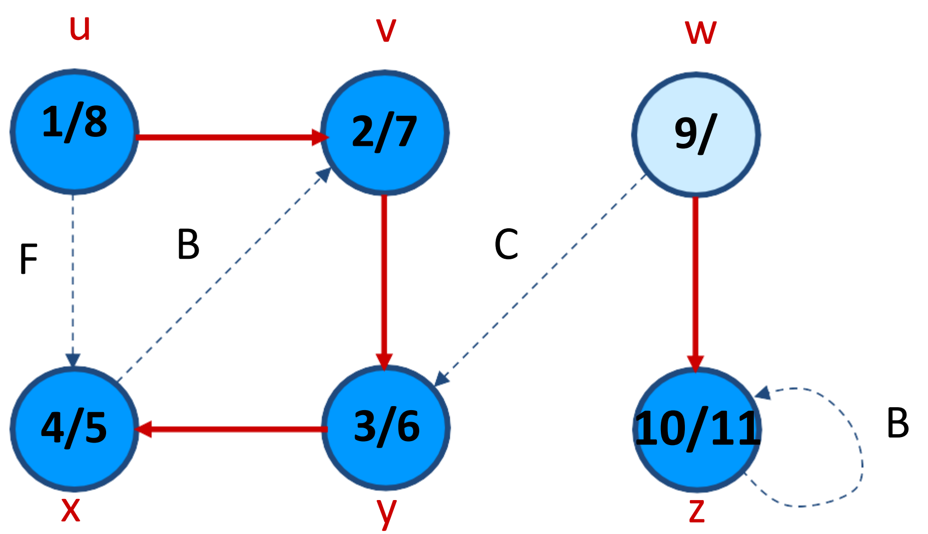
Example (DFS)



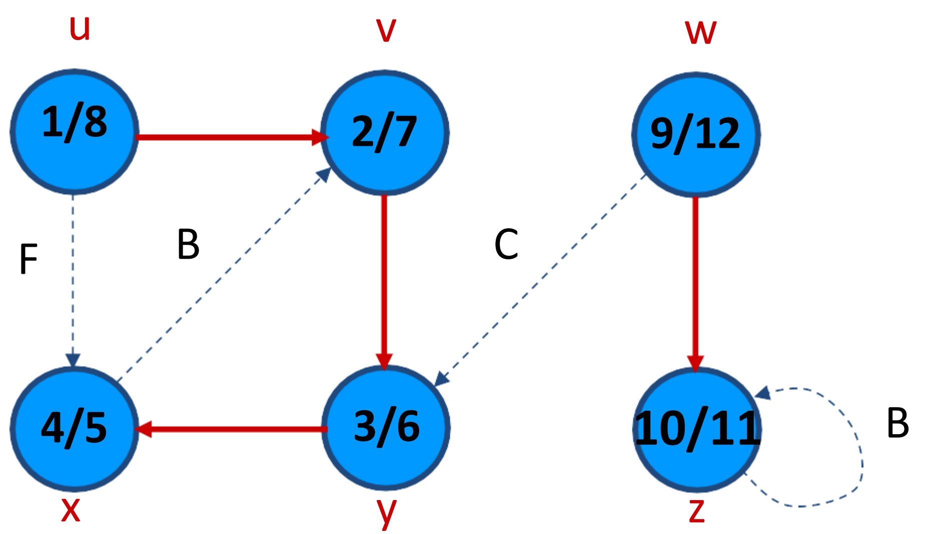
Example (DFS)



Example (DFS)



Example (DFS)!!!

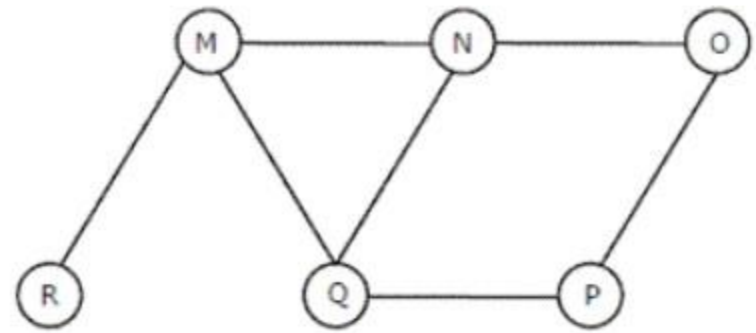


Recursive DFS Algorithm

```
Traverse( )
    for all nodes X
        visited[X]= False
    DFS( 1st node )
DFS( X )
    visited[X] = True
    for each successor Y of X
        if (visited[Y] = False)
            DFS( Y )
```

Quiz 1

- The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is

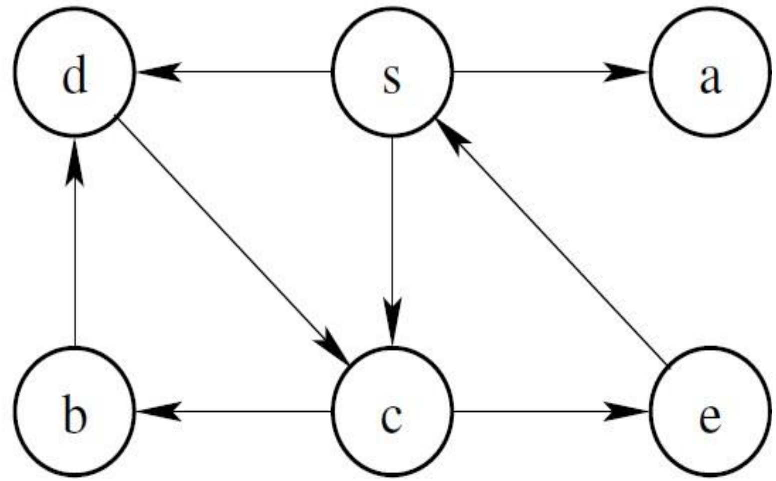


A	MNOPQR
B	NQMPOR
C	QMNPRO
D	QMNPOR

Quiz 2

Give the visited node order for each type of graph search, starting with s, given the following adjacency lists and accompanying figure:

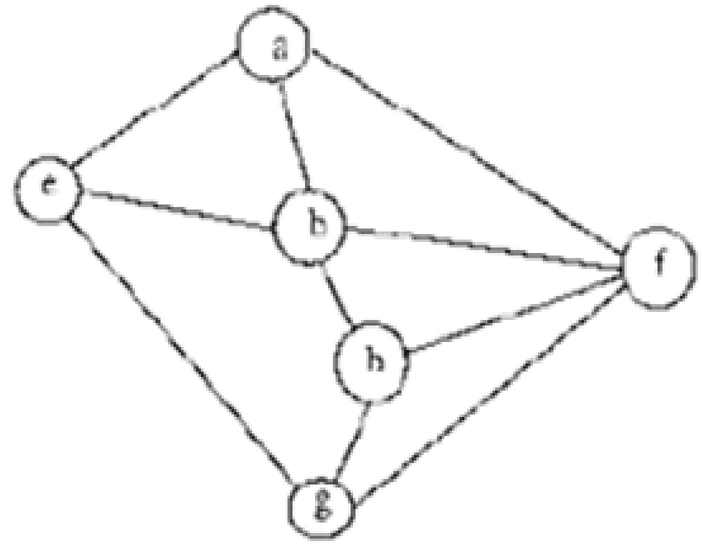
$adj(s) = [a, c, d]$,
$adj(a) = [\,]$,
$adj(c) = [e, b]$,
$adj(b) = [d]$,
$adj(d) = [c]$,
$adj(e) = [s]$.



- BFS?
- DFS?

Quiz 3

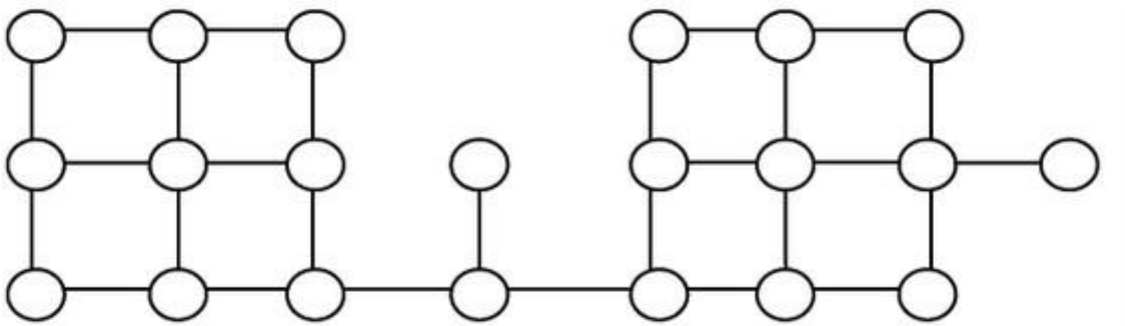
- Consider the following graph
  Among the following sequences
  I) a b e g h f
  II) a b f e h g
  III) a b f h g e
  IV) a f g h b e Which are <u>depth first traversals</u> of the above graph



A	I, II and IV only
B	I and IV only
C	II, III and IV only
D	I, III and IV only

Quiz 4

- Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is ?



A 17

B 18

C 19

D 20

Quiz 5

- Discuss the Order of BFS and DFS algorithms, with respect to V and E.

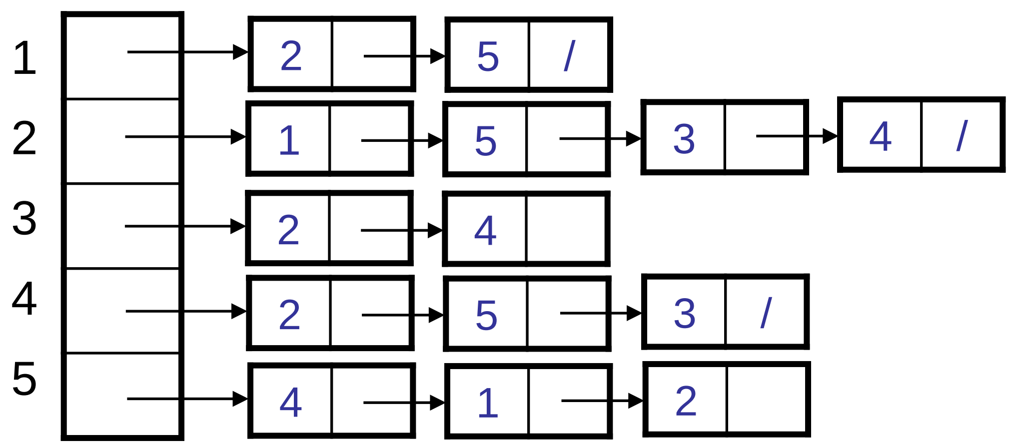
Exercises

Problem 1

- Given an adjacency-list representation, how long does it take to compute the **out-degree** of every vertex?
- How long does it take to compute the **in-degree** of every vertex?

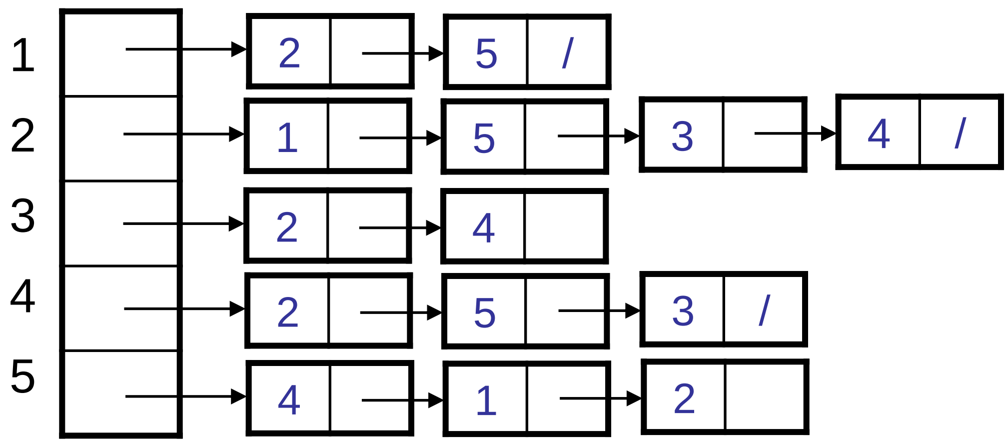
Problem 1

- Given an adjacency-list representation, how long does it take to compute the **out-degree** of every vertex?
  - For each vertex u, search Adj[u] → Θ(E)



Problem 1

- How long does it take to compute the **in-degree** of every vertex?
  - For each vertex u,
    search entire list of edges → $\Theta(VE)$



Problem 2

- The transpose of a graph $G=(V,E)$ is the graph $G^T=(V,E^T)$, where $E^T=\{(v,u) \epsilon V \times V: (u,v) \epsilon E\}$. Thus, G^T is G with all edges reversed.

(a) Describe an efficient algorithm for computing G^T from G, both for the adjacency-list and adjacency-matrix representations of G.

(b) Analyze the running time of each algorithm.

Problem 2 (cont'd)

Adjacency matrix

```
for (i=1; i<=V; i++)
    for(j=i+1; j<=V; j++)
        if(A[i][j] && !A[j][i]) {
            A[i][j]=0;
            A[j][i]=1;
        }
```

$O(V^2)$ complexity

	1	2	3	4	5
1	0	1	0	0	1
2	0	0	0	1	1
3	0	1	0	0	0
4	0	1	1	0	1
5	0	0	0	1	0

Problem 2 (cont'd)

Adjacency list

Allocate V list pointers for G^T (Adj'[]) ← O(V)

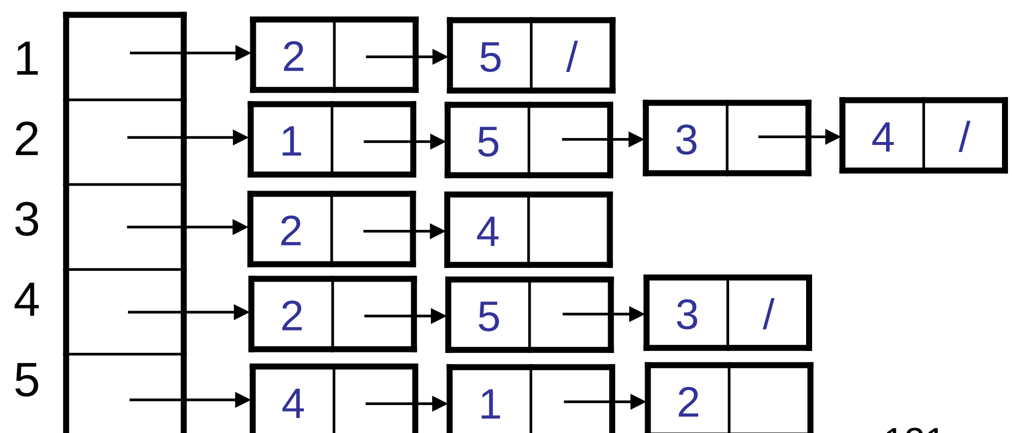
for(i=1; i<=V, i++)

    for every vertex v in Adj[i]

        add vertex i to Adj'[v]

← O(E)

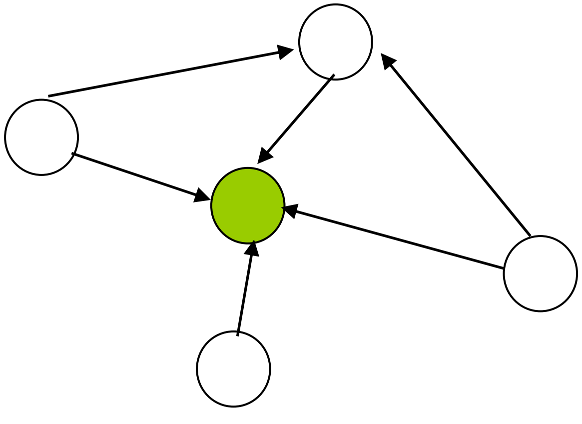
Total time: O(V+E)



Problem 3

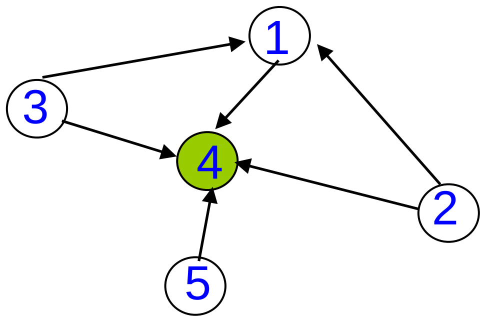
- When adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions.
- Show that determining whether a directed graph G contains a **universal sink** (a vertex of in-degree |V|-1 and out-degree 0) can be determined in time O(V).

Example



Problem 3 (cont.)

- Example



	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

Problem 3 (cont.)

- How many sinks could a graph have?
- How can we determine whether a given vertex **u** is a universal sink?
- How long would it take to determine whether a given vertex **u** is a **universal sink**?

Problem 3 (cont.)

- How many sinks could a graph have?
  - 0 or 1
- How can we determine whether a given vertex **u** is a universal sink?
  - The u-row must contain 0's only
  - The u-column must contain 1's only
  - A[u][u]=0
- How long would it take to determine whether a given vertex **u** is a **universal sink**?
  - O(V) time

Problem 3 (cont.)

```
IS-SINK(A, k)
let A be |V| × |V|
for j ← 1 to |V|                    ▷ Check for a 1 in row k
    do if a_kj = 1
          then return FALSE
for i ← 1 to |V|                    ▷ Check for an off-diagonal 0 in column k
    do if a_ik = 0 and i ≠ k
          then return FALSE
return TRUE
```

Problem 3 (cont.)

- How long would it take to determine whether a given graph contains a universal sink if you were to check every single vertex in the graph?
- **- $O(V^2)$**
- Can you come up with a **$O(V)$** algorithm?

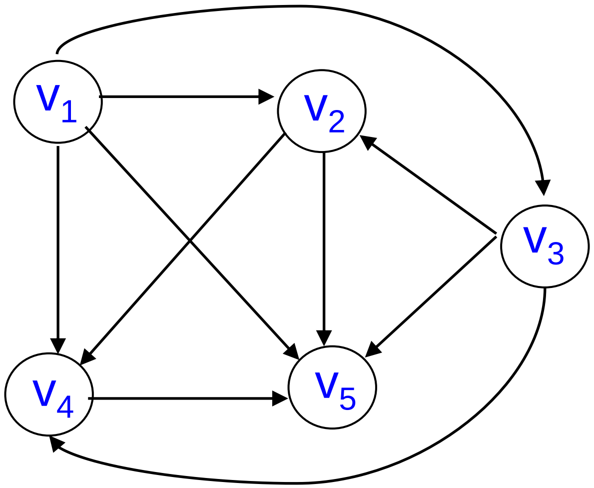
Problem 3 (cont.)

- Observations
  - If A[u][v]=1, then u cannot be a universal sink
  - If A[u][v]=0, then v cannot be a universal sink

UNIVERSAL-SINK(A)

let A be $|V| \times |V|$
$i \leftarrow j \leftarrow 1$
while $i \leq |V|$ and $j \leq |V|$
    do if $a_{ij} = 1$
        then $i \leftarrow i + 1$
        else $j \leftarrow j + 1$
$s \leftarrow 0$
if $i > |V|$
  then return "there is no universal sink"
elseif IS-SINK(A, i) = FALSE
  then return "there is no universal sink"
else return i "is a universal sink"

Problem 3 (cont.)



$$
\begin{array}{c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\left(\begin{array}{ccccc}
\mathbf{0} & \mathbf{1} & 1 & 1 & 1 \\
0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & 1 \\
0 & 1 & 0 & \mathbf{1} & 1 \\
0 & 0 & 0 & \mathbf{0} & \mathbf{1} \\
0 & 0 & 0 & 0 & \mathbf{0}
\end{array}\right)
\end{array}
\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
$$

- Loop terminates when **i > |V|** or **j > |V|**
- Upon termination, the only vertex that could be a sink is **i**
  - If i > |V|, there is no sink
  - If i < |V|, then j>|V|
    - * vertices k where 1 ≤ k < i can not be sinks
    - * vertices k where i < k ≤ |V| can not be sinks