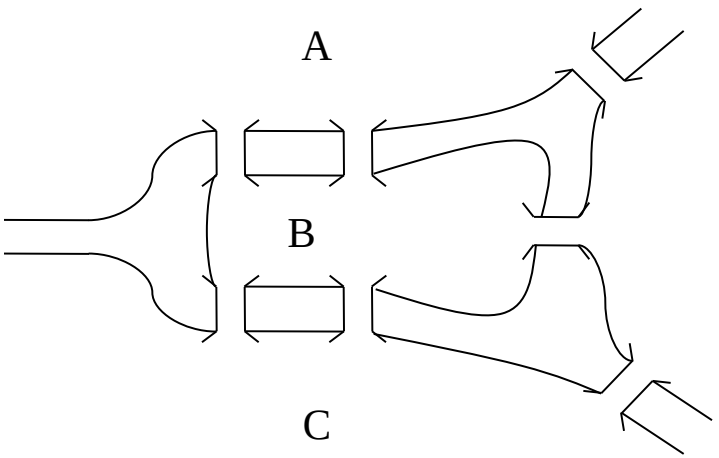
Data Structures and Algorithms W11

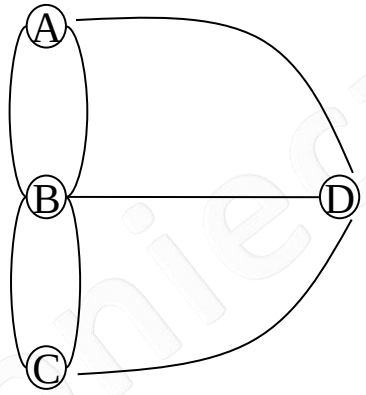
Graph Theory part.1
Introduction – definition, reprezentation
Connected Components
Bread-first and Depth-first search

Syllabus

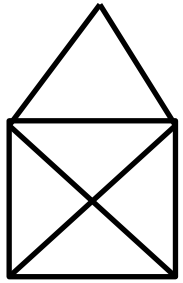
- Introduction into graph theory
  - Definitions: graph, digraph,…
  - Other definitions: path, cycle etc.
- Representation in computer memory
  - Adjacency matrix
  - Adjacency list
  - Incidence matrix
- Algorithm for connected components
- Breadth-first search (BFS)
- Depth-first search (DFS)

Seven Bridges of Königsberg (1736)





Degree(A) = 3
Degree(B) = 5
Degree(C) = 3
Degree(D) = 3



- An ***Euler tour*** of a graph is a cycle that traverses each edge of the graph exactly once, although it may visit a vertex more than once

A graph has an Euler tour if degree(*v*) is even for each vertex *v*.

- An **Euler walk** - in an graph is a path that uses each edge exactly once

An Euler walk exist if at most two vertices in the graph are of odd degree .

Graph - definition

- **A graph** G (**undirected graph**) is an ordered pair:

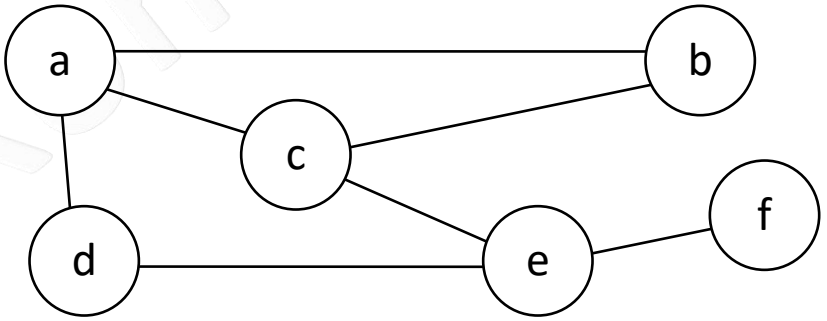
  G=(V,E)

  where:

  V – is a finite set of points called ***vertices***

  E – is a finite set of ***edges,*** each of which connect a pair of vertices. The vertices belonging to the edge are called its **endpoints** *of the edge*.

  $e \in E$ is **unordered** pair (u, v), where $u, v \in V$ (vertices *u* and *v* ***are connected***)



- An example:

  $G_1=(V_1,E_1)$

  $V_1=\{a,b,c,d,e,f\}$

  $E_1=\{(a,c),(b,a),(a,d),(d,e),(c,e),(e,f),(c,b)\}$

- In theory, there are also **infinite** graphs, but we will not deal with them in the lecture

Digraph (directed graph) - definition

- **Digraph (directed graph)** G is an ordered pair:

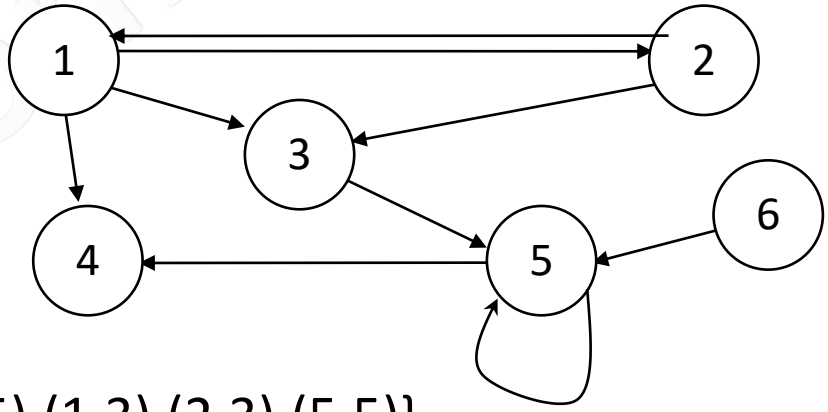
  G=(V,A)

  where :

  V – a finite set of points called ***vertices***

  A – a finite set of ***directed edges (arcs)***

  $e \in A$ is an **ordered** pair (u, v), where $u, v \in V$ (there is connection **from** *u* **to** *v)*), u is a **head**, v is a **tail**



- An example:

  $G_2=(V_2,E_2)$

  $V_2=\{1,2,3,4,5,6\}$

  $E_2=\{(1,4),(5,4),(1,2),(6,5),(2,1),(3,5),(1,3),(2,3),(5,5)\}$

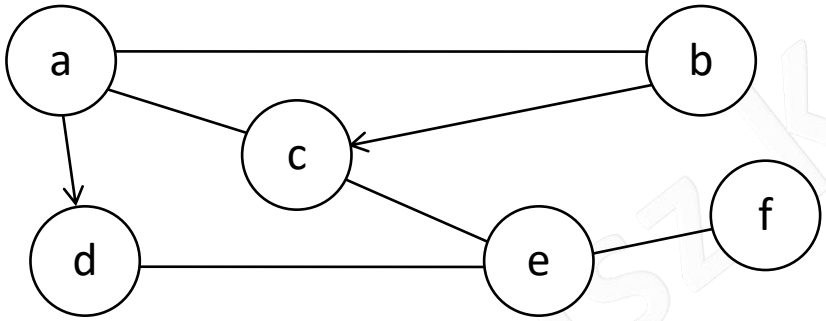
Others graphs

- **Mixed** graph:

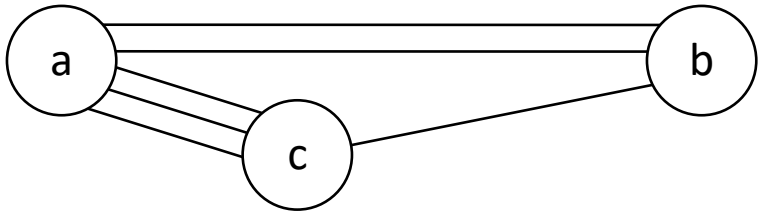
  G=(V,E,A)

  Where:

  V,E,A – like in previous definitions
- **Multihgraph** (**pseudograph**), where can be, that two or more edges connect the same two vertices



Mixed graph



Multigraph

Definitions 1/3

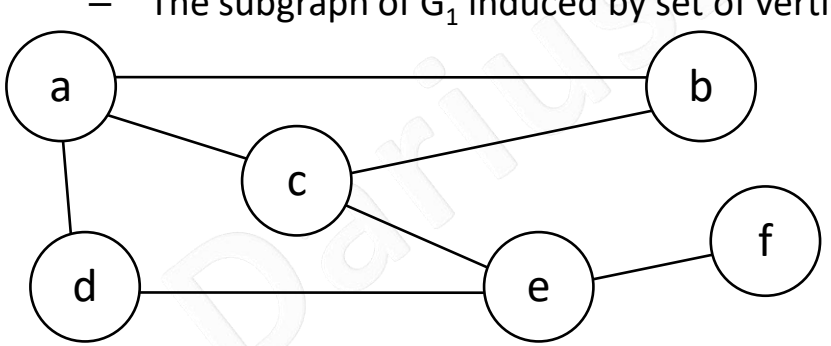
- **General remark:** how many books - so many definitions!

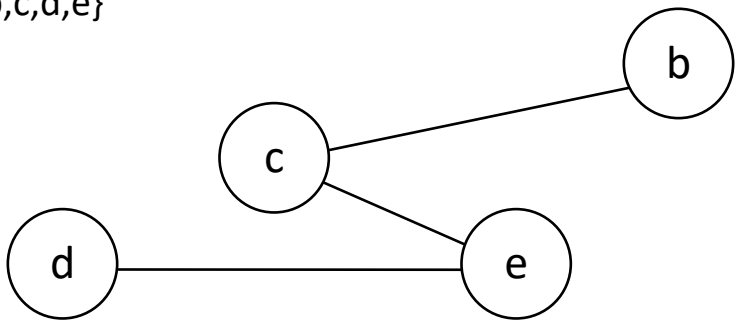
- An **edge** is **adjacent** to its **endpoints**.
- **Two edge** are **adjacent** if they have one the same endpoint.
  - In directed graph tow edges have to be adjacent to an endpoint which is a head for one edge and a tail for another edge.
- **Two vertices** u, v are **adjacent** if there is an edge (u,v)
- A **walk** is an alternating sequence of vertices and edges, starting and ending at a vertex, in which each edge is adjacent in the sequence to its two endpoints.
- A **Trail** - A walk without repeated edges (but with vertex repetition allowed)
- A **path** from vertex *v* to vertex *u* is a trail sequence $(v_0, v_1, v_2, ..., v_k)$ of vertices where:

  $v_0=v$, $v_k=u$ and $(v_i,v_{i+1})\in E$ for $i = 0, 1, \ldots, k-1$
  - Vertices v and u are called **endpoints** *of the path*
  - E.a. for the graph G_1 a path is p_1=(a,c,e,d,a,b)
- The **length** of a path is the number of edges in the path
  - The lenght of p_1 is equal 5
- A **simple path** - a walk with no repetitions of vertices or edges (beside maybe the first and the last vertex)
  - A path p_1 is not simple, a simple path is jest p_2=(b,c,e,d)

Definitions 2/3

- A **cycle** it is closed path where first and last vertex are the same.
- A **simple cycle** – Cycle without repetition of edges.
- A **loop/self-loop** is an edge both of whose endpoints are the same vertex.
  - It forms a cycle of length 1. These are not allowed in simple graphs.
- We say that the graph is **acyclic** when it contains no cycle.
- A graph is **connected** if every pair of vertices is connected by a path
  - For the undirected graph, the definition is used directly
  - For a directed graph it is transformed into a undirected *underlying graph* obtained by replacing all directed edges of the graph with undirected edges.
- A graph $G_l = (V_l , E_l)$ is a **subgraph** of $G = (V, E)$ if $V_l \subseteq V$ and $E_l \subseteq E$
- The subgraph of G **induced** by $V_l \subseteq V$ is the graph $G_l = (V_l , E_l)$, where $E_l = \{ (u, v) \in E \mid u, v \in V_l\}$
  - The subgraph of G_1 induced by set of vertices {b,c,d,e}





Definitions 3/3

- The **degree** of a vertex *v* in a graph is its number of incident edges deg(*v*)
  - In a directed graph, one may distinguish the in-degree (number of incoming edges) degIn(*v*) and out-degree (number of outgoing edges) degOut(*v*)
- An **isolated vertex** of a graph is a vertex whose degree is zero.

- A **complete graph** is one in which every two vertices are adjacent: all edges that could exist are present. The undirected complete graph has exactly $|E|=|V|*(|V|-1)/2=\Theta(n^2)$ edges.
- **Density of the graph** – the ratio of the number of edges to the largest possible number of edges.
  - We say that a graph is **dense**, when the number of its edges is of the order of the number of edges of the full graph
  - We say that a graph is **sparse** otherwise
    - Most often, we assume that the number of edges in a sparse graph $|E|=O(|V|)$
- A **clique** – a complete subgraph

- A **forest** – is an acyclic graph
- A **tree is** a connected acyclic graph
- If $G = (V, E)$ is a tree, then $|E| = |V| - 1$

Problems of graph theory

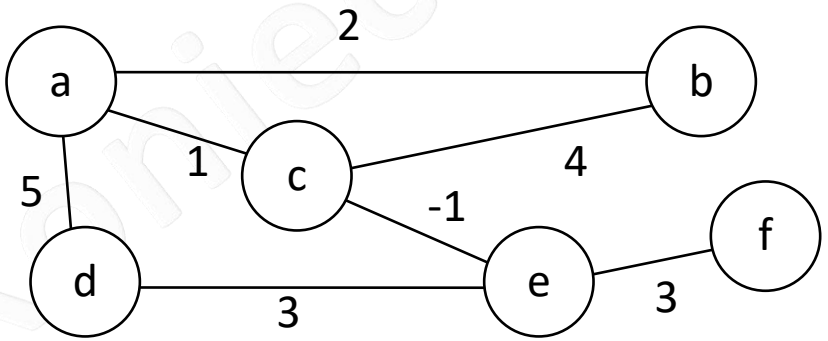
- **Graph coloring**: assigns different colors to the endpoints of each edge
  - Minimal number of colors

- **Graph isomorphism***:* a one-to-one incidence preserving correspondence of the vertices and edges of one graph to the vertices and edges of another graph.

- **Planar graph** – A planar graph is a graph that has an embedding onto the Euclidean plane.
  - We can draw such a graph without crossing edges
- **Eulerian graph** – An Eulerian graph is a graph that has an Eulerian **circuit** (a closed walk that uses every edge exactly once)
- **Hamiltonian graph** – it has a path which cover all of the vertices in the graph exactly once.

Weighted graph

- A **weighted** graph (**digraph**) *G* is a triple (*V*, *E*, *w*), where V – set of vertices, E – set of edges, while *w*: E $\rightarrow \Re$ z is a real-valued function defined on E.
  - Instead of writing w((v_1,v_2)) we will use w(v_1,v_2)
- The **weight of graph** is the sum of the weight of its edges
- The **weight of a path** is the sum of the weight of its edges



- example:
  $G_3=(V_3,E_3,w)$
  $V_3=\{a,b,c,d,e,f\}$
  $E_3=\{(a,c),(b,a),(a,d),(d,e),(c,e),(e,f),(c,b)\}$

$e \in E_3$	(a,c)	(b,a)	(a,d)	(d,e)	(c,e)	(e,f)	(c,b)
w(e)	1	2	5	3	-1	3	4

- The weight of graph G_3 is 17
- The weight of path p_1=(a,c,e,d,a,b) is 10

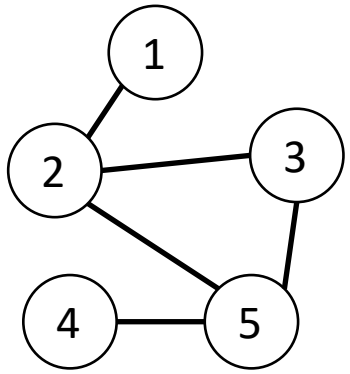
Why/where

- With graphs we can express :
  - nets
    - computer nets
    - traffic nets
    - urban nets
    - … nets
  - person connections
  - technology processes
  - work flow
  - …

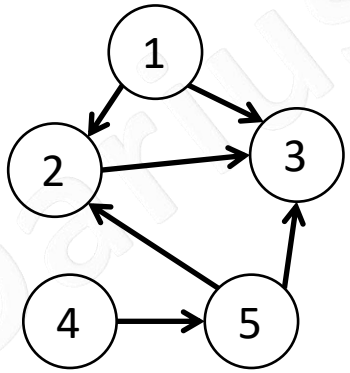
Graph's representation – adjacency matrix

- The **adjacency matrix** of graph G = (V, E) is an $n \times n$ array (n=|V|)$A = (a_{i,j})$, which elements are defined as follows :

$$a_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$



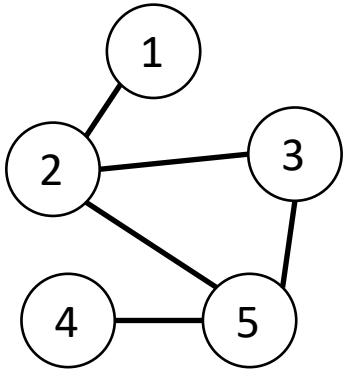
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

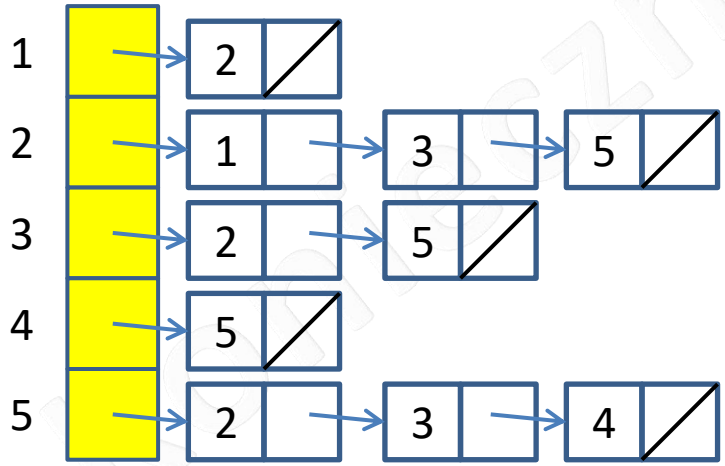


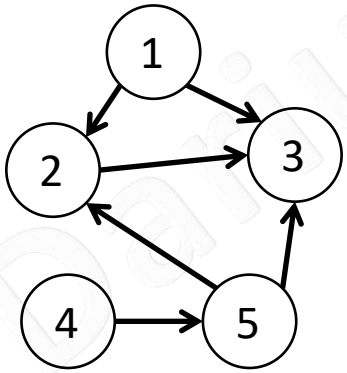
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

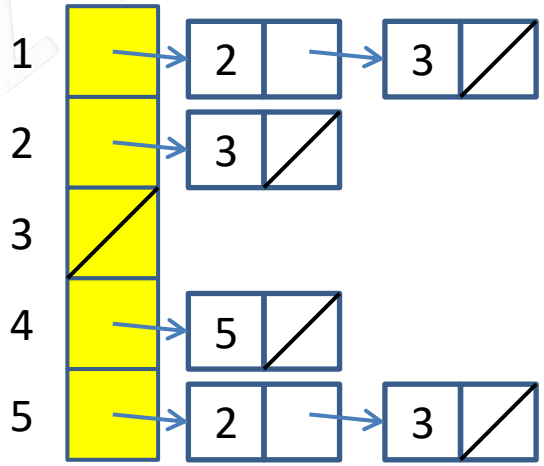
Graph's representation – adjacency list

- The **adjacency list** of graph $G = (V, E)$ is an array $Adj[1..|V|]$ of lists. For each $v \in V$, $Adj[v]$ is a linked list of all vertices adjacent to v.



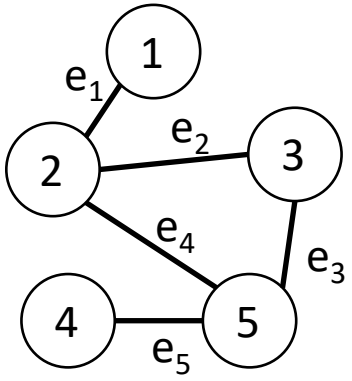






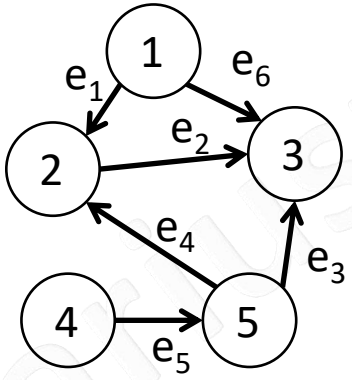
Graph's representation – incidence matrix

- Less popular, only for part of algorithms is useable
- It needs many memory O(|V|*|E|)
- Every column descibe one edge: value 1 or -1 are in rows where are endpoints for the egde



edges

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$



edges

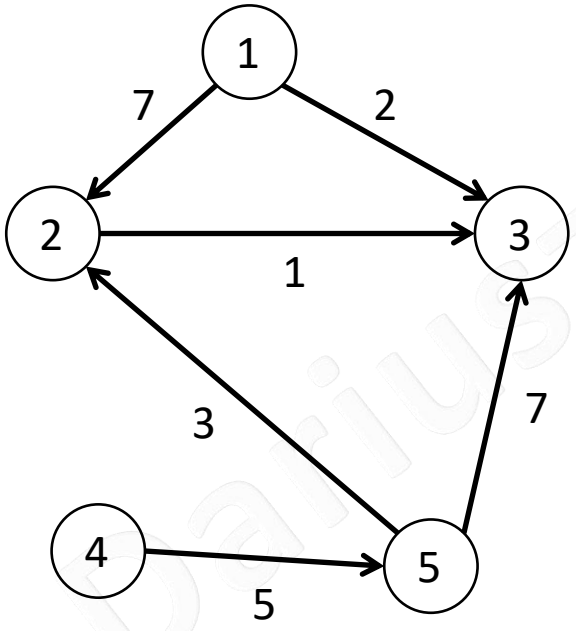
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 \end{bmatrix} \end{array}$$

- There are many other graph representations in the computer memory adapted to **specific** graphs, situations, algorithms.

Adjacency matrix for weighted graph

- The **adjacency matrix** for **weighted** graph G = (V, E, W) is an n × n array (n=|V|)$A = (a_{i,j})$, which elements are defined as follows :

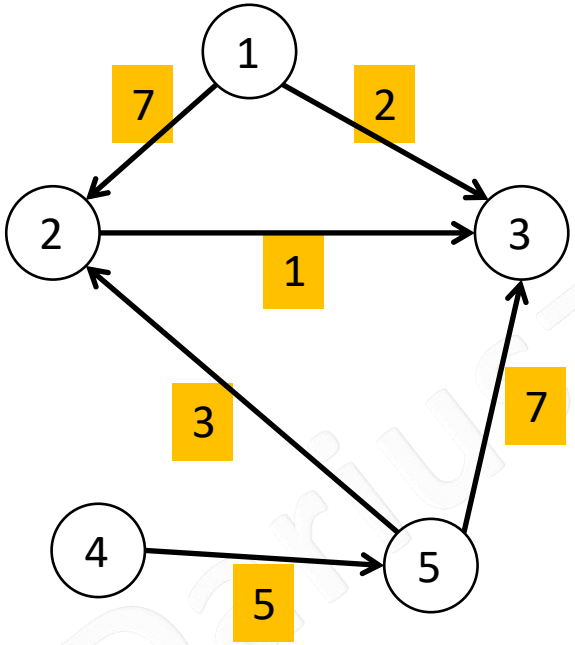
$$a_{i,j} = \begin{cases} w(v_i, v_j) & if \ (v_i, v_j) \in E \\ 0 & if \quad i = j \\ \infty & otherwise \end{cases}$$

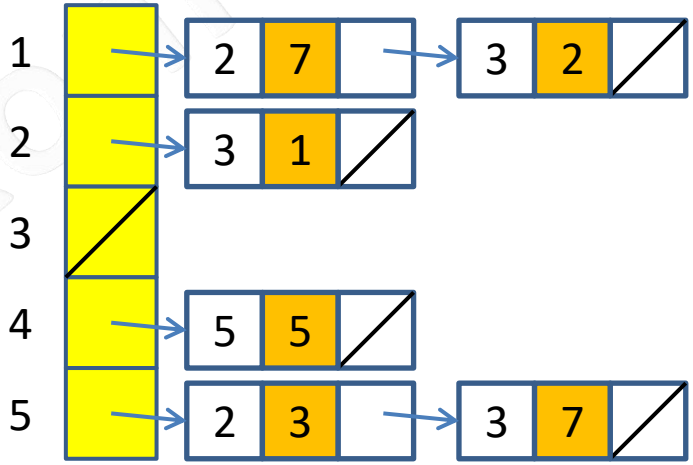


$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 7 & 2 & \infty & \infty \\ \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & 5 \\ \infty & 3 & 7 & \infty & 0 \end{bmatrix} \end{array}$$

Adjacency list for weighted graph

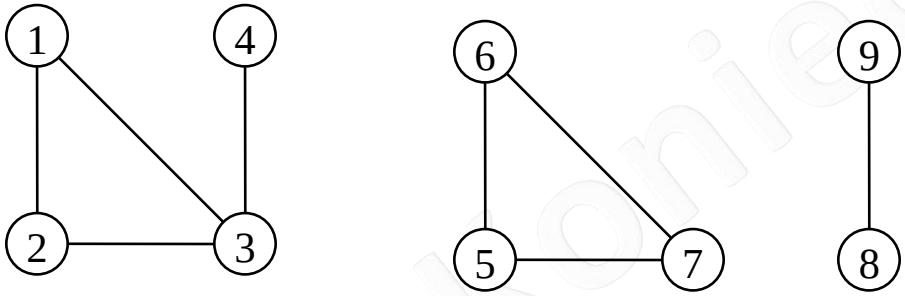
- Simple modification of the adjacency list to accommodate weighted graph. Every element stores a weight of a proper egde.





Connected components

- In an undirected graph, a **connected component** or **component** is a maximal connected subgraph.
- Two vertices are in the same connected component if and only if **there exists a path** between them.
- Problem: division of the graph into connected components.



A graph with three connected components.

```
ConnectedComponents(G,w)
{ 1} for each vertex u∈V[G] do
{ 2}     MakeSet(u)
{ 3} for each edge (u,v)∈E do
{ 4}     if FindSet(u) ≠ FindSet(v) then
{ 5}             Union(u,v)
```

Complexity $\Theta(|E|+|V|)$

- After finishing the algorithm, the representative of the set determines a connected component.

Connected components - example

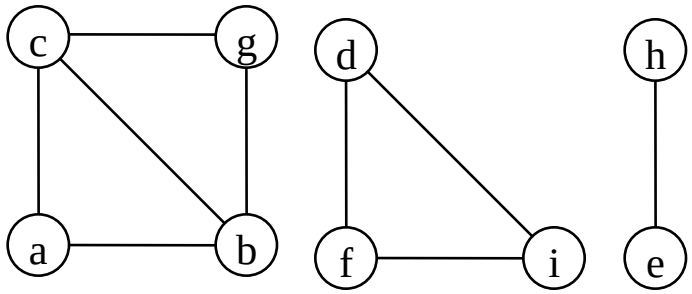
V = { a, b, c, d, e, f, g, h, i}

E = { (a,c), (d,f), (d,i), (f,i), (e,h), (b,g), (a,b), (b,c), (c,g) }

{a}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c,b,g}
{b}	{b}	{b}	{b}	{b}	{b}	{b,g}	
{c}							
{d}	{d}	{d,f}	{d,f,i}	{d,f,i}	{d,f,i}	{d,f,i}	{d,f,i}
{e}	{e}	{e}	{e}	{e}	{e,h}	{e,h}	{e,h}
{f}	{f}						
{g}	{g}	{g}	{g}	{g}	{g}		
{h}	{h}	{h}	{h}	{h}			
{i}	{i}	{i}					

Complexity (adjacent lists): O(|V|+|E|)

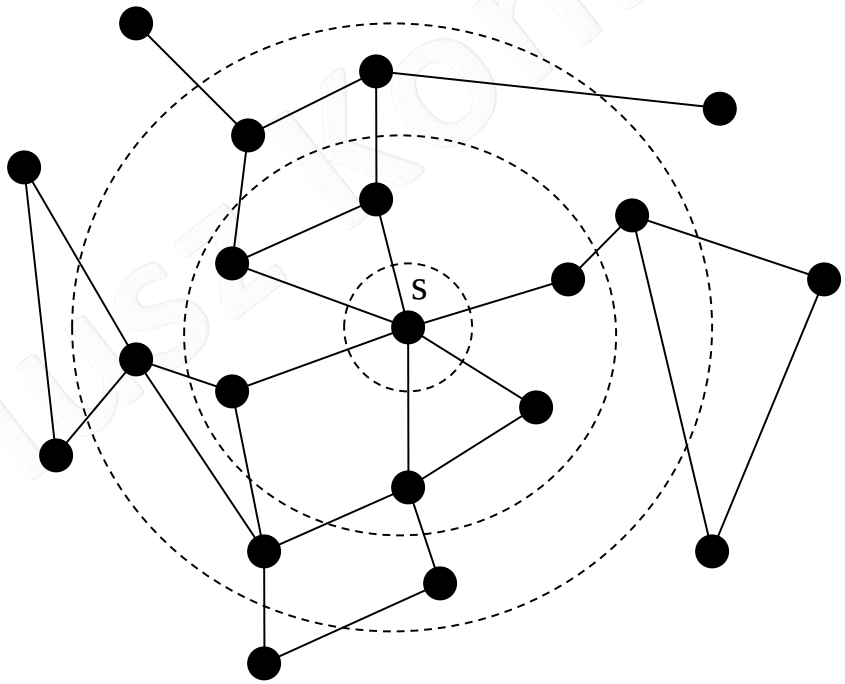
Complexity (adjacent matrix): $O(|V|^2)$



Breadth-first search

Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

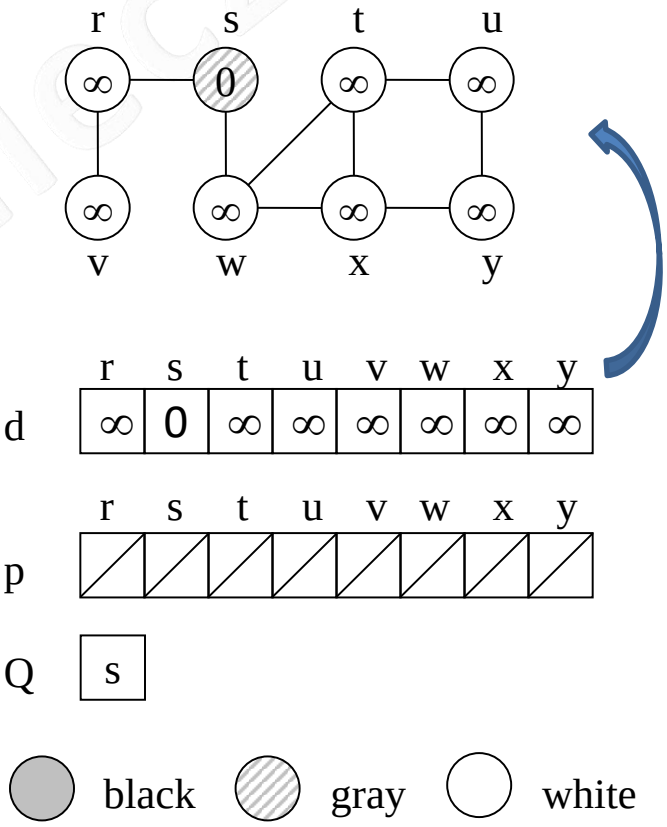
Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.



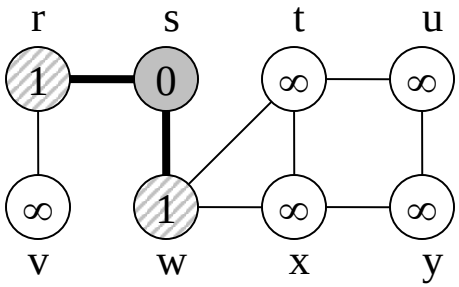
BFS - code

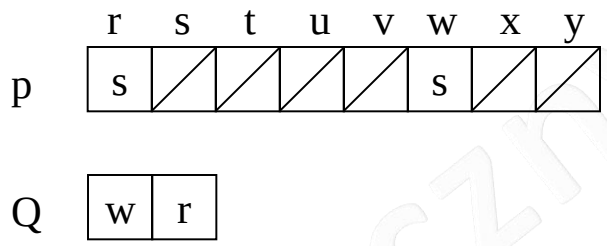
- Breadth-first search colors each vertex white, gray, or black (*color*)
- Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*
- Predecessor of *u* is stored in the variable p[*u*]
- The distance from the source *s* to vertex *u* computed by the algorithm is stored in *d*[*u*].

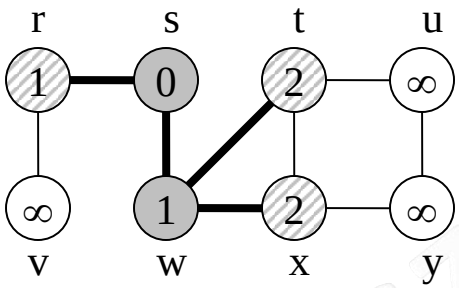
```
      BFS(G,s)
{ 1}  for every vertex u∈V[G]-{s}
{ 2}     do color[u]:=WHITE
{ 3}           d[u]:= ∞
{ 4}           p[u] := NIL
{ 5}  color[s]:=GREY
{ 6}  d[s]:=0
{ 7}  p[s]:=NIL
{ 8}  Enqueue(Q,s)
{ 9}  while  not Empty(Q)
{10}     do u:=Dequeue(Q)
{11}           for every v∈Adj[u]
{12}               do if color[v]=WHITE
{13}                  then color[v]:=GREY
{14}                       d[v]:=d[u] + 1
{15}                       p[v]:=u
{16}                       Enqueue(Q, v)
{17}           color[u]:=BLACK
```

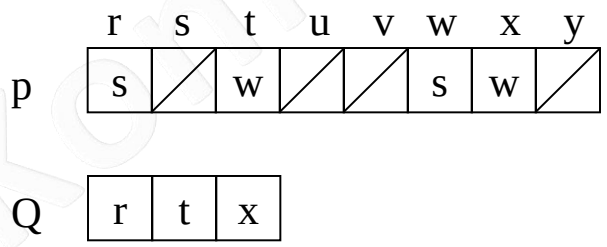


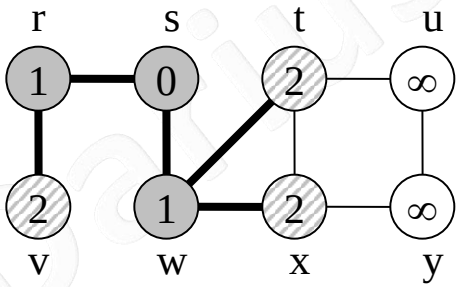
BFS – example 1/3

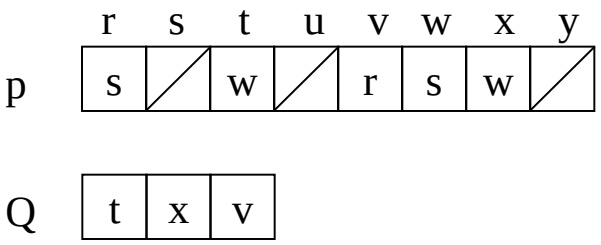




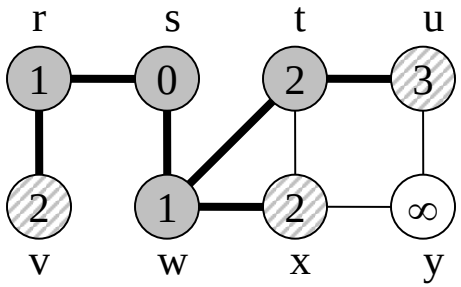






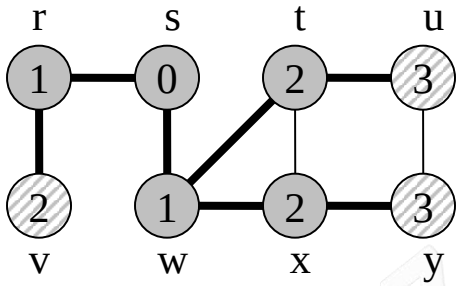


BFS – example 2/3



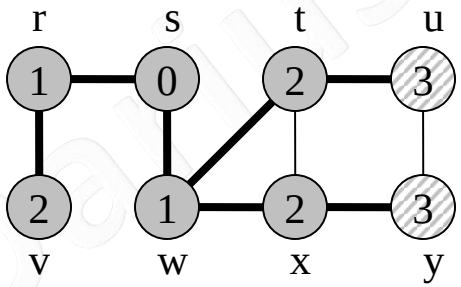
	r	s	t	u	v	w	x	y
p	s	/	w	t	r	s	w	/

Q			
	x	v	u



	r	s	t	u	v	w	x	y
p	s	/	w	t	r	s	w	x

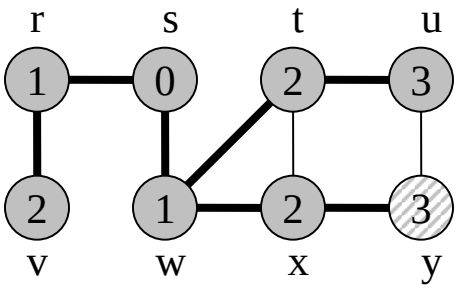
Q			
	v	u	y

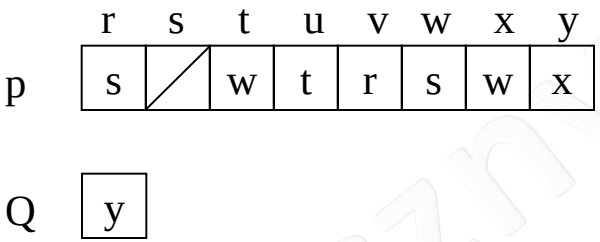


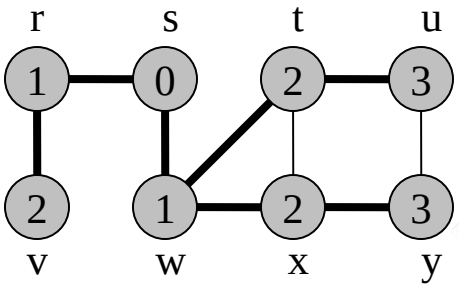
	r	s	t	u	v	w	x	y
p	s	/	w	t	r	s	w	x

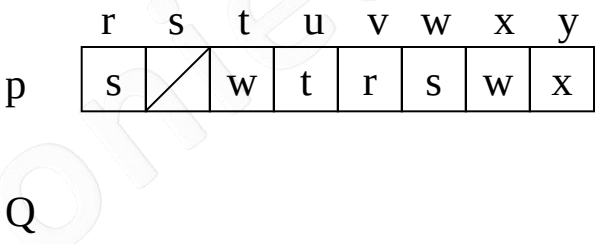
Q		
	u	y

BFS – example 3/3










```
    PrintPath(G,s,v)
{ 1} if v = s
{ 2}    then print s
{ 3}    else if p[v] =  NIL
{ 4}              then print „no path from” s „to” v „exists”
{ 5}              else PrintPath(G, s, p[v])
{ 6}                   print v
```

Complexity (adjacent lists): O(|V|+|E|)

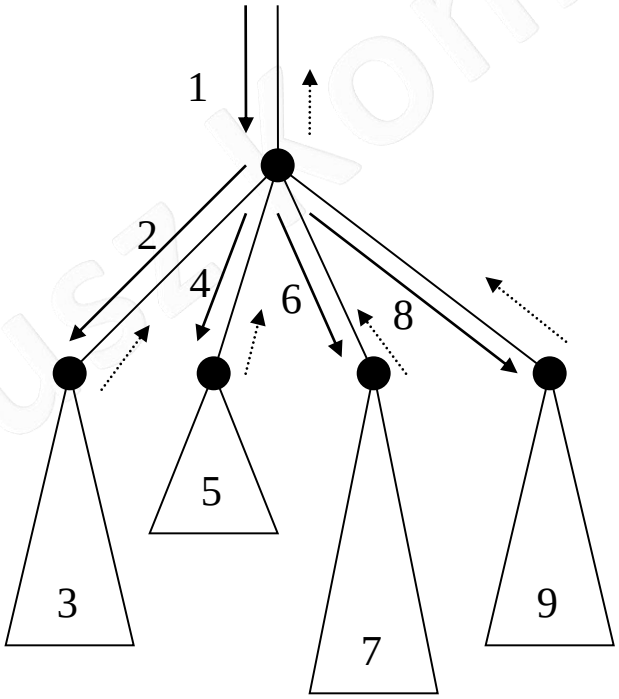
Complexity (adjacent matrix): $O(|V|^2)$

BFS - usage

- It creates tree of minimum paths (as number of edges)
- Can be a part of algorithm for another problem from graph theory.
  - In flow networks, searching for an extension path
- Idea used in solving problems in game theory (where the graph is built during computation)
  - Logic game – chess etc.

Depth-first search

- In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of *v*'s edges have been explored, the search "backtracks" to explore edges leaving the vertex from which *v* was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered
- Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex. Each vertex *v* has two timestamps: the first timestamp *t*[*v*] records when *v* is first discovered (and grayed), and the second timestamp *f* [*v*] records when the search finishes examining *v*'s adjacency list (and blackens *v*)

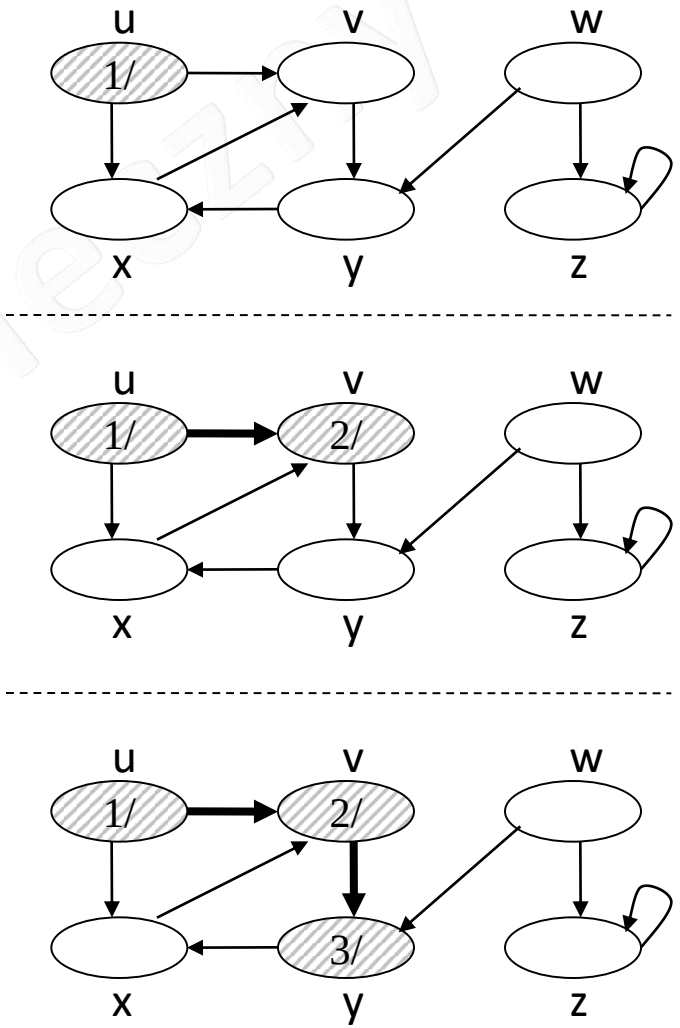


DFS - code

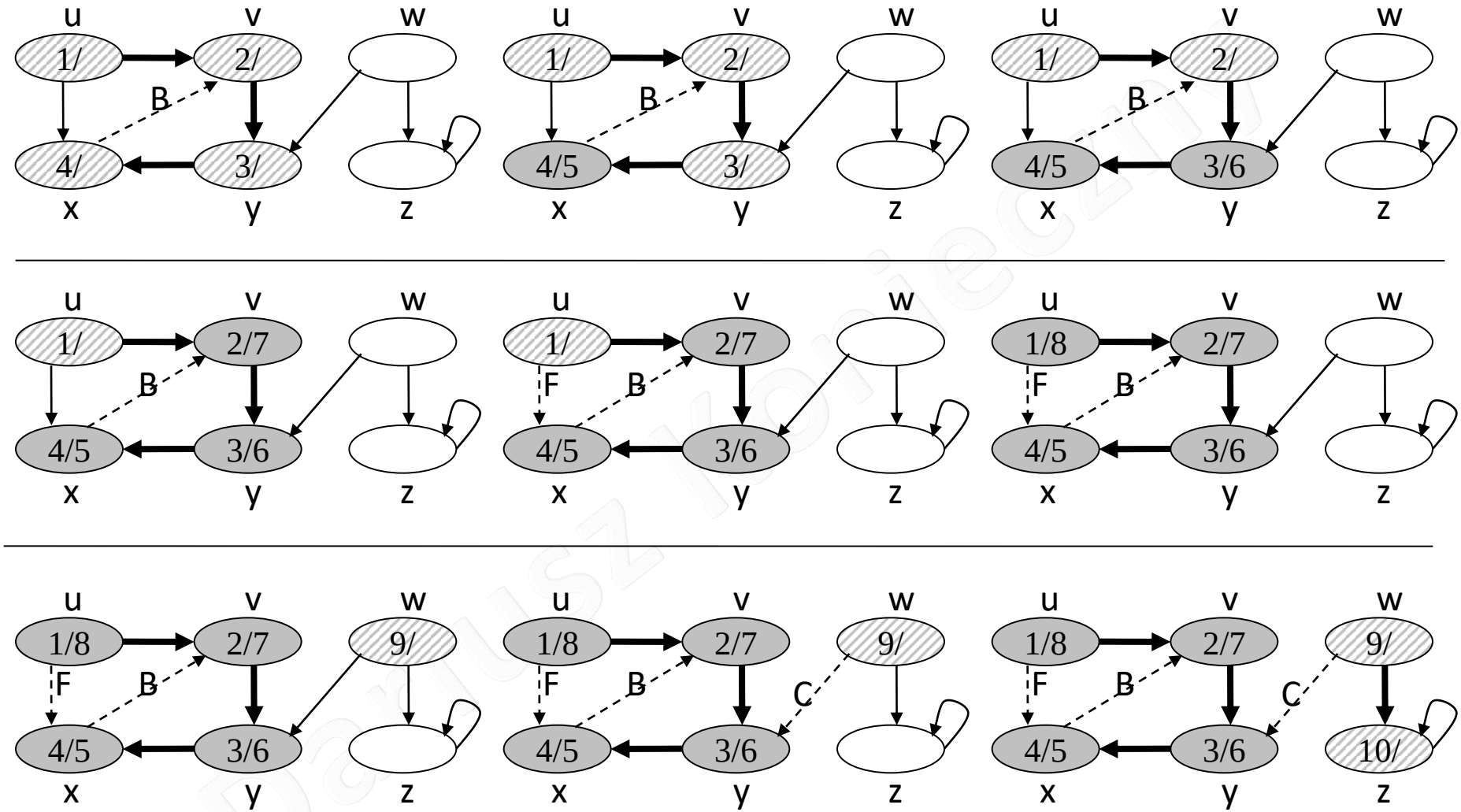
- Recursive version, „global” time in variable `time`

```
     DFS(G)
{ 1} for each vertex u∈V[G] do
{ 2}          color[u]:=WHITE
{ 3}          p[u] := NIL
{ 4} time:=0
{ 5} for each vertex u∈V[G] do
{ 6}          if color[u]=WHITE then
{ 7}              DFS_Visit(u)
```

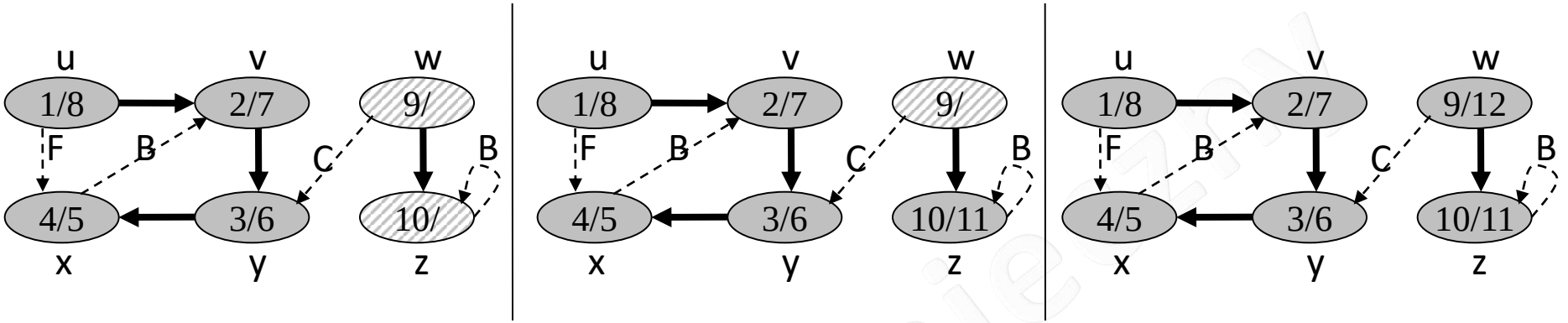
```
     DFS_VISIT(u)
{ 1} color[u]:=GREY
{ 2} time:=time+1
{ 3} t[u]:=time
{ 4} for each v∈Adj[u] do
{ 5}       if color[v]=WHITE then
{ 6}              p[v] := u
{ 7}              DFS_Visit(v)
{ 8} color[u]:=BLACK
{ 9} f[u] := time := time+1;
```



DFS – example 1/2



DFS – example 2/2



B –*back edge*
F – *forward edge*
C – *cross edge*

Complexity (adjacent lists): O(|V|+|E|)

Complexity (adjacent matrix): $O(|V|^2)$

DFS - usage

- A part of many other algorithms from graph theory
  - Searching bridges, or edges which divide a graph into dwo connected components
- Searching space of states fo a game, when BFS generates to big number of states, specially when the space generated is a tree:
  - DFS need only store information about a path from source to a vertex
  - Variants of DFS algorithm: min-max, alpha-beta, A* etc.