Graphs

Outline

- Definitions
- Graph implementation as data structure
- Visiting algorithms
- C implementations

Directed Graph

Directed graph G (also called digraph) is a pair (V,E), where

- V is the (finite) set of vertexes (or nodes)
- E is the (finite) set of of edges, corresponding to a binary relation on V

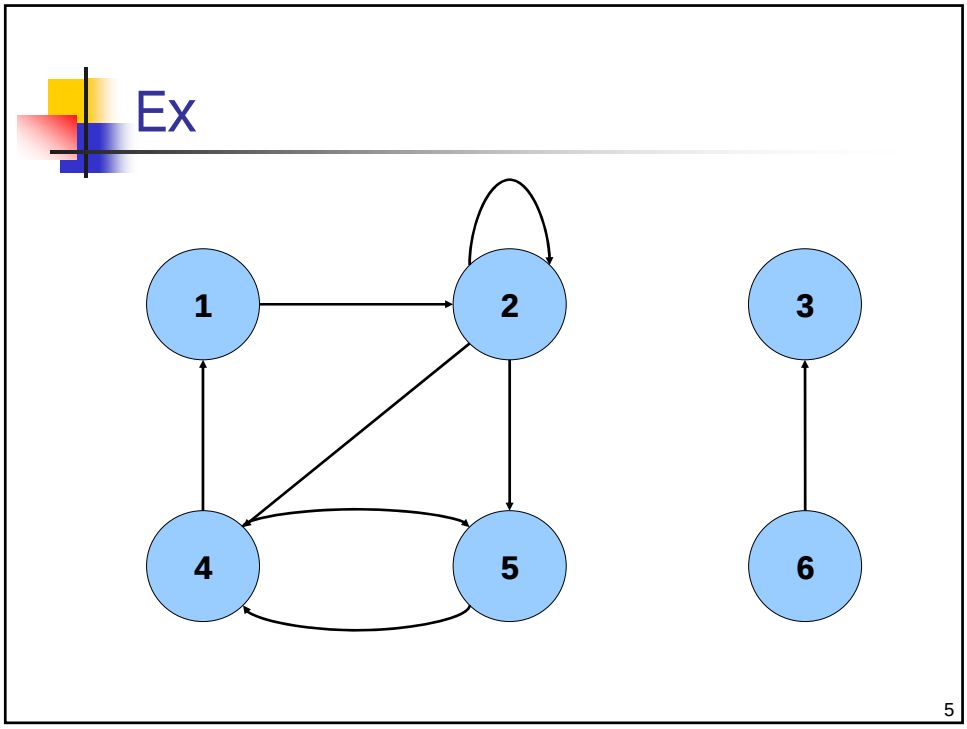
3

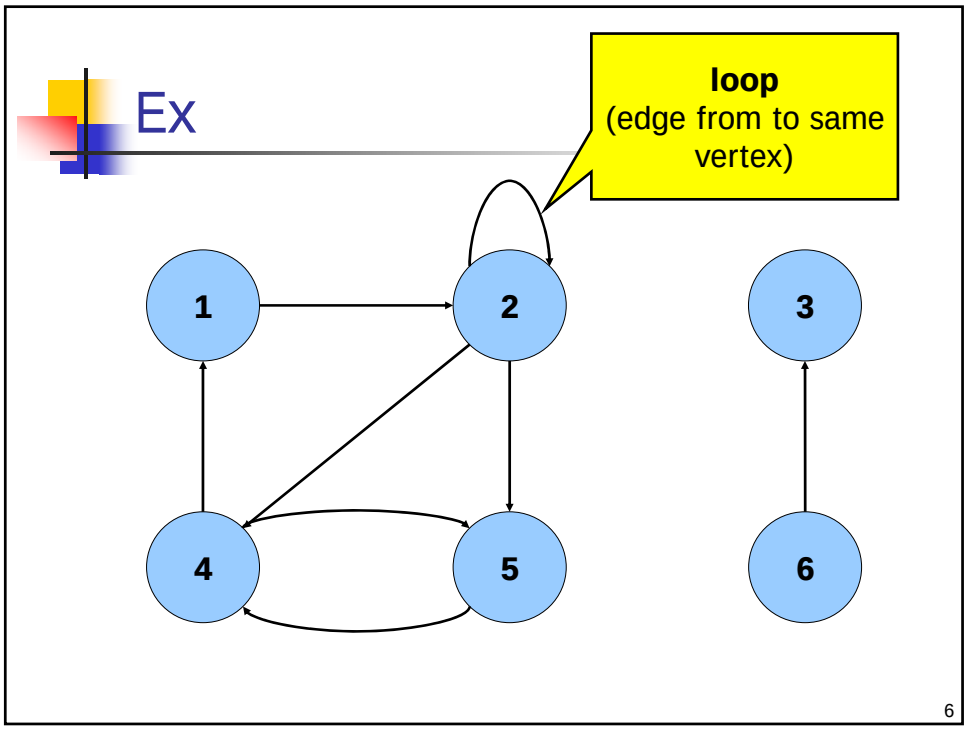
Representation

Graphs are often represented with

- Circles for vertexes
- Arrows for edges

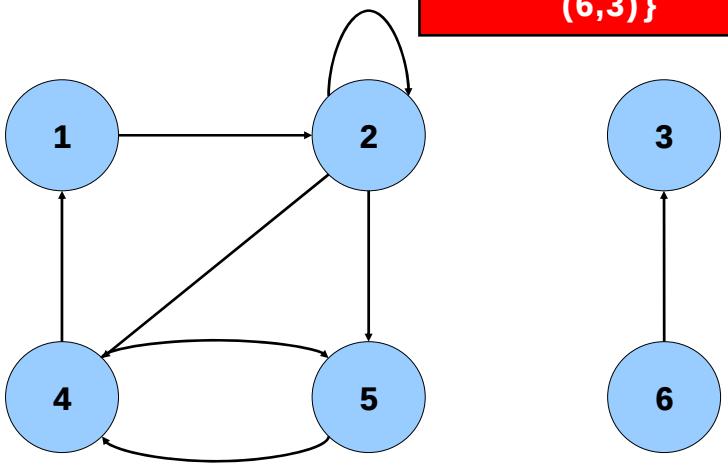
4


Ex
1
2
3
4
5
6
5


Ex
loop
(edge from to same vertex)
1
2
3
4
5
6
6

Ex

V={1,2,3,4,5,6}
E={(1,2),(2,2),(2,5),
(5,4),(4,5),(4,1),(2,4)
(6,3)}



7

Un directed graph

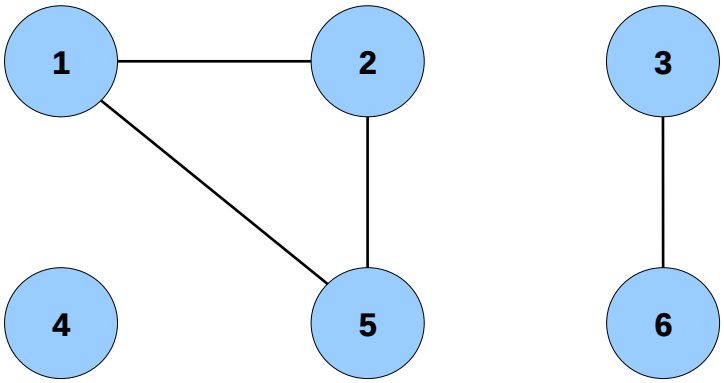
An undirected graph is a pair G=(V,E), E contains unordered pairs of vertexes

Edges are represented with lines

8

Ex

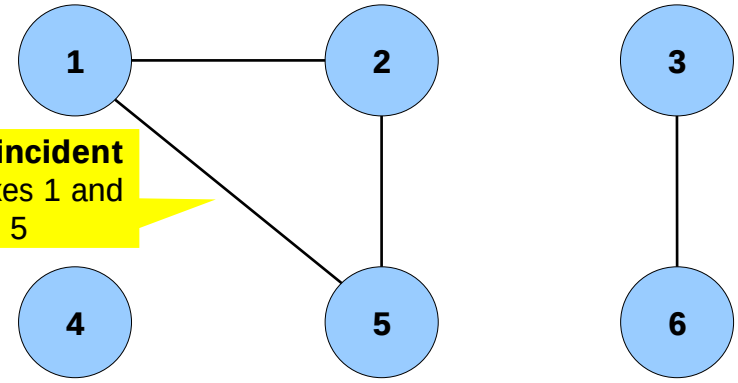
V={1,2,3,4,5,6}
E={(1,2),(2,5),
(5,1),(6,3)}



9

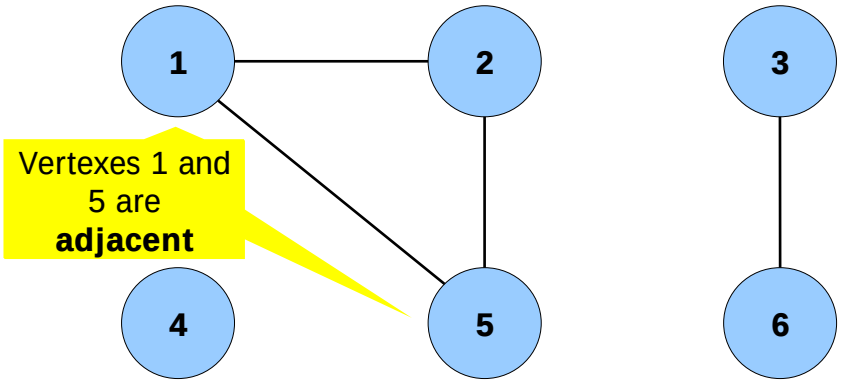
Ex

Edge **incident** vertexes 1 and 5



10

Ex



11

Degree

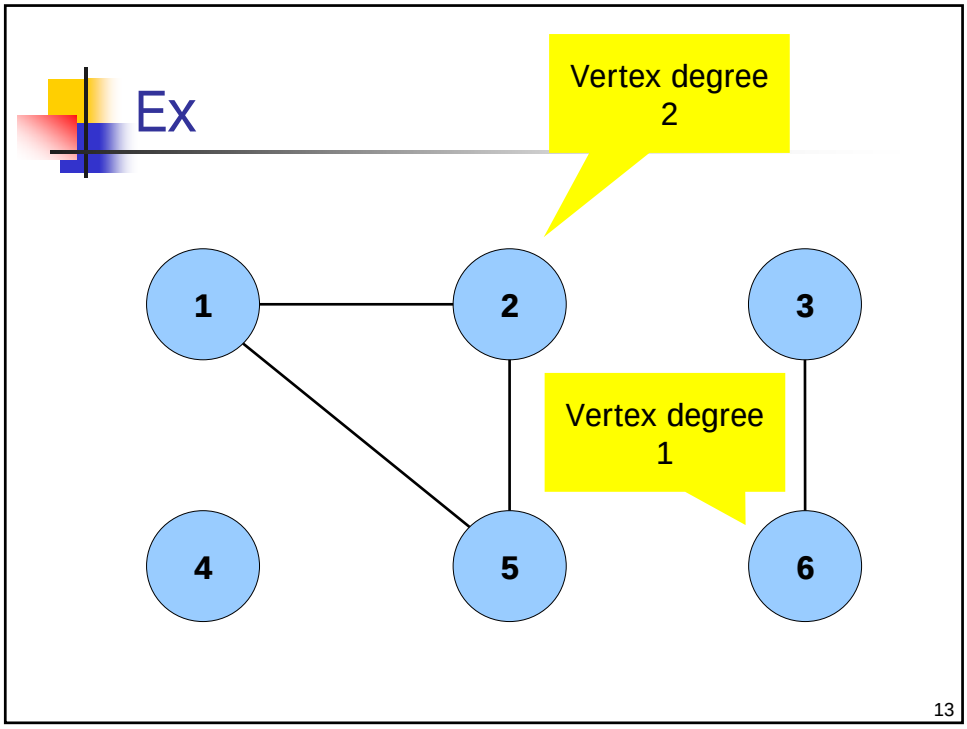
In an undirected graph the degree of a vertex is the number of incident edges

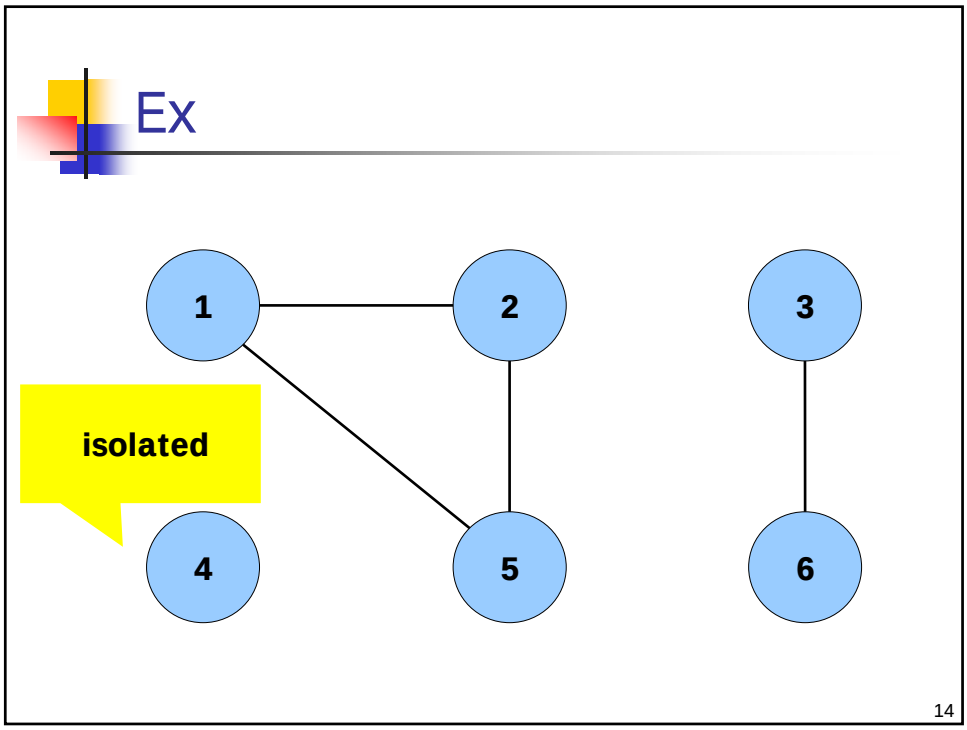
In a directed graph:

- The indegree is the number of edges entering a vertex
- The outdegree is the number of edges exiting a vertex
- The degree is the sum of indegree and outdegree

A vertex with degree 0 is defined isolated.

12


Ex
Vertex degree
2
1
2
3
Vertex degree
1
4
5
6
13


Ex
1
2
3
isolated
4
5
6
14

Path

A path from vertex u to vertex u' in graph G=(V,E) is a sequence of vertexes $(v_0,v_1,v_2,...,v_k)$ with $u=v_0$ and $u'=v_k$, and $(v_{i-1},v_i) \in E$ for i=1,2,...,k.

k is the lentgh of the path

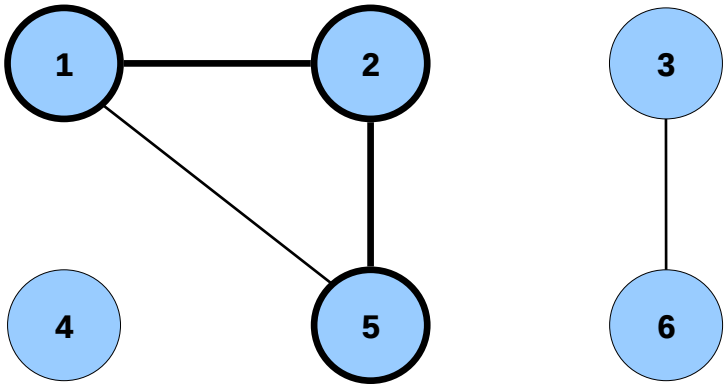
If a path exists from u to u' then u' is said reachable from u.

A path is simple if all vertexes in it are distinct

15

Ex

In bold path 1,2,5. Length is 2 and path is simple



16

Cycle

A cycle is a path with $v_0=v_k$.

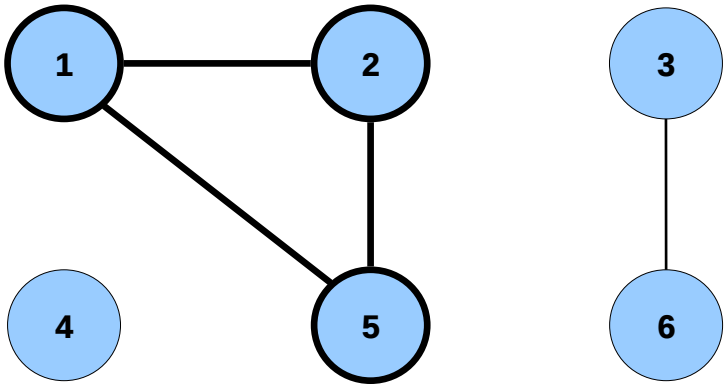
A loop is a cycle of length 1.

A graph with no cyclesis said acyclic

17

Ex

In bold path 1,2,5,1.
It is a cycle, of length 3.



18

Reachability (undirected g)

An undirected graph is connected if each pair of vertexes has a path that connects them

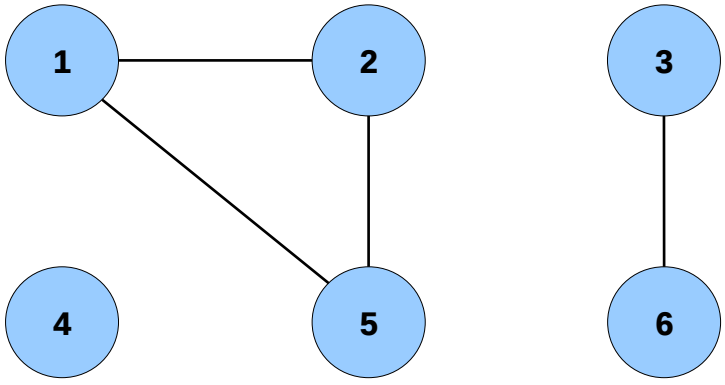
The connected subgraphs (of maximum size) are said connected components

A connected graph is composed ofonly one connected component.

19

Ex

The graph has 3 connected components: {1,2,5}, {3,6}, {4}



20

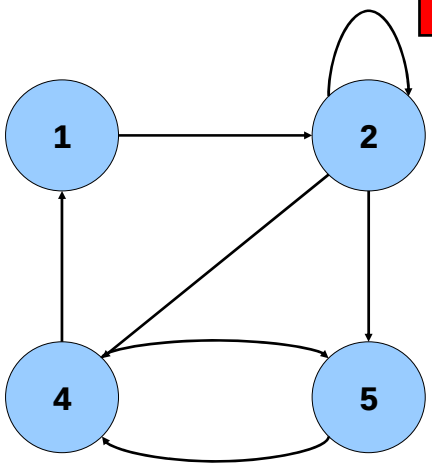
Reachability (directed g)

A directed graph is strongly connected if for each ordered pair of vertexes (u,u') exists a path from u to u'.

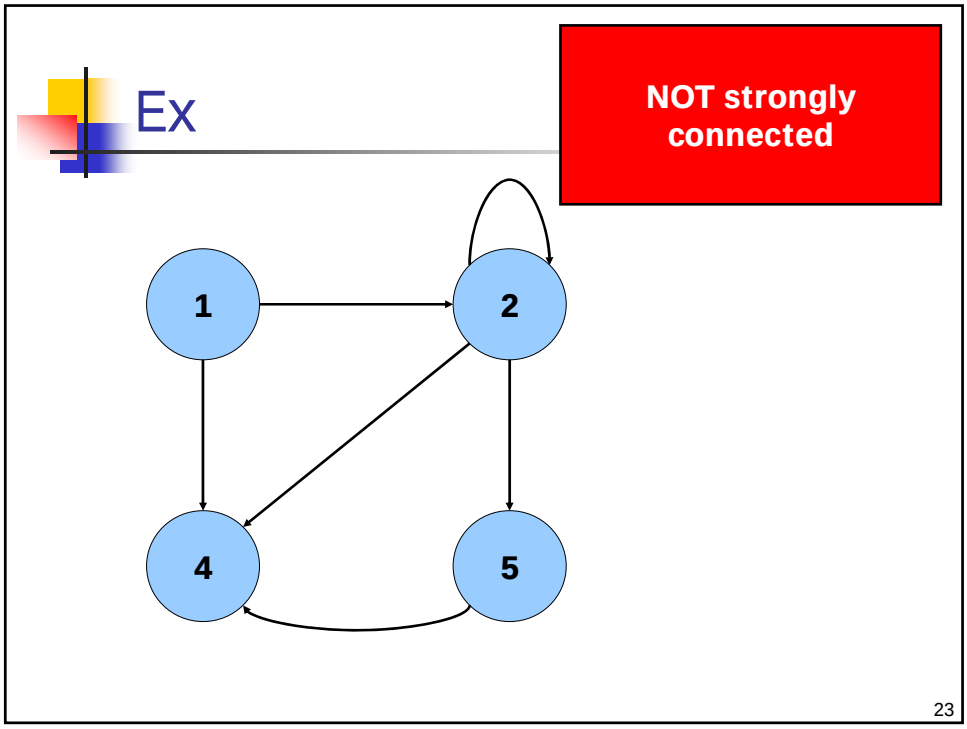
21

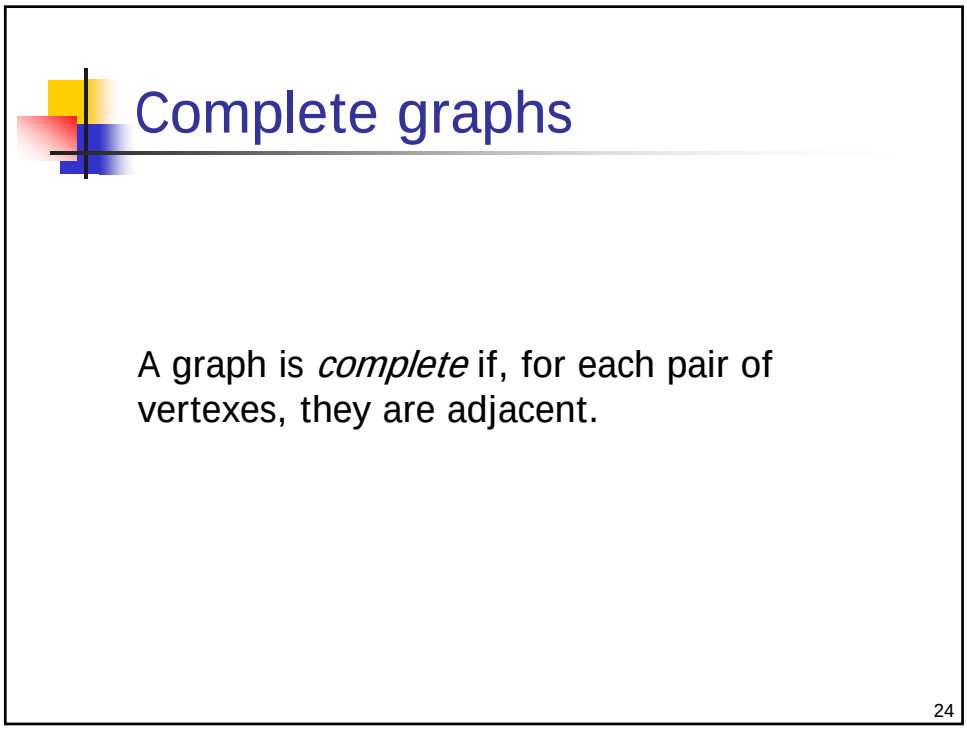
Ex

Strongly connected



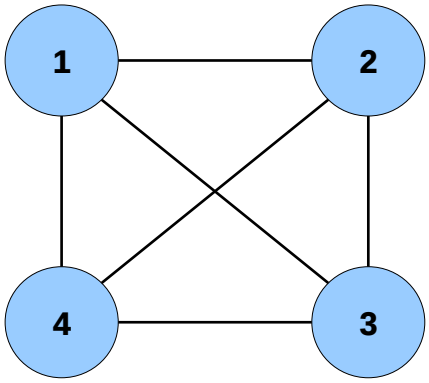
22


Ex
NOT strongly connected
1
2
4
5
23


Complete graphs
A graph is *complete* if, for each pair of vertexes, they are adjacent.
24

Ex

complete



25

Number of edges in complete graphs

A complete graph with n vertexes has edges:

- n^2, directed graph
- $n^2/2$, undirected graph

26

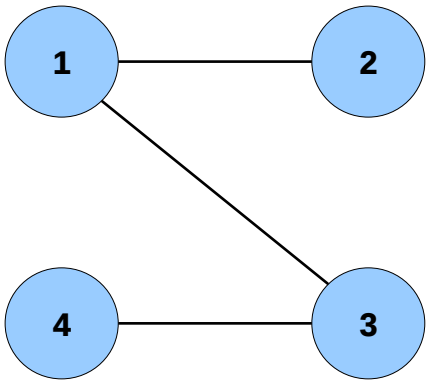
Density

Density of a graph is the ratio number of edges ($|E|$) and total possible number of edges

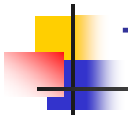
27

Ex

density 0.5



28



Trees and forestes

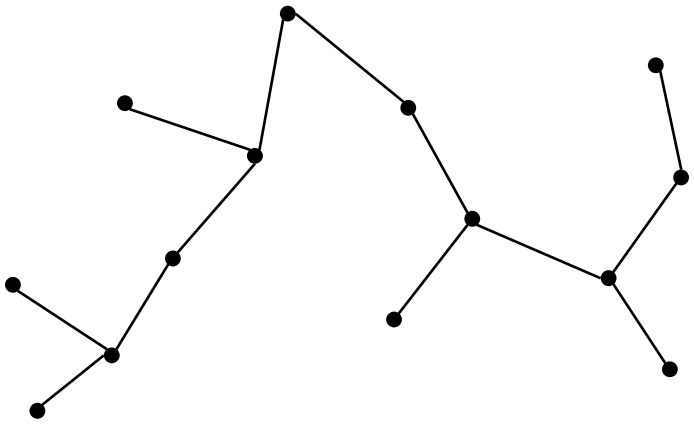
Forest : undirected, acyclic graph

Tree: undirected, acyclic, connected graph

29

Ex

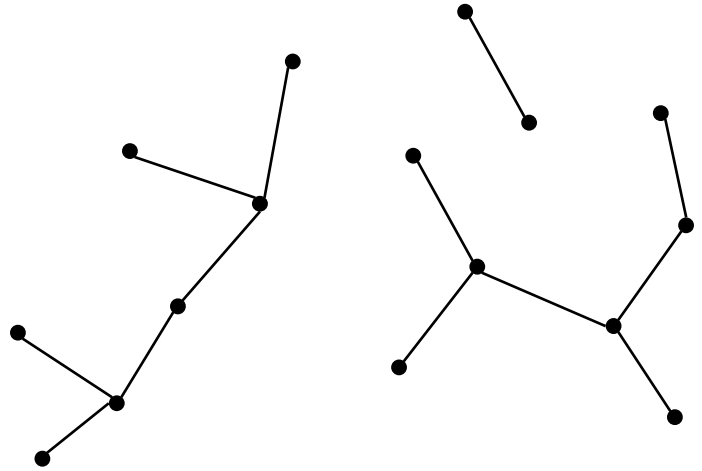
Tree



30

Ex

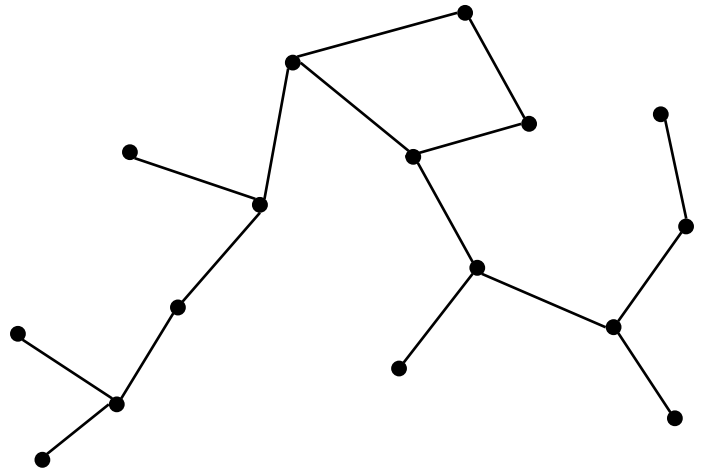
forest



31

Ex

Graph (no tree no forest because of cycle)



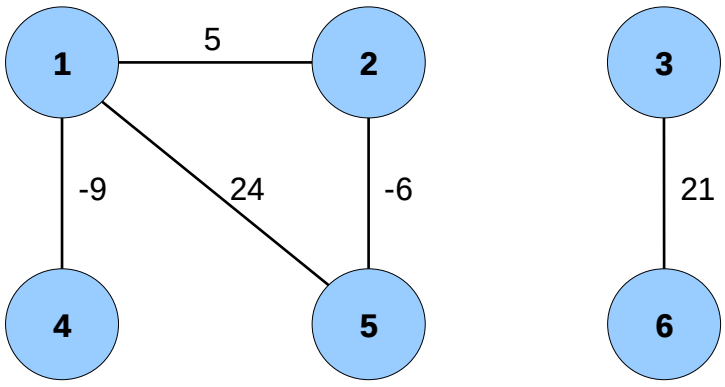
32

Weighted graphs

Each edge has a weight (number)

33

Ex



34

Applications of graphs

Ex:

- Telephone networks
- Flow chart.

35

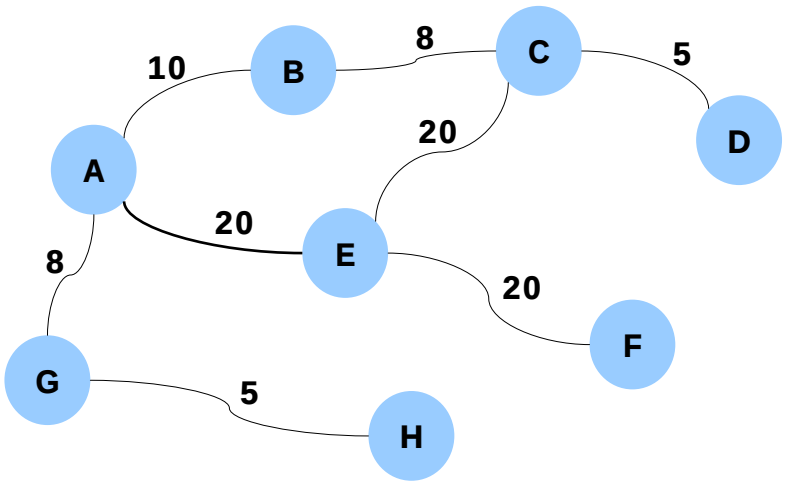
Telephone network (1)

Vertex: dispatching node

Edge: channel with certain bandwidth (weight)

36

Telephone network (2)



37

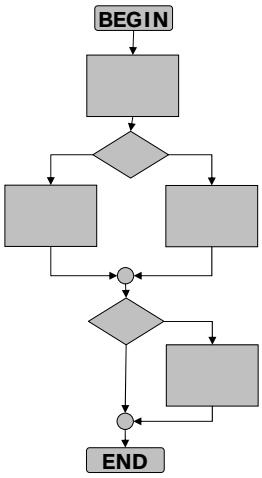
Telephone network (3)

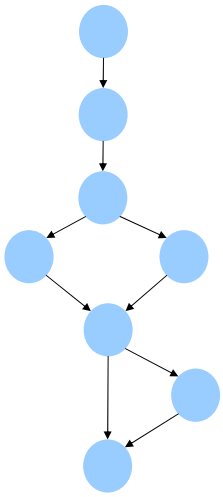
Questions:

- Is the graph connected? (if not some calls are not possible)
- What are critical edges? (if they fail the graph becomes unconnected)
- What are critical nodes? (if they fail the graph becomes unconnected)

38

Flow chart





39

Outline

- Definitions
- Graph implementation as data structure
- Visiting algorithms
- C implementations

40

Implementations

- Adjacency lists
- Adjacency matrix

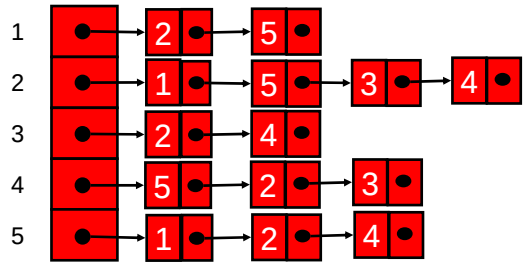
41

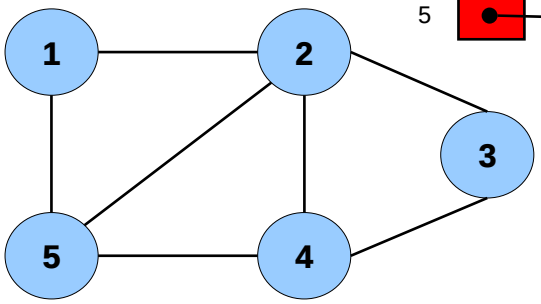
Adjacency list

Array A, element A[i] contains pointer to linked list of vertexes adjacent to i.

42

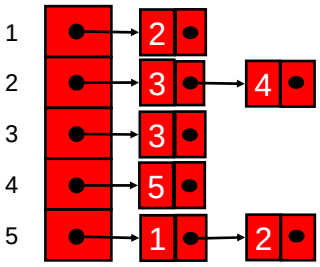
Ex (undirected g)

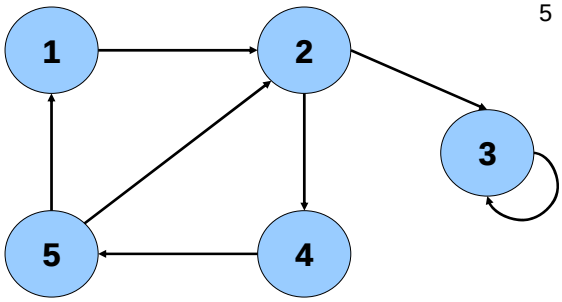




43

Ex (directed g)





44

Memory

Directed graph: lists contain |E| elements
Undirected graph: lists contain 2|E| elements

Memory usage is O(max(V,E)).

45

Limits

Checking adjacency u - u‘
To check it it is needed to visit all list of adjacent nodes

46

Adjacency matrix

Matrix of size $|V| \times |V|$. Element a_{ij} is

- 1 if $(i,j) \in E$
- 0 otherwise

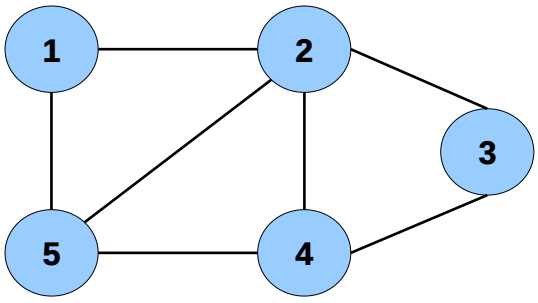
In undirected graphs the matrix is simmetric.

In weighted graphs element a_{ij} contains the weigth

47

Ex (undirected)

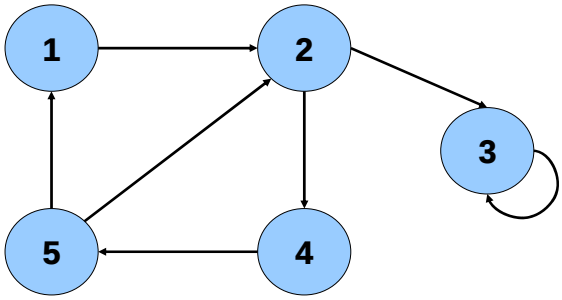
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



48

Ex (directed)

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



49

Comparison

Matrix, in comparison with lists:

- Require more memory (unless graph is very dense)
- Are faster (adjacency is computed in constant time)

50

Outline

- Definitions
- Graph implementation as data structure
- Visiting algorithms
- C implementations

51

Visiting algorithms

Visit: starting from a vertex (source) touch all vertexes connected to it

Two algorithms :

- Breadth visit
- Width visit

52

Breadth

breadth-first search, BFS consists in visiting per layers:

$l=0$, $S_l=\{source\}$

1. Visit all vertexes S_{l+1} not yet visited and adjacent to a vertex in S_l
2. $l=l+1$
3. Repeat from 2 until S_l is empty.

53

FIFO Queue

FIFO queue is used to represent layers:

- Each time a node is visited it is put in the queue
- At each step a node is extracted from the queue, its adjacents are checked. If not yet visited, they are.

54

Coloring

To support the visit vertexes are ‘colored’ as follows

- Initially all vertexes are white
- A vertex becomes gray when visited the first time
- A vertex becomes black when all adjacent vertexes, and not yet visited, are put in the queue

55

Pseudo-code

```
BFS(G, s)
 1  for ogni vertice u ∈ V[G] – {s}
 2       do  color[u] ← WHITE
 3           d[u] ← ∞
 4           π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← {s}
 9  while Q ≠ ∅
10       do  u ← head[Q]
11           for ogni v ∈ Adj[u]
12                do if color[v] = WHITE
13                     then  color[v] ← GRAY
14                           d[v] ← d[u] + 1
15                           π[v] ← u
16                           ENQUEUE(Q, v)
17           DEQUEUE(Q)
18           color[u] ← BLACK
```

56

BFS Tree

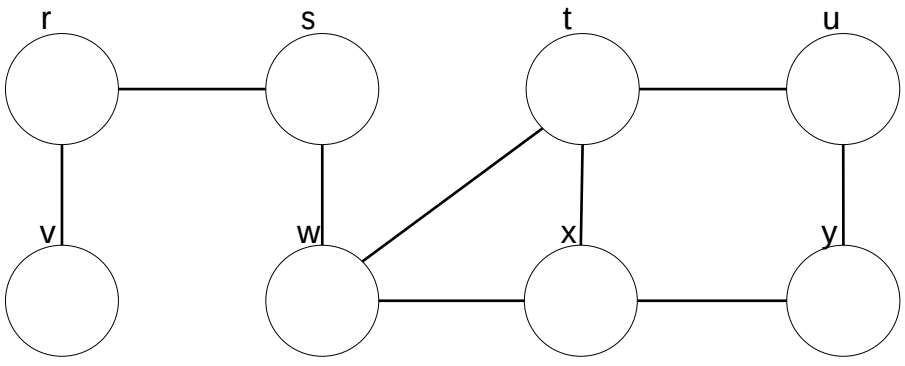
A width visit produces a BFS tree where

- The root is the source
- Vertexes are the ones of the graph
- Edges are a subset

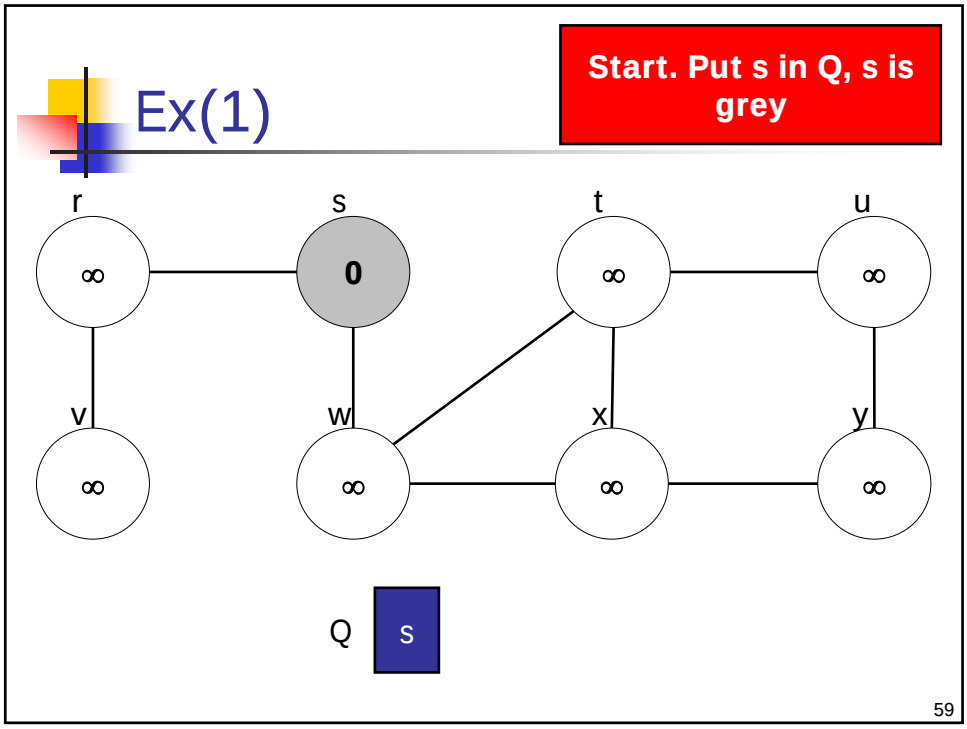
57

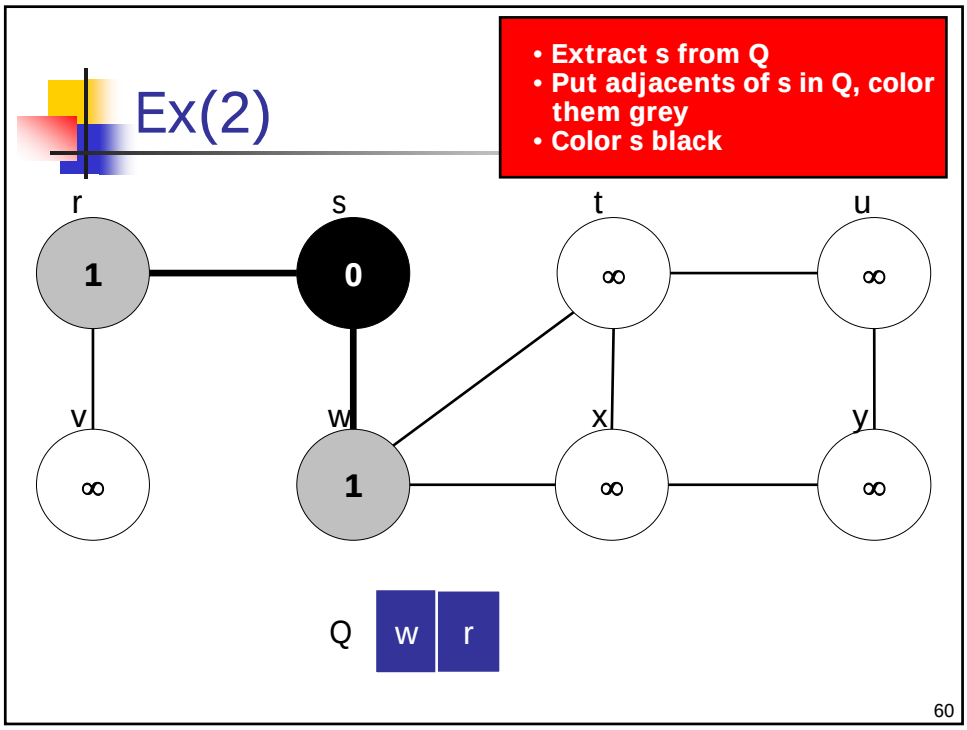
Ex (0)

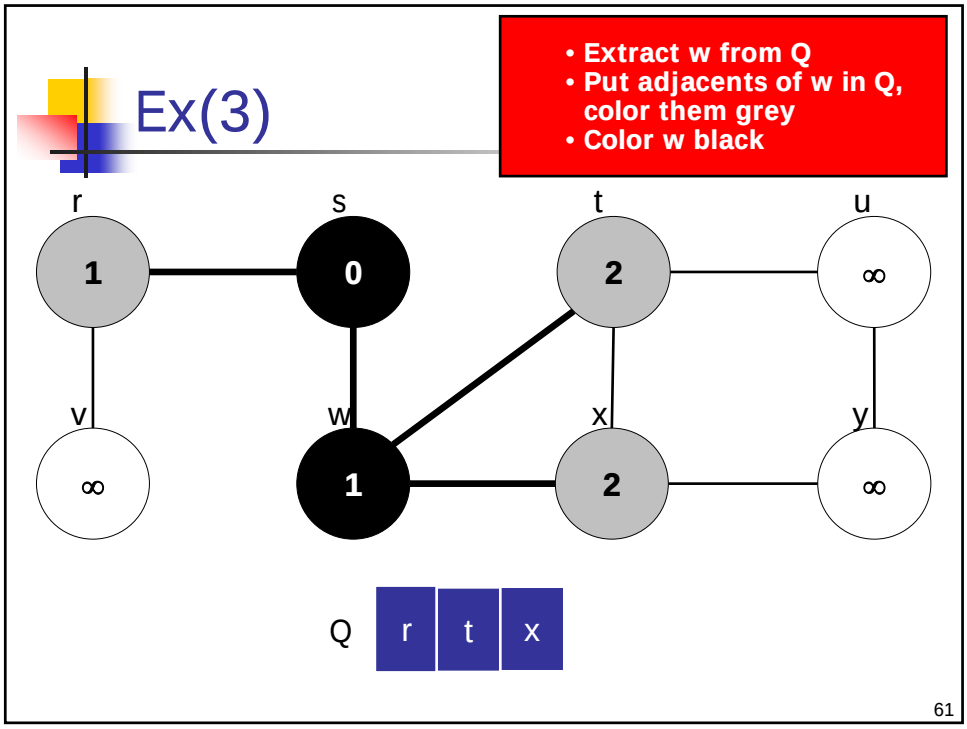
Source: s

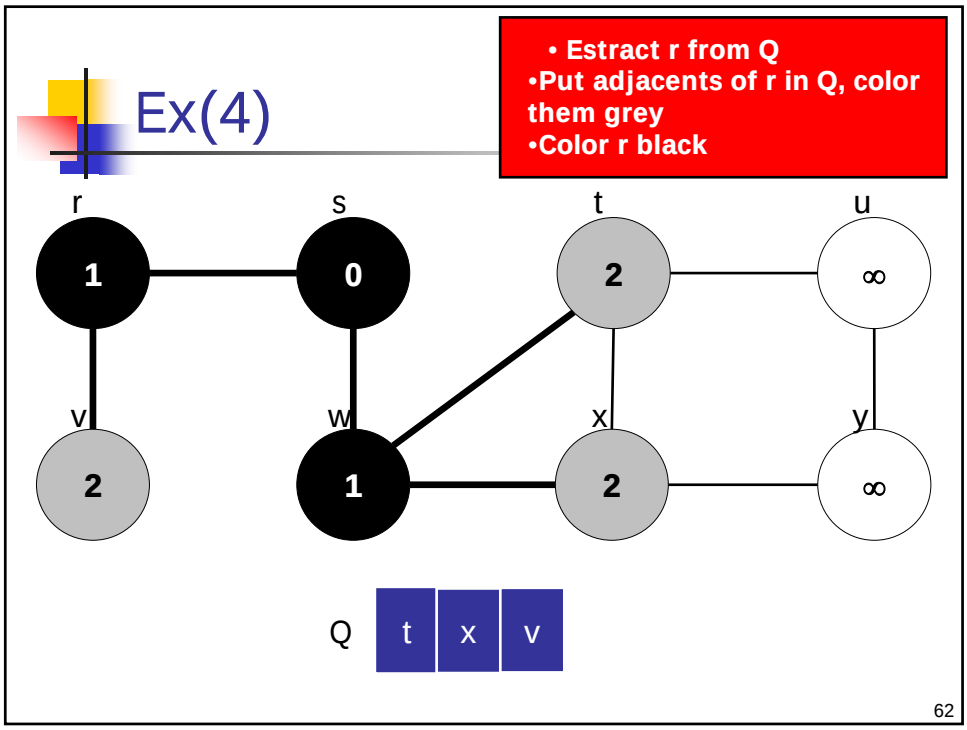


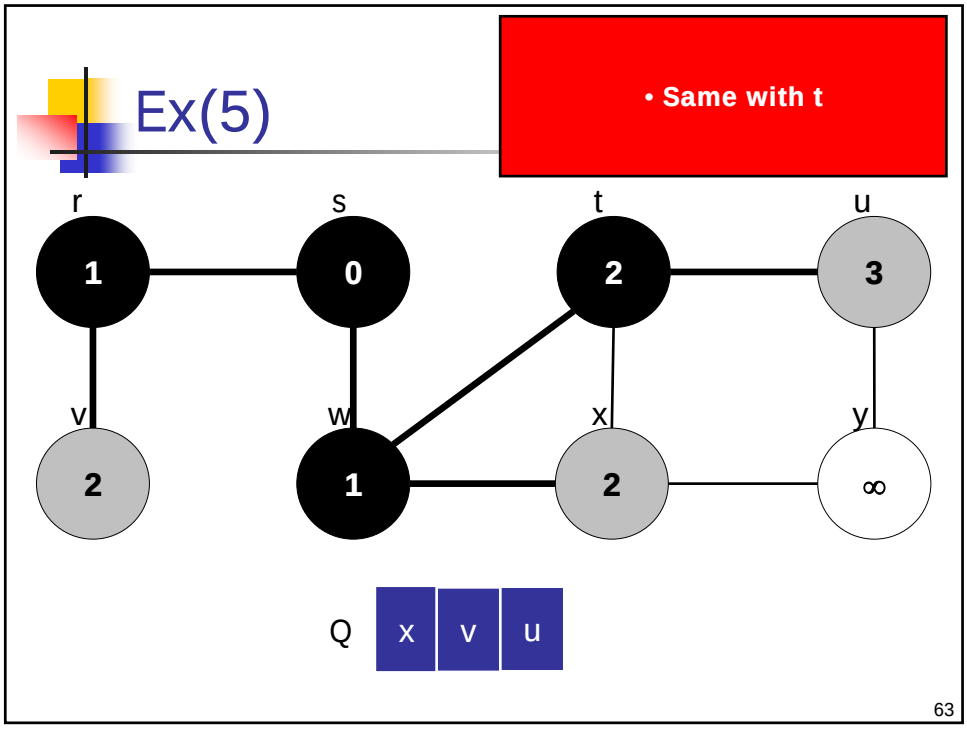
58

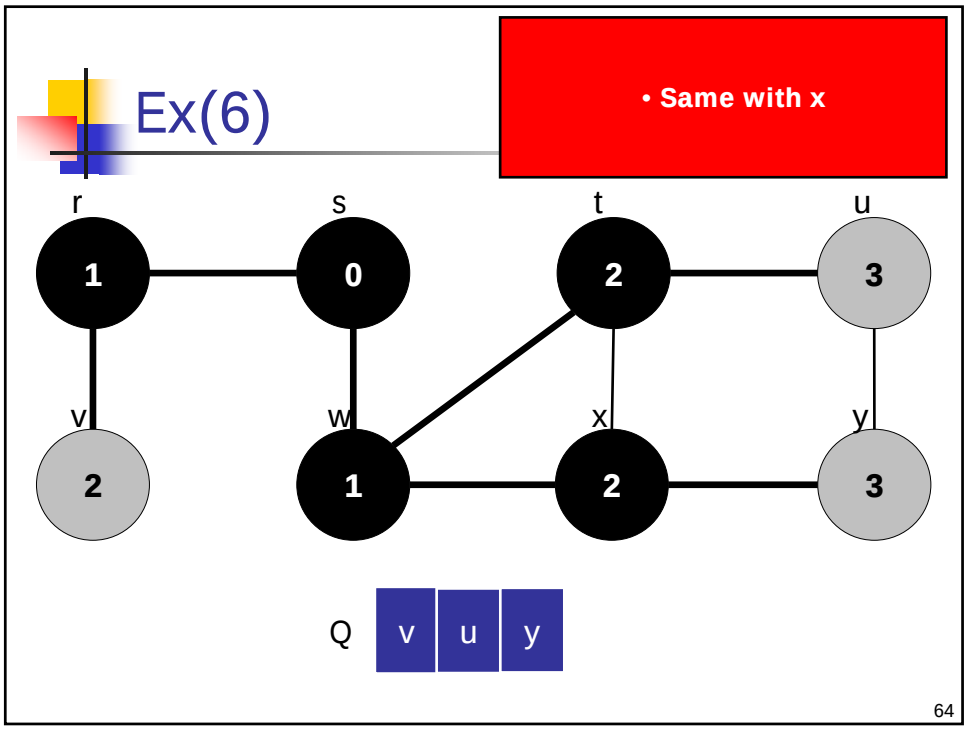

Ex(1)
Start. Put s in Q, s is grey
r
s
t
u
∞
0
∞
∞
v
w
x
y
∞
∞
∞
∞
Q
s
59

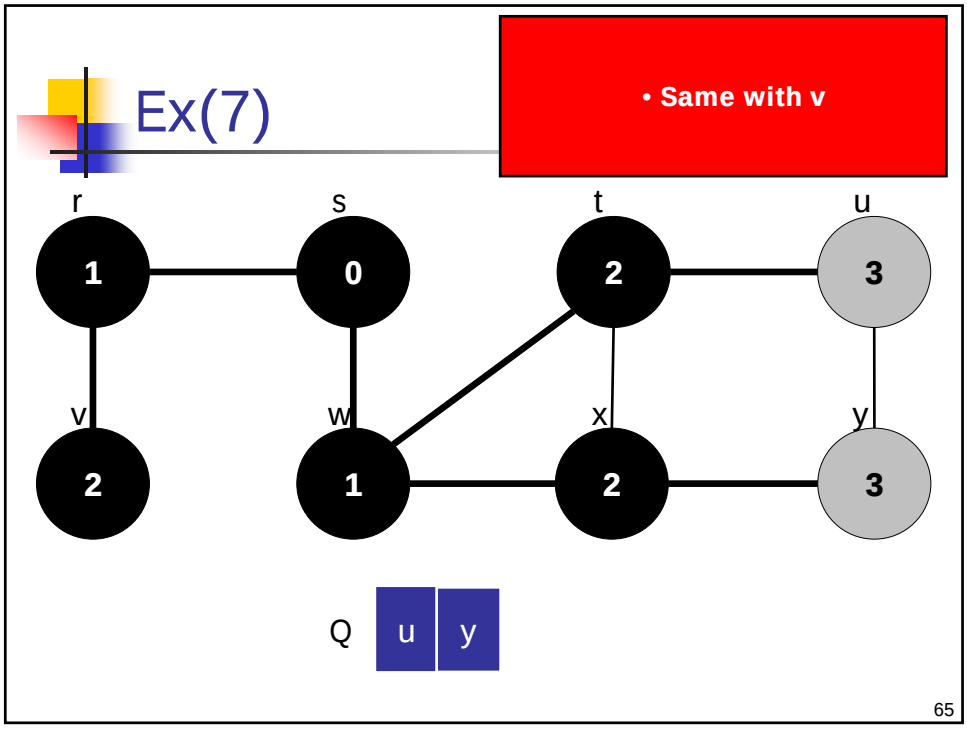

Ex(2)
• Extract s from Q
• Put adjacents of s in Q, color them grey
• Color s black
r
s
t
u
1
0
∞
∞
v
w
x
y
∞
1
∞
∞
Q
w
r
60

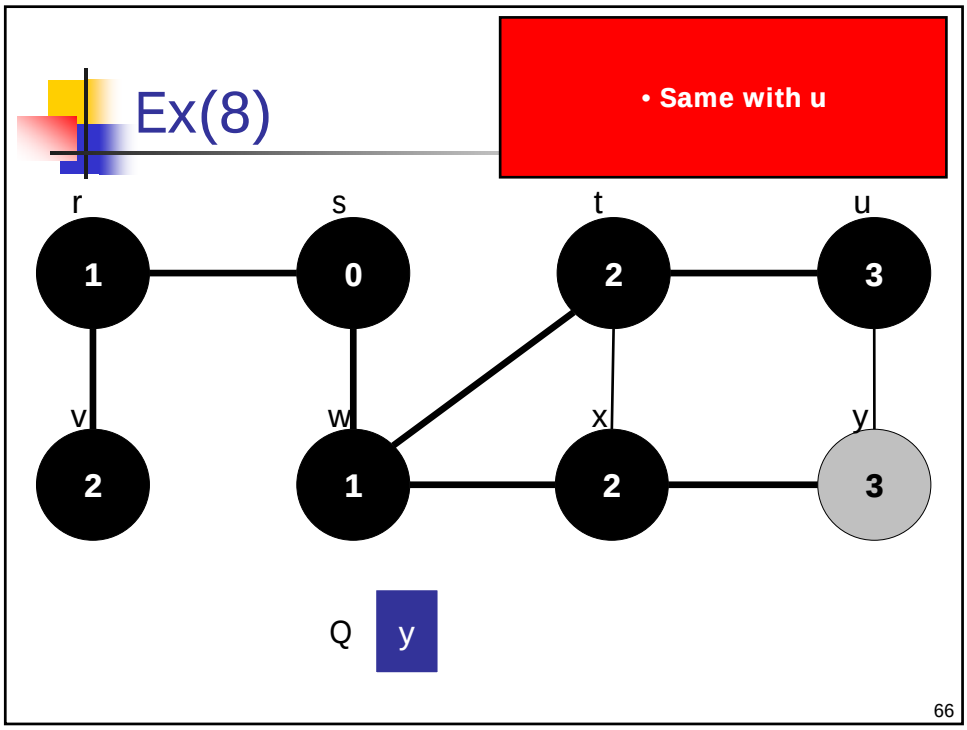

Ex(3)
• Extract w from Q
• Put adjacents of w in Q, color them grey
• Color w black
r
s
t
u
1
0
2
∞
v
w
x
y
∞
1
2
∞
Q
r
t
x
61

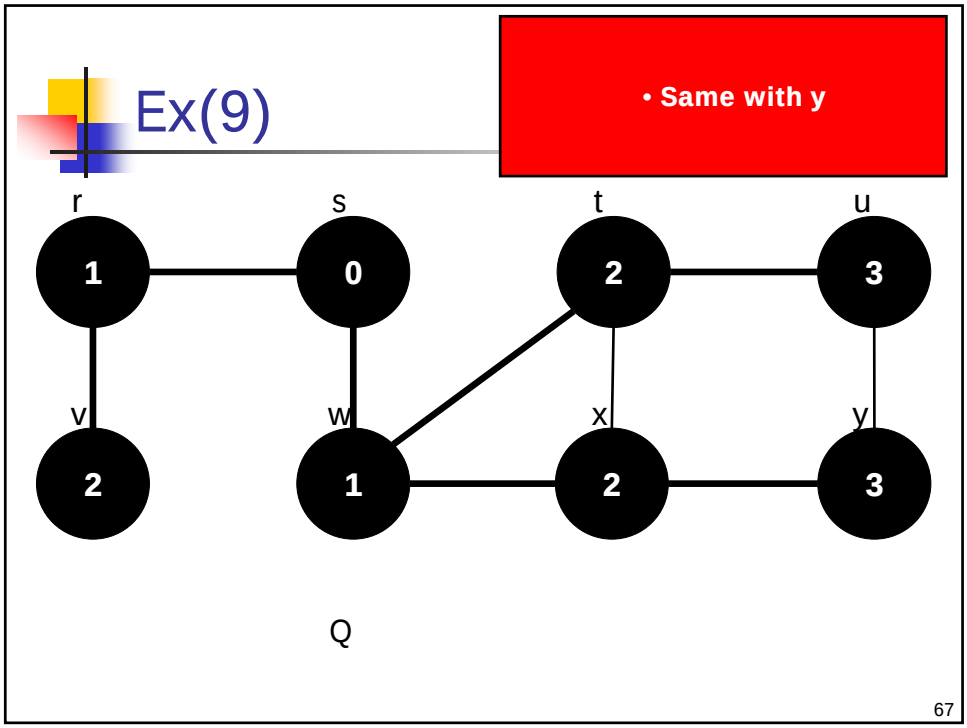

Ex(4)
• Estract r from Q
•Put adjacents of r in Q, color them grey
•Color r black
r
s
t
u
1
0
2
∞
v
w
x
y
2
1
2
∞
Q
t
x
v
62


Ex(5)
• Same with t
r
s
t
u
1
0
2
3
v
w
x
y
2
1
2
∞
Q
x
v
u
63


Ex(6)
• Same with x
r
s
t
u
1
0
2
3
v
w
x
y
2
1
2
3
Q
v
u
y
64


Ex(7)
• Same with v
r
s
t
u
1
0
2
3
v
w
x
y
2
1
2
3
Q
u
y
65


Ex(8)
• Same with u
r
s
t
u
1
0
2
3
v
w
x
y
2
1
2
3
Q
y
66


Ex(9)
• Same with y
r
s
t
u
1
0
2
3
v
w
x
y
2
1
2
3
Q
67


Complexity
Time for BFS is O(V+E).
68

Shortest path

Shortest path (s,v) = min number of arcs on a path s to v (assuming not weighted graph)

BFS computes shortest paths from source

69

Depth visit

Depth-First Search, DFS

At each step visits a vertex adjacent to the last vertex visited

When no adjacents, backtarck to last vertex with adjacents not visited yet

70

Pseudo-code (1)

```
DFS(G)
1   for ogni vertice u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4   time ← 0
5   for ogni vertice u ∈ V[G]
6       do if color[u] = WHITE
7          then DFS-VISIT(u)
```

71

Pseudo-code (2)

```
DFS-VISIT(u)
1   color[u] ← GRAY          ▷ Il vertice bianco u è stato appena scoperto
2   d[u] ← time ← time + 1
3   for ogni v ∈ Adj[u]      ▷ Si esplora l'arco (u, v)
4       do if color[v] = WHITE
5          then π[v] ← u
6               DFS-VISIT(v)
7   color[u] ← BLACK         ▷ Si rende u nero: la sua visita è finita.
8   f[u] ← time ← time + 1
```

72

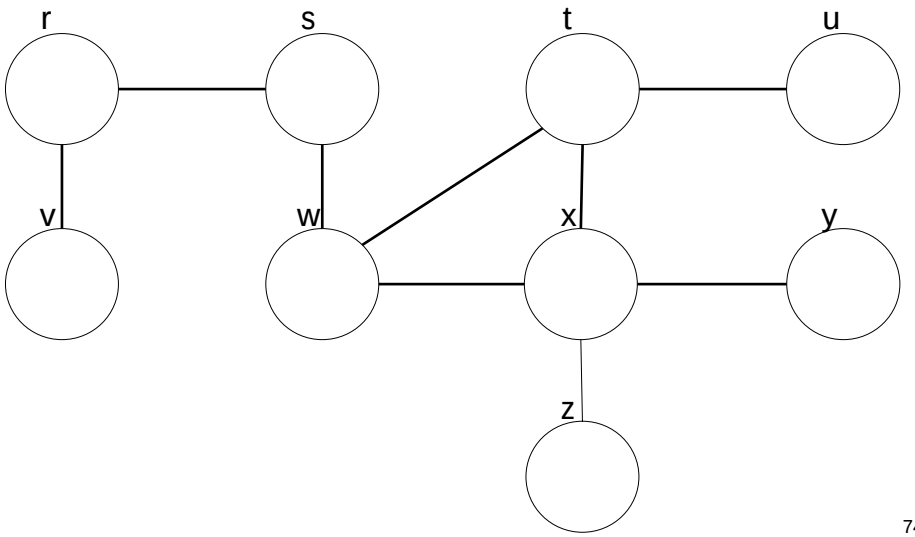
Coloring

- Initially all white
- Vertex becomes grey when visited first time
- Black when all adjacents visited

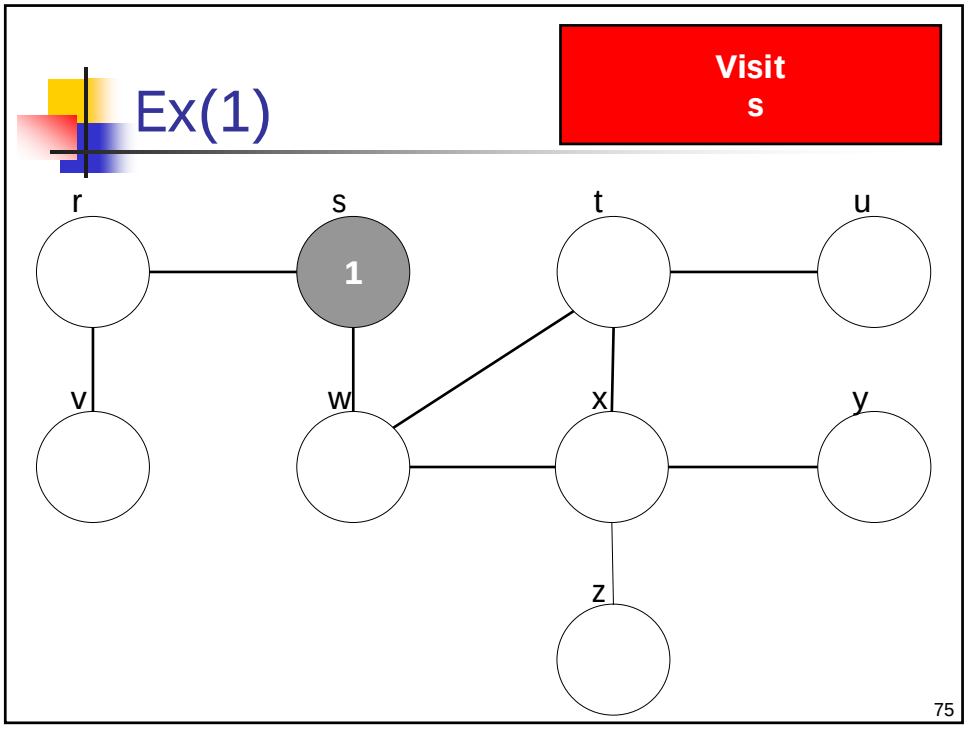
73

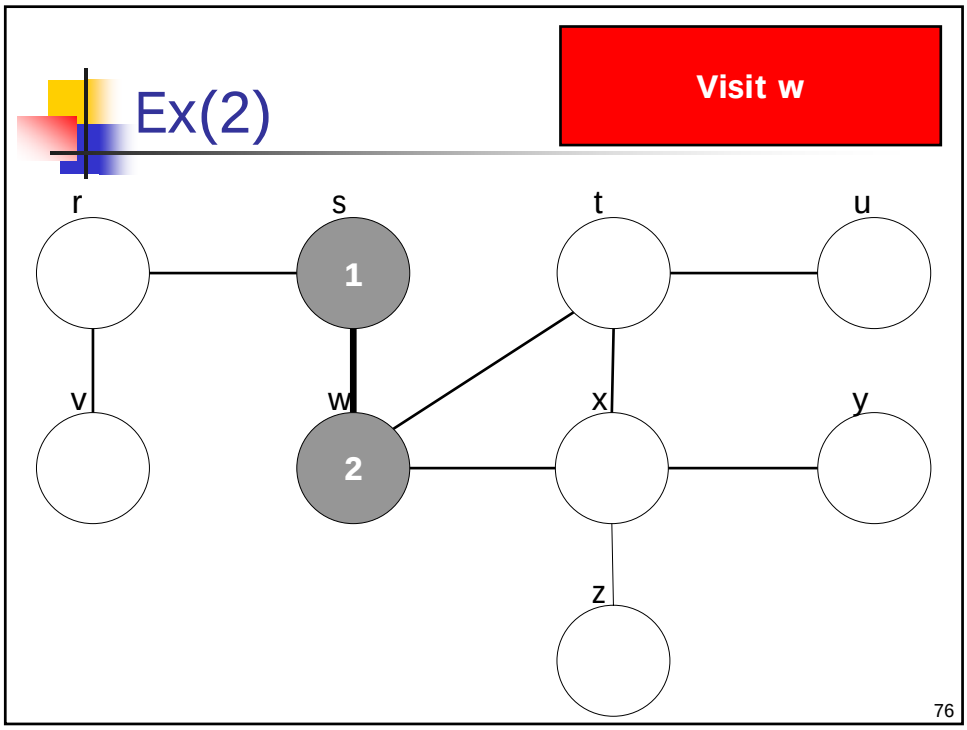
Ex (0)

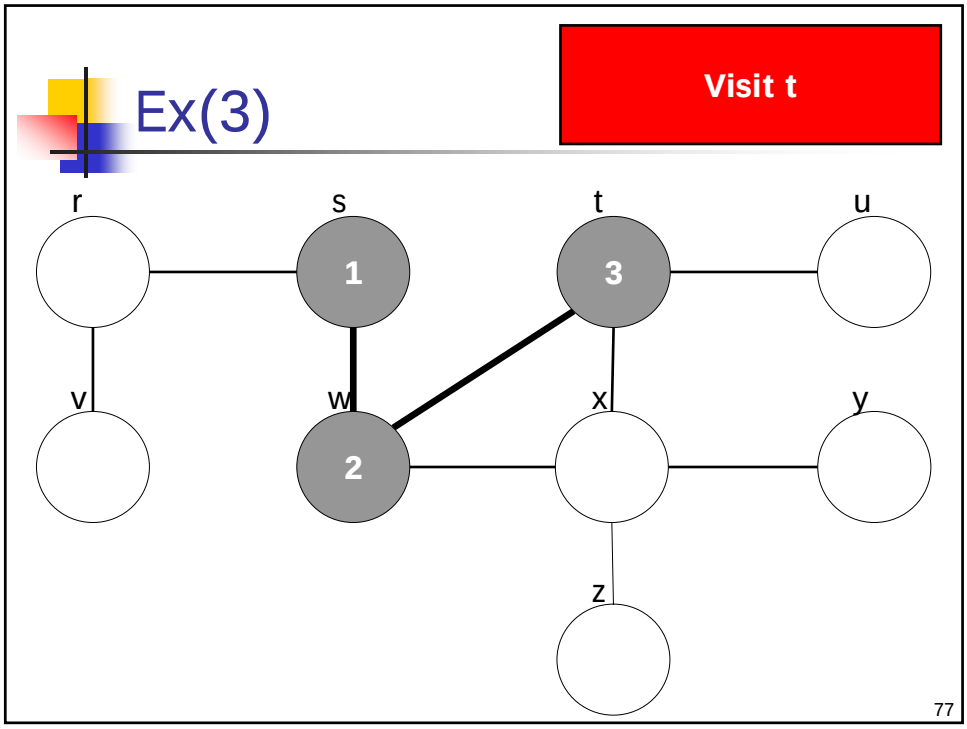
Source: s

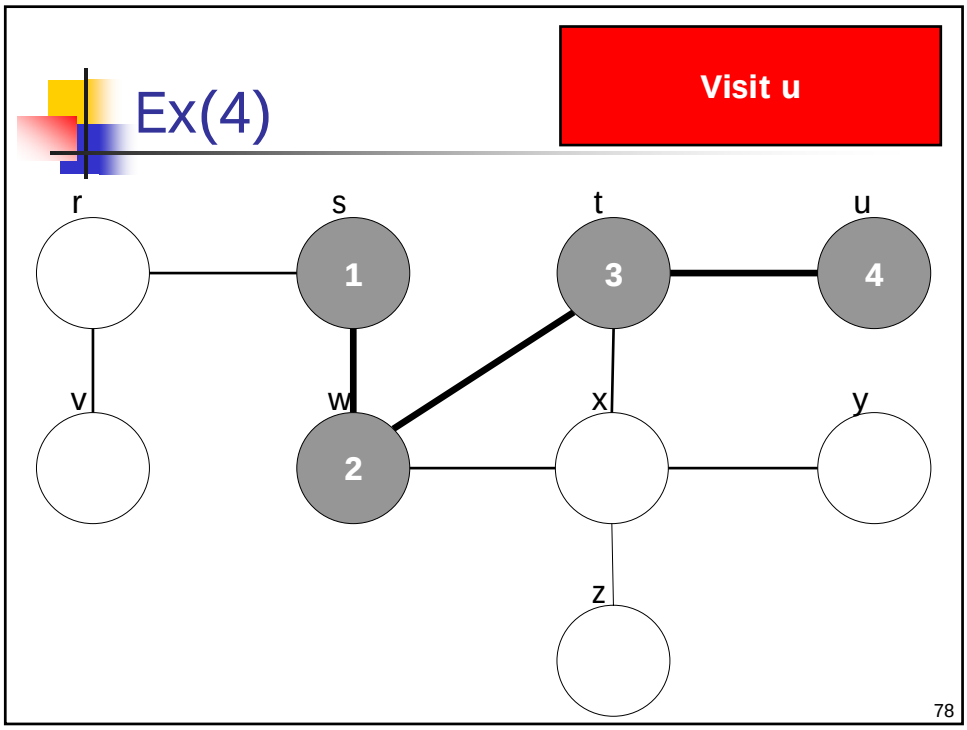


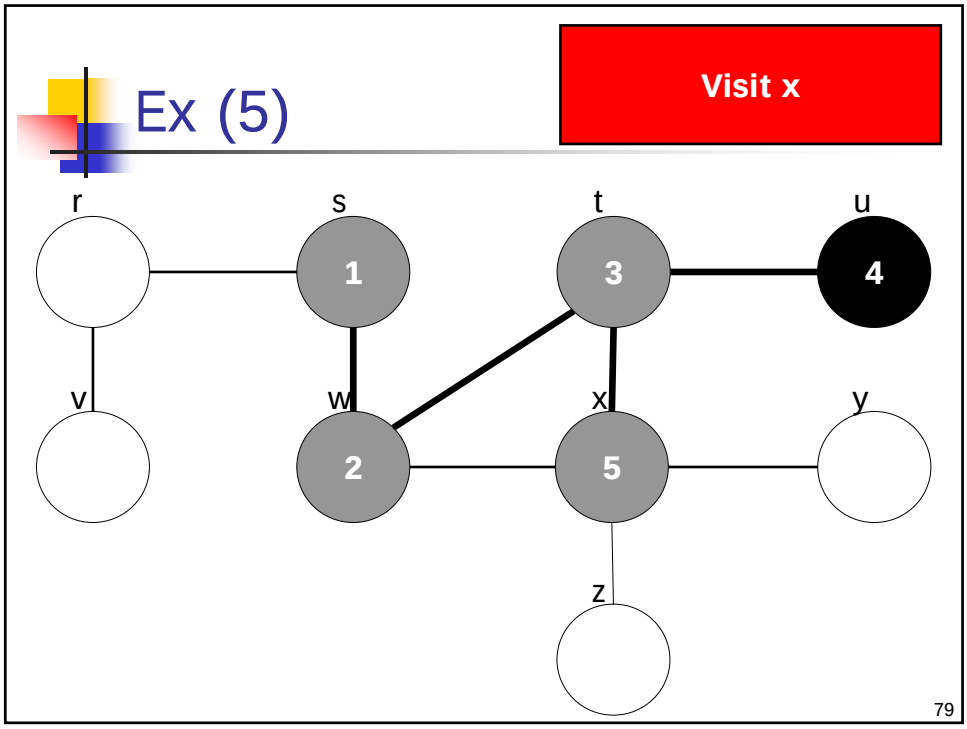
74

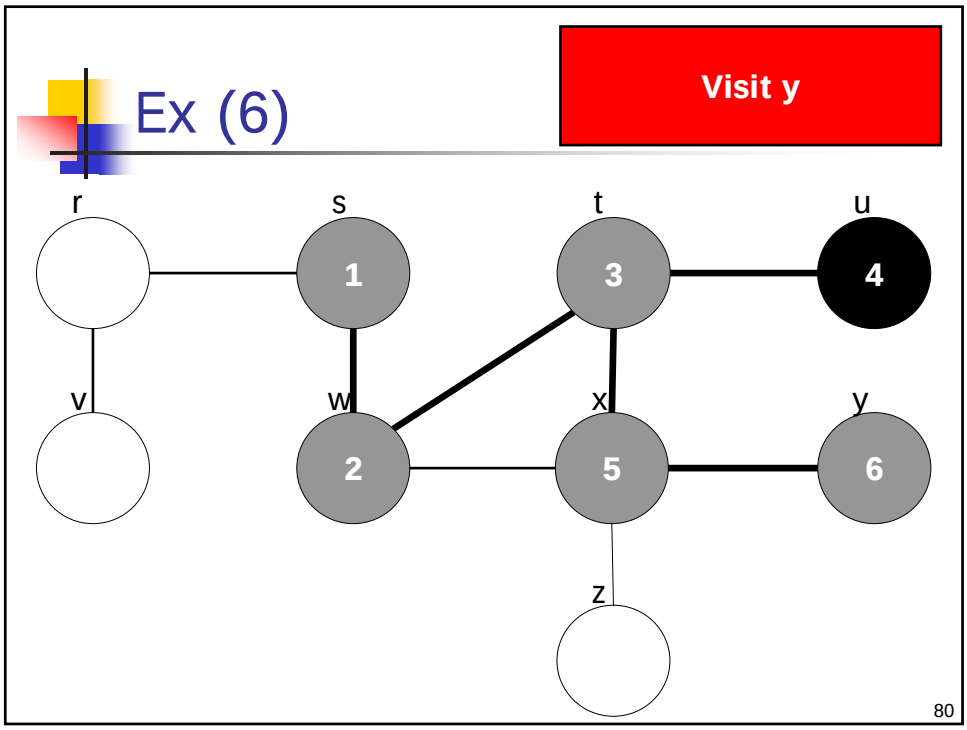

Ex(1)
Visit
s
r
s
t
u
1
v
w
x
y
z
75

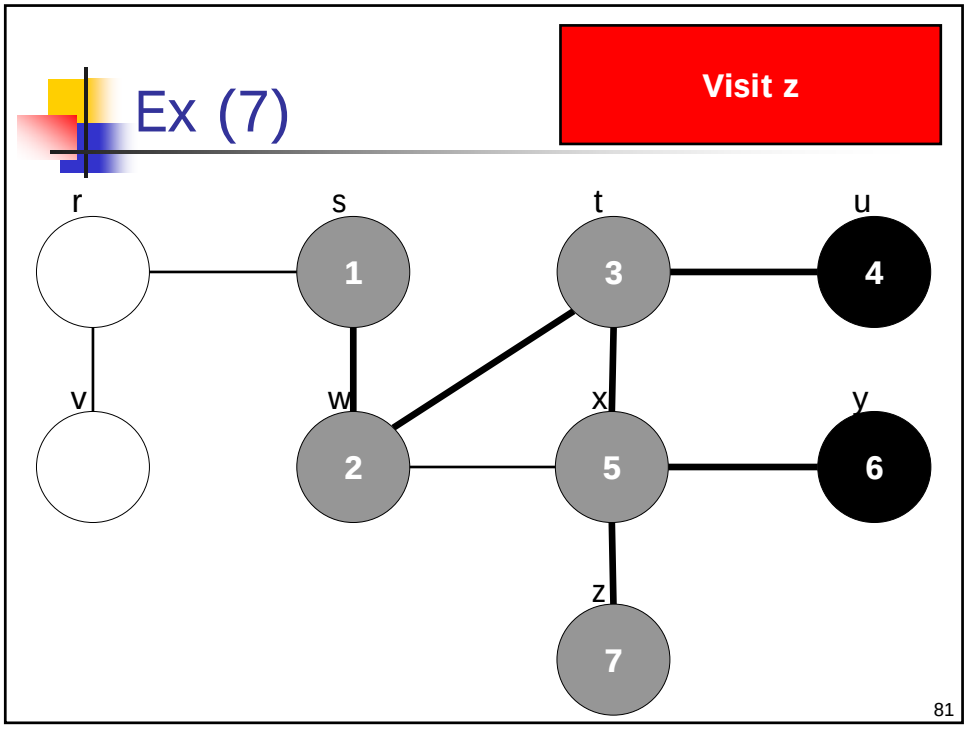

Ex(2)
Visit w
r
s
t
u
1
v
w
x
y
2
z
76

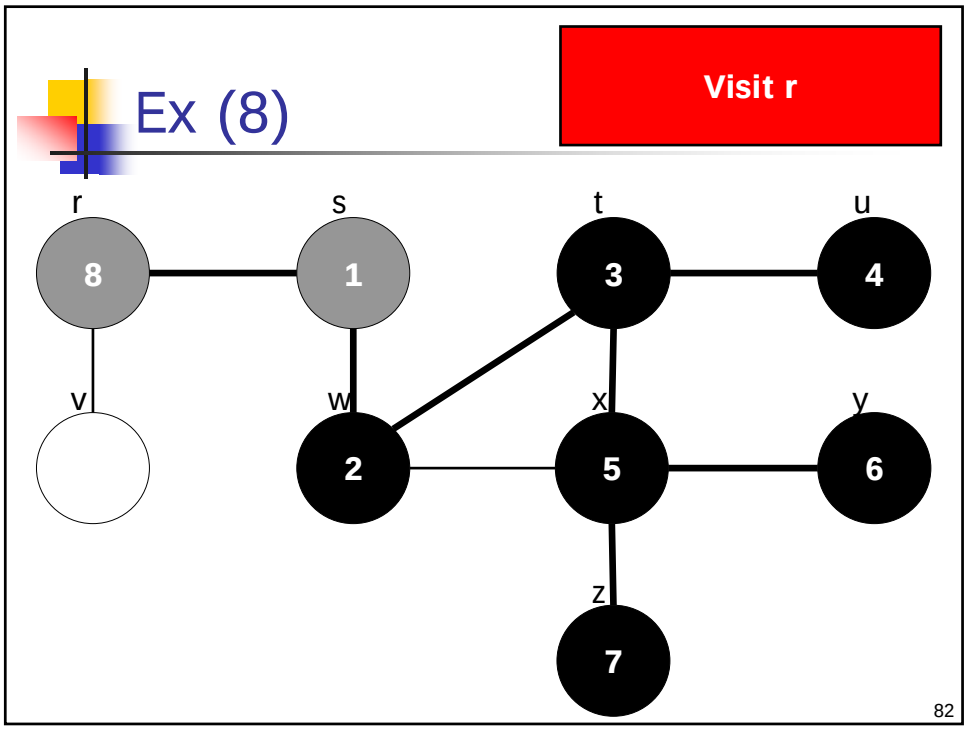

Ex(3)
Visit t
r
s
t
u
1
3
v
w
x
y
2
z
77

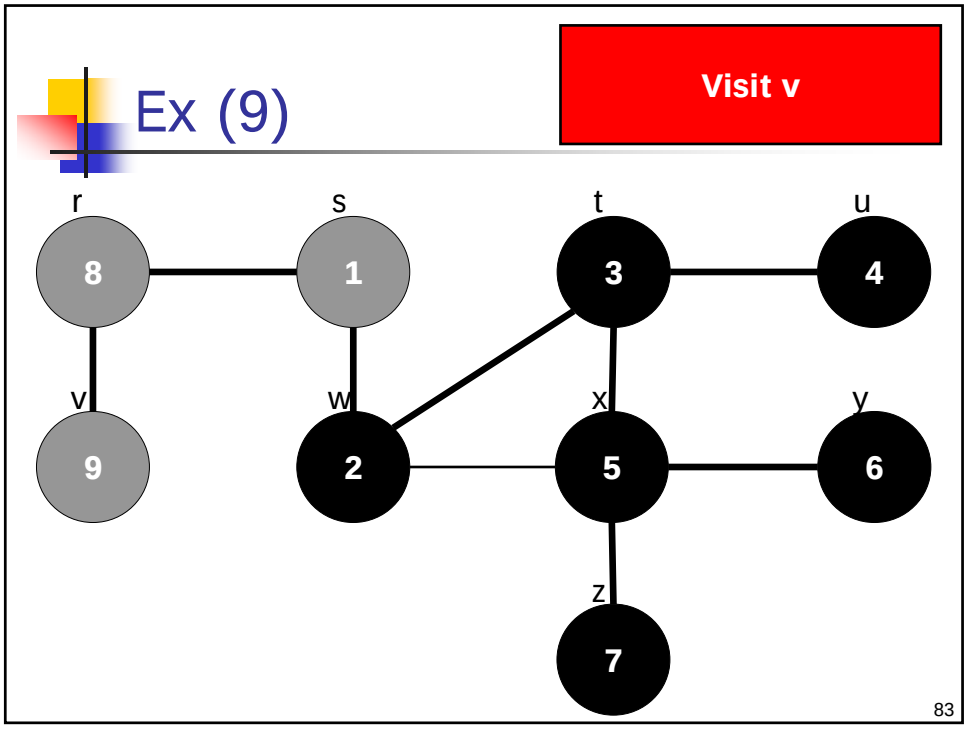

Ex(4)
Visit u
r
s
t
u
1
3
4
v
w
x
y
2
z
78

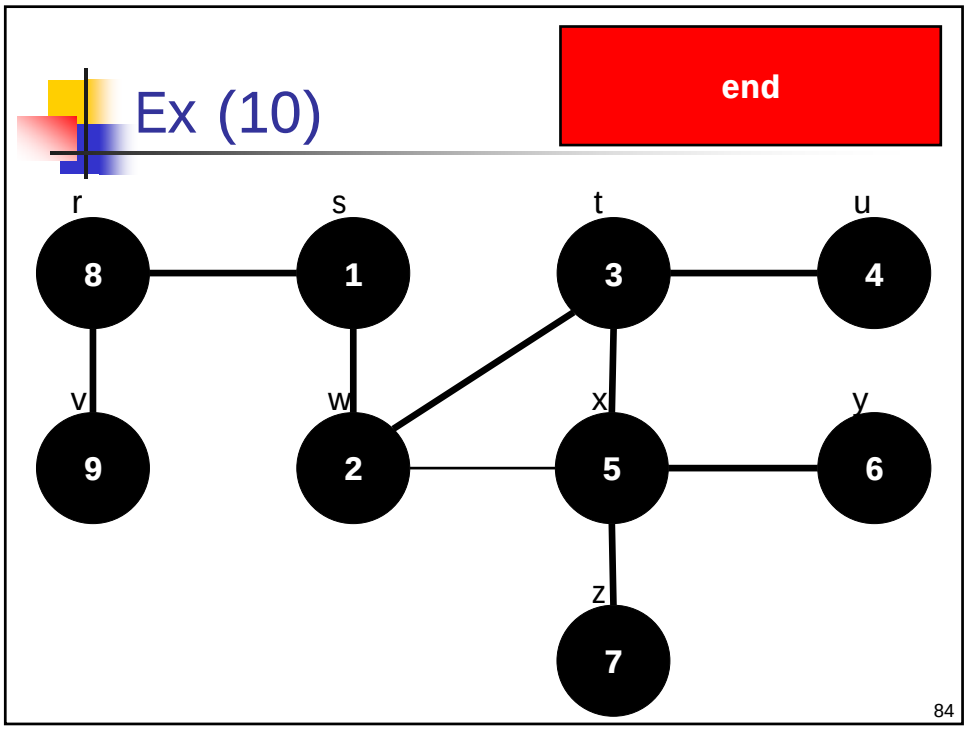

Ex (5)
Visit x
r
s
t
u
1
3
4
v
w
x
y
2
5
z
79


Ex (6)
Visit y
r
s
t
u
1
3
4
v
w
x
y
2
5
6
z
80


Ex (7)
Visit z
r
s
t
u
1
3
4
v
w
x
y
2
5
6
z
7
81


Ex (8)
Visit r
r
s
t
u
8
1
3
4
v
w
x
y
2
5
6
z
7
82


Ex (9)
Visit v
r
s
t
u
8
1
3
4
v
w
x
y
9
2
5
6
z
7
83


Ex (10)
end
r
s
t
u
8
1
3
4
v
w
x
y
9
2
5
6
z
7
84

Complexity

DFS is $\Theta(V+E)$.

85

DFS forest

DFS builds a forest foresta DFS, composed by one or more DFS trees

86

Outline

- Definitions
- Graph implementation as data structure
- Visiting algorithms
- C implementations

87

Lettura da file

Si supponga di voler leggere da un file la descrizione di un grafo e di volerla memorizzare in una lista di adiacenza.

Occorre definire

- Il formato del file
- Il formato della rappresentazione in memoria.

88

File format (one possibility)

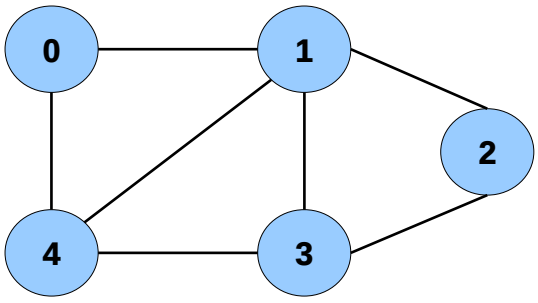
First line: number of vertexes *n*.

Next n blocks, each block:

- "*k“ is vertex number
- Number of adjacent nodes to k
- List of adjacents

89

Ex




```
5
*0
2
1
4
*1
4
0
4
3
2
*2
2
1
3
*3
3
2
1
4
*4
3
0
1
3
```

90

Adjacent list

- An array of pointers to VERTEX:

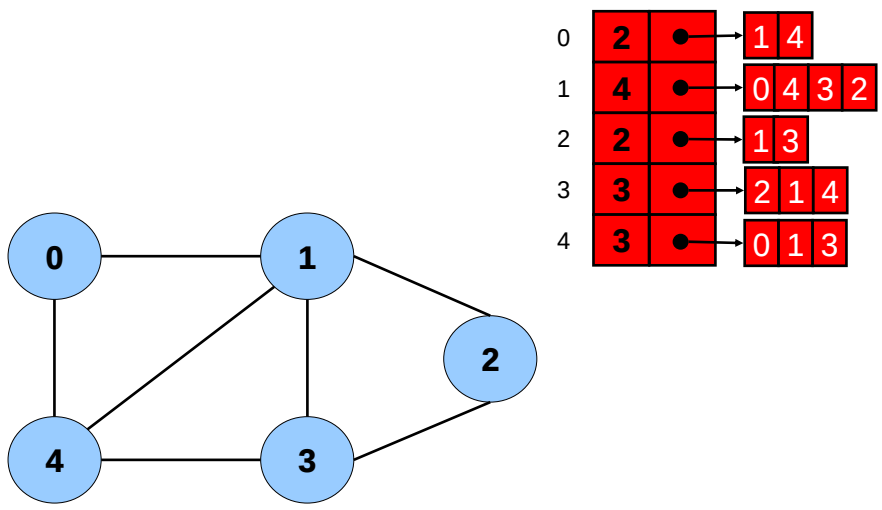
```
struct vertex{
     int  nadj;
     int  *adjlist;
}VERTEX;
```

- Each VERTEX points to array of integers (size == numer of adjacents

All arrays dynamically allocated when reading file

91

Ex



92

Graph.h

```
typedef struct vertex *VERTEXP;
struct vertex{
   int  nadj;
   int  *adjlist;
}VERTEX;
```

93

Readgrf (1)

```
#include <stdio.h>
#include "grafi.h"
VERTEXP  graph;
int      nvertex;
int readgrf(char name[])
{   FILE   *fin;
    int      nadj, n, i, j;
    if( (fin=fopen( name, "r")) == NULL)
    {   printf( "Errore in apertura file %s\n", name);
        return(0);   }
    fscanf( fin, "%d\n", &nvertex);
    if( (graph = (VERTEXP)malloc( nvertex * sizeof( VERTEX))) == NULL)
    {   printf( "Errore in allocazione graph\n");
        return( 0);   }
```

94

Readgrf (2)

```
for( i=0; i<nvertex; i++)
{   fscanf( fin, "*%d\n", &n);          /* number of vertex */
    if( n != i)
    {   printf( "Error in vertex order (%d)\n", n);
        return( 0); }
    fscanf( fin, "%d\n", &nadj);
    graph[i].nadj = nadj;
    if( nadj != 0)
    {   if( (graph[i].adjlist = (int *)malloc( nadj * sizeof( int))) == NULL)
        {   printf( "Error in  adjlist malloc %d\n", i);
            return( 0); }
        for( j=0; j<nadj; j++)
            fscanf( fin, "%d\n", &(graph[i].adjlist[j]));
    }
}
```


Readgrf (3)

```
    fclose( fin);
    return( 1);
}
```

Width visit

FIFO queue with two operations:

- Insert
- Extract :–1 if empty

Instead of coloring uses a vector *visited*

97

bfs

```
void bfs( int root)
{   int      vertex, i;
    vertex = root;
    visited[vertex] = 1;
    while( vertex != -1)
    {   visit( vertex);
        for( i=0; i<graph[vertex].nadj; i++)
        {   if( visited[graph[vertex].adjlist[i]] == 0)
            {   insert( graph[vertex].adjlist[i]);
                visited[graph[vertex].adjlist[i]] = 1;
            }
        }
        vertex = extract();
    }
}
```

98

Depth visit

Again uses *visited* vector

99

dfs

```
void dfs(int root)
{  int      i;
   visited[root] = 1;
   visit(root);
   for(i=0; i<graph[root].nadj; i++)
   {  if(visited[graph[root].adjlist[i]] == 0)
         dfs(graph[root].adjlist[i]);
   }
}
```

100