Algorithms and Data Structures 2010-2011

Lesson 5: *graphs and visits*

Luciano Bononi

<bononi@cs.unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

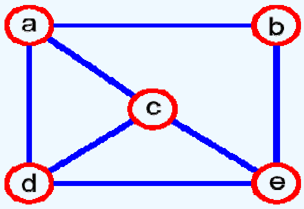
13/05/2011, Bologna

Outline of the lesson

- **Graphs**
  - Principles
  - Representations
    - Adjacency list
    - Adjacency matrix
    - Adjacency set (implemented as arrays)
  - Traversing graphs
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)
    - Connected components

Graphs: **definition**

- **DEFINITION:** a graph G = (V,E) is composed of
  - V: set of **vertices**
  - $E \subset V \times V$: set of **edges** connecting the vertices
- An **edge** e = (u,v) is a pair of vertices
- **Undirected graph**: a graph G = (V,E) in which every edge is undirected

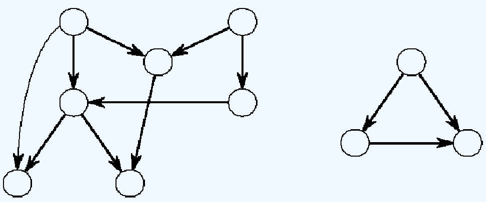


V= {a,b,c,d,e}

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

Graphs: **definition**

- **Directed graph** or **digraph**: a graph G = (V,E) in which E is a set of **ordered pairs** of vertices, called directed edges, arcs, or arrows



Graphs: **undirected** and **directed**

- **Undirected graph**
  - G=(V,E) with V={1, 2, 3, 4} and

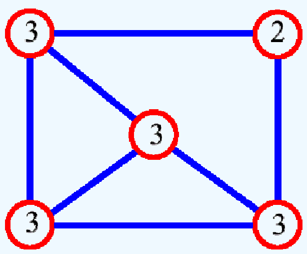
    E={[1, 2], [1, 3], [2, 4], [3, 2], [4, 2], [4, 3]}

- **Directed graph**
  - G=(V,E) with V={1, 2, 3, 4} and

    E={(1, 2), (1, 3), (2, 4), (3, 2), (4, 2), (4, 3)}

Graphs: **terminology**

- **adjacent vertices**: connected by an edge
- **degree** (of a vertex): # of adjacent vertices



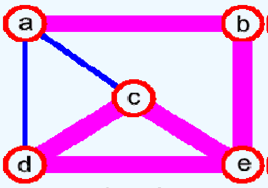
$$\sum_{v \in V} \deg(v) = 2(\#\text{ of edges})$$

"Since adjacent vertices each count the adjoining edge then it will be counted twice"

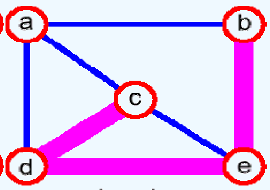
- **path**: sequence of vertices V_1 ,V_2 ,. . .V_k such that consecutive vertices V_i and V_{i+1} are adjacent

Graphs: **terminology**

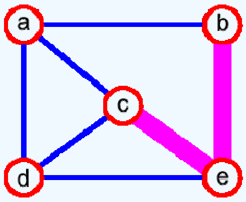
- **simple path**: no repeated vertices



a b e d c



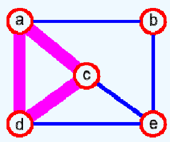
b e d c



b e c

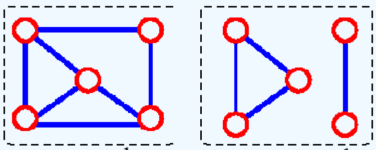
Graphs: **terminology**

- **cycle**: simple path, except that the **last vertex is the same as the first vertex**



a c d a

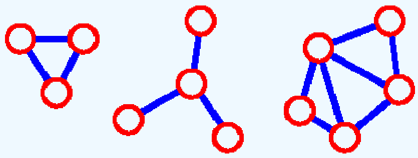
- **connected graph**: **any two** vertices are connected by some path



connected not connected

Graphs: **terminology**

- **sub-graph**: subset of vertices and edges forming a graph
- **connected component**: maximal connected sub-graph
  - two vertices are in the same connected component **if and only if** there exist a path between them
  - e.g., the graph below has 3 connected components

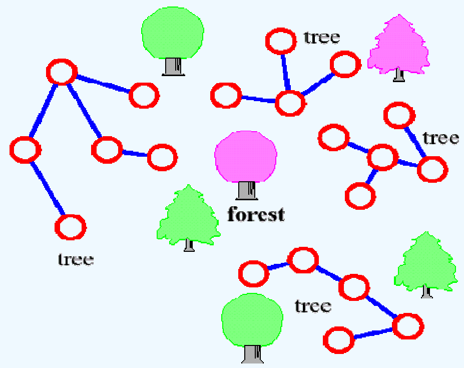


Graphs: **terminology**

- A **direct graph** is called in **strongly connected** if for every pair of vertices **u** and **v** there is a path from **u** to **v** and from **v** to **u**
- The **strongly connected components** of a directed graph are its maximal strongly connected sub-graphs
- These form a partition of the graph

Graphs: **terminology**

- **tree**: connected graph without cycles
- **forest**: collection of trees



Graph ADT: some **operations**

- **G = graph, v = vertex, e=edge**
- **Creates** a new Graph — Create(G)
- Returns **True** if the Graph is empty — Empty(G)

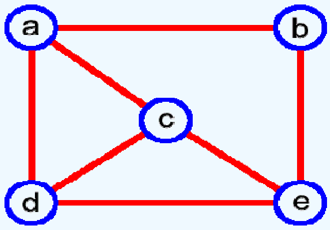
  or **False** if it has at lest one vertex
- **Inserts** a new vertex — InsVertex(G, v)
- **Inserts** a new edge — InsEdge(G, v_1, v_2)
- **Deletes** an existing vertex — DelVertex(G, v)
- **Deletes** an existing edge — DelEdge(G, v_1, v_2)
- Returns the **set** of **adjacent** vertices — AdjSet(G, v)

...

Graphs ADT: **implementation – adjacency matrix**

Implementation based on adjacency matrix

- Matrix M with entries for all pairs of vertices
  - M[i,j] = true *if there is an edge (i,j) in the graph*
  - M[i,j] = false *if there is **no** edge (i,j) in the graph*
- Space = $O(n^2)$ with n = number of vertices in the graph



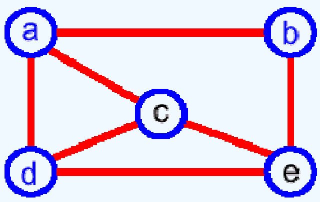
	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

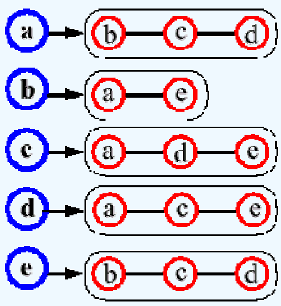
Graphs ADT: **implementation – adjacency list**

Implementation based on adjacency list

- The **adjacency list** of a vertex v: the sequence of vertices adjacent to v
- The graphs is obtained by the adjacency lists of all its vertices

$$\text{Space} = \Theta(n + \sum \deg(v)) = \Theta(n+m)$$





Graphs ADT: **implementation – adjacency set**

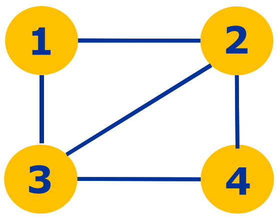
Implementation based on adjacency set

- Implemented using two **vectors**
  - **First** vector for **vertices**
  - **Second** vector for **edges**

- **EXAMPLE:**
  - **Undirected graph**
    - G=(V,E) with V={1, 2, 3, 4} and

      A={**[**1, 2**]**, **[**1, 3**]**, **[**2, 4**]**, **[**3, 2**]**, **[**4, 2**]**, **[**4, 3**]**}

Graphs ADT: implementation – adjacency set



1	**1**
2	**3**
3	**6**
4	**9**
5	**11**

VERTICES

1	**2**	A(1)
2	**3**	
3	**1**	A(2)
4	**3**	
5	**4**	
6	**1**	A(3)
7	**2**	
8	**4**	
9	**2**	A(4)
10	**3**	

EDGES

- Space complexity?

Graphs ADT: implementation

- **EXERCISE**

  What is the cost of the Graph ADT operations seen before in presence of an implementation based on:

  1. **Adjacency matrix**
  2. **Adjacency list**
  3. **Adjacency set (using arrays)**

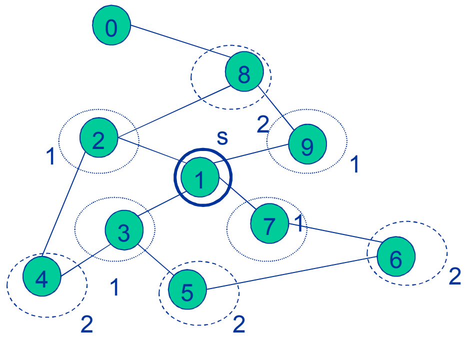
Graphs **traversal**

- The **graph traversal** refers to the problem of visiting all vertices in a graph in a particular manner

- Two common graph traversal algorithms:
  - **Breadth-First Search** (BFS)
    - application example: finds the shortest paths in an unweighted (?) graph
  - **Depth-First Search** (DFS)
    - application example: finds strongly connected components

Graphs traversal: **Breadth-First Search (BFS)**

- Given any source vertex **s**, the BFS visits the other vertices at **increasing distances** away from **s**
- In doing so, the BFS discovers paths from **s** to other vertices
- What do we mean by "*distance*"? **The number of edges on a path from s**



Example:

Consider s=vertex 1

Nodes at distance 1?

2, 3, 7, 9

Nodes at distance 2?

8, 6, 5, 4

Nodes at distance 3?

0

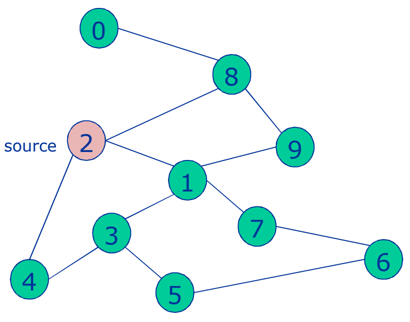
Graphs traversal: **BFS**

```
Procedure BFS(G : graph; u : vertex)
   Make(Q); Enqueue(Q, u);
   while not Empty(Q) do
      u := Dequeue(Q);
      /* visit the vertex u and mark it as visited */
      for each v in AdjSet(G, u)
            /* visit the edge (u, v) */
            if (v is not marked) and (v is not in Q) then
                  Enqueue(Q, v)
```

Graphs traversal: **BFS**

```
Procedure BFS(G : graph; u : vertex)
   Make(Q); Enqueue(Q, u);
   while not Empty(Q) do
      u := Dequeue(Q);
      /* visit the vertex u and mark it as visited */
      for each v in AdjSet(G, u)
            /* visit the edge (u, v) */
            if (v is not marked) and (v is not in Q) then
                  Enqueue(Q, v)
```


Graphs traversal: **BFS, example**



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

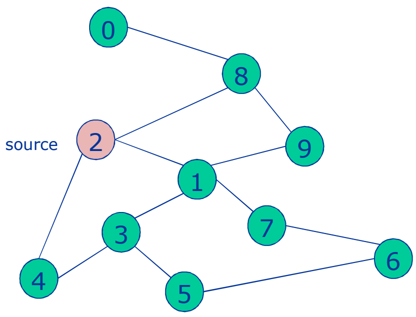
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Initialize visited table (all False)

Q = { }

Initialize Q to be empty

Graphs traversal: **BFS, example**



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

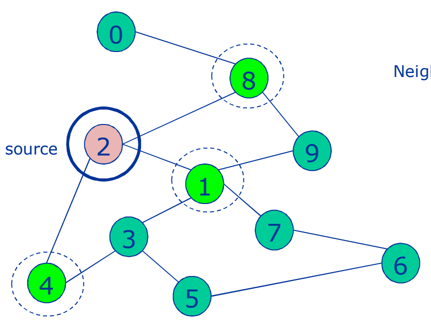
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.

Graphs traversal: **BFS, example**



Adjacency List

Neighbors →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

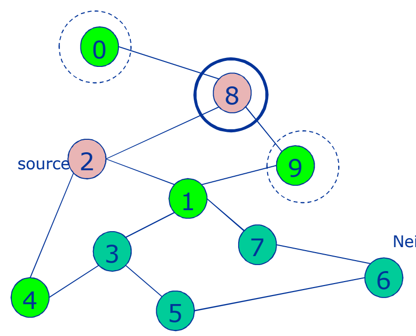
Mark neighbors as visited.

Q = {2} → { 8, 1, 4 }

Dequeue 2.

Place all unvisited neighbors of 2 on the queue

Graphs traversal: **BFS, example**



Neighbors → 8

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	T

Mark new visited Neighbors.

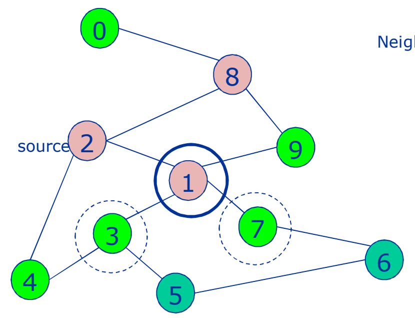
Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.

-- Place all unvisited neighbors of 8 on the queue.

-- Notice that 2 is not placed on the queue again, it has been visited!

Graphs traversal: **BFS, example**



Neighbors → 1

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Mark new visited Neighbors.

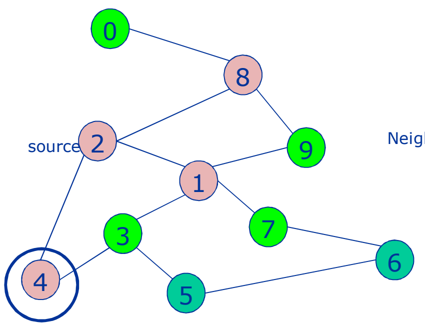
Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.

-- Place all unvisited neighbors of 1 on the queue.

-- Only nodes 3 and 7 haven't been visited yet.

Graphs traversal: **BFS, example**



Neighbors → Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

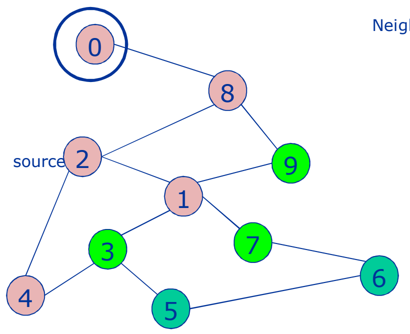
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unvisited neighbors!

Graphs traversal: **BFS, example**



Neighbors → Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

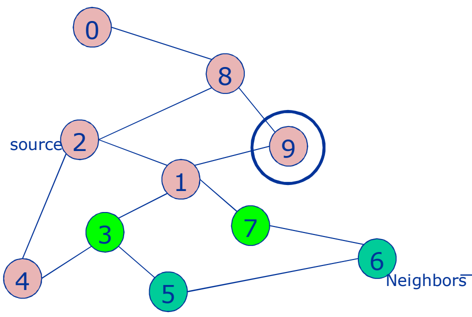
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!

Graphs traversal: **BFS, example**



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

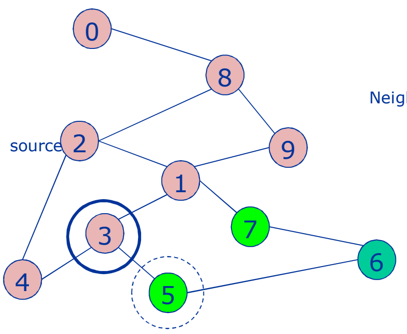
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.

-- 9 has no unvisited neighbors!

Graphs traversal: **BFS, example**



Adjacency List

Neighbors →

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

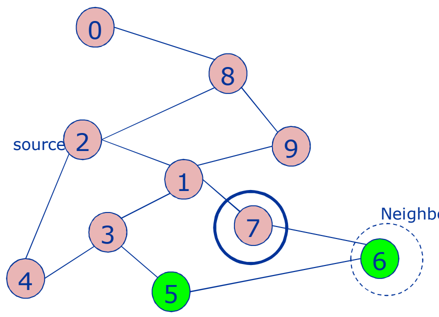
Mark new visited Vertex 5.

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.

-- place neighbor 5 on the queue.

Graphs traversal: **BFS, example**



Neighbors

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark new visited

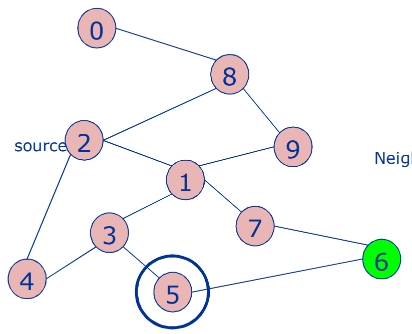
Vertex 6.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.

-- place neighbor 6 on the queue.

Graphs traversal: **BFS, example**



Neighbors

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

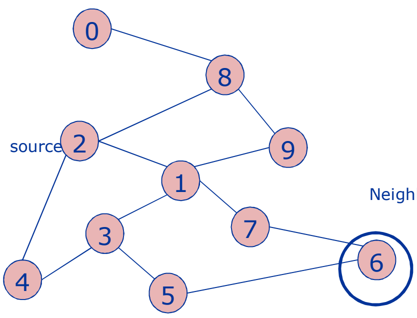
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

Dequeue 5.

-- no unvisited neighbors of 5.

Graphs traversal: **BFS, example**



Neighbors →

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

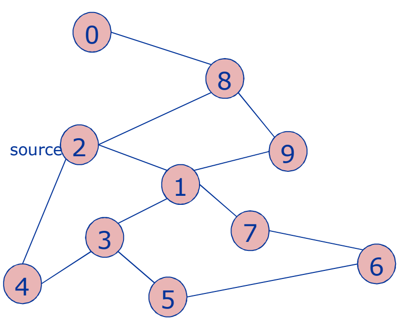
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 6 } → { }

Dequeue 6.

-- no unvisited neighbors of 6.

Graphs traversal: **BFS, example**



0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { } STOP!!! **Q is empty!!!**

What did we discover?

Look at "visited" tables.

There exists a path from source vertex 2 to all vertices in the graph

Time complexity of BFS

- Assuming: **n** = number of vertices, **m** = number of edges
- **Time complexity of BFS:**
  - **Assuming adjacency lists:** O(n+m)
    - each vertex in the graph is marked only once
    - for each vertex, the AdjSet is visited only once
  - **Assuming adjacency matrix:** $O(n^2)$
    - for each vertex it is necessary to scan the whole vector of adjacency
    - it is independent from the number of edges

Graphs traversal: **Depth-First Search (DFS)**

- DFS will continue to visit neighbors in a **recursive** pattern
- **Main principle**:
  - whenever we visit **v** from **u**, we recursively visit all unvisited neighbors of **v**
  - then we backtrack (return) to **u**

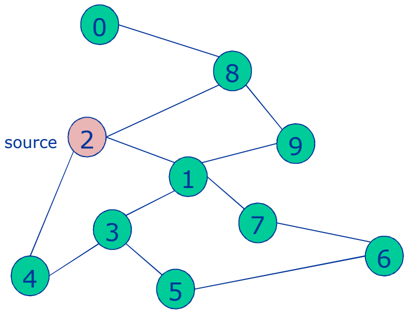
Graphs traversal: **DFS**

```
Procedure DFS(G : graph; u : vertex)
      /* visit the vertex u and mark it as visited */
      for each v in AdjSet(G, u)
            /* visit the edge (u, v) */
            if (v is not marked) then  DFS(G, v)
```


Graphs traversal: **DFS**

```
Procedure DFS(G : graph; u : vertex)
      /* visit the vertex u and mark it as visited */
      for each v in AdjSet(G, u)
            /* visit the edge (u, v) */
            if (v is not marked) then  DFS(G, v)
```

Graphs traversal: **DFS**, **example**



Adjacency List

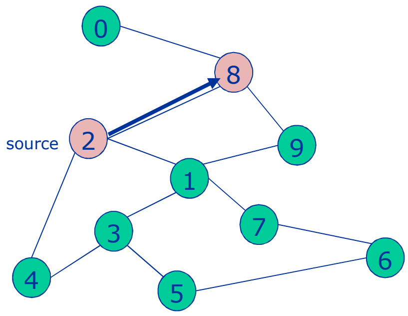
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Mark 2 as visited

Graphs traversal: **DFS**, **example**



Adjacency List

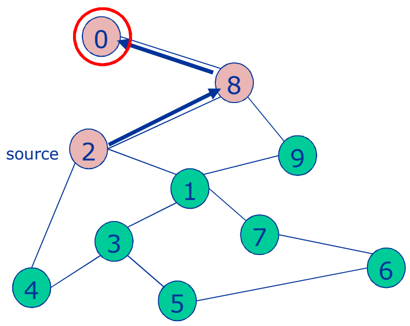
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Mark 8 as visited

Graphs traversal: DFS, example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

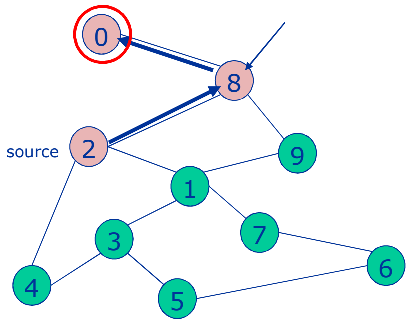
Visited Table (T/F)

0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Mark 0 as visited

Graphs traversal: **DFS, example**

Back to 8



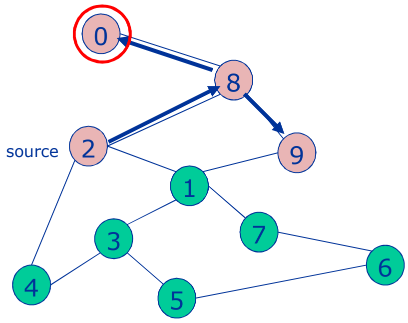
Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Graphs traversal: **DFS, example**



Adjacency List

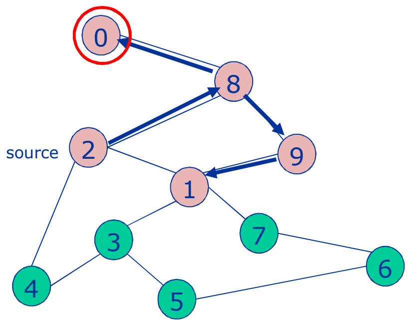
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	T

Mark 9 as visited

Graphs traversal: **DFS, example**



Adjacency List

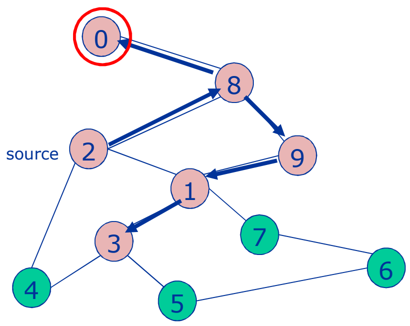
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	T

Mark 1 as visited

Graphs traversal: **DFS, example**



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

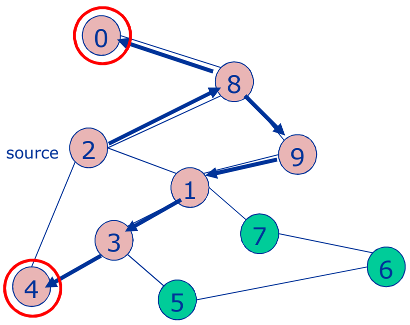
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	F
5	F
6	F
7	F
8	T
9	T

Pred

Mark 3 as visited

Graphs traversal: **DFS, example**



Adjacency List

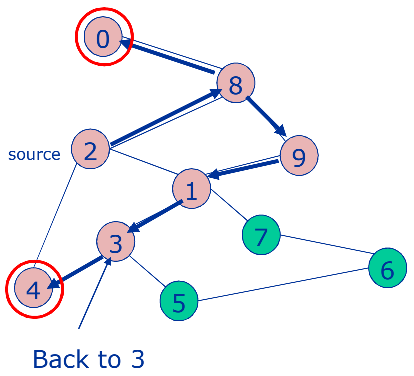
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

Mark 4 as visited

Graphs traversal: **DFS**, **example**



Adjacency List

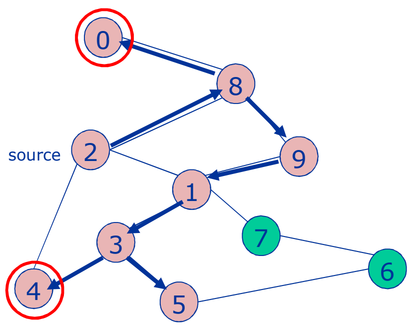
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

		Pred
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

Pred

Graphs traversal: **DFS**, **example**



Adjacency List

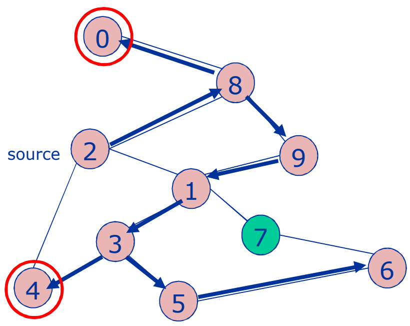
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T

Mark 5 as visited

Graphs traversal: **DFS, example**



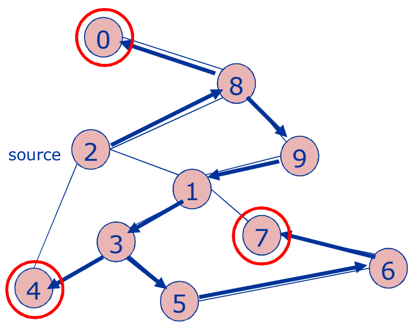
Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	F
8	T
9	T

Graphs traversal: **DFS, example**



Adjacency List

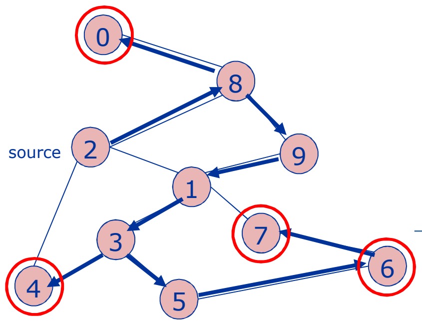
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark 7 as visited

Graphs traversal: **DFS, example**



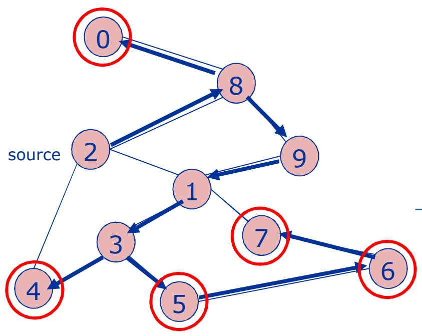
Adiacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS, example**



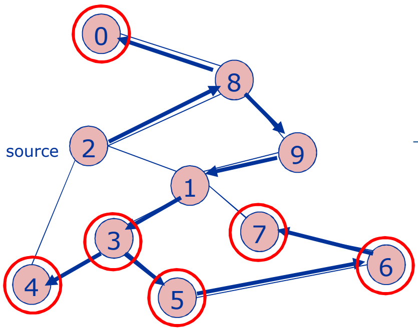
Adiacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS**, **example**



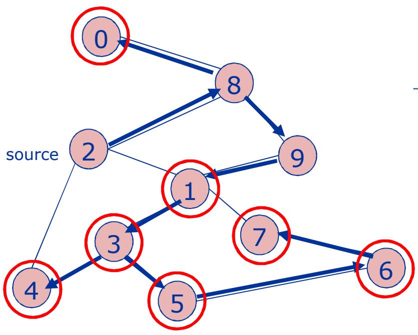
Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS**, **example**



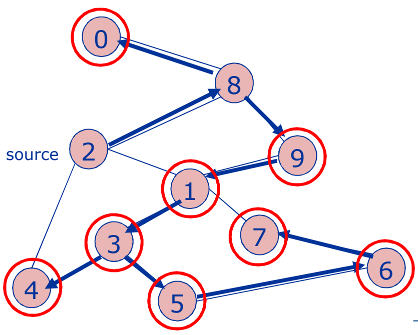
Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS**, **example**



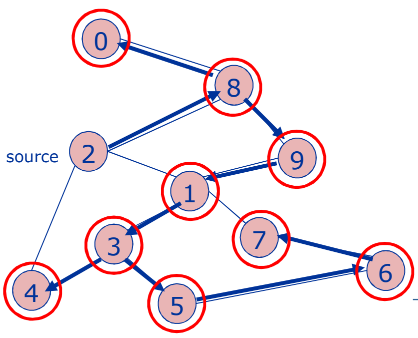
Adiacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS**, **example**



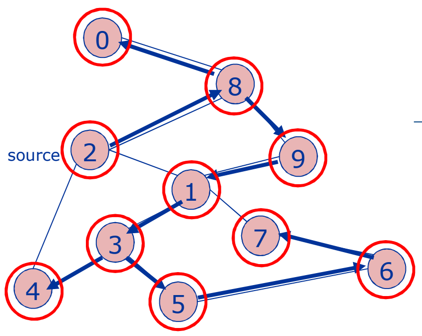
Adiacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Graphs traversal: **DFS, example**



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Traversal completed!

Time complexity of **DFS**

- Assuming: **n** = number of vertices, **m** = number of edges
- **Time complexity of DFS:**
  - **Assuming adjacency lists:** O(n+m)
    - we never visited a vertex more than once
    - we had to examine all edges of the vertices
  - **Assuming adjacency matrix:** $O(n^2)$
    - for each node it is necessary to scan the whole vector of adjacency
    - it is independent from the number of edges

Graphs: **exercise**

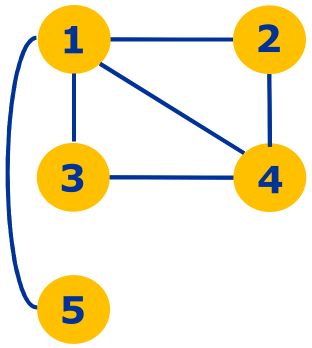
- Write the Depth-First-Search (DFS) procedure in pseudo-code. Given the undirected graph
  - G=(N, A)
    - N={1, 2, 3, 4,5}
    - A={[1, 2], [1, 3], [1,5], [3, 5], [3, 4], [4, 1], [4, 2]}

  Execute the DFS procedure starting from the vertex 5

  Plot the graph and show its representation based on adjacency set implemented using vertex and edges vectors

  Show the visited nodes and edges, assuming that the vectors are in not decreasing order

DFS exercise resolution



VERTICES

1	**1**
2	**5**
3	**7**
4	**10**
5	**13**
6	**15**

EDGES

1	**2**	A(1)
2	**3**	
3	**4**	
4	**5**	
5	**1**	A(2)
6	**4**	
7	**1**	A(3)
8	**4**	
9	**5**	
10	**1**	A(4)
11	**2**	
12	**3**	
13	**1**	A(5)
14	**3**	

DFS(5) = **5** (5, 1) **1** (1, 2) **2** (2, 1) (2, 4) **4** (4, 1)(4, 2) (4, 3) **3** (3, 1) (3, 4) (3, 5) (1, 3) (1, 4) (1, 5) (5, 3)

Graphs: exercise

- Write the Breadth-First-Search (BFS) procedure in pseudo-code. Given the directed graph
  - G=(N, A)
    - N={1, 2, 3, 4,5}
    - A={(1, 2), (1, 3), (1,5), (3, 5), (3, 4), (4, 1), (4, 2), (5, 2)}

  Execute the BFS procedure starting from the vertex 4

  Plot the graph and show its representation based on adjacency set implemented using vertex and edges vectors

  Show the visited nodes and edges, assuming that the vectors are in decreasing order

Connected components

- **EXERCISE:** write the pseudo-code to find the number and the composition of *connected components* in a given graph G

- **SUGGESTION**: the algorithm could be based on a slightly modified version of the DFS traversal

References

- Part of this material is inspired / taken by the following freely available resources:
  - http://www.cs.aau.dk/~simas/ad01/slides.html
  - http://www.cs.ust.hk/~huamin/COMP171/index.htm

Algorithms and Data Structures 2010 - 2011

Lesson 5: *graphs and visits*

Luciano Bononi
<bononi@cs.unibo.it>
http://www.cs.unibo.it/~bononi/
(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

29/04/2011, Bologna