Depth-First Search

CSE 2011
Fall 2009

Depth-First Search (DFS)

- DFS is another popular graph search strategy
  - Idea is similar to pre-order traversal (visit node, then visit children recursively)
- DFS will continue to visit neighbors in a recursive pattern
  - Whenever we visit v from u, we recursively visit all unvisited neighbors of v. Then we backtrack (return) to u.

DFS Algorithm

```
Algorithm DFS(s)
1.   for each vertex v
2.       do flag[v] := false;
3.   RDFS(s);

Algorithm RDFS(v)
1.   flag[v] := true;
2.   for each neighbor w of v
3.       do if flag[w] = false
4.           then RDFS(w);
```

Flag all vertices as not visited

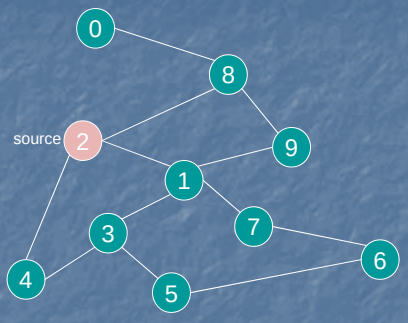
Flag yourself as visited

For unvisited neighbors, call *RDFS*(*w*) recursively

We can also record the paths using *prev*[ ].
Where do we insert the code for *prev*[ ]?

3

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

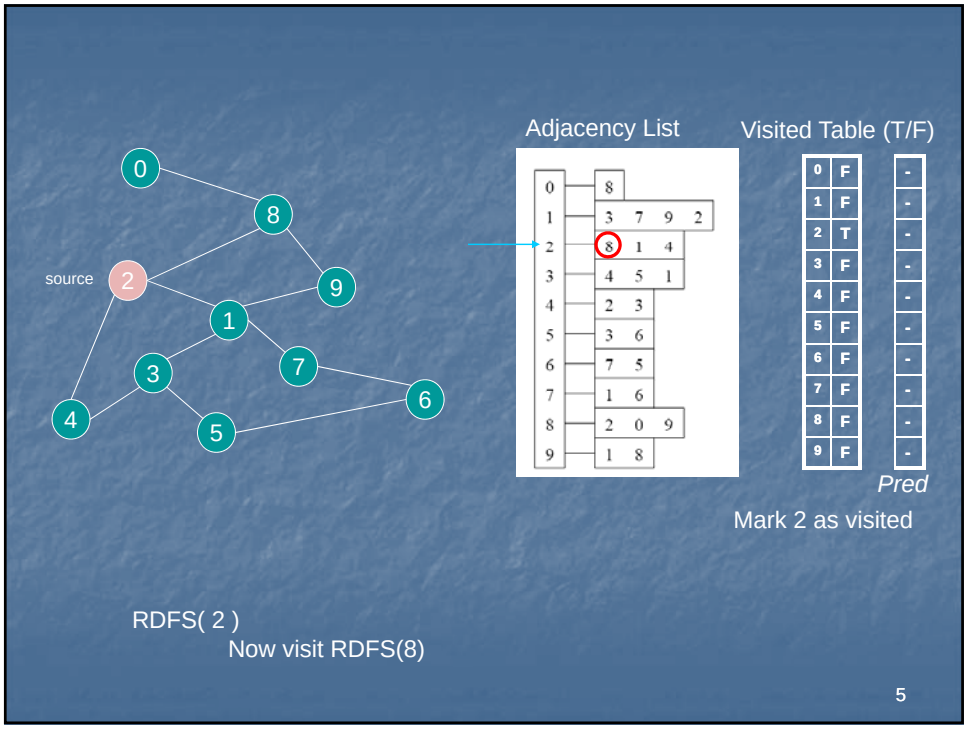
-
-
-
-
-
-
-
-
-
-

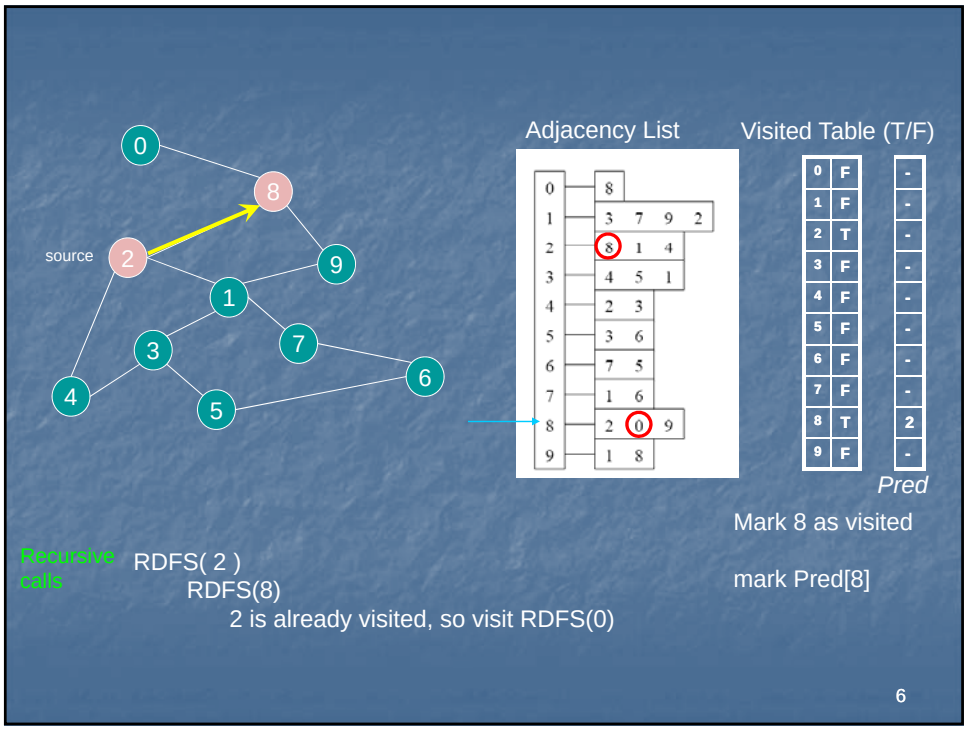
Pred

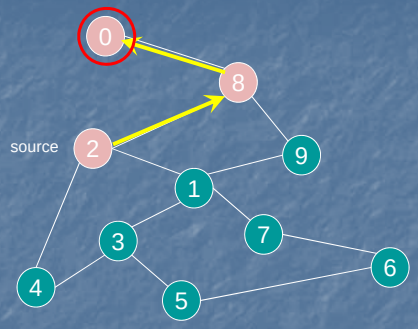
Initialize visited table (all False)

Initialize Pred to -1

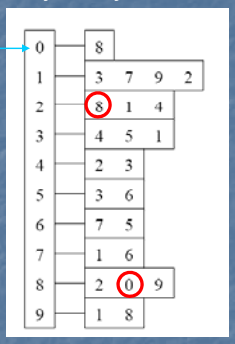
4


Adjacency List
Visited Table (T/F)
source
Pred
Mark 2 as visited
RDFS(2)
Now visit RDFS(8)
5


Adjacency List
Visited Table (T/F)
source
Pred
Mark 8 as visited
mark Pred[8]
Recursive calls
RDFS(2)
RDFS(8)
2 is already visited, so visit RDFS(0)
6



Adjacency List



Visited Table (T/F)

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

8
-
-
-
-
-
-
-
2
-

Pred

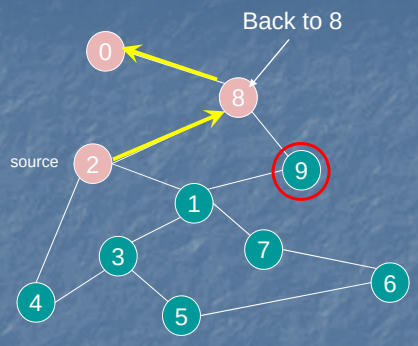
Mark 0 as visited

Mark Pred[0]

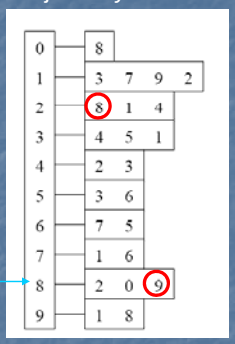
Recursive calls

RDFS(2)
RDFS(8)
RDFS(0) -> no unvisited neighbors, return to call RDFS(8)

7



Adjacency List



Visited Table (T/F)

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

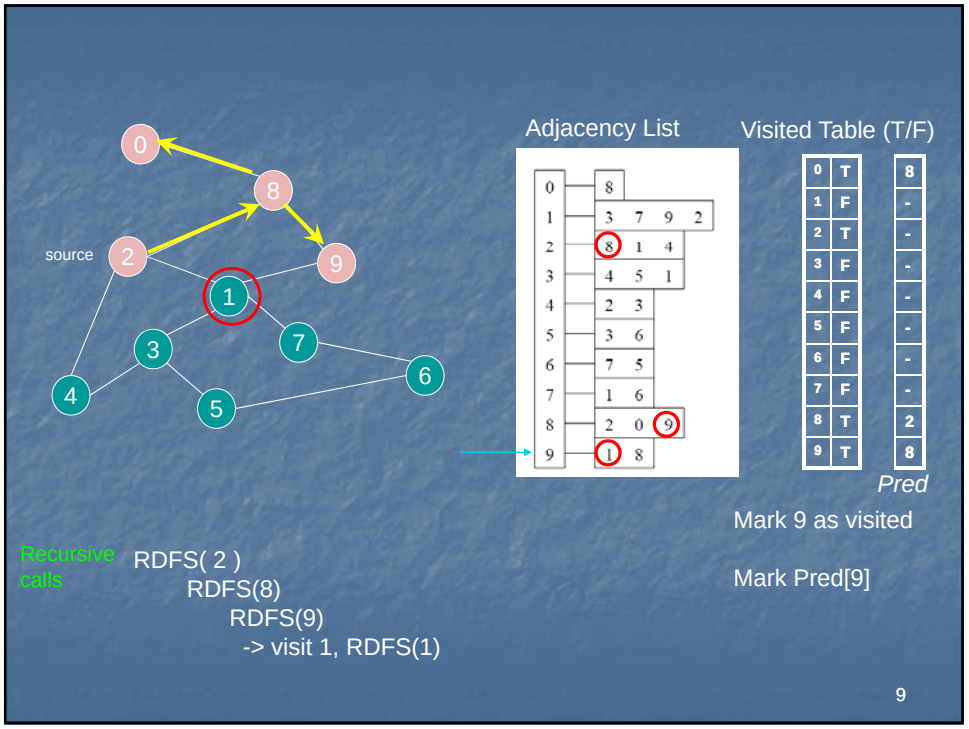
8
-
-
-
-
-
-
-
2
-

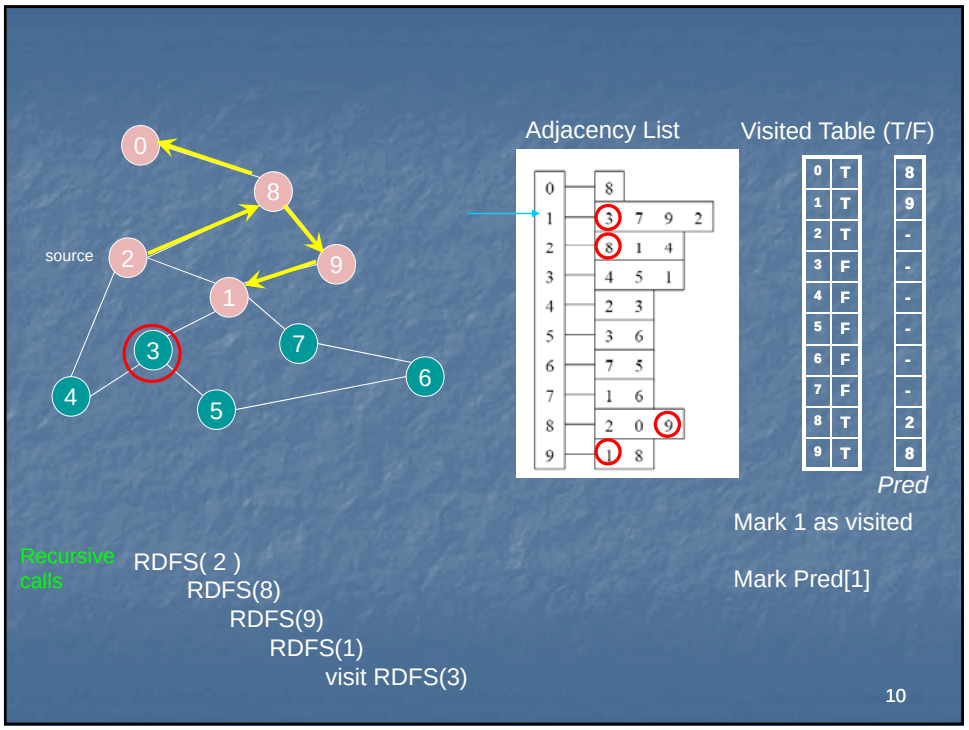
Pred

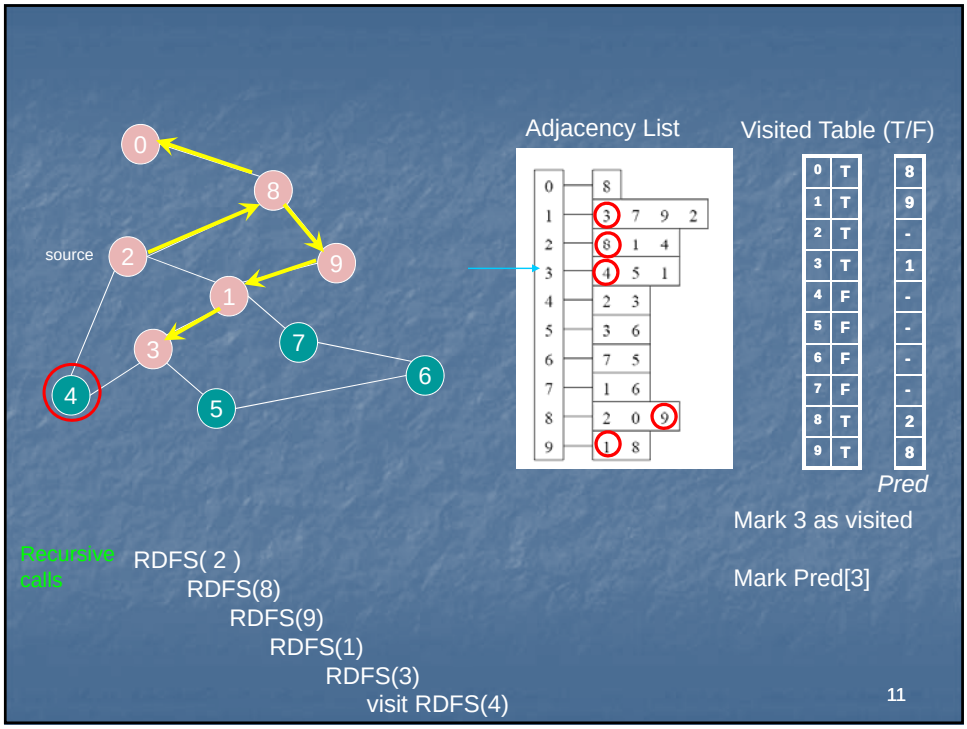
Recursive calls

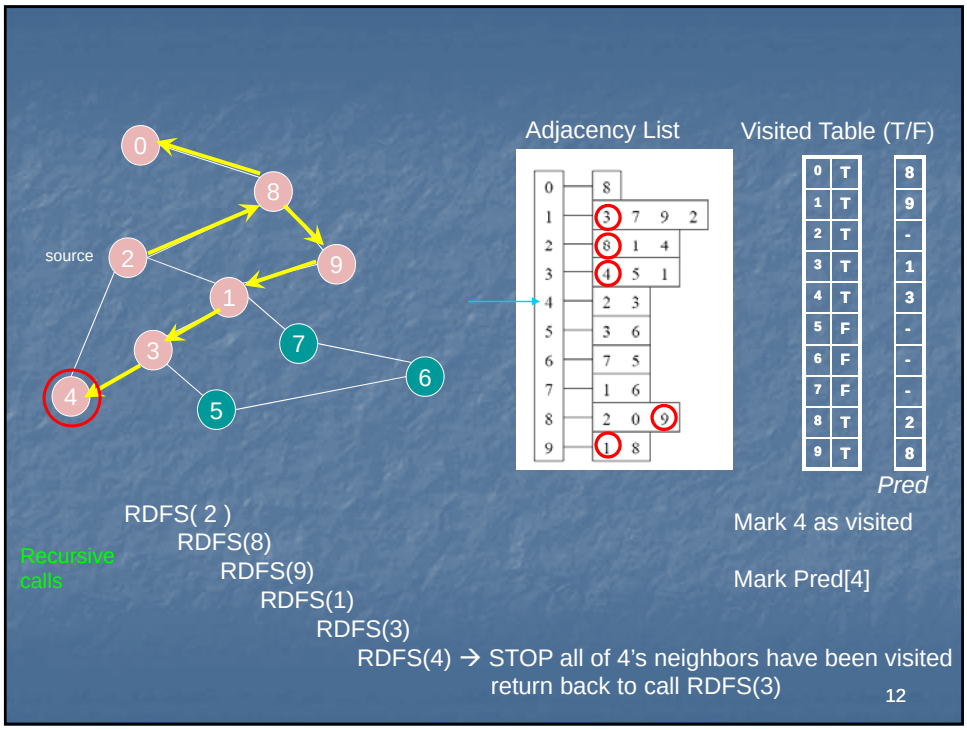
RDFS(2)
RDFS(8)
Now visit 9 -> RDFS(9)

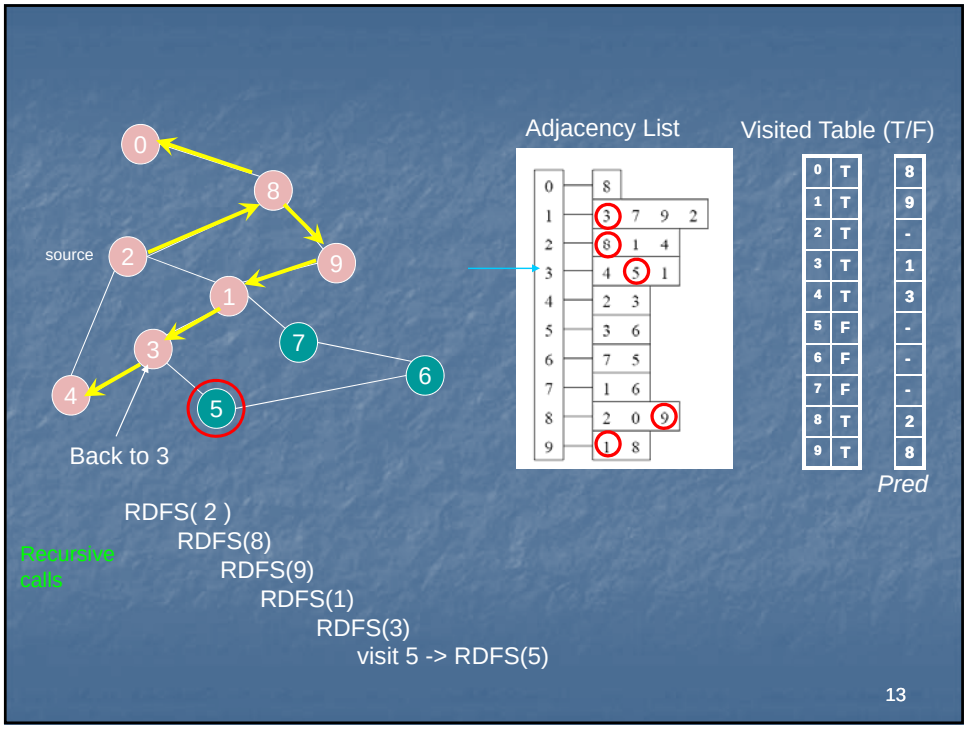
8

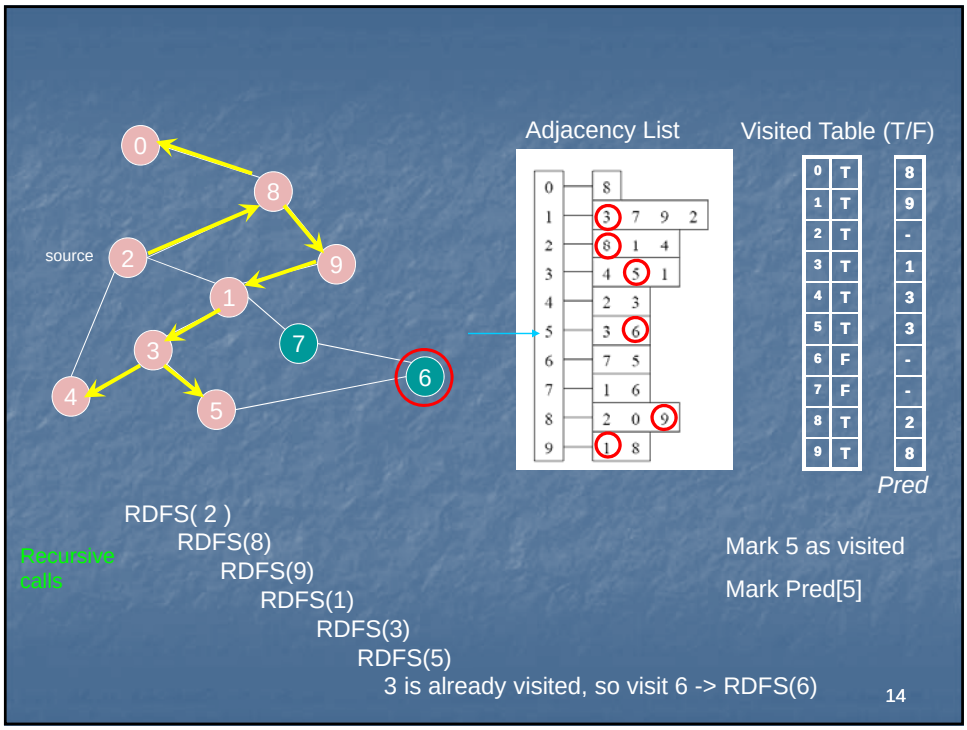

Adjacency List
Visited Table (T/F)
Pred
source
Mark 9 as visited
Mark Pred[9]
Recursive calls
RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)
9

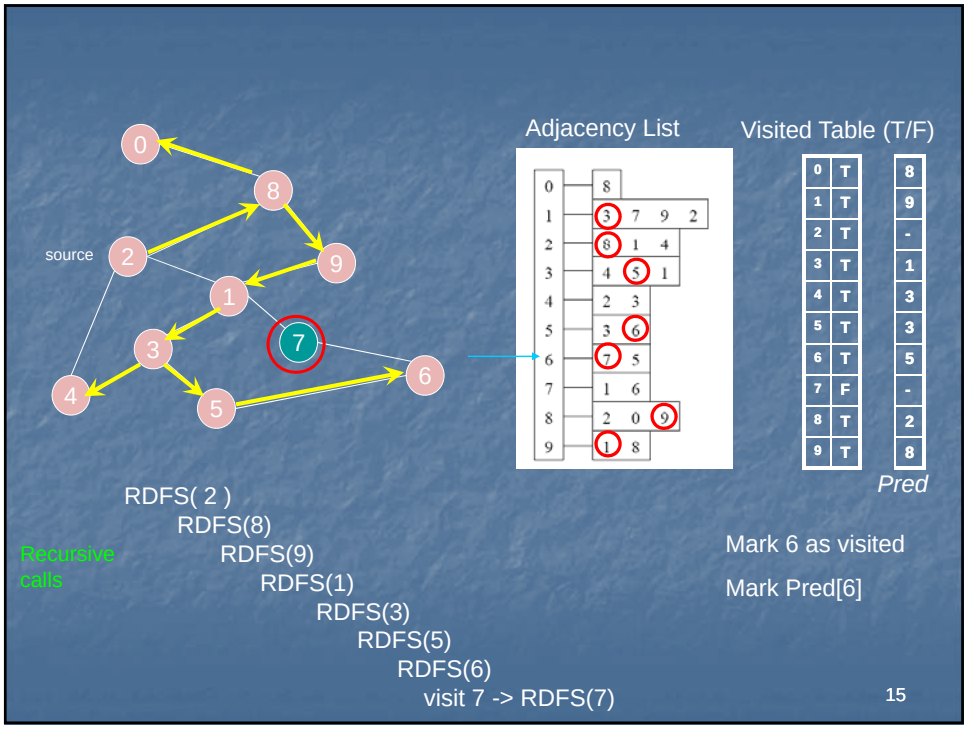

Adjacency List
Visited Table (T/F)
Pred
source
Mark 1 as visited
Mark Pred[1]
Recursive calls
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)
10

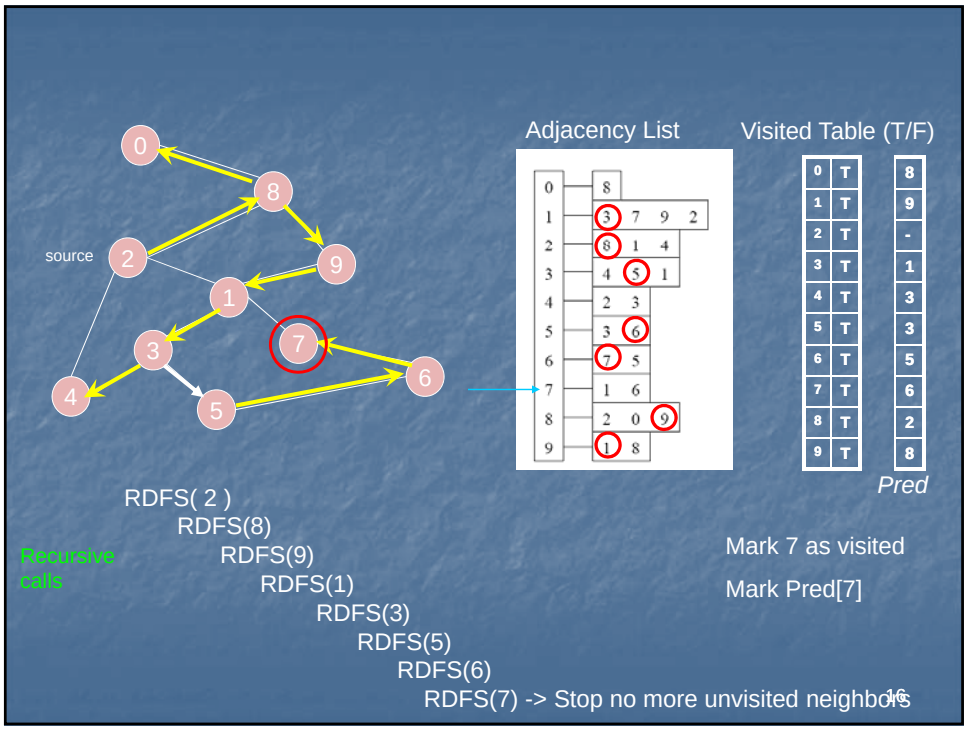

Adjacency List
Visited Table (T/F)
source
Pred
Mark 3 as visited
Mark Pred[3]
Recursive calls
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)
11

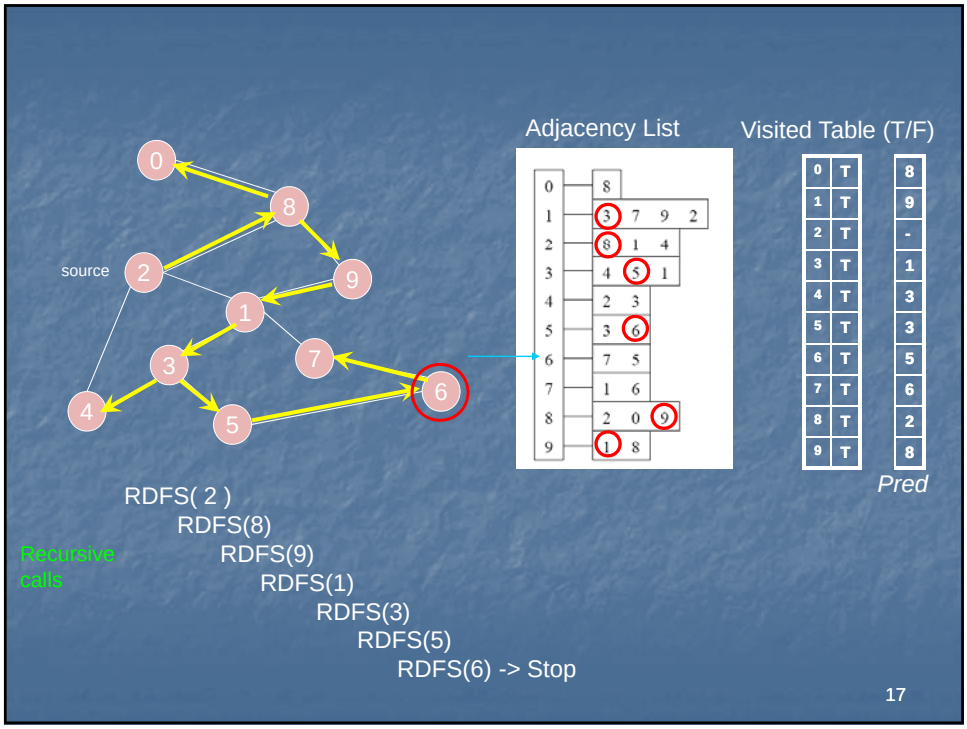

Adjacency List
Visited Table (T/F)
source
Pred
Mark 4 as visited
Mark Pred[4]
Recursive calls
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP all of 4's neighbors have been visited
return back to call RDFS(3)
12

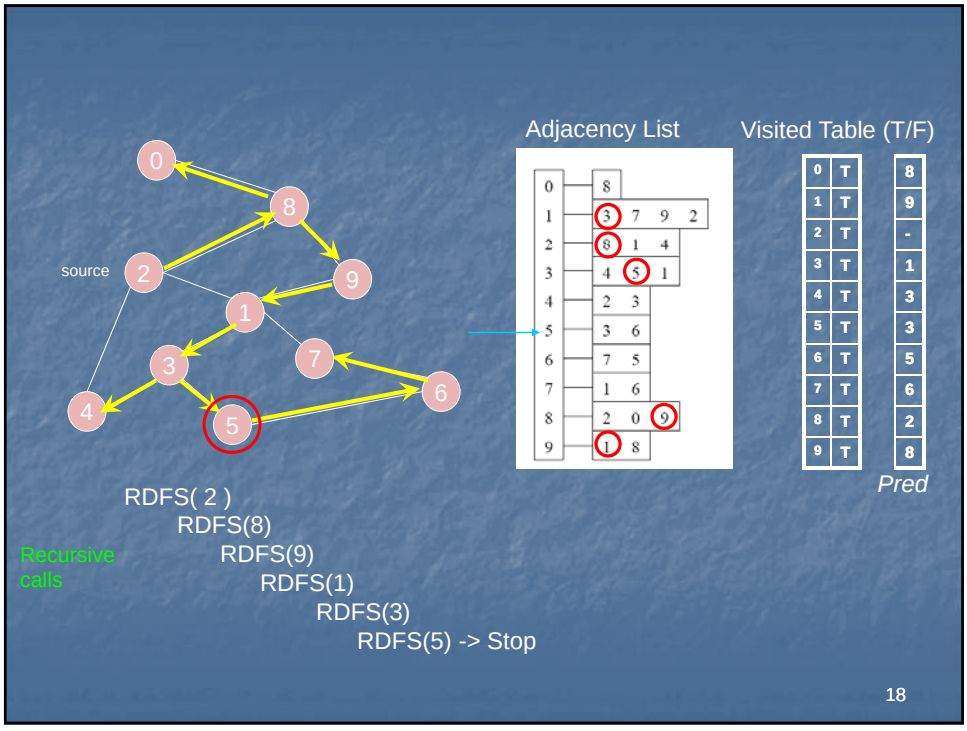

Adjacency List
Visited Table (T/F)
source
Back to 3
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit 5 -> RDFS(5)
Recursive calls
13

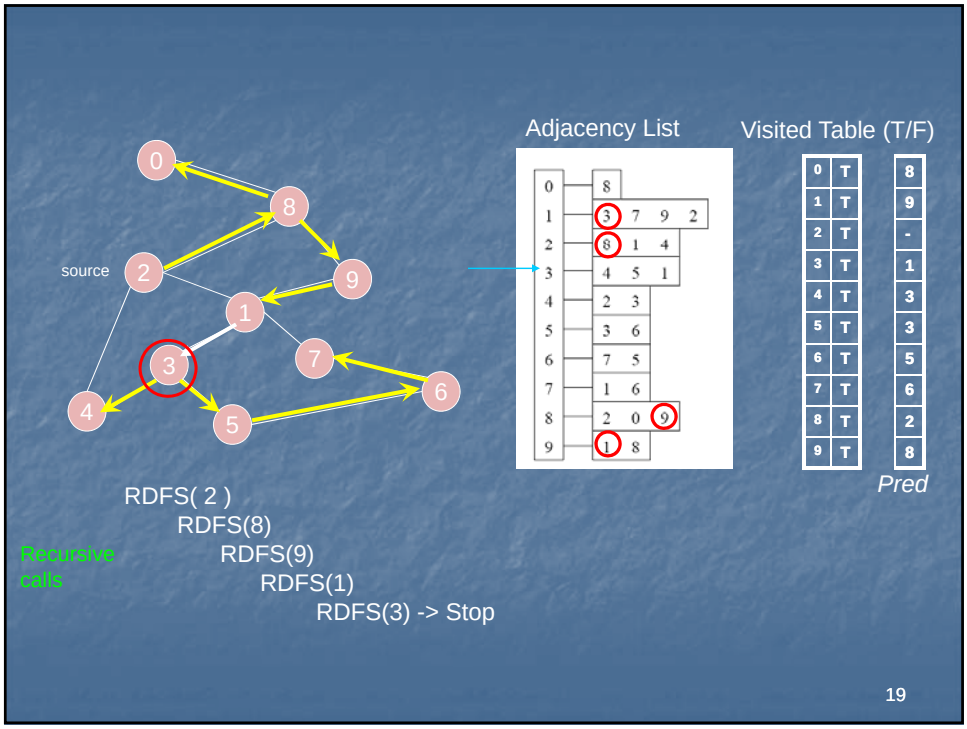

Adjacency List
Visited Table (T/F)
source
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
3 is already visited, so visit 6 -> RDFS(6)
Mark 5 as visited
Mark Pred[5]
Recursive calls
14

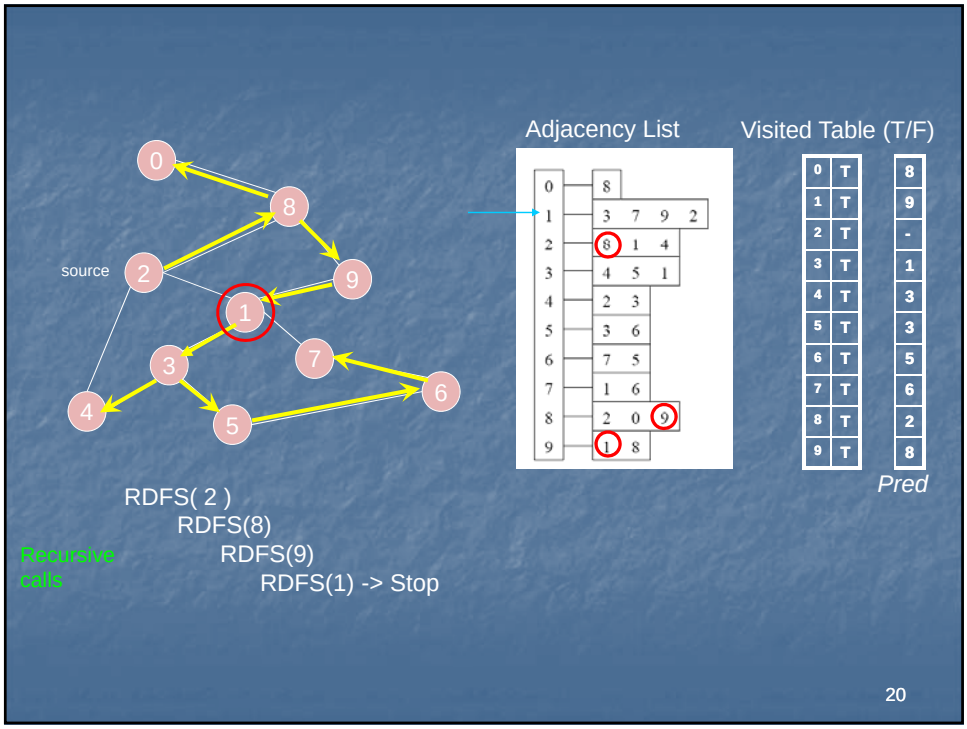

Adjacency List
Visited Table (T/F)
source
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)
Recursive calls
Mark 6 as visited
Mark Pred[6]
15

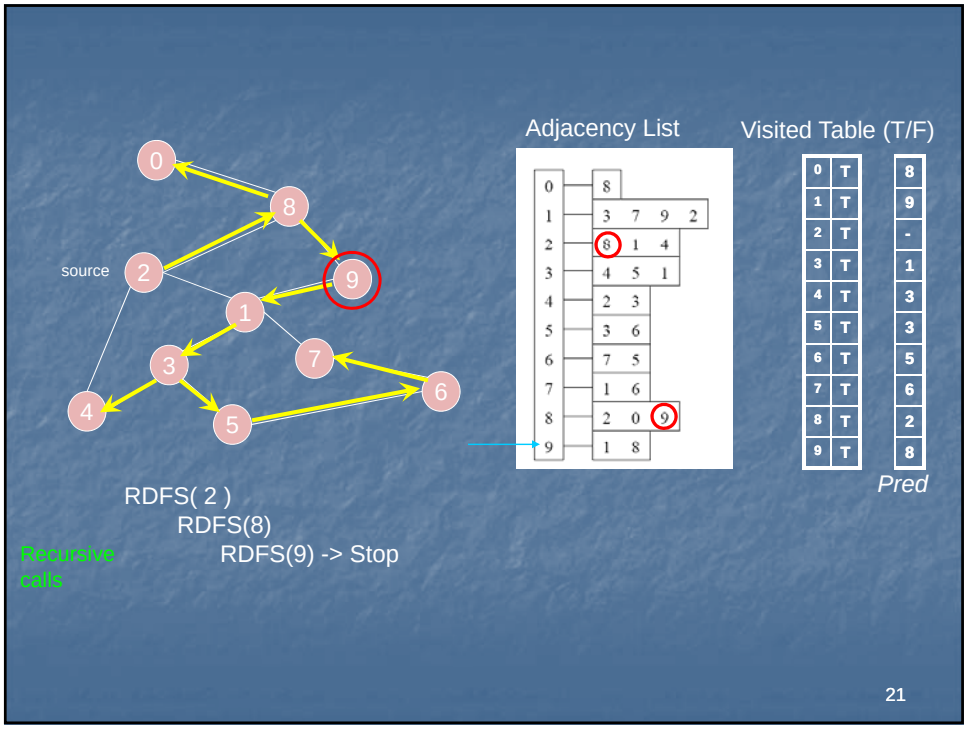

Adjacency List
Visited Table (T/F)
source
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) -> Stop no more unvisited neighbors
Recursive calls
Mark 7 as visited
Mark Pred[7]
16

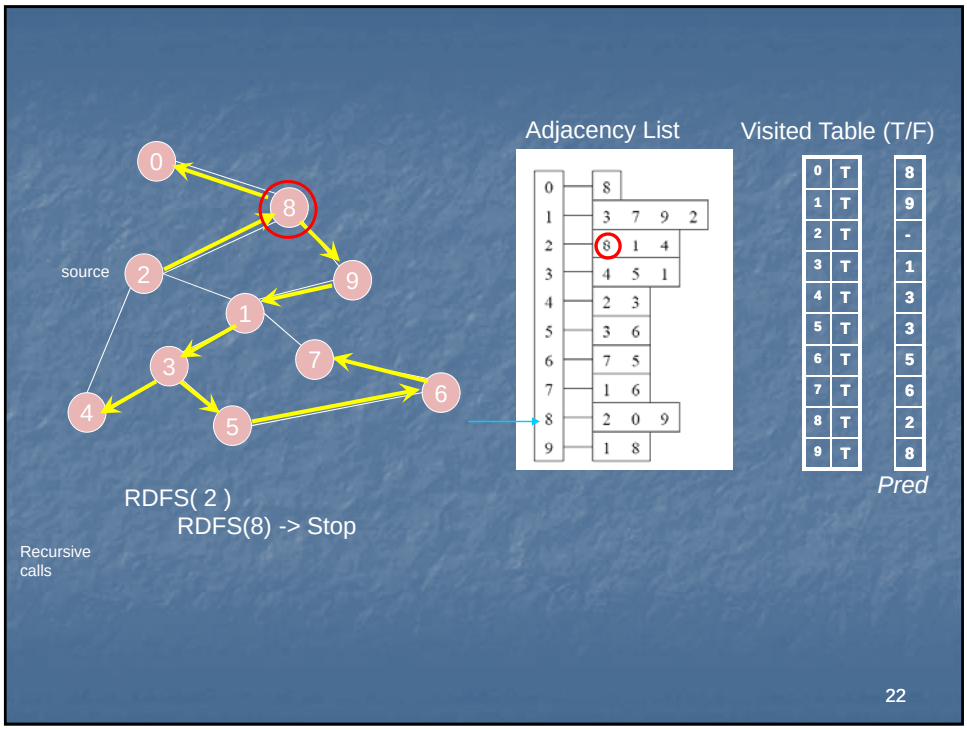

Adjacency List
Visited Table (T/F)
source
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop
Recursive calls
17


Adjacency List
Visited Table (T/F)
source
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> Stop
Recursive calls
18

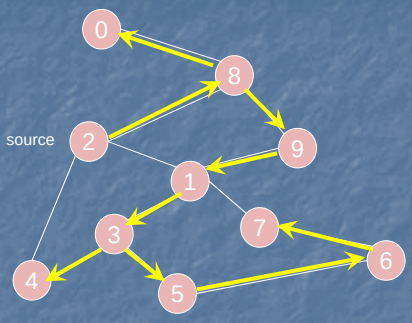

Adjacency List
Visited Table (T/F)
source
0
8
9
2
1
3
7
6
4
5
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> Stop
Recursive calls
19


Adjacency List
Visited Table (T/F)
source
0
8
9
2
1
3
7
6
4
5
Pred
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) -> Stop
Recursive calls
20

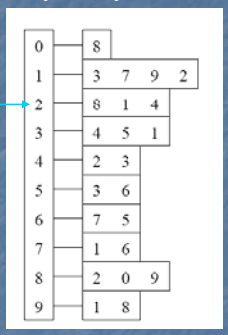

Adjacency List
Visited Table (T/F)
source
0
8
1
3 7 9 2
2
8 1 4
3
4 5 1
4
2 3
5
3 6
6
7 5
7
1 6
8
2 0 9
9
1 8
0 T
1 T
2 T
3 T
4 T
5 T
6 T
7 T
8 T
9 T
8
9
-
1
3
3
5
6
2
8
Pred
RDFS(2)
RDFS(8)
RDFS(9) -> Stop
Recursive calls
21


Adjacency List
Visited Table (T/F)
source
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
8
9
-
1
3
3
5
6
2
8
Pred
RDFS(2)
RDFS(8) -> Stop
Recursive calls
22

Example Finished



Adjacency List



Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
6
2
8

RDFS(2) -> Stop

Recursive calls finished

23

Time Complexity of DFS

- We never visited a vertex more than once.
- We had to examine the adjacency lists of all vertices.
  - $\Sigma_{vertex\ v}$ degree(v) = 2E
- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)
  - O(V + E)

24

Enhanced DFS Algorithm

- What if a graph is not connected (strongly connected)?
  - Use an enhanced version of DFS, which is similar to the enhanced BFS algorithm

```
BFSearch( G ) {
i = 1;     // component number
for every vertex v
    flag[v] = false;
for every vertex v
    if ( flag[v] == false ) {
        print ( "Component " +  i++ );
        BFS( v );
    }
}
```

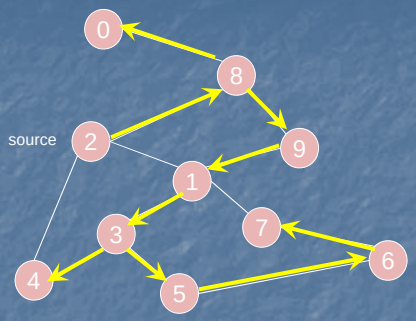
25

Applications of DFS

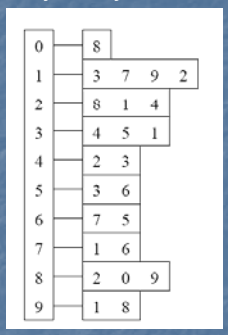
- Is there a path from source *s* to a vertex *v*?
- Is an undirected graph connected?
- Is a directed graph strongly connected?
- To output the contents (e.g., the vertices) of a graph
- To find the connected components of a graph
- To find out if a graph contains cycles and report cycles.
- To construct a DSF tree/forest from a graph

26

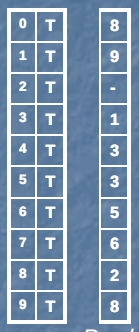
DFS Path Tracking



Adjacency List



Visited Table (T/F)



Pred

DFS find out path too

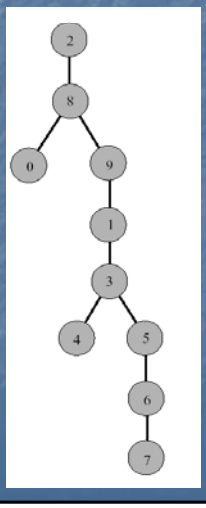
Algorithm *Path*(w)
1. **if** $pred[w] \neq -1$
2. **then**
3. *Path*$(pred[w])$;
4. output w

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->

27

DFS Tree

Resulting DFS-tree.
Notice it is much "deeper" than the BFS tree.



Captures the structure of the recursive calls

- when we visit a neighbor w of v, we add w as child of v
- whenever DFS returns from a vertex v, we climb up in the tree from v to its parent

28

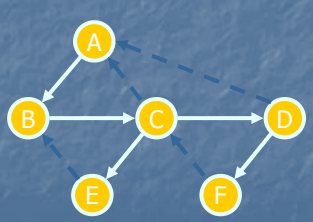
Applications – DFS vs BFS

- What can BFS do and DFS can't?
  - Finding shortest paths (in unweighted graphs)
- What can DFS do and BFS can't?
  - Finding out if a connected undirected graph is *biconnected*
    - A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph

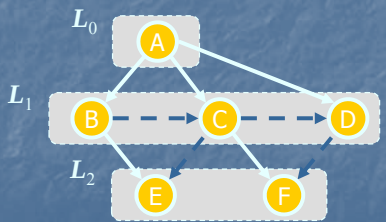
29

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS

30

Next time …

- Review — Dec. 8
- Final exam — Dec. 11

31