DFS : Depth-First Search

- DFS is another popular search strategy.
- It can do certain things that BFS cannot do. We will discuss some of these algorithms in COMP 271 (so you cannot get rid of DFS after COMP171).
- DFS idea :
  - Whenever we visit a vertex *v* from another vertex *u*, we **recursively** visit a neighbor of *v* that has not been visited before until all neighbors of v have been visited. Then we ***backtrack*** (return) to *u*.

Algorithm

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

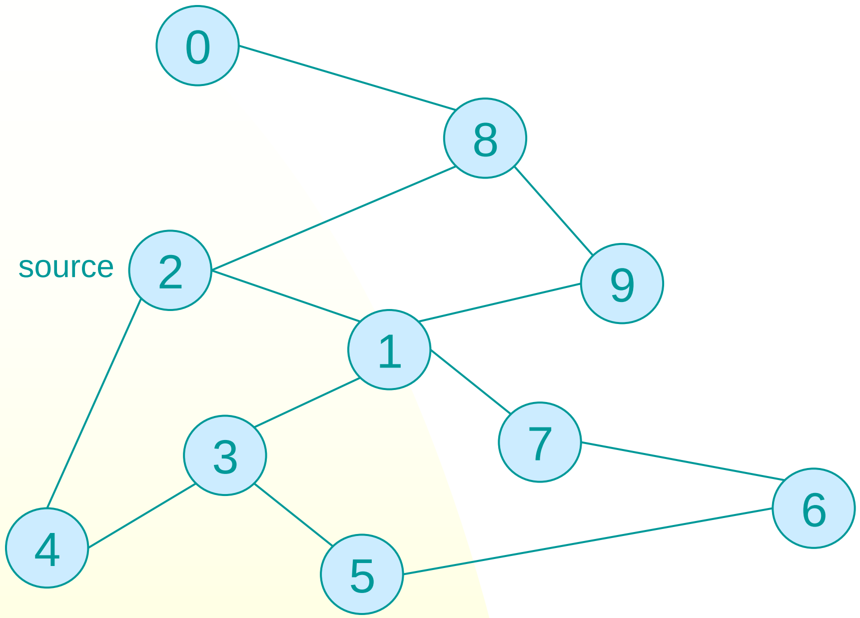
Flag all vertices as not visited

Algorithm *RDFS*(v)
1. $flag[v]$:= true;
2. **for** each neighbor $\boxed{w}$ of v
3. **do if** $flag[w]$ = false
4. **then** $\boxed{RDFS(w);}$

Visit *v*, and mark *v* as visited.

For each unvisited neighbor.
make a recursive call *RDFS(w)*.

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

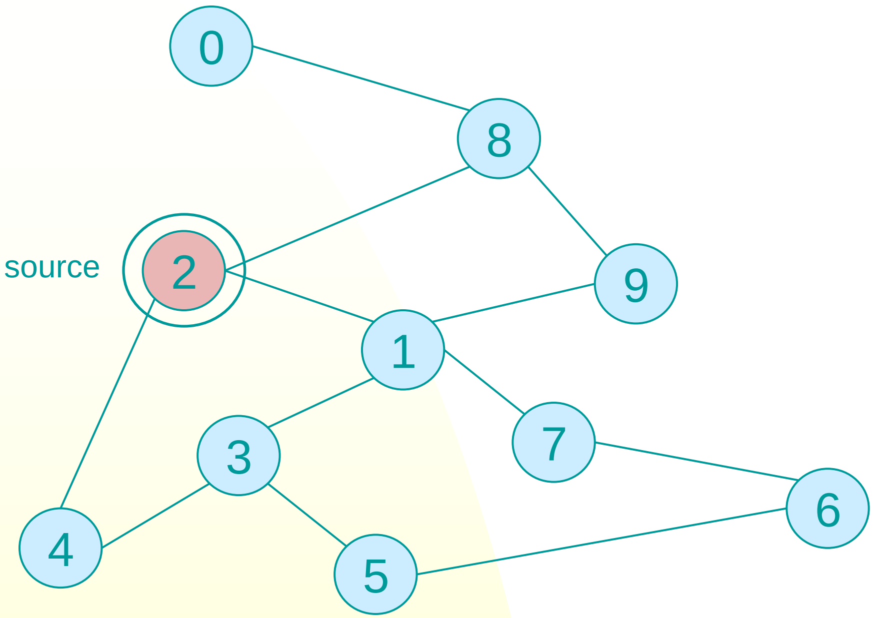
Visited Table (T/F)

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

Initialize visited table (all empty F)

Initialize Pred to -1

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

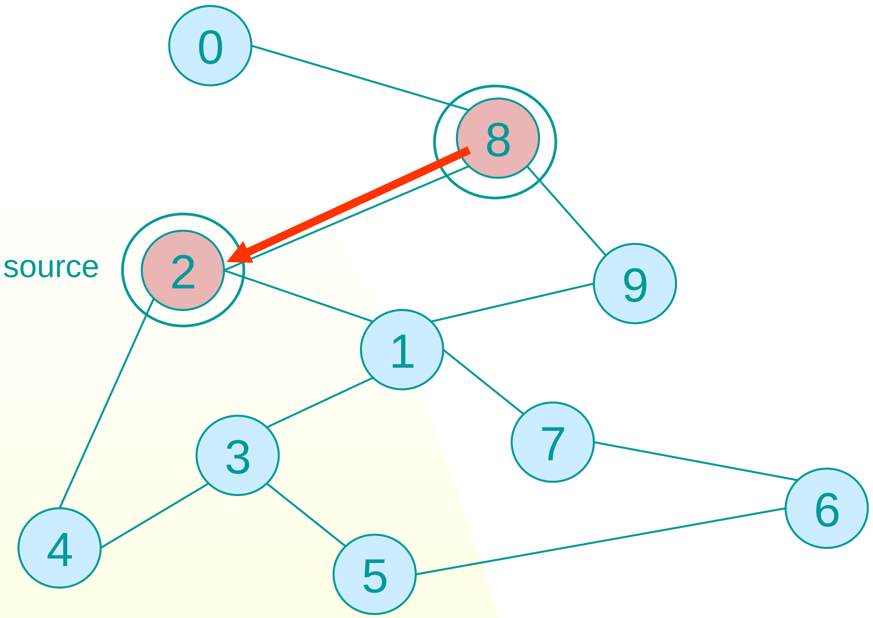
Mark 2 as visited

visit sequence= {2}

RDFS(**2**)

recursive call → RDFS(8)

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	$\boxed{0}$	9	
9	1	8		

Visited Table (T/F)

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

visit sequence= {2, 8}

Mark 8 as visited

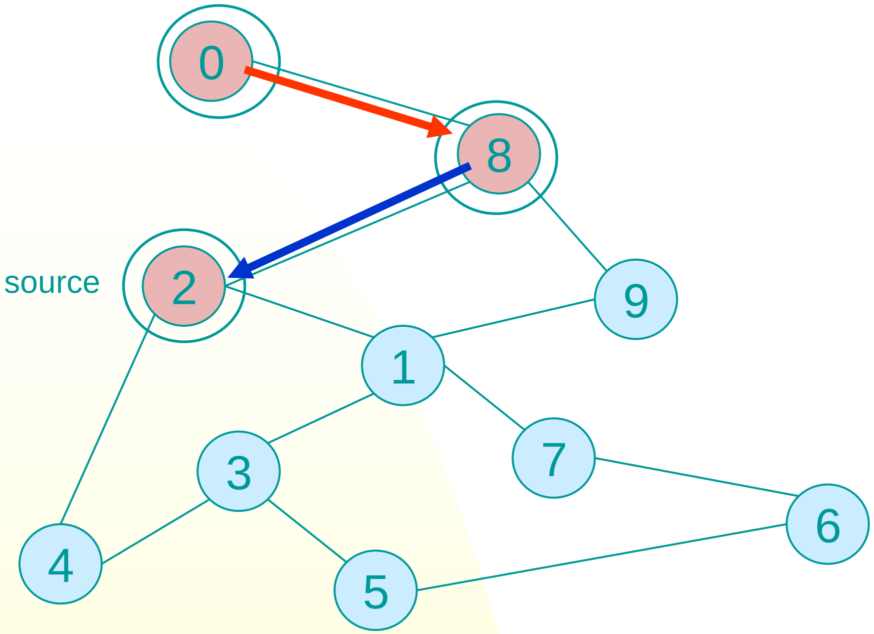
Recursive calls

RDFS(2)

RDFS(**8**)

recursive call→RDFS(0)

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

Pred

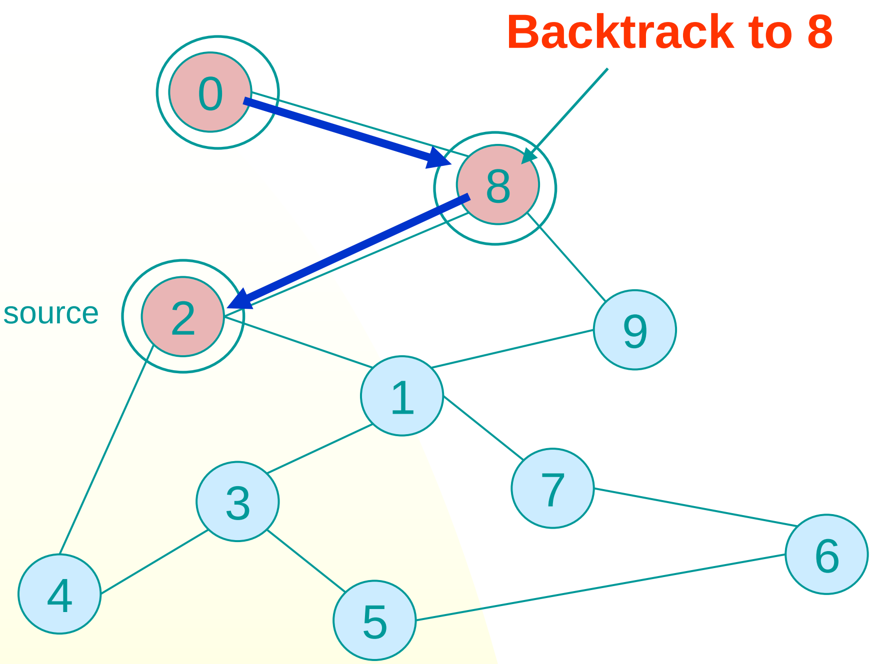
Mark 0 as visited

visit sequence= {2, 8, 0}

Recursive calls

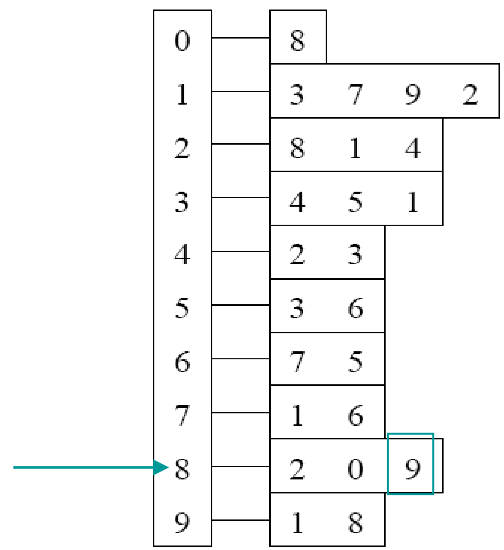
RDFS(2)
    RDFS(8)
        RDFS(**0**) -> no unvisited neighbor, return
            to (backtrack) RDFS(8)

Example



visit sequence= {2, 8, 0}

Adjacency List



Visited Table (T/F)

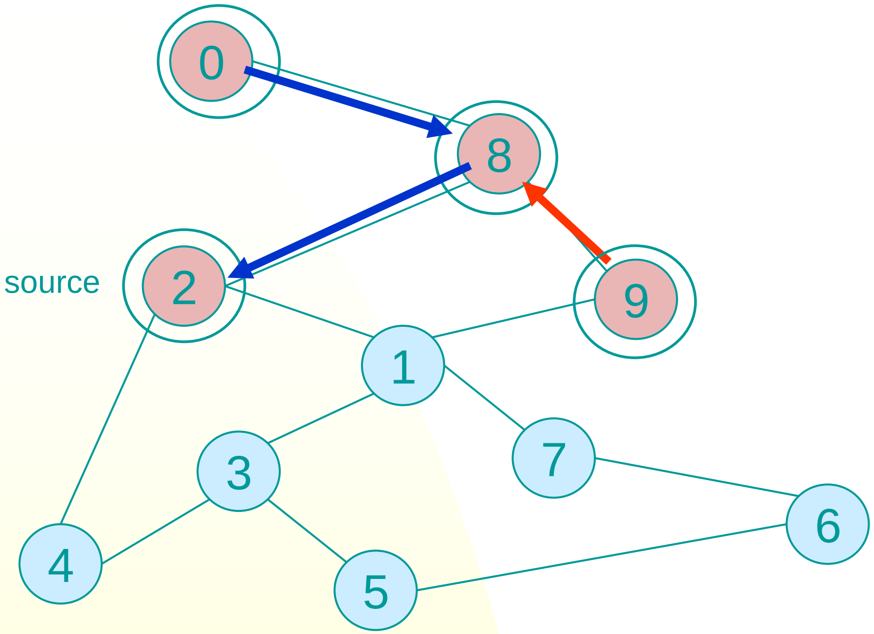
		Pred
0	T	8
1	F	-
2	T	-1
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

Pred

Recursive calls

RDFS(2)
    RDFS(8)
        recursive call→RDFS(9)

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

Pred

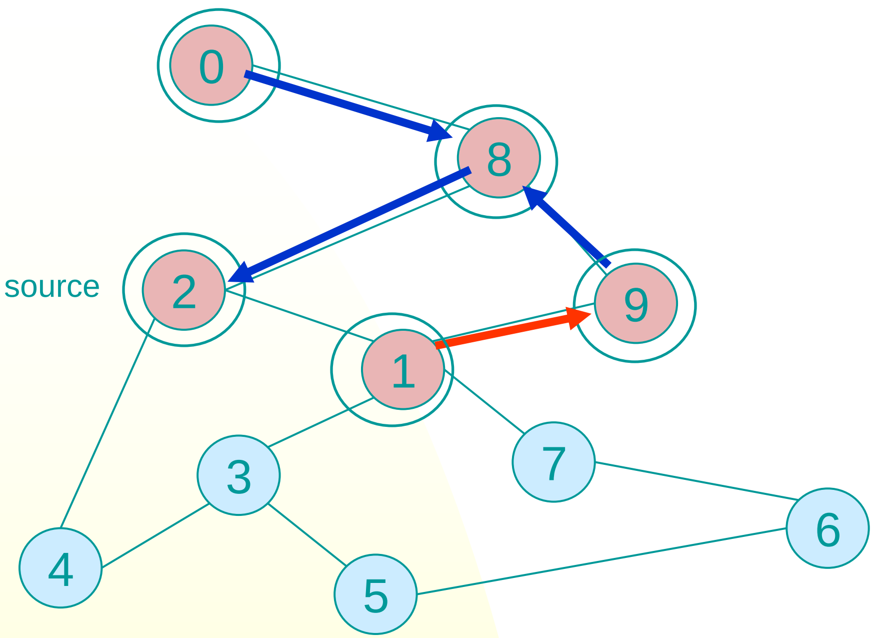
visit sequence= {2, 8, 0, 9}

Mark 9 as visited

Recursive calls

RDFS(2)
RDFS(8)
RDFS(**9**)
recursive call→RDFS(1)

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

Pred

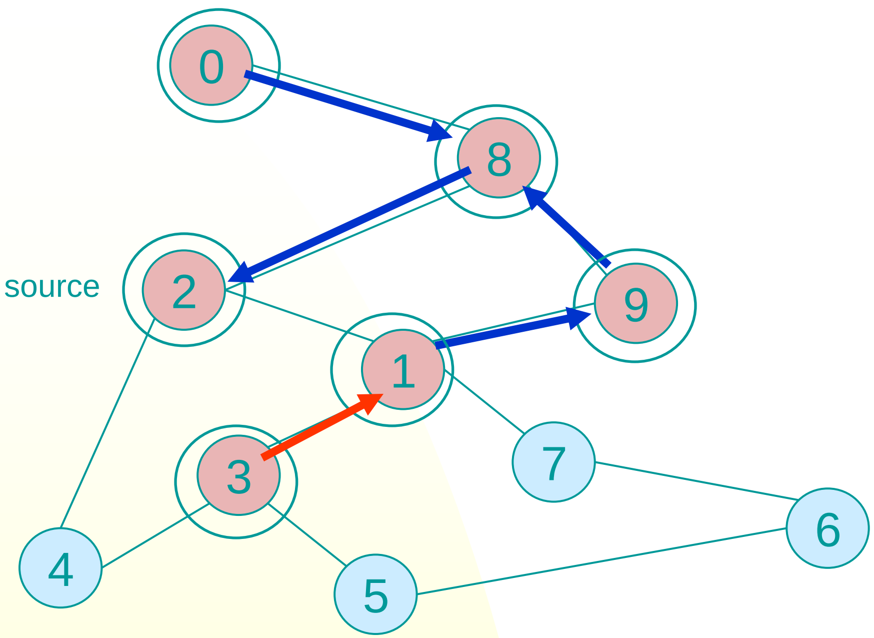
Mark 1 as visited

visit sequence= {2, 8, 0, 9, 1}

Recursive calls

RDFS(2)
 RDFS(8)
  RDFS(9)
   RDFS(**1**)
    recursive call→RDFS(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)

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

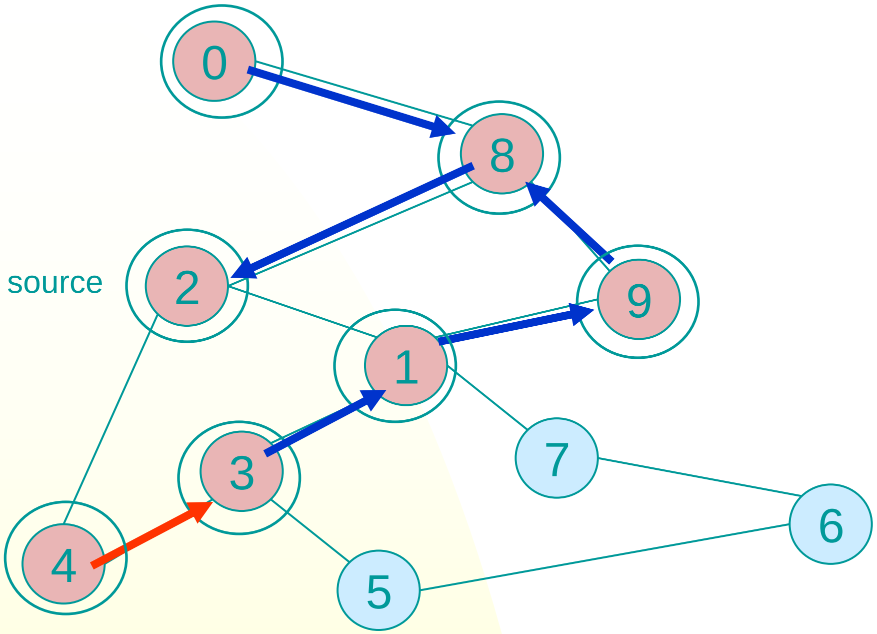
visit sequence= {2, 8, 0, 9, 1, 3}

Mark 3 as visited

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(**3**)
recursive call→RDFS(4)

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

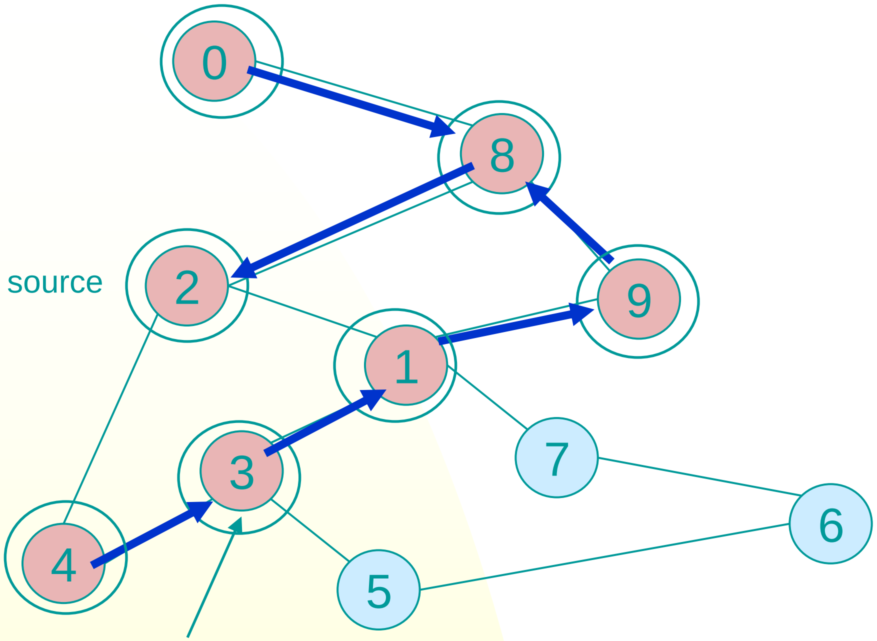
Mark 4 as visited

visit sequence= {2, 8, 0, 9, 1, 3, 4}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP all of 4's neighbors have been visited
backtrack (return back) to call RDFS(3)

Example



Backtrack to 3

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)

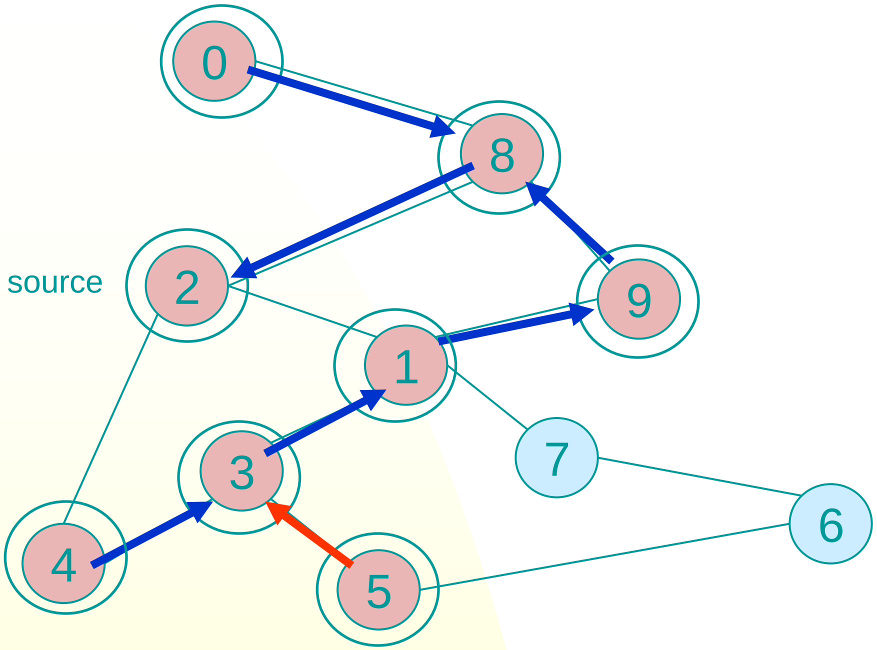
		Pred
0	T	8
1	T	9
2	T	-1
3	T	1
4	T	3
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

visit sequence= {2, 8, 0, 9, 1, 3, 4}

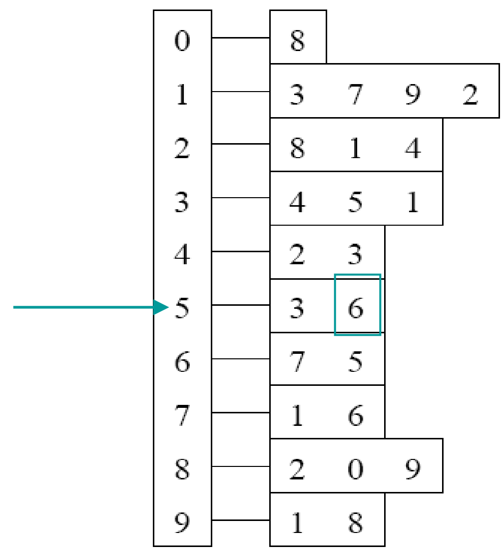
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
recursive call→RDFS(5)

Example



Adjacency List



Visited Table (T/F)

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

Pred

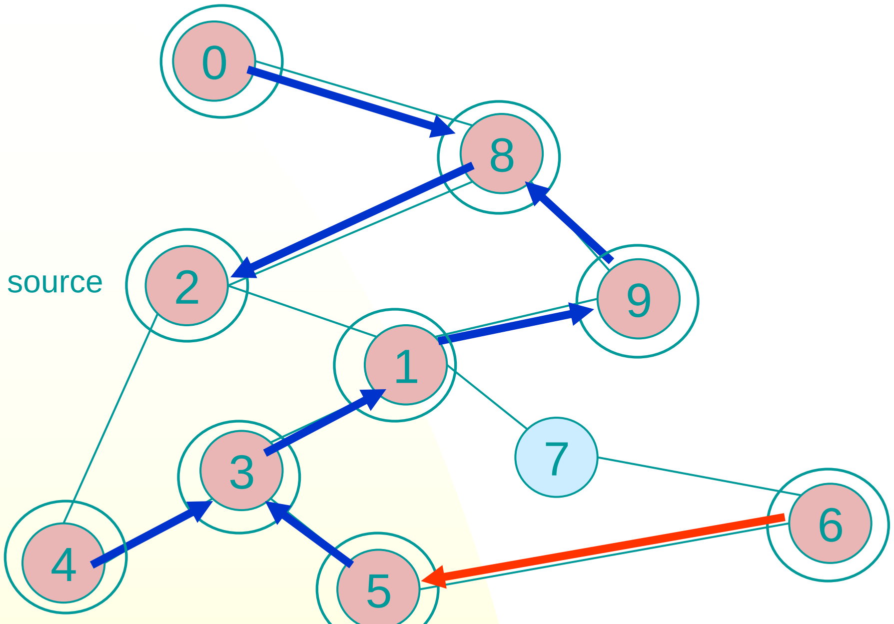
visit sequence= {2, 8, 0, 9, 1, 3, 4, 5}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
3 is visited, recursive call→RDFS(6)

Mark 5 as visited

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

Pred

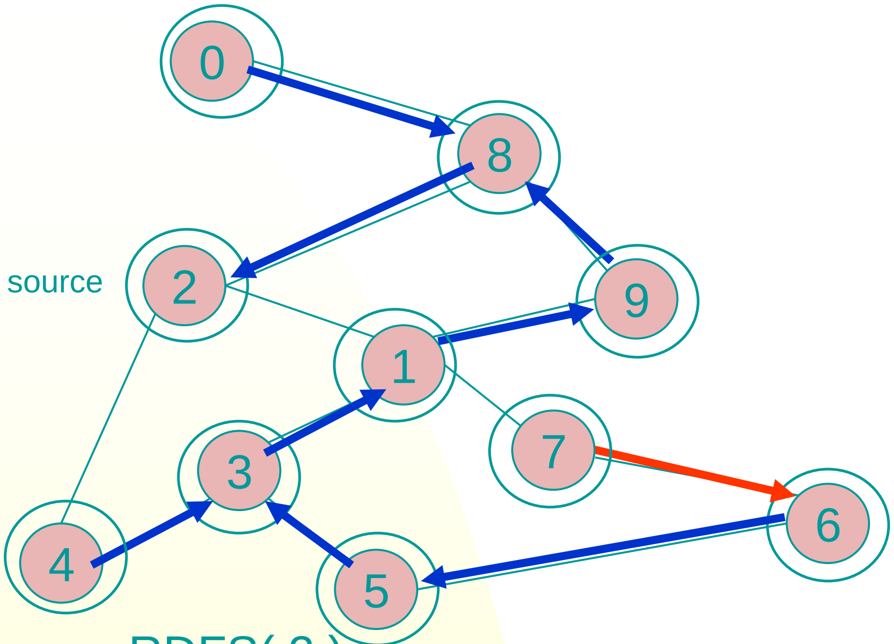
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(**6**)
recursive call→ RDFS(7)

Recursive calls

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6}

Mark 6 as visited

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

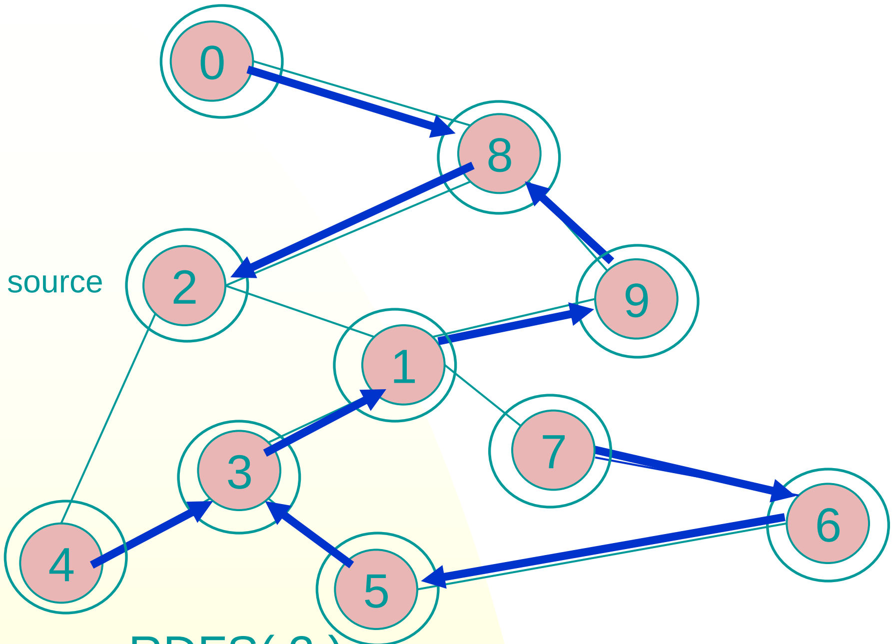
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7)

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

Mark 7 as visited

Example



RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) → no recursive call

Recursive calls

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		

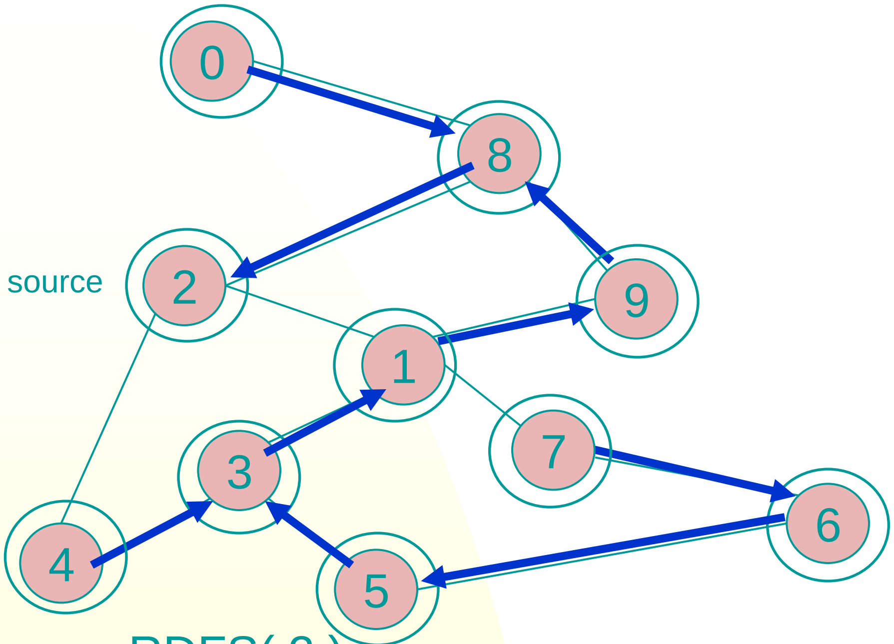
Visited Table (T/F)

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

Pred

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

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

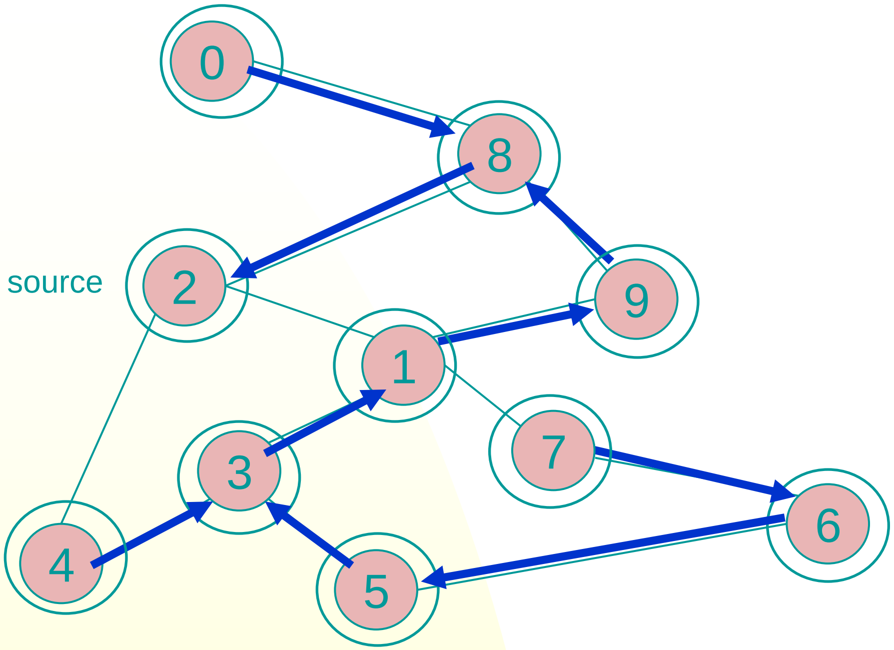
Pred

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) → no recursive call

Recursive calls

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

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

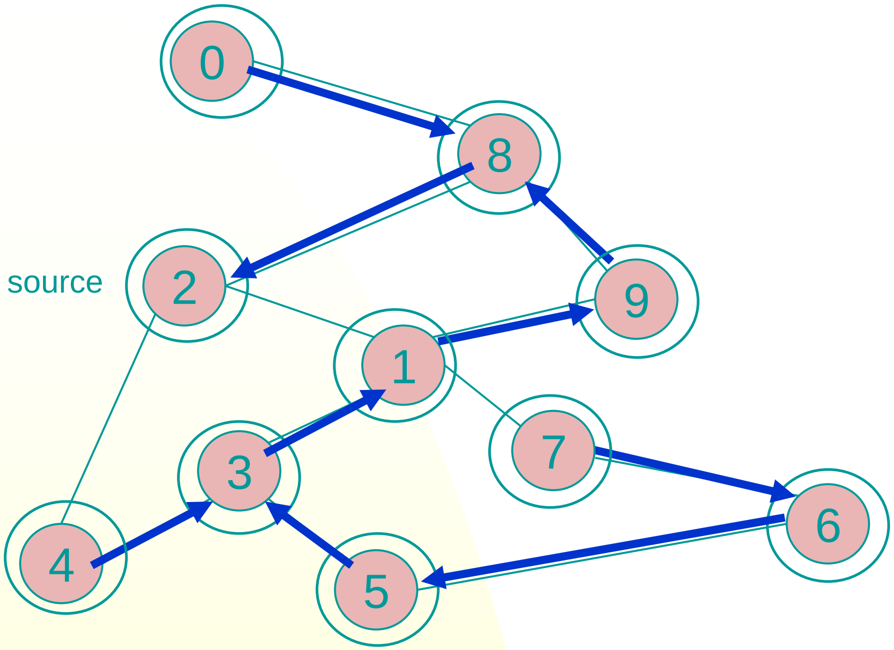
Pred

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) → no recursive call

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)

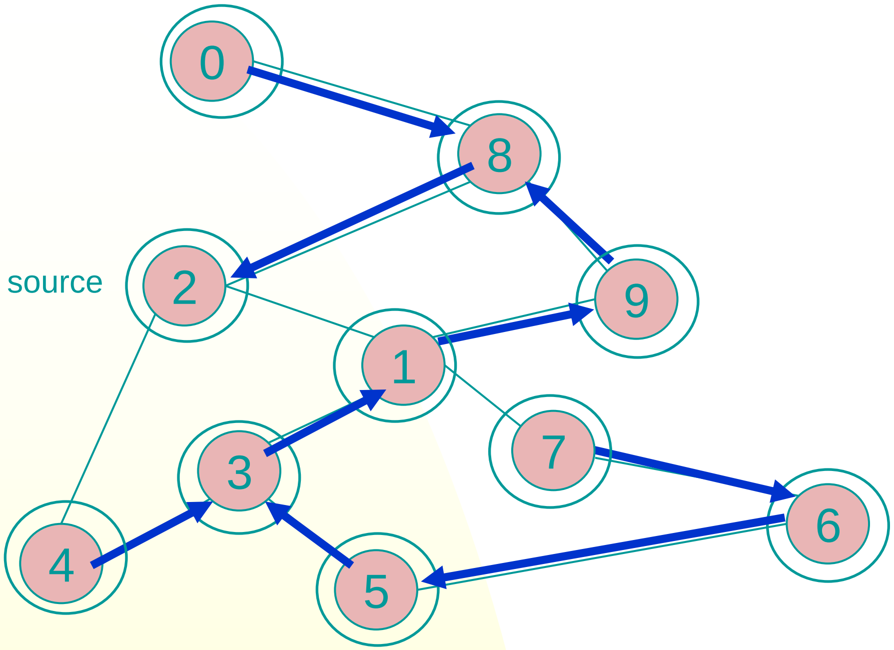
		Pred
0	T	8
1	T	9
2	T	-1
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) → no recursive call

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

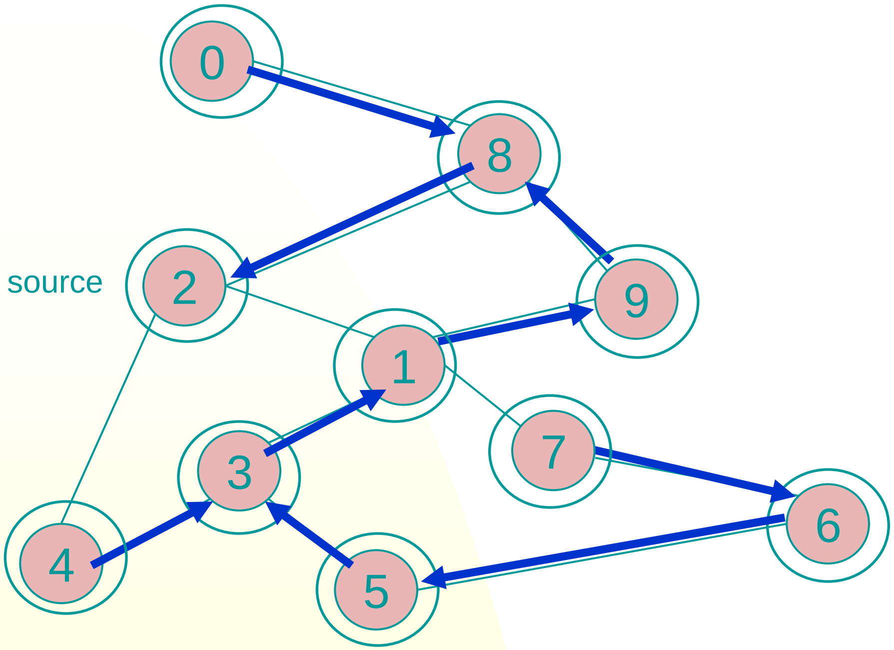
Pred

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) → no recursive call

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)

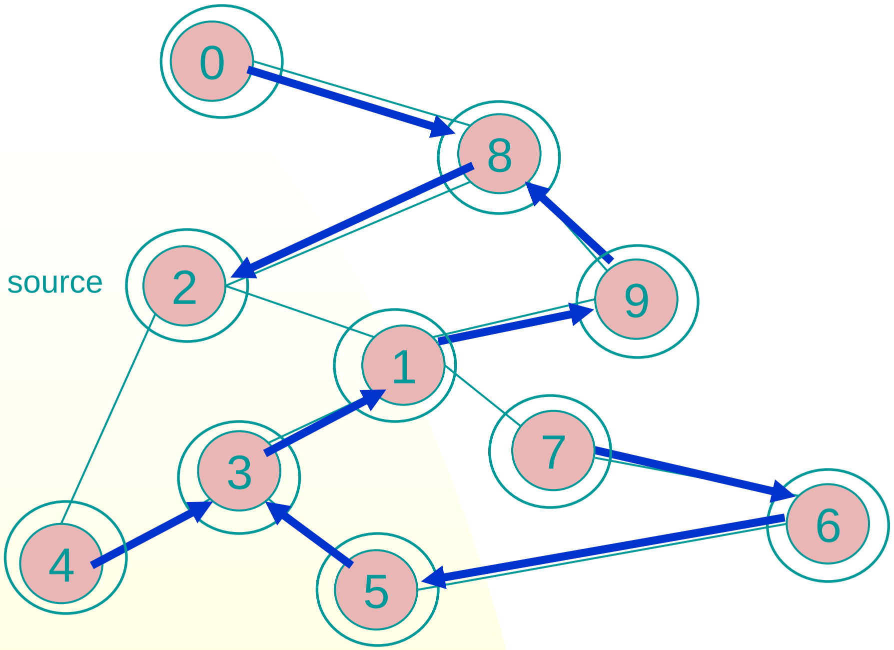
		Pred
0	T	8
1	T	9
2	T	-1
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9) → no recursive call

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

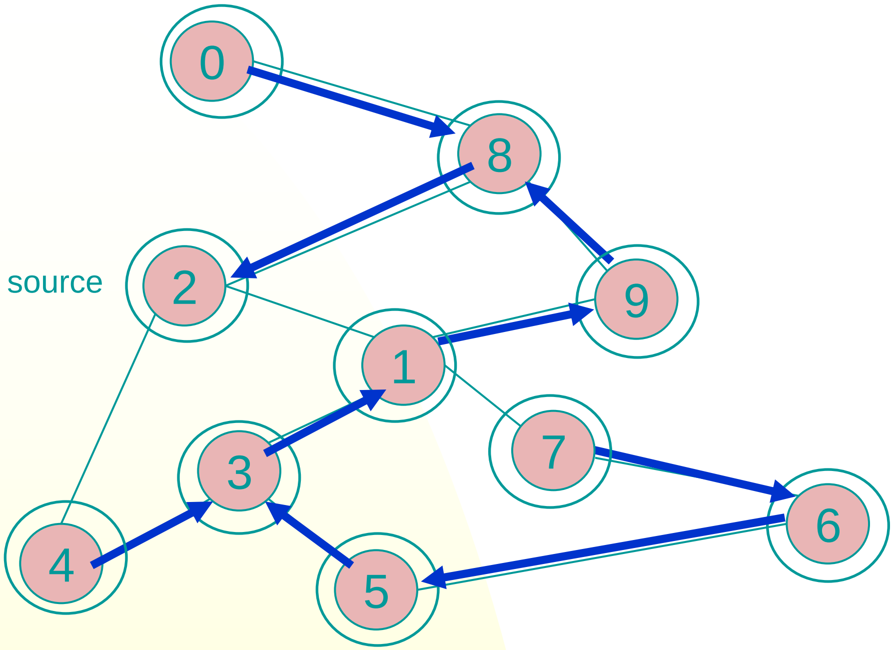
visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)

RDFS(8) → no recursive call

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

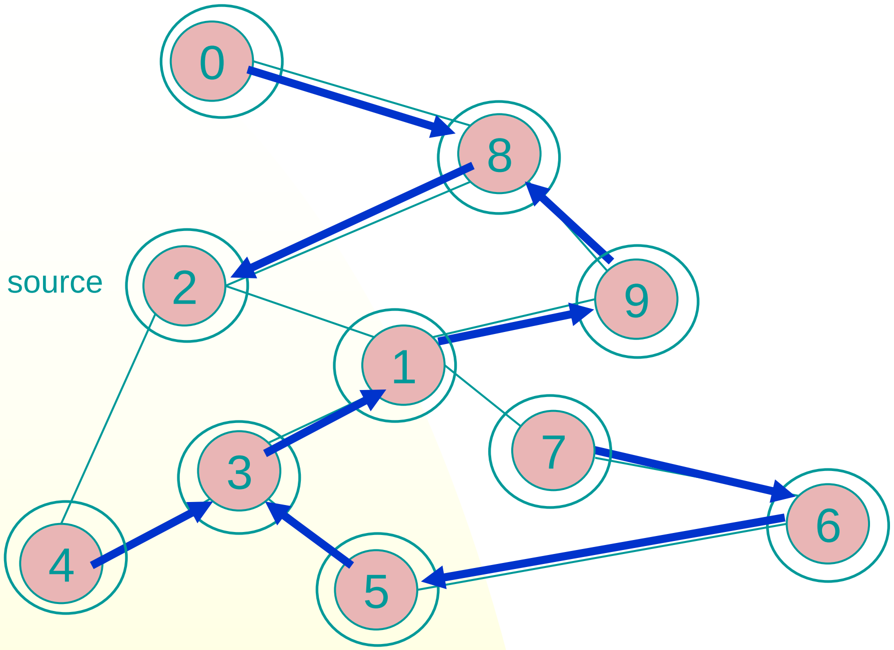
visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Pred

RDFS(2) → no recursive call

Recursive calls

Recover a path



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

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Algorithm *Path*(w)

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

Try some examples.

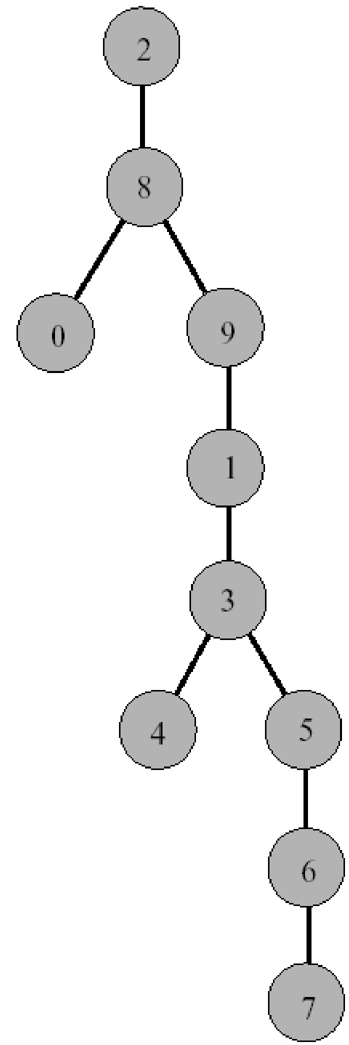
Path(0) ->

Path(6) ->

Path(7) ->

DFS Tree

- The edges that we traverse during DFS (or the edges that we backtrack along) form a tree. We usually call the rooted version (rooted at the source) the DFS tree.



Running time analysis

- The running time analysis is very similar to BFS. Let the graph be represented by an *adjacent list*, and *n* and *m* represent the number of vertices and edges in the graph respectively.

Running time analysis

- Each (connected) vertex is visited EXACTLY one time → so there are O(n) recursive call.
- For a particular vertex *v*, (in the recursive call) we need to find all its neighbors. For *adjacent list* representation, it takes O(degree(v)).
  - Hence, the total number of time for all vertices is

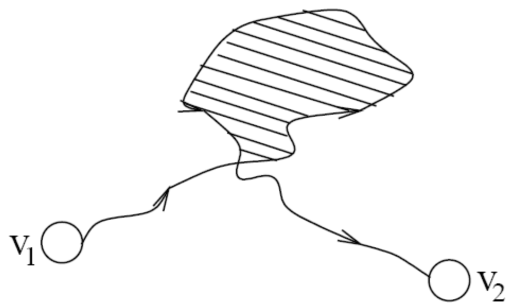
$$\sum_{\text{vertex } v} O(\deg(v)) = O(2m) = O(m)$$

- For *adjacency list* representation, the running time for DFS is

$$\underbrace{O(n)}_{\text{initialization}} + \underbrace{O(n)}_{\text{number of recursive call}} + \underbrace{O(2m)}_{\text{find out all the neighbors}} = O(n+m)$$

Some applications of BFS/DFS -- Connectivity

- A graph is *connected* if and only if there exists a path between *every* pair of distinct vertices.
- If a path is not simple, then it contains cycles. Since any cycle can be bypassed, the non-simple path contains a simple path.



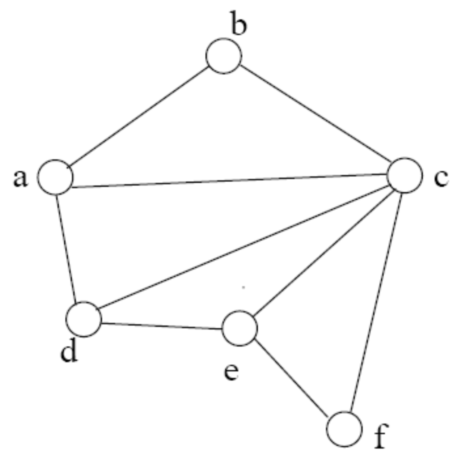
- Therefore, a graph is connected if and only if there exists a simple path between every pair of distinct vertices.

Some applications of BFS/DFS -- Connectivity

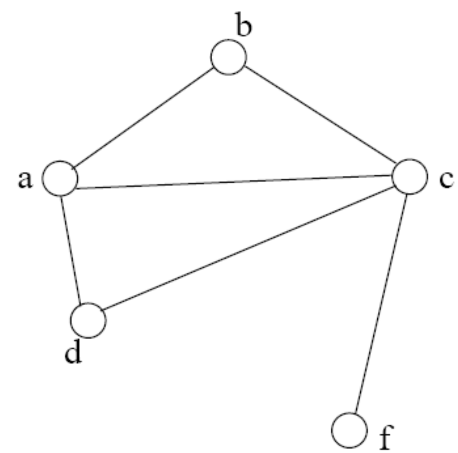
- One can use BFS or DFS to decide if a graph is connected. Run BFS or DFS using an arbitrary vertex as the source. At the end, if all vertices have been visited (all flags are marked *'T'*), then the graph is connected. Otherwise, the graph is disconnected.
- The running time is O(n+m).

Some applications of BFS/DFS – Connected Components

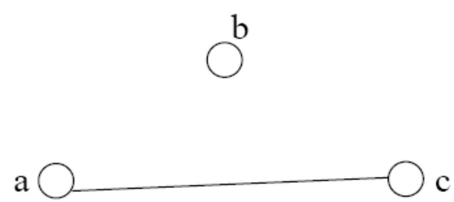
A graph $H(V_H, E_H)$ is a *subgraph* of $G(V_G, E_G)$ if and only if $V_H \subset V_G$ and $E_H \subset E_G$.



graph G



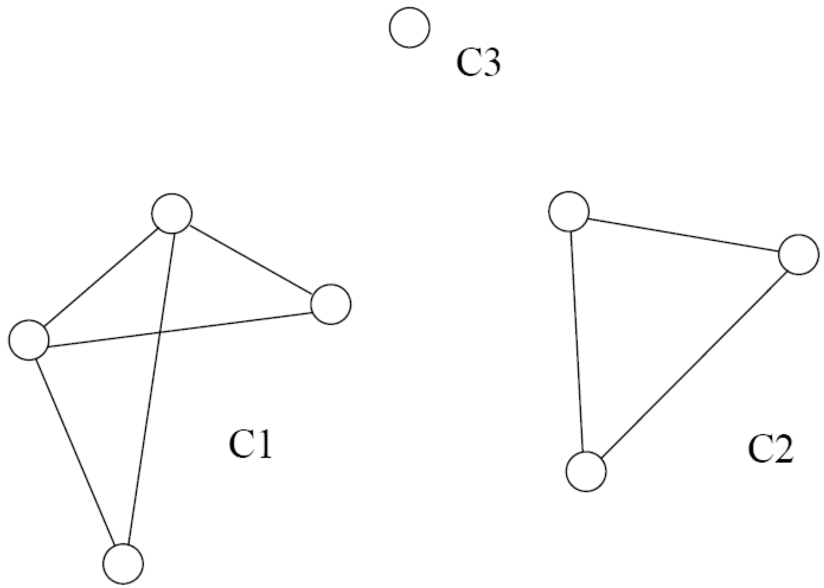
subgraph H 1



subgraph H 2

Some applications of BFS/DFS – Connected Components

- A connected component is a maximal connected subgraph of a graph.
- The set of connected components is unique for a given graph.



3 components: C1, C2, and C3

Some applications of BFS/DFS – Connected Components

Algorithm *DFSConn*(G)
Input: a graph G
Output: the connected components

1. **for** each vertex v
2. **do** $flag[v]$:= false;
3. **for** each vertex v
4. **do if** $flag[v]$ = false
5. **then** output "A new connected component:";
6. *RDFS*(v);

For each vertex

If not visited

Call *RDFS(v)*

This will find all connected vertices to "v"

Algorithm *RDFS*(v)

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

Basic DFS algorithm.

Some applications of BFS/DFS – Connected Components

Running time analysis

- Running time for each *i* connected-component

$$O(n_i + m_i)$$

- Question:
  - Can two connected components have the same edge?
  - Can two connected components have the same vertex?

- It follows

$$\sum_i O(n_i + m_i) = O(\sum_i n_i + \sum_i m_i) = O(n+m)$$