Graph Traversal Techniques: BFS and DFS

MTL 776

Graph Algorithms

Graph Representation

Two popular computer representations of a graph. Both represent the vertex set and the edge set, but in different ways.

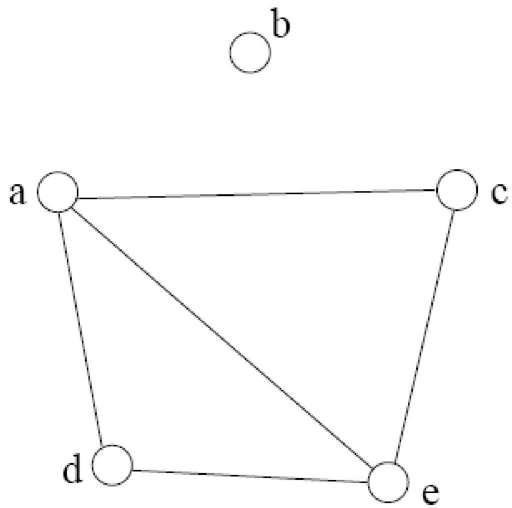
1. Adjacency Matrix

   Use a 2D matrix to represent the graph

2. Adjacency List

   Use a 1D array of linked lists

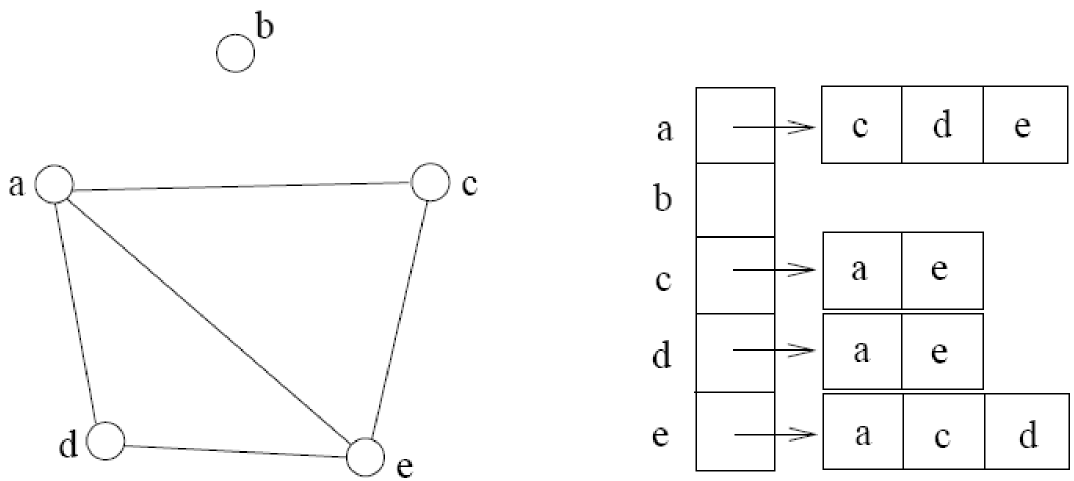
Adjacency Matrix



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

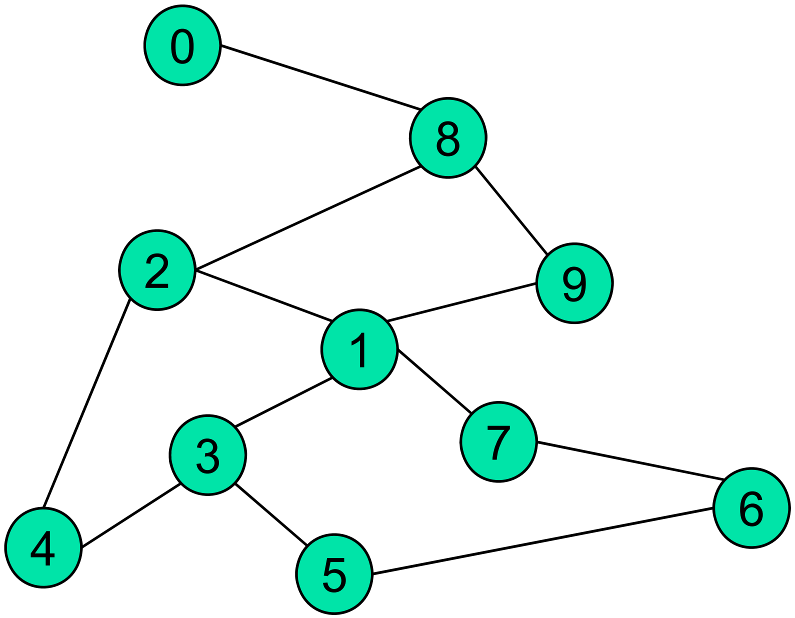
- 2D array A[0..n-1, 0..n-1], where ***n*** is the number of vertices in the graph
- Each row and column is indexed by the vertex id
  - e,g a=0, b=1, c=2, d=3, e=4
- *A[i][j]=1* if there is an edge connecting vertices *i* and *j*; otherwise, *A[i][j]=0*
- The storage requirement is $\Theta(n^2)$. It is not efficient if the graph has few edges. An adjacency matrix is an appropriate representation if the graph is dense: $|E|=\Theta(|V|^2)$
- We can detect in O(1) time whether two vertices are connected.

Adjacency List



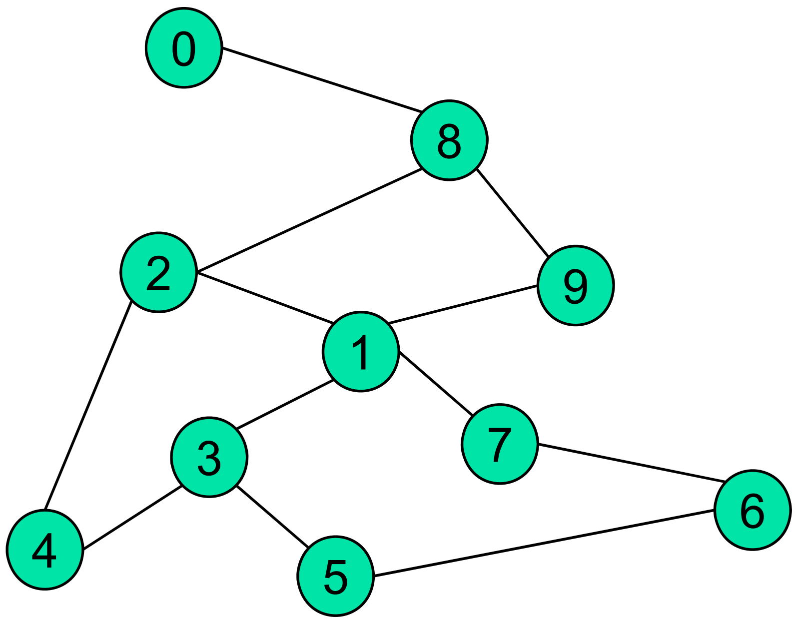
- If the graph is not dense, in other words, sparse, a better solution is an adjacency list
- The adjacency list is an array A[0..n-1] of lists, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id
- Each list *A[i]* stores the ids of the vertices adjacent to vertex *i*

Adjacency Matrix Example



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

Adjacency List Example



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

Storage of Adjacency List

- The array takes up $\Theta(n)$ space
- Define degree of v, deg(v), to be the number of edges incident to v. Then, the total space to store the graph is proportional to:

$$\sum_{\text{vertex } v} \deg(v)$$

- An edge $e=\{u,v\}$ of the graph contributes a count of 1 to deg(u) and contributes a count 1 to deg(v)
- Therefore, $\Sigma_{\text{vertex } v}\deg(v) = 2m$, where m is the total number of edges
- In all, the adjacency list takes up $\Theta(n+m)$ space
  - If $m = O(n^2)$ (i.e. dense graphs), both adjacent matrix and adjacent lists use $\Theta(n^2)$ space.
  - If $m = O(n)$, adjacent list outperform adjacent matrix

- However, one cannot tell in O(1) time whether two vertices are connected

Adjacency List vs. Matrix

- **Adjacency List**
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists

- **Adjacency Matrix**
  - Always require n^2 space
    - This can waste a lot of space if the number of edges are sparse
  - Can quickly find if an edge exists

Generic Search

Algorithm Generic GraphSearch

```
{   for i=1 to n do  {Mark[i]=0; P[i]=0; Num[i]=0; Com[i]=0;} count=1; comp=0;
For i=1 to n do { if (Mark[i]==0) comp++;search(i);}
Search(i){
    Mark[i]=1; P[i]=-1; Num[i]=count; Count++; com[i]=comp;
    S={i};
    While (  S ≠ emprtyset){
        select a vertex x from S;
        if ( x has an unmarked neighbor y)
                { Mark[y]=1; P[y]=x; Num[y]=count; count=count+1;
                  S=S∪ {y};com[y]=comp;
                }
        else S=S-{x}; }
    }
```

Queue implementation of Generic Search: BFS

Algorithm BFS

```
    {  for i=1 to n do  {Mark[i]=0; P[i]=0; Num[i]=0;} count=1;
For i=1 to n do { if (Mark[i]==0) search(i);  BFS(i);}  Q=createQueue();
BFS(i) search(i){
    Mark[i]=1; P[i]=-1; Num[i]=count; Count++;
    S={i}; Enque(Q,i);
    While (  S !=empty   Isemptyqueue(Q)!=0){
          select a vertex x from S; x=Front(Q);
          if ( x has an unmarked neighbor y)
                    { Mark[y]=1; P[y]=x; Num[y]=count; count=count+1;
                      S=S∪ {y}; Enque(Q,y);
                    }
          else S=S-{x}; Dequeue(Q); }
    }  Note that green colored text is replaced with red colored text in
    genertic search
```

Stack implementation of Generic Search: DFS

Algorithm DFS

```
  {  for i=1 to n do  {Mark[i]=0; P[i]=0; Num[i]=0;} count=1;
For i=1 to n do { if (Mark[i]==0) search(i); DFS(i);} S=createStack();
Search(i) DFS(i){
    Mark[i]=1; P[i]=-1; Num[i]=count; Count++;
    S={i}; Push(S,i);
    While (  S !=emptyset Isempty(S)!=0){
          select a vertex x from S;
          if ( x has an unmarked neighbor y)
                  { Mark[y]=1; P[y]=x; Num[y]=count; count=count+1;
                    S=S∪ {y}; Push(S,y);
                  }
          else S=S-{x}; Pop(S); }
    } Note that green colored text is replaced with red colored text in
    genertic search
```

BFS Algorithm

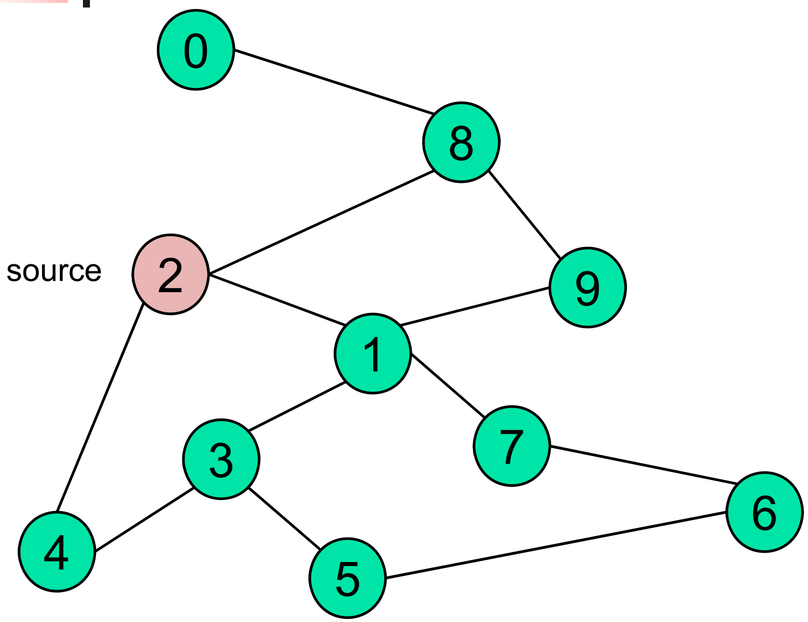
Algorithm *BFS*(s)

Input: s is the source vertex

Output: Mark all vertices that can be visited from s.

for each vertex v
    do $\mathit{flag}[v] :=$ false; **// flag[ ]: visited table**
$Q =$ empty queue; **Why use queue? Need FIFO**
$\mathit{flag}[s] :=$ true;
enqueue(Q, s);
while Q is not empty
    do $v :=$ *dequeue*(Q);
        for each w adjacent to v
            do if $\mathit{flag}[w] =$ false
                then $\mathit{flag}[w] :=$ true;
                    enqueue(Q, w)

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		

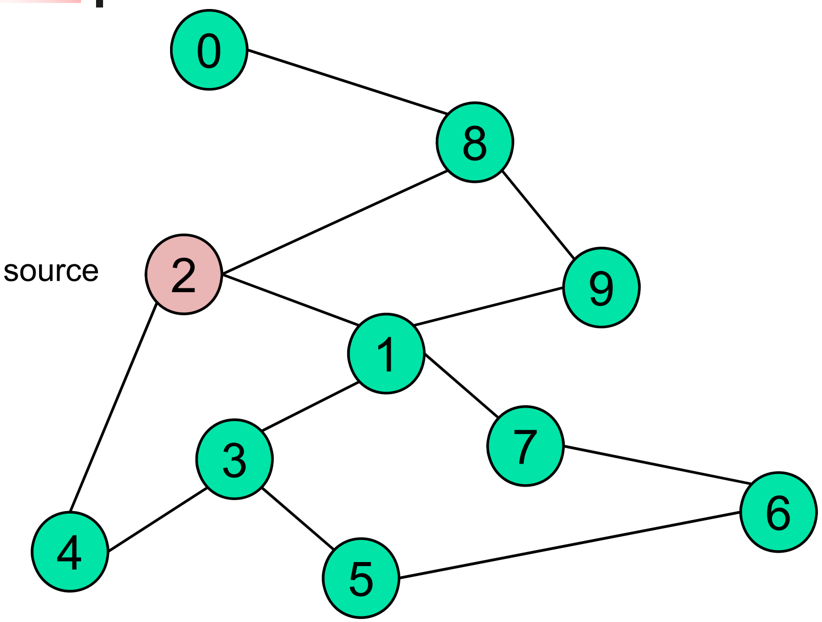
Visited Table (T/F)

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



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

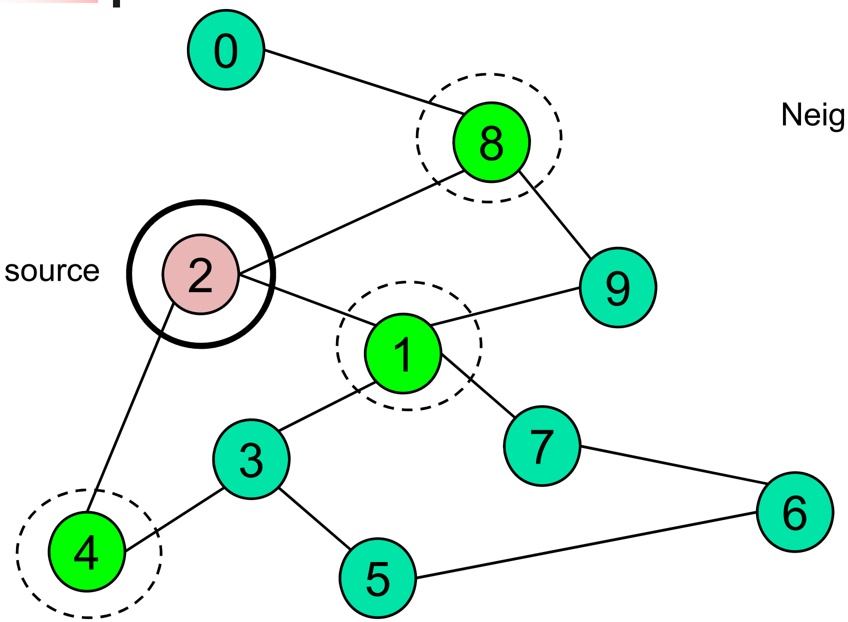
Visited Table (T/F)

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



Adjacency List

Neighbors →

	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		

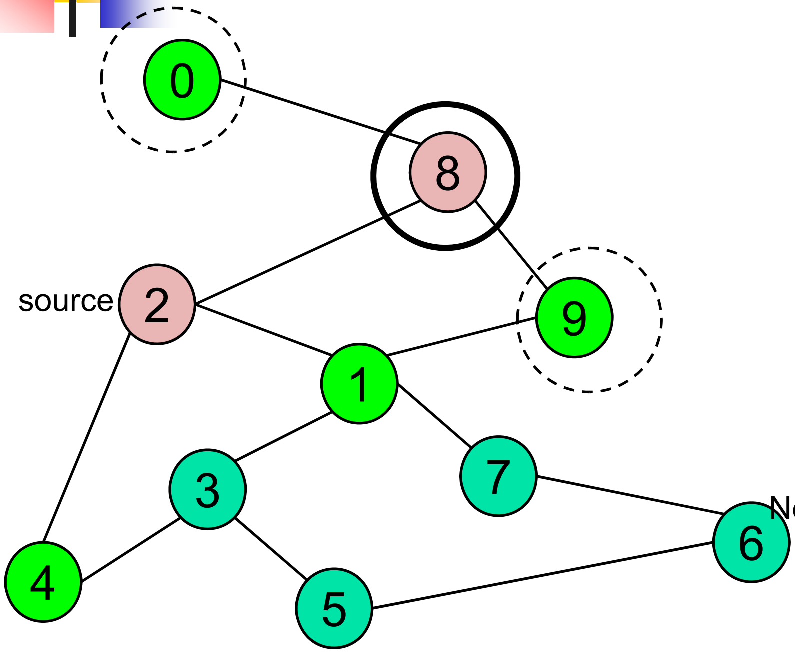
Visited Table (T/F)

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

Mark neighbors as visited 1, 4, 8

Q = {2} → { 8, 1, 4 }

Dequeue 2.
Place all unvisited neighbors of 2 on the queue



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

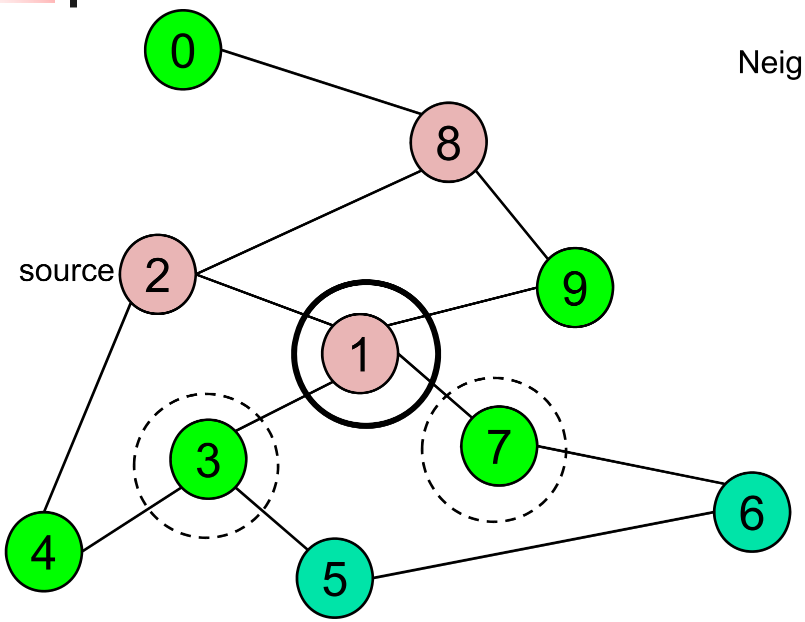
Visited Table (T/F)

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

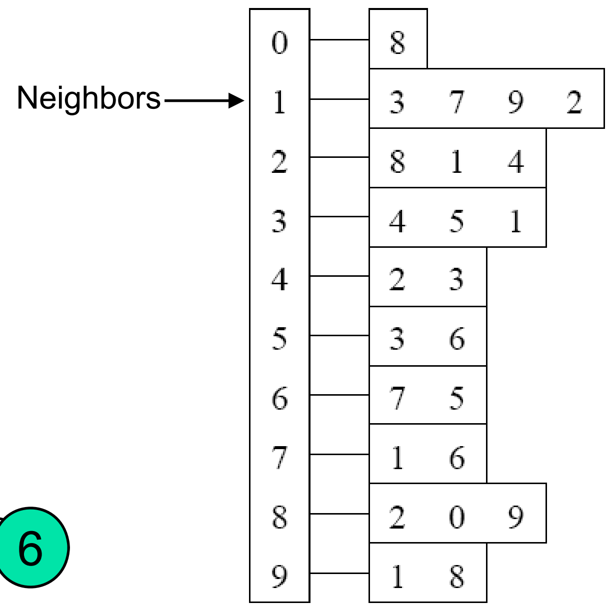
Mark new visited Neighbors 0, 9

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!



Adjacency List



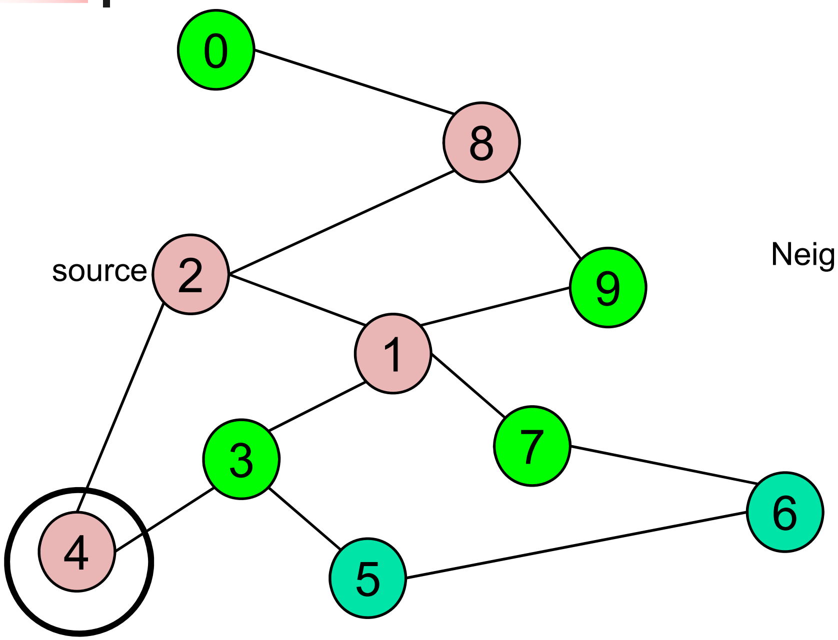
Visited Table (T/F)

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

Mark new visited Neighbors 3, 7

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.



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

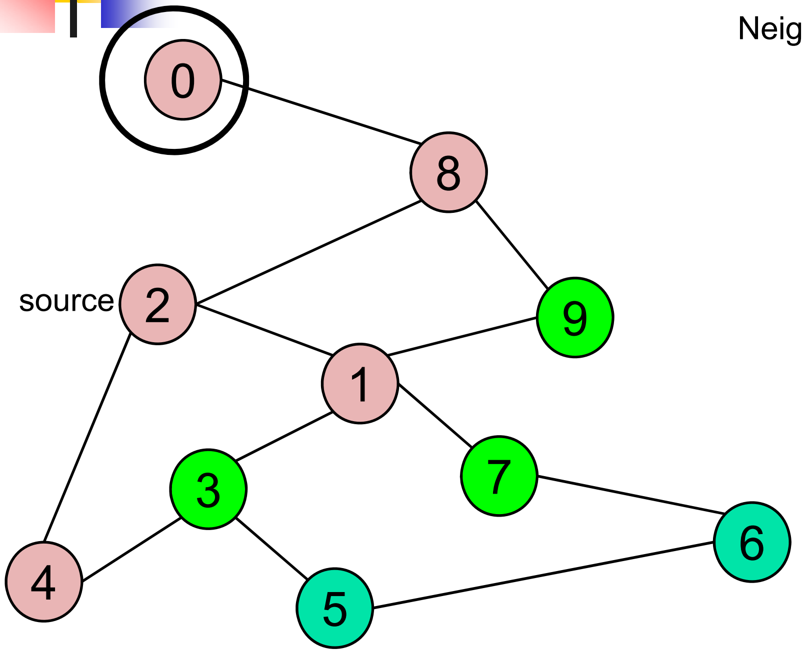
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	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!



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

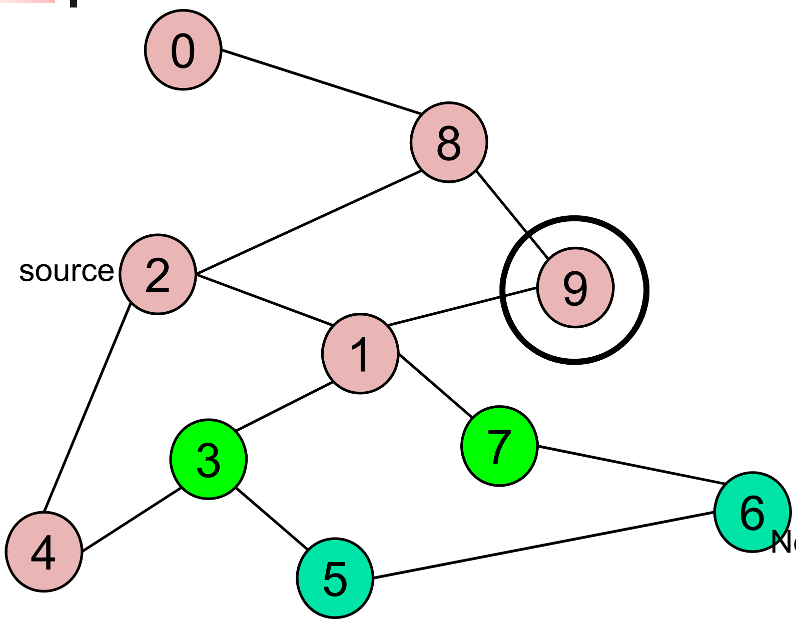
Visited Table (T/F)

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

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!



Adjacency List

Neighbors → 9

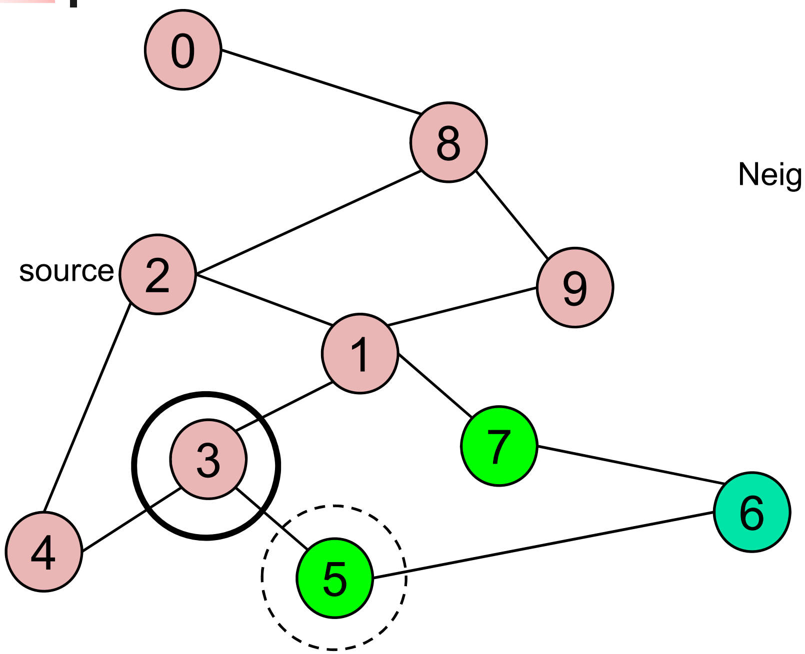
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

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!



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

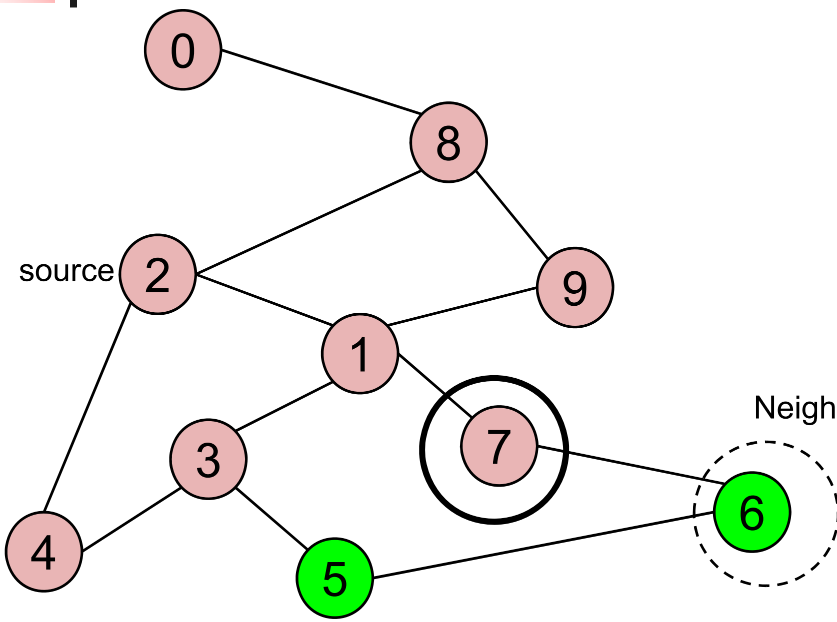
Visited Table (T/F)

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.



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

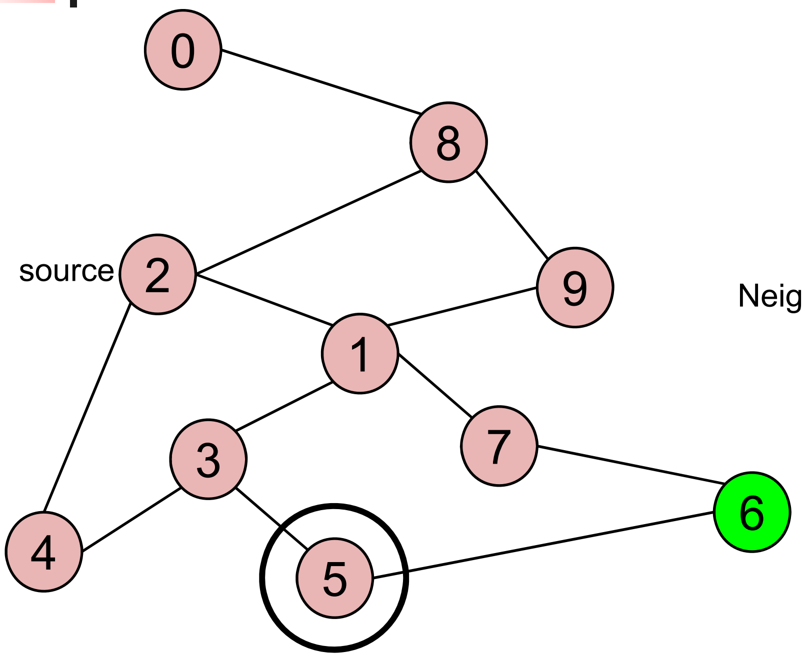
Visited Table (T/F)

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



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

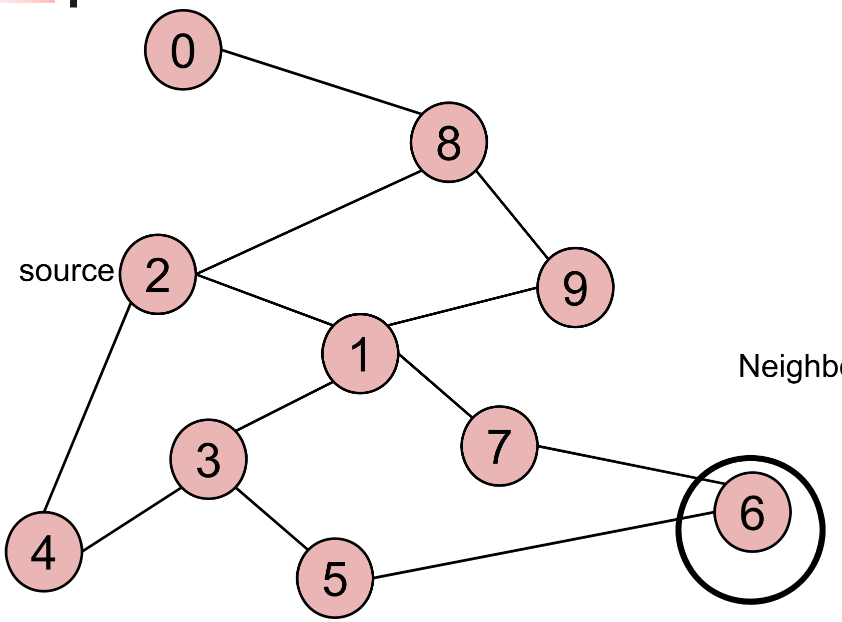
Visited Table (T/F)

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



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

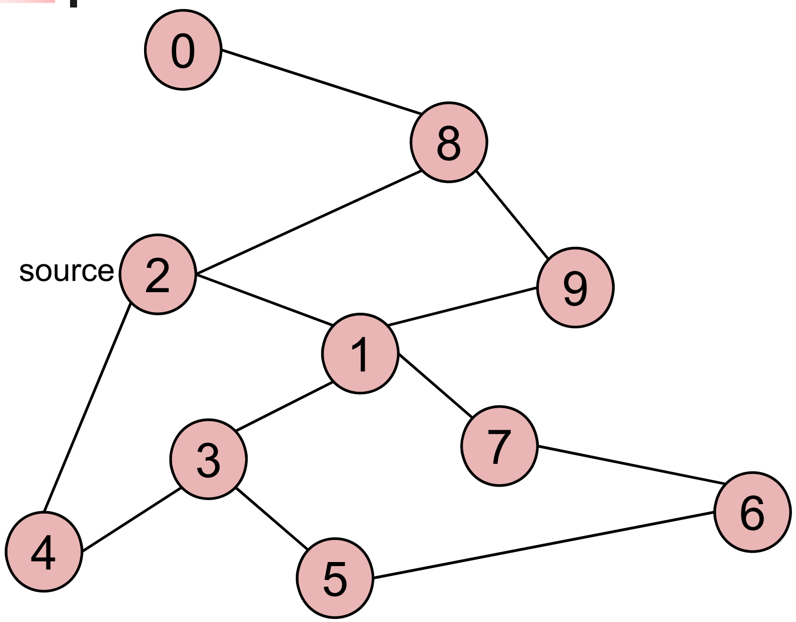
Visited Table (T/F)

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

Q = { 6 } → { }

Dequeue 6.

-- no unvisited neighbors of 6



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 = { } 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
(Using Adjacency List)

Assume adjacency list

- n = number of vertices m = number of edges

Algorithm *BFS*(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
    do v := dequeue(Q);
        for each w adjacent to v
            do if flag[w] = false
                then flag[w] := true;
                    enqueue(Q, w)
```

O(n + m)

Each vertex will enter Q at most once.

Each iteration takes time proportional to deg(v) + 1 (the number 1 is to account for the case where deg(v) = 0 --- the work required is 1, not 0).

Running Time

- Recall: Given a graph with m edges, what is the total degree?

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

- The total running time of the while loop is:

$$O(\Sigma_{\text{vertex } v} (\deg(v) + 1)) = O(n+m)$$

this is summing over all the iterations in the while loop!

Time Complexity of BFS

(Using Adjacency Matrix)

Assume adjacency list

- n = number of vertices   m = number of edges

Algorithm *BFS*(s)

Input: s is the source vertex

Output: Mark all vertices that can be visited from s.

```
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
    do v := dequeue(Q);
        for each w adjacent to v
            do if flag[w] = false
                then flag[w] := true;
                    enqueue(Q, w)
```

$\boxed{O(n^2)}$

Finding the adjacent vertices of v requires checking all elements in the row. This takes linear time O(n).

Summing over all the n iterations, the total running time is $O(n^2)$.

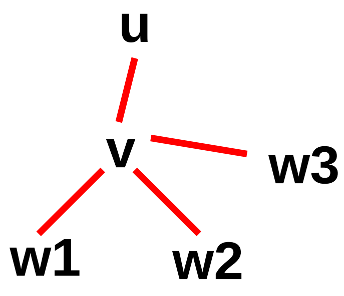
So, with adjacency matrix, BFS is $O(n^2)$ independent of the number of edges m. With adjacent lists, BFS is O(n+m); if $m=O(n^2)$ like in a dense graph, $O(n+m)=O(n^2)$.

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 can provide certain information about the graph that BFS cannot

DFS Algorithm

- 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.
  - Note: it is possible that w2 was unvisited when we recursively visit w1, but became visited by the time we return from the recursive call.



Stack implementation of Generic Search: DFS

Algorithm DFS

```
{   for i=1 to n do  {Mark[i]=0; P[i]=0; Num[i]=0;} count=1;
For i=1 to n do { if (Mark[i]==0) search(i); DFS(i);} S=createStack();
Search(i) DFS(i){
    Mark[i]=1; P[i]=-1; Num[i]=count; Count++;
    S={i}; Push(S,i);
    While (  S !=emptyset Isempty(S)!=0){
            select a vertex x from S;
            if ( x has an unmarked neighbor y)
                    { Mark[y]=1; P[y]=x; Num[y]=count; count=count+1;
                      S=S∪ {y}; Push(S,y);
                    }
            else S=S-{x}; Pop(S); }
```

} Note that green colored text is replaced with red colored text in genertic search

DFS Algorithm

Algorithm $DFS(s)$

```
for each vertex v
    do flag[v] := false;
RDFS(s);
```

Flag all vertices as not visited

Algorithm $RDFS(v)$

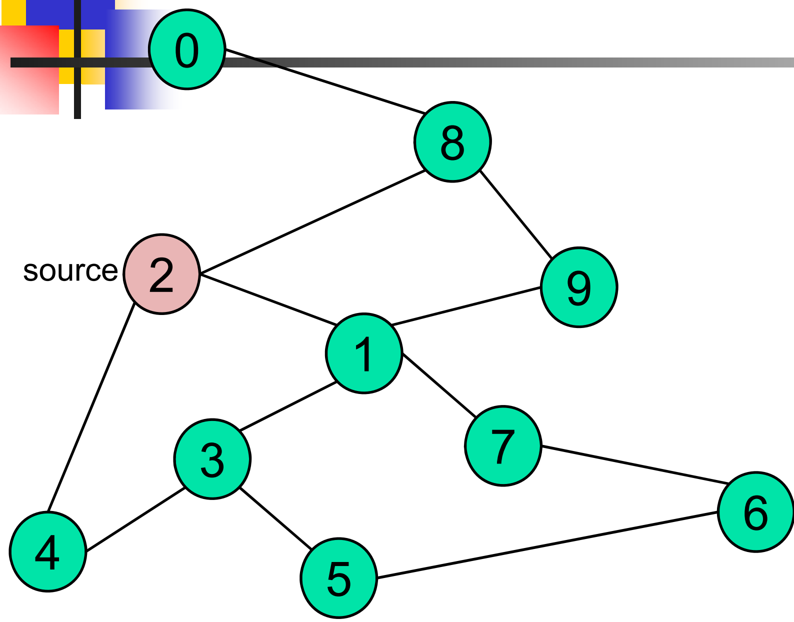
```
flag[v] := true;
for each neighbor w of v
    do if flag[w] = false
        then RDFS(w);
```

Flag yourself as visited

For unvisited neighbors, call RDFS(w) recursively

We can also record the paths using pred[ ].

Example



Adjacency List

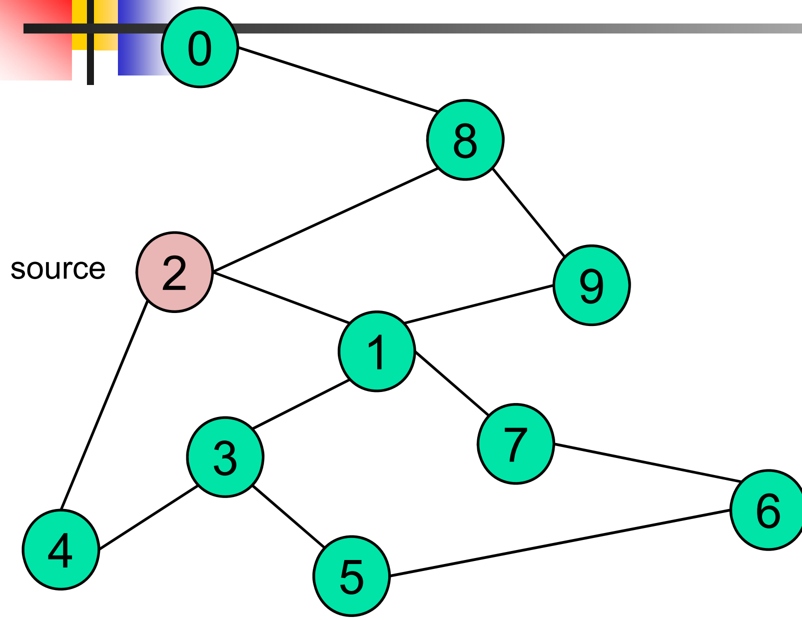
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

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 False)

Initialize Pred to -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		

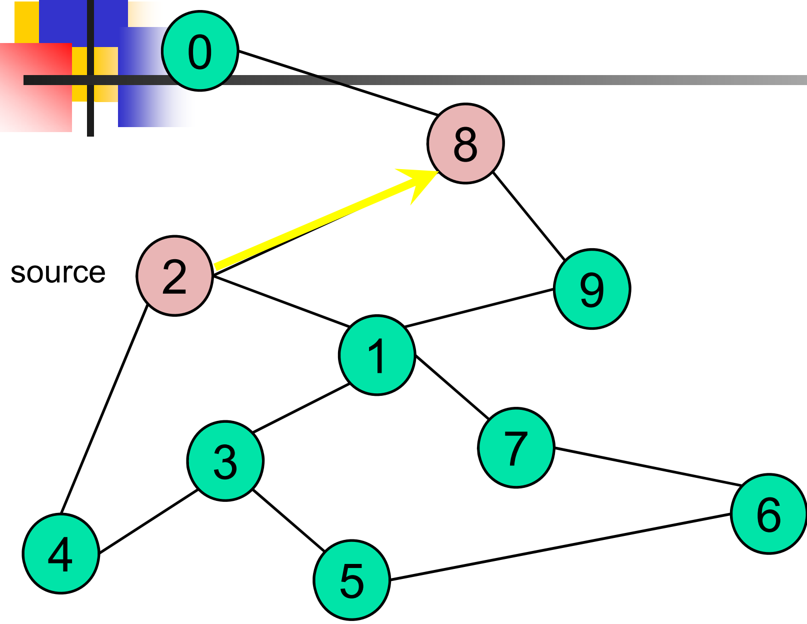
Visited Table (T/F)

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

Mark 2 as visited

RDFS(2)

Now visit RDFS(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)

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

Mark 8 as visited

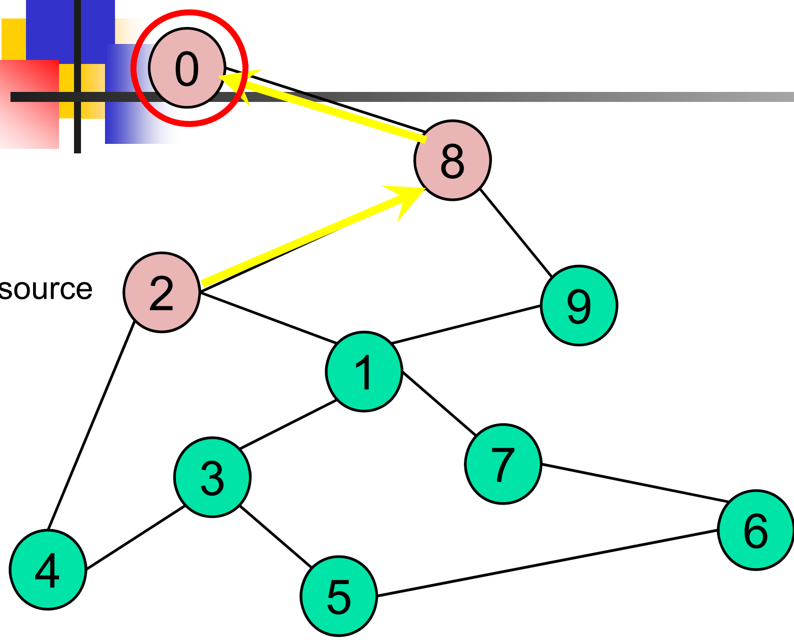
mark Pred[8]

Recursive calls

RDFS(2)

RDFS(8)

2 is already visited, so visit RDFS(0)



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

Mark 0 as visited

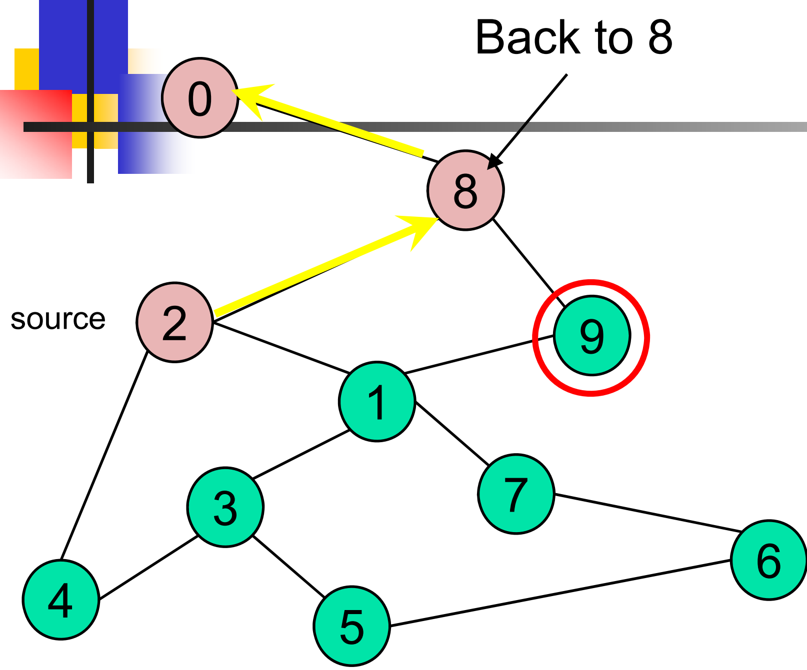
Mark Pred[0]

Recursive calls

RDFS(2)

RDFS(8)

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

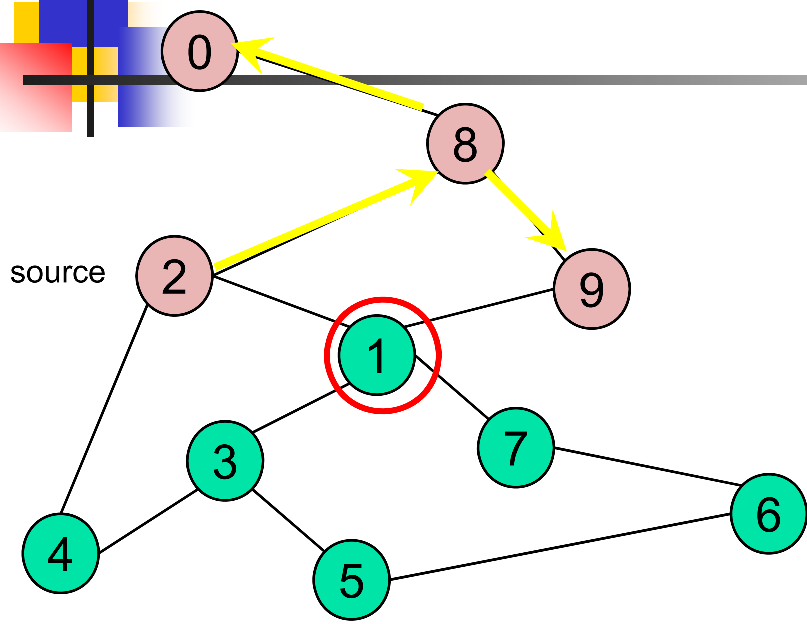
		Pred
0	T	8
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

Recursive calls

RDFS(2)

RDFS(8)

Now visit 9 -> RDFS(9)



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)

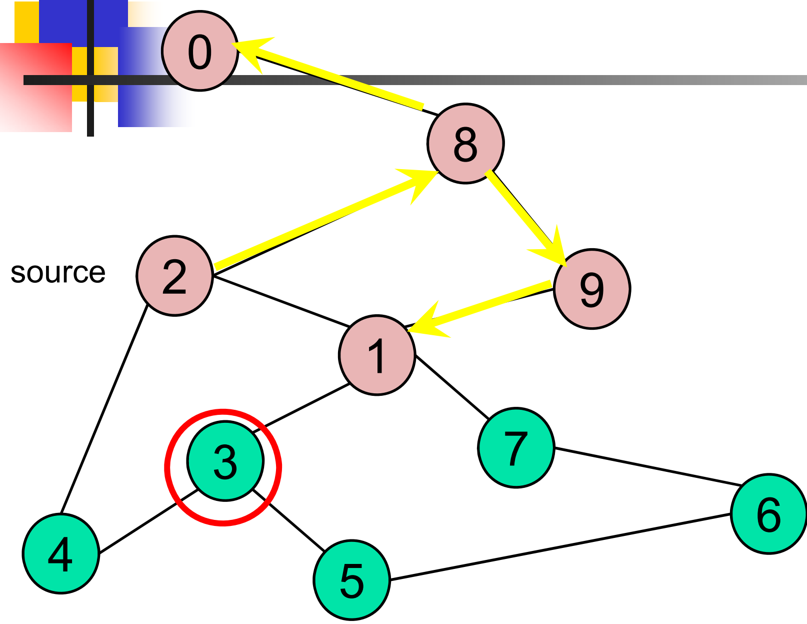
		Pred
0	T	8
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

Mark 9 as visited

Mark Pred[9]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(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)

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

Pred

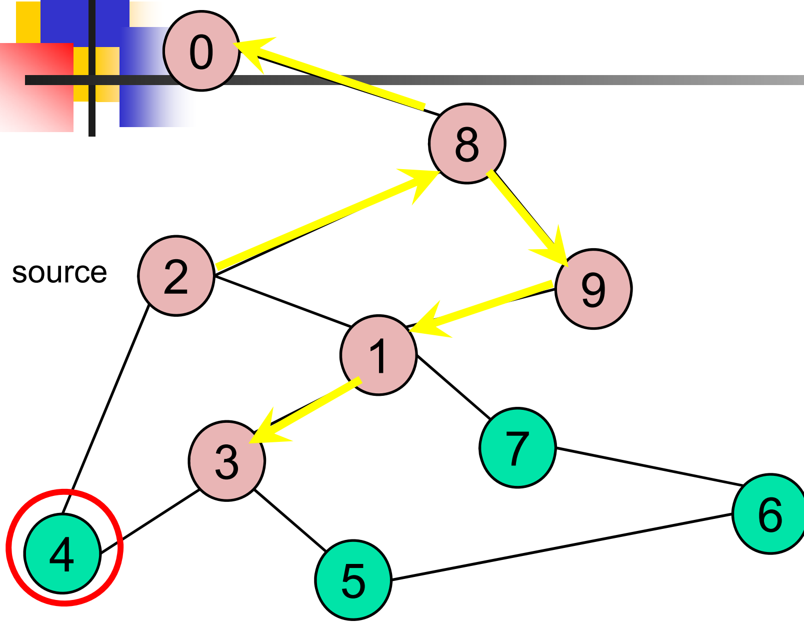
Mark 1 as visited

Mark Pred[1]

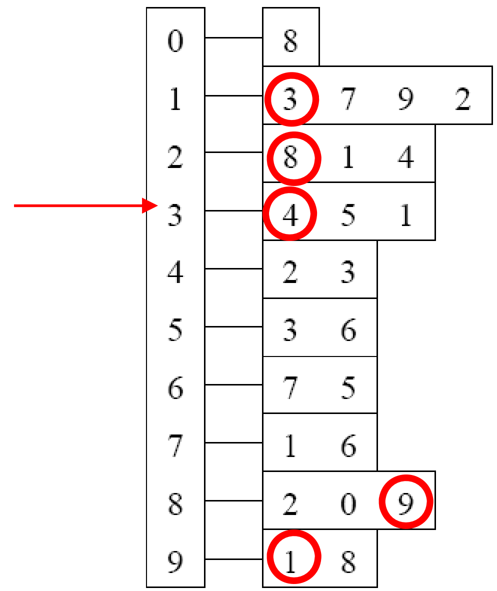
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)

source



Adjacency List



Visited Table (T/F)

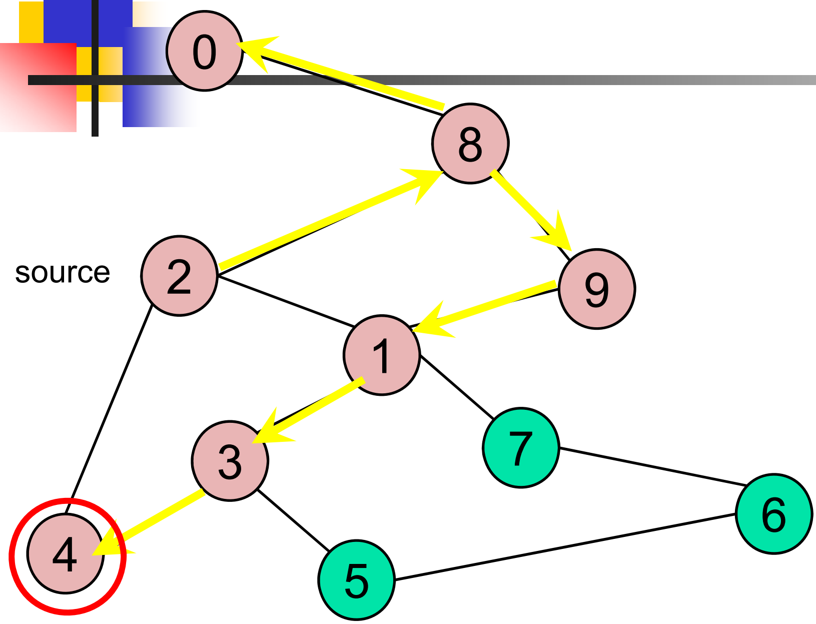
		Pred
0	T	8
1	T	9
2	T	-
3	T	1
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

Mark 3 as visited

Mark Pred[3]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)



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)

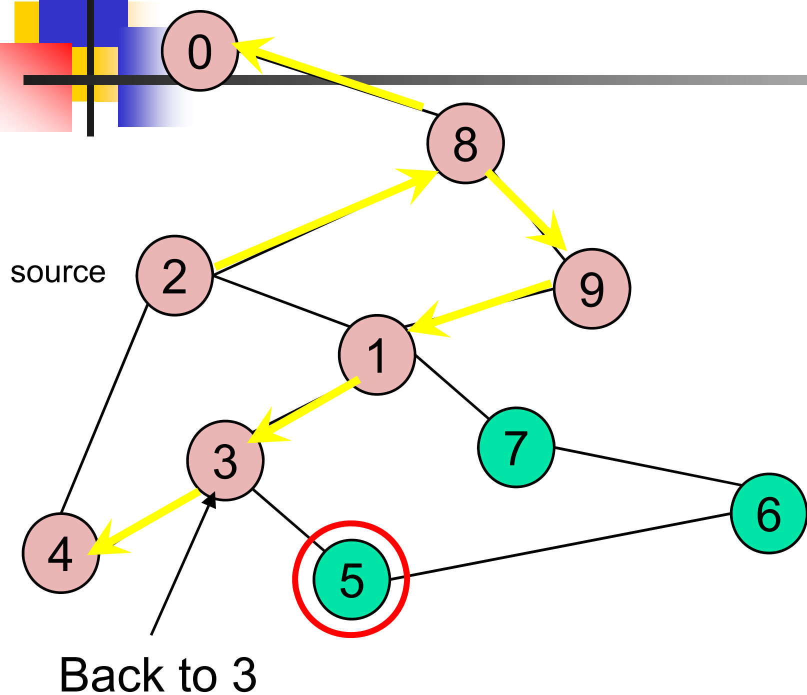
		Pred
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

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)



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

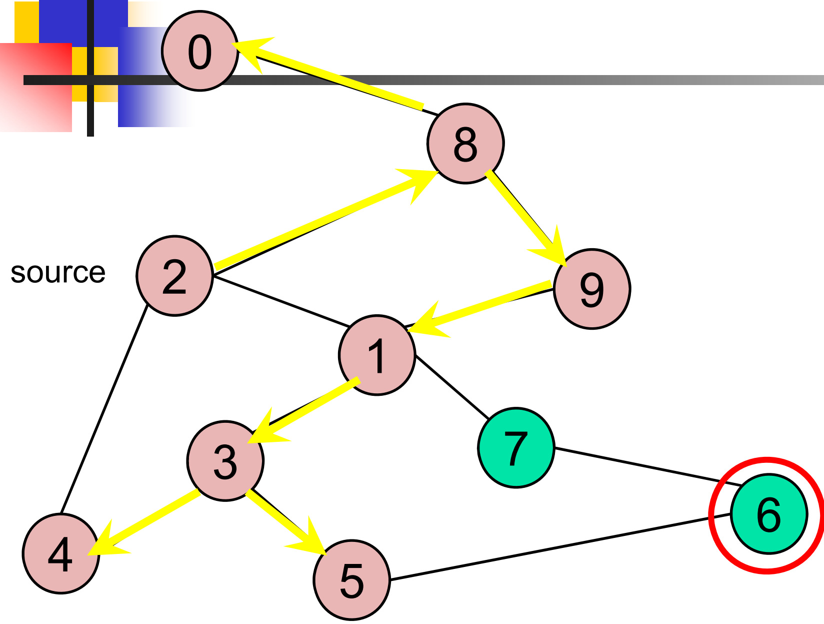
Visited Table (T/F)

		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

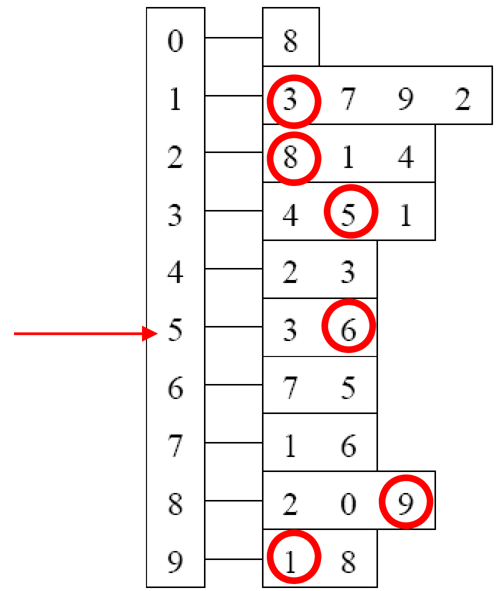
Pred

Recursive calls

RDFS(2)
  RDFS(8)
    RDFS(9)
      RDFS(1)
        RDFS(3)
          visit 5 -> RDFS(5)



Adjacency List



Visited Table (T/F)

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

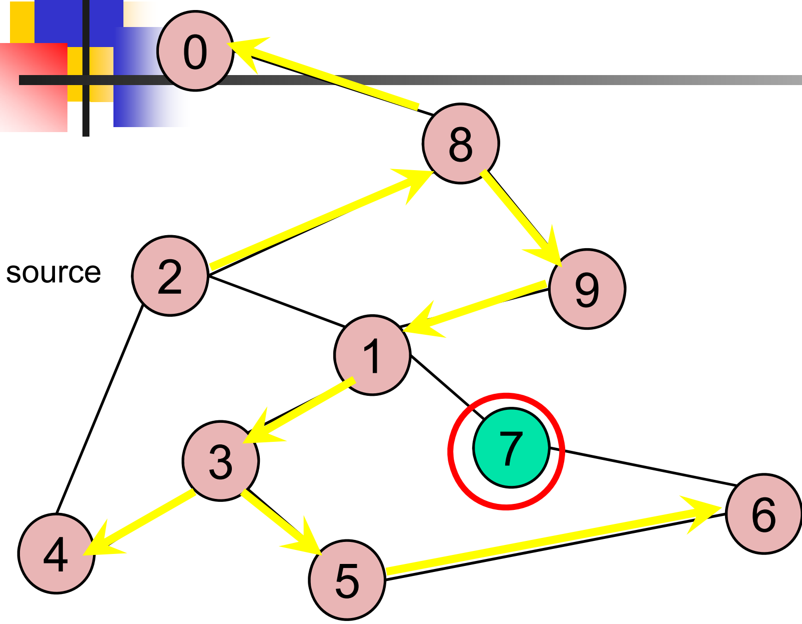
Pred

Recursive calls

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]



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

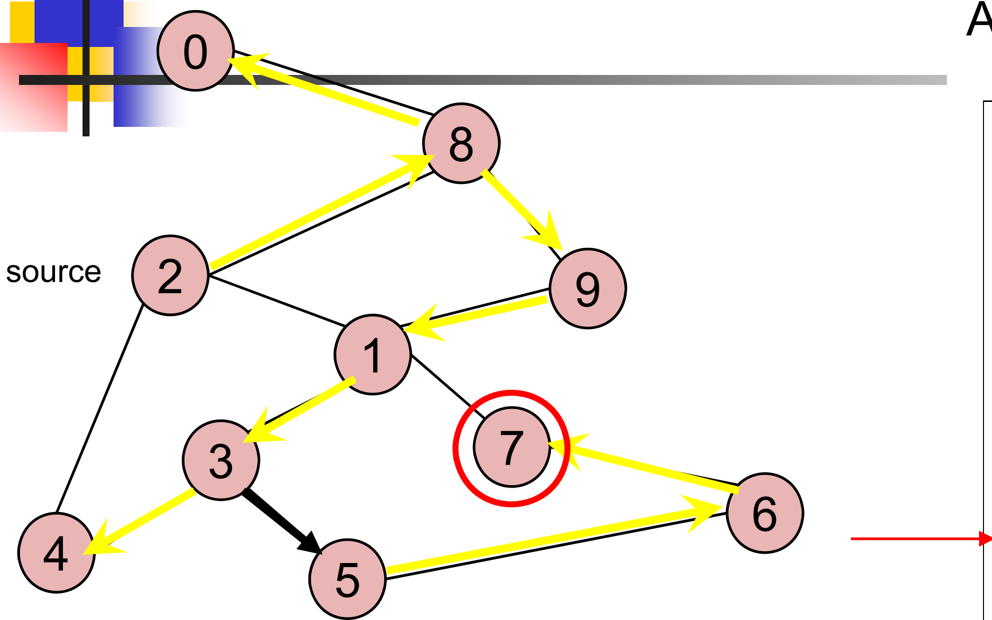
Pred

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

Mark 6 as visited

Mark Pred[6]



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	T	3
6	T	5
7	T	6
8	T	2
9	T	8

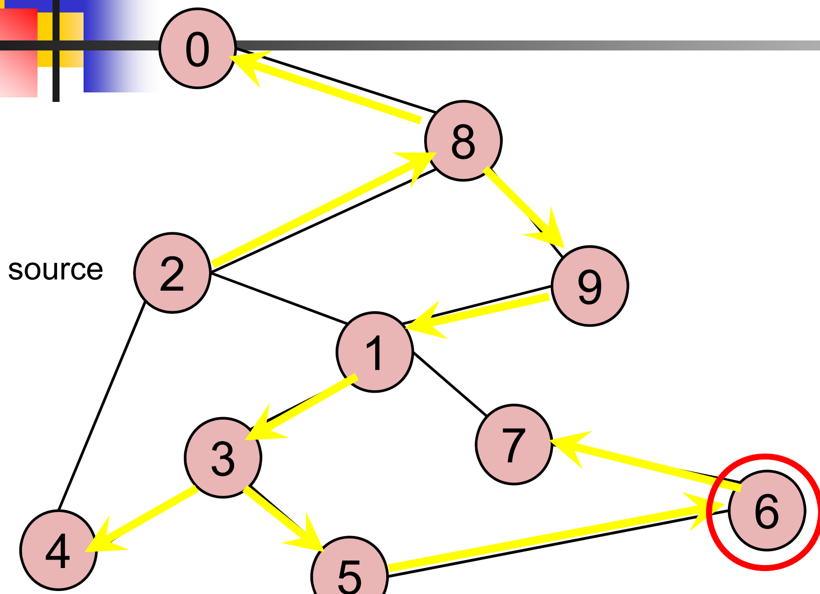
Pred

Recursive calls

RDFS(2)
  RDFS(8)
    RDFS(9)
      RDFS(1)
        RDFS(3)
          RDFS(5)
            RDFS(6)
              RDFS(7) -> Stop no more unvisited neighbors

Mark 7 as visited

Mark Pred[7]



Adjacency List

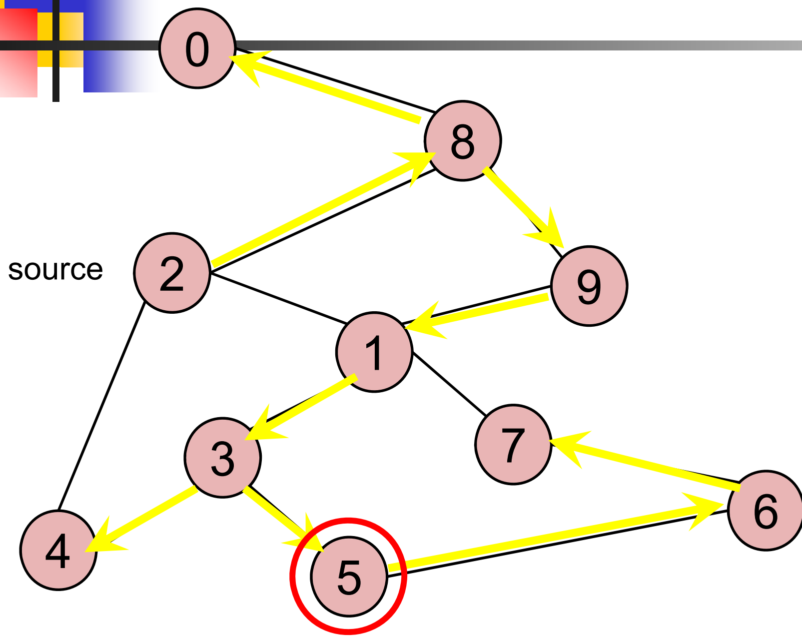
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

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

Recursive calls

RDFS(2)
- RDFS(8)
  - RDFS(9)
    - RDFS(1)
      - RDFS(3)
        - RDFS(5)
          - RDFS(6) -> Stop



Adjacency List

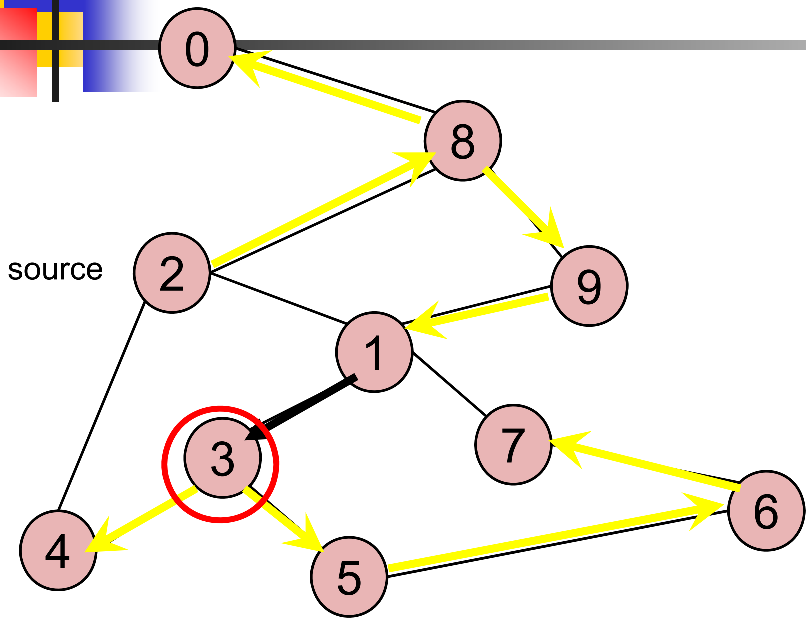
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

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

Recursive calls

RDFS(2)
    RDFS(8)
        RDFS(9)
            RDFS(1)
                RDFS(3)
                    RDFS(5) -> Stop



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

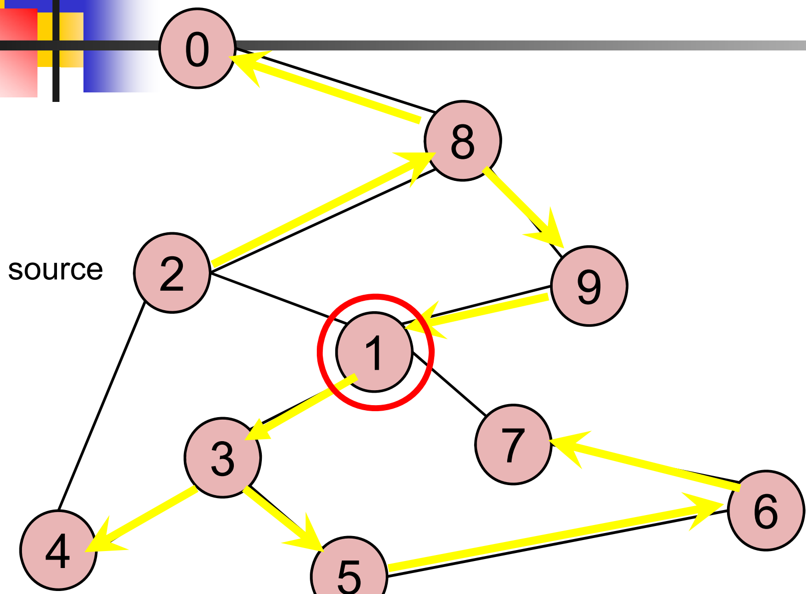
Visited Table (T/F)

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

Pred

Recursive calls

RDFS(2)
    RDFS(8)
        RDFS(9)
            RDFS(1)
                RDFS(3) -> Stop



Adjacency List

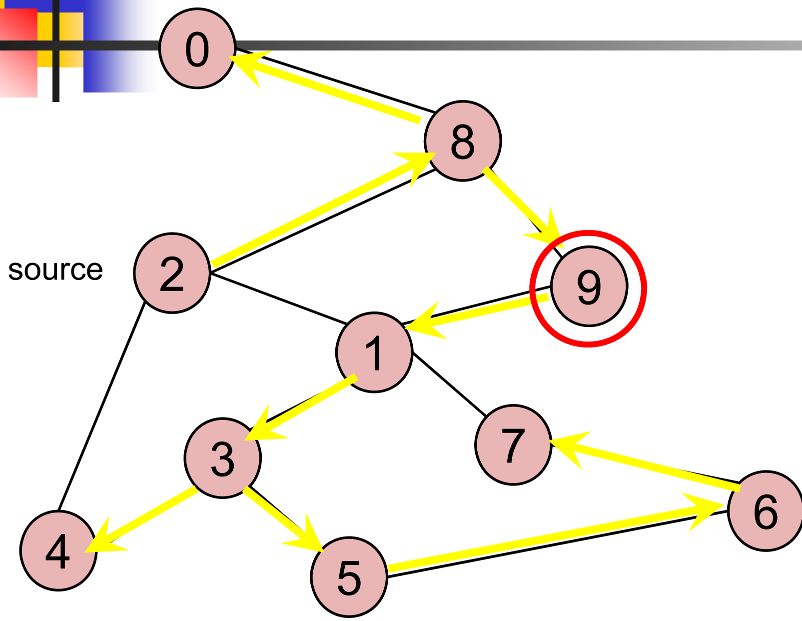
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

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

Recursive calls

RDFS(2)
    RDFS(8)
        RDFS(9)
            RDFS(1) -> Stop



Adjacency List

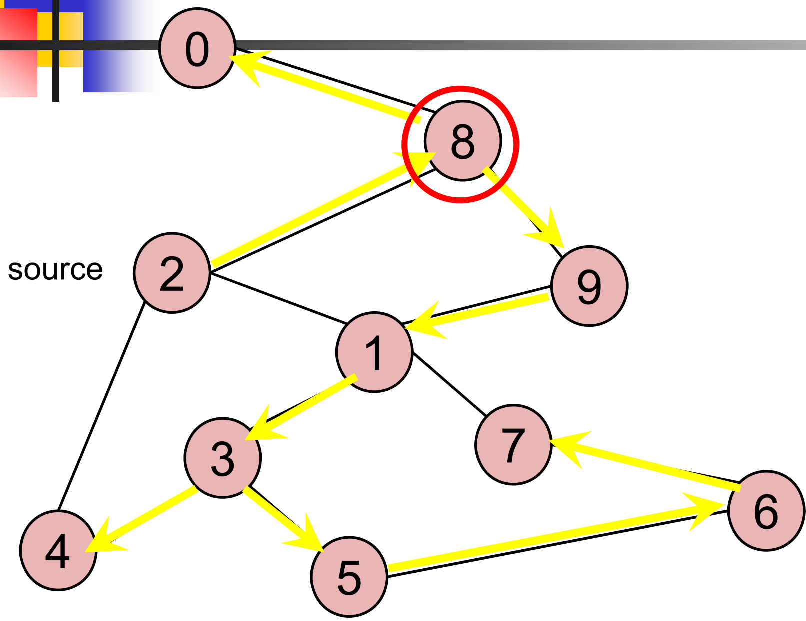
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

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

RDFS(2)
RDFS(8)
RDFS(9) -> Stop

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)

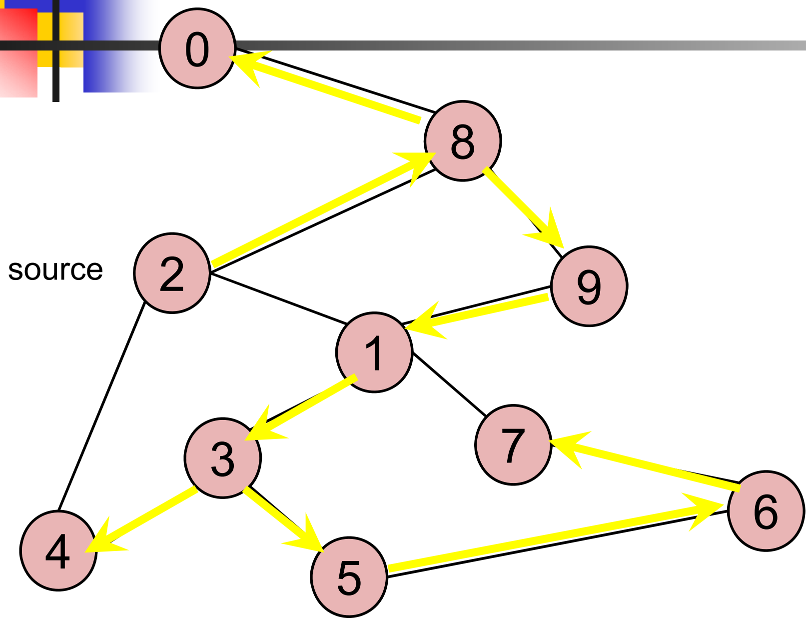
		Pred
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

RDFS(2)

RDFS(8) -> Stop

Recursive calls

Example Finished



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

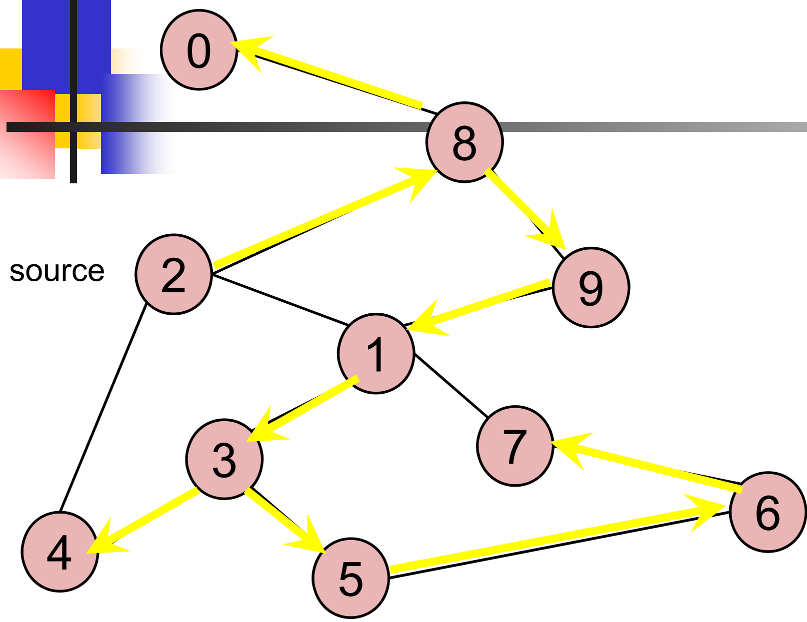
Visited Table (T/F)

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

RDFS(2) -> Stop

Recursive calls finished

DFS Path Tracking



DFS find out path too

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	T	3
6	T	5
7	T	6
8	T	2
9	T	8

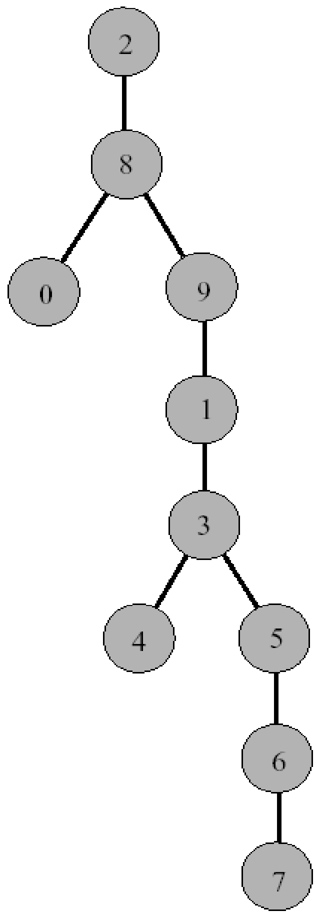
Pred

```
Algorithm Path(w)
    if pred[w] ≠ −1
        then
            Path(pred[w]);
    output w
```

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

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

Time Complexity of DFS
(Using adjacency list)

- We never visited a vertex more than once

- We had to examine all edges of the vertices
  - We know $\Sigma_{vertex\ v}$ degree(v) = 2m where m is the number of edges

- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)
  - O(n + m)

- You will also see this written as:
  - O(|v|+|e|) |v| = number of vertices (n)
    |e| = number of edges (m)