

Data Structures and Algorithms

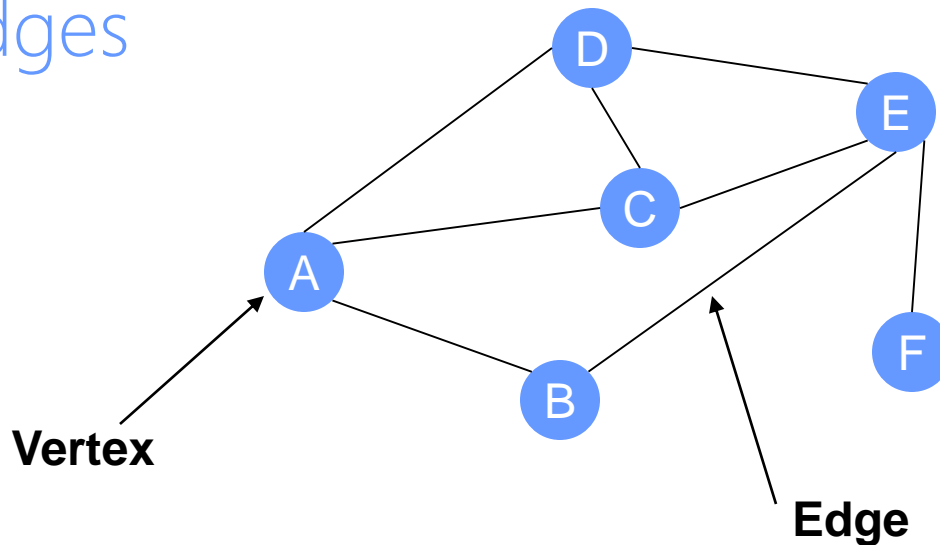
Graph and

Lecture 14: Breath First Search

Department of Computer Science & Technology
United International College

Graphs

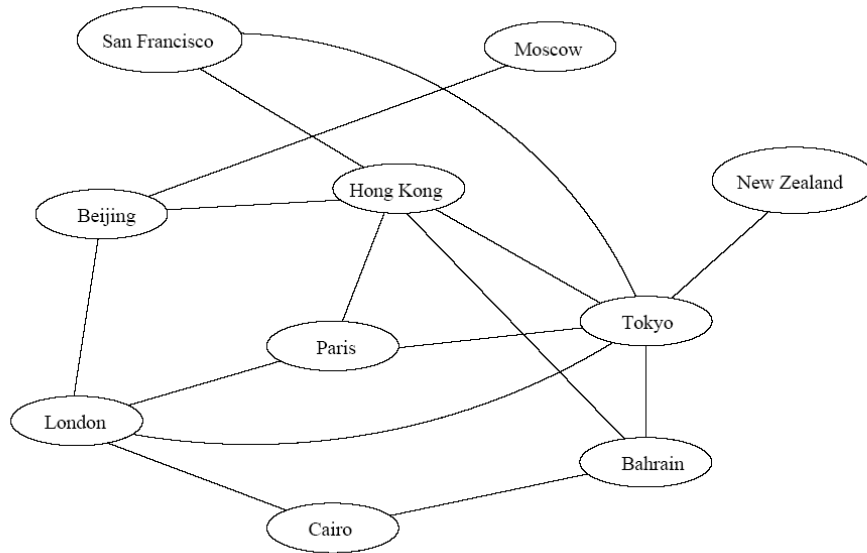
- Extremely useful tool in modeling problems
- Consist of:
 - Vertices
 - Edges



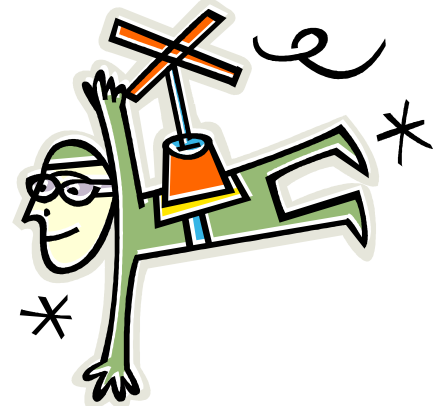
Vertices can be considered “sites” or locations.

Edges represent connections.

Application 1

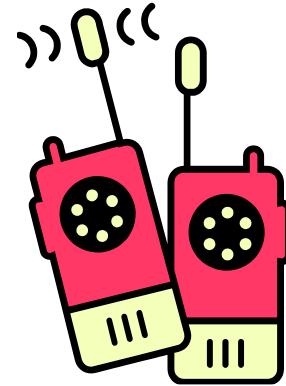
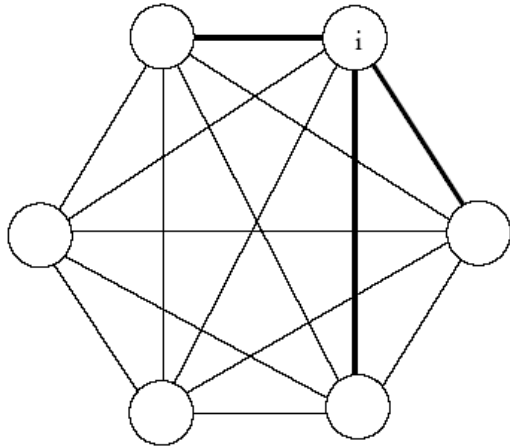


Air flight system



- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flights = a query on whether an edge exists
- A query on how to get to a location = does a **path** exist from A to B
- We can even associate costs to edges (**weighted graphs**), then ask “what is the cheapest path from A to B”

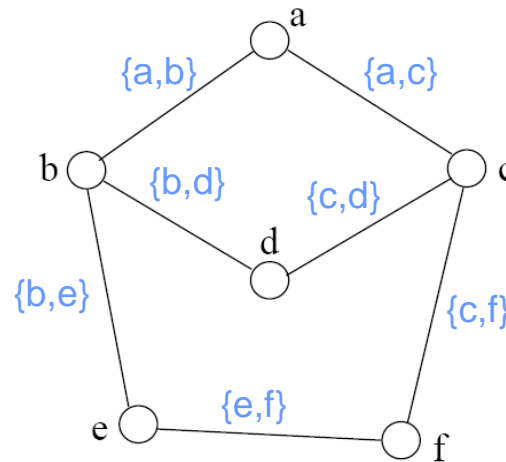
Application 2



- Represented by a **weighted complete graph** (every two vertices are connected by an edge)
- Each edge represents the **Euclidean distance** d_{ij} between two stations
- Each station uses a certain power i to transmit messages. Given this power i , only a few nodes can be reached (bold edges). A station reachable by i then uses its own power to relay the message to other stations not reachable by i .
- A typical wireless communication problem is: how to broadcast between *all* stations such that they are all connected and the power consumption is minimized.

Definition

- A graph $G=(V, E)$ consists a set of vertices, V , and a set of edges, E .
- Each edge is a pair of (v, w) , where v, w belongs to V
- If the pair is unordered, the graph is undirected; otherwise it is directed



$$V = \{a, b, c, d, e, f\}$$

$$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

An undirected graph

Terminology

1. If v_1 and v_2 are connected, they are said to be **adjacent** vertices
 - v_1 and v_2 are endpoints of the edge $\{v_1, v_2\}$
2. If an edge e is connected to v , then v is said to be **incident** on e . Also, the edge e is said to be **incident** on v .
3. $\{v_1, v_2\} = \{v_2, v_1\}$

If we are talking about directed graphs, where edges have direction. This means that $\{v_1, v_2\} \neq \{v_2, v_1\}$. Directed graphs are drawn with arrows (called arcs) between edges.



Graph Representation

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

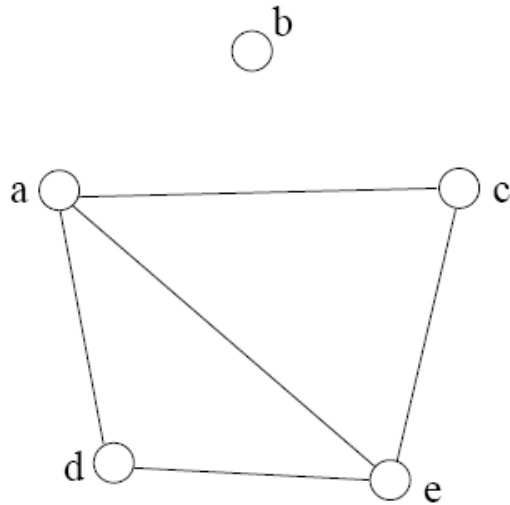
1. Adjacency Matrix

Use a 2D matrix to represent the graph

2. Adjacency List

Use a 1D array of linked lists

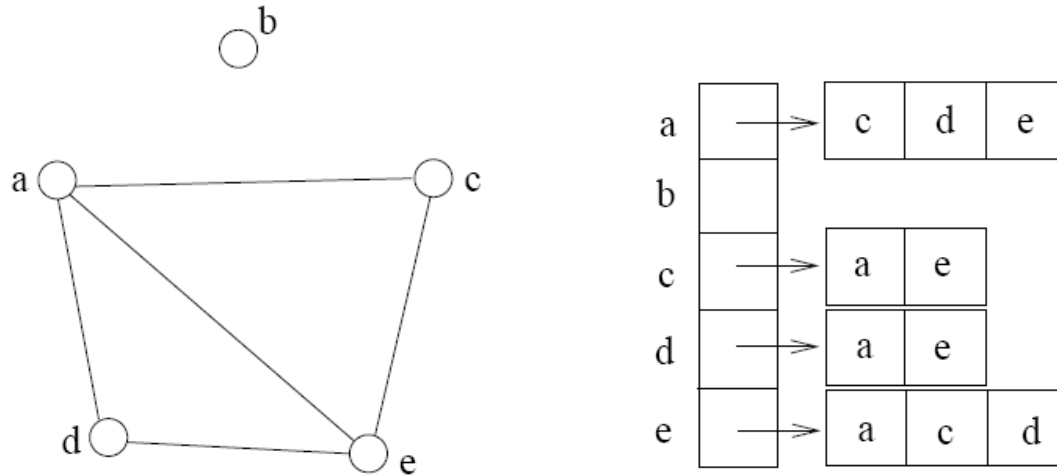
Adjacency Matrix



	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

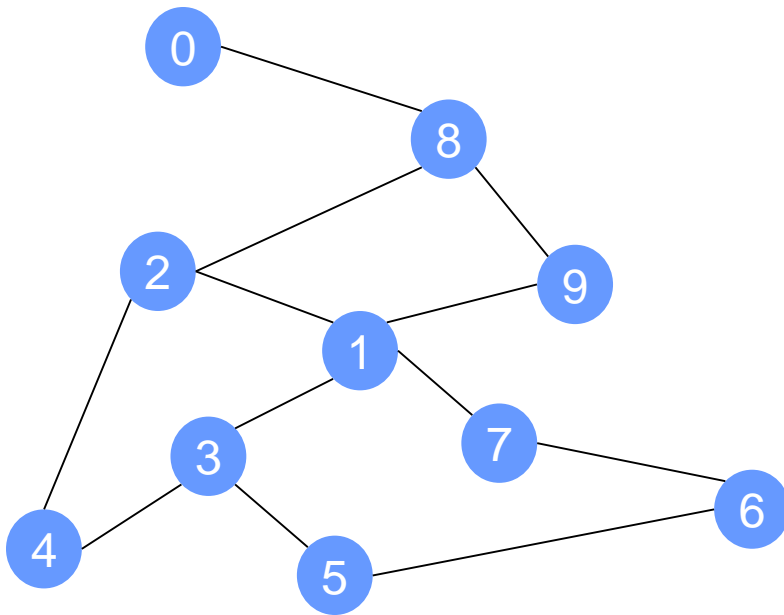
- 2D array $A[0..n-1, 0..n-1]$, where n is the number of vertices in the graph
- Each row and column is indexed by the vertex id
 - e.g. $a=0, b=1, c=2, d=3, e=4$
- $A[i][j]=1$ if there is an edge connecting vertices i and j ; otherwise $A[i][j]=0$.
- The storage requirement is $\Theta(n^2)$. It is not 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



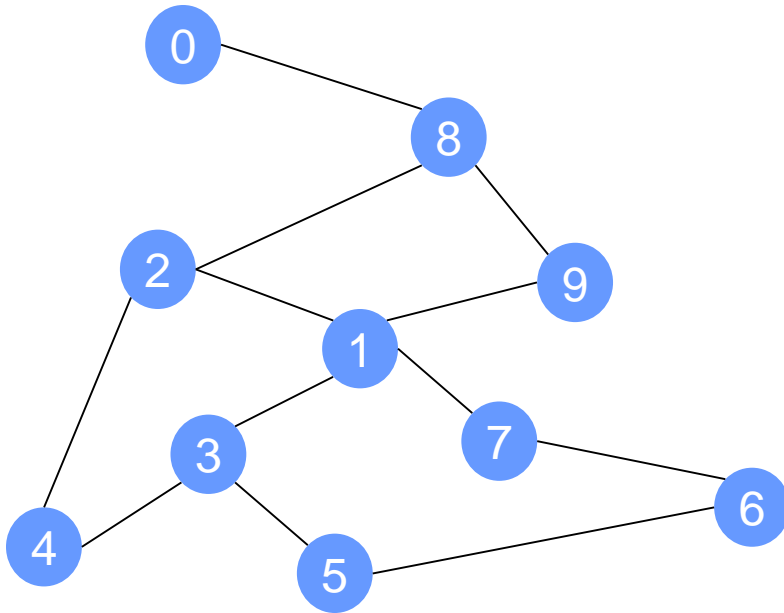
- If the graph is not dense, in other words, **sparse**, a better solution is an adjacency list
- The adjacency list is **an array $A[0..n-1]$ of lists**, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id
- Each **list $A[i]$** stores the **ids of the vertices 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, $\sum_{\text{vertex } v} \deg(v) = 2m$, 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 graph is sparse
 - Can quickly find if an edge exists

Path between Vertices

- A **path** is a sequence of vertices $(v_0, v_1, v_2, \dots, v_k)$ such that:
 - For $0 \leq i < k$, $\{v_i, v_{i+1}\}$ is an edge
 - For $0 \leq i < k-1$, $v_i \neq v_{i+2}$
That is, the edge $\{v_i, v_{i+1}\} \neq \{v_{i+1}, v_{i+2}\}$

Note: a path is allowed to go through the same vertex or the same edge any number of times!

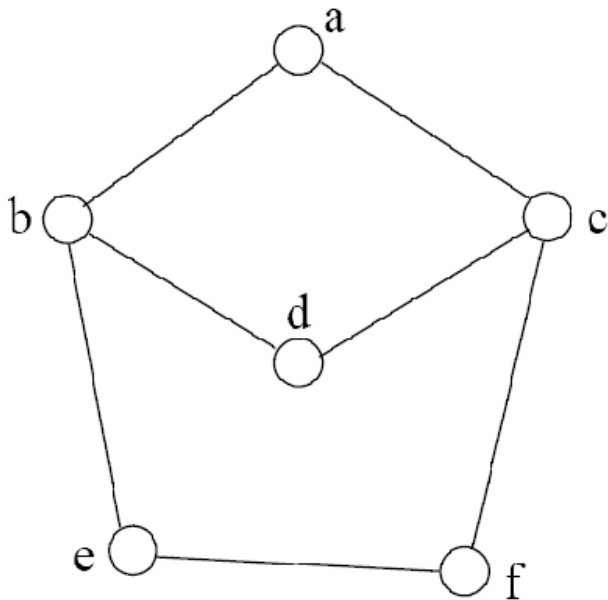
- The **length** of a path is the number of edges on the path

Types of paths



- A path is **simple** if and only if it does not contain a vertex more than once.
- A path is a **cycle** if and only if $v_0 = v_k$
 - The beginning and end are the same vertex!
- A path contains a cycle as its sub-path if some vertex appears twice or more

Path Examples



Are these paths?

Any cycles?

What is the path's length?

1. $\{a, c, f, e\}$
2. $\{a, b, d, c, f, e\}$
3. $\{a, c, d, b, d, c, f, e\}$
4. $\{a, c, d, b, a\}$
5. $\{a, c, f, e, b, d, c, a\}$

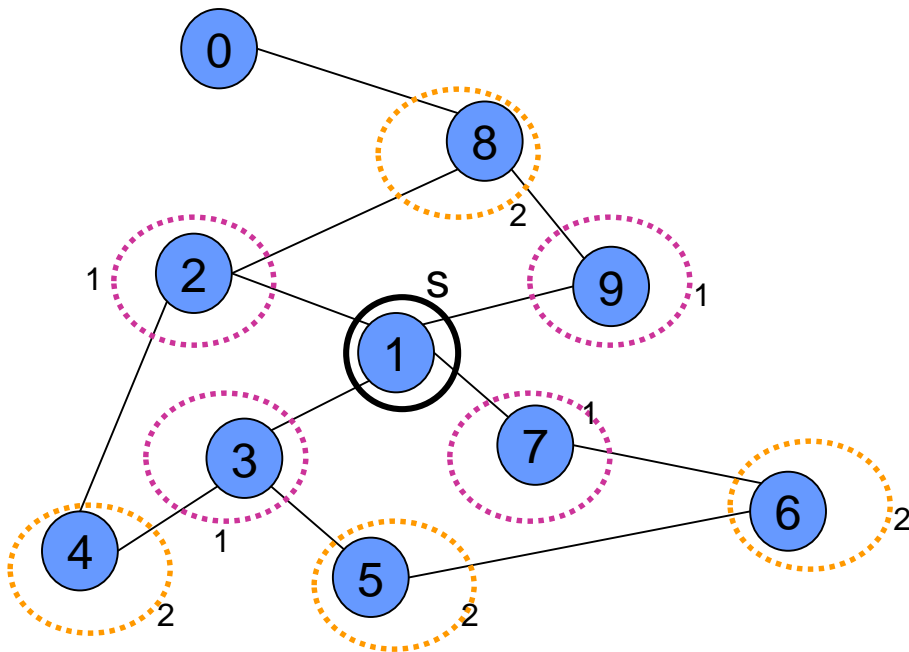
Graph Traversal



- Application example
 - Given a graph representation and a vertex s in the graph
 - Find all paths from s to other vertices
- Two common graph traversal algorithms
 - Breadth-First Search (BFS)
 - Find the shortest paths in an unweighted graph
 - Depth-First Search (DFS)
 - Topological sort
 - Find strongly connected components

BFS and Shortest Path Problem

- Given any source vertex s , BFS visits the other vertices at increasing distances away from s . In doing so, 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 $s = \text{vertex } 1$

Nodes at distance 1?

2, 3, 7, 9

Nodes at distance 2?

8, 6, 5, 4

Nodes at distance 3?

0

BFS Algorithm

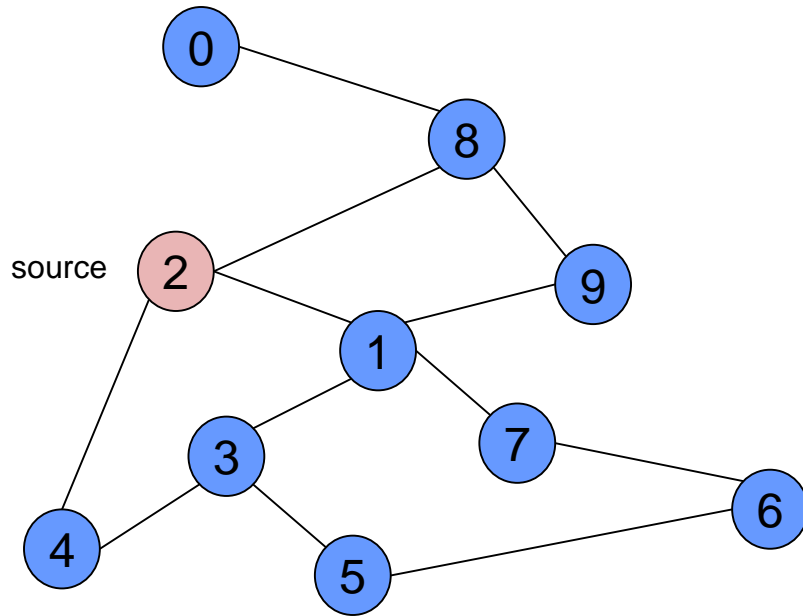
Algorithm $BFS(s)$

Input: s is the source vertex

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

1. **for** each vertex v
2. **do** $flag[v] := \text{false}$; // $flag[]$: visited table
3. $Q = \text{empty queue}$; **Why use queue? Need FIFO**
4. $flag[s] := \text{true}$;
5. $enqueue(Q, s)$;
6. **while** Q is not empty
7. **do** $v := dequeue(Q)$;
8. **for** each w adjacent to v
9. **do if** $flag[w] = \text{false}$
10. **then** $flag[w] := \text{true}$;
11. $enqueue(Q, w)$

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)

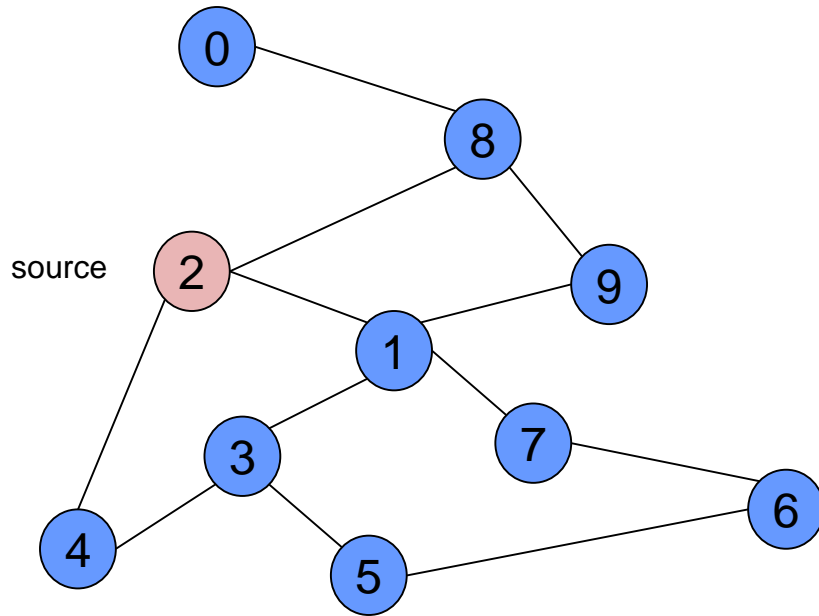
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

$Q = \{ \}$

Initialize **Q** to be empty

Initialize visited
table (all False)

BFS Example (Cont'd)


$$Q = \{ 2 \}$$

Place source 2 on the queue

Adjacency List

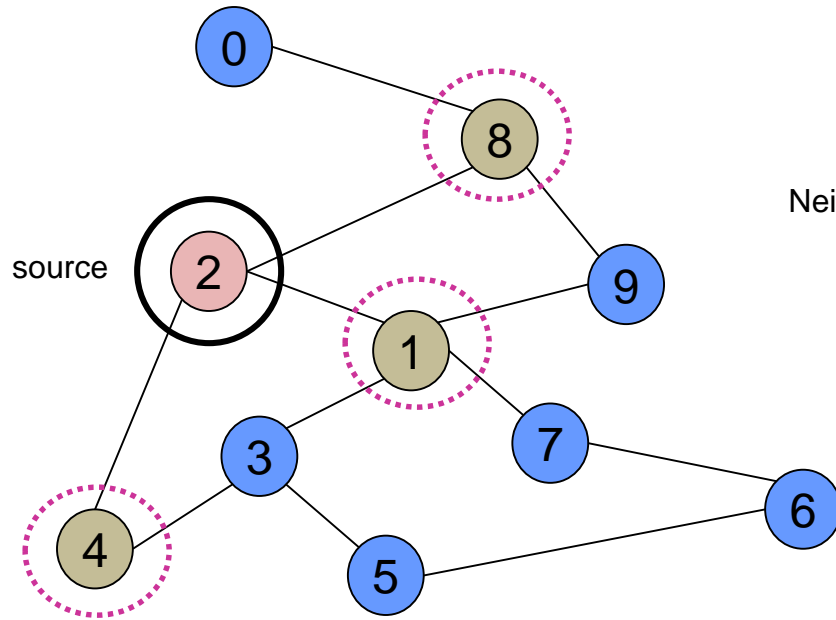
0	8				
1	3	7	9	2	
2	8	1	4		
3	4	5	1		
4	2	3			
5	3	6			
6	7	5			
7	1	6			
8	2	0	9		
9	1	8			

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

BFS Example (Cont'd)



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

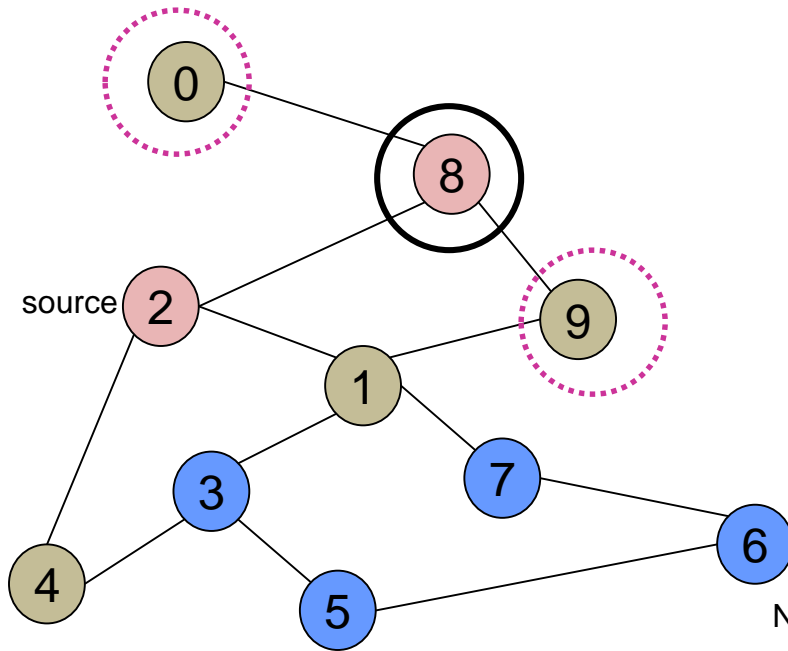
$Q = \{2\} \rightarrow \{8, 1, 4\}$

Dequeue 2.

Place all **unvisited** neighbors of 2 on the queue

Mark neighbors
as visited 1, 4, 8

BFS Example (Cont'd)



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

Neighbors →

Visited Table (T/F)

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

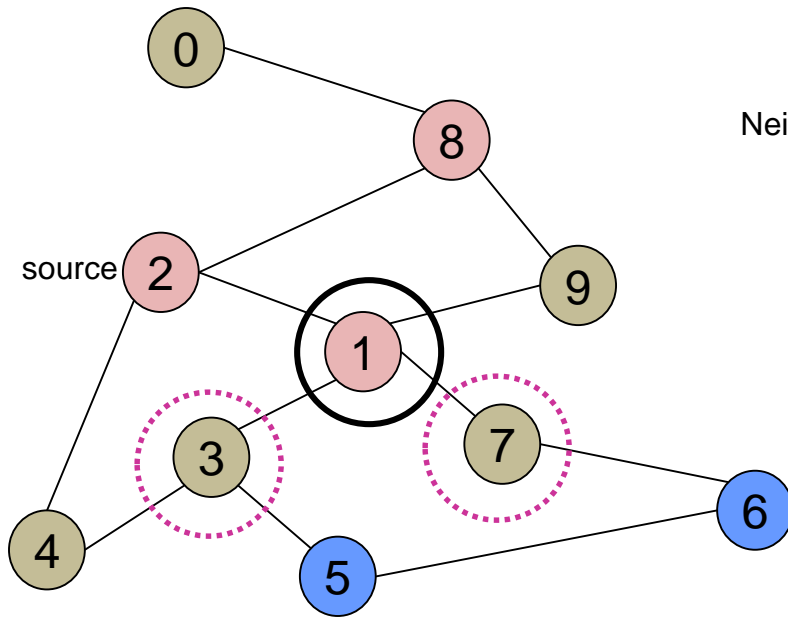
Mark new visited
Neighbors 0, 9

$Q = \{ 8, 1, 4 \} \rightarrow \{ 1, 4, 0, 9 \}$

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!

BFS Example (Cont'd)



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

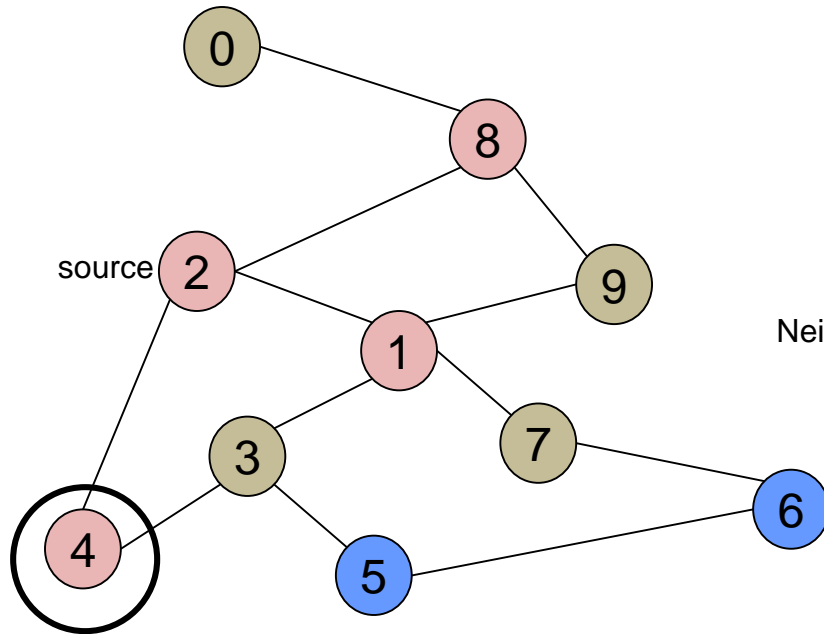
$Q = \{ 1, 4, 0, 9 \} \rightarrow \{ 4, 0, 9, 3, 7 \}$

Mark new visited
Neighbors 3, 7

Dequeue 1.

- Place all unvisited neighbors of 1 on the queue.
- Only nodes 3 and 7 haven't been visited yet.

BFS Example (Cont'd)



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)

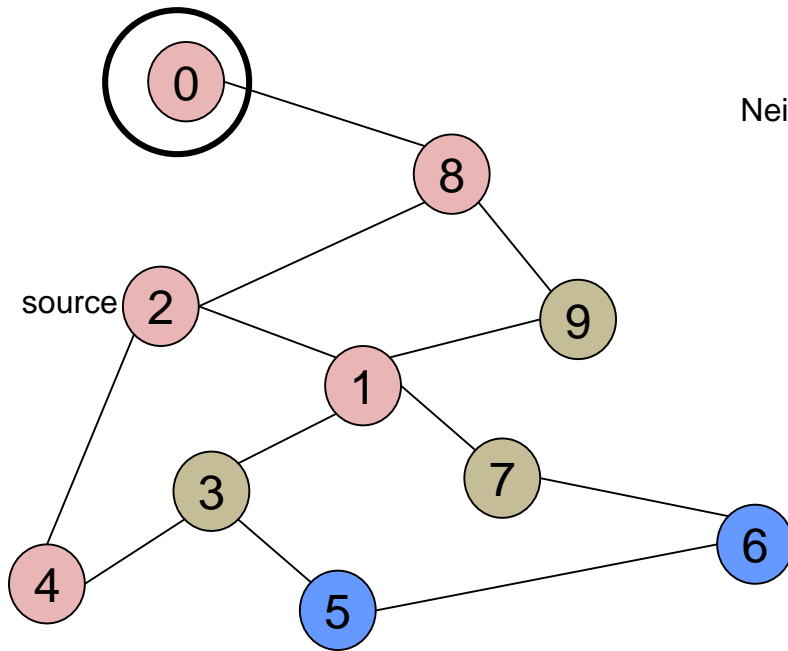
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

$Q = \{4, 0, 9, 3, 7\} \rightarrow \{0, 9, 3, 7\}$

Dequeue 4.

-- 4 has no unvisited neighbors!

BFS Example (Cont'd)



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)

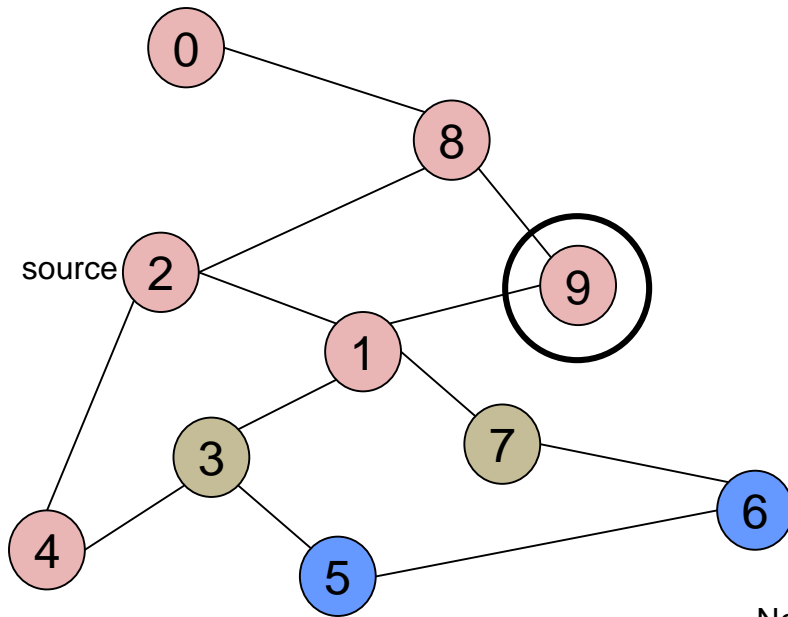
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

$Q = \{0, 9, 3, 7\} \rightarrow \{9, 3, 7\}$

Dequeue 0.

-- 0 has no unvisited neighbors!

BFS Example (Cont'd)



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)

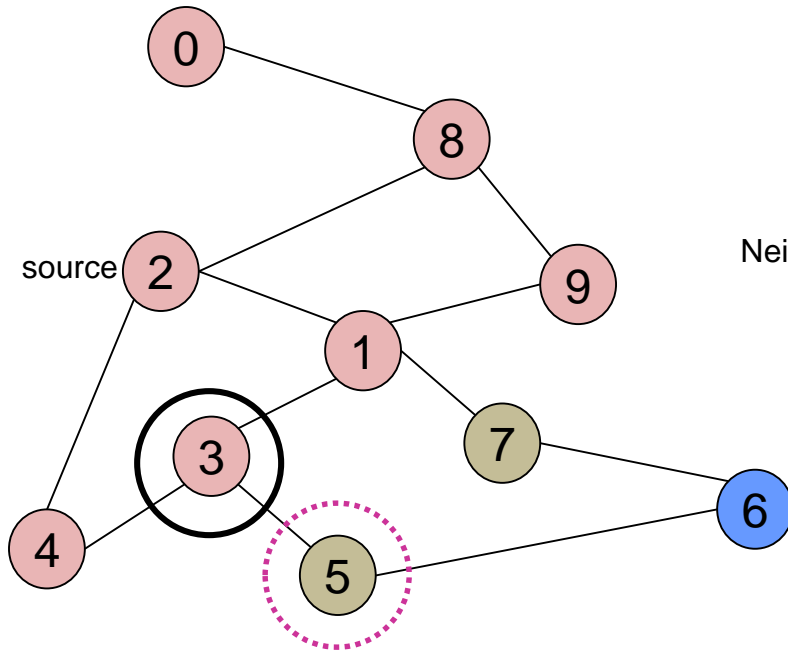
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

$Q = \{9, 3, 7\} \rightarrow \{3, 7\}$

Dequeue 9.

-- 9 has no unvisited neighbors!

BFS Example (Cont'd)



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	T
8	T
9	T

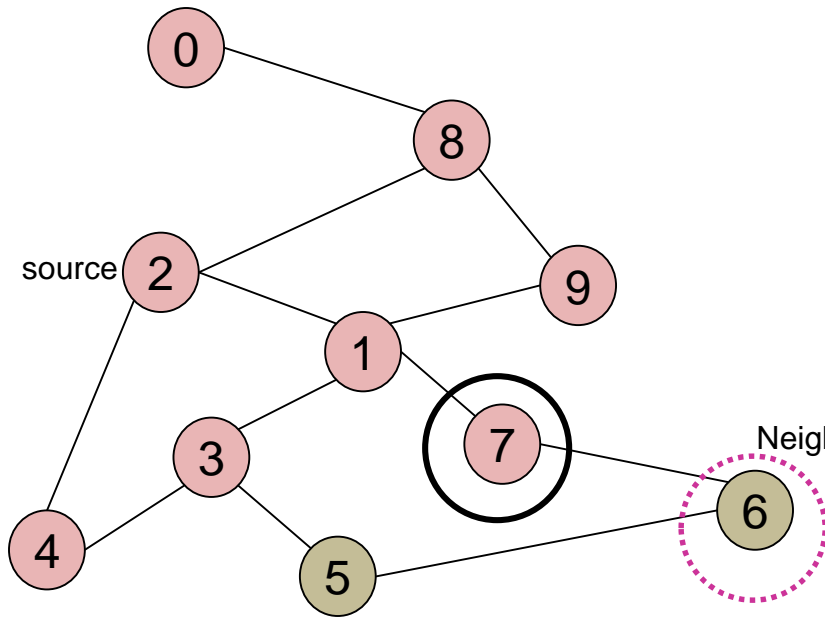
Mark new visited
Vertex 5

$Q = \{3, 7\} \rightarrow \{7, 5\}$

Dequeue 3.

-- place neighbor 5 on the queue.

BFS Example (Cont'd)



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

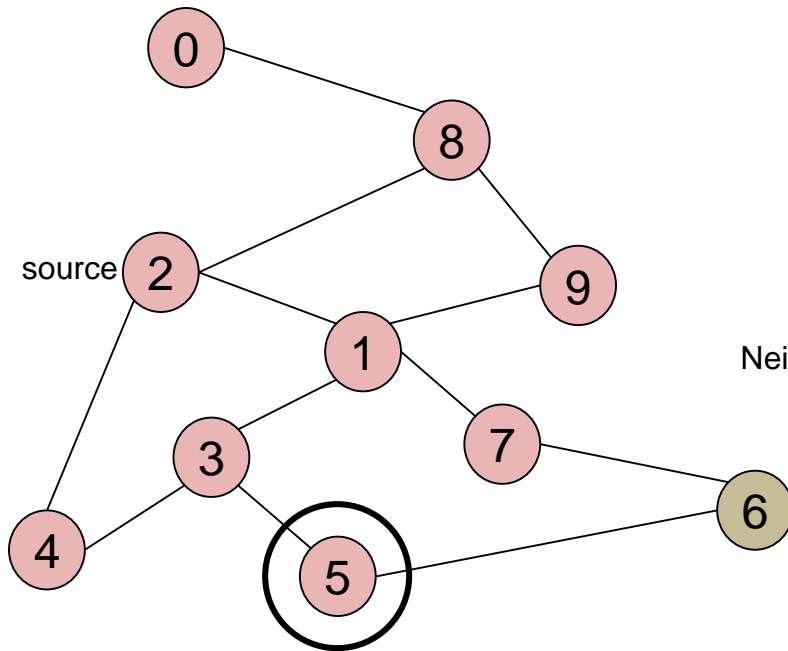
Mark new visited
Vertex 6

$Q = \{7, 5\} \rightarrow \{5, 6\}$

Dequeue 7.

-- place neighbor 6 on the queue

BFS Example (Cont'd)



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)

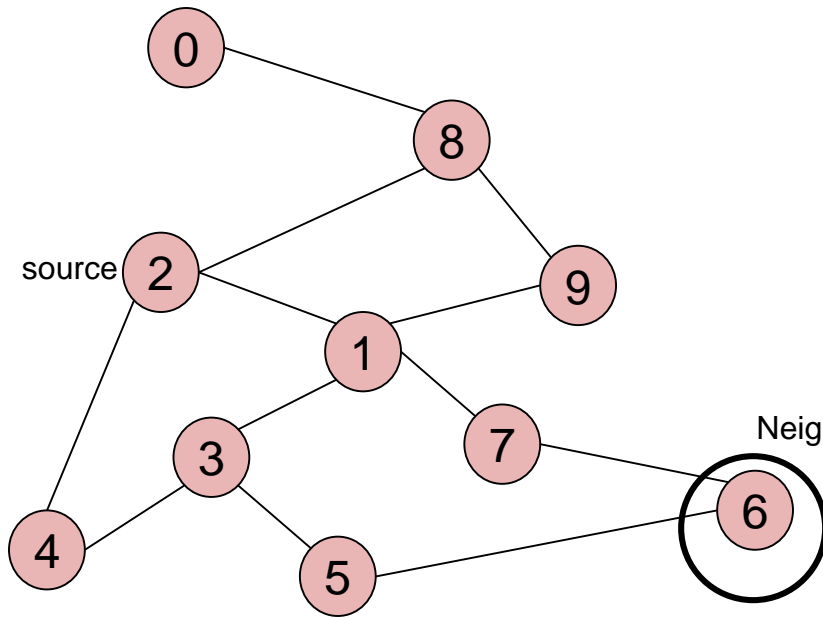
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

$Q = \{ 5, 6 \} \rightarrow \{ 6 \}$

Dequeue 5.

-- no unvisited neighbors of 5

BFS Example (Cont'd)



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)

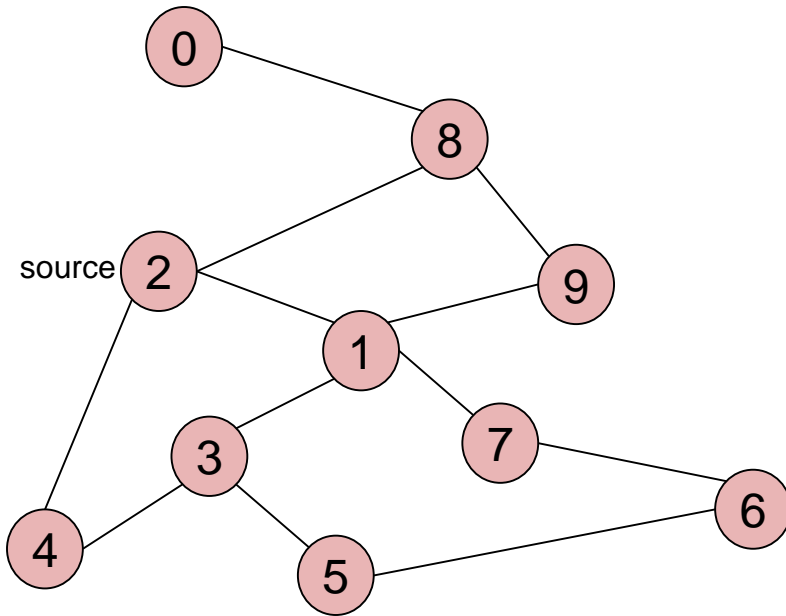
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

$Q = \{6\} \rightarrow \{ \}$

Dequeue 6.

-- no unvisited neighbors of 6

BFS Example (Cont'd)



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 .

```
1.  for each vertex  $v$ 
2.      do  $flag[v] := false$ ;
3.   $Q =$  empty queue;
4.   $flag[s] := true$ ;
5.   $enqueue(Q, s)$ ;
6.  while  $Q$  is not empty
7.      do  $v := dequeue(Q)$ ;
8.          for each  $w$  adjacent to  $v$ 
9.              do if  $flag[w] = false$ 
10.                  then  $flag[w] := true$ ;
11.                       $enqueue(Q, w)$ 
```

$O(n + m)$

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?

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

- The **total** running time of the while loop is:

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

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 .

```
1.  for each vertex  $v$ 
2.      do  $flag[v] := false$ ;
3.   $Q =$  empty queue;
4.   $flag[s] := true$ ;
5.   $enqueue(Q, s)$ ;
6.  while  $Q$  is not empty
7.      do  $v := dequeue(Q)$ ;
8.          for each  $w$  adjacent to  $v$ 
9.              do if  $flag[w] = false$ 
10.                  then  $flag[w] := true$ ;
11.                       $enqueue(Q, w)$ 
```

$O(n^2)$

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, $O(n+m)=O(n^2)$.