DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 12
Graph - Representation, Traversal & Shortest Path Algorithm

Instructor: Engr. Nasr Kamal

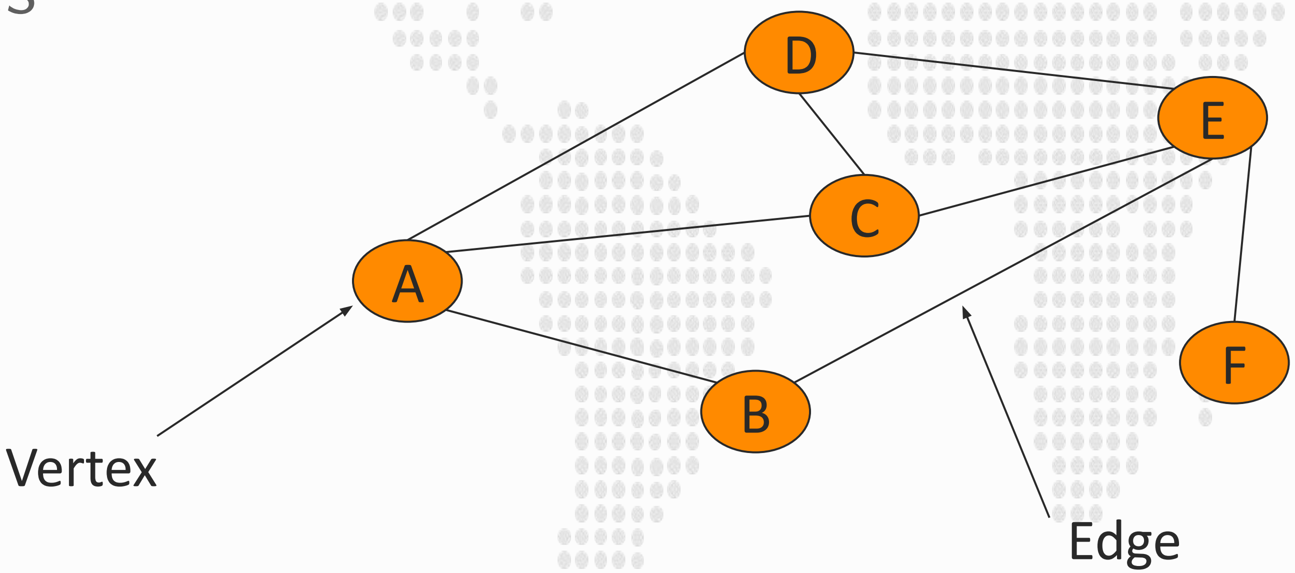
Department of Computer Science and Information Technology

Graph: Definition

- A *graph consists of a set of vertices and a set of edges.*
- *Think of a map of your* state.
- Each town is connected with other towns via some type of road.
- A map is a type of graph.
- Each town is a vertex and a road that connects two towns is an edge.
- Edges are specified as a *pair, (v1, v2), where v1 and v2 are two* vertices in the graph.
- A vertex can also have a *weight, sometimes also called* a cost.

Graphs

- A graph G=(V,E), where V is a set of vertices and E is a set of edges.
- Consist of:
  - Vertices
  - Edges

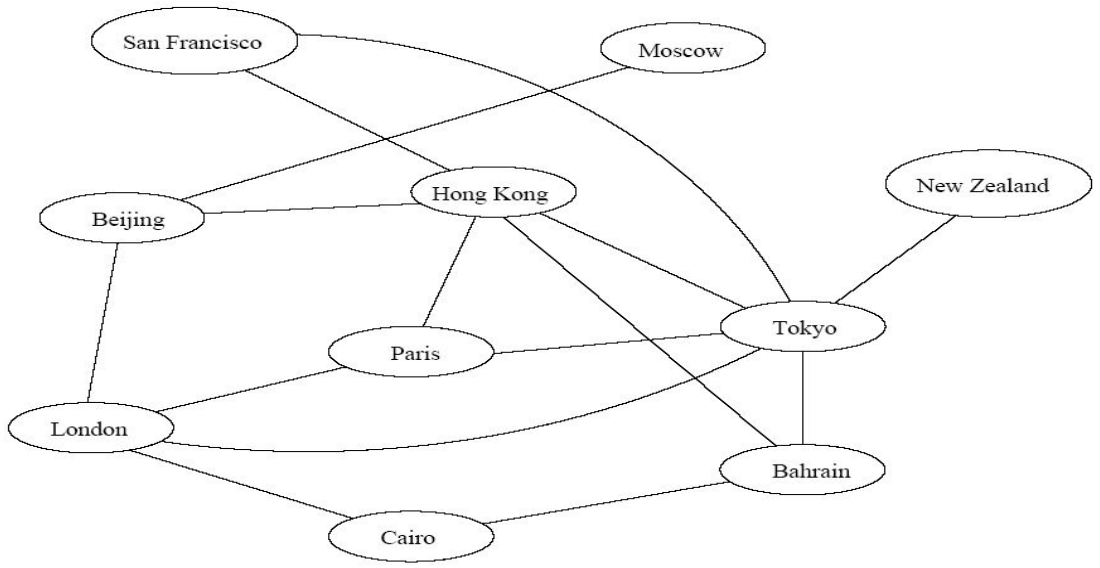


Vertices can be considered “sites” or locations.

Edges represent connections.

Applications

Air flight system



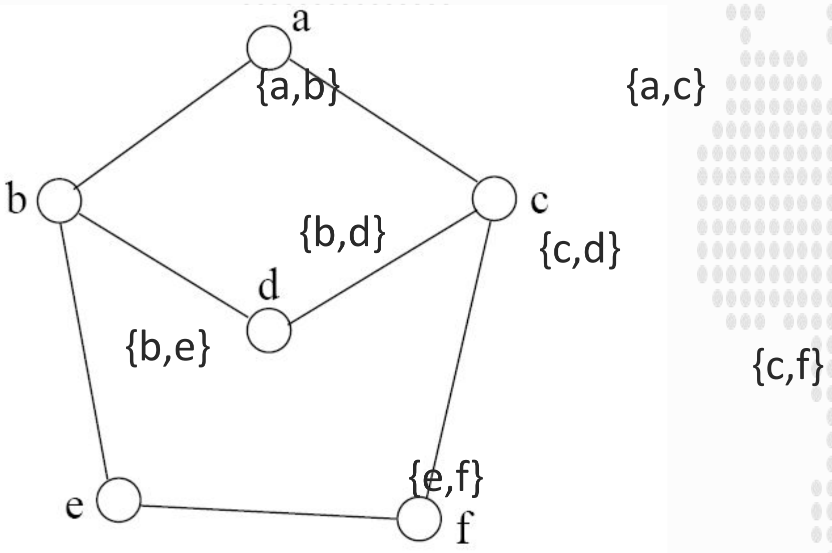
- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flights becomes a query on whether an edge exists
- A query on how to get to a location is “does a path exist from A to B”
- We can even associate costs to edges (weighted graphs), then ask “what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

Undirected Graph

Definition

Undirected graph

An undirected graph is specified by an ordered pair (V,E), where V is the set of vertices and E is the set of edges



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

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

Graph Representation

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

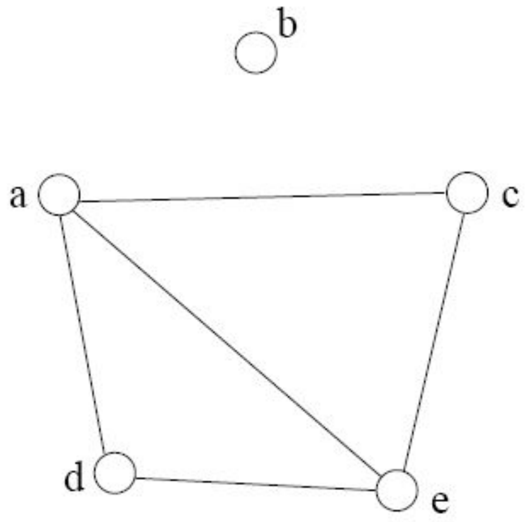
1. Adjacency Matrix

   Use a 2D matrix to represent the graph

1. Adjacency List

   Use 2D array of linked lists

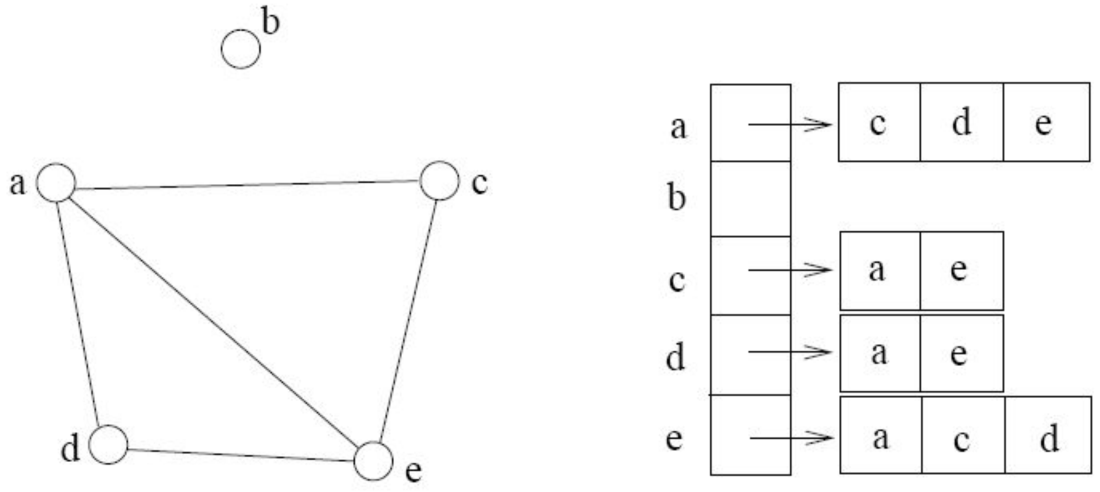
Adjacency Matrix



	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

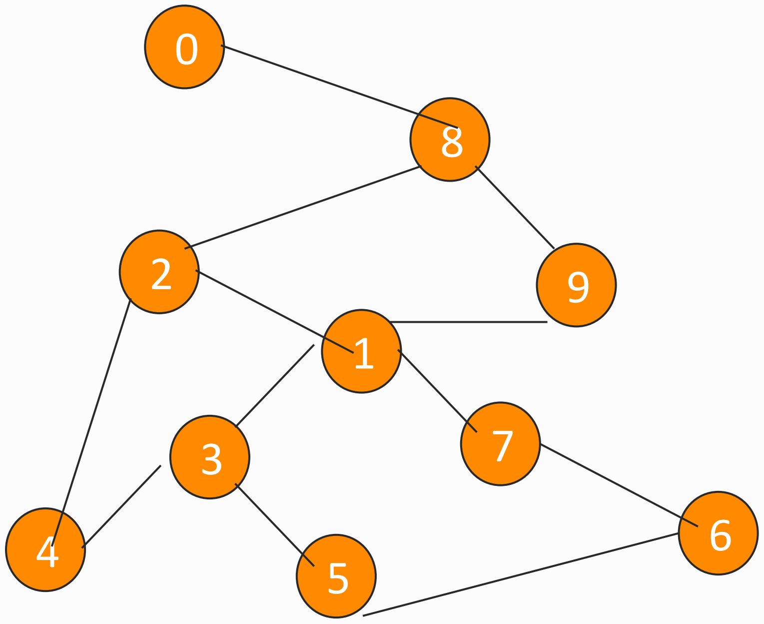
- 2D array A[0..n-1, 0..n-1], where *n* is the number of vertices in the graph
- Each row and column is indexed by the vertex id.
  - e,g a=0, b=1, c=2, d=3, e=4
- An array entry A [i] [j] is equal to 1 if there is an edge connecting
- vertices i and j. Otherwise, A [i] [j] is 0.
- The storage requirement is $\Theta(n^2)$. Not efficient if the graph has few edges.
- We can detect in O(1) time whether two vertices are connected.

Adjacency list



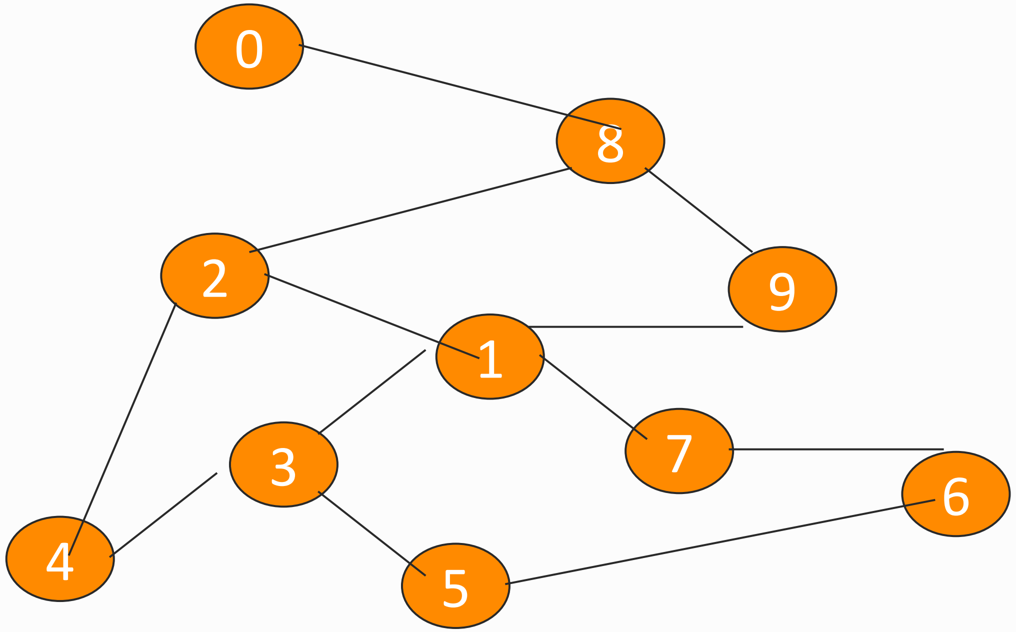
- The adjacency list is an array A[0..n-1] of lists, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id (as with adjacency matrix)
- The list A[i] stores the ids of the vertices adjacent to i.

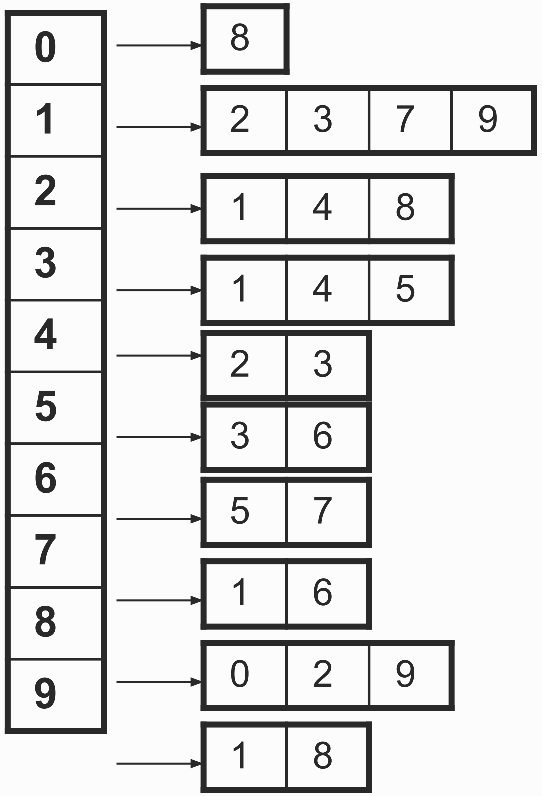
Examples



	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

Examples





Adjacency Lists vs. Matrix

Both representations have their use cases:

Adjacency Lists

- Efficient in terms of memory for sparse graphs, but slower for edge existence checks.
- More compact than adjacency matrices if graph has few edges
- Requires more time to find if an edge exists

Adjacency Matrix

- Fast to check if an edge exists, but uses more memory for sparse graphs.
- 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

Path between Vertices

A path is a sequence of vertices (v_0, v_1, v_2,… 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 if some vertex appears twice or more

Graph Traversal

Two common graph traversal algorithms

- **Breadth-First Search (BFS)**
- **Depth-First Search (DFS)**

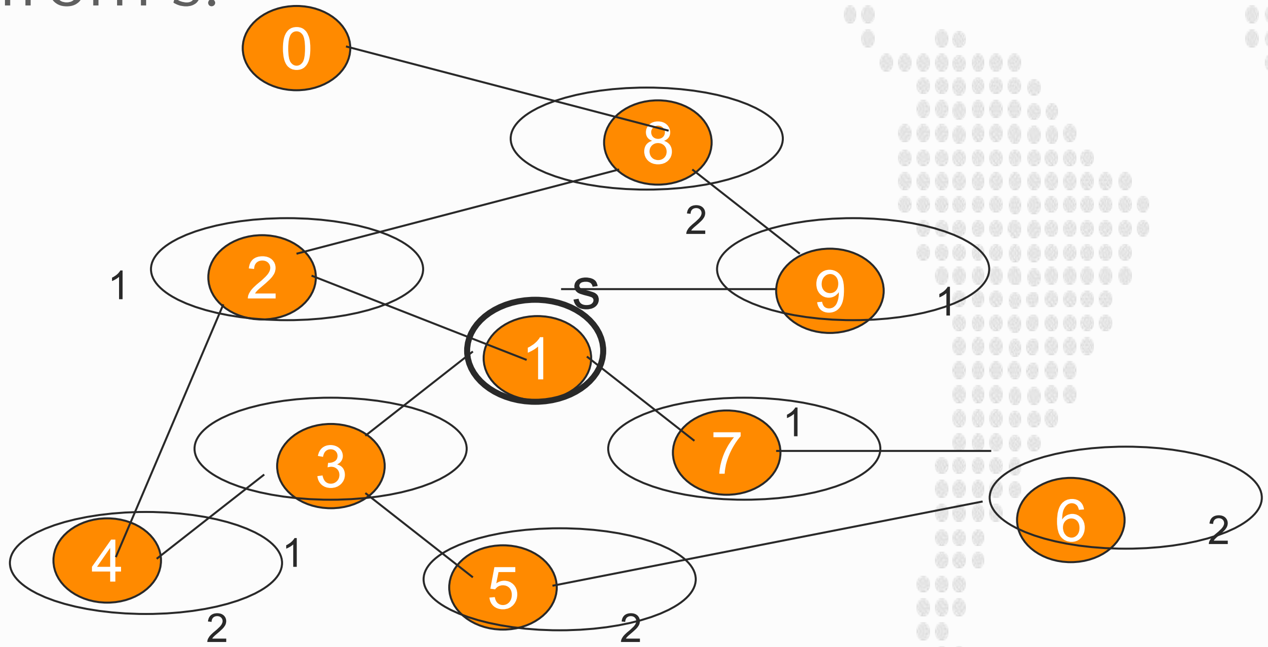
BFS and DFS are fundamental graph traversal algorithms, commonly used to explore nodes and edges of a graph

Breadth First Search - BFS

- BFS is a method for traversing a graph in **layers.**
- It is particularly useful for finding the shortest path in an unweighted graph, testing connectivity, and solving a variety of graph-related problems.
- Starting from a given node (often referred to as the **source node**), BFS explores all of the node's neighbors first, then the neighbors of those neighbors, and so on.
- It uses a **queue** data structure to keep track of nodes to visit next, ensuring that nodes are visited level by level.

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=vertex 1

Nodes at distance 1?
2, 3, 7, 9

Nodes at distance 2?
8, 6, 5, 4

Nodes at distance 3?
0

BSF 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 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)
```

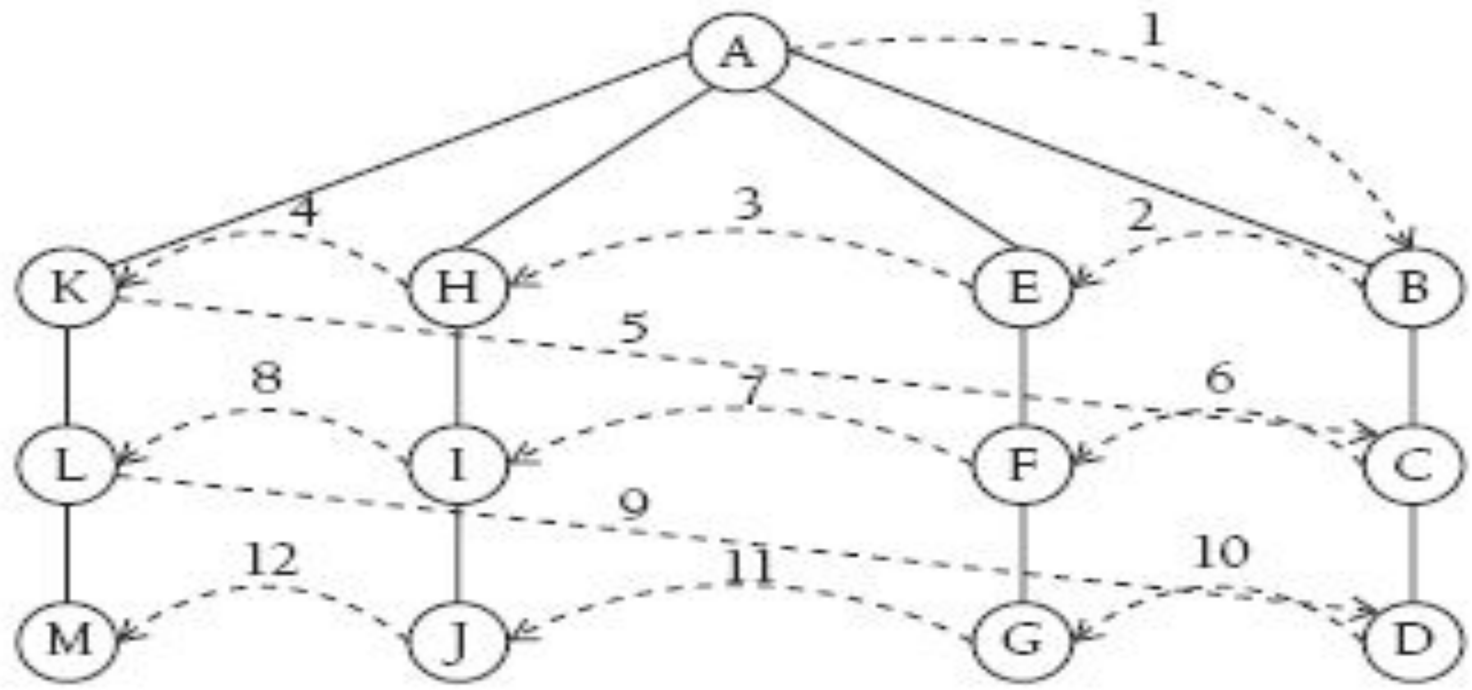
Why use queue? Need FIFO

BFS Algorithm

The algorithm for breadth-first search uses a queue. The algorithm is as follows:

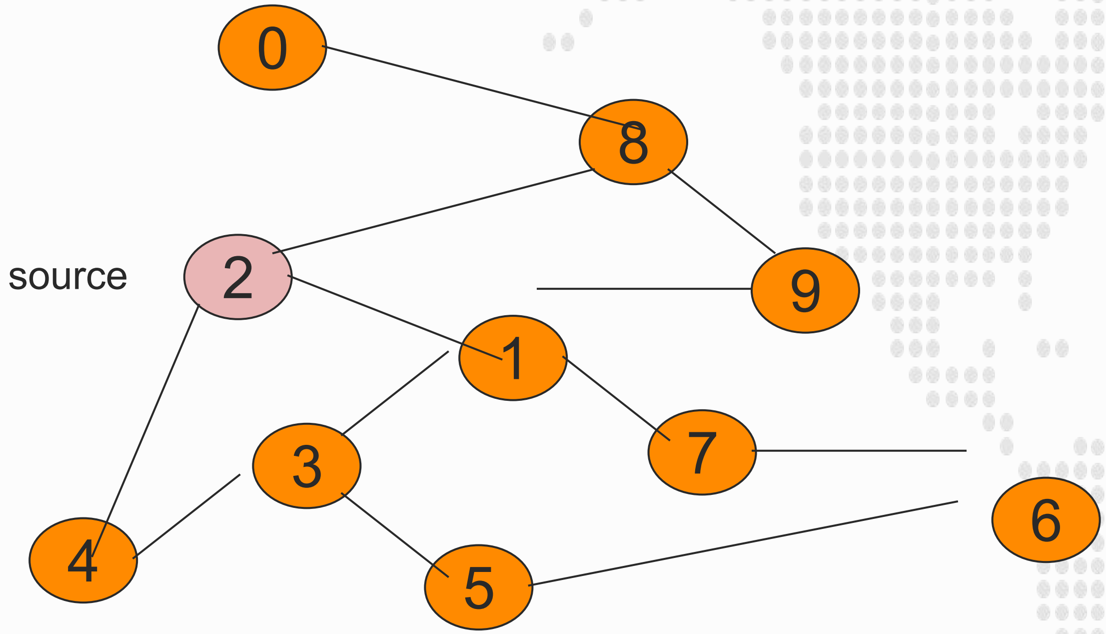
- Find an unvisited vertex that is adjacent to the current vertex, mark it as visited, and add to a queue.
- If an unvisited, adjacent vertex can't be found, remove a vertex from the queue (as long as there is a vertex to remove), make it the current vertex, and start over.
- If the second step can't be performed because the queue is empty, the algorithm is finished.

BFS



Breath-First Search.

Example



Q = { }

Initialize Q to be empty

Adjacency List

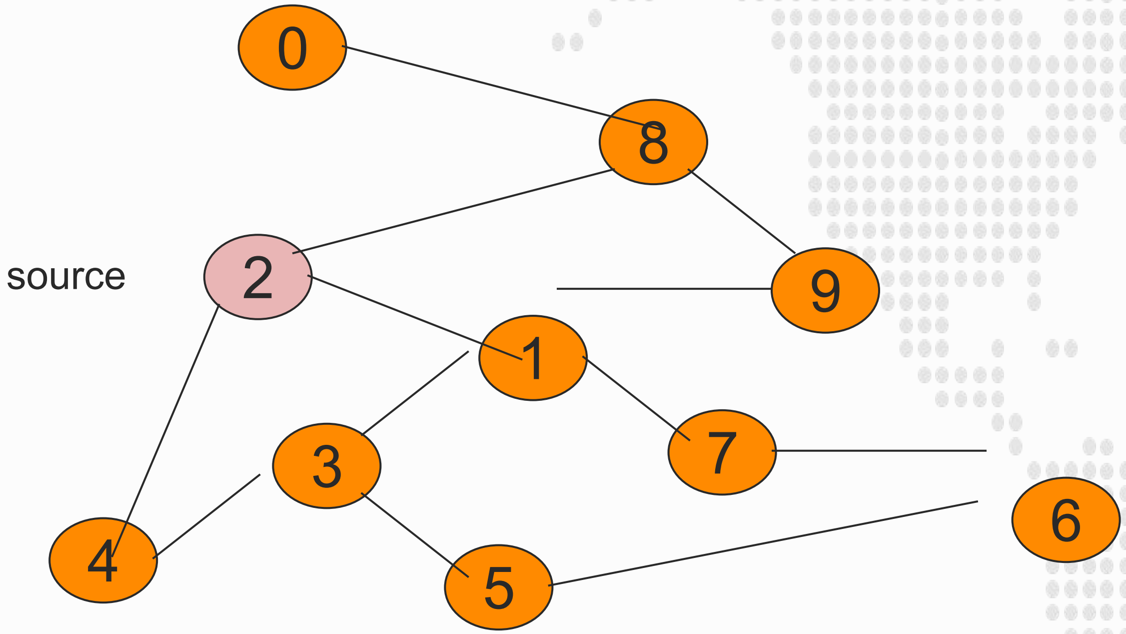
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

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

Initialize visited table (all False)

Example



Q = { 2 }

Place source 2 on the queue.

Adjacency List

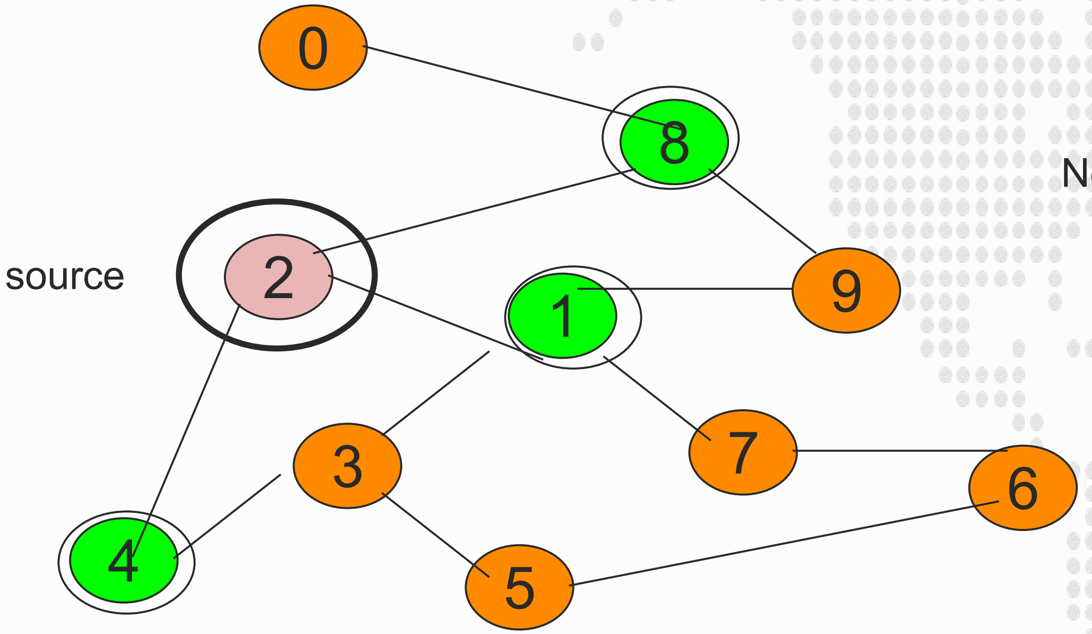
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

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.

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

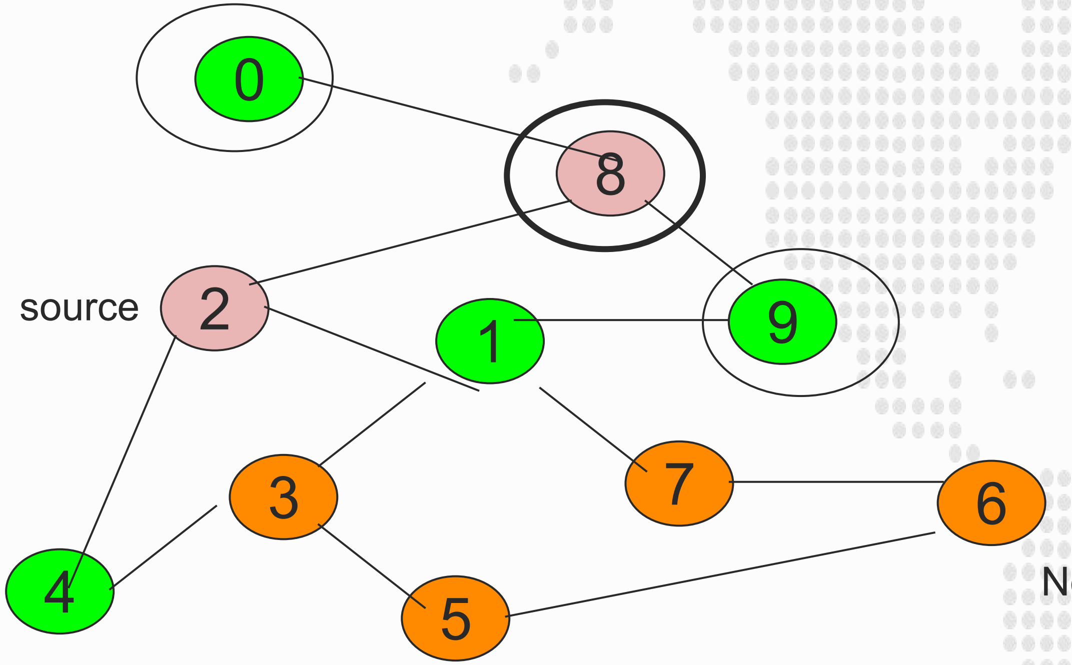
Q = {2} → { 8, 1, 4 }

Dequeue 2.

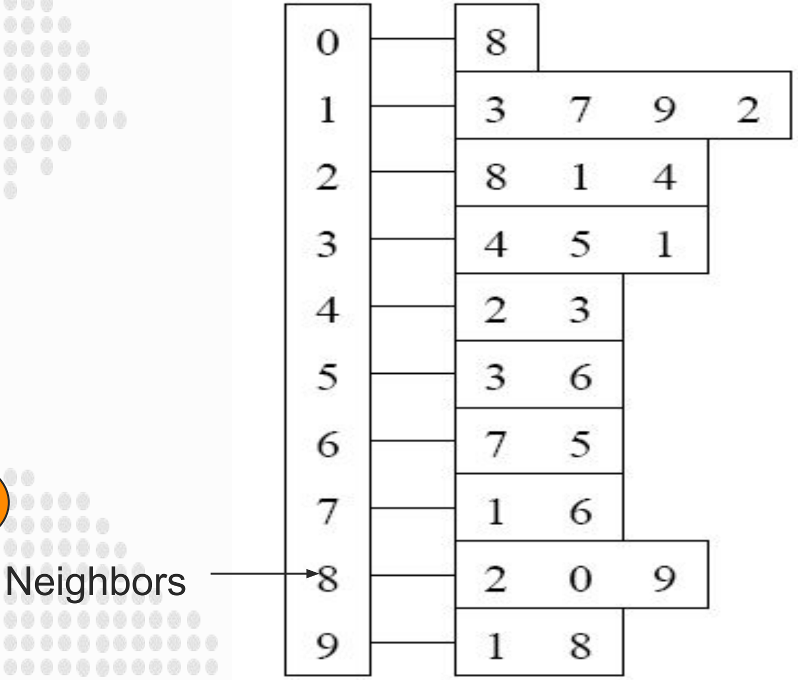
Place all unvisited neighbors of 2 on the queue

Mark neighbors as visited.

Example



Adjacency List



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.

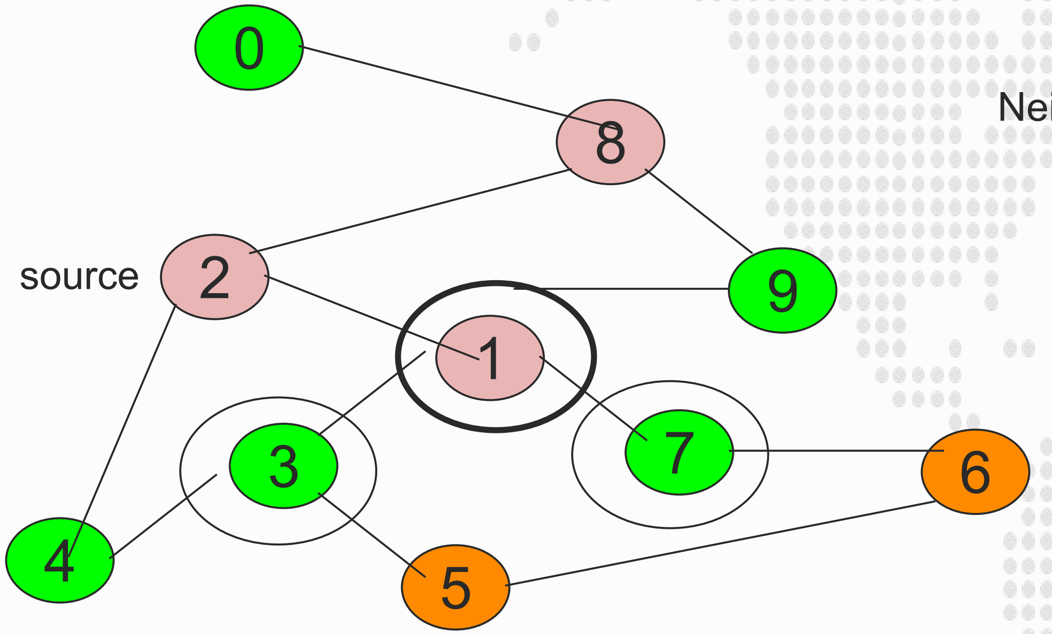
Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

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!

Example



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

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

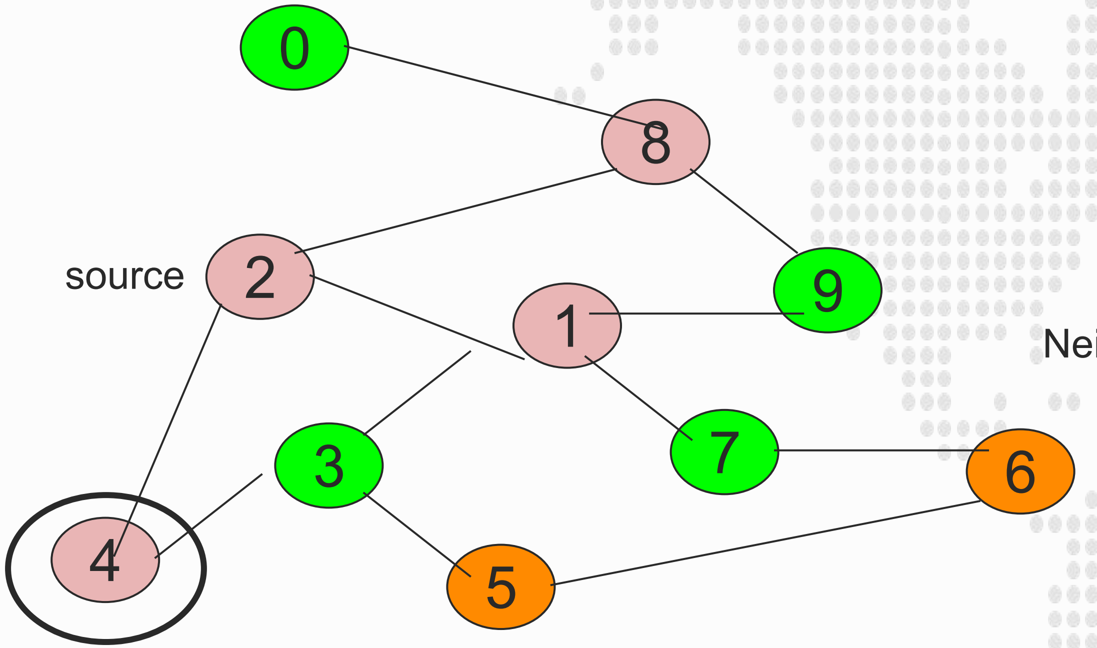
Dequeue 1.

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

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

Mark new visited Neighbors.

Example



Adjacency List

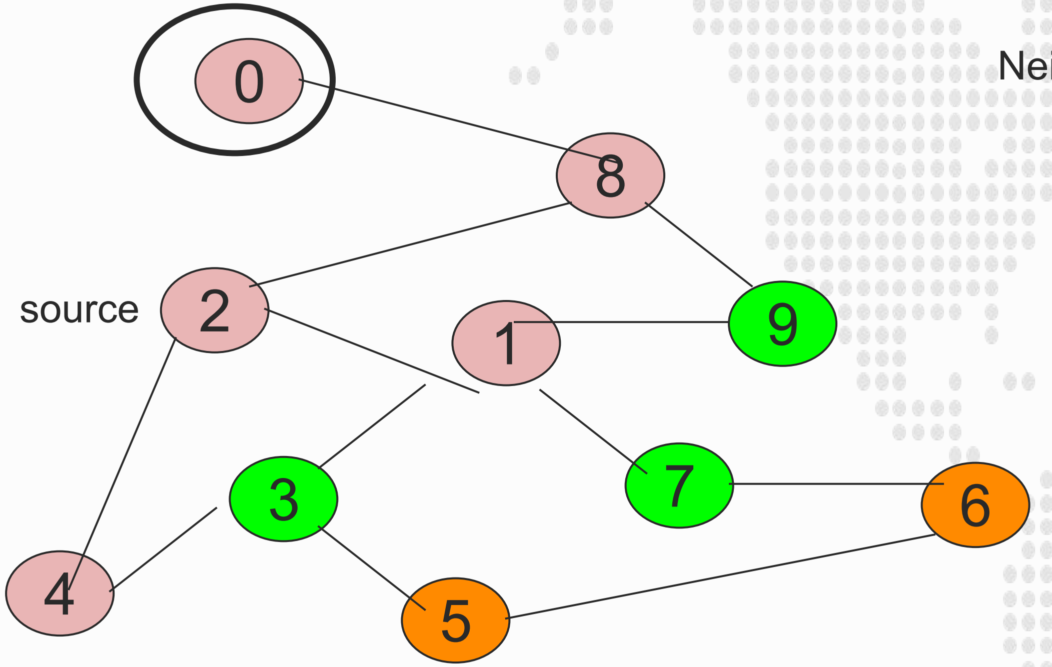
Visited Table (T/F)

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

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

Dequeue 4.
-- 4 has no unvisited neighbors!

Example



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)

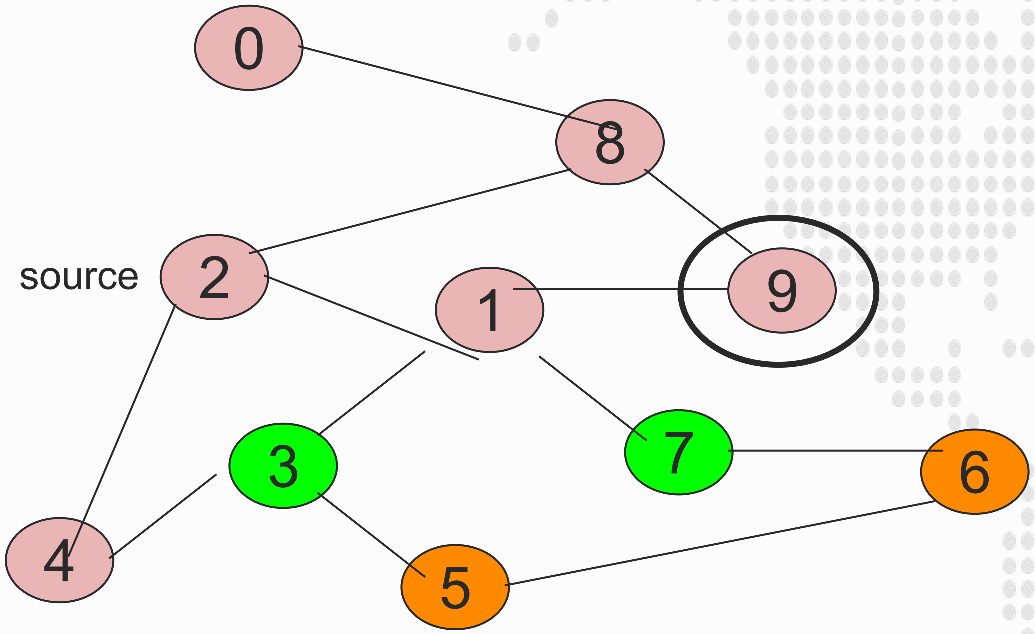
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!

Example

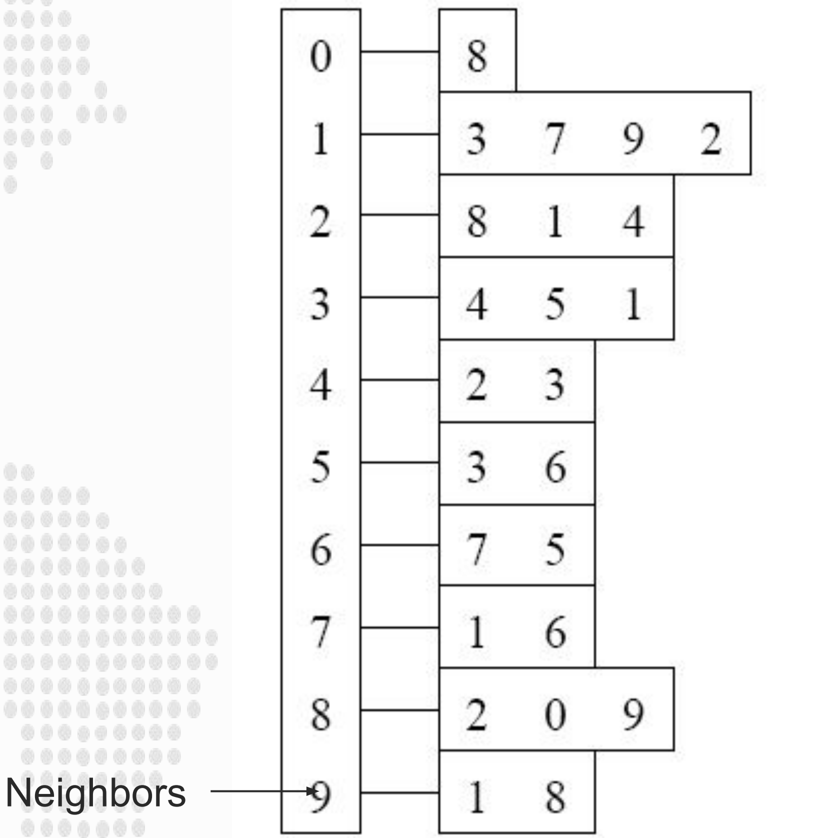


Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.

-- 9 has no unvisited neighbors!

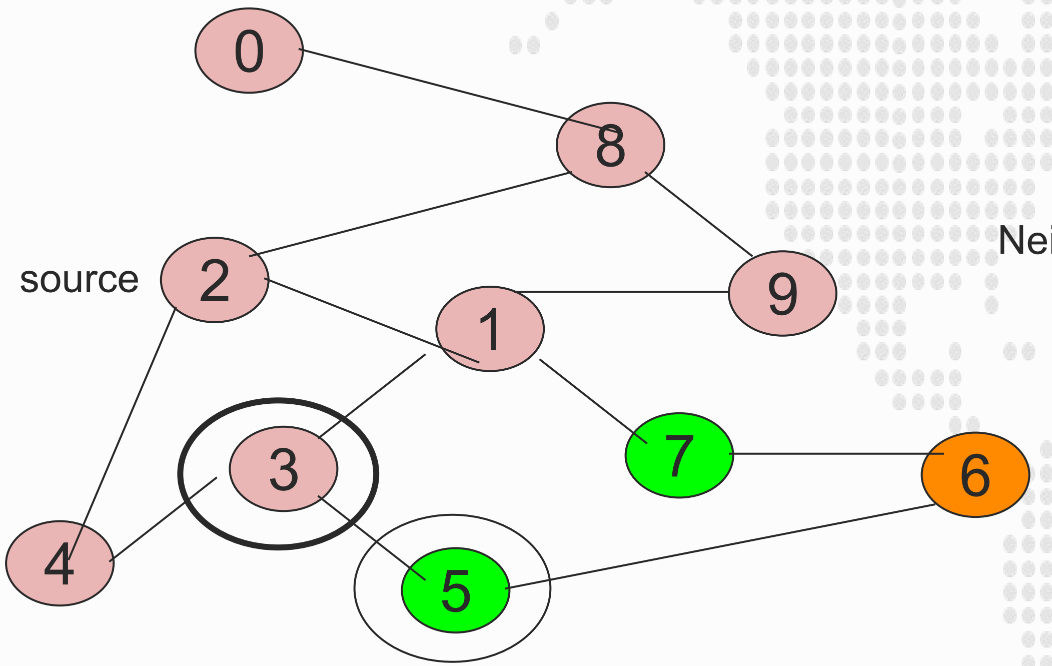
Adjacency List



Visited Table (T/F)

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

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)

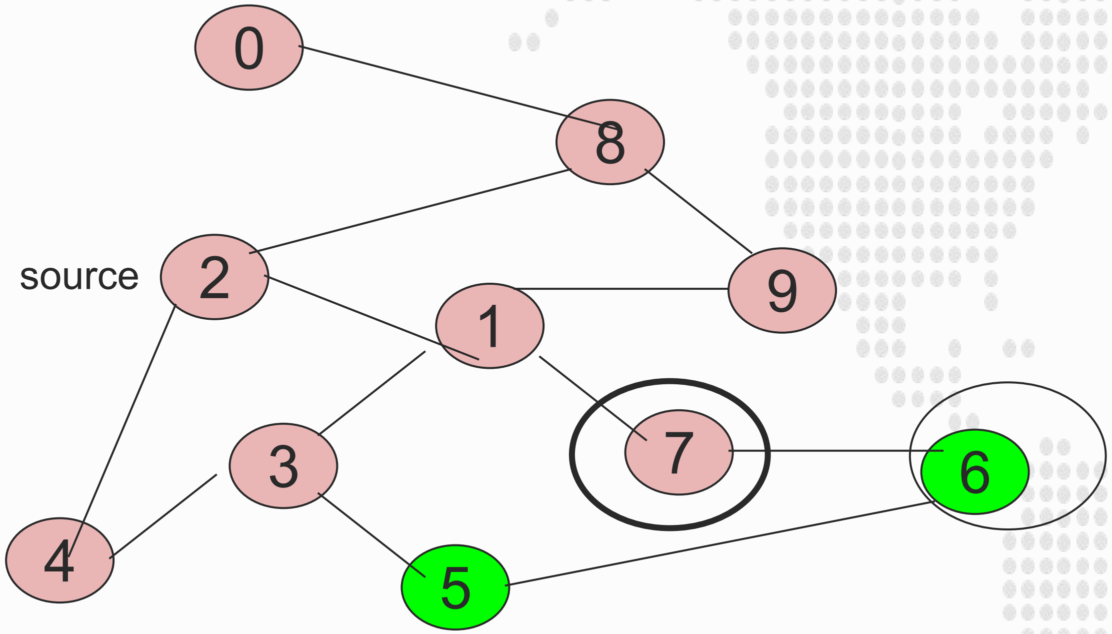
0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.

Mark new visited
Vertex 5.

Example



Q = { 7, 5 } → { 5, 6 }

Dequeue 7.

-- place neighbor 6 on the queue.

Adjacency List

Neighbors → 7

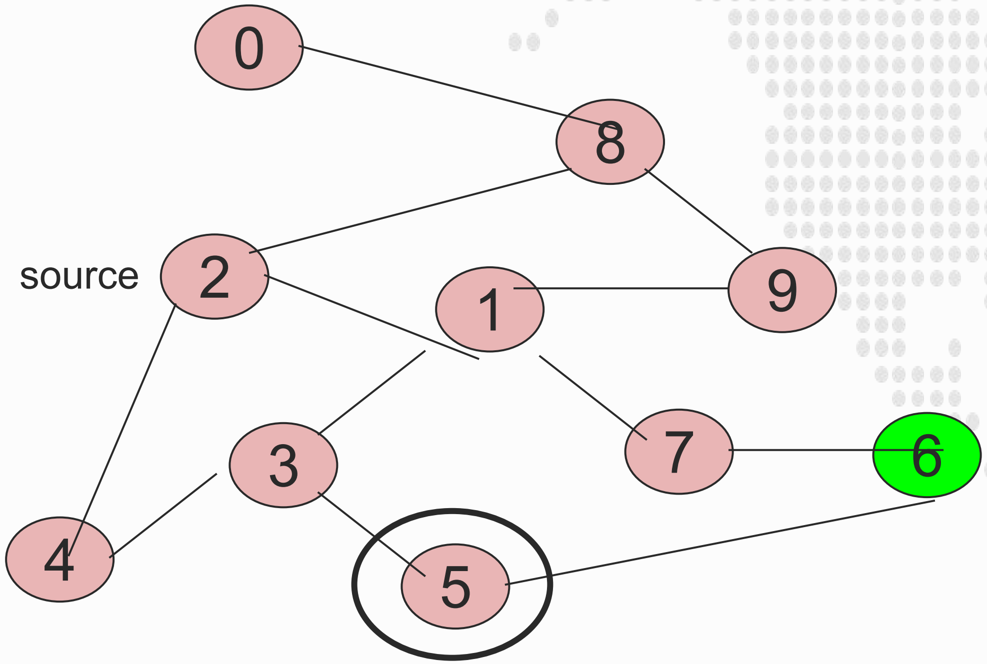
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

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.

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)

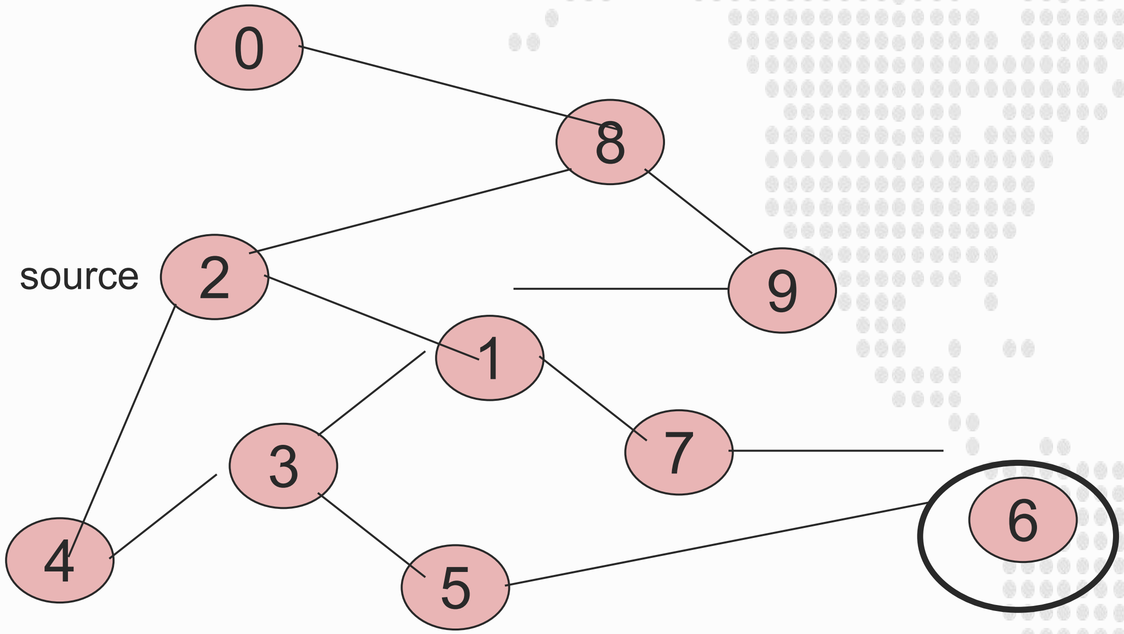
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

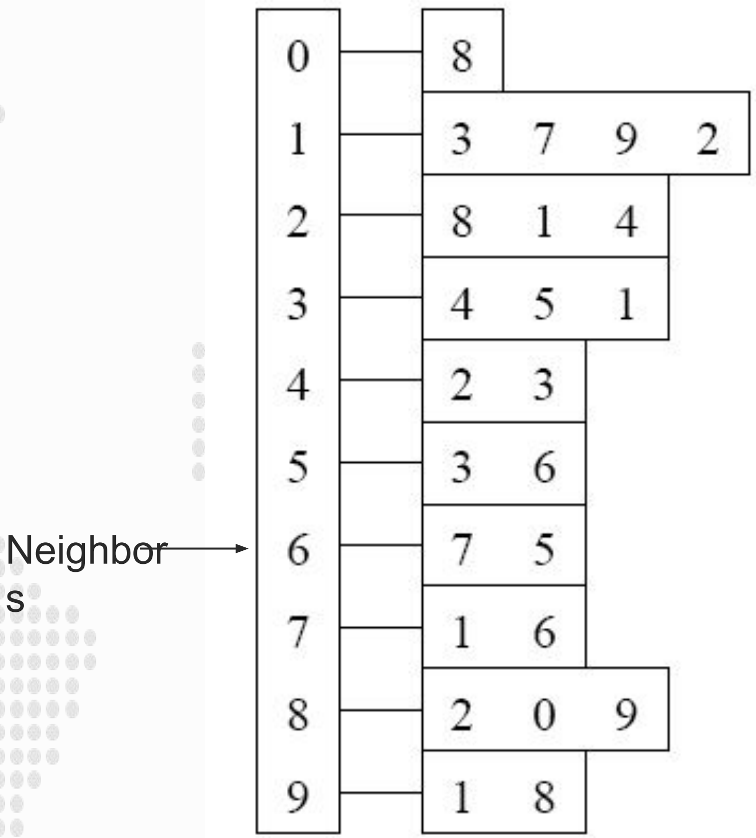
Dequeue 5.

-- no unvisited neighbors of 5.

Example



Adjacency List



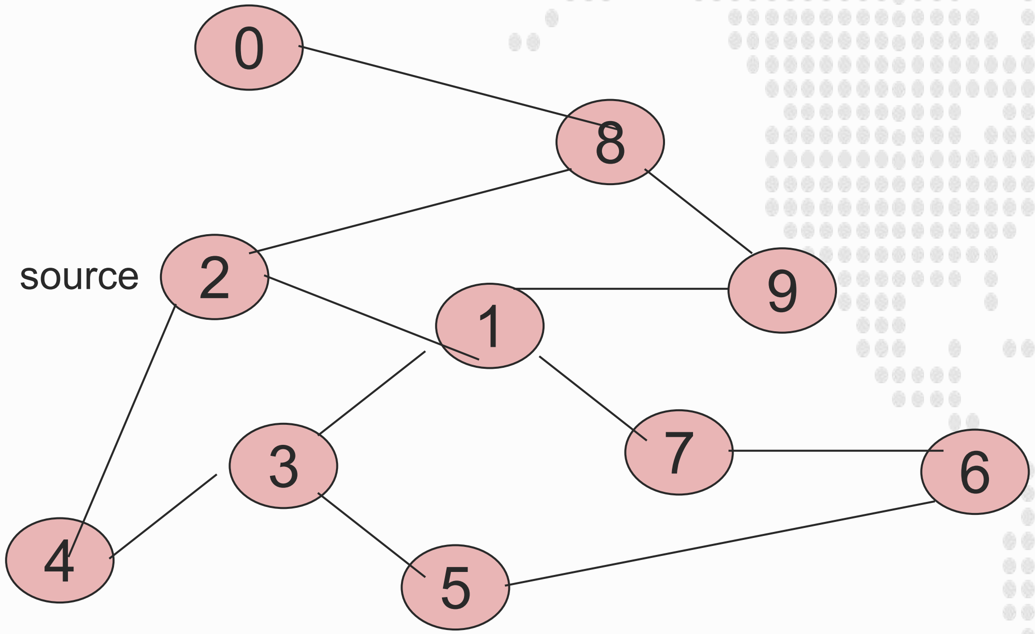
Visited Table (T/F)

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

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

Dequeue 6.
-- no unvisited neighbors of 6.

Example



Q = { }   STOP!!!   Q is 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	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

What did we discover?

Look at “visited” tables.

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

Shortest Path Recording

BFS we saw only tells us whether a path exists from source s, to other vertices v.

- It doesn't tell us the path!
- We need to modify the algorithm to record the path.

How can we do that?

Note: we do not know which vertices lie on this path until we reach v!

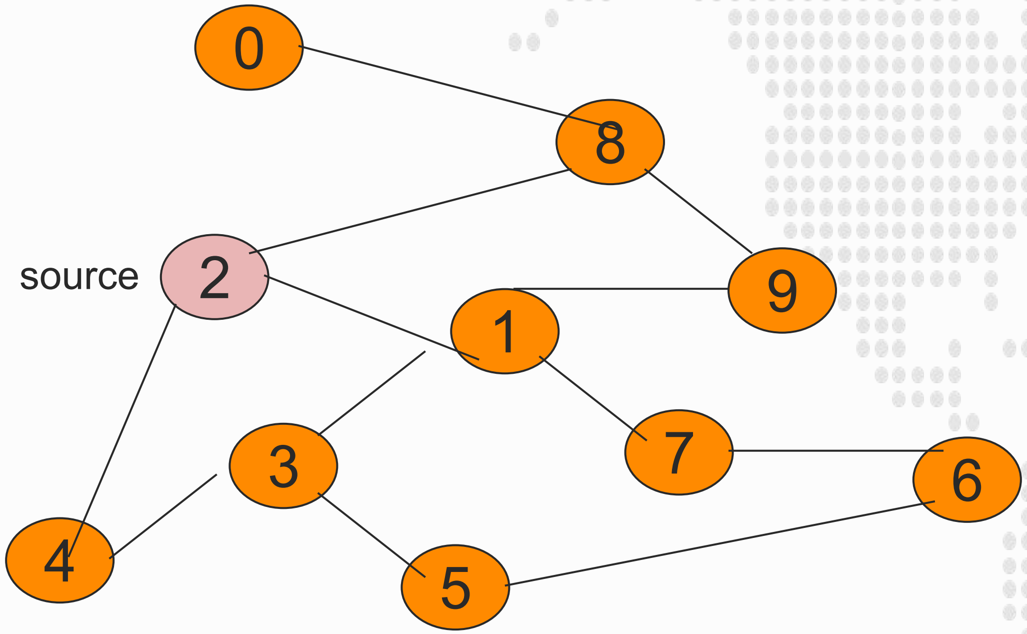
Efficient solution:

- Use an additional array *pred[0..n-1]*
- *Pred[w] = v means that vertex w was visited from v*

Steps to Modify BFS for Shortest Path

- **Track Distances**: Maintain an array or map to keep track of the shortest distance from the source to each node.
- **Track Predecessors**: Keep an array or map to store the predecessor (or parent) of each node in the shortest path. This will help in reconstructing the path at the end.
- **Perform BFS**: Start from the source node, explore neighbors level by level, and update distances and predecessors as you go.

Example



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		

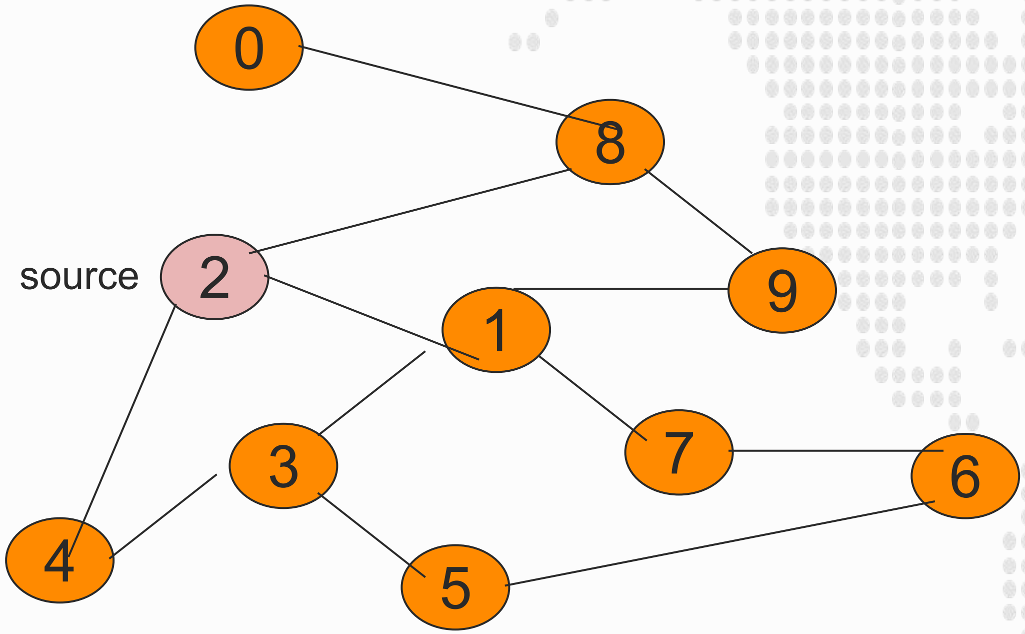
Visited Table (T/F)

		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

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)

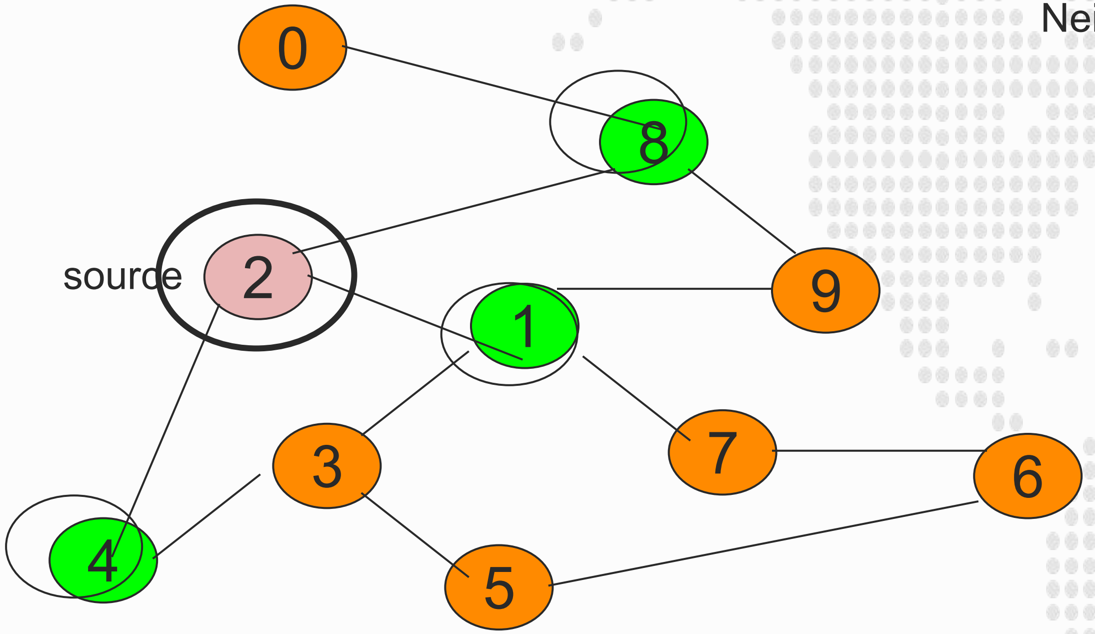
		Pred
0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Q = { 2 }

Place source 2 on the queue.

Flag that 2 has been visited.

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)

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

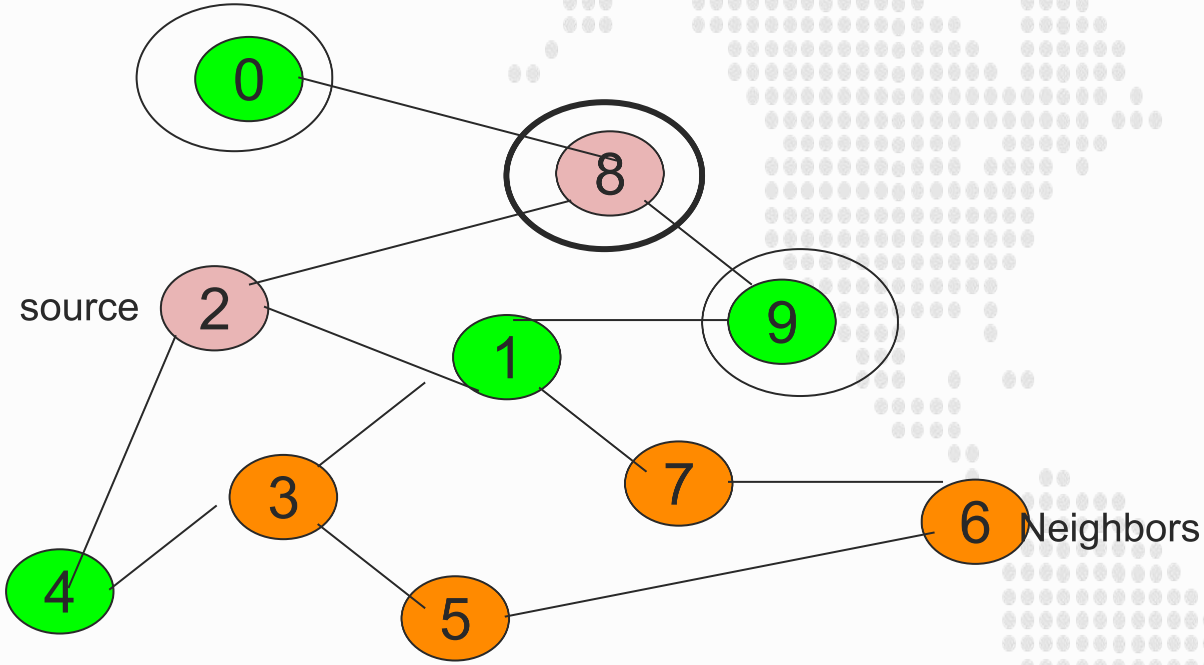
Mark neighbors as visited.

Record in Pred that we came from 2.

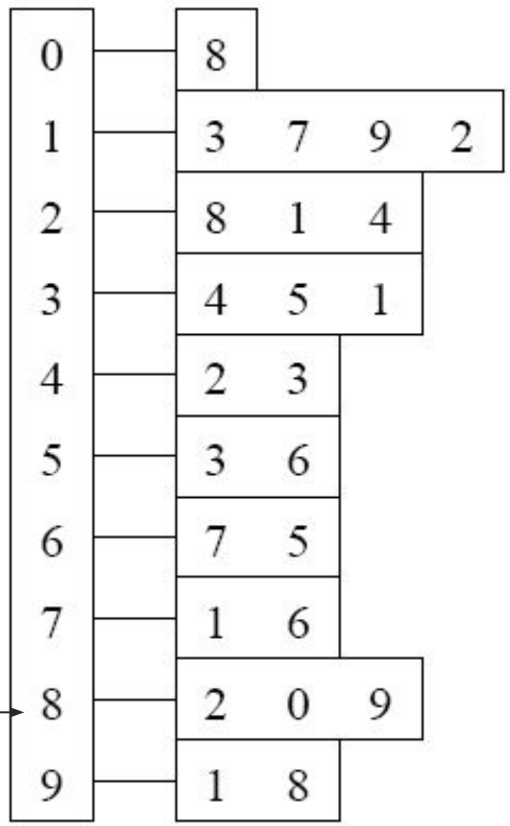
Q = {2} → { 8, 1, 4 }

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

Example



Adjacency List



Visited Table (T/F)

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

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.

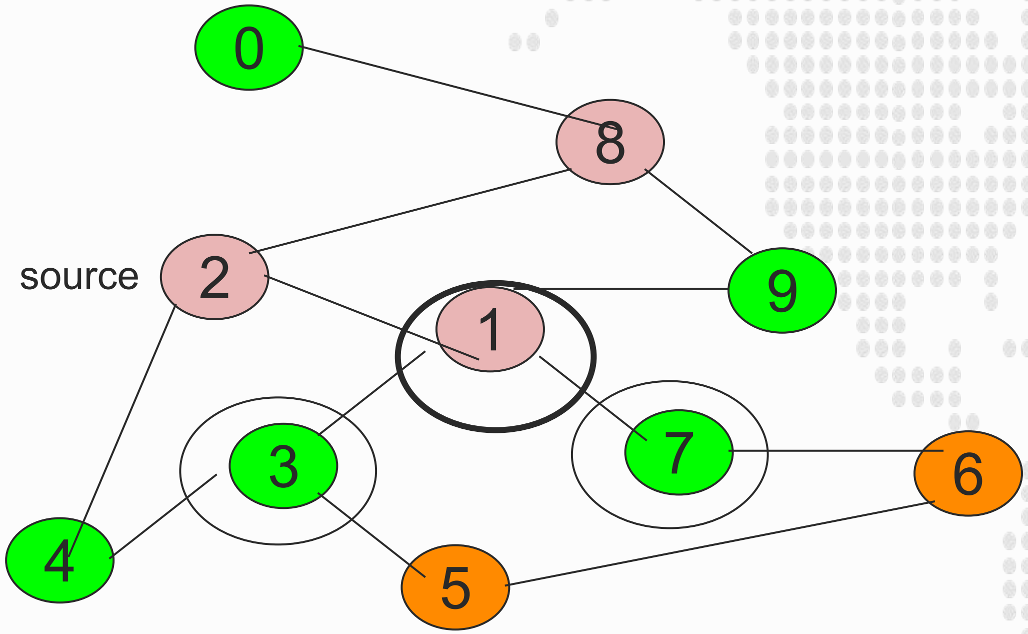
-- Place all unvisited neighbors of 8 on the queue.

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

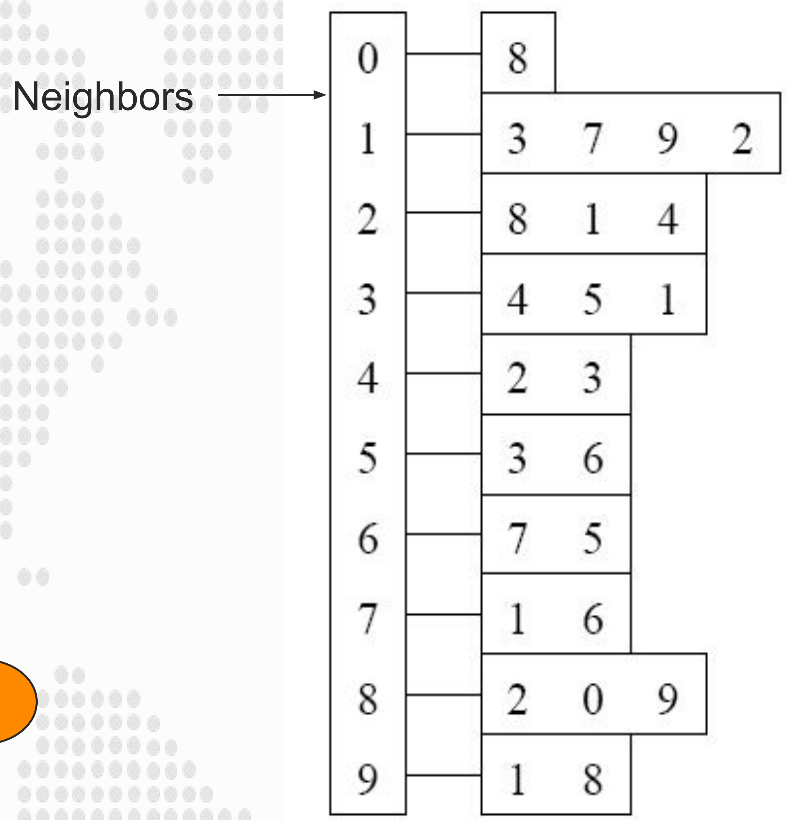
Mark new visited Neighbors.

Record in Pred that we came from 8.

Example



Adjacency List



Visited Table (T/F)

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

Mark new visited Neighbors.

Record in Pred that we came from 1.

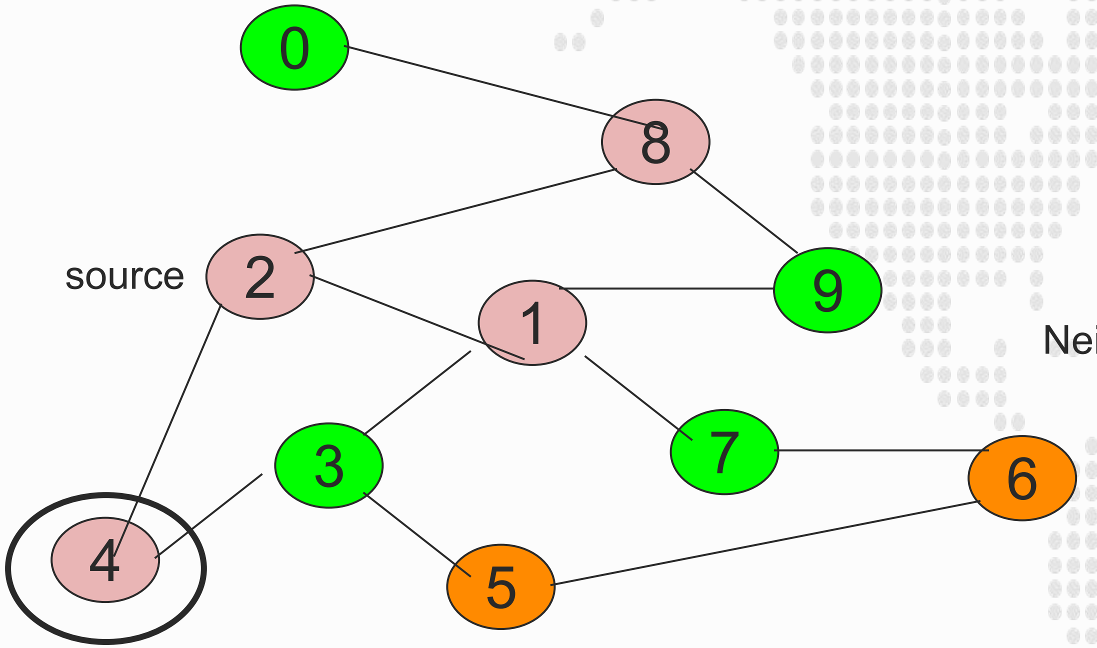
Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.

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

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

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

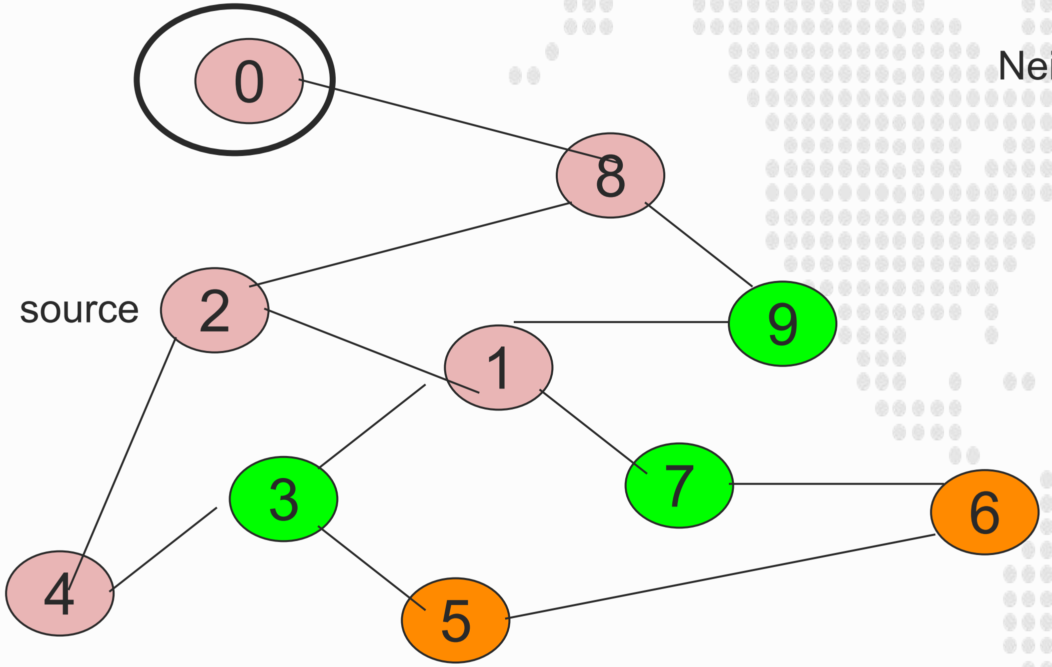
Visited Table (T/F)

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

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

Dequeue 4.
-- 4 has no unvisited neighbors!

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)

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

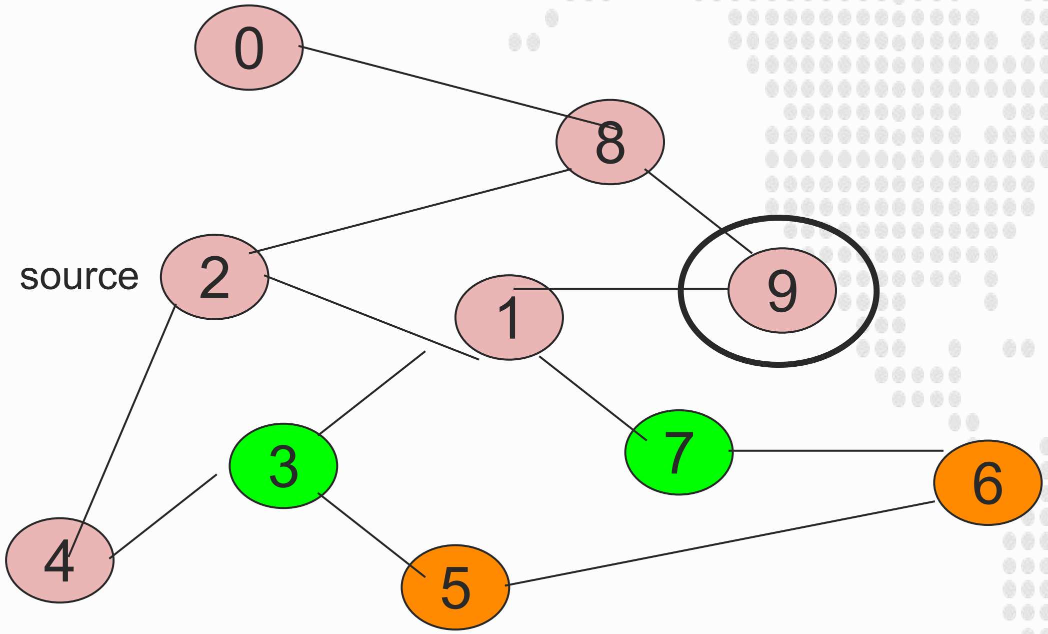
Pred

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!

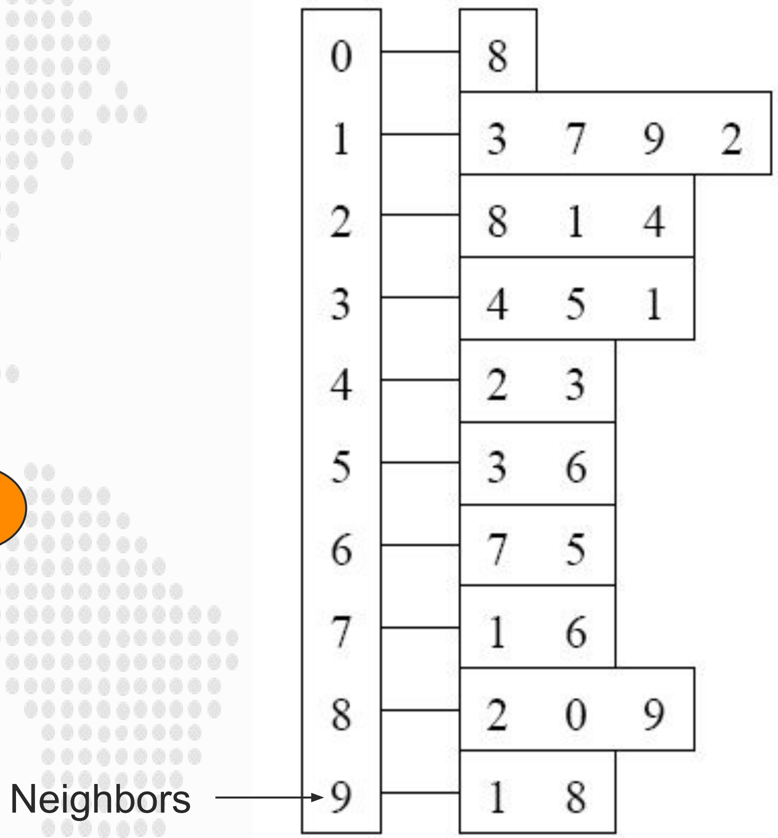
Example



Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 has no unvisited neighbors!

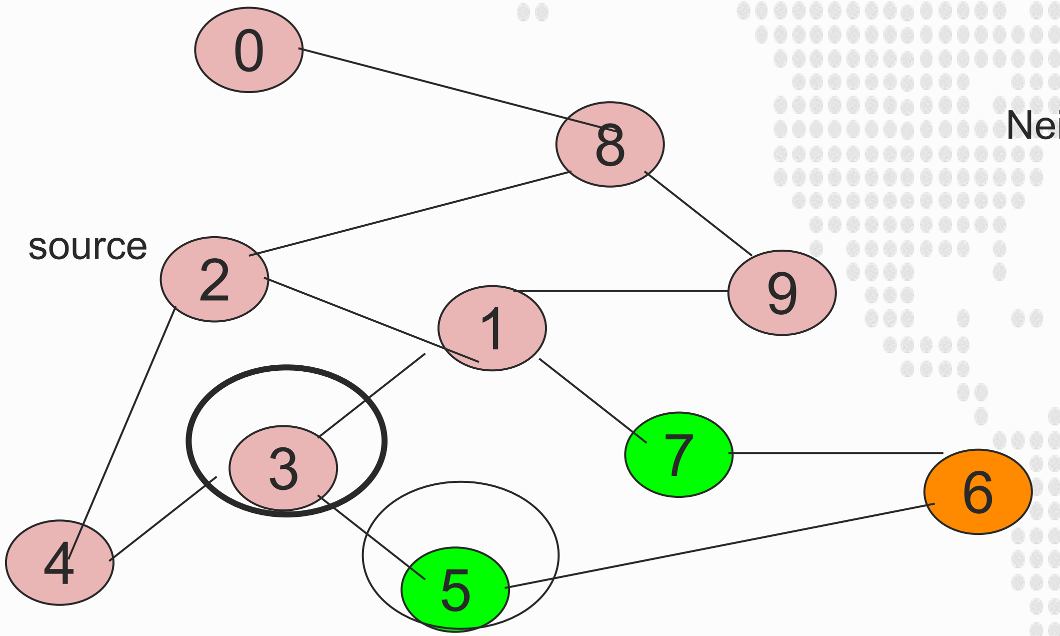
Adjacency List



Visited Table (T/F)

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

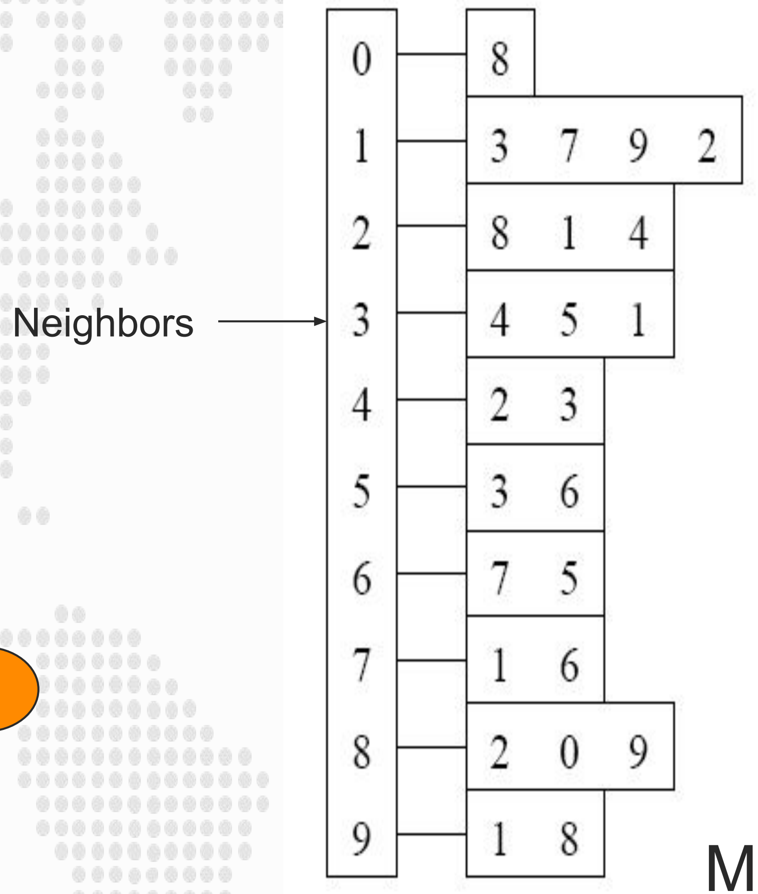
Example



Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.

Adjacency List



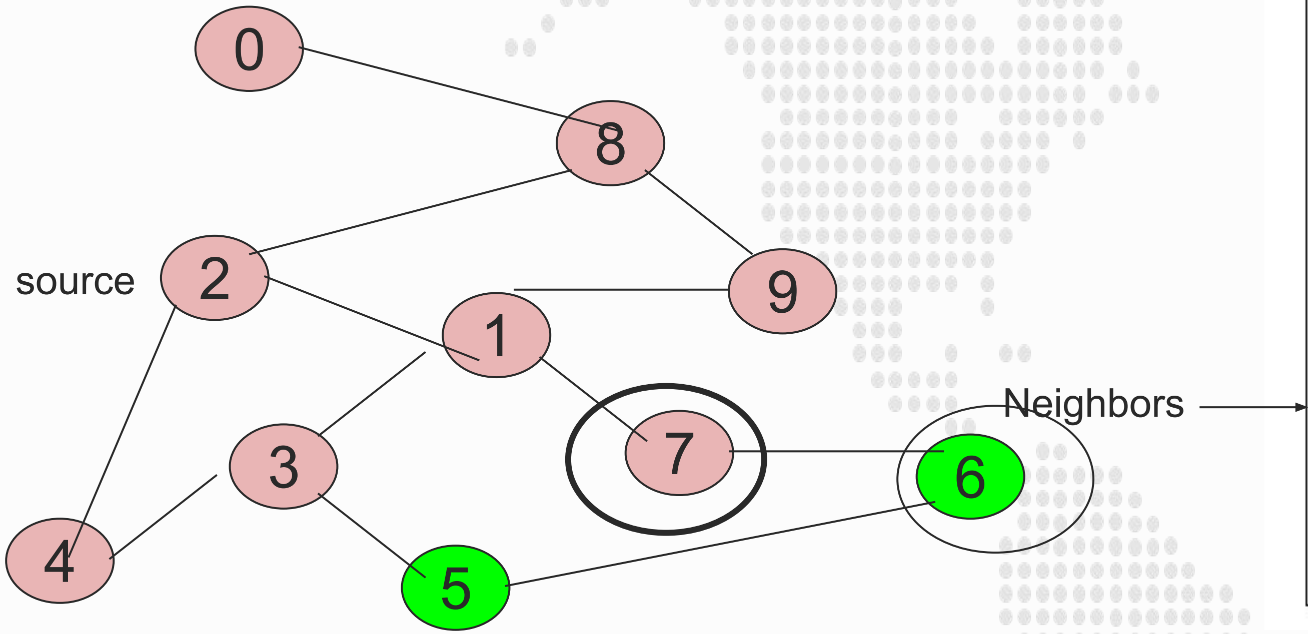
Visited Table (T/F)

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

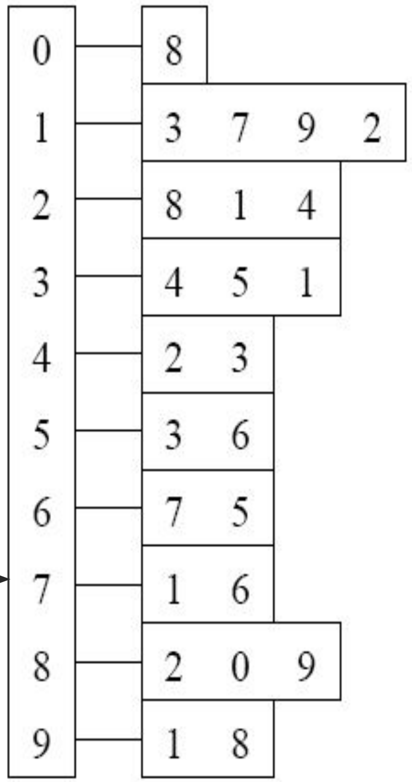
Mark new visited Vertex 5.

Record in Pred that we came from 3

Example



Adjacency List



Visited Table (T/F)

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

Pred

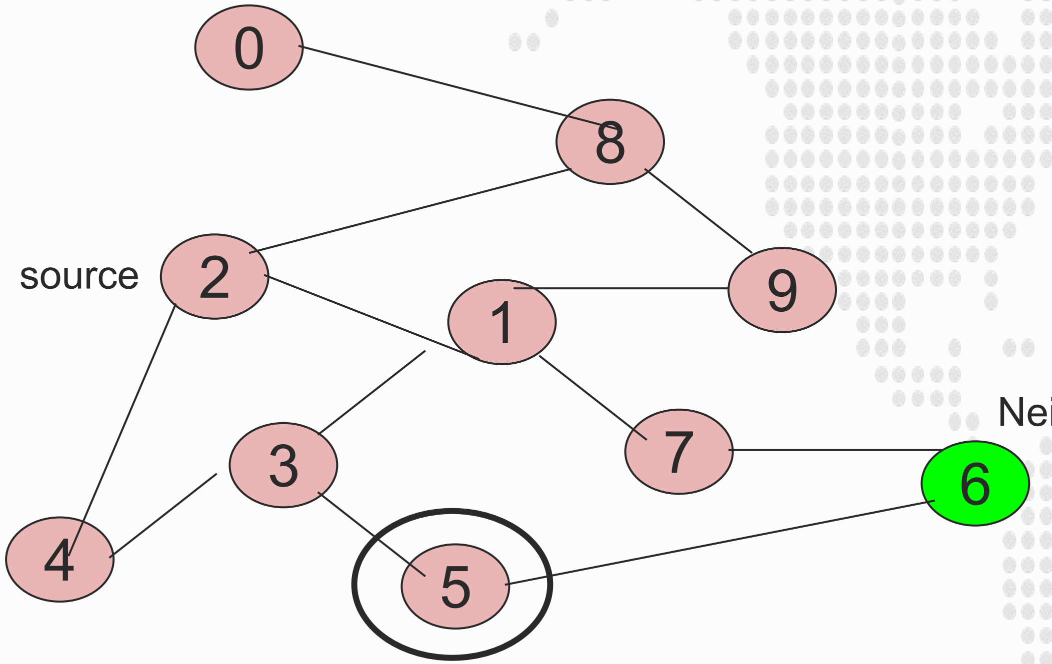
Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue.

Mark new visited
Vertex 6.

Record in Pred
that we came
from 7.

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)

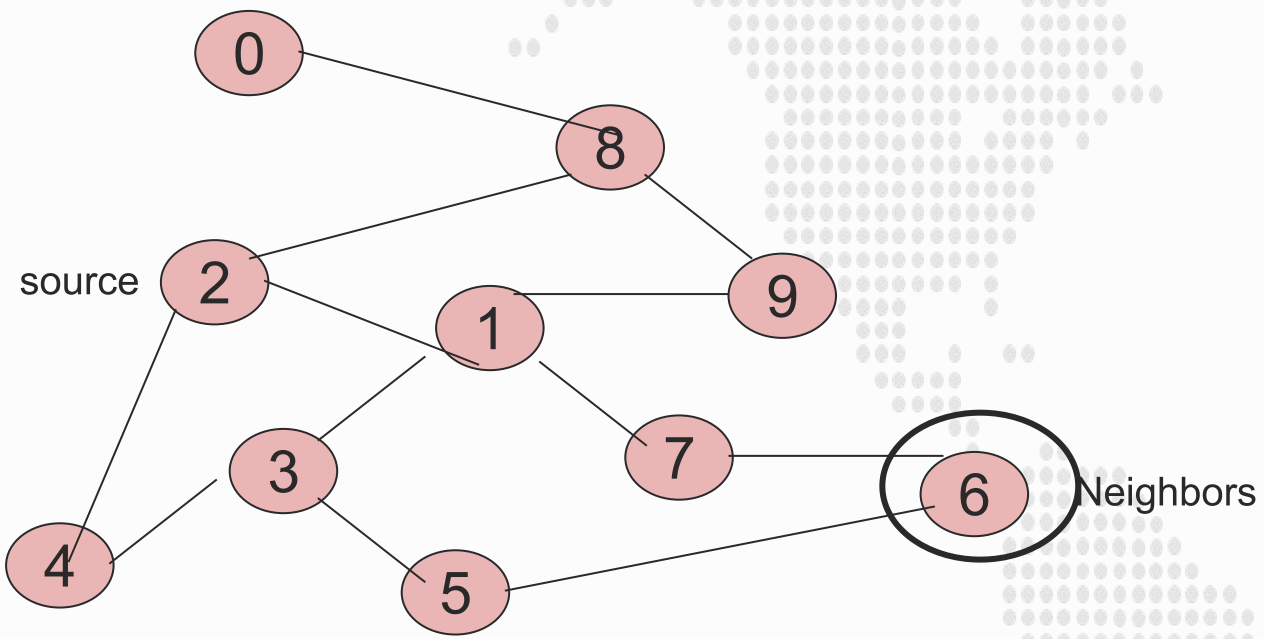
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Q = { 5, 6} → { 6 }

Dequeue 5.

-- no unvisited neighbors of 5.

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)

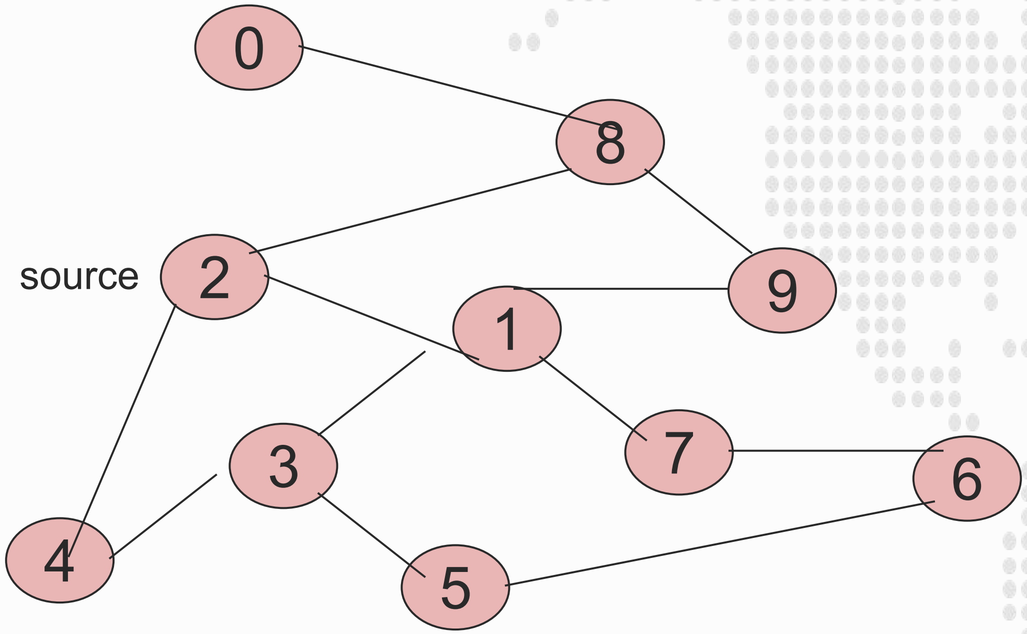
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

Q = { 6 } → { }

Dequeue 6.
-- no unvisited neighbors of 6.

Example



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

Adjacency List

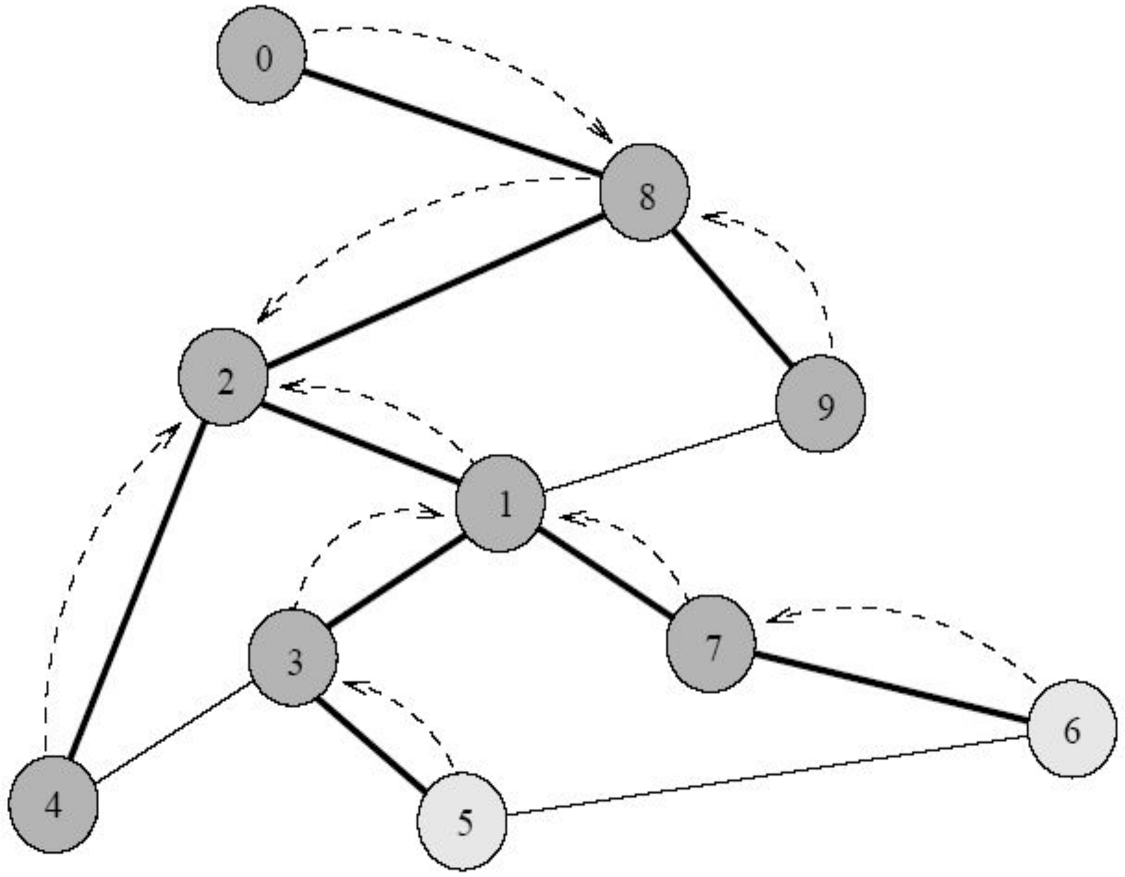
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

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

Pred now can be traced backward to report the path!

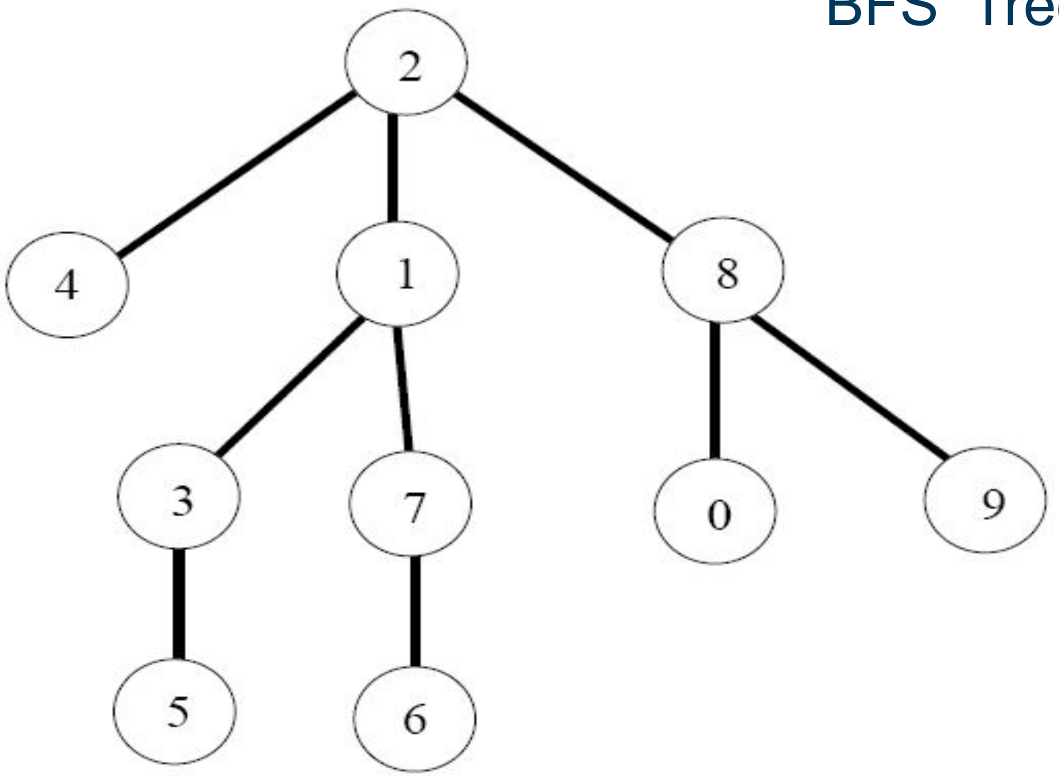
Path reporting



Nodes	Visited from/ Parent Nodes
0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

BFS Tree

- The paths found by BFS is often drawn as a rooted tree (called BFS tree), with the starting vertex as the root of the tree.



BFS Tree for Vertex s=2.

Weighted Graph

Weighted Graph

- A **weighted graph** is a graph in which each edge has an associated weight or cost.
- These weights could represent various values depending on the context, such as distances, costs, or times.
- Weighted graphs can be either **directed** or **undirected**.

Example

Consider a graph with four vertices A, B, C, and D, representing cities, and edges representing roads between them. The weights on the edges represent distances between the cities:

1. A to B has a weight of 5 (distance = 5 km).
2. A to C has a weight of 3.
3. B to D has a weight of 2.
4. C to D has a weight of 7.

```
         (5)
  A ----------> B
  |             |
(3)|            |(2)
  |             |
  V             V
  C  ---------> D
         (7)
```

Representation of Weighted Graph

2. Adjacency list Representation

An **adjacency list** for a weighted graph is an array of lists. Each list contains pairs representing the adjacent vertices and the weights of the edges to them.

- A -> B|5, C|3
- B -> D|2
- C -> D|7
- D -> (no outgoing edges)

Dijekstra’s Algorithms

Dijkstra's Algorithm is a **greedy algorithm** used to find the **shortest path** from a source vertex to all other vertices in a weighted graph. It guarantees the shortest path for graphs with **non-negative weights**.

Key Concepts

1. **Weighted Graph**: A graph where edges have weights representing costs, distances, or other measures.
2. **Shortest Path**: The path between two vertices that has the smallest sum of edge weights.
3. **Greedy Approach**: The algorithm selects the shortest path to expand at each step.

Algorithms Steps

Initialize:

- Create a distance array, initialized as infinity (∞\infty∞) for all vertices, except the source vertex (set it to 0).
- Mark all vertices as unvisited.
- Use a priority queue (or a similar structure) to always expand the vertex with the smallest tentative distance.

Iterate:

- Pick the unvisited vertex with the smallest distance.
- Update the distances to its neighbors if a shorter path is found through this vertex.
- Mark the current vertex as visited (processed).

Repeat:

- Continue until all vertices are visited or the shortest path to the target vertex is found.

Example with Step by Step Execution

Vertices: A, B, C, D, E, F

Edges (with weights):

A–B (4), A–C (2), B–C (1), B–D (5), C–D (8), C–E (10), D–E (2), D–F (6), E–F (3)

Step by Step Execution

1. Initialization:

   Start at vertex A.

   - Distance array: [A:0,B:∞,C:∞,D:∞,E:∞,F:∞]
   - Priority queue: [(A,0)]

2. Iteration 1 (Expand A):

- Current vertex: A (distance = 0).
- Neighbors of A: B (4), C (2).
- Update distances:
  - B:min(∞,0+4)=4
  - C:min(∞,0+2)=2
- Distance array: [A:0,B:4,C:2,D:∞,E:∞,F:∞]
- Priority queue: [(C,2),(B,4)]

Step by Step Execution

Iteration 2 (Expand C):

- Current vertex: C (distance = 2).
- Neighbors of C: B (1), D (8), E (10).
- Update distances:
  - B:min(4,2+1)=3
  - D:min(∞,2+8)=10
  - E:min(∞,2+10)=12
- Distance array: [A:0,B:3,C:2,D:10,E:12,F:∞]
- Priority queue: [(B,3),(B,4),(D,10),(E,12)]

Step by Step Execution

Iteration 3 (Expand B):

- Current vertex: B (distance = 3).
- Neighbors of B: D (5).
- Update distances:
    i. D:min(10,3+5)=8
- Distance array: [A:0,B:3,C:2,D:8,E:12,F:∞]
- Priority queue: [(D,8),(E,12),(D,10)]

Step by Step Execution

- Iteration 4 (Expand D):
  - Current vertex: D (distance = 8).
  - Neighbors of D: E (2), F (6).
  - Update distances:
    1. E:min(12,8+2)=10
    2. F:min(∞,8+6)=14
  - Distance array: [A:0,B:3,C:2,D:8,E:10,F:14]
  - Priority queue: [(E,10),(F,14)]

Step by Step Execution

Iteration 5 (Expand E):

- Current vertex: E (distance = 10).
- Neighbors of E: F (3).
- Update distances:
  - F:min(14,10+3)=13
  - Distance array: [A:0,B:3,C:2,D:8,E:10,F:13]
- Priority queue: [(F,13)][(F, 13)][(F,13)]

Step by Step Execution

- Iteration 6 (Expand F):
  - Current vertex: F (distance = 13).
  - All neighbors already processed.

Final Distance Array:
[A:0,B:3,C:2,D:8,E:10,F:13]

Shortest Path

Shortest Paths from A:

- To B: 3
- To C: 2
- To D: 8
- To E: 10
- To F: 13

Applications of Dijekstra’s Algorithm

1. **Navigation Systems**:
   - Google Maps and GPS use Dijkstra’s algorithm to find the shortest path.
2. **Network Routing**:
   - Used in protocols like OSPF (Open Shortest Path First).
3. **Scheduling**:
   - Optimizing tasks in resource-constrained environments.
4. **Game Development**:
   - AI pathfinding in grid-based games.