Graph Data Structures

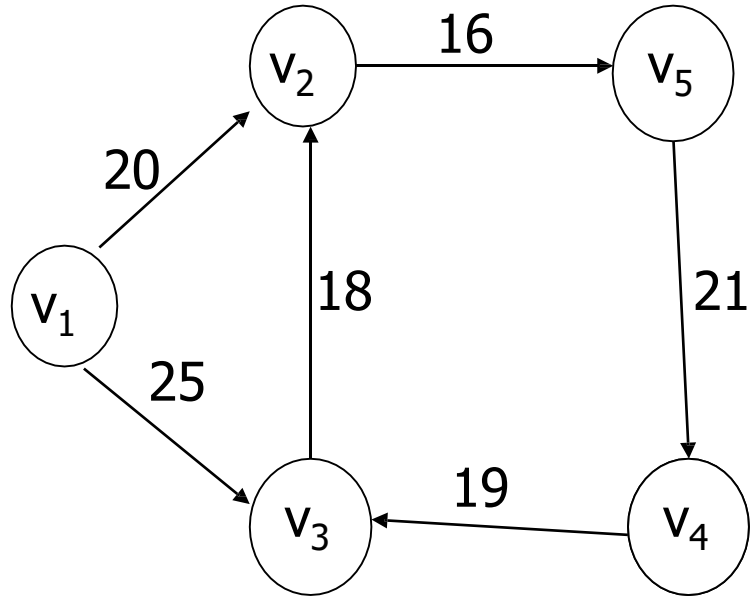
PROF. NAVRATI SAXENA

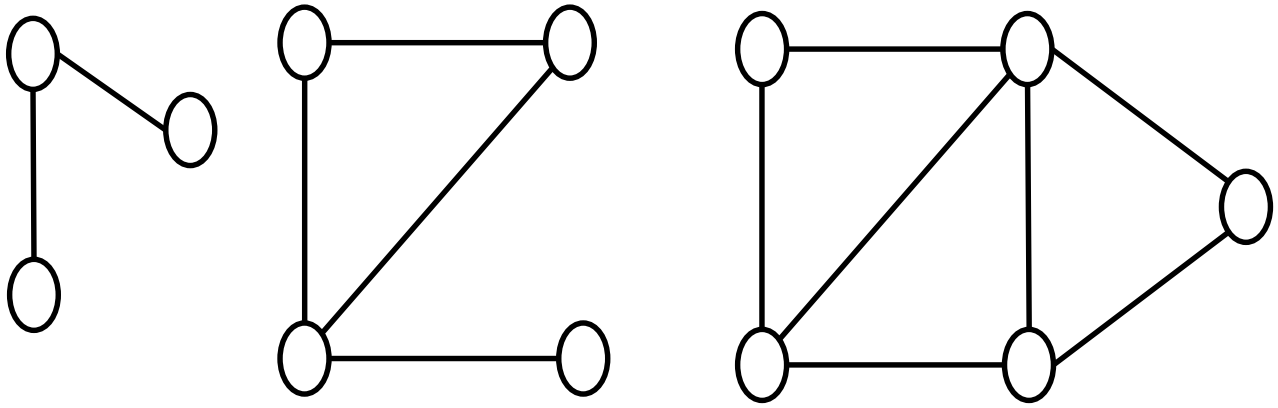
Graphs: Definition

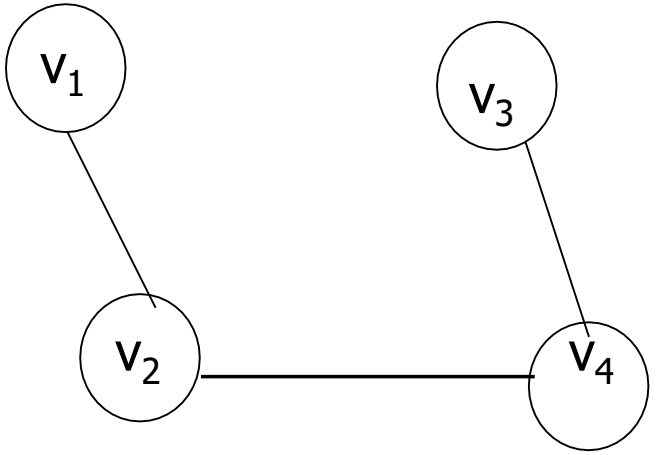
- A mathematical non-linear data structure
- Capable of representing many kinds of physical structures
- A graph G(V, E) is defined as a set of vertices (V) and a set of edges (E)
  - Vertices: Collection of nodes, represented as points or circles
  - Edges: Connect a pair of vertices and can have weights attached

Graphs: Examples

- Most un-restricted form of data organization
- Final destination for problem solving
- Many variances (Many Types of Graphs)
  - Directed and un-directed graphs
  - Connected and un-connected graphs
  - Weighted graphs

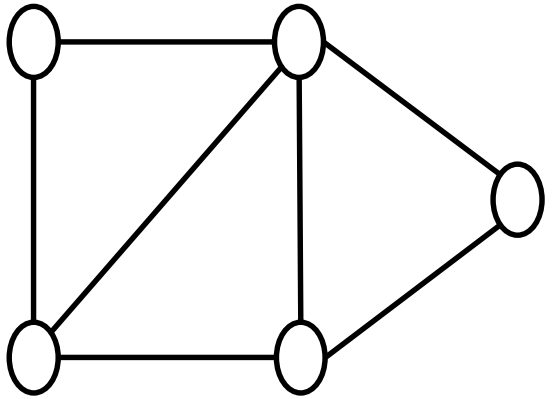


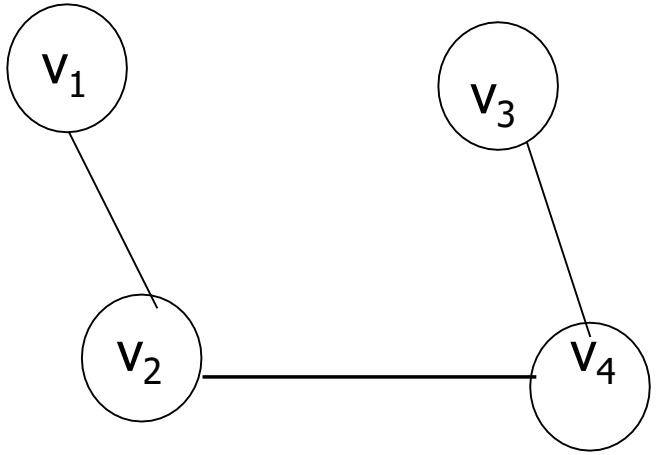




Types of Graph

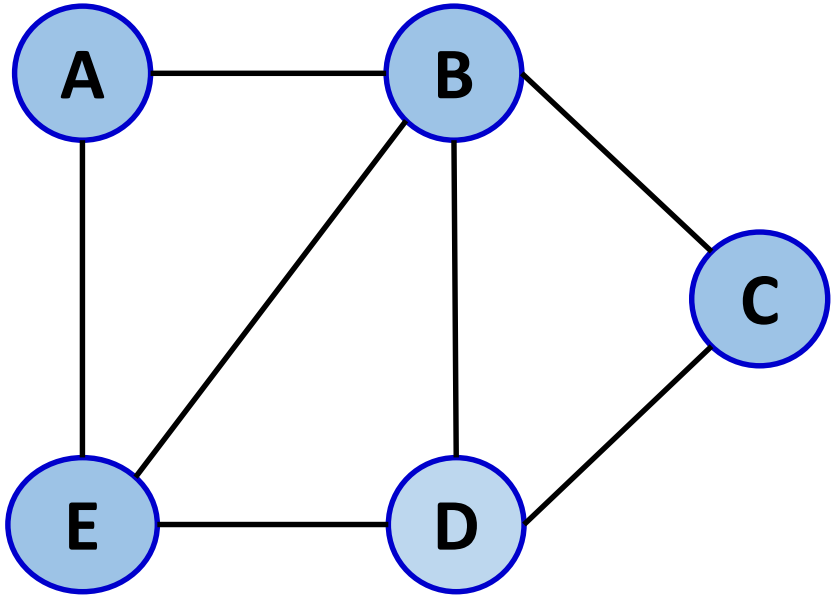
- Undirected graph
- Directed graph
- Weighted graph
- Connected graph
- Non-connected graph

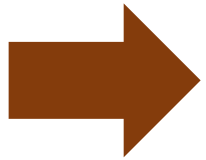




Adjacency Matrix Representation

1. Create a Matrix (e.g. use a 2-dimensional array)
2. Any index *i* represents a node
3. Any entry *(i, j)* in the matrix represents connectivity between two nodes *i, j*
    1. *entry (i, j) = 1 => an edge exists*
    2. *entry (i, j) = 0 => no edge exists*

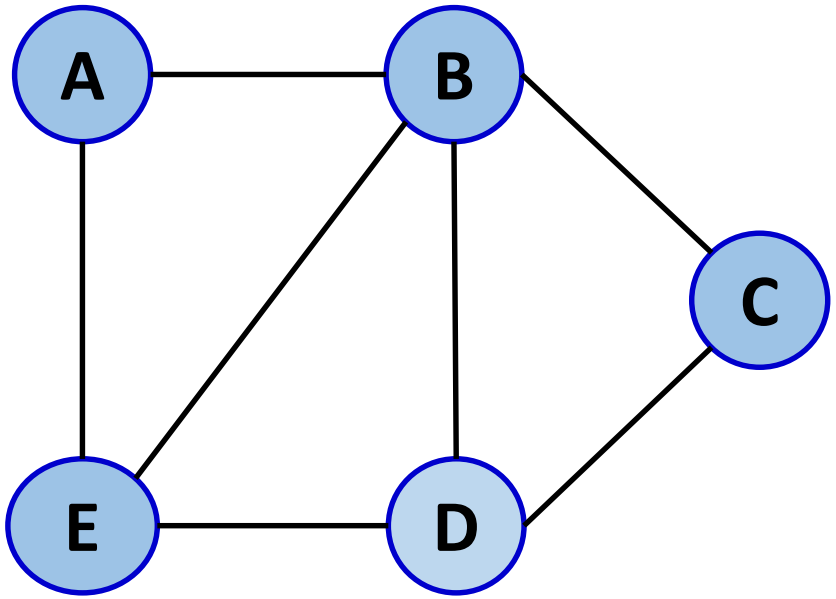


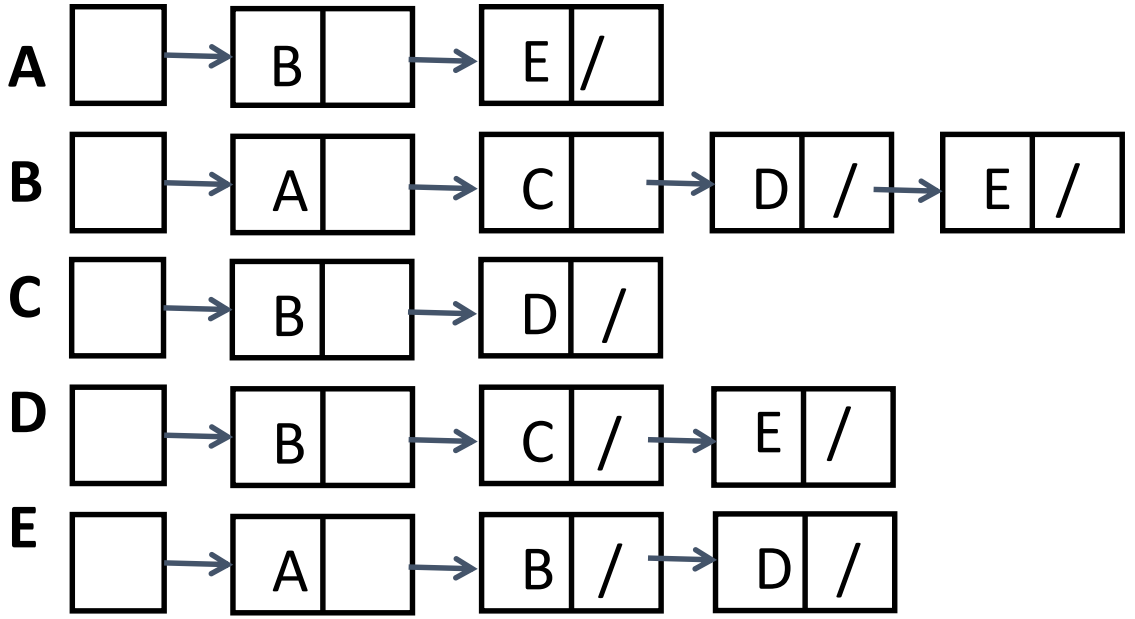


	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	1	1
C	0	1	0	1	0
D	0	1	1	0	1
E	1	1	0	1	0

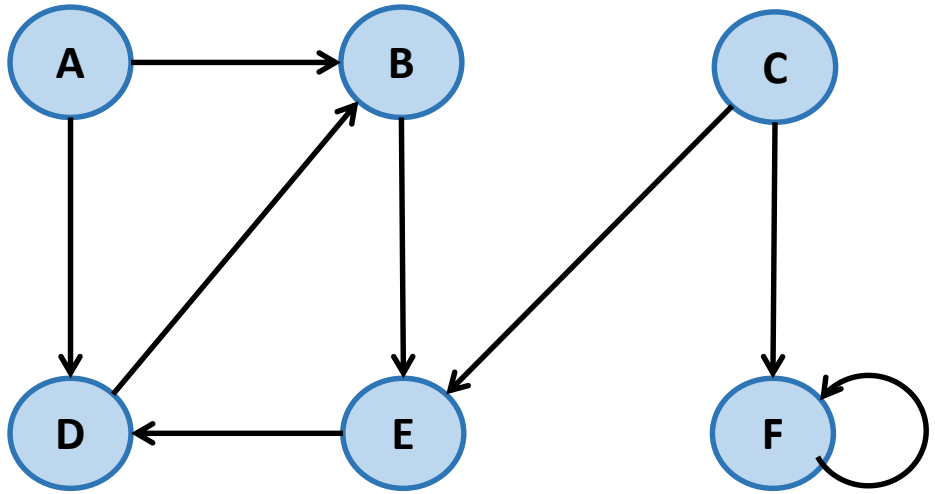
Adjacency List Representation

1. Create an array of Lists (e.g. use a linked-list)
2. The headers of the lists represents the nodes
3. Every list represents the nodes, connected from the header node



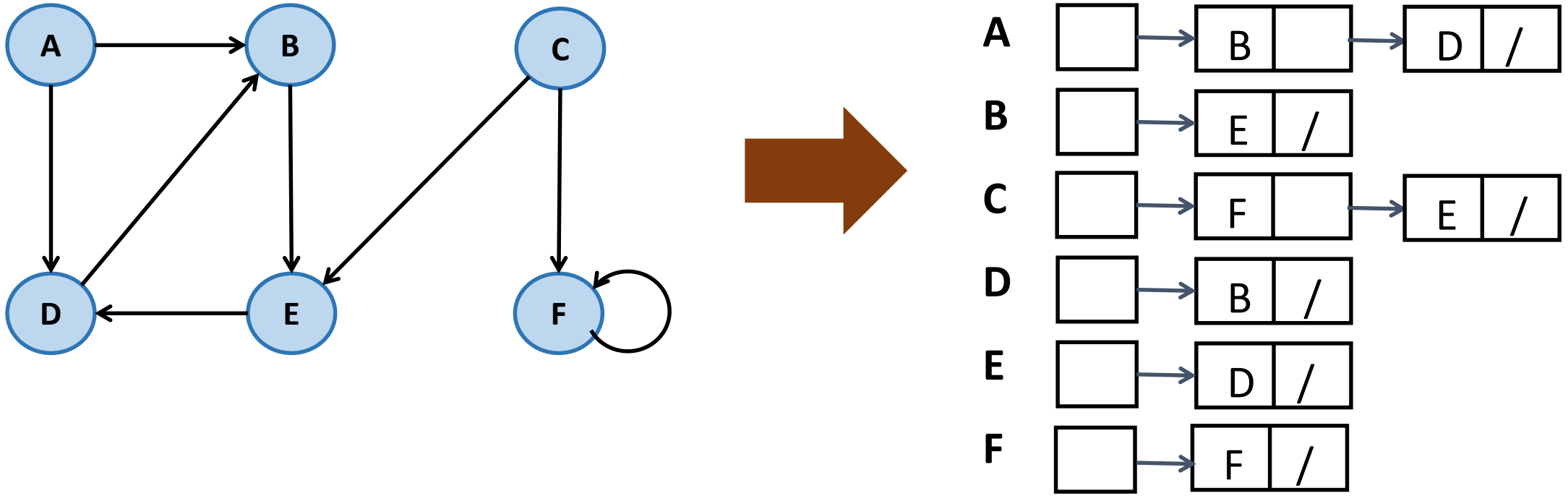


Adjacency Matrix for Directed Graph



	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	0	0	1	0
C	0	0	0	0	1	1
D	0	1	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	0	0	1

Adjacency List for Directed Graph



Major Graph Operations

- Insert
- Delete
- Search (Traversal)
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search (BFS Traversal)

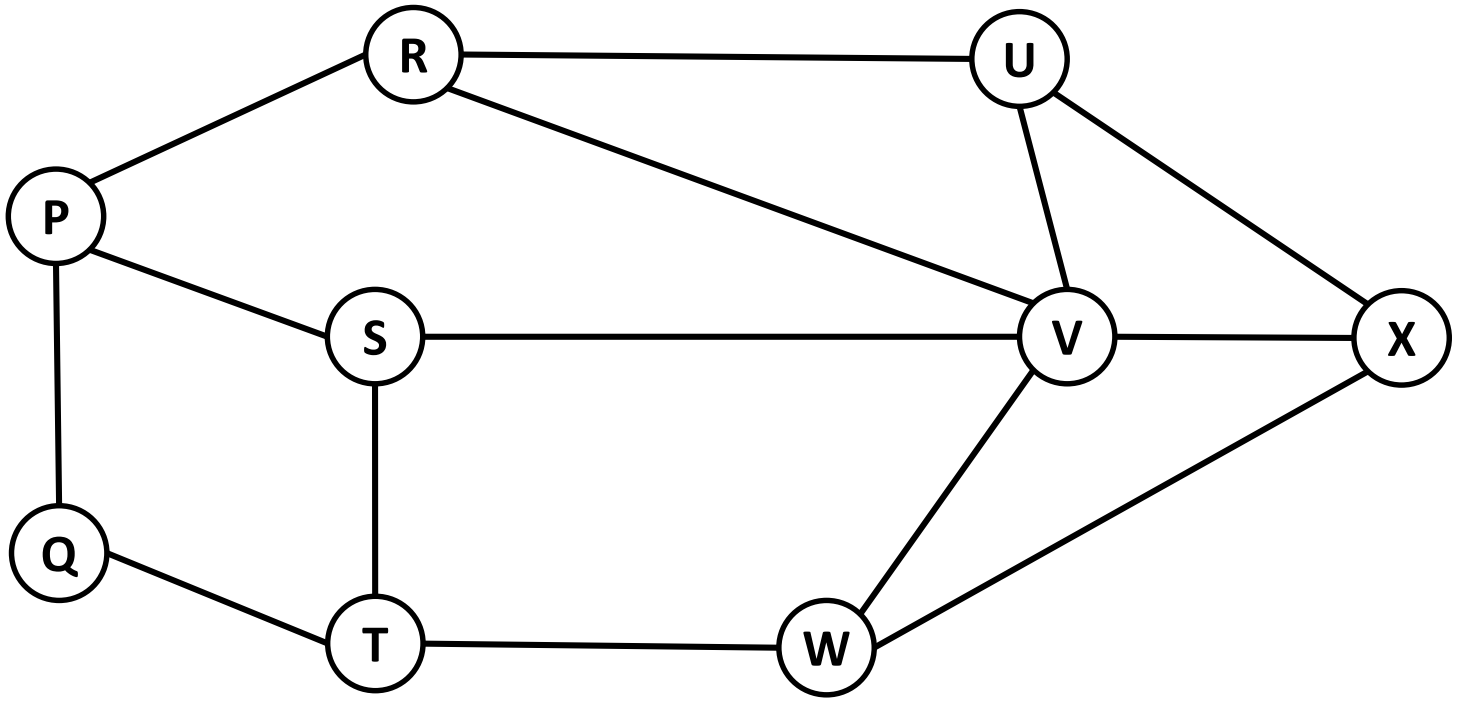
- **Idea**
  - Start from the starting node (this will be given to you)
  - Generate a traversal tree while traversing the graph
  - A node is traversed when its all successor nodes are generated
- **Implementation**: Use of a coloring scheme with a queue

Orange: Encountered but not traversed

Green: traversed

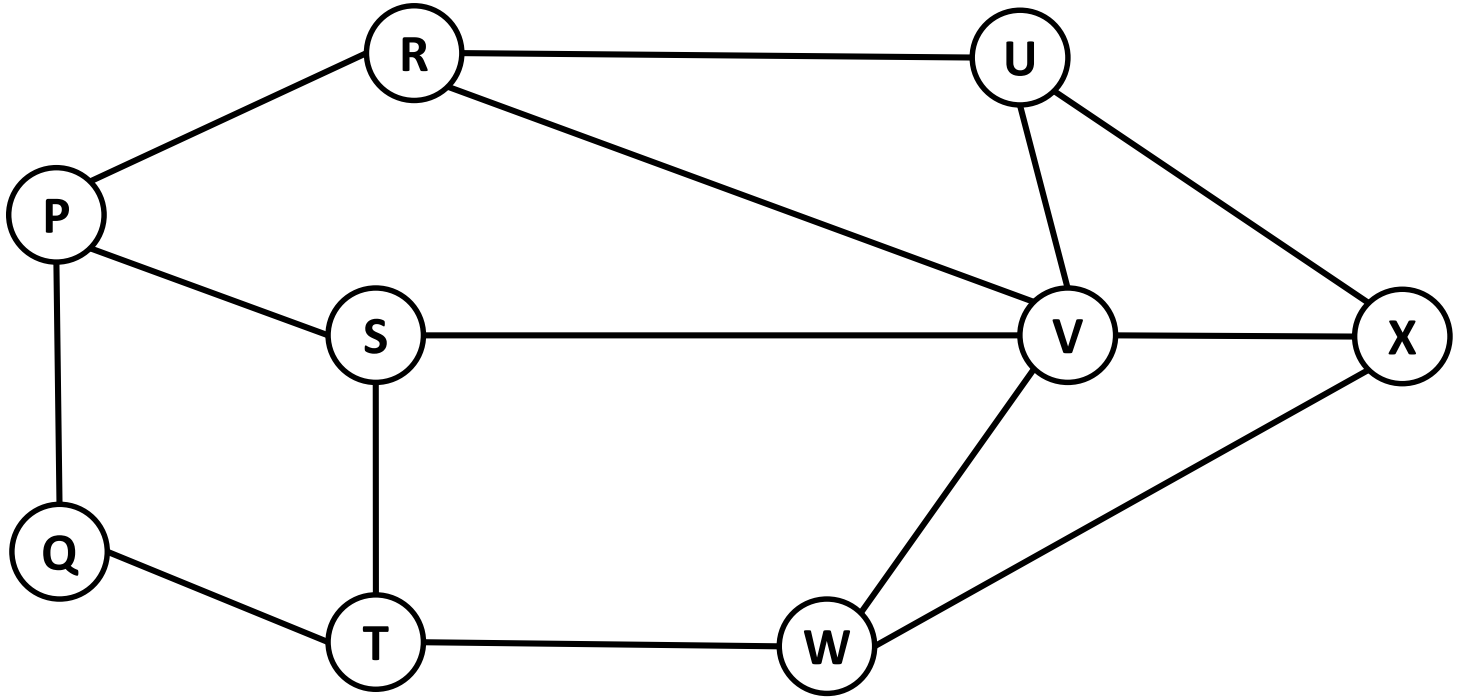
White: not accessed/encountered/generated

BFS Example – Initialization



Queue

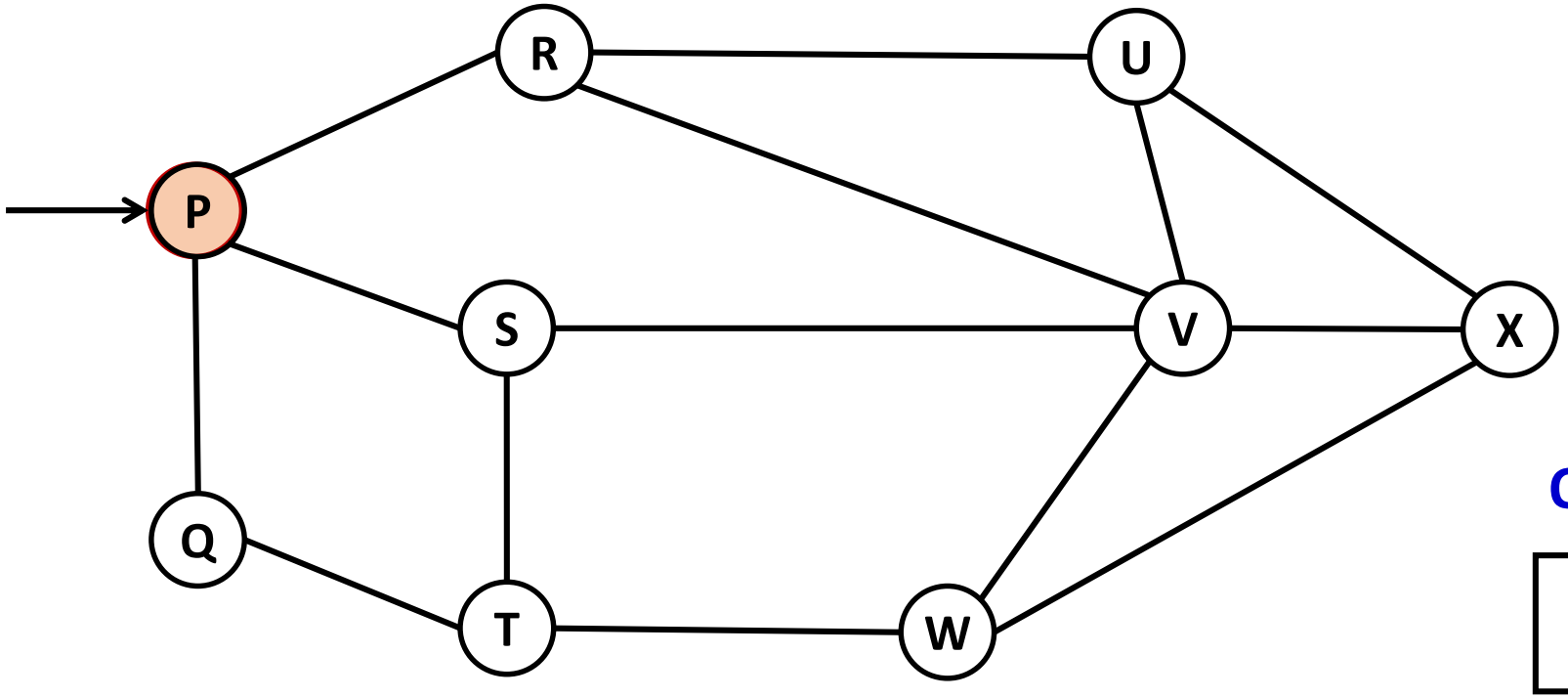
BFS Example: Step-1



Queue

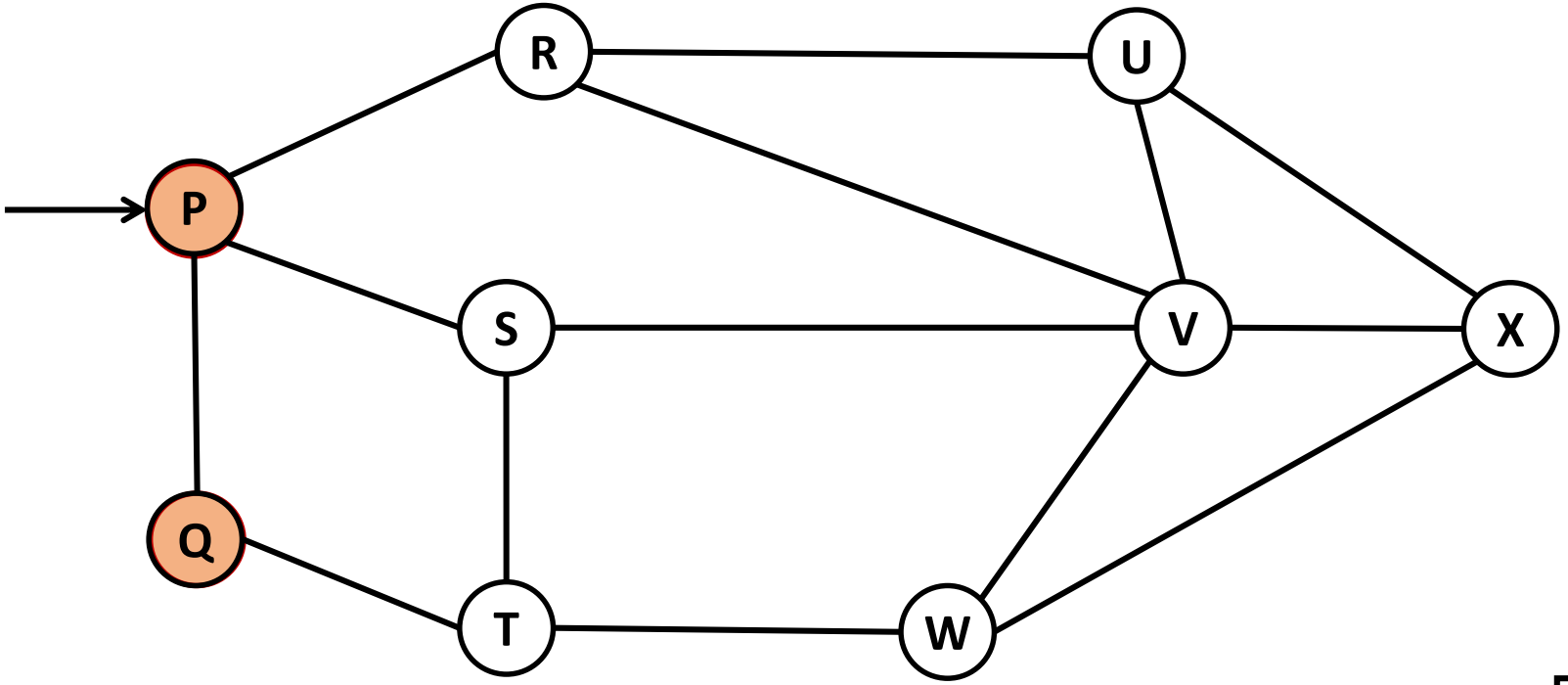
P						

BFS Example: Step-2



Queue

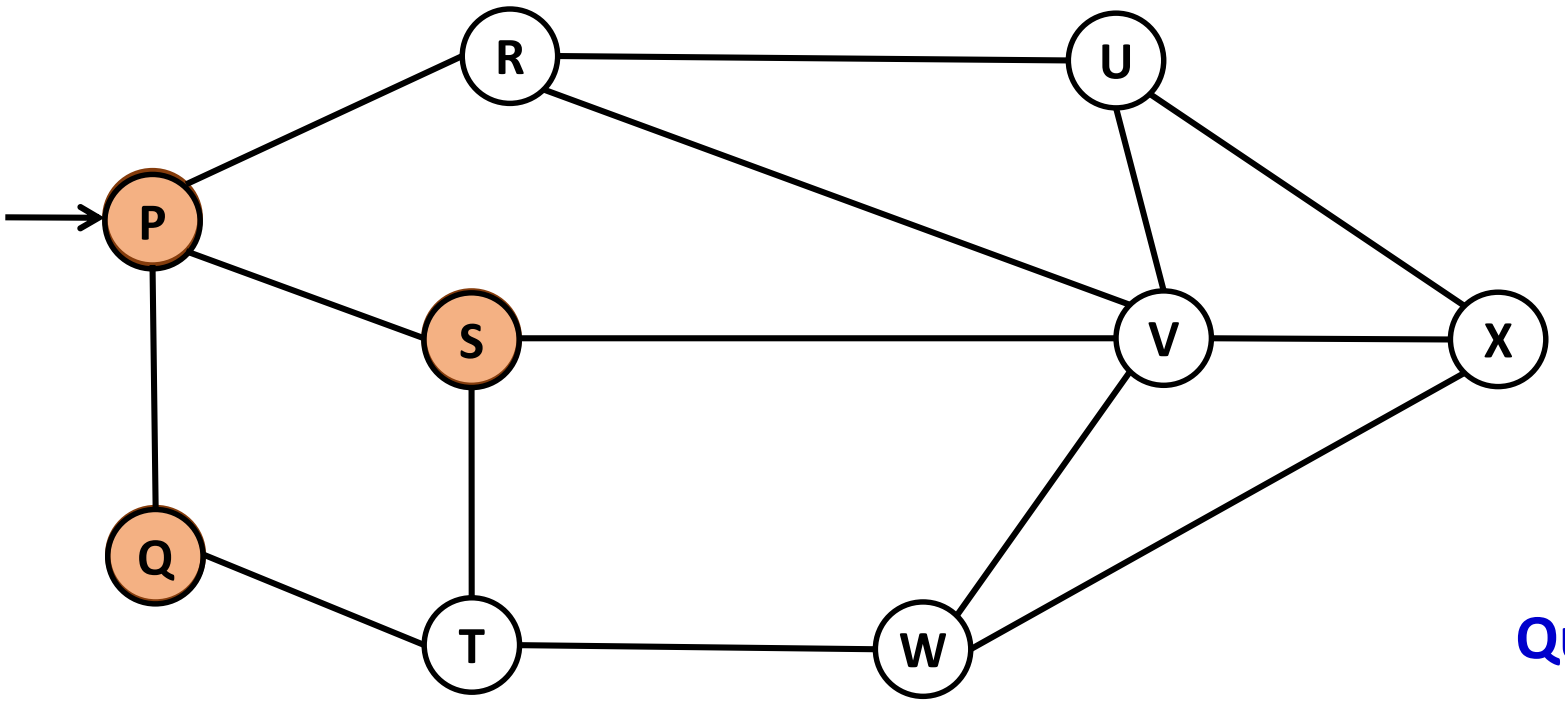
BFS Example: Step-3



Queue

P	Q					

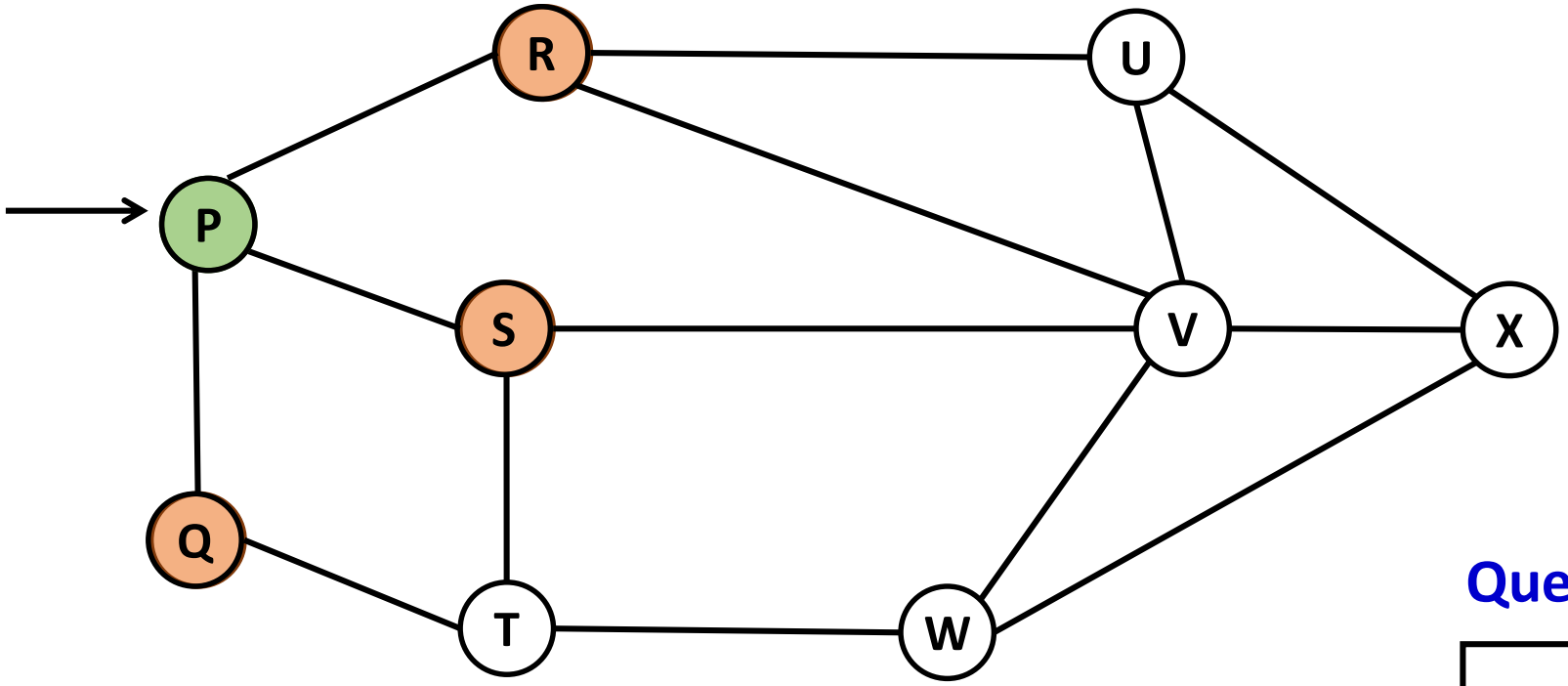
BFS Example: Step-4



Queue

P	Q	S				

BFS Example: Step-5



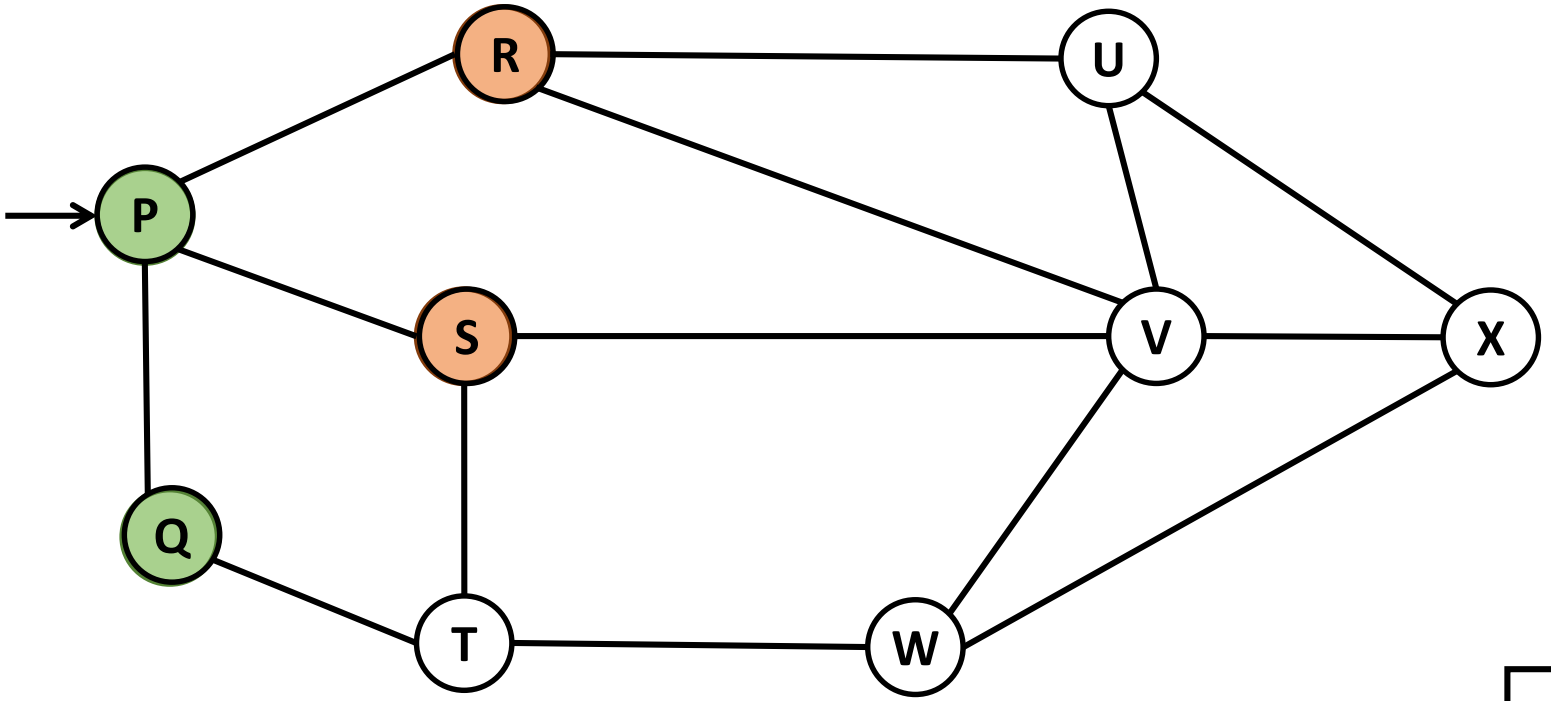
Queue

Q	S	R	T			

Traversed List

P

BFS Example: Step-6



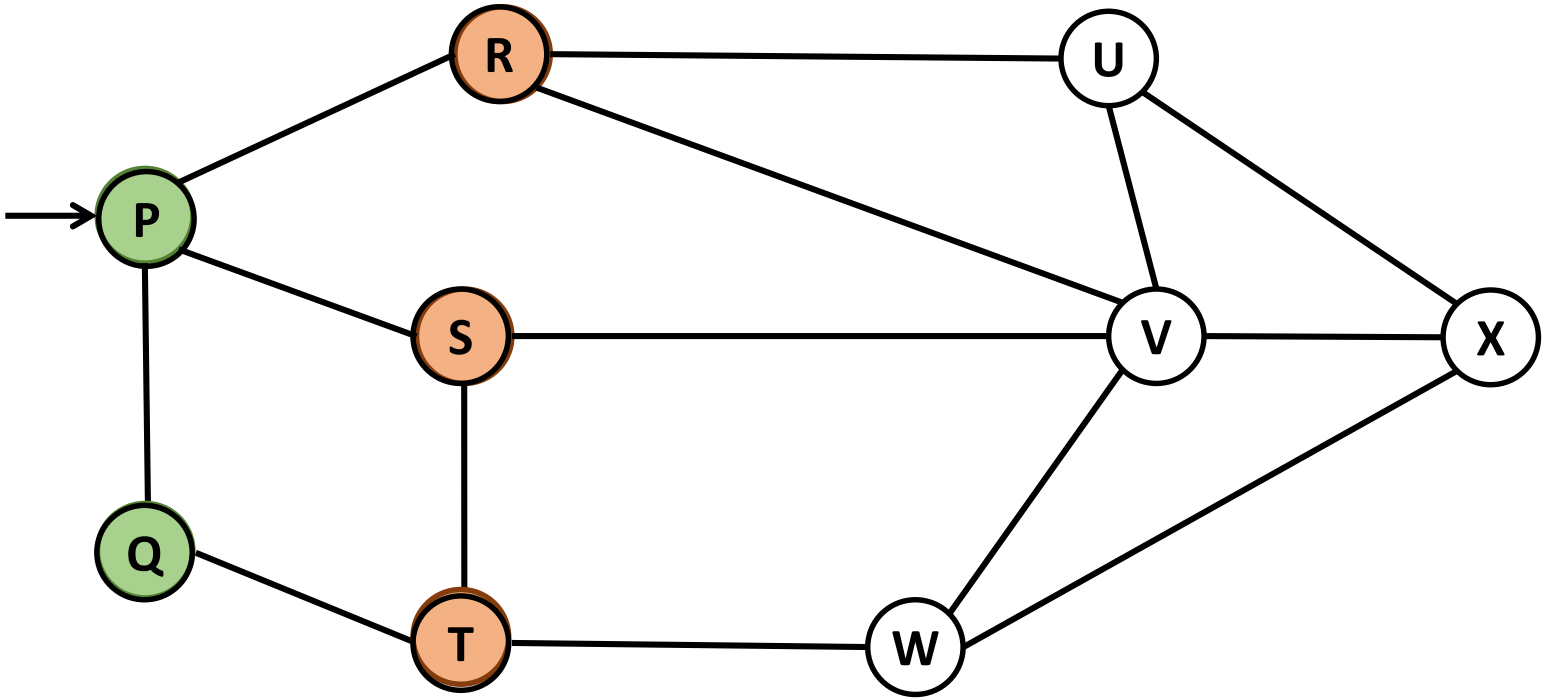
Queue

R	S	T				

Traversed List

PQ

BFS Example: Step-7



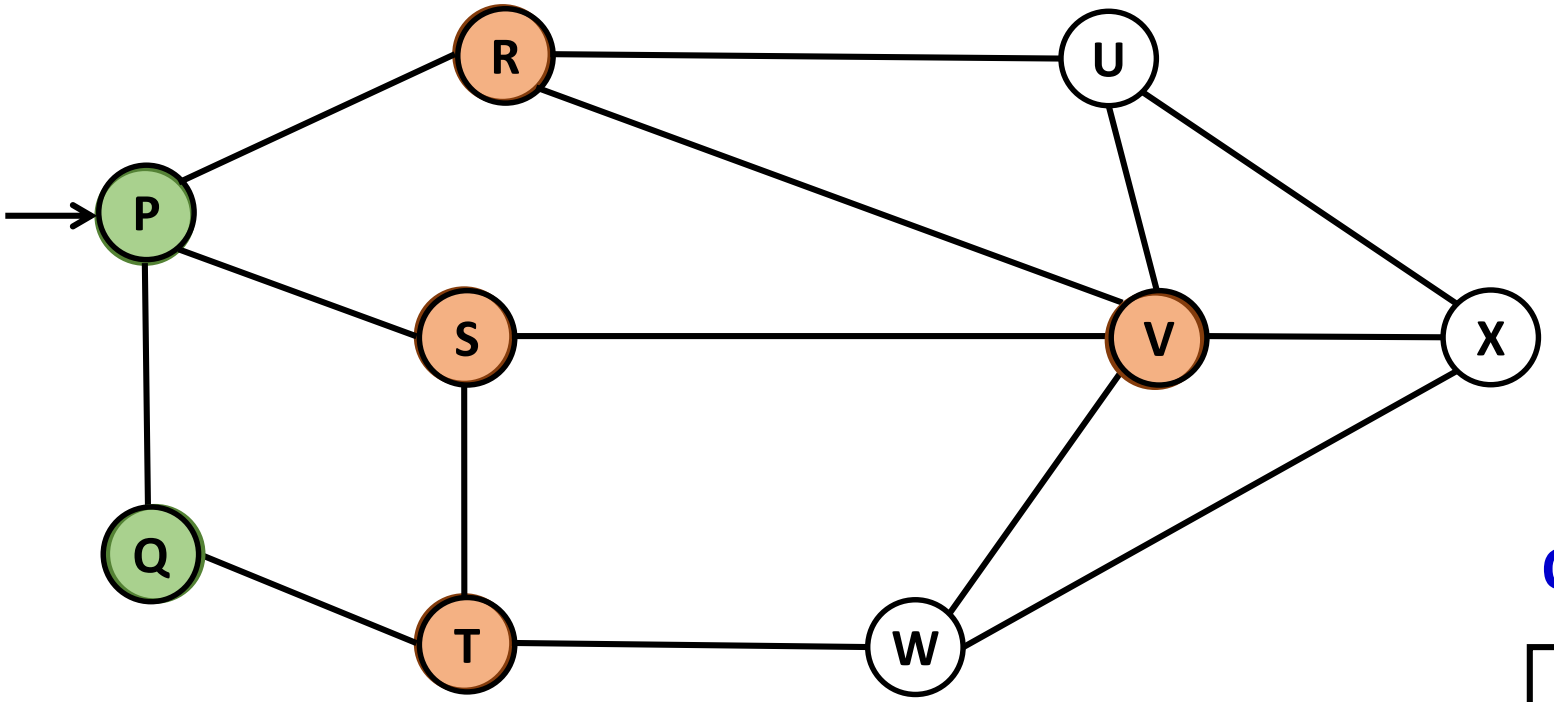
Queue

R	S	T				

Traversed List

PQ

BFS Example: Step-8



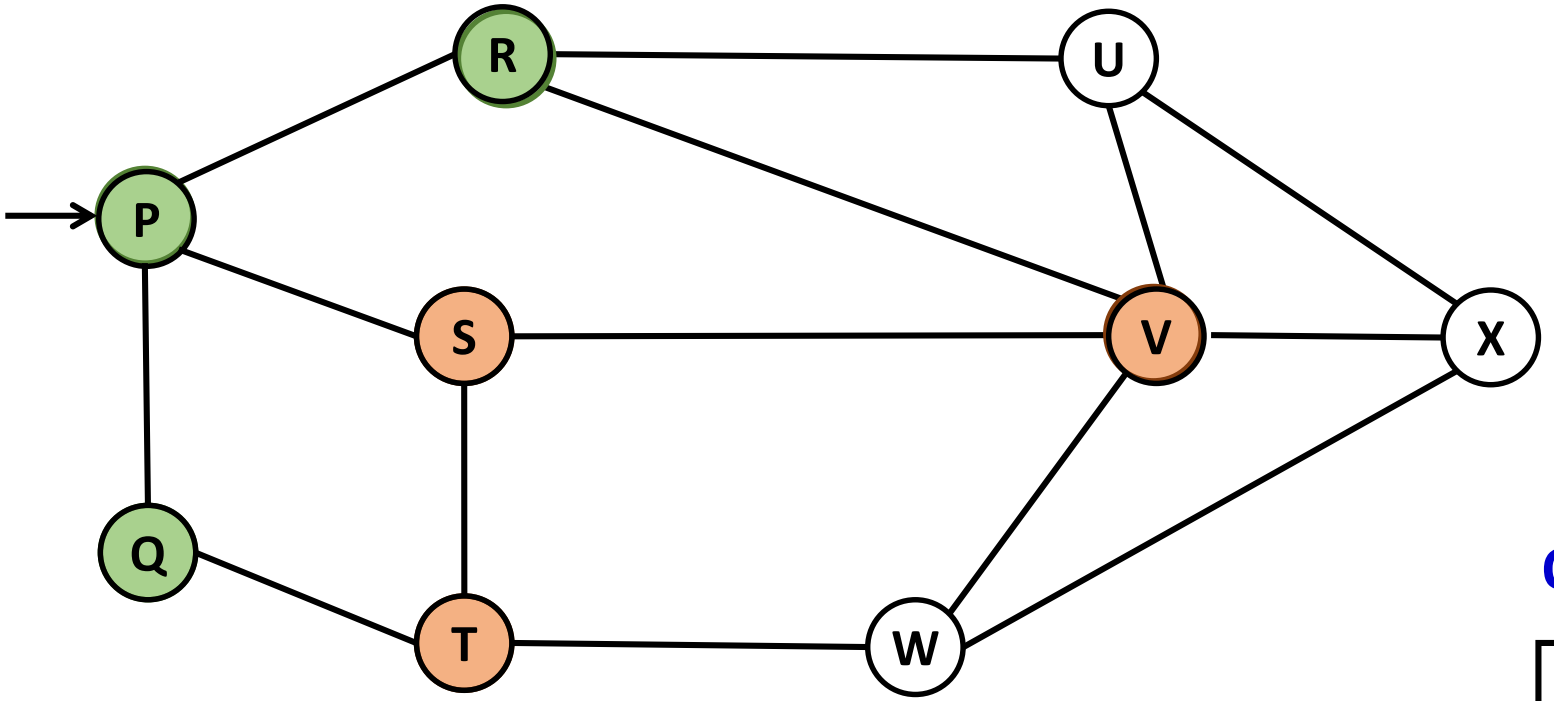
Queue

R	S	T	V			

Traversed List

PQ

BFS Example: Step-9



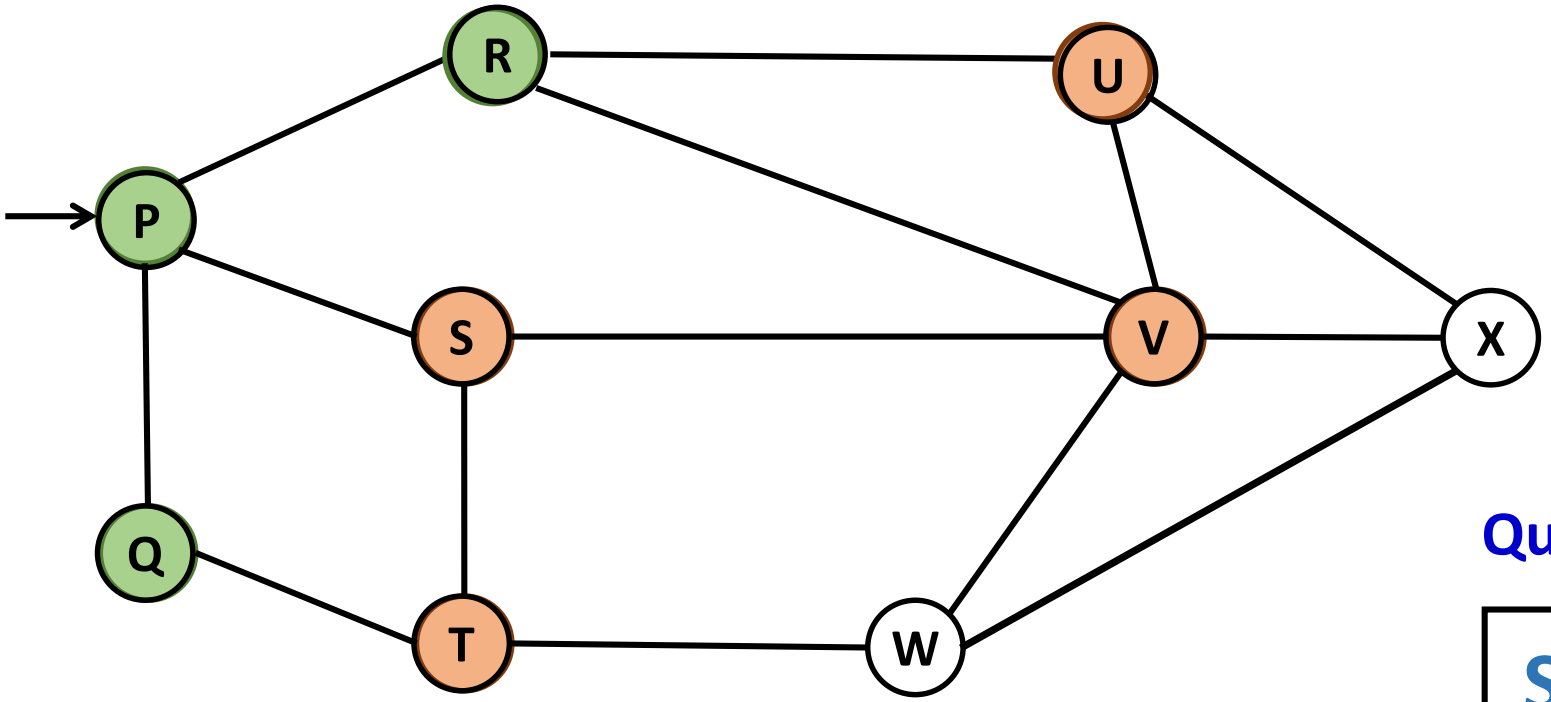
Queue

S	T	V				

Traversed List

PQR

BFS Example: Step-10



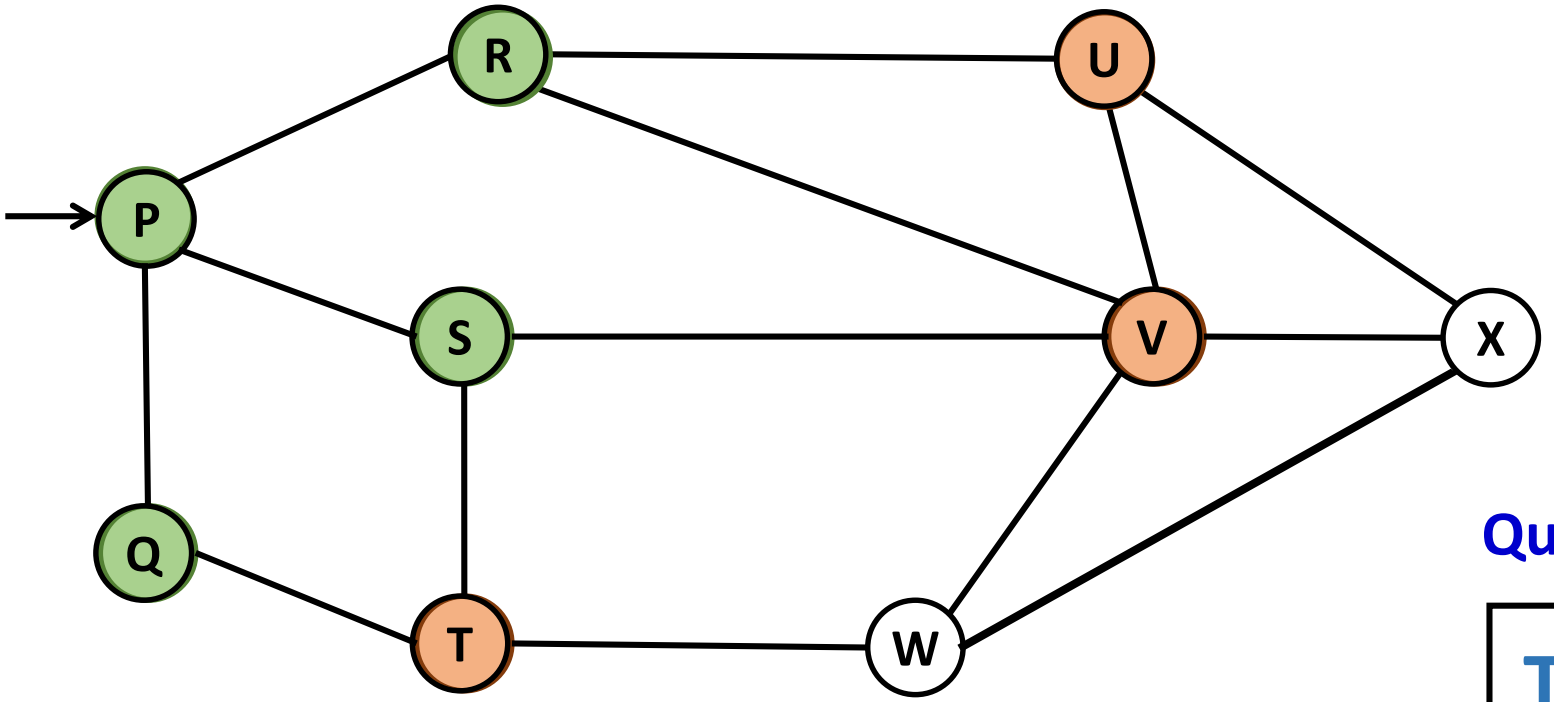
Queue

S	T	V	U			

Traversed List

PQR

BFS Example: Step-11



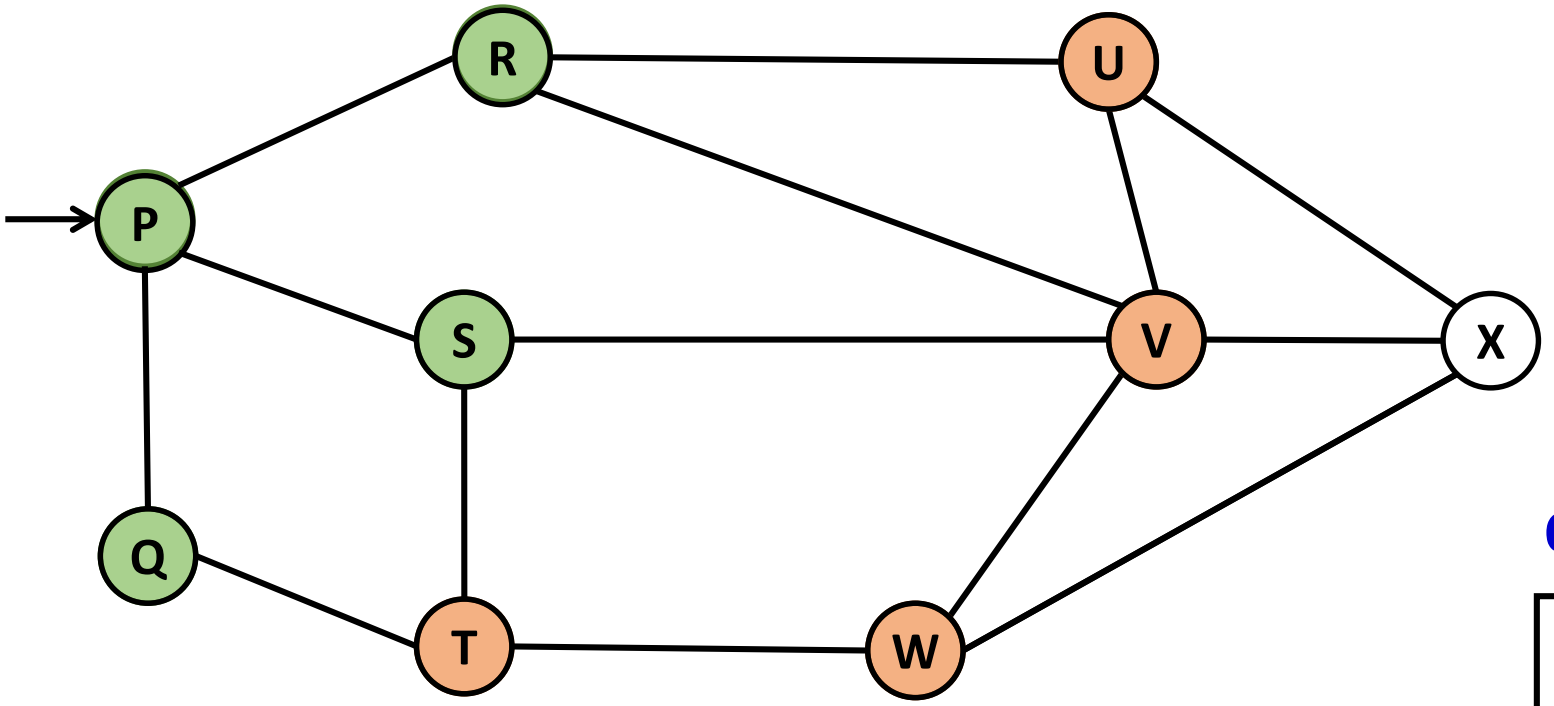
Queue

T	V	U				

Traversed List

PQRS

BFS Example: Step-12



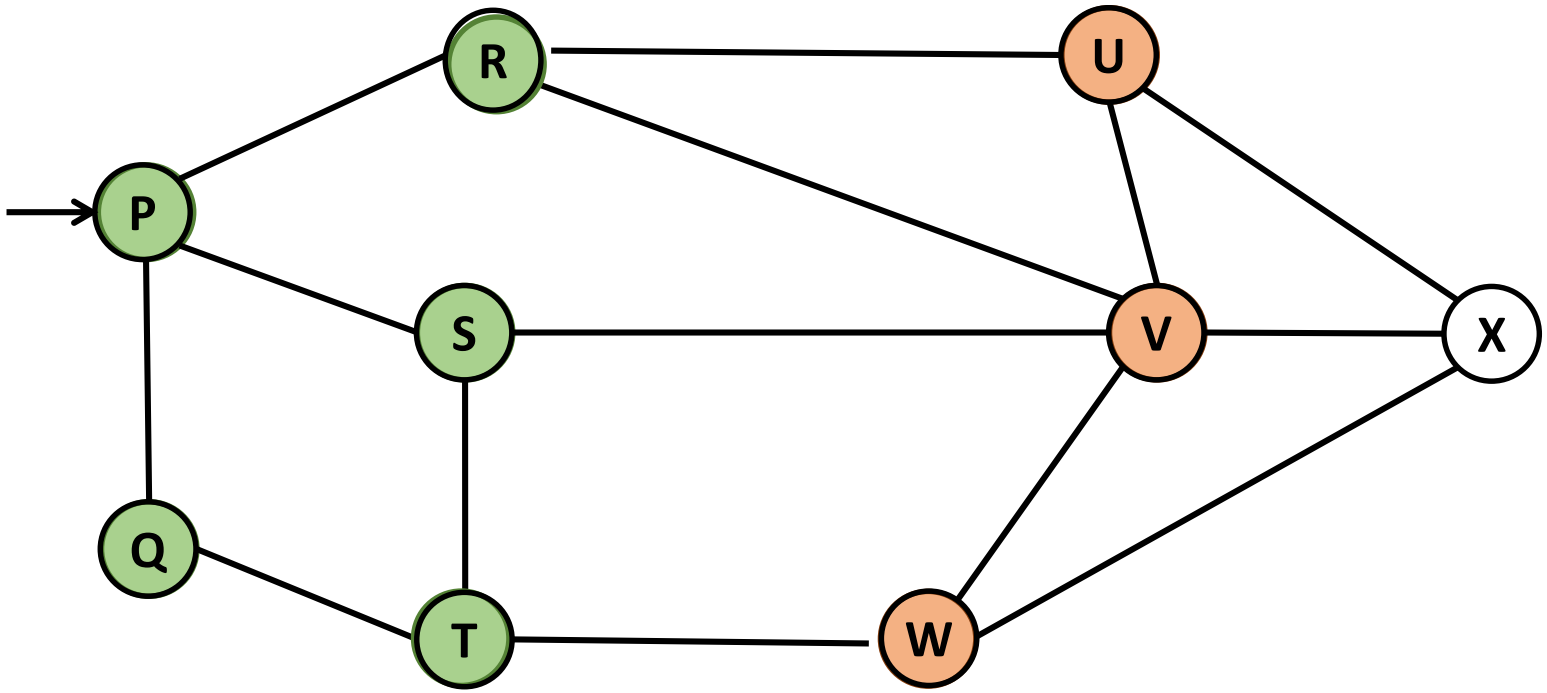
Queue

T	V	U	W			

Traversed List

PQRS

BFS Example: Step-13



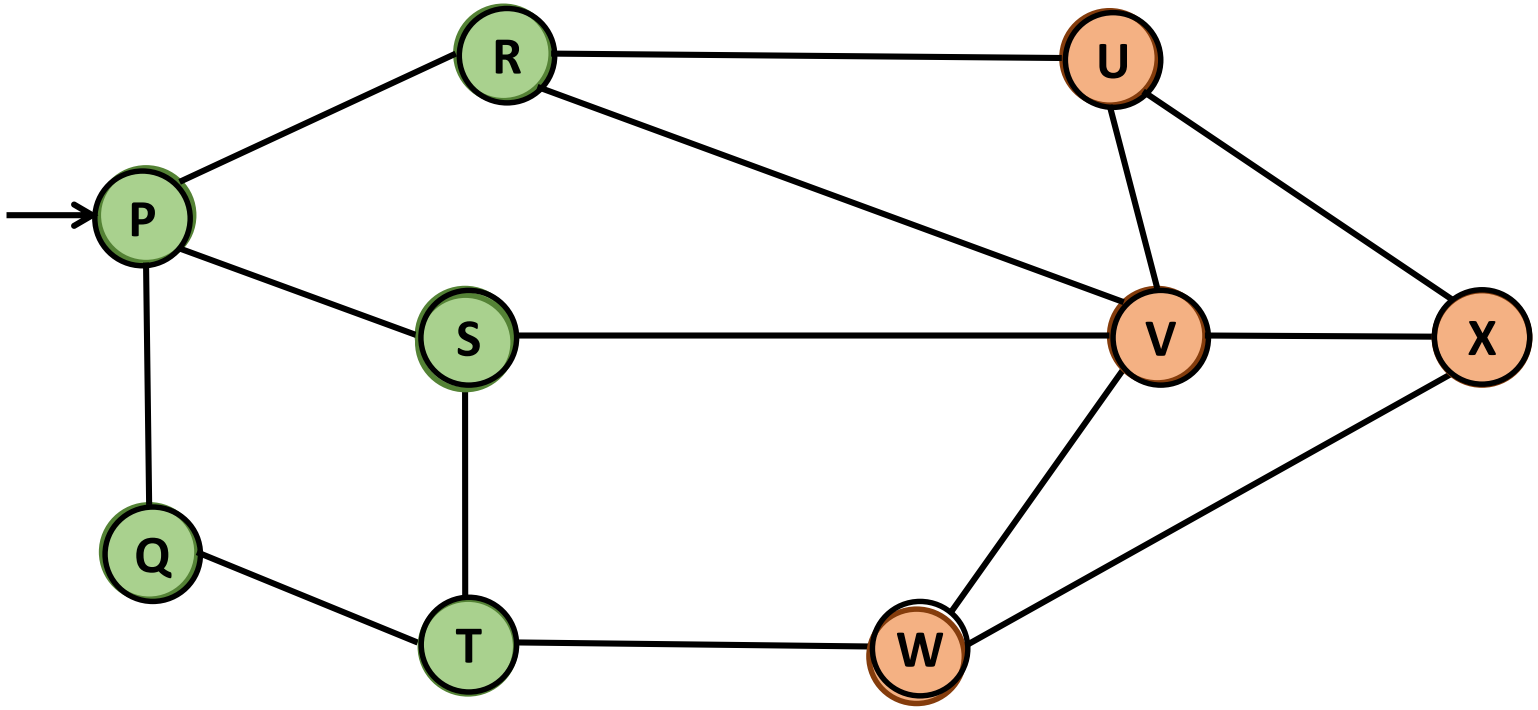
Queue

V	U	W				

Traversed List

PQRST

BFS Example: Step-14



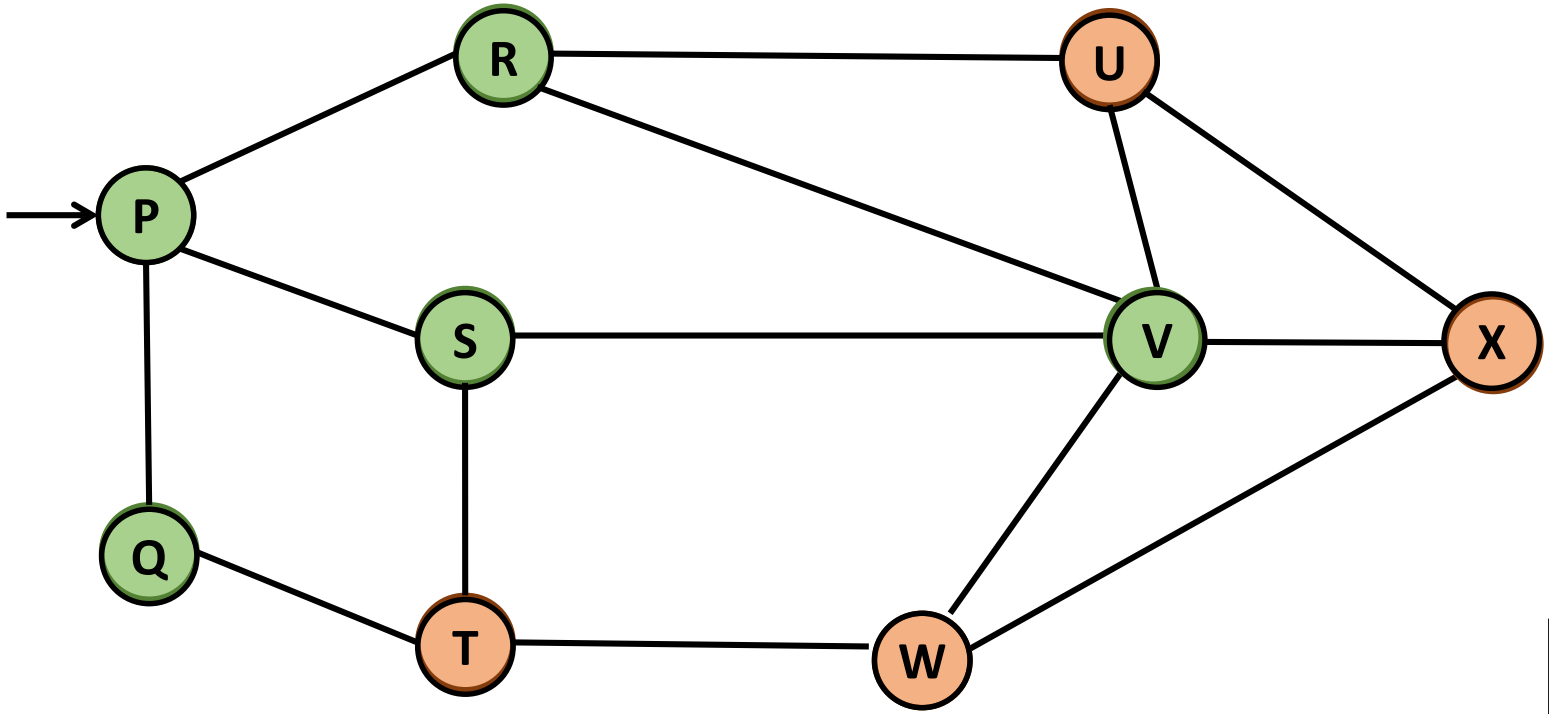
Queue

V	U	W	X			

Traversed List

PQRST

BFS Example: Step-15



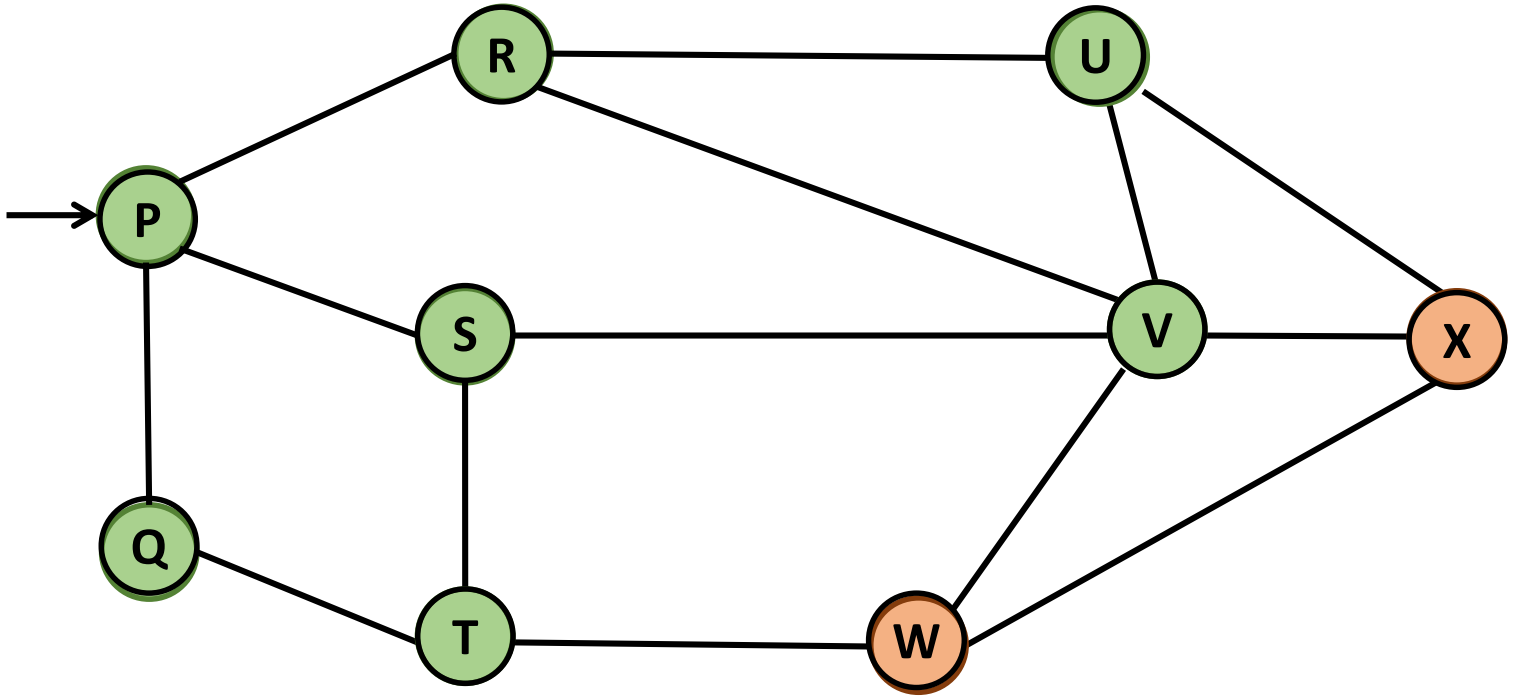
Queue

U	W	X				

Traversed List

PQRSTV

BFS Example: Step-16

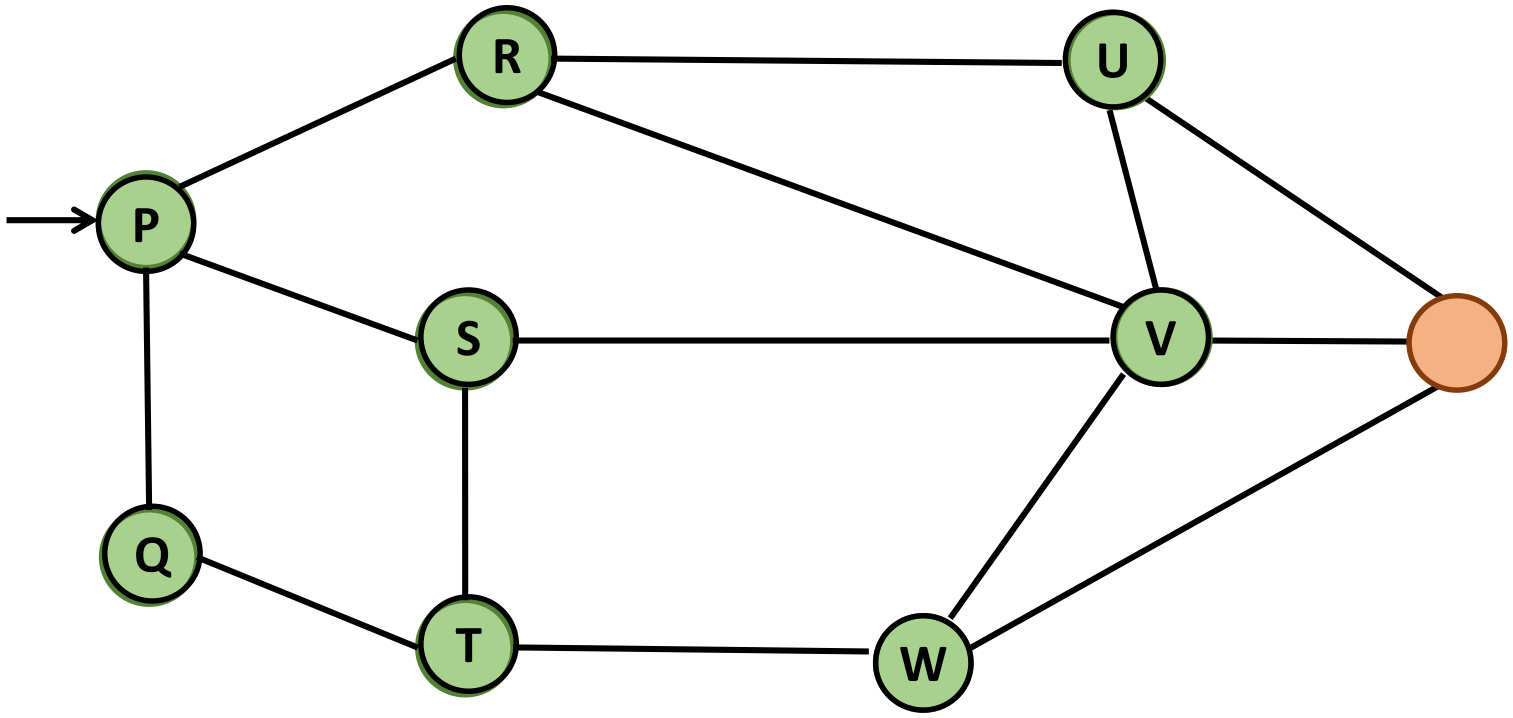


Queue

Traversed List

PQRSTVU

BFS Example: Step-17

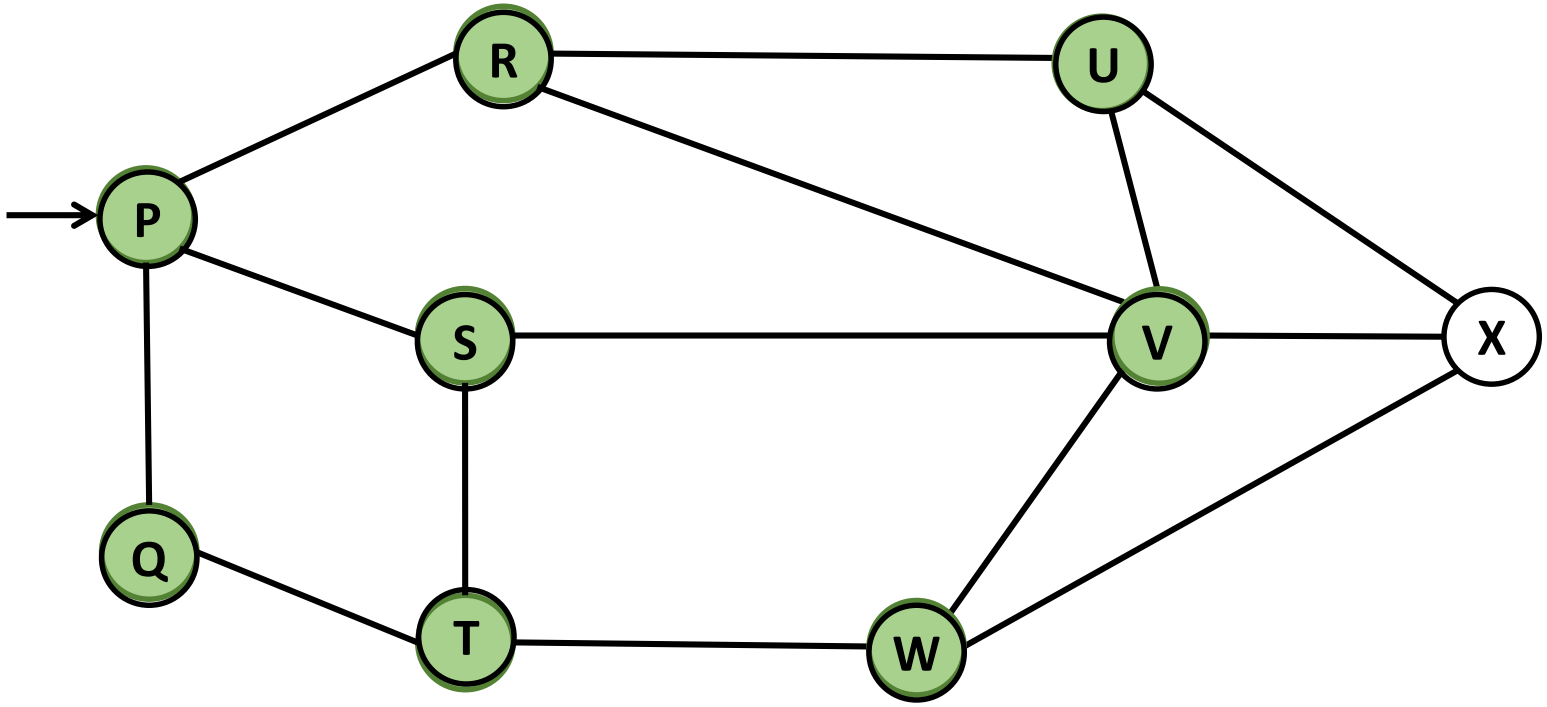


Queue

Traversed List

PQRSTVUW

BFS Example: Step-18

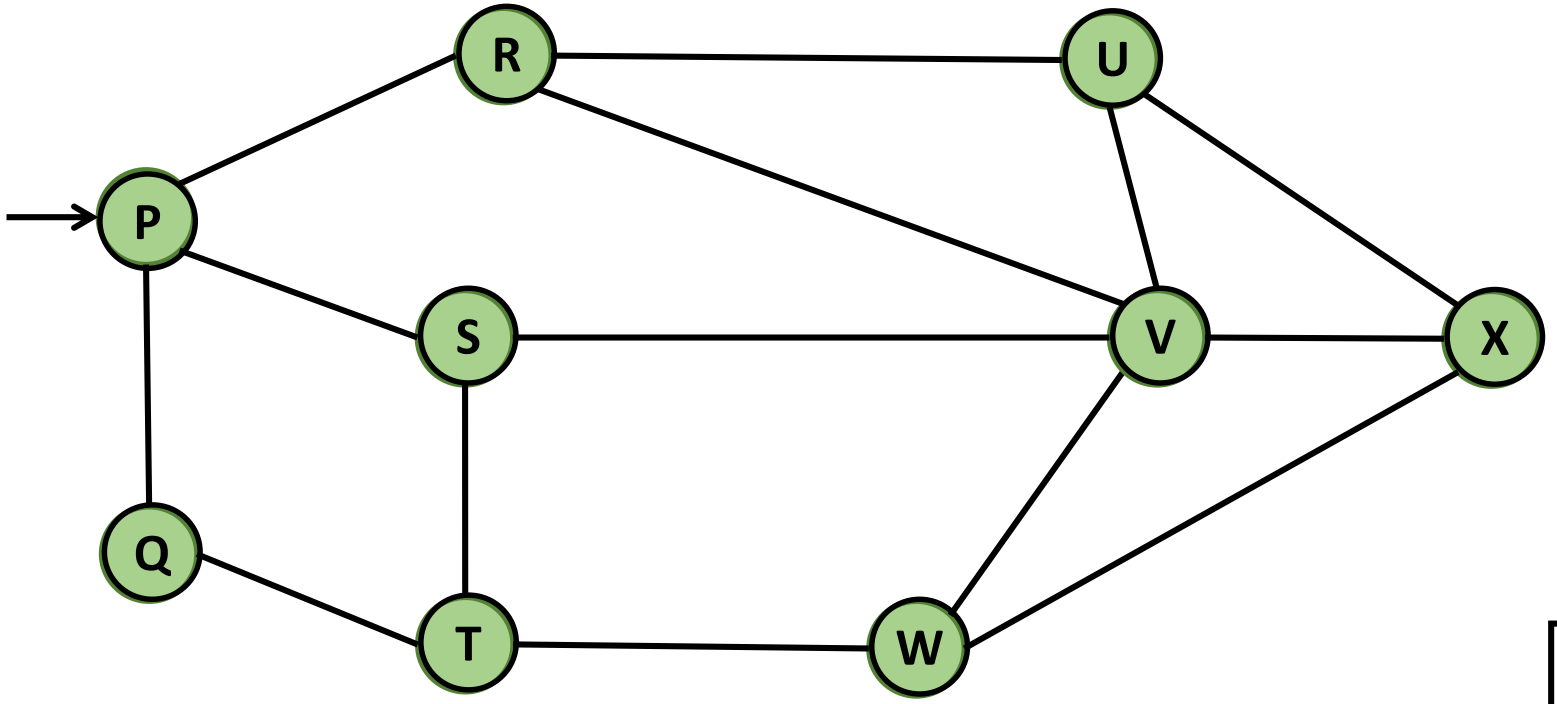


Queue

Traversed List

PQRSTVUWX

BFS Implementation: Step-23



Queue

Traversed List

PQRSTVUWX

Graph: Depth-First Traversal

- **Key Idea**
  1. Same as with tree traversal using DFS
  2. Use a stack
  3. Use three colors White, Orange, and Green, similarly

Depth-First Traversal

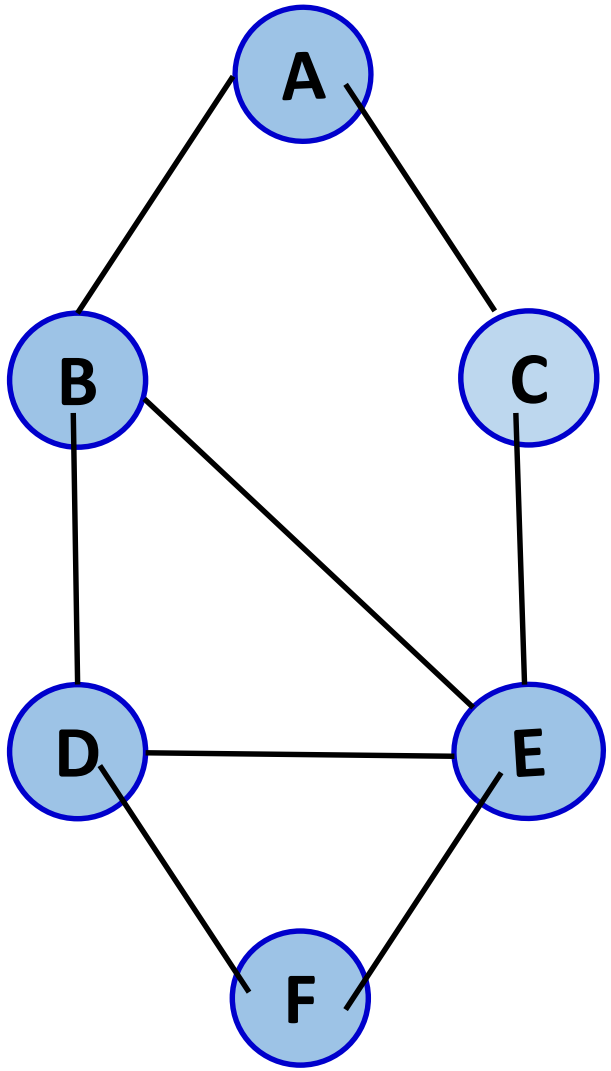
- **Idea**: If possible, go (in depth) forward

  else back track

- **Problem**: Since we have cycles, each node

  may be visited infinite times

- **Solution**: use a Boolean visited array



Problems Specific with Graphs

Minimum Spanning Tree

- Path Problems
  1. Simple Paths
  2. **Shortest Path Problem**
     - Single source shortest paths
     - All-pair shortest paths
  3. Find Cycles
  4. Euler Path and Circuit Problem
  5. Hamiltonian Path and Circuit Problem (or TSP)

Single Source Shortest Path Algorithms

Two famous algorithms for single source shortest paths

- **Dijkstra's Algorithm**
- Bellman-Ford's Algorithm

- Both use "edge relaxation" approach
- **Dijkstra's algorithm** (greedy approach) is faster
  - Can not be used with graphs having negative weight edges
- Bellman-Ford's algorithm (dynamic programming approach) can work with graphs having negative weight edges
  - But not with "negative cycles"

Greedy Algorithms

- Always makes local optimal choice at each stage with the intent of finding a global optimum.
- Might not always produce an optimal solution,
- Nonetheless yields local optimal solutions that approximate a global optimal solution in a reasonable amount of time.

Five components of Greedy Algorithms

1. **Candidate set**: From which a solution is created
2. **Selection function**: Chooses the best candidate to be added to the solution
3. **Feasibility function:** Determines if a candidate can be used to contribute to a solution
4. **Objective function:** Assigns a value to a solution, or a partial solution, and
5. **Solution function**: Indicates when we have discovered a complete solution

Thank you!

BFS: Algorithm

35

```
BFS(G, s)
1       for each vertex u ∈ V [G] do
2               color[u] ← WHITE
3               dist[u] ← ∞
4               π[u] ← NULL //traversed set
5       color[s] ← ORANGE
6               dist[s] ← 0
7               π[s] ← NIL
8       Q ← Φ
9       ENQUEUE (Q,s)
10      while Q !=Φ do
11              u ←DEQUEUE (Q)
12              for each v ∈ Adj [u] do
13                      if color [v] = WHITE
14                              color[v] ←ORANGE
```

```
void Graph::DFSUtil(int v, bool visited[])
{
    visited[v] = true;// Mark the current node as visited and print it
    print v;
    list<int>::iterator i;// Recur for all the vertices adjacent to this vertex
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}
void Graph::DFS()// The function to do DFS traversal. It uses recursive DFSUtil()
{
    bool *visited = new bool[V];// Mark all the vertices as not visited
    for (int i = 0; i < V; i++)
        visited[i] = false;
    for (int i = 0; i < V; i++) /* Call the recursive helper function to print
                        DFS traversal starting from all vertices one by one*/
        if (visited[i] == false)
            DFSUtil(i, visited);
}
```

Depth-First Traversal

```
DFS(G)
1for each vertex u ∈ V [G] do
2    color[u] ← WHITE
3    π[u] ← NIL
4    time ← 0
5for each vertex u ∈ V [G] do
6    if color[u] = WHITE then
7        DFS-VISIT(u)
```

```
DFS-VISIT(u)
1    color[u] ← ORANGE
2    time ← time +1
3    d[u] ← time          //discovery time
4    for each v ∈ Adj[u] do
5        if color[v] = WHITE
6            then π[v] ← u
7            DFS-VISIT(v)
8    color[u] ← GREEN
9    f [u] ← time ← time +1   //finish time
```

Thank you!