- **Graph:**

  A graph $G = (V, E)$ consists of two sets,

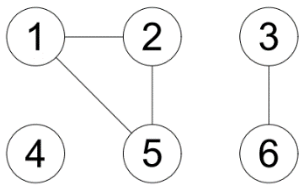
  V = set of vertices (nodes)

  E = set of edges = subset of $(V \times V)$

- **Representation**

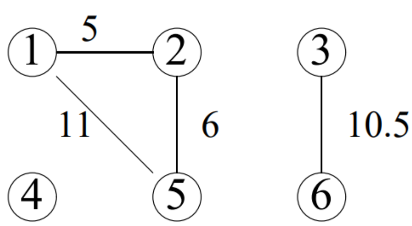
  1. **Adjacency Matrix:**

     - Undirected graph



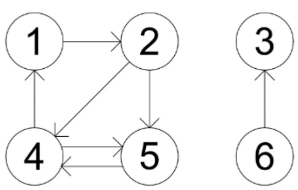
$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

     - Weighted undirected graph



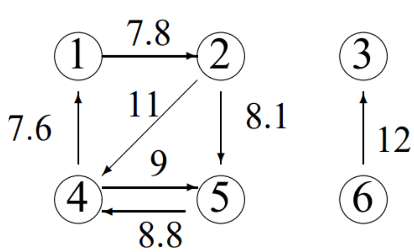
$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{bmatrix} \infty & 5 & \infty & \infty & 11 & \infty \\ 5 & \infty & \infty & \infty & 6 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10.5 \\ \infty & \infty & \infty & \infty & \infty & \infty \\ 11 & 6 & \infty & \infty & \infty & \infty \\ \infty & \infty & 10.5 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

     - Directed graph



$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

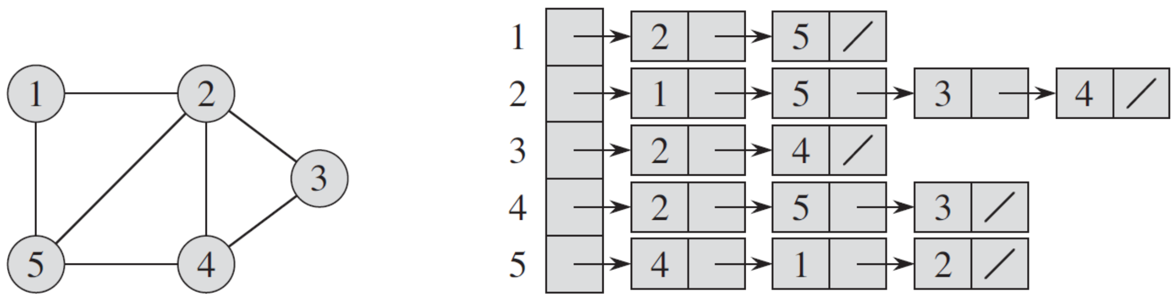
     - Weighted directed graph



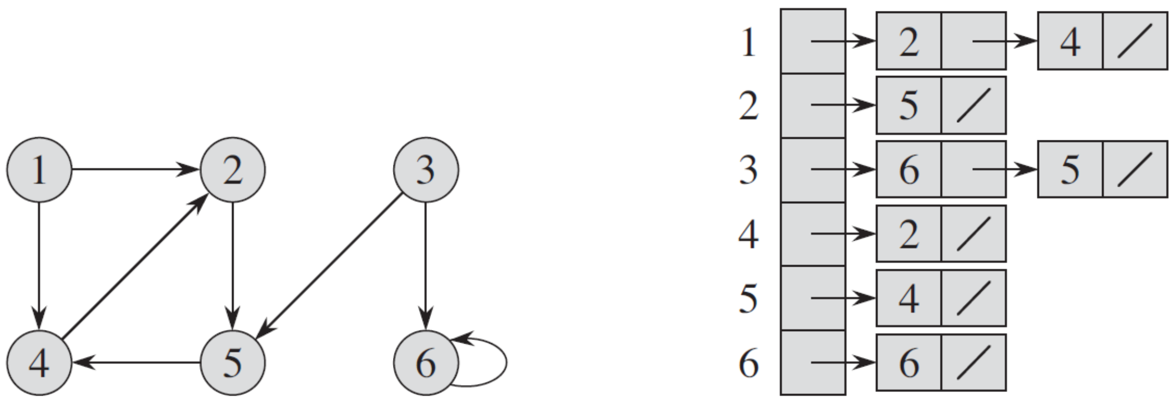
$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{bmatrix} \infty & 7.8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 11 & 8.1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ 7.6 & \infty & \infty & \infty & 9 & \infty \\ \infty & \infty & \infty & 8.8 & \infty & \infty \\ \infty & \infty & 12 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

2. **Adjacency List:**

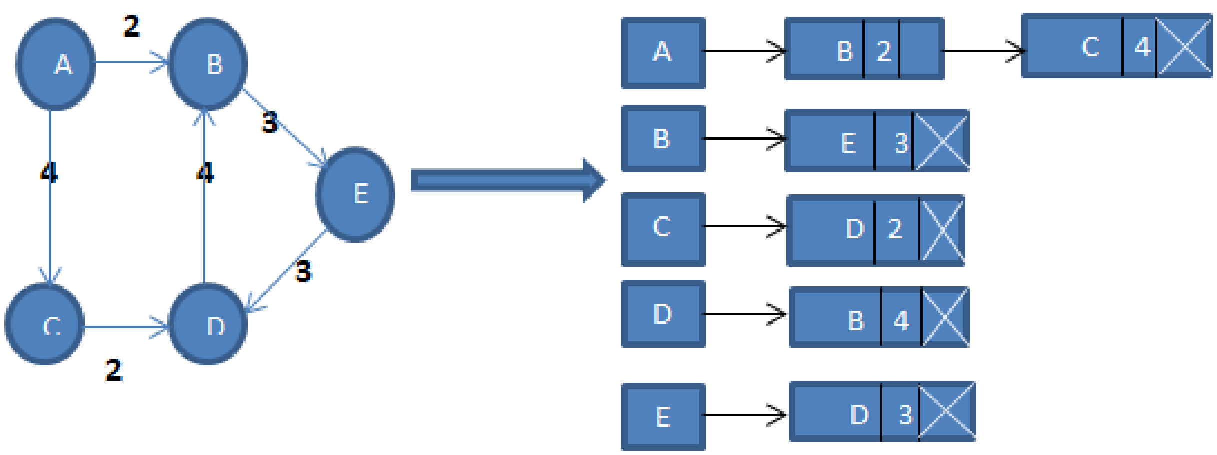
- Undirected graph



- Directed graph



- Weighted graph



- ## Shortest-paths Problem

1. **Single-source:**

   Find a shortest path from a given source vertex to each of the other vertices. This paradigm also works for the **single-destination shortest path** problem. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

2. **Single-pair:**

   Given 2 vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.

3. **All-pairs:**

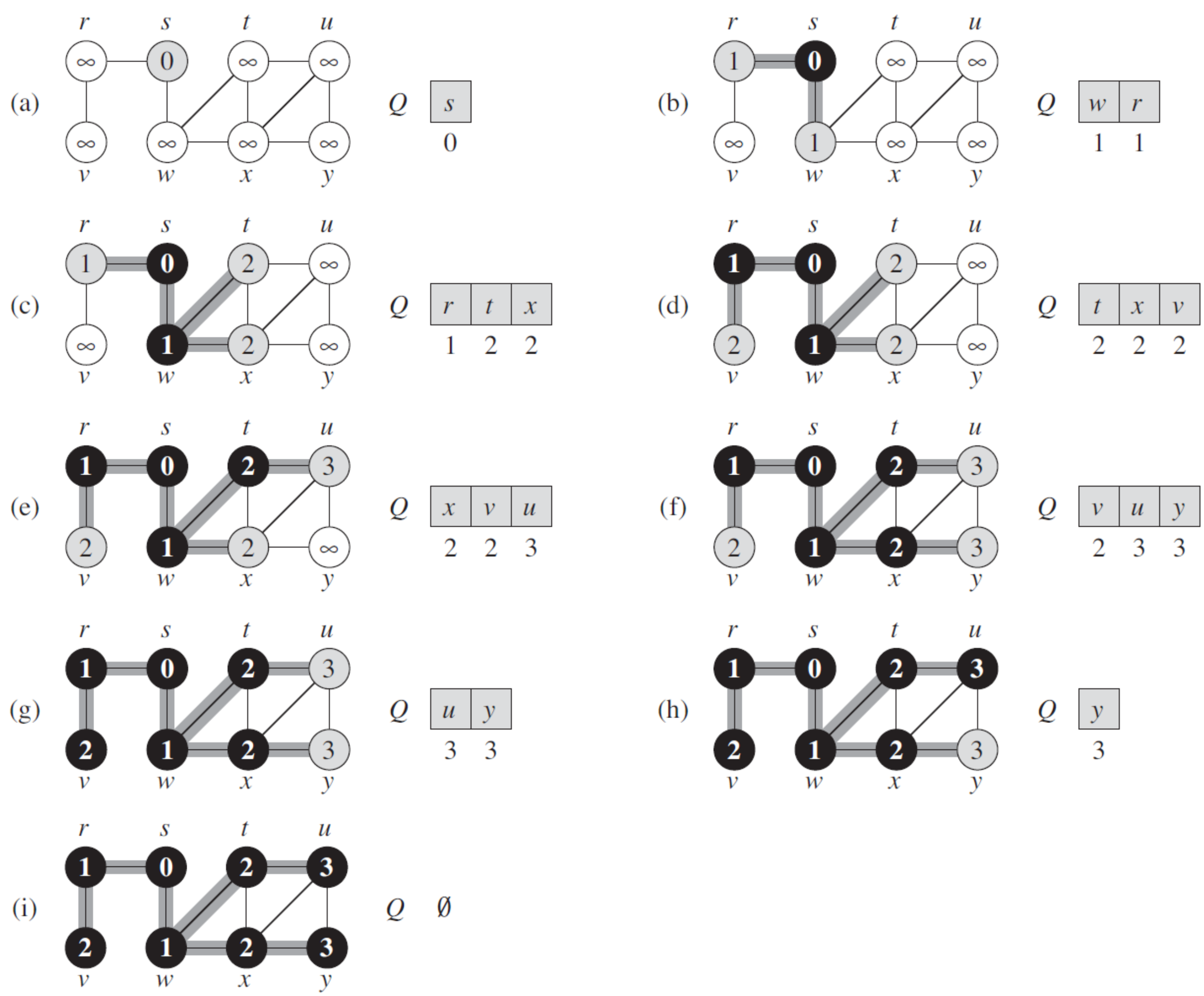
   Find shortest-paths for every pair of vertices.

- **Single-source shortest path problem:** Given a graph $G = (V, E)$ with weight function $w: E \to \mathbb{R}$ and a source vertex $s \in V$, find for all vertices $v \in V$ the minimum possible weight for path from s to v.

Different Algorithms:

- **Breadth-first search (BFS)** – If all the edge weights are equal.
- **Dijkstra** – If all the edge weights are positive.
- **Bellman-Ford** – If all the edge weights are positive and negative.

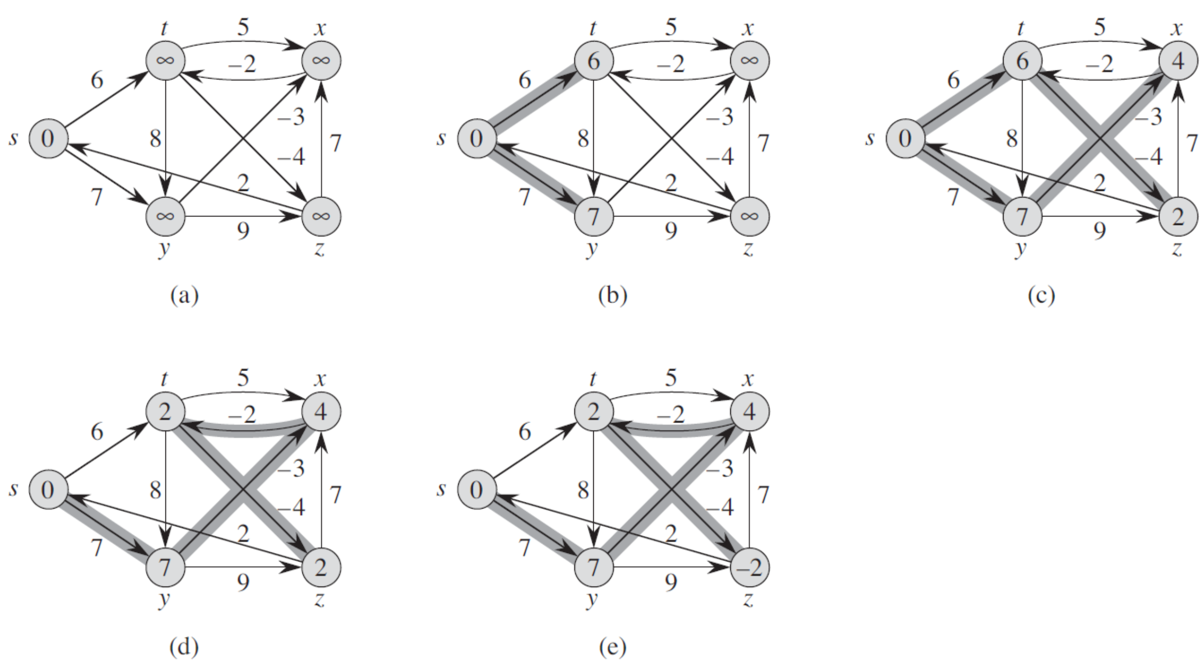
1. **Breadth-first search (BFS) –** $O(|V| + |E|)$



Algorithm:

```
BFS(G, s)
for each vertex u in V[G] − {s}
        color[u] ← white
        d[u] ← ∝
        π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q, s)
while Q ≠ Φ
    u ← dequeue(Q)
    for each v in Adj[u]
            if color[v] = white
                    color[v] ← gray
                    d[v] ← d[u] + 1
                    π[v] ← u
                    enqueue(Q, v)
    color[u] ← black
```

2. **Bellman-ford** – $O(|V||E|)$



Algorithm:

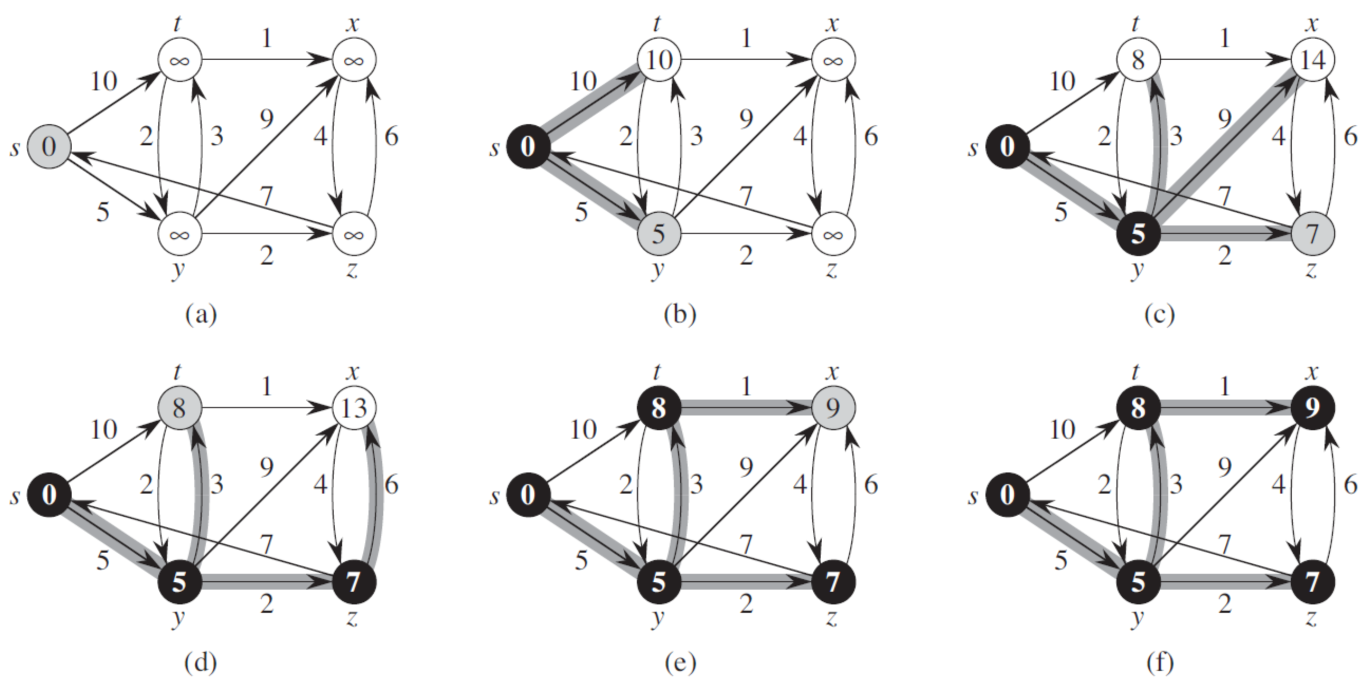
```
BELLMAN-FORD(V, E, w, s)
    INITIALIZE-SINGLE-SOURCE(V, s)
    for i ← 1 to |V| - 1
        do for each edge (u, v) ∈ E
              do RELAX(u, v, w)
    for each edge (u, v) ∈ E
        do if d[v] > d[u] + w(u, v)
              then return FALSE
    return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(V, s)
    for each v ∈ V
        do d[v] ← ∞
           π[v] ← NIL
    d[s] ← 0
```

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
      then d[v] ← d[u] + w(u, v)
           π[v] ← u
```

Negative Cycle Detection
This section detects negative cycle. If exists then the algorithm will return **False** otherwise **True**.

3. **Dijkstra's algorithm** – min-priority queue: Array $O(|V|^2)$, Binary heap $O(|E| \lg (|V|))$, Fibonacci heap $O(|V| \lg(|V|) + |E|)$



Algorithm

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(V, s)
    S ← ∅
    Q ← V[G]
    while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
            S ← S ∪ {u}
            for each vertex v ∈ Adj[u]
                do RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(V, s)
    for each v ∈ V
        do d[v] ← ∞
            π[v] ← NIL
    d[s] ← 0
```

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
        then d[v] ← d[u] + w(u, v)
            π[v] ← u
            also update the Priority Queue
                                    value
```

C++ STL – Priority Queue

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>    ///for greater function

using namespace std;

class mycomp{
public:
    bool operator()(const pair<int,int> &elm1, const pair<int,int> &elm2){
        return elm1.second>elm2.second;
    }
};

int main()
{
    priority_queue<int> pq; ///default max heap

    pq.push(10);
    pq.push(-5);
    pq.push(8);
    pq.push(2);
    pq.push(4);

    while(!pq.empty()){
        cout<< pq.top() <<" ";
        pq.pop();
    }
    cout<<endl;

    ///---------------------------------------------------------------------------

    priority_queue<int, vector<int>, greater<int> > pq1; ///use as min heap

    pq1.push(10);
    pq1.push(-5);
    pq1.push(8);
```

```
    pq1.push(2);
    pq1.push(4);

    while(!pq1.empty()){
        cout<< pq1.top() <<" ";
        pq1.pop();
    }
    cout<<endl;

    ///---------------------------------------------------------------------------

    priority_queue< pair<int, int>, vector< pair<int,int> >, mycomp > pq2; ///use as min
heap

    pair<int,int> p1(10,8);
    pair<int,int> p2(5,10);
    pair<int,int> p3(9,18);
    pair<int,int> p4(2,7);
    pair<int,int> p5(9,5);

    pq2.push(p1);
    pq2.push(p2);
    pq2.push(p3);
    pq2.push(p4);
    pq2.push(p5);

    while(!pq2.empty()){
        cout<<"("<<pq2.top().first <<" "<<pq2.top().second<<")"<<" ";
        pq2.pop();
    }
    cout<<endl;

    return 0;
}
```