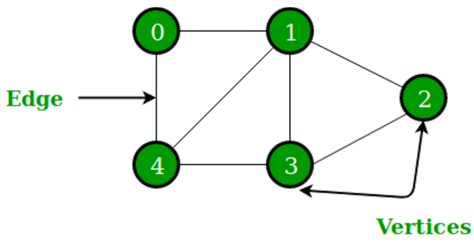
Graph

Contents

- Terminologies
- Graph representation
- Graph traversal
- Spanning tree
- Shortest path

Graph

- A graph consists of **a finite set of vertices** (or nodes) and **set of edges** which connect a pair of nodes.
- G = {V, E}
  - V: set of vertices. V = {v_1, v_2,…, v_n}
  - E: set of edges. E = {e_1, e_2,…,e_m}
- Example:
  - V = {0, 1, 2, 3, 4}
  - E = {01, 04, 12, 13, 14, 23, 34}

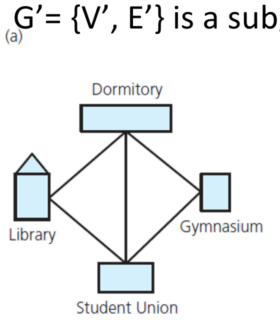


Terminologies

fit@hcmus

Terminologies

- A **subgraph** consists of a subset of a graph's vertices and a subset of its edges.
  - G'= {V', E'} is a subgraph of G = {V, E} if V' ⊆ V, E' ⊆ E



(a) A campus map as a graph;
(b) a subgraph

fit@hcmus

Terminologies

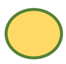
- **Vertex**: also called a **node**.
- **Edge**: connects two vertices.
- **Loop** (*self-edge*): An edge of the form (*v*, *v*).
- **Adjacent**: two vertices are **adjacent** if they are joined by an edge.

Terminologies

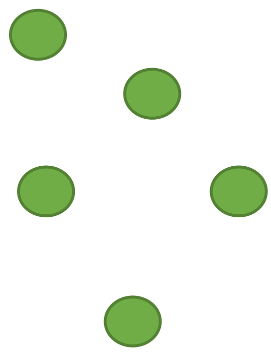
- **Path**: A sequence of edges that begins at one vertex and ends at another vertex.
  - If all vertices of a path is distinct, the path is **simple**.
- **Cycle**: A path that starts and ends at the same vertex and does not traverse the same edge more than once.
- **Acyclic** graph: A graph with no cycle.

Terminologies

- **Null graph**: A graph having no edges
- **Trivial graph**: A graph with only one vertex.



trivial graph



null graph

Terminologies

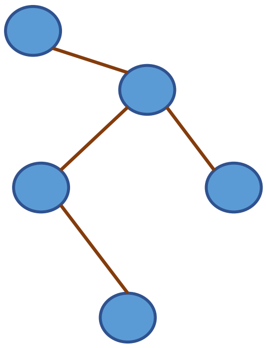
- **Path**: A sequence of edges that begins at one vertex and ends at another vertex.
  - If all vertices of a path is distinct, the path is simple.
- **Cycle**: A path that starts and ends at the same vertex and does not traverse the same edge more than once.
- **Acyclic graph**: A graph with no cycle.

Terminologies

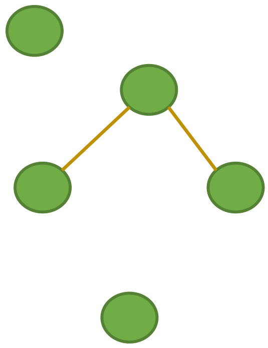
- **Connected graph:** A graph in which each pair of **distinct vertices** has a **path** between them.
- **Disconnected graph:** A graph does not contain at least two connected vertices.
- **Complete graph:** A graph in which each pairs of **distinct vertices** has an **edge** between them
- Graph cannot have duplicate edges between vertices.
  - **Multigraph**: does allow multiple edges

fit@hcmus

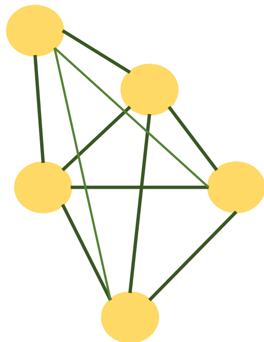
Terminologies



connected graph



disconnected graph



complete graph

fit@hcmus

Terminologies

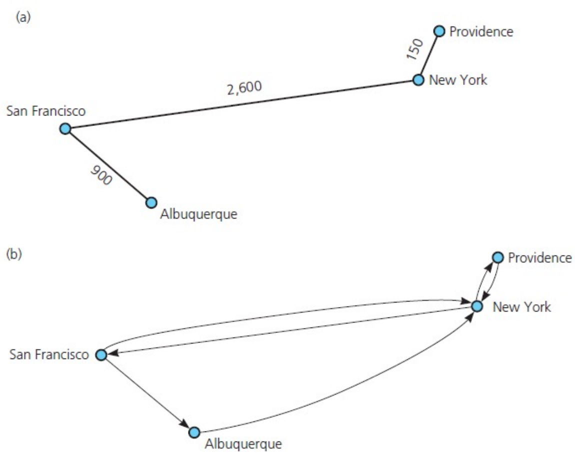
- **Undirected graph**: the graph in which edges do not indicate a direction.
- **Directed graph, or digraph**: a graph in which each edge has a direction.
- **Weighted graph**: a graph with numbers (weights, costs) assigned to its edges.

fit@hcmus

Terminologies

(a): undirected graph

(b): directed graph



fit@hcmus

Graph Representation

fit@hcmus

Graph Representation

- Adjacency Matrix
- Adjacency List

fit@hcmus

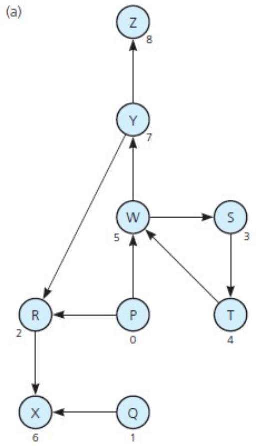
Adjacency Matrix

A[*n*][*n*] with *n* is the number of vertices.

- $A[i][j] = \begin{cases} 1 \; if \; there \; is \; an \; edge(i,j) \\ 0 \; if \; there \; is \; no \; edge \; (i,j) \end{cases}$
- $A[i][j] = \begin{cases} w \quad with \; w \; is \; the \; weight \; of \; edge(i,j) \\ \infty \qquad if \; there \; is \; no \; edge \; (i,j) \end{cases}$

Adjacency Matrix

(a)

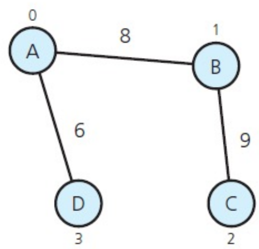


(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Adjacency Matrix

(a)



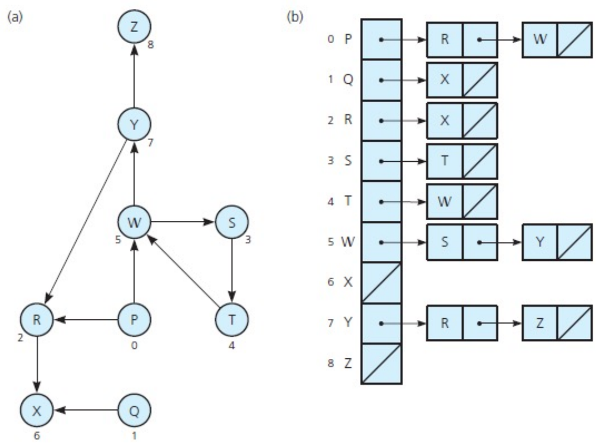
(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

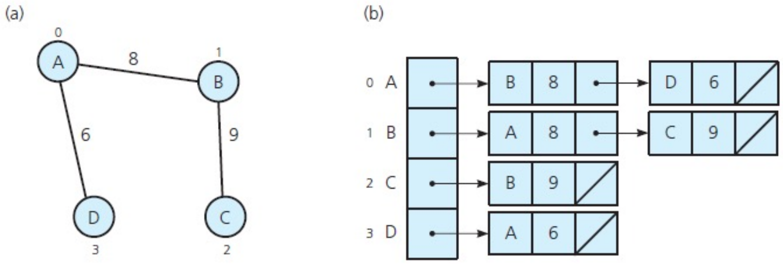
Adjacency List

- A graph with n vertices has n linked chains.
- The i^{th} linked chain has a node for vertex j if and only if having edge (i,j).

Adjacency List



Adjacency List



Graph Traversal

Graph Traversal

- Visits (all) the vertices that it can reach.
- **Connected component** is subset of vertices visited during traversal that begins at given vertex.

Depth-First Search

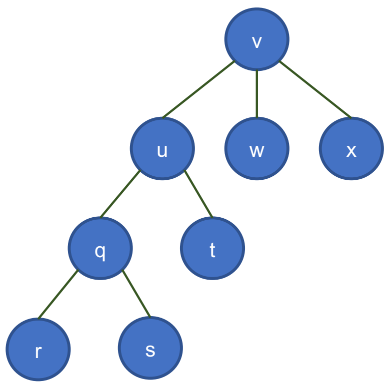
- Goes as far as possible from a vertex before backing up.

```
DFS(v: vertex)
{
      Mark v as visited
      for (each unvisited vertex u adjacent to v)
            DFS(u)
}
```

Depth-First Search

```
DFS(v: vertex)
      s = new empty stack
      s.push(v)
      Mark v as visited
      while (s is not empty)  {
            if (no unvisited vertices are adjacent to the
vertex on the top of the stack)
                  s.pop()
            else {
                  s.push(u)
                  Marked u as visited
            }
      }
```

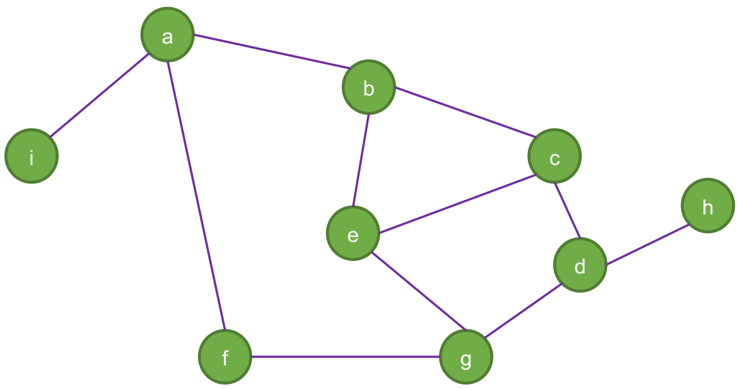

Depth-First Search



v - u - q - r - s - t - w - x

fit@hcmus

Depth-First Search



DFS starts at ***a***:
DFS starts at ***e***:

fit@hcmus

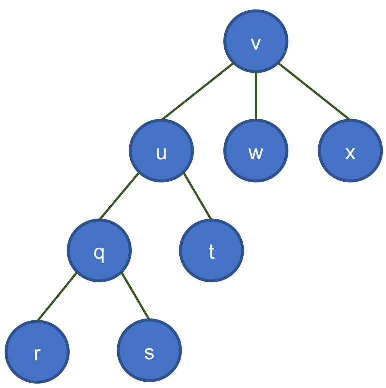
Breadth-First Search

- Visits all vertices adjacent to vertex before going forward.
- Breadth-first search uses a **queue**.

Breadth-First Search

```
BFS(v: Vertex)
     q = a new empty queue
     q.enqueue(v)
     Mark v as visited
     while (q is not empty){
          w = q.dequeue()
          for (each unvisited vertex u adjacent to w){
               Mark u as visited
               q.enqueue(u)
          }
     }
```

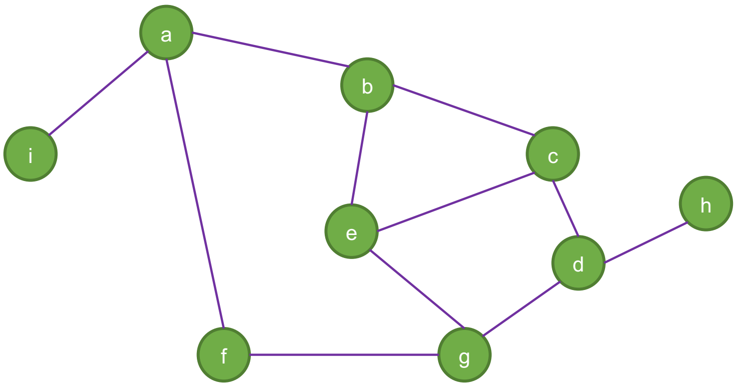

Breadth-First Search



v - u - w - x - q - t - r - s

fit@hcmus

Breadth-First Search



BFS starts at ***a***:
BFS starts at ***e***:

fit@hcmus

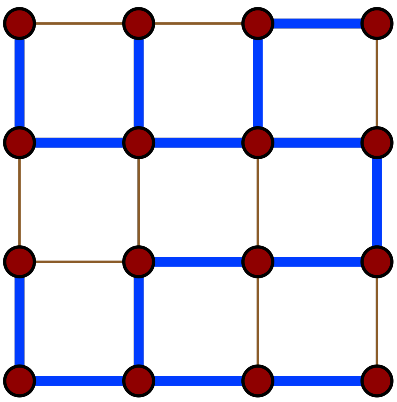
Minimum Spanning Tree

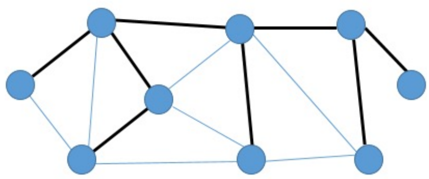
Spanning Tree

- A spanning tree
  - is a **subgraph** of undirected graph G
  - has **all** the vertices covered with **minimum** possible number of edges.

- does not have cycles
- cannot be disconnected.

Spanning Tree





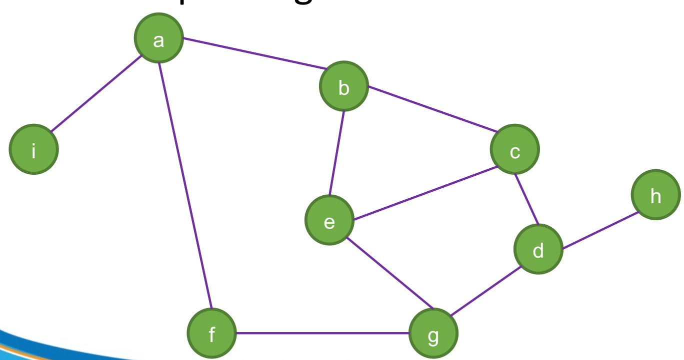
Spanning Tree

- A connected graph G can have **more than one** spanning tree.
- All possible spanning trees of graph G, **have the same** number of **edges** and **vertices**.
- The spanning tree **does not have any cycle** (loops).
- The spanning tree is **minimally connected**.
- The spanning tree is **maximally acyclic**.

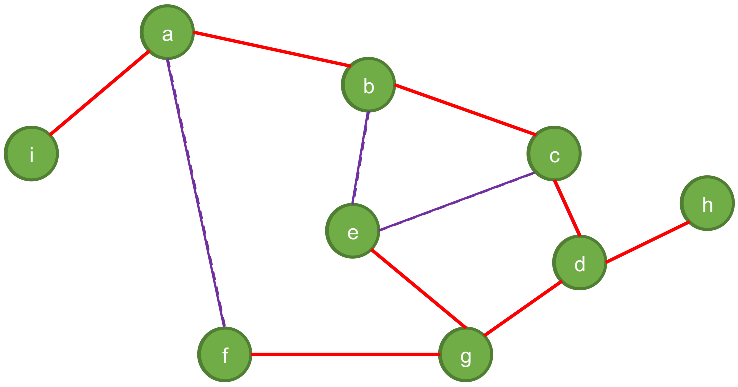
Spanning Tree

- Depth-first-search spanning tree

- Breadth-first-search spanning tree

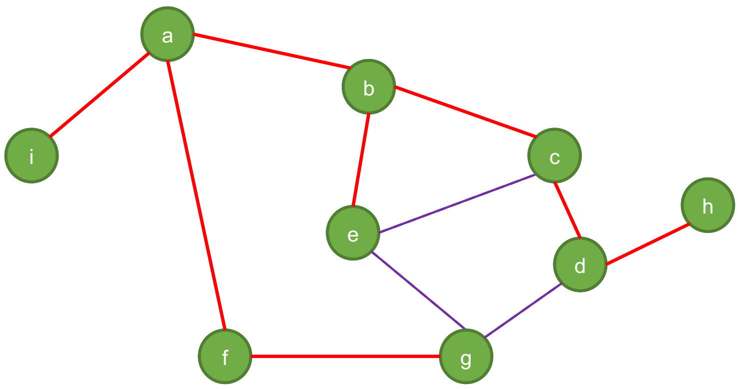


Spanning Tree



DFS spanning tree

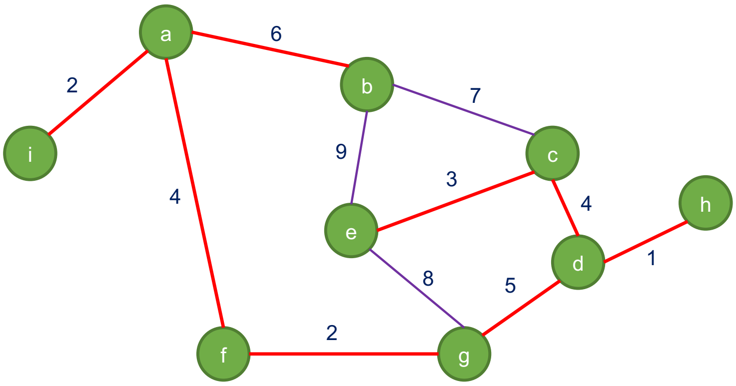
Spanning Tree



BFS spanning tree

Minimum Spanning Tree

- A minimum spanning tree is a spanning tree that has **minimum weight** than all other spanning trees of the same graph.



Prim's Minimum Spanning Tree

- Begins with any vertex.
- Initially, the tree T contains only the starting vertex.
- At each stage,
  - Select the least cost edge *e*(v, u) with *v* in T and *u* not in T.
  - Add *u* and *e* to T

Prim's Minimum Spanning Tree

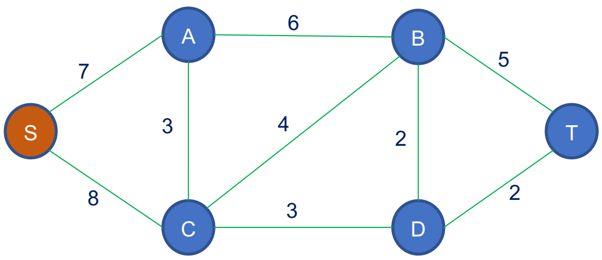
```
primAlgorithm(v: Vertex)
    Mark v as visited and include it in the minimum spanning tree
    while (there are unvisited vertices)
    {
          Find the least-cost edge e(v, u) from a visited vertex
                v to some unvisited vertex u
          Mark u as visited
          Add the vertex u and the edge e(v, u) to the minimum
                spanning tree
    }
```

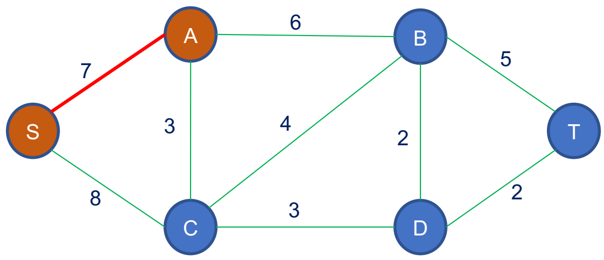

fit@hcmus

Prim's Minimum Spanning Tree

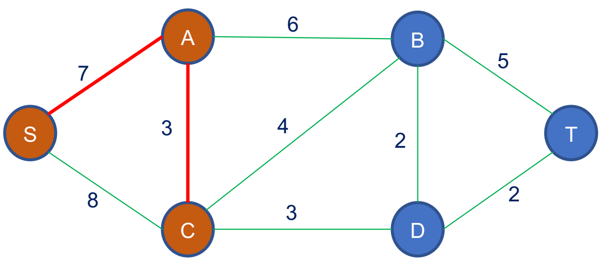
```
PrimSpanningTree(matrix[N][N], source)
{
      for v = 0 to N-1 {
            length[v] = matrix[source][v]
            parent[v] = source }
      Mark source //Add source to the spanning tree
      for step = 1 to N-1 {
            Find the vertex v such that length[v] is smallest and v
                  is not in spanning tree
            Mark v
            for all vertices u not in vertexSet
                  if (length[u] > matrix[v][u]){
                        length[u] = matrix[v][u]
                        parent[u] = v }
      }
}
```

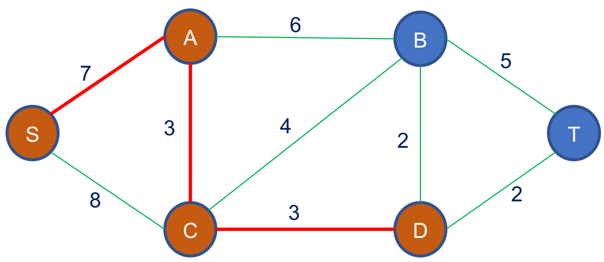
Prim’s Minimum Spanning Tree



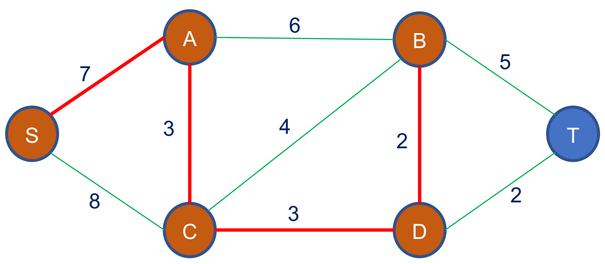


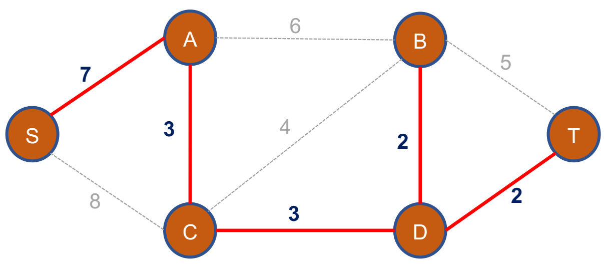
Prim’s Minimum Spanning Tree



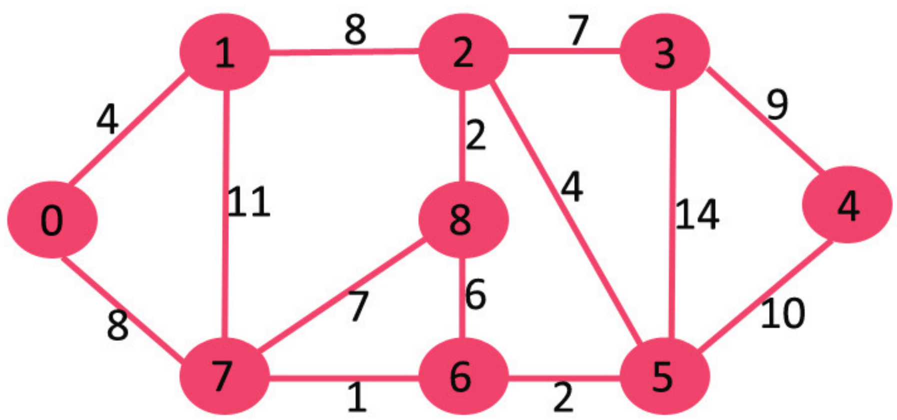


Prim's Minimum Spanning Tree

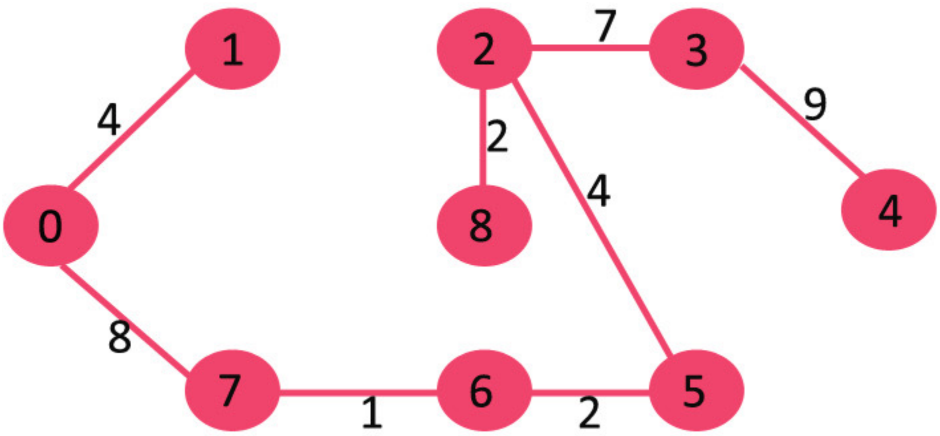




Example



Example



Shortest Path

Dijkstra's Shortest Path Algorithm

- Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.
- **Dijkstra's** algorithm is very **similar** to **Prim's** algorithm for minimum spanning tree.
- This algorithm is applicable to graphs with **non-negative weights** only.

Dijkstra's Shortest Path Algorithm

shortestPath(matrix[N][N], source, length[])

Input:

matrix[N][N]: adjacency matrix of Graph *G* with *N* vertices

source: the *source* vertex

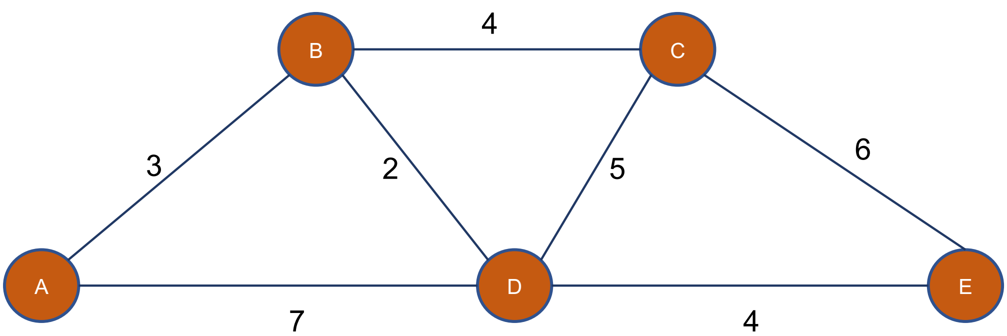
Output:

length[]: the length of the shortest path from *source* to all *vertices* in G.

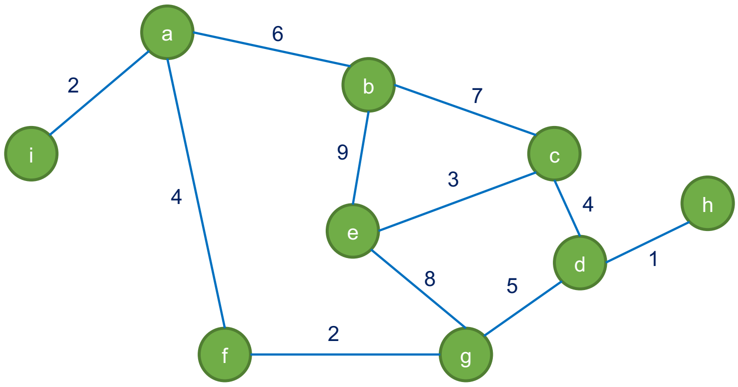
Dijkstra’s Shortest Path Algorithm

```
shortestPath(matrix[N][N], source, length[]){
      for v = 0 to N-1
            length[v] = matrix[source][v]
      length[source] = 0 //why?
      for step = 1 to N {
            Find the vertex v such that length[v] is smallest and
                  v is not in vertexSet
            Add v to vertexSet
            for all vertices u not in vertexSet
                  if (length[u] > length[v] + matrix[v][u]){
                        length[u] = length[v] + matrix[v][u]
                        parent[u] = v }
      }
}
```


Example



Example



Questions and Answers