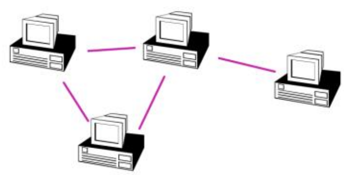
Graph

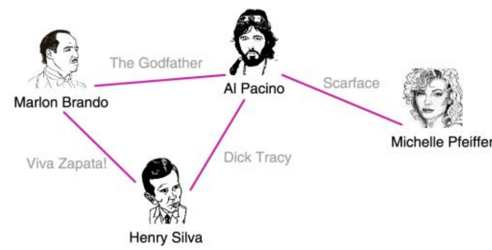
Contents

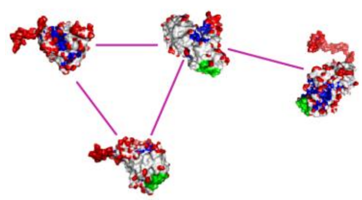
- Terminologies
- Graph representation
- Graph traversal
- Spanning tree
- Shortest path

fit@hcmus

Examples







fit@hcmus

Networks or Graphs

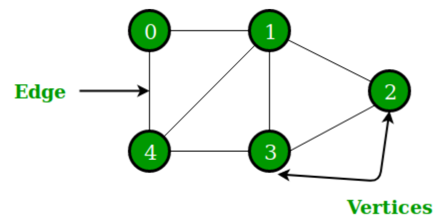
- **network** often refers to ***real systems***
  - www,
  - social network,
  - metabolic network.

- **graph**: mathematical representation of a ***network***
  - web graph,
  - social graph (a Facebook term)

fit@hcmus

Graph

- A graph consists of **a finite set of vertices** (or nodes) and **set of edges** which connect a pair of vertices (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}



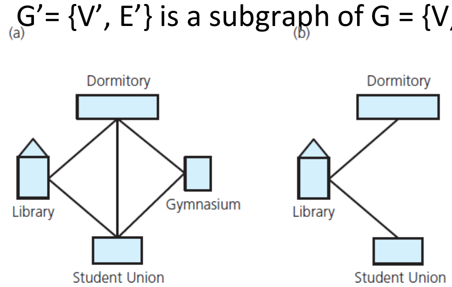
fit@hcmus

Terminologies

Terminologies

fit@hcmus

- 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

Terminologies

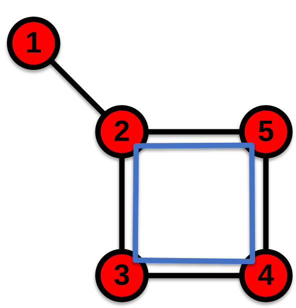
fit@hcmus

- **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.

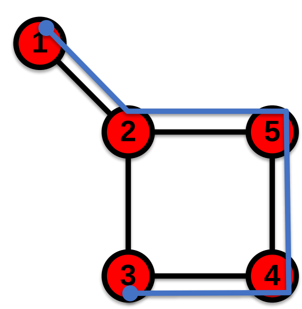
Terminologies

- **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



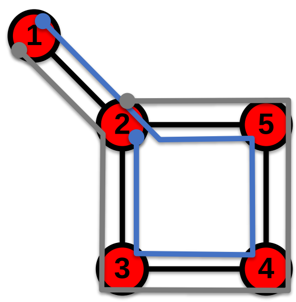
Cycle: A path with the same start and end node.



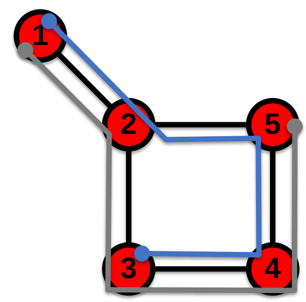
A path that does not intersect itself.

Terminologies

- **Eulerian Path**: A path that traverses each ***edge*** exactly once.
- **Hamiltonian Path**: A path that visits each ***vertex*** exactly once.



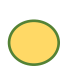
Eulerian Path



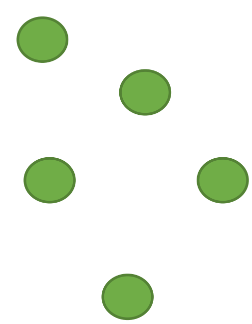
Hamiltonian Path

Terminologies

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



trivial graph



null graph

Terminologies

fit@hcmus

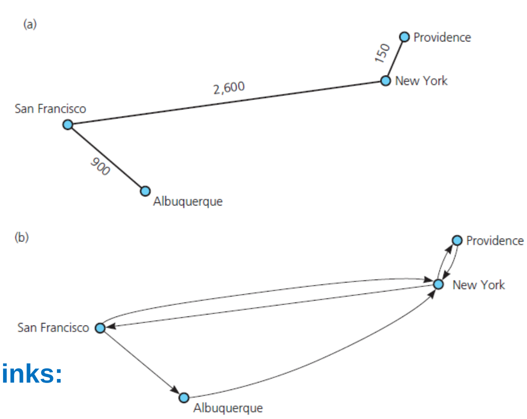
- **Undirected graph**: the graph in which edges do not indicate a direction.
- **Directed graph, or digraph**: a graph in which each edge has a direction.
- **Weighted graph**: a graph with numbers (weights, costs) assigned to its edges.

Terminologies

fit@hcmus

(a): undirected graph

(b): directed graph



Undirected direction/links:
- Co-authorship links
- Actor network
- Protein interactions

Directed directions/links:
- URLs on the www
- Phone calls
- Metabolic reactions

fit@hcmus

NETWORK	NODES	LINKS	DIRECTED UNDIRECTED	N	L
Internet	Routers	Internet connections	Undirected	192,244	609,066
WWW	Webpages	Links	Directed	325,729	1,497,134
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594
Mobile Phone Calls	Subscribers	Calls	Directed	36,595	91,826
Email	Email addresses	Emails	Directed	57,194	103,731
Science Collaboration	Scientists	Co-authorship	Undirected	23,133	93,439
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908
Citation Network	Paper	Citations	Directed	449,673	4,689,479
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930

16

fit@hcmus

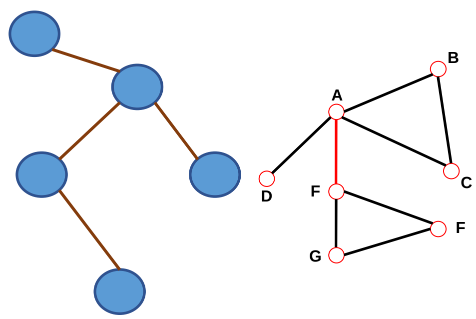
Terminologies

- **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.
- Graph cannot have duplicate edges between vertices.
  - **Multigraph**: does allow multiple edges

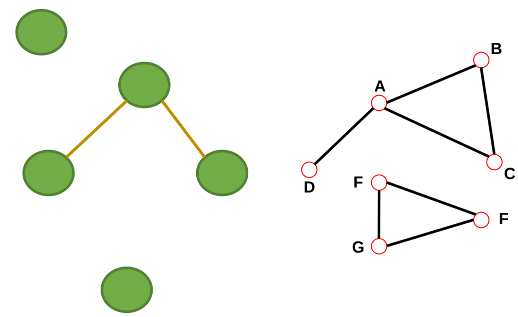
17

fit@hcmus

Terminologies



connected graph

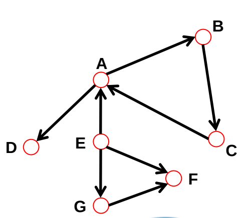


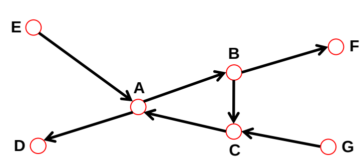
disconnected graph

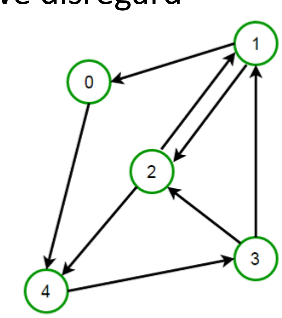
fit@hcmus

Terminologies

- **Strongly connected** directed graph: if there is a path in each direction between each pair of vertices of the graph.
- **Weakly connected** directed graph: it is connected if we disregard the edge directions.



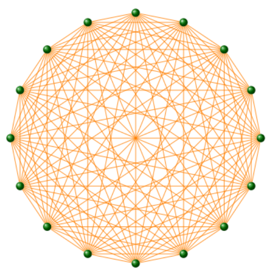


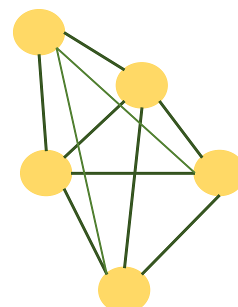


Terminologies

- **Complete graph:** A graph in which each pairs of **distinct vertices** has an **edge** between them

The maximum number of edges a graph *N* vertices can have?





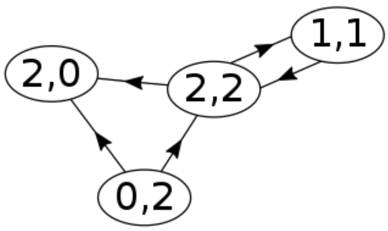
Terminologies

- **Degree** of a vertex *v* (denoted *deg(v)*): the number of edges connected to *v*.
- In directed graphs, we can define an *in-degree* and *out-degree* of vertex *v*.
  - **In-degree** of *v* (denoted $deg^{-}(v)$): number of head ends adjacent to *v*.
  - **Out-degree** of *v* (denoted $deg^{+}(v)$): number of tail ends adjacent to *v*.

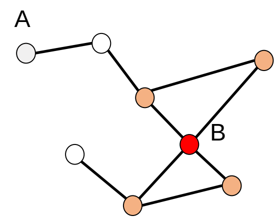
  $deg(v) = deg^{-}(v) + deg^{+}(v)$
- Note:
  - arc(*x*, *y*): direction from *x* to *y*. *x* is called *tail* and *y* is called *head* of the arc.

fit@hcmus

Terminologies



A directed graph with vertices labeled (indegree, outdegree)



deg(A) = 1; deg(B) = 4

fit@hcmus

Terminologies

- Let $G = \{V, E\}$
- If G is an undirected graph

$$\sum_{v \in V} \deg(v) = 2|E|$$

- If G is a directed graph

$$\sum_{v \in V} deg^{-}(v) = \sum_{v \in V} deg^{+}(v) = |E|$$

Graph Representation

Graph Representation

- Adjacency Matrix
- Adjacency List

fit@hcmus

Adjacency Matrix

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

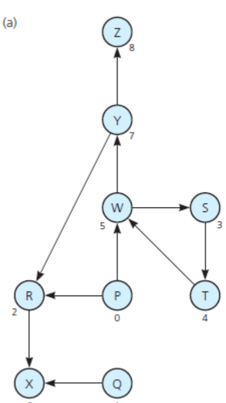
- $A[i][j] = \begin{cases} 1 \; if \; there \; is \; an \; edge(i,j) \\ 0 \; if \; there \; is \; no \; edge \; (i,j) \end{cases}$

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

fit@hcmus

Adjacency Matrix

(a)

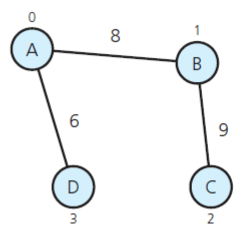


(b)

		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)



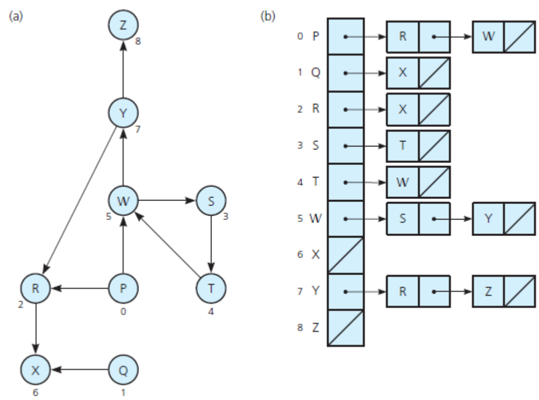
(b)

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

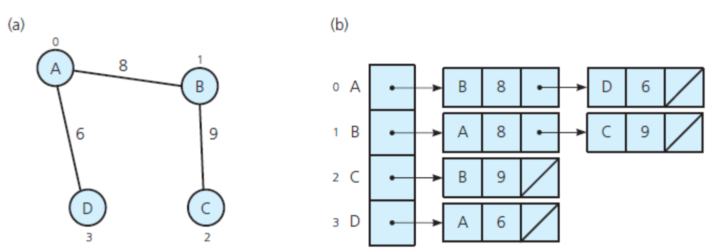
Adjacency List

- 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



fit@hcmus

Relative Advantages of Adjacency Lists and Matrices

- Faster to test if (x, y) in graph?
- Faster to find the degree of a vertex?
- Less memory on small graph?
- Less memory on big graph?
- Edge insertion or deletion?
- Faster to traverse the graph?
- Better for most problems?

fit@hcmus

Graph Traversal

Graph Traversal

fit@hcmus

- 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

fit@hcmus

- 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)
}
```

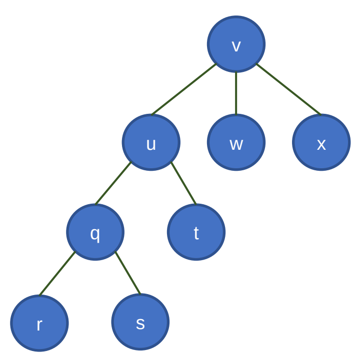
fit@hcmus

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
            }
      }
```

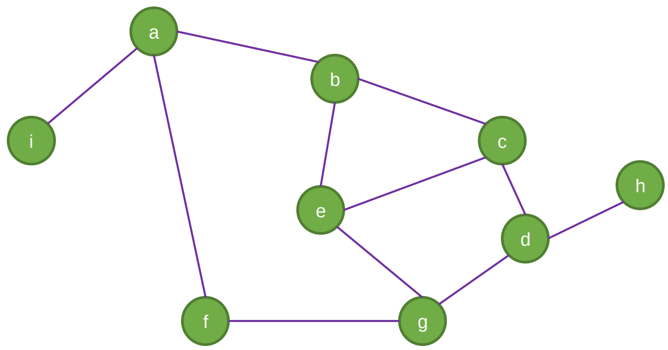

fit@hcmus

Depth-First Search



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

Depth-First Search



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

41

Breadth-First Search

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

42

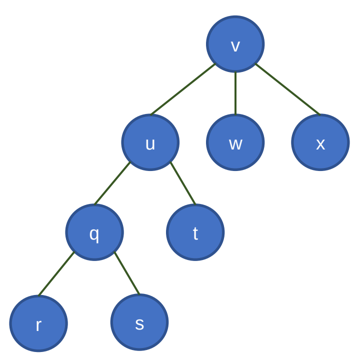
fit@hcmus

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)
        }
    }
```


fit@hcmus

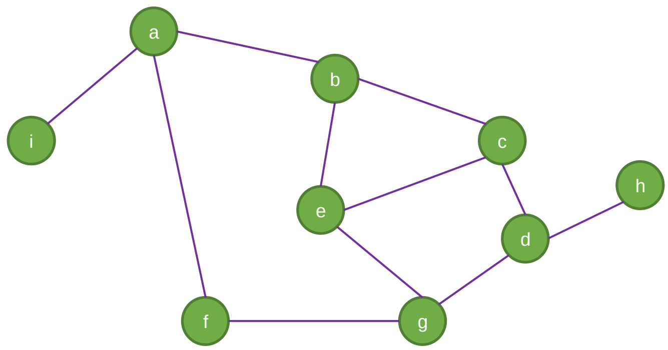
Breadth-First Search



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

fit@hcmus

Breadth-First Search



BFS starts at ***a***:

BFS starts at ***e***:

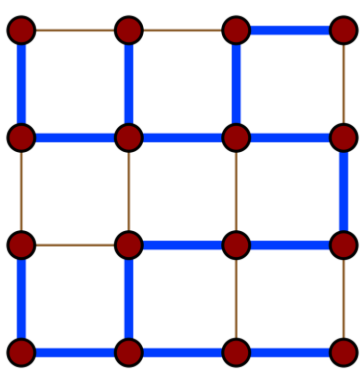
fit@hcmus

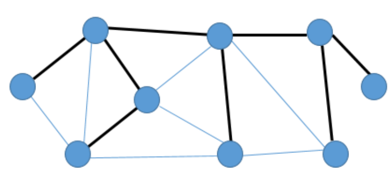
Minimum Spanning Tree

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





fit@hcmus

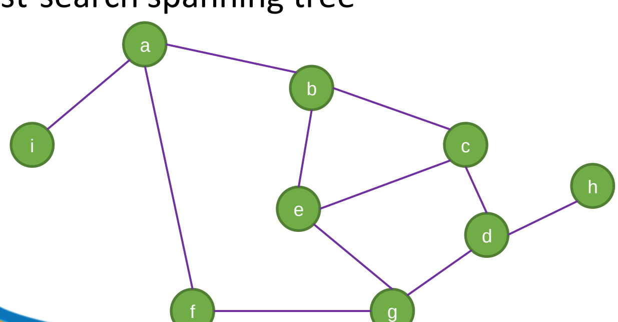
Spanning Tree

- 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**.

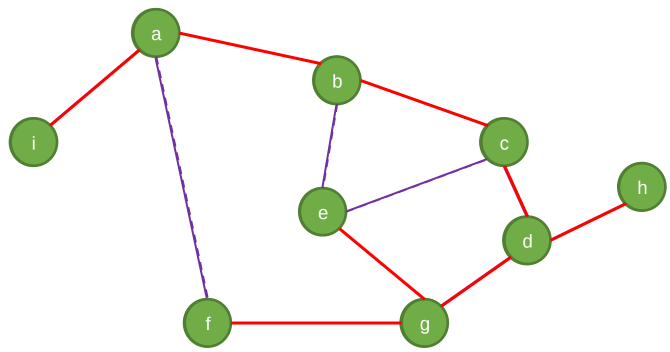
fit@hcmus

Spanning Tree

- Depth-first-search spanning tree
- Breadth-first-search spanning tree

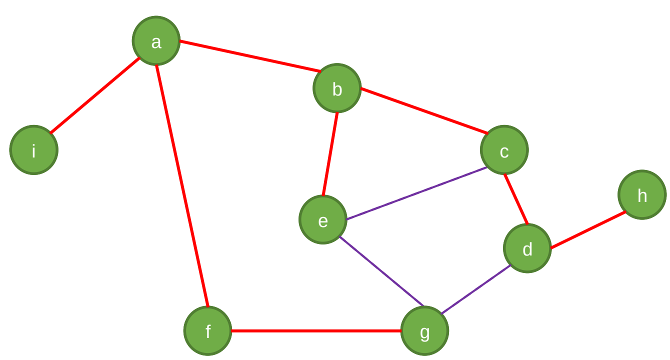


Spanning Tree



DFS spanning tree

Spanning Tree

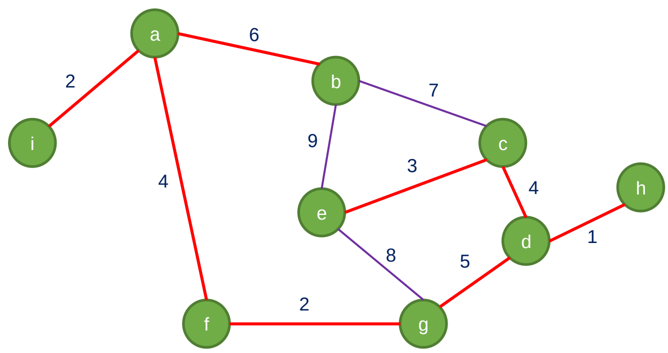


BFS spanning tree

Minimum Spanning Tree

fit@hcmus

- 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

fit@hcmus

- Begins with any vertex *s*.
- Initially, the tree *T* contains only the starting vertex *s*.
- 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

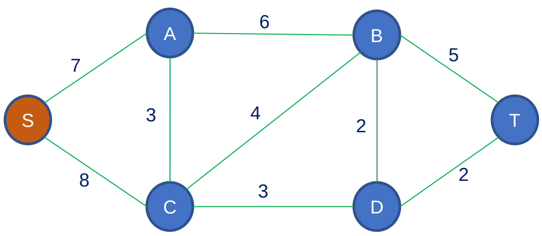
```
primAlgorithm(v: Vertex)
    Mark v as visited and include it in the minimum spanning tree
    while (there are unvisited vertices)
    {
        Find the least-cost edge e(v, u) from a visited vertex
            v to some unvisited vertex u
        Mark u as visited
        Add the vertex u and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

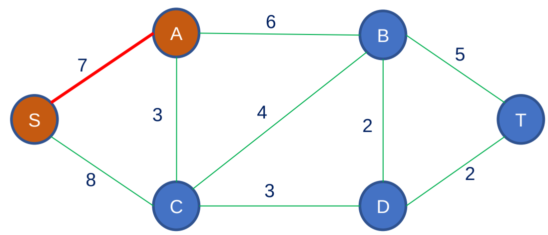

Prim's Minimum Spanning Tree

```
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

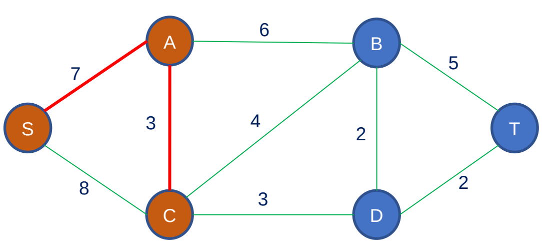
fit@hcmus

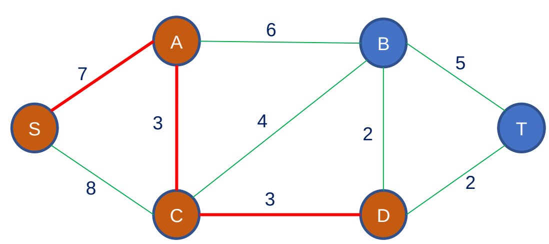




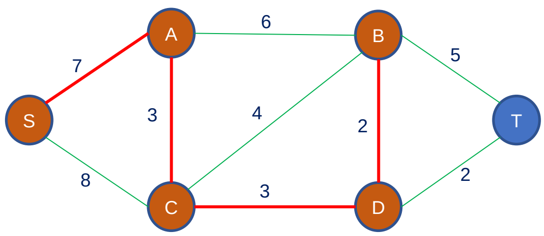
Prim’s Minimum Spanning Tree

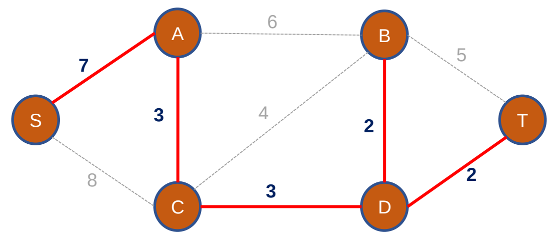
fit@hcmus





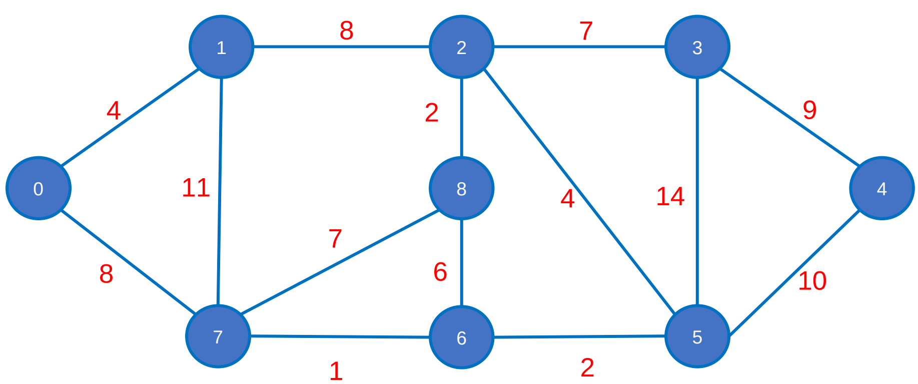
Prim's Minimum Spanning Tree





61

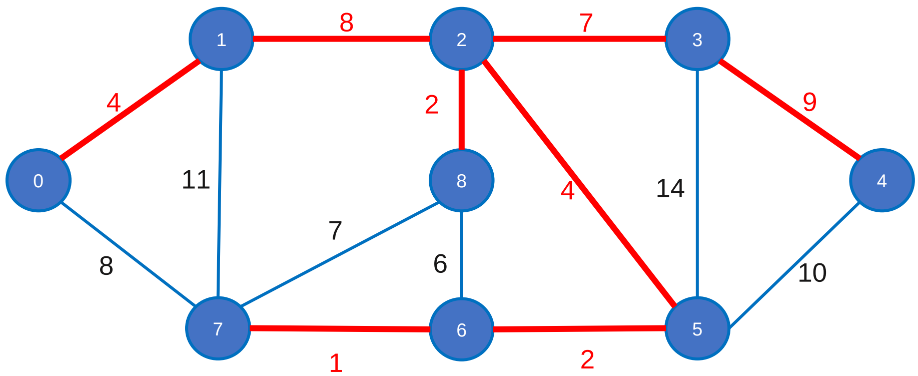
Example



62

fit@hcmus

Example



63

fit@hcmus

Shortest Path

68

Dijkstra's Shortest Path Algorithm

- Given a graph and a source vertex in the graph, find shortest paths from the 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.
```

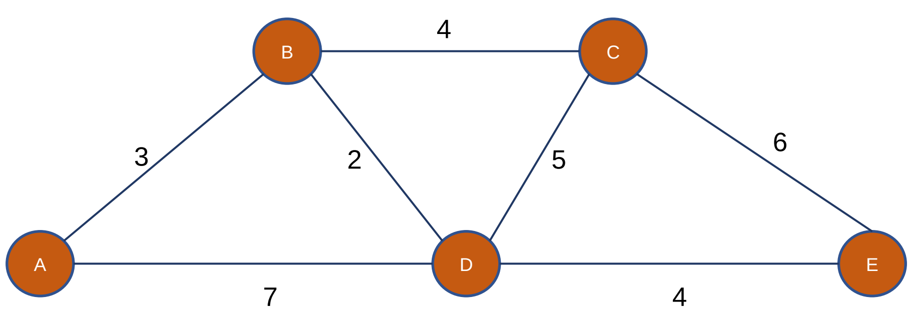
fit@hcmus

Dijkstra's Shortest Path Algorithm

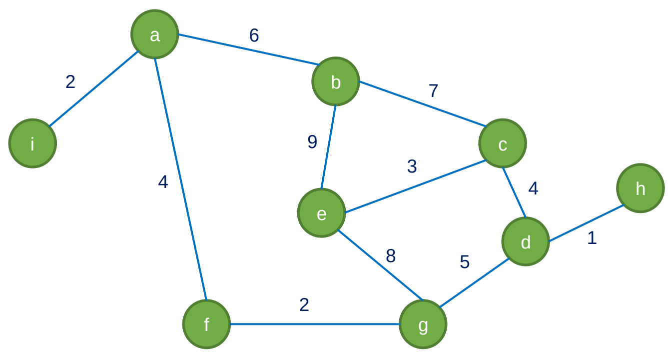
```
shortestPath(matrix[N][N], source, length[]){
	for v = 0 to N-1
		length[v] = matrix[source][v]
	length[source] = 0 //why?
	for step = 1 to N {
		Find the vertex v such that length[v] is smallest and
			v is not in vertexSet
		Add v to vertexSet
		for all vertices u not in vertexSet
			if (length[u]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


fit@hcmus

Example



Example



Questions and Answers