Graphs

Data structures

Fall 2018

Graphs

- G = (V,E)
- V is the vertex set.
- Vertices are also called nodes and points.
- E is the edge set.
- Each edge connects two different vertices.
- Edges are also called arcs and lines.
- Directed edge has an orientation (u,v).

$u \longrightarrow v$

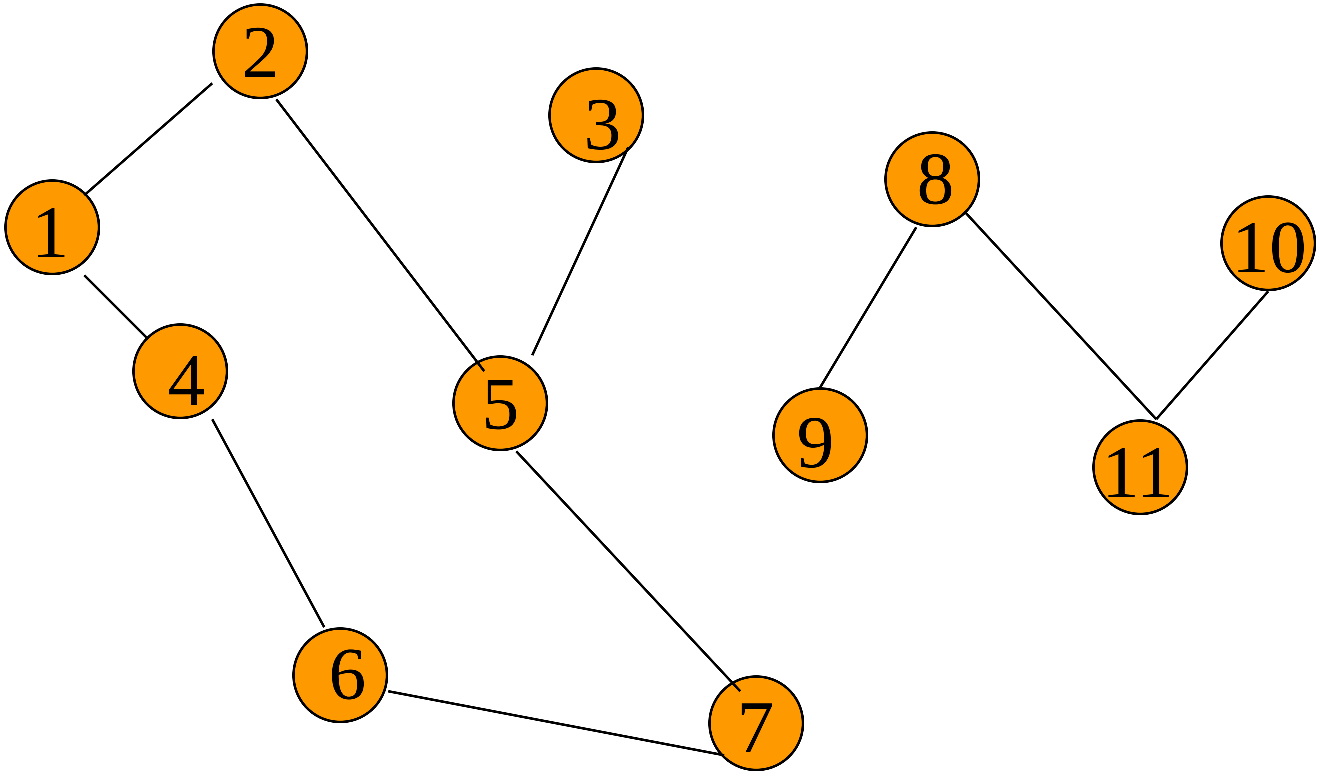
Graphs

- Undirected edge has no orientation (u,v).

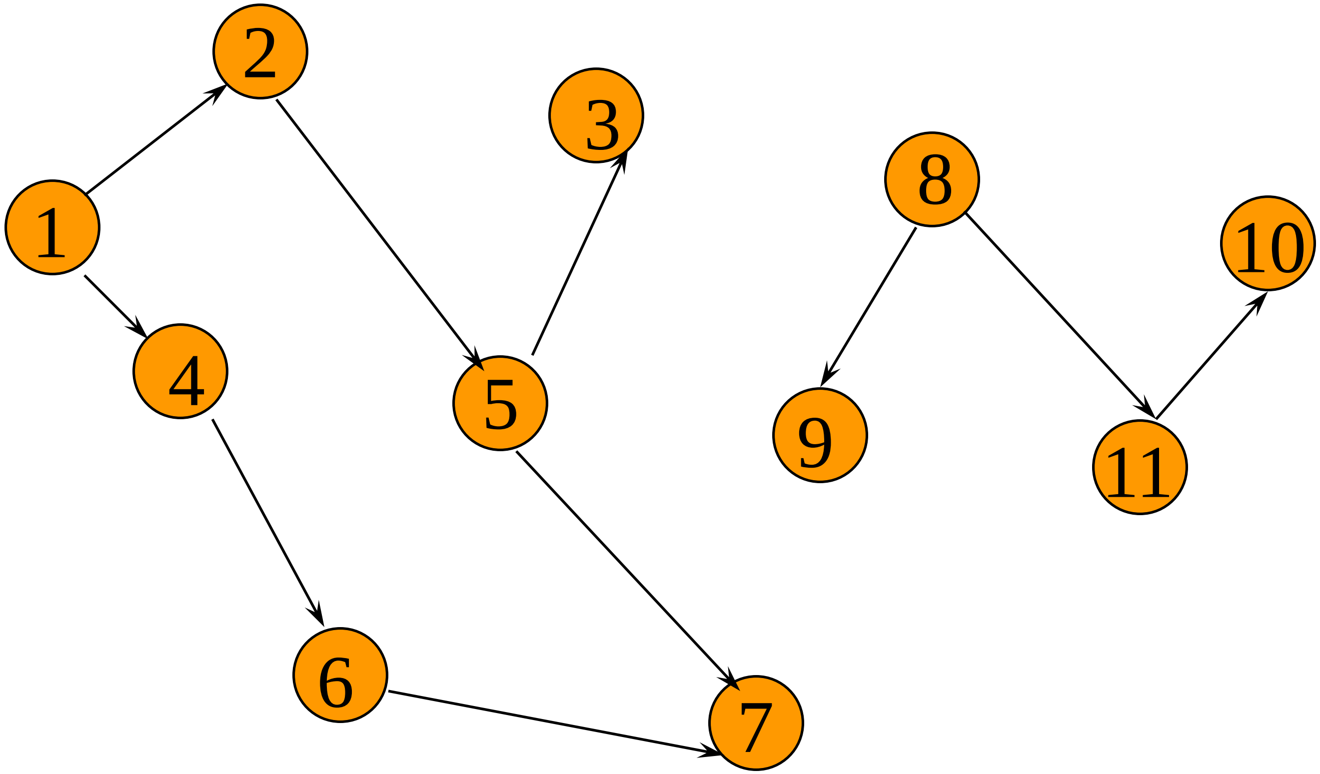
  u ——— v

- Undirected graph => no oriented edge.
- Directed graph => every edge has an orientation.

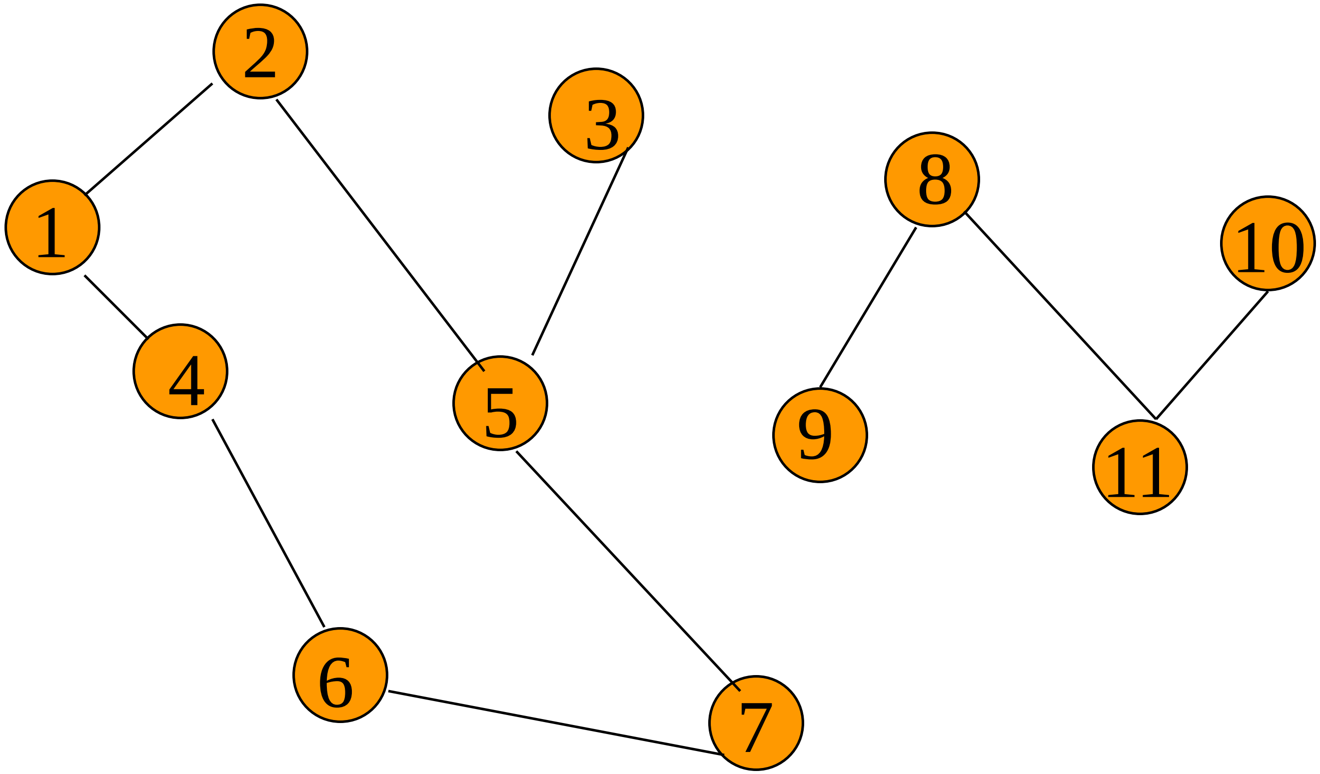
Undirected Graph



Directed Graph (Digraph)

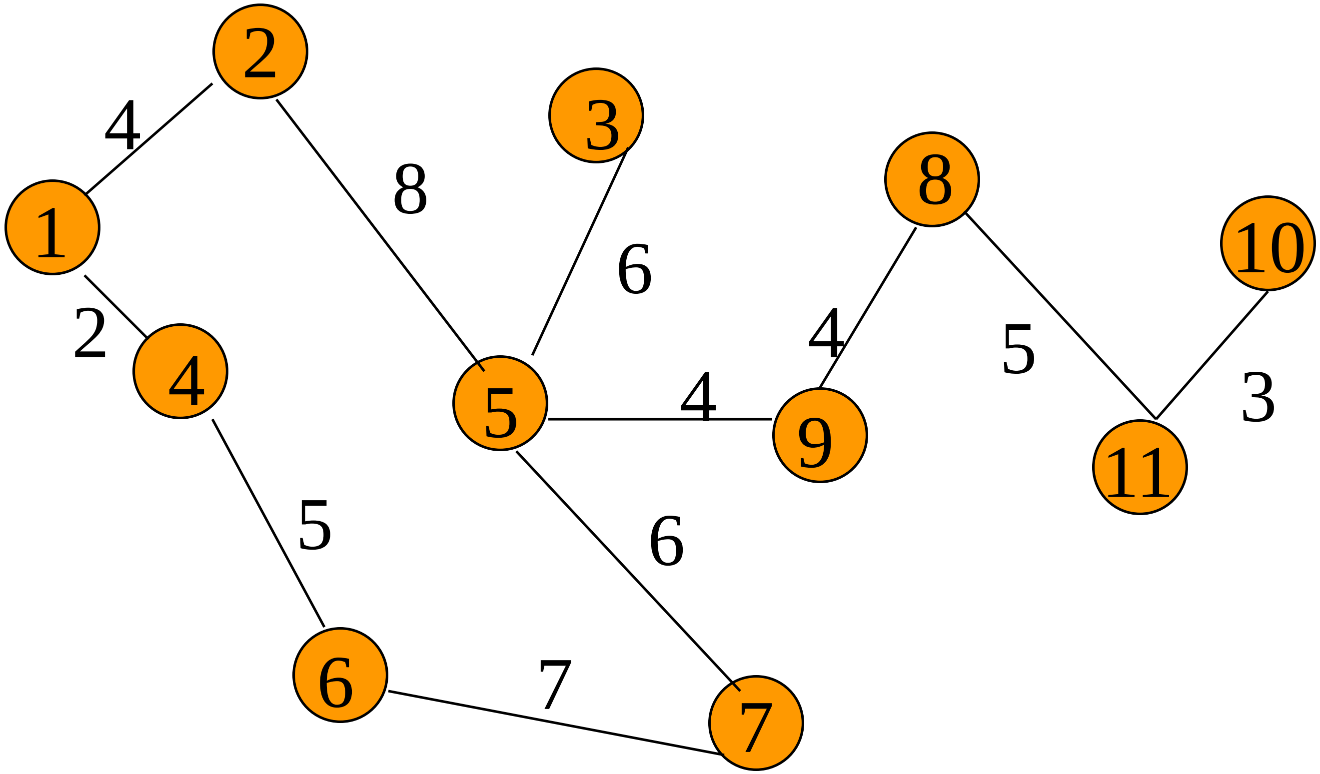


Applications—Communication Network



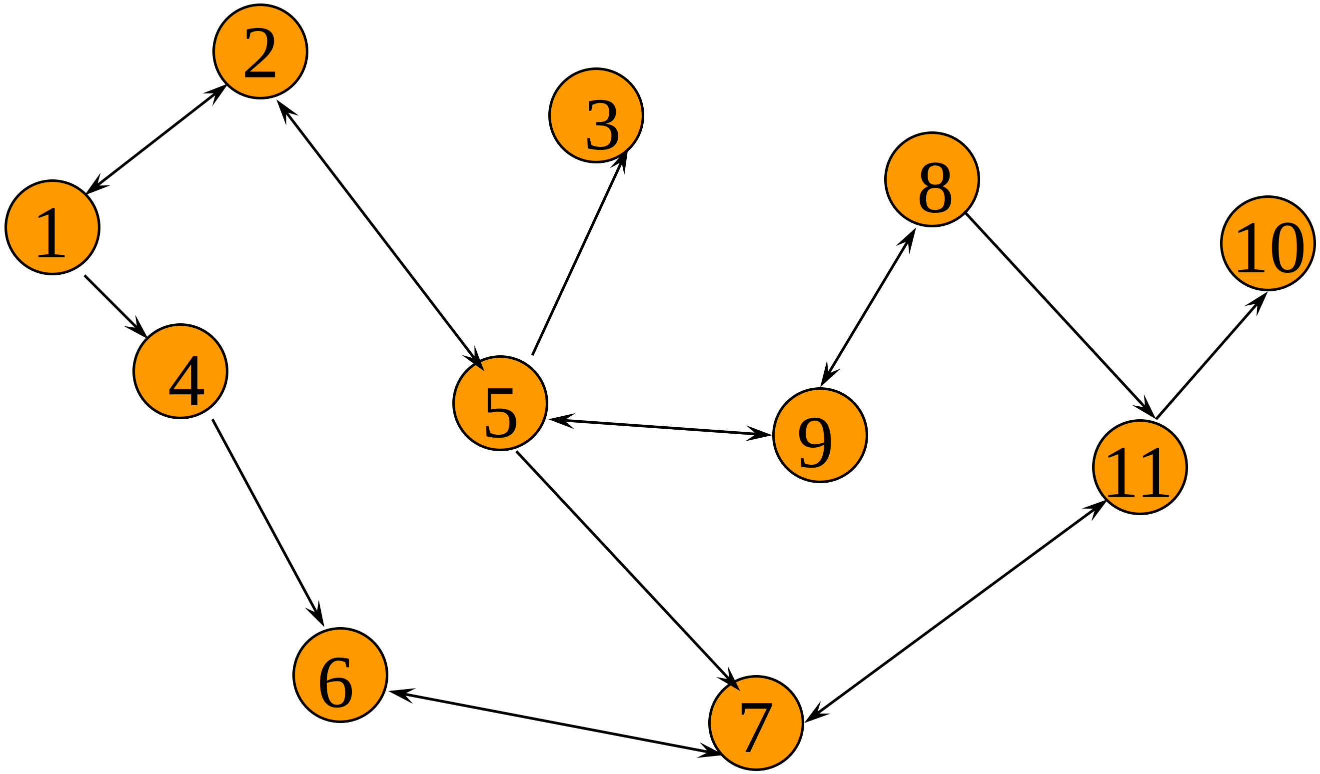
- Vertex = city, edge = communication link.

Driving Distance/Time Map



- Vertex = city, edge weight = driving distance/time.

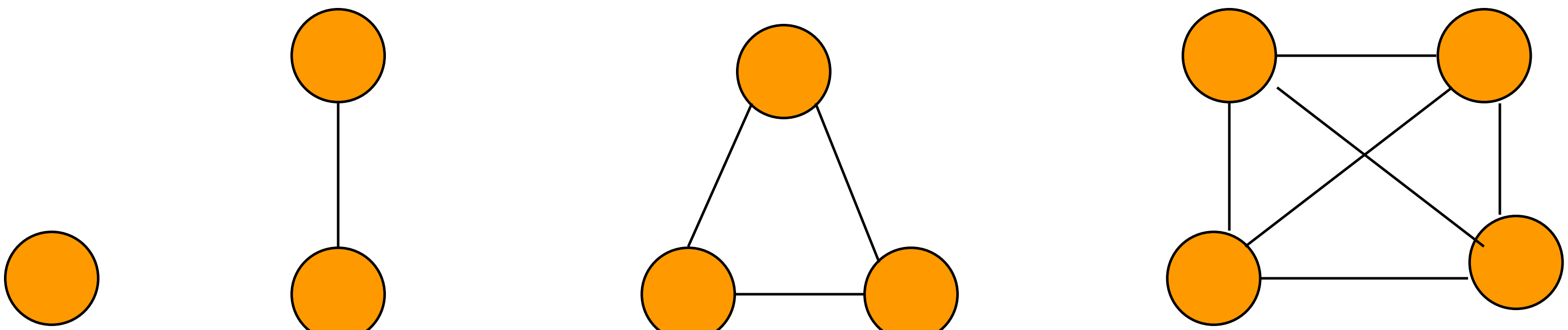
Street Map



- Some streets are one way.

Complete Undirected Graph

Has all possible edges.



n = 1 n = 2 n = 3 n = 4

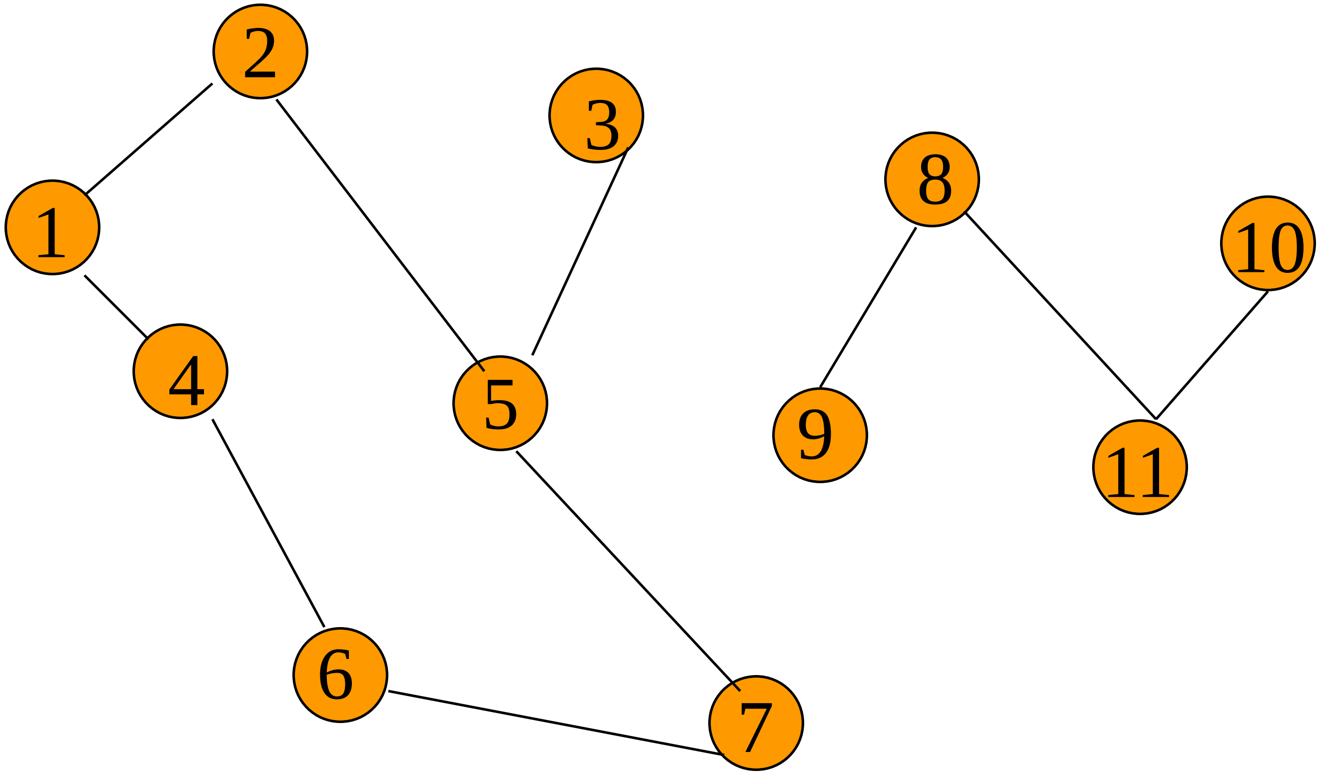
Number Of Edges—Undirected Graph

- Each edge is of the form (u,v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u,v) is the same as edge (v,u), the number of edges in a complete undirected graph is n(n-1)/2.
- Number of edges in an undirected graph is <= n(n-1)/2.

Number Of Edges—Directed Graph

- Each edge is of the form (u,v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u,v) is not the same as edge (v,u), the number of edges in a complete directed graph is n(n-1).
- Number of edges in a directed graph is <= n(n-1).

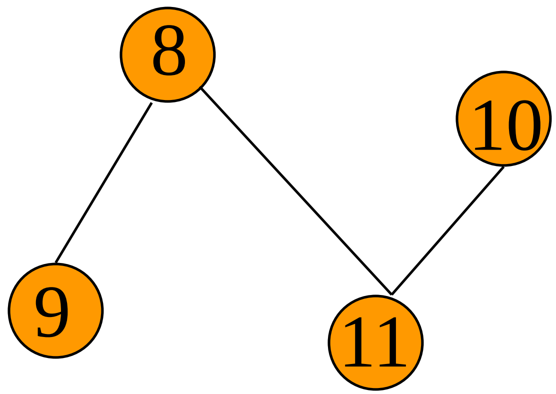
Vertex Degree



Number of edges incident to vertex.

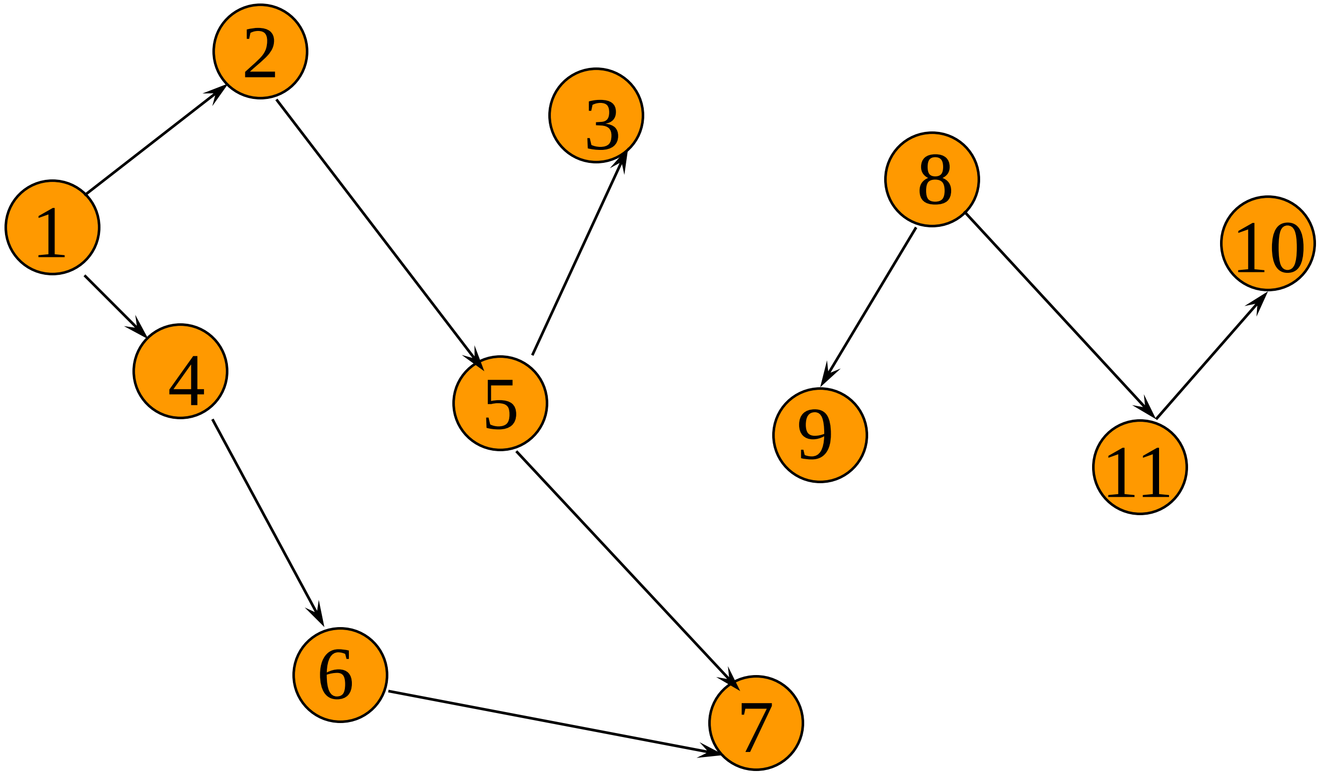
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



Sum of degrees = 2e (e is number of edges)

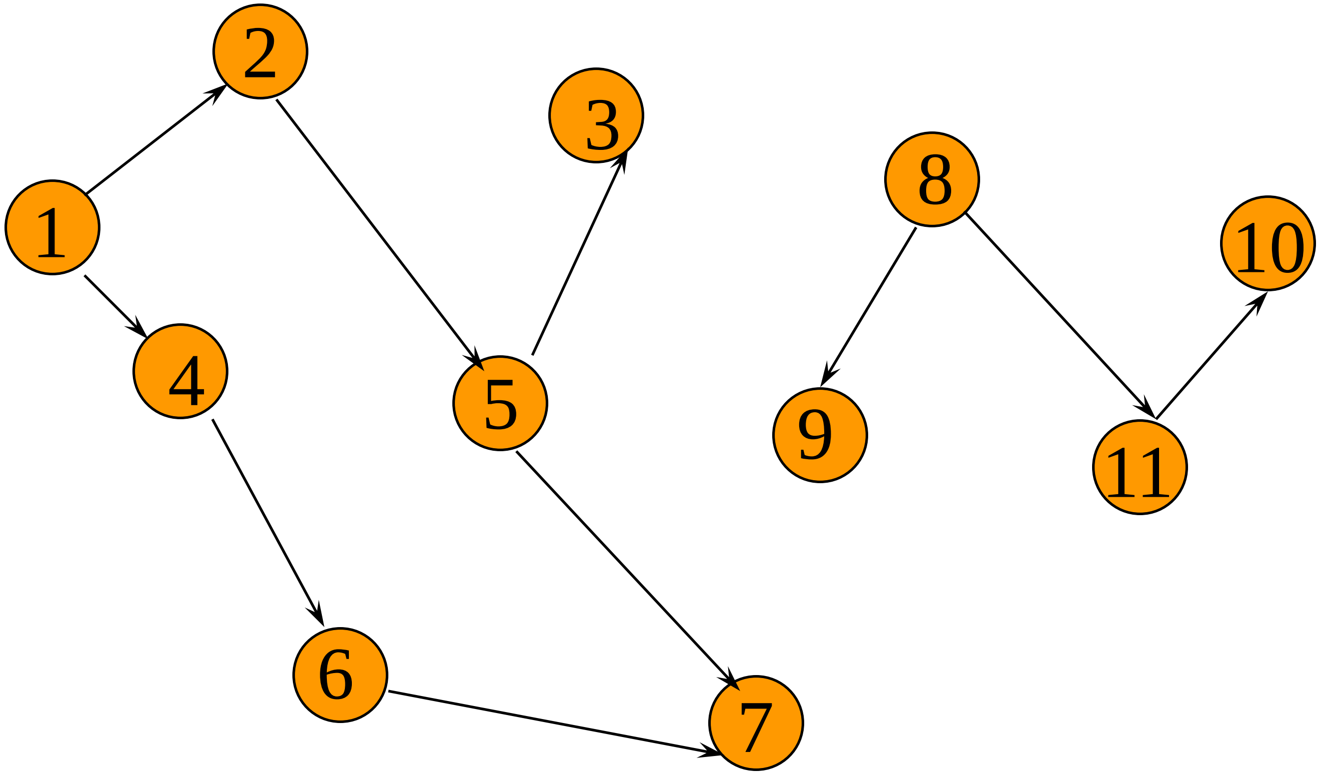
In-Degree Of A Vertex



in-degree is number of incoming edges

indegree(2) = 1, indegree(8) = 0

Out-Degree Of A Vertex



out-degree is number of outbound edges

outdegree(2) = 1, outdegree(8) = 2

Sum Of In- And Out-Degrees

each edge contributes 1 to the in-degree of some vertex and 1 to the out-degree of some other vertex

sum of in-degrees = sum of out-degrees = e,

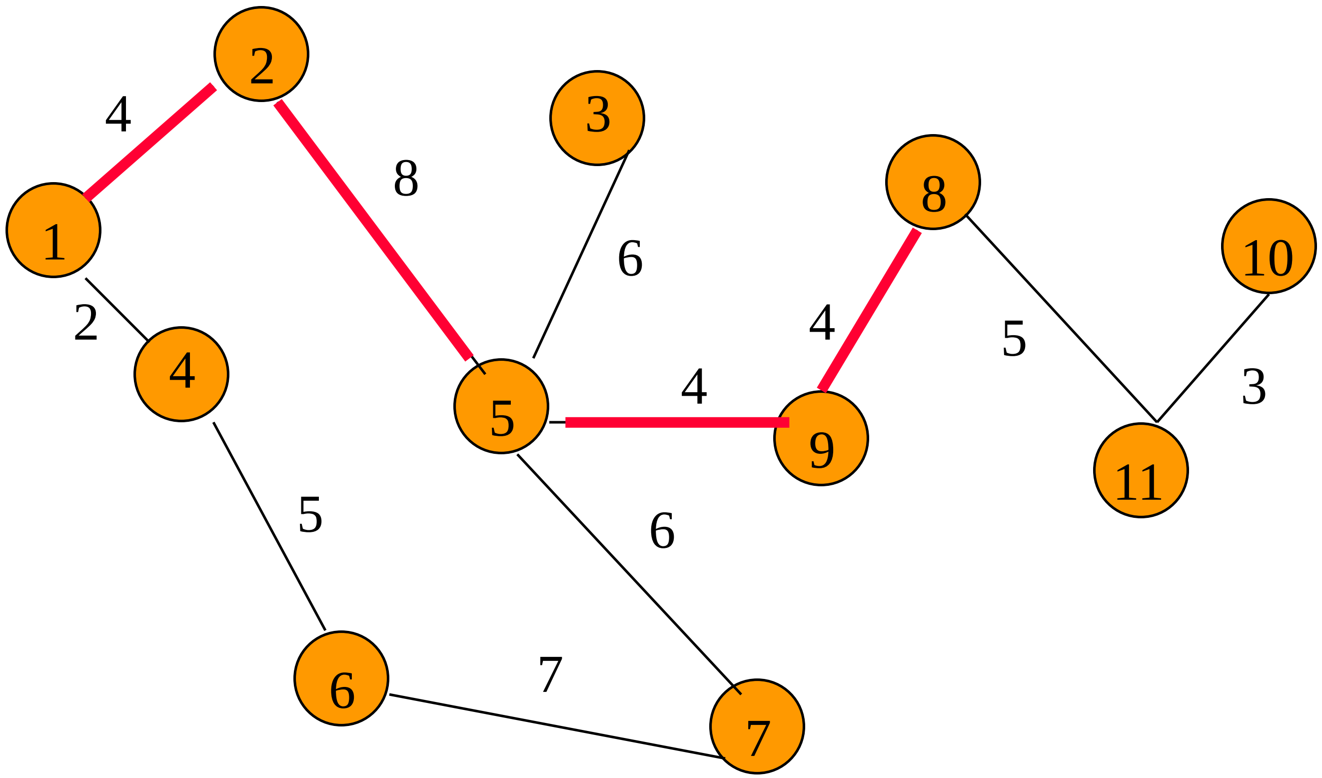
where e is the number of edges in the digraph

Sample Graph Problems

- Path problems.
- Connectedness problems.
- Spanning tree problems.

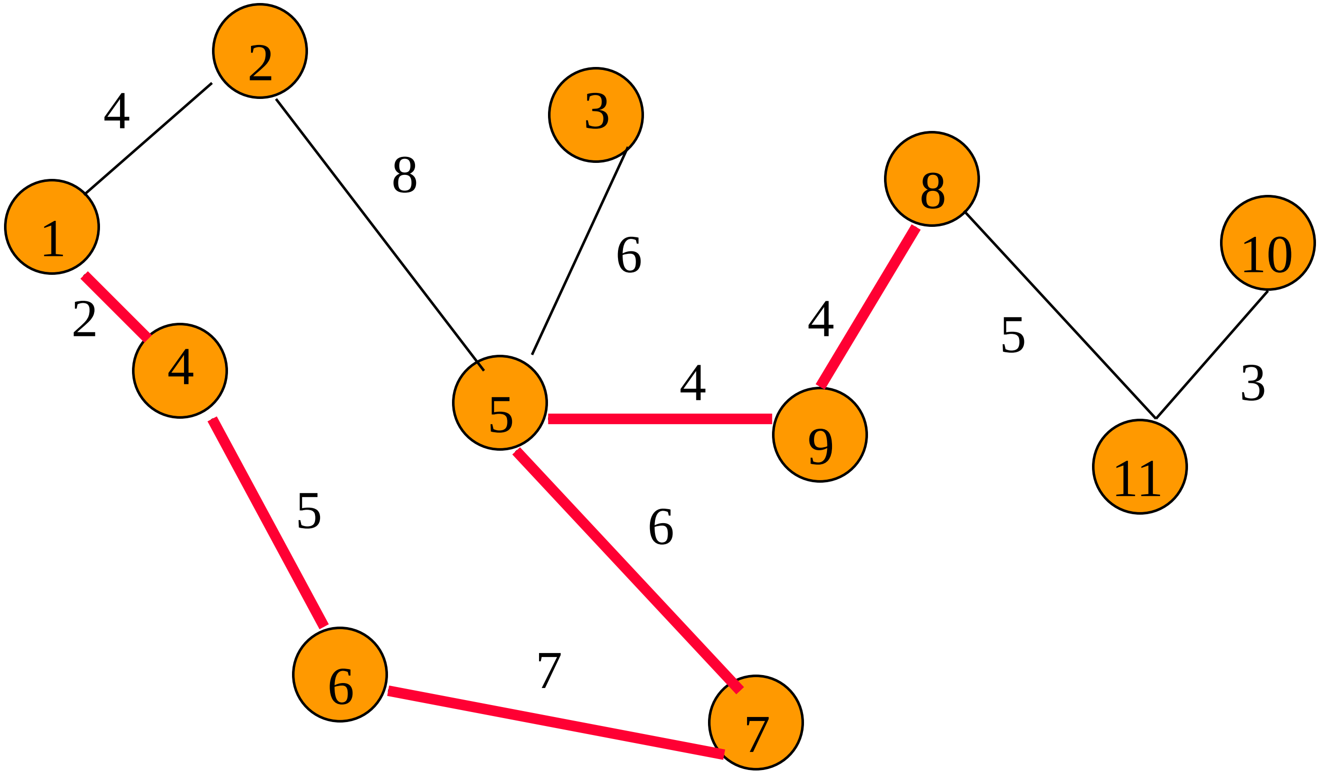
Path Finding

Path between 1 and 8.



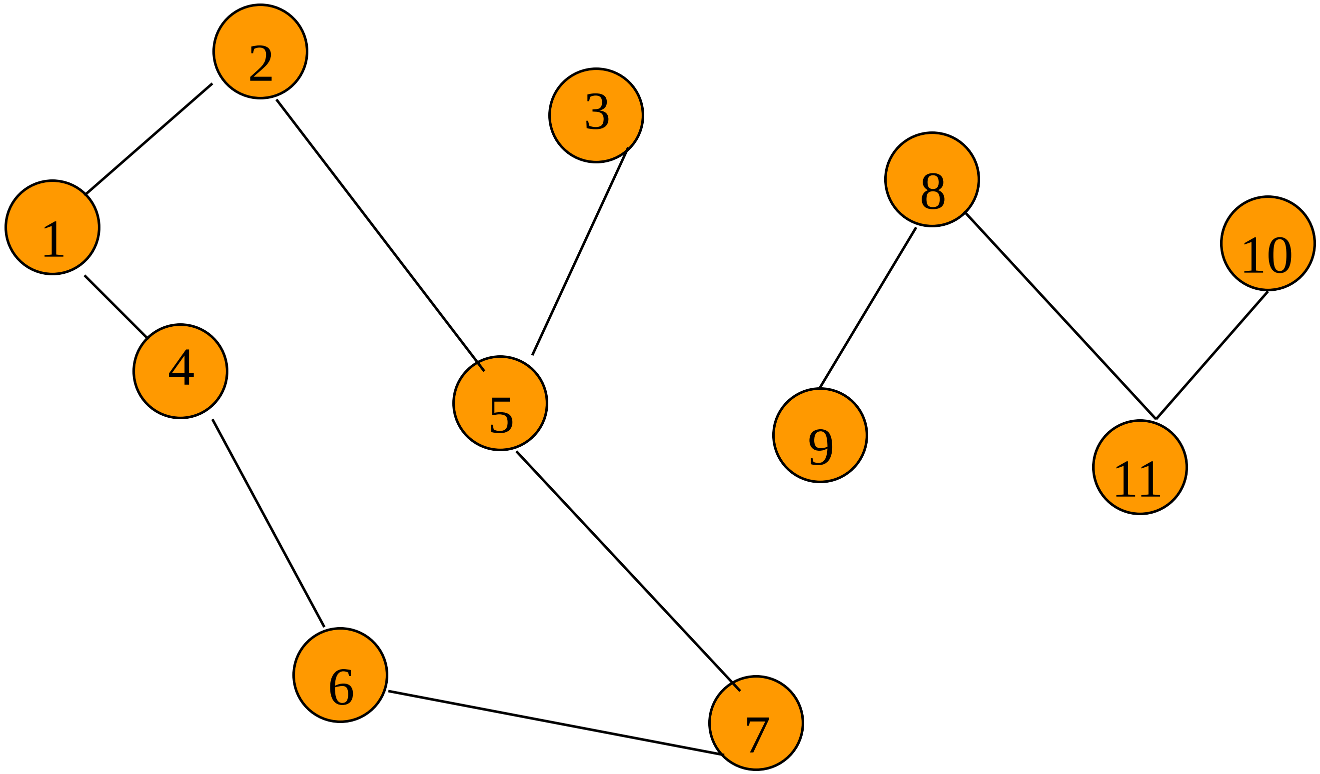
Path length is 20.

Another Path Between 1 and 8



Path length is 28.

Example Of No Path

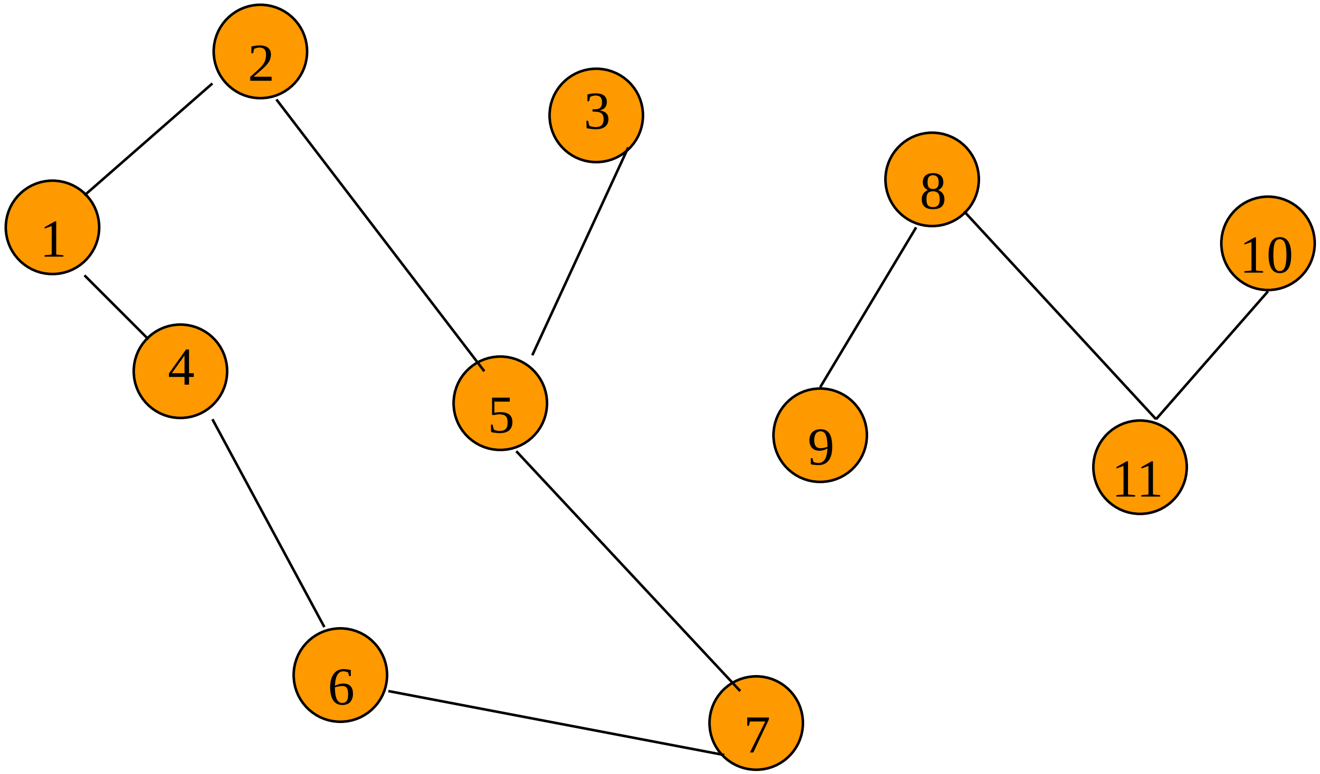


No path between 2 and 9.

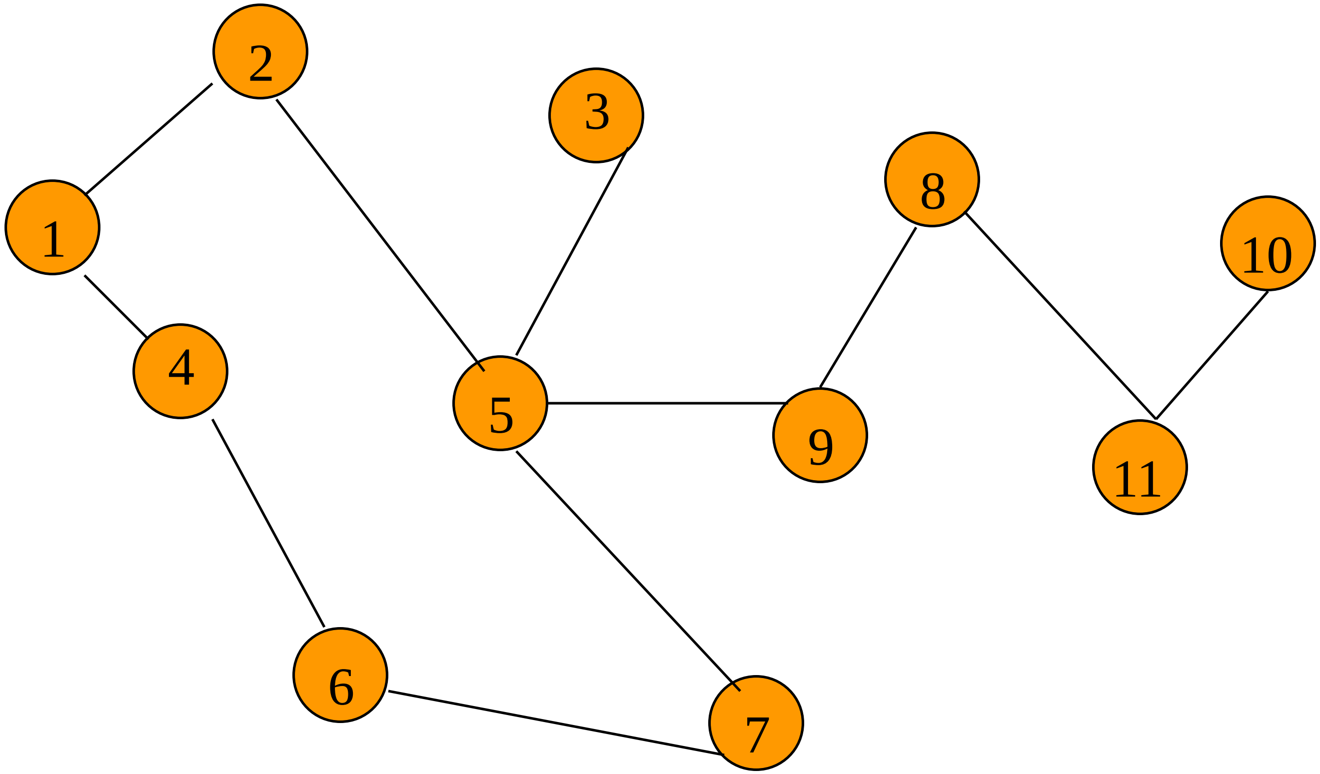
Connected Graph

- Undirected graph.
- There is a path between every pair of vertices.

Example Of Not Connected



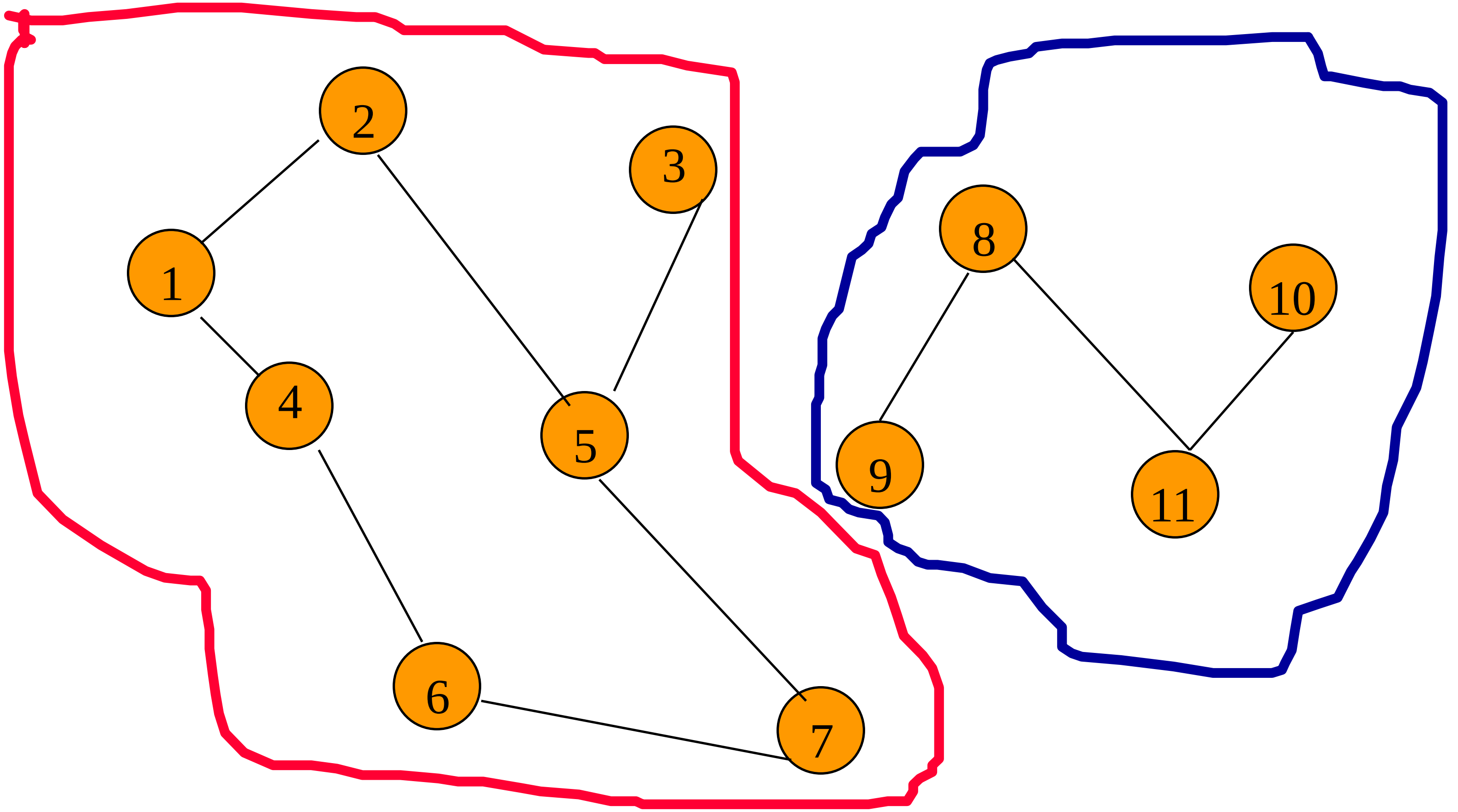
Connected Graph Example



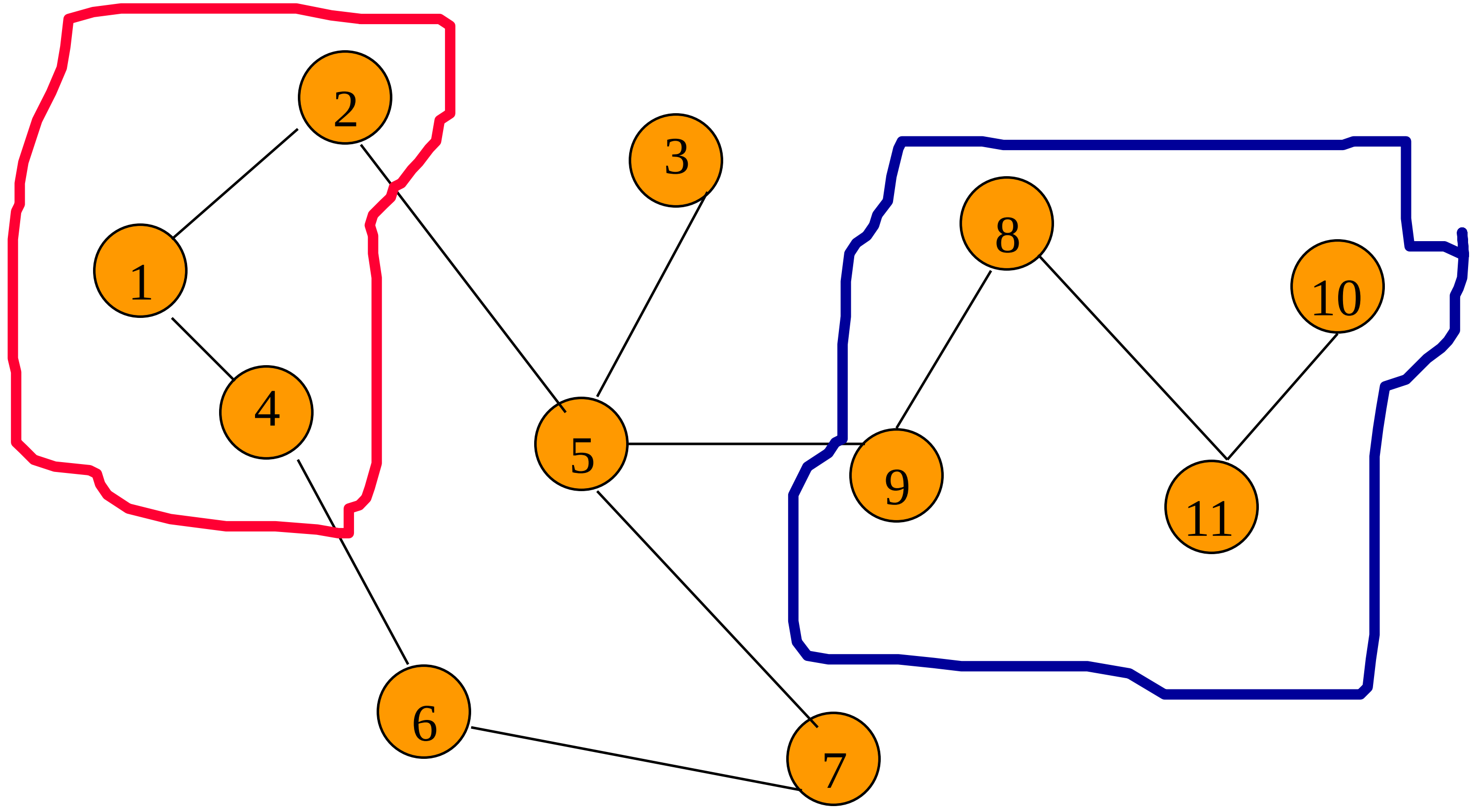
Connected Component

- A maximal subgraph that is connected.
  - Cannot add vertices and edges from original graph and retain connectedness.
- A connected graph has exactly 1 component.

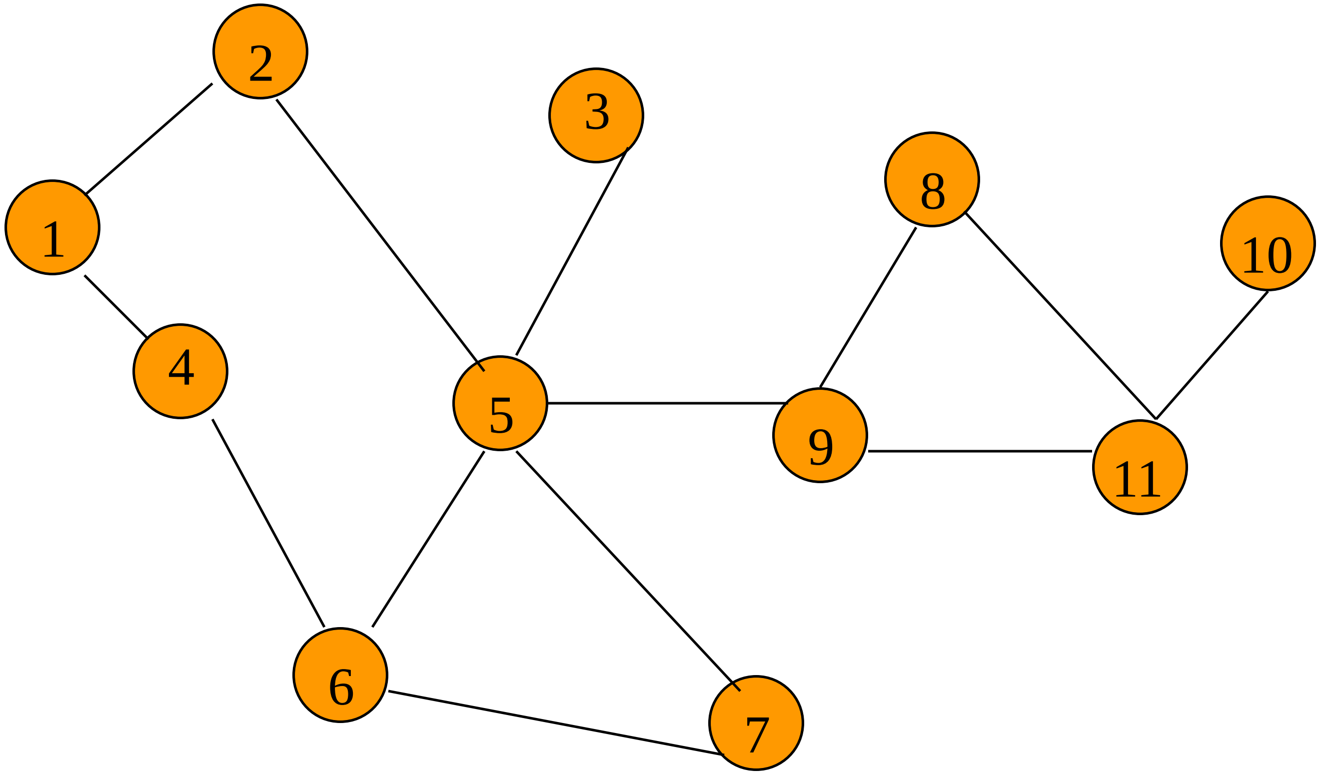
Connected Components



Not A Component



Communication Network

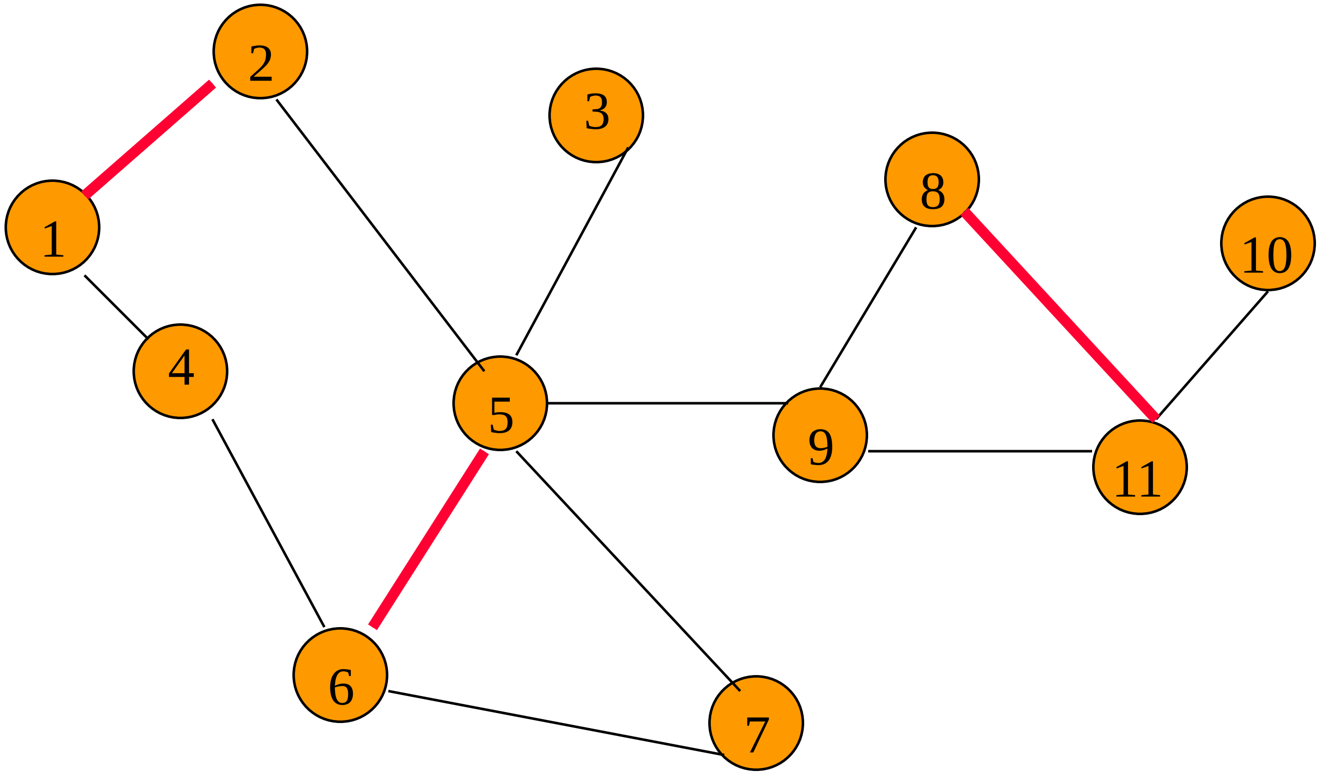


Each edge is a link that can be constructed (i.e., a feasible link).

Communication Network Problems

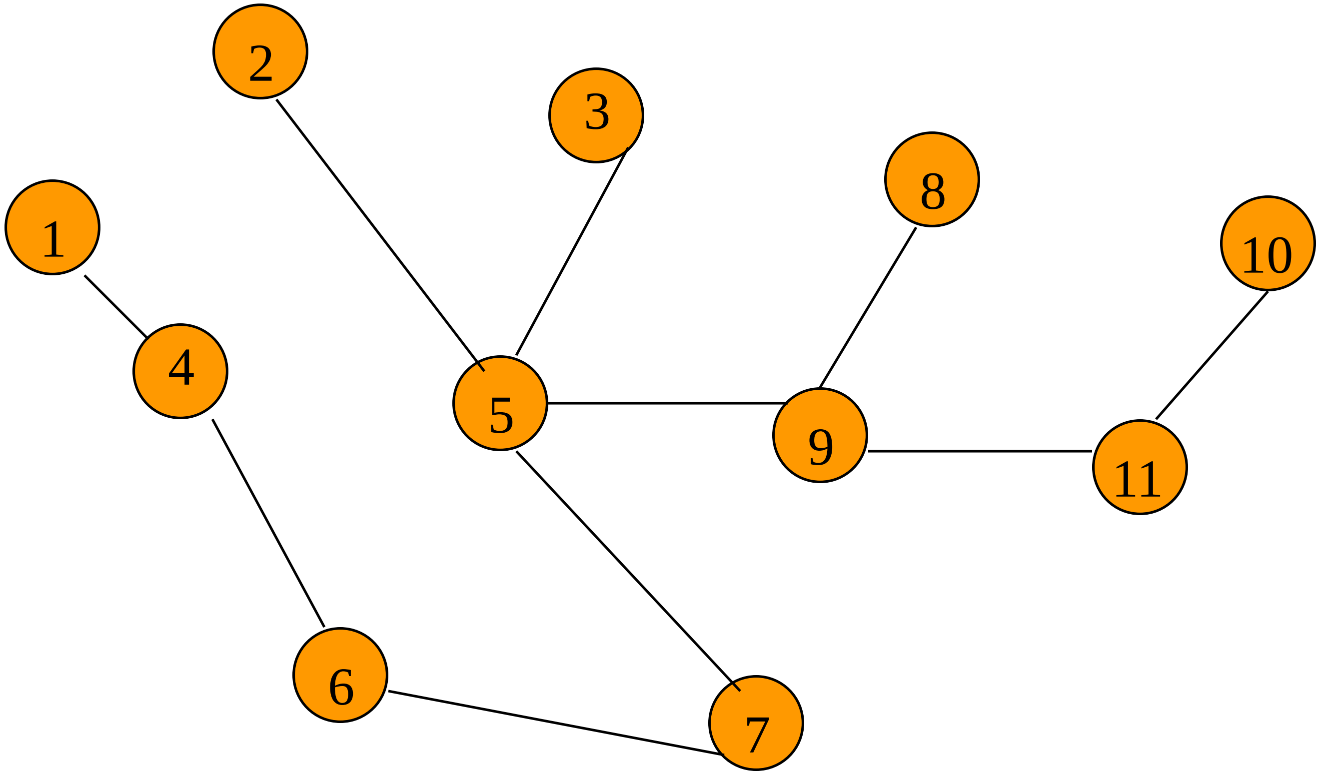
- Is the network connected?
  - Can we communicate between every pair of cities?
- Find the components.
- Want to construct smallest number of feasible links so that resulting network is connected.

Cycles And Connectedness



Removal of an edge that is on a cycle does not affect connectedness.

Cycles And Connectedness



Connected subgraph with all vertices and minimum number of edges has no cycles.

Tree

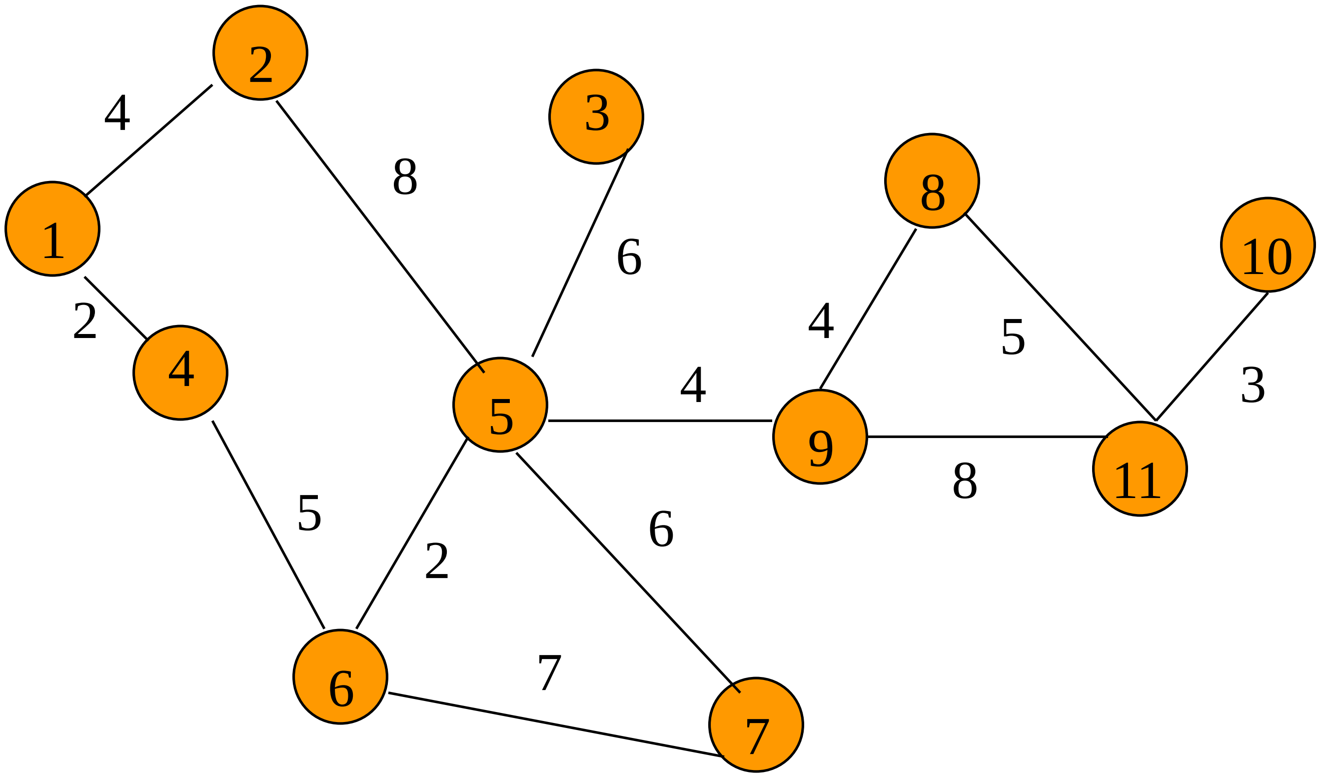
- Connected graph that has no cycles.
- n vertex connected graph with n-1 edges.

Spanning Tree

- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.

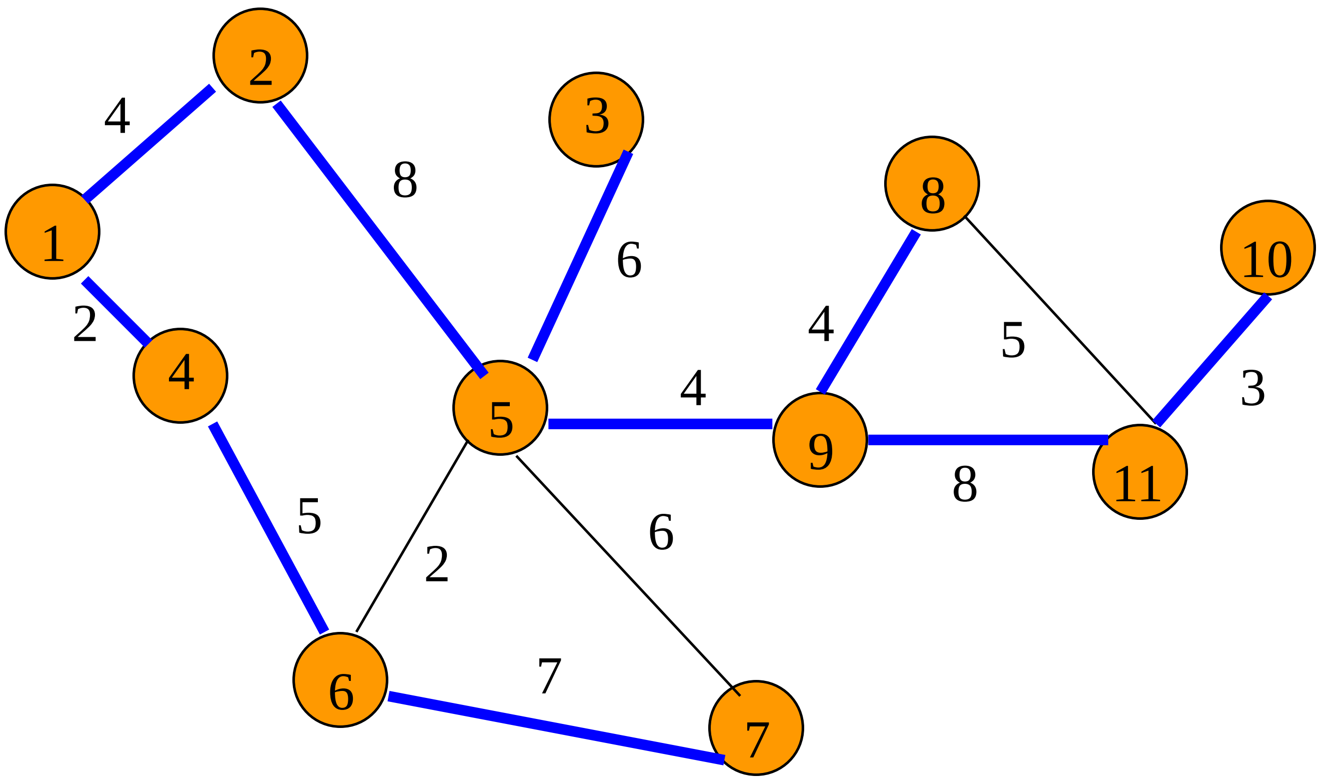
- If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

Minimum Cost Spanning Tree



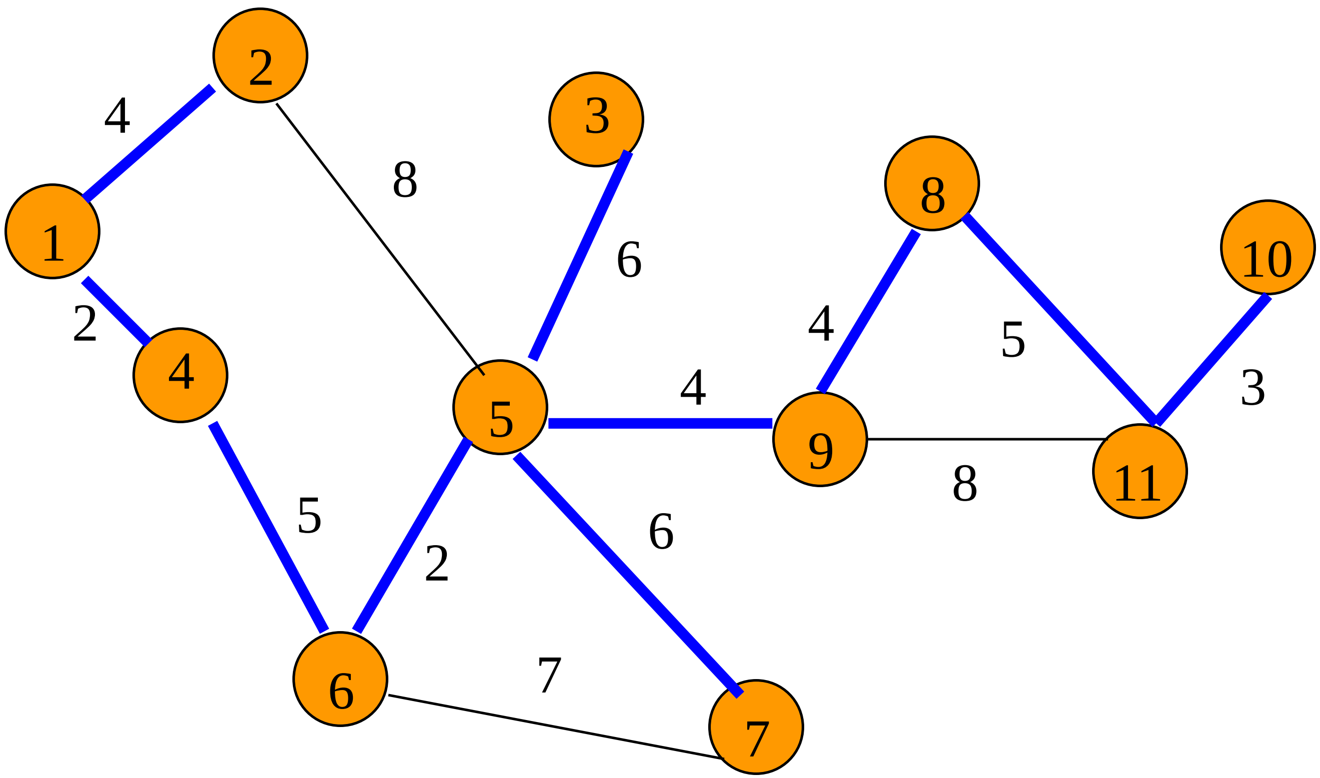
- Tree cost is sum of edge weights/costs.

A Spanning Tree



Spanning tree cost = 51.

Minimum Cost Spanning Tree



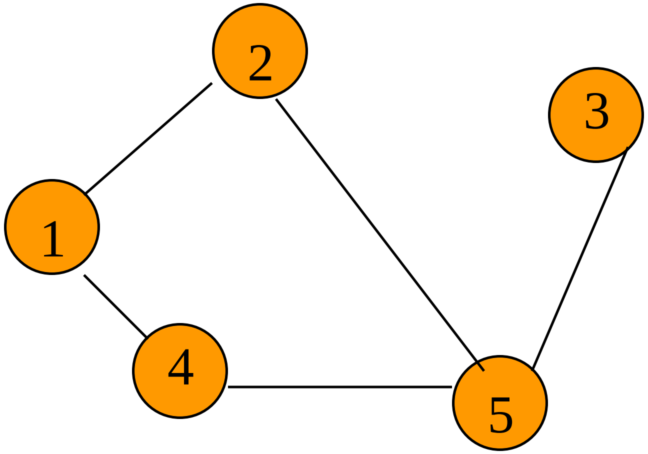
Spanning tree cost = 41.

Graph Representation

- Adjacency Matrix
- Adjacency Lists
  - Linked Adjacency Lists
  - Array Adjacency Lists

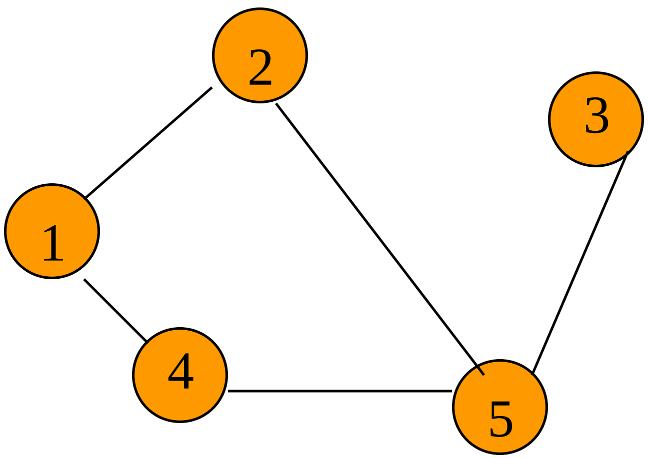
Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

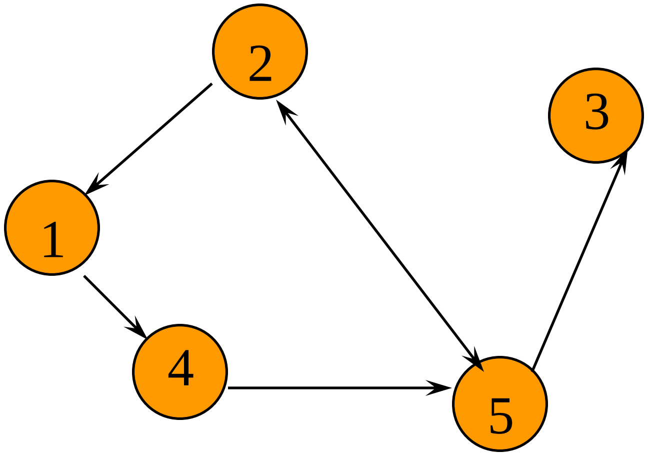
Adjacency Matrix Properties



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.
  - A(i,j) = A(j,i) for all i and j.

Adjacency Matrix (Digraph)



	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

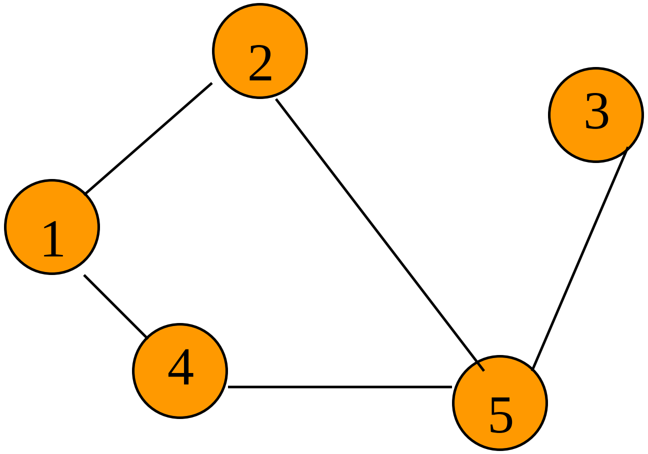
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

Adjacency Matrix

- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.

Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.
- An array of n adjacency lists.



aList[1] = (2,4)

aList[2] = (1,5)

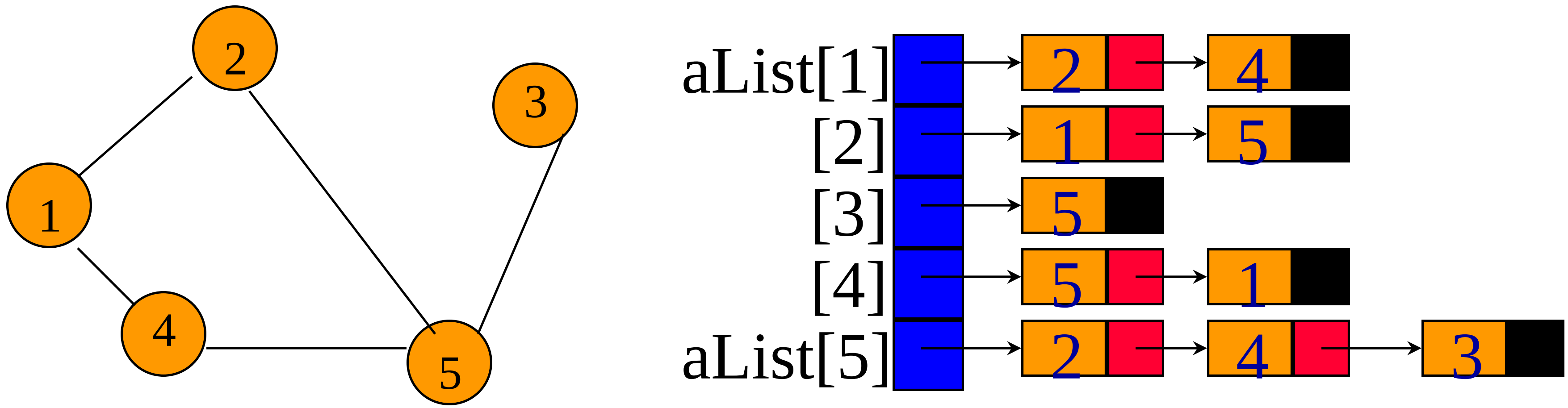
aList[3] = (5)

aList[4] = (5,1)

aList[5] = (2,4,3)

Linked Adjacency Lists

- Each adjacency list is a chain.



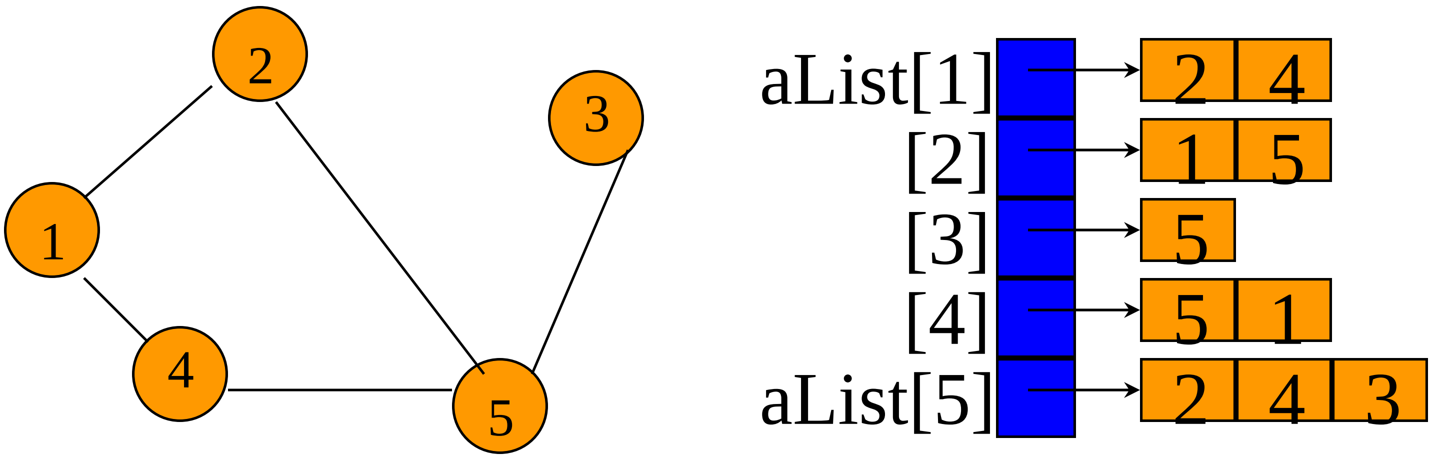
Array Length = n

# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Array Adjacency Lists

- Each adjacency list is an array list.



Array Length = n

# of list elements = 2e (undirected graph)

# of list elements = e (digraph)

Weighted Graphs

- Cost adjacency matrix.
  - C(i,j) = cost of edge (i,j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

Number Of Java Classes Needed

- Graph representations
  - Adjacency Matrix
  - Adjacency Lists
    - Linked Adjacency Lists
    - Array Adjacency Lists
  - 3 representations
- Graph types
  - Directed and undirected.
  - Weighted and unweighted.
  - 2 x 2 = 4 graph types
- 3 x 4 = 12 Java classes

Abstract Class Graph

```
package dataStructures;
import java.util.*;
public abstract class Graph
{
   // ADT methods come here

   // create an iterator for vertex i
   public abstract Iterator iterator(int i);

   // implementation independent methods come here
}
```

Abstract Methods Of Graph

```
// ADT methods
public abstract int vertices();
public abstract int edges();
public abstract boolean existsEdge(int i, int j);
public abstract void putEdge(Object theEdge);
public abstract void removeEdge(int i, int j);
public abstract int degree(int i);
public abstract int inDegree(int i);
public abstract int outDegree(int i);
```