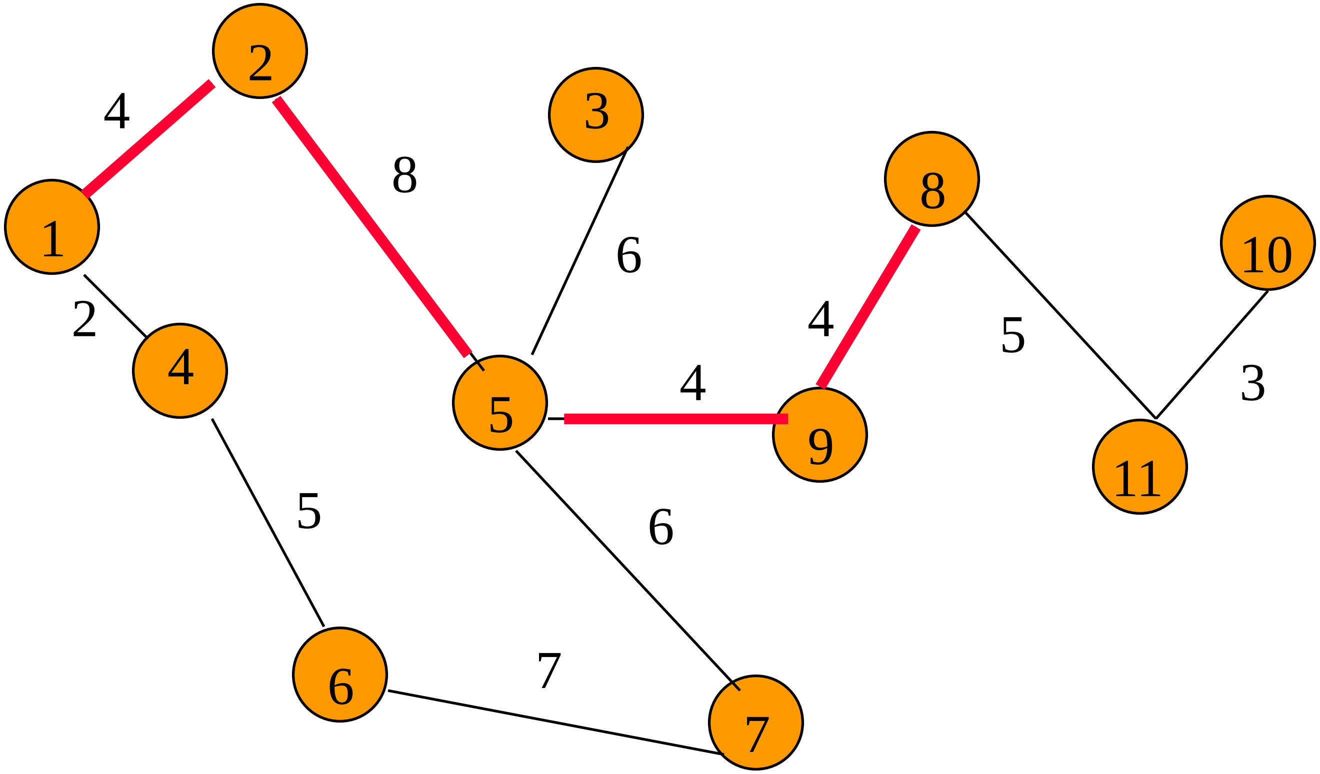
Graph Operations And Representation

Sample Graph Problems

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

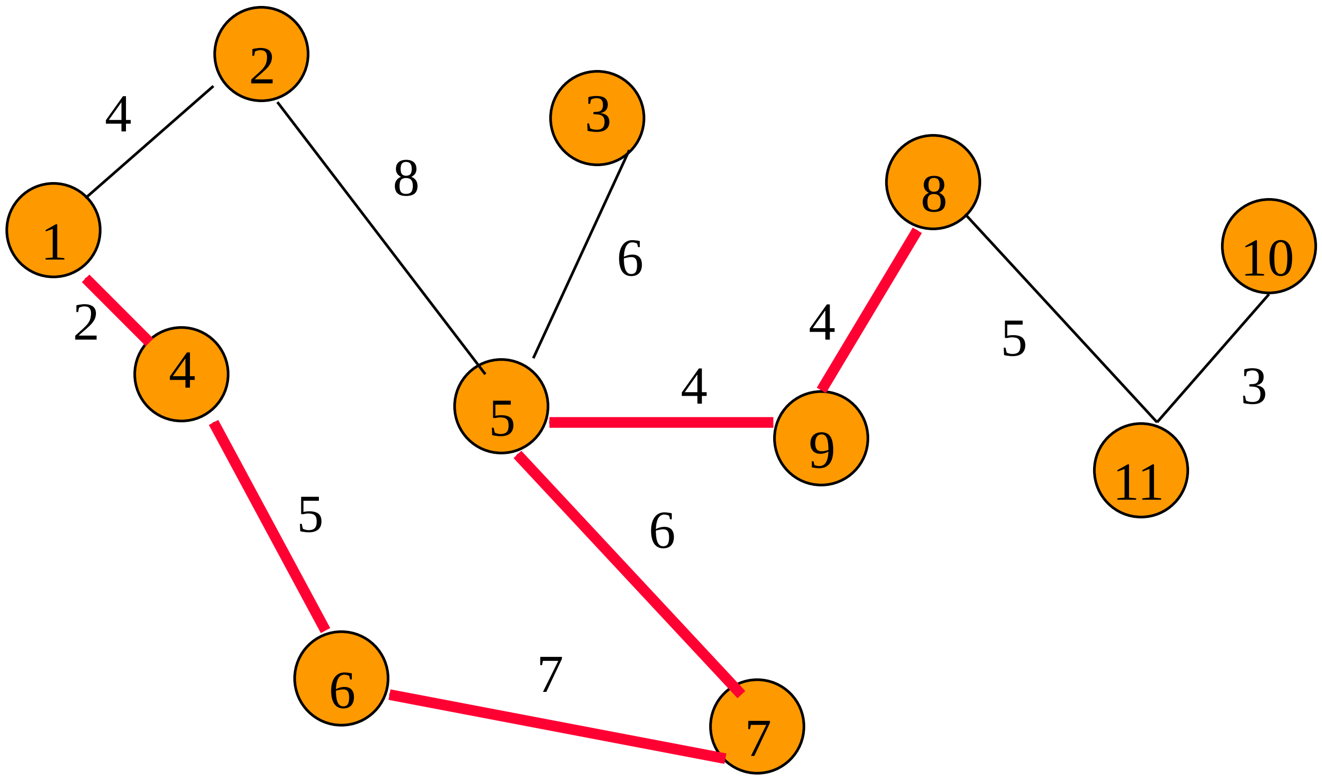
Path Finding

Path between 1 and 8.



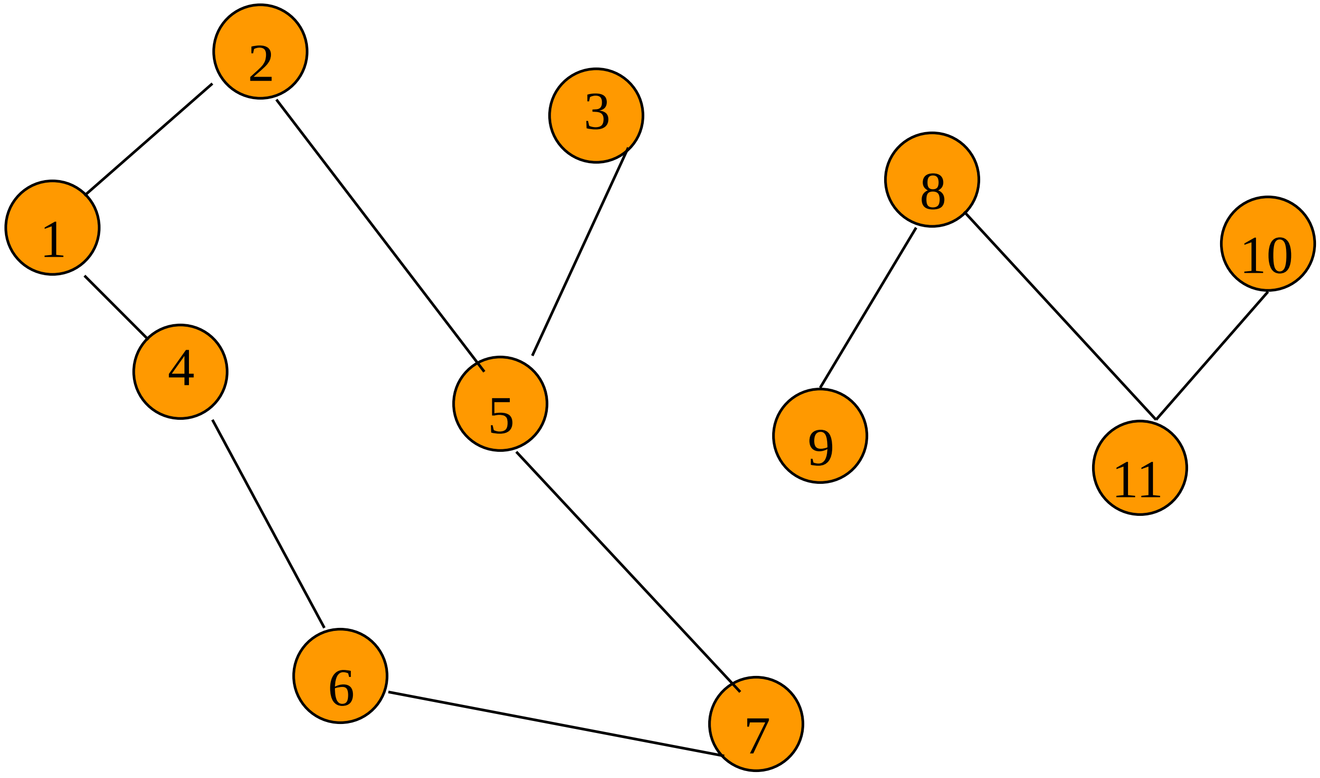
Path length is 20.

Another Path Between 1 and 8



Path length is 28.

Example Of No Path

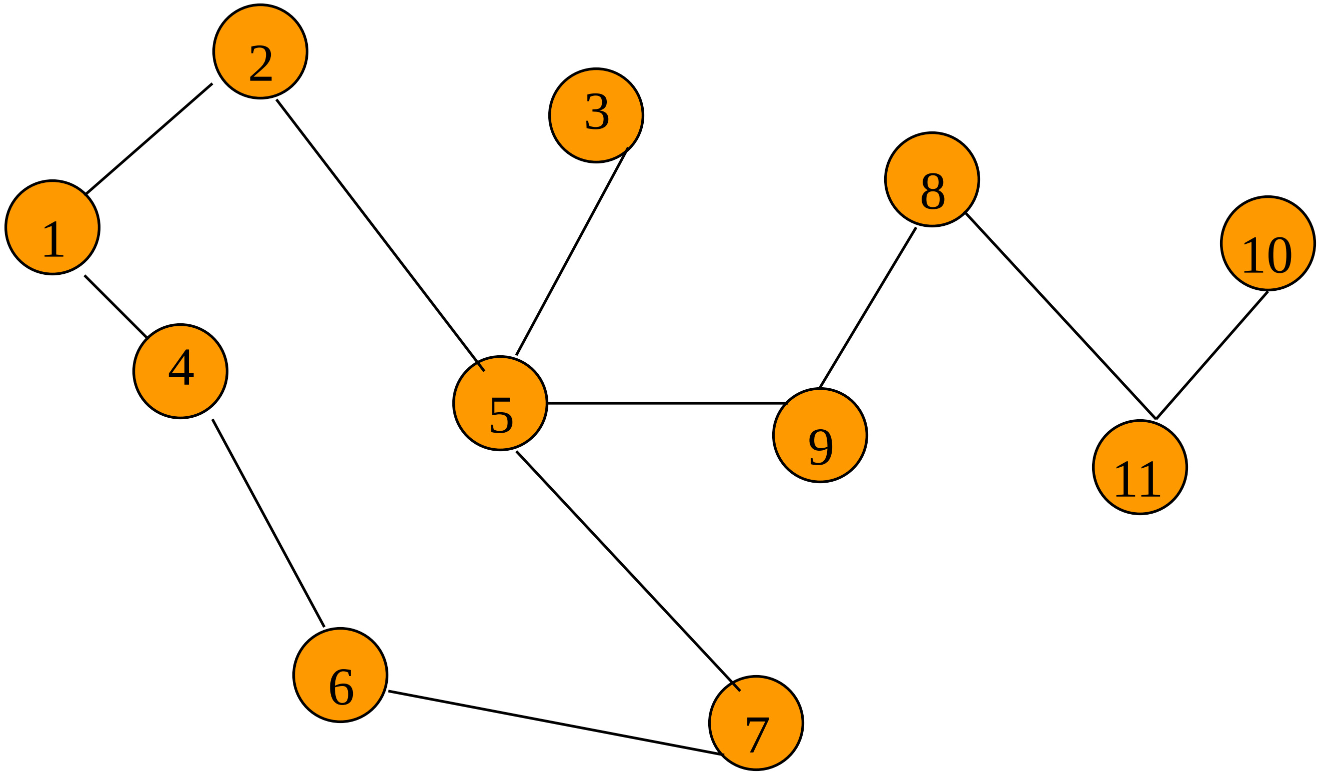


No path between 2 and 9.

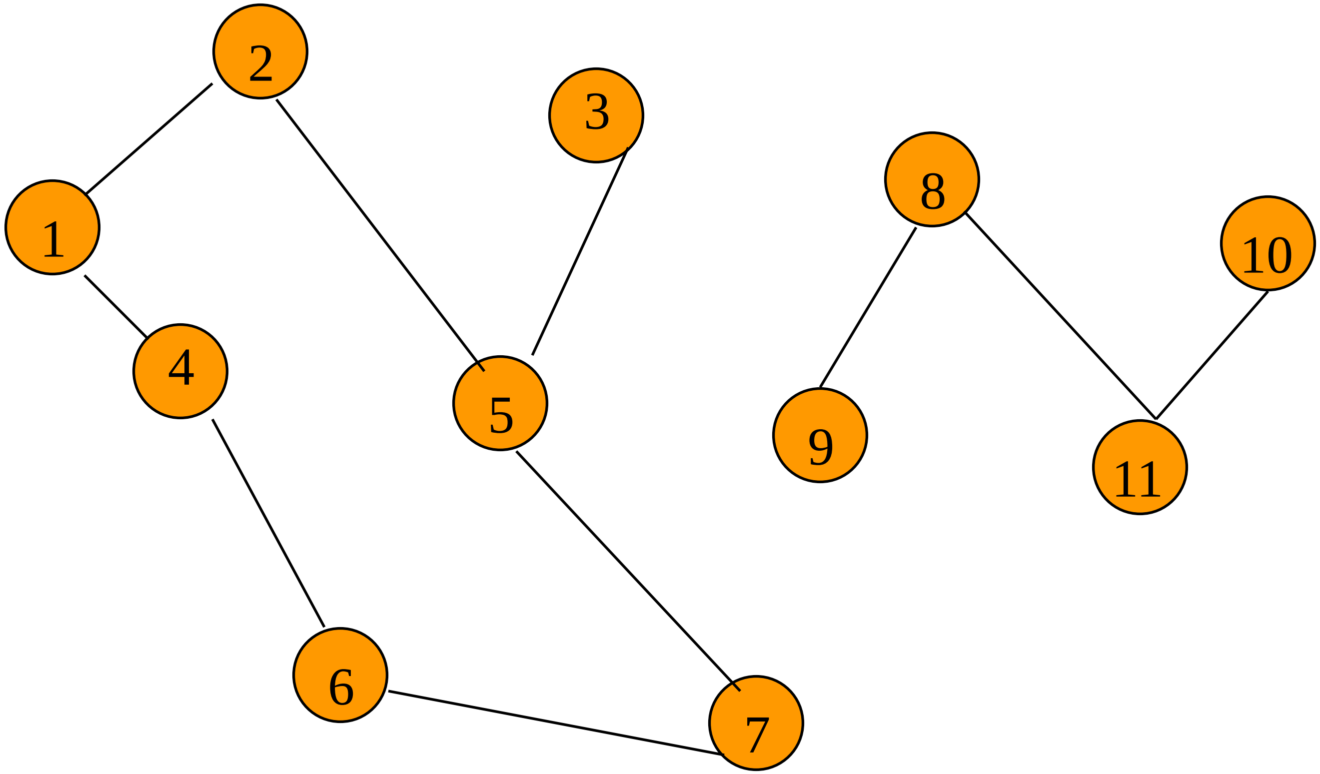
Connected Graph

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

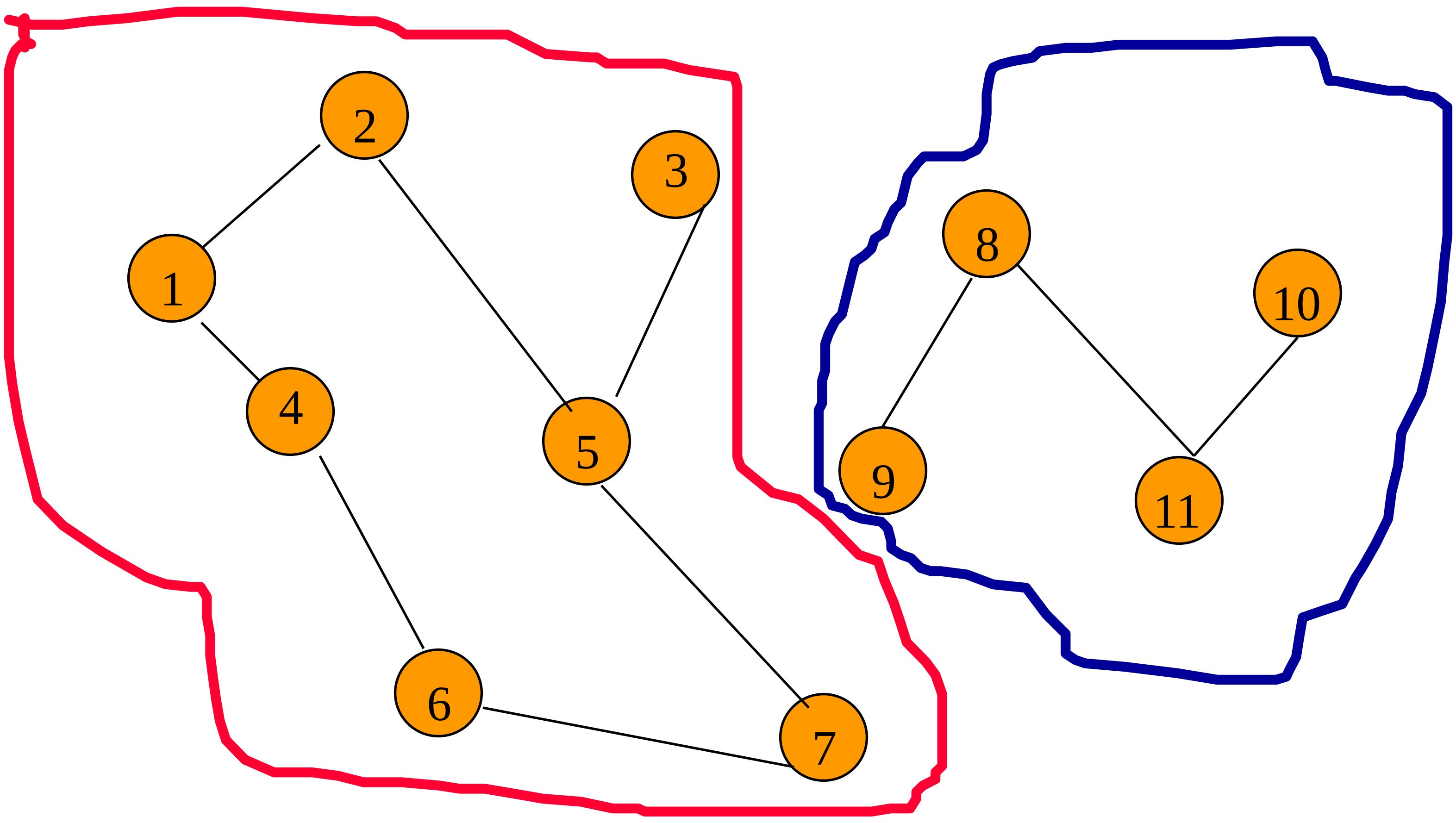
Connected Graph Example



Example Of Not Connected



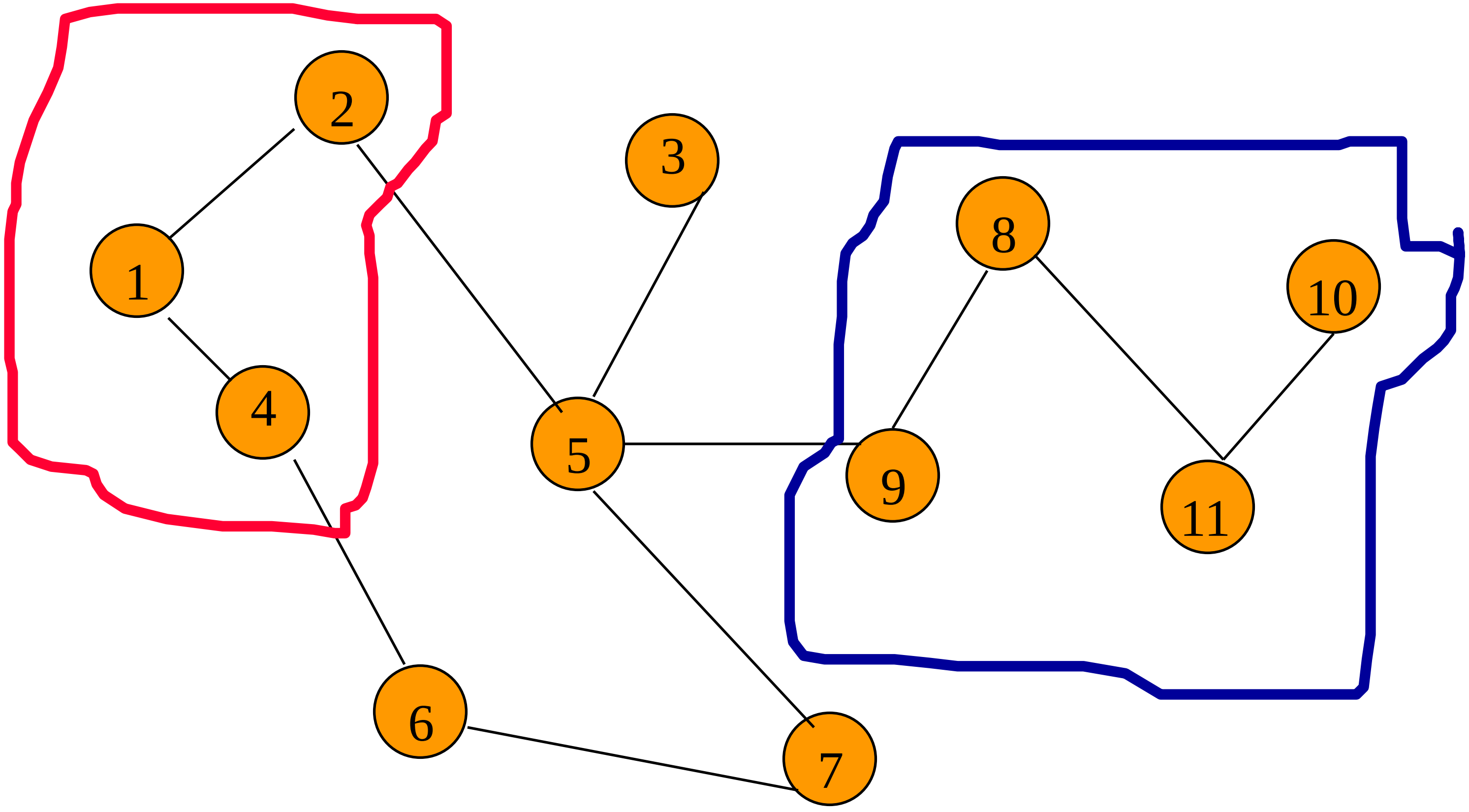
Connected Components



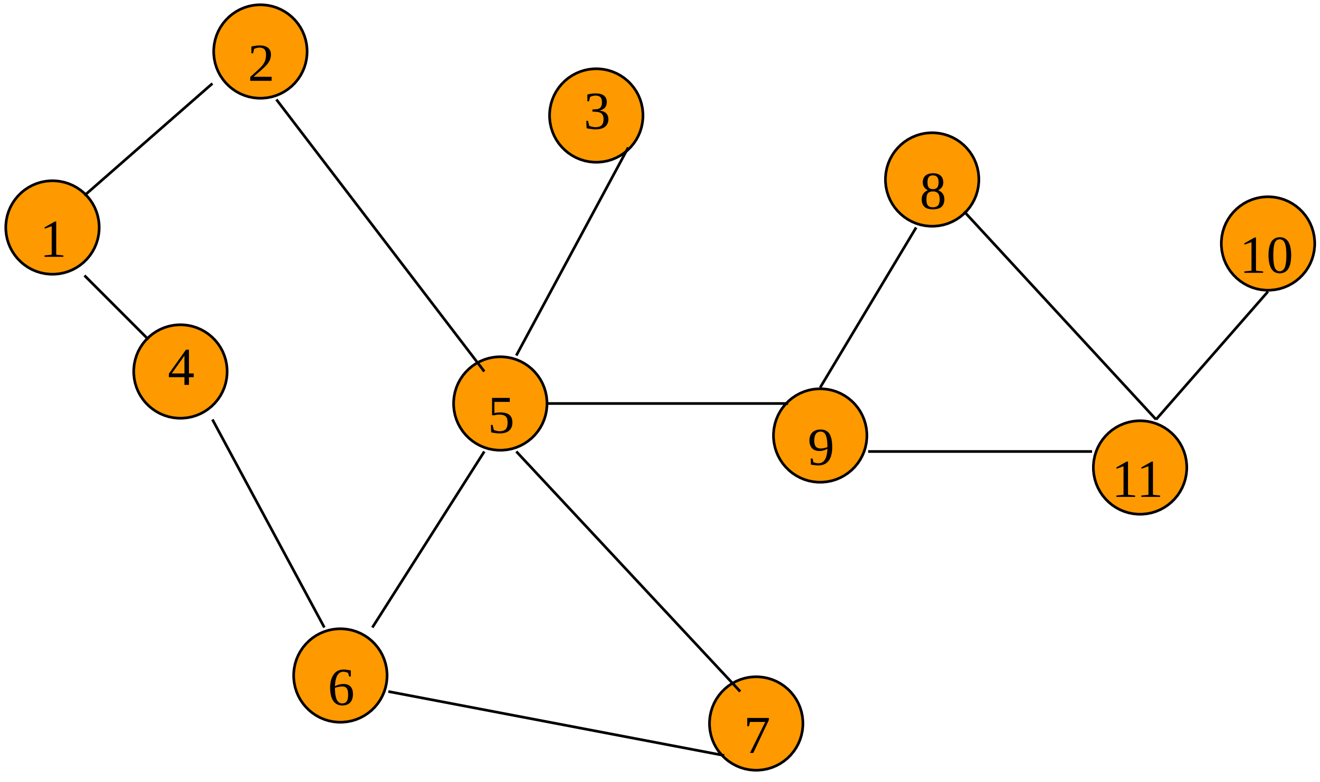
Connected Component

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

Not A Component



Communication Network

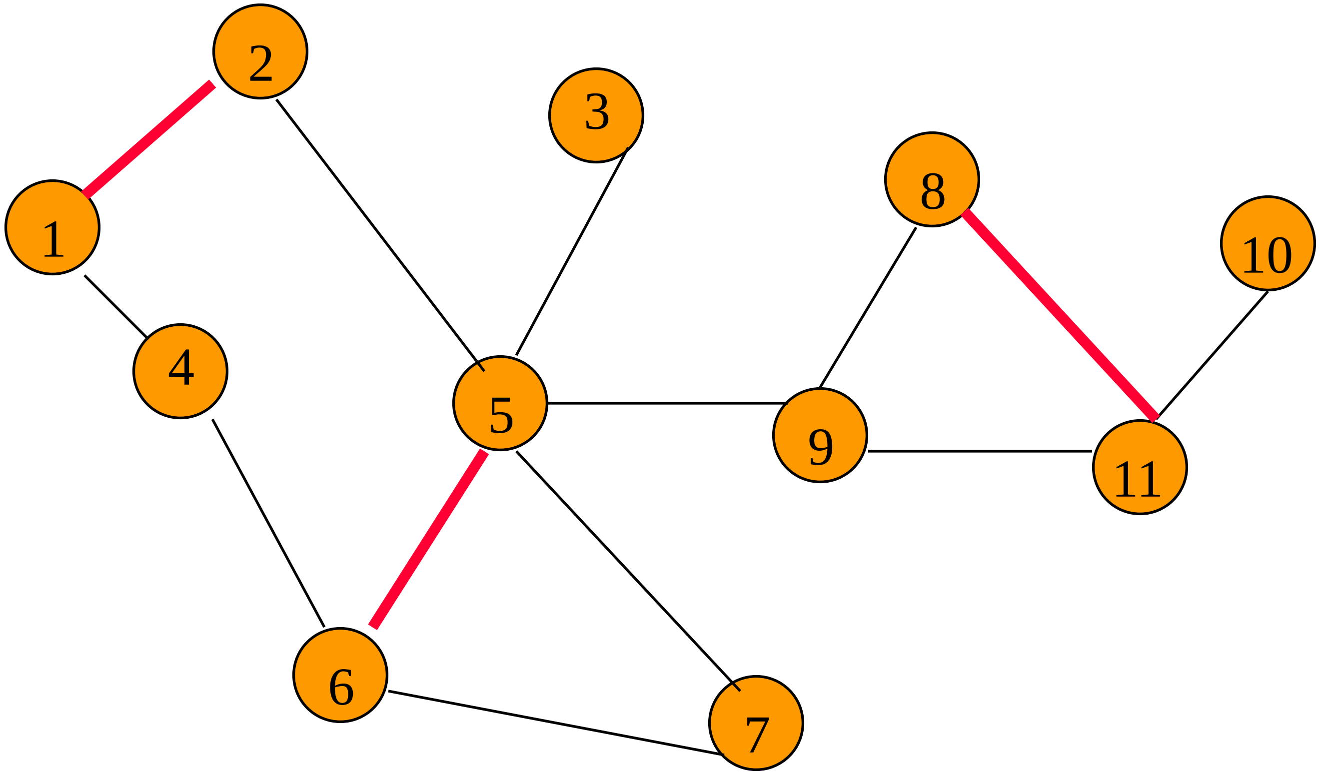


Each edge is a link that can be constructed (i.e., a feasible link).

Communication Network Problems

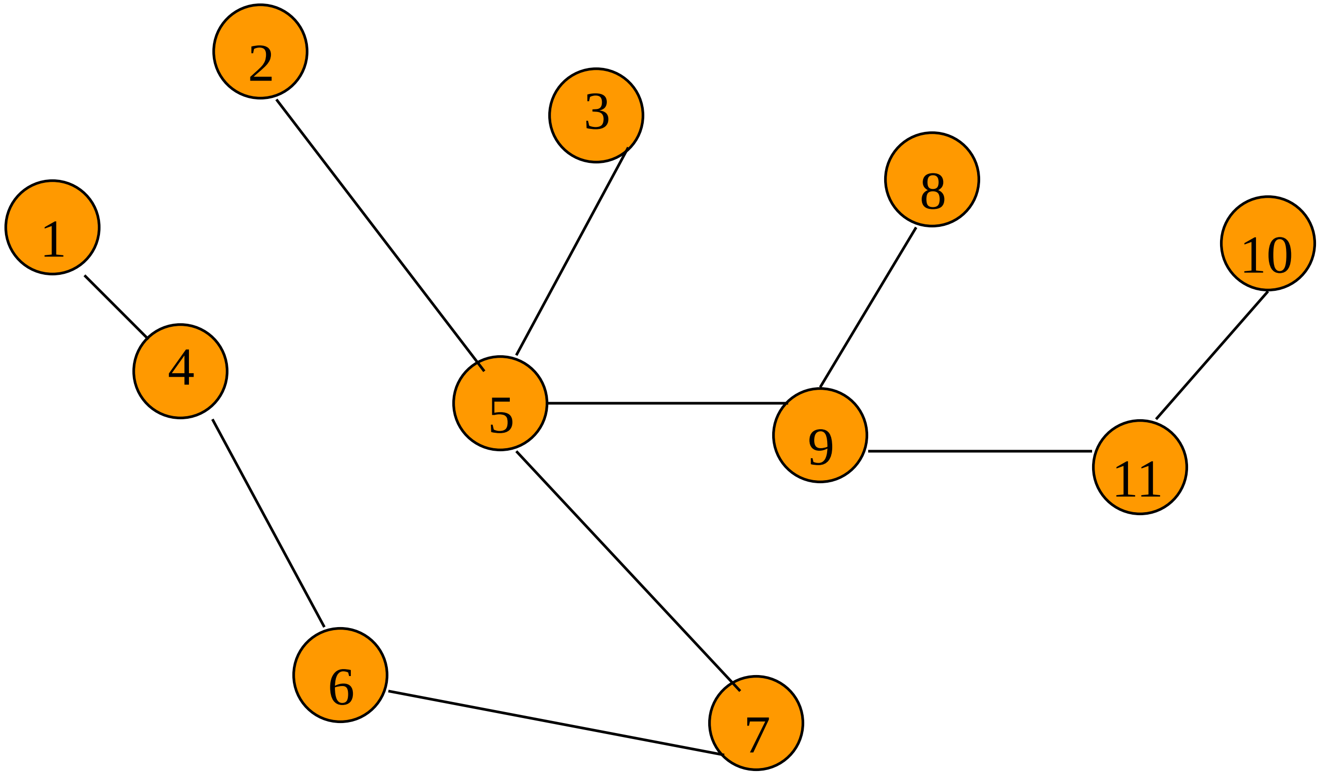
- Is the network connected?
  - Can we communicate between every pair of cities?
- Find the components.
- Want to construct smallest number of feasible links so that resulting network is connected.

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.

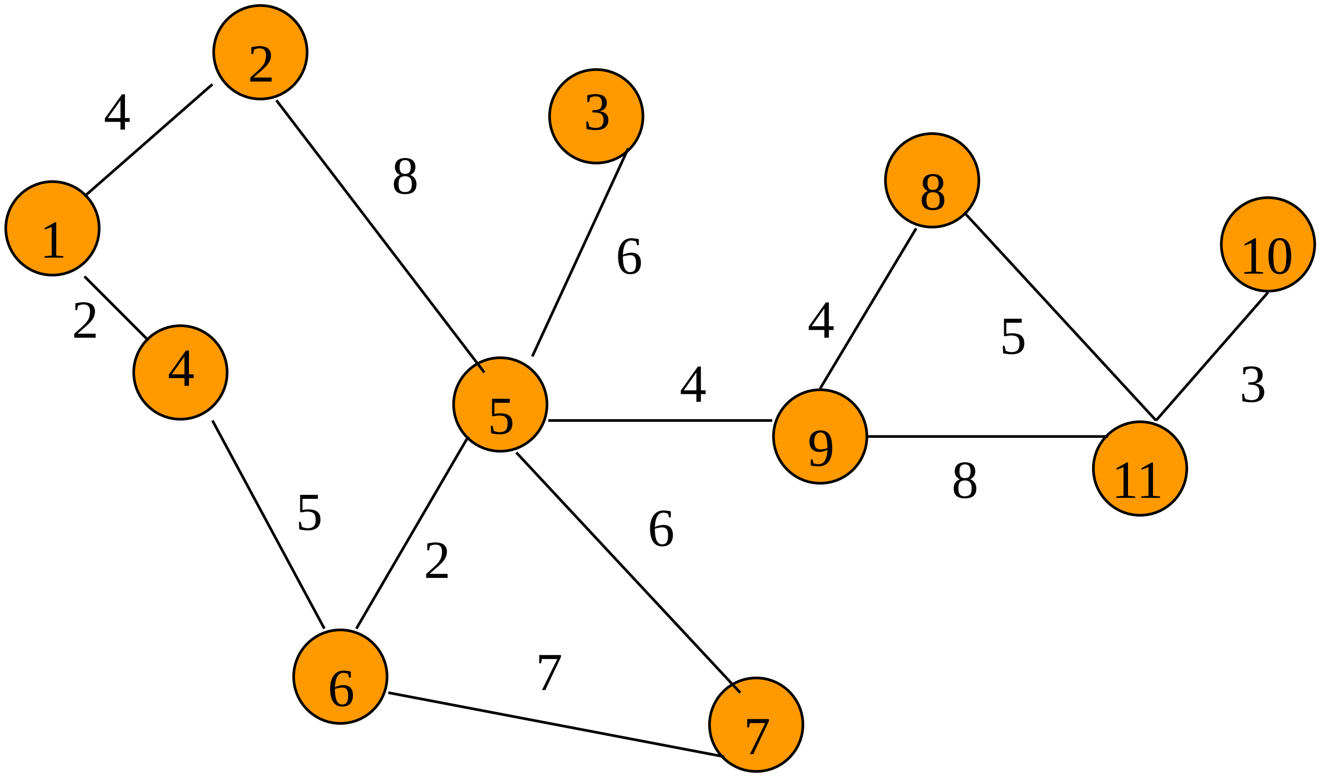
Tree

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

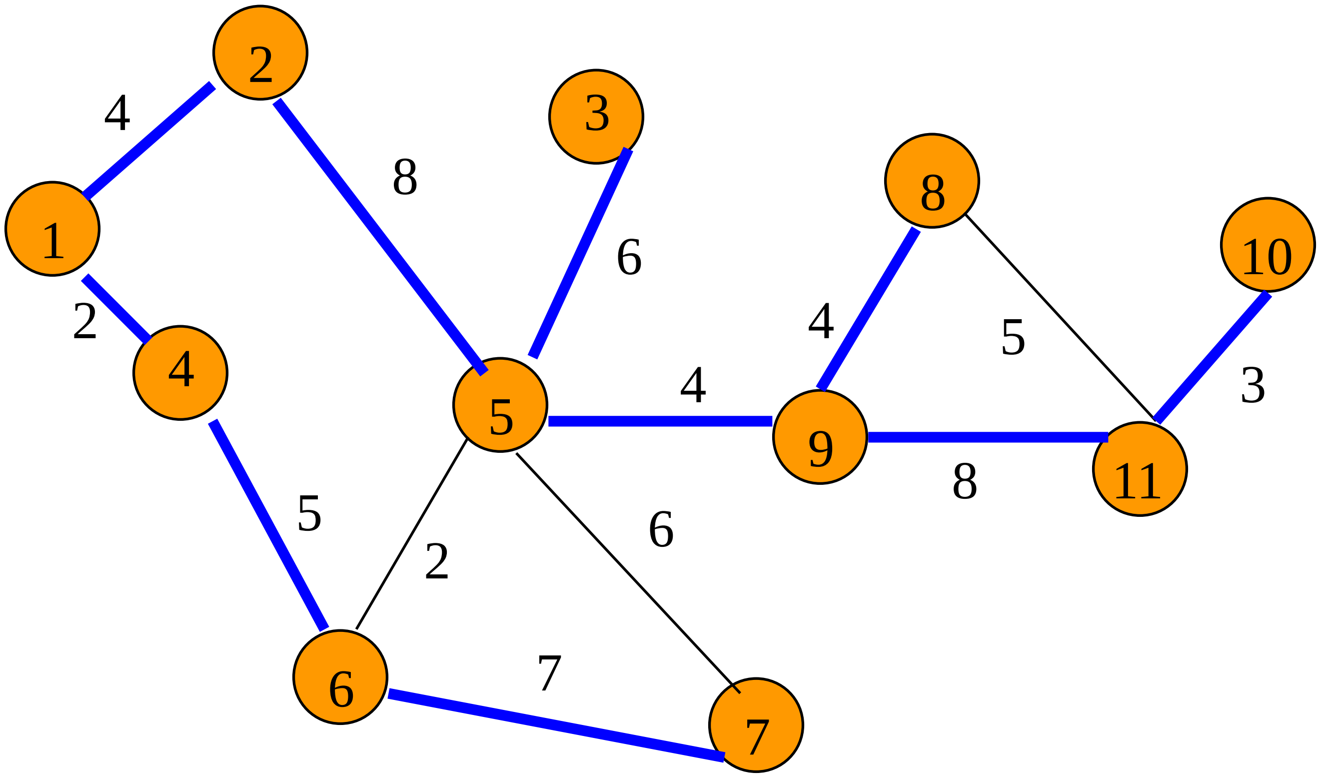
Spanning Tree

- Subgraph that includes all vertices of the original graph.
- Subgraph is connected
- Subgraph is a tree
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

An example Graph



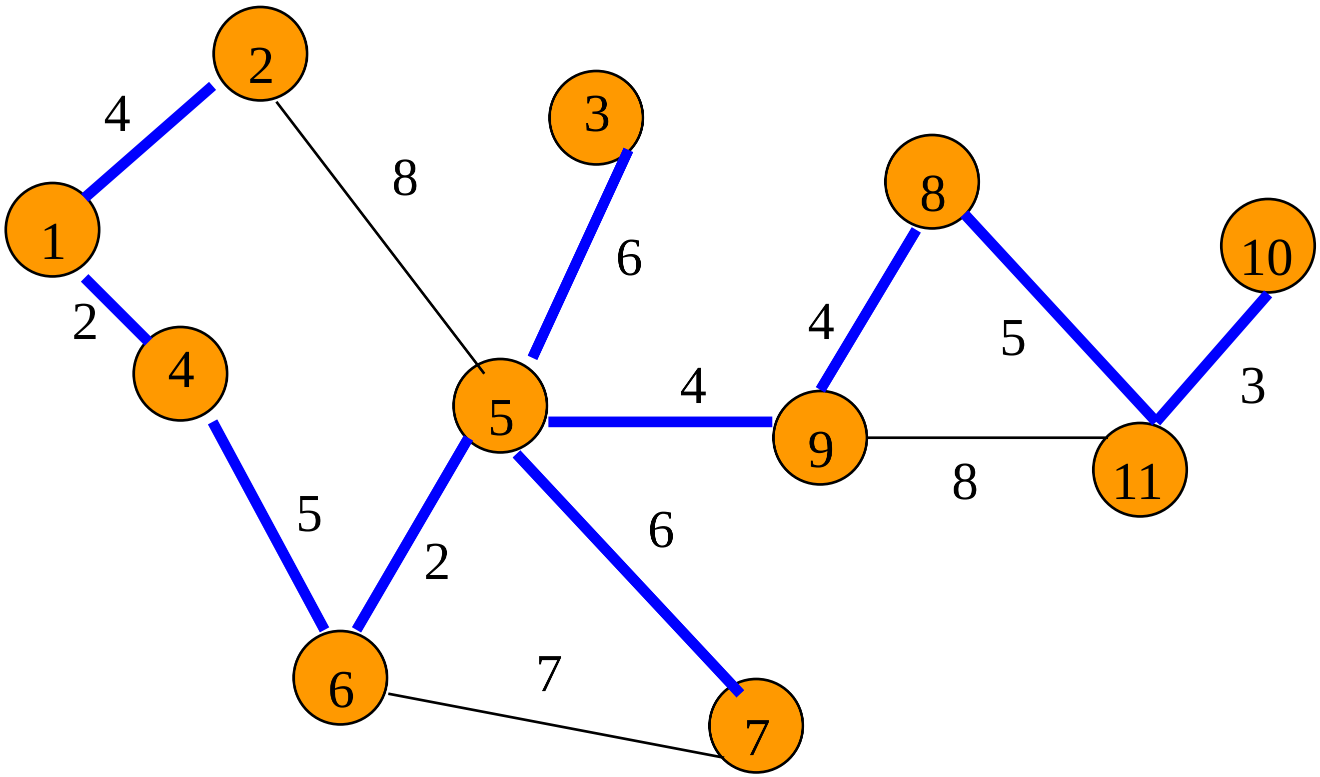
A Spanning Tree



Tree cost is sum of edge weights/costs.

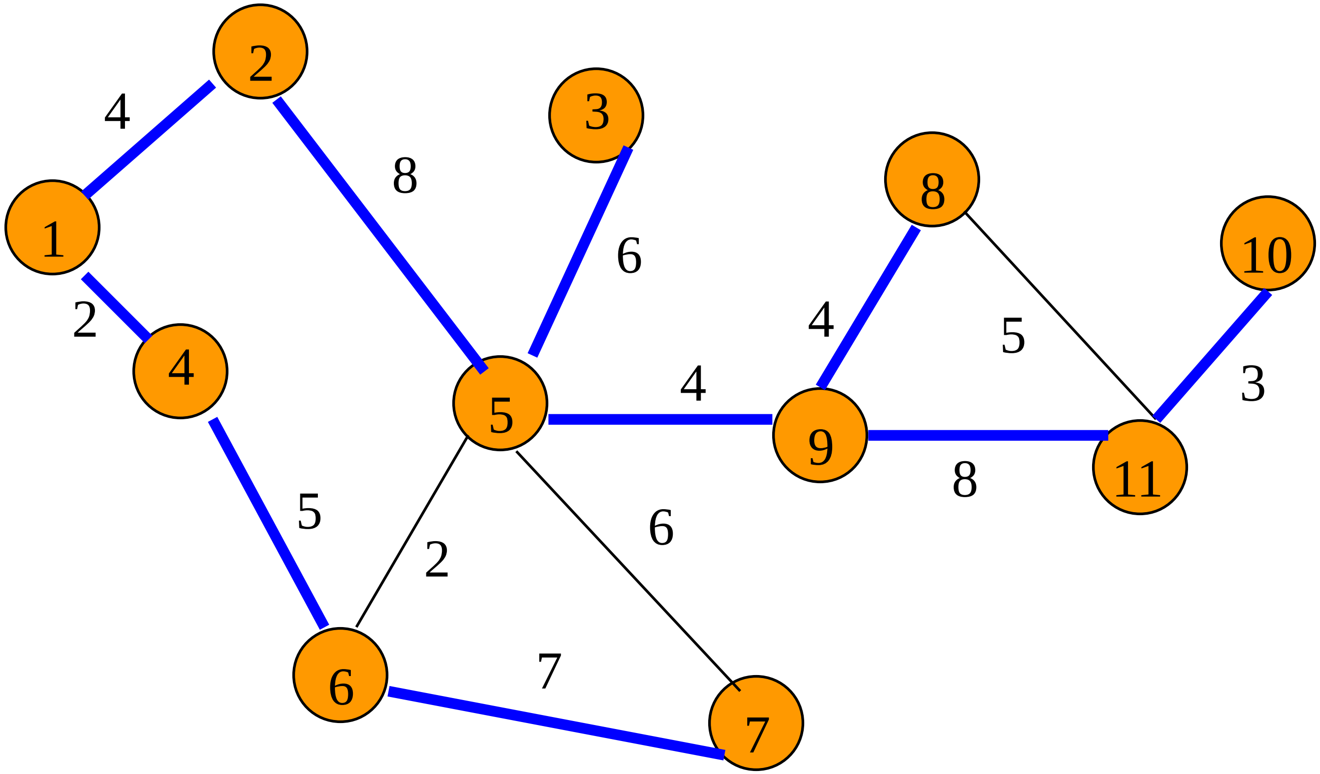
Spanning tree cost = 51.

Minimum Cost Spanning Tree



Spanning tree cost = 41.

A Wireless Broadcast Tree



Source = 1, weights = needed power.

Cost = 4 + 8 + 5 + 6 + 7 + 8 + 3 = 41.

Minimum Spanning Tree
Kruskal's Idea

- Build a minimum cost spanning tree T by adding edges to T one at a time
- Select the edges for inclusion in T in increasing order of the cost
- An edge is added to T if it does not form a cycle
- Since G is connected and has $n > 0$ vertices, exactly $n-1$ edges will be selected

Examples for Kruskal's Algorithm

0 10 5

2 12 3

1 14 6

1 16 2

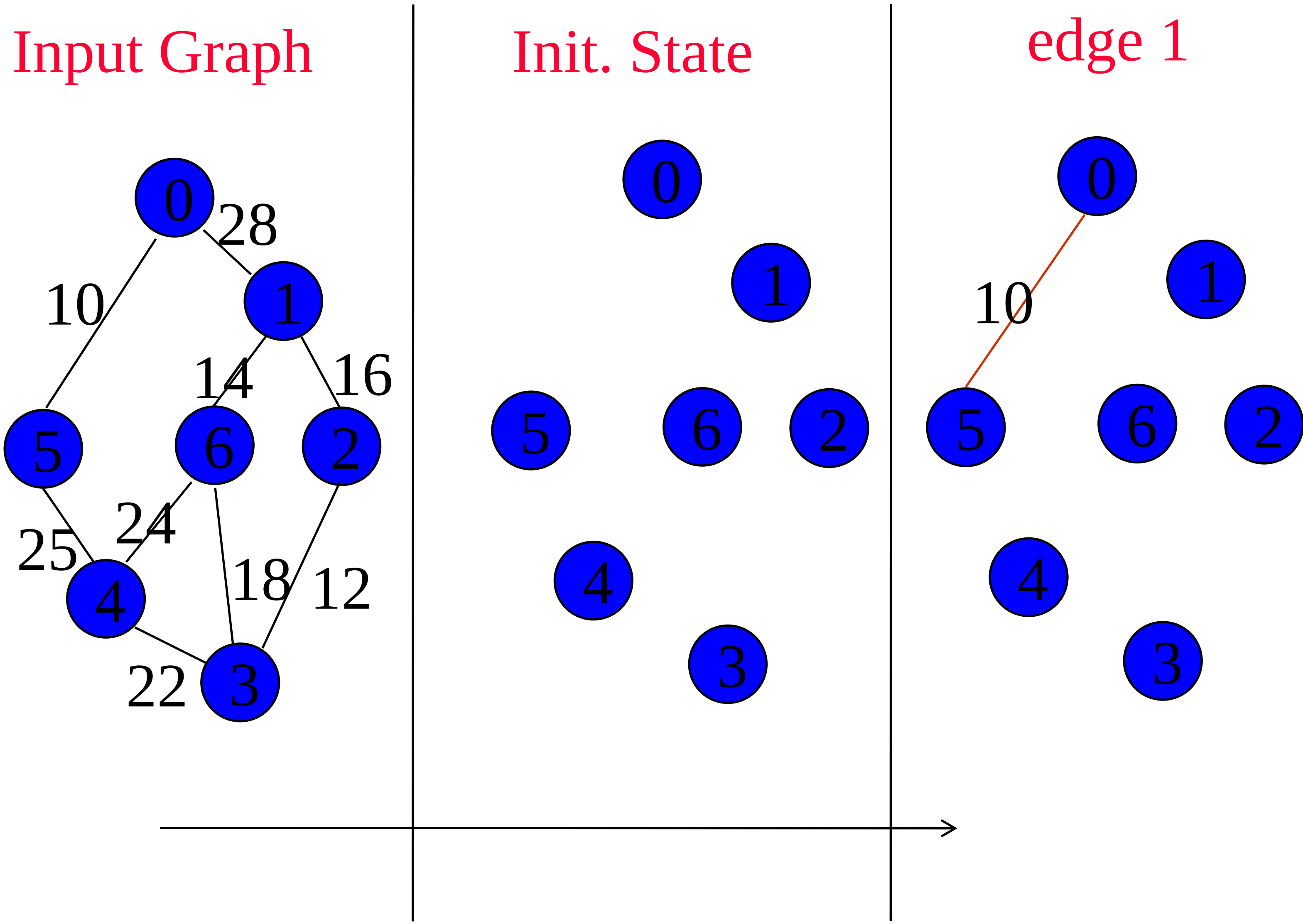
3 18 6

3 22 4

4 24 6

4 25 5

0 28 1



0 — 10 — 5

2 — 12 — 3

1 — 14 — 6

1 — 16 — 2

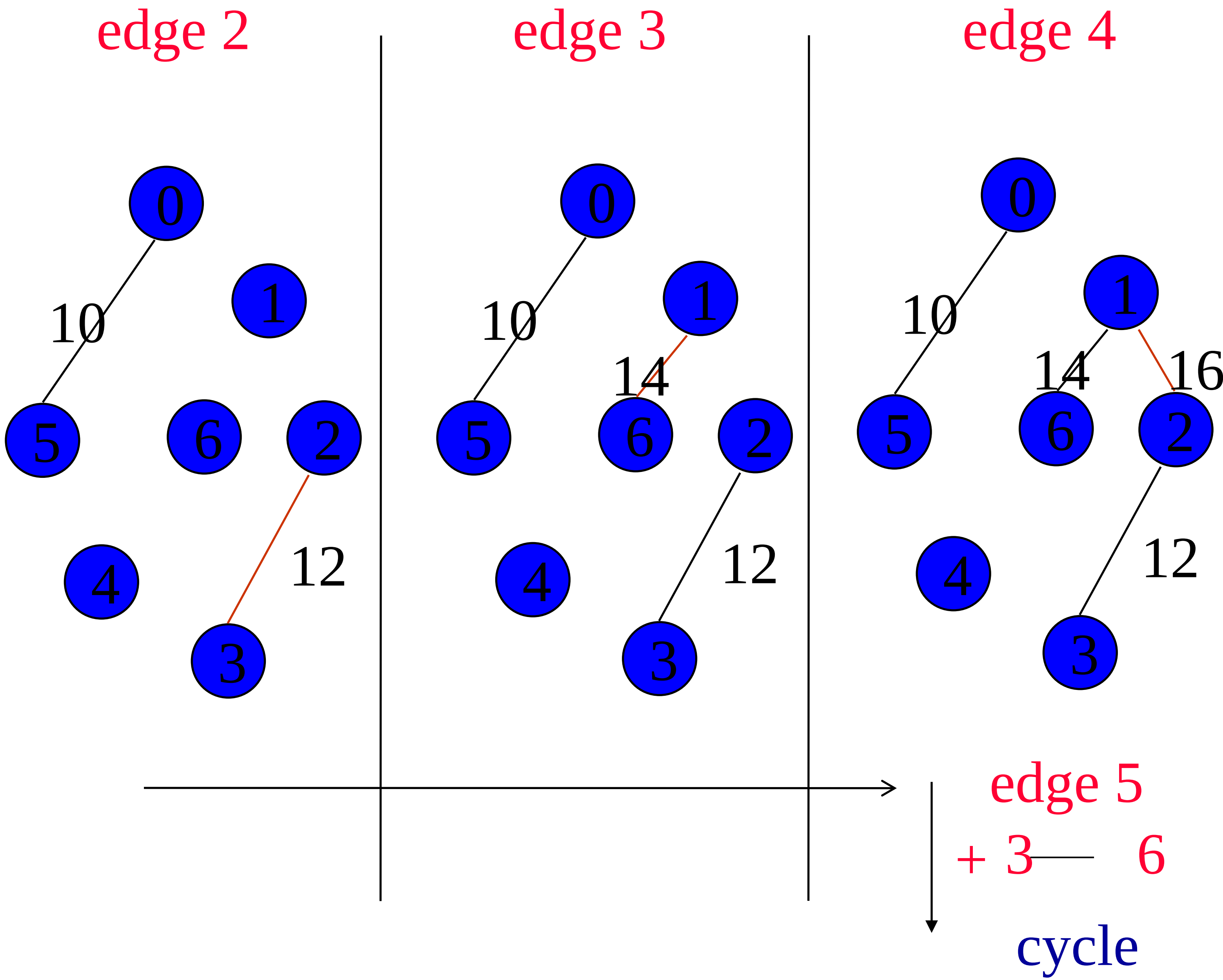
3 — 18 — 6

3 — 22 — 4

4 — 24 — 6

4 — 25 — 5

0 — 28 — 1



0 10 5

2 12 3

1 14 6

1 16 2

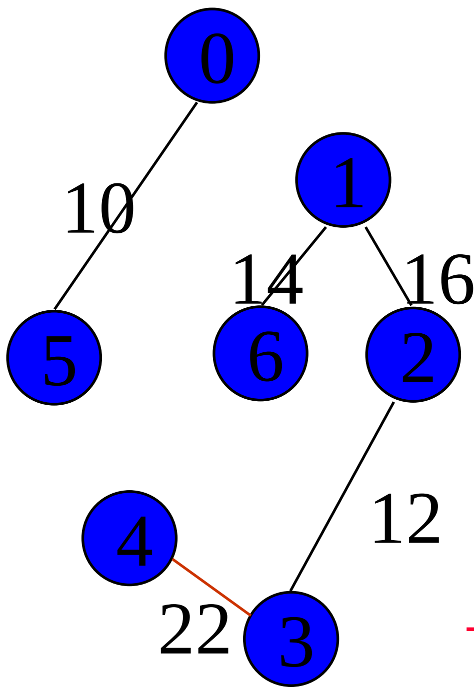
3 18 6

→ 3 22 4

4 24 6

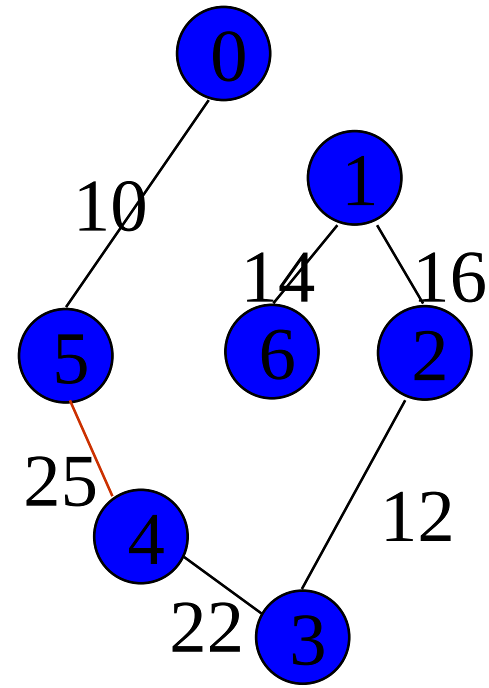
4 25 5

0 28 1



+ 4 — 6

cycle



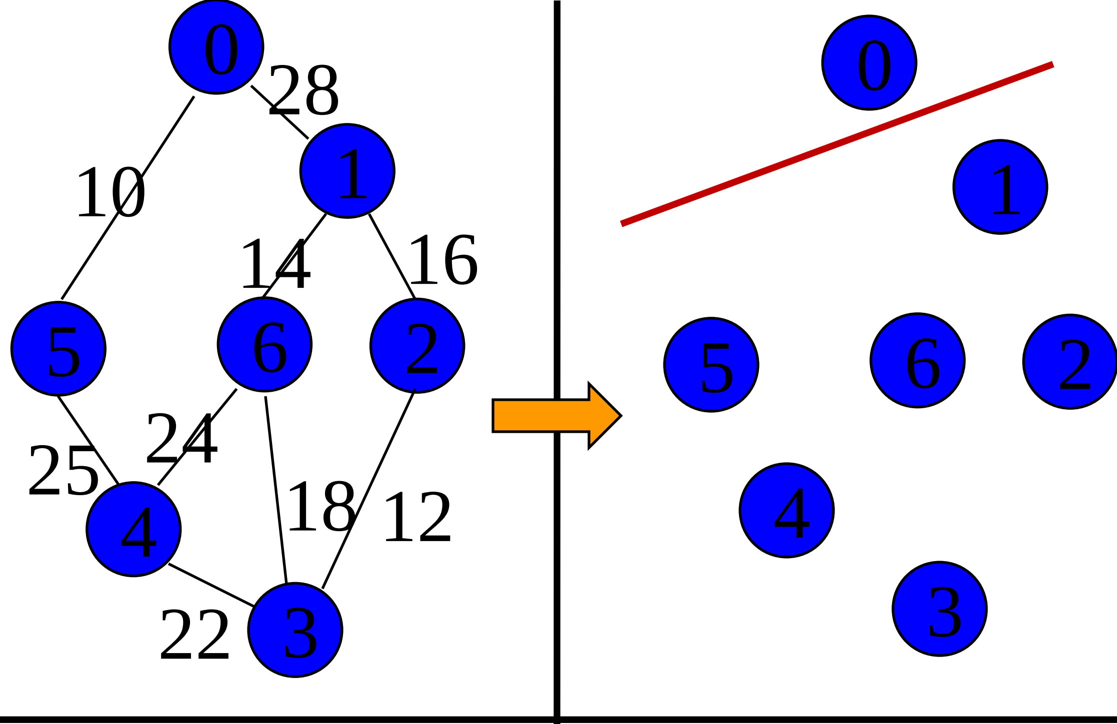
cost = 10 +25+22+12+16+14

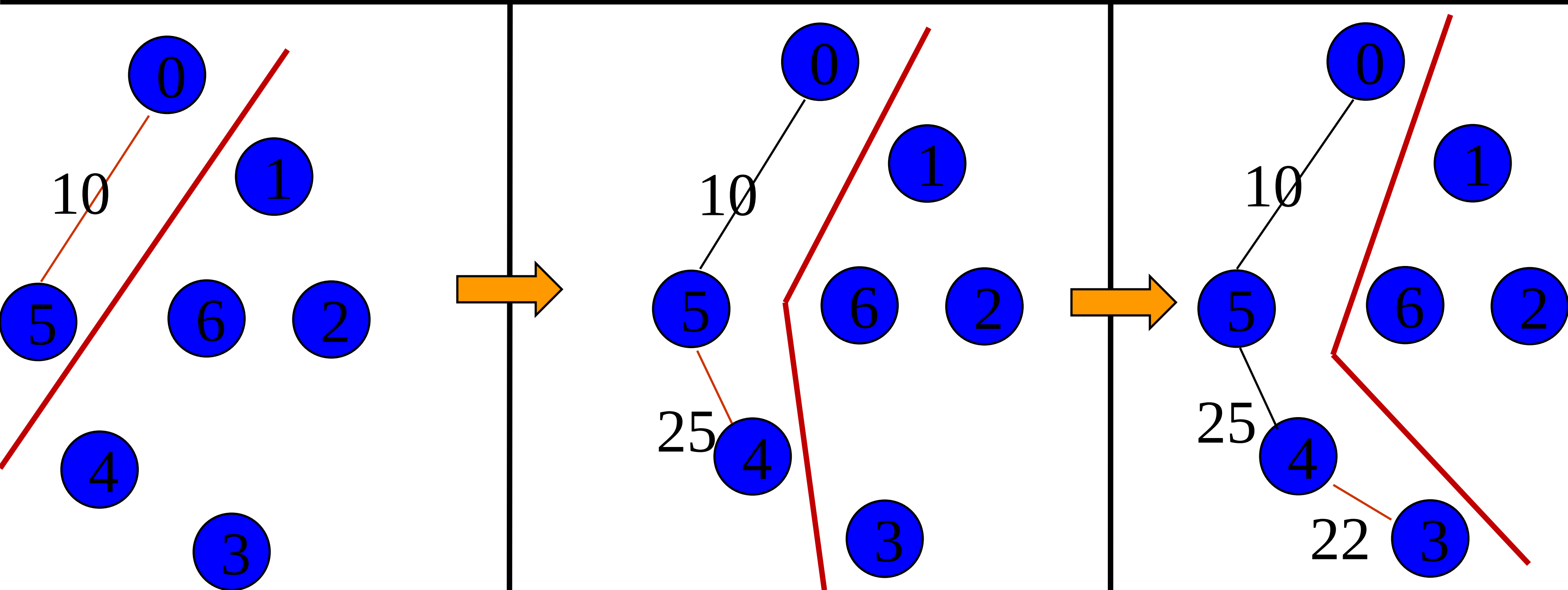
Implementation

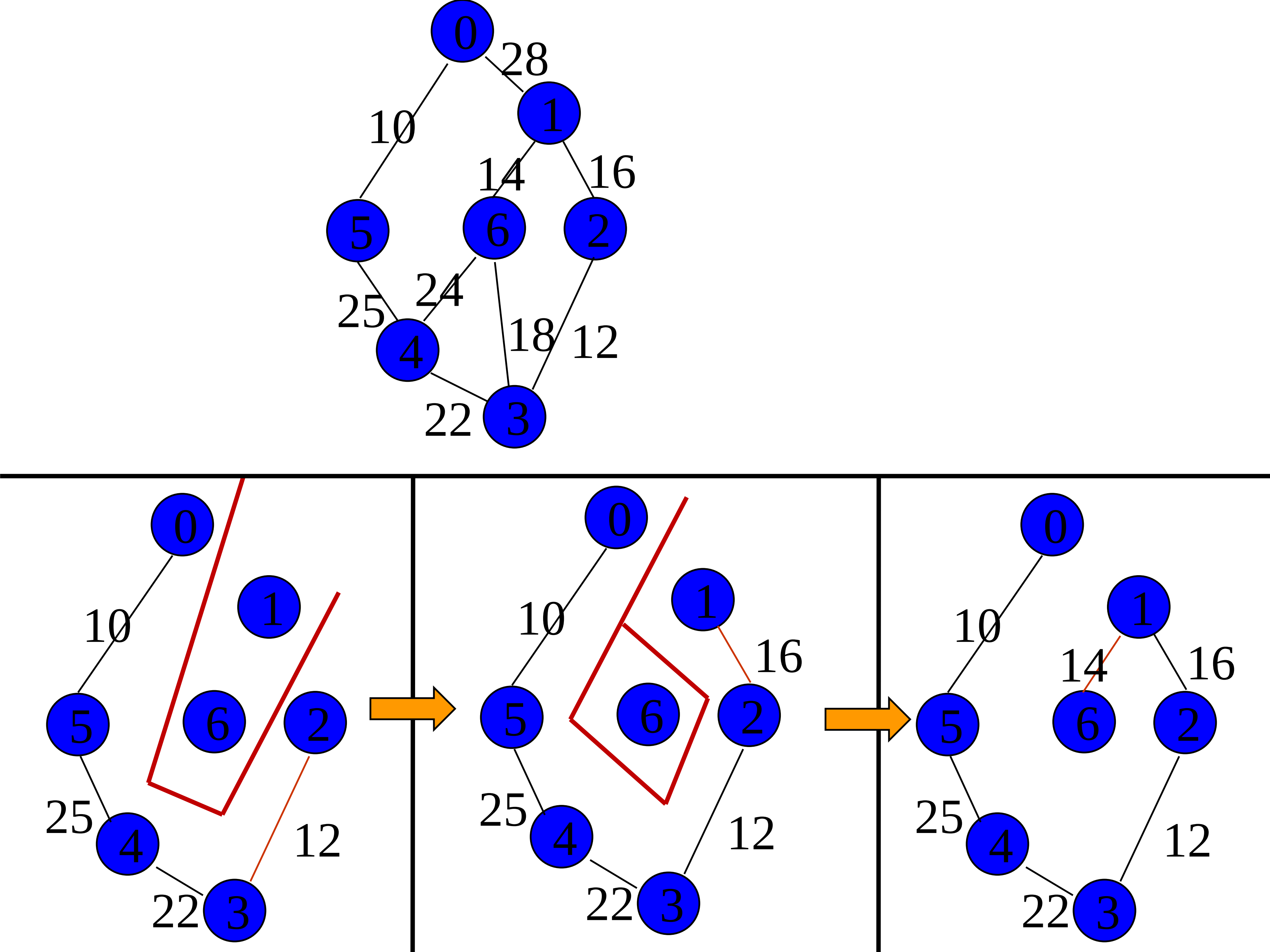
- Take following 3 arrays
  - A: All input edges, an edge as object (weight,u,v)
  - B: Output array, contain selected edges
  - C: Temp array, Sets of vertices

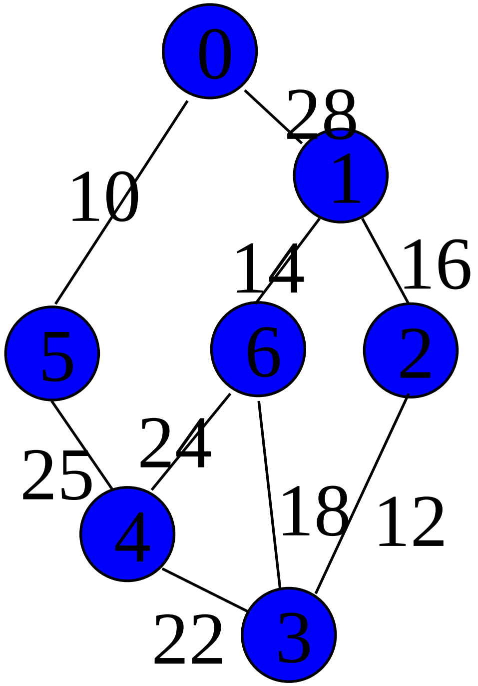
- Print: a) all edges in array B
  b) Tree cost

Example of Prim's Algorithm



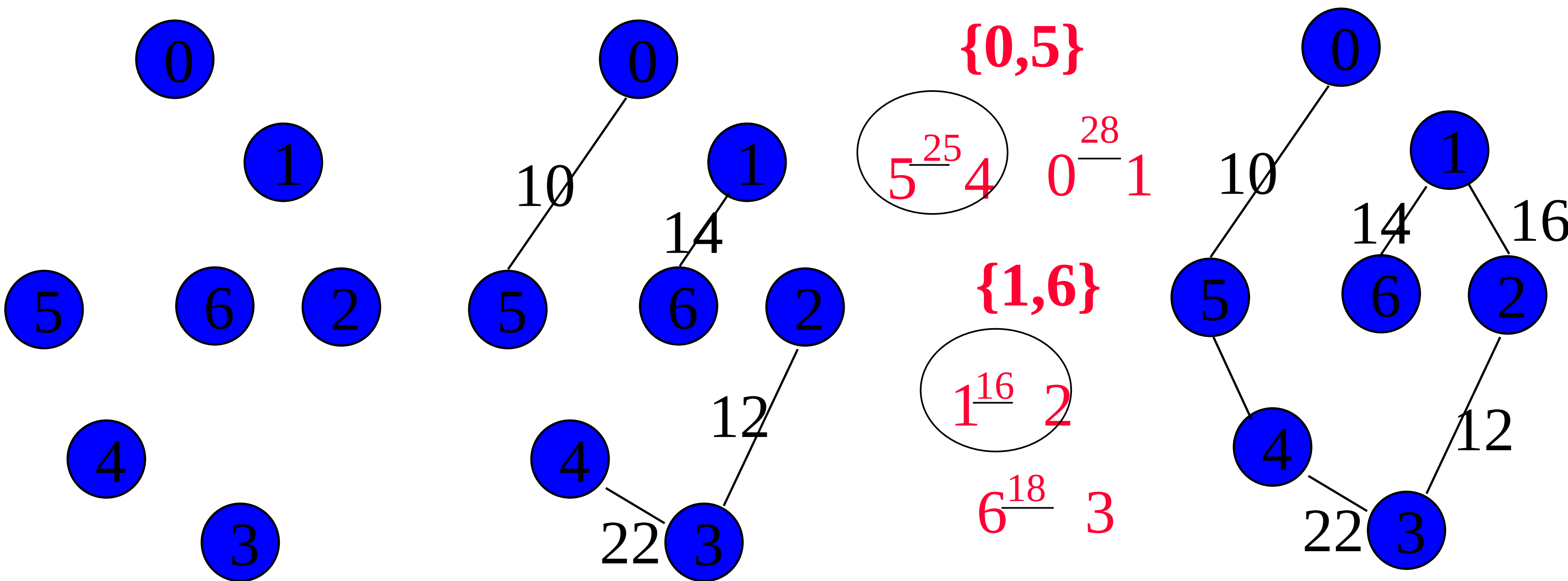



0
28
1
10
14
16
5
6
2
25
24
4
18
12
22
3
0
1
10
5
6
2
25
12
4
22
3
0
1
10
16
5
6
2
25
12
4
22
3
0
1
10
14
16
5
6
2
25
12
4
22
3



Sollin’s Algorithm

vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4

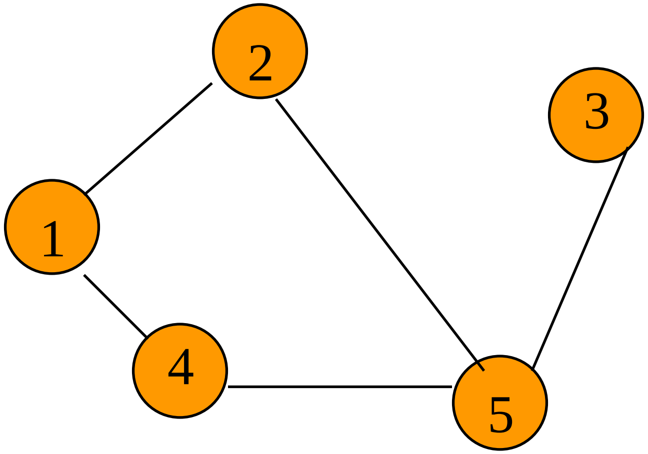


Graph Representation

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

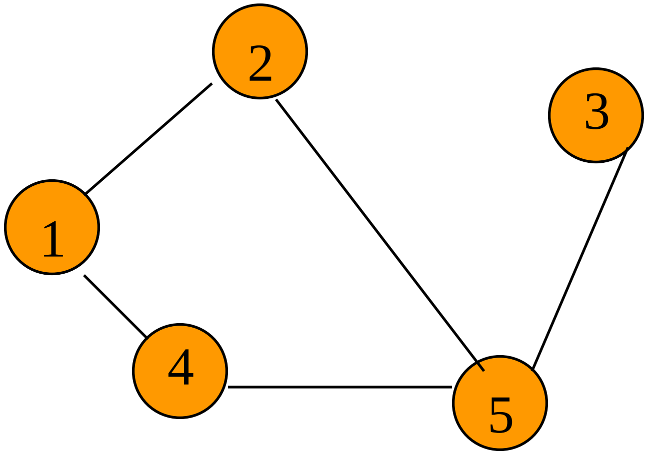
Adjacency Matrix

- 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

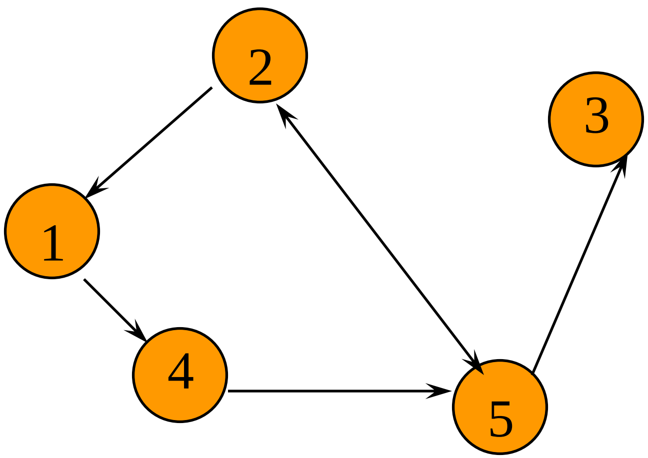
Adjacency Matrix Properties



	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

- 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

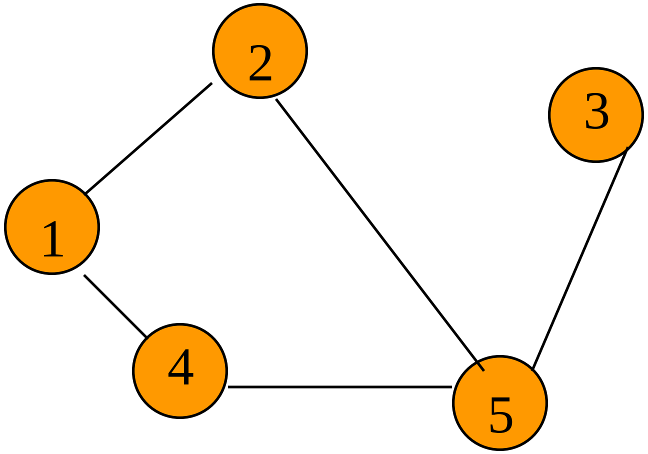
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

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 [i.e., $(n^2-n)/2$]
- $O(n)$ 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)

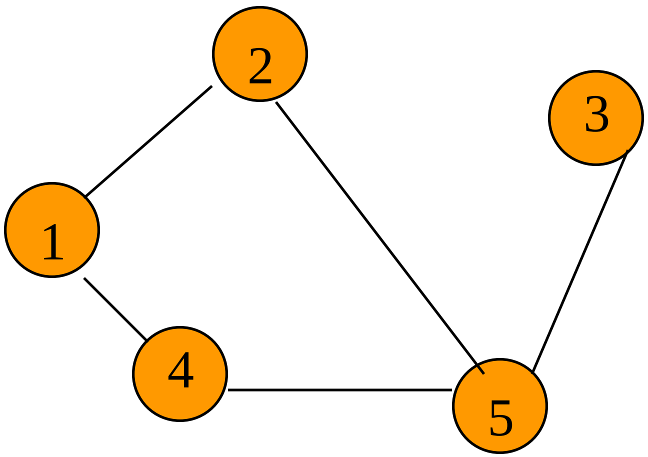
aList[3] = (5)

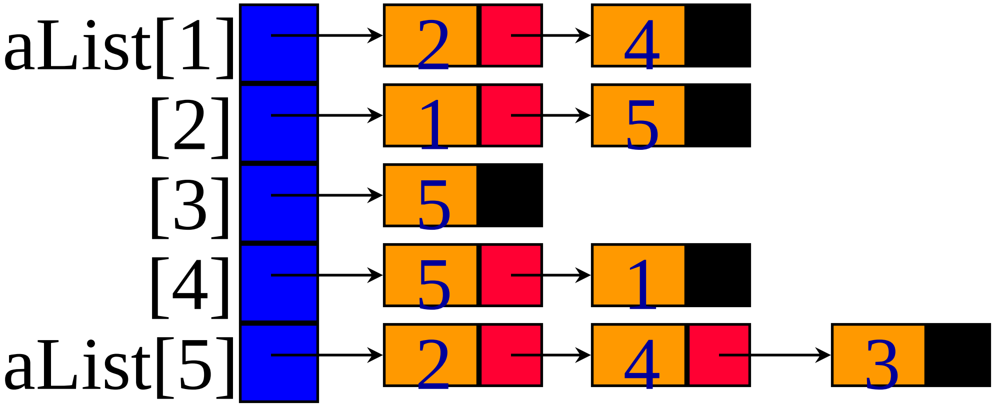
aList[4] = (5,1)

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

Linked Adjacency Lists

- Each adjacency list is a chain.





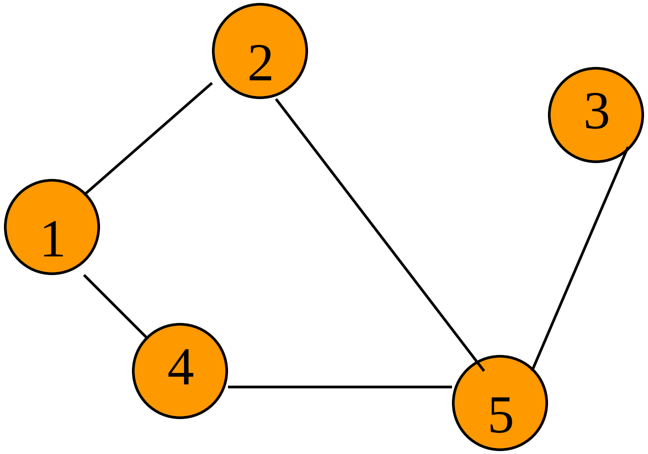
Array Length = n

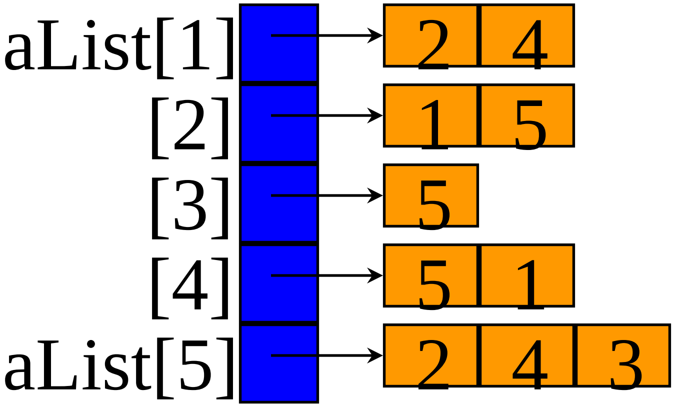
# of chain nodes = 2e (undirected graph)

# 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)

end

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

```
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);
```