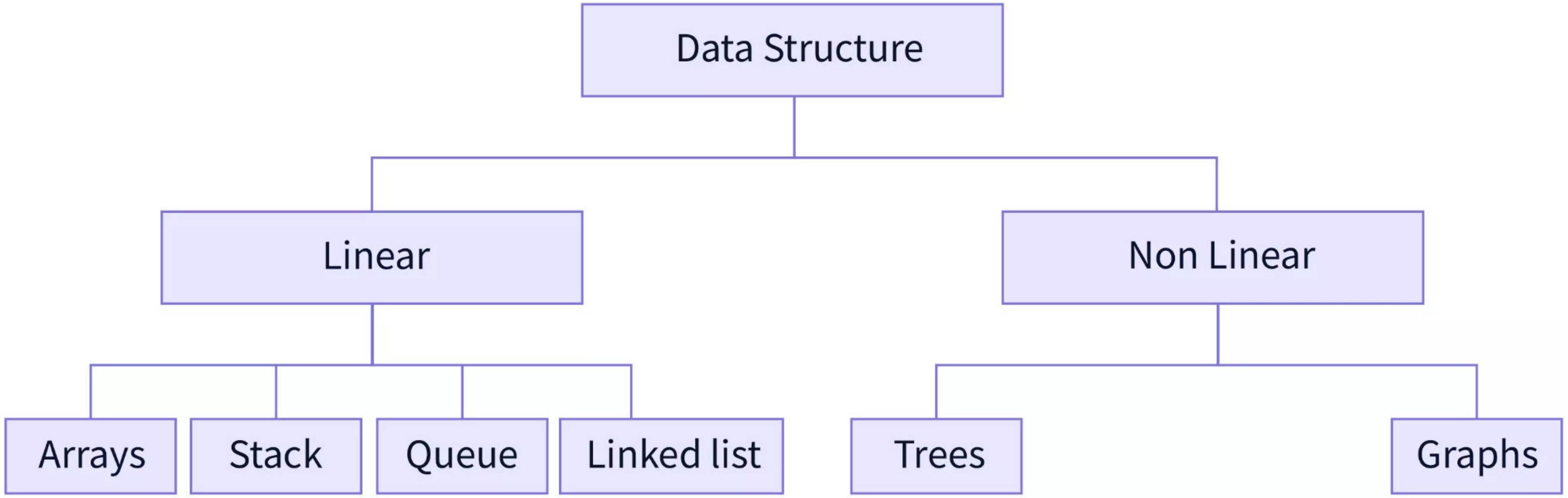
Design and Analysis of Algorithms

Data Structures



Das Bild kann nicht angezeigt werden.

Graph Data Structure

- A Graph is a non-linear data structure that consists of nodes (vertices) and edges which connects them.

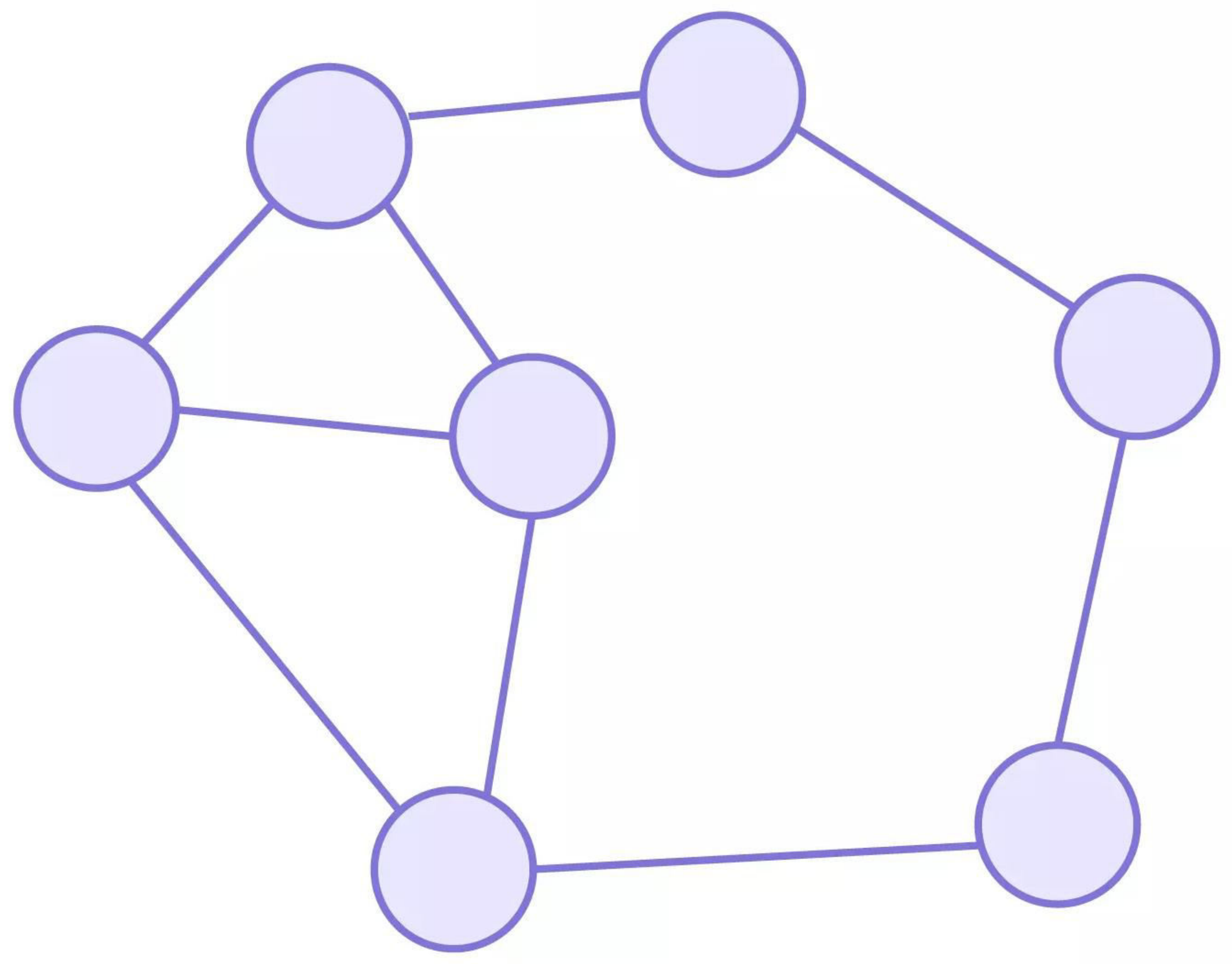
Das Bild kann nicht angezeigt werden.

Graph Data Structure

More precisely, a graph is a data structure (V, E) that consists of

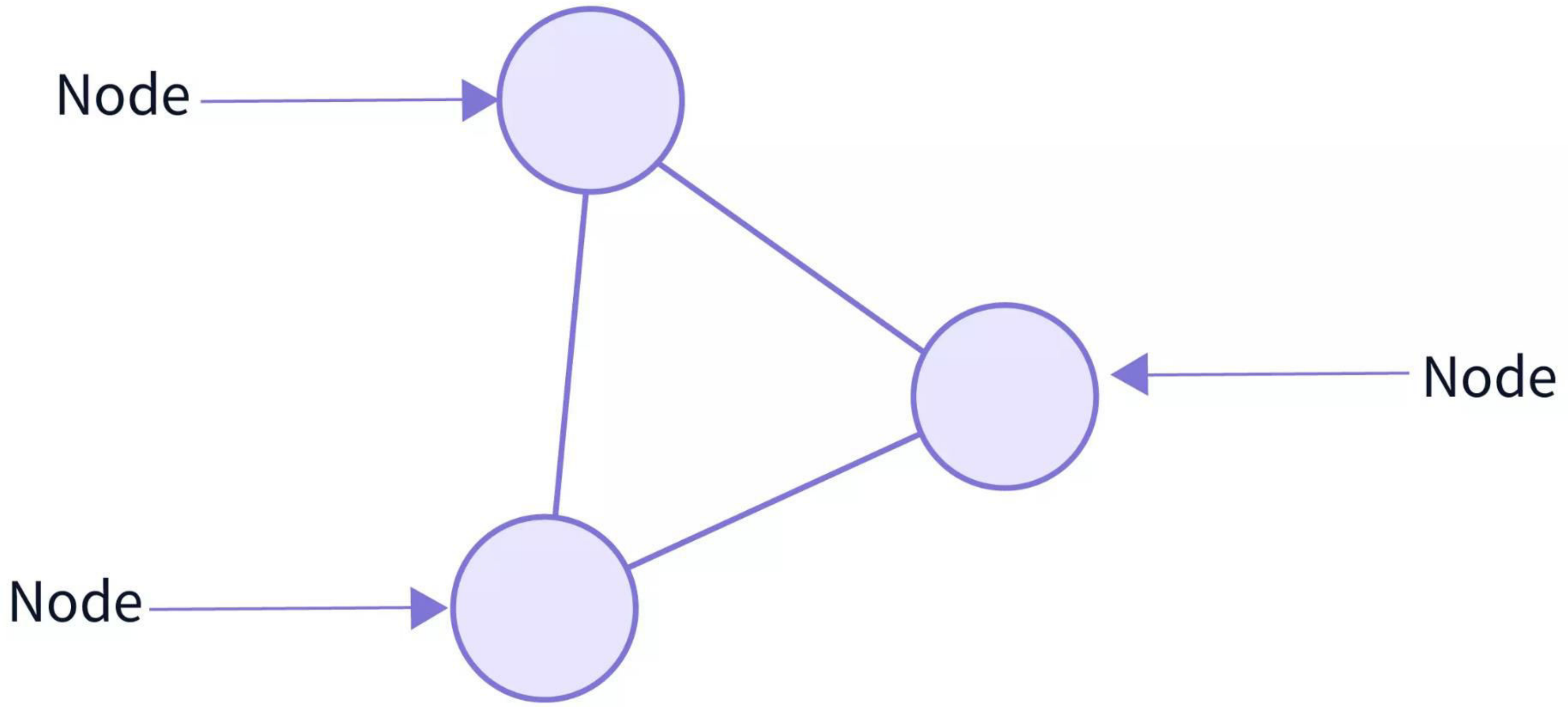
- A collection of vertices (V)
- A collection of edges (E), represented as ordered pairs (*u*, *v*)

Graph



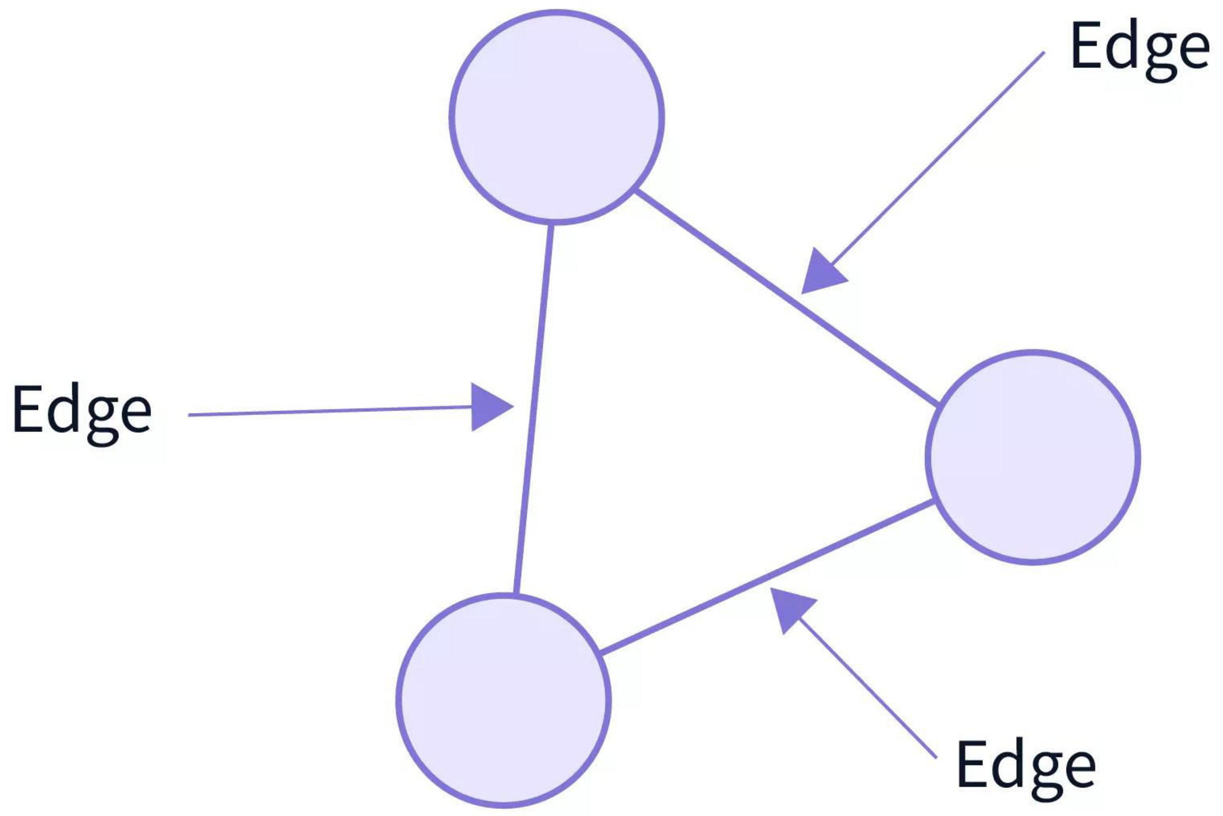
Nodes

- They are also called vertices.
- A node can represent anything such as any location, buildings, computers, etc.



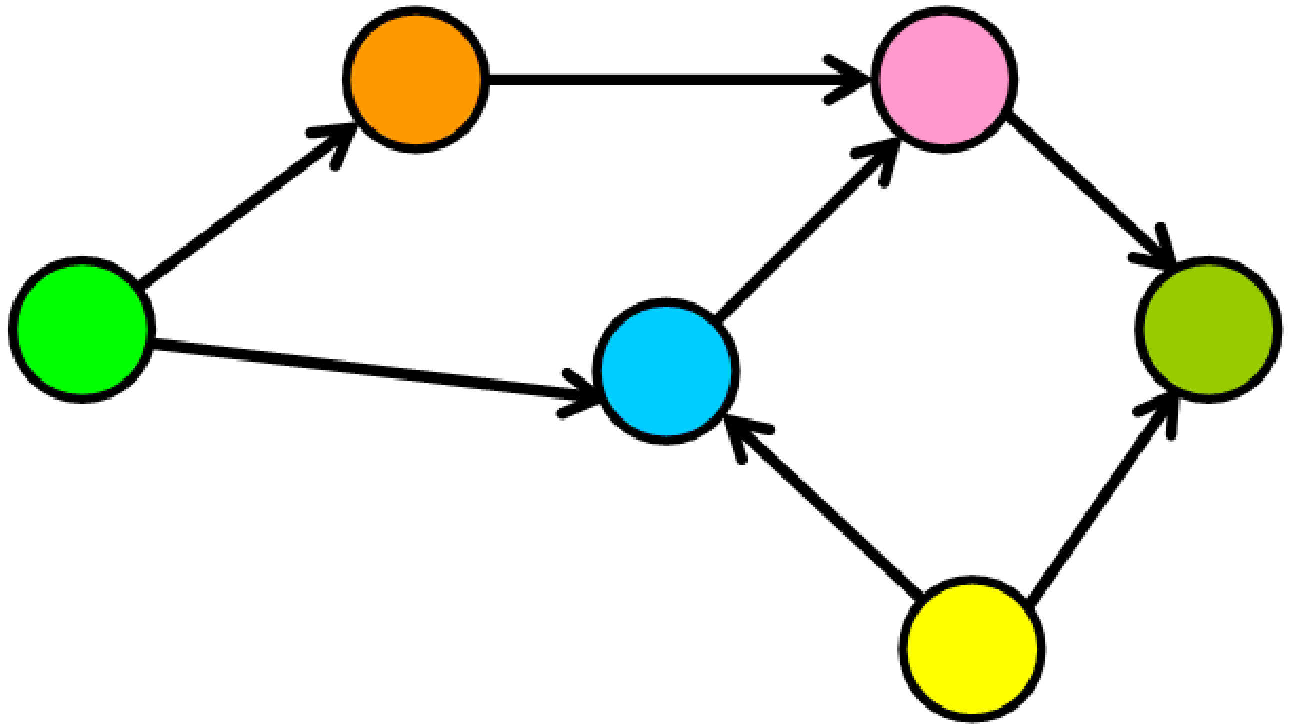
Edges

- Edges basically connects the nodes in a graph data structure.
- They represent the relationships between various nodes in a graph.



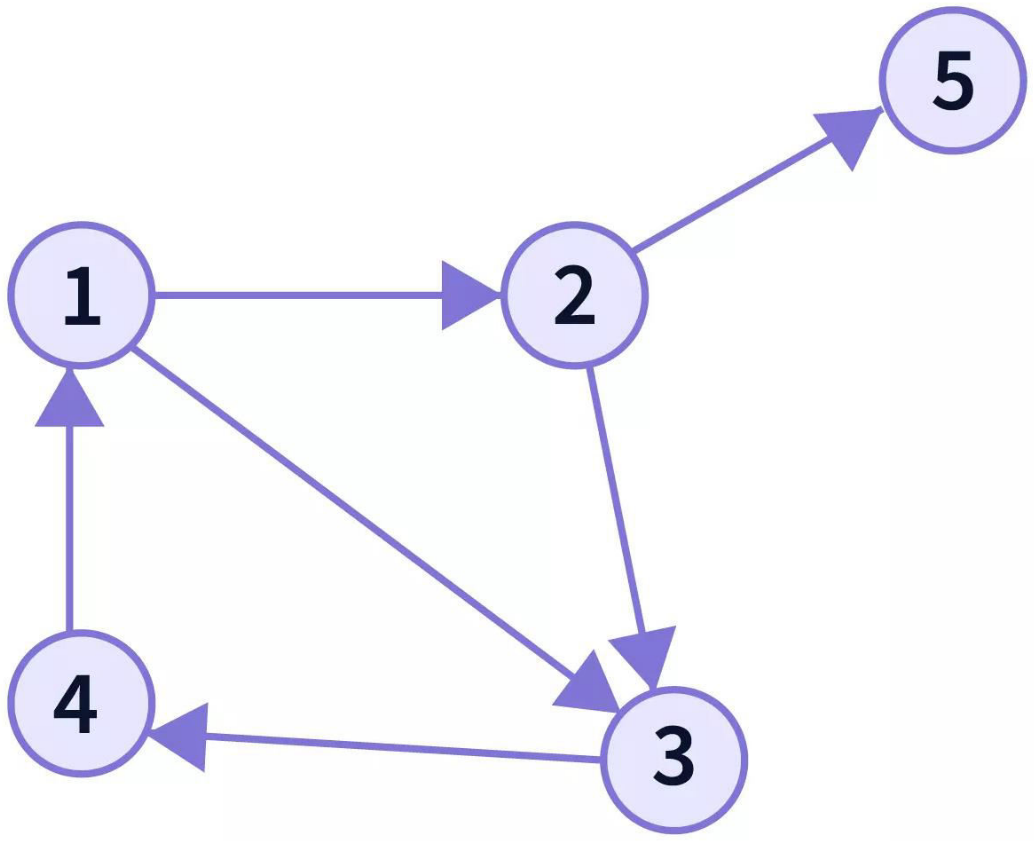
Vertices and Edges

- |V| = Total number of vertices in the graph.
- |E| = Total number of edges in the graph.
- In the example below, |V| = 6 and |E| = 7.



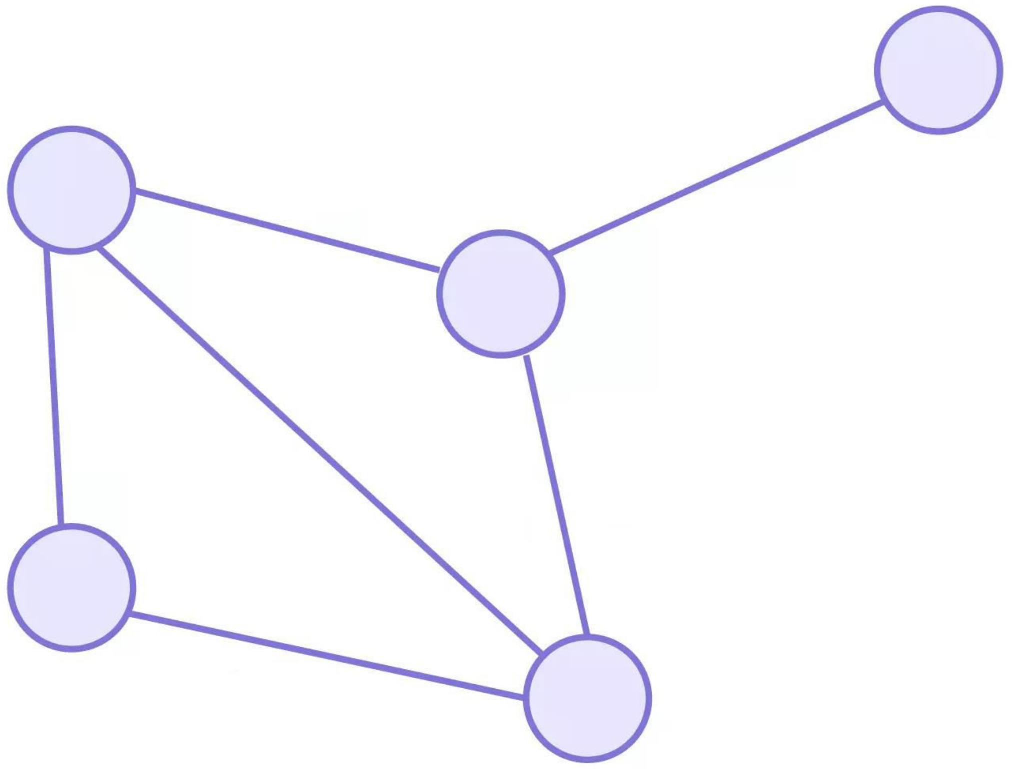
Directed Graphs

- Directed graphs are the graphs where the edges have directions from one node towards the other node.



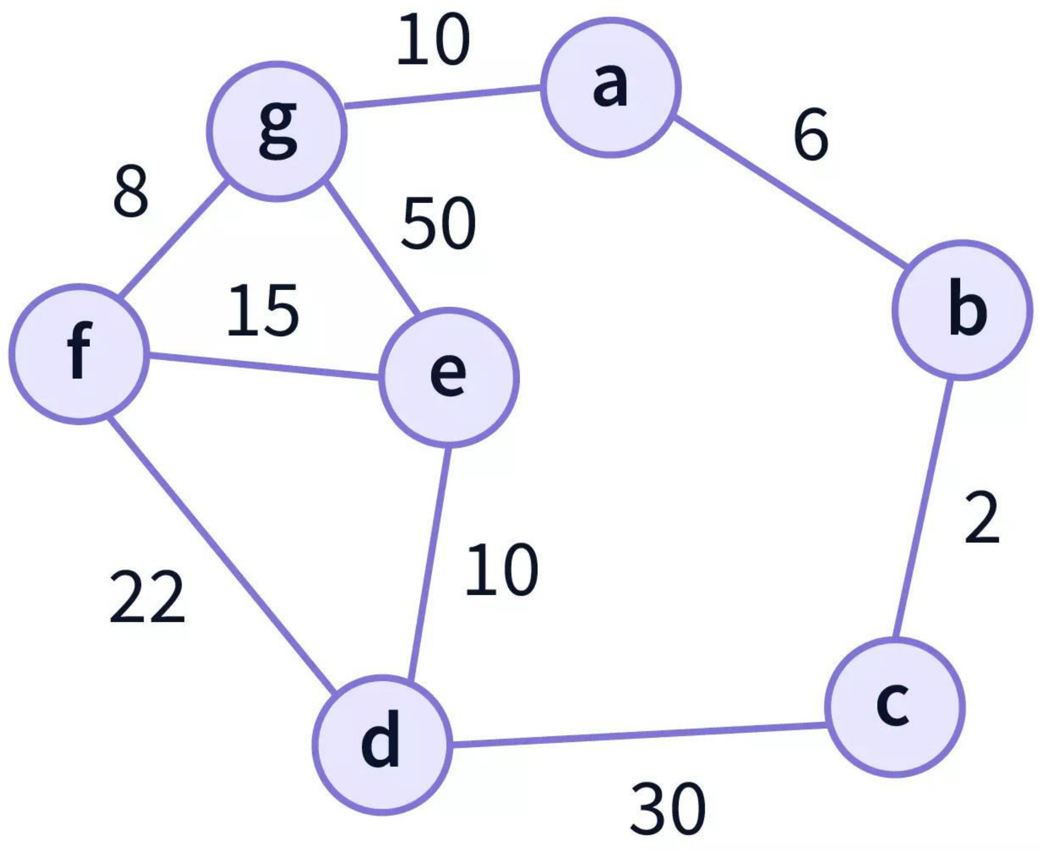
Undirected Graphs

- Undirected graphs have edges that do not have a direction.
- Hence, the graph can be traversed in either direction.



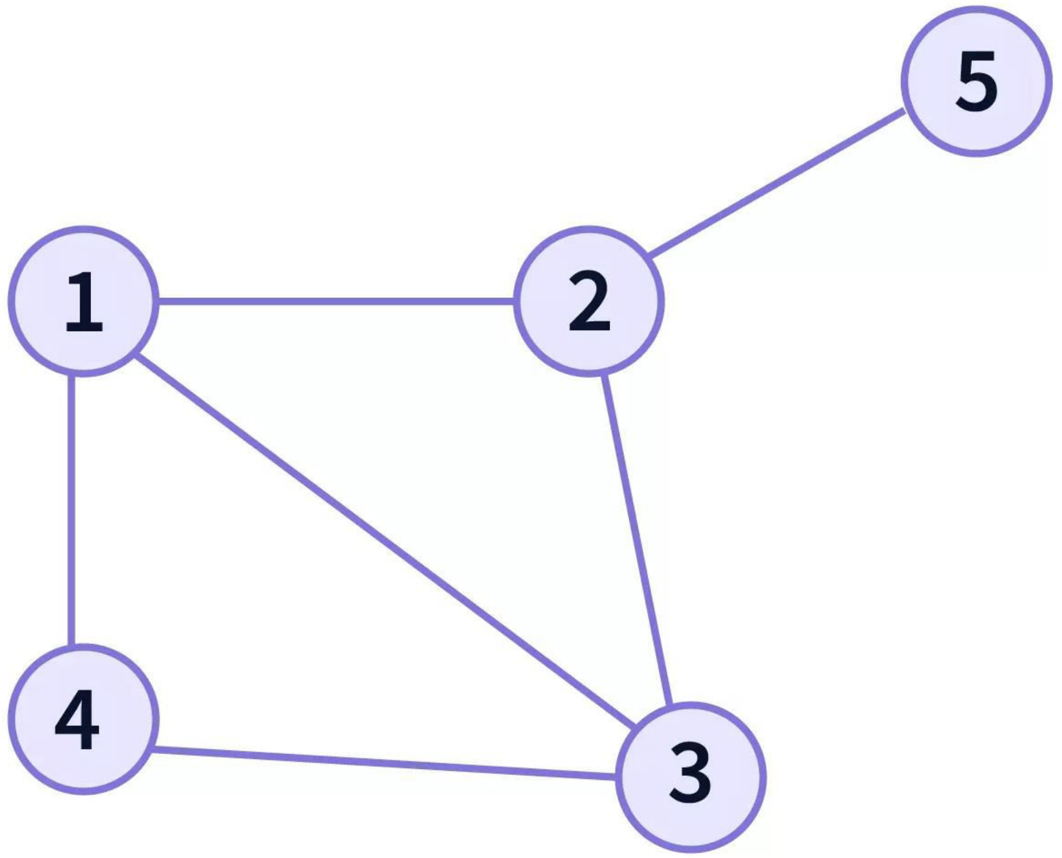
Weighted Graph

- A weighted graph is a graph with numbers assigned to its edges.
- These numbers are called weights or costs.



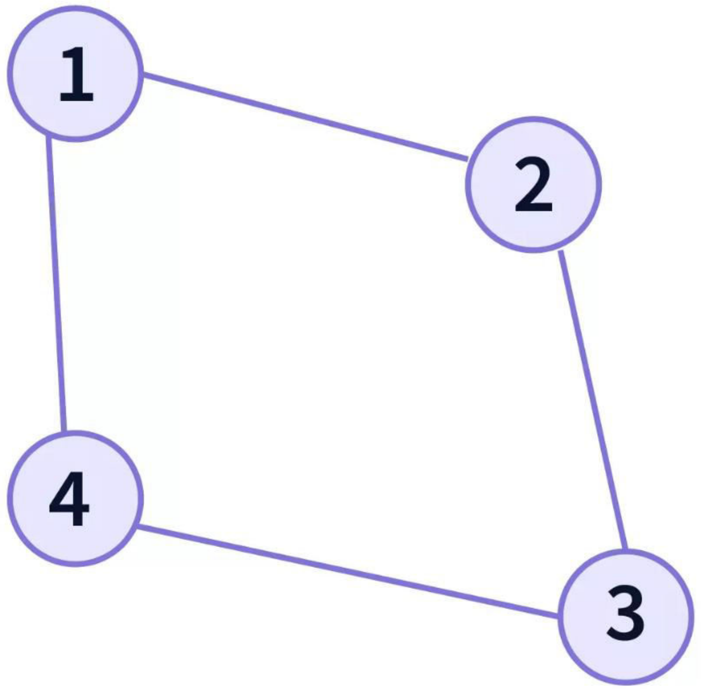
Unweighted Graph

- An unweighted graph is a graph with no edge weight.

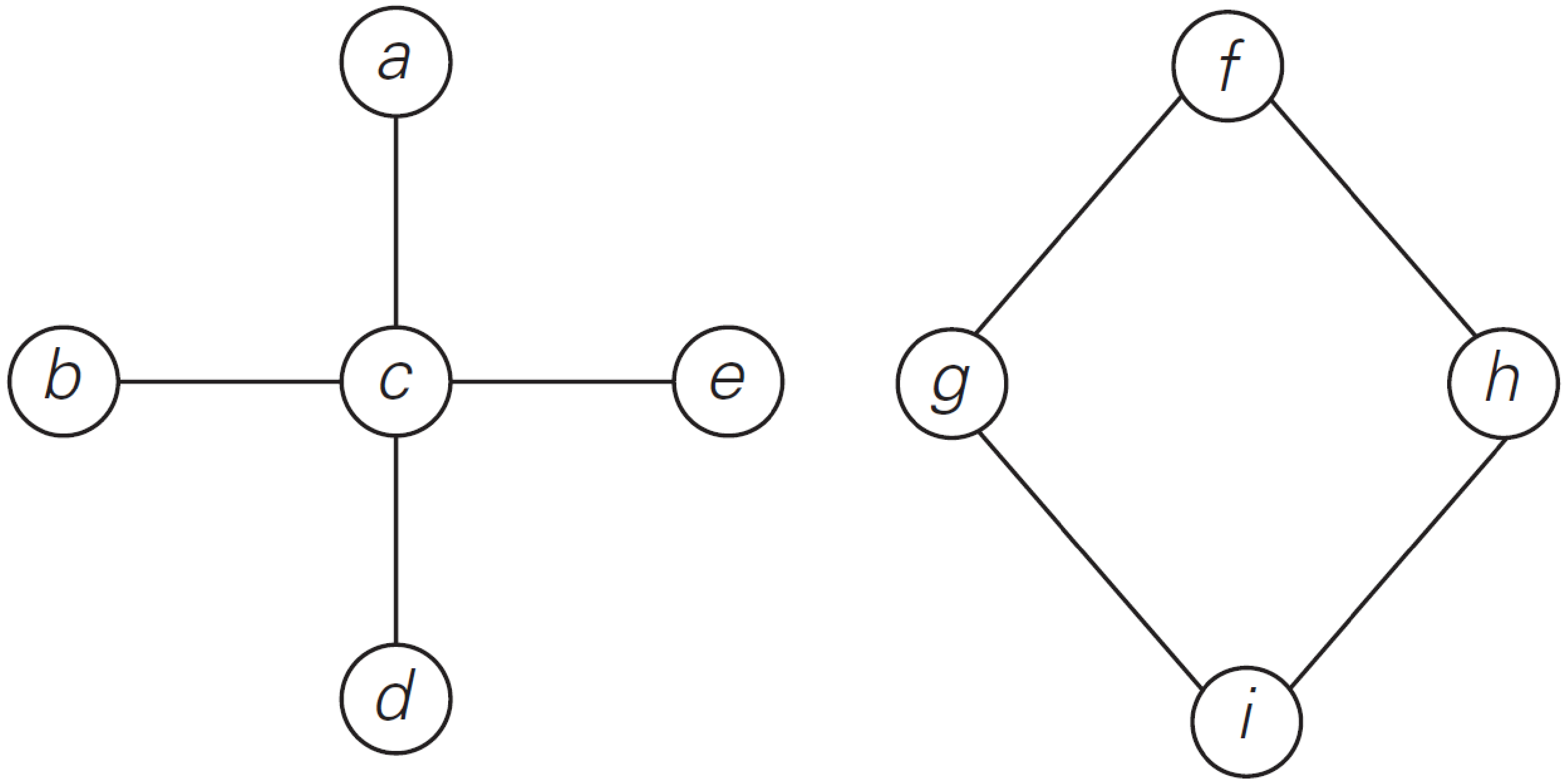


Connected Graph

- A connected graph is a graph in which there is always a path from a vertex to any other vertex.

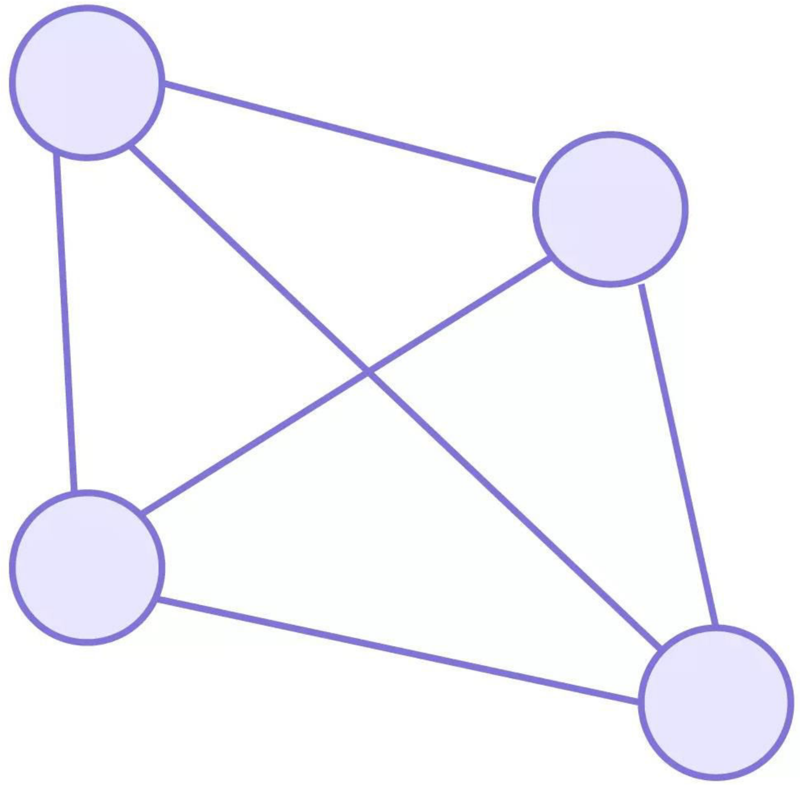


Disconnected Graph

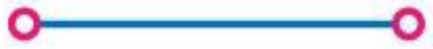


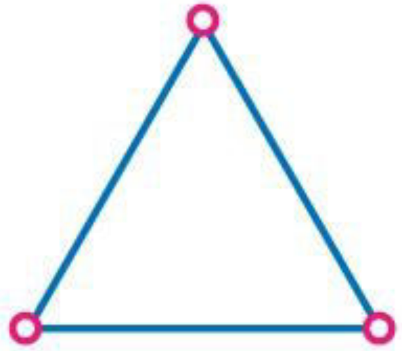
Complete Graph

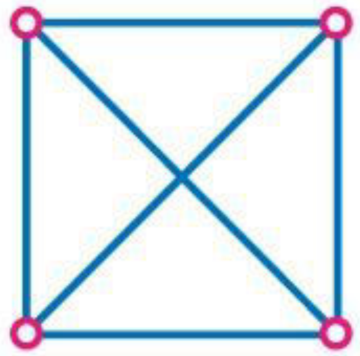
- In a complete graph, there is an edge between every single pair of node in the graph.
- Hence, every vertex has an edge to all other vertices.

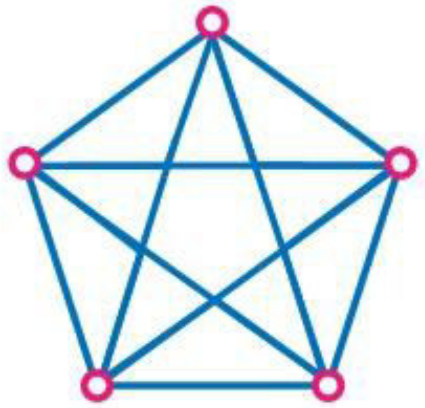


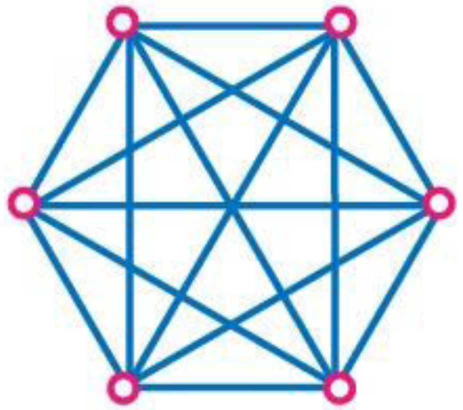
Complete Graph

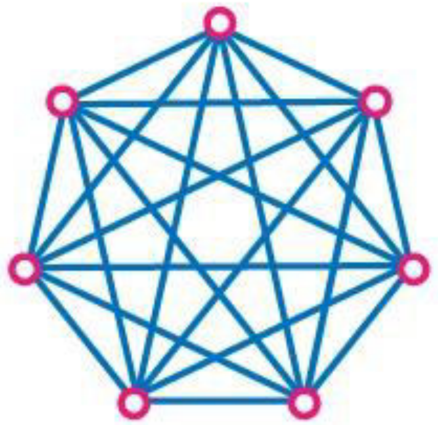

K_2


K_3

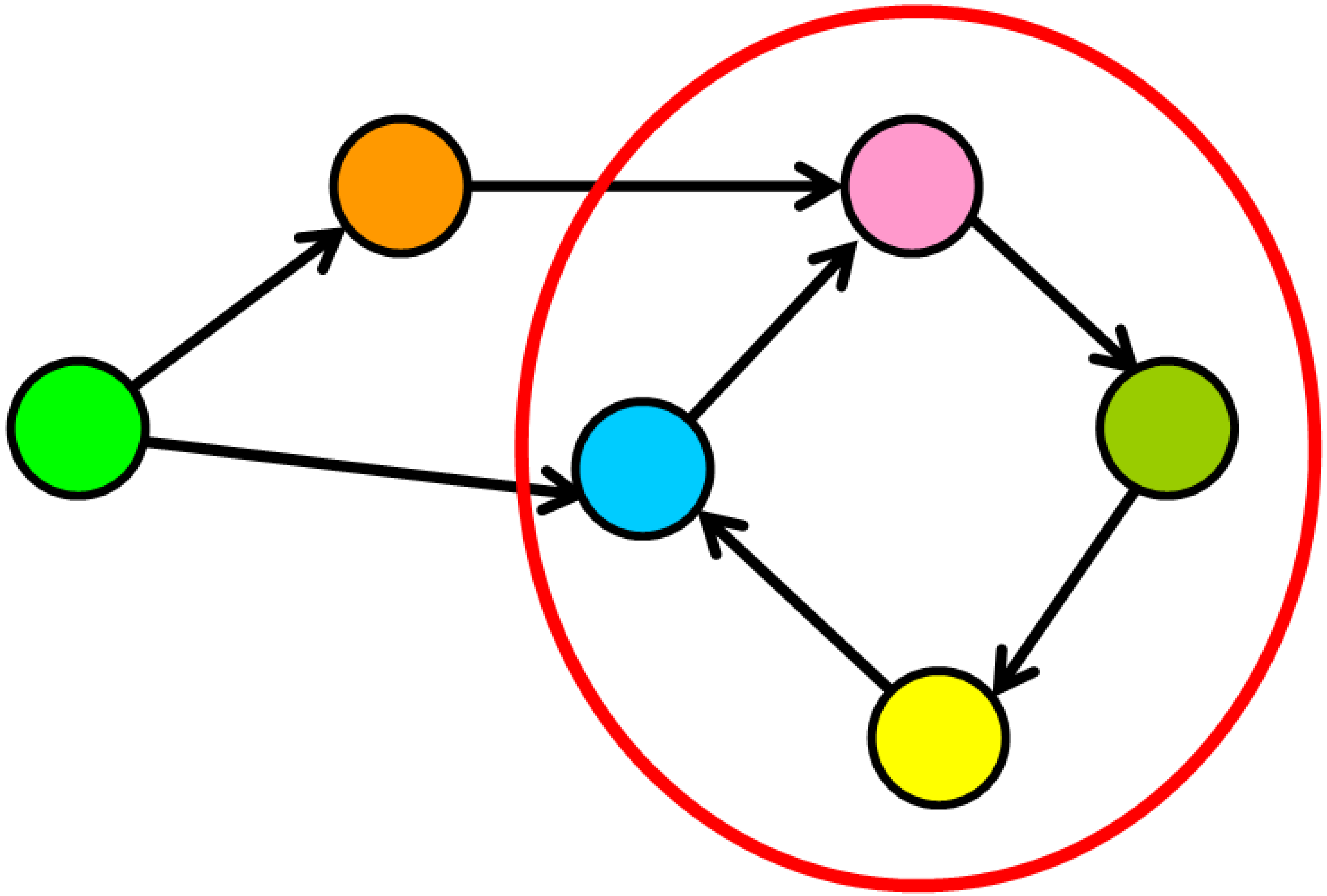

K_4


K_5


K_6

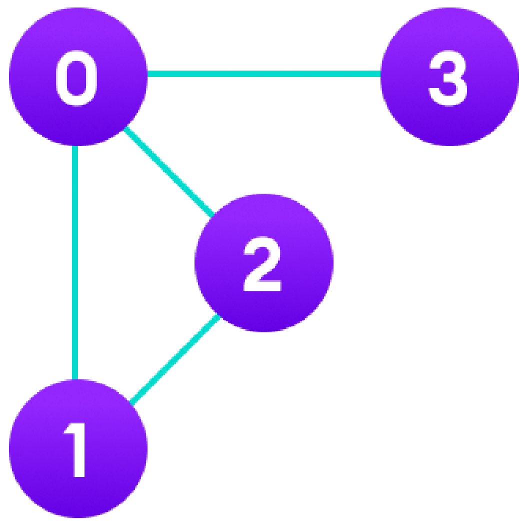

K_7

Cycles



Graph Representation: Adjacency Matrix

- An adjacency matrix is a 2D array of V×V vertices.
- Each row and column represent a vertex.



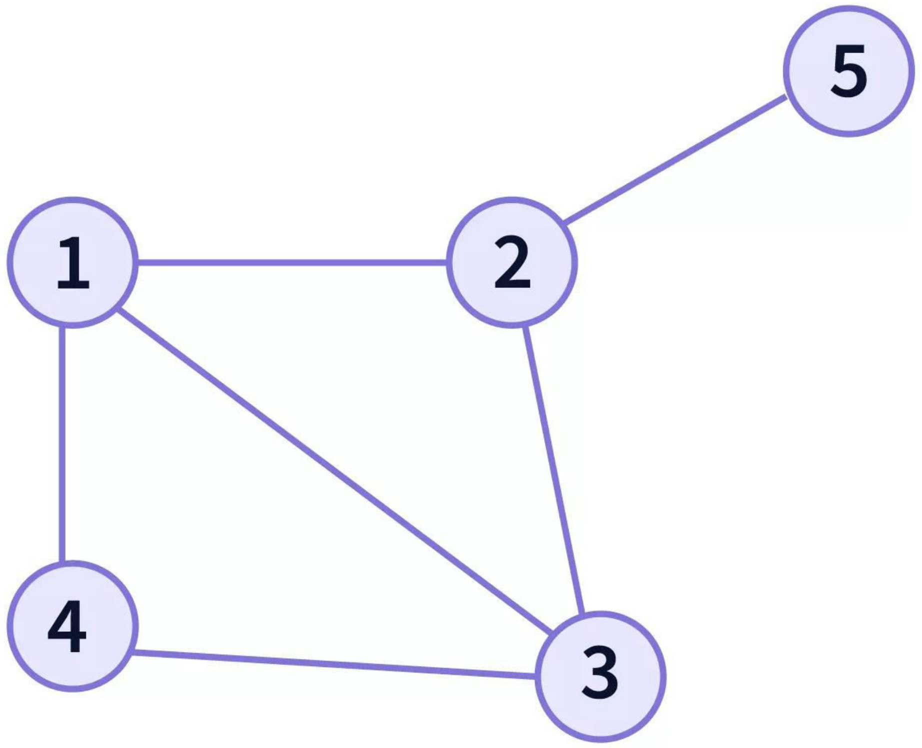
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph Representation: Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

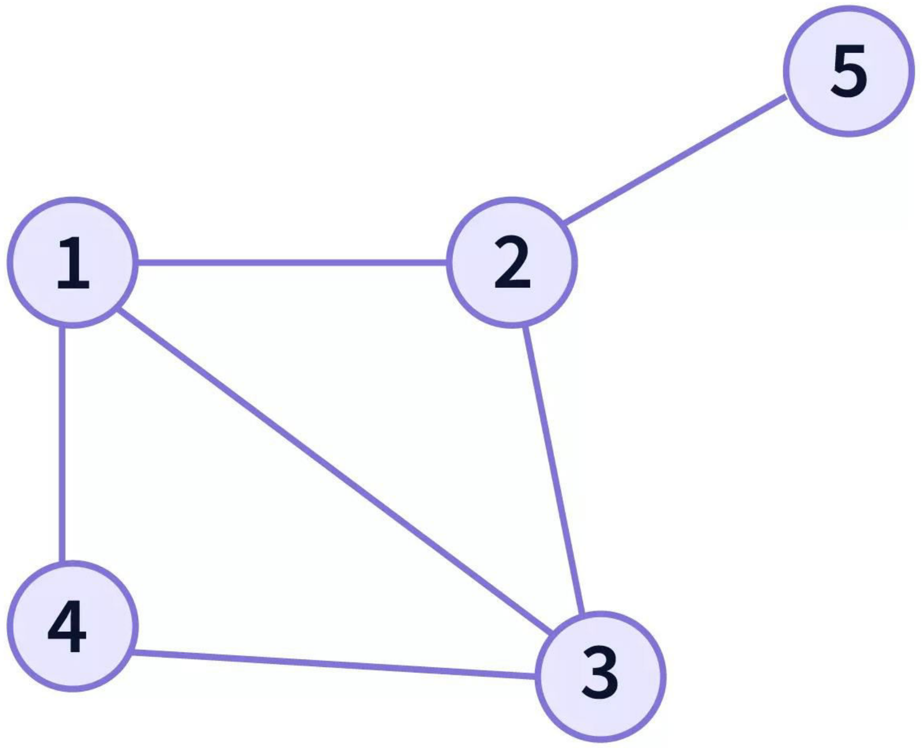
Das Bild kann nicht angezeigt werden.

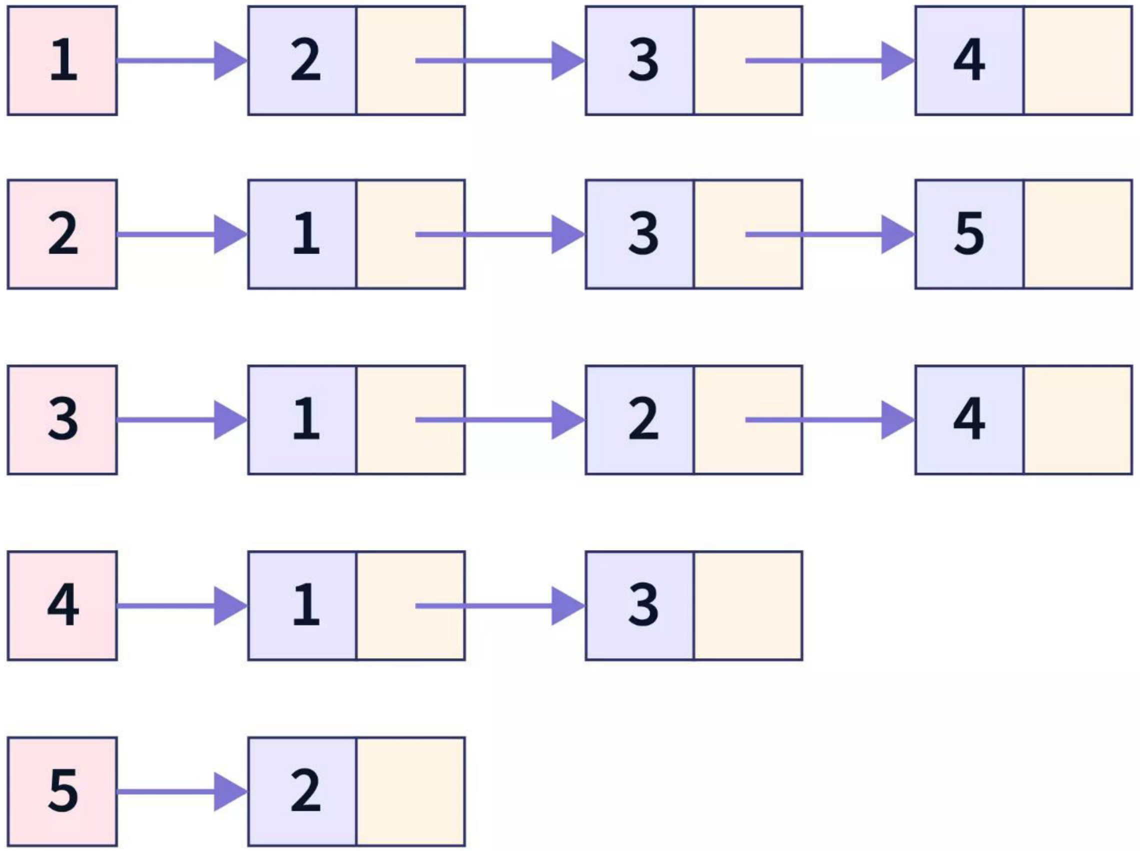
Graph Representation: Example 1



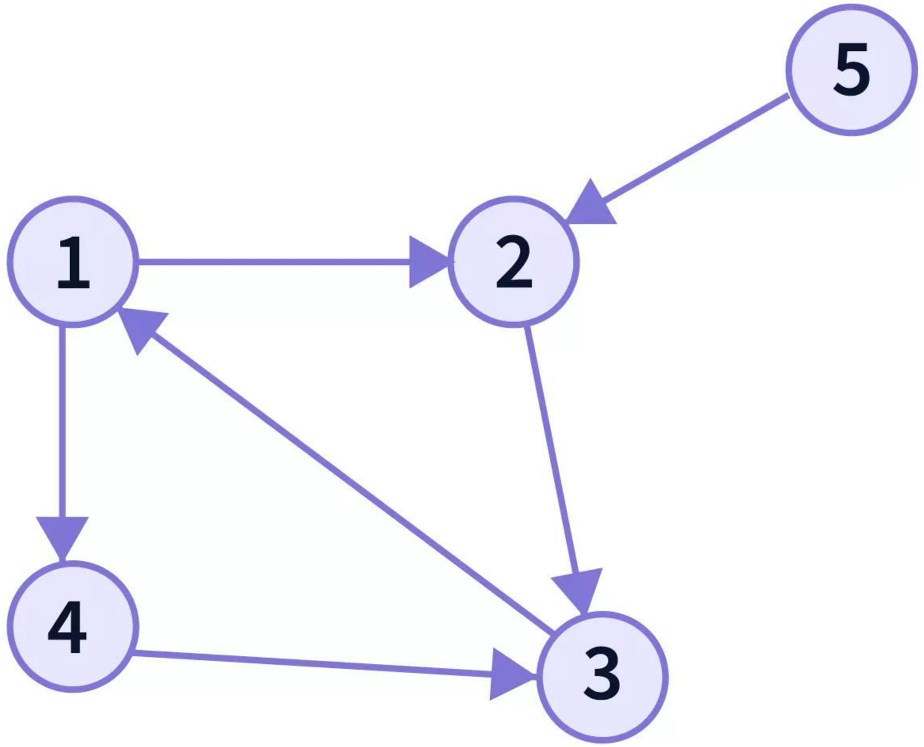
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	1
3	1	1	0	1	0
4	1	0	1	0	0
5	0	1	0	0	0

Graph Representation: Example 1



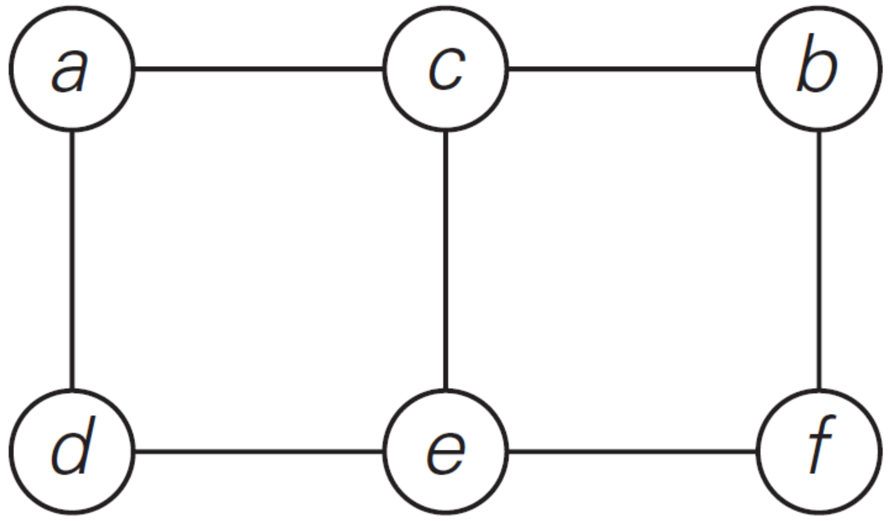


Graph Representation: Example 2



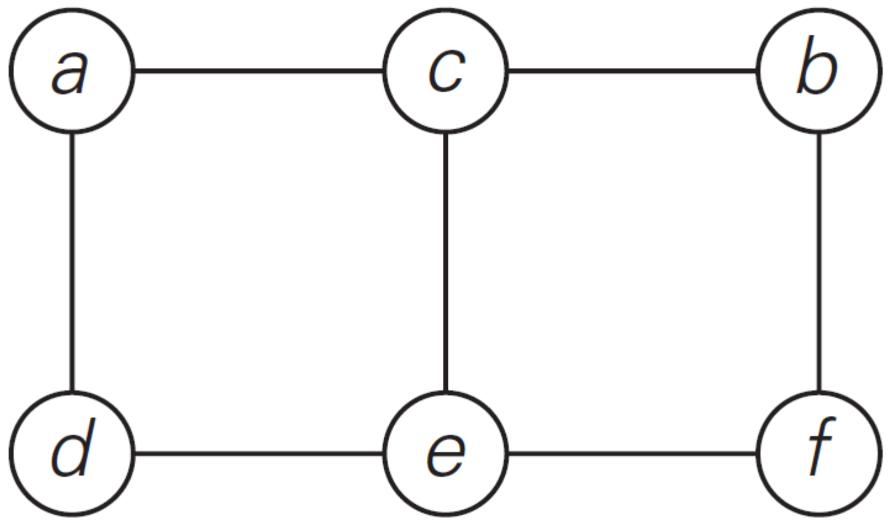
	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	1	0	0
5	0	1	0	0	0

Graph Representation: Example 3



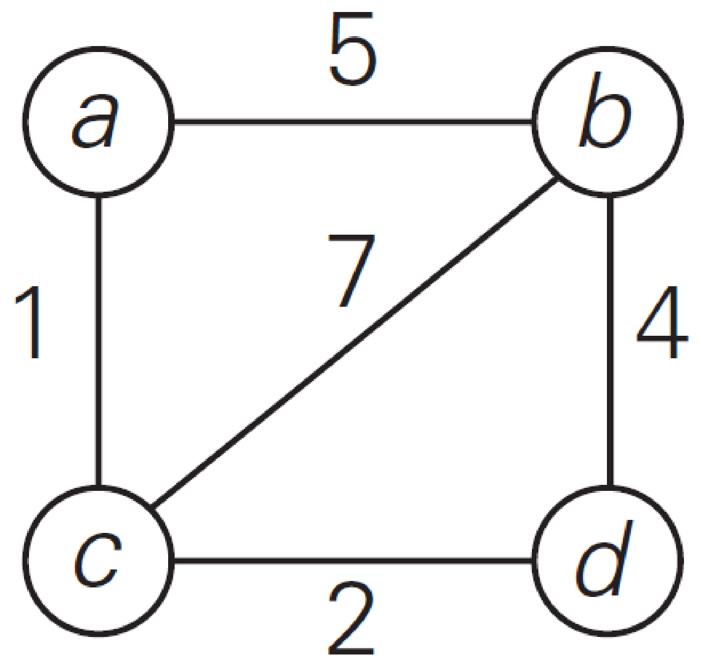
$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 0 & 0 & 1 & 1 & 0 & 1 \\ f & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

Graph Representation: Example 3



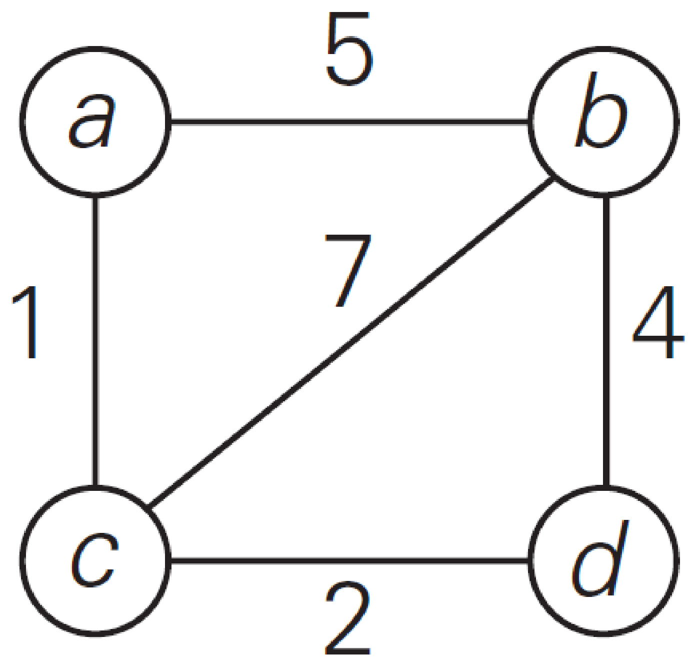
$$
\begin{array}{|c|}
\hline
a \\ \hline
b \\ \hline
c \\ \hline
d \\ \hline
e \\ \hline
f \\ \hline
\end{array}
\begin{array}{l}
\rightarrow c \rightarrow d \\
\rightarrow c \rightarrow f \\
\rightarrow a \rightarrow b \rightarrow e \\
\rightarrow a \rightarrow e \\
\rightarrow c \rightarrow d \rightarrow f \\
\rightarrow b \rightarrow e
\end{array}
$$

Graph Representation: Example 4



$$
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & \infty & 5 & 1 & \infty \\
b & 5 & \infty & 7 & 4 \\
c & 1 & 7 & \infty & 2 \\
d & \infty & 4 & 2 & \infty
\end{array}
$$

Graph Representation: Example 4



a	$\rightarrow b, 5 \rightarrow c, 1$
b	$\rightarrow a, 5 \rightarrow c, 7 \rightarrow d, 4$
c	$\rightarrow a, 1 \rightarrow b, 7 \rightarrow d, 2$
d	$\rightarrow b, 4 \rightarrow c, 2$

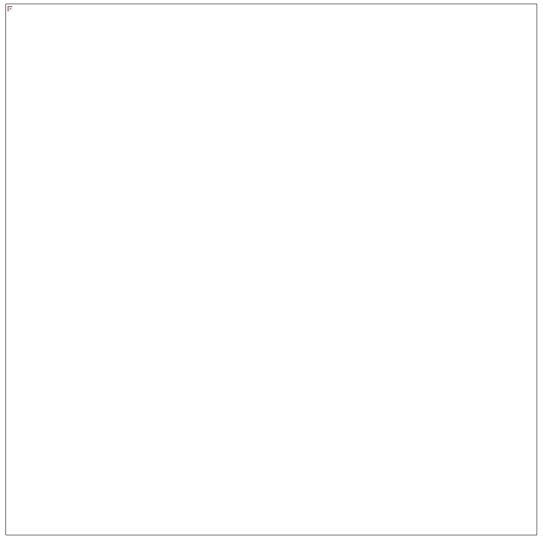
Graph Algorithms

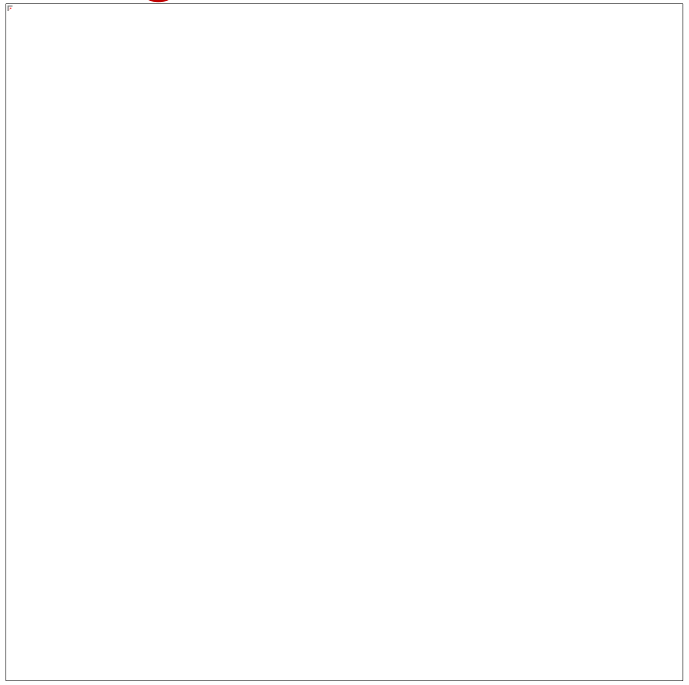
- Breadth First Search (BFS)
- Depth First Search (DFS)
- Dijkstra’s Algorithm
- Floyd-Warshall Algorithm
- Bellman-Ford Algorithm
- Ford Fulkerson Algorithm
- **Prim’s Algorithm**
- **Kruskal’s Algorithm**

Das Bild kann nicht angezeigt werden.

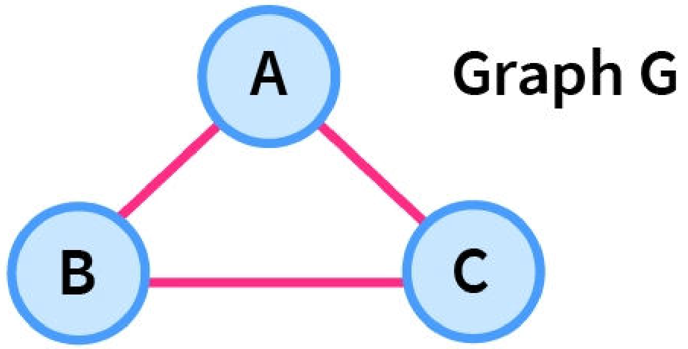
Spanning Tree

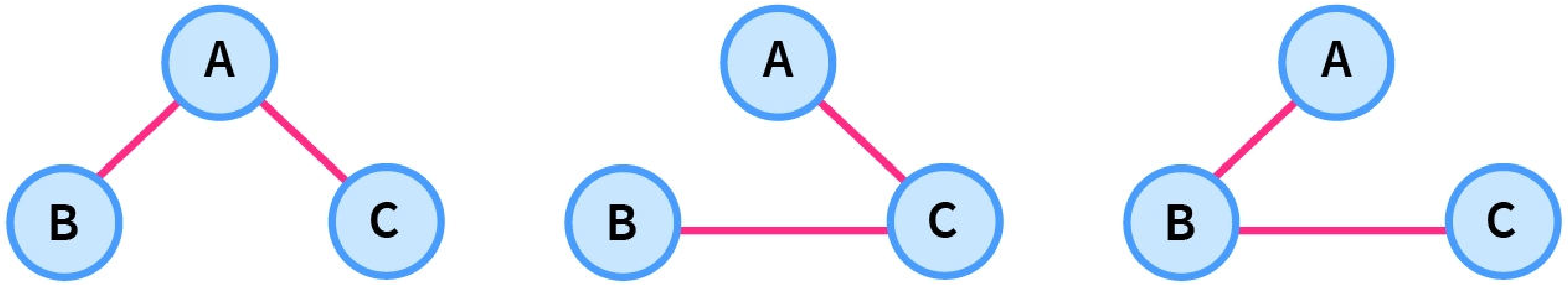
- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.
- If a vertex is missed, then it is not a spanning tree.





Spanning Tree: Example 1

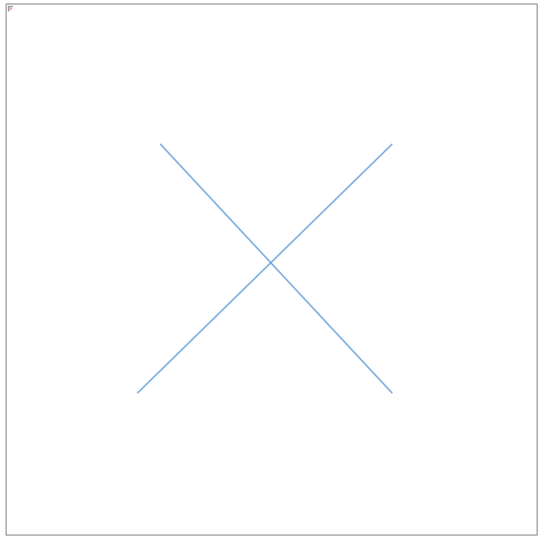




Spanning Trees, subgraph of G

Spanning Tree: Example 2

- The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n - 2)}$.
- If we have $n = 4$, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$.



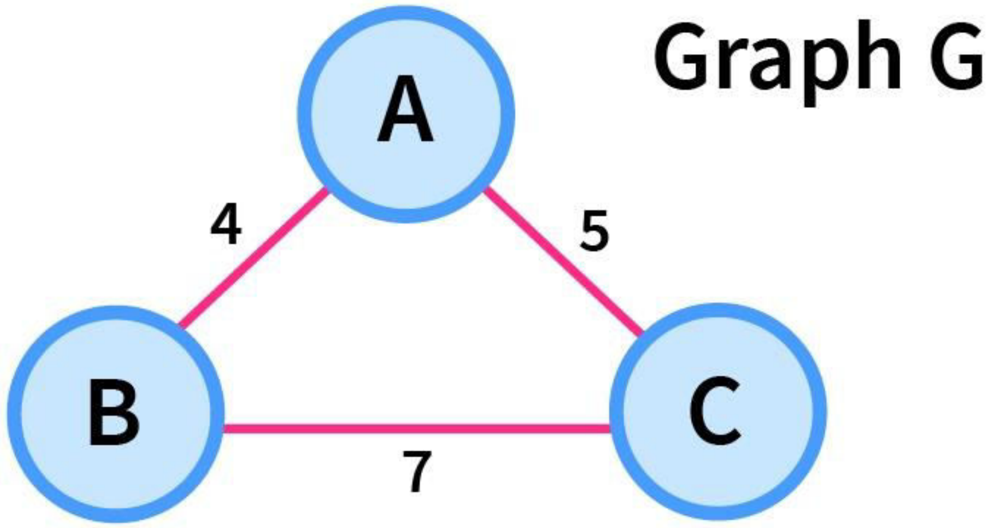
Das Bild kann nicht angezeigt werden.

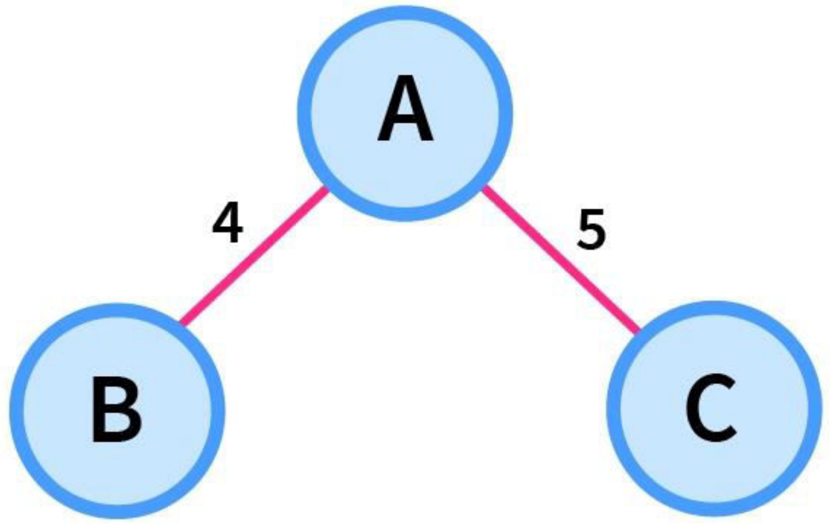
Spanning Tree: Example 2

Das Bild kann nicht angezeigt werden.

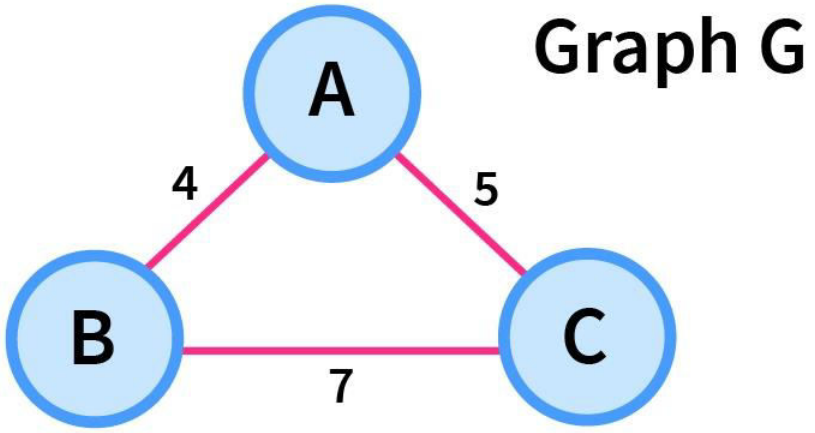
Minimum Spanning Tree (MST)

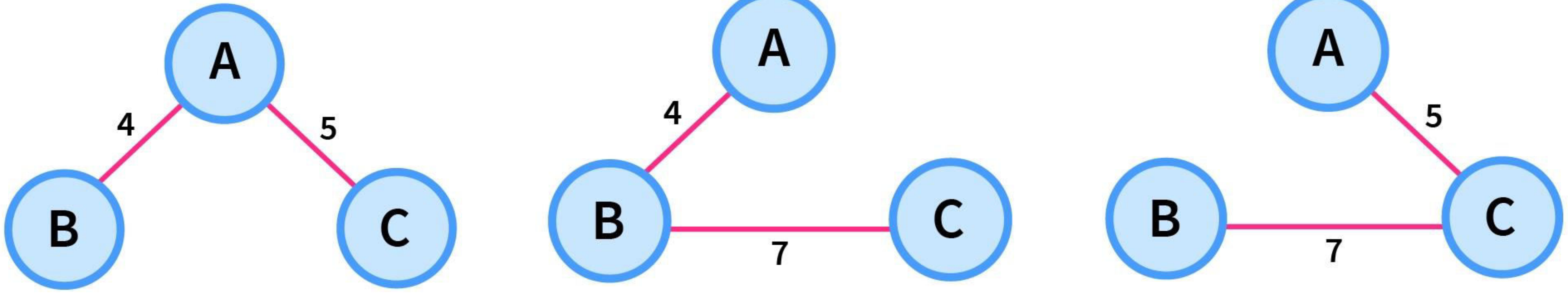
- A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.





Minimum Spanning Tree (MST): Example 1



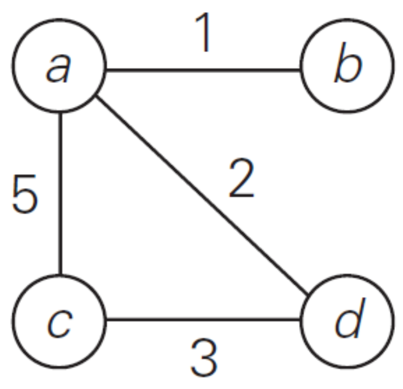


Das Bild kann nicht angezeigt werden.

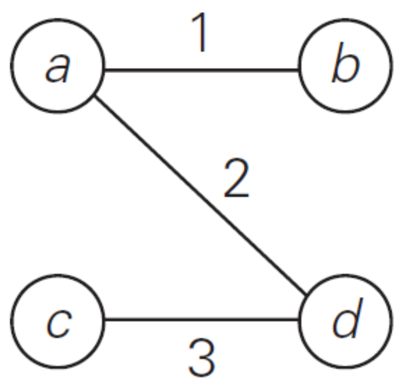
Minimum Spanning Tree (MST): Example 2

Das Bild kann nicht angezeigt werden.

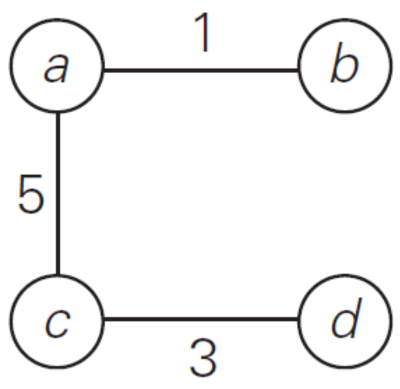
Minimum Spanning Tree (MST): Example 3



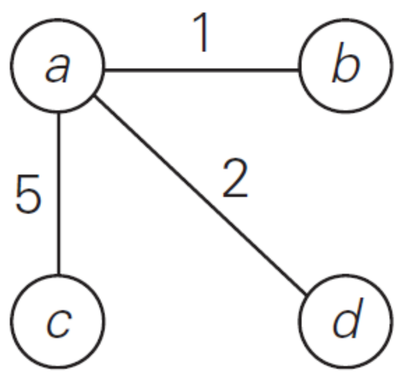
graph



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

Exhaustive Search

- If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles.
- First, the number of spanning trees grows exponentially.
- Second, generating all spanning trees for a given graph is not easy.

Properties of Spanning Tree

- Spanning trees do not have any cycles.
- A connected graph G can have more than one spanning tree.
- A Spanning tree always contains $n - 1$ edges, where n is the total number of vertices in the graph G.
- A Spanning tree is a minimally connected sub-graph.

Algorithm Design Techniques

- **Brute Force and Exhaustive Search**
- **Decrease-and-Conquer**
- **Divide-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- **Greedy Technique**
- Dynamic Programming
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Greedy Technique

- The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step, the choice made must be:

  Feasible: it has to satisfy the problem's constraints.

  Locally Optimal: it has to be the best local choice among all feasible choices available on that step.

  Irrevocable: once made, it cannot be changed later.

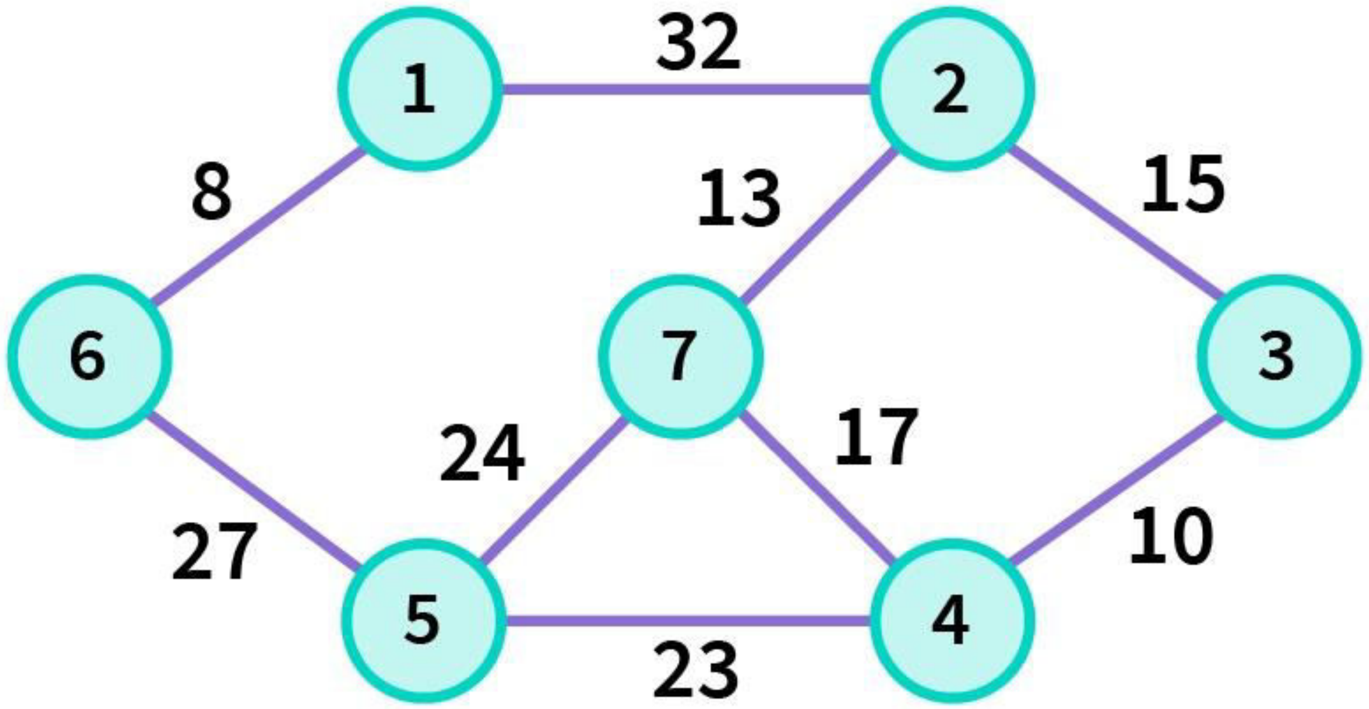
Prim’s Algorithm

- Prim’s algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- On each iteration, the algorithm expands the current tree in the greedy manner.
- The algorithm stops after all the graph’s vertices have been included in the tree being constructed.
- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in

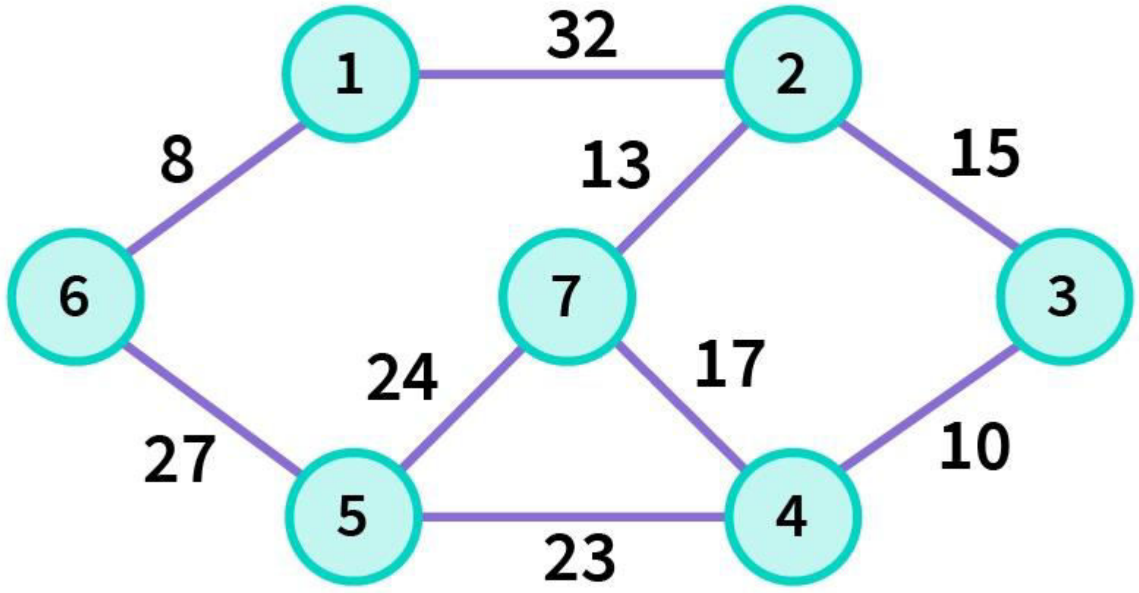
$$O(|E| \log |V|)$$

Prim’s Algorithm: Example 1

Prim's algorithm can be implemented using Fibonacci heap and it never accepts cycles.

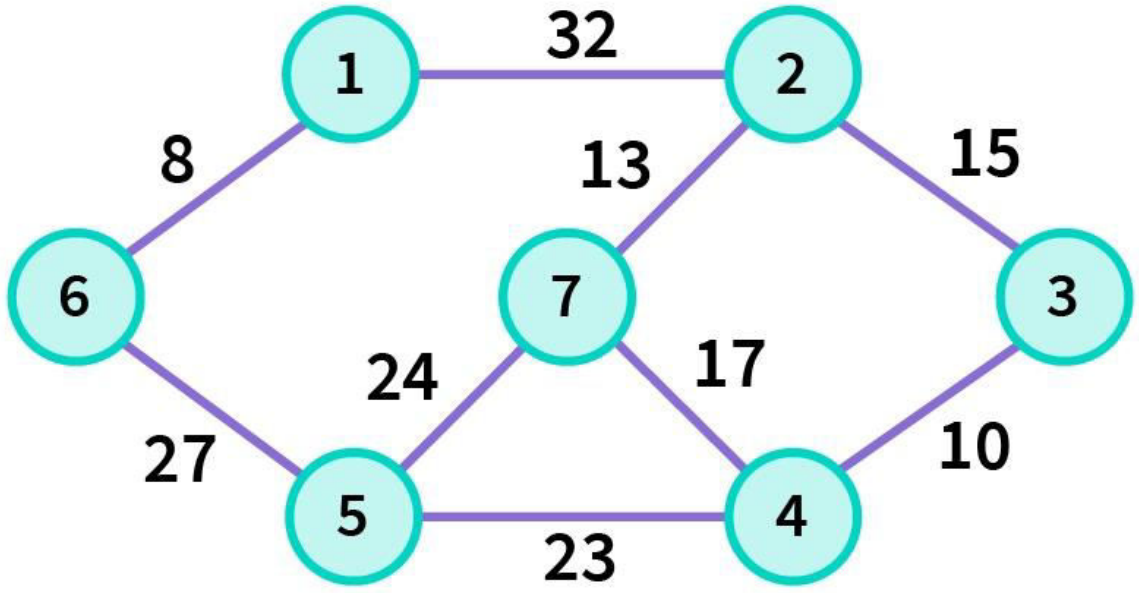


Prim’s Algorithm: Example 1



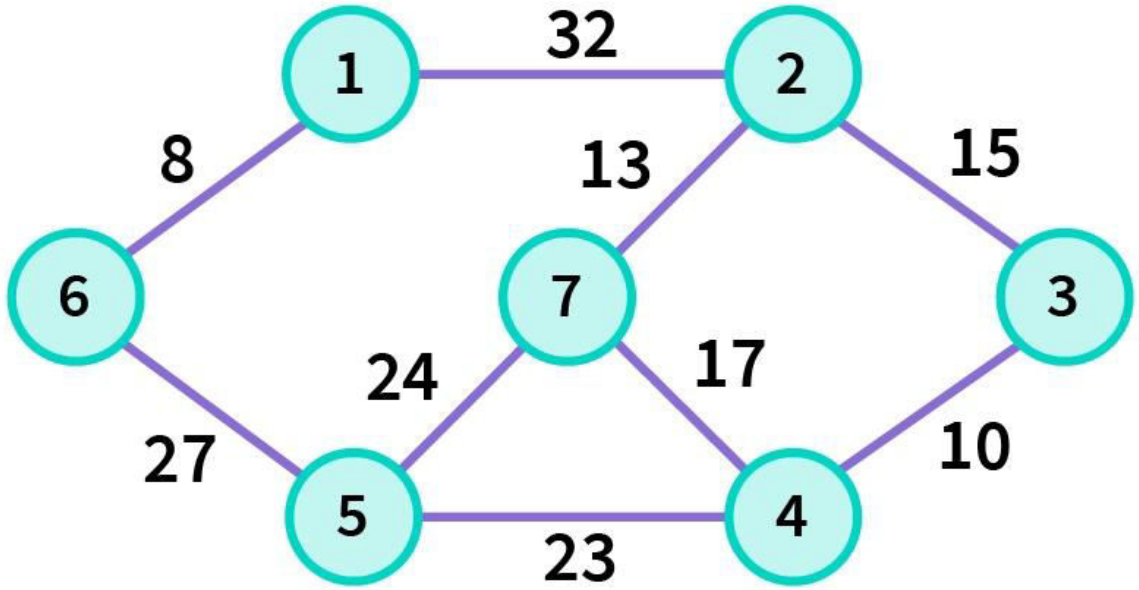
1	2	3	4	5	6	7
∞	∞	∞	∞	∞	∞	∞

Prim’s Algorithm: Example 1



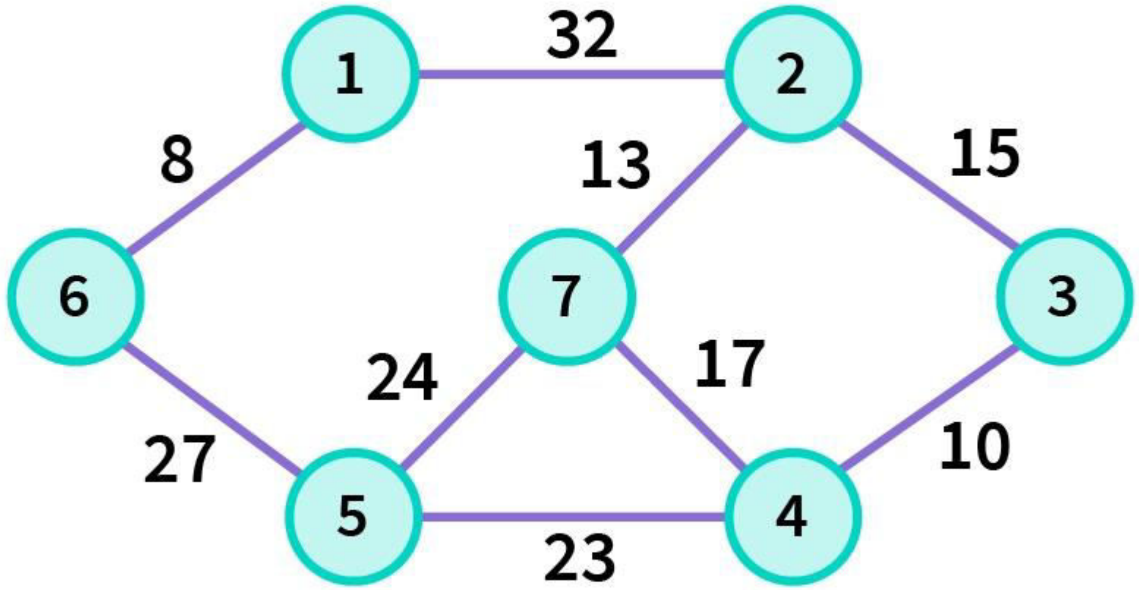
1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞

Prim’s Algorithm: Example 1



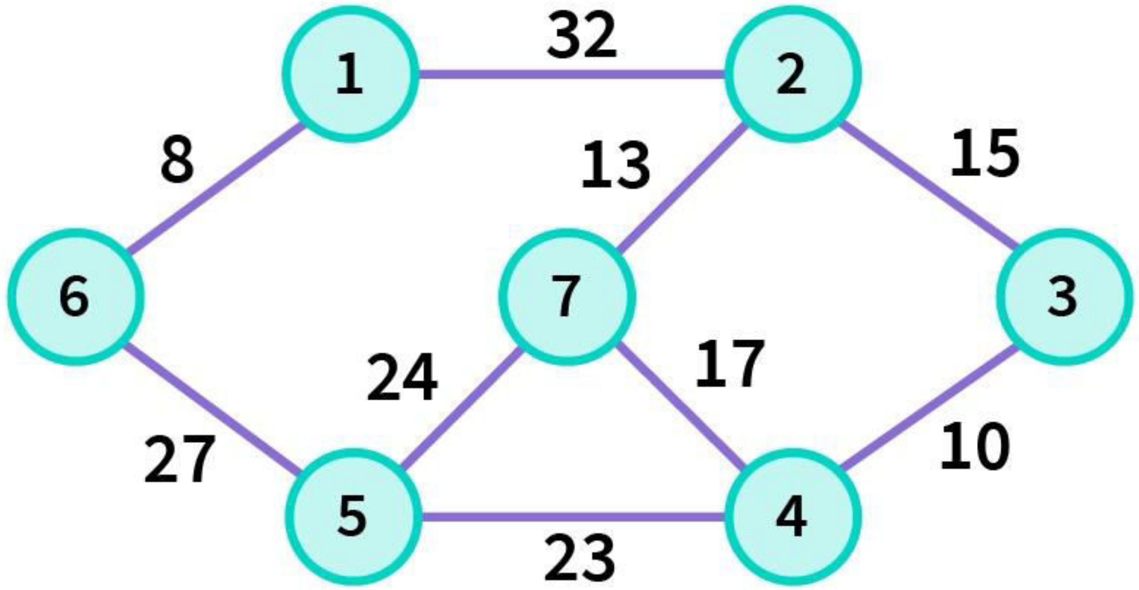
1	2	3	4	5	6	7
0	32	∞	∞	∞	∞	∞

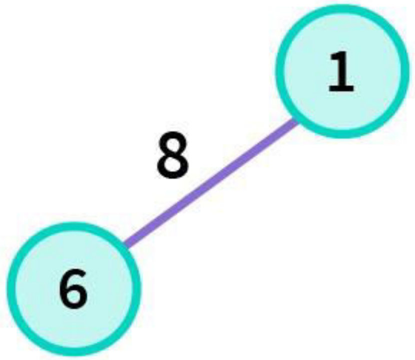
Prim’s Algorithm: Example 1



1	2	3	4	5	6	7
0	32	∞	∞	∞	8	∞

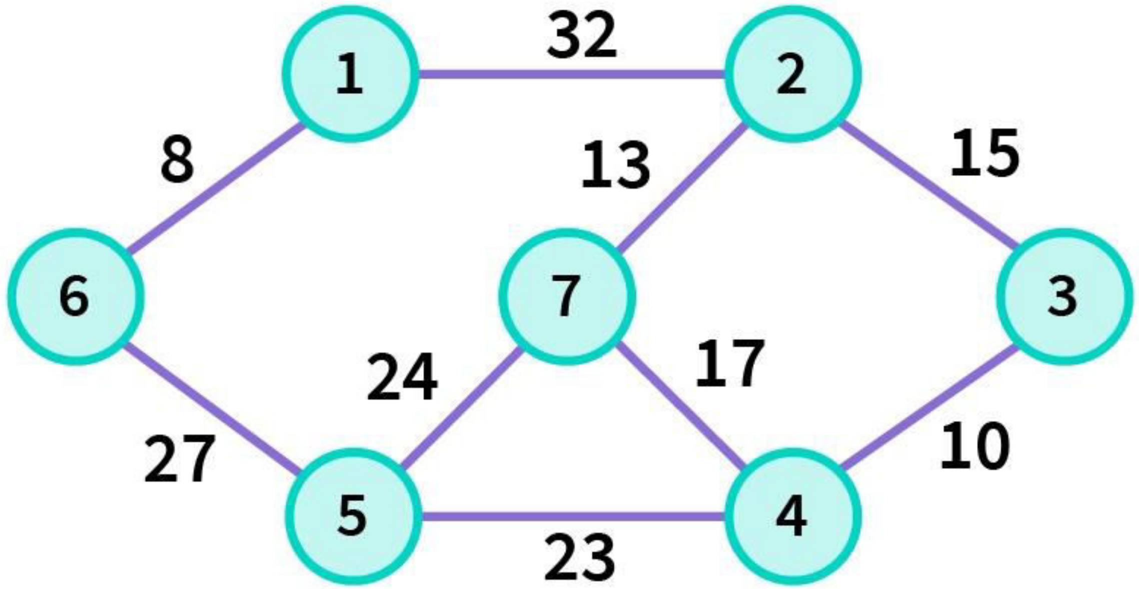
Prim’s Algorithm: Example 1

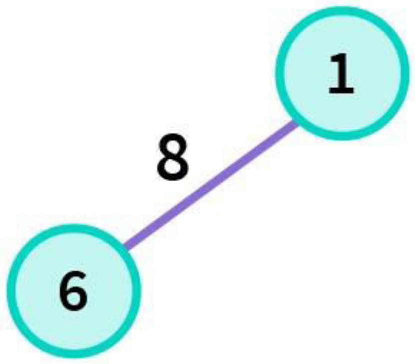




1	2	3	4	5	6	7
0	32	∞	∞	∞	8	∞

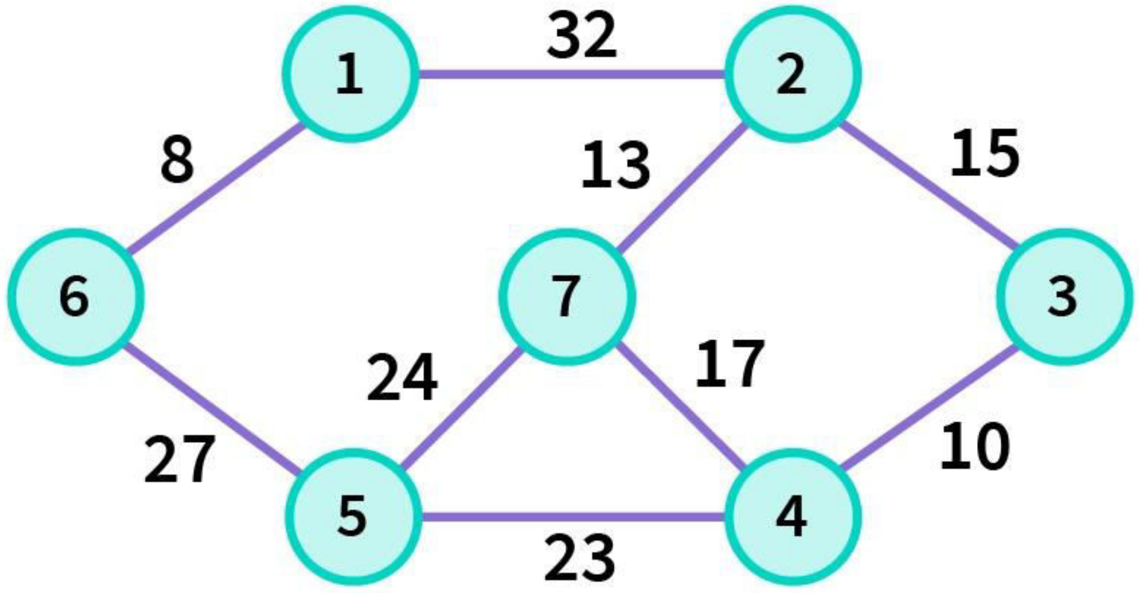
Prim's Algorithm: Example 1

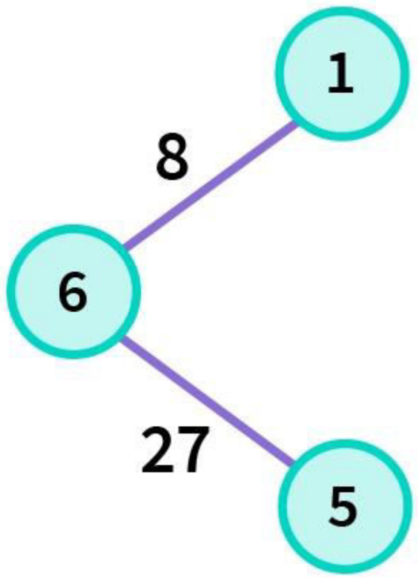




1	2	3	4	5	6	7
0	32	∞	∞	27	8	∞

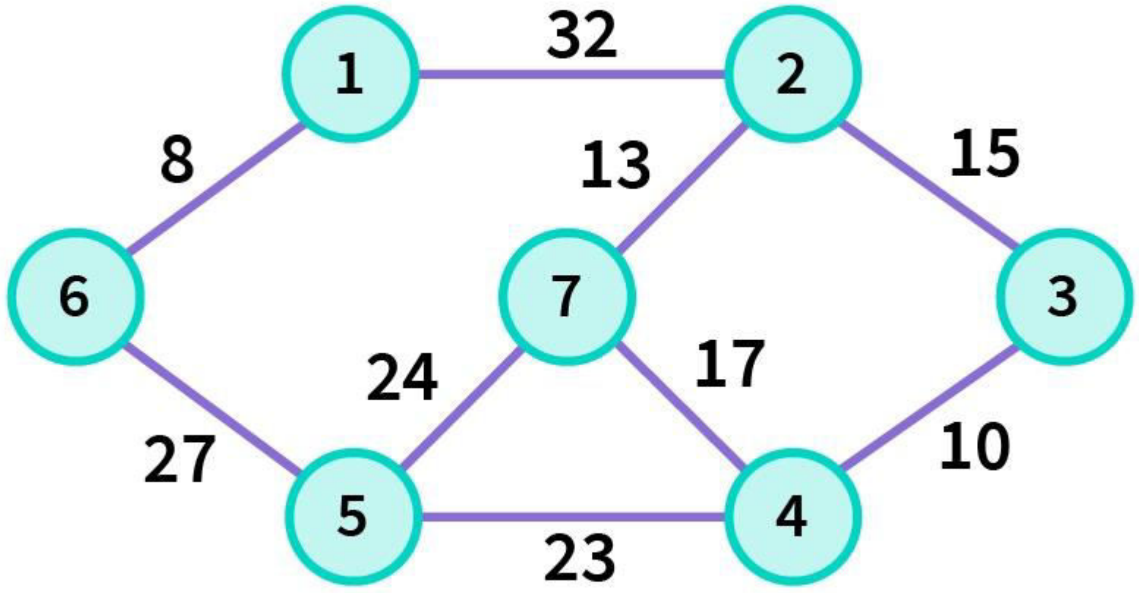
Prim’s Algorithm: Example 1

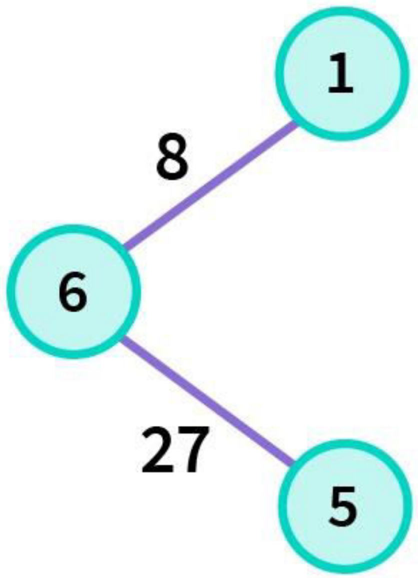




1	2	3	4	5	6	7
0	32	∞	∞	27	8	∞

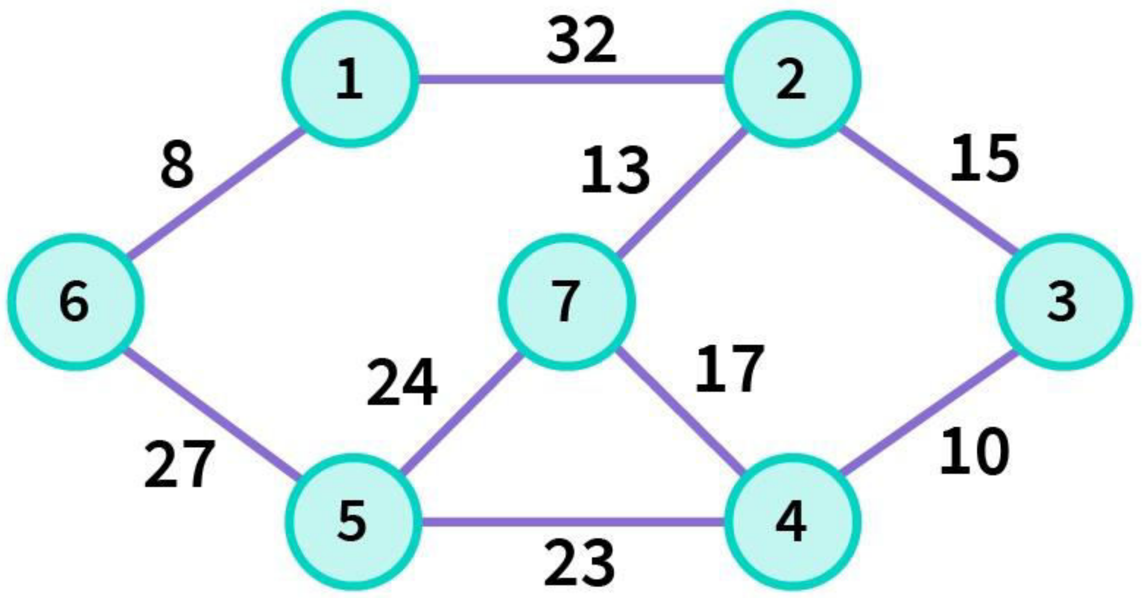
Prim's Algorithm: Example 1

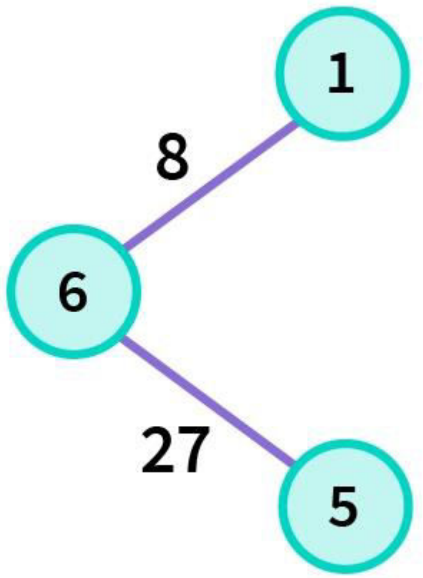




1	2	3	4	5	6	7
0	32	∞	23	27	8	∞

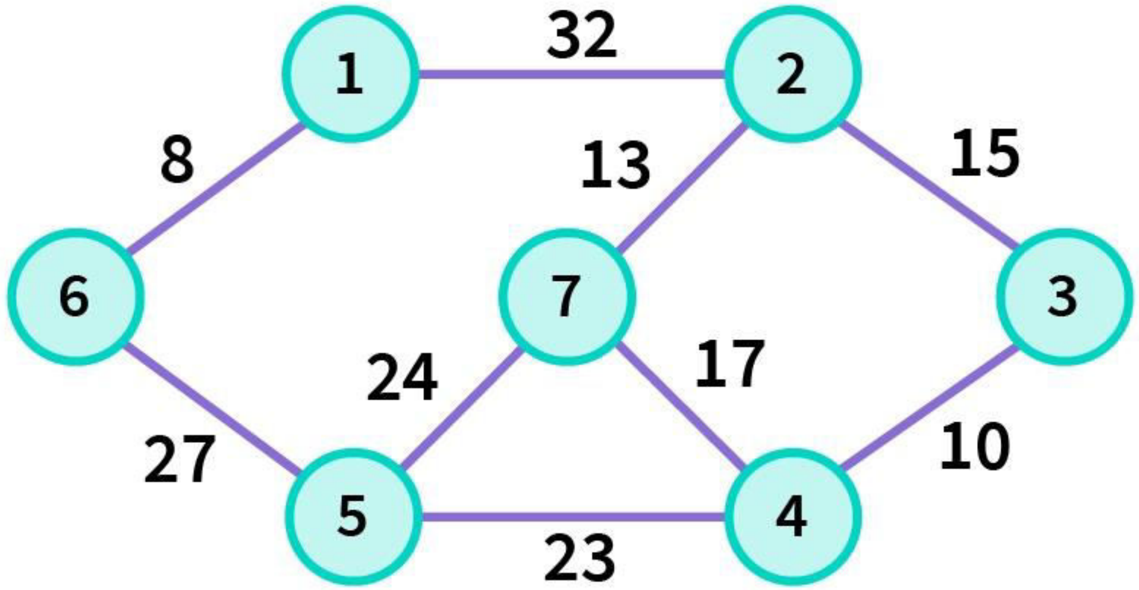
Prim’s Algorithm: Example 1

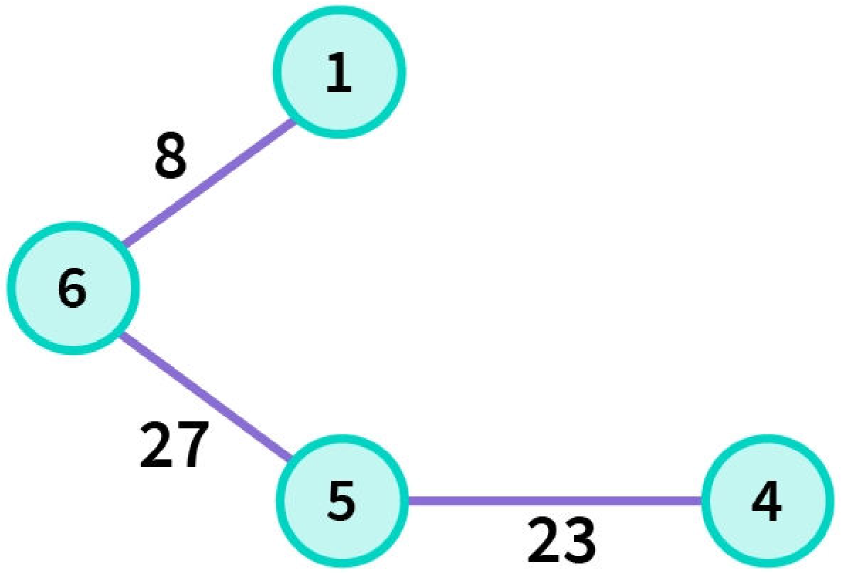




1	2	3	4	5	6	7
0	32	∞	23	27	8	24

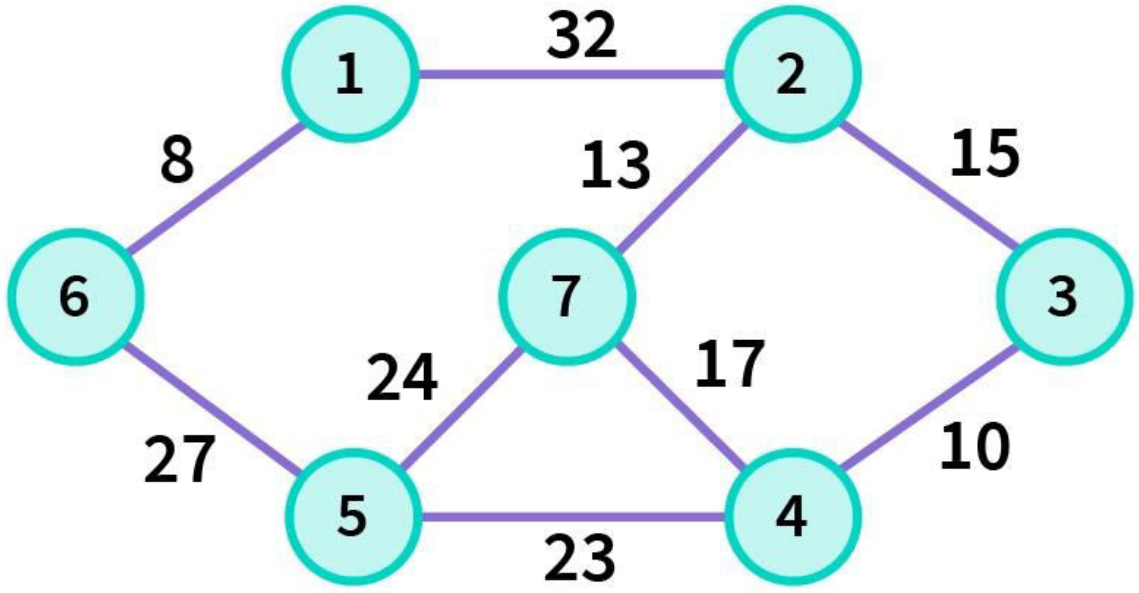
Prim’s Algorithm: Example 1

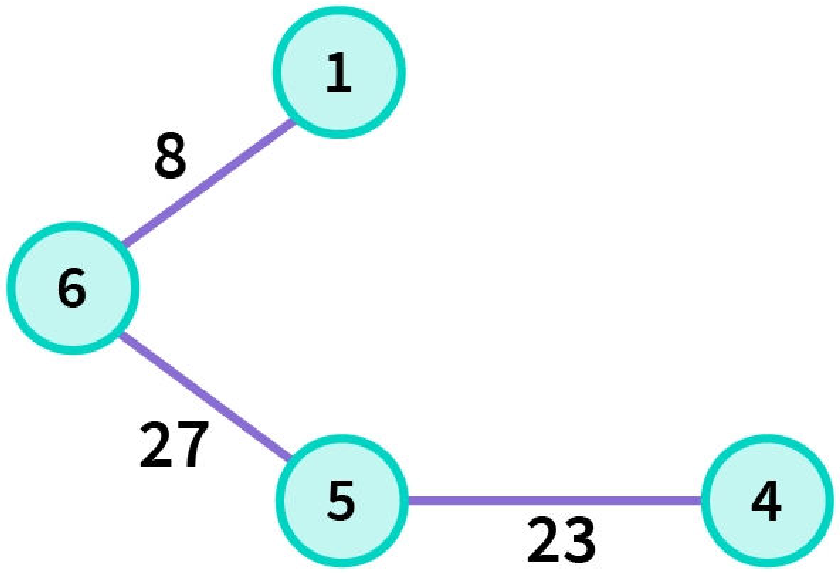




1	2	3	4	5	6	7
0	32	∞	23	27	8	24

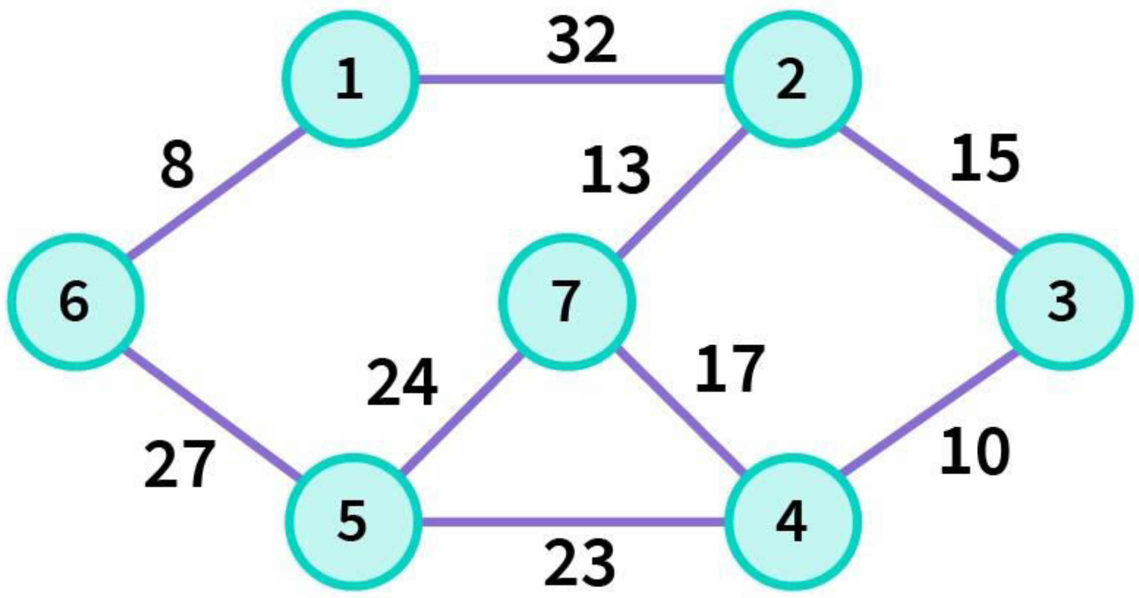
Prim’s Algorithm: Example 1

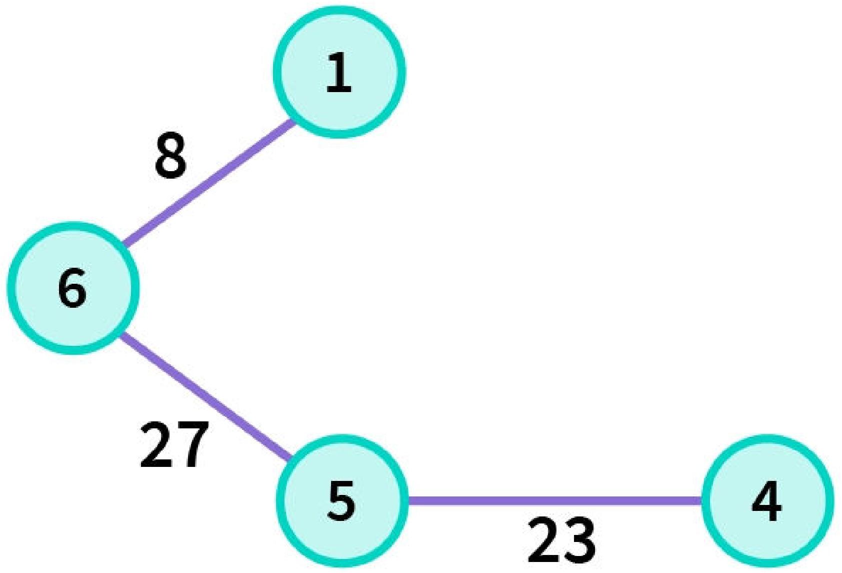




1	2	3	4	5	6	7
0	32	10	23	27	8	24

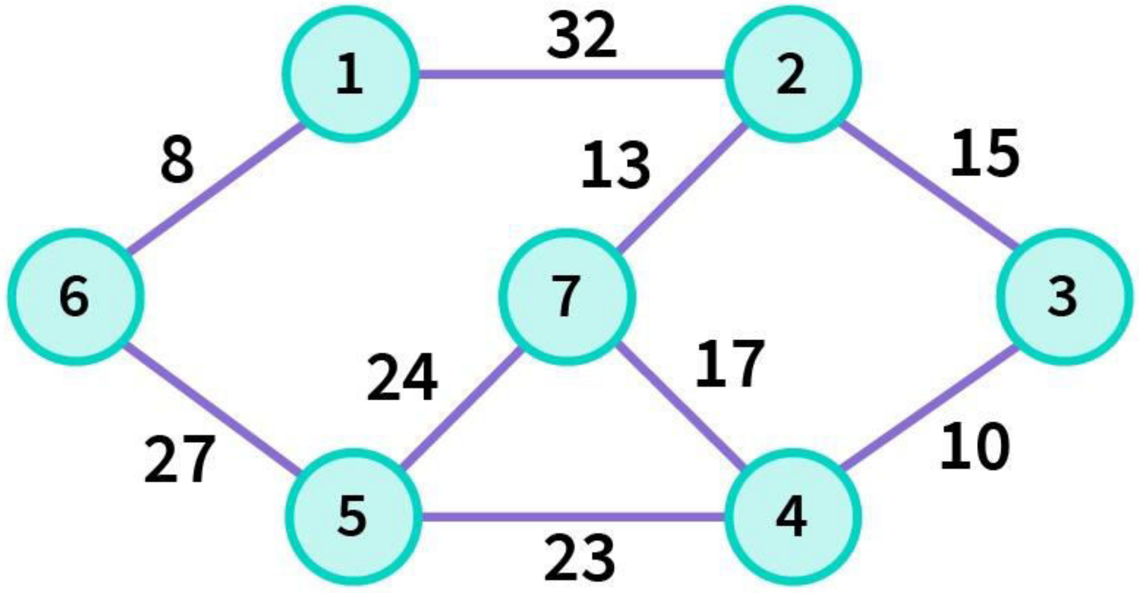
Prim’s Algorithm: Example 1

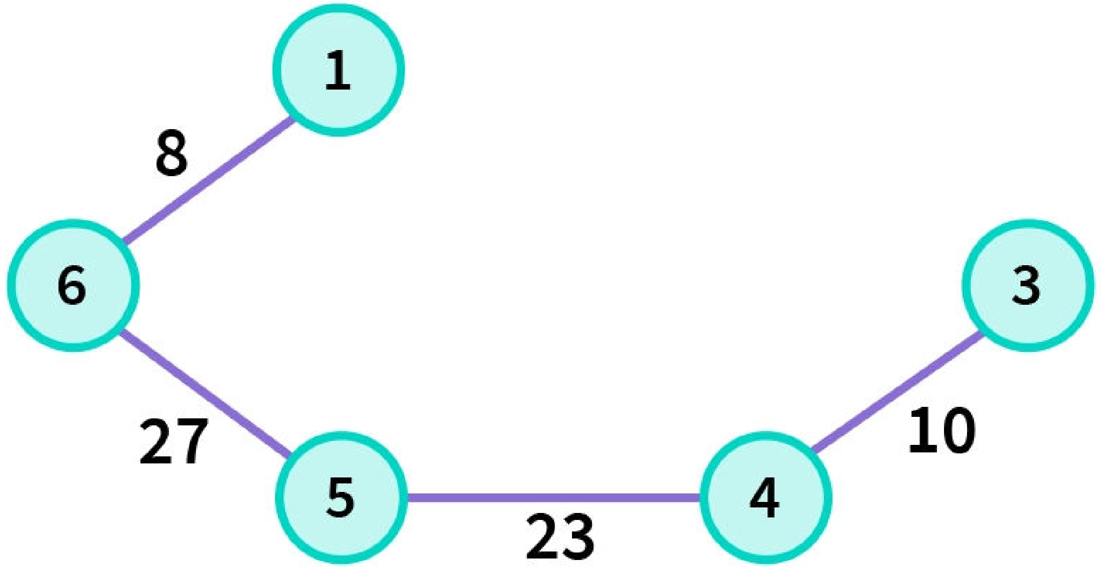




1	2	3	4	5	6	7
0	32	10	23	27	8	17

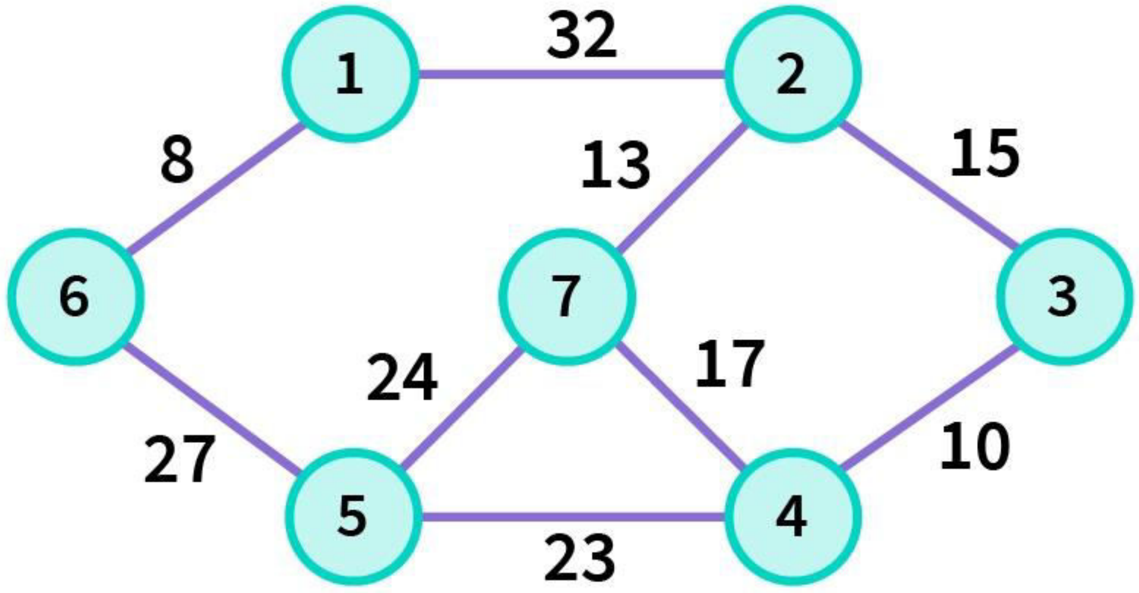
Prim's Algorithm: Example 1

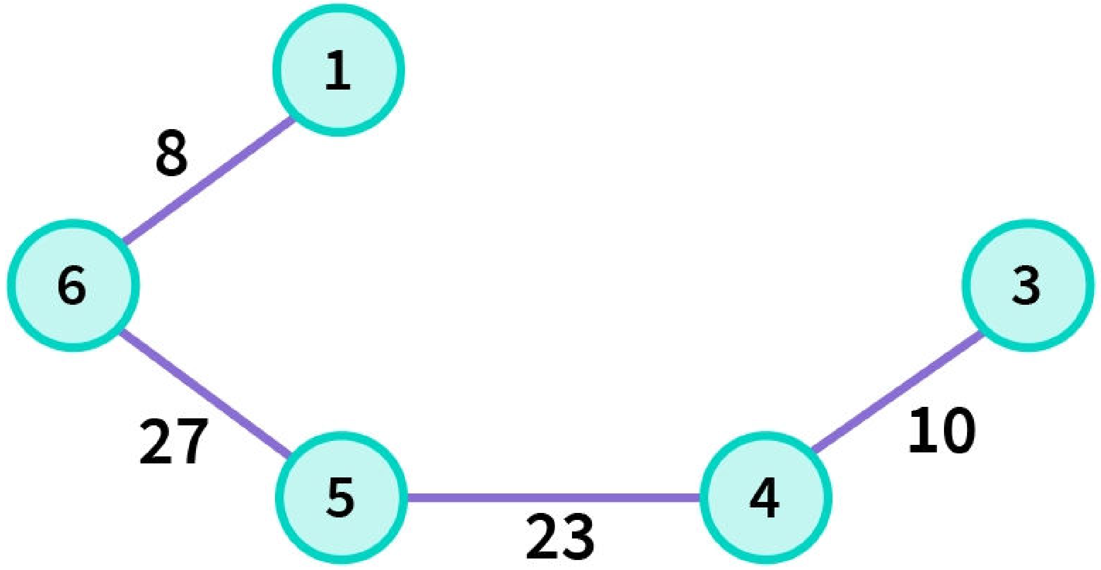




1	2	3	4	5	6	7
0	32	10	23	27	8	17

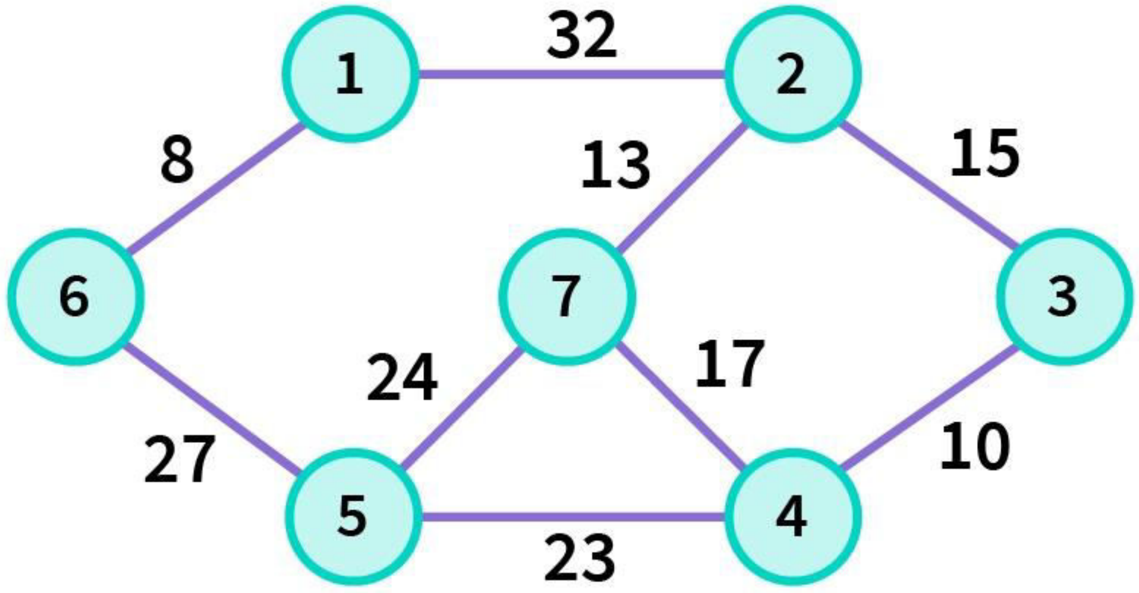
Prim’s Algorithm: Example 1

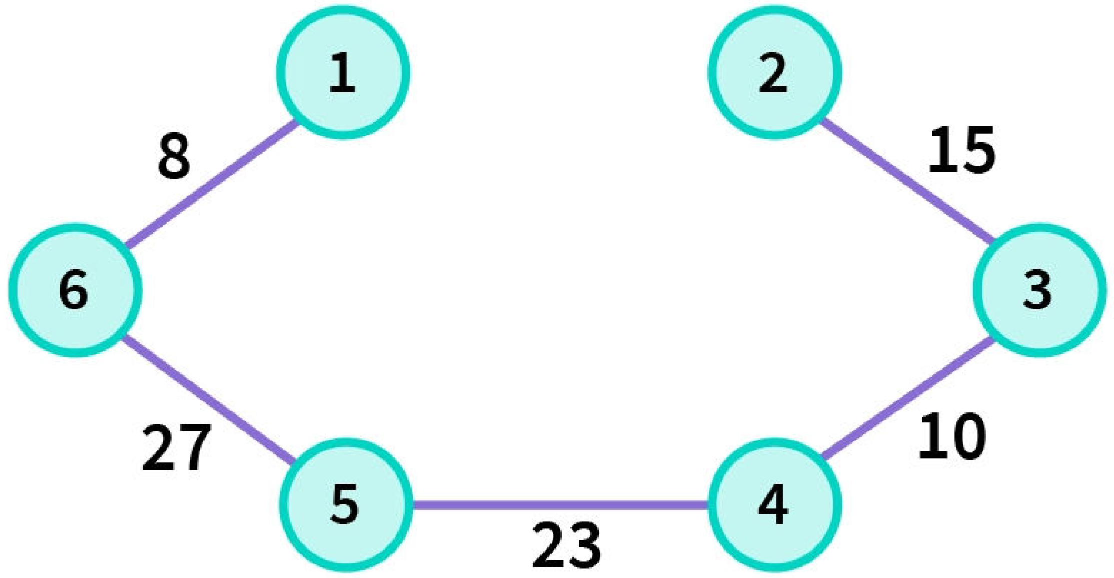




1	2	3	4	5	6	7
0	15	10	23	27	8	17

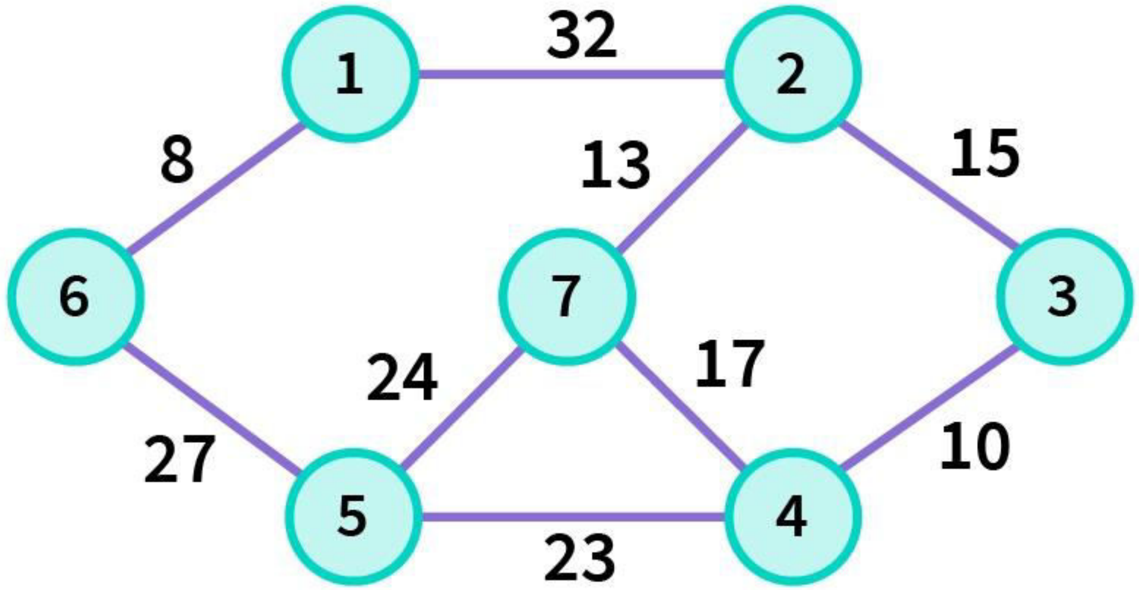
Prim’s Algorithm: Example 1

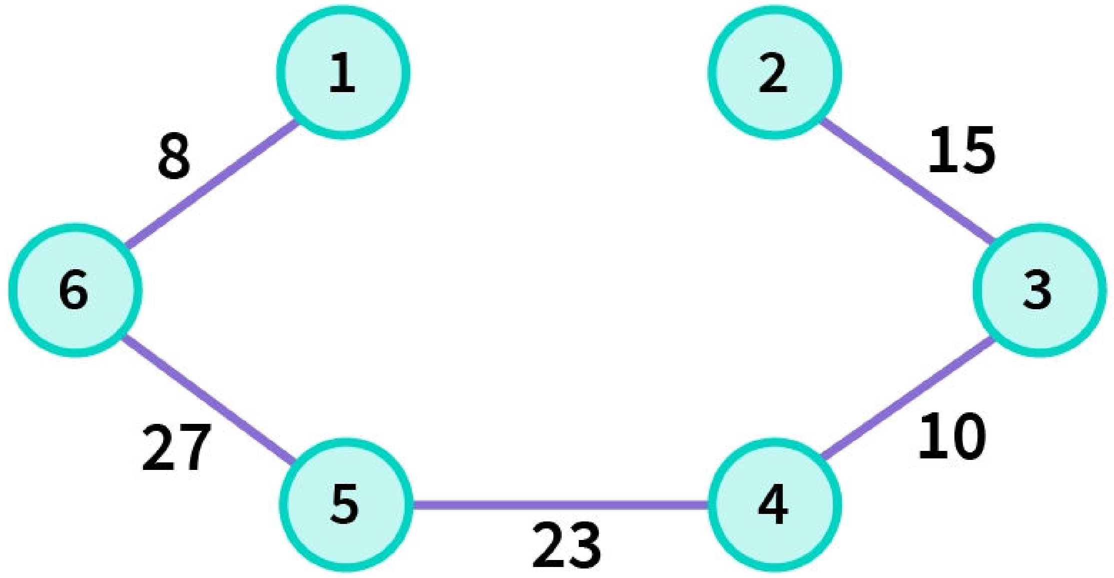




1	2	3	4	5	6	7
0	15	10	23	27	8	17

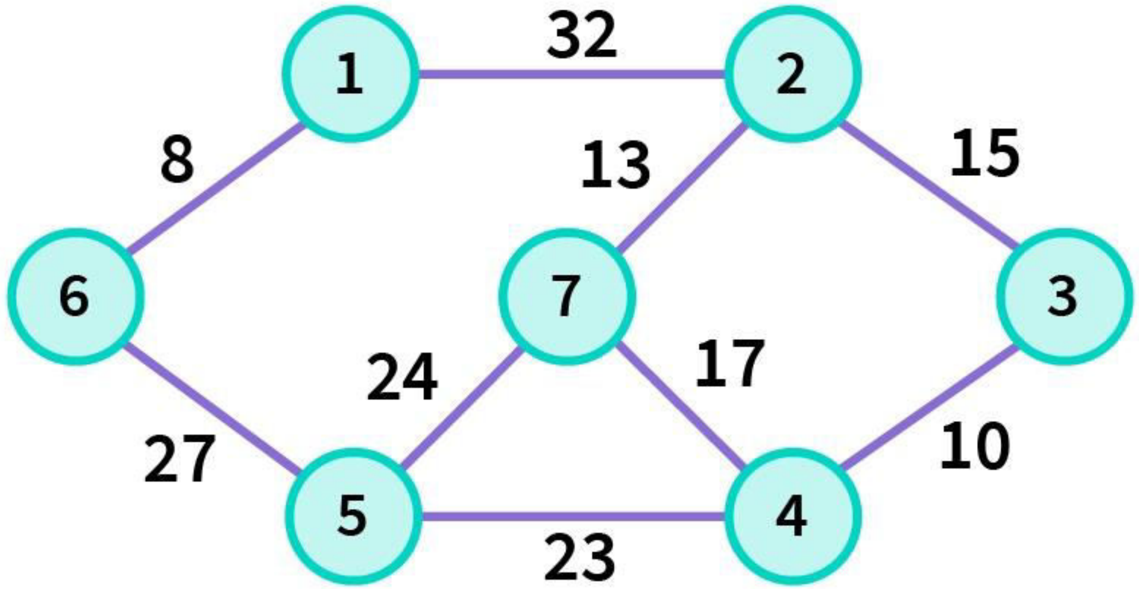
Prim’s Algorithm: Example 1

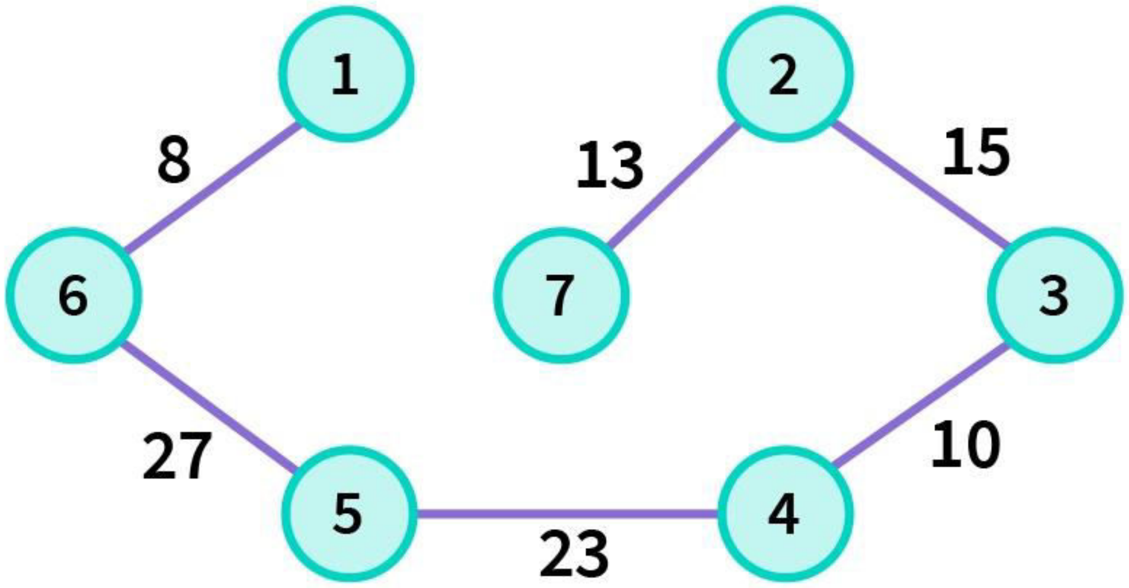




1	2	3	4	5	6	7
0	15	10	23	27	8	13

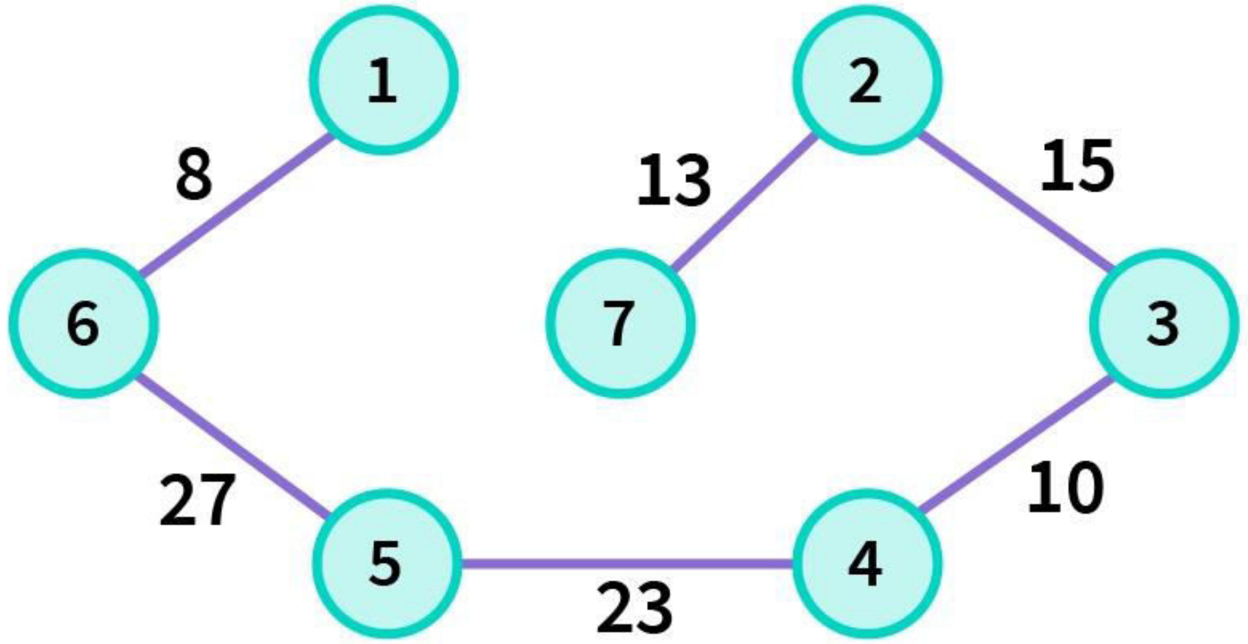
Prim’s Algorithm: Example 1





1	2	3	4	5	6	7
0	15	10	23	27	8	13

Prim’s Algorithm: Example 1



Total Weight = 8 + 27 + 23 + 10 +15 +13 = 96

1	**2**	**3**	**4**	**5**	**6**	**7**
0	15	10	23	27	8	13

Kruskal’s Algorithm

- There is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution.
- It is named Kruskal’s algorithm after Joseph Kruskal.
- The algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

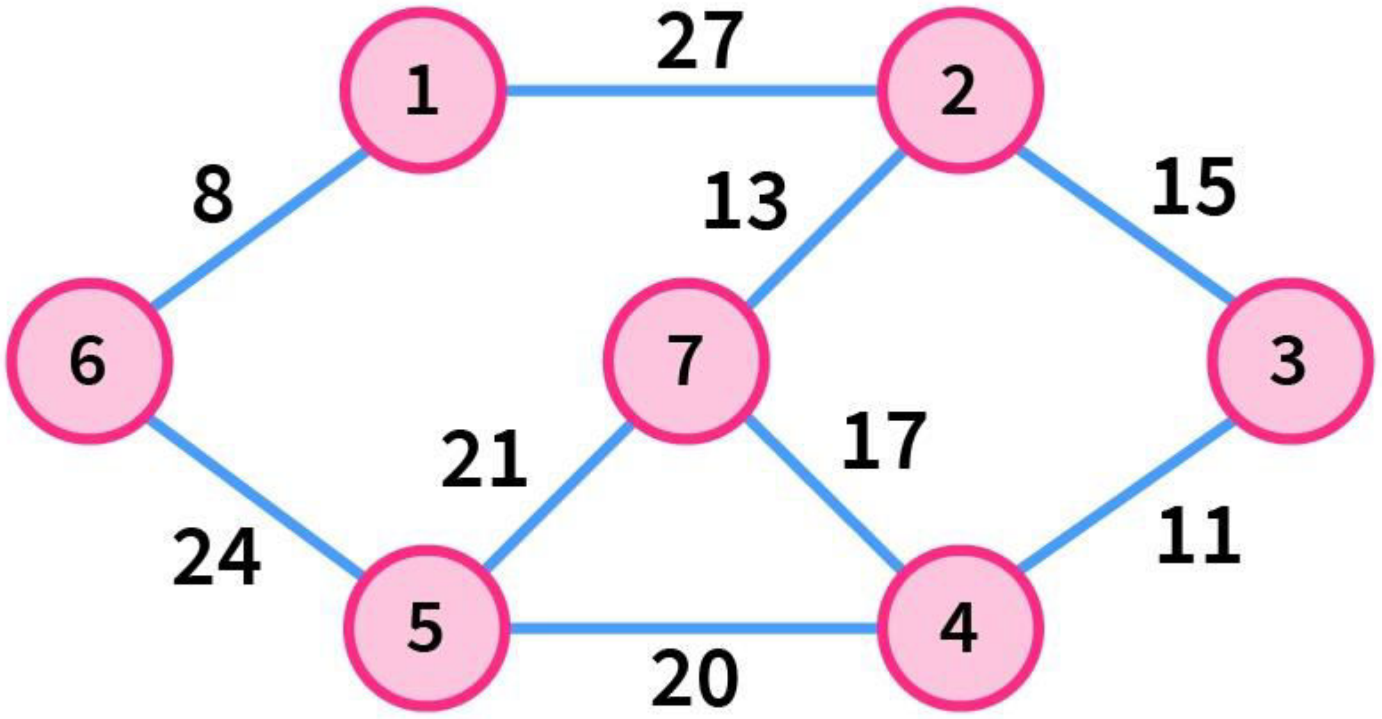
Kruskal’s Algorithm

- The algorithm begins by sorting the graph’s edges in nondecreasing order of their weights.
- Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.
- With an efficient sorting algorithm, the time efficiency of Kruskal’s algorithm will be in

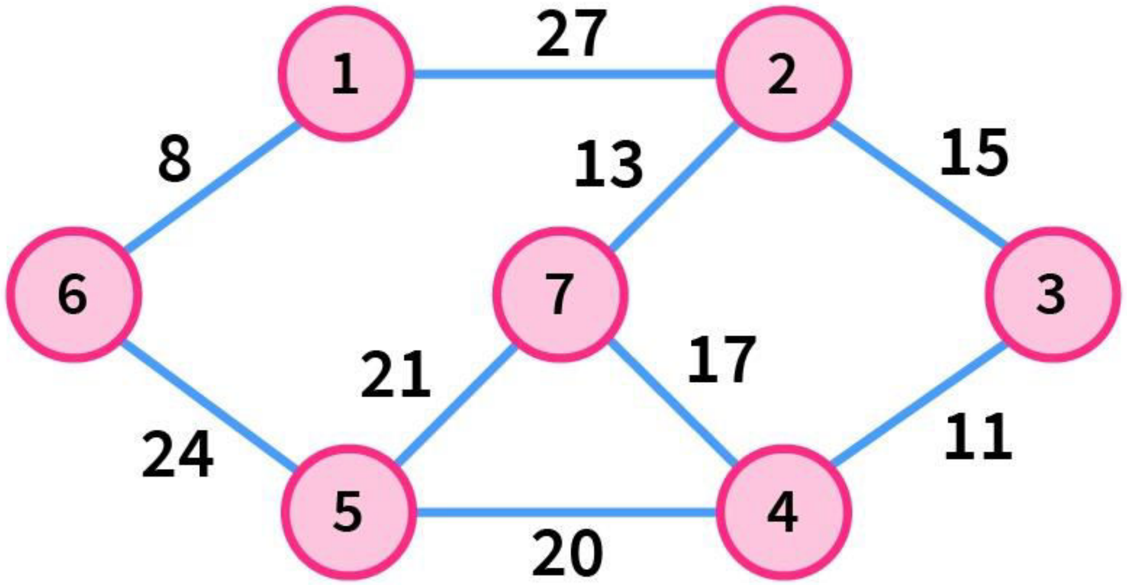
$$O(|E| \log |E|)$$

Kruskal’s Algorithm: Example 1

1- Sort all the edges in non-decreasing order of their weight.
2- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3- Repeat step#2 until there are (V-1) edges in the spanning tree.

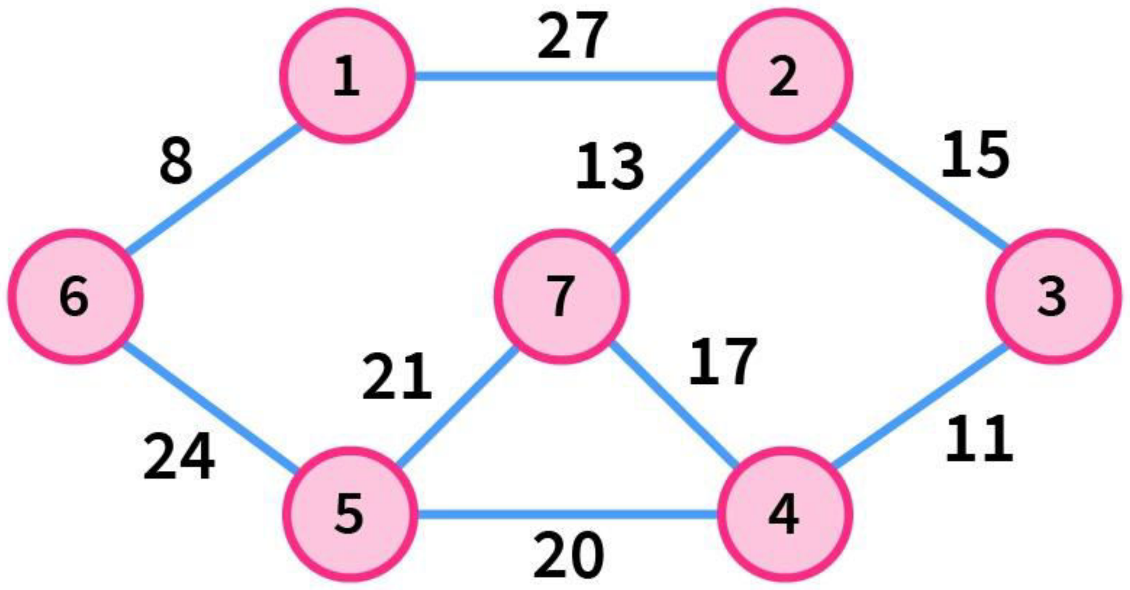


Kruskal’s Algorithm: Example 1



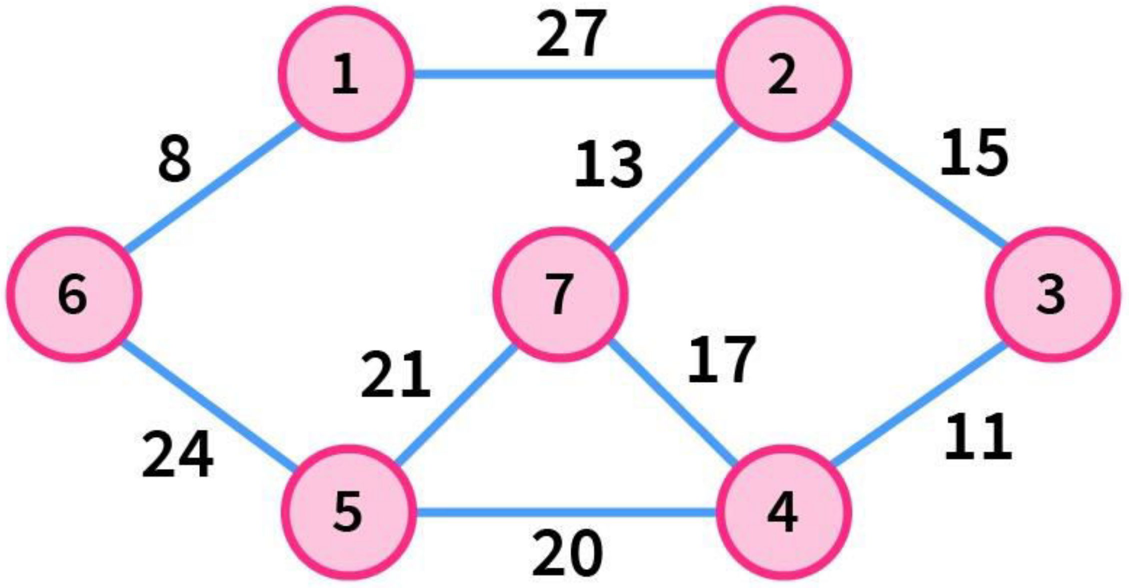
1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

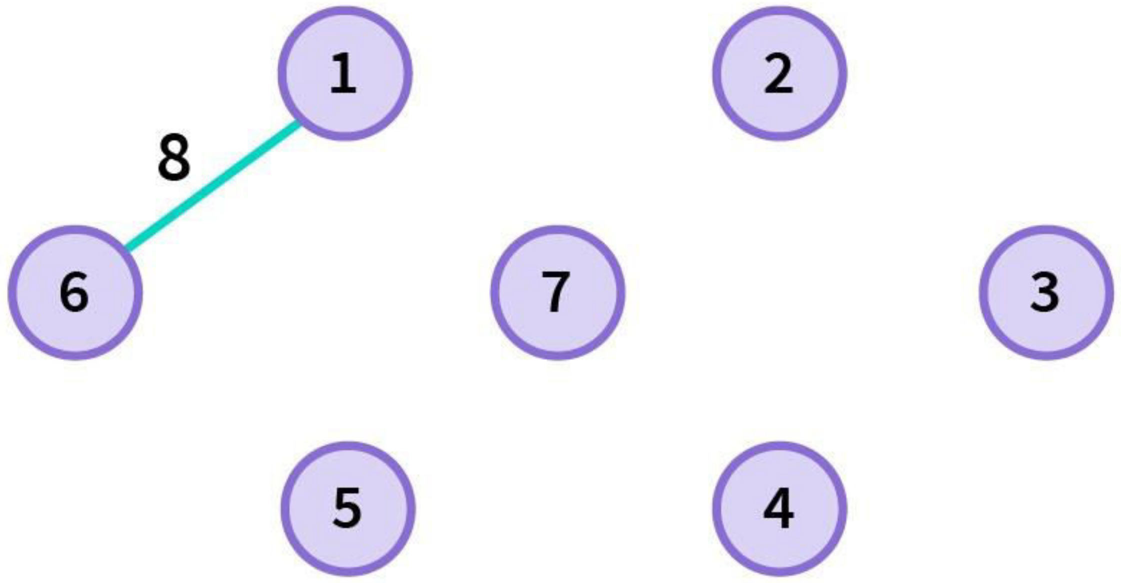
Kruskal’s Algorithm: Example 1



1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

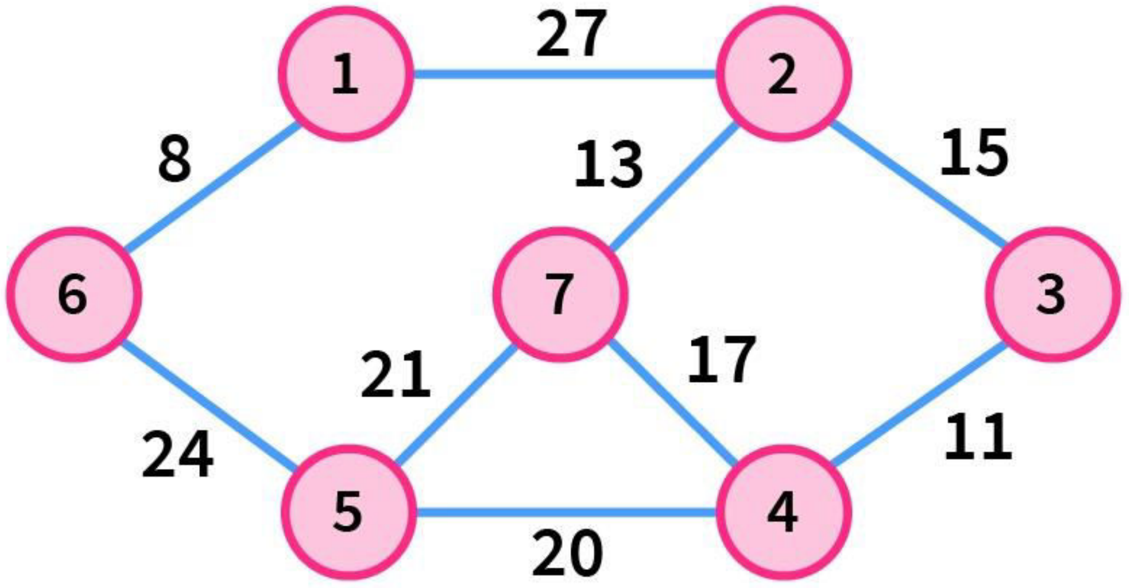
Kruskal's Algorithm: Example 1

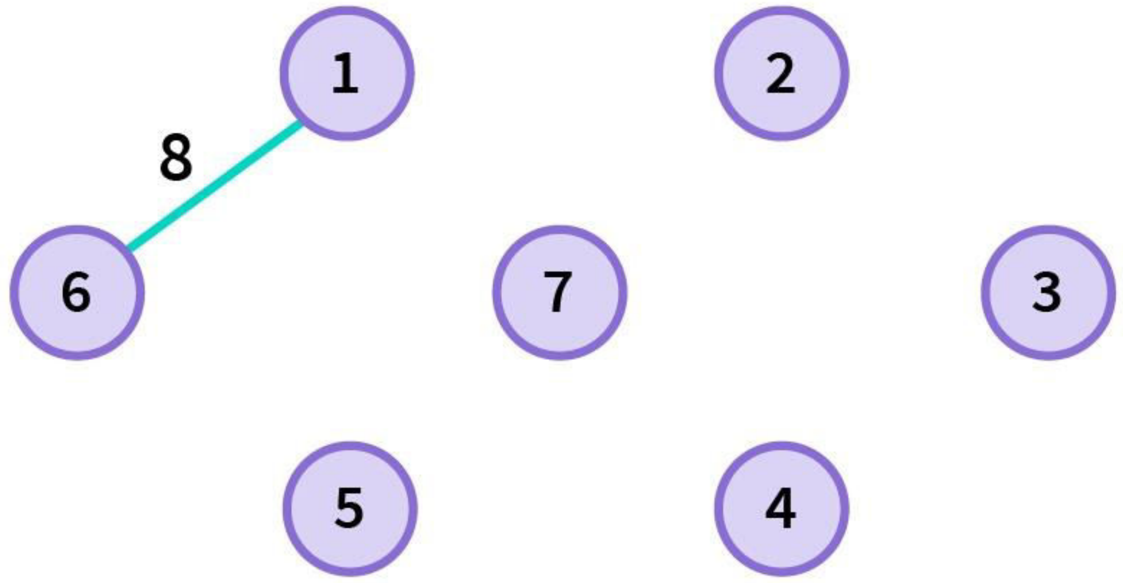




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

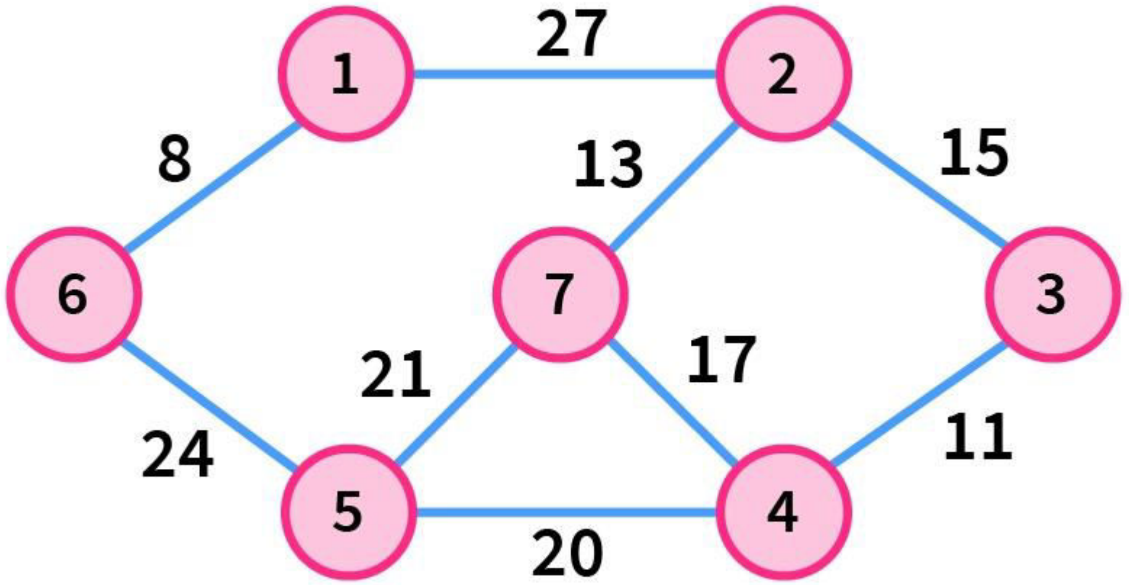
Kruskal’s Algorithm: Example 1

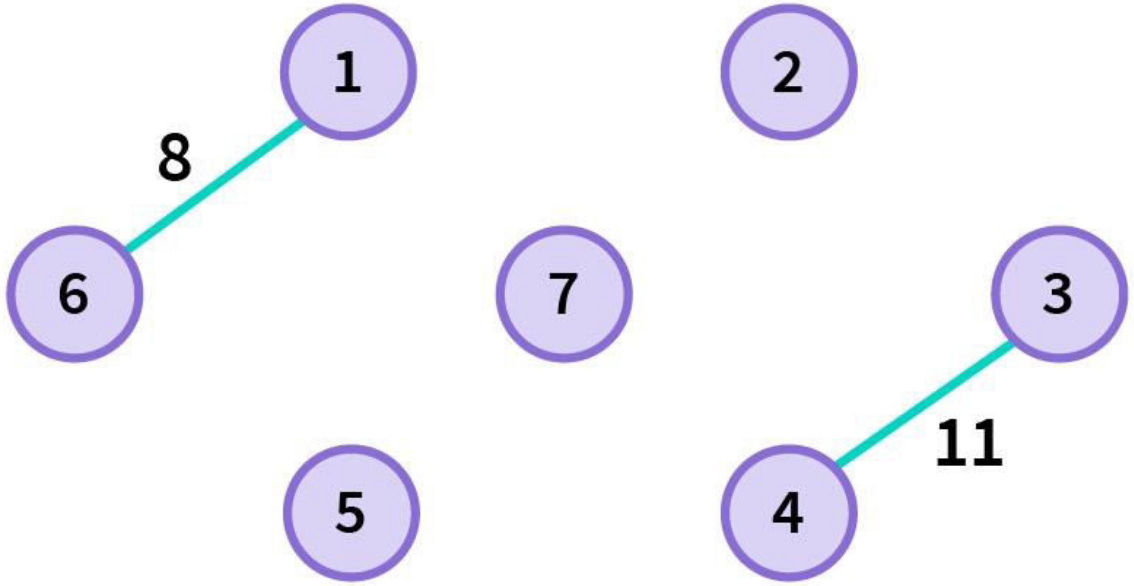




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

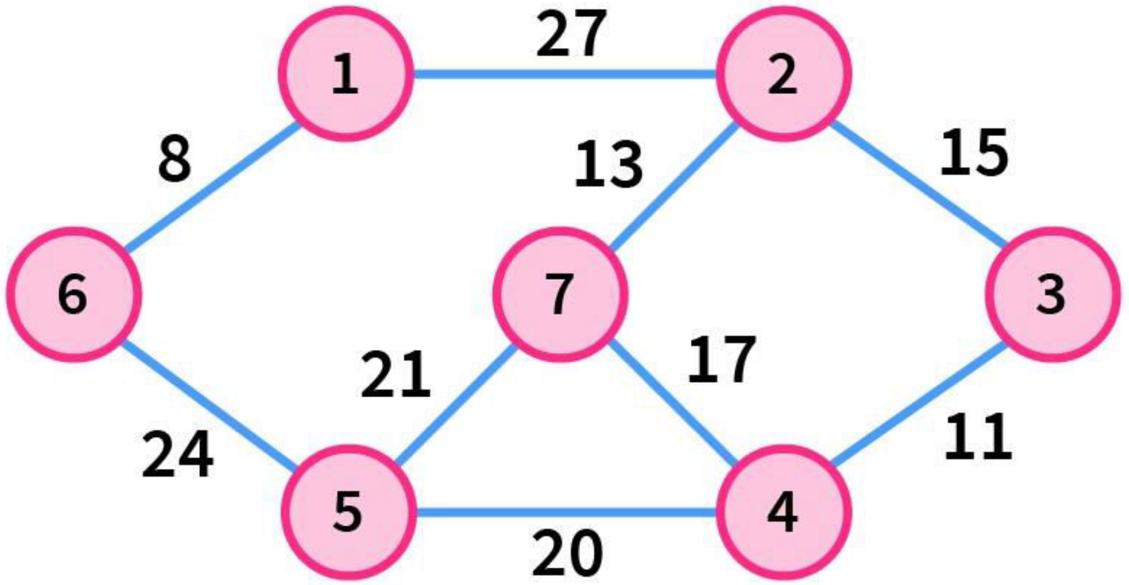
Kruskal’s Algorithm: Example 1

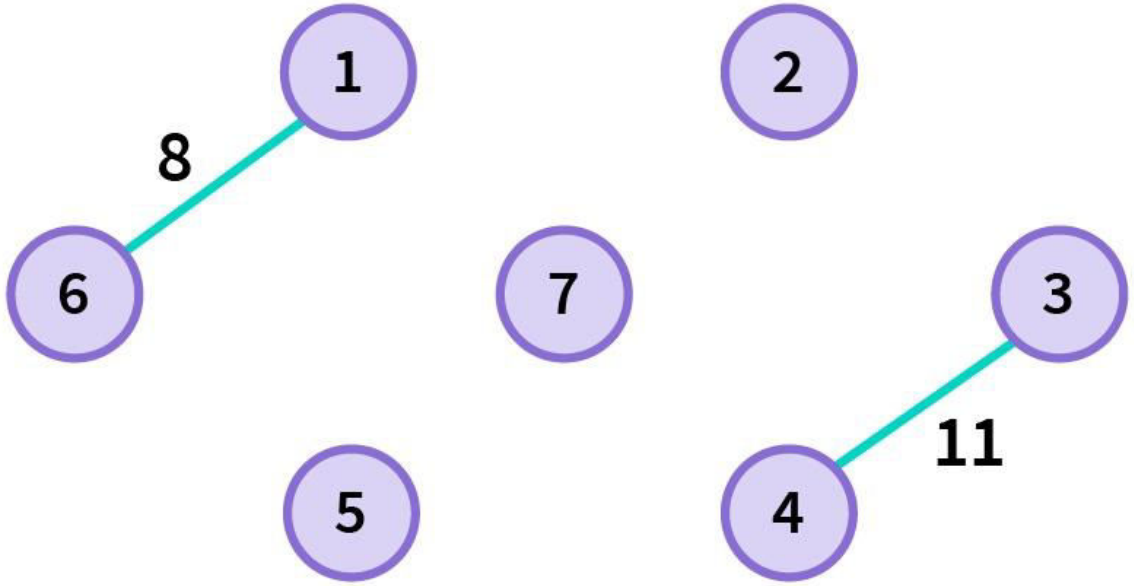




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

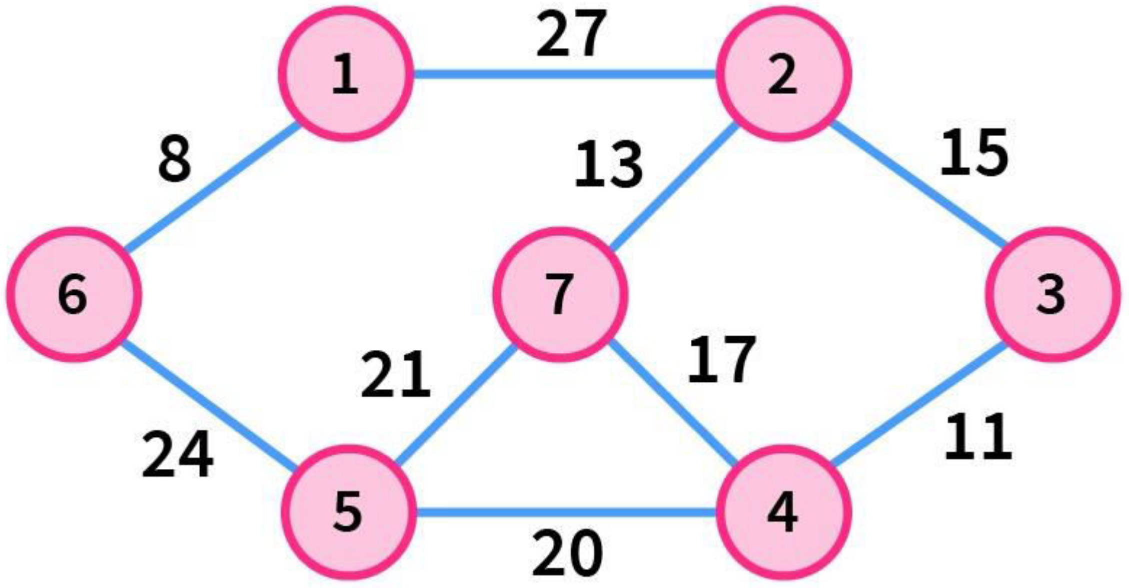
Kruskal’s Algorithm: Example 1

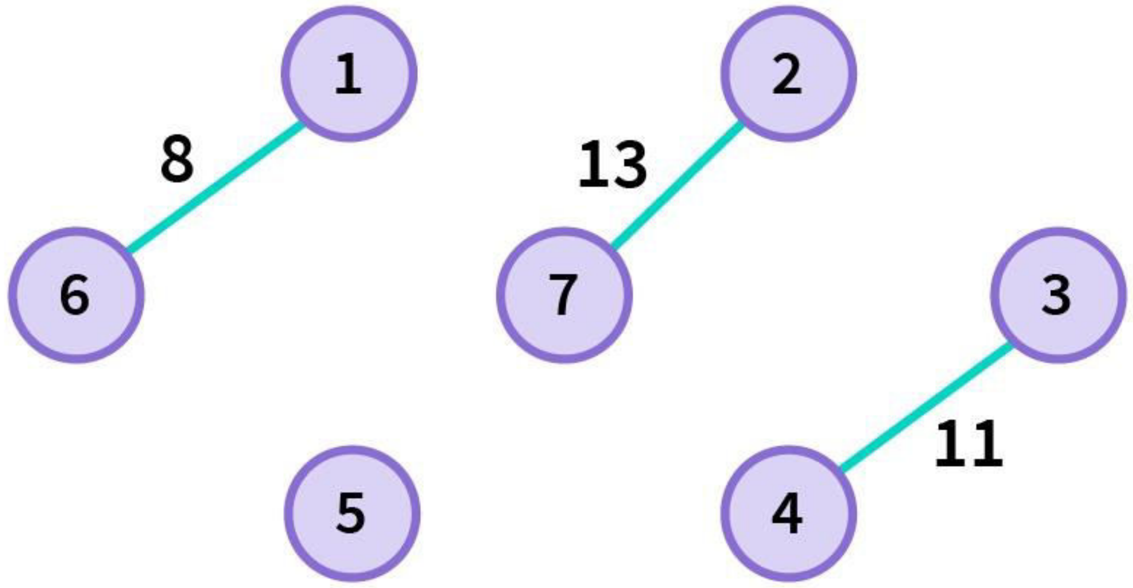




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

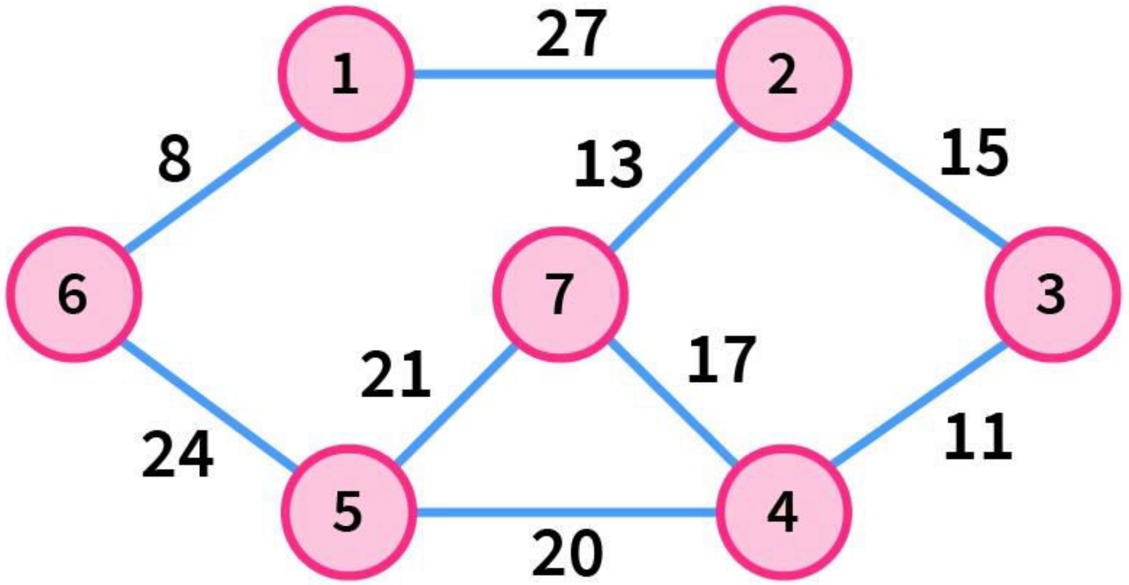
Kruskal’s Algorithm: Example 1

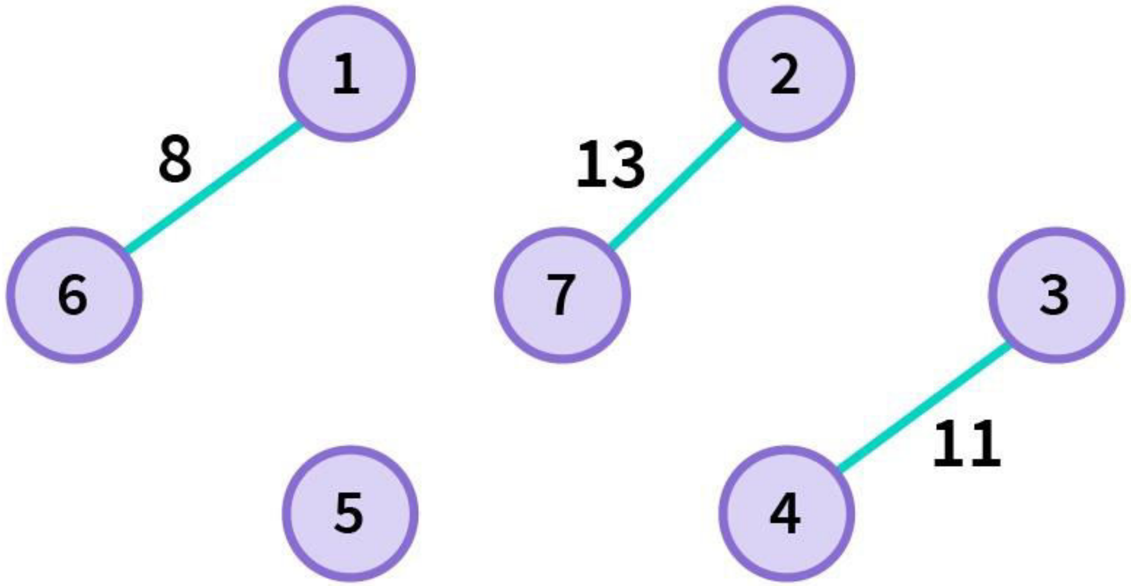




1-6 🟩	3-4 🟩	2-7 🟩	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

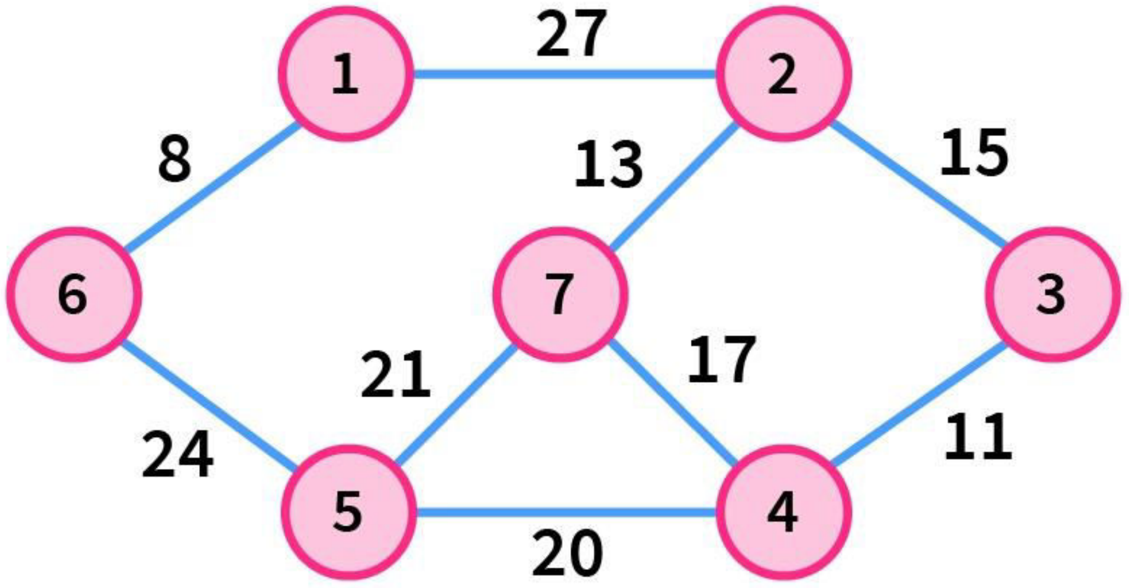
Kruskal's Algorithm: Example 1

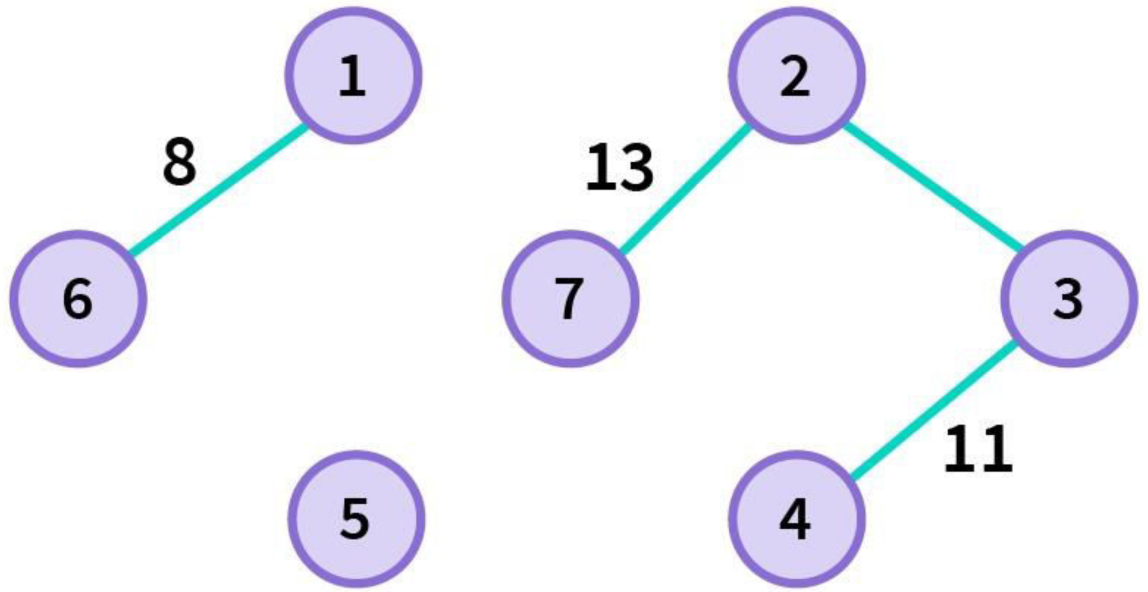




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

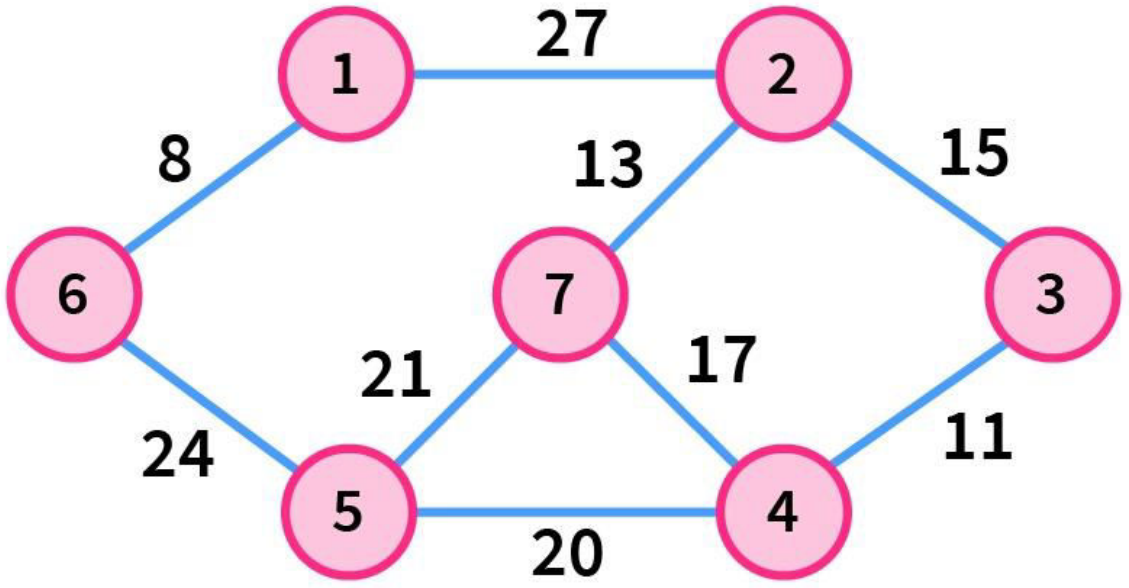
Kruskal’s Algorithm: Example 1

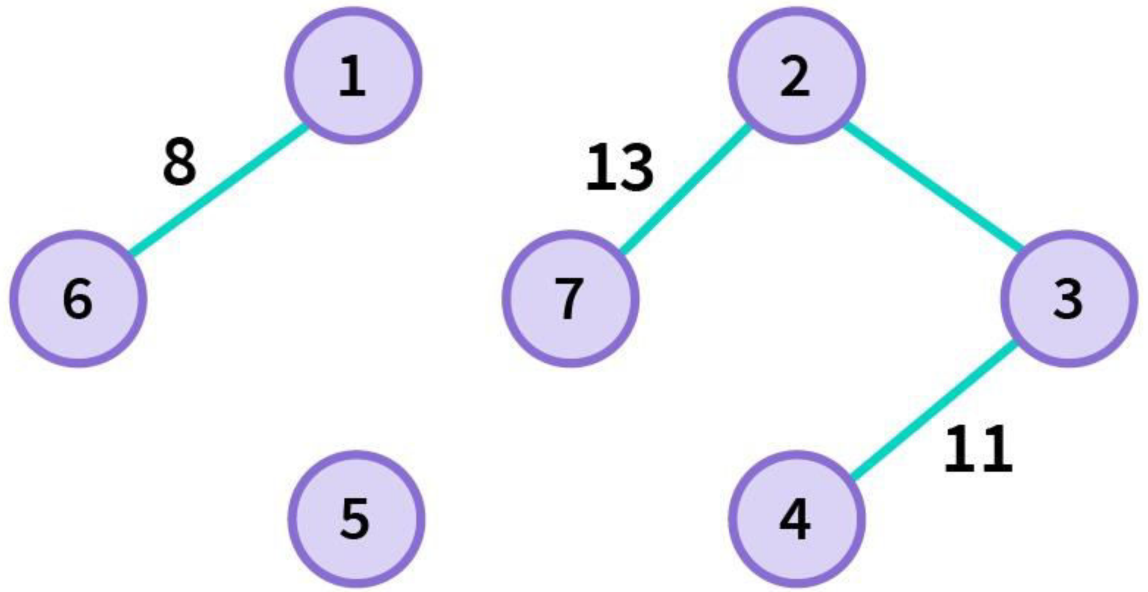




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

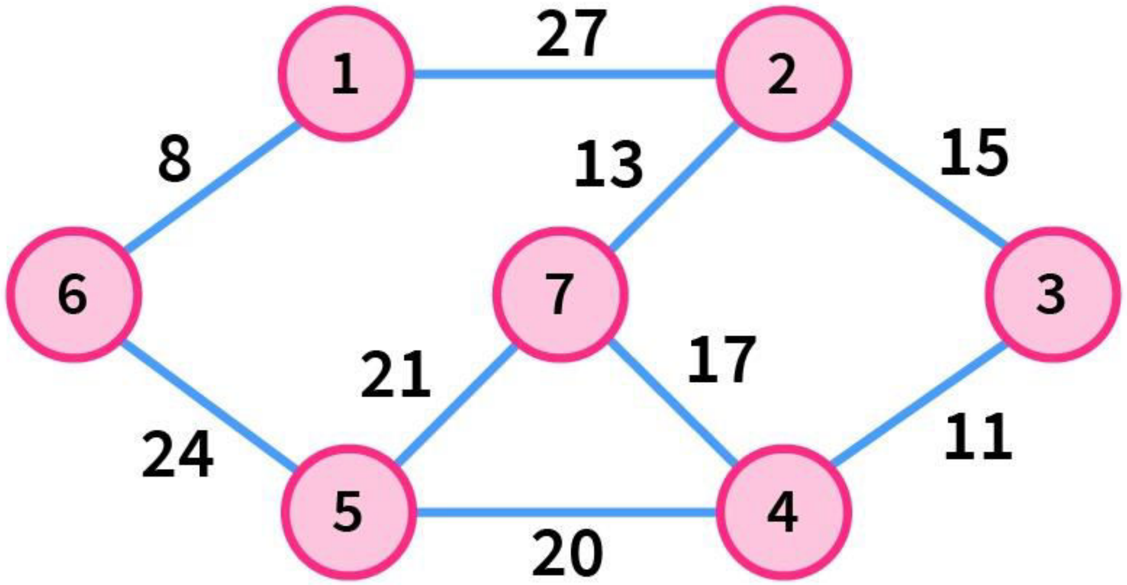
Kruskal's Algorithm: Example 1

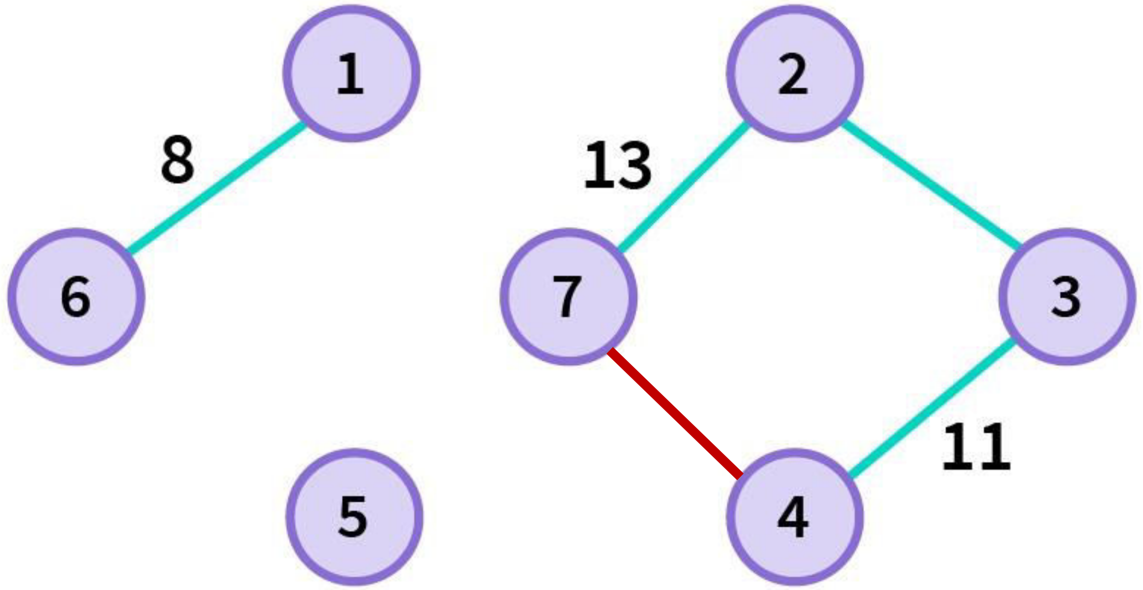




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

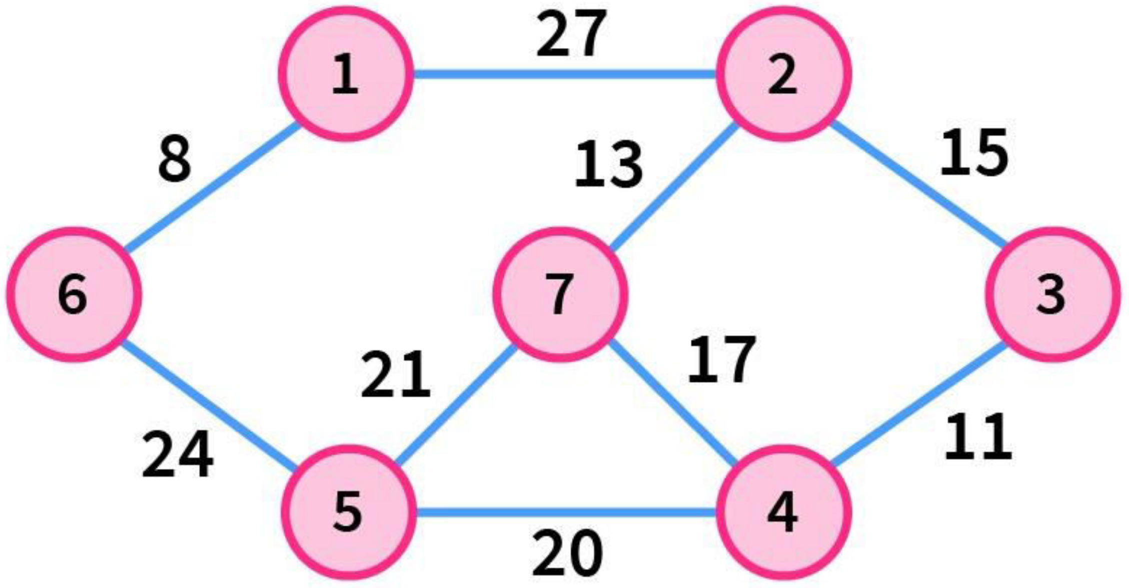
Kruskal’s Algorithm: Example 1

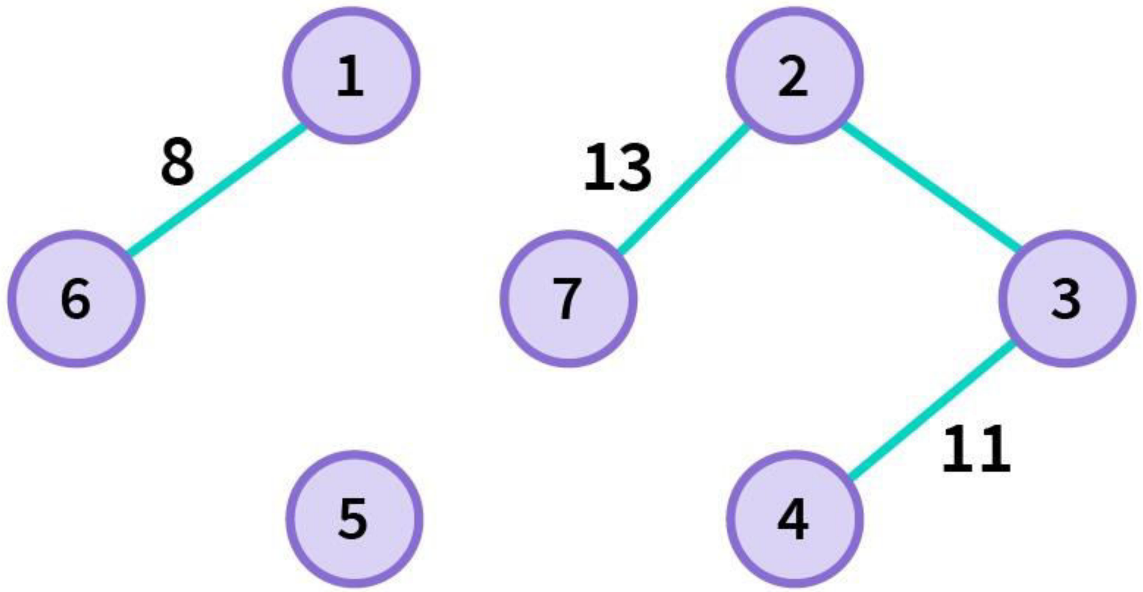




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

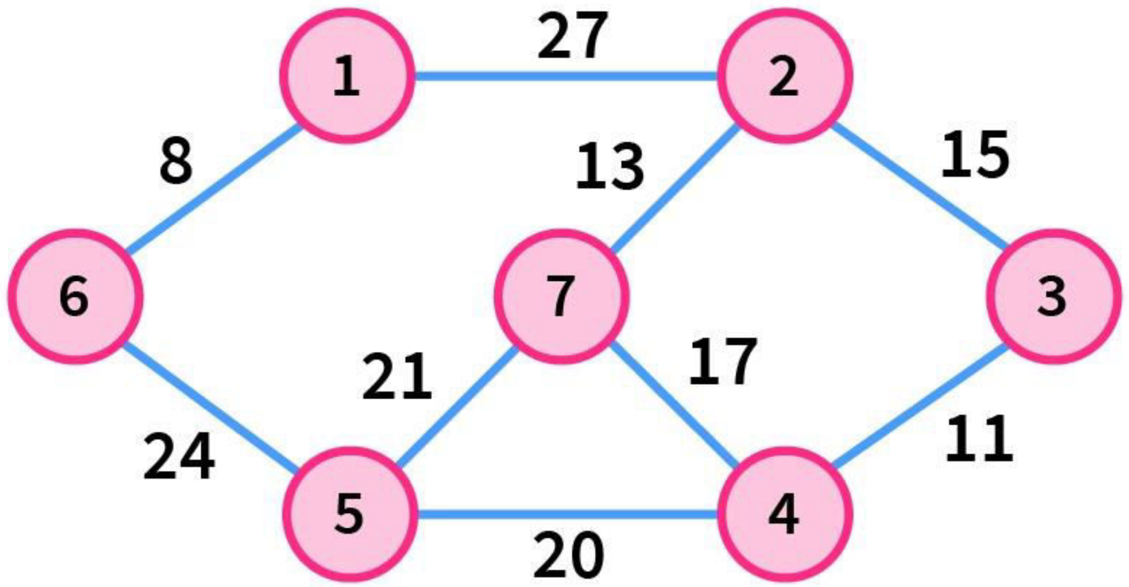
Kruskal’s Algorithm: Example 1

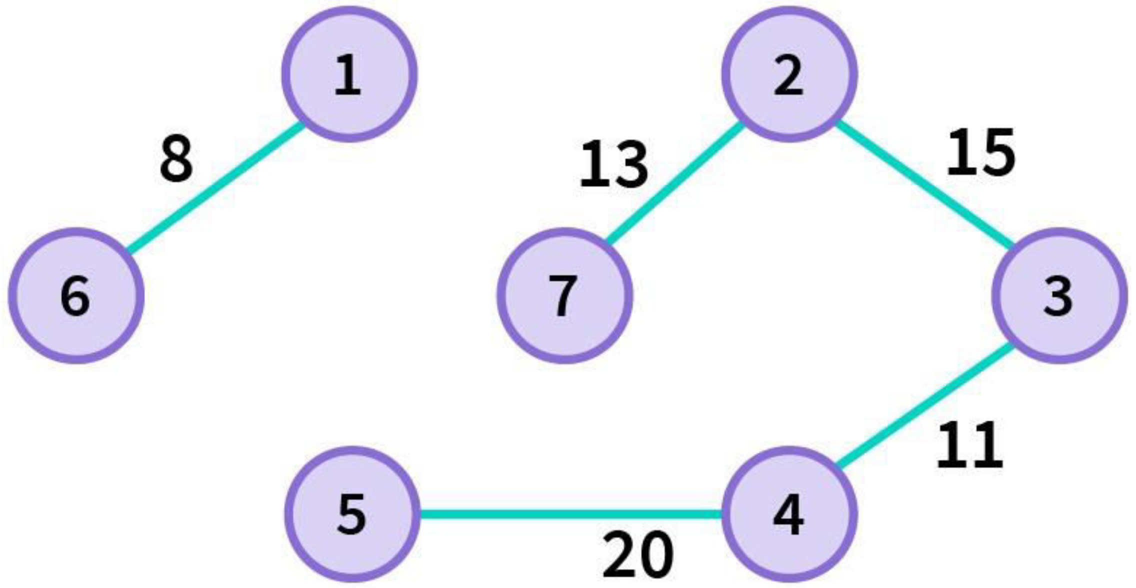




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

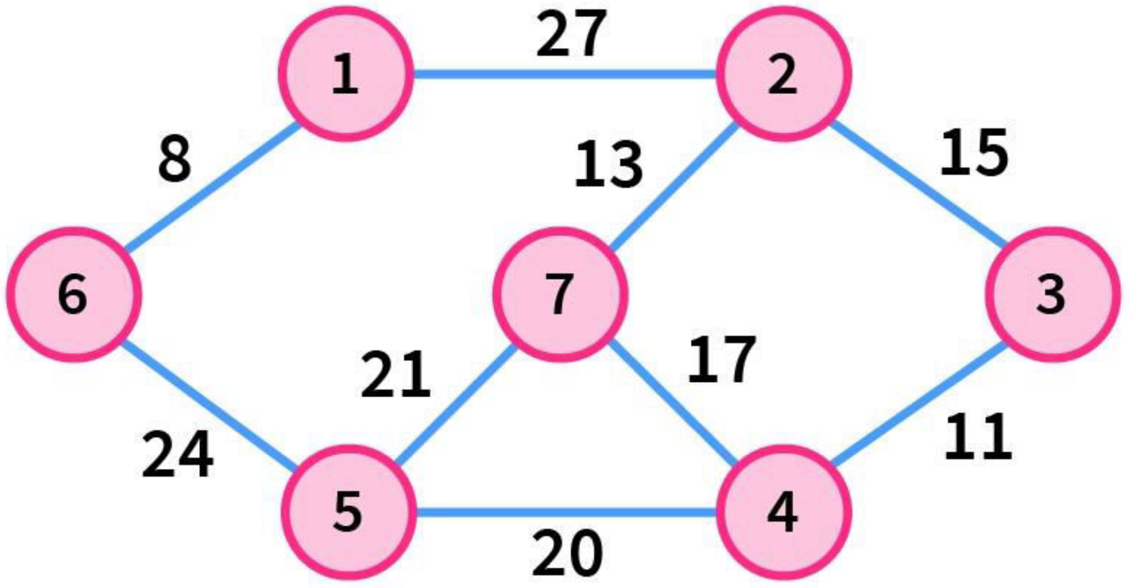
Kruskal’s Algorithm: Example 1

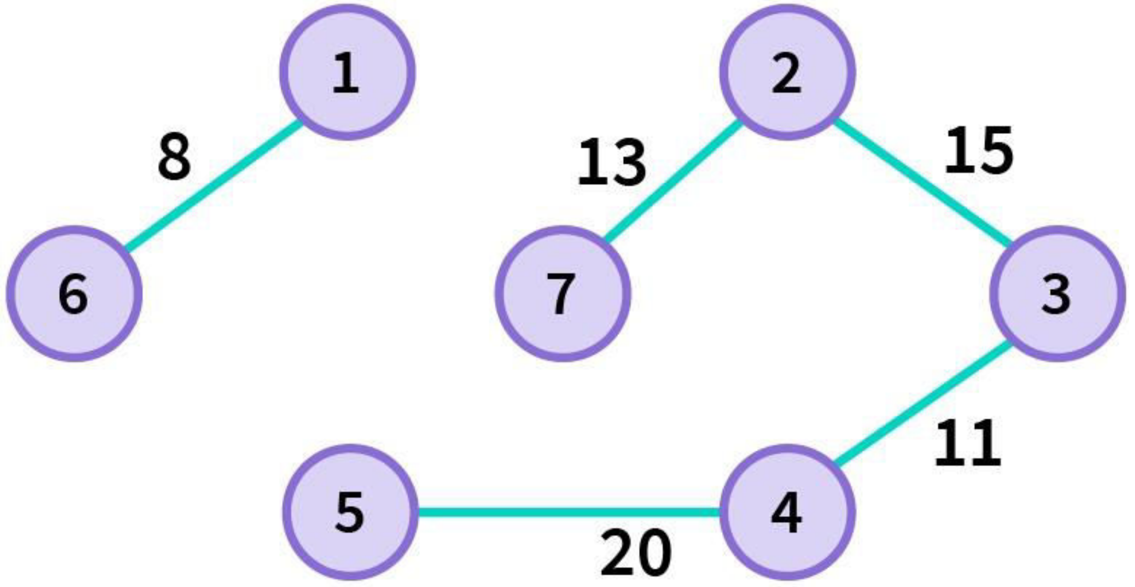




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

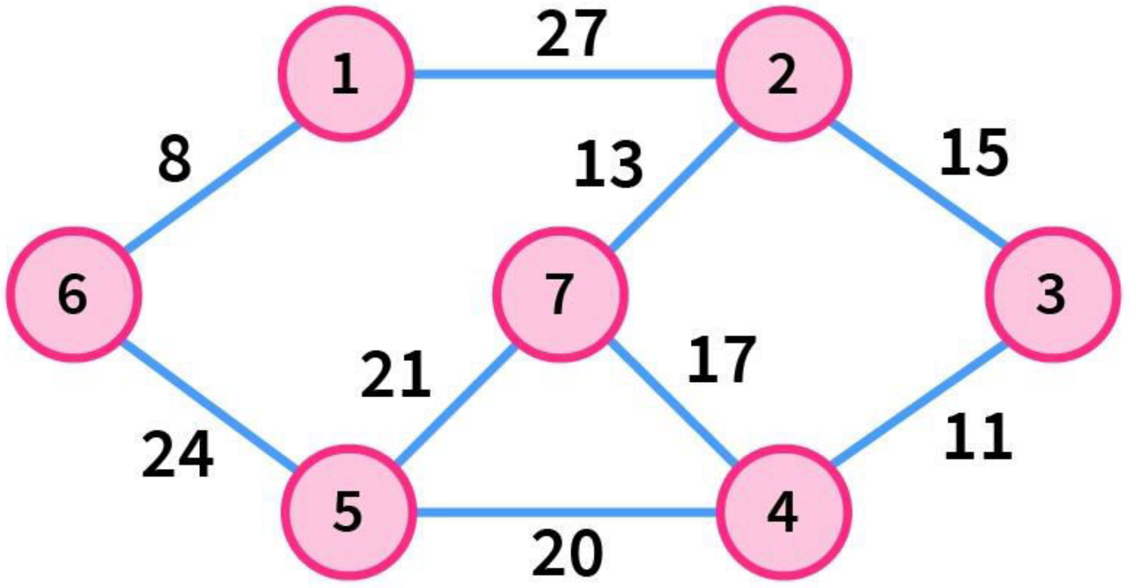
Kruskal’s Algorithm: Example 1

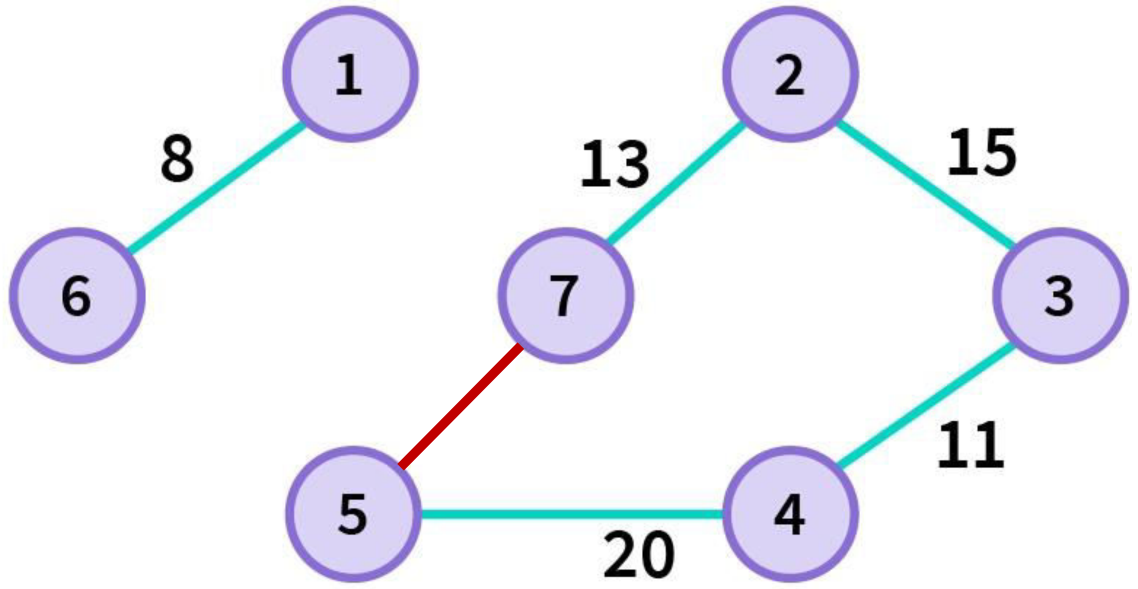




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

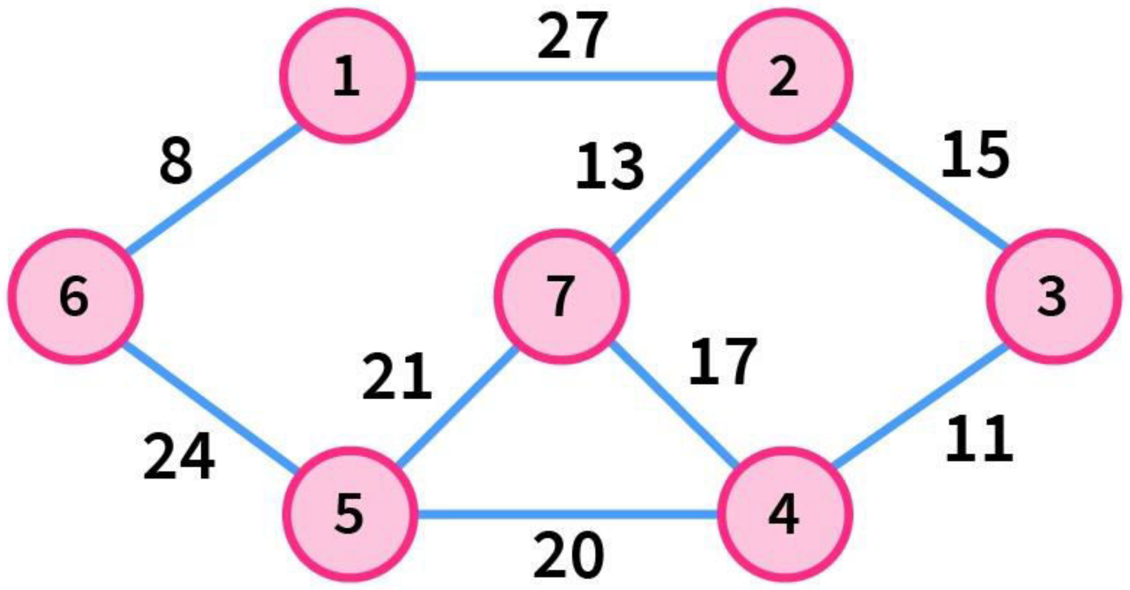
Kruskal’s Algorithm: Example 1

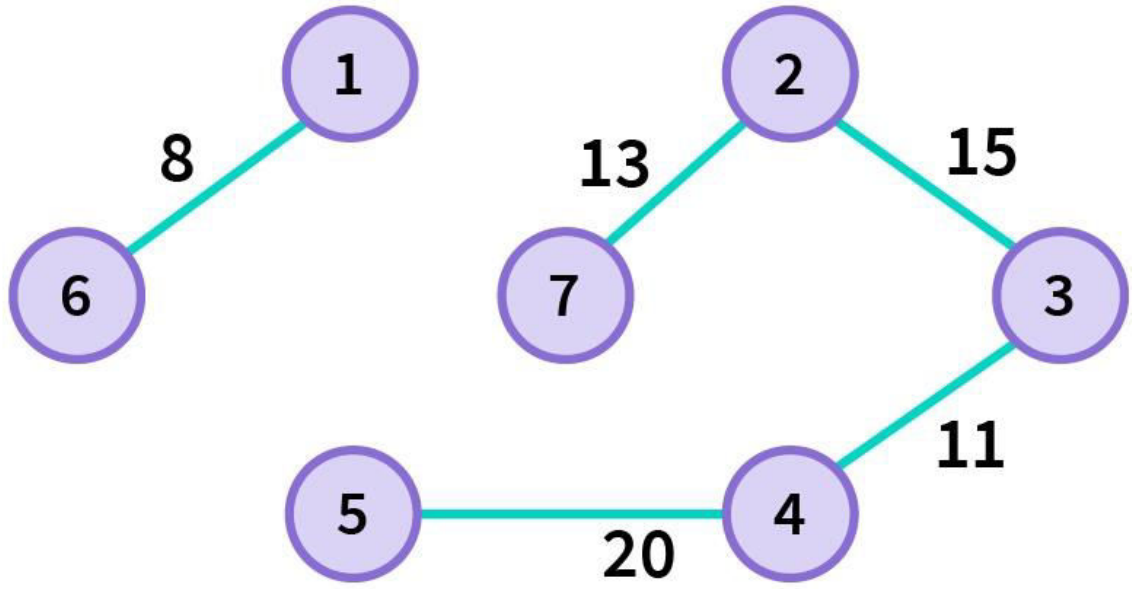




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

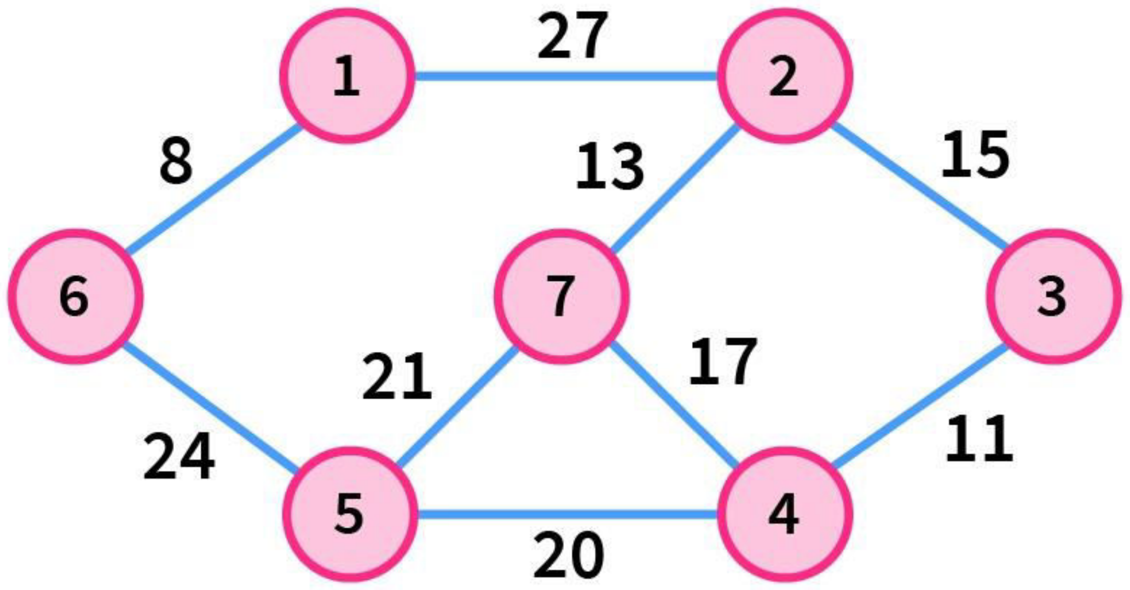
Kruskal’s Algorithm: Example 1

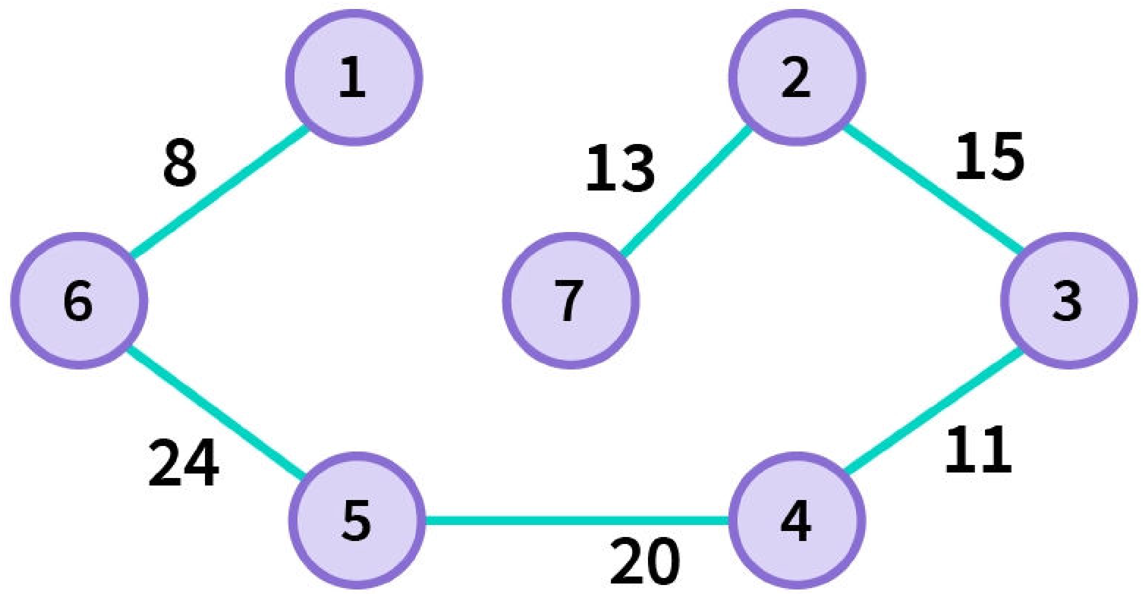




1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

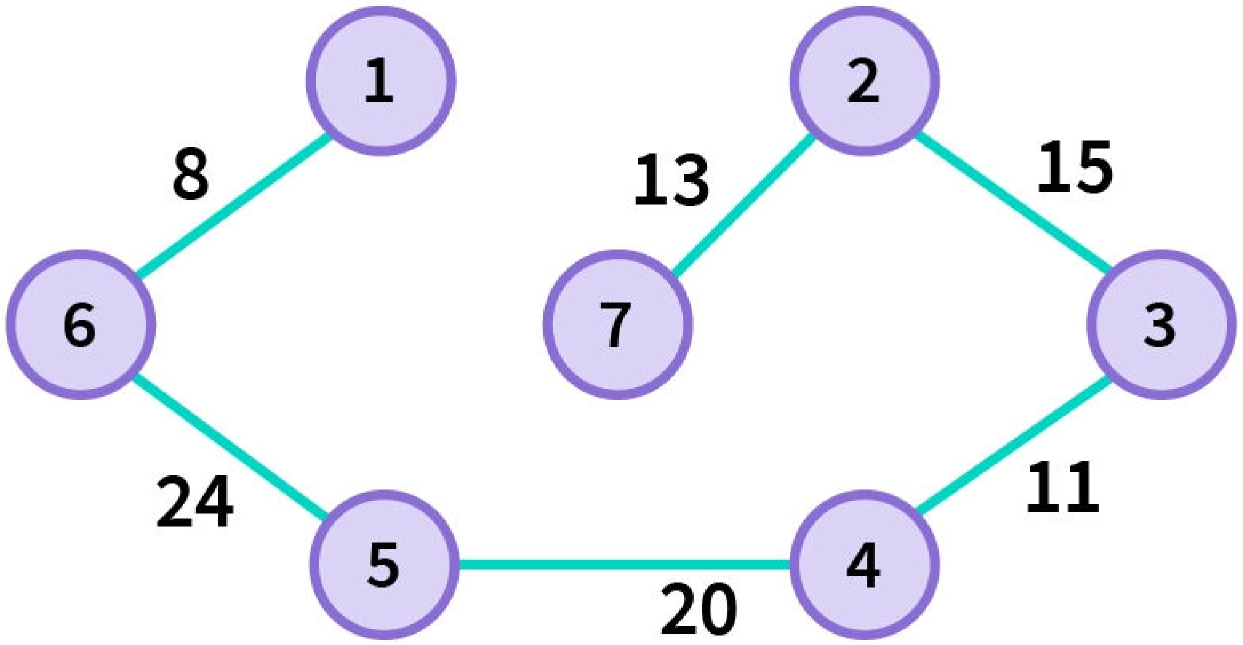
Kruskal’s Algorithm: Example 1





1-6	**3-4**	**2-7**	**2-3**	**4-7**	**4-5**	**5-7**	**5-6**	**1-2**
8	11	13	15	17	20	21	24	27

Kruskal’s Algorithm: Example 1



Total Weight = 8 + 24 + 20 + 11 +15 +13 = 91

1-6	3-4	2-7	2-3	4-7	4-5	5-7	5-6	1-2
8	11	13	15	17	20	21	24	27

Spanning Tree Applications

- Computer Network Routing Protocols
- Network Designing

  MST tells us the minimum amount of wire needed to connect all the nodes (servers).
- Laying Cables of Electrical Wiring
- Cluster Analysis
- Approximation of NP-Hard Problems, such as Traveling Salesman Problem (TSP).

Disjoint Set Data Structure or Union-Find Algorithm : *is defined as a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.*

A union-find algorithm is an algorithm that performs two useful operations on such a data structure:
Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
Union: Join two subsets into a single subset. Here first we have to check if the two subsets belong to same set. If no, then we cannot perform union.

Example:

Let us check an example to understand how the data structure is applied. For this consider the following problem statement

Problem: Given an undirected graph, the task is to check if the graph contains a cycle or not.

Initially create subsets containing only a single node which are the parent of itself. Now while traversing through the edges, if the two end nodes of the edge belongs to the same set then they form a cycle. Otherwise, perform union to merge the subsets together.

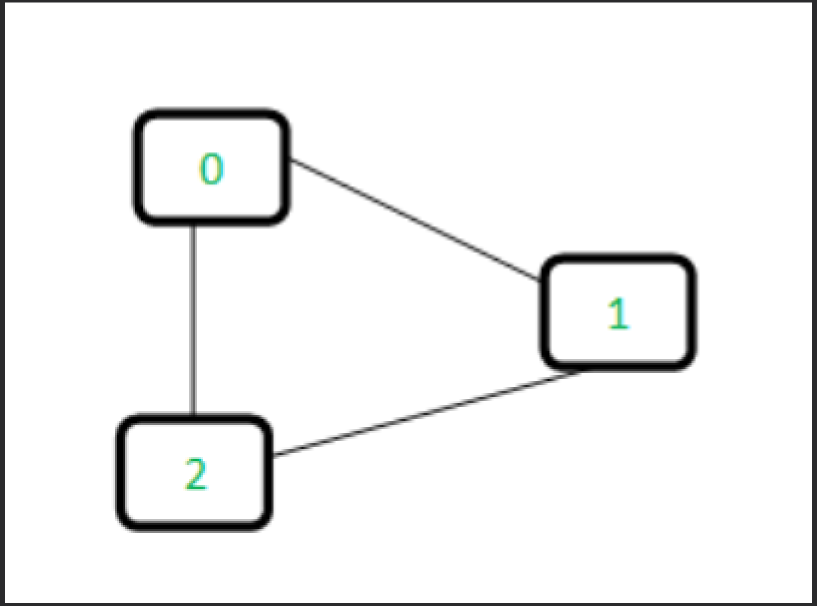
Note: *This method assumes that the graph doesn't contain any self-loops.*

Use an array to keep track of the subsets and which nodes belong to that subset. Let the array be ***parent[]****.*

Initially, all slots of parent array are initialized to hold the same values as the node.

parent[] = {0, 1, 2}*. Also when the value of the node and its parent are same, that is the root of that subset of nodes.*

Input: *The following is the graph*

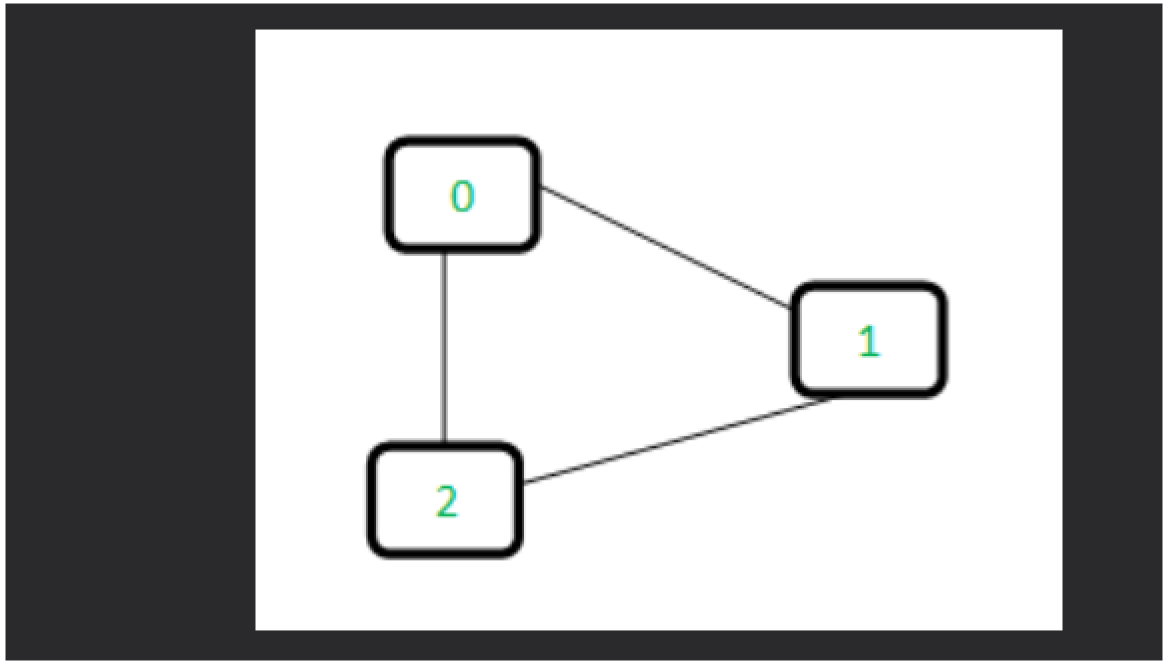


Output: *Yes*

Explanation: *There is a cycle of vertices {0, 1, 2}.*

parent[] = {0, 1, 2}.
Now process all edges one by one.
Edge 0-1:
=> Find the subsets in which vertices 0 and 1 are.
=> 0 and 1 belongs to subset 0 and 1.
=> Since they are in different subsets, take the union of them.
=> For taking the union, either make node 0 as parent of node 1 or vice-versa.
=> 1 is made parent of 0 (1 is now representative of subset {0, 1})
=> parent[] = {1, 1, 2}

Edge 1-2:
=> 1 is in subset 1 and 2 is in subset 2.
=> Since they are in different subsets, take union.
=> Make 2 as parent of 1. (2 is now representative of subset {0, 1, 2})
=> parent[] = {1, 2, 2}



Edge 0-2:
=> 0 is in subset 2 and 2 is also in subset 2.
=> Because 1 is parent of 0 and 2 is parent of 1. So 0 also belongs to subset 2
=> Hence, including this edge forms a cycle.
Therefore, the above graph contains a cycle.

Follow the below steps to implement the idea:

1- Initially create a **parent[]** array to keep track of the subsets.
2- Traverse through all the edges:
- Check to which subset each of the nodes belong to by finding the parent[] array till the node and the parent are the same.
- If the two nodes belong to the same subset then they belong to a cycle.
- Otherwise, perform union operation on those two subsets.

If no cycle is found, return false.

Nearest neighbor link solution better?

Prim’s on a graph to find minimum spanning tree:

1.Start at certain vertex

2.Choose nearest unvisited vertex (repeat)

   1. **This can be chosen from any vertex you have visited**

3.End when all vertices have been visited

Nearest neighbour when used for TSP solutions:

1.Start at certain vertex

2.Choose nearest unvisited vertex (repeat)

   1. **This can only be chosen from the vertex you are currently at**

3.Return to start when all vertices have been visited

So the main difference is that in Nearest neighbour you can only expand your selection from the vertices immediately connected to the one where you are currently at, whereas in Prim’s you can expand your selection from any vertex you have already visited.

References

- Graph in Data Structure - Scaler
- Graph Data Structure - Programiz
- Spanning Tree - Scaler
- Spanning Tree and Minimum Spanning Tree - Programiz
- Prim's Algorithm - Programiz
- Kruskal's Algorithm - Programiz
- Prim's Algorithm - Scaler
- Kruskal's Algorithm - Scaler
- Prims and Kruskal Algorithm - Scaler