Greedy Algorithms

Greedy Algorithm

- Most straight forward algorithm
- They are easy to invent, easy to implement and *– when they work –* efficient.
- Typically used to solve **Optimization Problems**
- *Crude Approach*, so many problems cannot be solved correctly.

Making Change (1) Problem

- Suppose, a country has following coins:

  100 paisa, 25 paisa, 10 paisa, 5 paisa & 1 paisa

- **Our Problem** is to devise an algorithm for paying a given amount using smallest possible number of coins.

- E.g. if we want to pay Rs. 2.89 (289 paisa)

  Then the best solution is to give 10 coins:

  2 X 100 paisa = 200 paisa (2 coins)

  3 X 25 paisa = 75 paisa (3 coins)

  1 X 10 paisa = 10 paisa (1 coin)

  4 X 1 paisa = 4 paisa (4 coins)

  TOTAL = 289 paisa (10 coins)

Making Change (1) Problem

- This is example of Greedy Algorithm
- For this problem we are always getting a **Optimal Solution**; however with a different series of values, or if the supply of some of the coins is limited, the greedy algorithm may not work.

- The algorithm is "greedy" because at every step it chooses the largest coin it can, without worrying whether this will prove to be a sound decision in the long run.

General Characteristics of Greedy Algorithm

- To construct the solution of our problem, we have a set of candidates. *(Available coins)*
- As algorithm proceeds, we accumulate two other sets. One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.
- There is a function that checks whether a particular set of candidates provides a solution to our problem.
- A second function checks whether a set of candidates is feasible.
- The selection function, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.
- Finally, an objective function gives the value of a solution.

Greedy Algorithm

```
function greedy(C: set) : set
        {C is the set of candidates}
    s = Ø  {we construct the solution in the set S}
    while  c <> Ø  and not solution(s) do
        x = select(c)
        c = c \ {x}
    if feasible ( s U {x} ) then s = s U {x}
    if solution(s) then return s
    else return "No Solution"
```

Graphs: Minimum Spanning Trees

- Let G = <N,A> be a *connected, undirected graph.*

  where N is the set of nodes and A is the set of edges. Each edge has given length.

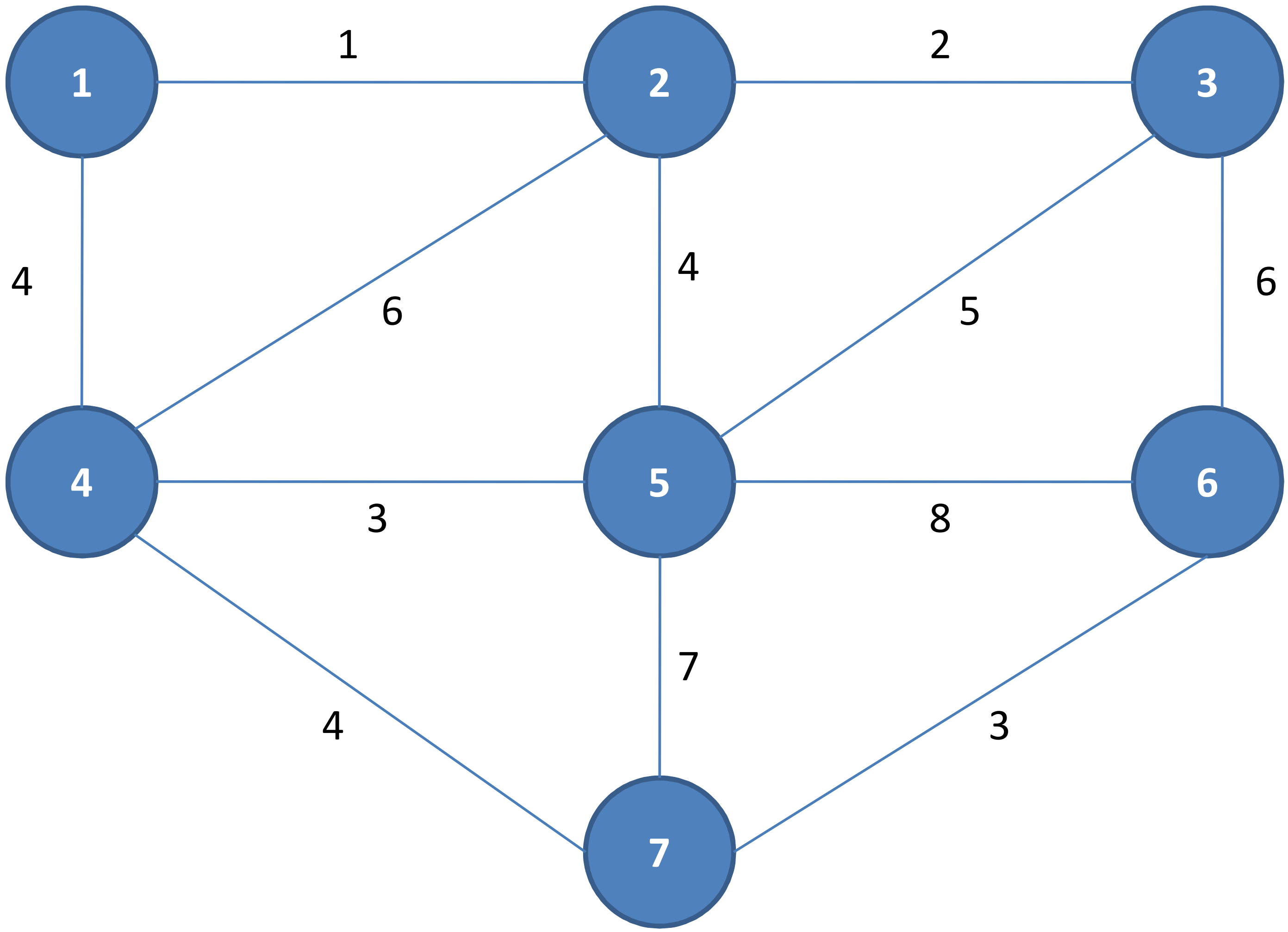
Problem: *The Problem is to find a subset T of the edges of G such that all the nodes remain connected, and the sum of the lengths of the edges in T is as small as possible.*

Note: A connected graph with n nodes must have at least n-1 edges, on other side, a graph with n nodes and more than n-1 edges contains at least one cycle.

Greedy Algorithm

- The candidates are the edges in G
- A set of edges in solution if it consists a spanning tree for nodes in N
- A set of edges is feasible if it does not include a cycle
- Objective is to minimize the total length

1. Kruskal's Algorithm (MST Problem)



Kruskal's Algorithm (MST Problem)

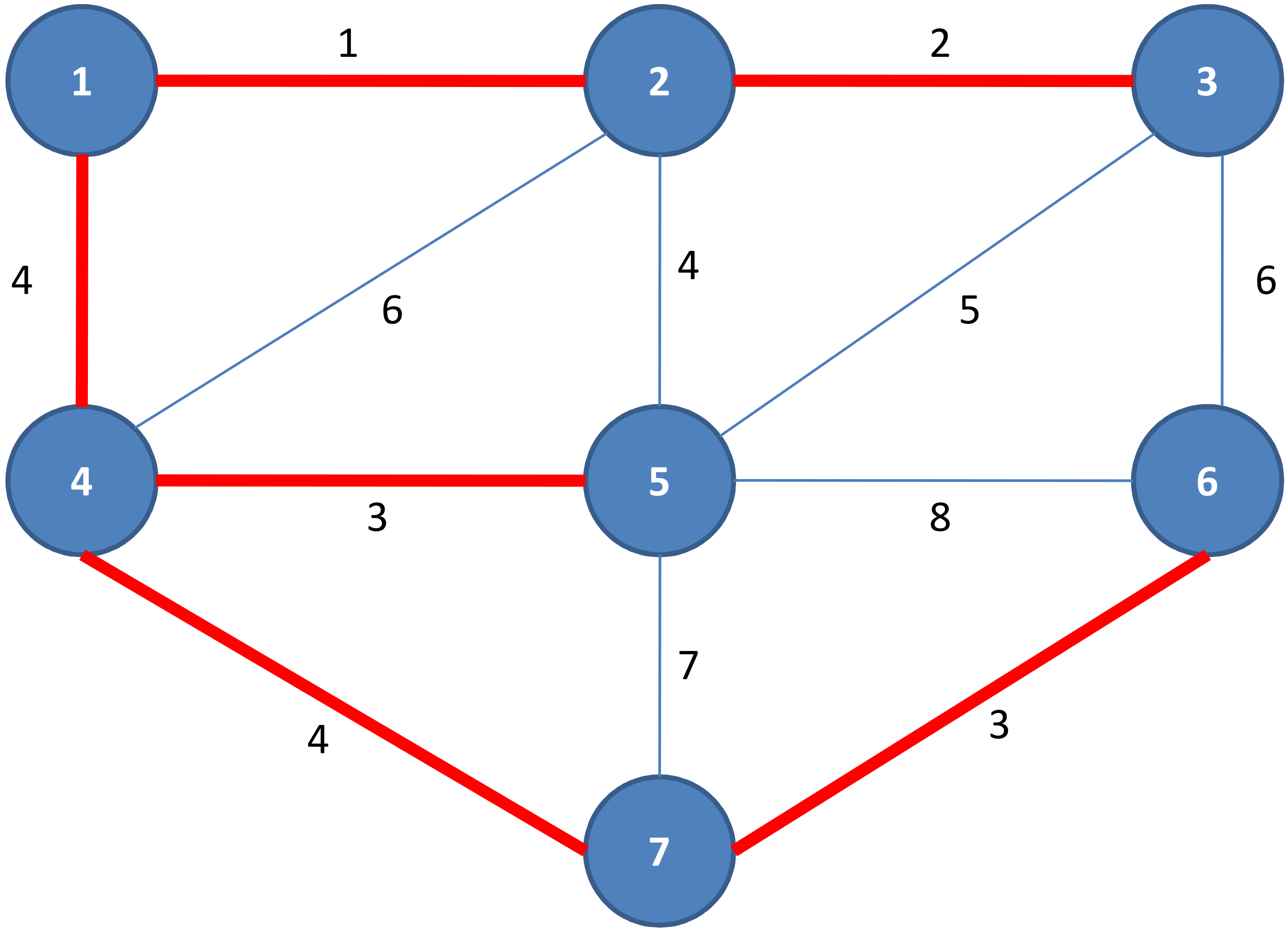
- Arrange all the edges of the graph in ***increasing order of their length***.
- So,

  {1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {2,5}, {4,7}, {3,5}, {2,4}, {3,6}, {5,7} and {5,6}

Kruskal's Algorithm (MST Problem)

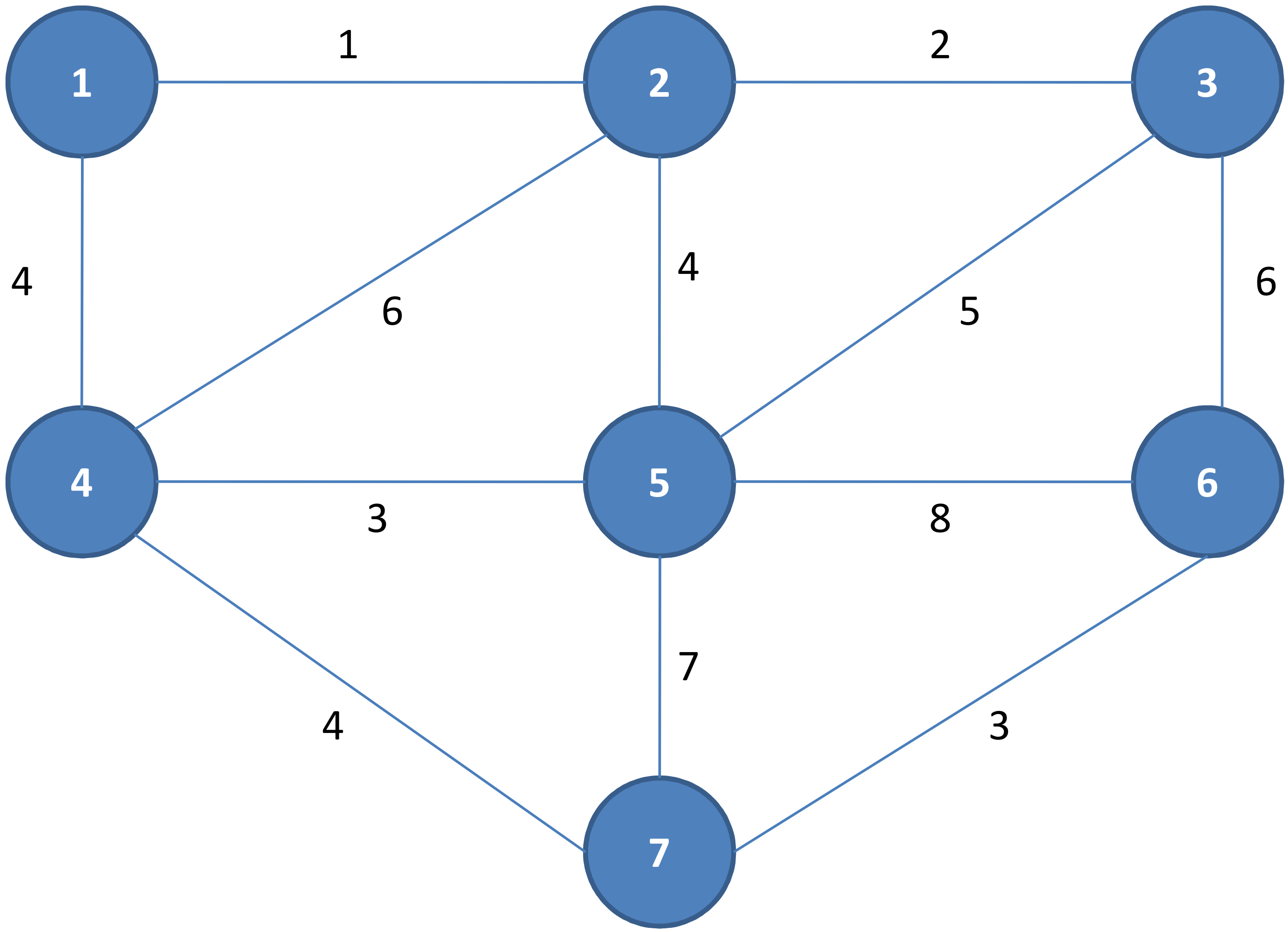
Step	Edge Considered	Connected Components
Initialization	--	{1} {2} {3} {4} {5} {6} {7}
1	**{1,2}**	{1,2} {3} {4} {5} {6} {7}
2	**{2,3}**	{1,2,3} {4} {5} {6} {7}
3	**{4,5}**	{1,2,3} {4,5} {6} {7}
4	**{6,7}**	{1,2,3} {4,5} {6,7}
5	**{1,4}**	{1,2,3,4,5} {6,7}
6	*{2,5}*	*Rejected*
7	**{4,7}**	{1,2,3,4,5,6,7}

Kruskal's Algorithm (MST Problem)



Total Length = 1+2+3+3+4+4= 17

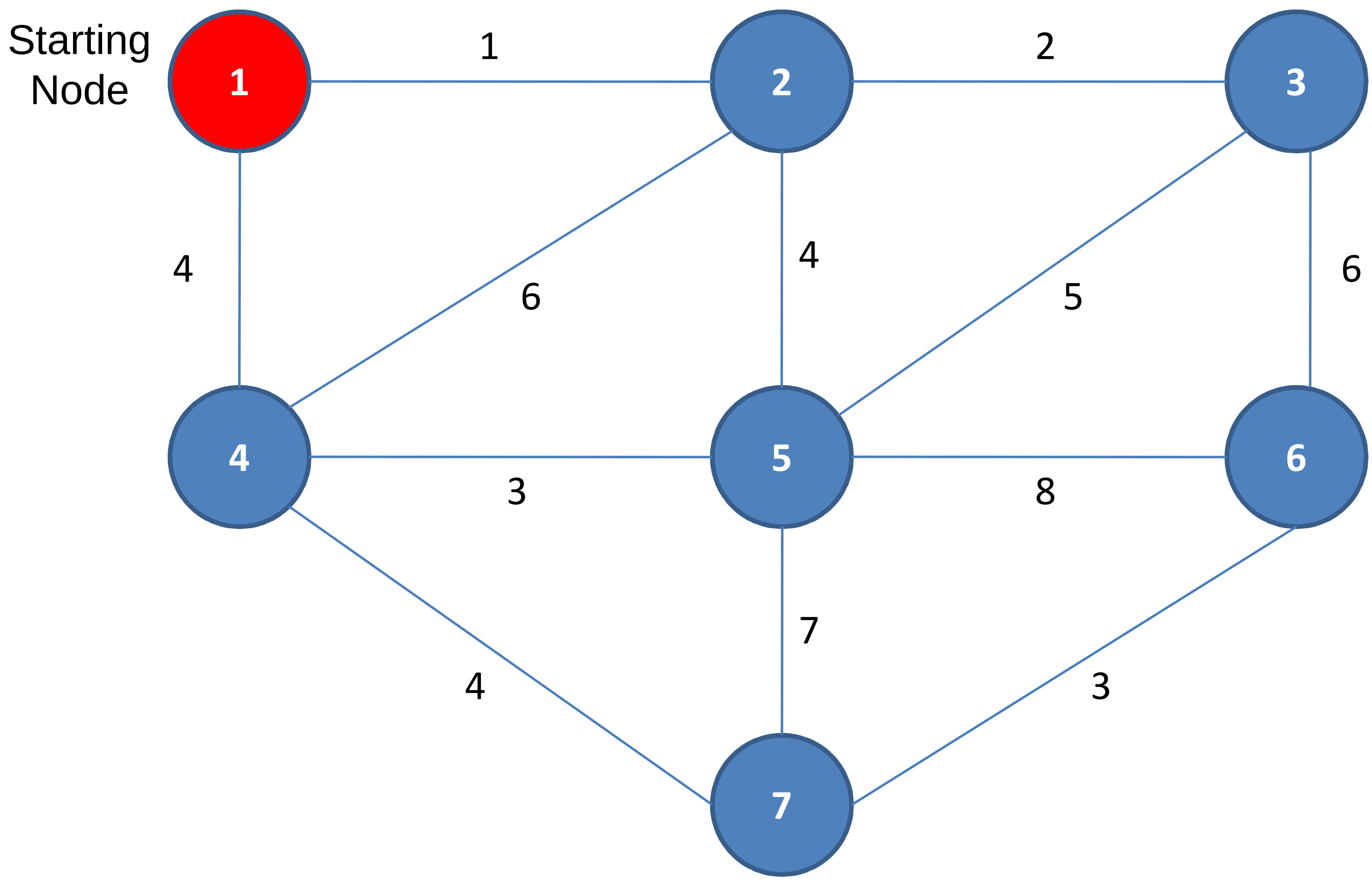
2. Prim's Algorithm (MST Problem)



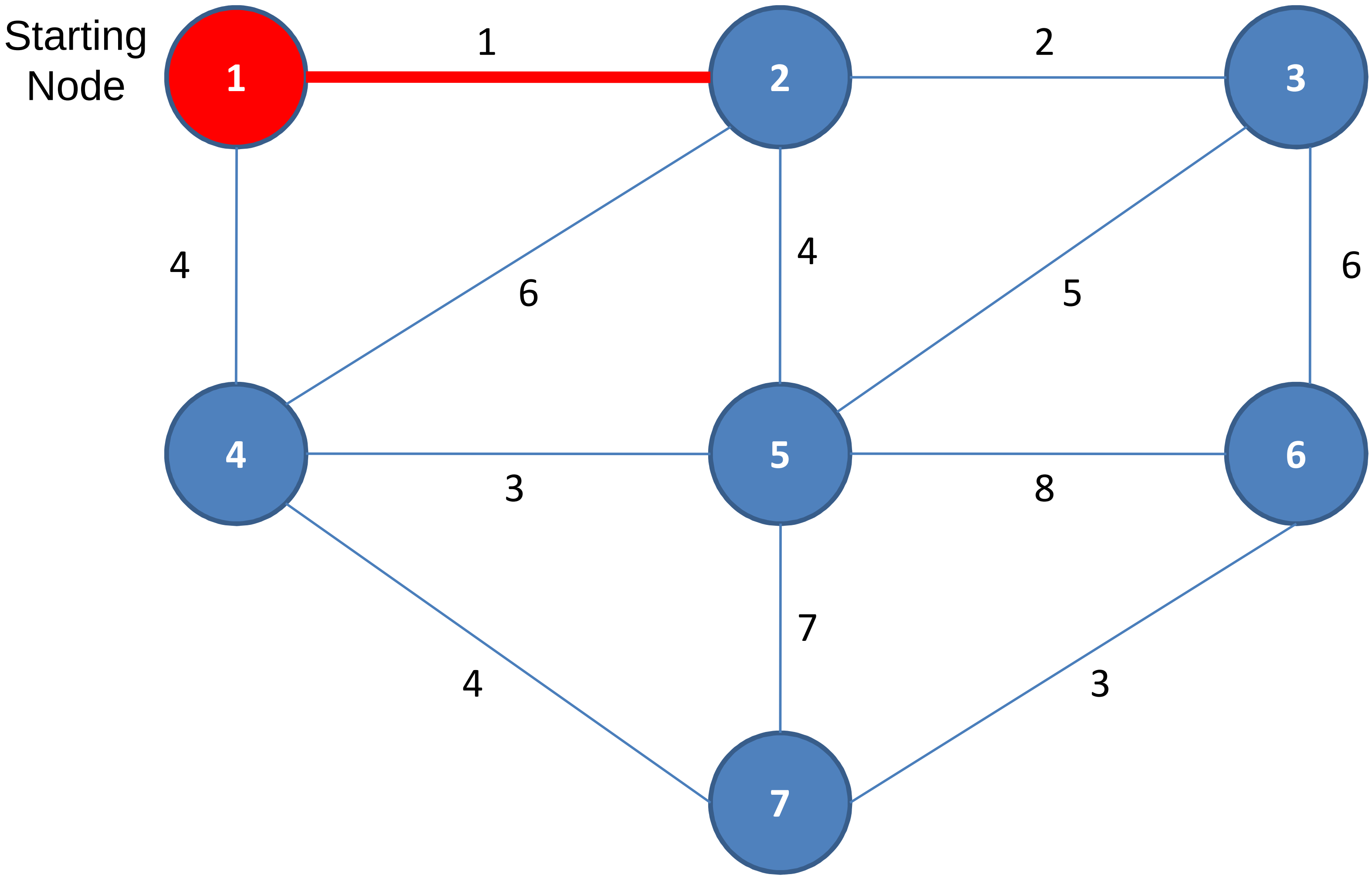
Prim's Algorithm (MST Problem)

- In this algorithm, the minimum spanning tree grows in a natural way, starting from an arbitrary root.
- At each stage we add a new branch to the tree already constructed.
- The algorithm stops when all the nodes have been reached.

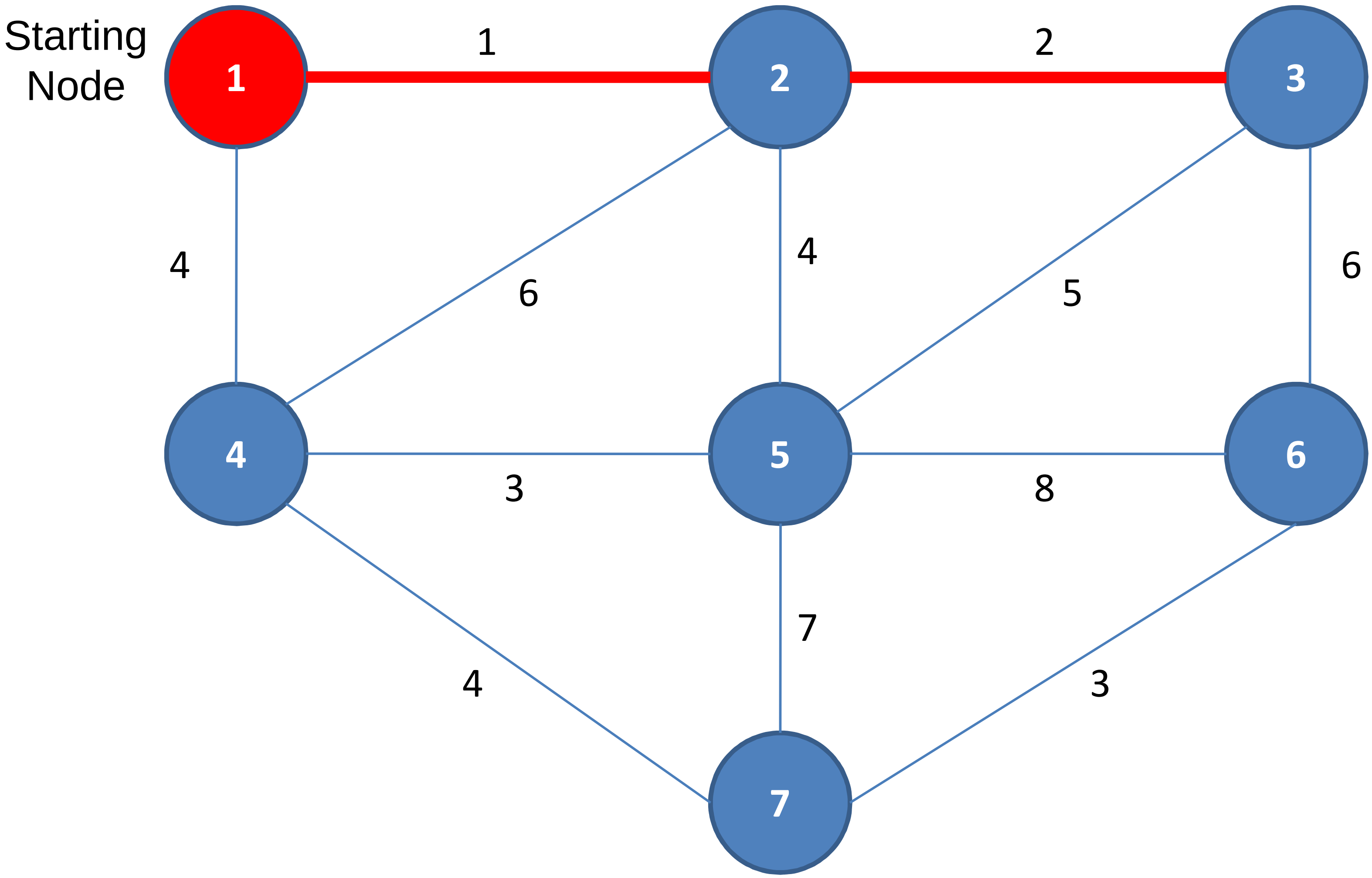
Prim's Algorithm (MST Problem)



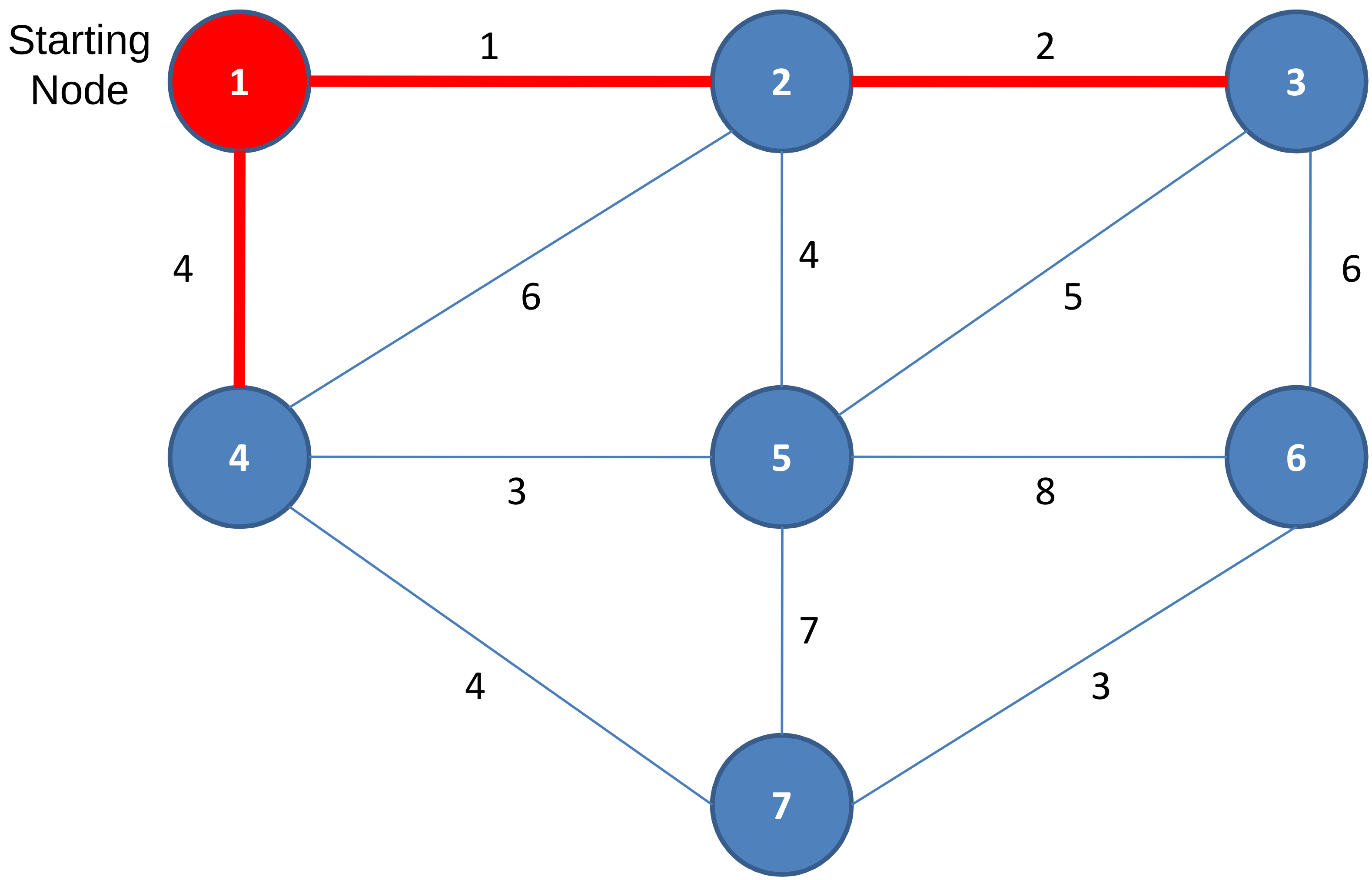
Prim's Algorithm (MST Problem)



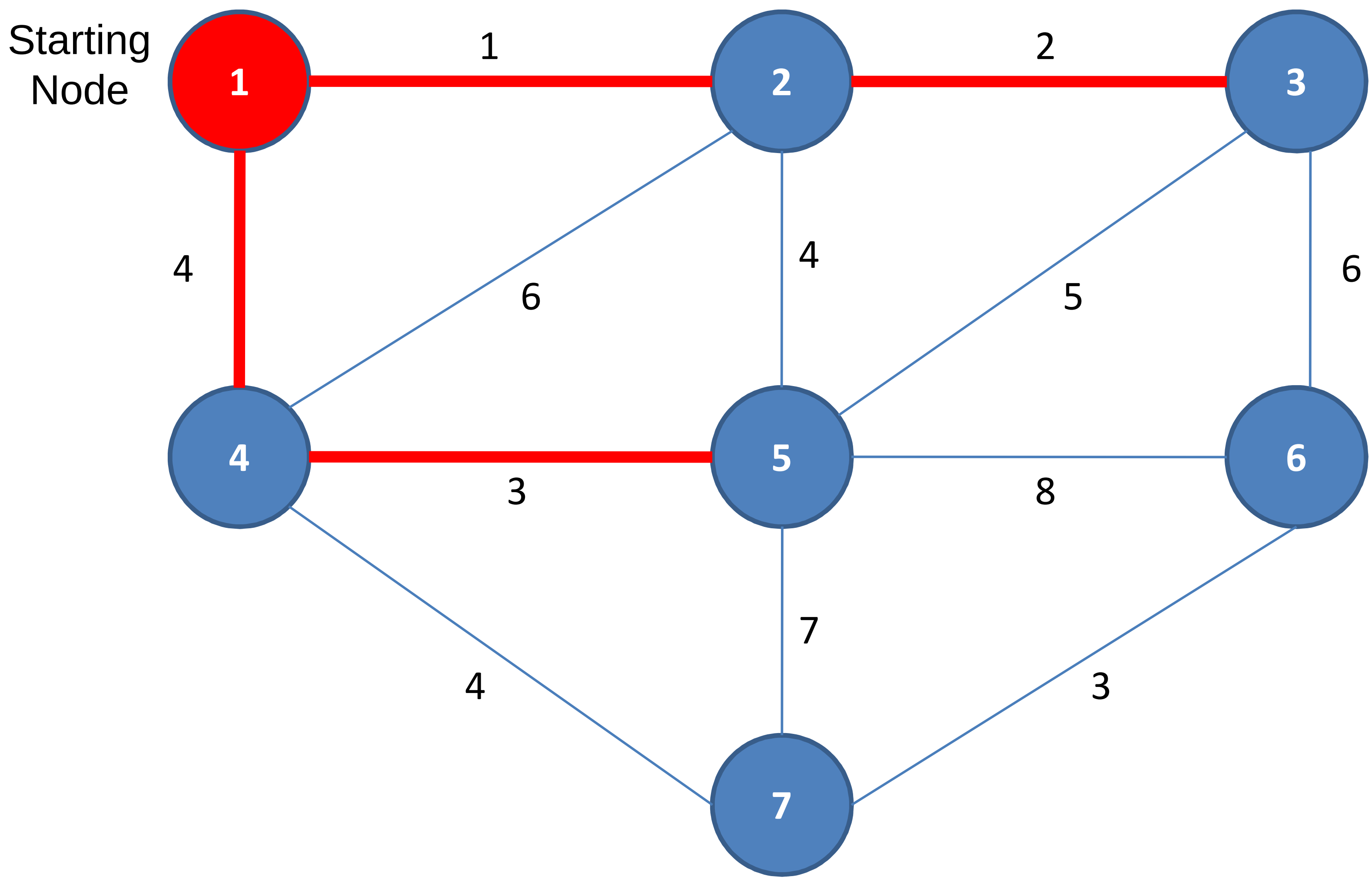
Prim's Algorithm (MST Problem)



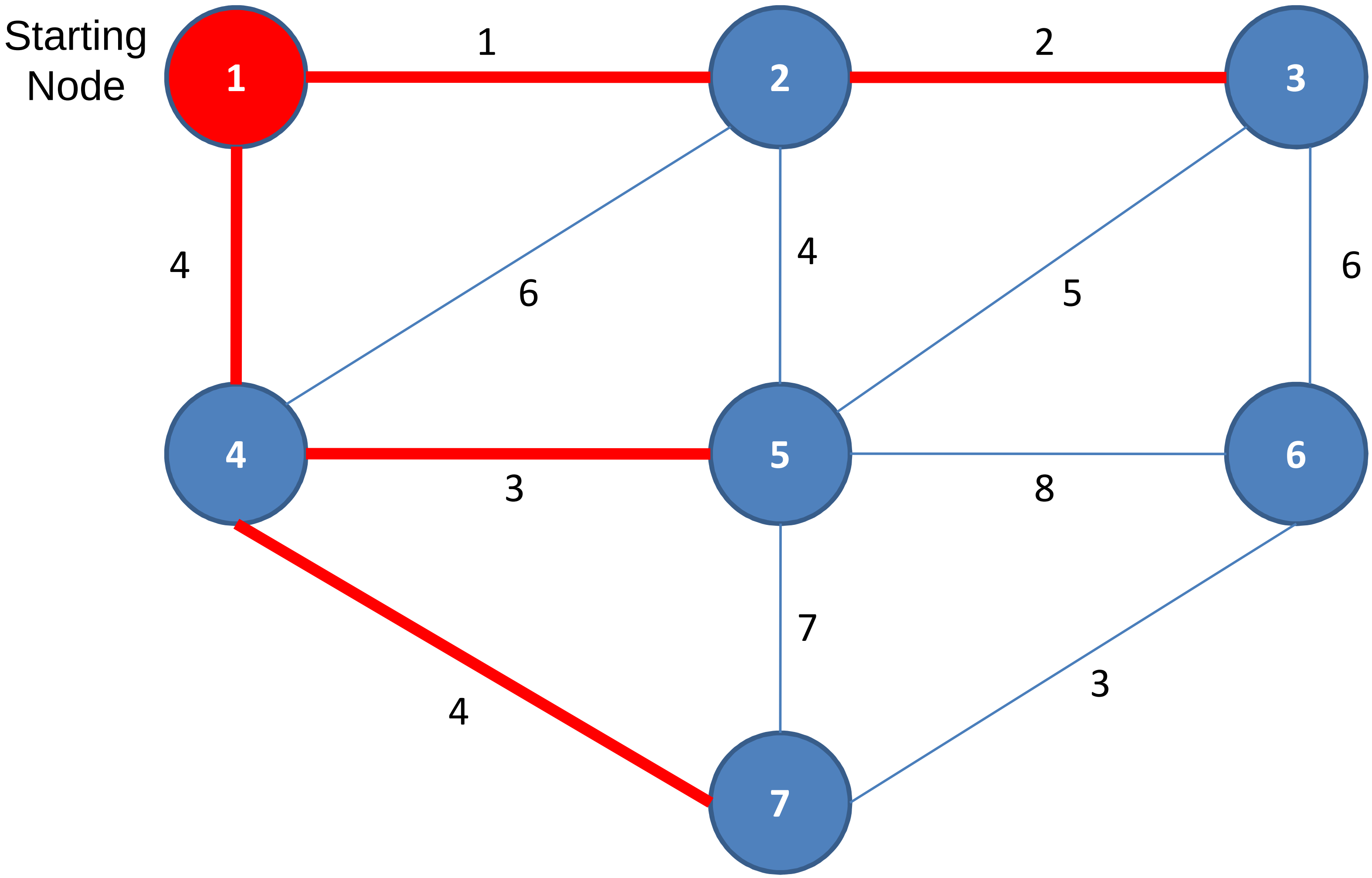
Prim's Algorithm (MST Problem)



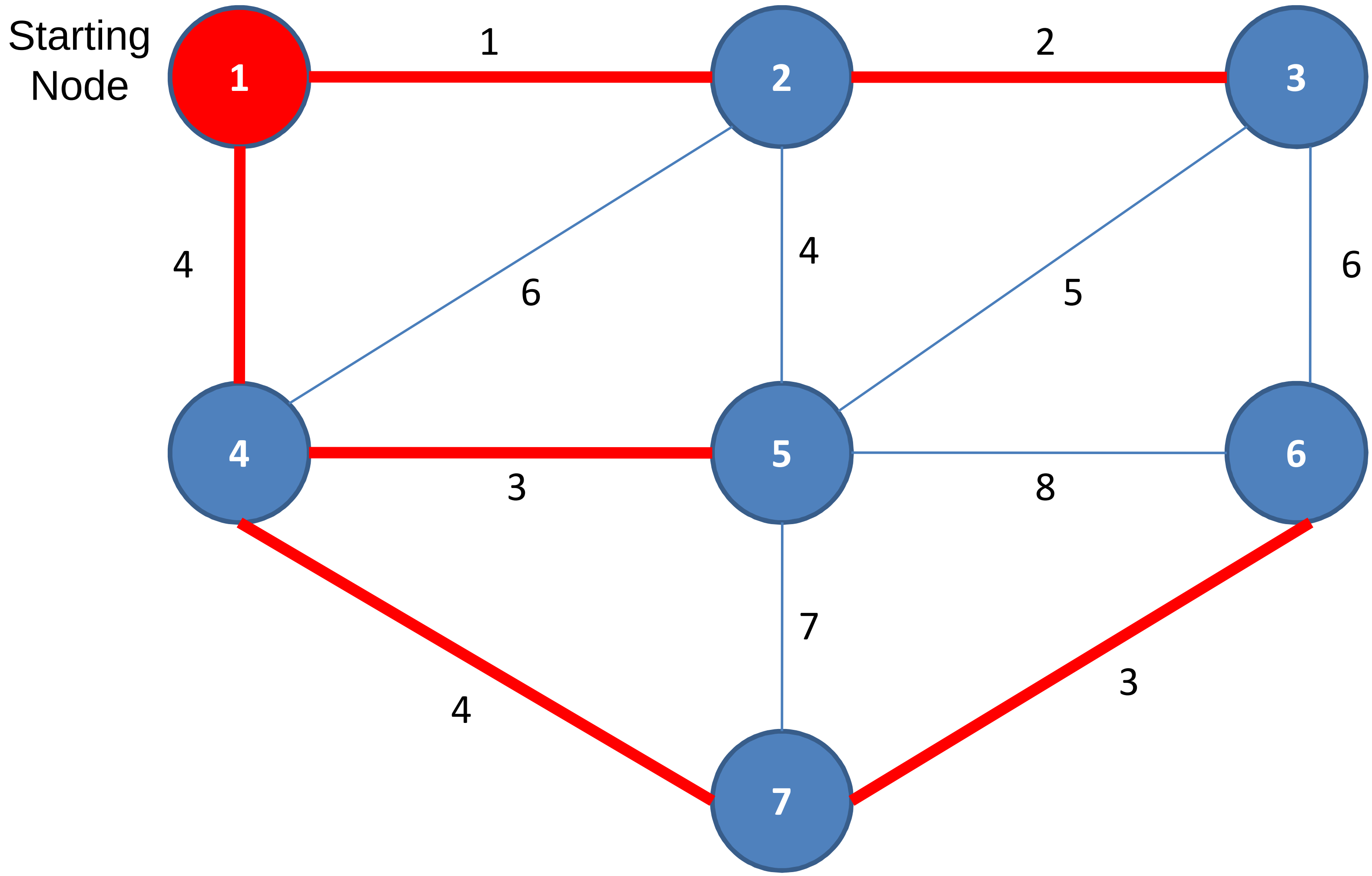
Prim's Algorithm (MST Problem)



Prim's Algorithm (MST Problem)



Prim's Algorithm (MST Problem)

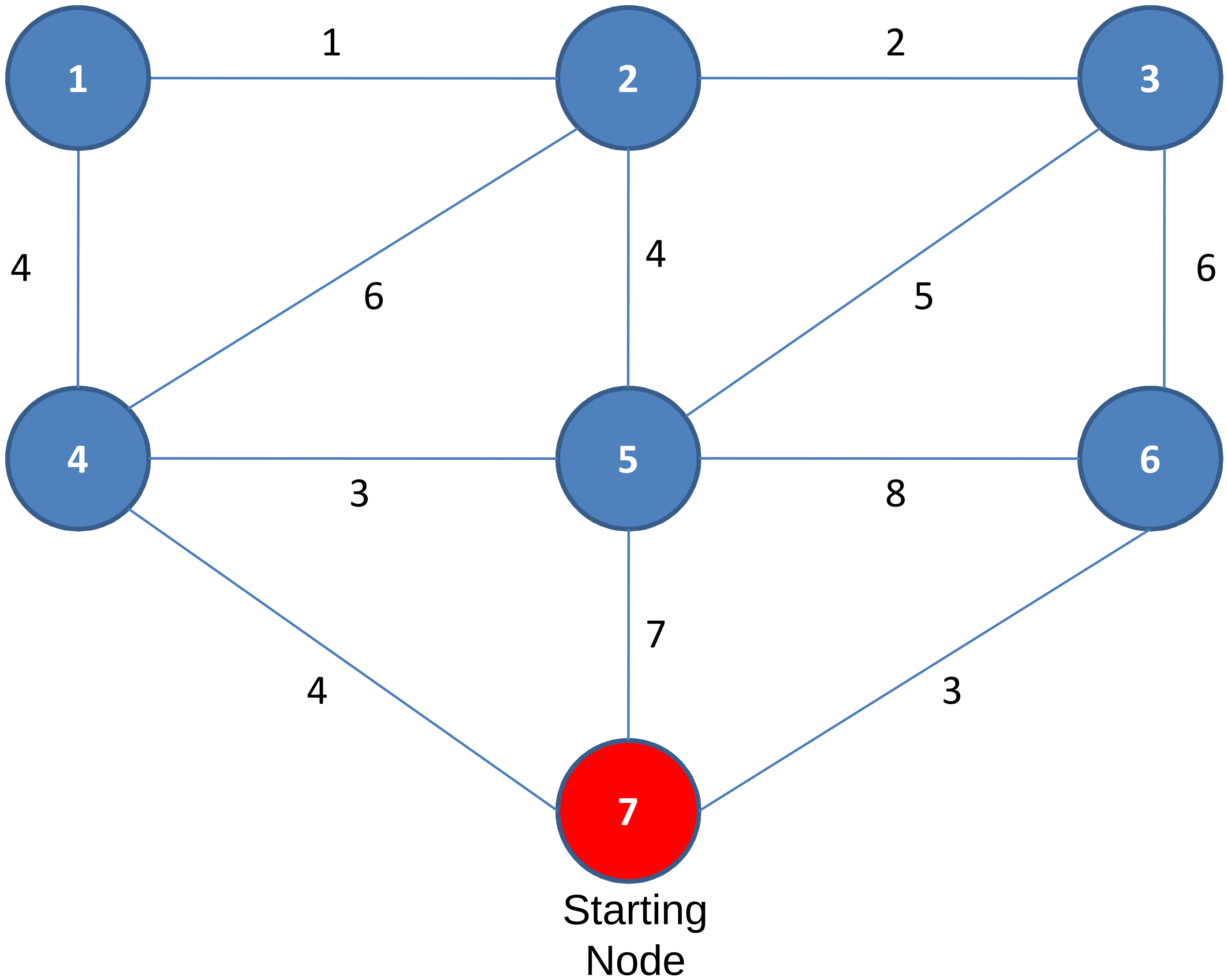


Total Length = 1+2+3+3+4+4= 17

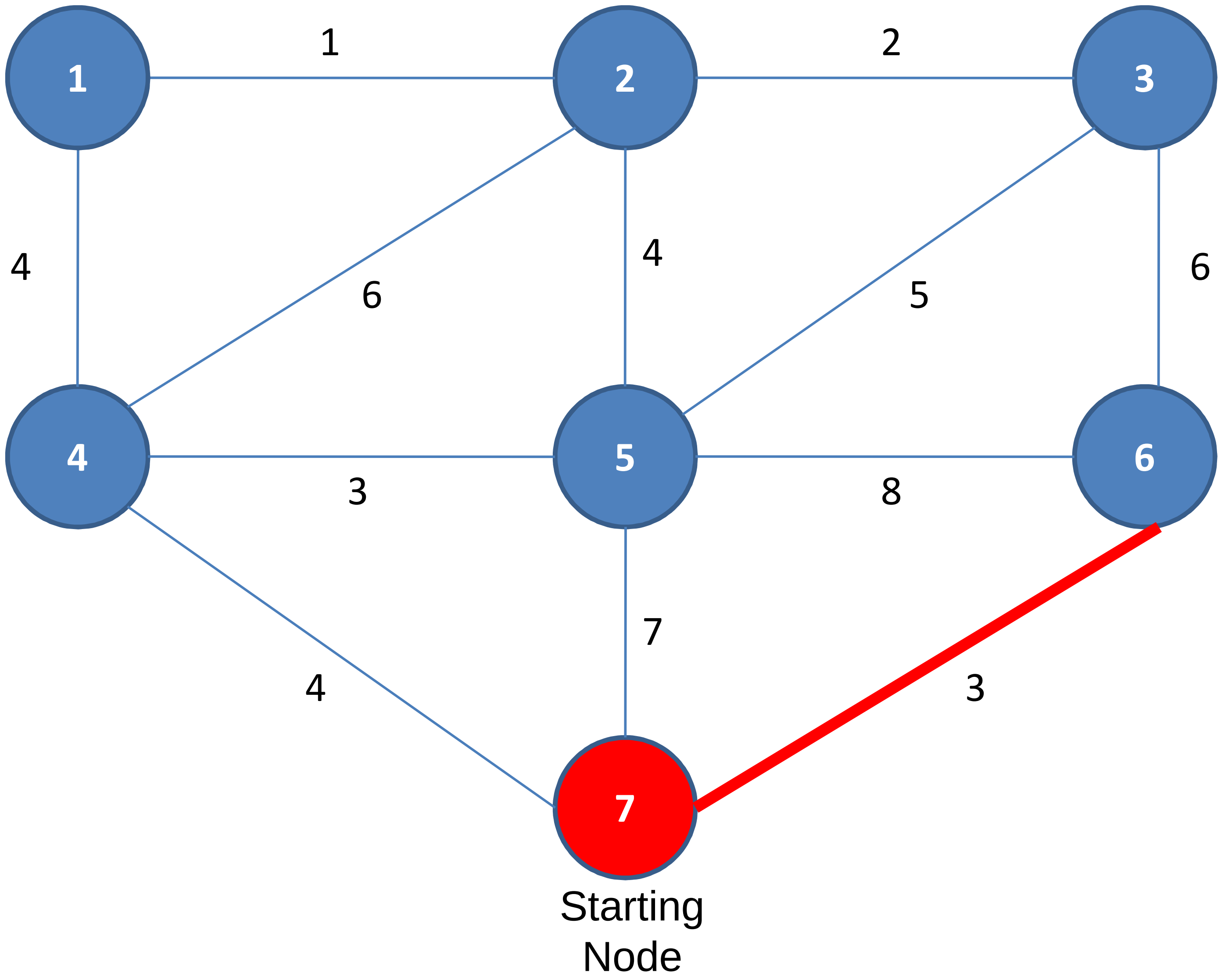
Prim's Algorithm (MST Problem)

Step	{U,V}	B
Initialization	--	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

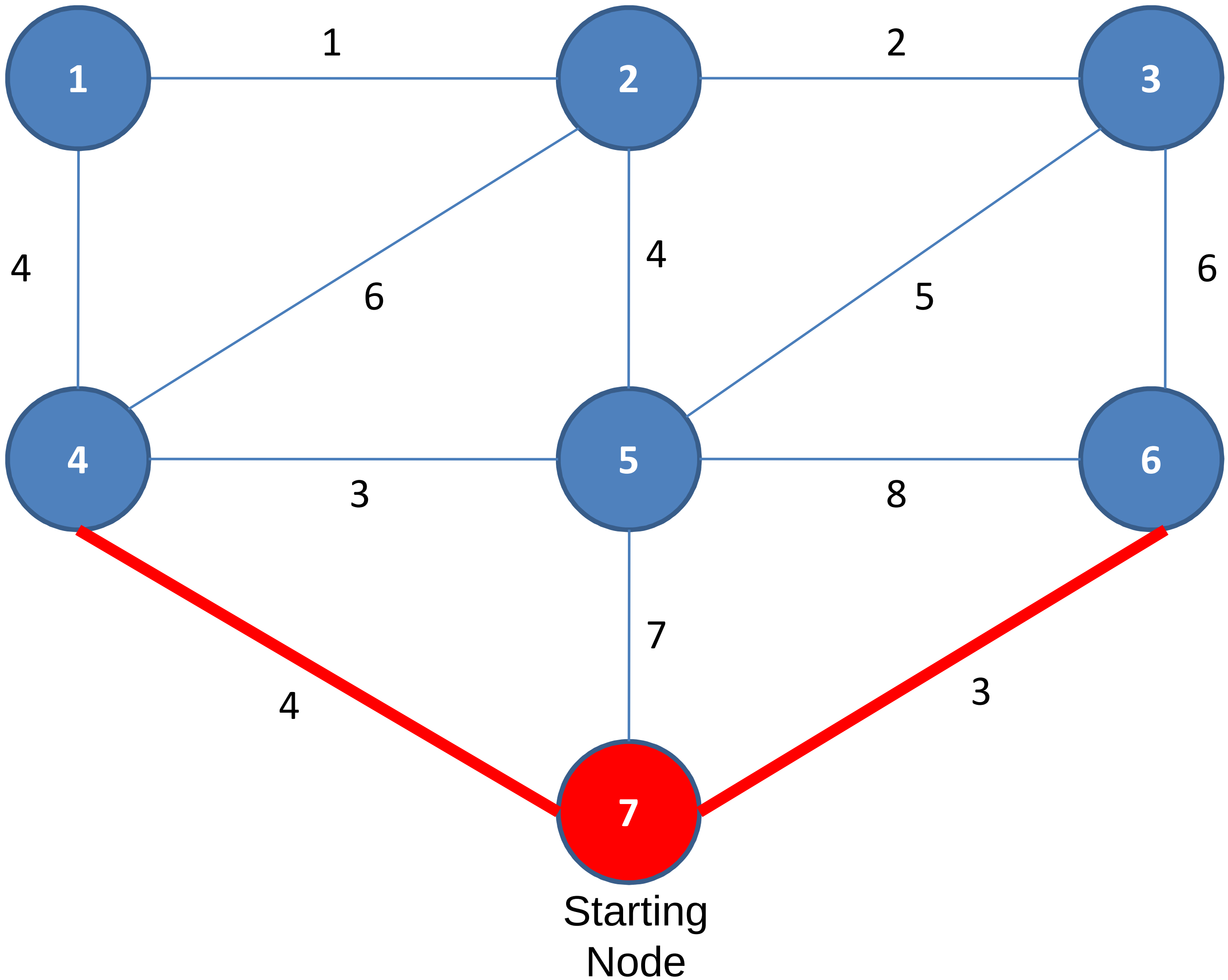
Prim's Algorithm (MST Problem)



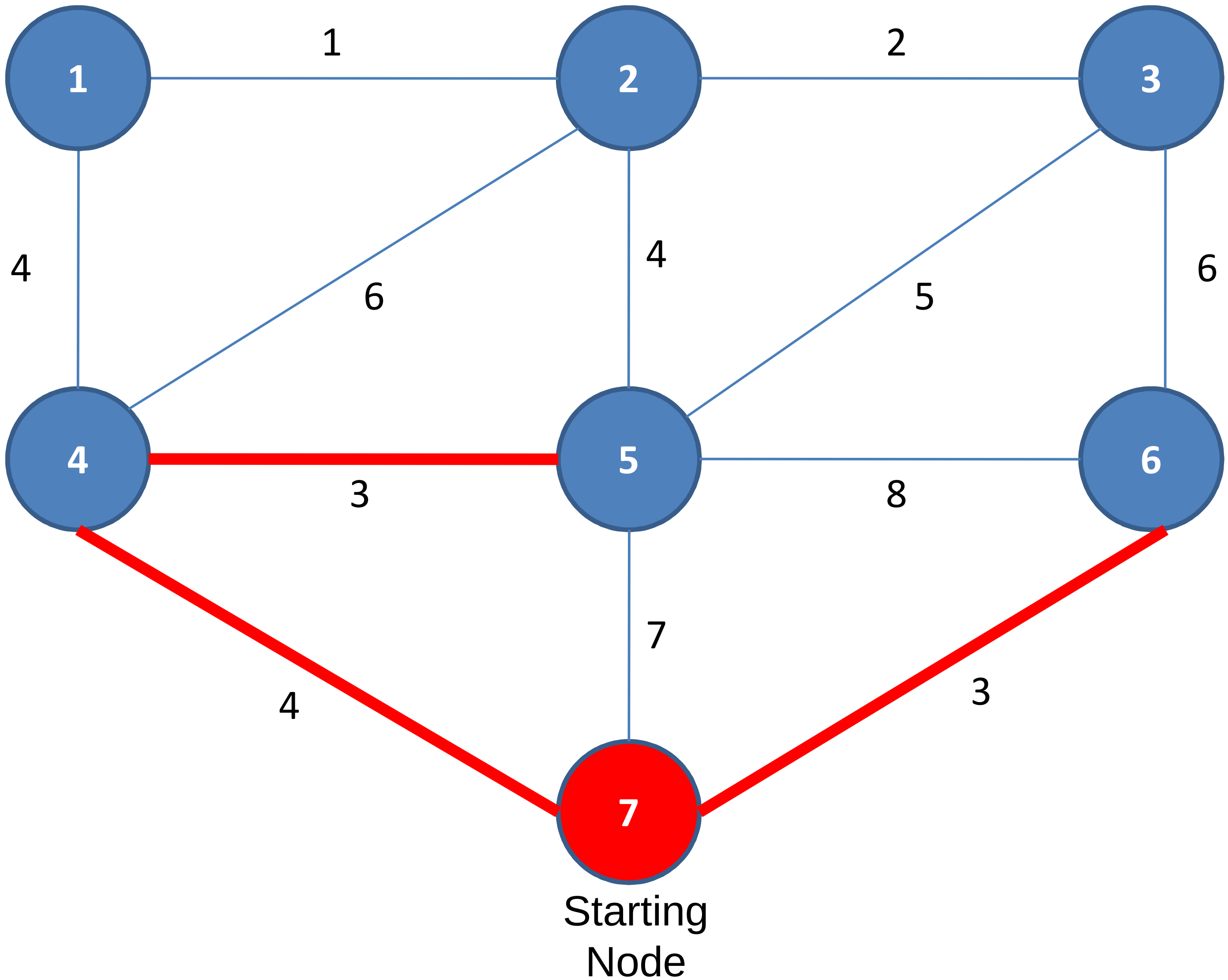
Prim's Algorithm (MST Problem)



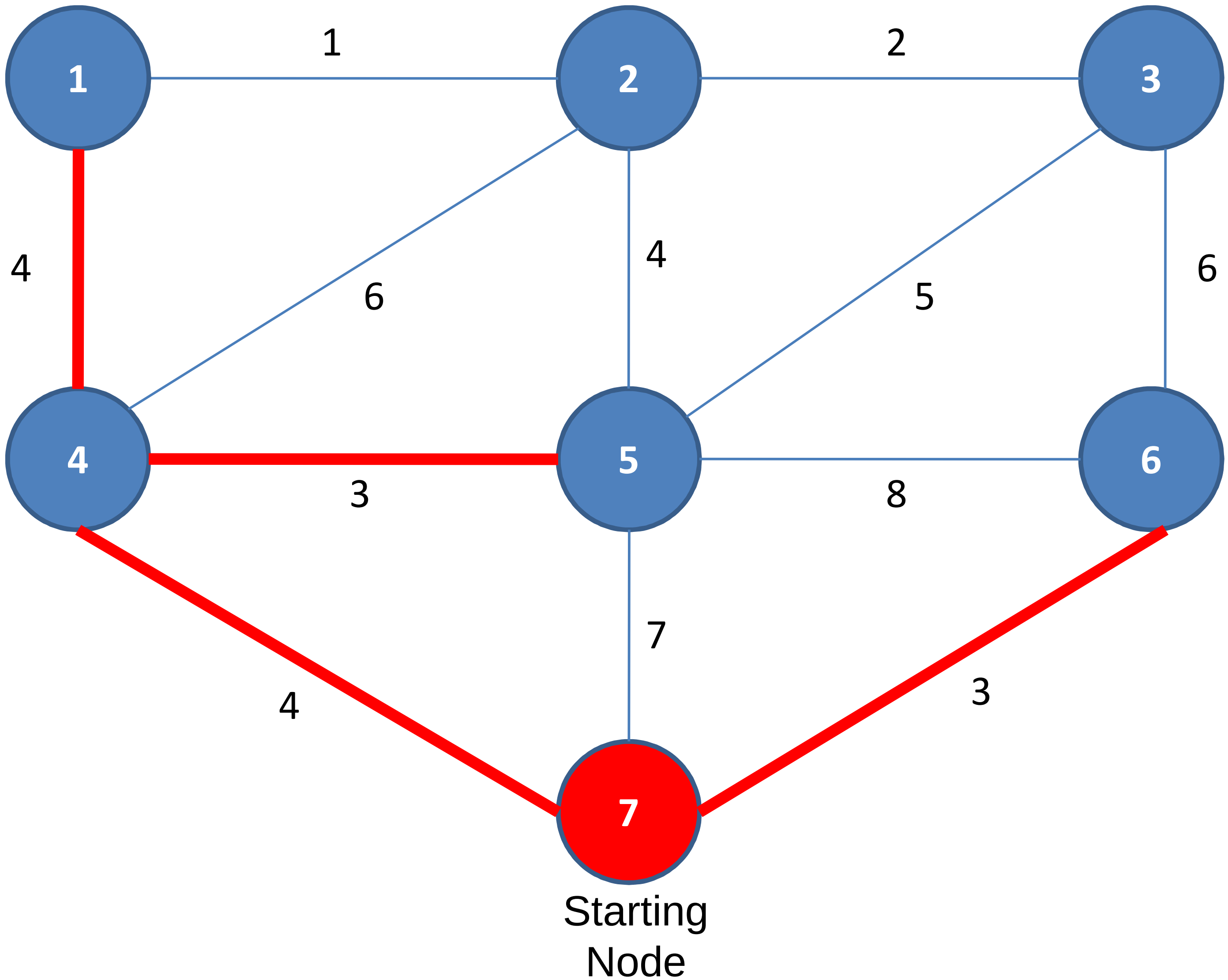
Prim's Algorithm (MST Problem)



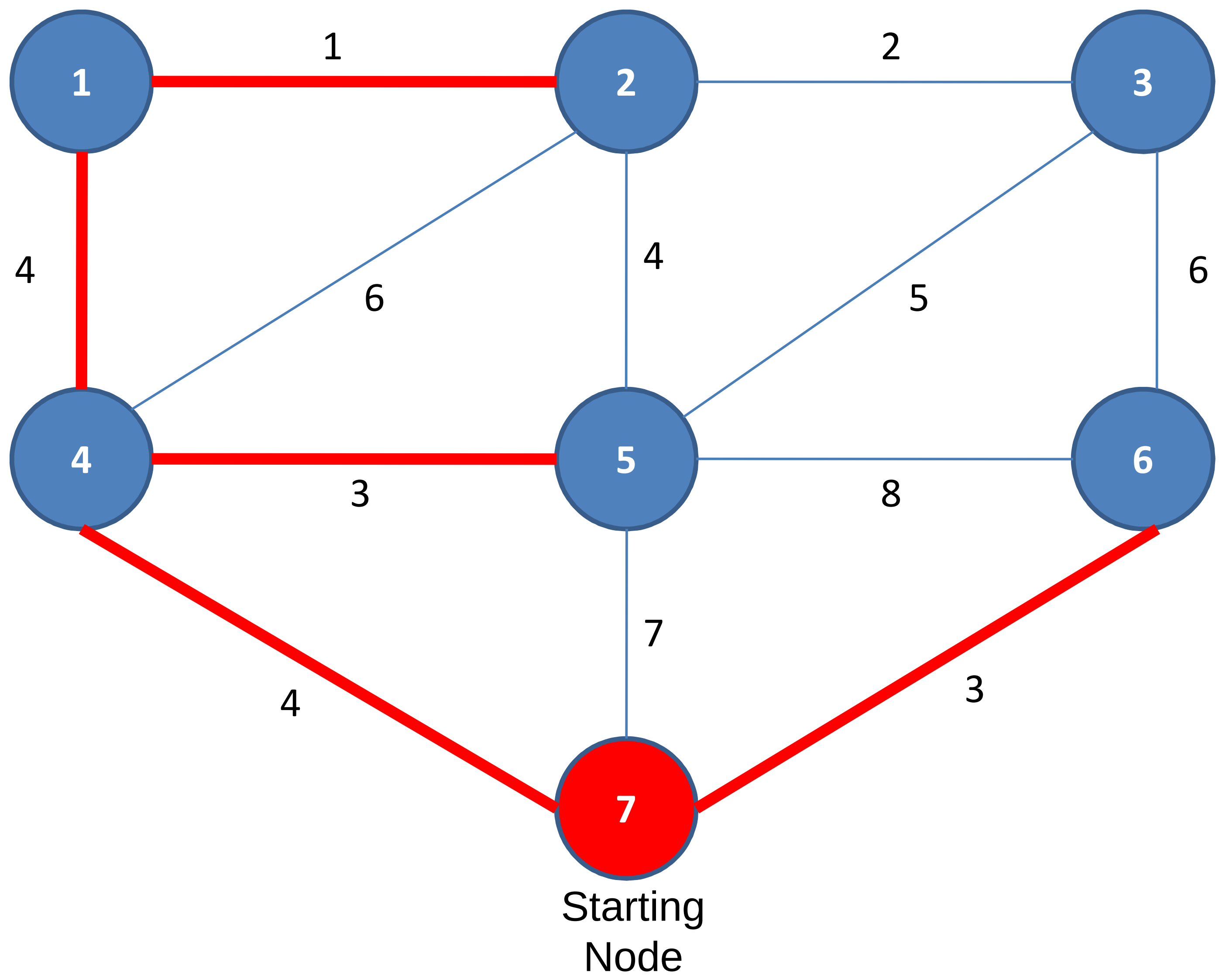
Prim's Algorithm (MST Problem)



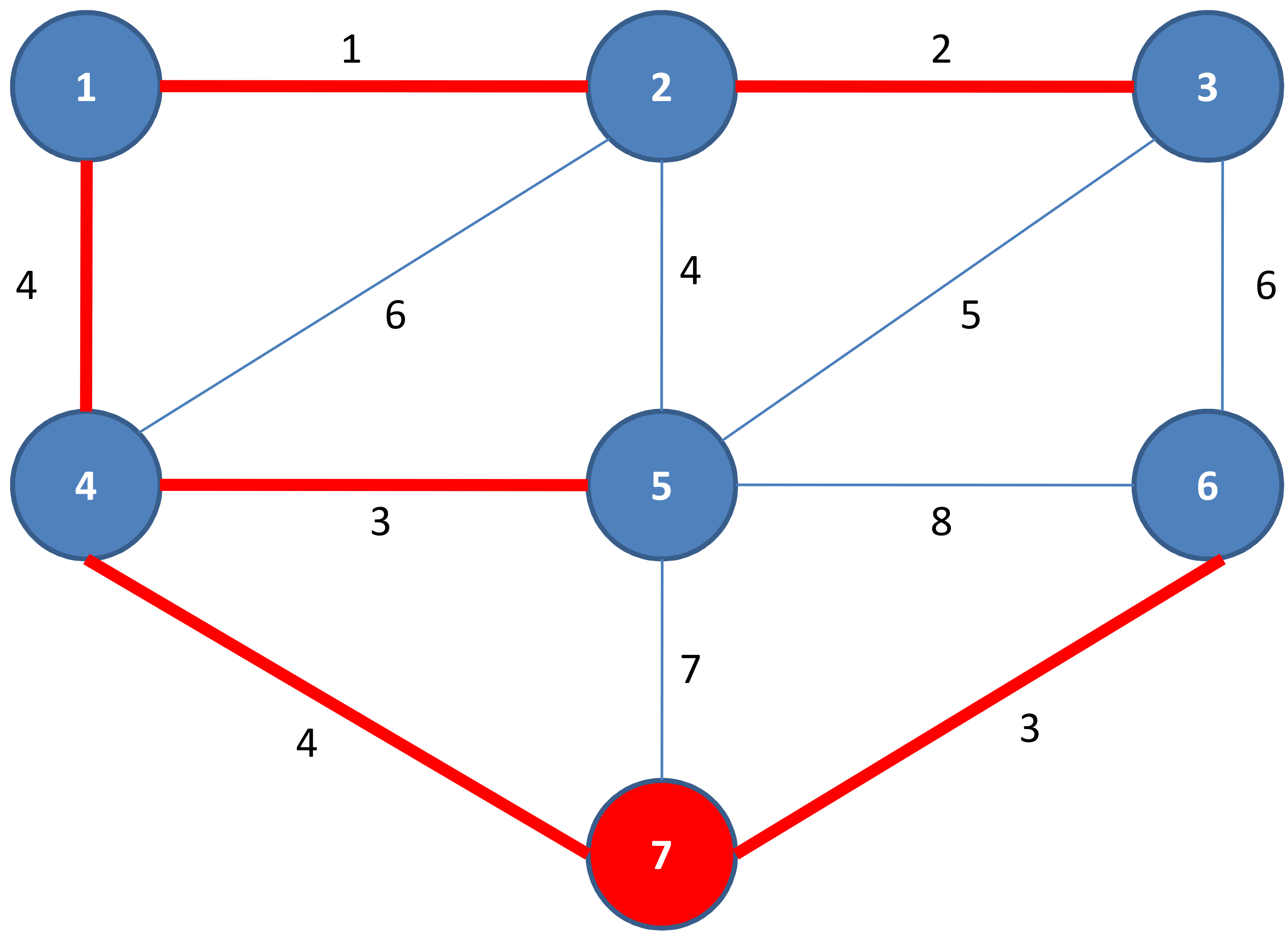
Prim's Algorithm (MST Problem)



Prim’s Algorithm (MST Problem)



Prim's Algorithm (MST Problem)



Total Length = 1+2+3+3+4+4= 17

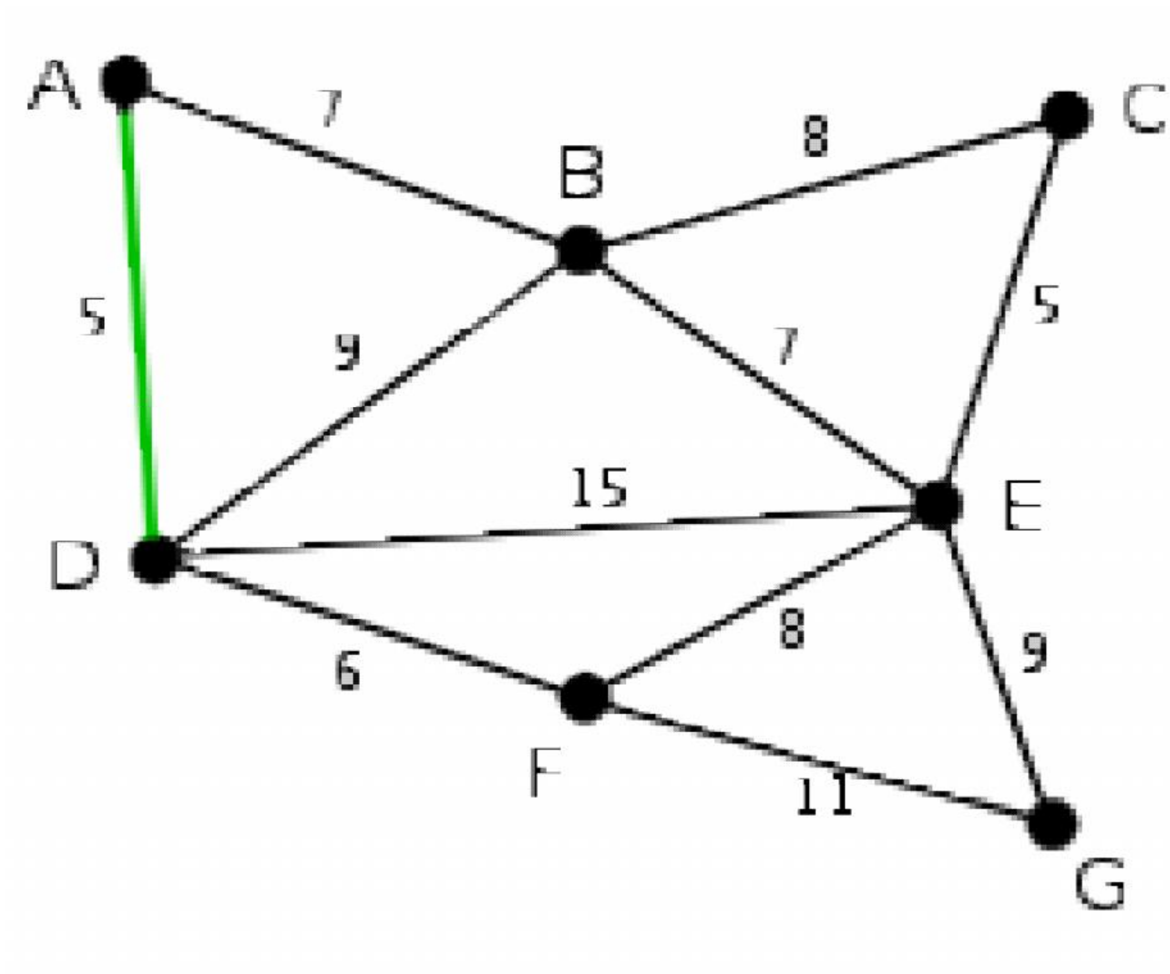
Prim's Algorithm (MST Problem)

Step	{U,V}	B
Initialization	--	{7}
1	{7,6}	{6,7}
2	{7,4}	{4,6,7}
3	{4,5}	{4,5,6,7}
4	{4,1}	{1,4,5,6,7}
5	{1,2}	{1,2,4,5,6,7}
6	{2,3}	{1,2,3,4,5,6,7}

Comparison of Kruskal’s & Prim’s Algorithm

- For a graph with **V** vertices **E** edges, Kruskal's algorithm runs in **O(E log V)** time and Prim's algorithm can run in **O(E + V log V)** amortized time.
- **Prim's algorithm is significantly faster** in the limit when you've got a really **dense graph** with many more edges than vertices. **Kruskal performs better** in typical situations (**sparse graphs**) because it uses simpler data structures.

Another Example (MST Problem)



Answer