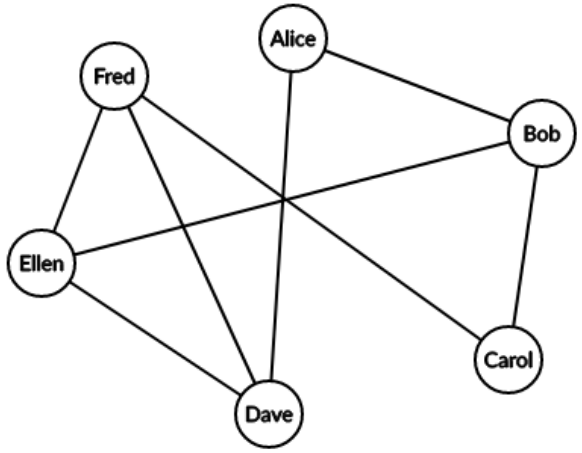
ALGODAY Vol. 1 May 2023

SPIROS MAGGIOROS – GRAPH THEORY WITH CODING

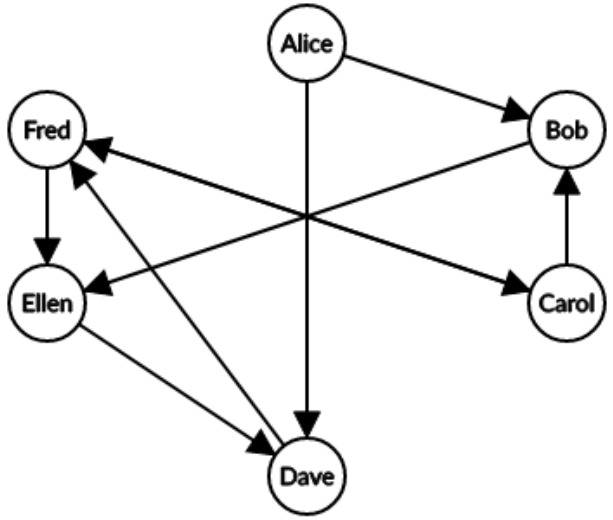
Introduction to Graph Theory - Non directed graphs

Graphs are one of the key algorithmic concepts and learning how to work with them is essential for anyone interested in computer science. So, what exactly is a graph? Basically, it's just a way to represent relationships between objects. Let's take a look at a simple example. Imagine we have a group of people and we know for each pair whether or not they are friends.



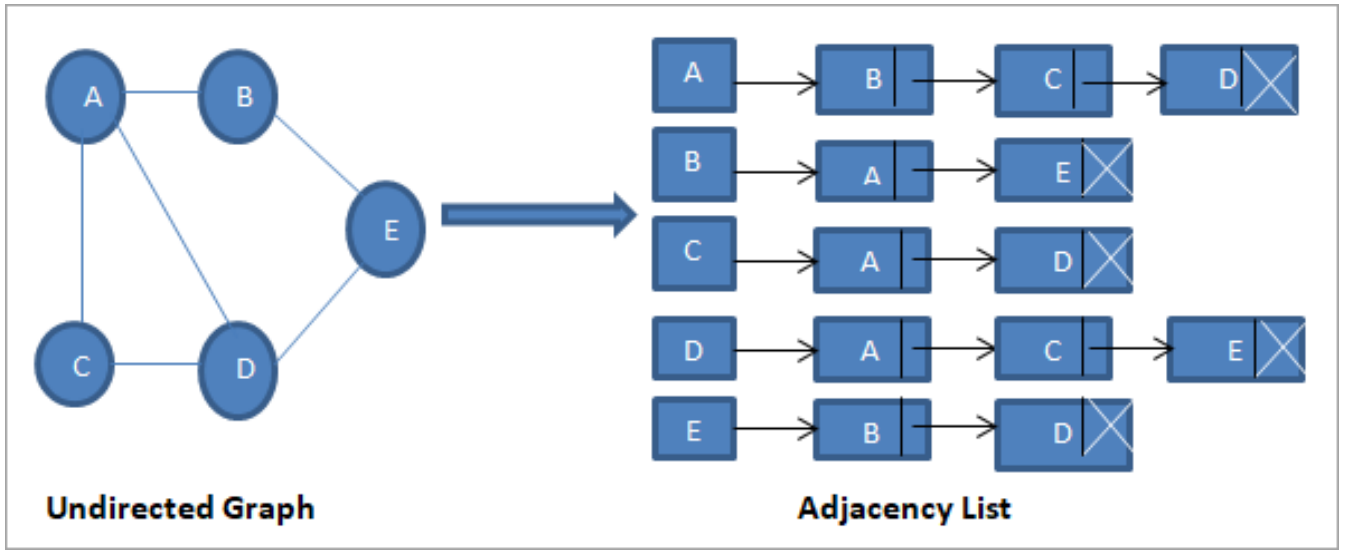
Directed Graphs

Unlike Facebook's friendship, Twitter's "following" relation only goes one way. If person A follows person B, that does not necessarily mean that person B follows person A. To represent their users, Twitter also uses graphs, but with a twist: the edges, which are now called arcs, are oriented. For example if Alice follows Bob, then there is an arc connecting the source node corresponding to Alice and the target node corresponding to Bob.



Graphs with code

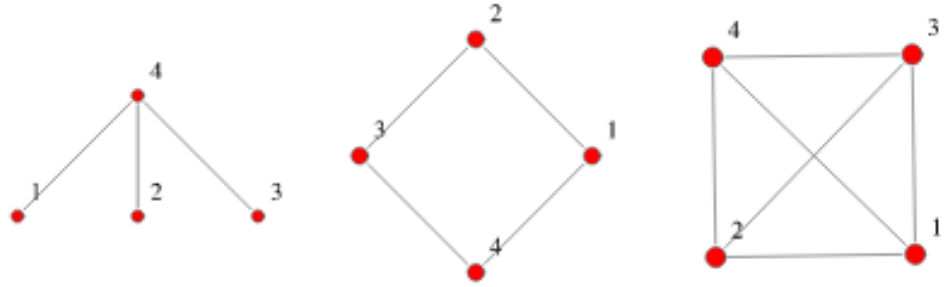
To represent a graph using the tools that a programming language is offering us , we can think that a each graph node , connects to other nodes(>=0).So basically , we need to map a value to an array of values. The best data structure to do that is a Hashmap.



Grids

Another way to represent graphs is using a grid. Let's assume we have a graph with n vertexes. Then , we create a nxn graph with boolean values , as 0 shows that we dont have an edge from i to j and 1 that he have an edge from i to j.

$$A[i,j] = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E \end{cases}$$



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Graph Traversals

DFS – BFS:

The **depth-first search**, or in short, **DFS**, traverses all the nodes that are reachable from a starting node. This algorithm is quite easy to understand and implement, making it a top option when trying to check if a graph is **connected** or not:

First it visits the starting node, which we call the **root**, and pushes it in a stack.

While the stack is not empty, the node at the top is examined and one of these two actions takes place:

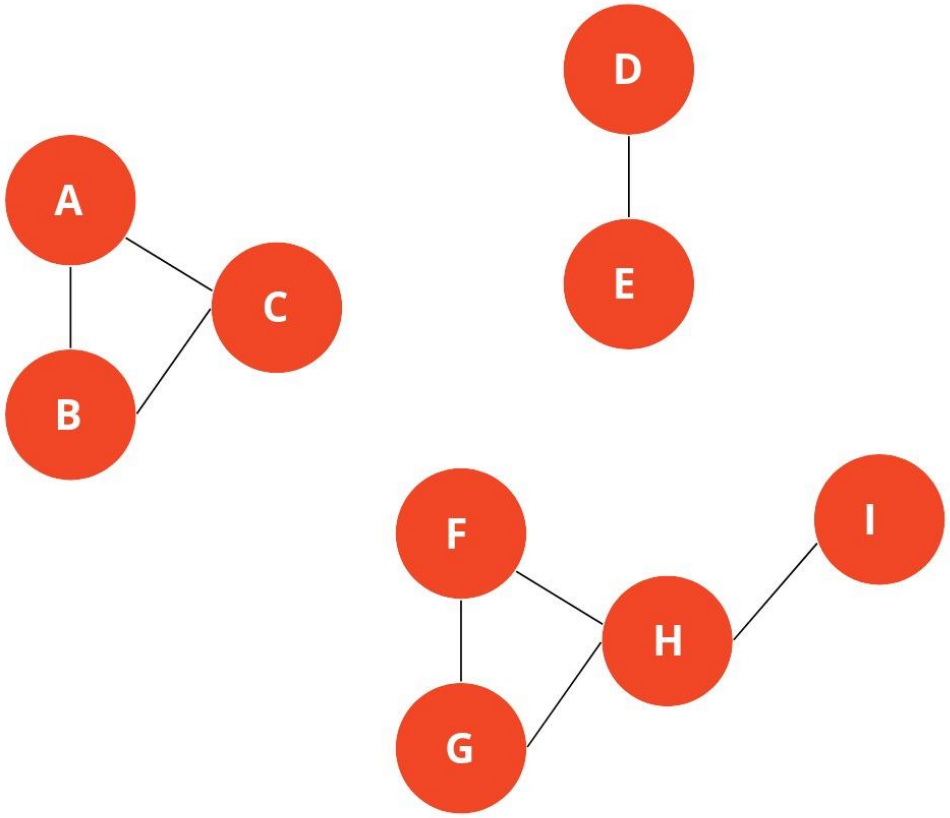
If the node has unvisited neighbours, one of them is chosen, visited and pushed in the stack.

Otherwise, if all the node's neighbours had previously been visited the node presents no further interest and it is popped from the stack.

When the stack becomes empty it means that all the accessible nodes have been visited, so the algorithm ends.

The **breadth-first search** algorithm, or in short, **BFS**, is one of the most common algorithms used to determine if a graph is **connected** or not. The steps of the BFS are the following:

1. Start by pushing the root into the queue.
2. While the queue is not emppty.
3. Pop the first node from the queue, visit all its neighbours and push into the queue those neighbours that have not been previously visited.
4. Repeat step 3.
5. Finally, when the queue becomes empty it means all the reachable nodes have been visited, so the algorithm finishes.
6. BFS-DFS visual(https://visualgo.net/en/dfsbfs)

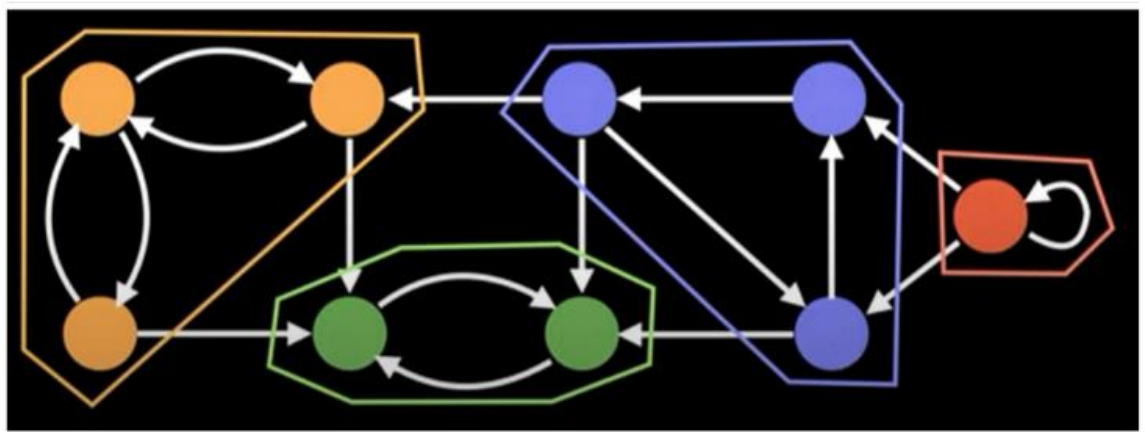


Connectivity check

We will use DFS to find how many connected components an undirected graph has.

1. We traverse all the nodes starting from A. If the node is unvisited , then we DFS.
2. When the operation stops , we increment our counter and we continue with the other nodes.

Tarjan's Algorithm for strongly connected components



Strongly connected components (SCC's) are self contained cycles within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle.

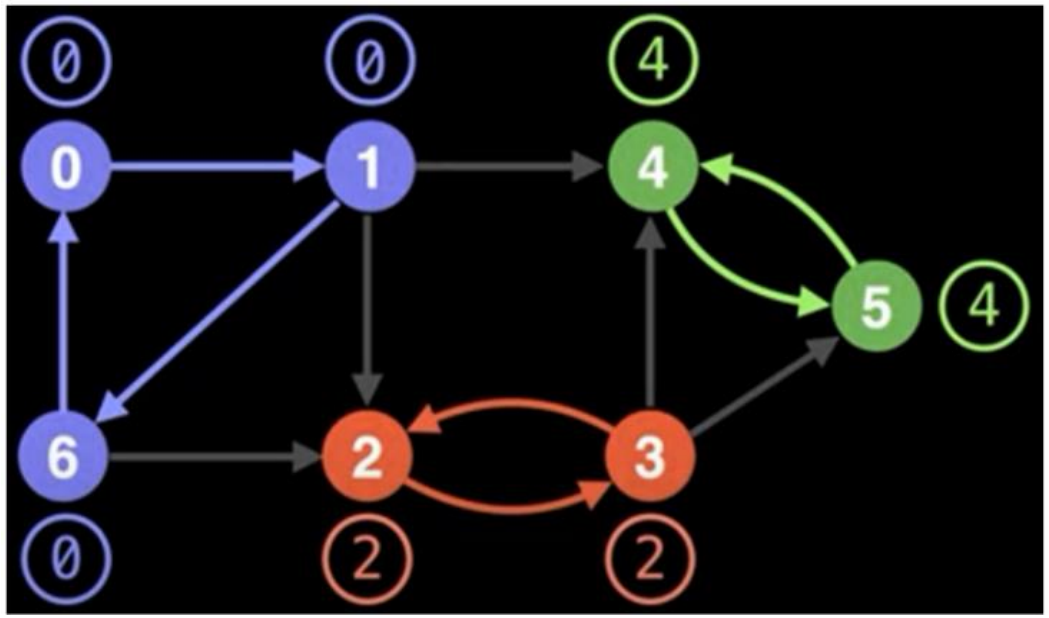
In this graph , there exists 4 SCC's.Note that

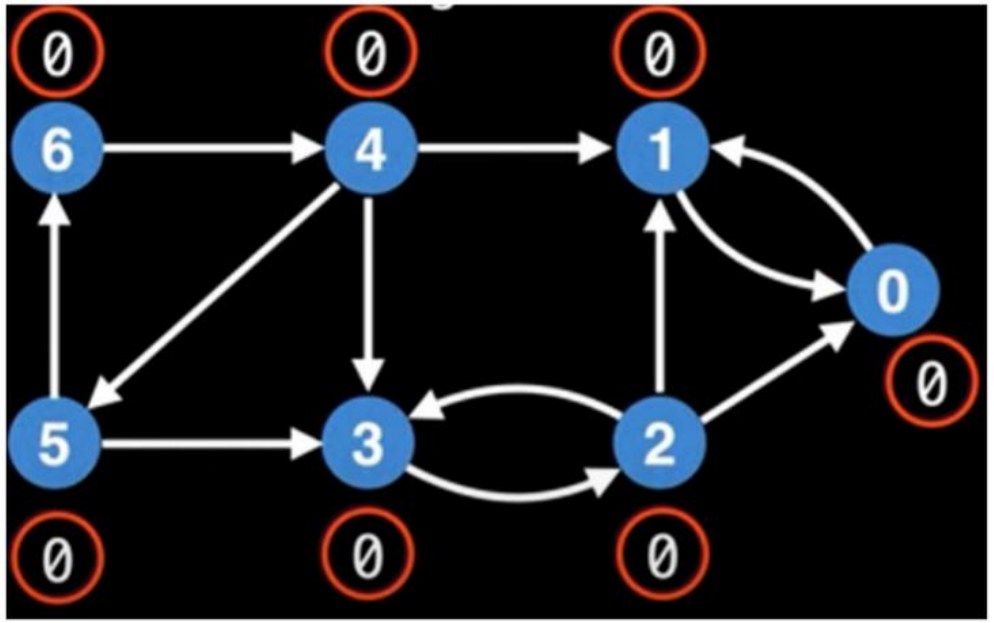
Every vertex can travel at every other vertex

In every SCC.

Low Link Values

The low-link value of a node is the smallest node ID reachable from that node when doing a depth-first search (DFS), including itself.There is a catch with doing a DFS on the graph, as it is highly dependent on the traversal order of the DFS, which is effectively random.





Kosaraju's algorithm for strongly connected components

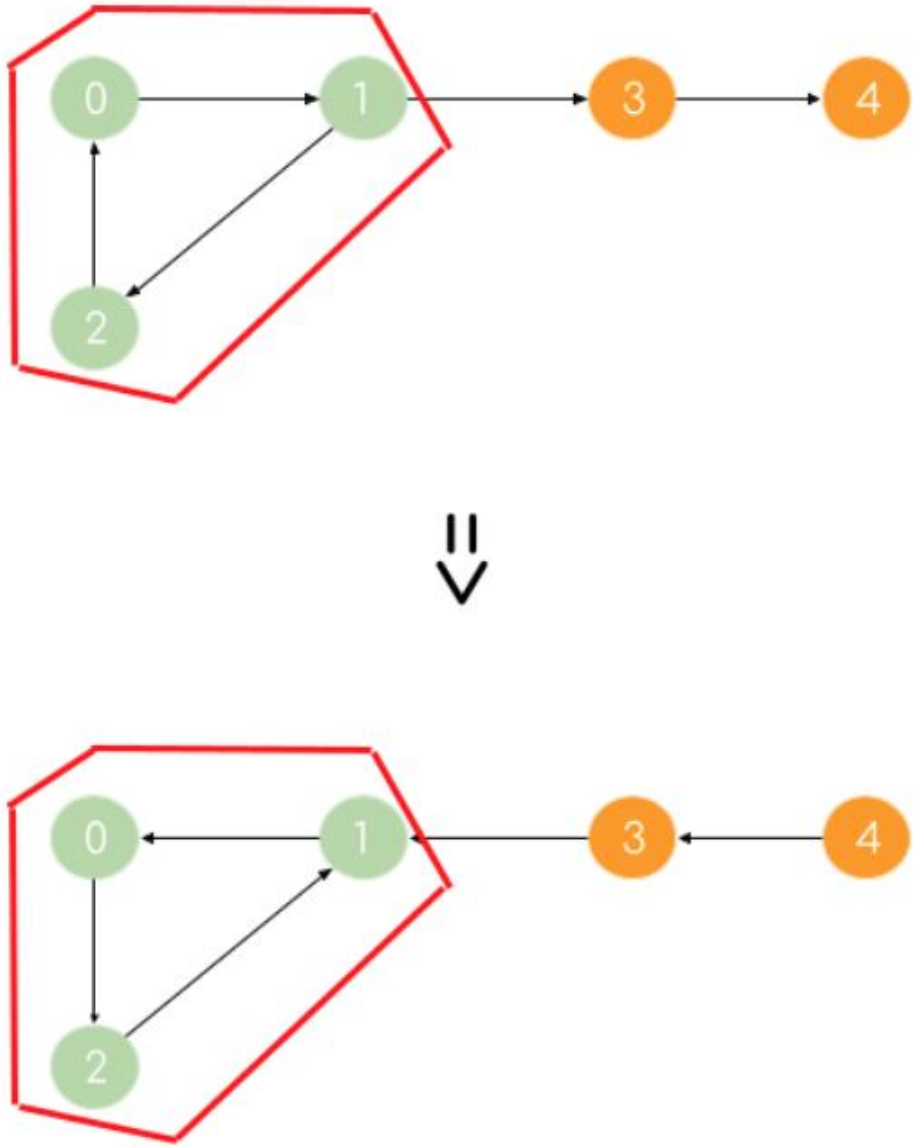
Idea: if we are in a SCC and a vertex u can travel to another vertex v in the SCC then , if we reverse all the edges of the graph , the vertex v can travel to the vertex u. The SCC still remain an SCC.

Algorithm:

1)Perform DFS traversal of the graph. Push node to stack before returning.

2)Find the transpose graph by reversing the edges.

3)Pop nodes one by one from the stack and again to DFS on the modified graph.



For the first DFS

DFS for the node 0 :

Visited -> {0 , 1 , 2}. Node 2 has no other unvisited neighbors , so we put it in the stack. Stack -> {2}.Back with recursion , we go to 1.

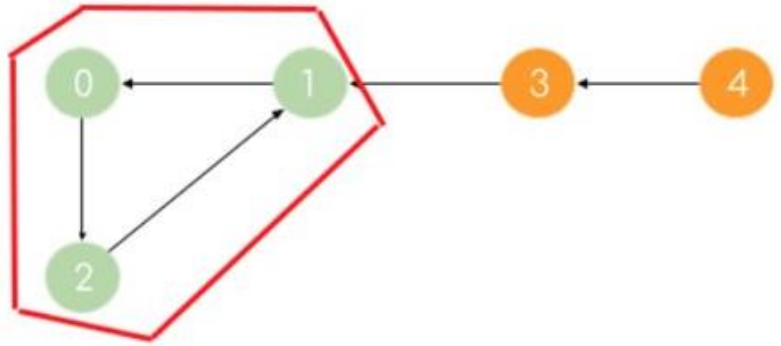
DFS for the node 1:

Visited -> {0 , 1 , 2 , 3 , 4}. 4 has no other unvisited neighbors , we add it to the stack. Stack -> {4,2}. Back with recursion , we go to 3. 3 has no other unvisited neighbors , we add it to the stack , so stack -> {3,4,2}.Back with recursion , we go to 1, 1 has no other unvisited neighbors , we add it to the stack , so stack -> {1,3,4,2}.Back with recursion , we go to 0, 0 has no other unvisitted neighbors , we add it to the stack , so stack -> {0,1,3,4,2}.

For the second DFS

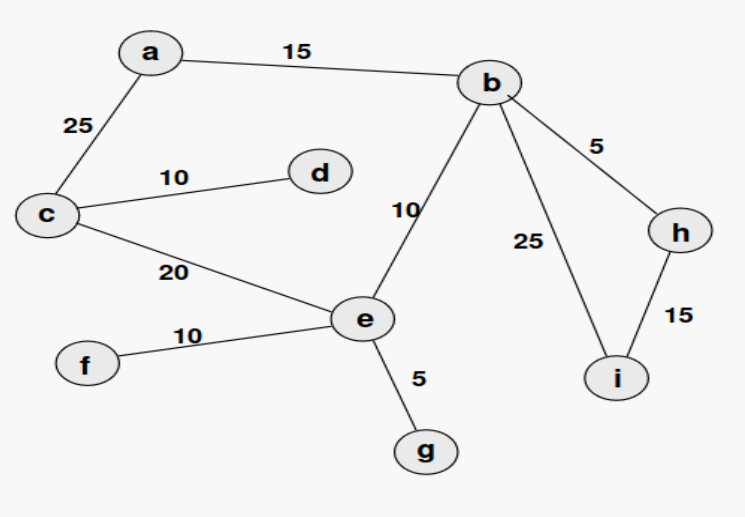
First , we reverse all the edges of the graph.

Stack is -> {0, 1, 3, 4, 2}.We dfs for the node 0 ,we pop 0 out of the stack, we visit {0,2,1}.Then , there are no more unvisited neighbors , so our first SCC is {0,2,1}.The next element in the stack is 1 , but , 1 is already visited so we pop and do nothing. The next element in the stack is 2 , but 2 is already visited, so we pop and do nothing. The next element is 3 , we dfs from the node 3 , there are no other neighbors , so the second SCC is {3} and we can easily see that the third SCC will be {4}.



Weighted Graphs

Weighted Graphs
In many applications, each edge of a graph has an associated numerical value, called a weight.
Usually, the edge weights are non-negative integers.
Weighted graphs may be either directed or undirected.



The weight of an edge is often referred to as the "cost" of the edge.
In applications, the weight may be a measure of the length of a route, the capacity of a line, the energy required to move between locations along a route, etc.

Finding shortest paths

Bellman Ford – Dijkstra

Dijkstra's algo has better time complexity , the only disadvantage is that it will not work for negative cycles , as if one exists then it will loop forever. Dijkstra's time complexity is O(|V|^2) but with a priority queue it goes down to O(|V| + |E|log|V|). Bellman Ford's time complexity is O(|V||E|) and there's no other implementation.

How Dijkstra works?

1. Initialize distances of all vertices as infinite.
2. Create an empty **p**riority_**q**ueue **q**. Every item of q is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs

---

3. Insert source vertex into q and make its distance as 0. That's because we need 0 distance to travel to our starting node , as we are already here.
4. While either q doesn't become empty
5. Extract minimum distance vertex from q. Let the extracted vertex be u.
6. Loop through all adjacent of u and do following for every vertex v

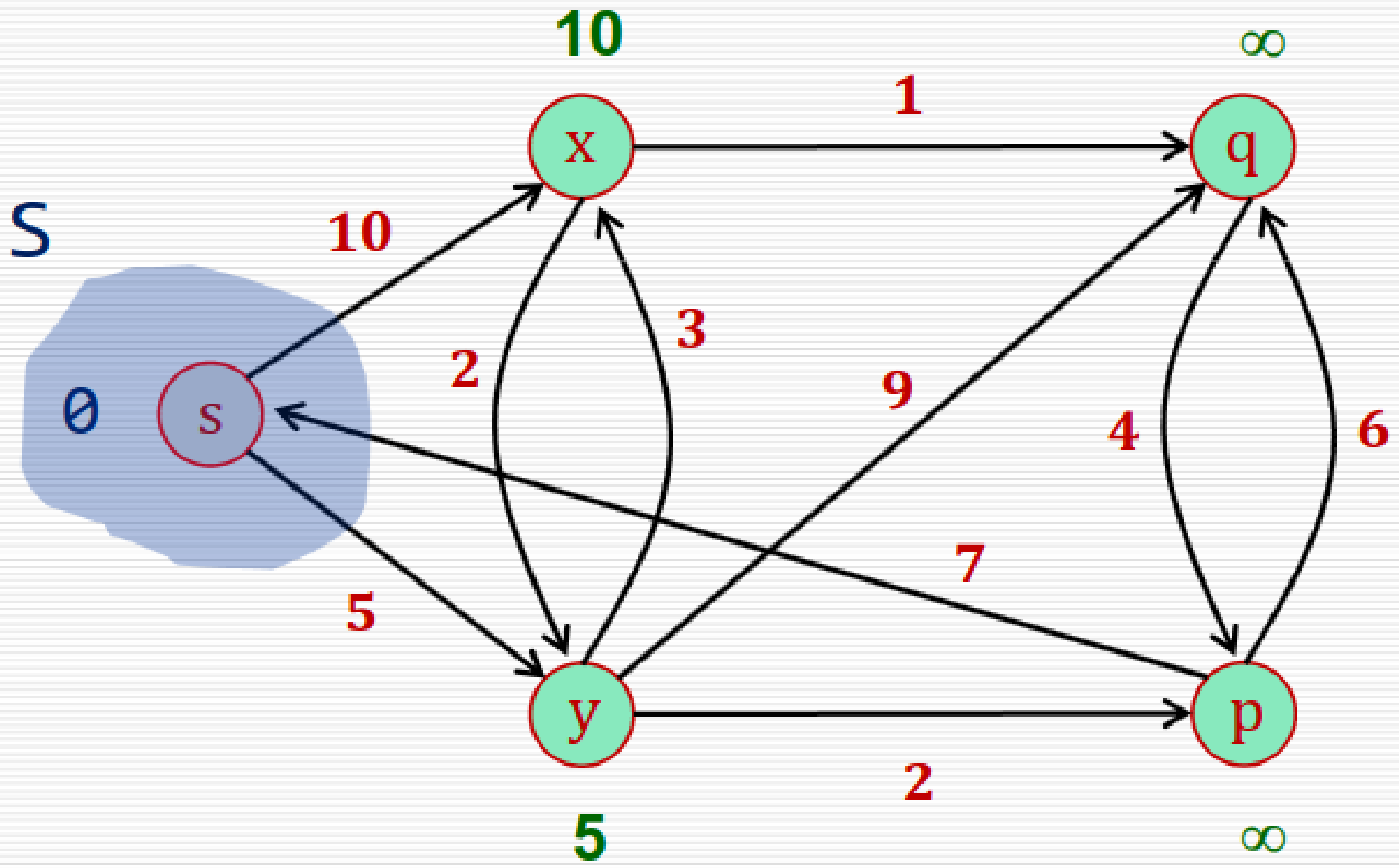
If dist[v] > dist[u] + weight(u, v)

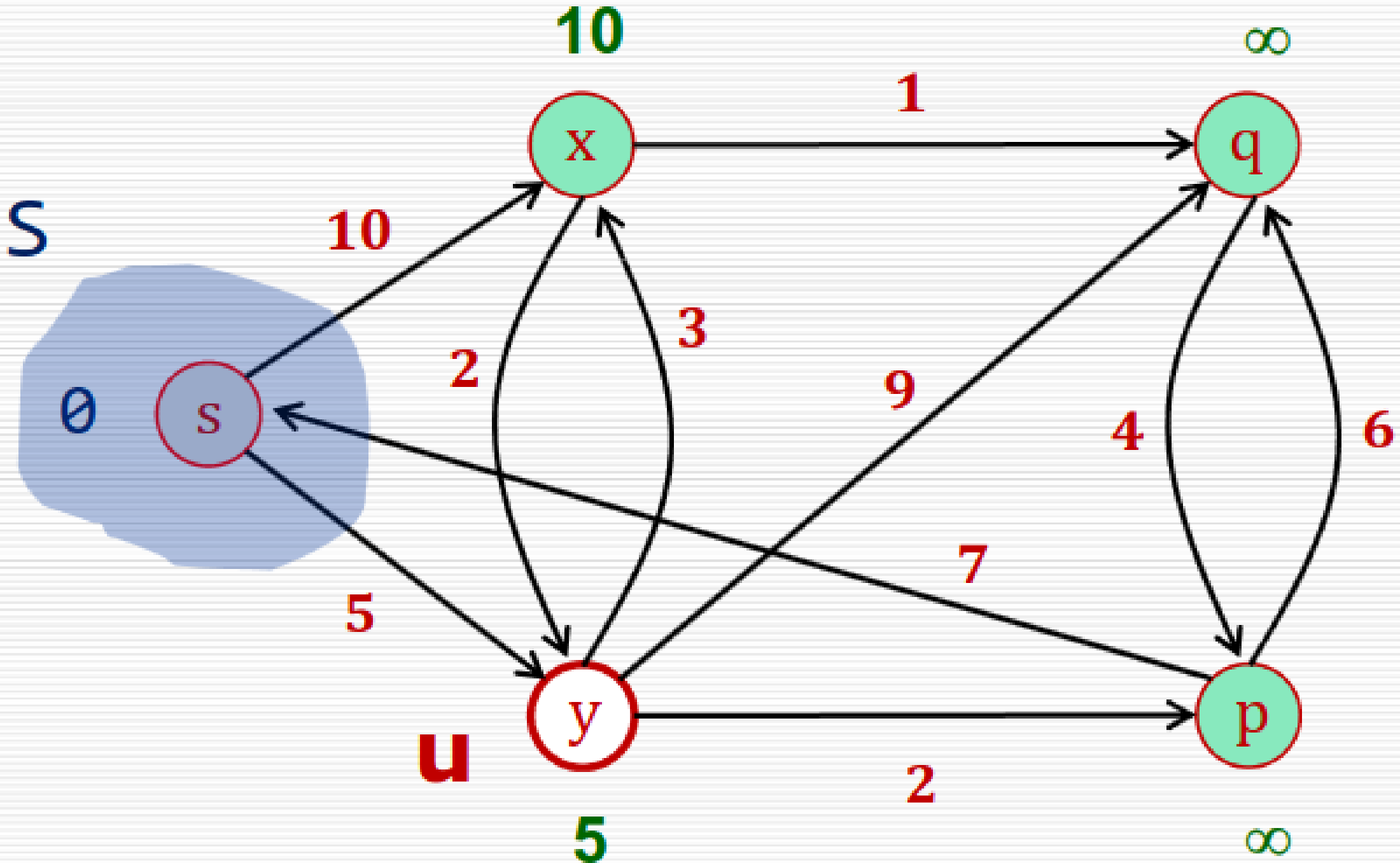
(i) Update distance of v,

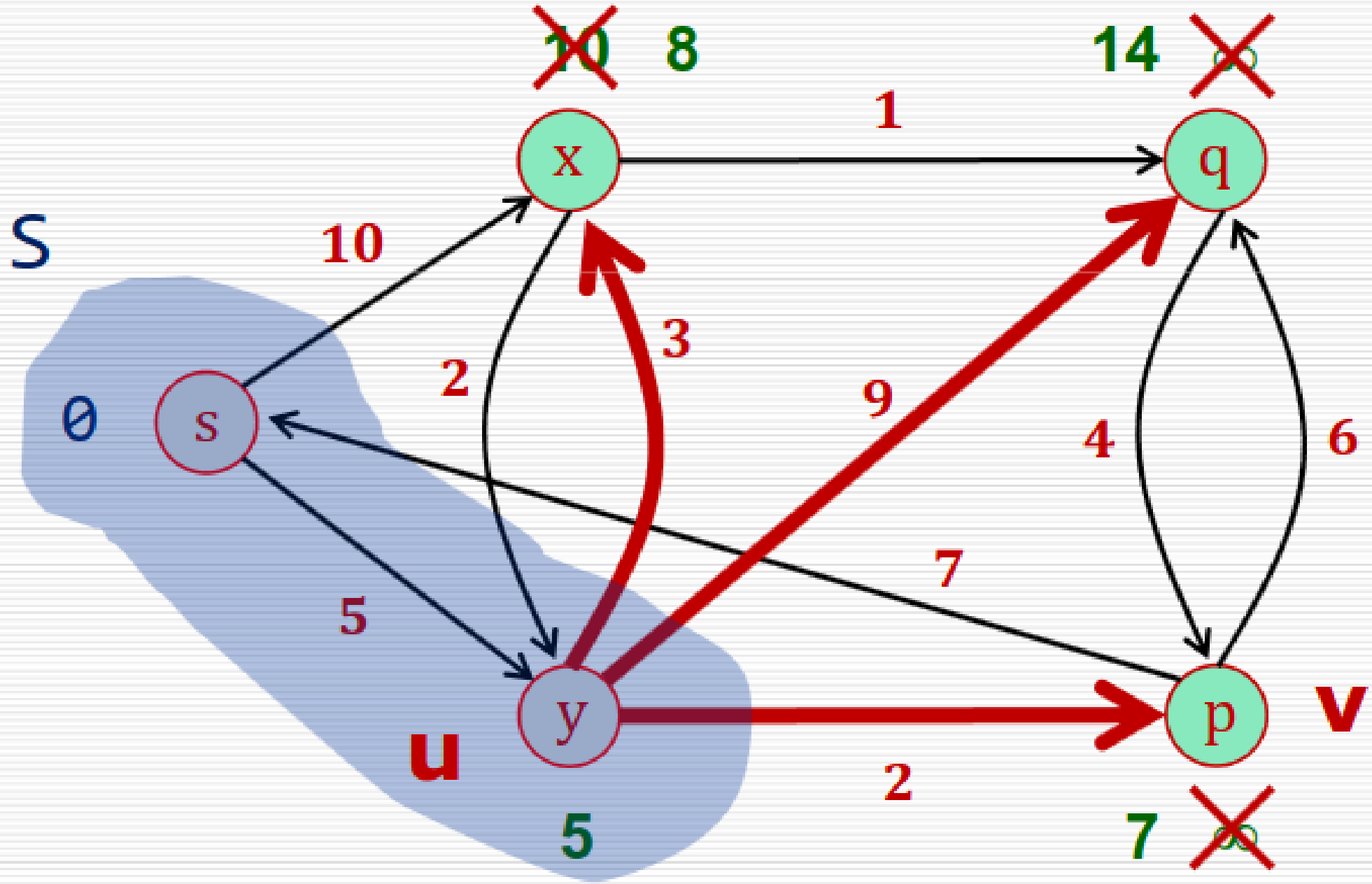
dist[v] = dist[u] + weight(u, v)

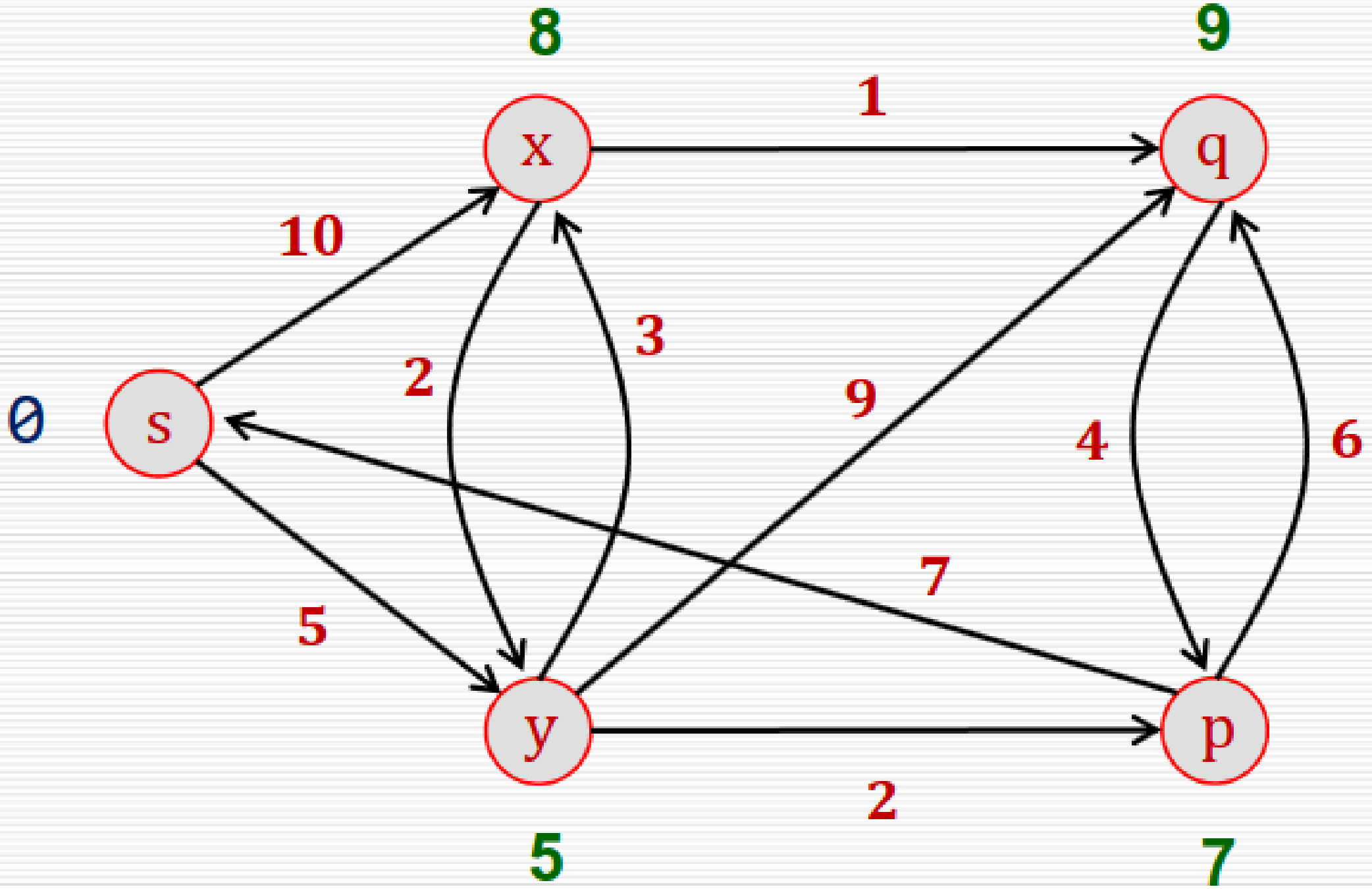
(ii) Insert v into the q (Even if v is

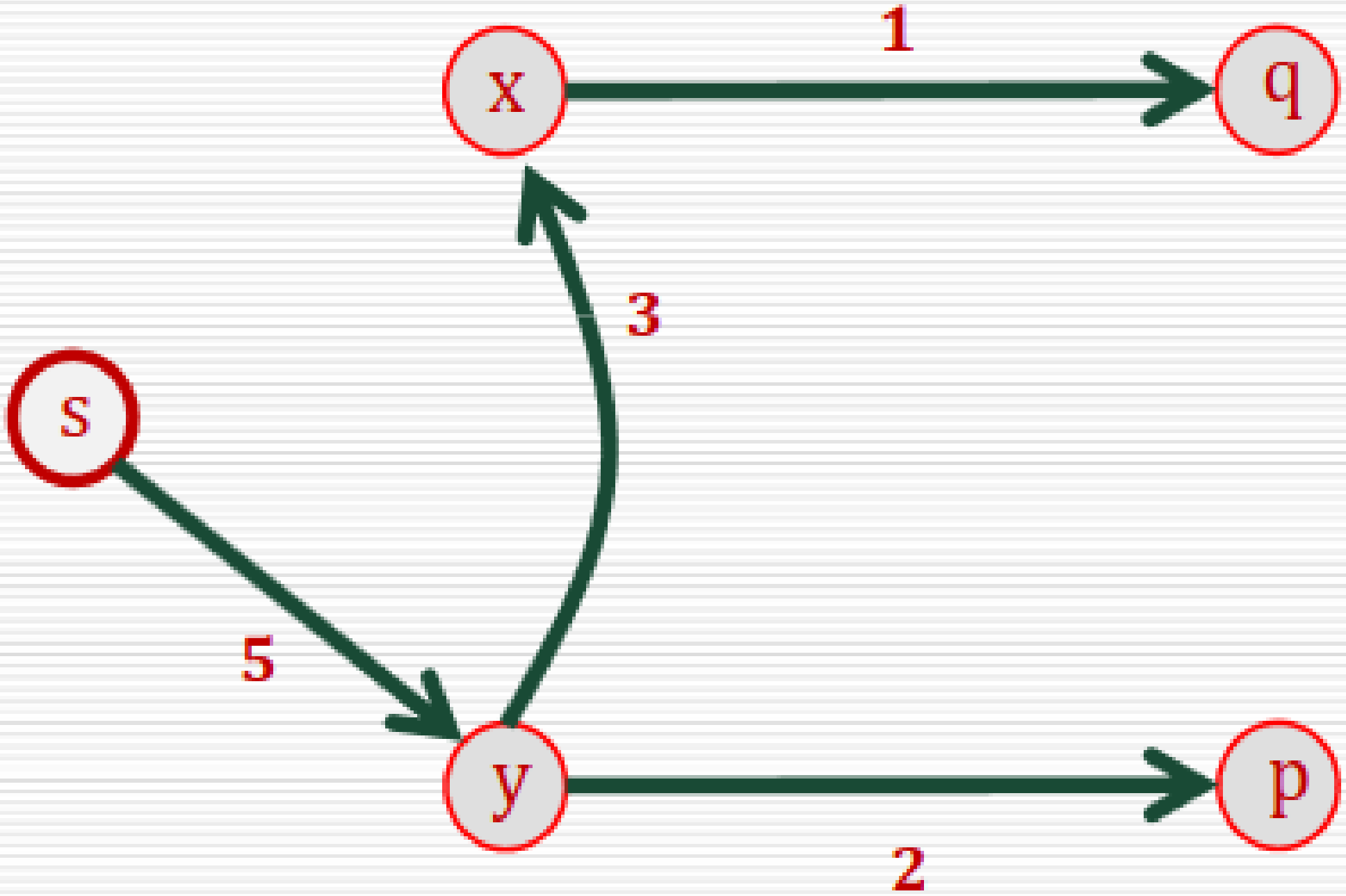
already there)


10
∞
1
x
q
S
10
2
3
9
4
6
0
s
7
5
y
p
2
5
∞


10
∞
1
x
q
S
10
3
2
9
4
6
0
s
7
5
y
p
u
2
5
∞


10
8
14
∞
1
x
q
S
10
3
2
9
0
s
4
6
7
5
y
p
v
u
2
5
7
∞


8
9
1
x
q
10
3
2
0
s
9
4
6
7
5
y
p
2
5
7


x
q
1
3
s
5
y
p
2

Minimum spanning trees

Prim-Kruskal :

Prim:

1. We pick any node from the graph.
2. For each neighbor , we pick the one with the less weight such that the new subgraph remains a tree.

Kruskal:

1. Each time we pick the edge with the less weight such that the new subgraph wont contain cycles.

Kruskal's time complexity: O(ElogV). Prim's time complexity: O(V^2) - O(ElogV) using fibonacci heaps , but only theoreticaly.

Kruskal's Algorithm

1. Sort(edges)
2. Select the edge with the minimum weight.
3. If the edge will not create a cycle in the tree , we add it , else we continue.

Have in mind that kruskal's algorithm prefers heap data structures(heap sort) and we don't care about connectivity as we create the MST starting by the least weighted edge.Applications of Kruskal algorithm are LAN connection, TV Network etc.

Union find for Kruskal's Algorithm

1. **Find:** Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
2. **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.

We use Union find in kruskal's algorithm to check if a cycle exists after inserting an edge.Let's assume we have this graph:

Edge 0-1 : different subsets , so union.After the operation 0 and 1 exists

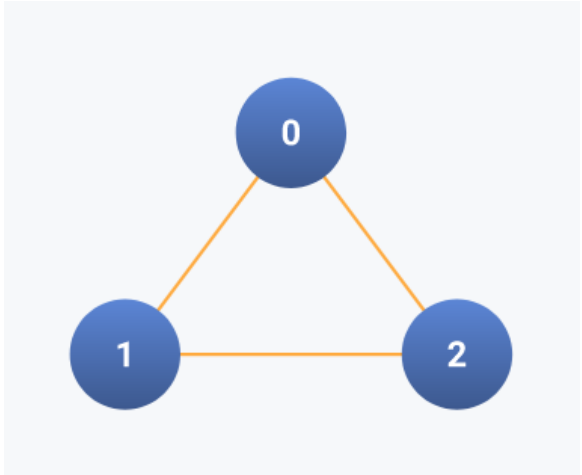
At the same subset so we have {0,1}

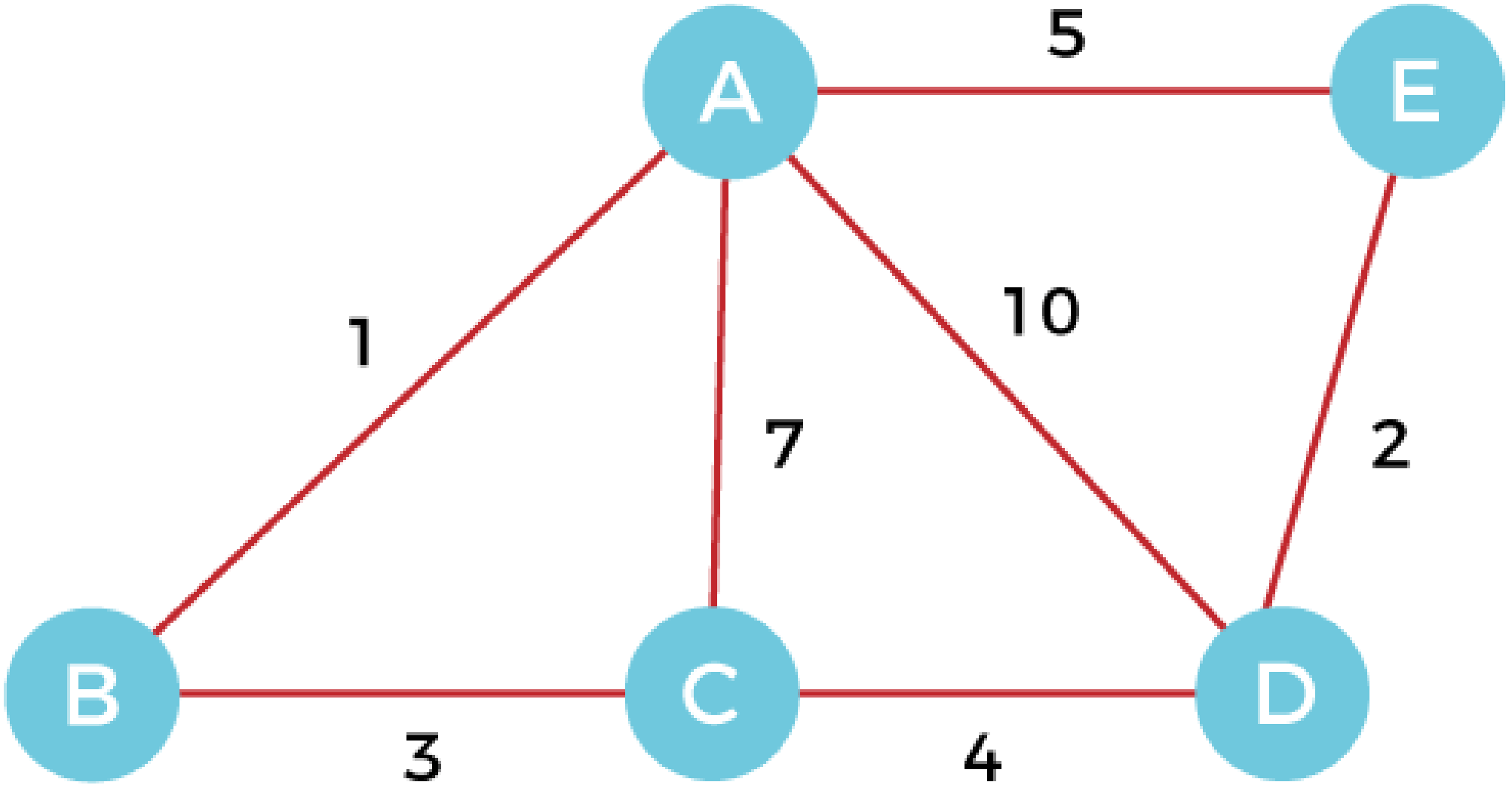
Edge 1 – 2: 1 exists at {0,1} , 2 exists at {2} so we union them and we'll

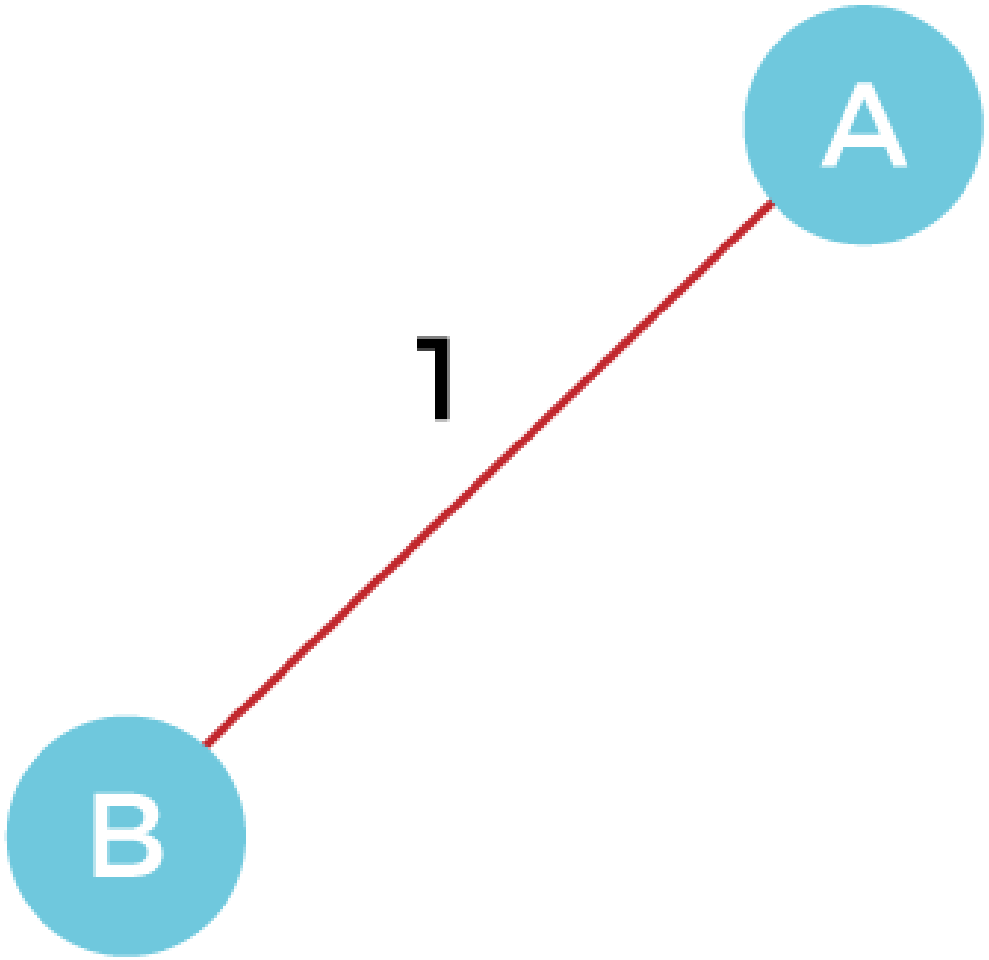
Have {0,1,2}

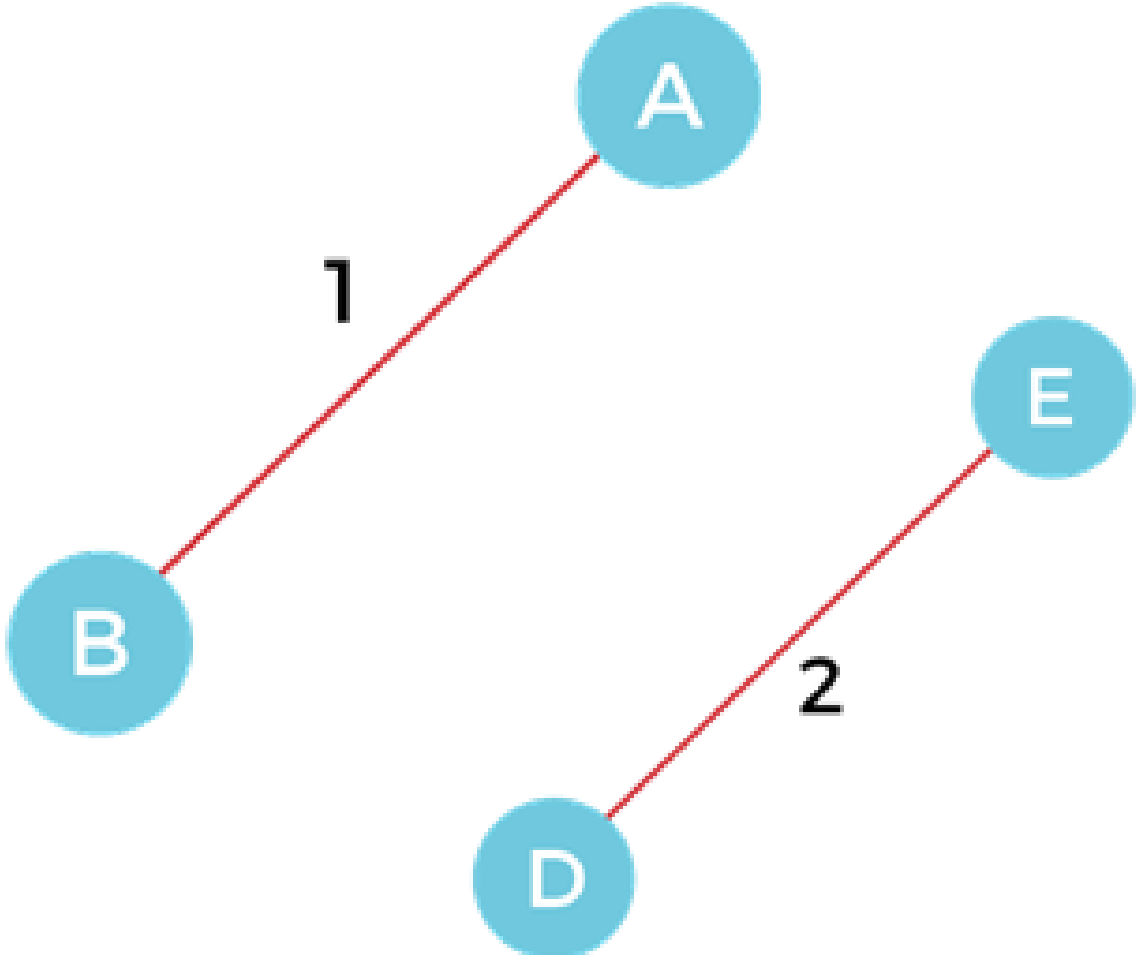
Edge 0-2: 0 exists at {0,1,2} , but , 2 exists at {0,1,2} as well , so we can't union

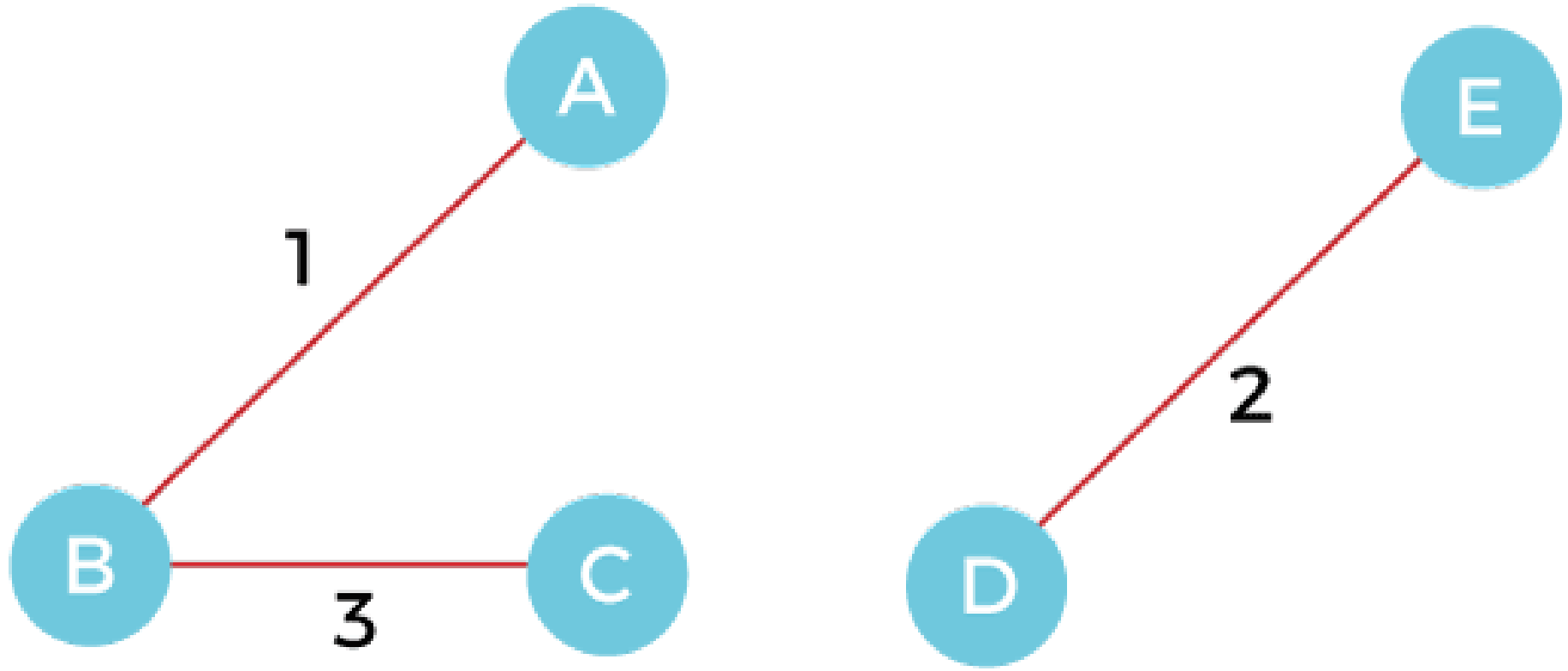
Them and so there exists a cycle in this graph.

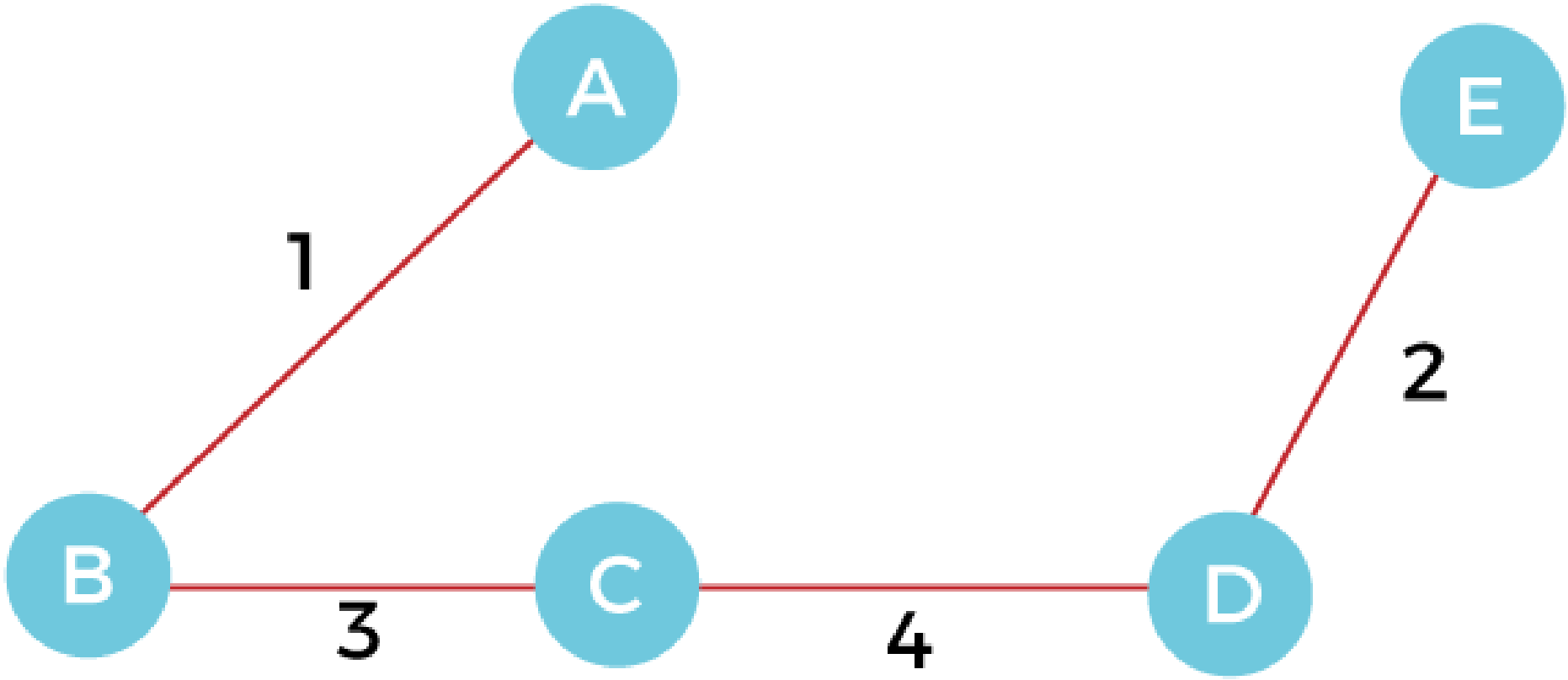



A
E
B
C
D
5
1
10
7
2
3
4


A
1
B


A
1
B
E
2
D


A
1
B
3
C
E
2
D

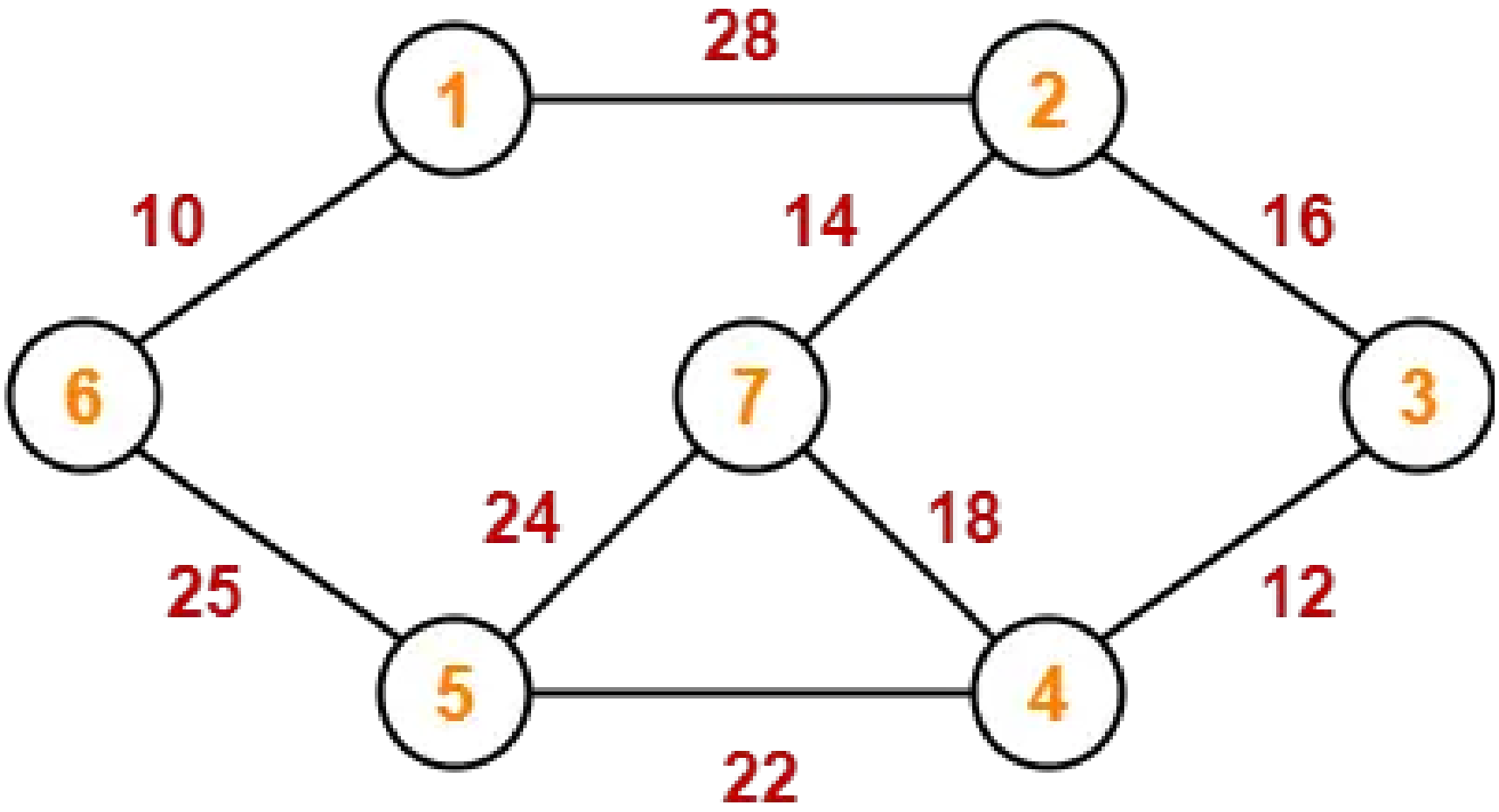

A
E
1
2
B
3
C
4
D

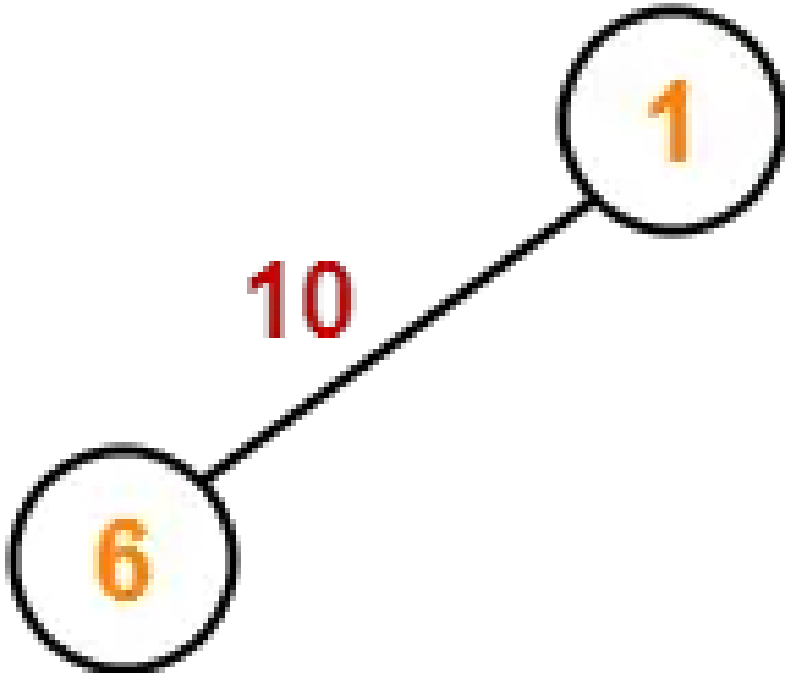
Prim's Algorithm

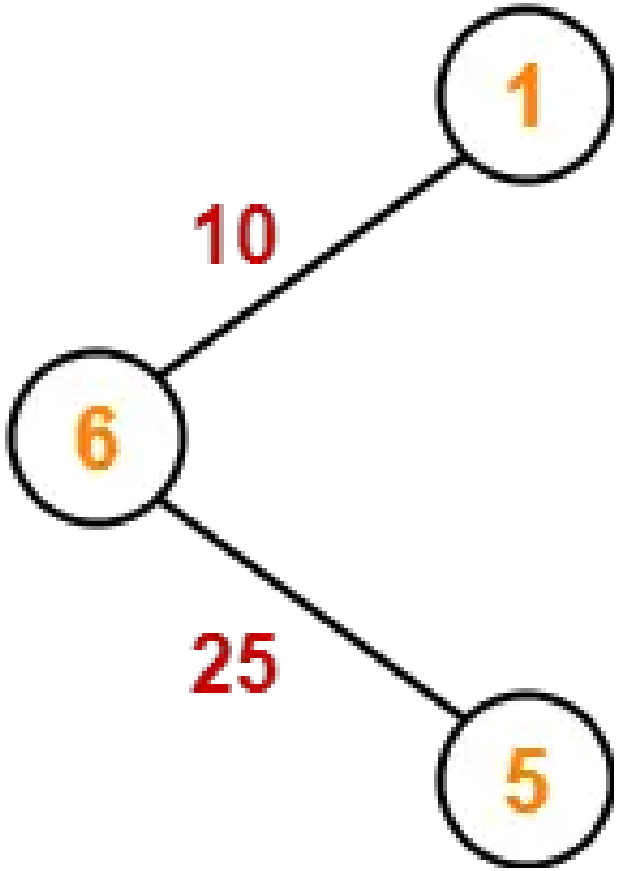
1. We always start from the starting node.
2. We then select(each time) the edge with the minimum weight from the so far constructed tree(remember we dont have any cycles)

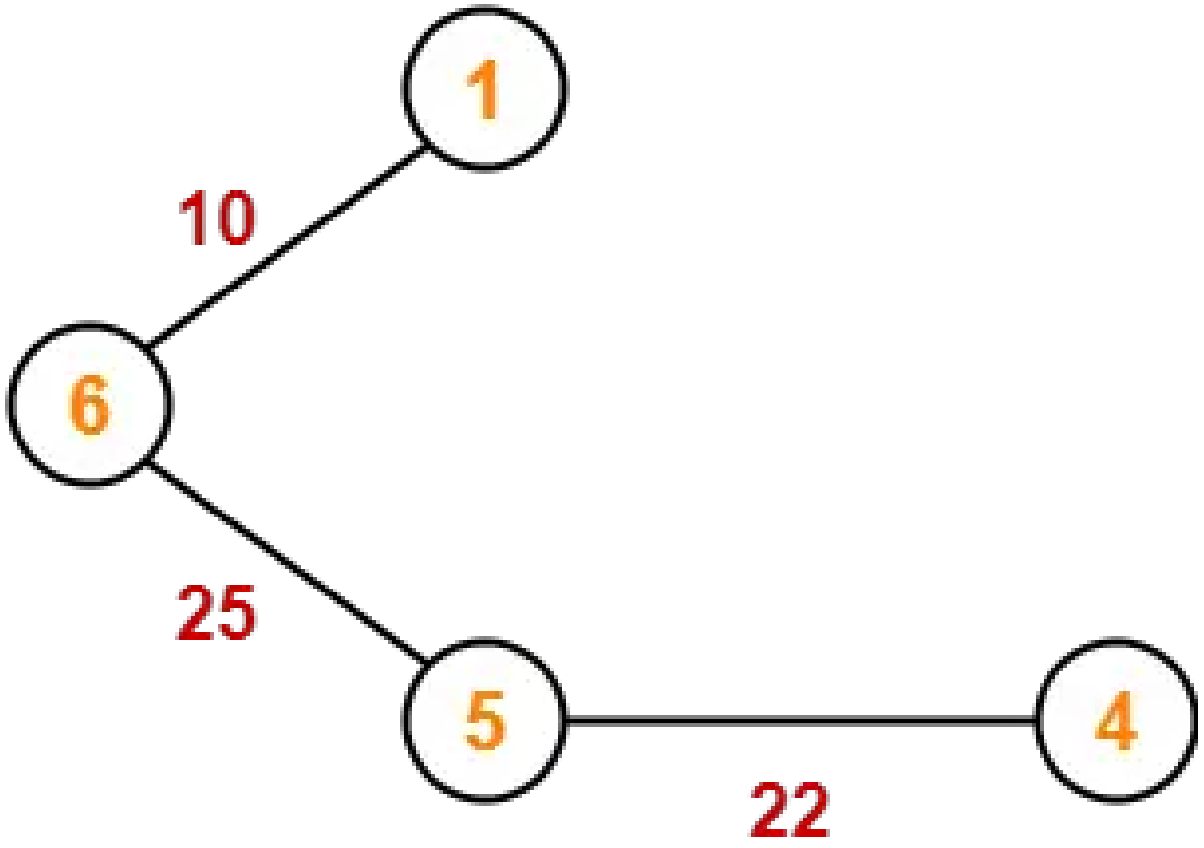
Applications of prim’s algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.Prim’s algorithm prefer list data structures.

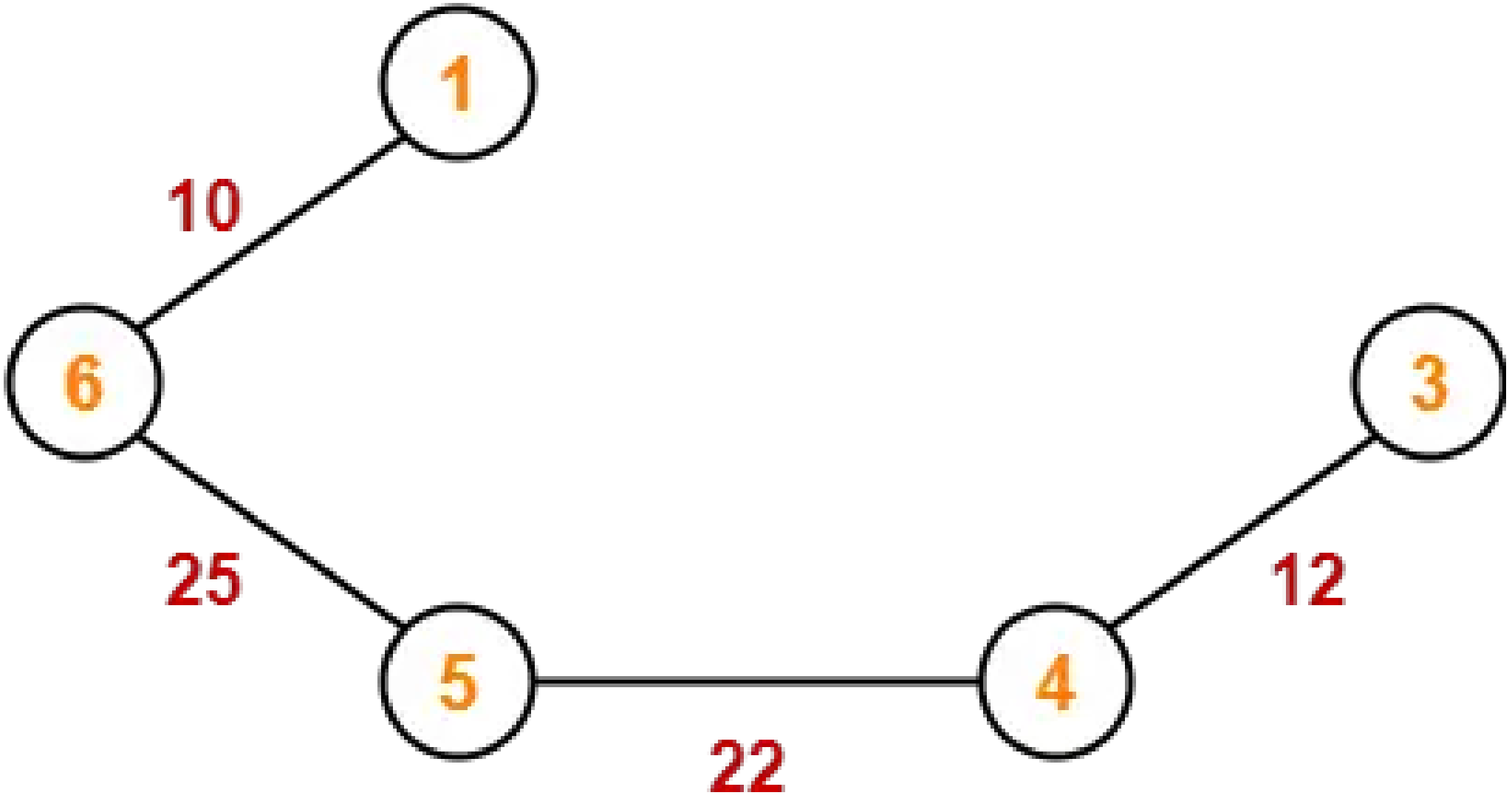
NOTE that both algorithms wont work in directed graphs.Why?

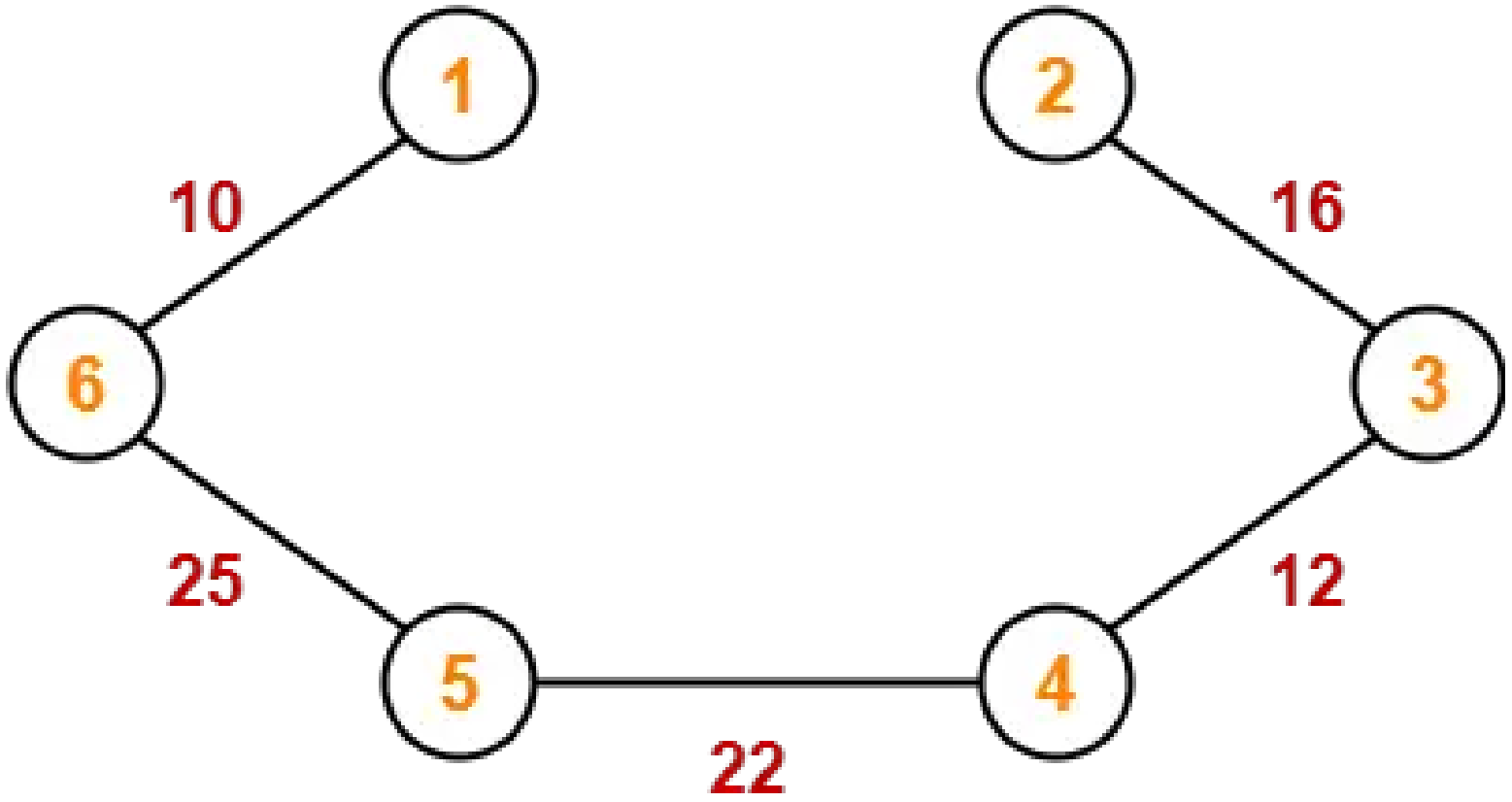

1
2
3
4
5
6
7
28
10
14
16
24
18
25
12
22

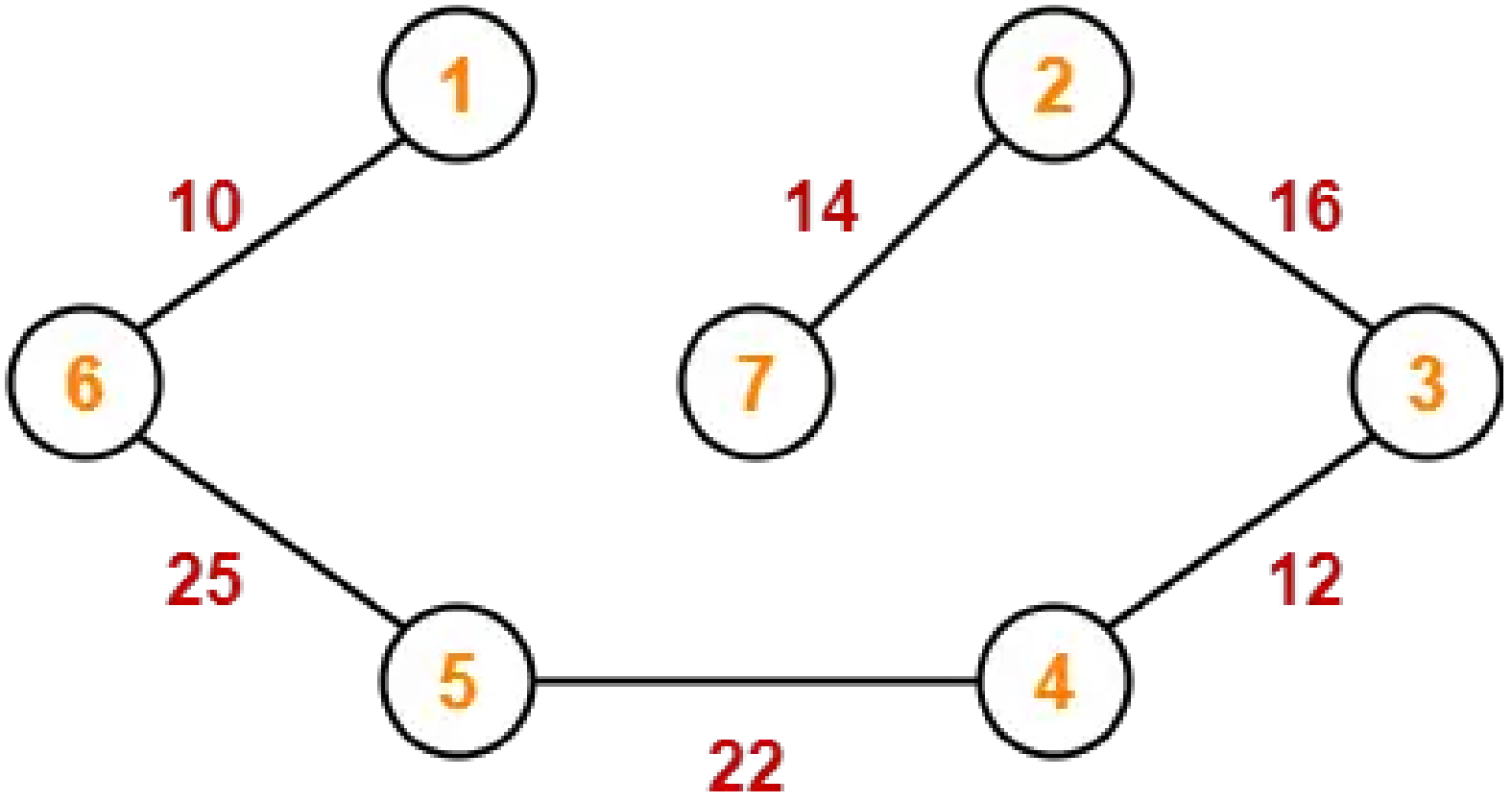

1
10
6


1
10
6
25
5

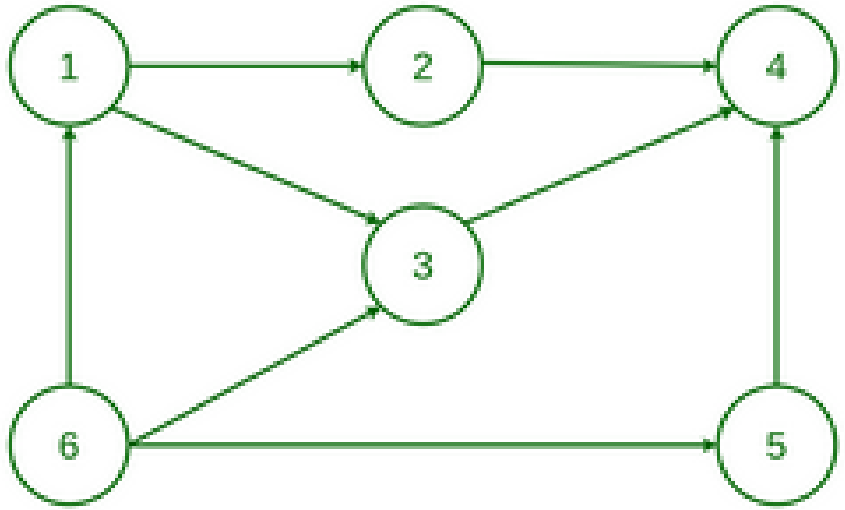

1
10
6
25
5
4
22


1
10
6
3
25
12
5
4
22


1
2
10
16
6
3
25
12
5
4
22

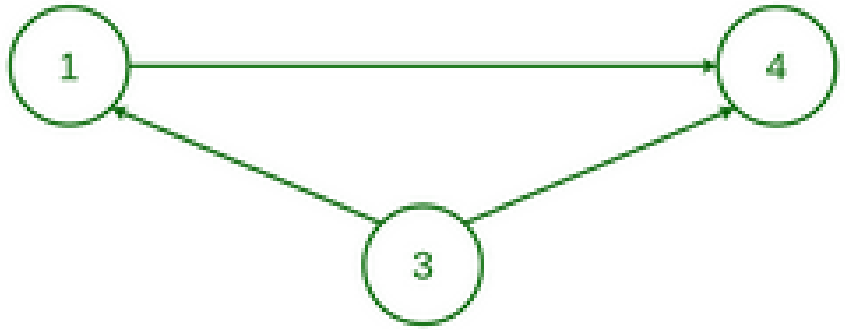

1
2
10
14
16
6
7
3
25
12
5
4
22

Why Prim's Algorithm fails for Directed Graph



As No node is reachable from node 4 and Prim's Algorithm assumes that every node is reachable due to which Prim's Algorithm fails for directed graph

Why Kruskal's Algorithm fails for Directed Graph



As There is no cycle in this directed graph but Kruskal's Algorithm assumes it a cycle by union-find method due to which Kruskal's Algorithm fails for directed graph