Lesson:

Graphs in Java-2

Pre Requisites:

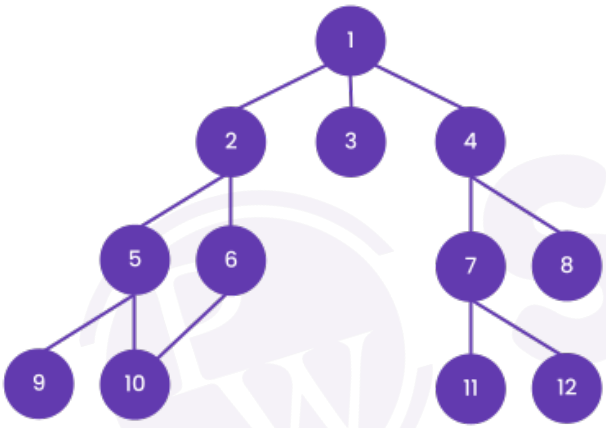
- Trees in Java
- Graphs in Java-1

Topics to be covered

- Cycle detection in a graph
- Dijkstra's algorithm
- Minimum Spanning tree
- Kruskal Algorithm
- Prim's algorithm

Cycle detection in a graph

A cycle in a graph means we start from a node and we reach it back again then it's a cycle.
For example, the following graph contains a cycle 2–5–10–6–2:



Cycle can exist in both undirected and directed graphs.

We will be using BFS to detect cycle in graph.

Approach:
When we do a Breadth–first search (BFS) from any vertex v in an undirected graph, we may encounter a cross-edge that points to a previously discovered vertex that is neither an ancestor nor a descendant of the current vertex.
Each "cross edge" defines a cycle in an undirected graph. If the cross edge is x —> y, then since y is already discovered, we have a path from v to y (or from y to v since the graph is undirected), where v is the starting vertex of BFS.
So, we can say that we have a path v ~~ x ~ y ~~ v that forms a cycle. (Here, ~~ represents one more edge in the path, and ~ represents a direct edge).

LP_Code1.java

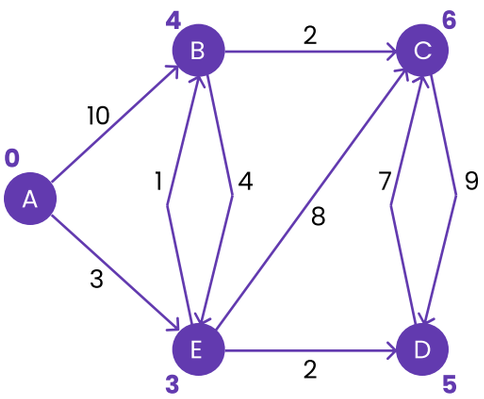
Output:

```
The graph contains a cycle
```

Dijkstra's Algorithm

This algorithm helps us to find the shortest cost/weight path in a directed graph from one source node to all the other nodes.
Given a source vertex s from a set of vertices V in a weighted digraph where all its edge weights w(u, v) are non-negative, find the shortest path weights d(s, v) from source s for all vertices v present in the graph.

for eg in the below graph



Vertex	Minimum Cost	Route
A —> B	4	A —> E —> B
A —> C	6	A —> E —> B —> C
A —> D	5	A —> E —> D
A —> E	3	A —> E

We know that the Breadth–first search (BFS) can be used to find the shortest path in an unweighted graph or even in a weighted graph having the same cost of all its edges. But if edges in the graph are weighted with different costs, then BFS generalizes to uniform-cost search. Instead of expanding nodes to their depth from the root, uniform-cost search expands the nodes in order of their cost from the root. A variant of this algorithm is known as Dijkstra's algorithm.

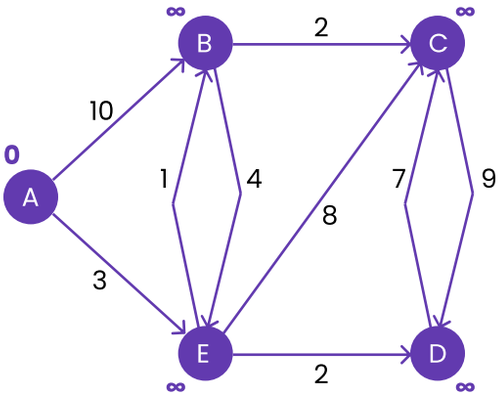
Dijkstra's Algorithm is an algorithm for finding the shortest paths between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other node. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the fastest route to the destination node has been determined.

Dijkstra's Algorithm is based on the principle of relaxation, in which more accurate values gradually replace an approximation to the correct distance until the shortest distance is reached. The approximate distance to each vertex is always an overestimate of the true distance and is replaced by the minimum of its old value with the length of a newly found path. It uses a priority queue to greedily select the closest vertex that has not yet been processed and performs this relaxation process on all of its outgoing edges.

Following is the pseudo code for dijkstra implementation

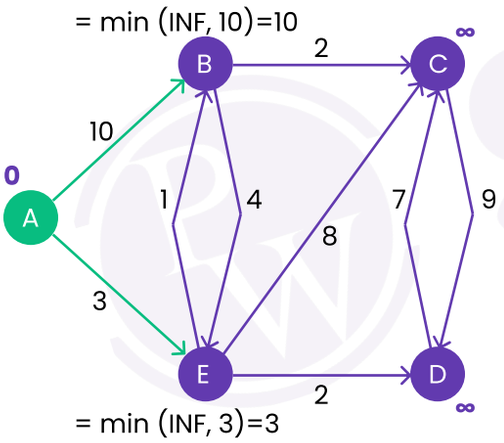
```
function Dijkstra(Graph, source)

    dist[source] = 0    // Initialization
    create vertex set Q

    for each vertex v in Graph
    {
        if v != source
        {
            dist[v] = INFINITY        // Unknown distance from source to v
            prev[v] = UNDEFINED       // Predecessor of v
        }
        Q.add_with_priority(v, dist[v])
    }
    while Q is not empty
    {
        u = Q.extract_min()           // Remove minimum
        for each neighbor v of u that is still in Q
        {
            alt = dist[u] + length(u, v)
            if alt < dist[v]
            {
                dist[v] = alt
                prev[v] = u
                Q.decrease_priority(v, alt)
            }
        }
    }
    return dist[], prev[]
```

Let's take the above example to show how dijkstra's work
We will start with vertex A, So vertex A has a distance 0, and the remaining vertices have an undefined (infinite) distance from the source. Let S be the set of vertices whose shortest path distances from the source are already calculated.



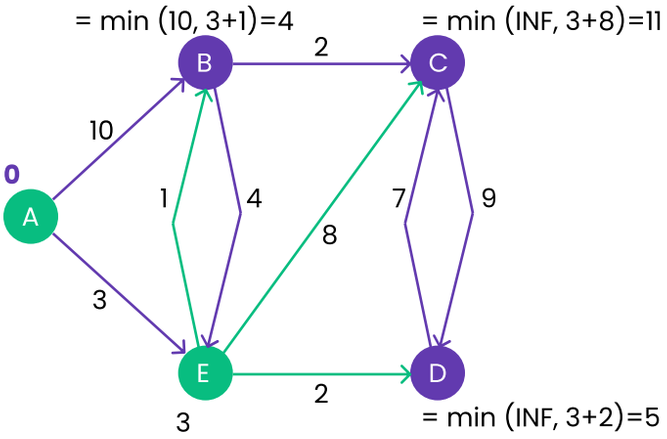
S = {}

Initially, S contains the source vertex. S = {A}.
We start from source vertex A and start relaxing A's neighbors. Since vertex B can be reached from a direct edge from vertex A, update its distance to 10 (weight of edge A–B). Similarly, we can reach vertex E through a direct edge from A, so we update its distance from INFINITY to 3.



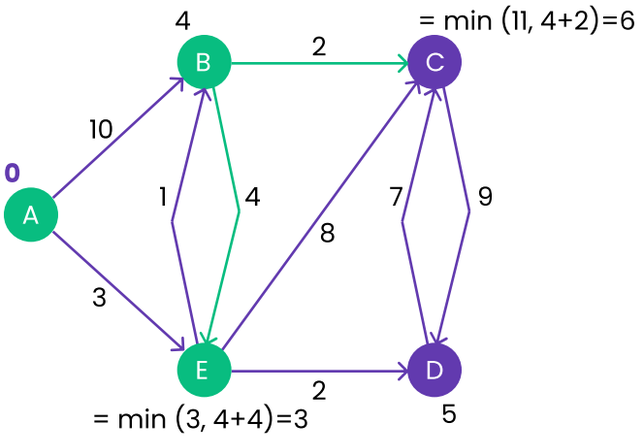
S = {A}

After processing all outgoing edges of A, we next consider a vertex having minimum distance. B has a distance of 10, E has distance 3, and all remaining vertices have distance INFINITY. So, we choose E and push it into set S. Now our set becomes S = {A, E}. Next, we relax with E's neighbors. E has 2 neighbors B and C. We have already found one route to vertex B through vertex A having cost 10. But if we visit a vertex B through vertex E, we are getting an even cheaper route, i.e., (cost of edge A–E + cost of edge E–B) = 3 + 1 = 4 < 10 (cost of edge A–B).

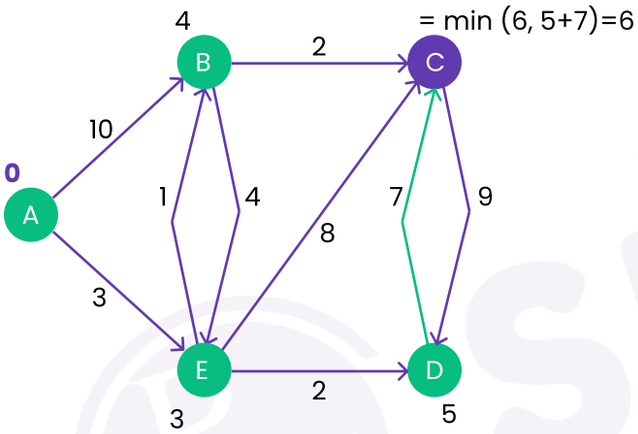


S = {A,E}

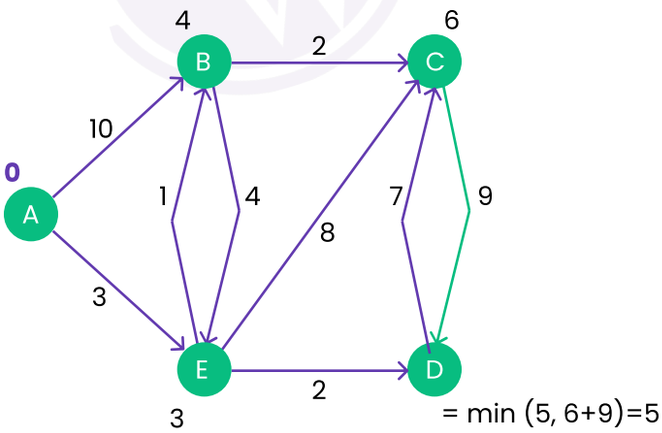
We repeat the process till we have processed all the vertices, i.e., Set S becomes full.



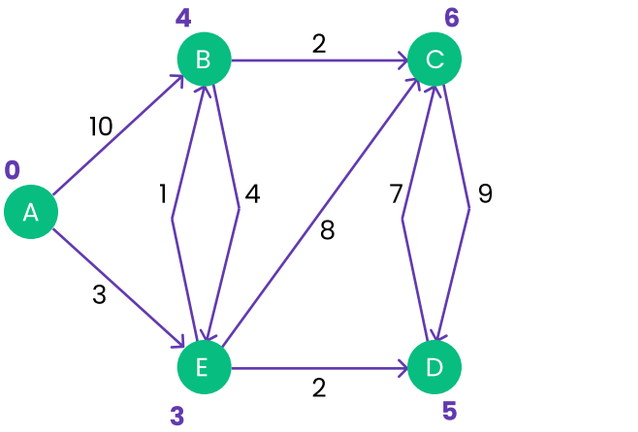
S = {A,E,B}



S = {A,E,B,D}



S = {A,E,B,D,C}



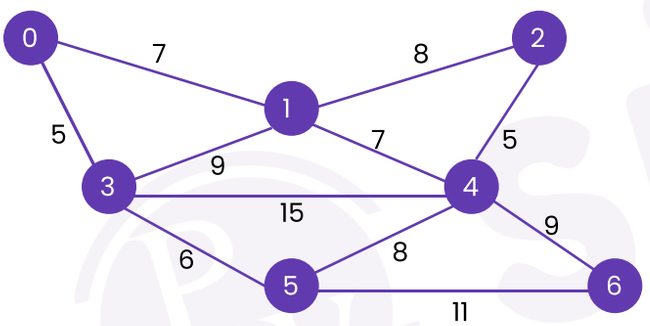
LP_Code2.java

Output:

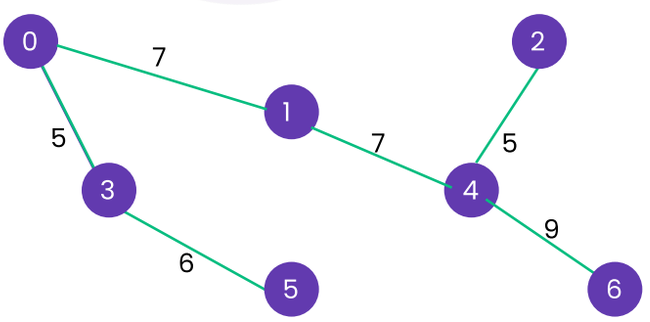
```
Path (0 -> 1): Minimum cost = 4, Route = [0, 4, 1]
Path (0 -> 2): Minimum cost = 6, Route = [0, 4, 1, 2]
Path (0 -> 3): Minimum cost = 5, Route = [0, 4, 3]
Path (0 -> 4): Minimum cost = 3, Route = [0, 4]
Path (1 -> 2): Minimum cost = 2, Route = [1, 2]
Path (1 -> 3): Minimum cost = 6, Route = [1, 4, 3]
Path (1 -> 4): Minimum cost = 4, Route = [1, 4]
Path (2 -> 3): Minimum cost = 9, Route = [2, 3]
Path (3 -> 2): Minimum cost = 7, Route = [3, 2]
Path (4 -> 1): Minimum cost = 1, Route = [4, 1]
Path (4 -> 2): Minimum cost = 3, Route = [4, 1, 2]
Path (4 -> 3): Minimum cost = 2, Route = [4, 3]
```

Minimum Spanning Tree

A Minimum Spanning Tree is a spanning tree of a connected, undirected graph. It connects all the vertices with minimal total weighting for its edges.



For example, consider the above graph. Its minimum spanning tree will be the following tree with exactly n-1 edges where n is the total number of vertices in the graph, and the sum of weights of edges is as minimum as possible:



Kruskal's Algorithm

We can use Kruskal's Minimum Spanning Tree algorithm, a greedy algorithm to find a minimum spanning tree for a connected weighted graph. Kruskal's Algorithm works by finding a subset of the edges from the given graph covering every vertex present in the graph such that they form a tree (called MST), and the sum of weights of edges is as minimum as possible.
Let G = (V, E) be the given graph. Initially, our MST contains only vertices of the given graph with no edges. In other words, initially, MST has V connected components, with each vertex acting as one connected component. The goal is to add minimum weight edges to our MST such that we are left with a single connected component that comprises all the graph's vertices. Following is the complete algorithm:

sort all edges in graph G in order of their increasing weights;
Repeat till we add V-1 edge successfully // as MST contains V-1 edges

```
    select the next edge with minimum weight from graph G;

    if (no cycle is formed by adding the edge in MST, i.e., the edge connects two
          different connected components in MST)
       add the edge to MST;
}
```

LP_Code3.java

Output:

```
[(0, 3, 5), (2, 4, 5), (3, 5, 6), (0, 1, 7), (1, 4, 7), (4, 6, 9)]
```

The time complexity of the above solution is O(n2), where n is the total number of vertices in the graph.

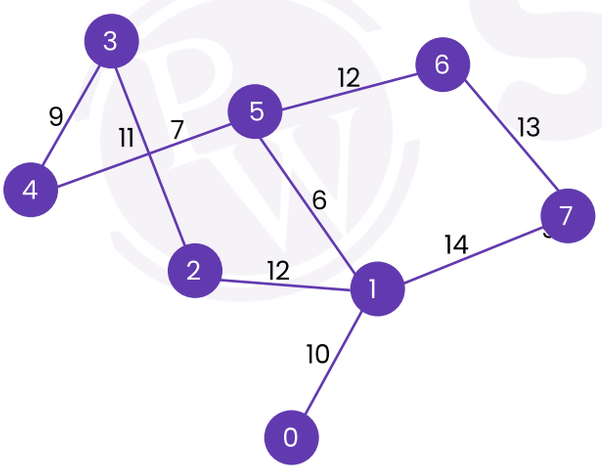
PRIM's Algorithm

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.
The steps for implementing Prim's algorithm are as follows:

- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree

For the input graph

LP_Code4.java



Output:

```
0 - 1
3 - 2
4 - 3
5 - 4
1 - 5
5 - 6
1 - 7
```

Next Class Teasers

- Dynamic Programming