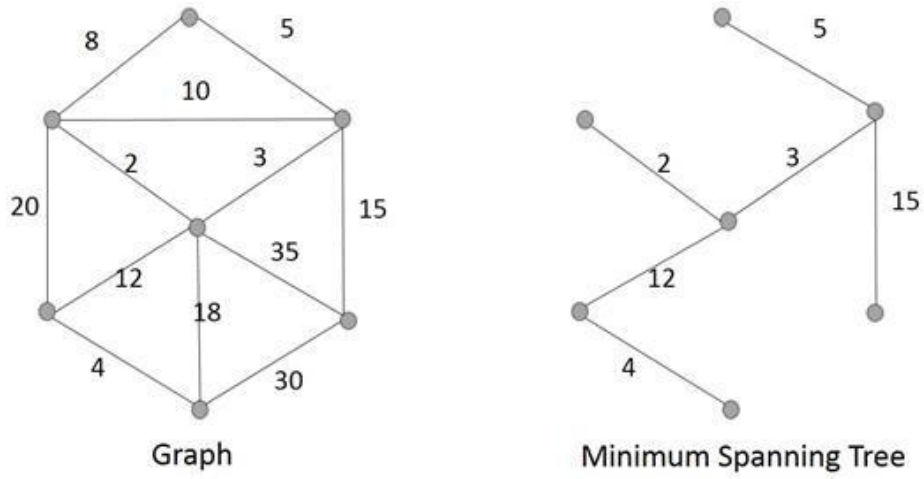
```
    Stack<Integer> stk = new Stack<Integer>();
    for (int i = 0; i < count; i++) {
        if (visited[i] == false) {
            dfsUtil2(gph, i, visited, stk);
        }
    }
    Graph gReversed = transposeGraph(gph);
    for (int i = 0; i < count; i++) {
        visited[i] = false;
    }

    Stack<Integer> stk2 = new Stack<Integer>();
    while (stk.isEmpty() == false) {
        int index = stk.pop();
        if (visited[index] == false) {
            stk2.clear();
            dfsUtil2(gReversed, index, visited, stk2);
            System.out.println(stk2);
        }
    }
}
```

Minimum Spanning Trees (MST)

A **Spanning Tree** of a graph G is a tree that contains all the vertices of the Graph.
A **Minimum Spanning Tree** is a spanning-tree whose sum of length / weight of edges is minimum as possible.

For example, if you want to setup communication between a set of cities, then you may want to use the least amount of wire as possible. MST can be used to find the network path and wire cost estimate.



Prim's Algorithm for MST

Prim's algorithm grows a single tree T, one edge at a time, until it becomes a spanning tree.
We initialize T with zero edges and U with a single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in us. Since each edge adds one new vertex to U, after n − 1 additions, U contains all the vertices of the spanning tree and T becomes a spanning tree.

Example 12.21:

```
// Returns the MST by Prim's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Prim(G)
    T = {}
    Let r be any vertex in G
    U = {r}
    for i = 1 to |V| - 1 do
        e = minimum-weight edge (u, v)
            With u in U and v in V-U
        U = U + {v}
        T = T + {e}
    return T
```

Prim's Algorithm using a priority queue to get the closest next vertex
Time complexity is O(E log V) where V vertices and E edges of the MST.

Example 12.22: Prims algorithm implementation for adjacency list representation of graph.

```
public static void prims(Graph gph) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];
    int source = 1;

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = 999999; // infinite
    }

    dist[source] = 0;
    previous[source] = -1;
    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);
    Edge node = new Edge(source, 0);
    queue.add(node);

    while (queue.isEmpty() != true) {
        node = queue.peek();
        queue.remove();
        visited[source] = true;
        source = node.dest;
        LinkedList<Edge> adl = gph.Adj.get(source);
        for (Edge adn : adl) {
            int dest = adn.dest;
            int alt = adn.cost;
            if (dist[dest] > alt && visited[dest] == false) {
                dist[dest] = alt;
                previous[dest] = source;
```

```
                    node = new Edge(dest, alt);
                    queue.add(node);
                }
            }
        }

        // printing result.
        int count = gph.count;
        for (int i = 0; i < count; i++) {
            if (dist[i] == Integer.MAX_VALUE) {
                System.out.println(" node id " + i + "  prev " + previous[i] + " distance :
Unreachable");
            } else {
                System.out.println(" node id " + i + "  prev " + previous[i] + " distance : " +
dist[i]);

            }
        }
    }
```

Example 12.23: Prims algorithm implementation for adjacency matrix representation of graph.

```
public static void prims(GraphAM gph) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    int source = 0;
    boolean[] visited = new boolean[gph.count];

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = Integer.MAX_VALUE; // infinite
        visited[i] = false;
    }

    dist[source] = 0;
    previous[source] = -1;

    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);

    Edge node = new Edge(source, 0);
    queue.add(node);

    while (queue.isEmpty() != true) {
        node = queue.peek();
        queue.remove();
        source = node.dest;
        visited[source] = true;
        for (int dest = 0; dest < gph.count; dest++) {
            int cost = gph.adj[source][dest];
            if (cost != 0) {
                int alt = cost;
                if (dist[dest] > alt && visited[dest] == false) {

                    dist[dest] = alt;
```

```
                        previous[dest] = source;
                        node = new Edge(dest, alt);
                        queue.add(node);
                    }
                }
            }
        }

        int count = gph.count;
        for (int i = 0; i < count; i++) {
            if (dist[i] == Integer.MAX_VALUE) {
                System.out.println(" \n node id " + i + "  prev " + previous[i] + " distance :
Unreachable");
            } else {
                System.out.println(" node id " + i + "  prev " + previous[i] + " distance : " +
dist[i]);
            }
        }
}

private static class Edge {
    private int dest;
    private int cost;

    public Edge(int dst, int cst) {
        dest = dst;
        cost = cst;
    }
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}
```

Kruskal's Algorithm

Kruskal's Algorithm repeatedly chooses the smallest-weight edge that does not form a cycle.
Sort the edges in non-decreasing order of cost: c (e1) ≤ c (e2) ≤ · · · ≤ c (em).
Set T to be the empty tree. Add edges to tree one by one, if it does not create a cycle.

```
// Returns the MST by Kruskal's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Kruskal(G)
```

```
Sort the edges E by their weights
T = { }
while |T | + 1 < |V | do
    e = next edge in E
    if T + {e} does not have a cycle then
        T = T + {e}
return T
```

Kruskal's Algorithm is O(E log V) using efficient cycle detection.

Shortest Path Algorithms in Graph

Single Source Shortest Path

For a graph G= (V, E), the single source shortest path problem is to find the shortest path from a given source vertex s to all the vertices of V.

Single Source Shortest Path for unweighted Graph.

Find single source shortest path for unweighted graph or a graph with all the vertices of same weight.

Or

Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.

Example 12.24:

```
public static void shortestPath(Graph gph, int source)// unweighted graph
{
    int curr;
    int count = gph.count;
    int[] distance = new int[count];
    int[] path = new int[count];
    for (int i = 0; i < count; i++) {
        distance[i] = -1;
    }
    Queue<Integer> que = new LinkedList<Integer>();
    que.add(source);
    distance[source] = 0;
    while (que.isEmpty() == false) {
        curr = que.remove();
        LinkedList<Edge> adl = gph.Adj.get(curr);
        for (Edge adn : adl) {
            if (distance[adn.dest] == -1) {
                distance[adn.dest] = distance[curr] + 1;
                path[adn.dest] = curr;
                que.add(adn.dest);
            }
        }
    }
    for (int i = 0; i < count; i++) {
        System.out.println(path[i] + " to " + i + " weight " + distance[i]);
```

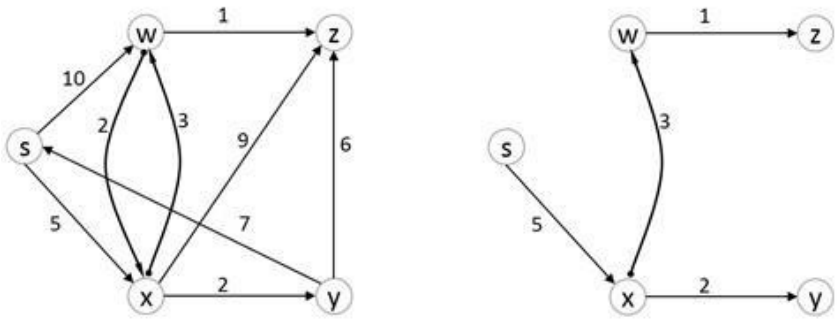
```
    }
}
```

Analysis:

- First, the starting point source is added to a queue.
- Breadth first traversal is performed
- Nodes that are closer to the source are traversed first and processed.

Dijkstra's algorithm

Dijkstra's algorithm is used for single-source shortest path problem for weighted edges with no negative weight. Given a weighted connected graph G, find shortest paths from the source vertex s to each of the other vertices. Dijkstra's algorithm is similar to prims algorithm. It maintains a set of nodes for which shortest path is known.



Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning as we do not know the actual path to the nodes and parents of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)

At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and a parent list fully populated.

```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
for each v in V do
    D[v] = infinite   // Unknown distance
    P[v] = null //unknown previous node
    add v to PQ //adding all nodes to priority queue

D[source] = 0 // Distance from source to source

while (PQ is not empty)
    u = vertex from PQ with smallest D[u]
    remove u from PQ
    for each v adjacent from u do
        alt = D[u] + length ( u , v)
```

```
        if  alt < D[v] then
            D[v] = alt
            P[v] = u
Return D[] , P[]
```

Time complexity is O(|E|log|V|) Where V is the number of vertices and E is the number of edges in the given graph.

Note: Dijkstra's algorithm does not work for graphs with negative edge weight.
Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Example 12.25: Dijkstra's algorithm for adjacency list implementation of graph.

```
public static void dijkstra(Graph gph, int source) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = 999999; // infinite
    }

    dist[source] = 0;
    previous[source] = -1;
    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);
    Edge node = new Edge(source, 0);
    queue.add(node);

    while (queue.isEmpty() != true) {
        node = queue.peek();
        queue.remove();
        source = node.dest;
        visited[source] = true;
        LinkedList<Edge> adl = gph.Adj.get(source);
        for (Edge adn : adl) {
            int dest = adn.dest;
            int alt = adn.cost + dist[source];
            if (dist[dest] > alt && visited[dest] == false) {

                dist[dest] = alt;
                previous[dest] = source;
                node = new Edge(dest, alt);
                queue.add(node);
            }
        }
    }

    int count = gph.count;
    for (int i = 0; i < count; i++) {
        if (dist[i] == Integer.MAX_VALUE) {
            System.out.println(" \n node id " + i + "  prev " + previous[i] + " distance :
Unreachable");
        } else {
```

```
            System.out.println(" node id " + i + "  prev " + previous[i] + " distance : " +
dist[i]);

        }
    }
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}
```

Example 12.26: Dijkstra's algorithm for adjacency matrix implementation of graph.

```
public static void dijkstra(GraphAM gph, int source) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = Integer.MAX_VALUE; // infinite
        visited[i] = false;
    }

    dist[source] = 0;
    previous[source] = -1;
    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);

    Edge node = new Edge(source, 0);
    queue.add(node);

    while (queue.isEmpty() != true) {
        node = queue.peek();
        queue.remove();
        source = node.dest;
        visited[source] = true;
        for (int dest = 0; dest < gph.count; dest++) {
            int cost = gph.adj[source][dest];
            if (cost != 0) {
                int alt = cost + dist[source];
                if (dist[dest] > alt && visited[dest] == false) {

                    dist[dest] = alt;
                    previous[dest] = source;
                    node = new Edge(dest, alt);
                    queue.add(node);
```

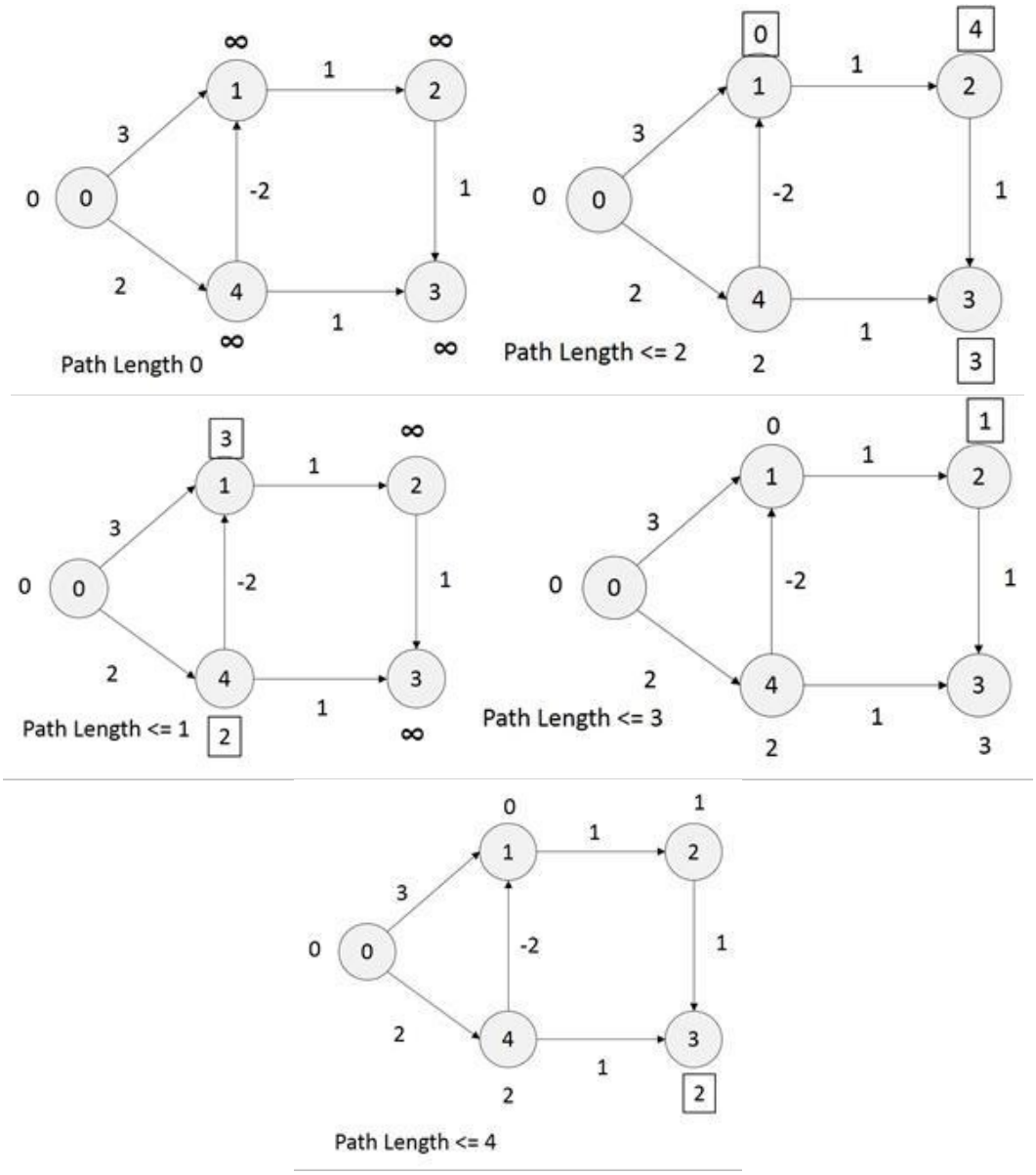
```
                    }
                }
            }
        }

        int count = gph.count;
        for (int i = 0; i < count; i++) {
            if (dist[i] == Integer.MAX_VALUE) {
                System.out.println(" \n node id " + i + "  prev " + previous[i] + " distance :
Unreachable");
            } else {
                System.out.println(" node id " + i + "  prev " + previous[i] + " distance : " +
dist[i]);

            }
        }
}

private static class Edge {
    private int dest;
    private int cost;

    public Edge(int dst, int cst) {
        dest = dst;
        cost = cst;
    }
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}
```

Bellman Ford Shortest Path

Single source shortest path from a source vertex s to all other vertices in a graph containing negative weight edges but no negative cycle is done using Bellman Ford algorithm. It does not work if there is some cycle in the graph whose total weight is negative. In this algorithm, distance of all the vertices is assigned to ∞ and the source vertex distance is assigned as 0. Then V-1 passes (V is total no of vertices) over all the edges is performed and the distance to destination is updated for each vertice.

We can checks over all the edges again and if there is change in the weights in the distance then it detect a negative weight cycle. If distance is changed after V-1 relaxation then we have a negative cycle. We can stop the algorithm if an iteration does not modify distance estimates. This is beneficial if shortest paths are likely to be less than V-1.

Time complexity is O(V.E), where V is number of vertices and E is the total number of edges.



Example 12.27:

```
public static void bellmanFordshortestPath(Graph gph, int source) {
    int count = gph.count;
    int[] distance = new int[count];
    int[] path = new int[count];

    for (int i = 0; i < count; i++) {
        distance[i] = 999999; // infinite
        path[i] = -1;
    }
    distance[source] = 0;
    // Outer loop will run (V-1) number of times.
    // Inner for loop and while loop runs combined will
    // run for Edges number of times.
    // Which make the total complexity as O(V*E)
```

```
    for (int i = 0; i < count - 1; i++) {
        for (int j = 0; j < count; j++) {
            LinkedList<Edge> adl = gph.Adj.get(j);
            for (Edge adn : adl) {
                int newDistance = distance[j] + adn.cost;
                if (distance[adn.dest] > newDistance) {
                    distance[adn.dest] = newDistance;
                    path[adn.dest] = j;
                }
            }
        }
    }
    for (int i = 0; i < count; i++) {
        System.out.println(path[i] + " to " + i + " weight " + distance[i]);
    }
}
```

All Pairs Shortest Paths

Given a weighted graph G(V, E), the all pair shortest path problem is used to find the shortest path between all pairs of vertices u, v ϵ V. Execute V instances of single source shortest path algorithm for each vertex of the graph.

The complexity of this algorithm will be $O(V^3)$

Hamiltonian Path and Hamiltonian Circuit

Hamiltonian path is a path in which every vertex is visited exactly once with no repeats, it does not have to start and end at the same vertex.

Hamiltonian path is a Np-Complete problem, so the only solution is possible using backtracking, which starts from a vertex s and try all adjacent vertices recursively. If we do not find the path, then we backtrack and try other vertices.

Example 12.28:

```
public static boolean hamiltonianPathUtil(GraphAM graph, int path[], int pSize, int added[])
{
    // Base case full length path is found
    if (pSize == graph.count) {
        return true;
    }
    for (int vertex = 0; vertex < graph.count; vertex++) {
        // there is a path from last element and next vertex
        // and next vertex is not already included in path.
        if (pSize == 0 || (graph.adj[path[pSize - 1]][vertex] == 1 && added[vertex] == 0)) {
            path[pSize++] = vertex;
            added[vertex] = 1;
            if (hamiltonianPathUtil(graph, path, pSize, added))
                return true;
            // backtracking
            pSize--;
            added[vertex] = 0;
        }
```

```
    }
    return false;
}

public static boolean hamiltonianPath(GraphAM graph) {
    int[] path = new int[graph.count];
    int[] added = new int[graph.count];

    if (hamiltonianPathUtil(graph, path, 0, added)) {
        System.out.println("Hamiltonian Path found :: ");
        for (int i = 0; i < graph.count; i++)
            System.out.println(" " + path[i]);

        return true;
    }
    System.out.println("Hamiltonian Path not found");
    return false;
}
```

Hamiltonian circuit is a Hamiltonian Path such that there is an edge from its last vertex to its first vertex. A **Hamiltonian circuit** is a circuit that visits every vertex exactly once and it must start and end at the same vertex.

Solution of Hamiltonian circuit is also NP-compete problem. The only difference is the base condition in which there should be a path from last node of the Hamiltonian path to the first element.

Example 12.29:

```
public static boolean hamiltonianCycleUtil(GraphAM graph, int path[], int pSize, int added[])
{
    // Base case full length path is found
    // this last check can be modified to make it a path.
    if (pSize == graph.count) {
        if (graph.adj[path[pSize - 1]][path[0]] == 1) {
            path[pSize] = path[0];
            return true;
        } else
            return false;
    }
    for (int vertex = 0; vertex < graph.count; vertex++) {
        // there is a path from last element and next vertex
        if (pSize == 0 || (graph.adj[path[pSize - 1]][vertex] == 1 && added[vertex] == 0)) {
            path[pSize++] = vertex;
            added[vertex] = 1;
            if (hamiltonianCycleUtil(graph, path, pSize, added))
                return true;
            // backtracking
            pSize--;
            added[vertex] = 0;
        }
    }
    return false;
}
```

```
public static boolean hamiltonianCycle(GraphAM graph) {
    int[] path = new int[graph.count + 1];
    int[] added = new int[graph.count];
    if (hamiltonianCycleUtil(graph, path, 0, added)) {
        System.out.println("Hamiltonian Cycle found :: ");
        for (int i = 0; i <= graph.count; i++)
            System.out.print(" " + path[i]);
        return true;
    }
    System.out.println("Hamiltonian Cycle not found");
    return false;
}

public static void main(String[] args) {
    int count = 5;
    GraphAM graph = new GraphAM(count);
    int[][] adj = {
            { 0, 1, 0, 1, 0 },
            { 1, 0, 1, 1, 0 },
            { 0, 1, 0, 0, 1 },
            { 1, 1, 0, 0, 1 },
            { 0, 1, 1, 1, 0 } };

    for (int i = 0; i < count; i++)
        for (int j = 0; j < count; j++)
            if (adj[i][j] == 1)
                graph.addDirectedEdge(i, j, 1);
    System.out.println("hamiltonianPath : " + hamiltonianPath(graph));
    System.out.println("hamiltonianCycle : " + hamiltonianCycle(graph));

    GraphAM graph2 = new GraphAM(count);
    int[][] adj2 = {
            { 0, 1, 0, 1, 0 },
            { 1, 0, 1, 1, 0 },
            { 0, 1, 0, 0, 1 },
            { 1, 1, 0, 0, 0 },
            { 0, 1, 1, 0, 0 } };

    for (int i = 0; i < count; i++)
        for (int j = 0; j < count; j++)
            if (adj2[i][j] == 1)
                graph2.addDirectedEdge(i, j, 1);

    System.out.println("hamiltonianPath :  " + hamiltonianPath(graph2));
    System.out.println("hamiltonianCycle :  " + hamiltonianCycle(graph2));
}
```

Euler path and Euler Circuit

Eulerian Path is a path in the graph that visits every edge exactly once.

Eulerian Circuit is an Eulerian Path, which starts and ends on the same vertex. Or **Eulerian Circuit** is a path in the graph that visits every edge exactly one and it starts and ends on the same vertex.

A graph is called Eulerian if there is an Euler circuit in it. A graph is called Semi-Eulerian if there is an Euler Path in the graph. If there is no Euler path possible in the graph, then it is called non-Eulerian.

A graph is Eulerian if all the edges have even number of number of edges in it. A graph is Semi-Eulerian if it has exactly two vertices with odd number of edges or odd degree. In all other cases, the graph is Non-Eulerian.

Example 12.30: Check if the graph is Eulerian.

```
public static int isEulerian(Graph graph) {
    int count = graph.count;
    int odd;
    int[] inDegree;
    int[] outDegree;
    LinkedList<Edge> adl;
    // Check if all non - zero degree nodes are connected
    if (isConnected(graph) == false) {
        System.out.println("graph is not Eulerian");
        return 0;
    } else {
        // Count odd degree
        odd = 0;
        inDegree = new int[count];
        outDegree = new int[count];

        for (int i = 0; i < count; i++) {
            adl = graph.Adj.get(i);
            for (Edge adn : adl) {
                outDegree[i] += 1;
                inDegree[adn.dest] += 1;
            }
        }
        for (int i = 0; i < count; i++) {
            if ((inDegree[i] + outDegree[i]) % 2 != 0) {
                odd += 1;
            }
        }
    }

    if (odd == 0) {
        System.out.println("graph is Eulerian");
        return 2;
    } else if (odd == 2) {
        System.out.println("graph is Semi-Eulerian");
        return 1;
    } else {
        System.out.println("graph is not Eulerian");
        return 0;
    }
}
```

Travelling Salesman Problem (TSP)

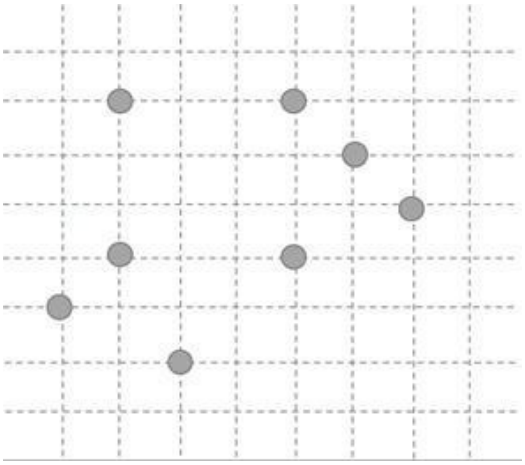
Problem: The travelling salesperson problem tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. Alternatively, find the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.

```
Algorithm TSP
Select a city
MinTourCost = infinite
For ( All permutations of cities ) do
    If( LengthOfPathSinglePermutation < MinTourCost )
        MinTourCost = LengthOfPath
```

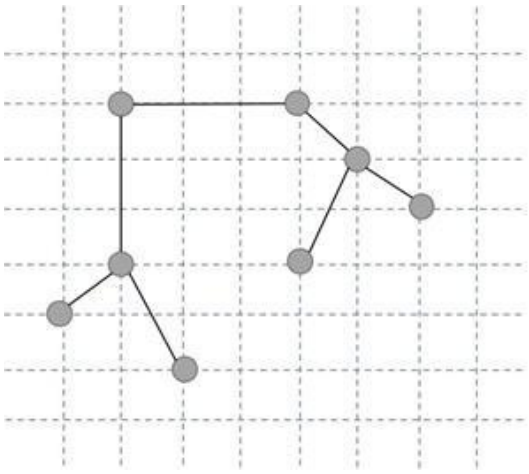
Total number of possible combinations = (n-1)!
Cost for calculating the path: Θ(n)
So the total cost of finding the shortest path: Θ(n!)

It is an NP-Hard problem there is no efficient algorithm to find its solution. Even if some solution is given, it is equally hard to verify that this is a correct solution or not. However, some approximate algorithms can be used to find a good solution. We will not always get the best solution, but will get a good solution.

Our approximate algorithm is based on the minimum spanning tree problem. In which we have to construct a tree from a graph such that every node is connected by edges of the graph and the total sum of the cost of all the edges is minimum.

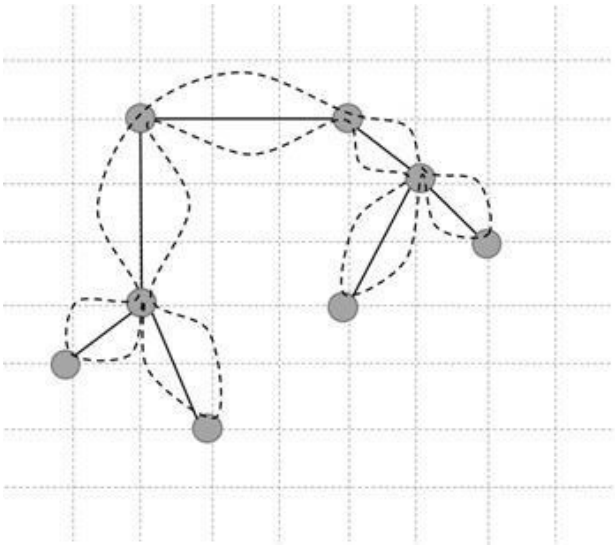


In the above diagram, we have a group of cities (each city is represented by a circle.) Which are located in the grid and the distance between the cities is same as per the actual distance. And there is a path from each city to another city which is a straight path from one to another.

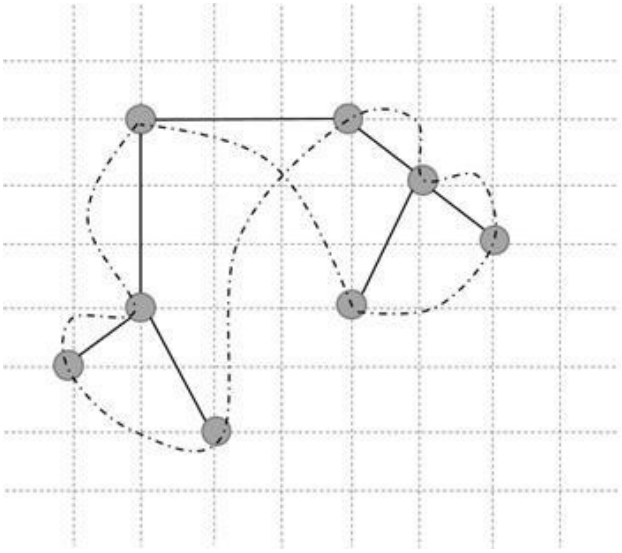


We have made a minimum spanning tree for the above city graph.

What we want to prove that the shortest path in a TSP will always be greater than the length of MST. Since in MST all nodes are connected to the next node, which is also the minimum distance from the group of node. Therefore, to make it a path without repeating the nodes we need to go directly from one node to other without following MST. At that point, when we are not following MST we are choosing an edge, which is greater, then the edges provided by MST. So TSP path will always be greater than or equal to MST path.



Now let us take a path from starting node and traverse each node on the way given above and then come back to the starting node. The total cost of the path is 2MST. The only difference is that we are visiting many nodes multiple times.



Now let us change our traversal algorithm so that it will become TSP in our traversal, we do not visit an already visited node we will skip them and will visit the next unvisited node. In this algorithm, we will reach the next node by as shorter path. (The sum of the length of all the edges of a polygon is always greater than a single edge.) Ultimately, we will get the TSP and its path length is not more than twice the optimal solution. Therefore, the proposed algorithm gives a good result.

Exercise

1. In the various path-finding algorithms, we have created a path array that just stores immediate parent of a node, print the complete path for it.

2. All the functions are implemented considering as if the graph is represented by adjacency list. Write all those functions for graph representation as adjacency matrix.

3. In a given start string, end string and a set of strings, find if there exists a path between the start string and end string via the set of strings.

   A path exists if we can get from start string to end the string by changing (no addition/removal) only one character at a time. The restriction is that the new string generated after changing one character has to be in the set.

   Start: "cog"
   End: "bad"
   Set: ["bag", "cag", "cat", "fag", "con", "rat", "sat", "fog"]
   One of the paths: "cog" -> "fog" -> "fag" -> "bag" -> "bad"

CHAPTER 13: STRING ALGORITHMS

Introduction

String in Java language is an array of character. We use string algorithm in so many tasks, when we are using some copy-paste, some string replacement, and some string search. When we are using some dictionary program, we are using string algorithms. When we are searching