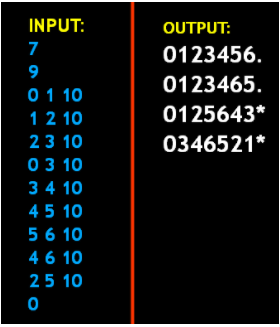
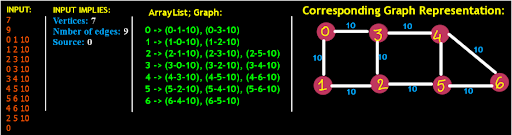
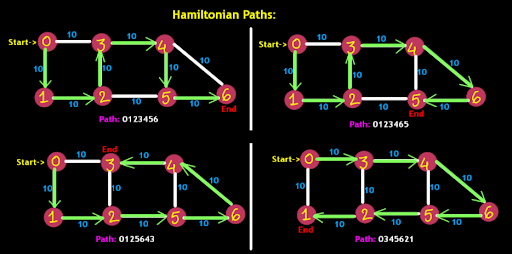
**1. PROBLEM DISCUSSION :**

Prerequisite for this problem is the “Find and Print All Paths”. In case you have not gone through this problem, it is advised that you solve and understand these first. In this problem, we are given a graph and a source vertex. We are required to find and print all "Hamiltonian Paths and Cycles" starting from source. The cycles must end with "\*" and paths with a ".". And we have to print the path in lexicographically increasing order. What is the Hamiltonian Path and Cycle? A Hamiltonian Path is such a path which visits all vertices without visiting any vertex twice i.e all the vertices in the graph are traversed exactly once. A Hamiltonian Path becomes a cycle if there is an edge between the first and last vertex. For example : Let’s consider the following graph :

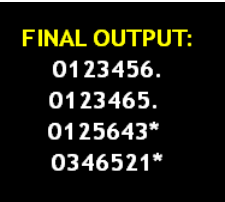
****

****

For this graph representation, we have four following possible Hamiltonian Paths.

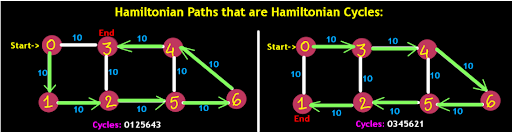
****

In all these paths, we can see that all the vertices of the graph are visited exactly once, so all these paths are Hamiltonian Paths. Out of these Hamiltonian Paths, two are Hamiltonian Cycles as there is edge between start and end vertex of the path. So we have four Hamiltonian Paths and out of those four, two are cycles. So we have to add "." at the end of paths and "\*" at the end of cycles. Therefore the output looks like this :

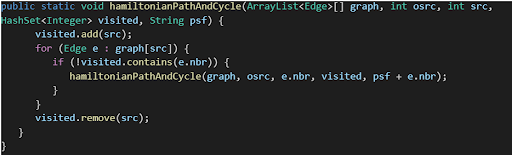
****

**2. APPROACH :**

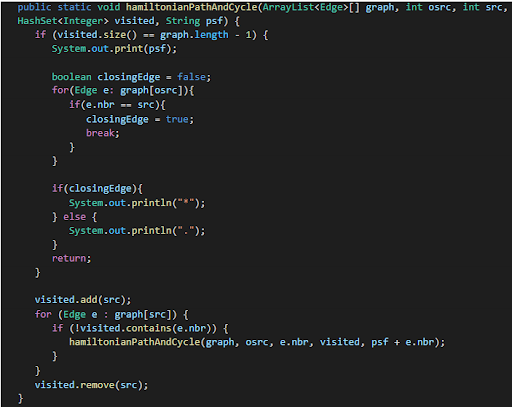
Before we jump to the code of this problem, let me tell you that this problem is very similar to the Print All Paths problem. In Print All Paths, we used to find the path between a given source and destination vertex. Each time we visited any vertex, we used to set the value corresponding to the vertex as true in the visited array. And when we were done exploring all neighbor's of this vertex then we again set the value as false corresponding to this vertex in the visited array. Also if the base case was hit, that is when source becomes equal to destination, we printed the path so far. We will follow a similar approach in this problem as well. But this time the base case will change. Since we need to visit each vertex of the graph this time, the base case will become the stage where all vertices have been visited. Also, also, also, we need to check that the path obtained is Hamiltonian Path or Hamiltonian Cycle.

****

For doing so, we check whether there is an edge between source and last visited vertex. If there is edge than it is Hamiltonian Cycle and we print, "\*". And if there is no edge then it is the Hamiltonian Path and we print, ".". Let's try to code this: If you remember, Print All Path has exactly the same code, except the base case, which we have not handled yet. And 2 minor changes can be seen . Instead of using an array to keep a check for visited vertices, we have used a Hashset and second is that there is an extra parameter, osrc (original source vertex). Both these changes concern the base case.

****

We made the first change which is regarding the check for visited, so that while coding the base case, when we need to count the visited number of vertices so far, we can simply use the size of this Hashset instead of using some for loop. Using a hashset not only eases the coding but also, accessing the hashset's size takes constant time. For the base case we check whether the size of the visited hashset is equal to one less than the size of the graph. Why 1 less? Well, when the call will be made for the last vertex, at that time this last vertex will not be present in the visited hashset. So, subtracting one from the size of Hashset compensates for the last vertex. So if the condition (visited.size() == graph.length - 1) is satisfied then the string psf (storing path so far) will get printed. But we are not done yet! Yes the second change, an extra parameter osrc, storing the original source. We also need to check whether the psf (path so far) is Hamiltonian Path or Cycle. To do so, we check whether there is an edge between the osrc vertex (original source vertex) and src vertex (source vertex at this call). For checking this, we define a variable, closing edge of type Boolean and store false in it. Then we run a for loop and visit each edge of the original source, and for each edge we check whether the neighbor of osrc equals src. If it does then we set the value of the closing edge to true and break the loop. Then we check if the closing edge is true or not. If it's true, then it implies src and osrc are neighbors and there is an edge between these two, making the particular path Hamiltonian Cycle. So we print "\*". Otherwise it means the closing edge is false and there is no edge between src and osrc, letting it be Hamiltonian Path only. So here we print ".".

****

Yes! Now that looks complete.

**3. CODE :**

Note: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the code.

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList< Edge>[] graph = new ArrayList[vtces];

for (int i = 0; i < vtces; i++) {

graph[i] = new ArrayList< >();

}

int edges = Integer.parseInt(br.readLine());

for (int i = 0; i < edges; i++) {

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

int src = Integer.parseInt(br.readLine());

HashSet<Integer> visited = new HashSet< >();

hamiltonianPathAndCycle(graph, src, src, visited, src + "");

}

public static void hamiltonianPathAndCycle(ArrayList< Edge>[] graph, int osrc, int src, HashSet< Integer> visited, String psf) {

if (visited.size() == graph.length - 1) {

System.out.print(psf);

boolean closingEdge = false;

for (Edge e : graph[osrc]) {

if (e.nbr == src) {

closingEdge = true;

break;

}

}

if (closingEdge) {

System.out.println("\*");

} else {

System.out.println(".");

}

return;

}

visited.add(src);

for (Edge e : graph[src]) {

if (!visited.contains(e.nbr)) {

hamiltonianPathAndCycle(graph, osrc, e.nbr, visited, psf + e.nbr);

}

}

visited.remove(src);

}

}

For more clarity of the question, watch the question video

Play Video

**4. ANALYSIS :**

Time Complexity :

The time complexity of the code is O(V+E) because of the DFS approach used in this solution, where V is the number of vertices and E is the number of edges in the graph.

Space Complexity :

The space complexity of the solution is O(V), since we used a hashset to store the number of vertices which acts as a visited array for this solution.