Representation of Graphs - Adjacency List, Adjacency Matrix & Other Representations

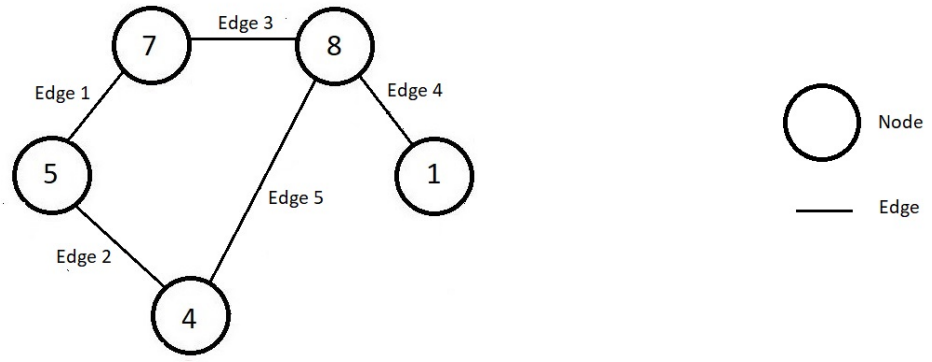
codewithharry.com/videos/data-structures-and-algorithms-in-hindi-84

In the last lecture, we introduced to you our new topic, graphs. We saw the applications of graphs and discussed their elements including their nodes/ vertices and edges. Moving ahead with our discussion on graphs, today we'll learn how graphs are represented, and various terminologies related to graphs.

Representation of graphs is actually a very important concept and once you understand how graphs are represented, learning other graph algorithms becomes much easier. The first lecture detailed the following points:

1. Graphs and their nodes & edges, directed and undirected edges and graphs.
2. Applications including their use to model real-world problems like managing a social network, website links, etc.
3. Another major application is their use to solve problems like is there a path between two locations on a map and if there is, which one is the shortest.

A simple undirected graph looks like this:



Let's now move to see how we represent graphs in various ways.

Ways to represent a graph:

Any representation should basically be able to store the nodes of a graph and their connections between them. And this can be accomplished in so many ways but primarily the most used way to represent a graph is,

1. **Adjacency List** - Mark the nodes with their neighbours
2. **Adjacency Matrix** - Aij = 1, if there is an edge between i and j, 0 otherwise.

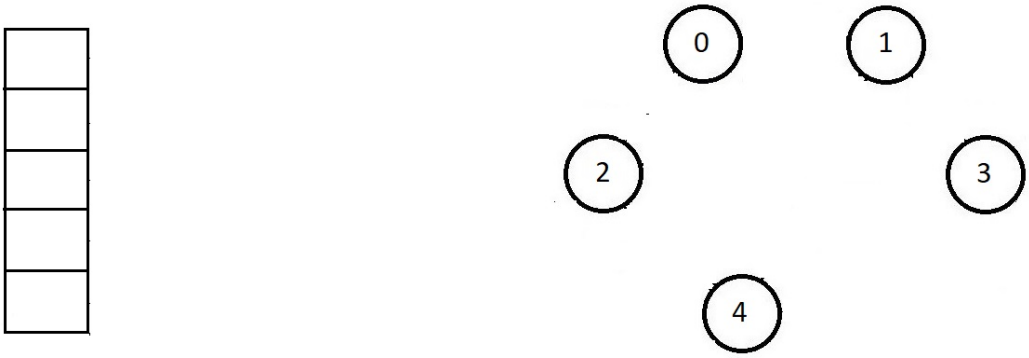
We'll see the above two in detail. Meanwhile, other representations include:

1. **Edge Set** - Store the pair of nodes/vertices connected with an edge. Example: {(0, 1), (0, 4), (1, 4)}.
2. Other implementations to represent a graph also exist. For example, Compact list representation, cost adjacency list, cost adjacency matrix, etc.

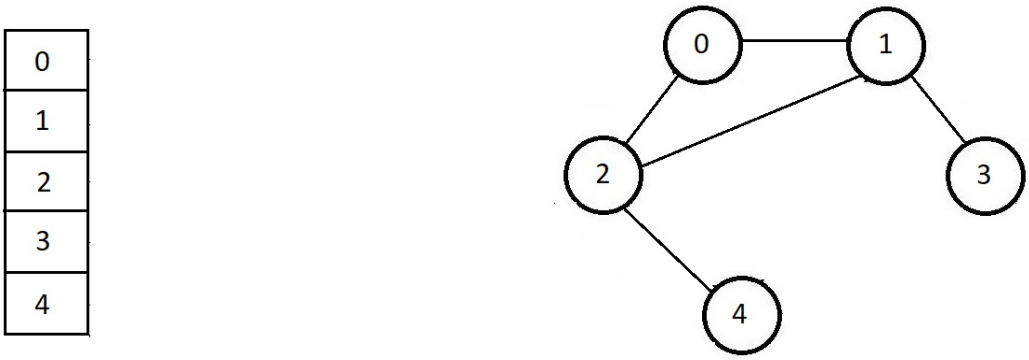
Let's now deal with each of these individually.

Adjacency List:

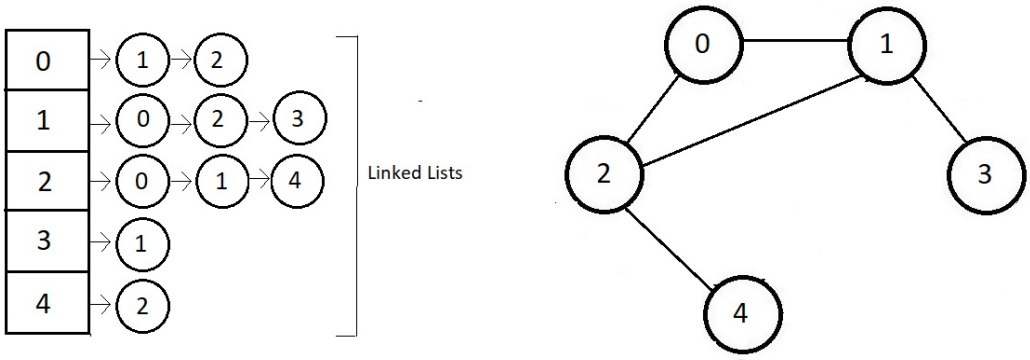
In this method of representation of graphs, we store the nodes, along with the list of their neighbors. Basically, we maintain a list of all the does, and along with it, we store the list of nodes a particular node is connected with. Consider the nodes I've illustrated below.



So first, we store the nodes we have in the graph.



Next, we look for the connections of each of these nodes we stored. Starting with 0, you can see that 0 is connected with both 1 and 2. So, we will store that beside node 0. Next, 1 is connected with all 0, 2, and 3. 2 is connected with both 0 and 4, and 3 is connected with only 1, and 4 is connected with only 2. So, this information gets stored as shown below.



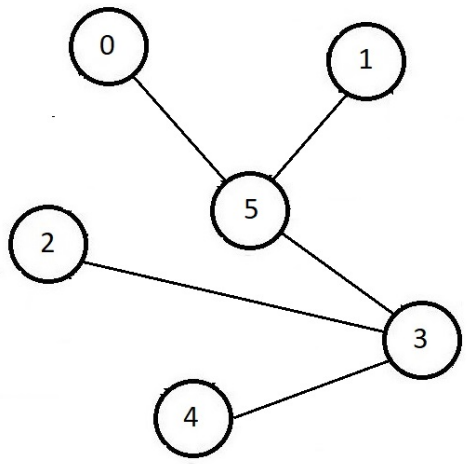
And the information about the connections of each of the nodes gets stored in separate linked lists as shown in the figure above. And each of the nodes itself acts as a pointer stored in an array pointing to the head of each of the linked lists. So, in this case, we would have an array of length 5 where the first index stores a pointer to the head of the adjacency linked list of the node 0.

And this was the adjacency list representation of graphs, and I can say that this is one of the most used representation methods. Next is the adjacency matrix.

Adjacency Matrix:

Adjacency matrix is another method of representation of graphs, where we represent our graph in the form of a matrix where cells are either filled with 0 or 1. Let's call the cell falling on the intersection of ith row and jth column be Aij, then the cell would be filled with 1 if there is an edge between node i and j, otherwise, the cell would be filled with a 0.

Consider the graph illustrated below:



Now, we'll make a 6x6 matrix as follows:

Now, we'll iterate through each of the cells, and see if there is an edge between the row number and the column number or not. If there is an edge, we'll place 1 in that cell, otherwise a zero. Since there are no self-loops even, we'll mark 0 to the cell having i=j. The filled adjacency matrix for the above graph would be:

	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

	0	1	2	3	4	5
0	0	0	0	0	0	1
1	0	0	0	0	0	1
2	0	0	0	1	1	0
3	0	0	1	0	1	1
4	0	0	0	1	0	0
5	1	1	0	1	0	0

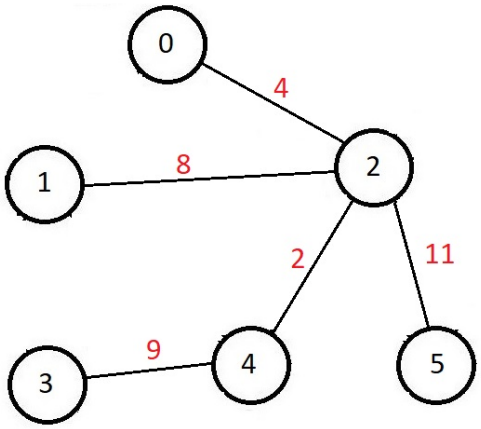
Now, one can very easily find whether there is an edge between any two nodes by simply looking for the cell representing the two nodes and checking if there is a 1 or a 0. So, this was the adjacency matrix representation of graphs.

We can further extrapolate the application of the adjacency matrix by replacing these ones in the matrix with the weights for a weighted graph. Weighted graphs have a value/cost for each of the respective edges. These costs could represent anything, be it distance or time or cost literally. There is one traveling salesman problem where we store the shortest path from a city to some other city. Let's look at the cost adjacency matrix in detail.

Cost Adjacency Matrix:

The cost adjacency matrix is another method of representation of weighted graphs, where we represent our graph in the form of a matrix where cells are either filled with 0 or the cost of the edge. Let's call the cell falling on the intersection of ith row and jth column be Aij, then the cell would be filled with the cost of the edge between node i and j if there is an edge between node i and j, otherwise, the cell would be filled with a 0 and if the cost could also be 0, then we'll fill -1 in the cell where there is no edge.

Consider the graph illustrated below:



Now, we'll make a 6x6 matrix as follows and iterate through each of the cells, and see if there is an edge between the row number and the column number or not. If there is an edge, we'll place 1 in that cell, otherwise a zero. Since there are no self-loops even, we'll mark 0 to the cell having i=j. The filled adjacency matrix for the above graph would be, assuming the cost could be zero as well:

So, this was the cost adjacency matrix representation of graphs. Other implementations are not that frequently used, so let's go through them quickly and in brief.

	0	1	2	3	4	5
0	-1	-1	4	-1	-1	-1
1	-1	-1	8	-1	-1	-1
2	4	8	-1	-1	2	11
3	-1	-1	-1	-1	9	-1
4	-1	-1	2	9	-1	-1
5	-1	-1	11	-1	-1	-1

Edge Set: Store the pair of nodes/vertices connected with an edge. Example: {(0, 1), (0, 2), (1, 2)} for a graph having nodes 0, 1 and 2 all connected with each other.

Cost Adjacency List: Similar to the adjacency list, but instead of just storing the node value, we'll also store the cost of the edge too in the linked list.

Compact List Representation: here, the entire graph is compressed and stored in just one single 1D array.

This was all about representing graphs. We'll be mainly using the adjacency list and adjacency matrix for all our purposes in the coming lectures. We'll see their programming as well in C and a lot more about graphs in the upcoming lecture.

Thank you for being with me throughout the session. I hope you enjoyed it. If you appreciate my work, please let your friends know about this channel too. Lectures on graphs have just started, and if you haven't saved this already, you just have to move on to codewithharry.com or my YouTube channel to access them. See you all there in the next lecture. Till then keep learning.