June15

14 June 2020 23:13

Graphs:

Python has no built-in data type or class for graphs, but it is easy to implement them in Python. One data type is ideal for representing graphs in Python, i.e. dictionaries.

Adjacent vertices:

Two vertices are adjacent when they are both incident to a common edge.

Path in an undirected Graph:

A path in an undirected graph is a sequence of vertices P = (v1, v2, ..., vn) ∈ V x V x ... x V such that vi is adjacent to v{i+1} for 1 ≤ i < n. Such a path P is called a path of length n from v1 to vn.

Simple Path:

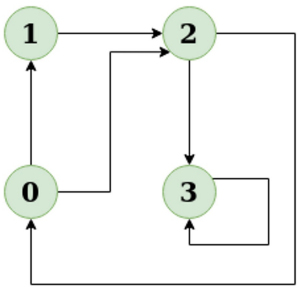
A path with no repeated vertices is called a simple path.

1. **Detect Cycle in a Directed Graph**

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false.

```
Input: n = 4, e = 6
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3
Output: Yes
```

```
The diagram clearly shows a cycle 0 -> 2 -> 0
```



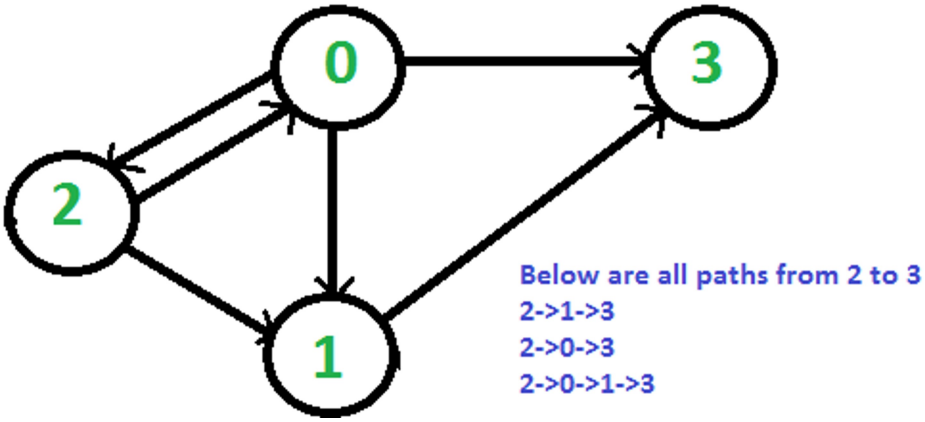
Algorithm:

- Create the graph using the given number of edges and vertices.
- Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
- Mark the current node as visited and also mark the index in recursion stack.
- Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
- If the adjacent vertices are already marked in the recursion stack then return true.
- Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

Complexity Analysis:
Time Complexity: O(V+E).
Time Complexity of this method is same as time complexity of DFS traversal which is O(V+E).
Space Complexity: O(V).
To store the visited and recursion stack O(V) space is needed.

1. **Print all paths from a given source to a destination**

Given a directed graph, a source vertex 's' and a destination vertex 'd', print all paths from given 's' to 'd'.
Consider the following directed graph. Let the s be 2 and d be 3. There are 4 different paths from 2 to 3.

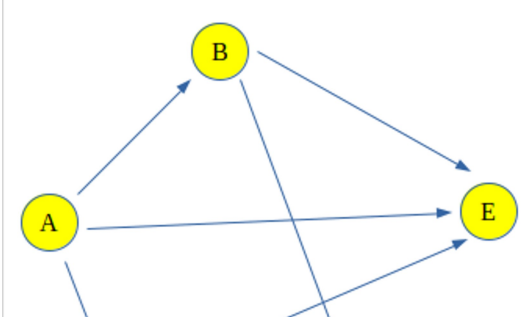


Algorithm:

The idea is to do Depth First Traversal of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination vertex, print contents of path[]. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn't go in a cycle.

3. **Count all possible paths between two vertices**

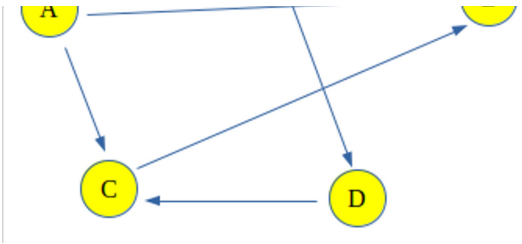
Count the total number of ways or paths that exist between two vertices in a directed graph. These paths don't contain a cycle, the simple enough reason is that a cycle contains an infinite number of paths and hence they create a problem.



Input: Count paths between A and E
Output : Total paths between A and E are 4
Explanation: The 4 paths between A and E are:

```
A -> E
A -> B -> E
A -> C -> E
A -> B -> D -> C -> E
```

Input : Count paths between A and C
Output : Total paths between A and C are 2
Explanation: The 2 paths between A and C are:



Output : Total paths between A and C are 2

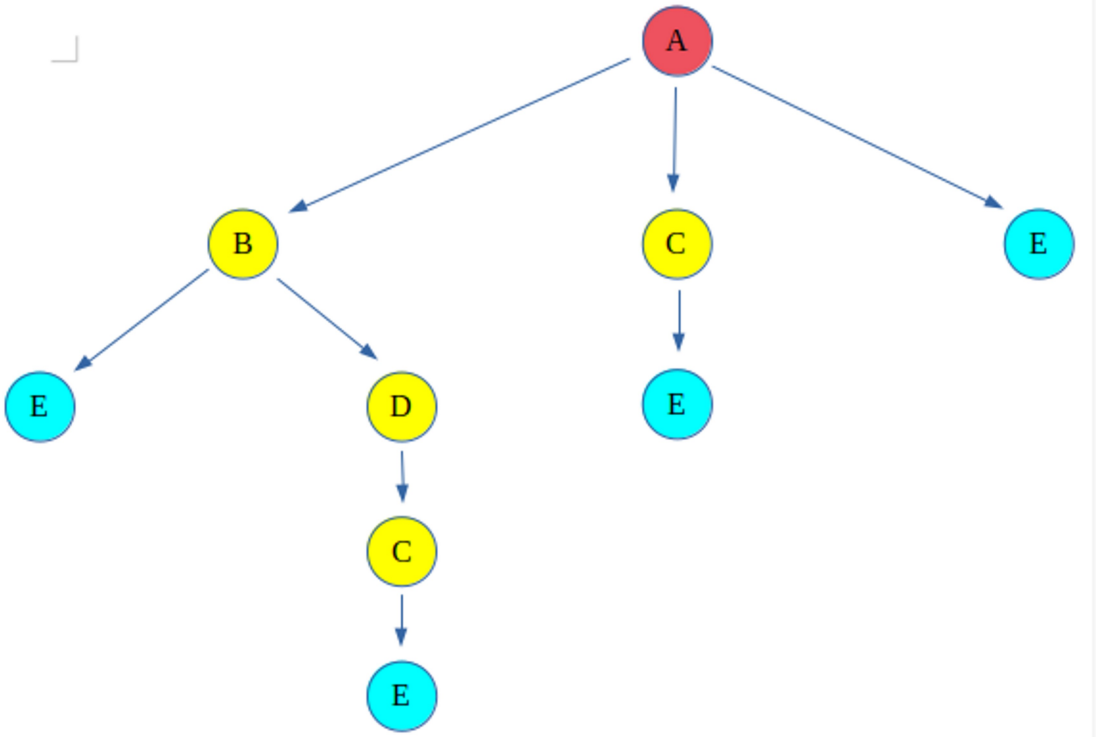
Explanation: The 2 paths between A and C are:

A -> C

A -> B -> D -> C

Approach:
The problem can be solved using backtracking, that says take a path and start walking on it and check if it leads us to the destination vertex then count the path and backtrack to take another path. If the path doesn't lead to the destination vertex, discard the path.
This type of graph traversal is called Backtracking.

Backtracking for above graph can be shown like this:
The red colour vertex is the source vertex and the light-blue colour vertex is destination, rest are either intermediate or discarded paths.



This give four paths between source(A) and destination(E) vertex.

Why this solution will not work for a graph which contains cycles?

The Problem Associated with this is that now if one more edge is added between C and B, it would make a cycle (B -> D -> C -> B). And hence after every cycle through the loop, the length path will increase and that will be considered as a different path, and there would be infinitely many paths because of the cycle.

Algorithm:

- Create a recursive function that takes index of node of a graph and the destination index. Keep

a global or a static variable count to store the count.
- If the current nodes is the destination increase the count.
- Else for all the adjacent nodes, i.e. nodes that are accessible from the current node, call the recursive function with the index of adjacent node and the destination.
- Print the Count.

Complexity Analysis:

Time Complexity: O(N!).
If the graph is complete then there can be around N! recursive calls, so the time Complexity is O(N!)
Space Complexity: O(1).
Since no extra space is required.

4. Depth First Search or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a Boolean visited array.

Approach:
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

Algorithm:
- Create a recursive function that takes the index of node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

Complexity Analysis:
Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
Space Complexity: O(V).
Since, an extra visited array is needed of size V.

5. Breadth First Search or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See

method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

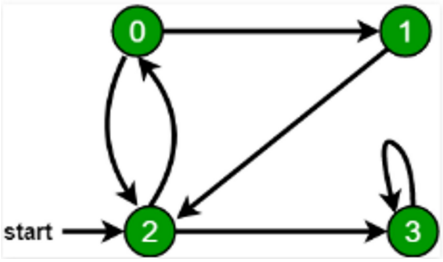
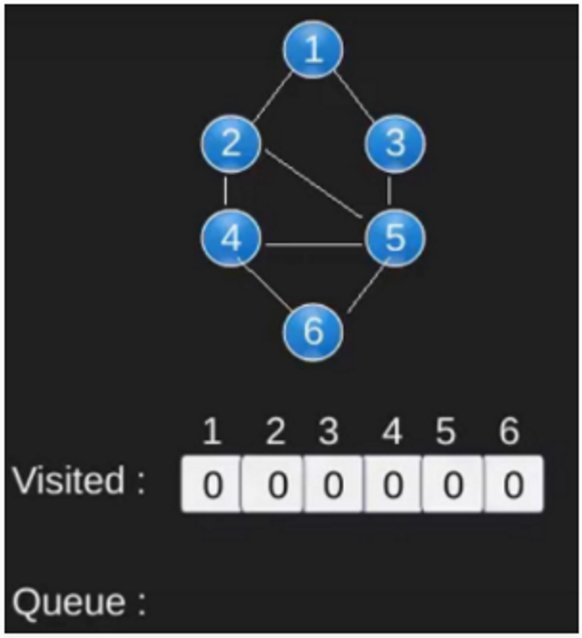
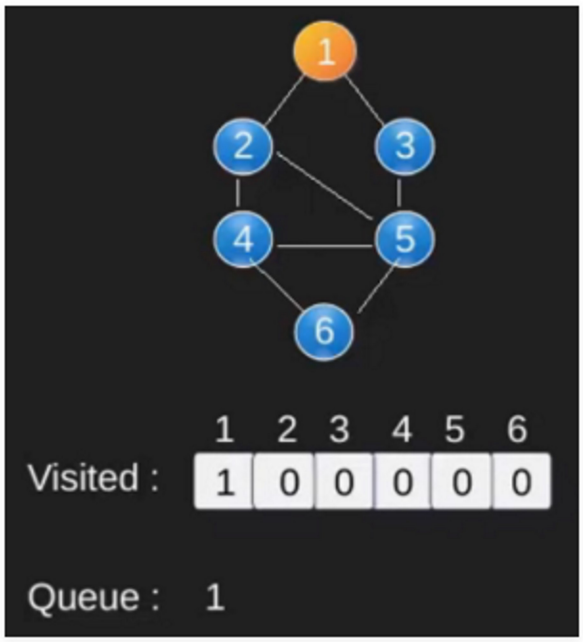
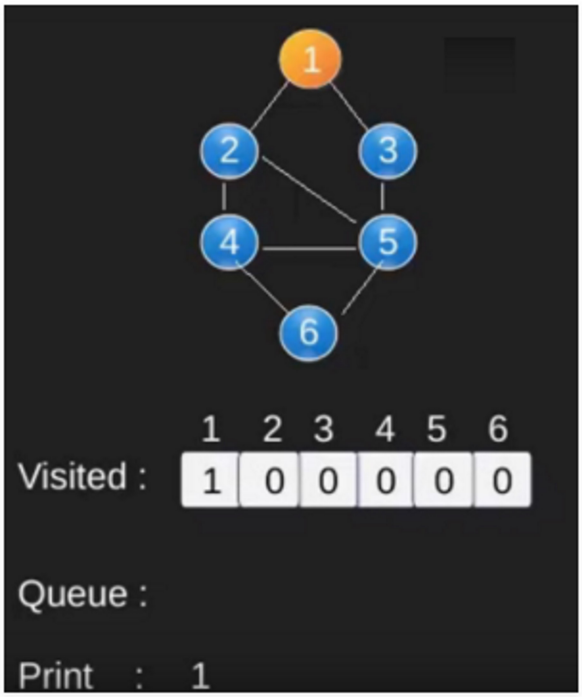
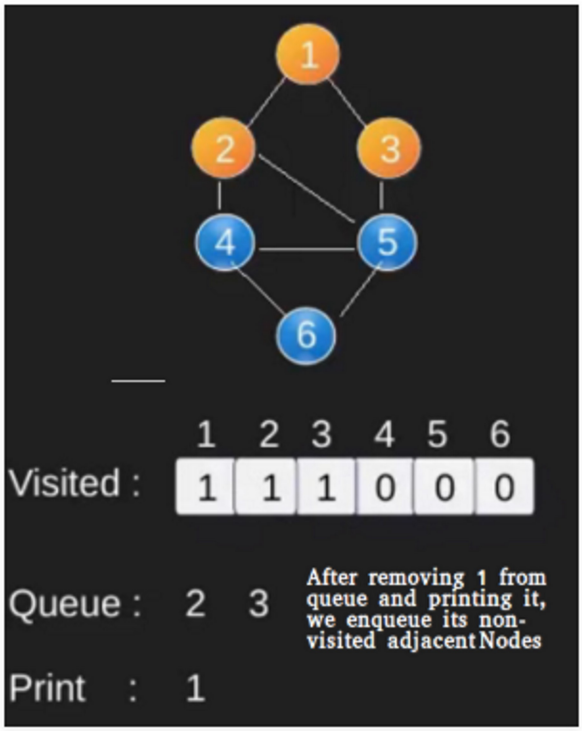


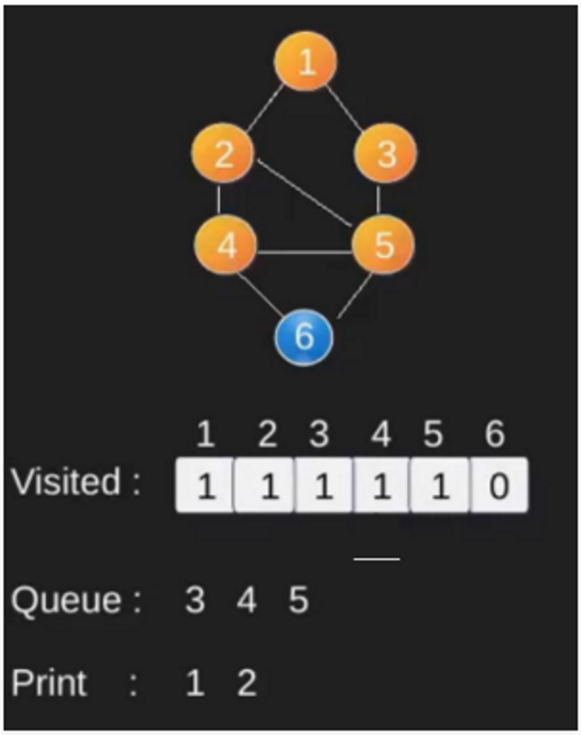
Illustration:

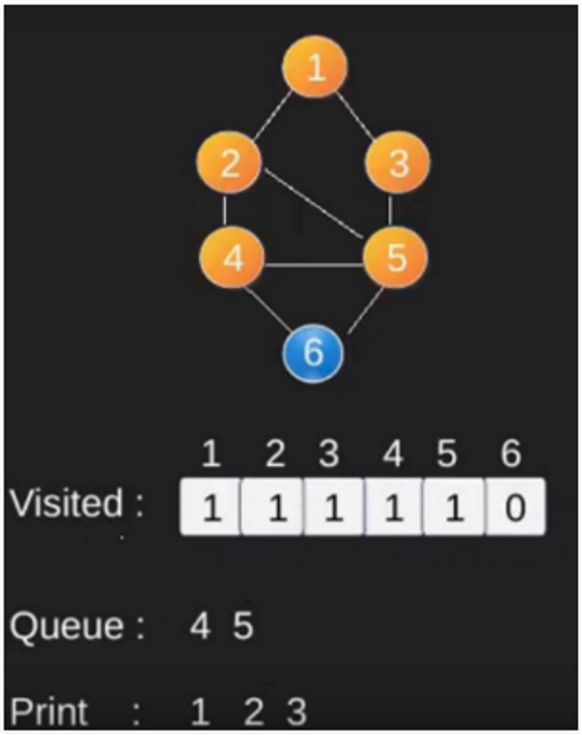


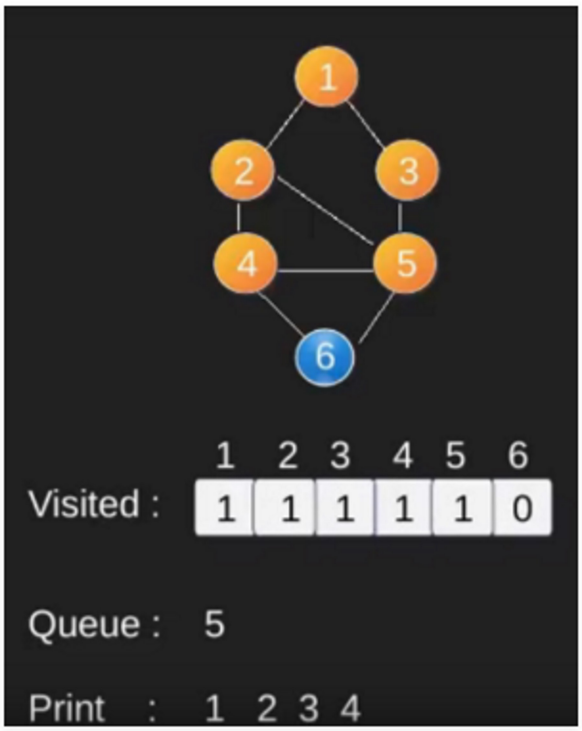


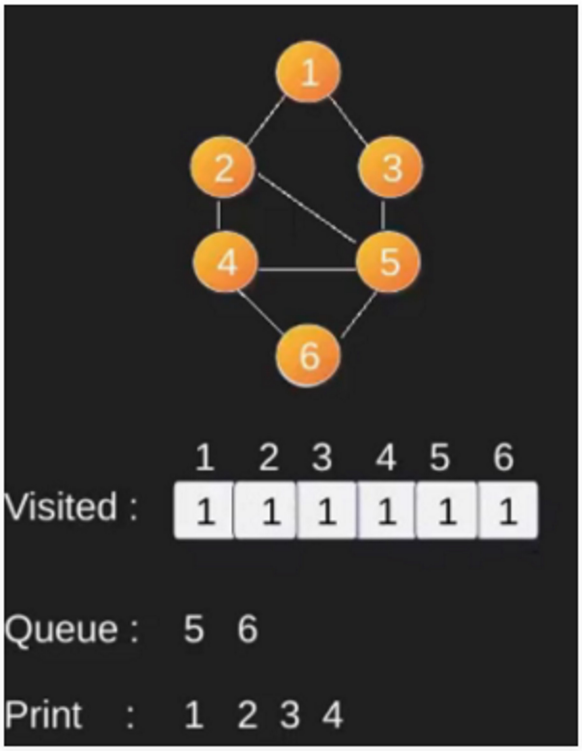

1
2
3
4
5
6
1 2 3 4 5 6
Visited :
1 0 0 0 0 0
Queue :
Print : 1

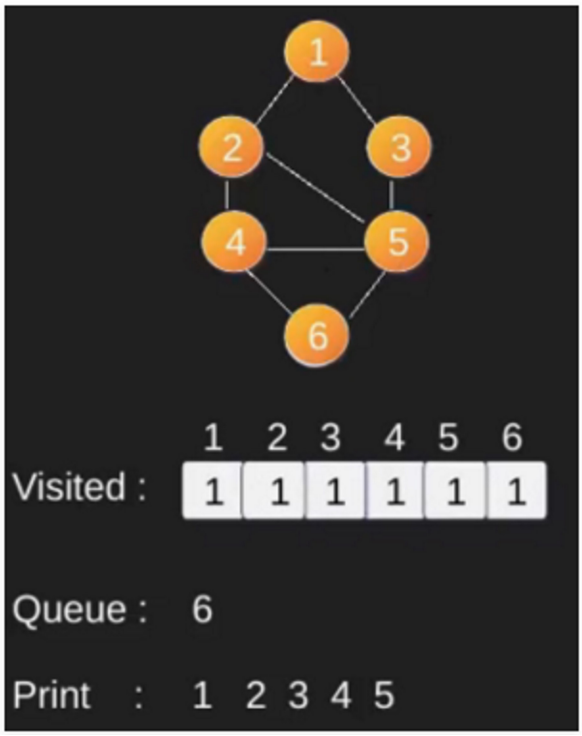

1
2
3
4
5
6
1 2 3 4 5 6
Visited :
1 1 1 0 0 0
Queue : 2 3
After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes
Print : 1

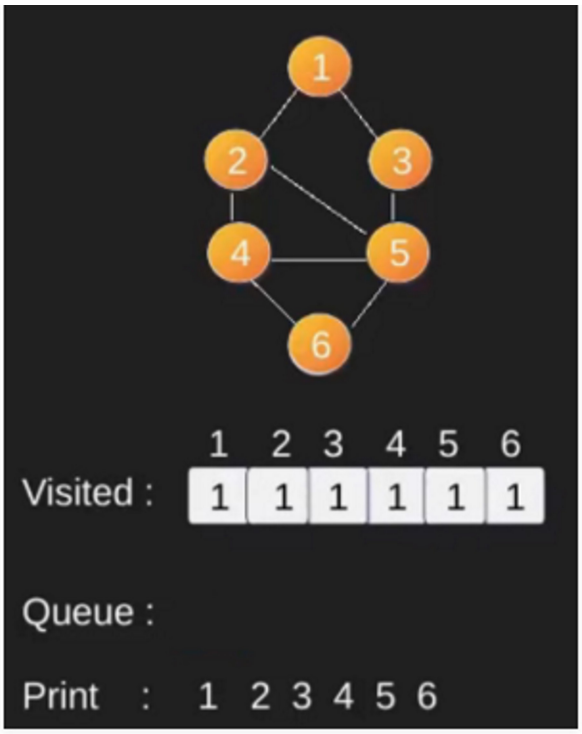

1
2
3
4
5
6
1 2 3 4 5 6
Visited :
1 1 1 1 1 0
Queue : 3 4 5
Print : 1 2


1
2
3
4
5
6
1 2 3 4 5 6
Visited :
1 1 1 1 1 0
Queue : 4 5
Print : 1 2 3









Note that the above method traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph).

6. Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find the shortest paths from source to all vertices in the given graph. It is a greedy approach.

Algorithm:

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While sptSet doesn't include all vertices:

- Pick a vertex u which is not there in sptSet and has minimum distance value.
- Include u to sptSet.
- Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of a distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

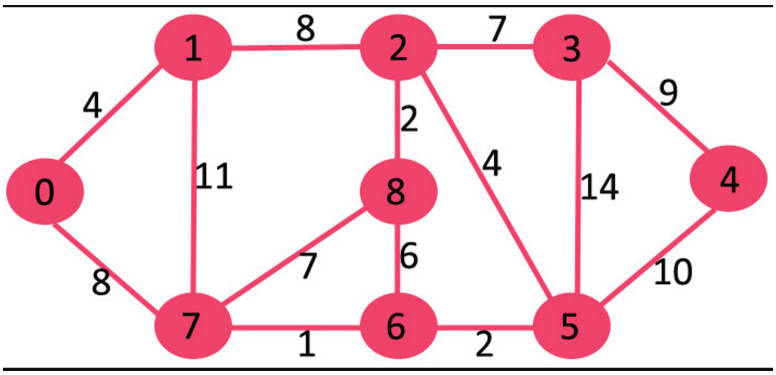
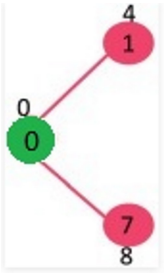
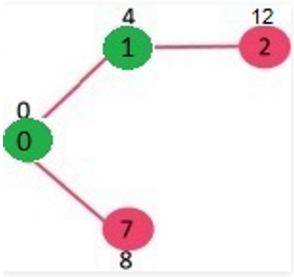


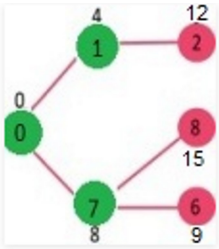
Illustration:

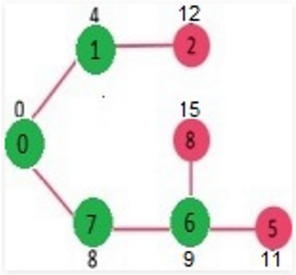
1.

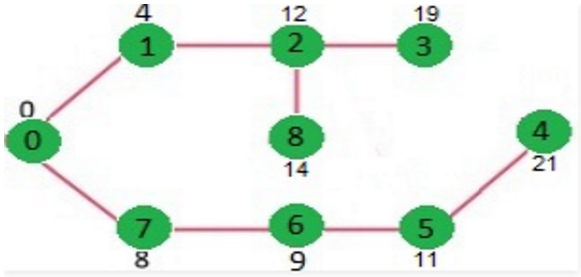


2.



3. 
4
12
1
2
0
0
8
15
7
6
8
9

4. 
4
12
1
2
0
0
15
8
7
6
5
8
9
11

5. 
4
12
19
1
2
3
0
0
8
4
14
21
7
6
5
8
9
11