Graph

Introduction

- Graph is a collection of nodes and edges.
- Edges connect two nodes.

Types

Unidirected and Directed

In directed the edges point from one node to another while in unidirected the edges have no particular direction.

Weighted and Unweighted

In weighted, the edges contain a value/weights whose significance can vary from question to question.

Cyclic and Acyclic

In a cyclic graph, we can come back to a node without visiting any edge twice.

Disconnected and Connected

In a connected graph, you can reach any node from any other node, through a sequence of connected edges.

Note: Tree is a connected acyclic graph.

Representation of Graph

Adjacency matrix

A graph with N nodes has an adjacency matrix M of dimension N x N, with M[i][j] = 1, if there is an edge directed from node i to j. If M[i][j] = M[j][i] = 1, then there is an undirected edge connecting i and j.

Adjacency List

Every node has a list of nodes, which denotes a direct edge from that node to every other node in the list.

Traversals

Breadth First Search (BFS)

- Also called level order traversal.
- Travel nodes in a level order fashion.

Pseudo Code:

```
function(graph):
    N = length of graph
    vis = boolean array of length N with false
    for source from 0 to N - 1:
        if vis[source]:
            continue
        queue = Queue()
        append source to queue
        while queue is not empty:
            beg = queue front
            pop front from queue
            print beg
            for next in graph[beg]:
                if not vis[next]:
                    vis[next] = true
                    append next to queue
```

- Time complexity: O(N + M)
- Space complexity: O(N)

Depth First Search (DFS)

Simply visit nodes until you can’t find any other node which is already not visited

Pseudo Code:

```
dfs(graph, visited, source):
    visited[source] = true
    print source
    for next in graph[source]:
        if not visited[next]:
            visited[next] = true
            dfs(graph, visited, next)
```

- Time complexity: O(N + M)
- Space complexity: O(N)

Multi Source BFS

In this we start our BFS from various starting nodes.

Question - 1 (Rotten Oranges)

Given a matrix mat of dimensions NxM, with following values:

1. 0 -> empty
2. 1 -> fresh
3. 2 -> rotten

Every minute, any fresh orange adjacent to a rotten orange gets rotten. Find the minimum time when all oranges get rotten. If not possible return -1.

Solution:
We can do the following:

1. Traverse the matrix and put every rotten orange coordinate in the queue as the source.
2. Perform BFS while maintaining a time matrix.
3. Find the maximum time for all rotten oranges.

Pseudo Code:

```
function(mat):
    N = length of mat
    M = length of mat[0]
    time = NxM matrix with values -1
    queue = Queue()
    for i from 0 to N - 1:
        for j from 0 to M - 1:
            if mat[i][j] == 2:
                append (i, j) in queue
                time[i][j] = 0

    while queue is not empty:
        (x, y) = front of queue
        pop front of queue
        for new_x, new_y in all 4 possible directions of (x, y):
            if time[new_x][new_y] == -1:
                append (new_x, new_y) in queue
                time[new_x][new_y] = time[x][y] + 1
                if mat[new_x][new_y] == 1:
                    mat[new_x][new_y] = 2

    ans = 0
    for i from 0 to N - 1:
        for j from 0 to M - 1:
            if mat[i][j] == 1:
                return -1
            if mat[i][j] == 2:
                ans = max(ans, time[i][j])

    return ans
```

Bipartite Graph

If we can divide the nodes into two sets such that no two nodes in the same set are adjacent to each other.
We can find if the given graph is a bipartite graph by coloring it into 2 colors.

Pseudo Code:

```
bool dfs(u, color):
    result = true
    for v neighbor of u:
        if color[v] == -1:
            color[v] = 1 - color[u]
            result = result or dfs(v, color)
        else if color[v] == color[u]:
```

```
            result = false
    return result

function(graph):
    N = node count in graph
    color = Array(N, -1)
    color[0] = 0
    return dfs(0, color)
```

Topological Sorting

1. Degree: No. of connections of a node.
2. Indegree: No. of incoming edges of a node.
3. Outdegree: No. of outgoing edges of a node.
4. Topological sort is used for dependency resolution.
5. We need to sort the nodes in such a way that the nodes which come earlier cannot be reached by nodes with come later by a sequence of directed edges.

We can use BFS to find a topological sorting.

Pseudo Code:

```
function(graph):
    N = node count in graph
    queue = Queue()
    indegree = array of indegrees of each node
    cnt = 0
    for i from 0 to N - 1:
        if indegree[i] == 0:
            append i in queue

    while queue is not empty:
        x = front of queue
        print(x)
        cnt = cnt + 1
        remove front of queue
        for y in graph[x]:
            indegree[y] = indegree[y] - 1
            if indegree[y] == 0:
                append y in queue

    if cnt != N:
        cyclic graph found
```

Minimum Spanning Tree (MST)

Given a connected weighted graph, a MST is a subgraph of it such that all nodes are connected and the sum of weights of edges is minimum.

Krushkal’s Algorithm

Sort the edges according to the weights, and choose edges in ascending order if they are not already in a connected component, and add the weight to the answer.

We need to use disjoin set union data structure to quickly find if two nodes are in the same connected component.

DSU

1. Initially each node is a tree itself.
2. We connect two trees when we add an edge.
3. To find if two nodes are in the same tree, we check for the root of the tree.
4. We can apply path compression to further optimize the solution.

```
parent = [] storing parent of each node, initially parent[x] = x

find(x, parent):
    while x != parent[x]:
        x = parent[x]
    return x

union(x, y):
    root_x = find(x)
    root_y = find(y)
    if root_x != root_y:
        parent[root_x] = root_y
            or
        parent[root_y] = root_x

find_with_comp(x):
    if x == parent[x]:
        return x
    parent[x] = find_with_com(parent[x])
```

Prim’s Algorithm

1. Add all edge options in a min heap, starting with a certain node.
2. Choose the best option.
3. Add newly discovered edges in the min heap.
4. Repeat from 2.

Shortest Path Algorithms

Dijkstra’s Algorithm

1. Used for non-negative weighted graphs.
2. Find the shortest distance of all other nodes from a particular node.

Pseudo Code:

```
function(graph, source):
    N = node count in graph
    heap = MinHeap()
    dist[N] = initialized to MAX
    append (0, source) to heap
    dist[source] = 0
    while heap is not empty:
        (cur_dist, x) = top of heap
        pop heap
        if cur_dist == dist[x]:
            for (y, w) in graph[x]:
                new_dist = cur_dist + w
                if dist[y] > new_dist:
                    dist[y] = new_dist
                    append (new_dist, y) to heap
```

- Time complexity: E * log(V)
- Space complexity: O(V)

Bellman-Ford Algorithm

1. Can detect negative cycles.
2. Find the shortest distance of all other nodes from a particular node.

Pseudo Code:

```
function(graph, source):
    dist[N] = initialized to MAX
    dist[source] = 0
    for iteration from 1 to N - 1:
        for edge in list of edges:
            (u, v, w) = edge
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w

    check for negative cycle with another iteration
```

- Time complexity: O(V * E)
- Space complexity: O(V)

Floyd-Warshall Algorithm

1. Calculates shortest path between all possible node pairs.

Pseudo Code:

```
function(graph):
    N = node count of graph
    for k from 0 to N - 1:
        for u from 0 to N - 1:
            for v from 0 to N - 1:
                dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
```

- Time complexity: O(V^3)
- Space complexity: O(V)