Graph Theory

Graph theory was first proposed by `Leonhard Euler`.

The result from koenigsberg bridge problem returns that,
“ A graph that has nodes with an odd number of degrees more than 2. Then the graph will not support a path that travels all nodes using each edge once”

Some words for graph

- Node : The points where edges are connected
- Edge : The connection between nodes
- Complete Graph : All nodes are related to each other with unique edge
- Cycle : last node is connected to first node makes a graph cycle
- Wheel : if a node is connected to each node of a cycle then this is wheel
- Bipartite : If nodes can be aligned in 2 groups that do not have any edge in the same group. Then this is a bipartite graph.

Graph Representation

We can represent graphs in 2 ways.

a. Adjacency Matrix
b. Adjacency List

Adjacency Matrix

A matrix that shows connection / edge between 2 nodes is the adjacency matrix. If I have n nodes in graph I have to take an $n \times n$ matrix to represent the graph.

```
1 —------------------ 2 —--------------- 3
|                     |                  |
|                     |                  |
4—--------------------5                  6
```

If here we create adjacency matrix,

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	0	1	0
3	0	1	0	0	0	1
4	1	0	0	0	1	0
5	0	1	0	1	0	0
6	0	0	1	0	0	0

starting with 1 indexing each row and col reads out its number. The (x,y) coordinate in the matrix if - 1 represents x and y has an edge. But if 0 then it is supposed that x,y coordinates do not share an edge.

Problems of using Adjacency Matrix

- Takes a lot of memory. For a graph of n nodes, it takes n^2 space.
- There is no benefit if the number of edges is less or more.
- Can not show multi-edge in any way

Adjacency List

In the adjacency list each node represents a list and the list stores the information of nodes they are connected to. For the graph above, the adjacency list is,

1 → 2 , 4
2 → 1 , 5
3 → 2 , 6
4 → 1 , 5
5 → 2 , 4
6 → 3

Now the constraint is O(E) the number of edges. Now edges can be as much as n^2. In that case the matrix and list will give the same result time… The less amount of edge means less amount of memory will be in use.

Code for representing graph [using list]

```
class graph{
    public:
    int v;
    vector<int> *adj;
    graph(int v){
        this->v = v;
        adj = new vector<int>[v];
    }

    void addEdge(int u, int v){
        adj[u].push_back(v);
        adj[v].push_back(u); //for undirected graph
    }

    void printGraph(){
        for(int i=0; i<v; i++){
            cout << i << " -> ";
            for(int j=0; j<adj[i].size(); j++){
                cout << adj[i][j] << " ";
            }
            cout << endl;
        }
    }
};
```

Breadth First Search (BFS)

This technique is used to traverse through the graph and find the shortest path between 2 nodes. **Only works on unweighted graphs.** We can use BFS to find an element in a graph. In BFS we search first a node and then push all its adjacent nodes to a queue and then popping one by one from the queue we look at everyone's adj nodes. We keep track of visited nodes by pushing to an array.

```
bool bfs(graph g, int source, int destination){
    queue<int> q;
    int level[100];

    init(level, 100, -1);

    q.push(source);

    while(!q.empty()){
```

```
        int u = q.front();
        q.pop();

        for(int i=0; i < g.adj[u].size(); i++){
            int v = g.adj[u][i];

            if(v == destination){
                return true;
            }

            if(level[v] == -1){
                level[v] = 1;
                q.push(v);
            }
        }
    }

    return false;
}
```

Finding Shortest Path using BFS

Finding the shortest path using bfs is a very easy task as it may differ only a line or two from the search purpose BFS. Using BFS we can,

- Find the minimum level distance from a node to another
- Shortest path finding from any node to another node.
- Bi-coloring a graph

Minimum Level Distance

```
int shortestPath(int source, int destination, graph g){
    int level[100];
    init(level, 100, -1);
    queue<int> q;

    q.push(source);
    level[source] = 0;

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i<g.adj[u].size(); i++){
```

```
            int v = g.adj[u][i];
            if(level[v] == -1){
                level[v] = level[u] + 1;
                q.push(v);
            }
        }
    }
    return level[destination];
}
```

Shortest Path

To find the shortest path I have to first do 2 things.

1. Find the shortest path by finding levels.
2. Labeling the node parents with a `previous array`

```
int previous[100];

void ShortestPath(graph g, int source, int destination){
    queue<int> q;
    int level[100];

    init(level, 100, -1);

    q.push(source);

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i < g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(level[v] == -1){
                level[v] = level[u] + 1;
                previous[v] = u;
                q.push(v);
            }
        }
    }

    cout << level[destination] << endl;
}
```

Now after finding the shortest path we can code the recursive function to print the path.

```
void printPath(int source, int destination){
    if(source == destination){
        cout << source;
    }
    else{
        printPath(source, previous[destination]);
        cout << " -> " << destination;
    }
}
```

Bi-Coloring

If we can color each node of a graph from 2 colors that no adjacent nodes share same color then ce can say “ *This Graph is Bi-Colorable* ”

```
bool isBicolorable(graph g, int source){
    int color[100];
    init(color, 100, -1);
    queue<int> q;

    q.push(source);
    color[source] = 1;

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(color[v] == -1){
                color[v] = 1 - color[u];
                q.push(v);
            }
            else if(color[v] == color[u]){
                return false;
            }
        }
    }
    return true;
}
```

Depth First Search (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It’s similar to the Depth First Traversal of a tree, but unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, DFS uses a boolean-visited array.

DFS can be used in,

- Detecting Cycle/loop in a graph
- Finding if a path exists or not
- If path exists, showing path
- Test for Bipartite Graph
- Finding strongly connected components

Finding Path using DFS

```
void dfs(graph g, int source, int destination){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;

    s.push(source);
    visited[source] = 1;

    while(!s.empty()){
        int u = s.top();
        s.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
            }
        }
    }
    if(visited[destination] == 1){
        cout << "Path exists" << endl;
    }
    else{
        cout << "Path does not exist" << endl;
    }
}
```

Printing the path

Now with a little of modification just like in BFS we did, we can make a recursive function that prints the parent of a node. By that, the path will be printed. We will again be needing a parent array.

```
int parent[100];

void dfs(graph g, int source, int destination){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;
    s.push(source);
    visited[source] = 1;
    while(!s.empty()){
        int u = s.top();
        s.pop();
        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
                parent[v] = u;
            }
        }
    }

    if(visited[destination] == 1){
        cout << "Path exists" << endl;
    }
    else{
        cout << "Path does not exist" << endl;
    }
}


void printPath(int source, int destination){
    if(source == destination){
        cout << source;
    }
    else{
        printPath(source, parent[destination]);
        cout << " -> " << destination;
    }
}
```

Detecteing Loop/Cycle in a graph

For undirecred graph we can check for a cycle like this,

```
int parent[100];

bool findCycle(graph g, int source){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;

    s.push(source);
    visited[source] = 1;

    while(!s.empty()){
        int u = s.top();
        s.pop();
        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
                parent[v] = u;
            }
            else if(visited[v] == 1 && parent[u] != v){
                return true;
            }
        }
    }

    return false;
}
```

For directed graph the process is little different. Here we have to keep track of nodes those are fully processed.

```
bool findCycleDG(graph g, int source) {
    int visited[100];
    int parent[100];
    init(visited, 100, 0);
    init(parent, 100, -1);

    stack<int> s;
    s.push(source);

    while (!s.empty()) {
```

```
        int u = s.top();
        s.pop();
        visited[u] = 1;

        for (int i = 0; i < g.adj[u].size(); i++) {
            int v = g.adj[u][i];
            if (visited[v] == 0) {
                s.push(v);
                parent[v] = u;
            } else if (visited[v] == 1) {
                return true;
            }
        }

        visited[u] = 2;
    }

    return false;
}
```
