DFS

```
#include<bits/stdc++.h>
using namespace std;

const int MAX_VERTICES = 10;
const int MAX_EDGES = 10;

class Graph{

    int adj[MAX_VERTICES][MAX_EDGES];
    int count[MAX_VERTICES];
    int vertices;
    bool isDirected;

    public:
    Graph(int v,bool directed){
        vertices = v;
        isDirected = directed;
        for(int i=0;i<vertices;i++){
            count[i] = 0;
            for(int j=0;j<MAX_EDGES;j++){
                adj[i][j] = -1;
            }
        }
    }


    void addEdge(int u,int v){
        adj[u][count[u]] = v;
        count[u]++;
        if(!isDirected){
            adj[v][count[v]] = u;
            count[v]++;
        }
    }

    void DFS(int start,bool visited[]){
        visited[start] = true;
        cout<<start<<" ";
        for(int i=0;i<count[start];i++){
            int neighbor = adj[start][i];
            if(!visited[neighbor]){
                DFS(neighbor,visited);
            }
        }
    }
```

```
    void DFS_Traversal(int start){
        bool visited[MAX_VERTICES] = {false};
        cout<<"DFS Traversal starting from vertex "<<start<<": ";
        DFS(start,visited);
        cout<<endl;
    }

};

int main() {
    int vertices, edges, directed;
    cout << "Enter number of vertices: ";
    cin >> vertices;

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Is the graph directed? (1 = Yes, 0 = No): ";
    cin >> directed;

    Graph g(vertices, directed);

    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    g.DFS_Traversal(0);

    return 0;
}
```

BFS

```cpp
#include<bits/stdc++.h>
using namespace std;

const int MAX_VERTICES = 10;
const int MAX_EDGES = 10;

class Graph{

    int adj[MAX_VERTICES][MAX_EDGES];
    int count[MAX_VERTICES];
    int vertices;
    bool isDirected;

    public:
    Graph(int v,bool directed){
        vertices = v;
        isDirected = directed;
        for(int i=0;i<vertices;i++){
            count[i] = 0;
            for(int j=0;j<MAX_EDGES;j++){
                adj[i][j] = -1;
            }
        }
    }

    void addEdge(int u,int v){
        adj[u][count[u]] = v;
        count[u]++;
        if(!isDirected){
            adj[v][count[v]] = u;
            count[v]++;
        }
    }
```

```
    void BFS_Traversal(int start){
        bool visited[MAX_VERTICES] = {false};
        int queue[MAX_VERTICES];
        int front = 0, rear = 0;
        visited[start] = true;
        queue[rear++] = start;

        cout<<"BFS Traversal starting from vertex "<<start<<": ";

        while(front<rear){
            int current = queue[front++];
            cout<<current<<" ";

            for(int i=0;i<count[current];i++){
                int neighbor = adj[current][i];
                if(!visited[neighbor]){
                    visited[neighbor] = true;
                    queue[rear++] = neighbor;
                }
            }
        }
        cout<<endl;
    }
};

int main() {
    int vertices, edges, directed;
    cout << "Enter number of vertices: ";
    cin >> vertices;

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Is the graph directed? (1 = Yes, 0 = No): ";
    cin >> directed;

    Graph g(vertices, directed);

    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    g.BFS_Traversal(0);

    return 0;
}
```

Kruskal Algorithm Implementation

```cpp
#include <iostream>
using namespace std;

const int MAX_VERTICES = 10;
const int MAX_EDGES = 20;

struct Edge {
    int u, v, weight;
};

class KruskalMST {
    int parent[MAX_VERTICES];
    Edge edges[MAX_EDGES];
    int vertices, totalEdges;

public:
    KruskalMST(int v, int e) {
        vertices = v;
        totalEdges = e;
    }

    void inputEdges() {
        cout << "Enter edges (u v weight):" << endl;
        for (int i = 0; i < totalEdges; i++) {
            cin >> edges[i].u >> edges[i].v >> edges[i].weight;
        }
    }

    void sortEdges() {
        for (int i = 0; i < totalEdges - 1; i++) {
            for (int j = 0; j < totalEdges - i - 1; j++) {
                if (edges[j].weight > edges[j + 1].weight) {
                    Edge temp = edges[j];
                    edges[j] = edges[j + 1];
                    edges[j + 1] = temp;
                }
            }
        }
    }

    int find(int vertex) {
        if (parent[vertex] == -1)
            return vertex;
        return find(parent[vertex]);
    }
```

```
    void unionSet(int u, int v) {
        int setU = find(u);
        int setV = find(v);
        if (setU != setV) {
            parent[setU] = setV;
        }
    }

    void buildMST() {
        for (int i = 0; i < vertices; i++)
            parent[i] = -1;

        sortEdges();

        int mstWeight = 0;
        cout << "\nEdges in Minimum Spanning Tree:\n";

        int edgeCount = 0;
        for (int i = 0; i < totalEdges && edgeCount < vertices - 1; i++) {
            int u = edges[i].u;
            int v = edges[i].v;
            int w = edges[i].weight;

            if (find(u) != find(v)) {
                cout << u << " - " << v << " : " << w << endl;
                mstWeight += w;
                unionSet(u, v);
                edgeCount++;
            }
        }

        cout << "Total weight of MST: " << mstWeight << endl;
    }
};

int main() {
    int v, e;
    cout << "Enter number of vertices: ";
    cin >> v;
    cout << "Enter number of edges: ";
    cin >> e;

    KruskalMST mst(v, e);
    mst.inputEdges();
    mst.buildMST();

    return 0;
}
```

Prim's Algorithm Implementation

```
#include <iostream>
using namespace std;

const int MAX_VERTICES = 10;
const int INF = 1e9;

class PrimMST {
    int adj[MAX_VERTICES][MAX_VERTICES];
    int vertices;

public:
    PrimMST(int v) {
        vertices = v;
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                adj[i][j] = 0;
            }
        }
    }

    void inputEdges(int edges) {
        for (int i = 0; i < edges; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            adj[u][v] = w;
            adj[v][u] = w;
        }
    }

    int selectMinKey(int key[], bool visited[]) {
        int min = INF, minIndex = -1;
        for (int i = 0; i < vertices; i++) {
            if (!visited[i] && key[i] < min) {
                min = key[i];
                minIndex = i;
            }
        }
        return minIndex;
    }
```

```
    void buildMST() {
        int key[MAX_VERTICES];
        int parent[MAX_VERTICES];
        bool visited[MAX_VERTICES];

        for (int i = 0; i < vertices; i++) {
            key[i] = INF;
            visited[i] = false;
        }

        key[0] = 0;
        parent[0] = -1;

        for (int count = 0; count < vertices - 1; count++) {
            int u = selectMinKey(key, visited);
            visited[u] = true;

            for (int v = 0; v < vertices; v++) {
                if (adj[u][v] && !visited[v] && adj[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = adj[u][v];
                }
            }
        }


        int totalWeight = 0;
        for (int i = 1; i < vertices; i++) {
            cout << parent[i] << " - " << i << " : " << adj[i][parent[i]] << endl;
            totalWeight += adj[i][parent[i]];
        }
        cout << "Total weight of MST: " << totalWeight << endl;
    }
};

int main() {
    int v, e;
    cout << "Enter number of vertices: ";
    cin >> v;
    cout << "Enter number of edges: ";
    cin >> e;

    PrimMST mst(v);
    cout << "Enter edges (u v weight):" << endl;
    mst.inputEdges(e);
    cout << "\nEdges in MST:\n";
    mst.buildMST();

    return 0;
}
```

Dijktra Algorithm Implementation

```cpp
#include <iostream>
using namespace std;

const int MAX_VERTICES = 100;
const int INF = 1e9;

class Dijkstra {
    int graph[MAX_VERTICES][MAX_VERTICES];
    int vertices;
    bool isDirected;

public:
    Dijkstra(int v, bool directed) {
        vertices = v;
        isDirected = directed;
        for (int i = 0; i < vertices; i++)
            for (int j = 0; j < vertices; j++)
                graph[i][j] = 0;
    }

    void inputEdges(int edges) {
        cout << "Enter edges (from to weight):\n";
        for (int i = 0; i < edges; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            graph[u][v] = w;
            if (!isDirected) {
                graph[v][u] = w;
            }
        }

    }

    int selectMinVertex(int dist[], bool visited[]) {
        int min = INF;
        int index = -1;

        for (int i = 0; i < vertices; i++) {
            if (!visited[i] && dist[i] < min) {
                min = dist[i];
                index = i;
            }
        }
        return index;
    }
```

```
    void dijkstra(int src) {
        int dist[MAX_VERTICES];
        bool visited[MAX_VERTICES];

        for (int i = 0; i < vertices; i++) {
            dist[i] = INF;
            visited[i] = false;
        }

        dist[src] = 0;

        for (int count = 0; count < vertices - 1; count++) {
            int u = selectMinVertex(dist, visited);
            if (u == -1) break;
            visited[u] = true;

            for (int v = 0; v < vertices; v++) {
                if (graph[u][v] && !visited[v] && dist[u] + graph[u][v] < dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                }
            }
        }

        cout << "Shortest distances from source " << src << ":\n";
        for (int i = 0; i < vertices; i++) {
            cout << "To " << i << " : " << (dist[i] == INF ? -1 : dist[i]) << endl;
        }
    }
};

int main() {
    int v, e;
    cout << "Enter number of vertices: ";
    cin >> v;
    cout << "Enter number of edges: ";
    cin >> e;

    char choice;
    cout << "Is the graph directed? (y/n): ";
    cin >> choice;
    bool isDirected = (choice == 'y' || choice == 'Y');

    Dijkstra d(v, isDirected);
    d.inputEdges(e);

    int src;
    cout << "Enter source vertex: ";
    cin >> src;

    d.dijkstra(src);

    return 0;
}
```

Detect a Cycle

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_VERTICES = 10;
const int MAX_EDGES = 10;

class Graph {
    int adj[MAX_VERTICES][MAX_EDGES];
    int count[MAX_VERTICES];
    int vertices;
    bool isDirected;

public:
    Graph(int v, bool directed) {
        vertices = v;
        isDirected = directed;
        for (int i = 0; i < vertices; i++) {
            count[i] = 0;
            for (int j = 0; j < MAX_EDGES; j++) {
                adj[i][j] = -1;
            }
        }
    }

    void addEdge(int u, int v) {
        adj[u][count[u]++] = v;
        if (!isDirected) {
            adj[v][count[v]++] = u;
        }
    }
```

```
    bool hasCycleUtil(int current, bool visited[], int parent) {
        visited[current] = true;

        for (int i = 0; i < count[current]; i++) {
            int neighbor = adj[current][i];

            if (!visited[neighbor]) {
                if (hasCycleUtil(neighbor, visited, current)) {
                    return true;
                }
            }
            else if (neighbor != parent) {
                // A visited node which is not the parent -> cycle found
                return true;
            }
        }

        return false;
    }

    void detectCycle() {
        bool visited[MAX_VERTICES] = {false};

        for (int i = 0; i < vertices; i++) {
            if (!visited[i]) {
                if (hasCycleUtil(i, visited, -1)) {
                    cout << "Cycle detected in the graph." << endl;
                    return;
                }
            }
        }

        cout << "No cycle found in the graph." << endl;
    }
};
```

```cpp
int main() {
    int vertices, edges, directed;
    cout << "Enter number of vertices: ";
    cin >> vertices;

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Is the graph directed? (1 = Yes, 0 = No): ";
    cin >> directed;

    Graph g(vertices, directed);

    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    g.detectCycle();

    return 0;
}
```