PROGRAM 3(B)

AIM:-Compute the transitive closure of a given directed graph using Warshall's algorithm.

Source code:-

```
#include<iostream>
using std::cout,std::endl,std::cerr;
#include<vector>
using std::vector;
#include<string>
using std::string,std::getline;
#include <unordered_map>
using std::unordered_map,std::make_pair;
#include<fstream>
using std::ifstream;
#include<sstream>
using std::stringstream;
#include<iomanip>
using std::setw;


vector<vector<bool>> & transitive_closure(vector<vector<bool>> & graph){
    for(unsigned k=0;k<graph.size();++k){
        for(unsigned row=0;row<graph.size();++row){
            for(unsigned col=0;col<graph.size();++col){
                graph[row][col]=graph[row][col] || (graph[row][k] && graph[k][col]);
            }
        }
    }
    return graph;
}

template <typename NodeType>
vector<vector<bool>> CreateGraph(ifstream & input,unordered_map<unsigned,NodeType>&
NumIndices,unordered_map<NodeType,unsigned> &KeyIndices ){
    unordered_map<NodeType,vector<NodeType>> map;
    unsigned ctr=0;
     string line;
    while (getline(input,line))
    {
        stringstream tokens(line);
        //taking input as adjacency list
        //filling maps containing conversions from given type to unsigned and vice-versa
        NodeType Key;
        tokens>>Key;
        if(KeyIndices.emplace(make_pair(Key,ctr)).second==true){
        NumIndices.emplace(make_pair(ctr,Key));
        ++ctr;}
        map.emplace(make_pair( Key,vector<NodeType>(0)));
        NodeType value;
        while(tokens>>value){
```

```
        if(KeyIndices.emplace(make_pair(value,ctr)).second==true){
        NumIndices.emplace(make_pair(ctr,value)).second==true;
        ++ctr;};
        map[Key].emplace_back(value);

        }
    }
    // creating adjacency matrix
    vector<vector<bool>> graph(map.size(),vector<bool>(map.size(),false));
    for(const auto &row:map){
        for (const auto & col:row.second){
            graph[KeyIndices.at(row.first)][KeyIndices.at(col)]=true;
        }
    }
    return graph;
}

int main(int argc, char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return EXIT_FAILURE;
    }
    ifstream input(argv[1]);
    if (!input){
        cerr<<"error opening this file";
        return EXIT_FAILURE;
    }
    unordered_map<unsigned,string> NumIndices;
    unordered_map<string,unsigned> KeyIndices;

   auto graph=CreateGraph(input,NumIndices,KeyIndices);
    input.close();

    transitive_closure(graph);

     cout<<setw(12)<<" ";
    for(auto i=0;i<graph.size();++i){
        cout<<setw(10)<<NumIndices[i];}
        cout<<endl;
        for(auto i=0;i<graph.size();++i){
            cout<<setw(12)<<NumIndices[i];
        for(auto j=0;j<graph.size();++j){
            cout<<setw(10)<<graph[i][j];
        }
        cout<<endl;
    }
}
```

output:-

input file

```
Delhi    Mumbai Kolkata Chennai Dubai London
Mumbai    Delhi Kolkata Chennai Dubai AbuDhabi Bangkok Singapore
Kolkata    Delhi Mumbai Chennai Dubai Bangkok Singapore
Chennai    Delhi Mumbai Kolkata Dubai Singapore KualaLumpur
Dubai    Delhi Mumbai Kolkata Chennai London Paris NewYork
London    Delhi Dubai Paris NewYork Toronto
Paris    Dubai London NewYork Toronto
NewYork    Dubai London Paris Toronto LosAngeles
Toronto    London Paris NewYork LosAngeles
LosAngeles    NewYork Toronto
AbuDhabi    Mumbai Singapore KualaLumpur
Bangkok    Mumbai Kolkata Singapore KualaLumpur
Singapore    Mumbai Kolkata Chennai AbuDhabi Bangkok KualaLumpur Sydney
KualaLumpur    Chennai AbuDhabi Bangkok Singapore Sydney
Sydney    Singapore KualaLumpur
```

terminal -output

```
D:\Code Domain\DAA (c++)>.\Warshall.exe digraph-string.txt
              Delhi    Mumbai   Kolkata   Chennai     Dubai    London  AbuDhabi   Bangkok SingaporeKualaLumpur     Paris   NewYork   TorontoLosAngeles    Sydney
      Delhi         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
     Mumbai         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
    Kolkata         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
    Chennai         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
      Dubai         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
     London         0         0         0         0         0         1         0         0         0         0         1         1         1         1         0
   AbuDhabi         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
    Bangkok         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
  Singapore         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
KualaLumpur         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
      Paris         0         0         0         0         0         1         0         0         0         0         1         1         1         1         0
    NewYork         0         0         0         0         0         1         0         0         0         0         1         1         1         1         0
    Toronto         0         0         0         0         0         1         0         0         0         0         1         1         1         1         0
 LosAngeles         0         0         0         0         0         1         0         0         0         0         1         1         1         1         0
     Sydney         1         1         1         1         1         1         1         1         1         1         1         1         1         1         1
```

PROGRAM -5

AIM:- From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijikstra’s algorithm.

Source code:-

```
#include <iostream>
using std::cout,std::endl,std::cerr;
#include<unordered_map>
using std::unordered_map;
#include <string>
using std::string,std::getline;
#include<sstream>
using std::stringstream;
#include<fstream>
using std::ifstream;
#include <vector>
using std::vector;
#include<utility>
using std::pair, std::move;
#include<limits>
#include<queue>

template <typename T>
using graph= unordered_map<T,vector<pair<T,unsigned int>>>;

template<typename T>
struct vertex
{   vertex()=default;
    vertex(T n, unsigned int dist=std::numeric_limits<unsigned int>::max(),T p=T() , bool r=false):
            name(n),distance(dist),parent(p),removed(r){}
    T name;
    unsigned int distance=std::numeric_limits<unsigned int>::max();
    T parent= T();
    bool removed=false;
};


template <typename T>
unordered_map<T,vertex<T>> &dijkstra(const T & source,const graph<T> & G ,
unordered_map<T,vertex<T>>& vertices){
    //Initialisation

    for(const auto& element:G){
        vertices[element.first]=vertex<T>(element.first);
    }

    //setting the distance of source vertex to zero
    vertices[source].distance=0;

    //creating the priority queue
```

```
    std::priority_queue<pair<unsigned int, T>,vector<pair<unsigned
int,T>>,std::greater<pair<unsigned int,T>>> min_heap;
    min_heap.push({0,(source)});

    while(!min_heap.empty()){

        T current_min=min_heap.top().second;
        min_heap.pop();
        vertices[current_min].removed=true;

        // cout<<it->first<<'\t'<<it->second.name<<" "<<it->second.parent<<" "<<it-
>second.distance<<"\n";
        // ++it;

        for (const auto & element: (*G.find(current_min)).second){
            const auto & adj_vertex=element.first;
            if (vertices[adj_vertex].removed==true){ continue;}
            else {
                auto & old_dist=vertices[adj_vertex].distance;
                unsigned int new_dist=vertices[current_min].distance+element.second;
                if(new_dist < old_dist){
                    old_dist=new_dist;
                    vertices[adj_vertex].parent=current_min;
                    min_heap.push({new_dist,adj_vertex});
            }
            }

        }
    }
    return vertices;
}

template<typename T>
const graph<T>& create_graph(ifstream & input_file, graph<T>&G){
    string line;
    while(getline(input_file,line)){
        G.insert({line[0],vector<pair<T,unsigned int>>(0)});
        stringstream adj_vertices(line.substr(3));
        string adj_line;
        T name;
        unsigned int weight;
        while(getline(adj_vertices,adj_line,'}')){
            name=adj_line[2];
            stringstream distance(adj_line.substr(4));
            distance>>weight;
            G[line[0]].push_back({name,weight});
        }
    }
    return G;

}
```

```
int main(int argc, char **argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    ifstream input_file(argv[1]);
    if (!input_file){
        cerr<<"error opening this file";
        return -1;
    }
    graph<char> G;
    create_graph(input_file,G);
    input_file.close();

    unordered_map<char,vertex<char>> vertices;
    dijkstra('a',G,vertices);
    cout<<"node\t"<<"parent\t"<<"distance from source"<<endl;
    for(auto i= vertices.begin();i!=vertices.end();++i){
        cout<<i->first<<'\t'<<i->second.parent<<'\t'<<i->second.distance<<'\n';
    }

}
```

output:-

input file:-

```
a    {b,10},{c,20},{h,34}
b    {c,3},{a,10},{f,81}
c    {b,3},{d,17},{a,20}
d    {e,2},{c,17}
e    {o,8},{n,31},{d,2}
f    {n,15},{g,15},{b,81}
g    {n,12},{h,23},{f,15}
h    {a,34},{g,23}
i    {l,10},{j,8}
j    {k,5},{l,8}
k    {o,4},{l,5},{j,5}
l    {k,5},{i,10}
m    {n,13},{o,19}
n    {o,20},{m,13},{f,15},{g,12},{e,31}
o    {e,8},{m,19},{n,20},{k,4}
```

terminal output:-

```
PS D:\Code Domain\DAA (c++)> .\dijkstra.exe .\undirected-weighted-graph.txt
node    parent  distance from source
m       o       59
l       k       49
o       e       40
b       a       10
n       o       60
a               0
c       b       13
d       c       30
e       d       32
f       g       72
g       h       57
h       a       34
i       l       59
j       k       49
k       o       44
PS D:\Code Domain\DAA (c++)>
```

PROGRAM 7(A)

AIM:- Print all the nodes reachable from a given starting node in a digraph using BFS method.

Source code:-

```
#include <unordered_map>
using std::unordered_map;
#include<vector>
using std::vector;
#include<fstream>
using std::ifstream;
#include<sstream>
using std::stringstream;
#include<string>
using std::getline,std::string;
#include<iostream>
using std::cout,std::endl,std::cerr;
#include<queue>
using std::queue;
#include<unordered_set>
using std::unordered_set;
#include<stdexcept>

template<typename KeyType>
using graph=unordered_map<KeyType,vector<KeyType>>;

template<typename KeyType>
void BFS(const graph<KeyType> & digraph, const KeyType& source){
    queue<KeyType> to_process;
    unordered_set<KeyType> visited;
    to_process.push(source);
    visited.insert(source);
    cout<<"The nodes reachable from the source node are:-"<<endl;
    while (!to_process.empty())
    {
        auto processed=to_process.front();
        cout<<processed<<" ";
        to_process.pop();
        try{
        for (const auto & neighbour: digraph.at(processed)){
            if(visited.find(neighbour)==visited.end()){
                to_process.push(neighbour);
                visited.insert(neighbour);
            }
        }
    }
    catch(std::out_of_range){
        cout<<endl;
        cout<<processed<<" was the terminating node"<<endl;
        cout<<"The unreachable nodes are:- "<<endl;
        for(const auto & element:digraph){
```

```
            if(visited.find(element.first)==visited.end()){
                cout<<element.first<<" ";
            }
        }
        break;
    }
  }
}

int main(int argc, char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    ifstream input(argv[1]);
    if (!input){
        cerr<<"error opening this file";
        return -1;
    }

    //initialisations
    graph<char> digraph;
    char source;
    cout<<"enter the source node"<<endl;
    std::cin>>source;

    string line;
    while(getline(input,line)){
        digraph.insert({line[0],vector<char>(0)});
        stringstream tokens(line.substr(1));
        char keys;
        while(tokens>>keys){
            digraph[line[0]].push_back(keys);
        }

    }
    input.close();
   //driver
    BFS(digraph,source);
}
```

output

input file

```
a	b c d
b	c
c
e	d h
d	f g
f	g h i
i	j
h	i
g
```

terminal output

```
PS D:\Code Domain\DAA (c++)> .\BFS.exe .\digraph-char.txt
enter the source node
a
The nodes reachable from the source node are:-
a b c d f g h i j
j was the terminating node
The unreachable nodes are:-
e
PS D:\Code Domain\DAA (c++)>
```

PROGRAM 7(B)

AIM:-Check whether a given graph is connected or not using DFS method.

Source code:-

```
#include<iostream>
using std::cout,std::endl,std::cerr;
#include<string>
using std::string,std::getline;
#include<vector>
using std::vector;
#include<fstream>
#include<sstream>
#include<unordered_map>
using std::unordered_map;
#include<unordered_set>
using std::unordered_set;
#include<stack>
using std::stack;

template <typename VertexType>
using graph=unordered_map<VertexType,vector<VertexType>>;

template <typename VertexType>
vector<VertexType> isConnected(const graph<VertexType>& G){
    stack<VertexType> to_process;
    to_process.push(G.cbegin()->first);
    unordered_set<VertexType> visited;
    visited.insert(G.cbegin()->first);

    while(!to_process.empty()){
    VertexType current_vertex=to_process.top();
    to_process.pop();

    for(const auto & neighbour: G.at(current_vertex)){
        if(visited.find(neighbour)==visited.end()){
            to_process.push(neighbour);
            visited.insert(neighbour);
        }
    }
    }
    vector<VertexType>unVisited;
    if(visited.size()==G.size()){ return unVisited;}
    else{
        for(const auto & pair :G){
            if(visited.find(pair.first)==visited.end()){
                unVisited.push_back(pair.first);
            }
        }
        return unVisited;
    }
```

```
}

int main(int argc, char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    std::ifstream input(argv[1]);
    if (!input){
        cerr<<"error opening this file";
        return -1;
    }

    //initialisations
    graph<char> G;

    string line;
    while(getline(input,line)){
        G.insert({line[0],vector<char>(0)});
        std::stringstream tokens(line.substr(1));
        char keys;
        while(tokens>>keys){
            G[line[0]].push_back(keys);
        }

    }
    input.close();

    auto result=isConnected(G);
    if (result.empty()){
        cout<<"Congratulations! Given graph is connected!"<<endl;
    }
    else{
        cout<<"Sadly the given graph is not connected!"<<endl;
        cout<<"The unreachable vertices from '"<<G.begin()->first<<"' are:-"<<endl;
        for(const auto & vertex:result){
            cout<<vertex<<" ";
        }
    }
}
```

input file

```
a    b c h
b    c a f
c    b d a
d    e c
e    o n d
f    n g b
g    n h f
h    a g
i    l j
j    k l
k    o l j
l    k i
m    n o
n    o m f g e
o    e m n k
```

terminal output

```
PS D:\Code Domain\DAA (c++)> .\DFS .\undirected-unweighted-graph.txt
Congratulations! Given graph is connected!
PS D:\Code Domain\DAA (c++)>
```

PROGRAM 10

AIM:-Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm

Source code

```
#include <iostream>
using std::cout,std::endl,std::cerr;
#include<fstream>
using std::ifstream;
#include<sstream>
using std::stringstream;
#include<string>
using std::string,std::getline;
#include<unordered_map>
using std::unordered_map;
#include<vector>
using std::vector;
#include<utility>
using std::pair,std::make_pair;
#include<queue>
#include<limits>

template<typename KeyType>
using graph=unordered_map<KeyType,vector<pair<KeyType,unsigned>>>;

template<typename T>
struct vertex
{   vertex()=default;
   vertex(T n, unsigned int infinity=std::numeric_limits<unsigned int>::max(),T p=T() , bool
r=false):
         name(n),cost(infinity),parent(p),removed(r){}
   T name;
   unsigned int cost=std::numeric_limits<unsigned int>::max();
   T parent= T();
   bool removed=false;
};

template<typename KeyType>
const graph<KeyType> & MCST(const graph<KeyType>& G, graph<KeyType>& tree){

   unordered_map<KeyType,vertex<KeyType>> vertices;
   //initialisation (necessary)
   for(const auto& element:G){
      vertices[element.first]=vertex<KeyType>(element.first);
   }
   std::priority_queue<pair<unsigned int, KeyType>,vector<pair<unsigned
int,KeyType>>,std::greater<pair<unsigned int,KeyType>>> min_heap;
   auto first_element=vertices.begin();
```

```
    (first_element ->second).cost=0;
     (first_element ->second).removed=true;
    tree.emplace(make_pair(first_element->first,vector<pair<KeyType,unsigned>>(0)));
    for(const auto & neighbours:G.at(first_element->first)){
        vertices[neighbours.first].cost=neighbours.second;
        vertices[neighbours.first].parent=first_element->first;
        min_heap.push(make_pair(neighbours.second,neighbours.first));
    }

    while(tree.size()<G.size()){
        KeyType current_min=min_heap.top().second;
         min_heap.pop();

        if(!vertices.at(current_min).removed){
        tree.emplace(make_pair(current_min,vector<pair<KeyType,unsigned>>(0)));

tree.at(vertices.at(current_min).parent).emplace_back(make_pair(current_min,vertices.at(current_m
in).cost));
        vertices.at(current_min).removed=true;

         for (const auto & element: G.at(current_min)){
            const auto & adj_vertex=element.first;
            if (vertices[adj_vertex].removed==true){ continue;}
            else {
                auto & old_cost=vertices[adj_vertex].cost;
                unsigned int new_cost=element.second;
                if(new_cost < old_cost){
                    old_cost=new_cost;
                    vertices[adj_vertex].parent=current_min;
                    min_heap.push({new_cost,adj_vertex});
            }
            }

        }
        }
    }
    return tree;
}

template<typename KeyType>
const graph<KeyType> & FillGraph(ifstream & input, graph<KeyType> & G){
    string line;
    while (getline(input,line))
    {
     stringstream primary(line);
     KeyType Vertex;
     primary>>Vertex;
     vector<pair<KeyType,unsigned>> & neighbours=G[Vertex];
     string adj_vertex;
     while(primary>>adj_vertex){
        auto comma = adj_vertex.find_first_of(",");
```

```
        KeyType adj_vertex_name=adj_vertex.substr(0,comma);   //may need to change data type
here.
        unsigned weight=std::stoul(adj_vertex.substr(comma+1));
        neighbours.emplace_back(make_pair(adj_vertex_name,weight));
     }
    }
    return G;
}

int main(int argc,char**argv){
    if(argc!=2){
        cerr<<"Usage: "<<argv[0]<<"  name of input file";
        return -1;
    }
    ifstream input(argv[1]);
    if (!input){
        cerr<<"error opening this file";
        return -1;
    }
    graph <string> G;
    FillGraph(input,G);
    input.close();

    decltype(G) tree;
    MCST(G,tree);
    for (const auto & elements:tree)
    {
        cout<<elements.first<<'\t';
        for(const auto & children:elements.second){
            cout<<children.first<<","<<children.second<<"  ";
        }
        cout<<endl;
    }
}
```

input file

```
a    b,10 c,20 h,34
b    c,3 a,10,f,81
c    b,3 d,17 a,20
d    e,2 c,17
e    o,8 n,31 d,2
f    n,15 g,15 b,81
g    n,12 h,23 f,15
h    a,34 g,23
i    l,10 j,8
j    k,5 l,8
k    o,4 l,5 j,5
l    k,5 i,10
m    n,13 o,19
n    o,20 m,13 f,15 g,12 e,31
o    e,8 m,19 n,20 k,4
```

terminal output

```
PS D:\Code Domain\DAA (c++)> .\prim undirected_weighted_graph.txt
h
a
m       n,13  o,19
i
g       h,23
e       d,2
j
l       i,10
f
o       k,4  e,8
c       b,3
b       a,10
n       g,12  f,15
k       j,5  l,5
d       c,17
PS D:\Code Domain\DAA (c++)>
```