Discrete Mathematics

# Semester Project

# Project is made by:

Muhammad Hamza 01-134182-033

Maham Khan 01-134182-022

## Prim’s Algorithm

In [computer science](https://en.wikipedia.org/wiki/Computer_science), Prim's (also known as Jarník's) algorithm is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) that finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [weighted](https://en.wikipedia.org/wiki/Weighted_graph) [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph). This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the [edges](https://en.wikipedia.org/wiki/Graph_theory) in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

## Implementation in C++

#include <set>

#include <vector>

#include <iostream>

/\*

\*\* This program is a basic implementation of Prim's algorithm to find minimal

\*\* spanning tree. Input is done through std::cin into a 2D array of integers

\*\* then passed into a 2D vector. Sets are counted using std::vector's member

\*\* function called size() and nodes (vertices) are formed from 0 to

\*\* std::vector::size(), all of the nodes are put into unvisited set and they are

\*\* removed from it and added into visited set one by one comparing weights of

\*\* available edges and choosing the one with the minimum weight. The edge is then

\*\* added to the tree which is then returned to main and is printed.

\*\*

\*\* Input is taken of only half the total size of the array because the other

\*\* half is just the repeated edges, and any node's weight from itself is zero

\*/

typedef unsigned short int uint2;

typedef unsigned long long int uint64;

/\*

\*\* An edge contains a starting point, an ending point, and a weight associated

\*\* to it. I've made a structure so it is possible to make a bundled vector array

\*/

struct Edge {

uint2 start;

uint2 end;

uint64 weight;

};

/\* An array of edges is same as a tree \*/

typedef std::vector<Edge> Tree;

/\* Overloading of edge to std::ostream \*/

std::ostream& operator<<(std::ostream& \_os, const Edge& ed) {

\_os << "Edge " << ed.start + 1 << " to " << ed.end + 1 << ", Weight: " <<

ed.weight;

return (\_os);

}

/\* This is the primary method of this program, this method takes a 2D Vector

\*\* of integers and returns a tree and its maximum weight.

\*/

std::pair<Tree, uint64> Prims\_MST(const std::vector<std::vector<int>>&

graph) {

/\*

\*\* STL <set> is a library used here which lets us save the vertices that are

\*\* not visited yet. The library contains const data so editing is impossible

\*\* once the data is added. However, ability to delete the data is there so we

\*\* used that ability to save the unvisited nodes and then delete them as we

\*\* visit the nodes one by one.

\*/

std::set<uint2> unvisitedNodes;

std::set<uint2> visitedNodes;

/\*

\*\* we have used std::vector here typedef'd as "Tree" (see Line17) because it is

\*\* a standard template library and has certain advantages over normal

\*\* arrays. The ability to use iterators and auto as well as faster insertion

\*\* makes vectors better compared to cstyle arrays

\*/

Tree MST;

// Starting point is always visitedNodes with no cost

visitedNodes.insert(0);

// Insert all nodes to unvisitedNodes

for (auto i = 1; i < graph.size(); ++i) {

unvisitedNodes.insert(i);

}

while (!unvisitedNodes.empty()) {

// Shortest edge is initialized with -1 as start and end, and INT\_MAX as weight

Edge shortestEdge = {-1, -1, INT\_MAX};

// Put all edges (with their weights) from nodes that are in visitedNodes

for (auto node : visitedNodes) {

auto const& weightsInRows = graph[node];

for (int end\_node = 0; end\_node < weightsInRows.size(); ++end\_node) {

auto const weight = weightsInRows[end\_node];

if (weight > 0 && weight < shortestEdge.weight

&& unvisitedNodes.count(end\_node)) {

// New edge from node to end\_node formed with weight: weight

shortestEdge = {node, (int)end\_node, weight};

}

}

}

// Add the shortest path to the result

MST.push\_back(shortestEdge);

// Mark the secondary node as visitedNodes

unvisitedNodes.erase(shortestEdge.end);

visitedNodes.insert(shortestEdge.end);

}

uint64 MSTWeight = 0;

for (const auto& iter : MST) {

MSTWeight += iter.weight;

}

return std::pair<Tree, uint64>(MST,MSTWeight);

}

std::vector<std::vector<int>> makeGraph() {

uint2 NodesCount;

std::cout << "Enter the number of vertices: ";

std::cin >> NodesCount;

int\*\* GraphMatrix = new int\*[NodesCount];

for (int i = 0; i < NodesCount; ++i) {

GraphMatrix[i] = new int[NodesCount];

}

system("cls");

for (int i = 0; i < NodesCount; i++) {

std::cout << "Enter 0 if there is no edge between the vertices\n\n\n";

for (int j = i + 1; j < NodesCount; j++) {

std::cout << "Enter Row " << i + 1 << " Column " << j + 1

<< "\'s value: ";

std::cin >> GraphMatrix[i][j];

if (!GraphMatrix[i][j]) {

GraphMatrix[i][j] = INT\_MAX;

}

}

system("cls");

}

for (int i = 0; i < NodesCount; i++) {

if (GraphMatrix[i][i]) {

GraphMatrix[i][i] = 0; // Moving from Vertex N to N costs none

}

}

for (int i = 0; i < NodesCount; i++) {

for (int j = 0; j < NodesCount; j++) {

if (GraphMatrix[i][j]) {

GraphMatrix[j][i] = GraphMatrix[i][j]; // Cost from N-M == M-N

}

}

}

std::vector<int>\* matrixRow = new std::vector<int>;

std::vector<std::vector<int>> graph;

for (int i = 0; i < NodesCount; i++) {

for (int j = 0; j < NodesCount; j++) {

matrixRow->push\_back(GraphMatrix[i][j]);

}

graph.push\_back(\*matrixRow);

matrixRow->clear();

}

delete matrixRow;

for (int i = 0; i < NodesCount; i++) {

delete[] GraphMatrix[i];

}

delete[] GraphMatrix;

return graph;

}

int main() {

std::pair<Tree, uint64> pair = Prims\_MST(makeGraph());

for (auto const& iter : pair.first) {

std::cout << iter << '\n';

}

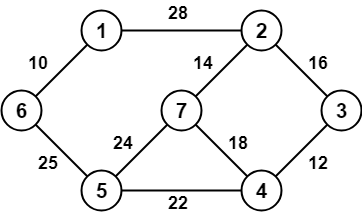
std::cout << "\n" << "The weight of minimum spanning tree is: "

<< pair.second;

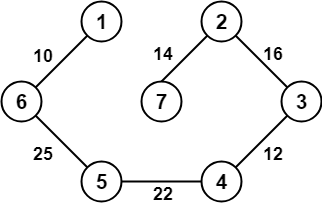
}

## Testing:

Consider this graph:



The Minimum Spanning Tree for the graph is:



The weight of the minimum spanning tree is 99 units.

Now, in our program, we implemented this graph and found these results

