# Experiment 3: Neighbour Table Determination

**Aim:** To create neighbor table for a given network topology

**Objective:** After carrying out this experiment, students will be able to:

* Generate neighbor table for all the nodes in a given topology.
* Analyse how this is useful in the process of routing data

**Problem statement:** You are required to write a program that calculates neighbor table for all the nodes in a given network. Consider a network with 10 nodes that is deployed in an area of 500 m2. Your program should initially determine the distance between each node and all other nodes. Then the range of the nodes is given as input to the user. Using this range information, determine the neighbors of all the nodes.

**Analysis:** While analyzing your program, you are required to address the following points:

* How this is useful in the process of routing data?
* For a 3D topology, how would your program need to be changed?

**MARKS DISTRIBUTION**

|  |  |  |
| --- | --- | --- |
| **Component** | **Maximum Marks** | **Marks Obtained** |
| Preparation of Document | 7 |  |
| Results | 7 |  |
| Viva | 6 |  |
| **Total** | **20** |  |

Submitted by: Satyajit Ghana

Register No: 17ETCS002159

1. Algorithm/Flowchart
2. Program

**main.cpp**

#include <iostream>

#include "network.hpp"

int main(int, char\*\*) {

    using namespace std;

    Network mynet(10);

    // Generate the Random Nodes

    mynet.generate\_random\_nodes\_with\_dimensions(10, 50);

    mynet.print\_nodes\_pos();

    vector<int> ranges(10);

    cout << "Enter the Ranges for the 10 Nodes : ";

    for (int i = 0; i < 10; i++)

        cin >> ranges[i];

    // Set the Node Ranges given by the user and calculate the neighbour table

    mynet.set\_node\_ranges(ranges);

    mynet.print\_neighbour\_table();

}

**network.cpp**

#include "network.hpp"

#include <iostream>

#include <random>

Network::Network(int n\_nodes) : n\_nodes(n\_nodes), neighbour\_table(n\_nodes), node\_ranges(n\_nodes), nodes\_pos(n\_nodes) {}

void Network::generate\_random\_nodes\_with\_dimensions(int length, int breadth) {

    std::random\_device random\_device;

    std::mt19937 random\_engine(random\_device());

    std::uniform\_int\_distribution<int> l\_dist(0, length);

    std::uniform\_int\_distribution<int> b\_dist(0, breadth);

    this->nodes\_pos.clear();

    for (int i = 0; i < this->n\_nodes; i++) {

        nodes\_pos.push\_back({l\_dist(random\_engine), b\_dist(random\_engine)});

    }

    std::cout << this->n\_nodes << " Random Nodes Generated" << std::endl;

}

void Network::print\_nodes\_pos() {

    int i = 0;

    for (auto& node : this->nodes\_pos) {

        std::cout << "NODE " << i << " : (" << node.first << ", " << node.second << ")" << std::endl;

        i++;

    }

}

void Network::set\_node\_ranges(std::vector<int>& ranges) {

    if (ranges.size() != this->nodes\_pos.size()) {

        throw "ranges length does not match nodes\_pos length";

    }

    this->node\_ranges.clear();

    this->node\_ranges.assign(ranges.begin(), ranges.end());

    this->gen\_fully\_connected\_network(this->node\_ranges);

}

template <typename T>

double Network::calc\_dist(std::pair<T, T> pos1, std::pair<T, T> pos2) {

    return std::sqrt(std::abs(

        ((pos1.first - pos2.first) \* (pos1.first - pos2.first) + (pos1.second - pos2.second) \* (pos1.second - pos2.second))));

}

// Generates the network from the given ranges

void Network::gen\_fully\_connected\_network(std::vector<int>& ranges) {

    if (this->node\_ranges.size() != this->nodes\_pos.size()) {

        throw "WTF did you do ? and HTF did you do that ? i'm not even public ";

    }

    this->neighbour\_table.clear\_graph();

    for (int node\_idx = 0; node\_idx < this->nodes\_pos.size(); node\_idx++) {

        for (int node\_jdx = 0; node\_jdx < this->nodes\_pos.size(); node\_jdx++) {

            double dist = Network::calc\_dist(this->nodes\_pos.at(node\_idx), this->nodes\_pos.at(node\_jdx));

            if (dist <= node\_ranges.at(node\_idx)) {

                this->neighbour\_table.add\_edge(node\_idx, node\_jdx, dist);

            }

        }

    }

}

void Network::print\_neighbour\_table() {

    if (this->node\_ranges.empty()) {

        std::cout << "EMPTY" << std::endl;

        return;

    }

    this->neighbour\_table.print\_graph();

}

**network.hpp**

#pragma once

#include <math.h>

#include <vector>

#include "graph.hpp"

class Network {

   public:

    Network(int nodes);

    int n\_nodes;

    std::vector<std::pair<int, int>> nodes\_pos;

    std::vector<int> node\_ranges;

    Graph neighbour\_table;

    void set\_node\_ranges(std::vector<int>& ranges);

    void generate\_random\_nodes\_with\_dimensions(int length, int breadth);

    void print\_nodes\_pos();

    void print\_neighbour\_table();

    template <typename T>

    static double calc\_dist(std::pair<T, T>, std::pair<T, T>);

   private:

    void gen\_fully\_connected\_network(std::vector<int>& ranges);

};

**graph.cpp**

#include "graph.hpp"

#include <iostream>

Graph::Graph(int nodes) : adj\_list(nodes), order(nodes) {}

void Graph::add\_edge(int src, int dest, double weight) {

    if (src >= adj\_list.size() || dest >= adj\_list.size()) {

        throw "Tried to Add Edge for Vertex that does not exist";

    }

    adj\_list.at(src).push\_back({dest, weight});

}

void Graph::print\_graph() {

    std::cout << "GRAPH" << std::endl;

    int i = 0;

    for (auto& row : adj\_list) {

        std::cout << "NODE " << i << " -> ";

        for (auto& ele : row) {

            std::cout << ele.first << " : " << ele.second << " , ";

        }

        std::cout << std::endl;

        i++;

    }

}

**graph.hpp**

#pragma once

#include <vector>

// Weighted DiGraph using Adjacency List

class Graph {

   public:

    Graph(int);

    std::vector<std::vector<std::pair<int, double>>> adj\_list;

    const int order;

    void clear\_graph() {

        this->adj\_list.clear();

        this->adj\_list.resize(order);

    };

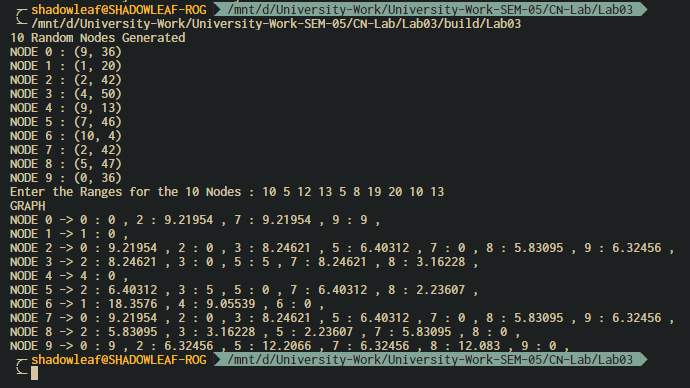
    void add\_edge(int src, int dest, double weight);

    bool is\_empty() { return adj\_list.empty(); };

    void print\_graph();

};

1. Results



1. Analysis and Discussions
2. Conclusions
3. Comments
   1. Limitations of the experiment
   2. Limitations of the results obtained
   3. Learning
   4. Recommendations