Advanced Algorithms CS354

Project Report

Title: **Exact and Rapid Linear Clustering of Networks with Dynamic Programming.**

Team Members:

Naveen 21CSB0B69

Junaid 21CSB0B36

Vishal 21CSB0B11

Presented to:

Dr. Manish Kumar Bajpai

Introduction:

Many complex networks live in ultra low-dimensional spaces. Spatial networks are an obvious example since their nodes are situated in two or three dimensions— locations on a map or the voxels of a connectome, say— and their edges denote relationships such as geographical proximity or physical connections—cables, roads, axons, and so on. But it also makes sense to think of several other types of networks as low-dimensional. In this project we implement a rapid clustering algorithm for one-dimensional space.

Problem Statement:

- Given N points in a one-dimensional plane, node j having score as x□
- The task is to partition these nodes into q Contiguous communities,
- such that if three nodes i, j, and k have decreasing scores xi > xj > xk, then nodes i and k can be in the same community if and only if j is also part of that community.
- We call these communities a layers and represent them as L = { L_1, L_2, ..., Lq } where Lr represents the rth set of contiguous nodes

Key points:

➢ **Input Description**: Takes a one-dimensional network where nodes have positions representing observed covariates (e.g., Score)

➢ **Objective Optimization:** The algorithm aims to find the best partition of nodes into contiguous communities. It optimizes an objective function of the form { Q(L) = ∑ f(L□) for p = 1 to q }, where (f(L□)) is a Quality Function of nodes within a layer (L□).

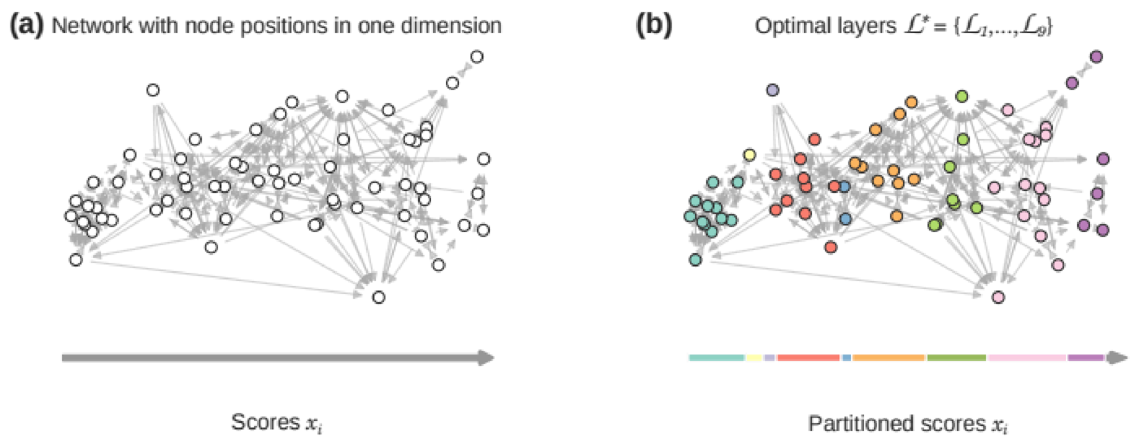
➢ **Layer Representation**: The identified communities are represented as layers { L_1, L_2, ..., Lq }. Each layer contains nodes grouped together based on their positions and connectivity.

➢ **Optimality Assurance:** The algorithm guarantees a provably optimal partition with respect to the specified objective function. This ensures the best possible partition given the objective and constraints.

➢ **Score Comparison:** score of each node k is denoted a x□. For any two nodes j and k if x□ > x□ then node j has higher score than k

➢ **Community Partitioning Task:** The goal is to partition nodes into contiguous communities based on a specific objective function.

➢ **Membership Condition:** given three nodes m, n, o if (x□ > x□ > x_o) and m and o belongs in the same layer, then n should also belong in the same layer.


(a) Network with node positions in one dimension
Scores x_i
(b) Optimal layers $\mathcal{L}^* = \{\mathcal{L}_1,\ldots,\mathcal{L}_9\}$
Partitioned scores x_i

Main Contribution:

➢ The main contribution of this paper is a fast method for finding optimal contiguous communities when the nodes have positions in a single dimension.

➢ The algorithm takes an **embedded network** and an **objective function** as input, and returns an optimal partition in as few as O(n^2) steps where n is the number of nodes, independent from the number of communities q in the partition.

Class of Hardness:

First we show that this problem is not NP as given a certificate, we cannot verify its validation in polynomial time. As to verify it we will have to check using all possible ways, which is exponential, and hence cannot be verified in polynomial time.

$$\sum_{q=1}^{n} \binom{n-1}{q-1} = 2^{n-1}$$

Also we can prove that we can reduce The Graph partition problem which is a known NP-hard problem into an instance of our linear clustering problem, in polynomial time.

Hence The hardness of The Linear Clustering Problem is NP-Hard

Terminology:

- ➢ $L_{i,j}$ to refer to a layer that contains node i through j ≥ i inclusively
- ➢ $f(L_{i,j})$ to refer to its quality
- ➢ Q^* is the quality of the best partition

$$Q(\mathcal{L}) = \sum_{r=1}^{q} f(L_r),$$

Objective Functions:

Objective function 1, (where m_r is the number of edges connecting pairs of nodes)

$$Q(\mathcal{L}) = \sum_{r=1}^{q} m_r$$

is not very interesting, in the sense that one can trivially maximize the number of internal edges by creating a large layer that contains (nearly) all the nodes and edges—thereby saying nothing of substance about the graph itself

Hence, one usually introduces a penalty / regularization term to rule out uninteresting solutions.

Objective Function 2,

$$Q(\mathcal{L}) = \sum_{r=1}^{q}(m_r - n_r^2),$$

Where n_r is the number of nodes in the layer r, and m_r is the number of edges connecting pairs of nodes.

Existing Solutions:

➢ Existing algorithms, such as the critical gap method and other greedy strategies, only offer approximate solutions to this problem.

➢ Brute Force -> for an unknown number of layers q, the size of the solution space grows exponentially.

$$\sum_{q=1}^{n} \binom{n-1}{q-1} = 2^{n-1}$$

Proposed Solution:

➢ **Preprocessing**:

We label nodes by scores, such that node 1 has score x1 and subsequent nodes have non-increasing scores x1 ≥ x2 ≥ . . . ≥ xn

In this step we collapse all nodes with equal scores into “super-nodes.”

➢ **DP Approach**:

The DP approach sets up this calculation by introducing Q∗☐ , the maximum of Q when optimized over possible separations L of the j nodes with the highest scores while ignoring the bottom n − j nodes.

➢ We compute Q∗ j through recursion, we assume that we already know the maxima Q∗ 1 , Q∗ 2 , ..., Q∗ k−1 of the objective function when optimized over partitions of the first k − 1 nodes (the k − 1 nodes with the highest scores). The base case is Q∗ 0 = 0.

$$Q_j^* = \max_{k=1,\ldots,j} \Big\{ Q_{k-1}^* + f(L_{k,j}) \Big\},$$

➢ The associated optimal partition L ∗ which is the object we actually want.

$$\mathcal{L}^* = \mathrm{argmax}_{\mathcal{L}} \Big\{ Q(\mathcal{L}) \Big\},$$

➢ We can determine the partition L ∗ associated with the optimal solution Q∗ by backtracking through stored indices once the calculation is over.

➢ Starting from the end, the stored index k immediately gives us the content of the bottom layer, which comprises node k through n.

➢ We then retrieve the index k ′ that maximized Q∗𝜿 and construct the second-to-last layer, Lk′ ,k. We continue this process and build the optimal partition L ∗ by recursing until we reach node 1.

- The algorithm is guaranteed to return an optimal solution, and thus, we call it an exact algorithm.

Complexity Calculation:

The algorithm's most costly step is in the inner loop, namely the maximization in below equation :

$$Q_j^* = \max_{k=1,\dots,j} \left\{ Q_{k-1}^* + f(L_{k,j}) \right\},$$

Each evaluation of Eq requires j calls to the layer-quality function f(·), meaning that a complete execution of the algorithm visits this steps j times for j = 1, ..., n.

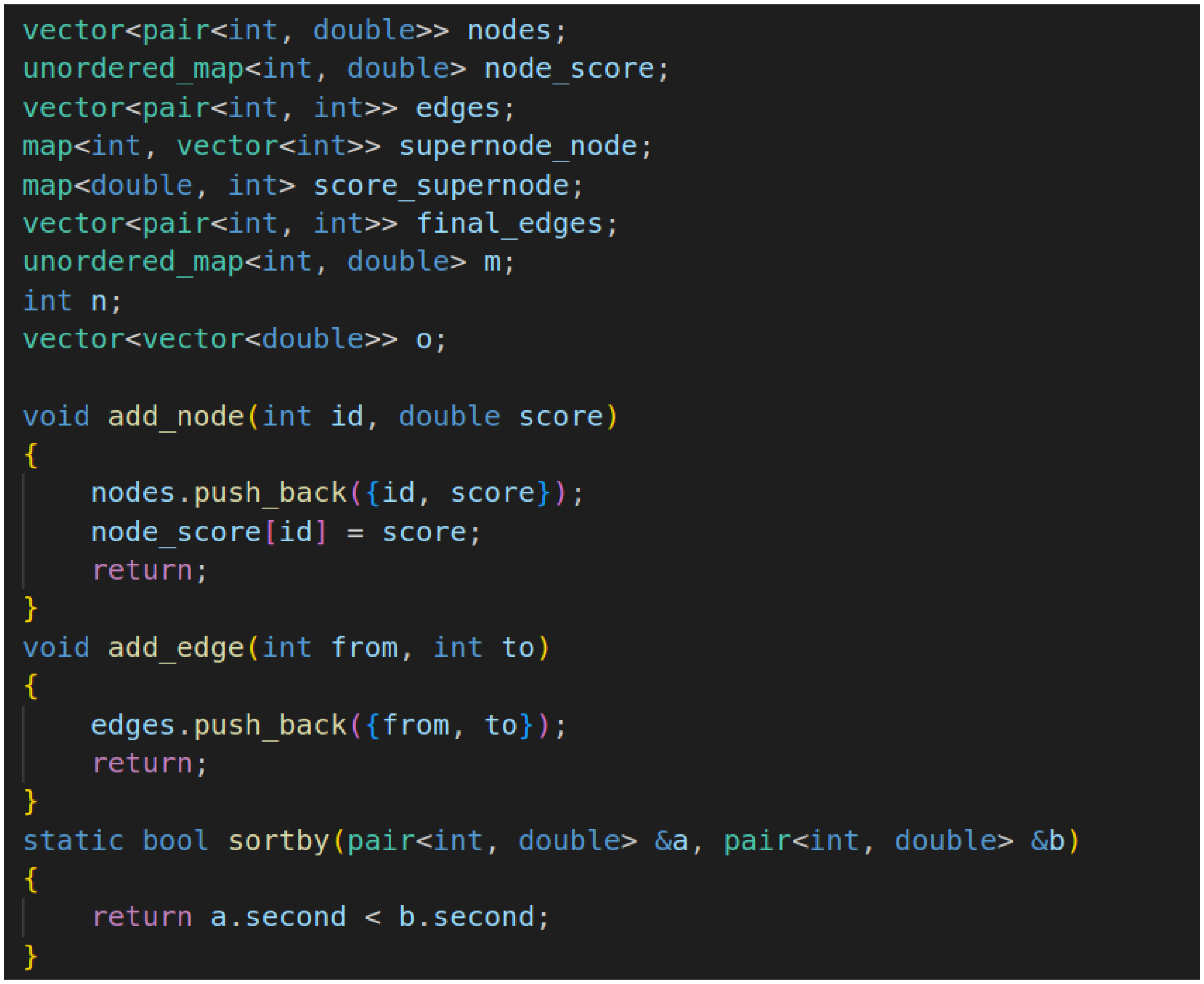
The layer quality is thus called $\sum_{j=1}^{n} j = \binom{n+1}{2}$ times, which implies an overall time-complexity that scales at least as O(n²) with the number of nodes n.

This bound is achieved if f(·) can be evaluated in constant time, but when calls to f(·) depend on n, the total running time may increase faster than O(n²).

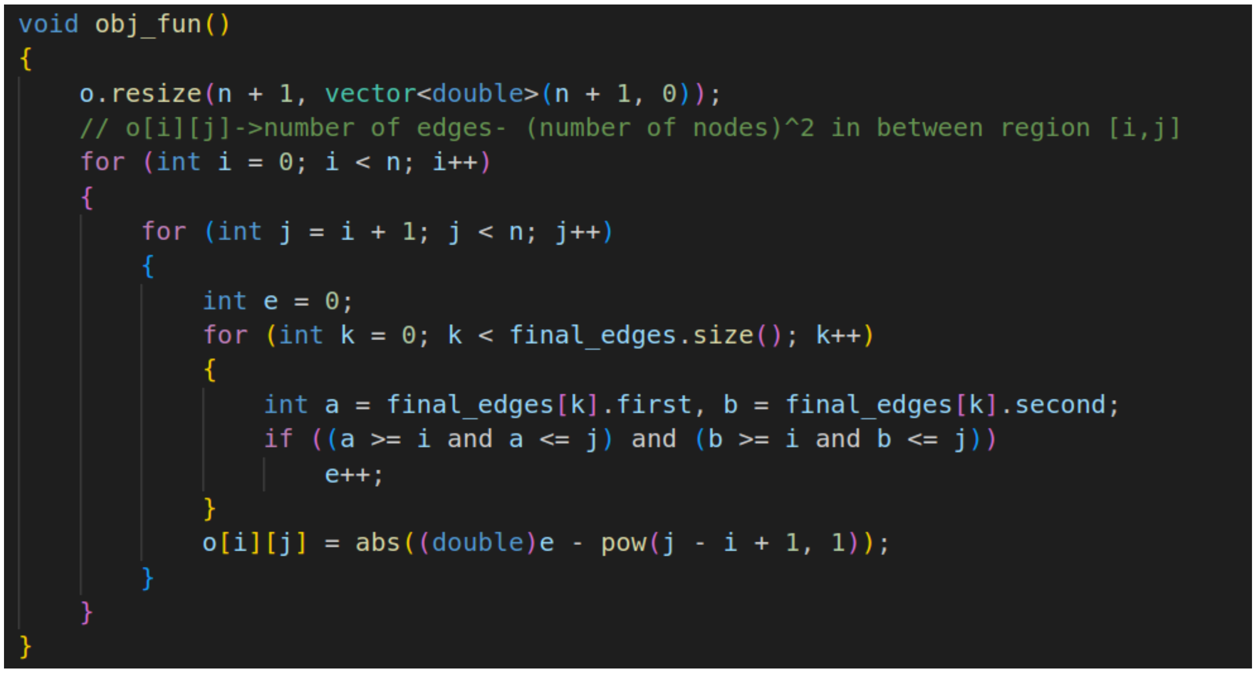
For example, if it takes at least as many operations as there are nodes in a layer to evaluate f(·), then the number of operations will be lower bounded by->

$$\sum_{j=1}^{n} \sum_{k=1}^{j} (j-k) = \binom{n+1}{3} = O(n^3),$$

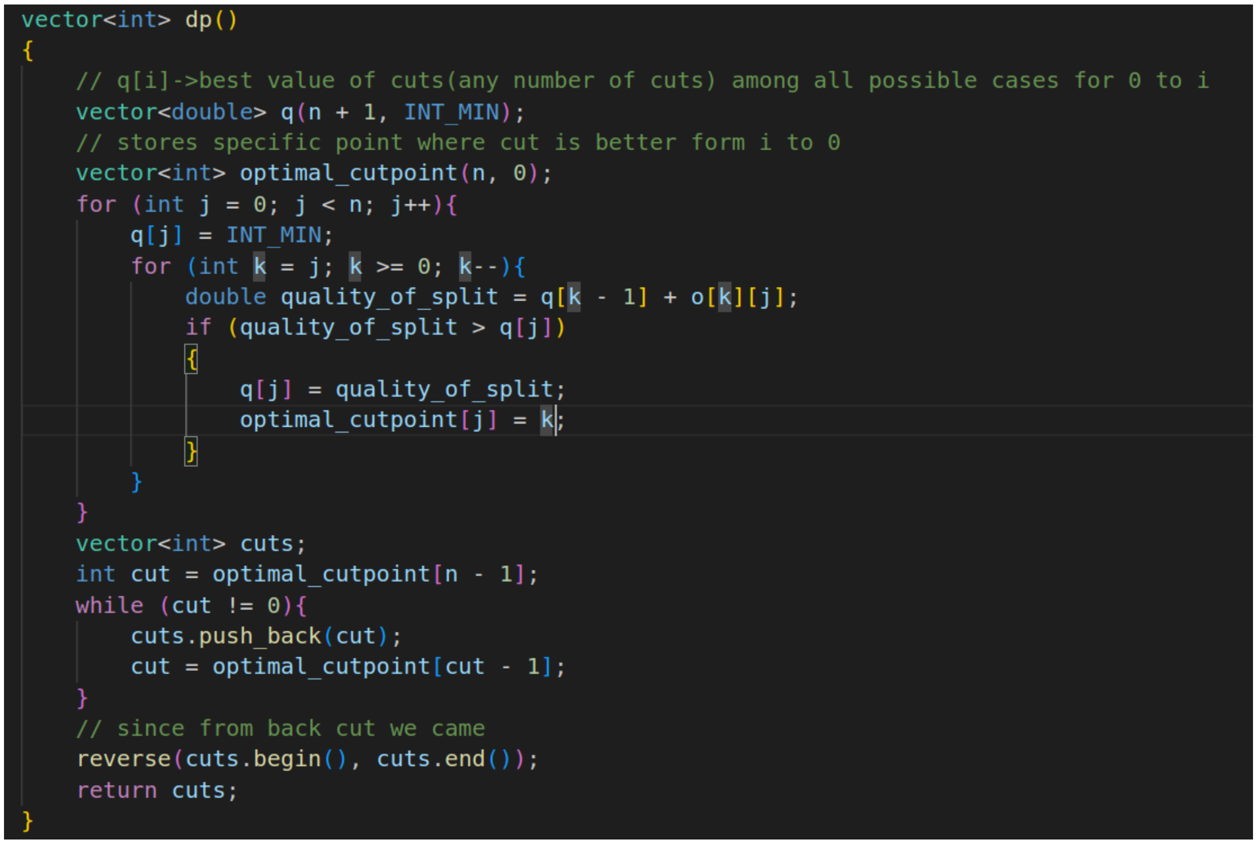
Implementation:



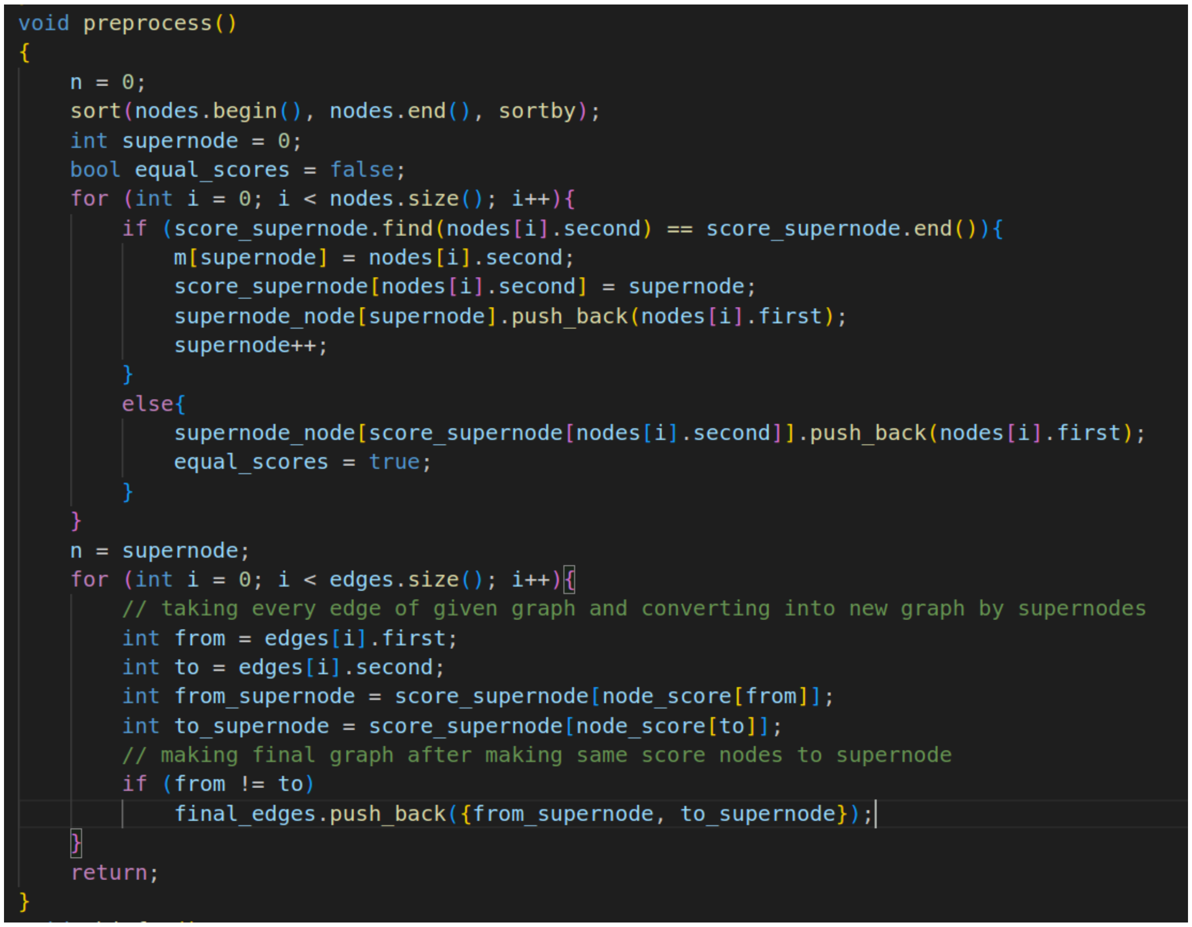
```cpp
vector<pair<int, double>> nodes;
unordered_map<int, double> node_score;
vector<pair<int, int>> edges;
map<int, vector<int>> supernode_node;
map<double, int> score_supernode;
vector<pair<int, int>> final_edges;
unordered_map<int, double> m;
int n;
vector<vector<double>> o;

void add_node(int id, double score)
{
    nodes.push_back({id, score});
    node_score[id] = score;
    return;
}
void add_edge(int from, int to)
{
    edges.push_back({from, to});
    return;
}
static bool sortby(pair<int, double> &a, pair<int, double> &b)
{
    return a.second < b.second;
}
```



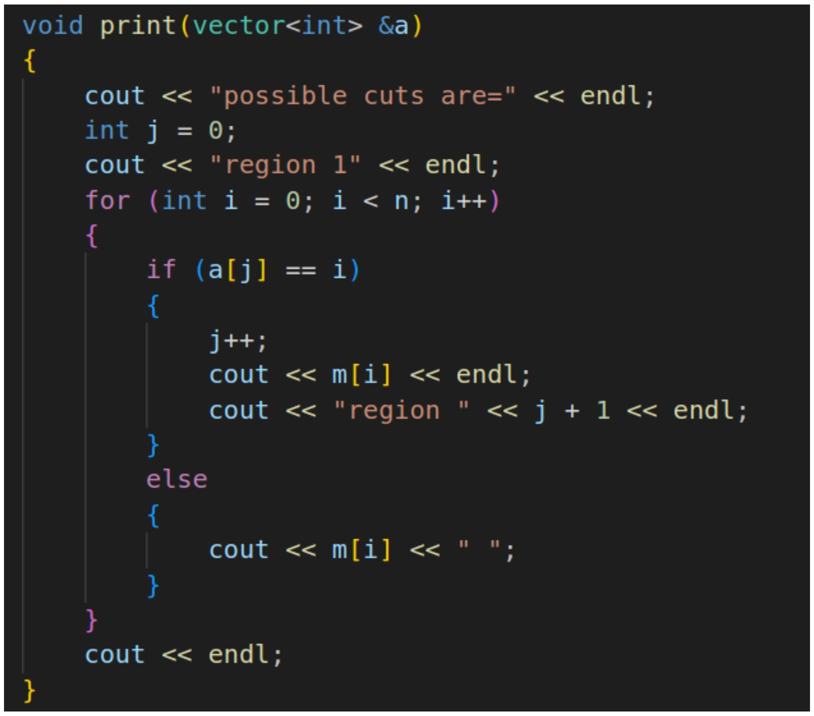
```
void obj_fun()
{
    o.resize(n + 1, vector<double>(n + 1, 0));
    // o[i][j]->number of edges- (number of nodes)^2 in between region [i,j]
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            int e = 0;
            for (int k = 0; k < final_edges.size(); k++)
            {
                int a = final_edges[k].first, b = final_edges[k].second;
                if ((a >= i and a <= j) and (b >= i and b <= j))
                    e++;
            }
            o[i][j] = abs((double)e - pow(j - i + 1, 1));
        }
    }
}
```



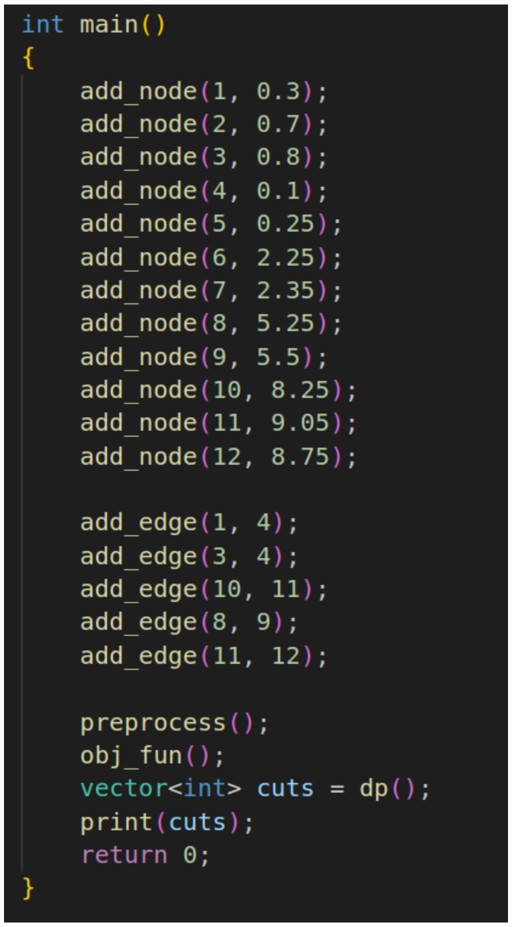
```
vector<int> dp()
{
    // q[i]->best value of cuts(any number of cuts) among all possible cases for 0 to i
    vector<double> q(n + 1, INT_MIN);
    // stores specific point where cut is better form i to 0
    vector<int> optimal_cutpoint(n, 0);
    for (int j = 0; j < n; j++){
        q[j] = INT_MIN;
        for (int k = j; k >= 0; k--){
            double quality_of_split = q[k - 1] + o[k][j];
            if (quality_of_split > q[j])
            {
                q[j] = quality_of_split;
                optimal_cutpoint[j] = k;
            }
        }
    }
    vector<int> cuts;
    int cut = optimal_cutpoint[n - 1];
    while (cut != 0){
        cuts.push_back(cut);
        cut = optimal_cutpoint[cut - 1];
    }
    // since from back cut we came
    reverse(cuts.begin(), cuts.end());
    return cuts;
}
```



```cpp
void preprocess()
{
    n = 0;
    sort(nodes.begin(), nodes.end(), sortby);
    int supernode = 0;
    bool equal_scores = false;
    for (int i = 0; i < nodes.size(); i++){
        if (score_supernode.find(nodes[i].second) == score_supernode.end()){
            m[supernode] = nodes[i].second;
            score_supernode[nodes[i].second] = supernode;
            supernode_node[supernode].push_back(nodes[i].first);
            supernode++;
        }
        else{
            supernode_node[score_supernode[nodes[i].second]].push_back(nodes[i].first);
            equal_scores = true;
        }
    }
    n = supernode;
    for (int i = 0; i < edges.size(); i++){
        // taking every edge of given graph and converting into new graph by supernodes
        int from = edges[i].first;
        int to = edges[i].second;
        int from_supernode = score_supernode[node_score[from]];
        int to_supernode = score_supernode[node_score[to]];
        // making final graph after making same score nodes to supernode
        if (from != to)
            final_edges.push_back({from_supernode, to_supernode});
    }
    return;
}
```

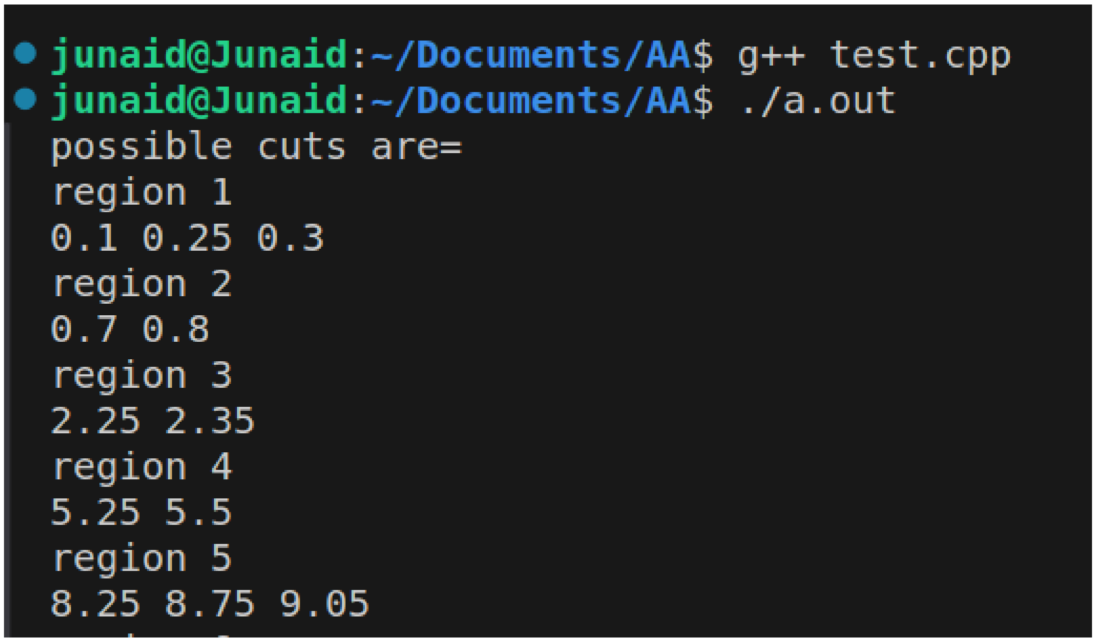


```cpp
void print(vector<int> &a)
{
    cout << "possible cuts are=" << endl;
    int j = 0;
    cout << "region 1" << endl;
    for (int i = 0; i < n; i++)
    {
        if (a[j] == i)
        {
            j++;
            cout << m[i] << endl;
            cout << "region " << j + 1 << endl;
        }
        else
        {
            cout << m[i] << " ";
        }
    }
    cout << endl;
}
```



```cpp
int main()
{
    add_node(1, 0.3);
    add_node(2, 0.7);
    add_node(3, 0.8);
    add_node(4, 0.1);
    add_node(5, 0.25);
    add_node(6, 2.25);
    add_node(7, 2.35);
    add_node(8, 5.25);
    add_node(9, 5.5);
    add_node(10, 8.25);
    add_node(11, 9.05);
    add_node(12, 8.75);

    add_edge(1, 4);
    add_edge(3, 4);
    add_edge(10, 11);
    add_edge(8, 9);
    add_edge(11, 12);

    preprocess();
    obj_fun();
    vector<int> cuts = dp();
    print(cuts);
    return 0;
}
```

Result:



```
junaid@Junaid:~/Documents/AA$ g++ test.cpp
junaid@Junaid:~/Documents/AA$ ./a.out
possible cuts are=
region 1
0.1 0.25 0.3
region 2
0.7 0.8
region 3
2.25 2.35
region 4
5.25 5.5
region 5
8.25 8.75 9.05
```

Limitations:

➢ Only used for finding optimal contiguous communities when the nodes have single dimension – the linear clustering problems.

➢ The quadratic scaling of its complexity with n is too rapid for use with very large networks.

➢ The DP strategy is limited to linear clustering, because the recursive structure of breaks down in more than one dimension.

Possible Improvements:

➢ Can be used for finding optimal contiguous communities for the nodes in the two-dimensional hyperbolic space H^2. This can be done by interpreting the “popularity” r_i along the radial dimension and as the “similarity” along the angular dimension.

➢ It might be possible to adopt a **divide-and-conquer strategy** and split the network into chunks of less-than-linear size that could then be processed independently. This would lead to better scaling, at the cost of some accuracy.

Conclusion:

In conclusion, our project implemented a rapid linear clustering algorithm for networks with one-dimensional space using Dynamic Programming.

References:

Paper : https://arxiv.org/abs/2301.10403

CPP (used for implementation) : https://en.cppreference.com/w/

SpatialNetworks:https://www.sciencedirect.com/science/article/abs/pii/S037015731000308X?via%3Dihub

A survey of graph layout problems :
https://dl.acm.org/doi/10.1145/568522.568523