

Assignment 2

ELL 880 Social Network Analysis

Implementation of Page Rank Algorithm, Strongly connected components, Giant component, Articulation vertices and Bridges

Mohit Sinha
2019EET2345



Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY
DELHI
2020-21

Implementation of Page Rank, Strongly connected components, Giant component, Articulation vertices and Bridges

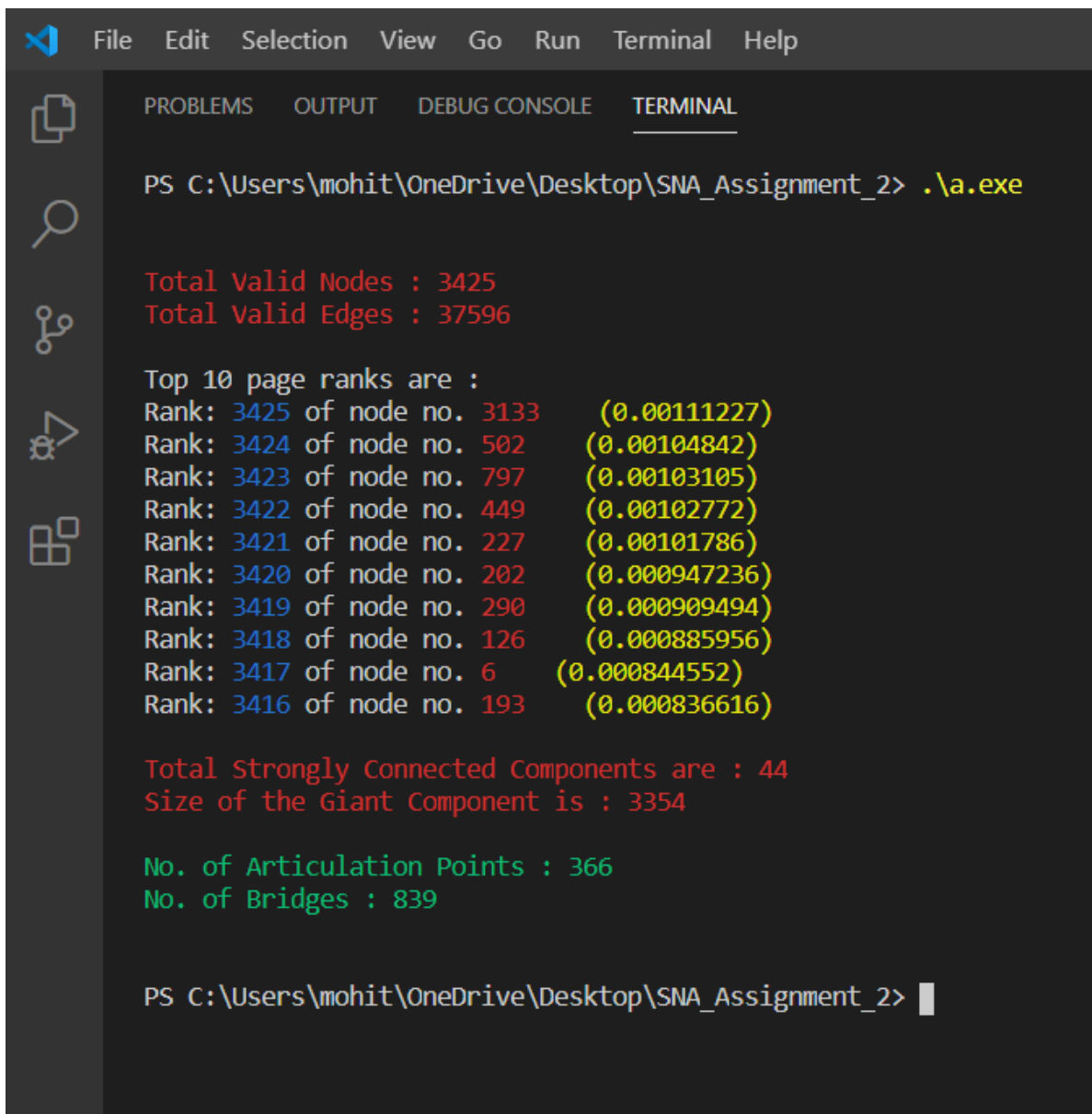
1. Dataset

I have used the flight and Airport data available from the website www.openflight.org

2. Results

The code output screenshot is shown below. Since the output contains the page rank of approx. 3500 nodes and it cannot be shown in a single screenshot, so only the few top 'K' nodes sorted in the highest page rank along with the values are shown below. The rest can be seen by changing the 'K' value in the code. As a bonus, I also wrote the code for strongly connected components, giant component, articulation nodes and bridges.

3. Output



```
File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\mohit\OneDrive\Desktop\SNA_Assignment_2> .\a.exe

Total valid Nodes : 3425
Total valid Edges : 37596

Top 10 page ranks are :
Rank: 3425 of node no. 3133 (0.00111227)
Rank: 3424 of node no. 502 (0.00104842)
Rank: 3423 of node no. 797 (0.00103105)
Rank: 3422 of node no. 449 (0.00102772)
Rank: 3421 of node no. 227 (0.00101786)
Rank: 3420 of node no. 202 (0.000947236)
Rank: 3419 of node no. 290 (0.000909494)
Rank: 3418 of node no. 126 (0.000885956)
Rank: 3417 of node no. 6 (0.000844552)
Rank: 3416 of node no. 193 (0.000836616)

Total Strongly Connected Components are : 44
Size of the Giant Component is : 3354

No. of Articulation Points : 366
No. of Bridges : 839

PS C:\Users\mohit\OneDrive\Desktop\SNA_Assignment_2> |
```

4. Code

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

#define RED_COLOR "\x1B[31m"
#define GREEN_COLOR "\x1B[32m"
#define BLUE_COLOR "\x1B[34m"
#define YELLOW_COLOR "\x1B[33m"
#define COLOR_RESET "\x1B[0m"

class Graph{
private:
    int n;
    unordered_map<int, string> info;
    vector<int> *edges = new vector<int>[n];
    vector<int> *edgesT = new vector<int>[n];
    long double *pastIteration = new long double[n];
    long double *presentIteration = new long double[n];
    void calPageRank();
    vector<set<int>> getSCC();
    void dfs1(int u, vector<int>& disc, vector<int>& low, vector<bool>& visited, vector<bool>& ap, vector<pair<int, int>> & bridges, int parent);
    void dfs2(int start, vector<bool>& visited, stack<int>& finishedStack);
    void dfs3(int start, vector<bool>& visited, set<int>& components);
public:
    void loadGraph(); // Loads the graph from the input .txt file
    void printRank(int k); // Top k page ranks
    void countSCC(); // Gives the count of strongly connected components using Kosaraju algorithm
    void articulationPointsAndBridges(); // Articulation vertices and bridges whose removal increases no. of components in a graph
    ~Graph(); // Destructor
};

void Graph::loadGraph(){
    unordered_map<string, int> mp;
    cin >> n;
    string u, v;
    int count = 0;
    set<pair<string, string>> edgesInfo;
    while(cin >> u){
        if(mp.find(u) == mp.end()){
            mp[u] = count;
            info[count] = u;
            count++;
        }

        cin >> v;
        if(mp.find(v) == mp.end()){
            mp[v] = count;
            info[count] = v;
            count++;
        }
    }
}
```

```

        if(edgesInfo.find({u, v}) == edgesInfo.end()){

            edges[mp[u]].push_back(mp[v]);
            edgesT[mp[v]].push_back(mp[u]);

            edgesInfo.insert({u, v});
        }

    }

    cout << "Total Valid Nodes : " << count << endl;
    cout << "Total Valid Edges : " << edgesInfo.size() << endl;
    calPageRank();
}

/*-----Util function to calculate page rank-----*/
void Graph::calPageRank(){
    // first initialisation
    for(int i = 0; i < n; i++){
        pastIteration[i] = (long double)1 / (long double)n;
    }
    int t = 5000; // no. of iterations
    while(t--){
        for(int i = 0; i < n; i++){
            long double sum = 0;

            for(int node : edgesT[i]){
                long double numerator, denominator;
                numerator = pastIteration[node];
                denominator = (long double)edges[node].size();
                sum = sum + (numerator / denominator);
            }
            presentIteration[i] = sum;
        }

        for(int j = 0; j < n; j++){
            pastIteration[j] = presentIteration[j];
        }
    }
}

/*-----member funtion to print the top k page ranks-----*/
void Graph::printRank(int k){
    multimap<long double, int> mpp;
    for(int i = 0; i < n; i++){
        mpp.insert({presentIteration[i], i});
    }

    vector<pair<long double, int>> ans;
    for(auto pp : mpp){
        ans.push_back(pp);
    }
    cout << endl;
}

```

```

        cout << "Top " << k << " page ranks are : " << endl;
        for(int i = (int)ans.size() - 1; i >= 0 && k > 0; i--, k--){
            cout << "Rank: " << i + 1 << " of node no. " << "(" << ans[i].second << "
            (Airport code : " << info[ans[i].second] << ")" << "(" << ans[i].first << ")" << "\n";
        }
    }

void Graph::dfs2(int start, vector<bool>& visited, stack<int>& finishedStack){
    visited[start] = true;
    for(auto it = edges[start].begin(); it != edges[start].end(); it++){
        if(visited[*it] == false){
            dfs2(*it, visited, finishedStack);
        }
    }
    finishedStack.push(start); // at the end of dfs1 push it onto stack
}

void Graph::dfs3(int start, vector<bool>& visited, set<int>& components){
    visited[start] = true;
    components.insert(start);
    for(auto it = edgesT[start].begin(); it != edgesT[start].end(); it++){
        if(visited[*it] == false){
            dfs3(*it, visited, components);
        }
    }
}

vector<set<int>> Graph::getSCC(){
    vector<bool> visited(n, false);
    stack<int> finishedStack;
    for(int i = 0; i < n; i++){
        if(visited[i] == false){
            dfs2(i, visited, finishedStack);
        }
    }

    vector<bool> visited2(n, false); // again make new visited array
    vector<set<int>> output;
    while(!finishedStack.empty()){
        set<int> components;
        int topElement = finishedStack.top();
        finishedStack.pop();
        if(visited2[topElement] == false){
            dfs3(topElement, visited2, components); //here dfs3 is implemented on transpose of
            edges i.e all original direction of edges are changed i.e using edgesT adjacency list
            output.push_back(components);
        }
    }

    return output;
}

/*-----Util function to calculate total strongly connected components-----*/
void Graph::countSCC(){

```

```

vector<set<int>> output = getSCC();
//<<1, 3, 5>, <2, 4, 6>, <....>> vector of set i.e, output is vector of components stored
as sets
cout << endl;
int giantComponentSize = 0;
for(int i = 0; i < output.size(); i++){
    giantComponentSize = max(giantComponentSize, (int)output[i].size());
}

cout << "Total Strongly Connected Components are : " << (int)output.size() << endl;
cout << "Size of the Giant Component is : " << giantComponentSize << endl;
}

void Graph::dfs1(int u, vector<int>& disc, vector<int>& low,
vector<bool>& visited, vector<bool>& ap, vector<pair<int, int> >& bridges, int parent){
    static int time=1;
    disc[u] = low[u] = time++;
    int child=0;
    visited[u] = true;
    for (auto& ele: edges[u]){
        if (!visited[ele]){
            child++;
            dfs1(ele, disc, low, visited, ap, bridges, u);
            low[u] = min(low[u], low[ele]);
            if (parent==-1 && child>1) ap[u] = true;
            else if (parent!=-1 && low[ele]>=disc[u]) ap[u] = true;
            if (low[ele]>disc[u]) bridges.push_back({u, ele});
        }
        else if (ele != parent){
            low[u] = min(low[u], disc[ele]);
        }
    }
}

/*-----to calculate articulation points and bridges in a graph-----*/
void Graph::articulationPointsAndBridges(){
    vector<bool> visited(n, false);
    vector<bool> ap(n, false);
    vector<pair<int, int> > bridges;
    vector<int> disc(n, INT_MAX);
    vector<int> low(n, INT_MAX);
    int count=0;
    for (int i=0;i<n;i++){
        if (!visited[i]) dfs1(i, disc, low, visited, ap, bridges, -1);
    }
    for (int i=0;i<n;i++){
        if (ap[i]) count++;
    }
    cout << endl;
    cout << "No. of Articulation Points : ";
    cout << count << endl;

    /*-----to print each Articulated vertices-----*/

```

```

/*
cout << "Articulation Vertices:" << endl;
for (int i=0;i<n;i++){
    if (ap[i]) cout << i << " ";
}
*/

auto compare = [&](auto a, auto b){
return a.first<b.first || (a.first==b.first && a.second<b.second);
};

sort(bridges.begin(), bridges.end(), compare);
cout << "No. of Bridges : ";
cout << bridges.size() << endl;

/*-----to print each bridges-----*/
/*
cout << "Bridges are : " << endl;
for (auto& ele: bridges){
cout << ele.first << " " << ele.second << endl;
}
*/
}

/*-----Destructor-----*/
Graph::~~Graph(){
    delete[] edges;
    delete[] edgesT;
    delete[] pastIteration;
    delete[] presentIteration;
}

int main(){

    // for getting input from input.txt
    freopen("edges.txt", "r", stdin);
    // for writing output to output.txt
    freopen("out.txt", "w", stdout);

    cout << "\n\n";
    Graph G; // Creates the object of the Class Graph
    G.loadGraph(); // Loads the graph from the input .txt file
    G.printRank(10); // Top k page ranks
    G.countSCC(); // Gives the count of strongly connected components
                  // using Kosaraju algorithm
    G.articulationPointsAndBridges(); // Articulation vertices and bridges whose removal
                                     // increases no. of components in a graph

    cout << "\n\n";
    return 0;
}

```