Algorithms HW3

화학부 2016-14043 이준영

1. **Environment**
    a. Language : C++17
    b. Compiler : clang++
    c. IDE : Visual Studio Code
    d. OS : MacOS Monterey 12.3
    e. tasks.json (for build)

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: clang++ build active file",
      "command": "/usr/bin/clang++",
      "args": [
        "-std=c++17",
        "-fdiagnostics-color=always",
        "-g",
        //"${workspaceFolder}/inputMaker/*.cpp",
        "${workspaceFolder}/*.cpp",
        "-o",
        //"${workspaceFolder}/inputMaker/inputMaker"
        "${workspaceFolder}/hw3",
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind" : "build",
        "isDefault": true
      },
      "detail": "compiler: /usr/bin/clang++"
    }
  ]
}
```

    f. execution code (in terminal)

```
./hw3 input/input.txt input/adj_mat.txt adj_mat
./hw3 input/input.txt input/adj_list.txt adj_list
```

```
./hw3 input/input.txt input/adj_arr.txt adj_arr
```

2. **Code**

   a. Used libraries

```
#include <iostream> //printing result on console
#include <iostream> //printing result on console
#include <sstream> //reading input and writing output
#include <fstream> //reading input and writing output
#include <string>
#include <vector>
#include <chrono>  //checking running time
#include <algorithm>  //sorting vecto
```

   b. Overview

   Each graph representation corresponds to the classes AdjMat, AdjList, and AdjArray. The classes work similarly. They execute DFS on the graph and make a stack which contains the vertices in finished time order. They make a reversed graph using the original graph, then execute DFS on the reversed graph in stack order. The strongly connected components are stored in 2d vectors and printed in txt files.

```
int main(int argc, char **argv){
    string inputPath = argv[1];
    string outputPath = argv[2];
    string TYPE = argv[3];

    if(TYPE == "adj_mat"){
        AdjMat MAT;
        MAT.readInput(inputPath);
        MAT.makeSCC(outputPath);
    }
    else if(TYPE == "adj_list"){
        AdjList LIST;
        LIST.readInput(inputPath);
        LIST.makeSCC(outputPath);
    }
    else if(TYPE == "adj_arr"){
        AdjArray ARR;
        ARR.readInput(inputPath);
        ARR.makeSCC(outputPath);
    }
    else cout << "Wrong Argument" << endl;

    return 0;
}
```

   c. Adjacency Matrix

The followings are data structures used in class AdjMat. Reversed graph was made by transposing the original graph.

```
class AdjMat{
private:
    int numVtx;
    vector<vector<int>> mat;    //Adjacency Matrix
    vector<vector<int>> matReverse; //Transpose of Adjacency Matrix (reversed graph)
    vector<bool> visited;   //visited vertices in DFS
    vector<bool> rVisited;  //visited vertices in DFS of reverse graph
    vector<int> finStack;   //Stack by finished time of DFS
    vector<vector<int>> SCC;    //Strongly connected components
};
```

In this homework, making reversed graph was the bottleneck. The code for making reversed graph follows as:

```
void makeMatReverse(){
        vector<vector<int>> initRev(numVtx, vector<int>(numVtx, 0));
        matReverse = initRev;
        for(int i = 0; i < numVtx; i++){
            for(int j = 0; j < numVtx; j++){
                matReverse[i][j] = mat[j][i];
            }
        }
    }
```

d. Adjacency List

```
class AdjList{
private:
    int numVtx;
    vector<listNode*> headList;     //List storing header of listNode pointer
    vector<listNode*> reverseList;      //Reversed graph
    vector<bool> visited;       //visited vertices in DFS
    vector<bool> rVisited;      //visited verticess in DFS of reversed graph
    vector<int> finStack;       //Stack by finished time of DFS
    vector<vector<int>> SCC;     //Strongly Connected Components
};
```

The code for making reversed graph follows as:

```
void makeReverseList(){
        vector<listNode*> tempReverse(headList.size(), nullptr);
        reverseList = tempReverse;
        for(int i = 0; i < headList.size(); i++){
            listNode* tempLast = headList[i];

            while(tempLast != nullptr){
                int outN = tempLast->outNode - 1;
                listNode* newNode = new listNode(i + 1);
```

```
                if(reverseList[outN] == nullptr) {
                    reverseList[outN] = newNode;
                }
                else{
                    listNode* tempBack = reverseList[outN];
                    while(tempBack->nextNode != nullptr) tempBack = tempBack->nextNode;
                    tempBack->nextNode = newNode;
                }
                tempLast = tempLast->nextNode;
            }
        }
    }
```

e. Adjacency Array

```
class AdjArray{
private:
    int numVtx;
    vector<int> Vtx;        //array for storing end point of Edge for each vertex
    vector<int> Edge;       //array for stroring outgoing edge
    vector<int> rVtx;       //for reversed graph
    vector<int> rEdge;      //for reversed graph
    vector<bool> visited;       //visited vertices in DFS
    vector<bool> rVisited;      //visited vertices in DFS of reversed graph
    vector<int> finStack;       //Stack by finished time of DFS
    vector<vector<int>> SCC;    //Strongly Connected Components
};
```

The code for making reversed graph follows as:

```
void makeReverse(){
        vector<int> tempRVtx(Vtx.size(), 0);
        vector<int> tempREdge(Edge.size(), 0);
        rVtx = tempRVtx;
        rEdge = tempREdge;

        for(int v = 0; v < Vtx.size(); v++){
            int startIdx;
            if(v == 0) startIdx = 0;
            else startIdx = Vtx[v - 1];
            int endIdx = Vtx[v];

            for(int e = startIdx; e < endIdx; e++){
                int out = Edge[e];
                rEdge.insert(rEdge.begin() + rVtx[out - 1], v + 1);

                for(int i = out - 1; i < rVtx.size(); i++){
                    rVtx[i]++;
                }
            }
        }
    }
```

3. **Input**

a. inputMaker

I made *inputMaker* program to make input files with various numbers of vertices and dense/sparse graphs. I adjusted the number of edges per vertex. In the sparse graph, the number of edges per vertex is smaller than (#total vertices)/2. On the other hand, in the dense graph, the number of edges per vertex is greater than (#total vertices)/2.

```
#include <iostream>
#include <fstream>
#include <string>
#include <random>
#include <vector>
#include <algorithm>

#define N 10  //Number of vertices

using namespace std;

int main(){
    random_device rd;
    mt19937 mersenne(rd());
    uniform_int_distribution<> out(0, N - 1);

    bool DENSE = true;

    ofstream inputWrite;
    inputWrite.open("input.txt");

    inputWrite << N << endl;

    for(int node = 1; node <= N; node++){
        bool PASS = false;
        int numOut;
        while(!PASS){
            numOut = out(mersenne);
            if((DENSE && numOut >= N/2) || (!DENSE && numOut < N/2)) PASS = true;
        }
        inputWrite << numOut;
        vector<int> saveOut;

        while(saveOut.size() != numOut){
            int newOut = out(mersenne) + 1;
            if(newOut != node){
                auto itr = find(saveOut.begin(), saveOut.end(), newOut);
                if(itr == saveOut.end()) saveOut.push_back(newOut);
            }
        }
        sort(saveOut.begin(), saveOut.end());
        for(int i = 0; i < saveOut.size(); i++){
            inputWrite << " " << saveOut[i];
        }
        if(node != N) inputWrite << "\n";
    }

    inputWrite.close();
```

```
    return 0;
}
```

4. **Experiments**
   a. Sparse graph

# Vertices	Matrix (ms)	List (ms)	Array (ms)
100	1	0	3
500	29	91	1222
1000	171	1313	25018
3000	666	48994	Too Large

   b. Dense graph

# Vertices	Matrix (ms)	List (ms)	Array (ms)
100	1	3	7
500	35	719	11618
1000	140	10545	93598
3000	1548	450239	Too Large

   c. Discussion

   The tendency of the running times in dense and sparse graphs are similar. The adjacency matrix is the fastest one and the adjacency array is the slowest one. The main bottleneck is making reversed graph. Matrix and list took O(V^2) in reversing the graph, while array took O(V^3) time. Time complexity of DFS is O(V^2) in the adjacency matrix, O(V + E) in adjacency list and array. The bottleneck is the reversing the graph. If there exists a fancy algorithm to reverse the graph, the time complexity can be drasticallly decrease especially in the adjacency array.

5. **Example Running**
   a. input.txt

   20
   6 4 7 10 15 16 20
   5 1 4 5 10 18
   2 9 19
   8 1 2 5 6 8 9 14 20
   9 1 2 8 9 11 12 13 15 16
   0

9 1 2 4 6 8 12 14 15 16
6 9 13 14 15 18 19
7 1 3 10 12 17 19 20
8 3 4 8 9 12 13 18 20
7 1 2 3 6 13 15 18
7 5 6 7 11 15 17 18
6 4 9 12 18 19 20
0
0
2 1 11
7 3 5 6 12 13 18 20
6 3 5 8 9 12 17
6 1 4 5 10 13 20
2 9 11

b. adj_mat.txt / adj_list.txt / adj_arr.txt

1 2 3 4 5 7 8 9 10 11 12 13 16 17 18 19 20
14
15
6
0ms