Algorithm Programming Assignment #3 **Report**

B07901021 潘世軒

1. Data structure

```
class Graph
{
    public:
        Graph();
        Graph(char, int);
        void E_construct(int, int, int);
        void test();
        void cyclebreak(fstream&);
        void sort_E();
       void MST_Kruskal();
       void dir_find_delete_edge(vector<Edge>&);
       void add_edge_back(vector<Edge>&);
       void output_u(fstream&);
       void output_d(fstream&);
   private:
       void Makeset(int);
       void Union(int, int);
       int Findset(int);
       void Link(int, int);
       void MergeSort(vector<Edge>&);
       void MergeSortSubVector(vector<Edge>&, int, int);
       void Merge(vector<Edge>&, int, int, int, int);

       bool DFS_cycle_detecting(Edge&);
       bool DFS_visit(int, int, vector<char>&);

       void DFS_cycle_detecting_check();
       void DFS_visit_check(int, vector<char>&);

       char type;
       int n;
       // weight[i][j] means the weight of edge(i,j)(-100~100)
       // if no edges between (i,j), weight[i][j] = 666
       vector<vector<int> > weight;
       vector<vector<int> > A;
       vector<Edge> E;
       vector<vector<int> > adj;
       vector<Vertex> V;
};
```

使用一個叫做 Graph 的資料結構，裏頭包含 vertex, edge，type 是用來表達這個 graph 是 undirected 還是 directed 的，n 則是用來表達 vertice 數目(有時候直接用 n 取代 V.size 比較方便)，weight 則是用來表達 edge weight(如 comment)，在我們進行 cycle breaking 後，會將刪除某些 edge 的結果儲存在 A，最後 output 的時候將 A 和 weight 進行比對來輸出刪掉的邊，最後又多創了一個 adj 的二維陣列，以便跑 DFS。Graph 裡的一大堆函式會在下方的 Algorithm 中解釋。

2. Algorithm

在本次作業中，我將幾乎所有的 function 都定義在 Graph 裡面，一方面是我如果我要在別的地方使用這個演算法，可以直接呼叫 graph 的 member function，感覺比較直覺，一方面是這樣是直接改 graph 裡的各種屬性，不需

要用外部 function 寫 call by referenc。，根據輸入建立好 Graph 後，直接跑 G.cyclebreak(fout) 開始進行拆解 cycle。

```
void Graph::cyclebreak(fstream& fout){
    if(type == 'u'){
        // undirected:
        // (1) find the maximum spanning tree
        // (2) compare the MST to origin edge and find the deleted edge
        MST_Kruskal();
        output_u(fout);
    }
    else if(type == 'd'){
        // directed:
       // (1) treat all edges as undirected edges and find the minimum spanning tree
        // (2) compare the MST to origin edge and find the deleted
        // (3) trying add back edges with "positive weight" in decreasing order
        // (4) run DFS to check whether the edge added will cause cycle in the Graph
        // (5) if not, add it back
        // (6) compare the final result to origin edge and find the deleted edge

        MST_Kruskal();
        vector<Edge> d;
        dir_find_delete_edge(d);
        add_edge_back(d);
        output_d(fout);
    }
    else{
        cout << "type error" << endl;
    }
}
```

(1) Undirected graph
Undirected graph 的部分，要求刪掉的 edge weight 總和越小越好，並且所有頂點都要被連接，因此我將課本中的 MST-Kruskal 稍微修改，改成找 Maximum spanning tree，這樣代表要刪掉的的邊 weight 總和會是 min，改動的地方只有將 edge 根據 weight sort 時的順序改為由大到小(先嘗試把大的加進 MST 中以找到 Maximum spanning tree)，將要刪掉的邊記在上述的 A 矩陣裡再和 weight 比對就可以找到哪些邊是要被刪掉的。

(2) Directed graph
一開始想要使用的方法是跑 DFS 去 detect cycle，並刪掉這個 cycle 中最小 weight 的 edge，但是後來覺得這個方法似乎有點不切實際，因為也許會找到很多 cycle，每個 cycle 又要繞整圈去找，感覺比較麻煩，因此不採用這個方法。
後來選用如上方 code 所示的方法，先將所有的 directed edge 都當作是無向的去跑 MST (max)，跑完後的結果，可以確保剩下來的 edge 絕對不會造成 cycle，接下來將這些被刪掉的 edge，由大排到小開始嘗試一一把 weight>0 的 edge 加進去，加進一條邊後，check 他加進去會不會造成新的 cycle (偵測辦法：若(u, v)加入後會造成 cycle，這個 cycle 一定會包含(u, v)，因此從 v 開始跑 DFS，如果跑得到 u 的話就代表有 cycle 存在！)，如果不會的話就代表我可

以安心的把這個 edge 加進去(並更新上述 A 矩陣以及 adj 矩陣)，當把所有 positive edge 都加進去後，比對 A 矩陣和 weight 就可以寫 output 了。

3. Findings

這次作業比較困難的部分處理有向圖的部分，但我認爲只要想到一個合理的演算法就可以交了，因爲其實我覺得最後刪掉的總 weight，其實和刪除或者加入 edge 的順序有關係，我嘗試過將上方紅字的這個排序改成 quicksort(unstable)，讓同樣 weight 的東西先後順序可能改變，導致 directed case 中有些會變好，有些結果則會變差，但很難去評斷這個順序到底怎麼排比較好，因此最後就隨便取一種 sort 方法來做，並祈禱測資對我有利。

我認爲另一個難題是這個 graph 到底要怎麼建立，課本中的演算法多半寫的很簡單，看看就過去了，但實際上要寫成 code 時卻會先卡一陣子這些資料結構到底要怎麼實現，我覺得我這次寫的 graph 的 coding 沒有很漂亮，因爲有些 private member 其實是可以移除的，但是我後來寫法是需要什麼就把它加進去(比方說 DFS 需要 adj matrix，就加，MST 需要 vertex.rank, p 這些東西，就寫一個 class vertex，加入 graph 中)，但感覺如果客家一點的話應該是有很多東西可以捨去的，總之最後有算出答案了就先維持這樣吧 XD，有空在來修的好看一點。

最後，因爲這次作業比較難用肉眼驗證正確性，因此我自己大爆寫了一個確定答案是否正確的 cpp 檔，讀入原本的 input file，以及算出來的 output file，可以確認是否有 cycle，是否所有 vertex 都走的到，以及是否誤刪根本沒有的邊，也有把這個 check 的檔案傳給幾個同學並請他們試用看看有沒有 bug，但目前還不知道這個 checker 到底會不會有問題 XD，只能說我跑的結果丟去 checker 是沒有問題的。爲了展現誠意，把 checker 丟在後面給看報告的人過目(其實可以看前面註解就好)，我覺得我在做的事情很符合工程師該做的事情，設計了一個東西不知道其對或錯，因此再設計一個東西來驗證其正確性，感覺蠻酷的！

```cpp
// this is the checker for PA3, NTUEE, algorithm, 2020 fall
// produced by b07901021 SHIH-HSUAN PAN in 2020/12/20
// compile:    g++ correct.cpp -o correct
// run:        ./correct <original_input_file> <output_file_you_produced>
//
// (1) the program will first read the original input file and construct graph
//     (only consider the edge without considering the weight)
// (2) then the output file you produced will be readed
// (3) according to your output file, some edges in the graph will be removed
//     also, the program will check whether the edge you removed is in the original input file
//     however, the order of undirected edges doesnt follow the order in the original input file
// (4) then the program will run DFS-smiliary algorithm to check the existence of cycles
// (5) treating all directed edges as undirected, then start traverse all vertices to see
//     whether all vertices are weakly connected
//
// summary:
// the program will check
// (1) whether you delete some edges not in the original input file
// (2) whether all vertices are weakly connected
// (3) whether there is any cycle in your graph aftering removing edges you specified
//
// if there is any problem about this checker, please send email to me to discuss about it
// email: b07901021@ntu.edu.tw

#include <cstring>
#include <fstream>
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

class Graph
{
    public:
        Graph();
        Graph(char , int);
        void E_construct(int, int);
        void E_delete(int, int);
        void DFS_cycle_detecting_check();
        void DFS_visit_check(int, vector<char>&, vector<int>&);
        void weakly_connected_check();
        void traverse(int, vector<char>&, vector<vector<int> >&);
    private:
        char type;
        int n;
        vector<vector<int> > adj;
};

Graph::Graph(char t, int num){
    type = t;
    n = num;
    for(int i = 0; i < n; i++){
        vector<int> blank;
        adj.push_back(blank);
    }
}
void Graph::E_construct(int u, int v){
    adj[u].push_back(v);
    if(type == 'u') adj[v].push_back(u);
```

```
}

void Graph::E_delete(int u, int v){
    if(type == 'd'){
        cout << "deleting edge(" << u << ", " << v << ")" << endl;
        vector<int>::iterator it = find (adj[u].begin(), adj[u].end(), v);
        if (it != adj[u].end()) adj[u].erase(it);
        else cout << "wrong!! your output file deletes edge not in original input file " << endl;
    }
    else{
        vector<int>::iterator it1 = find (adj[u].begin(), adj[u].end(), v);
        vector<int>::iterator it2 = find (adj[v].begin(), adj[v].end(), u);
        if (it1 != adj[u].end() && it2 != adj[v].end()) {
            adj[u].erase(it1);
            adj[v].erase(it2);
            // cout << "deleting edge(" << u << ", " << v << ")" << endl;
        }
        else{
            cout << "wrong!! your output file deletes edge not in original input file" << endl;
        }
    }
}

void Graph::DFS_cycle_detecting_check(){
    vector<char> color(n, 'w');
    vector<int> pi(n, -1);
    for(int i = 0; i < n; i++){
        if(color[i] == 'w')
            DFS_visit_check(i, color, pi);
    }
}
void Graph::DFS_visit_check(int u, vector<char>& color, vector<int>& pi){
    color[u] = 'g';
    for(int i = 0; i < adj[u].size(); i++){
        int v = adj[u][i];
        if(color[v] == 'w'){
            pi[v] = u;
            DFS_visit_check(v, color, pi);
        }
        else if(color[v] == 'g' && type == 'd'){
            cout << "wrong!! there are cycles in your graph" << endl;
        }
        else if(color[v] == 'g' && type == 'u'){
            if(v != pi[u])
                cout << "wrong!! there are cycles in your graph" << endl;
        }
    }
    color[u] = 'b';
}
void Graph::weakly_connected_check(){
    vector<vector<int> > adj_matrix;
    vector<char> color(n, 'w');
    for(int i = 0; i < n; i++){
        vector<int> temp(n, 0);
        adj_matrix.push_back(temp);
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < adj[i].size(); j++){
            int u = i;
            int v = adj[i][j];
            adj_matrix[u][v] = 1;
            adj_matrix[v][u] = 1;
        }
    }
    traverse(0, color, adj_matrix);
    for(int i = 0; i < n; i++){
        if(color[i] == 'w'){
            cout << "wrong!! all vertices are not weakly connected" << endl;
            break;
        }
    }

}
void Graph::traverse(int u, vector<char>& color, vector<vector<int> >& m){
    color[u] = 'g';
    for(int i = 0; i < m[u].size(); i++){
        if(m[u][i] == 1){
            int v = i;
            if(color[v] == 'w'){
                traverse(v, color, m);
            }
        }
    }
    color[u] = 'b';
}

int main(int argc, char* argv[])
{
    if(argc != 3) {
       cout << "wrong arguments" << endl;
       return 0;
    }

    //////////// read the input file ////////////

    cout << "reading..." << endl;
    char buffer[200];
    fstream fin(argv[1]);
    fstream fin_d(argv[2]);
    char type; // 'u' means undirected edge
               // 'd' means directed edge
    fin >> type;
    int m, n; // n: total number of vertices
              // m: total number of edges
    fin >> n >> m;

    vector<int> data;
    int num;
    while (fin >> num)
        data.push_back(num); // data[3i] will be the start point of an edge.
                             // data[3i+1] will be the end point of an edge.
                             // data[3i+2] will be the weight.

    cout << "processing original input file..." << endl;
    Graph G(type, n);
    for(int i = 0; i < m; i++){
        int u = data[3*i];
        int v = data[3*i+1];
        int w = data[3*i+2];
        G.E_construct(u, v);
    }

    cout << "processing output file..." << endl;
    vector<int> data_d;
    int num_d, total;
    fin_d >> total;
    cout << "sum of deleted edge weight = " << total << endl;
    while (fin_d >> num_d){
        data_d.push_back(num_d);
    }
```

```cpp
    for(int i = 0; i < data_d.size(); i+=3){
        int u = data_d[i];
        int v = data_d[i+1];
        int w = data_d[i+2];
        G.E_delete(u, v);
    }

    cout << "checking cycle existance" << endl;
    G.DFS_cycle_detecting_check();

    cout << "checking connected" << endl;
    G.weakly_connected_check();

    cout << "checking finished" << endl;
}

```