PA3

B06901061 電機三 王廷峻

Directory Structure

```
.
├── README.md   # spec
├── case1.txt   # test case
├── case2.txt
├── case3.txt
├── case4.txt
├── graph.cpp   # graph class
├── graph.h
├── main.cpp    # cli
├── makefile
├── script.sh   # script to execute
├── vertex.cpp  # node class
└── vertex.h
```

Main.cpp

將指令分成三類：

1. `lf <case.txt>`：allocate 新的記憶體儲存 graph，使用 `Graph::buildForwardTables` 建立所有 routers 的 routing tables。
2. `of`：輸出 routing tables 至 `<case>_out1.txt`
3. `rm r<number-of-router>`：Part2 詳述

```cpp
int main(int argc, char* argv[]){
    // ...
    while(getline(cin, cmd)){
        stringstream ss(cmd);
        string oprt, arg;
        ss >> oprt >> arg;

        if(oprt == "lf"){
            delete graph;       // avoid memory leaking

            filename = arg.substr(0, arg.length()-4);
            vector<vector<int> > cost_table = readfile(arg);
            graph = new Graph(cost_table.size(), cost_table);
            graph -> buildForwardTables();
            graph -> print();
        }
```

```
        else if(oprt == "of"){
            if(graph == NULL){
                cout << "Please load file first !\n";
            }
            else{
                string outfilename = filename + "_out1.txt";
                graph -> writeResult(outfilename);
            }
        }
        else if(oprt == "rm"){
            if(graph == NULL){
                cout << "Please load file first !\n";
            }
            else{
                int deleteRouterID = stoi(arg.substr(1)) - 1;
                graph -> deleteRouter(deleteRouterID);
                graph -> buildForwardTables();
                graph -> print();
            }
        }
        else{
            cout << "Invalid command !!!\n";
            delete graph;
            break;
        }
        cout << "RouterMgr>> ";
    }
    return 0;
}
```

graph.cpp/h

```
class Graph{
    public:
        friend ostream& operator << (ostream&, const Graph&);

        Graph(int n_vertex, vector<vector<int> > matrix):
_n_vertex(n_vertex), _tmp_src_next_vertex(n_vertex, -2){
            _graph.reserve(n_vertex);
            _forward_table.reserve(n_vertex);
            buildGraph(matrix);
        }
        // Print the routing tables
        void print();
        // Given topological matrix and   build graph
        void buildGraph(vector<vector<int> >);
        // Output routing tables to file
        void writeResult(string);
        // Given router ID and delete
```

```
        void deleteRouter(unsigned);
        // Build routing tables
        void buildForwardTables();

        // Auxiliary function for routing tables building
        // Given router ID as the source, calculate costs of each router
        void shortest_path(unsigned);
        // Given router ID, build its table
        void buildForwardTable(unsigned);
        // Find next step
        unsigned recursiveFindPrevVert(unsigned, unsigned);

        void setVertexToDefault();
        void setGraphToDefault();

    private:
        // array of routers
        vector<Vertex>                      _graph;
        // array of routing tables
        vector<vector<tuple<int, int> >>    _forward_table;
        // array of next step vertex with respect to a certain source
        vector<int>                         _tmp_src_next_vertex;
        int                                 _n_vertex;
};
```

Vertex.cpp/h

```
class Vertex{
    public:
        friend ostream& operator << (ostream& os, const Vertex& v);

        Vertex(unsigned id): _id(id), _cost(UINT_MAX), _visit(false),
_from(0){}
        unsigned                            getID()             const;
        unsigned                            getCost()           const;
        bool                                getVisit()          const;
        int                                 getWeight(Vertex*)  const;
        Vertex*                             getFromVertex()     const;
        vector<tuple<Vertex*, int> >        getOutEdge()        const;

        void                                setCost      (unsigned);
        void                                setVisit     (bool);
        void                                setFromVertex(Vertex*);

        bool                                isVoid();

        void                                buildOutEdge (Vertex*, int);
        void                                removeOutEdge(Vertex*);
        vector<Vertex*>                     relax        ();
```

```
22
23      private:
24          unsigned                                _id;
25          // cost to arrive the vertex
26          unsigned                                _cost;
27          // whether the vertex is in shortest tree
28          bool                                    _visit;
29          // previous vertex
30          Vertex*                                 _from;
31          vector<tuple<Vertex*, int> >            _outEdge;
32  };
33
34  #endif
```

Part1

How to build routing tables?

1. Call `Graph::buildForwardTables()`: 建立 routing tables
2. For each vertex call `Graph::shortest_path(unsigned)`：計算 source 到每個 router 的距離（i.e., `Vertext::_cost`，以及要走到每個 router 前的 vertex（i.e., `Vertext::_from`）
3. For each vertex call `Graph::buildForwardTable(unsigned)`：利用 `Vertext::_from` 計算從 source 到每個 router 需要選擇哪條 output edge 以達最短路徑。組合上述資訊和 `Vertext::_cost` 可建構 source 的 routing table
4. For each vertex call `Graph::setVertexToDefault()`：將 `Vertext::_cost` `Vertext::_from` 等資訊改回預設值，以便下一個 router 做 shortest path 搜尋。

Graph::buildForwardTables()

```
1  void Graph:: buildForwardTables(){
2      for( int i = 0 ; i < _graph.size() ; ++i ){
3          if(!_graph[i].isVoid()){
4              shortest_path(i);
5              buildForwardTable(i);
6              setVertexToDefault();
7          }
8      }
9  }
```

Graph::shortest_path(unsigned)

為了將 time complexity 壓在 $O((E+V)*\log V)$，採用 STL 的 Priority Queue (pq)。

1. 首先將 source 的 `Vertex::_cost` 設定為 0、`Vertex::_from` 設定為自己，並放入 pq
2. 每次挑選 `Vertex::_cost` 最小的 vertex 出來，並且對此 vertex 周圍的 vertex 做 relaxation，如果有被成功更新 `Vertex::_cost` 則會回傳至 `v_relaxed` 並且塞入 pq。此做法可能會將同樣的 vertex 重複塞入 pq，因此需透過 `Vertex::_visit` 判定是否已經在 shortest path tree 中
3. 最後將 `Vertex::_visit` 設定為 True，代表已經在 shortest path tree 中

```cpp
void Graph:: shortest_path(unsigned src){
    priority_queue<Vertex*, vector<Vertex*>, Comparator> pq;

    _graph[src].setCost(0);
    _graph[src].setFromVertex(&_graph[src]);
    pq.push(&_graph[src]);

    while(!pq.empty()){
        Vertex* u = pq.top();
        pq.pop();

        if(u -> getVisit())
            continue;

        vector<Vertex*> v_relaxed = u -> relax();
        for( int i = 0, n = v_relaxed.size() ; i < n ; ++i ){
            pq.push(v_relaxed[i]);
        }
        u -> setVisit(true);
    }
}
```

Graph::buildForwardTable(unsigned)

利用 `Graph::recursiveFindPrevVert(unsigned, unsigned)` 找到source 走到目標 id 需要走到哪個 neigherbor vertex，並和 `Vertex::_cost` 組合成 routing table (i.e., `forward_table`)

```cpp
void Graph:: buildForwardTable(unsigned src){
    for( int i = 0 ; i < _graph.size() ; ++i ){
        if(!_graph[i].isVoid()){
            if(_tmp_src_next_vertex[i] == -2){
                recursiveFindPrevVert(i, src);
            }
        }
    }

    vector<tuple<int, int> > forward_table(_graph.size(), make_tuple(-1, -2));
    for( int i = 0 ; i < _graph.size() ; ++i ){
        if(!_graph[i].isVoid()){
            forward_table[i] = make_tuple(_graph[i].getCost(), _tmp_src_next_vertex[i]);
        }
    }
    _forward_table[src] = forward_table;
}
```

Graph::recursiveFindPrevVert(unsigned, unsigned)

給定目標 id 和 source，利用建表 `_tmp_src_next_vertex` 和 `Vertex::_from` 遞迴搜尋 source 走到目標 id 需要走到哪個 neigherbor vertex。

```
unsigned Graph:: recursiveFindPrevVert(unsigned id, unsigned src){
    // Exist next step
    if( _tmp_src_next_vertex[id] != -2 ){
        return _tmp_src_next_vertex[id];
    }
    // Next step = itself
    if( id == src ){
        _tmp_src_next_vertex[id] = id;
        return _tmp_src_next_vertex[id];
    }
    // Next step = a neighbor of source, which views source as ancestor
    vector<tuple<Vertex*, int> > outEdge = _graph[src].getOutEdge();
    for( int i = 0 ; i < outEdge.size() ; i++ ){
        unsigned out_id = get<0>(outEdge[i]) -> getID();
        unsigned out_from_id = get<0>(outEdge[i]) -> getFromVertex() -> getID();
        if(out_id == id and out_from_id == src ){
            _tmp_src_next_vertex[id] = id;
            return id;
        }
    }
    // Exist _from vertex
    Vertex* self = &_graph[id];
    if (self -> getFromVertex() != NULL)
        _tmp_src_next_vertex[id] = recursiveFindPrevVert(self -> getFromVertex() -> getID(), src);

    return _tmp_src_next_vertex[id];
}
```

Part2

How to delete a specified router?

1. call `Graph::deleteRouter(unsiged)`：將此 vertex 的 neighbors 的 outEdge 中把 此 vertex 移除（呼叫 `Vertex::removeOutEdge(Vertex*)`）
2. 原先的位置將改有 void vertex（i.e., Vertex with id = UNIT_MAX） 取代，之所以不直接將 vertex erase 掉的原因在於：希望維持使用 hash 的形式，透過 id 在 $O(1)$ 的時間內找到對應的 vertex。之後在取用 `Graph::_graph` 前只要利用 `Vertex::isVoid()` 判斷是否需要跳過即可。

Graph::deleteRouter(unsigned)

```
void Graph:: deleteRouter(unsigned id){
    cout << "Delete Target: " << id << endl;

```

```
    for( int i = 0, n = _graph.size() ; i < n ; ++i ){
        if(!_graph[i].isVoid()){
            if(_graph[i].getID() == id){
                Vertex* rm = &_graph[i];

                for( int j = 0, m = rm -> getOutEdge().size() ; j < m ; ++j ){
                    Vertex* neighbor = get<0>(rm -> getOutEdge()[j]);
                    neighbor -> removeOutEdge(rm);
                }

                _graph[i] = Vertex(UINT_MAX);
                setGraphToDefault();
                break;
            }
        }
    }
}
```

Vertex::removeOutEdge(Vertex*)

```
void Vertex:: removeOutEdge(Vertex* vptr){
    vector<tuple<Vertex*, int> >::iterator cur = _outEdge.begin();
    while(cur != _outEdge.end()){
        if(get< 0 >(*cur) == vptr){
            cout << "Successfully remove [" << vptr -> getID() + 1 << "]
";
            cout << "from [" << getID() + 1 << "]\n";
            _outEdge.erase(cur);
            break;
        }
        ++cur;
    }
}
```