資料結構與物件導向程式設計
HW2 報告

110550088 李杰穎

May 28, 2022

1 Part 1: Directed Graph

此部分是給定一個有向圖，這個有向圖可能是無環也有可能是有環的，所以我們必須先判斷給定的圖是否有環。若有環，則我們必須把圖上的強連通分量找出來。若無環，則我們要印出這張有向圖的拓樸排序。

1.1 Implementation Detail

以下是 `PartI.h`，內有 Part 1 會用到的所有函數及變數。

Code 1: `PartI.h`

```
#ifndef PARTI_H
#include "SolverBase.h"
#include <algorithm>
#include <vector>
#include <map>
#include <iostream>
#include <fstream>

using namespace std;

class PartI : public SolverBase
{
int n, m, cnt;
vector<vector<pair<int, int>>> graph; // Declare an adjacency list to save the graph (pair fisrt:
  ↪ vertex second: weight)
vector<vector<pair<int, int>>> rev_graph; // Declare an adjacency list to save the "reverse" graph
  ↪ (pair fisrt: vertex second: weight)
map<pair<int, int>, int> scc_graph;  // Use map to save edges and theirs weights
vector<pair<int, vector<int>>> scc_vertex; // Save the id of each vertices in every SCC
vector<int> order; // Save topological order
vector<int> scc; // Save the id of corresponding SCC of each vertex
```

```cpp

map<int, int> finish; // Store the status of vertex when traversing
bool isAyclic = true; // To store that this graph is acyclic or not

public:
    void read(std::string); // Read input from file
    void solve(); // Main solve function
    void write(std::string); // Write output to file
    void dfs(int); // DFS
    void scc_dfs(int, int); // Run dfs to get SCC
    void scc_revdfs(int); // Run reverse dfs to get SCC
    void kosaraju(); // The main part of Kosaraju's Algorithm
    void buildGraph(); // Build coarse graph
    void buildGraphDFS(int); // Use dfs to build the coarse graph
};

#define PARTI_H
#endif
```

以下為 Part 1 的具體實作，其細節我都已經寫在註解中。主要就是先利用 DFS 判斷圖是否有環，若判斷無環，則根據先前 DFS 的結果輸出 topological ordering。若判斷有環，則會利用 Kosaraju 演算法來找出強連通分量。

Kosaraju’s algorithm 主要演算步驟如下：

1. 對有向圖 G 取反圖，得到 G 的反向圖 G^R
2. 利用 DFS 找出 G^R 的拓樸排序
3. 對 G 按照拓樸排序進行 DFS
4. 在同一次 DFS 跑到的點都在同一個強連通分量中

以下就是 Part 1 實作的所有程式碼，程式碼的解釋部分我已經寫在註解。

Code 2: `PartI.cpp`

```cpp
#include ”PartI.h”

using namespace std;

bool operator<(const pair<int, int>& a, const pair<int, int>& b){
    return (a.first < b.first) || (a.first == b.first && a.second < b.second);
}

void PartI::read(string file) {
    cout << ”Part I reading...” << endl;
    ifstream ifs(file); // Use input file stream to read input from file
    ifs >> n >> m; // Get # of vertices n and # of edges m
    // Resize the vector
    graph.resize(n);
    rev_graph.resize(n);
```

```
    scc.resize(n);
    int u, v, w;

    // Build graph
    for(int i=0;i<m;i++){
        ifs >> u >> v >> w;
        graph[u].push_back({v, w});
        rev_graph[v].push_back({u, w});
    }
    // Close the ifstream
    ifs.close();

    // Because we need to traverse from smallest index to the largest, thus we need to sort the adj. list
    for(auto i: graph){
        sort(i.begin(), i.end());
    }
}

void PartI::solve() {
    cout << ”Part I solving...” << endl;
    for(int i=0;i<n;i++){
        // Run DFS for not yet traversed vertex
        if(finish[i] == 0){
            dfs(i);
        }
    }

    if(isAyclic){
        // If is acylic, then reverse the order to get real topological ordering
        reverse(order.begin(), order.end());
        return;
    }

}

void PartI::write(string file) {
    cout << ”Part I writing...” << endl;
    ofstream ofs(file); // use output filestream to write output to file
    if(isAyclic){
        // Write the topological ordering to file
        for(auto i: order){
            ofs << i << ” ”;
        }
        ofs << endl;
        // After writing the to the file, directly exit the function
        return;
    }
    // If not ayclic, then run Kosaraju's algorithtm to find SCC
    kosaraju();
    // After running Kosaraju's algorithtm, vertex will have a label which indicates that which SCC it belongs to.
    // This information will save in vector<int> scc.
    // cnt is # of SCC in the given graph.
    scc_vertex.resize(cnt, {1e9, {}}); // scc_vertex will save the minimum index and every vertex in each SCC.
    // Iterate all vertices in the graph.
    for(int i=0;i<n;i++){
        int u = scc[i];
```

```
        scc_vertex[u].first = min(scc_vertex[u].first, i);
        scc_vertex[u].second.push_back(i);
    }
    sort(scc_vertex.begin(), scc_vertex.end()); // Sort the SCCs based on the minimum index.

    // Relabel the vertices to match the new index value of every SCC
    for(int i=0;i<cnt;i++){
        for(auto j: scc_vertex[i].second){
            scc[j] = i;
        }
    }

    // Build coarse graph
    buildGraph();
    // After running the above function, we will get the coarse graph and save as map.

    ofs << cnt << ” ” << scc_graph.size() << endl; // Output the # of vertcies and edges

    // Because map will sort by key, we don't need to sort the map.
    for(auto i: scc_graph){
        ofs << i.first.first << ” ” << i.first.second << ” ” << i.second << endl;
    }

    // Close output filestream
    ofs.close();
    return;
}

void PartI::dfs(int v){
    // Back edge, cyclic
    if(finish[v] == 1){
        isAyclic = false;
        return;
    }

    // forward edge or cross edge
    if(finish[v] == 2){
        return;
    }

    // mark 1
    finish[v] = 1;
    for(auto i: graph[v]){
        dfs(i.first);
    }

    // mark 2
    finish[v] = 2;
    order.push_back(v);
}

void PartI::scc_revdfs(int v){
    // DFS runs on reverse graph
    finish[v] = 1;
    for(auto i: rev_graph[v]){
        if(finish[v] == 0){
            scc_revdfs(i.first);
        }
    }
```

```
    order.push_back(v);
}

void PartI::scc_dfs(int cur, int s){
    // DFS runs on graph and lable the vertices
    scc[cur] = s;
    for(auto i: graph[cur]){
        int u = i.first;
        if(scc[u] == -1){
            scc_dfs(u, s);
        }
    }
}

void PartI::kosaraju(){
    order.clear();  // Clear the original order
    finish.clear(); // Clear the vector
    scc.resize(n);  // Resize the vector that store vertex belongs to SCC
    fill(scc.begin(), scc.end(), -1); // Fill the vector with -1
    for(int i=0;i<n;i++){
        if(finish[i] == 0){ // If not yet traversed, then run DFS on reverse graph.
            scc_revdfs(i);
        }
    }
    cnt = 0; // Store the # of SCC
    reverse(order.begin(), order.end());
    // Use topological ordering of reverse graph to get SCC
    for(auto i: order){
        if(scc[i] == -1){    // Not in any SCC
            scc_dfs(i, cnt); // Run DFS
            cnt ++;
        }
    }
}

void PartI::buildGraphDFS(int v){
    finish[v] = 1;
    for(auto i: graph[v]){ // Iterate all edges that connect to vertex v
        int u = i.first;
        if(scc[v] != scc[u]){
            scc_graph[{scc[v], scc[u]}] ++; // If two vertices belongs to different SCC, then this
            ↪  edge will appear in coarse graph and weight will plus 1
        }
        if(finish[u] == 0){ // If not yet visited, then run DFS
            buildGraphDFS(u);
        }
    }
}

void PartI::buildGraph(){
    finish.clear(); // Clear the vector that records visited vertices
    for(int i=0;i<n;i++){
        if(finish[i] == 0){
            buildGraphDFS(i);
        }
    }
}
```

1.2 Discussion

1.2.1 Time Complexity

此演算法時間複雜度的來源主要是多次 DFS，而一次 DFS 的時間複雜度為 $O(V+E)$，其中 V 為點的數量、E 為邊的數量。而 Kosaraju 演算法的複雜度也只是 2 次 DFS。其中還有用 map 維護 `finish`，其插入和查詢的總時間複雜度為 $O(V \log V)$。故總時間複雜度應為 $O(V + V \log V + E)$。

1.2.2 Your Discovery

在本次作業中，我發現可以僅需要一次 DFS 就可以得到 topological ordering，且 DFS 的起始點不一定要是入度為 0 的點，從任一點開始，只要 traverse 所有點，就可以得到正確的 topological ordering。

1.2.3 Which is the better algorithm in which condition

事實上，其實有另一個演算法可以計算強連通分量，此演算法即為 Tarjan 演算法。Tarjan 演算法雖然在時間複雜度上與 Kosaraju 演算法相同，但其常數部分差了一倍。Tarjan 演算法只需要一次 DFS 即可以找到強連通分量。但 Tarjan 演算法實作上比較麻煩，故在本次作業中，我仍採用 Kosaraju 演算法。

1.2.4 Encountered Problems

一開始在寫作業時沒有很理解 `read` 函數中 string 引數所代表的意義，後來查看 `main.cpp` 後才發現原來是檔名，於是想到可以用 C++ 內建的 `fstream` 來開啟檔案，並利用與 `cin, cout` 相同的方式來將變數輸入進來，也可以將變數輸出到檔案中。

2 Part 2: Shortest Path

在此部分中，我們需要實作兩種圖上最短路演算法，分別為 Dijkstra 演算法及 Bellman-Ford 演算法。

Dijsktra 演算法僅能處理無負邊權的圖，而 Bellman-Ford 演算法可以處理負邊權的圖，也可以偵測出圖上的負環。

在本次作業中，Dijkstra 演算法的部分我們會將輸入的邊權取絕對值，使圖上不會出現負邊權的情況。

2.1 Implementation Detail

以下為 `PartII.h`，有本部分會用到的函數及變數。

Code 3: `PartII.h`

```cpp
#ifndef PARTII_H
#include ”SolverBase.h”
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;

class PartII : public SolverBase
{
    int n, m; // # of vertices and edges
    bool hasNegCyc = false; // has negative cycle in the graph or not
    vector<vector<pair<int, int>>> adj; // Declare an adjacency list to save the graph (pair fisrt: vertex second: weight)
    vector<vector<pair<int, int>>> abs_adj; // Declare an adjacency list to save the graph (pair fisrt: vertex second: abs(weight))
    vector<int> d, abs_d; // The shortest distance from node 0 to every other vertices
public:
    void read(std::string); // Read input from file
    void solve(); // Main solve fucntion
    void write(std::string); // Write the output to file
    void dijkstra(int); // Dijkstra's Algorithm
    bool bellmanFord(int); // Bellman-Ford's Algorithm
};

#define PARTII_H
#endif
```

以下就是 Part 2 的所有程式碼，解釋的部分已放在註解中。

Code 4: `PartII.cpp`

```cpp
#include ”PartII.h”

#define INF 1e9

using namespace std;

void PartII::read(string file) {
    cout << ”Part II reading...” << endl;
    ifstream ifs(file); // Decalre a input filestream to get input from file
    ifs >> n >> m; // input # of vertices and edges
    // Resize the vectors
    d.resize(n);
    abs_d.resize(n);
    adj.resize(n);
```

```
    abs_adj.resize(n);

    // Build graph from input
    int u, v, w;
    for(int i=0;i<m;i++){
        ifs >> u >> v >> w;
        adj[u].push_back({v, w});
        abs_adj[u].push_back({v, abs(w)});
    }

    ifs.close();
}
void PartII::solve() {
    cout << "Part II solving..." << endl;
    dijkstra(0); // Run from Dijkstra's algorithm from node 0

    // Run Bellman-Ford algorithm from node 0.
    // And if function return false, it means there is negative cycle in the graph
    hasNegCyc = !bellmanFord(0);
}

void PartII::write(string file) {
    cout << "Part II writing..." << endl;
    ofstream ofs(file);
    ofs << abs_d[n-1] << endl; // Output the Dijkstra algorithm's output

    if(hasNegCyc){
        ofs << "Negative loop detected!" << endl; // If has negative cycle, then output "Negative
        ↪  loop detected!"
    } else {
        ofs << d[n-1] << endl; // Otherwise, output the Bellman-Ford algorithm's output
    }

    ofs.close(); // Close output file stream
}


void PartII::dijkstra(int src){
    fill(abs_d.begin(), abs_d.end(), INF); // Fill the distance vector w/ INF value
    abs_d[src] = 0; // The distance from node 0 to node 0 is 0

    // Use priority_queue to get the vertex with the smallest distance from node 0 in every step
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; // first:
    ↪  weight, second: vertex

    // Push the initial node 0 to pq
    pq.push({0, 0});

    // If pq is not empty, keep running the relax process
    while(!pq.empty()){
        auto edge = pq.top();
        pq.pop();
        int u = edge.second;
        for(auto i: abs_adj[u]){
            int w = i.first;
            int v = i.second;
            int alt = abs_d[u] + i.first; // Alternate path's distance

            // If alternate distance less than current distance save in distance vector
```

```cpp
            if(alt < abs_d[v]){
                abs_d[v] = alt;    // Update the distance in distance vector
                pq.push({alt, v}); // Update the new distance to pq
            }
        }
    }
}

bool PartII::bellmanFord(int src){
    fill(d.begin(), d.end(), INF); // Fill the distance vector w/ INF value
    d[src] = 0; // The distance from node 0 to node 0 is 0

    // Run n-1 times of relaxation process
    for(int i=0;i<n-1;i++){
        for(int u=0;u<n;u++){
            for(auto k: adj[u]){
                int v = k.first;
                int w = k.second;
                int alt = d[u] + w;

                // If alternate distance less than current distance save in distance vector
                if(alt < d[v]){
                    d[v] = alt; // Update the distance in distance vector
                }
            }
        }
    }


    for(int u=0;u<n;u++){
        for(auto k: adj[u]){
            int v = k.first;
            int w = k.second;
            int alt = d[u] + w;
            if(alt < d[v]){
                // If there still have edge that can be relaxed in nth times of relaxation process.
                // Then there exists a negative cycle
                return false;
            }
        }
    }

    return true;
}
```

2.2 Discussion

2.2.1 Time Complexity

以 Dijkstra 演算法來說，一般來說其時間複雜度為 $O(V^2)$，其中 V 為點的數量。但因為我在本次作業中使用 `priority_queue` 來加速取出最小值的過程，而 `priority_queue` 本質上是一個 Binary Heap，所以 Dijsktra 演算法的時間複雜度降至 $O((E+V)\log V)$。

至於 Bellman-Ford 演算法，其時間複雜度即為 $O(E \cdot V)$。

2.2.2 Your Discovery

在實作 Dijkstra 演算法中的 relaxation process 時，雖然實際上我們要做的應該是去更新 `priority_queue` 中的已存在的節點，但實際上，我們只要把更小的值推進去就好，因為 `priority_queue` 永遠會把較小的值拿出來，等到較大值拿出來時，這個較大的值也不會對 distance vector 產生更新。

2.2.3 How to find path of the shortest path?

我們可以在程式中額外宣告一個 `parent` 陣列，此陣列是要記錄最短路徑中，每一個節點的前一個節點為何，我們只需要在 relaxing 的時候更新這個陣列即可。跑完兩種演算法後，我們從終點 backtracking 回到起點，最後就可以找到最短路徑了。

2.2.4 How Bellman-Ford Algorithm detect negative cycle?

因為在圖中，我們可以發現最短路徑最多就經過 $V-1$ 條邊，所以我們對所有邊 relaxing $V-1$ 次必定能找到最短路徑。但如果圖內有 negative loop 則此張圖就沒有最短路徑，因為我們可以透過重複經過此 negative loop 來使最短距離越來越小。而 Bellman-Ford 演算法即是在跑完 $V-1$ 迭代過程後再額外跑一次迭代，如果有任何一條邊可以被 relaxing 到，則此圖中就有 negative loop。

2.2.5 Which is the better algorithm in which condition?

如上所述 Dijkstra 演算法無法處理有負邊的情況，也無法判斷是否有負環。而 Bellman-Ford 演算法可以處理負邊也可以處理負環。但 Dijkstra 演算法的整體複雜度較佳，故如果確定圖上沒有負邊，則我們應該優先考慮使用 Dijkstra 演算法，但如果不確定是否有負邊，則我們應該使用 Bellman-Ford 演算法。