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`

```
1  #ifndef PARTI_H
2  #include "SolverBase.h"
3  #include <algorithm>
4  #include <vector>
5  #include <map>
6  #include <iostream>
7  #include <fstream>
8
9  using namespace std;
10
11 class PartI : public SolverBase
12 {
13 int n, m, cnt;
14 vector<vector<pair<int, int>>> graph; // Declare an adjacency list to save the graph (pair fisrt:
   ↪  vertex second: weight)
15 vector<vector<pair<int, int>>> rev_graph; // Declare an adjacency list to save the "reverse" graph
   ↪  (pair fisrt: vertex second: weight)
16 map<pair<int, int>, int> scc_graph;  // Use map to save edges and theirs weights
17 vector<pair<int, vector<int>>> scc_vertex; // Save the id of each vertices in every SCC
18 vector<int> order; // Save topological order
19 vector<int> scc; // Save the id of corresponding SCC of each vertex
```

```
20
21  map<int, int> finish; // Store the status of vertex when traversing
22  bool isAyclic = true; // To store that this graph is acyclic or not
23
24  public:
25      void read(std::string); // Read input from file
26      void solve(); // Main solve function
27      void write(std::string); // Write output to file
28      void dfs(int); // DFS
29      void scc_dfs(int, int); // Run dfs to get SCC
30      void scc_revdfs(int); // Run reverse dfs to get SCC
31      void kosaraju(); // The main part of Kosaraju's Algorithm
32      void buildGraph(); // Build coarse graph
33      void buildGraphDFS(int); // Use dfs to build the coarse graph
34  };
35
36  #define PARTI_H
37  #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`

```
1   #include "PartI.h"
2
3   using namespace std;
4
5   bool operator<(const pair<int, int>& a, const pair<int, int>& b){
6       return (a.first < b.first) || (a.first == b.first && a.second < b.second);
7   }
8
9   void PartI::read(string file) {
10      cout << "Part I reading..." << endl;
11      ifstream ifs(file); // Use input file stream to read input from file
12      ifs >> n >> m; // Get # of vertices n and # of edges m
13      // Resize the vector
14      graph.resize(n);
15      rev_graph.resize(n);
```

```
16      scc.resize(n);
17      int u, v, w;
18
19      // Build graph
20      for(int i=0;i<m;i++){
21          ifs >> u >> v >> w;
22          graph[u].push_back({v, w});
23          rev_graph[v].push_back({u, w});
24      }
25      // Close the ifstream
26      ifs.close();
27
28      // Because we need to traverse from smallest index to the largest, thus we need to sort the
   ↪  adj. list
29      for(auto i: graph){
30          sort(i.begin(), i.end());
31      }
32  }
33
34  void PartI::solve() {
35      cout << "Part I solving..." << endl;
36      for(int i=0;i<n;i++){
37          // Run DFS for not yet traversed vertex
38          if(finish[i] == 0){
39              dfs(i);
40          }
41      }
42
43      if(isAyclic){
44          // If is acylic, then reverse the order to get real topological ordering
45          reverse(order.begin(), order.end());
46          return;
47      }
48
49  }
50
51  void PartI::write(string file) {
52      cout << "Part I writing..." << endl;
53      ofstream ofs(file); // use output filestream to write output to file
54      if(isAyclic){
55          // Write the topological ordering to file
56          for(auto i: order){
57              ofs << i << " ";
58          }
59          ofs << endl;
60          // After writing the to the file, directly exit the function
61          return;
62      }
63      // If not ayclic, then run Kosaraju's algorithtm to find SCC
64      kosaraju();
65      // After running Kosaraju's algorithtm, vertex will have a label which indicates that which SCC
   ↪  it belongs to.
66      // This information will save in vector<int> scc.
67      // cnt is # of SCC in the given graph.
68      scc_vertex.resize(cnt, {1e9, {}}); // scc_vertex will save the minimum index and every vertex
   ↪  in each SCC.
69      // Iterate all vertices in the graph.
70      for(int i=0;i<n;i++){
71          int u = scc[i];
```

```cpp
72              scc_vertex[u].first = min(scc_vertex[u].first, i);
73              scc_vertex[u].second.push_back(i);
74          }
75      sort(scc_vertex.begin(), scc_vertex.end()); // Sort the SCCs based on the minimum index.
76
77      // Relabel the vertices to match the new index value of every SCC
78      for(int i=0;i<cnt;i++){
79          for(auto j: scc_vertex[i].second){
80              scc[j] = i;
81          }
82      }
83
84      // Build coarse graph
85      buildGraph();
86      // After running the above function, we will get the coarse graph and save as map.
87
88      ofs << cnt << " " << scc_graph.size() << endl; // Output the # of vertcies and edges
89
90      // Because map will sort by key, we don't need to sort the map.
91      for(auto i: scc_graph){
92          ofs << i.first.first << " " << i.first.second << " " << i.second << endl;
93      }
94
95      // Close output filestream
96      ofs.close();
97      return;
98  }
99
100 void PartI::dfs(int v){
101     // Back edge, cyclic
102     if(finish[v] == 1){
103         isAyclic = false;
104         return;
105     }
106
107     // forward edge or cross edge
108     if(finish[v] == 2){
109         return;
110     }
111
112     // mark 1
113     finish[v] = 1;
114     for(auto i: graph[v]){
115         dfs(i.first);
116     }
117
118     // mark 2
119     finish[v] = 2;
120     order.push_back(v);
121 }
122
123 void PartI::scc_revdfs(int v){
124     // DFS runs on reverse graph
125     finish[v] = 1;
126     for(auto i: rev_graph[v]){
127         if(finish[v] == 0){
128             scc_revdfs(i.first);
129         }
130     }
```

```cpp
131        order.push_back(v);
132 }
133
134 void PartI::scc_dfs(int cur, int s){
135      // DFS runs on graph and lable the vertices
136      scc[cur] = s;
137      for(auto i: graph[cur]){
138          int u = i.first;
139          if(scc[u] == -1){
140              scc_dfs(u, s);
141          }
142      }
143 }
144
145 void PartI::kosaraju(){
146      order.clear();  // Clear the original order
147      finish.clear(); // Clear the vector
148      scc.resize(n);  // Resize the vector that store vertex belongs to SCC
149      fill(scc.begin(), scc.end(), -1); // Fill the vector with -1
150      for(int i=0;i<n;i++){
151          if(finish[i] == 0){ // If not yet traversed, then run DFS on reverse graph.
152              scc_revdfs(i);
153          }
154      }
155      cnt = 0; // Store the # of SCC
156      reverse(order.begin(), order.end());
157      // Use topological ordering of reverse graph to get SCC
158      for(auto i: order){
159          if(scc[i] == -1){     // Not in any SCC
160              scc_dfs(i, cnt); // Run DFS
161              cnt ++;
162          }
163      }
164 }
165
166 void PartI::buildGraphDFS(int v){
167      finish[v] = 1;
168      for(auto i: graph[v]){ // Iterate all edges that connect to vertex v
169          int u = i.first;
170          if(scc[v] != scc[u]){
171              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
172          }
173          if(finish[u] == 0){ // If not yet visited, then run DFS
174              buildGraphDFS(u);
175          }
176      }
177 }
178
179 void PartI::buildGraph(){
180      finish.clear(); // Clear the vector that records visited vertices
181      for(int i=0;i<n;i++){
182          if(finish[i] == 0){
183              buildGraphDFS(i);
184          }
185      }
186 }
```

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);
```

```cpp
15      abs_adj.resize(n);

16
17      // Build graph from input
18      int u, v, w;
19      for(int i=0;i<m;i++){
20          ifs >> u >> v >> w;
21          adj[u].push_back({v, w});
22          abs_adj[u].push_back({v, abs(w)});
23      }
24
25      ifs.close();
26 }
27 void PartII::solve() {
28      cout << "Part II solving..." << endl;
29      dijkstra(0); // Run from Dijkstra's algorithm from node 0
30
31      // Run Bellman-Ford algorithm from node 0.
32      // And if function return false, it means there is negative cycle in the graph
33      hasNegCyc = !bellmanFord(0);
34 }
35
36 void PartII::write(string file) {
37      cout << "Part II writing..." << endl;
38      ofstream ofs(file);
39      ofs << abs_d[n-1] << endl; // Output the Dijkstra algorithm's output
40
41      if(hasNegCyc){
42          ofs << "Negative loop detected!" << endl; // If has negative cycle, then output "Negative
            ↪  loop detected!"
43      } else {
44          ofs << d[n-1] << endl; // Otherwise, output the Bellman-Ford algorithm's output
45      }
46
47      ofs.close(); // Close output file stream
48 }
49
50
51 void PartII::dijkstra(int src){
52      fill(abs_d.begin(), abs_d.end(), INF); // Fill the distance vector w/ INF value
53      abs_d[src] = 0; // The distance from node 0 to node 0 is 0
54
55      // Use priority_queue to get the vertex with the smallest distance from node 0 in every step
56      priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; // first:
        ↪  weight, second: vertex
57
58      // Push the initial node 0 to pq
59      pq.push({0, 0});
60
61      // If pq is not empty, keep running the relax process
62      while(!pq.empty()){
63          auto edge = pq.top();
64          pq.pop();
65          int u = edge.second;
66          for(auto i: abs_adj[u]){
67              int w = i.first;
68              int v = i.second;
69              int alt = abs_d[u] + i.first; // Alternate path's distance
70
71              // If alternate distance less than current distance save in distance vector
```

```
72        if(alt < abs_d[v]){
73            abs_d[v] = alt;     // Update the distance in distance vector
74            pq.push({alt, v}); // Update the new distance to pq
75        }
76     }
77   }
78 }
79
80 bool PartII::bellmanFord(int src){
81    fill(d.begin(), d.end(), INF); // Fill the distance vector w/ INF value
82    d[src] = 0; // The distance from node 0 to node 0 is 0
83
84    // Run n-1 times of relaxation process
85    for(int i=0;i<n-1;i++){
86        for(int u=0;u<n;u++){
87            for(auto k: adj[u]){
88                int v = k.first;
89                int w = k.second;
90                int alt = d[u] + w;
91
92                // If alternate distance less than current distance save in distance vector
93                if(alt < d[v]){
94                    d[v] = alt; // Update the distance in distance vector
95                }
96            }
97        }
98    }
99
100
101    for(int u=0;u<n;u++){
102        for(auto k: adj[u]){
103            int v = k.first;
104            int w = k.second;
105            int alt = d[u] + w;
106            if(alt < d[v]){
107                // If there still have edge that can be relaxed in nth times of relaxation process.
108                // Then there exists a negative cycle
109                return false;
110            }
111        }
112    }
113
114    return true;
115 }
```

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?

2.2.4 How Bellman-Ford Algorithm detect negative cycle?

2.2.5 Which is the better algorithm in which condition?