第二次

物件導向與資料結構

期末作業報告書

目錄

Part I — Implementation Details

Step 1：使用Adjacency List作為graph的紀錄方式，並從file輸入資訊建立Adjacency List

```
for (int i = 0; i < num_of_edges; i++)
{
    int from, to;
    fle >> from >> to >> weight;
    graph[from].push_back(to);
}
```

Step 2：為確保稍後的DFS尋找順序是index由小到大，使用sort將每一個list裡的index排好(這裡的sort用的是selection sort)

```
for (int i = 0; i < num_of_vertex; i++)
{
    sortIntVectorSmallToBig(graph[i]);
}
```

Part I — Implementation Details

Step 3：進行第一次DFS搜尋，取得在DFS搜尋時，graph結束的先後順序，這裡宣告一種叫FFI (Finish, Found, Index) 的structure來記錄DFS開始時間與結束時間

```cpp
struct FoundFinishIndex
{
    int found;
    int finish;
    int index;
```

```cpp
FFI[vertex_in].found = ++time;
FFI[vertex_in].index = vertex_in;
if_found[vertex_in] = true;
for (auto itr = graph[vertex_in].begin(); itr != graph[vertex_in].end(); itr++)
{
    if (if_found[*itr] == false)
    {
        DFS(*itr, time, true, 0);
    }
}
FFI[vertex_in].finish = ++time;
return;
```

部分DFS函數

Step 4：將graph所有edges倒置

```cpp
void PartI::makeGraphTransposeUsingNewVector(vector<vector<int>>& new_gT)
{
    new_gT.resize(num_of_vertex);
    for (int i = 0; i < num_of_vertex; i++)
    {
        for (auto itr = graph[i].begin(); itr != graph[i].end(); itr++)
        {
            new_gT[*itr].push_back(i);
        }
    }
    return;
}
```

製作一個倒置graph

Part I — Implementation Details

Step 5：依照剛剛蒐集來的結束時間，將所有端點的結束時間由大排到小 (selection sort)

Step 6：使用剛剛排好的頂點資訊在先前作的相反graph進行第二次DFS，由於是以結束時間為順序進行的搜尋，這次DFS將會「中斷」許多次，以每一次中斷為分隔，建立SCC queue，儲存每個SCC中開始搜尋的Index，這個Index將作為SCC裡所有端點的predecessor，存在Coarse Graph裡

搜尋的起點作為predecessor存在coarse graph

```
for (int i = 0; i < num_of_vertex; i++)
{
    if (if_found[FFI[i].index] == false)
    {
        DFS(FFI[i].index, time, false, FFI[i].index);
        scc_queue.push_back(FFI[i].index);
    }
}
```

DFS結束後在SCC queue記錄該起點

```
coarseGraph[vertex_in].first = vertex_in;
coarseGraph[vertex_in].second = predecessor;
if_found[vertex_in] = true;
for (auto itr = graph_transpose[vertex_in].begin(); itr != graph_transpose[vertex_in].end(); itr++)
{
    if (if_found[*itr] == false)
    {
        isAyclic = false;
        DFS(*itr, time, false, predecessor);
    }
}
```

Part I — Implementation Details

Step 6.5：由於這個graph跟原本的比起來是完全相反的，所以如果這是一個acyclic圖，照著結束時間搜尋是打不進DFS的recursion裡的，DFS整個depth不起來，所以在DFS裡設一個assignment，如果他進得去就知道他不是acyclic

```
coarseGraph[vertex_in].first = vertex_in;
coarseGraph[vertex_in].second = predecessor;
if_found[vertex_in] = true;
for (auto itr = graph_transpose[vertex_in].begin(); itr != graph_transpose[vertex_in].end(); itr++)
{
    if (if_found[*itr] == false)
    {
        isAyclic = false;
        DFS(*itr, time, false, predecessor);
    }
}
```

Is acyclic預設是true，但如果打得進DFS的這裡
證明他不是acyclic，所以一但有辦法進來這裡就給他false

Part I — Implementation Details

Step 7：因為題目要求要拿一個SCC裡的最小端點作為代表，將coarse graph裡的最小端點分別挑出來當該SCC的代表，記錄在SCC queue

```
for (int i = 0; i < scc_size; i++)
{
    min = scc_queue[i];
    for (int j = 0; j < CG_size; j++)
    {
        if (coarseGraph[j].second == scc_queue[i] && coarseGraph[j].first < min)
        {
            min = coarseGraph[j].first;
        }
    }
    if (min != scc_queue[i])
    {
        for (int j = 0; j < CG_size; j++)
        {
            if (coarseGraph[j].second == scc_queue[i])
            {
                coarseGraph[j].second = min;
            }
        }
    }
    scc_queue[i] = min;
}
```

Step 8：如果該graph是acyclic，結束函數，照著DFS結束的順序印出所有端點，直接就是拓樸排序

Part I — Implementation Details

Step 9：若非acyclic，依題目要求將所有SCC的代表由小排到大 (還是selection sort)

Step 10：依照題目要求，將由小到大的SCC index們更換編號為0，且由小到大的正整數這裡宣告一種structure，FromToWeight，紀錄edges的資訊，同時尋找是否有重複的edges，有則增加該edges的weight

```
struct FromToWeight
{
    int from;
    int to;
    int weight;
```

```
for (int i = 0; i < scc_size; i++)
{
    for (int j = 0; j < scc_size; j++)
    {
        if (i == j)
        {
            continue;
        }
        int weight = findEdges(scc_queue[i], scc_queue[j]);
        if (weight != 0)
        {
            q.push_back(FromToWeight(i, j, weight));
```

尋找是否有重複edges

Part I — Discussion — 時間複雜度

確保DFS順序由小到大: $O(VE^2)$ $\begin{cases} O(V) \\ O(E^2) \end{cases}$

第一次DFS: $O(V+E)$

$O(VE)$

$O(V^2)$

第二次DFS: $O(V+E)$

Acyclic: $O(VE^2)$

Cyclic: $O(V^2E + VE^2)$

$O(V^2)$

$O(V^2)$

$O(V)$

$O(V)$

確保符合題目輸出規範: $O(V^2E)$

$O(E)$

整個Solve()函數

```
for (int i = 0; i < num_of_vertex; i++)
{
    sortIntVectorSmallToBig(graph[i]);
}
searchByDFSAndUpdateFinishVector(true); // results saved to FFI
makeGraphTransposeUsingNewVector(graph_transpose); //results save
sortFFIFromBigtoSmallByFinish(); //sort FFI
searchByDFSAndUpdateFinishVector(false);
if (isAyclic)
{
    return;
}
coarseGraphUseSmallestVertexAsIndex();
scc_size = scc_queue.size();
sortIntVectorSmallToBig(scc_queue);
if (scc_size > 1)
{
    for (int i = 0; i < scc_size; i++)
    {
        for (int j = 0; j < scc_size; j++)
        {
            if (i == j)
            {
                continue;
            }
            int weight = findEdges(scc_queue[i], scc_queue[j]);
            if (weight != 0)
            {
                q.push_back(FromToWeight(i, j, weight));
```

Part I — Discussion — 我的心得

我在學最短路徑的時候原本以為DFS挺普通的，不像Dijkstra能處理有Weight的圖，也不像BFS那樣能找出最短路徑，直到寫了這份作業才知道，原來“Depth First”是這麼重要的概念，他能獲得的是「持續」的資訊。因為是深度優先，所以才能以某個點為主，持續探索完全部分支，進而獲得特殊的資訊，未來我在解Graph相關問題時，也會試著利用DFS深度優先的特性，用它來發現其他演算法無法發現的資訊。

將graph倒過來探索也是令我驚訝的想法，我這輩子沒有想過有把graph倒過來玩的這回事，我認為這是相當跳脫框架的思考，希望我未來在解題卡關時，也能保有這樣厲害的跳脫框架能力。

Part I — Discussion — 遇到的難題

我在寫這題時遇到的最難問題就是「理解」的部分，我並沒很多有向圖的經驗，所以在理解DFS的結束順序與有向圖之間的神祕特性時花了很多時間，不過一旦理解之後，很多想法都會比較能想出來，比如acyclic graph的偵測之類的，讓答案形成符合題目要求的格式也是相當令人心累，不如說我有一半的時間都花在這裡了。

希望在測測資的時候一切順利，真的不要給我出事欸。

(Dijkstra)

Part II — Implementation Details

Step 1：使用Adjacency List作為graph的紀錄方式，並從file輸入資訊，建立Adjacency List，並宣告了一個包含to和weight的structure

```
struct ToWeight
{
    int to;
    int weight;
```

```
for (int i = 0; i < num_of_edge; i++)
{
    int from, to, weight;
    fle >> from >> to >> weight;
    graph[from].push_back(ToWeight(to, weight));
}
```

Step 2：將Dijkstra所使用的distance vector初始化好，填入MAX

```
dijkstra_distance.resize(num_of_vertex);

for (int i = 0; i < num_of_vertex; i++)
{
    dijkstra_distance[i] = MAX_FOR_DIJKSTRA;
}

dijkstra_distance[0] = 0;
```

(Dijkstra)

Part II — Implementation Details

Step 3：創立一個min heap，把所有端點的index和distance都塞進去

```
MinHeap min_heap;
min_heap.initHeapWithNumOfVertex(num_of_vertex, dijkstra_distance);
```

Step 4：進入while迴圈，不斷拿出最小distance的端點出來探索，直到heap被拿光光

```
while (min_heap.isThisEmpty() == false)
{
    int vertex = min_heap.extractHeap();
    for (auto itr = graph[vertex].begin(); itr != graph[vertex].end(); itr++)
```

(Dijkstra)

Part II — Implementation Details

Step 5：如果探索到某個端點的距離比以前紀載的更近，則relax它，並decrease min heap 裡該端點的distance

```
if (dijkstra_distance[to] > (dijkstra_distance[from] + weight))
{
    dijkstra_distance[to] = dijkstra_distance[from] + weight;
    return true;
}
return false;
```

Step 6：Heap空了以後，拿出最大index端點的distance，大功告成

(Bellman Ford)

Part II — Implementation Details

Step 1：將Bellman Ford所使用的distance vector初始化好，填入MAX

```
bellmanFord_distance.resize(num_of_vertex);
for (int i = 0; i < num_of_vertex; i++)
{
    bellmanFord_distance[i] = MAX_FOR_DIJKSTRA;
}
bellmanFord_distance[0] = 0;
return;
```

(Bellman Ford)

Part II —— Implementation Details

Step 2：進入V-1次的for loop

```
for (int i = 0; i < num_of_vertex - 1; i++)
{
    for (int j = 0; j < num_of_vertex; j++)
    {
        for (auto itr = graph[j].begin(); itr != graph[j].end(); itr++)
        {
            bellmanFord_relax(j, (*itr).to, (*itr).weight);
        }
    }
}
```

Step 3：搜尋所有edges

Step 4：如果到某個端點的距離更近，則relax

(Bellman Ford)

Part II —— Implementation Details

Step 5：再搜尋所有edges一次

```cpp
for (int i = 0; i < num_of_vertex; i++)
{
    for (auto itr = graph[i].begin(); itr != graph[i].end(); itr++)
    {
        if (bellmanFord_relax(i, (*itr).to, (*itr).weight) == true)
        {
            //std::cout << "Negative";
            return -1;
        }
    }
}
```

Step 6：如果還能出現更小距離，說明這個圖有負環，回傳-1，
若否，則回傳index端點的距離

Part II — Discussion — 如何偵測負環

最後再掃一次所有edges
如果還能出現更小距離，說明這個圖有負環
我的作法是讓relax函數自己就能回傳bool
如果能鬆弛，relax回傳true
Bellman Ford收到true則終止，負環判定

```
bellmanFord_init();
for (int i = 0; i < num_of_vertex - 1; i++)
{
    for (int j = 0; j < num_of_vertex; j++)
    {
        for (auto itr = graph[j].begin(); itr != graph[j].end(); itr++)
        {
            bellmanFord_relax(j, (*itr).to, (*itr).weight);
        }
    }
}

for (int i = 0; i < num_of_vertex; i++)
{
    for (auto itr = graph[i].begin(); itr != graph[i].end(); itr++)
    {
        if (bellmanFord_relax(i, (*itr).to, (*itr).weight) == true)
        {
            return -1;
        }
    }
```

Part II — Discussion — 時間複雜度

Dijkstra

O(V)
O(V)
O(V)
O(logV)
O(V)
$O(V^2)$
O(1)

```
dijkstra_init();
MinHeap min_heap;
min_heap.initHeapWithNumOfVertex(num_of_vertex, dijkstra_distance);

while (min_heap.isThisEmpty() == false)
{
    int vertex = min_heap.extractHeap();
    for (auto itr = graph[vertex].begin(); itr != graph[vertex].end(); itr++)
    {
        if (dijkstra_relax(vertex, (*itr).to, abs((*itr).weight)) == true)
        {
            min_heap.decreaseVertexDistance((*itr).to, dijkstra_distance[(*itr).to]);
        }
    }
}
```

Dijkstra時間複雜度：$O(V^2)$

Part II — Discussion — 時間複雜度

Bellman Ford

O(VE)

O(V)

O(E)

O(1)

```
bellmanFord_init();
for (int i = 0; i < num_of_vertex - 1; i++)
{
    for (int j = 0; j < num_of_vertex; j++)
    {
        for (auto itr = graph[j].begin(); itr != graph[j].end(); itr++)
        {
            bellmanFord_relax(j, (*itr).to, (*itr).weight);
        }
    }
}

for (int i = 0; i < num_of_vertex; i++)
{
    for (auto itr = graph[i].begin(); itr != graph[i].end(); itr++)
    {
        if (bellmanFord_relax(i, (*itr).to, (*itr).weight) == true)
        {
            return -1;
        }
    }
```

O(E)

O(1)

Bellman Ford 時間複雜度：O(VE)

Part II — Discussion — 找出最短路徑

我在Part2所寫的Dijkstra和Bellman Ford為求更少的記憶體用量(題目也沒有要求)，是不會紀錄predecessor的，但如果要求最短路徑的話，可以開一個紀錄predecessor的陣列，在relax的時候順便紀錄predecessor，最後再選出要走到的目標點，用recursion的方式搜尋整個predecessor陣列的先後關係，挑出它的最短路徑會經過的端點。

Part II — Discussion — 我的心得

Weighted graph在某些狀況下比起unweighted，似乎是更能表現現實生活狀況的一種圖，因此我認為這兩種演算法都是非常重要的基本知識。兩個演算法都有著一個類似的概念，那就是不斷遍歷再鬆弛整張圖，最後慢慢獲得正確答案，這是一個扎實又穩固的想法，也許這個概念未來將可以用在其他地方。另外，這次也用到許多C++的iterator，深感iterator在std::vector和std::list等容器中的便利性，除此之外，雖然已經寫過很多次了，但這次也練習了如何寫Min heap，希望未來能有用吧。

Part II — Discussion — 遇到的難題

Dijkstra和Bellman Ford都是相當有名的演算法，老師在上課的時候也仔細的解釋過它的概念和時間複雜度了，所以不像前面有一部份要自己生出來的Part1，本次作業的Part2寫起來並不難，實現的過程也比較快一些。

Part II — Discussion — 各自使用情形

- Dijkstra
  - 應用圖: positive weighted directed graph
  - 如果搭配Min heap，在某些時候的時間複雜度可以降到 V+ElogV
  - 所以在positive weighted directed graph的時候，使用搭配Min heap的Dijkstra可能是比較好的選擇
- Bellman Ford
  - 應用圖: positive / negative weighted directed graph
  - 可以偵測負環
  - 在出現負數weight就使用Bellman Ford