Home work #4 B10815057 廖聖郝

1. Shortest Path

節點 class，每條邊使用 pair<Node<N,W>*,W>來儲存邊的終點指標與權重，父節點則是為了讓之後能夠印出路徑，所以要儲存路徑中上個節點的指標，以便之後能夠遞迴找到整條路徑。

```
template<typename N,typename W>
//N 為名稱資料型別 W為權重資料型別
class Node {
public:
	void init(N n,W w) {//初始化節點
		name = n;//初始化節點名稱
		weight_from_start = w;//初始化節點權重
	}
	bool not_init() { return !name.size(); }//如果還沒初始化，名字size == 0
	void add_link(Node<N, W>* link,W weight) { link_to.push_back(make_pair(link,weight)); }
	//加入新的link，相當於指向目標節點的指標與該條邊的權重
	N name = "";//節點名稱
	vector<pair<Node<N,W>*,W> > link_to;//所有的link，以vector儲存
	W weight_from_start;//該節點權重，Dijkstra Algorithm中紀錄從起始節點到該節點的權重(最小權重)
	bool have_been_here = false;//Dijkstra Algorithm中紀錄該節點是否已經走過了
	Node<N, W>* parent = nullptr;//父節點，Dijkstra Algorithm中紀錄從哪個節點走來該節點(最小權重)
};
```

找尋節點的函式

```
template<typename N, typename W>

Node<N, W>* find_node_ptr(Node<N, W>* all, N name,int total) {

//從所有節點中找到此名稱的節點，並回傳其指標，如果找不到就回傳null

	for (int i = 0;i < total;i++) {

		if (all[i].name == name) {

			return &all[i];

		}

	}

	return nullptr;

}
```

初始化新節點的函式

```
template<typename N, typename W>
Node<N, W>* init_new_node(Node<N, W>* all, N name, int total) {//從所有節點陣列中初始化新的節點
	for (int i = 0;i < total;i++) {
		if (all[i].not_init()) {//找到第一個還沒初始化的節點
			all[i].init(name,0);//初始化該節點
			return &all[i];//回傳該節點指標，方便後續link的處理
		}
	}
}
```

輸入最終節點，使用父節點往上遞迴直到起始節點，然後印出路徑

```
template<typename N, typename W>
void print_path(Node<N, W>* now) {//印出從起始點到終點的遞迴函式，從終點開始往父節點遞迴直到起始點，
	if (now->parent == nullptr) {//起始節點的父節點為null
		cout << now->name;//起始點前面不加空格
		return;//遞迴到起始點後結束，開始回收(輸出)之前的遞迴直到
	}
	print_path(now->parent);//遞迴呼叫
	cout << ' ' << now->name;//除了起始點外前面要加空格，這樣最後一個輸出後面就沒有空格了
}
```

處理輸入資料，建立節點與所有邊

```
	int node_num, edge_num;//暫存輸入，節點數、邊數
	string start_node, end_node;//暫存輸入，起始節點名稱、最終點名稱
	cin >> node_num >> edge_num >> start_node >> end_node;//輸入第一行
	Node<string, int>* all_node = new Node<string, int>[node_num];
	//用陣列儲存所有節點，所以先配置一塊記憶體
	for (int i = 0;i < edge_num;i++) {//輸入每個邊的資訊
		string from,to;//暫存輸入，起點、終點名稱
		int w;//暫存輸入，該條邊權重
		cin >> from >> to >> w;

		Node<string, int>* from_ptr = find_node_ptr(all_node, from, node_num);//找到起點的指標
		if (!from_ptr) from_ptr = init_new_node(all_node, from, node_num);
		//如果指標是null，代表起點還沒被初始化，所以初始化起點

		Node<string, int>* to_ptr = find_node_ptr(all_node, to, node_num);//找到終點的指標
		if (!to_ptr) to_ptr = init_new_node(all_node, to, node_num);
		//如果指標是null，代表終點還沒被初始化，所以初始化終點

		from_ptr->add_link(to_ptr, w);//在起點內加入這條邊
	}
	//所有節點初始化完畢，邊的資訊也存到每條邊的起點中了
```

Dijkstra’s Algorithm，從起始節點開始走，把往外連接的節點賦予自己的權重加上邊的權重，然後加入可能列表，如果該節點曾經被賦予權重，就要比較新的權重與舊的權重，選小的更新到此節點的權重，最後還要連接最佳路徑的父節點。

```
Node<string, int>* start_ptr = find_node_ptr(all_node, start_node, node_num);//起始節點指標
Node<string, int>* end_ptr = find_node_ptr(all_node, end_node, node_num);//最終節點指標
Node<string, int>* now_ptr = start_ptr;//當前指標，當前所在的節點指標
vector<Node<string, int>*> next_posible;//下個可能前往節點的指標陣列
while (true) {
	now_ptr->have_been_here = true;//設當前節點已走過，避免之後再走回來
	for (auto i : now_ptr->link_to) {//跑完當前節點連接出去的邊(方向向外)
		if (i.first->have_been_here) continue;//此節點已走過，跳過
		if (find(next_posible.begin(), next_posible.end(), i.first)!= next_posible.end()) {
		//如果該節點已經在可能列表當中，檢查是否需要更新
			if (i.first->weight_from_start > (now_ptr->weight_from_start + i.second)) {
			//如果新的這條路比之前走的路權重更低，代表需要更新成現在的路，以找到最短路徑
				i.first->weight_from_start = (now_ptr->weight_from_start + i.second);
				//更新成更小的權重
				i.first->parent = now_ptr;//更新成新的路線，也就是更新父節點
			}
		}
		else {//該節點已經在可能列表當中
			next_posible.push_back(i.first);//放進可能列表
			i.first->weight_from_start = now_ptr->weight_from_start + i.second;//賦予權重
			i.first->parent = now_ptr;//賦予路線(父節點)
		}
	}
	if (next_posible.empty()) break;
	//如果可能列表為空，代表沒有路可以走了，所有計算已完成，結束迴圈

	//從可能列表中尋找最小權重節點當作下一個節點
	int min_weight = std::numeric_limits<int>::max();
	//最小權重，先設為整數最大，以確保第一次迴圈能夠直接賦值
	for (auto i : next_posible) {
		if (i->weight_from_start < min_weight) {//找到更小的權重
			now_ptr = i;//更新下個迴圈的當前指標
			min_weight = i->weight_from_start;//更新最小權重
		}
	}
	next_posible.erase(find(next_posible.begin(), next_posible.end(), now_ptr));
	//下個節點找到後，就從可能列表中刪除
}
```

印出最終節點跟起始節點之間的路徑與權重，並釋放記憶體

```
print_path(end_ptr);//呼叫遞迴函式，印出路徑
cout << endl << end_ptr->weight_from_start;//印出最終節點的權重，就是該路徑的total cost
delete[] all_node;//釋放記憶體
```

2. Path with maximum Probability

邊的 class，節點並無實體的 class，而是用 string 代表，每條邊儲存 2 個端點的名稱(string)與邊的 probability。

```
template<typename N,typename P>
//N 為名稱資料型別 P為probability資料型別
class Edge {
public:
	Edge(N n1,N n2,P p):node1(n1),node2(n2),probability(p){}
	N node1, node2;//一條邊有2個端點
	P probability;//該條邊的probability
};
```

用 vector 儲存所有集合，所以該函式回傳節點位於哪個集合之中(以陣列 index 表示)

```
int which_set(vector<set<string> >& all_set,string target) {//回傳該節點在哪個set當中(vector的index)
	for (int i = 0;i < all_set.size();i++) {//跑過每個集合
		for (auto j : all_set[i]) {//跑過集合裡的每個元素
			if (target == j) {
				return i;//回傳index
			}
		}
	}
	return -1;
}
```

處理輸入，建立所有邊，存入 vector(all_edge)當中，all_node 用於儲存所有遇到的節點名稱，因為集合的特性，所以不用處理重複 insert 的狀況

```
int edge_num;//邊數
string node1, node2;//暫存輸入，2個端點的名稱
float probability,result =  1.0f;
//暫存輸入，邊的probability result儲存結果，預設為1之後才能把所有probability乘起來
vector<Edge<string, float> > all_edge;//用vector儲存所有的邊
set<string> all_node;//節點沒有實際的資料結構，用string來替代
cin >> edge_num;//輸入邊數
for (int i = 0;i < edge_num;i++) {//跑過每個邊
	cin >> node1 >> node2 >> probability;//輸入每個邊的資訊
	all_edge.push_back(Edge<string, float>(node1, node2, probability));//加到所有邊當中
	all_node.insert(node1);//節點名稱加入集合當中，因為是集合，所以沒有重複新增的問題
	all_node.insert(node2);//節點名稱加入集合當中，因為是集合，所以沒有重複新增的問題
}
```

先把所有節點建立一個集合，集合內只有該節點，每個集合都放到vector(all_set)當中

```
//互相可以連通(非相鄰)的一群節點就是集合
vector<set<string> > all_set;//用vector儲存所有的集合
for (auto i : all_node) {//一開始，設每個節點都是一個集合
    set<string> tmp;//暫存集合
    tmp.insert(i);//新增該節點
    all_set.push_back(tmp);//放入所有節點中
}
```

使用 C++內建的排序函式，因為排序的資料型別(Edge)沒有定義如何判斷大小，所以建立一個 lambda 函式來判斷 2 個 Edge 的大小，函式內部直接比較 probability 即可。

```
sort(all_edge.begin(), all_edge.end(), [](const Edge<string, float>& a, const Edge<string, float>& b)
{return a.probability > b.probability;});
```

排序後，all_edge 內的資料變為由大到小，依序由大到小的處理每個邊，若邊上的 2 個節點屬於同一個集合，加入這條邊後會形成一個環，就不符合最小生成樹的規則，所以直接忽視這條邊，反之，若處於不同集合，連接這條邊就是合法的，把結果乘上該邊的 probability，然後合併 2 個集合。

```
for (auto i : all_edge) {//從大probability的邊跑到小probability的邊
    int node1_set_index = which_set(all_set, i.node1);
    //第一個端點所在的集合，用陣列(vector)的index表示
    int node2_set_index = which_set(all_set, i.node2);
    //第二個端點所在的集合，用陣列(vector)的index表示
    if (node1_set_index != node2_set_index) {
    //如果不在同一個集合裡，代表不會形成一個環，這條邊屬於MST的一員
        //由於這2個集合間有邊連接了，所以將2個集合合併為一個集合
        all_set[node1_set_index].insert(all_set[node2_set_index].begin(),
all_set[node2_set_index].end());//把端點2所在的集合加到端點1所在的集合
        all_set.erase(all_set.begin() + node2_set_index);//刪除端點2所在的集合
        result *= i.probability;//結果乘上該條邊的probability
    }
}
```

處理輸出，根據題目要求，結果小於 0.05 視為 0，其餘的四捨五入到小數點後四位

```
if (result < 0.05) cout << 0;//依題目要求，結果小於0.05的，視為0
else cout << fixed << setprecision(4) << result;//四捨五入到小數第四位輸出
```

3. Source code

GIST　　IMGUR 圖片

4-1

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <algorithm>
#include <limits>
using namespace std;

template<typename N,typename W>  <T> 提供適用於 IntelliSense 的樣本範本引數
//N 為名稱資料型別 W為權重資料型別
class Node {
public:
    void init(N n,W w) {//初始化節點
        name = n;//初始化節點名稱
        weight_from_start = w;//初始化節點權重
    }
    bool not_init() { return !name.size(); }//如果還沒初始化，名字size == 0
    void add_link(Node<N, W>* link,W weight) { link_to.push_back(make_pair(link,weight)); }//加入新的link，相當於指向目標節點的指標與該條邊的權重
    N name = "";//節點名稱
    vector<pair<Node<N,W>*,W> > link_to;//所有的link，以vector儲存
    W weight_from_start;//該節點權重，Dijkstra Algorithm中紀錄從起始節點到該節點的權重(最小權重)
    bool have_been_here = false;//Dijkstra Algorithm中紀錄該節點是否已經走過了
    Node<N, W>* parent = nullptr;//父節點，Dijkstra Algorithm中紀錄從哪個節點走來該節點(最小權重)
};

template<typename N, typename W>
Node<N, W>* find_node_ptr(Node<N, W>* all, N name,int total) {//從所有節點中找到此名稱的節點，並回傳其指標，如果找不到就回傳null
    for (int i = 0;i < total;i++) {
        if (all[i].name == name) {
            return &all[i];
        }
    }
    return nullptr;
}

template<typename N, typename W>
Node<N, W>* init_new_node(Node<N, W>* all, N name, int total) {//從所有節點陣列中初始化新的節點
    for (int i = 0;i < total;i++) {
        if (all[i].not_init()) {//找到第一個還沒初始化的節點
            all[i].init(name,0);//初始化該節點
            return &all[i];//回傳該節點指標，方便後續link的處理
        }
    }
}
```

```cpp
template<typename N, typename W>
void print_path(Node<N, W>* now) {//印出從起始點到終點的遞迴函式，從終點開始往父節點遞迴直到起始點，
    if (now->parent == nullptr) {//起始節點的父節點為null
        cout << now->name;//起始點前面不加空格
        return;//遞迴到起始點後結束，開始回收(輸出)之前的遞迴直到
    }
    print_path(now->parent);//遞迴呼叫
    cout << ' ' << now->name;//除了起始點外前面要加空格，這樣最後一個輸出後面就沒有空格了
}

int main() {
    int node_num, edge_num;//暫存輸入，節點數、邊數
    string start_node, end_node;//暫存輸入，起始節點名稱、最終點名稱
    cin >> node_num >> edge_num >> start_node >> end_node;//輸入第一行
    Node<string, int>* all_node = new Node<string, int>[node_num];//用陣列儲存所有節點，所以先配置一塊記憶體
    for (int i = 0;i < edge_num;i++) {//輸入每個邊的資訊
        string from,to;//暫存輸入，起點、終點名稱
        int w;//暫存輸入，該條邊權重
        cin >> from >> to >> w;

        Node<string, int>* from_ptr = find_node_ptr(all_node, from, node_num);//找到起點的指標
        if (!from_ptr) from_ptr = init_new_node(all_node, from, node_num);//如果指標是null，代表起點還沒被初始化，所以初始化起點

        Node<string, int>* to_ptr = find_node_ptr(all_node, to, node_num);//找到終點的指標
        if (!to_ptr) to_ptr = init_new_node(all_node, to, node_num);//如果指標是null，代表終點還沒被初始化，所以初始化終點

        from_ptr->add_link(to_ptr, w);//在起點內加入這條邊
    }
    //所有節點初始化完畢，邊的資訊也存到每條邊的起點中了
```

```
    Node<string, int>* start_ptr = find_node_ptr(all_node, start_node, node_num);//起始節點指標
    Node<string, int>* end_ptr = find_node_ptr(all_node, end_node, node_num);//最終節點指標
    Node<string, int>* now_ptr = start_ptr;//當前指標，當前所在的節點指標
    vector<Node<string, int>*> next_posible;//下個可能前往節點的指標陣列
    while (true) {
        now_ptr->have_been_here = true;//設當前節點已走過，避免之後再走回來
        for (auto i : now_ptr->link_to) {//跑完當前節點連接出去的邊(方向向外)
            if (i.first->have_been_here) continue;//此節點已走過，跳過
            if (find(next_posible.begin(), next_posible.end(), i.first)!= next_posible.end()) {//如果該節點已經在可能列表當中，檢查是否需要更新
                if (i.first->weight_from_start > (now_ptr->weight_from_start + i.second)) {//如果新的這條路比之前走的路權重更低，代表需要更新成現在的路，以找到最短路徑
                    i.first->weight_from_start = (now_ptr->weight_from_start + i.second);//更新成更小的權重
                    i.first->parent = now_ptr;//更新成新的路線，也就是更新父節點
                }
            }
            else {//該節點已經在可能列表當中
                next_posible.push_back(i.first);//放進可能列表
                i.first->weight_from_start = now_ptr->weight_from_start + i.second;//賦予權重
                i.first->parent = now_ptr;//賦予路線(父節點)
            }
        }
        if (next_posible.empty()) break;//如果可能列表為空，代表沒有路可以走了，所有計算已完成，結束迴圈

        //從可能列表中尋找最小權重節點當作下一個節點
        int min_weight = std::numeric_limits<int>::max();//最小權重，先設為整數最大，以確保第一次迴圈能夠直接賦值
        for (auto i : next_posible) {
            if (i->weight_from_start < min_weight) {//找到更小的權重
                now_ptr = i;//更新下個迴圈的當前指標
                min_weight = i->weight_from_start;//更新最小權重
            }
        }
        next_posible.erase(find(next_posible.begin(), next_posible.end(), now_ptr));//下個節點找到後，就從可能列表中刪除
    }
    print_path(end_ptr);//呼叫遞迴函式，印出路徑
    cout << endl << end_ptr->weight_from_start;//印出最終節點的權重，就是該路徑的total cost
    delete[] all_node;//釋放記憶體
    return 0;
}
```

4-2

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <iomanip>
using namespace std;

template<typename N,typename P>
//N 為名稱資料型別 P為probability資料型別
class Edge {
public:
    Edge(N n1,N n2,P p):node1(n1),node2(n2),probability(p){}
    N node1, node2;//一條邊有2個端點
    P probability;//該條邊的probability
};

int which_set(vector<set<string> >& all_set,string target) {//回傳該節點在哪個set當中(vector的index)
    for (int i = 0;i < all_set.size();i++) {//跑過每個集合
        for (auto j : all_set[i]) {//跑過集合裡的每個元素
            if (target == j) {
                return i;//回傳index
            }
        }
    }
    return -1;
}
```

```cpp
int main() {
    int edge_num;//邊數
    string node1, node2;//暫存輸入，2個端點的名稱
    float probability,result =  1.0f;//暫存輸入，邊的probability result儲存結果，預設為1之後才能把所有probability乘起來
    vector<Edge<string, float> > all_edge;//用vector儲存所有的邊
    set<string> all_node;//節點沒有實際的資料結構，用string來替代
    cin >> edge_num;//輸入邊數
    for (int i = 0;i < edge_num;i++) {//跑過每個邊
        cin >> node1 >> node2 >> probability;//輸入每個邊的資訊
        all_edge.push_back(Edge<string, float>(node1, node2, probability));//加到所有邊當中
        all_node.insert(node1);//節點名稱加入集合當中，因為是集合，所以沒有重複新增的問題
        all_node.insert(node2);//節點名稱加入集合當中，因為是集合，所以沒有重複新增的問題
    }
    //互相可以連通(非相鄰)的一群節點就是集合
    vector<set<string> > all_set;//用vector儲存所有的集合
    for (auto i : all_node) {//一開始，設每個節點都是一個集合
        set<string> tmp;//暫存集合
        tmp.insert(i);//新增該節點
        all_set.push_back(tmp);//放入所有節點中
    }

    sort(all_edge.begin(), all_edge.end(), [](const Edge<string, float>& a, const Edge<string, float>& b) {return a.probability > b.probability;});
    //對所有邊比較probability排序，依題目要求，由大而小
    for (auto i : all_edge) {//從大probability的邊跑到小probability的邊
        int node1_set_index = which_set(all_set, i.node1);//第一個端點所在的集合，用陣列(vector)的index表示
        int node2_set_index = which_set(all_set, i.node2);//第二個端點所在的集合，用陣列(vector)的index表示
        if (node1_set_index != node2_set_index) {//如果不在同一個集合裡，代表不會形成一個環，這條邊屬於MST的一員
            //由於這2個集合間有邊連接了，所以將2個集合合併為一個集合
            all_set[node1_set_index].insert(all_set[node2_set_index].begin(), all_set[node2_set_index].end());//把端點2所在的集合加到端點1所在的集合
            all_set.erase(all_set.begin() + node2_set_index);//刪除端點2所在的集合
            result *= i.probability;//結果乘上該條邊的probability
        }
    }
    if (result < 0.05) cout << 0;//依題目要求，結果小於0.05的，視為0
    else cout << fixed << setprecision(4) << result;//四捨五入到小數第四位輸出
    return 0;
}
```