2023 OOP&DS Homework 2

111550093, 朱驛庭

Part. 1 (20%):

1. Implementation
   - 主要概念：
     把每個發電站視為一個圖中的點，兩個發電站的能源消耗即為邊的權重，因此該問題即是要找出一個最小生成樹，我選擇用 Prim's Algorithm 來解決這個問題。
   - 主函式 main()

```cpp
ll num_vertices;
cin >> num_vertices;
// input the adjacency matrix of the graph
vector<vector<ll> > graph(num_vertices, vector<ll>(num_vertices, 0));
ll u, v ,w;
// input the weight for each edge
while(cin >> u >> v >> w)
{
    graph[u][v] = w;
    graph[v][u] = w;
}
MST(num_vertices, graph);
return 0;
```

     宣告一個二維的 long long vector，接著把輸入的邊的權重存進 adjacent matrix 中，要注意的是這個圖是無向的，因此兩個方向的邊都要存。接著就開始建立一個 Minimum Spanning Tree，這邊另外寫了一個函式來讓程式變得更容易閱讀。
   - Prim's Algorithm：MST()
     - 宣告各種變數

```cpp
#define Edge pair<ll, pair<ll,ll> >
typedef long long ll;
```

```cpp
//Use Prim's Algorithm to find the MST and its cost
ll min_cost = 0;
ll in_set = 1;
priority_queue <Edge, vector<Edge>, greater<Edge> > pq;
vector<bool> sel(num_vertices, false);
```

       一開始，先宣告兩個 long long 變數：min_cost, in_set，min_cost 是用來記錄該最小生成樹的總權重，in_set 則是用來記錄目前有多少個點被尋訪過。接著宣告一個 C++ STL 中 priority queue 類型的容器，存放的類型為 Edge，即一個存放邊資訊的 3-tuple：(w,u,v)，分別是邊的權重以及起點終點，另外因為 priority queue 預設是由大到小排列，因此要在其第三個參數的地方改寫成

greater<Edge>。另外我們也會需要一個 bool 類型的 vector 來記錄每一個點是否已被加入當前的樹中。

■ 初始化 priority queue

```
sel[0] = true;
// add all edges from vertex 0 to the priority queue
for(ll i = 0; i < num_vertices; i++) if(graph[0][i] != 0)
{
    pq.push(make_pair(graph[0][i], make_pair(0, i)));
}
```

以第一個點（index=0）當作起點，並將第一個點連接的所有邊放到 priority queue 裡面。

■ 主要迴圈

```
while( in_set < num_vertices)
{
    Edge e = pq.top();
    pq.pop();
    // if both vertices are already in the set, ignore this edge
    if(sel[e.second.second] == true) continue;
    // else add the edge to the MST
    min_cost += e.first;
    ll x = e.second.second;
    // add all edges from vertex x to the priority queue
    for(ll i = 0; i < num_vertices; i++) if(graph[x][i] != 0 && sel[i] == false)
    {
        pq.push(make_pair(graph[x][i], make_pair(x, i)));
    }
    sel[x] = true;
    in_set ++;
}
cout << min_cost << endl;
```

停止條件：每個點都被加到樹裡面。

每次將 priority queue 中最小的邊拿出來，因為放在裡面的邊的起點都被選過了，所以只需檢查終點有沒有都被加到樹裡面。如果有的話直接跳過這輪，沒有的花就把該點加到樹裡面，並同時把該邊的權重加到 min_cost，最後該點連到的所有沒標記的邊也丟進 priority queue 中。

2. Time complexity

結論：時間複雜度為$O\big((V+E)\log E\big)$

```
void MST(int num_vertices, vector<vector<ll> > graph)
{
    //Use Prim's Algorithm to find the MST and its cost
    ll min_cost = 0;  O(1)
    ll in_set = 1;  O(1)
    priority_queue <Edge, vector<Edge>, greater<Edge> > pq;    O(1)
    vector<bool> sel(num_vertices, false);  O(1)
    sel[0] = true;
    // add all edges from vertex 0 to the priority queue
    for(ll i = 0; i < num_vertices; i++) if(graph[0][i] != 0)  V Times
    {                                                                  O(V log V)
        pq.push(make_pair(graph[0][i], make_pair(0, i)));  O(log V)
    }
    while( in_set < num_vertices)
    {
        Edge e = pq.top();  O(1)
        pq.pop();  O(1)
        // if both vertices are already in the set, ignore this edge
        if(sel[e.second.second] == true) continue;   O(1)
        // else add the edge to the MST
        min_cost += e.first;  O(1)
        ll x = e.second.second;  O(1)                              E Times
        // add all edges from vertex x to the priority queue
        for(ll i = 0; i < num_vertices; i++) if(graph[x][i] != 0 && sel[i] == false)  O(E log V)
        {
            pq.push(make_pair(graph[x][i], make_pair(x, i)));  O(log V)
        }
        sel[x] = true;  O(1)
        in_set ++;  O(1)
    }
    cout << min_cost << endl;  O(1)
}
```

- 初始化 priority queue： $O(V\log V)$

  總共要尋訪過所有連接起點的點，最多可能有$V-1$個，而將一個邊加入 priority queue 的複雜度是$O(\log V)$，因此該步驟的總複雜度是$O(V\log V)$。

- 迴圈找最小的邊： $O(E\log V)$

  總共要尋訪過所有目前點集合外且有邊連接的點，因此一共需要E次，將一個邊加入 priority queue 的複雜度是$O(\log V)$，因此該步驟的總複雜度是$O(E\log V)$。

兩個步驟是線性的，因此總複雜度是兩者相加為：$O\big((V+E)\log E\big)$

3. Challenges/ discussion

- Challenges：

  原本沒有想到要用 priority queue 來找權重最小的邊，因此在 while 裡面每次都還要做兩層 for 迴圈，需要複雜度$O(V^2)$才能獲取最小的邊，因此後來改使用 priority queue，寫起來也比較簡潔。

- Which is the better algorithm in which condition：

Kruskal 演算法的複雜度是$O(E\log E)$，因此在邊的數量較少時效率較高，在這個問題中邊的數量比較多，因此選擇 Prim 演算法會比較好。

Part. 2 (20%):

1. Implementation
   - 核心概念：
     這是一個最短路徑問題，但麻煩的是每邊的權重不固定，要根據輸入的交通工具的 size 判斷，因此在建立圖之前會需要依照條件選擇三者之中時間較少者，作為該邊的權重，之後就可以用 Bellman-Ford 演算法來計算每一筆測資的最短路徑。
   - 自定義變數：Edge, Graph
     - Edge：u, v 為起終點，w 為權重
     - Graph：V, E 為點集合和邊集合的元素個數，edge 為 Edge 類型的陣列
     - createGraph()：回傳一個 Graph 類型的指標

```
// Struct for the edges of the graph
struct Edge {
  ll u;  //start vertex of the edge
  ll v;  //end vertex of the edge
  ll w;  //w of the edge (u,v)
};

// Graph - it consists of edges
struct Graph {
  ll V;        // Total number of vertices in the graph
  ll E;        // Total number of edges in the graph
  struct Edge* edge;  // Array of edges
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(ll V, ll E) {
  struct Graph* graph = new Graph;
  graph->V = V;  // Total Vertices
  graph->E = E;  // Total edges

  // Array of edges for graph
  graph->edge = new Edge[E];
  return graph;
}
```

- 輸入資料處理

```
ll num_vertices, num_edges;
cin >> num_vertices >> num_edges;
Graph* graph = createGraph(num_vertices, num_edges);
vector<vector<ll> > edges(num_edges, vector<ll>(4,0));
// input the start vertex, end vertex, and the time it takes to travel from start to e
for(ll i = 0; i < num_edges; i++)
{
    //u, v, w, t1, t2, t3
    ll u, v;
    cin >> u >> v;
    cin >> edges[i][0] >> edges[i][1] >> edges[i][2] >> edges[i][3];
    graph->edge[i].u = u-1;
    graph->edge[i].v = v-1;
}
ll s[3];
cin >> s[0] >> s[1] >> s[2];
// define each edge as the minimum time it takes to travel from one vertex to another
for(ll i = 0; i < num_edges; i++)
{
    ll min_time = INF;
    for(ll j = 0; j < 3; j++)
    {
        if(s[j] <= edges[i][0] && min_time > edges[i][j+1]) min_time = edges[i][j+1];
    }
    graph->edge[i].w = min_time;
}
```

建立一個 Graph，並使用一個二維的 vector 來儲存每個邊的資料。接著輸入三種交通工具的寬度，判斷每一邊的寬度是否大於這些寬度，並把其中最小的存成圖的權重。

```
ll num_test_cases;
cin >> num_test_cases;
for(ll i = 0; i < num_test_cases; i++)
{
    ll start, end;
    cin >> start >> end;
    SPT(start-1, end-1, graph);
}
```

接著依據每次的數據去生成一顆最短路徑樹，這邊使用自定義的函式 SPT( )來處理。

- Bellman-Ford 演算法：SPT( )

首先，宣告一個 vector 來記錄每個點到起始點的距離，把起點的距離紀錄為一。接著每次更新每一條邊，如果該點的距離大於邊的權重＋另一點的距離，則更新該點，總共更新 V-1 個回合。

```
void SPT(ll start,ll end, Graph* graph)
{
    //Construct Shortest Path Tree using Bellman-Ford Algorithm
    ll V = graph->V;
    ll E = graph->E;

    vector<ll> dist(V, INF);
    dist[start] = 0;
    for(ll i = 0; i < V - 1; i++)
    {
        for(ll j = 0; j < E; j++)
        {
            int u = graph->edge[j].u;
            int v = graph->edge[j].v;
            int w = graph->edge[j].w;
            if (dist[u] != INF && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
            if (dist[v] != INF && dist[v] + w < dist[u])
                dist[u] = dist[v] + w;
        }
    }
    cout << dist[end] << endl;
}
```

2. Time complexity

結論：時間複雜度為$O(VE)$

```
void SPT(ll start,ll end, Graph* graph)
{
    //Construct Shortest Path Tree using Bellman-Ford Algorithm
    ll V = graph->V;
    ll E = graph->E;

    vector<ll> dist(V, INF);
    dist[start] = 0;
    for(ll i = 0; i < V - 1; i++)
    {
        for(ll j = 0; j < E; j++)
        {
            int u = graph->edge[j].u;
            int v = graph->edge[j].v;
            int w = graph->edge[j].w;
            if (dist[u] != INF && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
            if (dist[v] != INF && dist[v] + w < dist[u])
                dist[u] = dist[v] + w;
        }
    }
    cout << dist[end] << endl;
}
```

V Times

E Times

$O(1)$

$O(VE)$

3. Challenges/ discussion

- Which is the better algorithm in which condition：
  另外一個最短路徑問題的演算法是 Dijkstra 演算法，其時間複雜度為$O(V^2)$，但在這個問題中邊的數量和點的數量差不多，因此和 Bellman-Ford 演算法的效率差不多。
- Challenges：
  一開始看到題目會不太知道要怎麼記錄整張圖，因為每個邊的權重不是固定的，後來想到可以先把所有邊的資訊存起來，等到輸入各個交通工具的寬度後再去決定邊的權重，這樣比起直接全部存起來簡潔很多。