2023 OOP&DS Homework 2

110705013 (Student-id), 沈昱宏 (Name)

Part. 1 (20%):

1. Implementation

   Code with explanation:

```
#include <iostream>
using namespace std;

/*

I created two struct - edge & vertice

find() : find the ancestor of a vertice, the vertices chained
together have the same ancestor.

merge() & sort() : sort the edges (from min cost to max cost)

kruskal() : find the minimum spanning using kruskal's algorithm

*/
struct edge{
    int x;          // one vertice
    int y;          // the other vertice
    int cost;       // the cost(supply power) between vertices
};

edge edges[1002*1002];
edge tmp[1002*1002];

struct vertice{
    int index;      // index of this vertice
    int ancestor;   // ancestor of this vertice
};

vertice vertices[1002];

int find(int a){
    while(vertices[a].ancestor!=a){
        a=vertices[a].ancestor;
    }
    return a;
}

void merge(int front,int mid,int end){
    int index=front,count=front;
    int rs=mid+1;
    while(count<=mid && rs<=end){
        if(edges[count].cost>edges[rs].cost){
            tmp[index++]=edges[rs++];

        }
        else{
            tmp[index++] = edges[count++];

        }
    }
    while(count<=mid){
        tmp[index++] = edges[count++];
    }
    while(rs<=end){
        tmp[index++] = edges[rs++];
    }
    for(int i=front;i<=end;i++){
        edges[i]=tmp[i];
    }
}
```

```
void sort(int front,int end){
    if(front < end){
        int mid = (front+end)/2;
        sort(front,mid);
        sort(mid+1,end);
        merge(front,mid,end);
    }
}

int kruskal(int Num){

    for(int i=0;i<Num;i++){              // reset the vertices
        vertices[i].index = vertices[i].ancestor = i;
    }
    int ct = 0;
    int ans = 0;
    int edgefound = 0;

    while(edgefound<Num-1){              // while not finish finding
        edge newedge = edges[ct++];      // use unused edge with minimum cost
        if(find(newedge.x)!=find(newedge.y)){   // if ancestor is different
            ans+=newedge.cost;                  // add the cost of the edge (needed)

            if(find(newedge.x)<find(newedge.y)){    // update the ancestors
                vertices[find(newedge.y)].ancestor = find(newedge.x);
            }
            else{
                vertices[find(newedge.x)].ancestor = find(newedge.y);
            }
            edgefound++;
        }
    }
    return ans;
}
int main()
{
    int Num;
    cin>>Num;
    int firstnode,secondnode,distance;
    int count = 0;
    while((cin >> firstnode) && !cin.eof()){
        cin>>secondnode>>distance;
        edges[count].x = firstnode;
        edges[count].y = secondnode;
        edges[count++].cost = distance;
    }
    sort(0,count-1);

    int ans;
    ans = kruskal( Num);
    cout<<ans<<endl;
    return 0;
}
```

Sort the edges → kruskal algorithm

2. Time complexity

Sorting: O(E*log(E)). Choosing tree: O(E)*O(V), since $V^2/2 < E$, V = O(log(V)), choosing tree is also O(E*log(E)).

Total time complexity = O(E*log(E))

3. Challenges/ discussion

The implementation of how to find the ancestors in Kruskal's algorithm took me some time. Before I started to implement the algorithm, my original thought was to use pointers to link the vertices, but when I started to write the code, I realized that I can do it with a table indicating the relationship between vertices and the corresponding ancestors. The change made the implementation a lot easier.

Part. 2 (20%):

1. Implementation
   Explanation in code

```cpp
#include <iostream>
using namespace std;

// define a structure for storing
struct Path{
    int roadWidth;
    int Ts[3];
};

int main()
{
    int N, M;
    cin >> N >> M;

    // create a graph
    Path ** graph;
    graph = new Path *[N + 1];
    for(int i = 0; i < N+1; i++){
        graph[i] = new Path [N + 1];
    }
    // reset the graph
    for(int i = 0; i < N+1; i++){
        for(int j = 0; j < N+1; j++){
            graph[i][j].roadWidth = 0;
            for(int k = 0; k < 3; k++){
                graph[i][j].Ts[k] = 9999999;
            }
        }
    }
    // input into the graph
    int from, to, width, truckTime, bikeTime, carTime;
    for(int i = 0;i<M;i++){
        cin >> from >> to >> width >> truckTime >> bikeTime >> carTime;
        graph[from][to].roadWidth = graph[to][from].roadWidth = width;
        graph[from][to].Ts[0] = graph[to][from].Ts[0] = truckTime;
        graph[from][to].Ts[1] = graph[to][from].Ts[1] = bikeTime;
        graph[from][to].Ts[2] = graph[to][from].Ts[2] = carTime;
    }

    // For the three kind of vehicle, if the vehicleW is bigger than road width,
    // set the time for the road using the vehicle to 9999999, which is the same
    // as their is no path.
    int vehicleW;
    for(int i = 0; i < 3; i++){
        cin >> vehicleW;
        for(int j = 0; j < N + 1; j++){
            for(int k = 0; k < N + 1; k++){
                if(graph[j][k].roadWidth < vehicleW){   // width for the road is not wide enough
                    graph[j][k].Ts[i] = 9999999;
                }
            }
        }
    }
    int p;

    // the minimum possible path for each pair of vertex
    int ** mingraph;
    mingraph = new int *[N + 1];
    for(int i = 0; i < N+1; i++){
        mingraph[i] = new int [N + 1];
    }
    int smallest;
    for(int i=0;i<N+1;i++){
        for(int j=0;j<N+1;j++){
            smallest = graph[i][j].Ts[0];                   // here is to find the minimum
            if(smallest > graph[i][j].Ts[1]) smallest = graph[i][j].Ts[1];
            if(smallest > graph[i][j].Ts[2]) smallest = graph[i][j].Ts[2];
            mingraph[i][j] = smallest;
        }
    }

    // since we may need to ask for the shortest path for more than source,
    // so I implement the Floyd_Warshall algorithm to find the shortest path from each node.
    for(int i = 0; i < N + 1; i++){
        for(int j = 0; j < N + 1; j++){
            for(int k = 0; k < N + 1; k++){
                if(mingraph[j][k] >= mingraph[j][i] + mingraph[i][k]){
                    mingraph[j][k] = mingraph[j][i] + mingraph[i][k];
                }
            }
        }
    }
    // input query and output the minimum path of three vehicle
    cin >> p;
    while(p--){
        cin >> from >> to;
        cout << mingraph[from][to] << endl;
    }

    return 0;
}
```

2. Time complexity
   N: number of node; p: number of testcase.
   Input: O(N); Check width of road: O(N^2); Floyd-Marshell: O(N^3); Output: O(p)
   Total complexity: O(N^3 + p)

3. Challenges/ discussion
   Choice of shortest path algorithm: At first, I was thinking to implement Dikjastra algorithm to solve for each p. Using Dikjastra will be faster if p if relatively small. According to the test case limit given by TAs, using Dikjastra's algorithm O(p*N2) will be faster than Floyd-Marshell algorithm.