Student: 陳玉山
ID：0710185

**REPORT HOMEWORK 8**

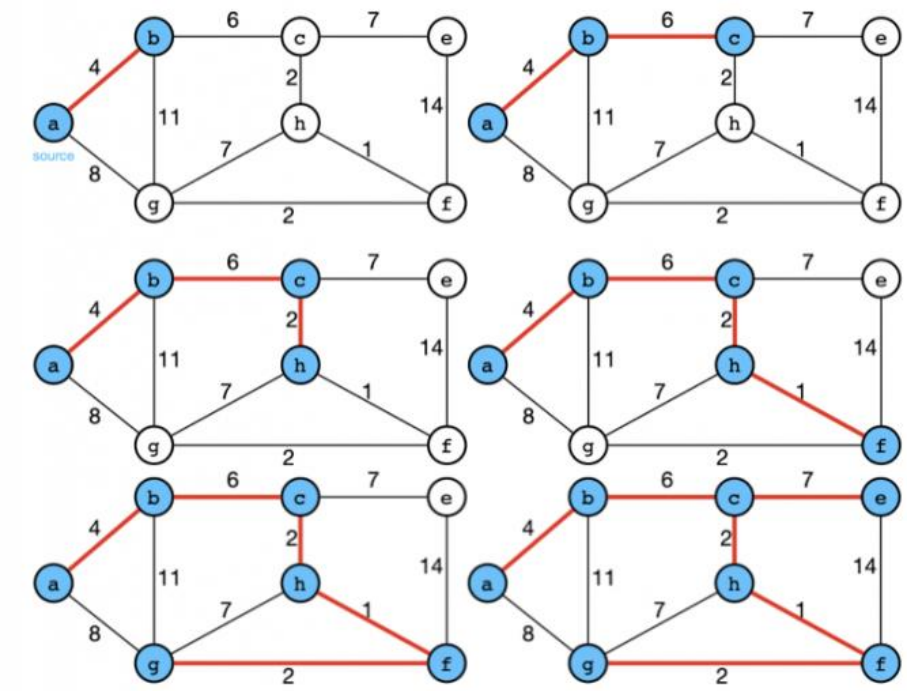
## I. Introduction

- In this homework, we try to solve the Minimum spanning tree problem by using two different methods, Prims and Kruskal. Each method includes different approaches, we may discuss them in the methodology section.

## II. Methodology

### 1) Prims method

For this approach, we will start at a certain node, then add all the connected edges of that current node to the priority queue. Taking advantage of this data structure, we can achieve the smallest edges and add them to the list of the minimum set of edges. After discovering a node, we move to the next node that is popping out recently and repeat the process of finding the smallest edges. If the smallest edge is connected with the visited node, we then take out a node from the priority-queue to visit. In this way, the smaller nodes will be visited first and then moved to the larger nodes. The process stops when all the nodes are visited with the requirement that there are no cycles. For my homework, I choose node 6 to start with.




*1: The process of Prim's algorithm (Medium.com)*

Student: 陳玉山
ID：0710185

```
u = 6 v = 7 weight = 1
u = 6 v = 5 weight = 2
u = 5 v = 2 weight = 4
u = 2 v = 8 weight = 2
u = 2 v = 3 weight = 7
u = 2 v = 1 weight = 8
u = 1 v = 0 weight = 4
u = 3 v = 4 weight = 9
1 2 4 2 7 8 4 9
total = 37
Visited = 1 1 1 1 1 1 1 1 1
```

*Figure 2: The result from running Prim's algorithm.*

```
priority_queue<link *, vector<link *>, compare> pq;
vector<link *> stack;
void prim(int start)
{ // start = 6
    link *min_path;
    edge_visited[start] = true;
    while (1)
    {
        if (visit_count == V)
            break;
        // cout << "something!";
        for (int j = 0; j < V; j++)
        {
            if (graph[start][j] != 0)
            {
                link *temp = create_link(start, j, graph[start][j]);
                pq.push(temp);
            }
        }
        min_path = pq.top();
        pq.pop();

        while (edge_visited[min_path->v] == true && pq.empty() == false)
        {
            // cout << "pq.size()= " <<pq.size() << endl;
```

Student: 陳玉山
ID：0710185

```
    while (edge_visited[min_path->v] == true && pq.empty() == false)
    {
        // cout << "pq.size()= " <<pq.size() << endl;

        min_path = pq.top();
        pq.pop();
    }

    cout << "u = " << min_path->u << " v = " << min_path->v << " weight = " << min_path->value << endl
    // cout << "count = " << visit_count << endl;

    start = min_path->v;
    stack.push_back(min_path);
    edge_visited[min_path->v] = true;
    visit_count++;
}
}
```

**2) Kruskal's Algorithm**

- For this algorithm, we will add all the edges into a list, then sort all of them by the value of the edges. For this approach, we can quickly take out the smallest edges and then add them to the list of minimum spanning trees. In the process, we need to make sure that the edges won't form any cycles. In my opinion, the process of checking for forming cycle is the most complicated one in this algorithm.

```
bool isCycle()
{


    for (int i =0; i<set.size();i++){
        cout << "set[i] = " << set[i];
    }


cout<<endl << "set.size() = " << set.size() <<endl;


int *visited = new int[V];
const int temp_size = V;
int **graph_temp = new int *[temp_size];

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

    graph_temp[i] = new int(temp_size);
    visited[i] = 0;
    pop[i] = false;
}
```

Student: 陳玉山
ID：0710185

```
for (int i = 0; i < V; i++)
{
    // edge_visited[i] = false;
    for (int j = 0; j < V; j++)
    {

        graph_temp[i][j] = 0;
        // path_visited[i][j] = false;
    }
}
// cout << "u v " << mst[2]->u << " " << mst[2]->v ;
cout << endl<<endl;

cout << "mst[0] -> u = "<<mst[mst.size()-1] -> u << endl;
start1 = mst[mst.size()-1] -> u;
visited[mst[mst.size()-1] -> u] = 1;
for (int i = 0; i < mst.size(); i++)
{

    graph_temp[mst[i]->u][mst[i]->v] = mst[i]->value;
    graph_temp[mst[i]->v][mst[i]->u] = mst[i]->value;
}
```

```
else if (j == V - 1 )
{

    cout << "stack size " << new_stack.size() << endl;



    start1 = new_stack[new_stack.size() - 1];
    cout << "pop = " << start1 << endl;
    visited[start1]--;
    new_stack.pop_back();
    pop[start1] = true;
}
else if (graph_temp[start1][j] != 0 && visited[j] == 1 && pop[start1] == false)
{
    visited[j] = 2;
}
else if (graph_temp[start1][j] != 0 && visited[j] == 2 && visited_count >= 2 && pop[start1] ==
{

    cout << "start = , j = " << start1 << " " << j << endl;
    return true;
    // for (int i = 0; i < temp_size; i++)
```

Student: 陳玉山
ID：0710185

```
while (1)
{
    flag = true;
    cout << " visit_count "<<visit_count << " "<<endl;
    if (isAllVisited(visited) && new_stack.size() == 0)
        break;
    print_visited(visited);

    for (int j = 0; j < V; j++)
    {
        //  cout << "somthing" << endl;
        // cout << "start = " << start1 << " " << "j = " << j << endl;
        cout << "stack size = " << new_stack.size();
        if (graph_temp[start1][j] != 0 && visited[j] == 0)
        {
            cout << "True";
            visited[j] = 1;
            cout << "visited[j] = " <<visited[j];
            cout << "start " <<start1 << endl;
            new_stack.push_back(start1);
            start1 = j;
            visited_count++;
            break;
        }
```

```
else if (j == V - 1 )
{

    cout << "stack size " << new_stack.size() << endl;



    start1 = new_stack[new_stack.size() - 1];
    cout << "pop = " << start1 << endl;
    visited[start1]--;
    new_stack.pop_back();
    pop[start1] = true;
}
else if (graph_temp[start1][j] != 0 && visited[j] == 1 && pop[start1] == false)
{
    visited[j] = 2;
}
else if (graph_temp[start1][j] != 0 && visited[j] == 2 && visited_count >= 2 && pop[start1] ==
{


    cout << "start = , j = " << start1 << " " << j << endl;
    return true;
    // for (int i = 0; i < temp_size; i++)
```

*Figure 3: isCycle() function.*

- The algorithm will check whether the added edges form the cycle every time the new edge is added.

III. Discussion

1) When dealing with a sparse graph we can use Kruskal's algorithm, and Prim's algorithm will run faster when we apply a dense graph. Because Kruskal's algorithm first has to sort all the edges, adding the minimum edge to the minimum spanning tree, then it has to make sure that these edges won't form a cycle. For this reason, it

would spend much time when the graph goes large. Meanwhile, Prim's algorithm just needs to store all the edges in the priority queue, and pop out whenever needed, it doesn't have to visit all the newly formed edges to check for some special requirements.

2) In that case, Prim's algorithm won't work because it always finds the connected edges to do the comparison, when there is a gap between the two graphs, it can't find the connection and compare. Therefore, the calculation may be not correct.

- Although the process of checking for the cycle is running well when I test it independently on different cpp file, I have a problem when trying to move that function to the homework. I think that because memory is overloaded when trying to run that function inside some while loops.

- By the way, I want to say thank you so much to the teacher and TAs for your hard work and help us to have a better fundamental knowledge of Algorithms. Wish you have a nice summer holiday!