## 图结构和算法

### 1、图的深度优先搜索

将图中每个顶点的访问标志设为 FALSE, 之后搜索图中每个顶点，如果未被访问，则以该顶点V0为起始点出发，访 问此顶点，然后依次从V0的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和V0有路径相通的顶 点都被访问到。

### 2、图的广度优先搜索

从图中的某个顶点V0出发，并在访问此顶点之后依次访问V0的所有未被访问过的邻接点，之后按这些顶点被访问 的先后次序依次访问它们的邻接点，直至图中所有和V0有路径相通的顶点都被访问到。

### 3、最小生成树算法

#### (1)Prim算法

普里姆算法：取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w。在添加的顶点 w 和已经 在生成树上的顶点v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。之后继续 往生成树上添加顶点，直至生成树上含有 n-1 个顶点为止。

leetcode例题：

leetcode1135

想象一下你是个城市基建规划者，地图上有 N 座城市，它们按以 1 到 N 的次序编号。

给你一些可连接的选项 conections，其中每个选项 conections[i] = [city1, city2, cost] 表示将城市 city1 和城市 city2 连接所要的成本。（连接是双向的，也就是说城市 city1 和城市 city2 相连也同样意味着城市 city2 和城市 city1 相连）。

返回使得每对城市间都存在将它们连接在一起的连通路径（可能长度为 1 的）最小成本。

该最小成本应该是所用全部连接代价的综合。如果根据已知条件无法完成该项任务，则请你返回 -1。

————————————————

原文链接：https://blog.csdn.net/qq\_21201267/article/details/107796632

```

struct cmp

{

bool operator()(const pair<int,int>& a, const pair<int,int>& b) const

{

return a.second > b.second;//小顶堆, 距离小的优先

}

};

class Solution {

public:

int minimumCost(int N, vector<vector<int>>& connections) {

vector<bool> vis(N+1, false);

vector<vector<pair<int,int>>> edges(N+1,vector<pair<int,int>>());

for(auto& c : connections)

{

edges[c[0]].push\_back({c[1],c[2]});

edges[c[1]].push\_back({c[0],c[2]});

}

priority\_queue<pair<int,int>, vector<pair<int,int>>, cmp> q;

int to, distance, total = 0, edge = 0;

vis[1] = true;

for(auto& e : edges[1])

q.push(e);

while(!q.empty())

{

to = q.top().first;

distance = q.top().second;

q.pop();

if(!vis[to])

{

vis[to] = true;

total += distance;

edge++;

if(edge == N-1)

return total;

for(auto& e : edges[to])

q.push(e);

}

}

return -1;

}

};

```

#### (2)Kruskal算法

克鲁斯卡尔算法：先构造一个只含 n 个顶点的子图 SG，然后从权值最小的边开始，若它的添加不使 SG 中产生回 路，则在 SG 上加上这条边，如此重复，直至加上 n-1 条边为止。

```

class dsu

{

vector<int> f;

public:

dsu(int n)

{

f.resize(n);

for(int i = 0; i < n; ++i)

f[i] = i;

}

void merge(int a, int b)

{

int fa = find(a);

int fb = find(b);

f[fa] = fb;

}

int find(int a)

{

int origin = a;

while(a != f[a])

{

a = f[a];

}

return f[origin] = f[a];

}

};

class Solution {

public:

int minimumCost(int N, vector<vector<int>>& connections) {

dsu u(N+1);

sort(connections.begin(), connections.end(),[&](auto a, auto b){

return a[2] < b[2];//距离短的边优先

});

int edge = 0, p1, p2, dis, total = 0;

for(int i = 0; i < connections.size(); ++i)

{

p1 = connections[i][0];

p2 = connections[i][1];

dis = connections[i][2];

if(u.find(p1) != u.find(p2))//两个还未链接

{

u.merge(p1,p2);

edge++;

total += dis;

}

if(edge == N-1)

break;

}

return edge==N-1 ? total : -1;

}

};

```

针对上述两个方法球最小路径即最小生成树，由于自己用unordered\_map<int,unordered\_set<pair<int,int>>>存储边出现错误，但是应该是可以使用的，需要尝试。自己写得代码如下：

```

#include <iostream>

#include <vector>

#include <utility>

#include <queue>

#include <unordered\_map>

#include <unordered\_set>

#include <algorithm>

using namespace std;

class PrimSolution {

public:

    struct cmp {

        // pair 类型数据之间可以使用 <, >做比较，但优先比较 first, 如果 first 相同才会比较 second.

        // 以下是对括号 () 的操作符重载，用结构体模仿函数的行为，这是仿函数的标准做法。

        bool operator()(const pair<int, int>& a, const pair<int, int>& b) const {

            return a.second > b.second;  //小顶堆, 距离小的优先

        }

    };

    int minimumCost(int N, vector<vector<int>>& connections) {

        vector<bool> vis(N + 1, false);

        vector<vector<pair<int, int>>> edges(N + 1, vector<pair<int, int>>());

        for (auto& c : connections) {

            // edges[i] 是个vector,里面存了i点直接相连的点及其对应的边的权值。

            edges[c[0]].push\_back({ c[1],c[2] });

            edges[c[1]].push\_back({ c[0],c[2] });

        }

        priority\_queue<pair<int, int>, vector<pair<int, int>>, cmp> q;

        int to, distance, total = 0, edge = 0;

        vis[1] = true;  // 第1点是起始点。

        for (auto& e : edges[1])

            q.push(e); // 把第 1 点的所有边加入优先队列；

        while (!q.empty()) {

            to = q.top().first; // to 是 连通分量a中的所有点的相关边里，权值最小的边的另一个顶点

            distance = q.top().second; // 权值最小的边的权值

            q.pop();  // 去掉刚刚处理的边

            if (!vis[to]) { // 如果刚刚处理的点（边）以前访问过，那么说明已经在这个连通分量 a 里面了，不需要再考虑

                vis[to] = true;

                total += distance;

                edge++;

                if (edge == N - 1)  // 放入连通分量 a 的 边的数量等于N-1，最小生成树就形成了。

                    return total;

                // 把刚刚处理的边的另一顶点（to）也加入了 连通分量a， 那么也要把 to  的相关 边 也加入连通分量的相连边这个队列中。

                for (auto& e : edges[to])

                    q.push(e);

            }

        }

        return -1;  // 如果连通分量 a 的 边的数量无法等于N-1，说明是非连通图，没有生成树。

    }

};

class KruskalSolution

{

public:

    vector<int> parent;

    int find(int x)

    {

        if (x != parent[x]) parent[x] = find(parent[x]);

        return parent[x];

    }

    void Union(int x,int y)

    {

        int fx = find(x);

        int fy = find(y);

        if (fx != fy)

        {

            parent[fy] = fx;

        }

    }

    bool isconnected(int x,int y)

    {

        if (find(x) != find(y)) return false;

        else return true;

    }

    static bool cmp(vector<int>& a, vector<int>& b)

    {

        return a[2] < b[2];

    }

    int minimumCost(int N, vector<vector<int>>& connections) {

        for (int i = 0; i <= N; i++)

        {

            parent.push\_back(i);

        }

        sort(connections.begin(), connections.end(), cmp);

        int nums = 0,total=0;

        for (int i = 0; i < connections.size(); i++)

        {

            int x = connections[i][0], y = connections[i][1],dis=connections[i][2];

            if (!isconnected(x, y))

            {

                nums++;

                total += dis;

                if (nums == N - 1)

                {

                    return total;

                }

                Union(x, y);

            }

        }

        return -1;

    }

};

int main()

{

    int N = 3;

    vector<vector<int>> connections = { {1, 2, 5},{1, 3, 6},{2, 3, 1 }};

    KruskalSolution test;//PrimSolution

    int ans=test.minimumCost(N, connections);

    cout << ans;

    return 0;

}

```

### 4、最短路径算法

#### (1)单源最短路径算法：Dijkstra

思路：

Dijkstral算法为求解一个点到其余各点最小路径的方法，其算法为： 假设我们求解的是顶点v到其余各个点的最短距离。n次循环至n个顶点全部遍历： 1. 从权值数组中找到权值最小的，标记该边端点k 2. 打印该路径及权值 3. 如果存在经过顶点k到顶点i的边比v->i的权值小 4. 更新权值数组及对应路径

leetcode743网络延迟问题:

![1bbe9c9458e8afd715bdacfb239c22ae.png](en-resource://database/577:1)

```

class Solution {

public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {

        int inf=INT\_MAX/2;

        vector<vector<int>> g(n,vector<int>(n,inf));

        for(vector<int>& t:times)

        {

            int x=t[0]-1,y=t[1]-1,time=t[2];

            g[x][y]=time;

        }

        vector<int> dis(n,inf);

        vector<int> used(n,0);

        dis[k-1]=0;

        for(int i=0;i<n;i++)

        {

            int x=-1;

            for(int y=0;y<n;y++)

            {

                if(!used[y]&&(x==-1||dis[y]<dis[x]))

                {

                    x=y;

                }

            }

            used[x]=true;

            for(int y=0;y<n;y++)

            {

                dis[y]=min(dis[y],dis[x]+g[x][y]);

            }

        }

        int ans=\*max\_element(dis.begin(), dis.end());

        if(ans==inf) return -1;

        else return ans;

    }

};

```

#### (2)Bellman-ford

#### (3)基于队列优化的Bellman-ford

### 5、并查集

#### (1)Quick Find

```

public class UnionFind {

int root[];

public UnionFind(int size) {

root = new int[size];

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

root[i] = i;

}

}

public int find(int x) {

return root[x];

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

for (int i = 0; i < root.length; i++) {

if (root[i] == rootY) {

root[i] = rootX;

}

}

}

};

public boolean connected(int x, int y) {

return find(x) == find(y);

}

}

```

#### (2)Quick Union

相比Quick Find效率更高

```

public class UnionFind {

int root[];

public UnionFind(int size) {

root = new int[size];

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

root[i] = i;

}

}

public int find(int x) {

while(x!=root[x])

{

x=root[x];

}

return x;

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

root[rootY]=rootX;

}

};

public boolean connected(int x, int y) {

return find(x) == find(y);

}

}

```

#### (3)按秩合并的并查集

```

public class UnionFind {

int root[];

public UnionFind(int size) {

root = new int[size];

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

root[i] = i;

rank[i]=1;

}

}

public int find(int x) {

while(x!=root[x])

{

x=root[x];

}

return x;

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if(rank[rootX]>rank[rootY])

{

root[rootY]=rootX;

}

else if(rank[rootX]<rank[rootY])

{

root[rootX]=rootY;

}

else

{

root[rootY]=rootX;

rank[rootX]++;

}

}

};

public boolean connected(int x, int y) {

return find(x) == find(y);

}

}

```

#### (4)路径压缩的并查集

将所有与根节点联通的子节点全部移动到直接与根节点相连，属于最高效的方法，但是有可能有些问题不需要这么高效的解决方法。方法就是通过find的递归查找来做到这一点高效。

```

public class UnionFind {

int root[];

public UnionFind(int size) {

root = new int[size];

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

root[i] = i;

}

}

public int find(int x) {

if(x!=root[x]) root[x]=find(root[x]);

return root[x];

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

root[rootY]=rootX;

}

};

public boolean connected(int x, int y) {

return find(x) == find(y);

}

}

```

### 6、拓扑排序

leetcode

210. 课程表 II:

深度优先搜索和广度优先搜索的拓扑排序