图论第一次内容

图的存储方式

1. 二维数组(vector数组)
2. 邻接表
3. 链式前向星

二维数组/vector数组：g[a][b]表示从a到b有一条权值为g[a][b]的有向边。适合稠密图($n^2 \approx m$)

邻接表：和前向星类似。

链式前向星：适合边多的图。

```
const int maxn = 1e5 + 10;
struct node{
    int nex, to, val;
}Edge[maxn];        //nex表示与当前边起点一样的另一条边在node数组中的位置，to表示那条边的去向
int head[maxn];     //head[i]表示从i出发的点的第一个点边node数组中的位置。
int tot = 0;        //当前已有边的个数。
inline void add(int from, int to, int val){
    Edge[++tot].to = to;
    Edge[tot].val = val;
    Edge[tot].nex = head[from]; //当前结点指向以前的头结点
    head[from] = tot;           //当前结点变为头结点
}
void dfs(int x, int fa){        //以x为起点，遍历前向星
    for(int i = head[x]; i != -1; i = Edge[i].nex){
        printf("%d %d %d\n", x, Edge[i].to, Edge[i].val);
        int v = Edge[i].to;
        if(v == fa) continue;
        dfs(v);
    }
}
```

```
int main()
{
    int from, to, val;
    memset(head, -1, sizeof(head));
    while(cin >> from >> to >> val, from && to && val)
        add(from, to, n);
    int i;
    cin >> i;
    Use(i);
}
```

最短路

1.Dijkstra算法

Dijkstra很好的运用了贪心算法，其思想是一直找离已加入顶点集合的最短边，更新邻点，下面是实现代码：

<1.朴素Dijkstra算法：

> 【题意】：给定一个n个点m条边的有向图，图中可能存在重边和自环，所有边权均为正值。请你求出1号点到n号点的最短距离，如果无法从1号点走到n号点，则输出-1。

直接Dijkstra即可，**注意是有向边，如果是无向边，则需要将两个端点都要存进去**，数组大小开两倍。这个图的范围为一个稠密图，适合朴树的Dijkstra算法，其时间复杂度为$O(v^2)$；

```
#include<bits/stdc++.h>

using namespace std;
const int maxn = 1e5+10;
const int inf = 0x3f3f3f3f;
struct node{
    int nex, to, val;
}Edge[maxn];
int head[maxn], n, m, tot = 0;
int dis[maxn], v[maxn];
void add(int from,int to,int val)
{
    Edge[++tot].nex = head[from];
    Edge[tot].to = to;
    Edge[tot].val = val;
    head[from] = tot;
}
void Dijkstra(int s)              //s到任一节点的距离
{
    for(int i = 0;i <= n; ++i)
        dis[i] = (i == s) ? 0 : inf;      //加0方便处理（也可以不加）
    for(int i = 1;i <= n; ++i)
    {
        int pos = 0;
        for(int j = 1;j <= n; ++j)
```

```
            if(!v[j] && dis[j] < dis[pos])  pos = j;
        v[pos] = 1;
        for(int j = head[pos]; j != -1;j = Edge[j].nex) //找以pos开头的边判断端点是否被访问过
            if(!v[Edge[j].to] && dis[pos] + Edge[j].val < dis[Edge[j].to]){
                dis[Edge[j].to] = dis[pos] + Edge[j].val;
            }
    }
}

int main()
{
    while(cin >> n >> m)
    {
        memset(head, -1, sizeof(head));
        memset(v, 0, sizeof(v));
        for(int i = 1;i <= m; ++i){
            int a, b, val;
            cin >> a >> b >> val;
            add(a, b, val);
            //add(b, a, val);       //如果是无向图要加上这个
        }
        Dijkstra(1);
        dis[n] == inf ? cout << -1 << endl : cout << dis[n] << endl;
    }
}
```

<2:堆优化的Dijkstra算法

堆优化的Dijkstra算法适合稀疏图，其时间复杂度为$O(vloge)$；**无向图数组开两倍！！！**

> 还是上面的题意，有自环和重边，求到n号节点的最小值，不存在输出-1.

show code:

```
#include<bits/stdc++.h>

using namespace std;
const int maxn = 1e5+10;
const int inf = 0x3f3f3f3f;
int n, m, head[maxn], dis[maxn], v[maxn],tot=0;
struct Node{
    int dis, pos;           //dis保存距离 pos保存这个dis的点是pos
    Node(int a = 0, int b = 0): dis(a), pos(b) {}
    friend bool operator < (const Node &a, const Node &b){
        return a.dis > b.dis;
    }       //重载小于号， 令dis越大的Node越小，这样优先队列默认按降序排列，top就是dis最小的
    //相当于大根堆变成小根堆
};
struct node{
    int nex, to, val;
}edge[maxn];

inline void add(int from, int to, int val)
```

```
{
    edge[++tot].nex = head[from];
    edge[tot].to = to;
    edge[tot].val = val;
    head[from] = tot;
}
void Dijkstra(int s)
{
    priority_queue<Node> q;
    for(int i = 1;i <= n; ++i)
        dis[i] = (i == s) ? 0 : inf;
    q.push(Node(0, s));
    while(!q.empty())
    {
        int pos = q.top().pos;      //找到距离最小的点
        q.pop();                     //找到距离离源点最大的点将它更新一波
        if(v[pos])  continue;
        v[pos] = 1;
        for(int i = head[pos]; i != -1; i = edge[i].nex){
            if(dis[pos] + edge[i].val < dis[edge[i].to]){
                dis[edge[i].to] = dis[pos]+edge[i].val;
                q.push(Node(dis[edge[i].to], edge[i].to));
            }
        }
    }
}

int main()
{
    while(cin >> n >> m)
    {
        memset(head, -1, sizeof(head));
        memset(v, 0, sizeof(v));
        tot = 0;
        for(int i = 1; i <= m; ++i){
            int a, b, val;
            cin >> a >> b >> val;
            add(a, b, val);
        }
        Dijkstra(1);
        dis[n] == inf ? puts("-1") : cout << dis[n] << endl;
    }
}
```

2.Floyd算法

Floyd（弗洛伊德）算法的原理是：一直看能不能走别的点使得两点的距离更短(大概就是这个意思)，其核心思想有点区间dp的意思，可以处理带负权的边，其时间复杂度为 $O(v^3)$ 看代码：

> 【题意】：n个点，m条边，q个询问，存在负权和重边，求任意两点之间的最短路径

```
#include<bits/stdc++.h>
```

```
using namespace std;
const int inf = 0x3f3f3f3f;
const int maxn = 210;
int d[maxn][maxn], n, m, q;
void Floyd()
{
    for(int k = 1; k <= n; ++k)  //枚举中间点
        for(int i = 1; i <= n; ++i)         //枚举变得两个顶点
            for(int j = 1; j <= n; ++j){
                if(d[i][k] == inf || d[k][j] == inf)  continue;
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
}

int main()
{
    scanf("%d %d %d",&n,&m,&q);
    for(int i = 1;i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            d[i][j] = (i == j) ? 0:inf;       //初始化
    for(int i = 1; i <= m; ++i){
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        d[a][b] = min(d[a][b], c);         //处理重边
    }
    Floyd();
    for(int i = 1;i <= q; ++i){
        int a, b;
        scanf("%d %d", &a, &b);
        if(d[a][b] == inf)    printf("impossible\n");
        else{
            printf("%d\n", d[a][b]);
        }
    }
}
```

要想输出路径，其实也很简单，只要在加个Path数组记录一下路径即可：

```
void Floyd()
{
    for(int i=1;i<=n;++i){
        for(int j=1;j<=n;++j)
            path[i][j]=(d[i][j]==inf)?-1:j;     //初始化路径，输入之后再初始化
    }
    for(int k = 1; k <= n; ++k)
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j <= n; ++j){
                if(d[i][k] == inf|| d[k][j] == inf)  continue;
                if(d[i][j] > d[i][k] + d[k][j]){
                    d[i][j] = d[i][k] + d[k][j];
                    path[i][j] = path[i][k];
                }
            }
```

```
}
void print(int start, int end)
{
    printf("%d ", start);
    while(start != end)
    {
        printf("%d ", path[start][end]);
        start = path[start][end];
    }
}
```

Floyd还可以用来传递闭包：

```
void floyd()
{
    for(int k = 1; k <= n; ++k){
        for (int i = 1;i <= n; ++i){
            for (int j = 1; j <= n; ++j){
                d[i][j] = d[i][j] | (d[i][k] && d[k][j]);
            }
        }
    }
}
```

3.Bellman-Ford算法

Bellman-Ford是基于迭代思想，它扫描所有的边（x，y，z）,如果：$dis[y] > dis[x] + z$，则更新 $dis[y] = dis[x] + z$，一直到算法结束。

其优点是**允许图中存在负权边，且时间复杂度为**$O(ve)$；

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1e5+10;
const int inf = 0x3f3f3f3f;
struct node
{
    int from, to, val;
}edge[maxn];
int  dist[maxn], n, m;                //n为点的个数，m为边的个数
bool Bellman_Ford(int s)
{
    for(int i = 1; i <= n; ++i)
        dist[i] = (i == s) ? 0 : inf;        //初始化
    for(int i = 1; i < n; ++i)
        for(int j = 1; j <= m; ++j){
            if(dist[edge[j].from] + edge[j].val < dist[edge[j].to])
                dist[edge[j].to] = dist[edge[j].from] + edge[j].val;
        }
    bool flag = 1;
    // 判断是否有负环路
```

```
    for(int i = 1; i <= m; ++i)
        if(dist[edge[i].to] > dist[edge[i].from] + edge[i].val){
            flag = 0;
            break;
        }
    return flag;
}
int main()
{
    scanf("%d %d", &n, &m);        //s为源点
    for(int i = 1; i <= m; ++i)
        scanf("%d %d %d", &edge[i].from, &edge[i].to, &edge[i].val);
    if(Bellman_Ford(1) && dist[n] != inf)
        printf("%d\n", dist[n]);
    else{
        printf("impossible\n");
    }
    system("pause");
}
```

4.SPFA算法

SPFA算法又叫做队列优化的Bellman-Ford算法，其时间复杂度为$O(ke)$，其中k是一个很小的常数。不过在特殊情况下其时间复杂度会退化到$O(ev)$。

它的算法流程和Bellman-Ford类似：先建立一个队列，队列中只包含一个起点s，接着扫描队头所有的出边（x，y，z）,如果：$dis[y] > dis[x] + z$，则更新$dis[y] = dis[x] + z$，直至算法结束。

裸模板：

```
#include<bits/stdc++.h>

using namespace std;
const int inf = 0x3f3f3f3f;
const int maxn = 1e5+10;;
int n,m,w;
struct Edge{
    int nex, to, val;
}edge[maxn];
int head[maxn], dis[maxn], vis[maxn], mark[maxn], tot;
void init()
{
    memset(head, -1, sizeof(head));
    tot = 0;
}
void add(int from, int to, int val)
{
    edge[++tot].to = to;
    edge[tot].val = val;
    edge[tot].nex = head[from];
    head[from] = tot;
}
bool SPFA(int s)
```

```
{
    for(int i = 1;i <= n; ++i){
        mark[i] = vis[i] = 0;                     //mark记录每个点入队列次数
        dis[i] = inf;
    }
    queue<int> q;
    q.push(s);                          //我们只需要判断负环，随便找一个起点就好
    dis[s]=0;
    vis[s]=1;                           //入队列
    mark[s]++;
    while(!q.empty())
    {
        int u = q.front(); q.pop();
        vis[u] = 0;                 //出队列
        for(int i = head[u]; i != -1; i = edge[i].nex)
        {
            int v = edge[i].to;
            if(dis[v] > dis[u] + edge[i].val)
            {
                dis[v] = dis[u] + edge[i].val;
                if(!vis[v])                     //不在队列中的时候入队
                {
                    q.push(v);
                    mark[v]++;
                    vis[v] = 1;
                }
                if(mark[v] >= n)      return false; //存在负环
            }
        }
    }
    return true;
}
int main()
{
    init();
    int u, v, z;
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= m; i++)
    {
        scanf("%d %d %d", &u, &v, &z);
        add(u, v, z);
    }
    if(SPFA(1) && dis[n] != inf)  printf("%d\n", dis[n]);
    else        printf("impossible\n");
}
```

最小生成树

在一个n个顶点的图选出n-1条边，使这n-1条边的权值和最小，这样的树叫做最小生成树。

1.Prime

该算法很好的运用了贪心算法，其基本思想是先随便选取一个结点，找出与该结点权值最小的结点，将该结点与之前的结点相连，并将该点加入集合，如此循环，直至找出所有的点，这样找出来的树就是最小生成树。由于该算法的时间复杂度与定点有关而与边数无关，所以**适合求稠密网的生成树。**

```
#include<bits/stdc++.h>

using namespace std;
const int inf=0x3f3f3f3f;
const int maxn=1e3;
int n,m,cost[maxn][maxn],dis[maxn],v[maxn];
int prime()
{
    int res=0,pos;
    memset(dis,inf,sizeof(dis));
    memset(v,0,sizeof(v));
    for(int i=1;i<=n;++i)
        dis[i]=cost[1][i];
    v[1]=1;
    for(int i = 1;i < n; ++i)
    {
        pos = 0;
        for(int j = 2; j <= n; ++j){
            if(!v[j] && dis[j] < dis[pos])
                pos = j;
        }
        if(pos == 0)  return inf;
        v[pos] = 1;
        res += dis[pos];
        for(int j = 2; j <= n; ++j)
            if(!v[j]) dis[j] = min(dis[j], cost[pos][j]);
    }
    return res;
}

int main()
{
    ios::sync_with_stdio(false);
    while(cin>>n>>m)
    {
        memset(cost,inf,sizeof(cost));
        for(int i=1;i<=m;++i){
            int a,b,c;
            cin>>a>>b>>c;
            cost[a][b]=cost[b][a]=min(cost[a][b],c);
        }
        prime()==inf?cout<<"impossible"<<endl:cout<<prime()<<endl;
    }
    return 0;
}
```

2.Kruskal

先将所有边排个序，然后从权值小的开始加，如果两个点不在同一个连通分支里面，则将它们加入到一个连通分支里面，直到一共加入了n - 1条边。时间复杂度$O(mlogm)$。

```
#include<bits/stdc++.h>

using namespace std;
const int inf = 0x3f3f3f3f;
const int maxn = 2e5+10;
int fa[maxn], ran[maxn], n, m;      //并查集中的父结点数组和秩序数组
struct node{
    int l, r, val;        //左右端点及权值
}arr[maxn];
inline void init(){
    for(int i = 1; i <= n; ++i)
        fa[i] = i, ran[i] = 0;
}
int ffind(int x){
    return x == fa[x] ? x : fa[x] = ffind(fa[x]);
}
void unite(int x, int y)
{
    int fx = ffind(x);
    int fy = ffind(y);
    if(fx == fy)      return;
    if(ran[fx] < ran[fy]) fa[fx]=fy;
    else{
        fa[fy] = fx;
        if(fa[fx] == fa[fy])  ran[fx]++;
    }
}
bool operator < (node &a, node &b){
    return a.val < b.val;
}
int Kruskal()
{
    int res = 0,tot = 0;
    for(int i = 1; i <= m; ++i)
    {
        if(ffind(arr[i].l) != ffind(arr[i].r))
        {
            tot++;
            res += arr[i].val;
            unite(arr[i].l, arr[i].r);
        }
        if(tot == n - 1)    return res;
    }
    return inf;
}

int main()
{
```

```
    while(~scanf("%d %d", &n, &m))
    {
        for(int i = 1; i <= m; ++i)
            cin >> arr[i].l >> arr[i].r >> arr[i].val;
        sort(arr + 1, arr + m + 1);
        init();
        int res = Kruskal();
        res == inf ? puts("impossible") : cout << res << endl;
    }
}
```

拓扑排序

定义：拓扑排序是指在**有向无环图**中，将所有的结点进行排序，最终得出的序列称为拓扑序。

```
#include<bits/stdc++.h>

using namespace std;
const int maxn = 1e5 + 10;
int n, m, tot = 0, head[maxn], seq[maxn], d[maxn];
struct node{
    int nex, to;
}edge[maxn];
void init()
{
    tot = 0;
    memset(head, -1, sizeof(head));
    memset(seq, 0, sizeof(seq));        //保存拓扑排序的结点
    memset(d, 0, sizeof(d));            //保存每个结点的入度
}
void add(int from, int to)
{
    edge[++tot].to = to;
    edge[tot].nex = head[from];
    head[from] = tot;
}
void topsort()
{
    queue<int> q;
    //如果编号要按顺序输出可有priority_queue<int,vector<int>,greater<int> >
    for(int i = 1; i <= n; ++i)
        if(!d[i])   q.push(i);      //先把起点压入队列
    int cnt = 0;
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        seq[++cnt] = u;
        for(int i = head[u];i != -1;i = edge[i].nex)
            if(--d[edge[i].to] == 0)   q.push(edge[i].to);  //前驱全部完成才能压
    }
    if(cnt != n){
```

```
        printf("-1\n");     //存在环
        return;
    }
    for(int i = 1;i <= n; ++i){
        printf("%d%c",seq[i],i == n ? '\n' : ' ');
    }
}

int main()
{
    while(~scanf("%d %d", &n, &m)){
        init();
        for(int i = 1; i <= m; ++i){
            int u,v;
            scanf("%d %d", &u, &v);
            add(u, v);
            d[v]++;
        }
        topsort();
    }
}
```