常用代码模板3----搜索与图论

树与图的存储

树是一种特殊的图，与图的存储方式相同。
对于无向图中的边ab，存储两条有向边a->b, b->a。
因此我们可以只考虑有向图的存储。

(1) 邻接矩阵：`g[a][b]` 存储边a->b

(2) **邻接表(3种方法)：**

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;

// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

```
struct Node
{
    int b, w, next;
} edge[M];
int idx, head[N];
void add(int a, int b, int w)
{
    edge[idx].b = b, edge[idx].w = w, edge[idx].next = head[a];
    head[a] = idx ++;
}
idx = 1;        // 初始化
```

```
// 使用vector
vector<pair<int, int>>G[N + 5];     // 后继节点, 权值
void add(int a, int b, int w)
{
    G[u].push_back({v, w});
}
G.clear();      // 初始化
for (int i = 0; i < G[u].size(); i++) {
    int v = G[u][i].first;      // 遍历整个图
    // do something
}
```

树与图的遍历

时间复杂度 O(n+m)O(n+m), nn 表示点数，mm 表示边数

(1) 深度优先遍历

```
int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
```

(2) 宽度优先遍历

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

while (q.size())
{
    int t = q.front();
    q.pop();

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!s[j])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

拓扑排序

时间复杂度 $O(n+m)$, n表示点数, m表示边数

```
bool topsort()
{
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];
```

```
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

朴素dijkstra算法

时间复杂是$O(n^2 + m)$, n **表示点数，** m **表示边数**

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化版dijkstra

时间复杂度 $O(mlogn)$, n**表示点数，** m **表示边数**

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定
```

```
// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Bellman-Ford算法

时间复杂度$O(nm)$, n **表示点数，** m **表示边数**

```
int n, m;           // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离
int backup[N];
struct Edge         // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
} edges[M];

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，
    // 由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for(int i = 0; i < k; ++i)
    {
        memcpy(backup, dist, sizeof dist);
        for(int j = 0; j < m; ++j)
```

```
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if(backup[a] + w < dist[b]) dist[b] = backup[a] + w;
        }
    }
    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

---

spfa 算法(队列优化的Bellman-Ford算法)

时间复杂度 平均情况下 $O(m)$，最坏情况下 $O(nm)$, n 表示点数，m 表示边数

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

---

spfa判断图中是否存在负环

时间复杂度是 $O(nm)$, n 表示点数， m 表示边数

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。

    queue<int> q;
    for (int i = 1; i <= n; i ++ )
    {
        q.push(i);
        st[i] = true;
    }

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

---

floyd算法

时间复杂度是 $O(n^3)$, n 表示点数

```
初始化:
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
```

```
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
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] = min(d[i][j], d[i][k] + d[k][j]);
}
```

朴素版prim算法

时间复杂度是 $O(n^2 + m)$, n **表示点数，** m **表示边数**

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

Kruskal算法

时间复杂度是 $O(mlogm)$, n**表示点数，** m**表示边数**

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
```

```
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通，则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

染色法判别二分图

时间复杂度是 $O(n+m)$, n 表示点数， m 表示边数

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示为染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
```

```
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

匈牙利算法

时间复杂度是$O(nm)$, n **表示点数，** m **表示边数**

```
int n1, n2;     // n1表示第一个集合中的点数，n2表示第二个集合中的点数
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
    if (find(i)) res ++ ;
}
```

AOE关键路径

AOE 网性质:

1. 只有在某**顶点**所代表的事件发生后，从该**顶点**出发的各个有向边所代表的**活动**才能开始
2. 只有在进入某**顶点**的各有向边所代表的**活动**都已经结束，该顶点所代表的**事件**才能发生

关键路径性质：每条边(活动)的最早开始时间 = 最晚开始时间 $\Leftrightarrow VE[i] = VL[j] - edge[i,j]$

$VE[j] = max\{VE[i_p] + edge[i_p, j]\}, p = 1, 2, \ldots, k$

$VL[i] = min\{VL[j_p] - edge[i, j_p]\}, p = 1, 2, \ldots, k$

$EE[i,j] = VE[i]$

$EL[i,j] = VL[j] - edge[i,j]$

活动的最迟发生时间 = 活动的弧头所指向的事件的最迟发生时间 - 这个活动所消耗的时间

```
const int N = 10010, M = 50010;
// 一种新的前向*存储图的方式
struct Node
{
    int b, w, next;
} aov[2][M];
int head[2][N], idx[2];
int indegree[2][N];      // 0代表正向图, 1代表反向图
int ve[N], vl[N];        // 顶点的最早和最晚开工时间
int n, m, Mx;            // Mx代表整个工程的最晚开工时间

void add(int a, int b, int w, int type)
{
    int t = idx[type];
    aov[type][t].b = b;
    aov[type][t].w = w;
    aov[type][t].next = head[type][a];
    head[type][a] = idx[type] ++;
}
void topsort(int val[], int type)
{
    queue<int> q;
    for(int i = 1; i <= n; ++i)
        if(indegree[type][i] == 0)
            q.push(i);
    if(type) fill(vl + 1, vl + 1 + n, ve[n]);    // 相当于赋值INF

    while(q.size())
    {
        int t = q.front();
        q.pop();
        for(int i = head[type][t]; i; i = aov[type][i].next)
        {
            int j = aov[type][i].b, w = aov[type][i].w;;
            indegree[type][j] --;
            if(indegree[type][j] == 0) q.push(j);
            // 更新最早和最晚开工时间
            if(!type) val[j] = max(val[j], val[t] + w);
            else val[j] = min(val[j], val[t] - w);
        }
    }
}
```

```
void calc()
{
    // 寻找字典序最小的关键路径
    for(int k = 1; k < n; )
    {
        int v = n + 1;
        for(int i = head[0][k]; i; i = aov[0][i].next)
        {
            int ver = aov[0][i].b, w = aov[0][i].w;
            if(ve[k] == vl[ver] - w)        // [K, ver]这条边属于关键路径
                v = min(v, ver);
        }
        cout << k << ' '<< v << endl;
        k = v;
    }
}

int main()
{
    idx [0] = idx[1] = 1;
    for(int i = 1; i <= m; ++i)
    {
        int a, b, w;
        cin >> a >> b >> w;
        add(a, b, w, 0);        // 正向建图
        add(b, a, w, 1);        // 反向建图
        indegree[0][b] ++;      // 入度
        indegree[1][a] ++;      // 反向入读
    }
    topsort(ve, 0);             // 正向计算最早开始时间, 取Max
    topsort(vl, 1);             // 反向计算最晚开始时间, 取Min
    cout << ve[n] << endl;      // 输出关键路径长度
    calc();
    return 0;
}
```