图论

Flood Fill

算法用途：找到某个点所在的**连通块**

基于不同的用途，可以额外维护更多的信息，比如**连通块中点的个数**

基本实现有两种：BFS 与 DFS

时间复杂度均为 $O(n+m)$，其中 n 为点数，m 为边数

基础:

原题链接：AcWing 1113. 红与黑

BFS 写法

- `st[i][j]` 表示 (i,j) 已经遍历过
- 只要当前点没有出界、没有被遍历过、当前点在连通块内，我们就将当前点加入到队列中去

- 如果我希望统计连通块中点的数量，那么我需要在**每次遍历一个新的点时就统计一次**，即在 `for` 循环内统计
- 如果我希望统计连通块的数量，那么在 Flood Fill 的入口出进行通统计

完整代码：

```
#include <iostream>
#include <cstring>
#include <queue>

#define x first
#define y second

using namespace std;
typedef pair<int, int> PII;
const int N = 25;

char g[N][N];
bool st[N][N];

int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

int n, m;

void bfs(int sx, int sy, int& sum)
{
    queue<PII>q;
    q.push({sx, sy});
    st[sx][sy] = true;

    while(q.size())
    {
        auto it = q.front();
        q.pop();
        int x = it.x, y = it.y;
        for(int i = 0; i < 4; i ++)
        {
            int nx = x + dx[i], ny = y + dy[i];
            if(nx < 0 || ny < 0 || nx >= m || ny >= n) continue;
            if(st[nx][ny]) continue;
            if(g[nx][ny] == '#') continue;

            q.push({nx, ny});
            st[nx][ny] = true;
            sum++;

        }
    }
}

int main()
{
```

```
    while(cin >> n >> m, n || m)
    {
        memset(g, 0, sizeof g);
        memset(st, false, sizeof st);
        for(int i = 0; i < m; i ++)
            cin >> g[i];
        int cnt = 0;
        for(int i = 0; i < m; i ++)
            for(int j = 0; j < n; j ++)
                if(g[i][j] == '@')
                {
                    int sum = 0;
                    bfs(i, j, sum);
                    cnt += sum;
                }
        cout << cnt + 1 << endl;
    }

    return 0;
}
```

DFS 写法

```
#include <iostream>
#include <cstring>


using namespace std;
typedef pair<int, int> PII;
const int N = 25;

char g[N][N];
bool st[N][N];

int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

int n, m;

int dfs(int sx, int sy)
{
    st[sx][sy] = true;
    int cnt = 1;

    for(int i = 0; i < 4; i ++)
    {
        int nx = sx + dx[i], ny = sy + dy[i];
        if(nx < 0 || ny < 0 || nx >= m || ny >= n) continue;
        if(g[nx][ny] == '#') continue;
        if(st[nx][ny]) continue;

        cnt += dfs(nx, ny);
```

```
    }
    return cnt;
}

int main()
{
    while(cin >> n >> m, n || m)
    {
        memset(g, 0, sizeof g);
        memset(st, false, sizeof st);
        for(int i = 0; i < m; i ++)
            cin >> g[i];
        int cnt = 0;
        for(int i = 0; i < m; i ++)
            for(int j = 0; j < n; j ++)
                if(g[i][j] == '@')
                {
                    cnt += dfs(i, j);
                }
        cout << cnt << endl;
    }

    return 0;
}
```

变式

原题链接：AcWing 1233. 全球变暖

对于每一块连通块（陆地），我们 BFS 时统计出连通块中点的数量 cnt ，同时再统计边界的数量 $bount$ （只要某个点上下左右存在海洋，那么就是边界）

只要 $cnt == bount$ 那么这块陆地将会被淹没，我们统计所有不会被淹没的陆地即可

完整代码：

```
#include <iostream>
#include <queue>

#define x first
#define y second

using namespace std;

const int N = 1e3 + 10;
typedef pair<int, int> PII;

char g[N][N];
bool st[N][N];

int n;
```

```cpp
int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

void bfs(int sx, int sy, int& cnt, int& bound)
{
    queue<PII>q;
    q.push({sx, sy});
    st[sx][sy] = true;
    while(q.size())
    {
        auto t = q.front();
        q.pop();

        st[t.x][t.y] = true;
        cnt++;

        bool is_bount = false;

        for(int i = 0; i < 4; i ++)
        {
            int nx = t.x + dx[i], ny = t.y + dy[i];
            if(nx >= 0 && ny >= 0 && nx < n && ny < n)
            {
                if(st[nx][ny]) continue;
                if(g[nx][ny] == '.') is_bount = true;
                else
                {
                    q.push({nx, ny});
                    st[nx][ny] = true;
                }
            }
        }
        if(is_bount) bound++;
    }

}

int main()
{
    cin >> n;
    for(int i = 0; i < n; i ++)
        cin >> g[i];
    int ans = 0;
    for(int i = 0; i < n; i ++)
        for(int j = 0; j < n; j ++)
            if(!st[i][j] && g[i][j] == '#')//当前点没有被遍历过并且是陆地
            {
                int cnt = 0, bound = 0;//cnt表示陆地数量，bound表示会被淹没的数量
                bfs(i, j, cnt, bound);
                if(cnt == bound) ans++;
            }
    cout << ans << endl;
    return 0;
}
```

BFS

BFS 中，每个点只会被遍历一次，因此时间复杂度为 $O(n+m)$，其中 n 为点数，m 为边数

BFS 需要额外记录点的坐标，因此需要用 `pair<int, int>` 来存储 其次，我们还需要确定哪些点已经遍历过，因此需要用 `st[i][j]` 来标记 (i,j) 这个点是否已经遍历过

在初始化时，我们将**起点**压入队列，**并用 `st` 标记起点**

原题链接：AcWing 1562. 微博转发

如果 A 关注了 B，那么 A 便会转发 B 的帖子，因此我们建立一条从 B 指向 A 的**有向边**

由于层数最大为 L 层，因此我们需要统计**所有路径长度不超过 L 的点的数量**

由于点的边权全部都是 1，因此我们直接用 BFS 来统计即可

- 点的边权不同，我们考虑 SPFA
- 如果点的边权只有 0 和 1，考虑双端队列
- 如果点的边权只有 1，考虑 BFS

完整代码：

```
#include <iostream>
#include <cstring>
#include <queue>

using namespace std;

const int N = 1e5 + 10;

int h[N], e[N], ne[N], idx;
bool st[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

int n, L;

int bfs(int u)
{
    memset(st, false, sizeof st);

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

    int cnt = 0;
```

```
    for(int i = 1; i <= L; i ++)//由于我们需要遍历 L 层，因此在队列遍历的时候只能遍历本层的节点，因此需要取队列的大小
    {
        int sz = q.size();
        while(sz--)
        {
            int t = q.front();
            q.pop();
            for(int j = h[t]; j != -1; j = ne[j])
            {
                int k = e[j];
                if(!st[k])
                {
                    st[k] = true;
                    q.push(k);
                    cnt++;
                }
            }
        }
    }
    return cnt;
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> L;
    for(int i = 1; i <= n; i ++)
    {
        int cnt = 0;
        cin >> cnt;
        for(int j = 1; j <= cnt; j ++)
        {
            int x;
            cin >> x;
            add(x, i);
        }
    }
    int k;
    cin >> k;
    while(k--)
    {
        int q;
        cin >> q;
        cout << bfs(q) << endl;
    }
    return 0;
}
```

拓扑排序

基础

只有有向图才有拓扑排序，无向图是没有拓扑排序的，算法思路如下：

1. 统计所有点的入度数，将入度为 0 的点加入队列中
2. 取队头元素 cur ，枚举 cur 所有出边指向的点 j，将点 j 的入度减一
3. 如果 j 的入度为 0 ，那么将其加入队列中
4. 最后统计队列中的元素，是否恰好为 N 个，即可知道是否存在拓扑排序

原题链接：AcWing 848. 有向图的拓扑序列

代码如下：

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 1e5 + 10;

int h[N], e[N], ne[N], idx;
int q[N], d[N];//q为队列，d用于记录每个点的入度数
               //对于孤立的点，入度数直接为0

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

int n, m;

bool topsort()
{
    int hh = 0, tt = -1;
    for(int i = 1; i <= n; i ++)//先将入度为0的点加入队列
    {
        if(d[i] == 0) q[++tt] = i;
    }
    while(hh <= tt)
    {
        int t = q[hh++];
        for(int i = h[t]; i != -1; i = ne[i])//每次删除一个点及其所有出边，如果出现入
度为0的点，则将其加入队列中
        {
            int j = e[i];
            if(--d[j] == 0) q[++tt] = j;
        }
    }
    return tt == n - 1;//只需要检验队列中的元素是否达到n个即可
}

int main()
```

```
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 1; i <= m; i ++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        d[b]++;
    }

    if(topsort())
    {
        for(int i = 0; i < n; i ++) cout << q[i] << " ";
    }
    else cout << "-1" << endl;

    return 0;
}
```

变式

有向无环图的构造

原题链接：AcWing 3696. 构造有向无环图

看到 $\mathrm{DAG(DirectedAcyclicGraph)}$，马上反应出是拓扑排序

在一个既包含无向边又包含有向边的图中，如果有向边本身构成环，那么不可能会有拓扑排序

如果有向边没有构成环，对于每一条无向边而言，只要**从前指向后**，就同样能保证不会构成环（构成环的条件为**至少存在一条边从后指向前**）

也就是说，如果有向边构成的图本身存在拓扑排序，那么整个图就一定存在拓扑排序，对于无向边而言，只需要从前指向后即可（在拓扑排序中的前后关系）

在代码实现上，我们需要一个数组 `pos[i]` 来记录拓扑排序中第 i 个点的编号，即拓扑排序的顺序

```
#include <iostream>
#include <cstring>

using namespace std;

const long long N = 2e5 + 10, M = N;

int h[N], e[M], ne[M], idx;

int q[N], d[N], pos[N];

void add(int a, int b)
{
```

```
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

struct Edge
{
    int a, b;
}edge[M];

int n, m;

bool topsort()
{
    int hh = 0, tt = -1;
    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;
        }
    }
    return tt == n - 1;
}

int main()
{
    int T;
    cin >> T;
    while(T--)
    {
        memset(h, -1, sizeof h);
        memset(d, 0, sizeof d);
        idx = 0;
        scanf("%d%d", &n, &m);
        int cnt = 0;
        for(int i = 1; i <= m ; i ++)
        {
            int t, a, b;
            scanf("%d%d%d", &t, &a, &b);
            if(!t) edge[cnt++] = {a, b};
            else
            {
                add(a, b);
                d[b]++;
            }
        }
        if(!topsort()) cout << "NO" << endl;
        else
        {
```

```
                cout << "YES\n";
                for(int i = 1; i <= n; i ++)
                    for(int j = h[i]; j != -1; j = ne[j])
                        printf("%d %d\n", i, e[j]);
                for(int i = 0; i < n; i ++)
                    pos[q[i]] = i;
                for(int i = 0; i < cnt; i ++)
                {
                    int x = edge[i].a, y = edge[i].b;
                    if(pos[x] > pos[y]) swap(x, y);
                    printf("%d %d\n", x, y);
                }
            }
        }
    return 0;
}
```

Dijkstra

基础

Dijkstra 算法**只能适用于边权为正的情况**

朴素 Dijkstra

朴素 Dijkstra 算法只能用在稠密图当中，用**邻接矩阵存储**，即边数是点数的平方

时间复杂度为 $O(n^2)$ ，因为会将所有点全部遍历一次

用于求某个点到其余所有点的最短距离，这里以 1 号点为例

定义集合 S 表示当前为确定最短距离的点的集合，$dist_i$ 表示第 i 号点到起点的最短距离，算法步骤如下：

1. 初始化 $dist$ 数组，全部为正无穷，并将起点初始化成 0
2. 循环 n 次，每次找到一个不在集合 S 中并且距离起点最短的点 t
3. 将 t 加入集合内，并用 t 更新其余点到起点的距离
4. 考察终点是否为正无穷，进而可以得到起点到终点的最短距离

完整代码如下：

```
const int N = 510;
int g[N][N];
bool st[N];//当前点是否在已确定最短距离
int dist[N];//当前点到起点的距离
int n;
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);//先初始化dist数组
    dist[1] = 0;//以1作为起点，因此初始化1的距离

    for(int i = 1; i <= n; i ++)
```

```
    {
        int t = -1;
        //每次找不在集合中并距离起点最短的点
        for(int j = 1; j <= n; j ++)
        {
            if(!st[j] && (t == -1 || dist[j] < dist[t])) t = j;
        }
        //将当前点加入到集合内，表示当前点已确定最短距离
        st[t] = true;
        //用t更新其他点到起点的距离
        for(int i = 1; i <= n; i ++)
            dist[i] = min(dist[i], dist[t] + g[t][i]);
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

使用时需要先对 `g[i][j]` 初始化，全部初始化成 INF ，`g[i][i]` 初始化成 0

堆优化 Dijkstra

用于稀疏图，即点数与边数同数量级，用**邻接表存储**，时间复杂度为 $O(n\log m)$

朴素版 Dijkstra 内，每次找距离起点最近的点都需要遍历一次所有点，这里的时间复杂度为 $O(n^2)$ ，考虑用堆来优化，此处时间复杂度变为 $O(1)$

在用点 t 更新其他点的距离时，相当于遍历了 t 的所有出边，原先时间复杂度为 $O(nm)$ ，此时由于是在堆中修改，因此变为 $O(n\log m)$

代码如下:

```
typedef pair<int, int> PII;
int dist[N];//到1号点的距离
bool st[N];//该点是否已确认过最短距离

int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    priority_queue<PII, vector<PII>, greater<PII>>q;
    q.push({0, 1});//顺序一定是：距离，点的编号
    d[1] = 0;

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

        int ver = q.second, dis = q.first;
        if(st[ver]) continue;//如果当前点重复，直接跳过
        st[ver] = true;
        for(int i = h[ver]; i != -1; i = ne[i])
```

```
        {
            int j = e[i];//e[i]表示点的编号，w[i]表示ver到e[i]的距离
            if(dist[j] > dis + w[i])//如果从1到j的距离大于从1到ver加上从ver到j的距
离，就更新d[j]
            {
                dist[j] = dis + w[i];
                q.push({d[j], j});
            }
        }
    }
    if(d[n] == 0x3f3f3f3f) return -1;
    return d[n];
}
```

变式

将其余点映射到虚拟节点

原题链接：AcWing 1488. 最短距离

本题很明显是求单源最短路，但有一个问题是，我们需要对每个点都执行一次 Dijkstra 时间上一定会超时

由于点的编号从 1 开始，因此我们可以将所有商店都映射到编号为 0 的点，这样在求每个点到最近商店的距离时只需要求该点到 0 号点的距离即可

具体地，我们从编号为 0 的点连一条**长度为 0 的边到各个商店**，这样就相当于将各个商店都映射到 0 这个点了

需要注意的是，此时的边数变为 $3N$，因为每条无向边需要连 $2N$，还需要额外再连 N 条从 0 指向各个商店的边

完整代码：

```
#include <iostream>
#include <cstring>
#include <queue>
#include <unordered_map>

using namespace std;

const int N = 1e5 + 10;

int h[N], e[3 * N], w[3 * N], ne[3 * N], idx;


typedef pair<int, int> PII;

int d[N];
bool st[N];

void add(int a, int b, int c)
{
```

```
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

int n, m;

void dijkstra()
{
    memset(d, 0x3f, sizeof d);
    memset(st, false, sizeof st);
    d[0] = 0;
    priority_queue<PII, vector<PII>, greater<PII>>q;
    q.push({0, 0});

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

        int ver = t.second, dis = t.first;
        if(st[ver]) continue;
        st[ver] = true;

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

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    while(m--)
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
        add(b, a, c);
    }
    cin >> m;
    while(m--)
    {
        int x;
        cin >> x;
        add(0, x, 0);//设定0为指向各个商店，距离为0的标点
    }
    dijkstra();
    cin >> m;
```

```
    while(m--)
    {
        int x;
        cin >> x;
        cout << d[x] << endl;
    }
    return 0;
}
```

构建虚拟节点 | 只更新部分点到起点的距离

原题连接：AcWing 903. 昂贵的聘礼

设点数为 n ，地位限制为 m

先不考虑地位，如果求 1 号点到其他所有点的最小距离，这个时候直接**购买这个点本身的花费**很难表示，所以转变思路：

新建一个虚拟节点（设其编号为 0），该节点指向原先的 n 个节点，路径长度为直接购买第 i 个节点的花费

此时问题转化成，求从虚拟节点到 1 号点的最短距离（由于边的指向是从虚拟节点开始，因此**起点要从 1 号点变为虚拟节点**）

然后接着考虑地位的问题，设第 i 号点的地位为 $level[i]$，由于与 $level[1]$ 的差距不能超过 m ，因此以 1 号点为中心，区间大小为 $[level[1]-m, level[1]+m]$ 内的点都可以被选择

具体地，用 i 循环 $[leve[1]-m, level[1]]$ ，每次所表示的区间大小为 $[i, i+m]$，此时便可以遍历所有的区间

只要下个点的等级处于区间范围内，便**可以更新从起点到下个点的距离**

由于加入虚拟点后，点的个数变为 $n+1$ ，因此需要循环 $n+1$ 次，并且在更新距离时，也需要将虚拟节点算入（也就是从 0 开始）

完整代码：

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 110, INF = 0x3f3f3f3f;

int n, m;
int w[N][N], level[N];
int dist[N];
bool st[N];

int dijkstra(int l, int r)
{
    memset(dist, 0x3f, sizeof dist);
```

```
    memset(st, false, sizeof st);
    dist[0] = 0;//以虚拟点为起点

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

        st[t] = true;

        for(int j = 0; j <= n; j ++)
            if(level[j] >= l && level[j] <= r) dist[j] = min(dist[j], dist[t] +
w[t][j]);
    }
    return dist[1];//以1号点为终点

}

int main()
{
    cin >> m >> n;

    memset(w, 0x3f, sizeof w);//w需要初始化
    for(int i = 0; i <= n; i ++)
        w[i][i] = 0;
    for(int i = 1; i <= n; i ++)
    {
        int price, cnt;
        cin >> price >> level[i] >> cnt;
        w[0][i] = min(w[0][i], price);//从虚拟节点连到当前节点
        while(cnt--)
        {
            int id, cost;
            cin >> id >> cost;
            w[id][i] = min(w[id][i], cost);//连一条边
        }
    }
    int ans = INF;
    for(int i = level[1] - m; i <= level[1]; i ++)//这里枚举小于酋长地位的，下面再加
上m就可以枚举到大于酋长地位的
        ans = min(ans, dijkstra(i, i + m));//酋长地位没说是最大的
    cout << ans << endl;
    return 0;
}
```

SPFA

基础

SPFA 求最短路

只要图中没有负环，就可以使用 SPFA 来求最短路问题（有负环就不存在最短路了）

既可以适用于边权为正，也可以适用于边权为负，时间复杂度一般是 $O(m)$ ，最坏是 $O(nm)$

算法思路：在用点来更新距离时，如果这个点到起点的距离本身被更新过，就用该点去更新其他点，否则忽视掉该点，具体思路如下：

`st[i]` 表示点 i **是否在队列中**，`dist[i]` 表示点 i 到起点的距离

- 初始化：`dist[1] = 0, st[1] = true` ，起点入队
- 取队头元素 t 并将其出队，此时有 `st[t] = false`
- 遍历 t 的所有出边指向的点 j ，如果 `dist[j]` 距离变小并且**点** j **不在队列中**，将 j 加入队列中

完整代码：

```
const int N = 1e5 + 10;
int h[N], e[N], w[M], ne[N], idx;
int dist[N];
bool st[N];

bool spfa()
{
    memset(dist, 0x3f, sizeof dist);
    queue<int>q;
    q.push(1);
    st[1] = true;

    while(q.size())
    {
        int 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[i] + w[i];
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    //边权为负，此时最大值可能会减小
    return d[n] > 0x3f3f3f3f / 2 ? false : true;
}
```

相关题目：AcWing 851. spfa求最短路

SPFA 判断是否存在负环

在一个**只有** n 个点的有向图中（允许边权为负），如果从起点到点 i 的最短路径中有 n 条边，那么说明这条最短路径当中一共有 $n+1$ 个点，根据抽屉原理可知，一定至少有两个点是重合的，此时说明一定存在环

由于这条路径为**当前最短路径**，说明环的权值一定为负，否则不可能出现最短路径长度变小（事实上存在负环的图是没有最短路径的）

因此只需要在 SPFA 的基础上，新加一个 `cnt[i]` 数组，表示从起点到 点 i 最短路径当中边的个数，只要存在 `cnt[i] >= n`，此时说明一定存在负环

在初始化时，需要将所有点全部入队而不能将起点入队，这是因为从起点到各个点的最短路径中，不一定包含负环

而对于距离的初始化而言，初始化起点的距离为 0 ，其余点为 INF ，因为需要以起点作为基准来考察其余点到起点的距离是否发生改变

完整代码：

```
const int N = 1e5 + 10;
int h[N], e[N], w[N], ne[N], idx;
int dist[N];//表示起点到当前点的距离
int cnt[N]; //表示从起点到当前点最短路径中的边的数量
bool st[N]; //表示当前点是否在队列中

bool spfa()
{
    memset(dist, 0x3f, sizeof dist);
    d[1] = 0;//以1号点作为起点

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

    while(q.size())
    {
        int 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;
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return false;
}
```

相关题目：AcWing 852. spfa判断负环

变式

原题连接：AcWing 3305. 作物杂交

这道题本质上基于 SPFA 的推导

设 $f(i,j)$ 表示所有在 i 步以内生成 j 的方法的集合，属性为时间的最小值

设最后一步中，**任意**两个作物 x,y 生成 j ，此时有：

j 的时间为从起点到 x 的时间与从起点到 y 的时间取较大值，再加上最后一步的时间($w[i]$ 表示 i 生成对应作物的时间)，即：

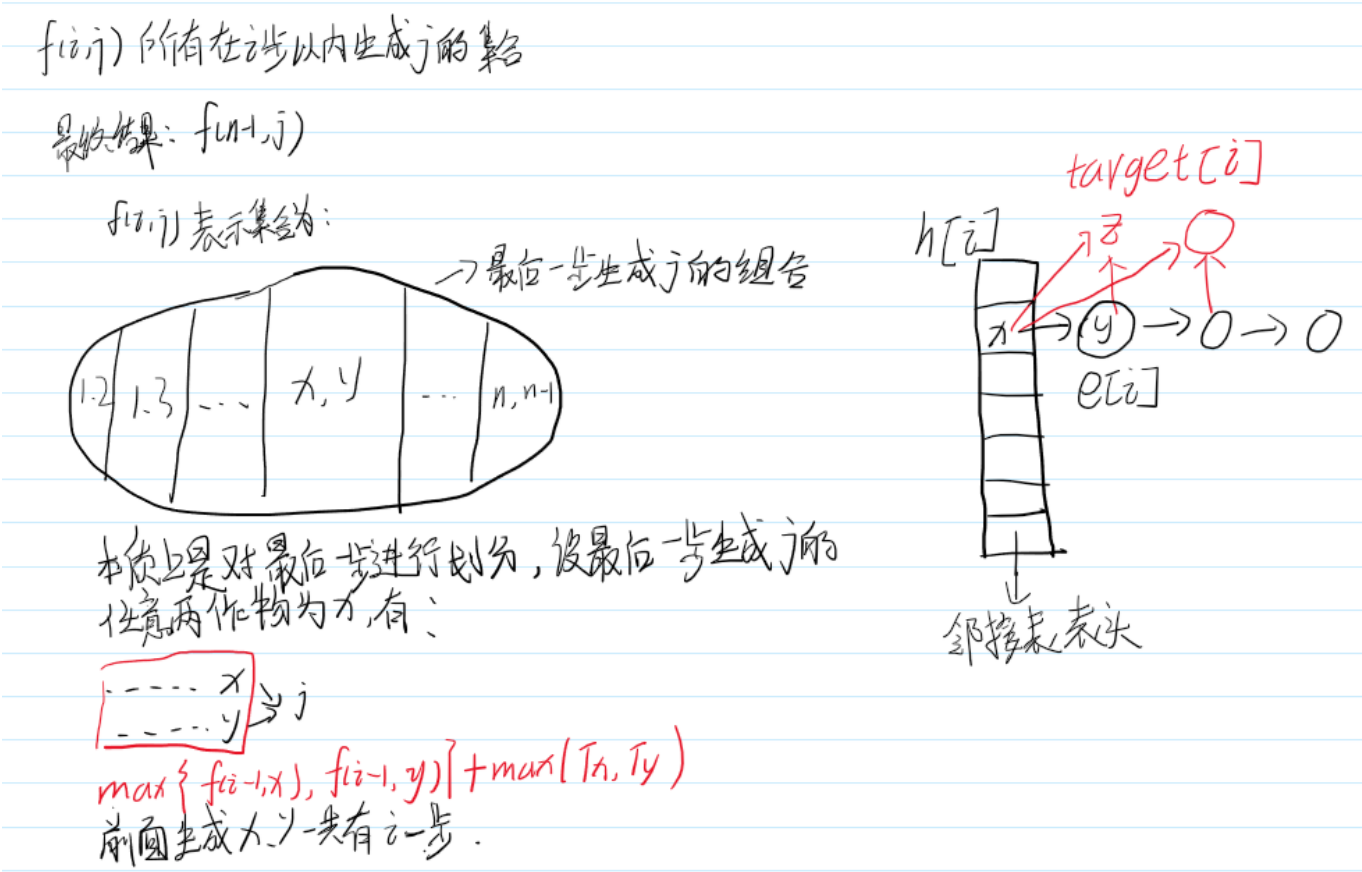
$$f(i,j) = \max(f(i-1,x), f(i-1,y)) + \max(w[x], w[y])$$

由于 i 的状态只依赖于 $i-1$ ，因此可以将第一维优化掉，设 $f(j)$ 表示生成 j 的方法的集合，属性为时间最小值，有

$$f(j) = \max(f(x), f(y)) + \max(w[x], w[y])$$

这几乎就是 SPFA 了，仿照 SPFA 的优化思路，只将当前更新过距离的点加入到队列中，这道题便也就结束了

有一个细节的地方需要注意的是，由于有两个点生成第三个点，因此在存图的时候，邻接表的表头表示 x ，邻接表引出的点表示 y ，还需要一个额外数组 `target` 表示由 x 与 y 生成的对应点，并且由于对称性，**边需要存两倍**



完整代码如下：

```
#include <iostream>
#include <cstring>
#include <queue>

using namespace std;

const int N = 2010, M = 2e5 + 10;

int h[N], e[M], ne[M], w[N], target[M], idx;
queue<int>q;
int dist[N];//从起点到i点的最小时间
bool st[N];//当前队列当中是否存在该元素

void add(int a, int b, int c)
{
    e[idx] = b, target[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

void spfa()
{
    while(q.size())
    {
        int x = q.front();
        q.pop();
```

```
        st[x] = false;

        for(int i = h[x]; i != -1; i = ne[i])
        {
            int y = e[i], z = target[i];
            if(dist[z] > max(dist[x], dist[y]) + max(w[x], w[y]))
            {
                dist[z] = max(dist[x], dist[y]) + max(w[x], w[y]);
                if(!st[z])
                {
                    q.push(z);
                    st[z] = true;
                }
            }
        }
    }
}

int n, m, k, T;

int main()
{
    memset(h, -1, sizeof h);
    memset(dist, 0x3f, sizeof dist);
    scanf("%d%d%d%d", &n, &m, &k, &T);
    for(int i = 1;i <= n; i ++) cin >> w[i];
    while(m--)
    {
        int x;
        cin >> x;
        q.push(x);
        dist[x] = 0;//初始化距离，这些都是起点
        st[x] = true;
    }
    while(k--)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
        add(b, a, c);
    }
    spfa();
    cout << dist[T] << endl;
    return 0;
}
```

Floyd

基础

用于求解多源汇最短路问题，对**边权没有要求**，负权也可以使用

采用**邻接矩阵存储**，设 $d[i][j]$ 表示图中从 i 到 j 的距离，当做完一遍 Floyd 之后，$d[i][j]$ 表示图中从 i 到 j 的**最短距离**

算法步骤如下

```
void Floyd()
{
    for(int k = 1; k <= n; k ++)//n为点数
        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]);
}
```

Floyd 本质上基于动态规划，设 $f(k,i,j)$ 表示从 i 出发，**只经过**点 $1,2,3,\cdots,k$ 这些点到底 j 的最短距离

设最后一个点经过 k ，那么有：

$$f(k,i,j)=f(k-1,i,k)+f(k-1,k,j)$$

实际含义为：先从 i 走 $1,2,3,\ldots,k-1$ 这些点到达 k ，再从 k 出发，走 $1,2,3,\ldots,k-1$ 这些点到达 j

由于第 k 层的状态只取决于 $k-1$ 层，因此可以将第一维优化掉，得到三重循环，这便是 Floyd 算法的思路

相关题目：AcWing 854. Floyd求最短路