普通图与树

单链表

其实很不方便，倒是多用在图中；正常不如直接双链表

```
const int N = 1e5 + 2;
int head, e[N], ne[N], idx;

void init() { head = -1, idx = 0; }

void insert(int k, int elm) {
    if (k == -1) { e[idx] = elm, ne[idx] = head, head = idx++; return; }
    e[idx] = elm, ne[idx] = ne[k], ne[k] = idx++;
}

void remove(int k) {
    if (k == -1) { head = ne[head]; return; }
    ne[k] = ne[ne[k]];
}

int tar = head;
while (tar != -1) {
    cout << e[tar] << " ";
    tar = ne[tar];
}
cout << endl;
```

双链表

先用这个了

```
const int N = 1e5 + 2;
int e[N], l[N], r[N], idx;
// 0 <-> 1
void init() { r[0] = 1, l[1] = 0, idx = 2; }

void insert_r(int k, int elm) {
    e[idx] = elm, l[idx] = k, r[idx] = r[k];
    l[r[k]] = idx, r[k] = idx++;
}

void insert_l(int k, int elm) {
    insert_r(l[k], elm);
}
```

```cpp
void remove(int k) {
    r[l[k]] = r[k], l[r[k]] = l[k];
}

// 打印
int pos = r[0];
while (pos != 1) {
    cout << e[pos] << " ";
    pos = r[pos];
}
cout << endl;
```

图

邻接表形式

```cpp
const int N = 1e5 + 2, M = N << 1; // N: vertex cnt, M: edge cnt

int n;
int h[N], e[M], ne[M], idx;
int w[N]; // 如果还带权的话

void init() {
    memset(h, -1, sizeof h);
}

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

// 遍历点u邻接的所有其他点j or 邻接的边 or 树的所有孩子
for (int i = h[u]; i != -1; i = ne[i]) {
    int j = e[i];  /* ~i */
    ...
}
```

邻接矩阵形式

```cpp
const int N = 1e5 + 2; // N: vertex cnt

int n;
int g[N][N];

void init() {
    memset(g, 0x3f, sizeof g); // INF means not connected
```

```
}

void add(int a, int b, int w) {
    g[a][b] = w;  // 注意：自环和重边的特殊处理
}
```

树

Acwing 285 输入只给边信息的树，用图的方式存储，不过多一个root

```
int h[N], e[N], ne[N], idx;
bool has_father[N];

int root = 1;
while (has_father[root]) root++;

// 树中dfs
void dfs(int u) {
    // 遍历所有子树
    for (int i = h[u]; ~i; i = ne[i]) {
        int son = e[i];
        dfs(son);

        // do something
    }
}
```

最小生成树 MST

总括

- Prim
    - 朴素 O(n^2)
    - 堆优化 O(m logn) 不常用
- Kruskal ： O(mlogm) = O(mlogn) 关键路径在排序
  常用：稠密图 - Prim朴素； 稀疏图 - Kruskal （因为简单些）

PRIM：

```
const int N = 500+2, INF = 0x3f3f3f3f;

int n, m;
int g[N][N];
int d[N]; // distance to set
bool flag[N]; // is in set

int prim() {
```

```
    memset(d, 0x3f, sizeof d);

    int res = 0;
    for (int i = 0; i < n; ++i) {
        // find min weight edge that is out of set
        int t = -1;
        for (int j = 1; j <= n; ++j) {
            if (!flag[j] && (t == -1 || d[t] > d[j])) {
                t = j;
            }
        }

        // add it to set
        if (i/* remove init state i = 0 */ && d[t] == INF) return INF;
        if (i) res += d[t];
        flag[t] = 1;

        // update distance to set
        for (int j = 1; j <= n; ++j)
            d[j] = min(d[j], g[t][j]);
    }

    return res;
}

int main() {
    memset(g, 0x3f, sizeof g);
    cin >> n >> m;
    int a, b, c;

    while (m --) {
        cin >> a >> b >> c;
        if(a != b) g[a][b] = g[b][a] = min(g[a][b], c);
    }

    int t = prim();
    if (t == INF) puts("impossible");  // there's INF edge in the graph <=>
not connectable
    else printf("%d\n", t);
}
```

Kruskal

```
const int N = 1e5 + 2, M = 2e5 + 2;
const int INF = 0x3f3f3f3f;

struct Edge {
    int a, b, w;
    bool operator< (const Edge& other) const {
```

```cpp
        return w < other.w;
    }
};
Edge e[M];
int n, m;
int p[N];

int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal() {
    int res = 0, cnt = 0; // cnt: mst edges cnt
    for (int i = 0; i < m; ++i) {
        Edge& edge = e[i];
        int pa = find(edge.a), pb = find(edge.b);
        if (pa != pb) {
            res += edge.w;
            cnt++;
            p[pb] = pa;
        }
    }
    return (cnt < n - 1) ? INF : res;
}

int main() {
    scanf("%d%d", &n, &m);
    int x, y, z;
    for (int i = 0; i < m; ++i) {
        scanf("%d%d%d", &x, &y, &z);
        e[i] = {x, y, z};
    }
    sort(e, e+m);
    for (int i = 1; i <= n; ++i) p[i] = i;

    int t = kruskal();
    if (t == INF) puts("impossible");
    else printf("%d\n", t);

    return 0;
}
```

最短路径 SP

总体分类

- 单源最短路
    - 全部正权边

        - 朴素Dijkstra : O(n^2)
        - 堆优化Dijkstra : O(m log n)
    - 存在负权边
        - Bellman-Ford : O(nm)
        - SPFA : 优化Bellman-Ford 平均O(m)，最坏O(nm) 不一定都能做，要求不能有负环
- 多源汇最短路
    - Floyd算法

【朴素dijk】：邻接矩阵

```
const int N = 500+2, M = 1e5 + 2;

int g[N][N]; // use matrix for graph
int n, m;

int d[N];     // shortest path
int flag[N]; // vertex set of those shortest path is determinated

void add(int x, int y, int z) {
    g[x][y] = min(g[x][y], z);
}

int dijkstra(int target) {
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    int t;
    for (int i = 0; i < n; ++i) {
        // find min distance
        t = -1;
        for (int j = 1; j <= n; ++j) {
            if (!flag[j] && (t == -1 || d[j] < d[t]))
                t = j;
        }

        // add to set
        if (t == target) break;
        flag[t] = 1;
        // printf("%d\n", t);

        // update distance
        for (int j = 1; j <= n; ++j) {
            d[j] = min(d[j], d[t] + g[t][j]);
            // printf("[update] d[%d] -> %d\n", j, d[j]);
        }
    }
    return d[target] != 0x3f3f3f3f ? d[target] : -1;
```

```
}

int main() {
    memset(g, 0x3f, sizeof g);
    cin >> n >> m;
    int a, b, d;
    for (int i = 0; i < m; ++i) {
        cin >> a >> b >> d;
        add(a, b, d);
    }
    cout << dijkstra(n) << endl;
}
```

【优先队列优化dijk】邻接表

```
#include <queue>
const int N = 1.5e5 + 2; // N = M
typedef pair<int, int> pii;

int h[N], e[N], ne[N], idx; // use matrix for graph
int w[N]; // edge weight: distance
int n, m;

int d[N];    // shortest path
int flag[N]; // vertex set of those shortest path is determinated

void add(int x, int y, int z) {
    e[idx] = y, w[idx] = z, ne[idx] = h[x], h[x] = idx++;
}

int dijkstra(int target) {
    memset(d, 0x3f, sizeof d);
    d[1] = 0;

    priority_queue<pii, vector<pii>, greater<pii>> heap; // min heap
    heap.push({0, 1});

    while (!heap.empty()) {
        // find min distance to next vertex
        auto t = heap.top(); heap.pop();
        int& cur = t.second, & dist = t.first;

        // add vertex to determinate set
        if (flag[target]) break;
        if (flag[cur]) continue;
        flag[cur] = 1;

        // update distance
        for (int i = h[cur]; i != -1; i = ne[i]) {
```

```
            int nxt = e[i];
            if (d[nxt] > dist + w[i]) {
                d[nxt] = dist + w[i];
                heap.push({d[nxt], nxt});
            }
        }
    }

    return d[target] != 0x3f3f3f3f ? d[target] : -1;
}

int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    int a, b, d;
    for (int i = 0; i < m; ++i) {
        cin >> a >> b >> d;
        add(a, b, d);
    }
    cout << dijkstra(n) << endl;
}
```

【Bellman-Ford】有边数限制的最短路

对于n点m边的图：循环n次，每次都对所有边 `a, b, w` 进行松弛操作 `d[b]=min(d[b], d[a] + w)`，这被称为三角不等式

有负权边：如果进一步有**负权回路**的图，则最短路不一定存在（负权环到不了的点就仍然存在）

- **外层迭代次数的含义**：循环k次后的 `d`，表示起始点经过不超过k条边的最短路
- 第n次更新时，如果还存在松弛操作，则说明有负环
    - 抽屉原理，说明存在一条最短路径有n条边，则对应n+1个点，而总共n个点，必有两个点相同，则说明**最短路径中成环**了，最短路径中有环则必然是负环了
      实现
- 虽说k次循环后，d表示不超过k条边的最短路，但这只是针对正确的最短路；超过k条边的非法最短路d值可能会串联着被更新，原因在于可能后更新的d直接使用本轮的更新结果，导致1轮更新多轮效果
    - 输入样例：可能迭代一轮就d[1] d[2]就都出来了，只不过d[1]是合法确定的最短路，d[2]则不是
    - => 所以每轮要记录更新前的副本，每次更新仅用上一轮的结果
- 这道题是Bellman-ford变种，并不是完全一样的

```
const int N = 500 + 2, M = 1e5 + 2;
const int INF = 0x3f3f3f3f;
struct Edge {
    int a, b, w;
};
```

```
int n, m, k;
Edge e[M];
int d[2][N];
int dist[N], backup[N];

int bellman_ford(int target) { // target = n
    memset(d, 0x3f, sizeof d);
    dist[1] = 0;

    for (int i = 0; i < k; ++i) {
        memcpy(backup, dist, sizeof(dist));
        for (int j = 0; j < m; ++j) {
            int a = e[j].a, b = e[j].b, w = e[j].w;
            dist[b] = min(dist[b], backup[a] + w);
        }
    }

    return dist[n] > INF / 2 ? -1 : dist[n];
}

int main() {
    cin >> n >> m >> k;
    int x, y, z;
    for (int i = 0; i < m; ++i) {
        cin >> x >> y >> z;
        e[i] = {x, y, z};
    }
    int t = bellman_ford(n);
    if (t > INF/2) puts("impossible");
    else cout << t << endl;
}
```

【SPFA】对Bellman-Ford的优化

- 对于其中的更新操作，d[b]被松弛更新当且仅当d[a]被松弛更新，所以按更新序遍历就可以一轮更新多轮（其实853已经能意识到，但853得避免这种情况）
- BFS遍历：queue中存可能可以更新的点，初始为起点1，每次更新一个点后，将这个点所有邻接点加入queue中
  SPFA一般也可过dijkstra的正权图，**可优先考虑**；且一般情况更快（此题交850，快200ms）；网格图容易卡SPFA

```
const int N = 1e5 + 2;
const int INF = 0x3f3f3f3f;

int n, m;
int h[N], e[N], w[N], ne[N], idx;
int d[N];
```

```cpp
int q[N<<1];
bool flag[N]; // check whether vertex is in the queue

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

int spfa(int target) {
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    int head = 0, tail = 0;
    q[0] = 1; flag[1] = 1;

    // assert no negative circle
    while (head <= tail) {
        int a = q[head++]; flag[a] = 0;
        for (int i = h[a]; i != -1; i = ne[i]) {
            int b = e[i];
            if (d[b] > d[a] + w[i]) {
                d[b] = d[a] + w[i];
                if (!flag[b]) { q[++tail] = b; flag[b] = 1; }
            }
        }
    }

    return d[target];
}

int main() {
    memset(h, -1, sizeof h);
    scanf("%d%d", &n, &m);
    int x, y, z;
    for (int i = 0; i < m; ++i) {
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
    }

    int t = spfa(n);
    if (t > INF/2) puts("impossible");
    else printf("%d\n", t);
}
```

【SPFA求负环】

`dist[x]` 表示1起点到x的最短路；`cnt[x]` 当前最短路的边数

- 则cnt[x]大于等于n时，就相当于bellman-ford循环n次没有停；所以同理可知**路径中存在环**，即负环

实现

- 队列的实现：这题用STL就过了，子实现queue M<<1/2/3/4的大小都不够(?)，会越界段错误：最终 `q[M * 1000]` 过了，相当于 O(m n) 最坏情况，自己的queue记得开到最坏情况
- 问的是全图是否存在负权环，并不是从1出发的最短路中是否有负环；所以一开始将所有点加入队列；否则可能起点1不可达负环，从而漏掉此负环判断

```
const int N = 2000+2, M = 10000+2;
const int INF = 0x3f3f3f3f;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[N], cnt[N];
int flag[N];

int q[M*N]; // should be enough: M * 1000 is enough, for worst cases

void add(int x, int y, int z) {
    e[idx] = y, w[idx] = z, ne[idx] = h[x], h[x] = idx++;
}

bool spfa(int target) {
    // no need to initialize: for only negative edge would trouble
    // memset(dist, 0x3f, sizeof dist);
    // dist[1] = 0;
    int head = 0, tail = -1;
    for (int i = 1; i <= n; ++i) {
        q[++tail] = i;
        flag[i] = 1;
    }

    while (head <= tail) {
        int a = q[head++]; flag[a] = 0;
        for (int i = h[a]; i != -1; i = ne[i]) {
            int b = e[i];
            if (dist[b] > dist[a] + w[i]) {
                dist[b] = dist[a] + w[i];
                cnt[b] = cnt[a] + 1;
                if (cnt[b] >= n) return true;
                if (!flag[b]) { q[++tail] = b, flag[b] = 1; }
            }
        }
    }
    return false;
}

int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    int x, y, z;
```

```
    for (int i = 0; i < m; ++i) {
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
    }
    puts(spfa(n) ? "Yes" : "No");
}
```

【Floyd求最短路】
实现

- 注意存在重边和自环，都是需要处理一下的：重边取最小，自环本来就是0
- 注意存在负边，只不过没有负环，所以输出还是要小处理一下 > INF/2
- 其实就是由 `dp[k][i][j] = dp[k-1][i][k] + dp[k-1][k][j]` 优化而来；`dp[k][i][j]` 表示从 i 到 j 只经过1,2,..,k 中转可得的最短路
    - 所以压缩后，k循环也应该在外侧；problem中第二个样例就是顺序错导致的

```
const int N = 200 + 2, INF = 0x3f3f3f3f;
int d[N][N]; // dist from i to j <- from dp[k][i][j]
int n, m;
void floyd() {
    for (int k = 1; k <= n; ++k) { // attention: k out
        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]);
            }
        }
    }
}

int main() {
    cin.tie(0); int k;
    cin >> n >> m >> k;

    memset(d, 0x3f, sizeof d);
    for (int i = 1; i <= n; ++i) d[i][i] = 0;
    int x, y, z;
    for (int i = 0; i < m; ++i) {
        cin >> x >> y >> z;
        d[x][y] = min(d[x][y], z);
    }
    floyd();

    while (k--) {
        cin >> x >> y;
        if (d[x][y] < INF/2) cout << d[x][y] << endl;
        else cout << "impossible" << endl;
```

```
    }
}
```

二分图

没看，也忘了

【染色法】 O(n+m)

```
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表示未染色，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】

```
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 ++ ;
}
```