Lesson2 (最短路算法)

> **最短路算法大纲**
>
> 源点：起点　汇点：终点
>
> **单源最短路**：从一个点到其它所有点的最短距离
>
> - **所有边权都是正数（正权图）**
>   - **朴素Dijkstra算法（贪心）**
>   - 时间复杂度O(n^2)　（n表示点数，m表示边数）
>   - 适合于**稠密图** (m~n^2)，用**邻接矩阵**存储
>   - **堆优化版的Dijkstra算法**
>   - 时间复杂度O(mlogn)
>   - 适用于**稀疏图**(m~n)，用**邻接表**进行存储
> - **存在负权边**
>   - **Bellman-Ford算法（离散数学）**
>   - 时间复杂度O(nm)
>   - 经过不超过k条边的最短路（只能用Bellman-Ford算法，不能使用SPFA）
>   - **SPFA算法**
>   - 对Bellman-Ford算法优化
>   - 一般时间复杂度O(m)，最坏为O(nm)
>
> **多源汇最短路**：源点汇点不确定，任意两点间的最短距离
>
> - **Floyd算法（DP）**
>   - 时间复杂度O(n^3)
>
> 考察重难点：如何**抽象问题并建图**

> **朴素Dijkstra算法（贪心）**
>
> 步骤
>
> ```
> 初始化距离 dist[1] = 0, dist[i] = +∞（实际中用大数代替）
>
> 集合S：当前已确定最短距离的点
> for i: 0 ~ n        O(n)
>     t <- 不在S中的距离最近的点（贪心）     O(n)
>     s <- t
>     用t更新其他点到起点的距离 dist[x] ? dist[t] + w
> ```
>
> 具体实现：　O(n^2)
>
> - 使用**邻接矩阵**进行存储
>
> ```cpp
> #include <iostream>
> #include <cstring>
> #include <algorithm>
>
> using namespace std;
>
> const int N = 510;
>
> int n, m;
> int g[N][N];    //朴素版Dijkstra算法使用临界矩阵
> int dist[N];    //dist[i]表示第i个结点到1号点的最短距离
> bool st[N];     //st[i]表示第i个结点是否已确定最短距离
>
> int dijkstra() {
>     //初始化dist数组
>     memset(dist, 0x3f, sizeof dist);
>     dist[1] = 0;
>
>     for (int i=0; i<n; i++) {
>         int t = -1;
>
>         for (int j=1; j<=n; j++)
> ```

```
            if (!st[j] && (t==-1 || dist[j]<dist[t]))
                t = j;

        st[t] = true;

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

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

int main(void) {
    scanf("%d%d", &n, &m);

    memset(g, 0x3f, sizeof g);

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        g[a][b] = min(g[a][b], c);
    }

    cout << dijkstra();

    return 0;
}
```

堆优化版Dijkstra（贪心）

稀疏图，使用**邻接表**存储

堆优化

- 手写堆（映射版）
- **优先队列**（不支持修改任意一个元素，因此每次修改则新插入元素，最多堆中有m个元素，存在冗余，时间复杂度O(mlogm) -> O(mlogn) ）

具体实现：O(mlogn)

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

using namespace std;

typedef pair<int, int> PII;

const int N = 200010, M = N;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[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 dijkstra() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

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

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

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

        //判断最短距离是否已确定
        if (st[ver]) continue;
        st[ver] = true;

```

```
        //更新所有ver的出边的距离
        for (int i=h[ver]; i!=-1; i=ne[i]) {
            int j = e[i];
            if (dist[j] > d+w[i]) {
                dist[j] = d+w[i];
                heap.push({dist[j], j});
            }
        }
    }

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

int main(void) {
    scanf("%d%d", &n, &m);

    memset(h, -1, sizeof h);

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }

    cout << dijkstra();

    return 0;
}

```

Bellman-Ford算法

步骤

```
存边方式
struct {
    int a, b, w;
} edge[M]

for n次
    (当有经过边数限制时，需备份上次dist数组，只用上次迭代结果来更新，防止出现串联现象)
    for 所有边 a-w->b
        dist[b] = min(dist[b], dist[a]+w);   (松弛操作)
```

特性

- 循环完n次后，对所有边满足 dist[b] <= dist[a] + w （三角不等式）
- 如果有**负权回路**，最短路则**不一定存在**，可能为-∞
- 迭代k次的意义：经过**不超过k条边**的**最短路距离**（只能用Bellman-Ford算法，不能用SPFA）
- 可以判断负环（一般用SPFA做）：依据抽屉原理，若第n次迭代时，仍有路径更新，则图中存在负权回路

具体实现：O(nm)

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

using namespace std;

const int N = 510, M = 100010;

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

int n, m, k;
int dist[N];
int last[N];    //dist前一次迭代结果备份，防止在有边数限制时出现串联

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

    for (int i=0; i<k; i++) {
        //备份上一次迭代结果
        memcpy(last, dist, sizeof dist);
```

```

        for (int j=0; j<m; j++) {
            auto e = edges[j];
            dist[e.b] = min(dist[e.b], last[e.a] + e.w);
        }
    }

    //返回-1可能存在二义性。
    if (dist[n] > 0x3f3f3f3f/2) return 0x3f3f3f3f;
    return dist[n];
}

int main(void) {
    scanf("%d%d%d", &n, &m, &k);

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edges[i] = {a, b, c};
    }

    int t = bellman_ford();

    if (t == 0x3f3f3f3f) puts("impossible");
    else printf("%d\n", t);

    return 0;
}
```

SPFA

特性

- 用**宽搜**对Bellman-Ford算法进行优化
- 只要没有负环，最短路问题都能使用SPFA
- 使用**邻接表**进行存储

步骤

```
队列存储所有变小的结点（存储待更新的点集）
queue <- 起点
while queue不空
    t <- 取队头
    更新t的所有出边 t-w->b
        更新成功则 queue <- b;
```

具体实现：**一般情况下O(m)**，最坏情况下O(nm)

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

using namespace std;

const int N = 100010, M = N;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[N];
int q[M], hh, tt = -1;
bool st[N];     //st[i]表示第i个结点是否在队列中，即是否为待更新的点

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

int spfa() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    //起点入队
    q[++tt] = 1;
    st[1] = true;

    while (hh <= tt) {
        int t = q[hh++];
        st[t] = false;  //已出队
```

```

        //更新t的所有出边
        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]) {
                    q[++tt] = j;
                    st[j] = true;
                }
            }
        }
    }

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

int main(void) {
    scanf("%d%d", &n, &m);

    memset(h, -1, sizeof h);

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }

    int t = spfa();

    if (t == 0x3f3f3f3f) puts("impossible");
    else printf("%d", t);

    return 0;
}
```

SPFA判负环（抽屉原理）

- 思路

  维护两个数组

  - dist[x]　当前从1号点到x号点的最短距离
  - cnt[x]　当前1号点到x号点的最短路的边数

```
更新操作
dist[x] = dist[t]+w[i];
cnt[x] = cnt[t]+1;

若cnt[x] >= n，则由抽屉原理可知图中存在负环
```

- 具体实现

  建议不用手写队列而用STL。手写队列容量不能确定，可能SF

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

using namespace std;

const int N = 2010, M = 10010;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[N], cnt[N];     //cnt[x]当前1号点到x最短路径的边数
queue<int> q;
bool st[N];

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

bool spfa() {
    //判断负环可以不用初始化，初始时将所有点加入队列
    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;
}

int main(void) {
    scanf("%d%d", &n, &m);

    memset(h, -1, sizeof h);

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }

    if (spfa()) puts("Yes");
    else puts("No");


    return 0;
}
```

Floyd算法（多源汇最短路）

特性

- 使用**邻接矩阵**进行存储
- 不能处理负环

思想

- 基于动态规划

  d[k, i, j] 表示从**i**只经过**1~k号中间点**时到达**j**的最短距离

  状态转移方程：d[k, i, j] = d[k-1, i, k] + d[k-1, k, j]

  去掉第一维：d[i, j] = d[i, k] + d[k, j];

步骤

```
d[i, j] 邻接矩阵
// O(n^3)
for (k=1; k<=n; k++)     //枚举阶段
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            d[i, j] = min(d[i, j], d[i, k]+d[k, j])
```

具体实现：O(n^3)

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

using namespace std;
```

```
const int N = 210, INF = 0x3f3f3f3f;

int n, m, Q;
int d[N][N];

void floyd() {
    for (int k=1; k<=n; k++)    //枚举阶段
        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(void) {
    scanf("%d%d%d", &n, &m, &Q);

    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;

    for (int i=0; i<m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        d[a][b] = min(d[a][b], c);
    }

    floyd();

    while (Q--) {
        int x, y;
        scanf("%d%d", &x, &y);

        if (d[x][y] > INF/2) puts("impossible");
        else printf("%d\n", d[x][y]);
    }

    return 0;
}
```