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

```java
static int N = 510, INF = 0x3f3f3f3f;

static int n, m;
static int[][] g = new int[N][N];
static int[] dist = new int[N];
static boolean[] st = new boolean[N];

static int dijkstra() {
    Arrays.fill(dist, INF);

    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;

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

    return dist[n] == INF ? -1 : dist[n];
}


public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) Arrays.fill(g[i], INF);

    while (m-- > 0) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ins.nextToken(); int c = (int)ins.nval;
        g[a][b] = Math.min(g[a][b], c);
    }

    out.println(dijkstra());

    out.flush();
}
```

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)

```
static int N = 150010, M = N, INF = 0x3f3f3f3f;

static int n, m;
static int idx;
static int[] h = new int[N], e = new int[M], w = new int[M], ne = new
int[M];
static int[] dist = new int[N];
static boolean[] st = new boolean[N];

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

static int dijkstra() {
    Arrays.fill(dist, INF);
    dist[1] = 0;

    PriorityQueue<PII> heap = new PriorityQueue<PII>((o1, o2) ->
o1.first-o2.first);
    heap.add(new PII(0, 1));

    while (!heap.isEmpty()) {
        PII t = heap.poll();

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

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

        for (int i=h[ver]; i!=-1; i=ne[i]) {    //更新ver所有出边
            int j = e[i];
            if (!st[j] && dist[j] > d+w[i]) {
                dist[j] = d+w[i];
                heap.add(new PII(dist[j], j));
            }
        }
    }

    return dist[n] == INF ? -1 : dist[n];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    Arrays.fill(h, -1);

    while (m-- > 0) {
        ins.nextToken(); int a = (int)ins.nval;
```

```
            ins.nextToken(); int b = (int)ins.nval;
            ins.nextToken(); int c = (int)ins.nval;
            add(a, b, c);
        }

        out.println(dijkstra());

        out.flush();
    }

    static class PII {
        int first, second;

        PII (int f, int s) {
            first = f; second = s;
        }
    }
```

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)

```
static int N = 510, M = 10010, INF = 0x3f3f3f3f;

static int n, m, k;
static Edge[] edge = new Edge[M];
static int[] dist = new int[N], last = new int[N];

static int bellmanFord() {
    Arrays.fill(dist, INF);
```

```java

    dist[1] = 0;

    for (int i=0; i<k; i++) {
        last = dist.clone();    //备份上一次迭代结果，防止串联

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

    if (dist[n] > INF/2) return INF;
    else return dist[n];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;
    ins.nextToken(); k = (int)ins.nval;

    for (int i=0; i<m; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ins.nextToken(); int c = (int)ins.nval;
        edge[i] = new Edge(a, b, c);
    }

    int t = bellmanFord();
    out.println((t == INF? "impossible" : t));

    out.flush();
}

static class Edge {
    int a, b, w;

    Edge (int aa, int bb, int ww) {
        a = aa; b = bb; w = ww;
    }
}
```

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)

```java
static int N = 100010, M = N, INF = 0x3f3f3f3f;

static int n, m;
static int idx;
static int[] h = new int[N], e = new int[M], w = new int[M], ne = new int[M];
static int[] dist = new int[N];
static int hh, tt = -1;
static int[] q = new int[M];    //存储待更新的点
static boolean[] st = new boolean[N];   //存储当前点是否在队列中

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

static int spfa() {
    Arrays.fill(dist, INF);
    dist[1] = 0;

    q[++tt] = 1;
    st[1] = true;

    while (hh <= tt) {
        int t = q[hh++];
        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]) {   //注意更新的位置
                    q[++tt] = j;
                    st[j] = true;
                }
            }
        }
    }

    return dist[n];
}


public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
```

```java
    ins.nextToken(); m = (int)ins.nval;

    Arrays.fill(h, -1);

    while (m-- > 0) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ins.nextToken(); int c = (int)ins.nval;
        add(a, b, c);
    }

    int t = spfa();
    out.println((t == INF ? "impossible" : t));

    out.flush();
}
```

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

```java
static int N = 2010, M = 10010;

static int n, m;
static int idx;
static int[] h = new int[N], e = new int[M], w = new int[M], ne = new int[M];
static Queue<Integer> q = new LinkedList<Integer>();
static boolean[] st = new boolean[N];
static int[] dist = new int[N], cnt = new int[N];    //cnt[i]表示初始某个点到结点i的路径边数。

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

static boolean spfa() {
    //存在负权边，所以dist可以不用初始化为INF
    for (int i=1; i<=n; i++) {
```

```java
        q.offer(i); st[i] = true;    //初始将所有点加入q中
    }

    while (!q.isEmpty()) {
        int t = q.poll();
        st[t] = false;  //出队

        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;  //到达j的路径边数=到达t的路径边数+1

                if (cnt[j] >= n) return true;

                if (!st[j]) {
                    q.offer(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    Arrays.fill(h, -1);

    while (m-- > 0) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ins.nextToken(); int c = (int)ins.nval;
        add(a, b, c);
    }

    out.println((spfa() ? "Yes" : "No"));

    out.flush();
}
```

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] = min(d[k-1, i, j], d[k-1, i, k] + d[k-1, k, j])

去掉第一维：d[i, j] = min(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)

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

static int n, m, q;
static int[][] dist = new int[N][N];

static void floyd() {
    for (int k=1; k<=n; k++)
        for (int i=1; i<=n; i++)
            for (int j=1; j<=n; j++)
                dist[i][j] = Math.min(dist[i][j], dist[i][k]+dist[k][j]);
}


public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;
    ins.nextToken(); q = (int)ins.nval;

    for (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
            dist[i][j] = (i == j ? 0: INF);

    while (m-- > 0) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ins.nextToken(); int c = (int)ins.nval;
        dist[a][b] = Math.min(dist[a][b], c);
    }

    floyd();

    while (q-- > 0) {
        ins.nextToken(); int x = (int)ins.nval;
        ins.nextToken(); int y = (int)ins.nval;
        out.println((dist[x][y] > INF/2 ? "impossible": dist[x][y]));
    }

    out.flush();
```

```
}
```