

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 ($m \sim n^2$)，用邻接矩阵存储
 - 堆优化版的Dijkstra算法
 - 时间复杂度 $O(m \log n)$
 - 适用于稀疏图 ($m \sim n$)，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

```
1  初始化距离  dist[1] = 0, dist[i] = +∞（实际中用大数代替）
2
3  集合S: 当前已确定最短距离的点
4  for i: 0 ~ n      O(n)
5      t <- 不在S中的距离最近的点（贪心）      O(n)
6      s <- t
7      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

```
1  static int N = 510, INF = 0x3f3f3f3f;
2
3  static int n, m;
4  static int[] [] g = new int[N][N];
5  static int[] dist = new int[N];
6  static boolean[] st = new boolean[N];
7
8  static int dijkstra() {
9      Arrays.fill(dist, INF);
10
11     dist[1] = 0;
12
13     for (int i=0; i<n; i++) {
14         int t = -1;
15
16         for (int j=1; j<=n; j++)
17             if (!st[j] && (t == -1 || dist[j] < dist[t]))
18                 t = j;
19
20         st[t] = true;
21
22         for (int j=1; j<=n; j++)
```

```

23         dist[j] = Math.min(dist[j], dist[t]+g[t][j]);
24     }
25
26     return dist[n] == INF ? -1 : dist[n];
27 }
28
29
30 public static void main(String[] args) throws Exception {
31     ins.nextToken(); n = (int)ins.nval;
32     ins.nextToken(); m = (int)ins.nval;
33
34     for (int i=1; i<=n; i++) Arrays.fill(g[i], INF);
35
36     while (m-- > 0) {
37         ins.nextToken(); int a = (int)ins.nval;
38         ins.nextToken(); int b = (int)ins.nval;
39         ins.nextToken(); int c = (int)ins.nval;
40         g[a][b] = Math.min(g[a][b], c);
41     }
42
43     out.println(dijkstra());
44
45     out.flush();
46 }

```

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 $O(m\log m)$ -> $O(m\log n)$)

具体实现: $O(m\log n)$

```

1  static int N = 150010, M = N, INF = 0x3f3f3f3f;
2
3  static int n, m;
4  static int idx;
5  static int[] h = new int[N], e = new int[M], w = new int[M], ne = new int[M];
6  static int[] dist = new int[N];
7  static boolean[] st = new boolean[N];
8
9  static void add(int a, int b, int c) {
10     e[idx] = b; w[idx] = c; ne[idx] = h[a]; h[a] = idx++;
11 }
12
13 static int dijkstra() {
14     Arrays.fill(dist, INF);
15     dist[1] = 0;
16
17     PriorityQueue<PII> heap = new PriorityQueue<PII>((o1, o2) -> o1.first-o2.first);
18     heap.add(new PII(0, 1));
19
20     while (!heap.isEmpty()) {
21         PII t = heap.poll();
22
23         int ver = t.second, d = t.first;
24
25         if (st[ver]) continue; //判断最短距离是否已经确定
26         st[ver] = true;
27
28         for (int i=h[ver]; i!=-1; i=ne[i]) { //更新ver所有出边
29             int j = e[i];
30             if (!st[j] && dist[j] > d+w[i]) {
31                 dist[j] = d+w[i];
32                 heap.add(new PII(dist[j], j));
33             }
34         }
35     }
36
37     return dist[n] == INF ? -1 : dist[n];
38 }
39
40 public static void main(String[] args) throws Exception {
41     ins.nextToken(); n = (int)ins.nval;

```

```
42     ins.nextToken(); m = (int)ins.nval;
43
44     Arrays.fill(h, -1);
45
46     while (m-- > 0) {
47         ins.nextToken(); int a = (int)ins.nval;
48         ins.nextToken(); int b = (int)ins.nval;
49         ins.nextToken(); int c = (int)ins.nval;
50         add(a, b, c);
51     }
52
53     out.println(dijkstra());
54
55     out.flush();
56 }
57
58 static class PII {
59     int first, second;
60
61     PII (int f, int s) {
62         first = f; second = s;
63     }
64 }
```

Bellman-Ford算法

步骤

```
1  存边方式
2  struct {
3      int a, b, w;
4  } edge[M]
5
6  for n次
7      (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)
8      for 所有边 a-w->b
9          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 $\text{dist}[b] \leq \text{dist}[a] + w$ (三角不等式)
- 如果有**负权回路**, 最短路则**不一定存在**, 可能为 $-\infty$
- 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: $O(nm)$

```
1  static int N = 510, M = 10010, INF = 0x3f3f3f3f;
2
3  static int n, m, k;
4  static Edge[] edge = new Edge[M];
5  static int[] dist = new int[N], last = new int[N];
6
7  static int bellmanFord() {
8      Arrays.fill(dist, INF);
9
10     dist[1] = 0;
11
12     for (int i=0; i<k; i++) {
13         last = dist.clone();    //备份上一次迭代结果, 防止串联
14
15         for (int j=0; j<m; j++) {
16             Edge e = edge[j];
17             dist[e.b] = Math.min(dist[e.b], last[e.a]+e.w);
18         }
19     }
20
21     if (dist[n] > INF/2) return INF;
22     else return dist[n];
23 }
24
25 public static void main(String[] args) throws Exception {
26     ins.nextToken(); n = (int)ins.nval;
27     ins.nextToken(); m = (int)ins.nval;
28     ins.nextToken(); k = (int)ins.nval;
29
30     for (int i=0; i<m; i++) {
```

```
31         ins.nextToken(); int a = (int)ins.nval;
32         ins.nextToken(); int b = (int)ins.nval;
33         ins.nextToken(); int c = (int)ins.nval;
34         edge[i] = new Edge(a, b, c);
35     }
36
37     int t = bellmanFord();
38     out.println((t == INF? "impossible" : t));
39
40     out.flush();
41 }
42
43 static class Edge {
44     int a, b, w;
45
46     Edge (int aa, int bb, int ww) {
47         a = aa; b = bb; w = ww;
48     }
49 }
```

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

```
1  队列存储所有变小的结点（存储待更新的点集）
2  queue <- 起点
3  while queue不空
4      t <- 取队头
5      更新t的所有出边 t-w->b
6      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)

```
1  static int N = 100010, M = N, INF = 0x3f3f3f3f;
2
3  static int n, m;
4  static int idx;
5  static int[] h = new int[N], e = new int[M], w = new int[M], ne = new int[M];
6  static int[] dist = new int[N];
7  static int hh, tt = -1;
8  static int[] q = new int[M];    //存储待更新的点
9  static boolean[] st = new boolean[N];    //存储当前点是否在队列中
10
11 static void add(int a, int b, int c) {
12     e[idx] = b; w[idx] = c; ne[idx] = h[a]; h[a] = idx++;
13 }
14
15 static int spfa() {
16     Arrays.fill(dist, INF);
17     dist[1] = 0;
18
19     q[++tt] = 1;
20     st[1] = true;
21
22     while (hh <= tt) {
23         int t = q[hh++];
24         st[t] = false;
25
26         for (int i=h[t]; i!=-1; i=ne[i]) {
27             int j = e[i];
28             if (dist[j] > dist[t]+w[i]) {
29                 dist[j] = dist[t]+w[i];
30
31                 if (!st[j]) {    //注意更新的位置
32                     q[++tt] = j;
33                     st[j] = true;
34                 }
35             }
36         }
37     }
38 }
```

```
39     return dist[n];
40 }
41
42
43 public static void main(String[] args) throws Exception {
44     ins.nextToken(); n = (int)ins.nval;
45     ins.nextToken(); m = (int)ins.nval;
46
47     Arrays.fill(h, -1);
48
49     while (m-- > 0) {
50         ins.nextToken(); int a = (int)ins.nval;
51         ins.nextToken(); int b = (int)ins.nval;
52         ins.nextToken(); int c = (int)ins.nval;
53         add(a, b, c);
54     }
55
56     int t = spfa();
57     out.println((t == INF ? "impossible" : t));
58
59     out.flush();
60 }
```

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

```
1 更新操作
2 dist[x] = dist[t]+w[i];
3 cnt[x] = cnt[t]+1;
4
5 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

```
1 static int N = 2010, M = 10010;
2
3 static int n, m;
4 static int idx;
5 static int[] h = new int[N], e = new int[M], w = new int[M], ne = new int[M];
6 static Queue<Integer> q = new LinkedList<Integer>();
7 static boolean[] st = new boolean[N];
8 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。
9
10 static void add(int a, int b, int c) {
11     e[idx] = b; w[idx] = c; ne[idx] = h[a]; h[a] = idx++;
12 }
13
14 static boolean spfa() {
15     //存在负权边，所以dist可以不用初始化为INF
16     for (int i=1; i<=n; i++) {
17         q.offer(i); st[i] = true;    //初始将所有点加入q中
18     }
19
20     while (!q.isEmpty()) {
21         int t = q.poll();
22         st[t] = false;    //出队
23
24         for (int i=h[t]; i!=-1; i=ne[i]) {
25             int j = e[i];
26
27             if (dist[j] > dist[t]+w[i]) {
28                 dist[j] = dist[t]+w[i];
29                 cnt[j] = cnt[t]+1;    //到达j的路径边数=到达t的路径边数+1
30
31                 if (cnt[j] >= n) return true;
32
33                 if (!st[j]) {
34                     q.offer(j);
35                     st[j] = true;
36                 }
37             }
38         }
39     }
40 }
```

```

38     }
39 }
40
41     return false;
42 }
43
44 public static void main(String[] args) throws Exception {
45     ins.nextToken(); n = (int)ins.nval;
46     ins.nextToken(); m = (int)ins.nval;
47
48     Arrays.fill(h, -1);
49
50     while (m-- > 0) {
51         ins.nextToken(); int a = (int)ins.nval;
52         ins.nextToken(); int b = (int)ins.nval;
53         ins.nextToken(); int c = (int)ins.nval;
54         add(a, b, c);
55     }
56
57     out.println((spfa() ? "Yes" : "No"));
58
59     out.flush();
60 }

```

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

```

1  d[i, j] 邻接矩阵
2  // o(n^3)
3  for (k=1; k<=n; k++)    //枚举阶段
4      for (i=1; i<=n; i++)
5          for (j=1; j<=n; j++)
6              d[i, j] = min(d[i, j], d[i, k]+d[k, j])

```

具体实现: $O(n^3)$

```

1  static int N = 210, INF = 0x3f3f3f3f;
2
3  static int n, m, q;
4  static int[][] dist = new int[N][N];
5
6  static void floyd() {
7      for (int k=1; k<=n; k++)
8          for (int i=1; i<=n; i++)
9              for (int j=1; j<=n; j++)
10                 dist[i][j] = Math.min(dist[i][j], dist[i][k]+dist[k][j]);
11 }
12
13
14 public static void main(String[] args) throws Exception {
15     ins.nextToken(); n = (int)ins.nval;
16     ins.nextToken(); m = (int)ins.nval;
17     ins.nextToken(); q = (int)ins.nval;
18
19     for (int i=1; i<=n; i++)
20         for (int j=1; j<=n; j++)
21             dist[i][j] = (i == j ? 0: INF);
22
23     while (m-- > 0) {
24         ins.nextToken(); int a = (int)ins.nval;
25         ins.nextToken(); int b = (int)ins.nval;
26         ins.nextToken(); int c = (int)ins.nval;
27         dist[a][b] = Math.min(dist[a][b], c);

```

```
28     }
29
30     floyd();
31
32     while (q-- > 0) {
33         ins.nextToken(); int x = (int)ins.nval;
34         ins.nextToken(); int y = (int)ins.nval;
35         out.println((dist[x][y] > INF/2 ? "impossible": dist[x][y]));
36     }
37
38     out.flush();
39 }
```