

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  #include <iostream>
2  #include <cstring>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 510;
8
9  int n, m;
10 int g[N][N];    //朴素版Dijkstra算法使用临界矩阵
11 int dist[N];    //dist[i]表示第i个结点到1号点的最短距离
12 bool st[N];     //st[i]表示第i个结点是否已确定最短距离
13
14 int dijkstra() {
15     //初始化dist数组
16     memset(dist, 0x3f, sizeof dist);
17     dist[1] = 0;
18
19     for (int i=0; i<n; i++) {
20         int t = -1;
21
22         for (int j=1; j<=n; j++)
23             if (!st[j] && (t==-1 || dist[j]<dist[t]))
24                 t = j;
25
26         st[t] = true;
27
28         // 用t更新其他点到起点的距离
29         for (int j=1; j<=n; j++)
30             dist[j] = min(dist[j], dist[t]+g[t][j]);
31     }
32
33     if (dist[n] == 0x3f3f3f3f) return -1;
34     return dist[n];
35 }
36
37 int main(void) {
38     scanf("%d%d", &n, &m);
39
40     memset(g, 0x3f, sizeof g);
41
42     for (int i=0; i<m; i++) {
43         int a, b, c;
44         scanf("%d%d%d", &a, &b, &c);
45         g[a][b] = min(g[a][b], c);
46     }
47
48     cout << dijkstra();
49
50     return 0;
51 }
```

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 $O(m\log m) \rightarrow O(m\log n)$)

具体实现: $O(m\log n)$

```
1  #include <iostream>
2  #include <cstring>
3  #include <algorithm>
4  #include <queue>
5
6  using namespace std;
7
8  typedef pair<int, int> PII;
9
10 const int N = 200010, M = N;
11
12 int n, m;
13 int h[N], e[M], w[M], ne[M], idx;
14 int dist[N];
15 bool st[N];
16
17 void add(int a, int b, int c){
18     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
19 }
20
21 int dijkstra() {
22     memset(dist, 0x3f, sizeof dist);
23     dist[1] = 0;
24
25     priority_queue<PII, vector<PII>, greater<PII>> heap;
26     heap.push({0, 1});
27
28     while (heap.size()) {
29         auto t = heap.top(); heap.pop();
30
31         int ver = t.second, d = t.first;
32
33         //判断最短距离是否已确定
34         if (st[ver]) continue;
35         st[ver] = true;
36
37         //更新所有ver的出边的距离
38         for (int i=h[ver]; i!=-1; i=ne[i]) {
39             int j = e[i];
40             if (dist[j] > d+w[i]) {
41                 dist[j] = d+w[i];
42                 heap.push({dist[j], j});
43             }
44         }
45     }
```

```

44     }
45 }
46
47     if (dist[n] == 0x3f3f3f3f) return -1;
48     return dist[n];
49 }
50
51 int main(void) {
52     scanf("%d%d", &n, &m);
53
54     memset(h, -1, sizeof h);
55
56     for (int i=0; i<m; i++) {
57         int a, b, c;
58         scanf("%d%d%d", &a, &b, &c);
59         add(a, b, c);
60     }
61
62     cout << dijkstra();
63
64     return 0;
65 }
66

```

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  #include <iostream>
2  #include <cstring>

```

```

3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 510, M = 100010;
8
9  struct Edge {
10     int a, b, w;
11 } edges[M];
12
13 int n, m, k;
14 int dist[N];
15 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
16
17 int bellman_ford() {
18     memset(dist, 0x3f, sizeof dist);
19     dist[1] = 0;
20
21     for (int i=0; i<k; i++) {
22         //备份上一次迭代结果
23         memcpy(last, dist, sizeof dist);
24
25         for (int j=0; j<m; j++) {
26             auto e = edges[j];
27             dist[e.b] = min(dist[e.b], last[e.a] + e.w);
28         }
29     }
30
31     //返回-1可能存在二义性。
32     if (dist[n] > 0x3f3f3f3f/2) return 0x3f3f3f3f;
33     return dist[n];
34 }
35
36 int main(void) {
37     scanf("%d%d%d", &n, &m, &k);
38
39     for (int i=0; i<m; i++) {
40         int a, b, c;
41         scanf("%d%d%d", &a, &b, &c);
42         edges[i] = {a, b, c};
43     }
44
45     int t = bellman_ford();
46
47     if (t == 0x3f3f3f3f) puts("impossible");
48     else printf("%d\n", t);
49
50     return 0;
51 }

```

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  #include <iostream>
2  #include <cstring>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 100010, M = N;
8
9  int n, m;
10 int h[N], e[M], w[M], ne[M], idx;
11 int dist[N];
12 int q[M], hh, tt = -1;
13 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
14
15 void add(int a, int b, int c) {
16     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
17 }
18
19 int spfa() {
20     memset(dist, 0x3f, sizeof dist);
21     dist[1] = 0;
22
23     //起点入队
24     q[++tt] = 1;
25     st[1] = true;
26
27     while (hh <= tt) {
28         int t = q[hh++];
29         st[t] = false;    //已出队
30
31         //更新t的所有出边
32         for (int i=h[t]; i!=-1; i=ne[i]) {
33             int j = e[i];
34             if (dist[j] > dist[t]+w[i]) {
35                 dist[j] = dist[t]+w[i];
36
37                 if (!st[j]) {
38                     q[++tt] = j;
39                     st[j] = true;
40                 }
41             }
42         }
43     }
44 }
```

```

42     }
43 }
44
45     if (dist[n] == 0x3f3f3f3f) return 0x3f3f3f3f;
46     return dist[n];
47 }
48
49 int main(void) {
50     scanf("%d%d", &n, &m);
51
52     memset(h, -1, sizeof h);
53
54     for (int i=0; i<m; i++) {
55         int a, b, c;
56         scanf("%d%d%d", &a, &b, &c);
57         add(a, b, c);
58     }
59
60     int t = spfa();
61
62     if (t == 0x3f3f3f3f) puts("impossible");
63     else printf("%d", t);
64
65     return 0;
66 }

```

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

```

1 dist[x] = dist[t]+w[i];
2 cnt[x] = cnt[t]+1;

```

3 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 #include <iostream>
2 #include <cstring>
3 #include <queue>
4 #include <algorithm>
5
6 using namespace std;
7
8 const int N = 2010, M = 10010;
9

```

```

10  int n, m;
11  int h[N], e[M], w[M], ne[M], idx;
12  int dist[N], cnt[N];    //cnt[x] 当前1号点到x最短路径的边数
13  queue<int> q;
14  bool st[N];
15
16  void add(int a, int b, int c) {
17      e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
18  }
19
20  bool spfa() {
21      //判断负环可以不用初始化，初始时将所有点加入队列
22      for (int i=1; i<=n; i++) {
23          q.push(i);
24          st[i] = true;
25      }
26
27      while (q.size()) {
28          int t = q.front(); q.pop();
29          st[t] = false;
30
31          for (int i=h[t]; i!=-1; i=ne[i]) {
32              int j = e[i];
33
34              if (dist[j] > dist[t]+w[i]) {
35                  dist[j] = dist[t]+w[i];
36                  cnt[j] = cnt[t]+1;
37
38                  if (cnt[j] >= n) return true;
39
40                  if (!st[j]) {
41                      q.push(j);
42                      st[j] = true;
43                  }
44              }
45          }
46      }
47
48      return false;
49  }
50
51  int main(void) {
52      scanf("%d%d", &n, &m);
53
54      memset(h, -1, sizeof h);
55
56      for (int i=0; i<m; i++) {
57          int a, b, c;
58          scanf("%d%d%d", &a, &b, &c);
59          add(a, b, c);
60      }
61
62      if (spfa()) puts("Yes");
63      else puts("No");
64
65
66      return 0;
67  }

```


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

```
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  #include <iostream>
2  #include <cstring>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 210, INF = 0x3f3f3f3f;
8
9  int n, m, Q;
10 int d[N][N];
11
12 void floyd() {
13     for (int k=1; k<=n; k++)    //枚举阶段
14         for (int i=1; i<=n; i++)
15             for (int j=1; j<=n; j++)
16                 d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
17 }
18
19 int main(void) {
20     scanf("%d%d%d", &n, &m, &Q);
21
22     for (int i=1; i<=n; i++)
23         for (int j=1; j<=n; j++)
24             if (i == j) d[i][j] = 0;
25             else d[i][j] = INF;
```

```
26
27     for (int i=0; i<m; i++) {
28         int a, b, c;
29         scanf("%d%d%d", &a, &b, &c);
30         d[a][b] = min(d[a][b], c);
31     }
32
33     floyd();
34
35     while (Q--) {
36         int x, y;
37         scanf("%d%d", &x, &y);
38
39         if (d[x][y] > INF/2) puts("impossible");
40         else printf("%d\n", d[x][y]);
41     }
42
43     return 0;
44 }
```