目录

1 最短路问题

1.1 应用场景

1.1.1 单源最短路问题

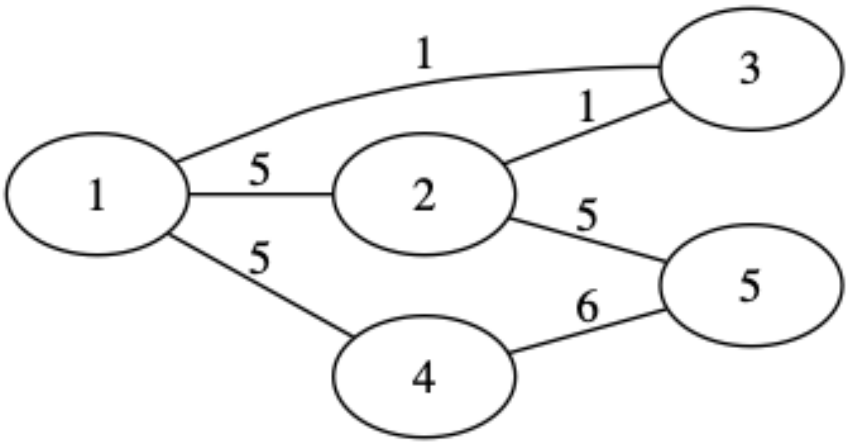
给定一个图（有向图或者无向图），求图中任意节点距离某一节点的最短距离。

1.1.2 多源最短路问题

给定一个图（有向图或者无向图），求图中任意两个节点之间的最短距离。

1.1.3 例子

- 求所有节点到节点 1 的最短距离



1.2 变量定义

1.2.1 定义图–邻接表

```
#include <vector>
const int maxn = 1010;
struct edge{
   int nextId;
   int dist;
};
vector <edge> node[maxn];
```

1.2.2 定义图–邻接矩阵

#+BEGIN$_{\text{SRC}}$ c++ //edge[i][j] 表示 i 指向 j 的边的权重 int edge[maxn][maxn];

1.2.3 定义距离原点距离

```
int dist[maxn];
```

1.3 BFS（广度优先搜索）

1.3.1 流程

1. 初始化

   ```
   const int INF = 1e9;
   queue <int> que;
   que.push(sNode);
   for (int i = 0 ;i<= n ;i++)
       dist[i] = INF;
   dist[sNode] = 0;
   ```

2. BFS 流程

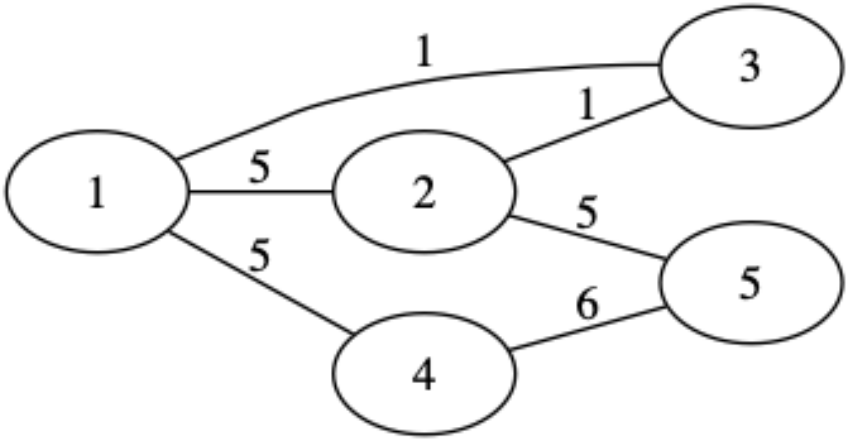
```
while (!que.empty()){
    int nId = que.front();
    que.pop();
    for (int i = 0 ;i< node[nId].size() ;i++){
        int nextId = node[nId][i].nextId;
        if (node[nId][i].dist + dist[nId]< dist[nextId]){
           dist[nextId] = node[nId][i].dist + dist[nId];
           que.push(nextId);
        }
    }
}
```

1.3.2 时间复杂度

节点个数 N，边个数 M $O(N^2)$

1.3.3 举例

- 求所有节点到节点 1 的最短距离



1. 初始化
    - 所有节点距离源节点的距离 dist

1	2	3	4	5
0	INF	INF	INF	INF

- 初始化 Queue 状态 [1]

2. 循环详情

(a) step 1

- 弹出队顶元素 1, 更新 dist

1	2	3	4	5
0	5	1	5	INF

- 把所有更新的元素 [2,3,4] 放入队列中 Queue 状态:[2,3,4]

(b) step 2

- 弹出队顶元素 2, 更新 dist

1	2	3	4	5
0	5	1	5	10

- 把所有更新的元素 [5] 放入队列中 Queue 状态:[3,4,5]

(c) step 3

- 弹出队顶元素 3, 更新 dist

1	2	3	4	5
0	2	1	5	10

- 把所有更新的元素 [2] 放入队列中 Queue 状态:[4,5,2]

(d) step 4

- 弹出队顶元素 4, 更新 dist

1	2	3	4	5
0	2	1	5	10

- 把所有更新的元素 [] 放入队列中 Queue 状态:[5,2]

(e) step 5

- 弹出队顶元素 5, 更新 dist

1	2	3	4	5
0	2	1	5	10

- 把所有更新的元素 [] 放入队列中 Queue 状态:[2]

(f) step 6

- 弹出队顶元素 2, 更新 dist

1	2	3	4	5
0	2	1	5	7

- 把所有更新的元素 [5] 放入队列中 Queue 状态:[5]

(g) step 7

- 弹出队顶元素 5, 更新 dist

1	2	3	4	5
0	2	1	5	7

- 把所有更新的元素 [] 放入队列中 Queue 状态:[]

(h) 队列为空结束

1.4 Bellman-Ford 算法

以边为思考中心的最短路径算法。可以发现负环的情况。

1.4.1 图结构存储

把所有边信息存储起来

```
const int maxm = 1010;
struct edge{
    int u;
    int v;
    int dist;
};
edge E[maxm];
```

1.4.2 流程

1. 初始化 初始化所有节点距离源节点的距离

```
const int INF = 1e9;
for (int i = 0 ;i<= n ;i++)
    dist[i] = INF;
dist[sNode] = 0;
```

2. Bellman-Ford 流程

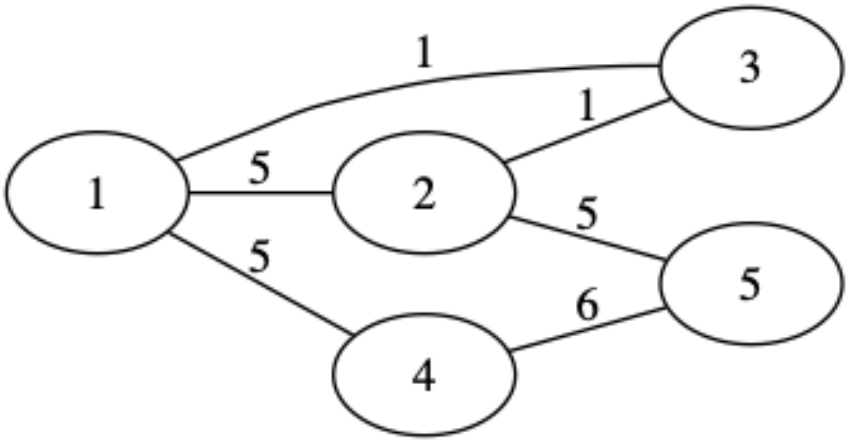
```
for (int i = 0 ;i< n-1 ;i++){
   bool isOk = false;
   for (int j = 0; j< m ;j++){
       int u = E[j].u;
       int v = E[j].v;
       int d = E[j].dist;
       if (dist[u] + d < dist[v]){
           dist[v] = dist[u] + d;
           isOk = true;
       }
       else if (dist[v] + d < dist[u]){
           dist[u] = dist[v] + d;
           isOk = true;
       }
   }
   if (!isOk) break;
}
```

1.4.3 时间复杂度

节点个数 N，边个数 M O(N*M)

1.4.4 举例

- 求所有节点到节点 1 的最短距离



1. 初始化

   - 所有节点距离源节点的距离 dist

1	2	3	4	5
0	INF	INF	INF	INF

2. 循环

   (a) step 1

   - 遍历所有边, 更新边两端端点距离源点的距离
     - 1 - 2 2 变为 5
     - 1 - 3 3 变为 1
     - 1 - 4 4 变为 5
     - 2 - 3 不能更新
     - 2 - 5 不能更新
     - 4 - 5 不能更新
   - 所有节点距离源节点的距离 dist

1	2	3	4	5
0	5	1	5	INF

   (b) step 2

- 遍历所有边, 更新边两端端点距离源点的距离
  - 1 - 2 不更新
  - 1 - 3 不更新
  - 1 - 4 不更新
  - 2 - 3 2 变成 2
  - 2 - 5 5 变为 10
  - 4 - 5 5 变为 11
- 所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	10

(c) step 3

- 遍历所有边, 更新边两端端点距离源点的距离
  - 1 - 2 不更新
  - 1 - 3 不更新
  - 1 - 4 不更新
  - 2 - 3 不更新
  - 2 - 5 5 变为 7
  - 4 - 5 不更新
- 所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	7

(d) step 4

- 遍历所有边, 更新边两端端点距离源点的距离
  - 1 - 2 不更新
  - 1 - 3 不更新
  - 1 - 4 不更新
  - 2 - 3 不更新
  - 2 - 5 不更新

– 4 - 5 不更新

- 所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	7

(e) 终止条件为全部不更新

1.5 SPFA 算法

优先队列优化的 Bellman-Ford 算法。

1.5.1 STL 优先队列

1. 优先队列方法

方法	功能
empty()	如果队列为空，则返回真
pop()	删除对顶元素，删除第一个元素
push()	加入一个元素
size()	返回优先队列中拥有的元素个数
top()	返回优先队列对顶元素，返回优先队列中有最高优先级的元素

2. 普通方法声明

```
#include <queue>
priority_queue <int> q;
//通过操作，按照元素从大到小的顺序出队
priority_queue <int,vector<int>, greater<int> > q2;
//通过操作，按照元素从小到大的顺序出队
```

3. 自定义优先级声明

```
#include <queue>
struct cmp {
```

```
    operator bool ()(int x, int y)
    {
         return  x > y;
    //x小的优先级高
    //也可以写成其他方式,
    //如:  return p[x] > p[y];
    //表示p[i]小的优先级高
    }
};
//定义方法
priority_queue<int, vector<int>, cmp> q;
```

4. 结构体声明方式

```
#include <queue>
struct node {
    int x, y;
    friend bool operator < (node a, node b)
    {
        return a.x > b.x;    //结构体中, x小的优先级高
    }
};
priority_queue<node>q;   //定义方法
//在该结构中, y为值, x为优先级。
//通过自定义operator<操作符来比较元素中的优先级。
//在重载”<”时, 最好不要重载”>”, 可能会发生编译错误
```

1.5.2 流程

1. 初始化

```
const int INF = 1e9;
```

```
struct upd{
   int nextId;
   int val;
   friend bool operator < (upd a, upd b)
   {
        return a.val > b.val;    //结构体中，x小的优先级高
   }
};
priority_queue <upd> que;
upd tmp; tmp.nextId = sNode; tmp.val = 0;
que.push(tmp);
for (int i = 0 ;i<= n ;i++)
    dist[i] = INF;
dist[sNode] = 0;
```

2. BFS 流程

```
while (!que.empty()){
    upd tmp = que.top();
    int nId = tmp.nextId;
    que.pop();
    for (int i = 0 ;i< node[nId].size() ;i++){
        int nextId = node[nId][i].nextId;
        if (node[nId][i].dist + dist[nId]< dist[nextId]){
           dist[nextId] = node[nId][i].dist + dist[nId];
           tmp.nextId = nextId;
           tmp.val = dist[nextId];
           que.push(tmp);
        }
    }
}
```

1.6 Dijkstra 算法

以点为思考中心的最短路径算法。

1.6.1 图结构存储

邻接表

1.6.2 流程

1. 初始化

```
const int INF = 1e9;
bool hasFind[maxn];
for (int i = 0 ;i<= n ;i++)
    dist[i] = INF;
dist[sNode] = 0;
memset(hasFind,0,sizeof hasFind);
hasFind[sNode] = true;
```

2. Dijkstra 算法

```
for (int i = 0 ;i< n-1 ;i++){
    int nId = -1 ;
    for (int j = 0 ;j< n ;j++){
        if (!hasFind[j]){
           if (nId == -1)
               nId = j;
           else if (dist[j]<dist[nId])
               nId = j;
        }
    }
    hasFind[nId] = true;
    for (int i = 0 ;i< node[nId].size() ;i++){
```

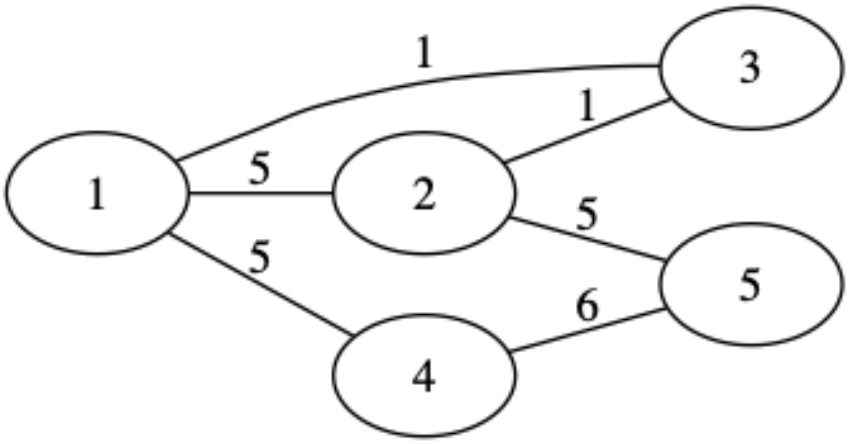
```
            int nextId = node[nId][i].nextId;
            if (node[nId][i].dist + dist[nId]< dist[nextId]){
               dist[nextId] = node[nId][i].dist + dist[nId];
               que.push(nextId);
            }
        }
    }
```

1.6.3 时间复杂度

节点个数 N，边个数 M O(N*N)

1.6.4 举例

- 求所有节点到节点 1 的最短距离



1. 初始化
    - 将源节点 1，放入已获取最短路径集合, 集合变为 {1}
    - 未获取最短路径节点结合 {2,3,4,5}
    - 根据节点 1 来更新所有节点距离源节点的距离 dist

1	2	3	4	5
0	5	1	5	INF

2. 流程

   (a) step 1:

   - 从未获取最短路径节点结合 {2,3,4,5} 中，选取距离源节点最近的节点 3
   - 将节点 3，放入已获取最短路径集合, 集合变为 {1,3}
   - 根据节点 3 来更新所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	INF

   (b) step 2:

   - 从未获取最短路径节点结合 {2,4,5} 中，选取距离源节点最近的节点 2
   - 将节点 2，放入已获取最短路径集合, 集合变为 {1,2,3}
   - 根据节点 2 来更新所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	7

   (c) step 3:

   - 从未获取最短路径节点结合 {4,5} 中，选取距离源节点最近的节点 4
   - 将节点 4，放入已获取最短路径集合, 集合变为 {1,2,3,4}
   - 根据节点 4 来更新所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	7

   (d) step 4:

   - 从未获取最短路径节点结合 {5} 中，选取距离源节点最近的节点 5
   - 将节点 5，放入已获取最短路径集合, 集合变为 {1,2,3,4,5}

- 根据节点 5 来更新所有节点距离源节点的距离 dist

1	2	3	4	5
0	2	1	5	7

(e) 终止条件，所有节点都放入到了已获取最短路径集合。

1.7 Floyd 算法

动态规划类型的最短路径算法。应用场景: 多源最短路问题

1.7.1 动态转移方程

dist[i][j] = min(dist[i][j],dist[i][k] + dist[k][j]);

1.7.2 初始化

```
//dist[i][j] 表示从i到j之间的最短距离
int dist[maxn][maxn];
for (int i = 0; i< n ;i++)
   for (int j = 0 ;j< n ;j++)
       dist[i][j] = edge[i][j];
```

1.7.3 Floyd 流程

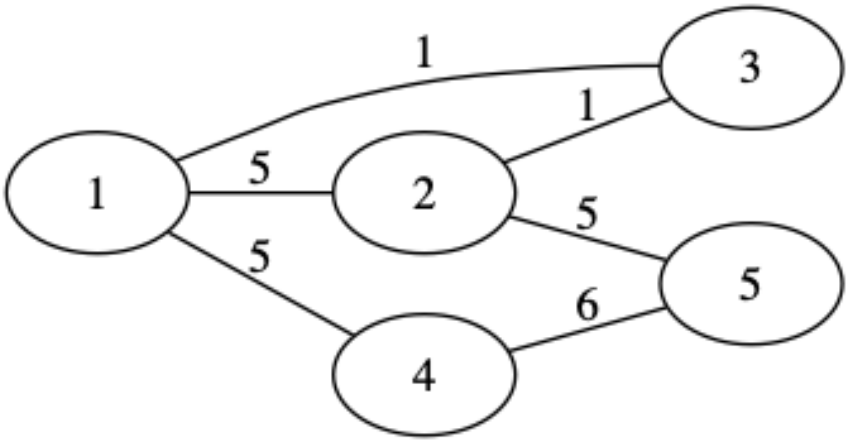
```
for (int k = 0 ;k< n ;k++)
    for (int i = 0 ;i< n ;i++)
        for (int j = 0 ;j< n ;j++)
            dist[i][j] = min(dist[i][j],dist[i][k] + dist[k][j]);
```

1.7.4 时间复杂度

节点个数 N，边个数 M $O(N^3)$

1.7.5 举例

- 求所有节点到节点 1 的最短距离



1. 初始化

   - dist 矩阵
     - dist[i][j] 表示节点 i 到节点 j 之间的最短路径长度
     - dist 初始化为 edge

	1	2	3	4	5
1	0	5	1	5	INF
2	5	0	1	INF	5
3	1	1	0	INF	INF
4	5	INF	INF	0	6
5	INF	5	INF	6	0

2. 流程

   (a) step 1

   - 通过节点 1 作为中转节点更新 dist
   - 更新公式 dist[i][j] = min(dist[i][1] + dist[1][j],dist[i][j]);

	1	2	3	4	5
1	0	5	1	5	INF
2	5	0	1	10	5
3	1	1	0	6	INF
4	5	10	6	0	6
5	INF	5	INF	6	0

(b) step 2

- 通过节点 2 作为中转节点更新 dist
- 更新公式 dist[i][j] = min(dist[i][2] + dist[2][j],dist[i][j]);

	1	2	3	4	5
1	0	5	1	5	10
2	5	0	1	10	5
3	1	1	0	6	6
4	5	10	6	0	6
5	10	5	6	6	0

(c) step 3

- 通过节点 3 作为中转节点更新 dist
- 更新公式 dist[i][j] = min(dist[i][3] + dist[3][j],dist[i][j]);

	1	2	3	4	5
1	0	2	1	5	7
2	2	0	1	7	5
3	1	1	0	6	6
4	5	7	6	0	6
5	7	5	6	6	0

(d) step 4

- 通过节点 4 作为中转节点更新 dist
- 更新公式 dist[i][j] = min(dist[i][4] + dist[4][j],dist[i][j]);

	1	2	3	4	5
1	0	2	1	5	7
2	2	0	1	7	5
3	1	1	0	6	6
4	5	7	6	0	6
5	7	5	6	6	0

(e) step 5

- 通过节点 5 作为中转节点更新 dist
- 更新公式 dist[i][j] = min(dist[i][5] + dist[5][j],dist[i][j]);

	1	2	3	4	5
1	0	2	1	5	7
2	2	0	1	7	5
3	1	1	0	6	6
4	5	7	6	0	6
5	7	5	6	6	0

1.8 相关题目

1.8.1 HDU 1874

1.8.2 HDU 2112

1.8.3 HDU 2680

1.8.4 HDU 1217

1.8.5 HDU 1221

1.8.6 POJ 1217