Floyd算法

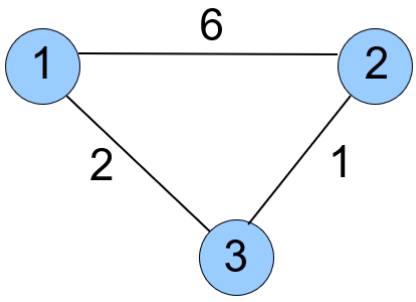
> Floyd(弗洛伊德)算法，是最简单的最短路径算法，可以计算图中·任意两点间的最短路径·。Floyed的时间复杂度是O(N^3)，适用于出现负边权的情况，不能处理负权回路。

算法思想：

三层循环，第一层循环中介点`k`，第二第三层循环起点终点`i`、`j`，如果点i到点k的距离加上点k到点j的距离小于原先点i到点j的距离，那么就用这个更短的路径长度来更新原先点i到点j的距离。

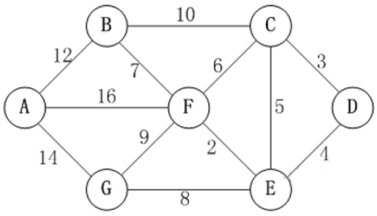
```
for(int k = 1; k <= n; k++)
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

如图所示，因为`dis[1][3]+dis[3][2]<dis[1][2]`，所以就用`dis[1][3]+dis[3][2]`来更新原先1到2的距离。



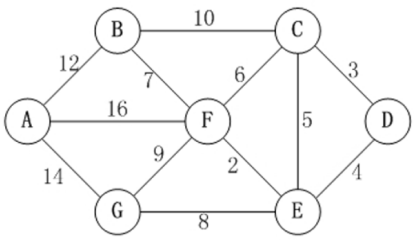
算法图解

建立邻接矩阵



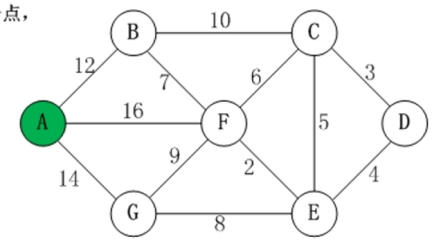
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第1步：
初始化矩阵S



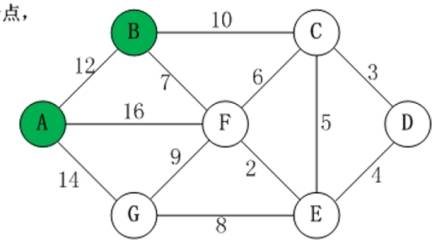
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：
以顶点A为中介点，
更新矩阵S。



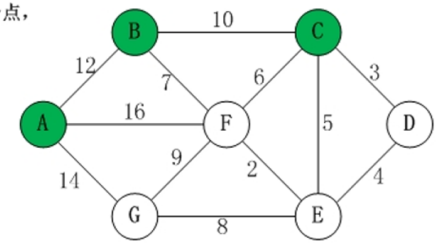
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：
以顶点B为中介点，
更新矩阵S。

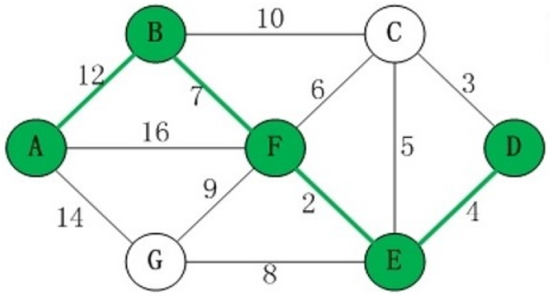


	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

第4步：
以顶点C为中介点，
更新矩阵S。

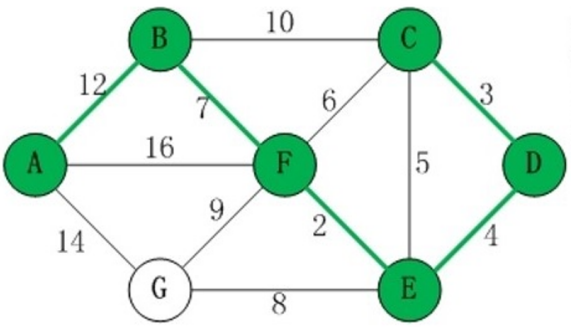


	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0



第5步：
选取顶点D

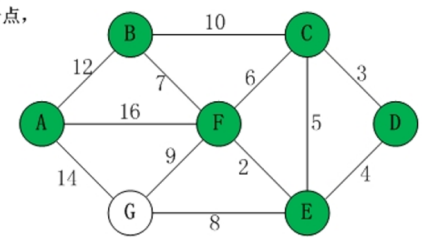
U={A, B, F, E, D}
V-U={C, G}



第6步：
选取顶点C

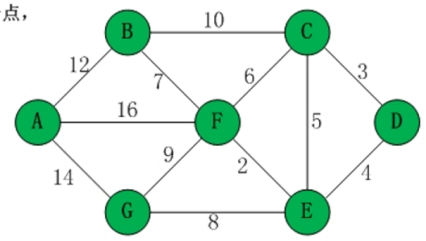
U={A, B, F, E, D, C}
V-U={G}

第7步：
以顶点F为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

第8步：
以顶点G为中介点，
更新矩阵S。



$$
\begin{array}{c}
 \\
A \\
B \\
C \\
D \\
E \\
F \\
G
\end{array}
\begin{bmatrix}
A & B & C & D & E & F & G \\
0 & 12 & 22 & 22 & 18 & 16 & 14 \\
12 & 0 & 10 & 13 & 9 & 7 & 16 \\
22 & 10 & 0 & 3 & 5 & 6 & 13 \\
22 & 13 & 3 & 0 & 4 & 6 & 12 \\
18 & 9 & 5 & 4 & 0 & 2 & 8 \\
16 & 7 & 6 & 6 & 2 & 0 & 9 \\
14 & 16 & 13 & 12 & 8 & 9 & 0
\end{bmatrix}
$$

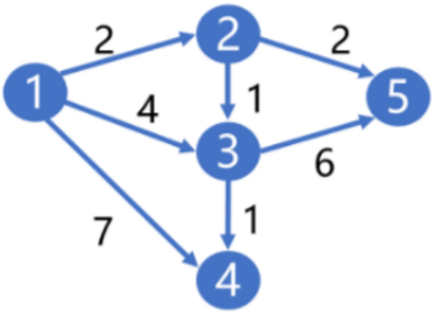
练习

- P108 信使

狄克斯特拉(Dijkstra)算法

狄克斯特拉（Dijkstra）算法是求解最短路径问题的算法，可以求得从起点到终点的路径中权重总和最小的那条路径路径。

如果我们想求出起点到一个点的最短路径，必然要先求出中转点的最短路径，也就是说，如果起点1到某一点v_0的最短路径要经过中转点v_i，那么中转点v_i一定是先于v_0被确定的最短路径的点。



中转点	终点	最短路径
1	1	0
1	2	2
1、2	3	3
1、2、3	4	4
1、2	5	4

求解顺序

算法实现

```
设起点为s，dis[v]表示从s到v的最短路径。
初始化：dis[v]=∞(v≠s)    dis[s]=0

for (i = 1; i <= n ; i++)
        1.在没有被访问过的点中找一个顶点u使得dis[u]是最小的。
        2.u标记为已确定最短路径
        3.for 与u相连的每个未确定最短路径的顶点v
            if(dis[u]+w[u][v] < dis[v]) {
                dis[v] = dis[u] + w[u][v];
}

dis[v]为s到v的最短距离
```

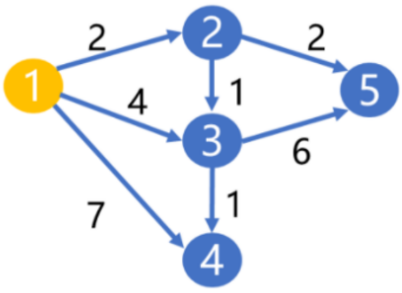
```
void dijkstra(){
    memset(dis, 0x3f, sizeof dis);
```

```
    dis[1] = 0;

    for(int i = 1; i <= n; i++){
        int u = -1;
        for(int j = 1; j <= n; j++){
            if(!book[j] && (u == -1 || dis[u] > dis[j]))
                u = j;
        }
        book[u] = true;

        for(int j = 1; j <= n; j++)
            dis[j] = min(dis[j], dis[u] + g[u][j]);
    }
}
```

算法图解

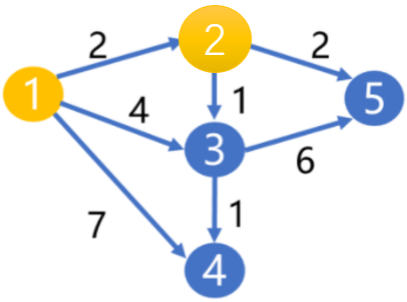
> 已经确定最短路径的点标黄点，未确定最短路径的点标为蓝点。

第一轮循环从1号点开始，对所有蓝点做出修改。



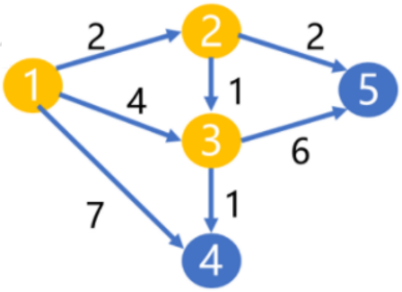
dis[] = | 0 | 2 | 4 | 7 | INF |

第二轮循环找到dis[2]最小，将2变为黄点，对所有蓝点做出修改。



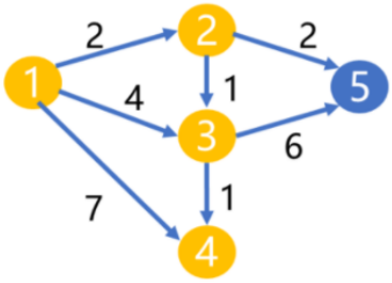
dis[] = | 0 | 2 | 3 | 7 | 4 |

第三轮循环找到dis[3]最小，将3变为黄点，对所有蓝点做出修改。



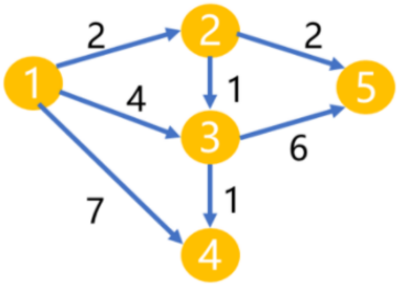
dis[] = | 0 | 2 | 3 | 4 | 4 |

第四轮循环找到dis[4]最小，将4变为黄点，对所有蓝点做出修改。



dis[] = | 0 | 2 | 3 | 4 | 4 |

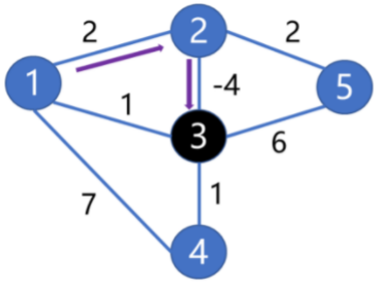
第五轮循环找到dis[5]最小，将5变为黄点，对所有蓝点做出修改。

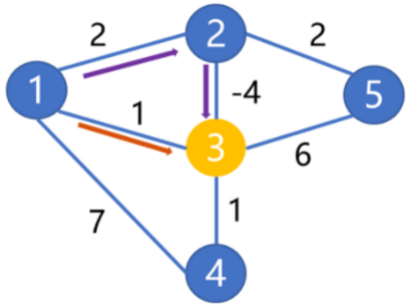


dis[] = | 0 | 2 | 3 | 4 | 4 |

朴素法Dijkstra

- 朴素法狄克斯特拉（Dijkstra）算法时间复杂度是O(N^2)
- 不能处理负边权情况





堆优化Dijkstra

- 时间复杂度`O(mlogn)`：

```
void dijkstra(){
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

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

    while(!heap.empty()){
        PII t = heap.top();
        heap.pop();

        int d = t.first, u = t.second;
        if(book[u]) continue;
        book[u] = true;
```

```
        for(int i = h[u]; ~i; i = ne[i]){
            int j = to[i];
            if(dis[j] > dis[u] + w[i]){
                dis[j] = dis[u] + w[i];
                heap.push({dis[j], j});
            }
        }
    }
}
```

练习

- P154 朴素法Dijkstra求最短路
- P155 堆优化Dijkstra求最短路

SPFA算法

算法简介

SPFA(Shortest Path Faster Algorithm)算法是求单源最短路径的一种算法，它是Bellman-ford的队列优化，它是一种十分高效的最短路算法。

很多时候，给定的图存在负权边，这时类似Dijkstra等算法便没有了用武之地，而Bellman-Ford算法的复杂度又过高，SPFA算法便派上用场了。SPFA的复杂度大约是O(kE)，E是边数，K是常数，平均值为2。

算法思想

初始时将起点加入队列。每次从队列中取出一个元素，并对所有与它相邻的点进行修改，若某个相邻的点修改成功，则将其入队。直到队列为空时算法结束。

这个算法，简单的说就是队列优化的bellman-ford，利用了每个点不会更新次数太多的特点发明的此算法。

SPFA 在形式上和广度优先搜索非常类似，不同的是广度优先搜索中一个点出了队列就不可能重新进入队列，但是SPFA中一个点可能在出队列之后再次被放入队列，也就是说一个点修改过其它的点之后，过了一段时间可能会获得更短的路径，于是再次用来修改其它的点，这样反复进行下去。

此外，SPFA算法还可以判断图中是否有负权环，即一个点入队次数超过N。

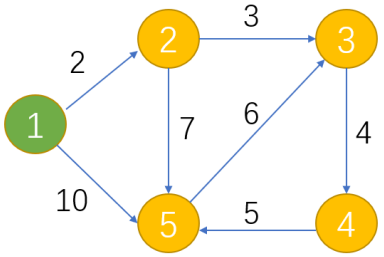
```
void spfa(){
    memset(dis, 0x3f, sizeof dis);
    dis[S] = 0;

    int hh = 0, tt = 1;
    q[0] = S, book[S] = true;
    while(hh != tt){
        int t = q[hh++];
```

```
        if(hh == N) hh = 0;
        book[t] = false;

        for(int i = h[t]; ~i; i = ne[i]){
            int j = to[i];
            if(dis[j] > dis[t] + w[i]){
                dis[j] = dis[t] + w[i];
                if(!book[j]){
                    q[tt++] = j;
                    if(tt == N) tt = 0;
                    book[j] = true;
                }
            }
        }
    }
}
```

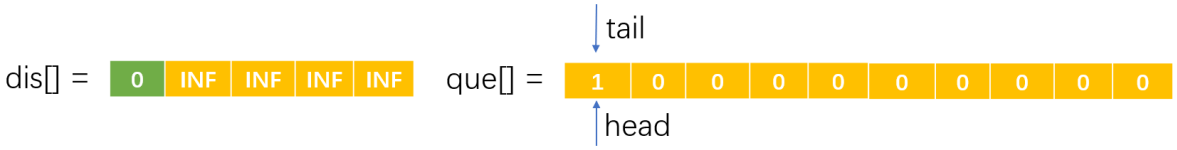
算法图解

> 5个顶点，7条边，7个边权，求顶点1到其他各个点最小值。

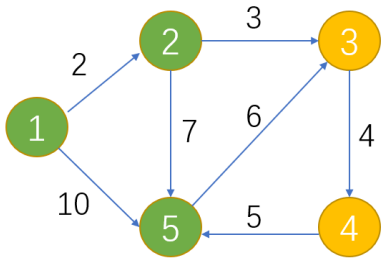
起点`1`入队，`dis[1]=0, book[1]=true`。



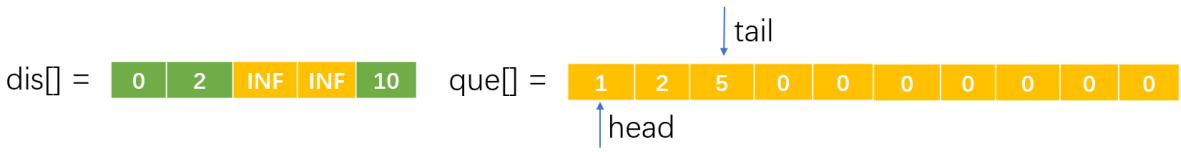

```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
0 2 5 9 9
```



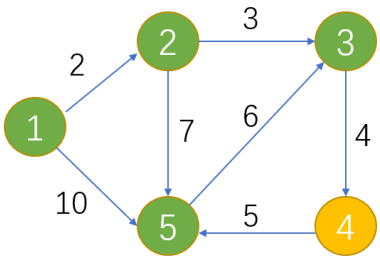
修改`2,5`。



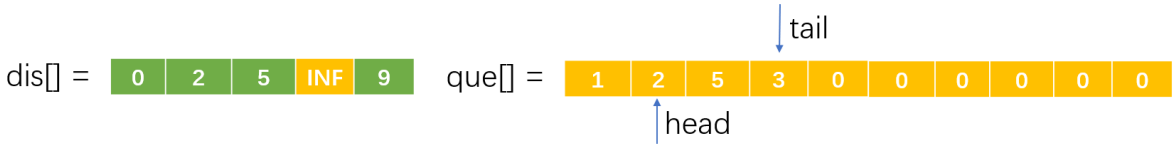

```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
0 2 5 9 9
```



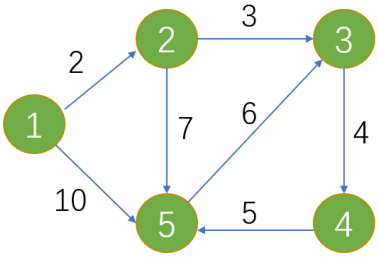
修改`3`。



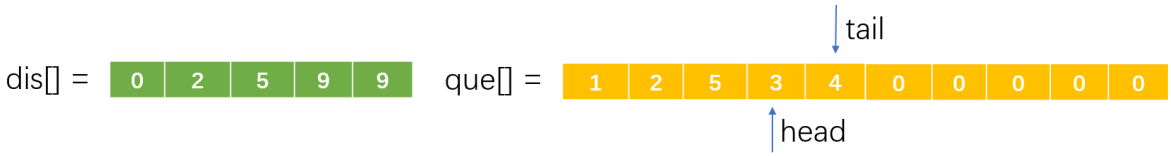

```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
0 2 5 9 9
```



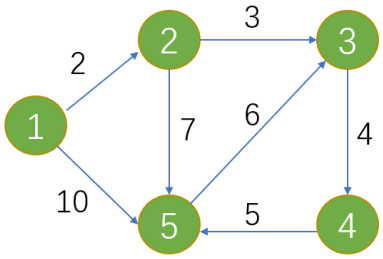
修改`4`。




```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
0 2 5 9 9
```



队列为空，算法结束。




```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
0 2 5 9 9
```

练习

- P95 热浪