图论之最短路径

0000.什么是图

00.图论简介

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象。**图 (Graph)** 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

示意图:

01.图的概念

00.组成部分

1. **顶点**：就是下**图中的圆圈。**上面的数字就是他的编号，用来描述不同的点。
2. **边**：下图中**连接两个圆圈的线**就是边。
3. **边权**：下图中边上面的数字就是边权。可以理解为**边的长度**，也就是两个点的距离。但实际上边**权是可以为负数**的，所以不是完全等于长度。

01.相关概念

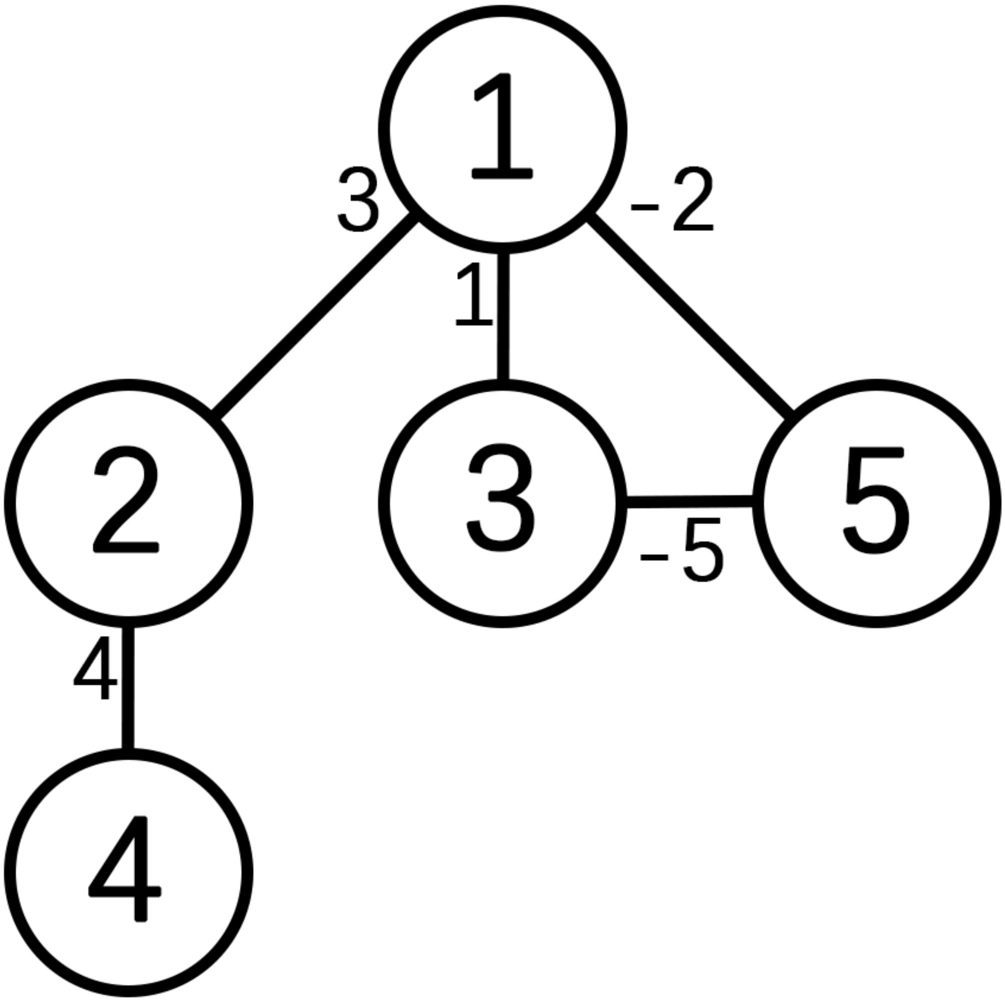
1. **无向图**：没有方向性的边组成的图。
2. **有向图**：有方向性的边组成的图。
3. **负权图**：**带有**负边权的图。
4. **正权图**：**没有**负边权的图。
5. **负环**：一个回路的**总权值**为负。
6. **重边**：一个点到另一个点有**多条边。**
7. **自环**：一个点有**自己到自己**的边。
8. **相邻**：两个点之间有**边**，则称之为相邻。

0010.图的储存

00.邻接矩阵

邻接矩阵的方法是用一个二维数组`e`，其中`e[i][j]`表示**i到j点的边权。**如果两个不相邻，则存为**正无穷。**

示意图:



	1	2	3	4	5
1	0	1	1	∞	1
2	1	0	∞	1	∞
3	1	∞	0	∞	1
4	∞	1	∞	0	∞
5	1	∞	1	∞	0

01.邻接表

邻接表是按边的方式进行的存储，我们需要把边进行编号从1->m。存储边的这一部分我们使用三个数组：`u` `v` `w`，其中`u[i]`表示**第i条边的起始点**，`v[i]`表示**第i条边的终止点**，`w[i]`表示**第i条边的边权**。然后我们还需要让边与顶点、边与边联系起来，我们使用`first[i]`表示**第i个点的第一条边**，`next[i]`表示第i条边的下一条边。

示意图:

```
4 5
1 4 9
4 3 8
1 2 5
2 4 6
1 3 7
```

①读入第一条边

	U	V	W	first	next
1	1	4	9	1	-1
2				-1	
3				-1	
4				-1	
5					

②读入第二条边

	U	V	W	first	next
1	1	4	9	1	-1
2	4	3	8	-1	-1
3				-1	
4				2	
5					

③读入第三条边

	U	V	W	first	next
1	1	4	9	3	-1
2	4	3	8	-1	-1
3	1	2	5	-1	1
4				2	
5					

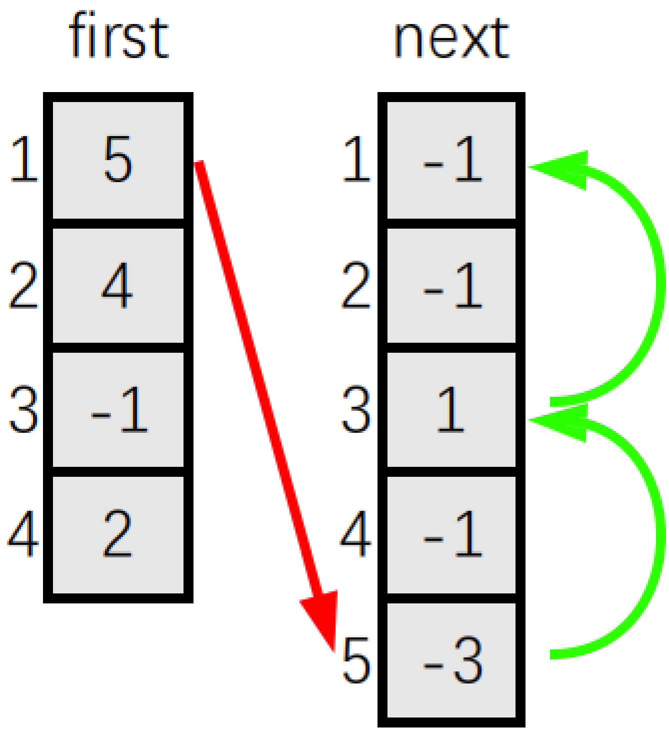
④读入第四条边

	U	V	W	first	next
1	1	4	9	3	-1
2	4	3	8	4	-1
3	1	2	5	-1	1
4	2	4	6	2	-1
5					

⑤读入第五条边

	U	V	W	first	next
1	1	4	9	5	-1
2	4	3	8	4	-1
3	1	2	5	-1	1
4	2	4	6	2	-1
5	1	3	7		3

eg遍历点1的边:

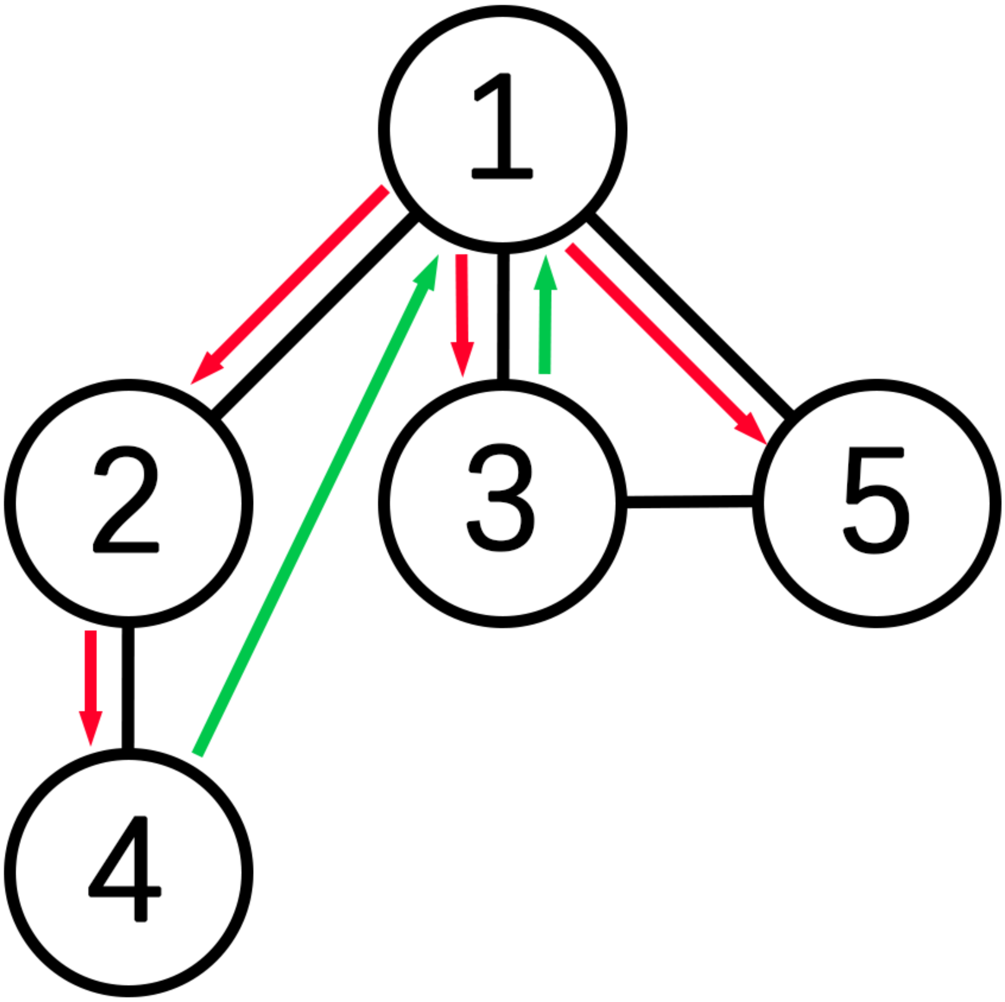


0011.图的遍历

00.DFS

00.伪代码

```
dfs(cur)
    cnt+1
    if cnt=n stop
    for i:1->n
        if cur-j连接 and j未遍历
            dfs(i)
```



01.模板代码

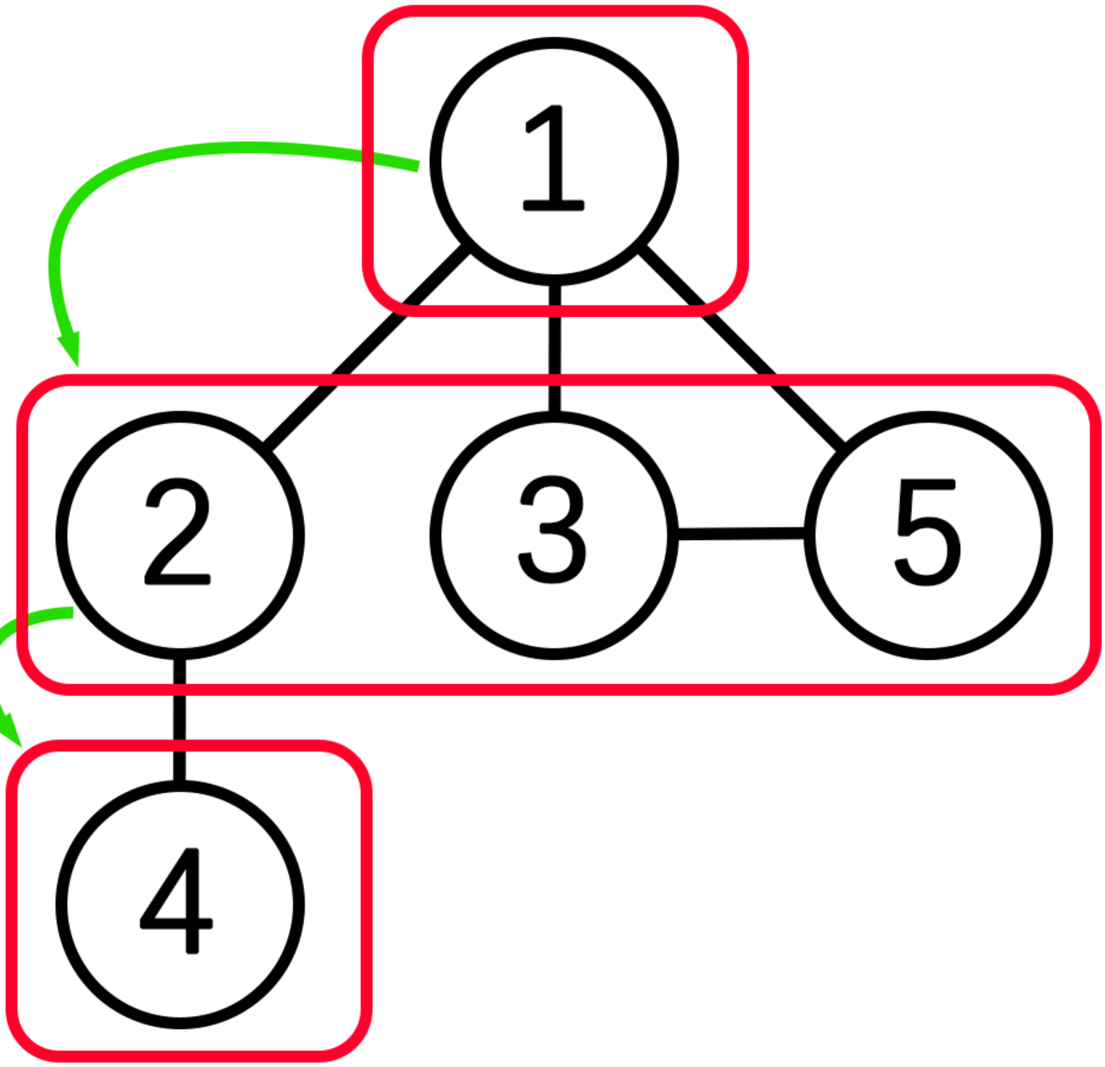
```
/*
 *cur: 当前的点
 *cnt:遍历的点个数
 *INF:正无穷
 *matrix:邻接矩阵
 *book:标记数组
 */
void depthFirstSearchMatrix(int cur){
    printf("%d ",cur);
    ++cnt;
    if(cnt==n) return;
    for(int i=1;i<=n;i++)
        if(matrix[cur][i]<INF&&book[i]==0){
            book[i]=1;
            depthFirstSearchMatrix(i);
        }

    return;
}
```

01.BFS

00.算法实现

使用一个队列que

1. 将源点s入队
2. 将对头p点**相邻**并且没有遍历的点入队，**将对头出队。**如果队列不为空，则执行2



01.伪代码

```
que.push s

while not que.empty
    cur=que.front
    for i:1->n
        if cur-i连接 i未遍历
            que.push i
    que.pop
```

10.模板代码

```
/*
 *INF:正无穷
 *cur: 当前的点
 *matrix:邻接矩阵
 *que:队列
 *book:标记数组
 */
void breadthFirstSearchMatrix(){
    queue que;
    int cur=s;
    for(int i=0;i<MAX_N;i++)book[i]=0;

    que.push(cur);
    book[cur]=1;

    while(!que.empty()){
        cur=que.front();
        for(int i=1;i<=n;i++){
            if(matrix[cur][i]!=INF&&!book[i]){
                que.push(i);
                book[i]=1;
            }
        }
        printf("%d ",cur);
        que.pop();
    }

    return;
}
```

0100.最短路径

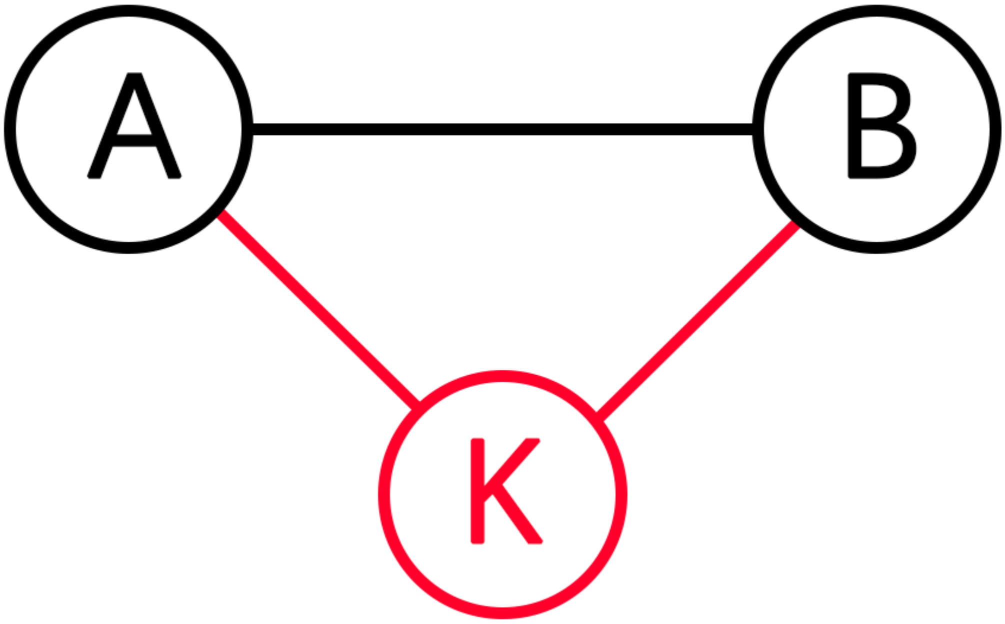
00.Floyd-Warshall

Floyd-Warshall，是解决全源最短路径问题的算法，可以求出图上**任意两个点间**的最短路径。

00.算法原理

要优化`S(i->j)`，**可以引入点k，使得**`S(i->k)+S(k->j)<S(i->j)`

示意图:



那么我们枚举k,i,j则可以求出全源最短路径。

10.伪代码

伪代码:

```
for k:1->n
    for i:1->n
        for j:1->n
            e[i][j]=min(e[i][k]+e[k][j],e[i][j])
```

11.模板代码

模板代码:

```
/*
 *k:中专点
 *i:起始点
 *j:终止点
 *INF:正无穷
 *matrix:邻接矩阵
 */
void Floyd_WarshallMatrix(){
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                matrix[i][j]=min(matrix[i][k]+matrix[k][j],matrix[i][j]);

    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++)
            if(matrix[i][j]==INF) printf("INF ");
            else printf("%d ",matrix[i][j]);
        putchar('\n');
```

```
    }

    return;
}
```

明显看出，其时间复杂度为O(N^3)

01.Dijkstra

Dijkstra/ˈdaɪkstrə/迪杰斯特拉算法是处理**一点到其余个点的短路**，也叫做"**单源最短路径**"。以下简称dij算法。实际上dij算法基于一个**没有负权**的图之上的一个**贪心**或者说**动归**算法。

朴素版

00.算法原理

我们找到一个与源点s最相近的点k，标记k。并且遍历k的所有出边，如果p没有被标记`S(s->k)+S(k->p)<S(s->p)`，**则让**`S(s->k)+S(k->p)=S(s->p)`

枚举k，则可以求出以源点s的最短路径。

01.伪代码

伪代码:

```
init dis

for i:1->n-1
    min=find(dis)
    book u
    for j:1->n
        if !book j
            dis[j]=min(dis[u]+S(u->j),dis[j])
```

10.模板代码

模板代码:

```
/*
 *matrix:邻接矩阵
 *vis:记录是否访问过
 *dis:记录距离
 *min:记录最小距离
 *u:记录最小距离的点
 *INF:正无穷
 */
void DijkstraMatrix(){
    int dis[MAX_N]={0},_min,cur;
    book[s]=1;
    for(int i=0;i<MAX_N;i++)book[i]=0;

    for(int i=1;i<=n;i++)
        dis[i]=matrix[s][i];

    for(int i=1;i<n;i++){
        _min=INF;
        for(int j=1;j<=n;j++)
```

```
            if(!book[j]&&dis[j]<_min){
                _min=dis[j];
                cur=j;
            }
        book[cur]=1;
        for(int j=1;j<=n;j++)
            if(!book[j])
                dis[j]=min(dis[cur]+matrix[cur][j],dis[j]);
    }

    for(int i=1;i<=n;++i)
        if(dis[i]==INF) printf("INF ");
        else printf("%d ",dis[i]);

    return;
}
```

11.时间复杂度分析

从伪代码部分可以看出，dij算法主要有两部分组成：

1. 找到最短的出边O(N)
2. 松弛O(N)

所以时间复杂度总和为：**O(O(N)+O(N))*N=O(N^2)**

堆优化

00.算法原理

1. 找到最小出边这一部分使用**邻接表**优化到O(M)，并且**使用堆寻找最小值优化到O(1)**
2. 松弛操作仍然需要**M**次，每次在堆上的修改需要**O(logN)**

所以时间复杂度总和为：**O(1)*N+O(logN)*M=O(MlogN)**

01.模板代码

```
/*
 *h:可以直接在中间插入的堆
 *list:邻接表
 *dis:记录距离
 *cur:当前的点
 *to:去到的点
 *INF:正无穷
 */
void DijkstraHeapList(){
    heap h;
    int dis[MAX_N],cur,to;
    for(int i=0;i<MAX_N;i++)dis[i]=INF;

    h.push((node){s,0});
    dis[s]=0;

    while(!h.empty()){
        cur=h.top().v;
        if(h.book[cur]==-1){
            h.pop();
            continue;
        }
```

```
        h.pop();
        for(int i=head[cur];~i;i=list[i].next){
            to=list[i].to;
            if((h.book[to]!=-1)&&(dis[to]>dis[cur]+list[i].w)){
                dis[to]=dis[cur]+list[i].w;
                h.push((node){to,dis[to]});
            }
        }
    }

    for(int i=1;i<=n;++i)
        if(dis[i]==INF) printf("INF ");
        else printf("%d ",dis[i]);

    return;
}
```

10.Bellman-Ford

00.算法原理

实际上可以知道最短路径中是没有环，这个时候分两种情况

1. 正环：如果有正环，那么去**掉这个环即可缩短**，所以不会有正环。
2. 负环：如果有负环，**一直走这个环即可一直缩短路程**，即不存在最短路，所有不会有负环。

根据这个结论还可知，最短路**最多经过n个点，n-1条边**，否则就有环了。这个结论介绍Bellman-Ford的基础，Bellman-Ford**可以判负环**也是更具这个结论。使用这个结论可以很简单写出Bellman-Ford算法：

1. 外层循环执行n-1次，注意**这里n-1次循环不代表遍历n-1个点**，这是有**n-1次松弛操作。**
2. 内层嵌套一个遍历m条边的循环。若第e条边可以使得：`dis[v[e]]>w[e]+dis[u[e]]`，则使`dis[v[e]]=w[e]+dis[u[e]]`完成松弛。
3. 两层循环完了之后，在进行一轮松弛，**若还可以松弛，则有负环**。

时间复杂度O(nm)，实际上有时不需要跑完n-1次循环，**当本次循环没有松弛操作时**，即可停止。

注意！！！！ 这里的判负环只是判断**源点出发联通的部分**是否有负环，真正的判负环应该创建一个超级源点，**与每个点都相邻，且边权为0**，然后以超级源点执行Bellman-Ford！！！！

01.伪代码

```
init dis

for i:0->n-1
    flag=false
    for e:1->m
        if dis[e]<dis[u[e]]+w[e]
            flag=true
            dis[e]=dis[u[e]]+w[e]
    if !flag stop
    else if i=n-1 NO ANSWER
```

10.模板代码

```
/*
 *list:邻接表
 *dis:记录距离
 *flag:记录是否有松弛操作
 *INF:正无穷
 */
bool Bellman_FordList(){
    int dis[MAX_N];
    bool flag;
    for(int i=0;i<MAX_N;i++)dis[i]=INF;
    dis[s]=0;

    for(int i=0;i<n;i++){
        flag=false;
        for(int e=1;e<=m;e++){
            if(dis[list[e].to]>dis[list[e].from]+list[e].w){
                flag=true;
                dis[list[e].to]=dis[list[e].from]+list[e].w;
            }
        }
        if(!flag) break;
        else if(i==n-1){
            printf("No answer!");
            return true;
        }
    }

    for(int i=1;i<=n;++i)
        if(dis[i]==INF) printf("INF ");
        else printf("%d ",dis[i]);

    return false;
}
```

11.SPFA

00.算法原理

实际上在Bellman-Ford算法中，很多的松弛操作是并不需要的。只有一个点被其他点松弛过，这个点才有可能继续，维护一个队列中有可以引起松弛的点。那么SPFA也可以判负环，只要一个点入队了n次是，即发生了n次松弛，即最短路经过了n条边，所以会有负环。具体来说：

1. 将源点s入队，执行2。
2. 取出对头u，将u相邻的点遍历v，若`dis[v]<dis[u]+S(u->v)`，则`dis[v]=dis[u]+S(u->v)`，并计入v的入队次数。

01.伪代码

```
init dis,cnt,book
que.push s

while not q.empty
    u=q.front
    q.pop
```

```
    for q->v
        if dis[u]<dis[v]+S(u->v)
            dis[u]=dis[v]+S(u->v)
            if not book v
                cnt[v]+1
                if cnt[v]==n
                    NO ANSWER
                q.push v
                book v
```

10.模板代码

```
/*
 *list:邻接表
 *q:队列
 *dis:记录距离
 *book:是否在队列中
 *cnt:记录入队此时
 *cur: 当前的点
 *to:去到的点
 *INF:正无穷
 */
bool SPFAList(){
    queue q;
    int dis[MAX_N],cnt[MAX_N]={0},cur,to;
    memset(book,0,sizeof(book));
    for(int i=0;i<MAX_N;i++)dis[i]=INF;

    q.push(s);
    dis[s]=0;
    book[s]=1;
    cnt[s]=1;

    while(!q.empty()){
        cur=q.front();
        q.pop();
        book[cur]=0;

        for(int i=head[cur];~i;i=list[i].next){
            to=list[i].to;
            if(dis[to]>dis[cur]+list[i].w){
                dis[to]=dis[cur]+list[i].w;
                if(!book[to]){
                    cnt[to]++;
                    if(cnt[to]>=n){
                        printf("No answer!");
                        return false;
                    }
                    q.push(to);
                    book[to]=1;
                }
            }
        }
    }

    for(int i=1;i<=n;++i)
```

```
        if(dis[i]==INF) printf("INF ");
        else printf("%d ",dis[i]);

    return true;
}
```

实际上SPFA**最坏时间复杂度仍然是O(nm)**，但在随机图中表现极为优异。于是就有了卡SPFA的说法

10.卡SPFA