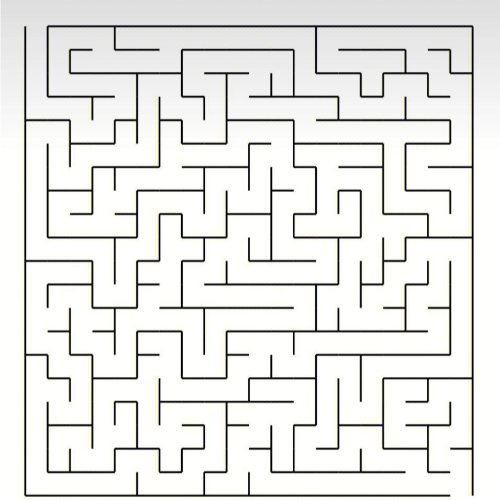
BFS、DFS、拓扑排序与最短路

2023年7月19日

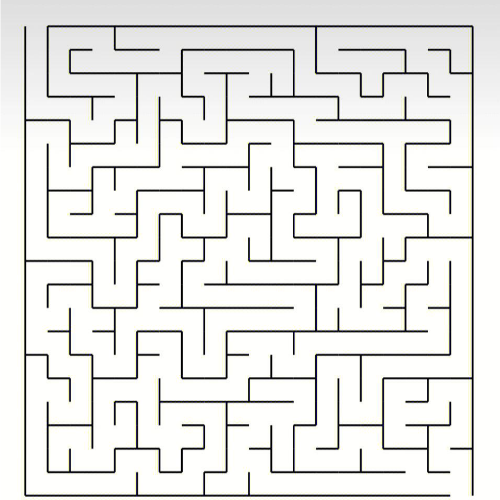
BIT-Epsilon

搜索

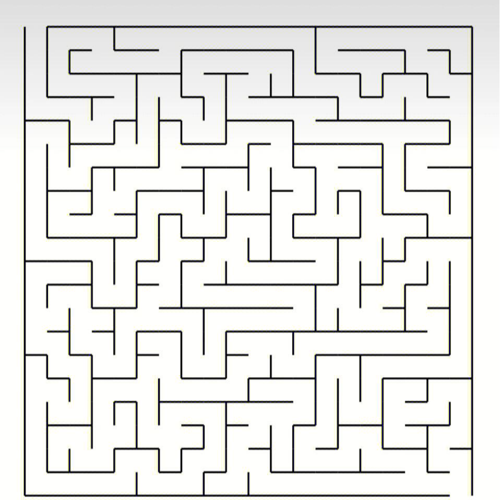
不知道大家有没有走过迷宫？



如果要让计算机程序来走迷宫，应该怎么实现？



如果要让计算机程序来走迷宫，应该怎么实现？



实际上，在计算机中并没有更高明的方法，基本上所有的算法都是基于暴力枚举所有的路径，也就是搜索。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在假设迷宫可以用一个网格图来表示，网格中的S代表该格是起点，G代表该格是终点，#代表该格是墙，无法通行，空格代表该格是空地，可以通行。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

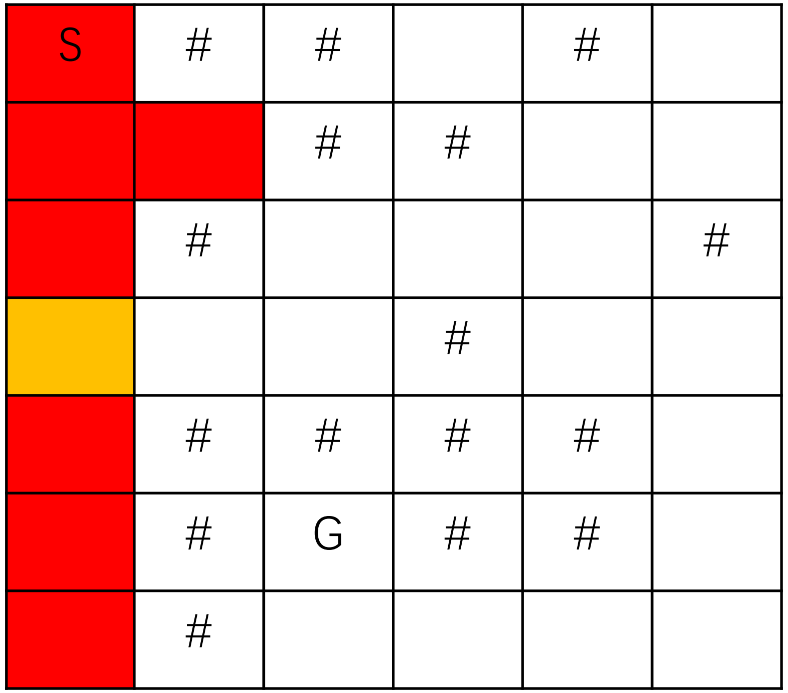
现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。



现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

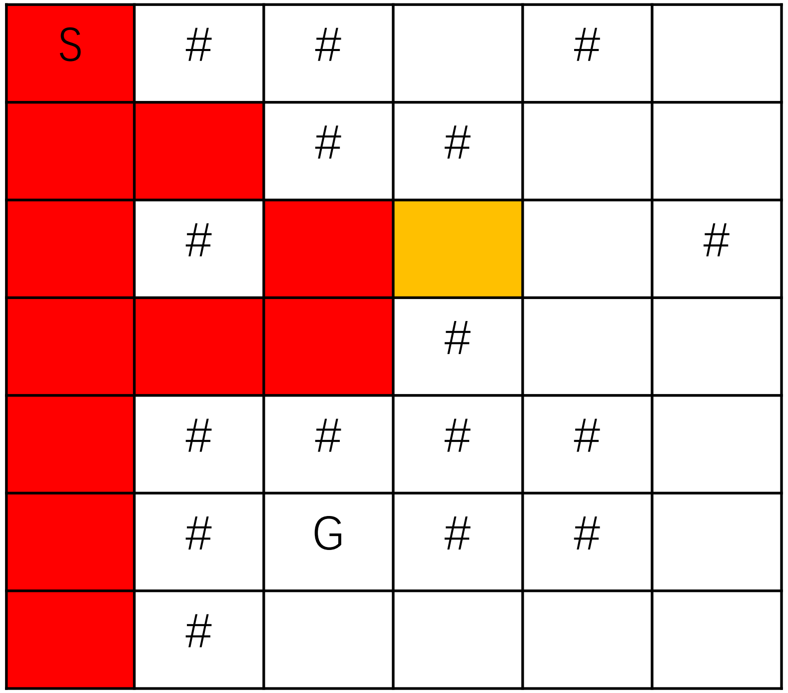
现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

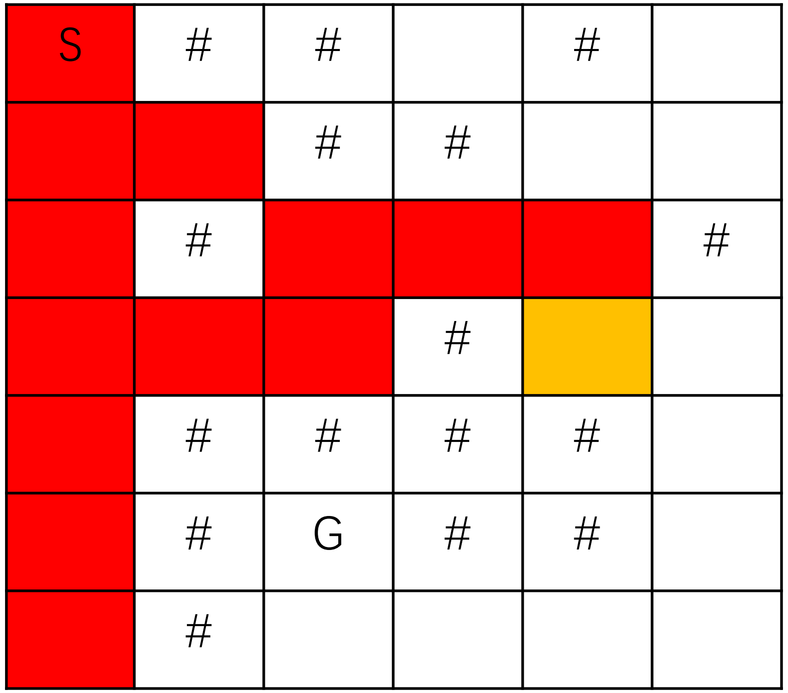
现在我们尝试用我们之前的方式来找一条由起点到终点的路径。



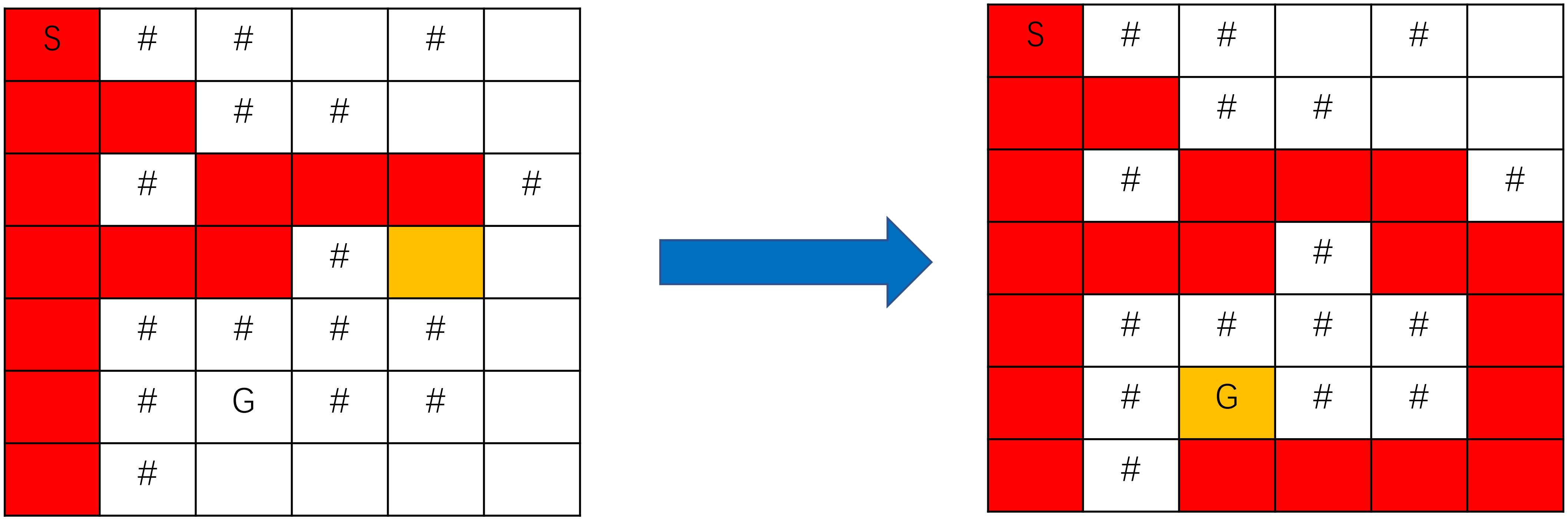
现在我们尝试用我们之前的方式来找一条由起点到终点的路径。

S	#	#		#	
		#	#		
	#				#
			#		
	#	#	#	#	
	#	G	#	#	
	#				

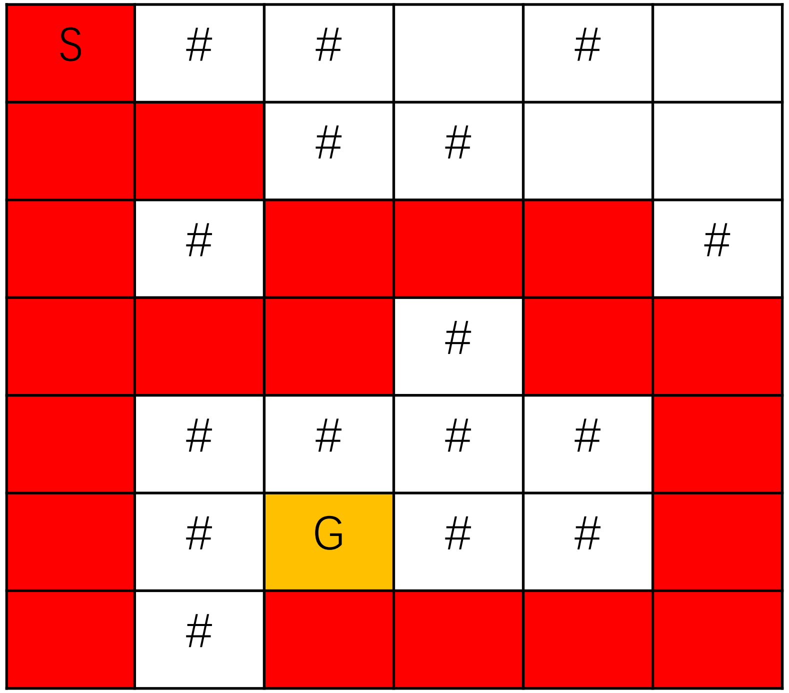
现在我们尝试用我们之前的方式来找一条由起点到终点的路径。



现在我们尝试用我们之前的方式来找一条由起点到终点的路径。



现在我们尝试用我们之前的方式来找一条由起点到终点的路径。



可以看到，我们上述的方法成功找到了一条从起点到终点的路径。
我们在路径的时候，采取的是一种“不撞南墙不回头”的策略，这就是深度优先搜索（dfs）。

深度优先搜索的代码实现

	0	1	2	3	4	5
0	S	#	#		#	
1			#	#		
2		#				#
3				#		
4		#	#	#	#	
5		#	G	#	#	
6		#				

现在假设我们到达了（1，0）点，我们接下去可以走的点有（1，1）点和（2，0）点。我们先走（1，1）点，如果沿着（1，1）点走不到终点，我们在回到（1，0）点后，再往（2，0）点走。

深度优先搜索的代码实现

	0	1	2	3	4	5
0	S	#	#		#	
1			#	#		
2		#				#
3				#		
4		#	#	#	#	
5		#	G	#	#	
6		#				

现在假设我们到达了（1，0）点，我们接下去可以走的点有（1，1）点和（2，0）点。我们先走（1，1）点，如果沿着（1，1）点走不到终点，我们在回到（1，0）点后，再往（2，0）点走。
那么，在（1，1）点上，我们又该怎么走呢？

深度优先搜索的代码实现

	0	1	2	3	4	5
0	S	#	#		#	
1			#	#		
2		#				#
3				#		
4		#	#	#	#	
5		#	G	#	#	
6		#				

现在假设我们到达了（1，0）点，我们接下去可以走的点有（1，1）点和（2，0）点。我们先走（1，1）点，如果沿着（1，1）点走不到终点，我们在回到（1，0）点后，再往（2，0）点走。
那么，在（1，1）点上，我们又该怎么走呢?
不难想到，在每一点上，我们采取的策略都是相同的。

因此我们可以将上述过程写成一个函数，每次走到一个新的点只需要调用这个函数。

深度优先搜索的代码实现

```
//设这个网格图的大小是MxN的
char G[M][N];   //用来记录哪些点是墙
int vis[M][N];  //用来记录哪些位置已经访问过
void dfs(int x, int y, int gx, int gy) {
    vis[x][y] = 1;      //标记当前点已经访问过
    if(x == gx && y == gy) {
        //到达终点，说明找到了一条由起点到终点的路径

        return;
    }

    int nextx, nexty;   //将要走的位置的坐标
    //尝试向上走
    nextx = x - 1;  nexty = y;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N   //保证不出边界
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {  //保证要走的位置还没有访问过
            dfs(nextx, nexty, gx, gy);
    }
    //尝试向左走
    nextx = x;  nexty = y - 1;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
```

```
    //尝试向下走
    nextx = x + 1;  nexty = y;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
    //尝试向右走
    nextx = x;  nexty = y + 1;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
    return;
}
```

深度优先搜索的代码实现

```
//设这个网格图的大小是MxN的
char G[M][N];   //用来记录哪些点是墙
int vis[M][N];  //用来记录哪些位置已经访问过
void dfs(int x, int y, int gx, int gy) {
    vis[x][y] = 1;      //标记当前点已经访问过
    if(x == gx && y == gy) {
        //到达终点，说明找到了一条由起点到终点的路径

        return;
    }

    int nextx, nexty;   //将要走的位置的坐标
    //尝试向上走
    nextx = x - 1;  nexty = y;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N   //保证不出边界
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {  //保证要走的位置还没有访问过
            dfs(nextx, nexty, gx, gy);
    }
    //尝试向左走
    nextx = x;  nexty = y - 1;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
```

```
    //尝试向下走
    nextx = x + 1;  nexty = y;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
    //尝试向右走
    nextx = x;  nexty = y + 1;
    if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
     && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
    }
    return;
}
```

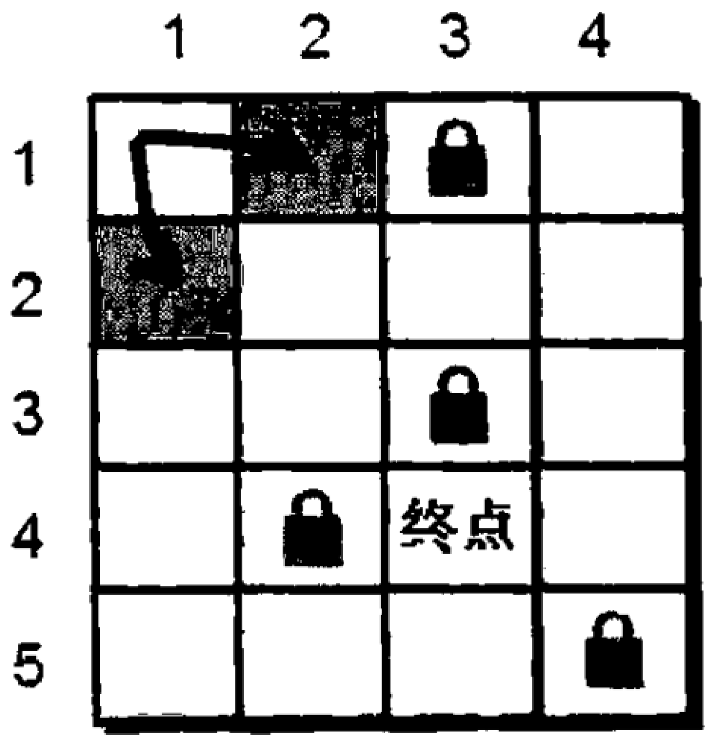
可以发现，向四个方向走的代码块中，除了坐标的具体值不同，其他都是相同的，因此我们可以简化上述代码。

深度优先搜索的代码实现

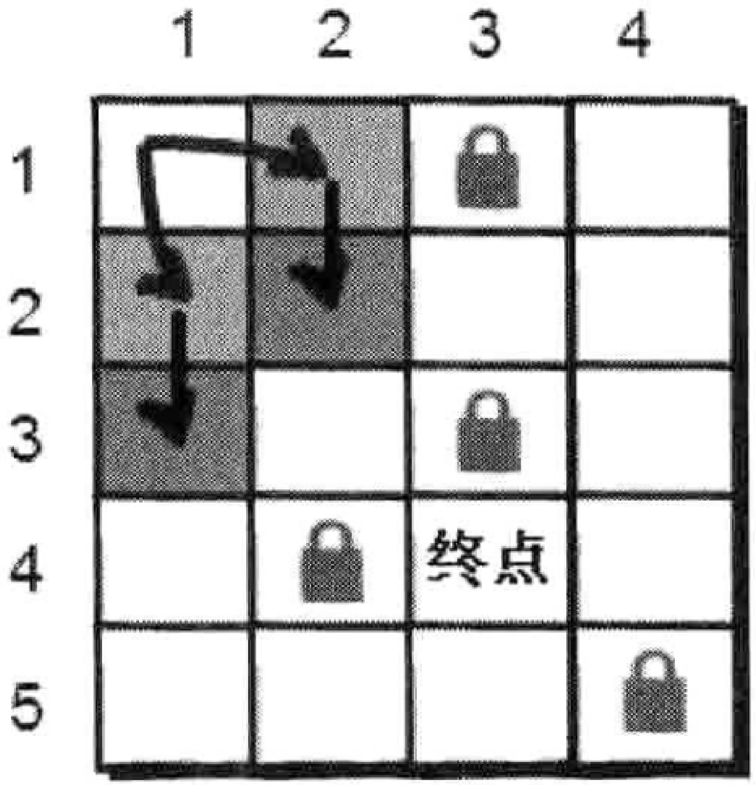
```
//设这个网格图的大小是MxN的
char G[M][N];    //用来记录哪些点是墙
int vis[M][N];   //用来记录哪些位置已经访问过
const int dx[4] = {0, 1, 0, -1}, dy[4] = {1, 0, -1, 0};
void dfs(int x, int y, int gx, int gy) {
    vis[x][y] = 1;        //标记当前点已经访问过
    if(x == gx && y == gy) {
        //到达终点，说明找到了一条由起点到终点的路径

        return;
    }

    int nextx, nexty;    //将要走的位置的坐标
    for(int d = 0; d < 4; d++) {
        int nextx = x + dx[d], nexty = y + dy[d];
        //随着d的遍历，我们遍历到了四个方向的坐标
        if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
         && !vis[nextx][nexty] && G[nextx][nexty] != '#')  {
            dfs(nextx, nexty, gx, gy);
        }
    }
    return;
}
```

上面我们讲了深度优先搜索（dfs），它的特点是“一条路走到黑”。
现在我们来介绍另外一种搜索方式：广度优先搜索（bfs），它是通过“层层拓展”的方式进行遍历的。

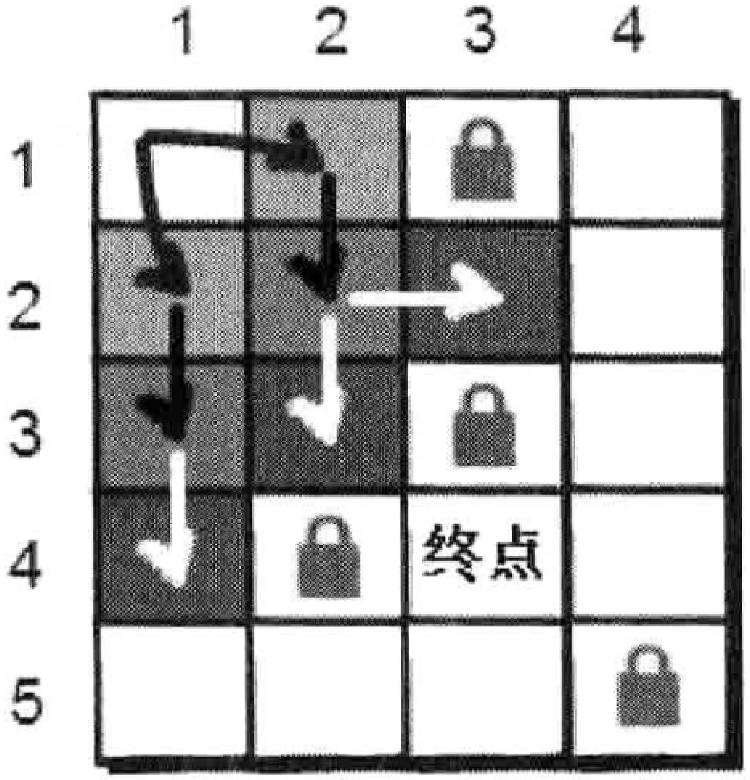
所谓的“层层拓展”指的是先遍历所有1步可以到达的点，再遍历所有2步可以到达的点……如此往复，直到遍历到终点。



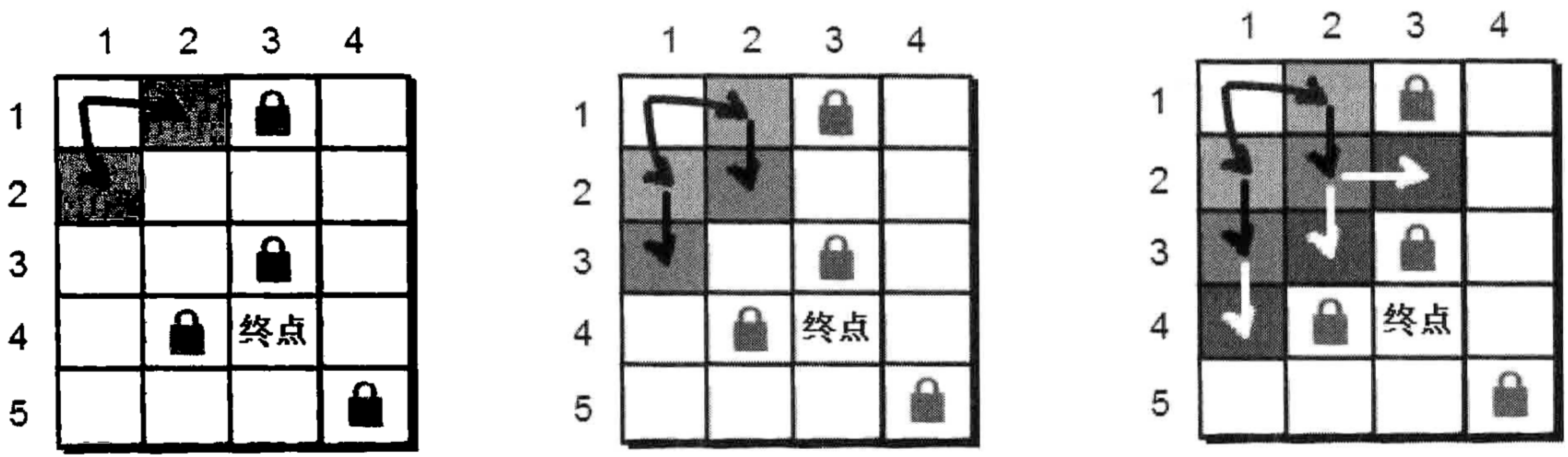
假设起点在（1，1）点。
可以一步到达的点有（1，2）和（2，1）。



从可以1步到达的点进行拓展，得到可以2步到达的点：（3，1）和（2，2）。



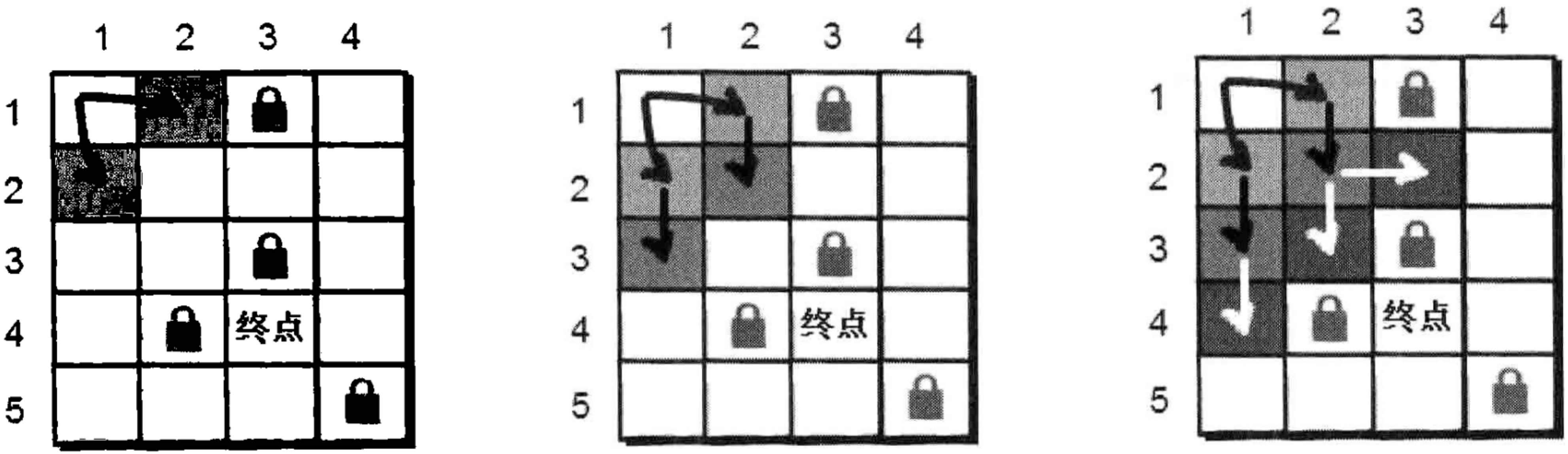
如此重复，直到拓展到终点。
当然，在这个过程中，每个点至多被遍历到一遍，需要一个二维数组记录每个点的访问情况。



上述算法可以用一个队列来辅助实现：

初始时，队列中只有起点（1，1）。

队列： （1，1）

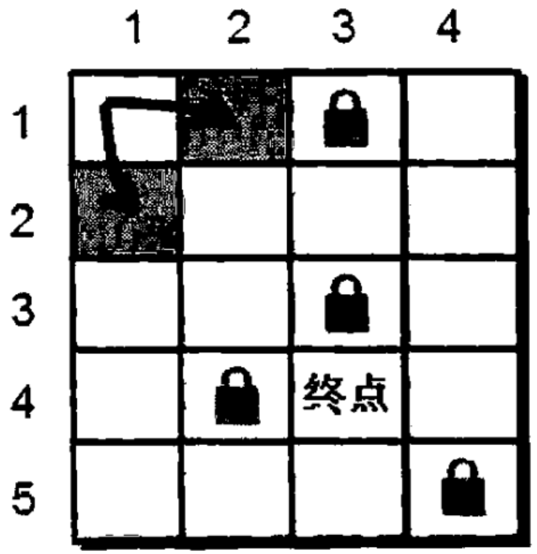


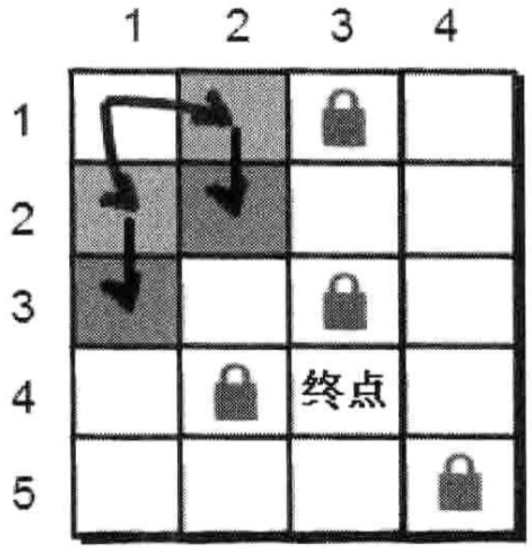
上述算法可以用一个队列来辅助实现：

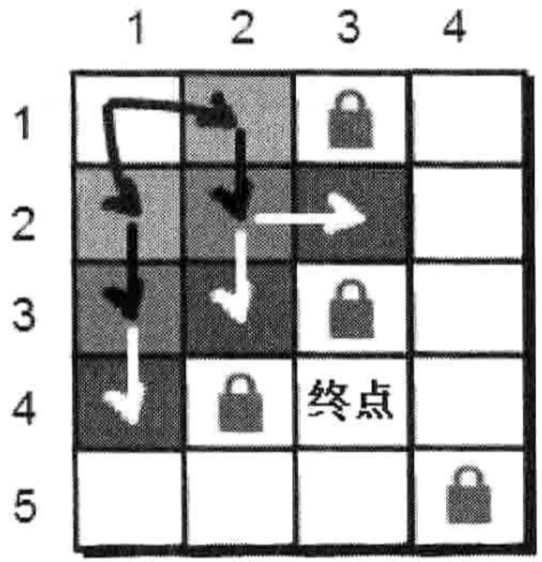
由队首（1，1）拓展到（1，2）和（2，1），将它们加入队列，并将（1，1）出队

队列：

(1，2)	(2，1)





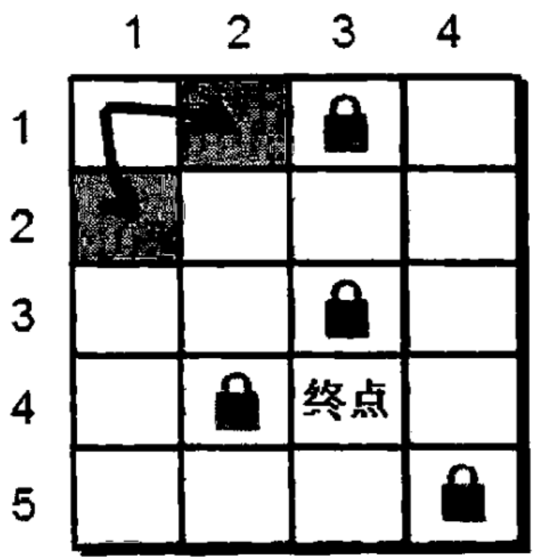


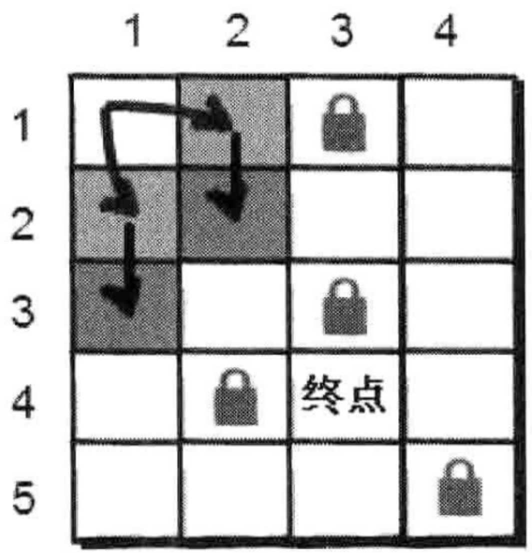
上述算法可以用一个队列来辅助实现:

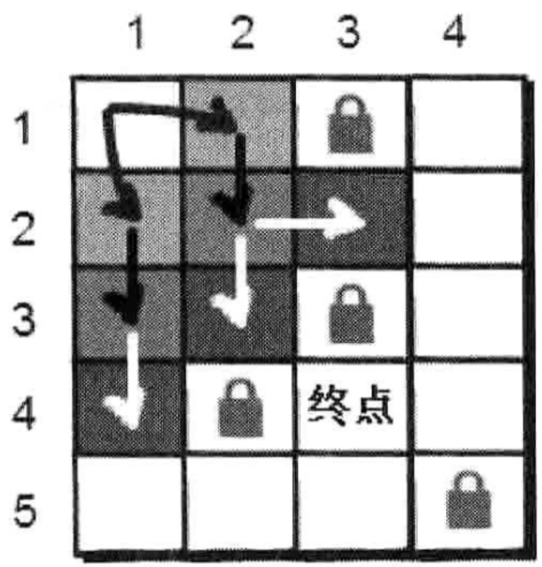
由队首（1，2）拓展到（2，2），将它们加入队列，并将队首（1，2）出队

队列:

(2，1)	(2，2)







上述算法可以用一个队列来辅助实现：

如此反复，直到遍历到终点或队列为空（不存在从起点到终点的路径）

广度优先搜索的代码实现

```
const int dx[] = {0, 1, 0, -1}, dy[] = {1, 0, -1, 0};
struct node {   //用于表示点坐标的结构体(可以直接用pair)
    int x, y;
};
int vis[M][N];  //用于标记每个点的访问情况
```

```
void bfs(int x, int y, int gx, int gy) {
    queue<node> q;  //队列
    q.push((node){x, y});
    vis[x][y] = 1;
    while(!q.empty()) {
        int xx = q.front().x, yy = q.front().y; //取出队首，并用其来拓展
        q.pop();
        if(xx == gx && yy == gy)    {
            //遍历到终点

            return;
        }

        for(int d = 0; d < 4; d++) {
            int nextx = xx + dx[d], nexty = yy + dy[x]; //尝试向四个方向拓展
            if(nextx >= 0 && nextx < M && nexty >= 0 && nexty < N
             && !vis[nextx][nexty] && G[nextx][nexty] != '#') {
                vis[nextx][nexty] = 1;
                q.push((node){nextx, nexty});
            }
        }
    }
    return;
}
```

广度优先搜索的其他应用

1. 求起点到其余点的最短距离。
   我们在广度优先搜索的过程中进行的是“层层拓展”：先拓展第1层：可以1步到达的点，再由第1层拓展到第2层：可以2步到达的点。从另一角度来讲，第1层中被拓展到的点到起点的距离就是1。
   更一般地，在第k层中被拓展到的点到起点的最短距离就是k。
2. 求联通块
   将某个点作为起点进行bfs，bfs过程中访问到的所有点就是与起点联通的所有点。

全排列问题
洛谷 P1706

题目描述

复制Markdown 展开

按照字典序输出自然数 1 到 n 所有不重复的排列，即 n 的全排列，要求所产生的任一数字序列中不允许出现重复的数字。

输入格式

一个整数 n。

输出格式

由 $1 \sim n$ 组成的所有不重复的数字序列，每行一个序列。

每个数字保留 5 个场宽。

输入输出样例

输入 #1 复制

```
3
```

输出 #1 复制

```
    1    2    3
    1    3    2
    2    1    3
    2    3    1
    3    1    2
    3    2    1
```

说明/提示

$1 \leq n \leq 9$。

题解
洛谷 P1706

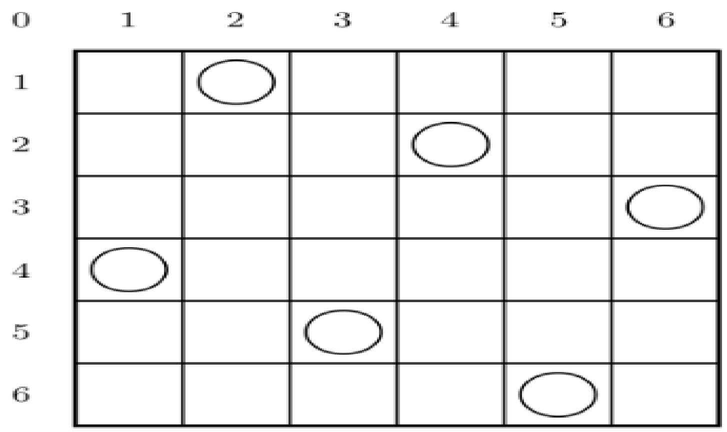
```
#include<bits/stdc++.h>
using namespace std;
int n;
int ans[15];//保存当前的方案
int use[15];//表示每个数是否被用过
void dfs(int x){//X表示当前搜索到那个数
    if(x>n){//如果N位都搜索完了，就输出方案并返回
        for(int i=1;i<=n;i++)
            printf("% 5d",ans[i]);//输出方案
        puts("");
        return;
    }
    for(int i=1;i<=n;i++)//从小到大枚举
    if(!use[i]){//判断这个数是否用过
        ans[x]=i;//保存到方案中
        use[i]=1;//标记这个数被使用了
        dfs(x+1);//进行下一步搜索
        use[i]=0;//撤销标记
    }
}
int main()
{

    scanf("%d",&n);//输入
    dfs(1);//从第一个数开始搜索；
}
```

八皇后问题
洛谷 P1219

题目描述

复制Markdown 展开

一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行、每列有且只有一个，每条对角线（包括两条主对角线的所有平行线）上至多有一个棋子。



洛谷

上面的布局可以用序列 2 4 6 1 3 5 来描述，第 i 个数字表示在第 i 行的相应位置有一个棋子，如下：

行号 1 2 3 4 5 6

列号 2 4 6 1 3 5

这只是棋子放置的一个解。请编一个程序找出所有棋子放置的解。
并把它们以上面的序列方法输出，解按字典顺序排列。
请输出前 3 个解。最后一行是解的总个数。

输入格式

一行一个正整数 n，表示棋盘是 $n \times n$ 大小的。

输出格式

前三行为前三个解，每个解的两个数字之间用一个空格隔开。第四行只有一个数字，表示解的总数。

输入输出样例

输入 #1 复制

```
6
```

输出 #1 复制

```
2 4 6 1 3 5
3 6 2 5 1 4
4 1 5 2 6 3
4
```

八皇后 题解
洛谷 P1219

```
#include<cstdio>
#include<iostream>
using namespace std;
int ans[14],check[3][28]={0},sum=0,n;
void eq(int line)
{
    if(line>n)
    {
        sum++;
        if(sum>3) return;
        else
        {
            for(int i=1;i<=n;i++) printf("%d ",ans[i]);
            printf("\n");
            return;
        }
    }
    for(int i=1;i<=n;i++)
    {
        if((!check[0][i])&&(!check[1][line+i])&&(!check[2][line-i+n]))
        {
            ans[line]=i;
            check[0][i]=1; check[1][line+i]=1; check[2][line-i+n]=1;
            eq(line+1);
            check[0][i]=0; check[1][line+i]=0; check[2][line-i+n]=0;
        }
    }
}
int main()
{
    scanf("%d",&n);
    eq(1);
    printf("%d",sum);
    return 0;
}
```

Meet in the middle
中间相遇法（译名不确定）

Meet in the middle算法的主要思想是将整个搜索过程分成两半，分别搜索，最后将两半的结果合并。

暴力搜索的复杂度往往是指数级的，而改用 meet in the middle 算法后复杂度的指数可以减半，即让复杂度从 O(a^b) 降到 O($a^{b/2}$)。

开关灯问题
P2963

题目描述

给出一张 n 个点 m 条边的无向图，每个点的初始状态都为 0。

你可以操作任意一个点，操作结束后该点以及所有与该点相邻的点的状态都会改变，由 0 变成 1 或由 1 变成 0。

你需要求出最少的操作次数，使得在所有操作完成之后所有 n 个点的状态都是 1。

输入格式

第一行两个整数 n, m。

之后 m 行，每行两个整数 a, b，表示在点 a, b 之间有一条边。

输出格式

一行一个整数，表示最少需要的操作次数。

输入输出样例

输入 #1 复制

```
5 6
1 2
1 3
4 2
3 4
2 5
5 3
```

输出 #1 复制

```
3
```

说明/提示

对于 100% 的数据，$1 \le n \le 35, 1 \le m \le 595, 1 \le a, b \le n$。

开关灯 题解
P2963

meet in middle 就是让我们先找一半的状态，也就是找出只使用编号为 1 到 mid 的开关能够到达的状态，再找出只使用另一半开关能到达的状态。如果前半段和后半段开启的灯互补，将这两段合并起来就得到了一种将所有灯打开的方案。具体实现时，可以把前半段的状态以及达到每种状态的最少按开关次数存储在 map 里面，搜索后半段时，每搜出一种方案，就把它与互补的第一段方案合并来更新答案。

复杂度O($(n+m)\ 2^{n/2}$)

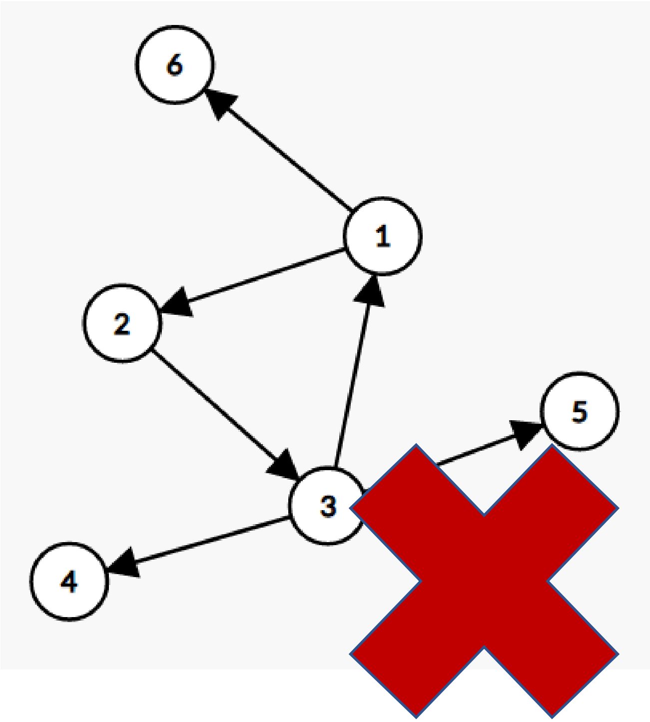
拓扑排序

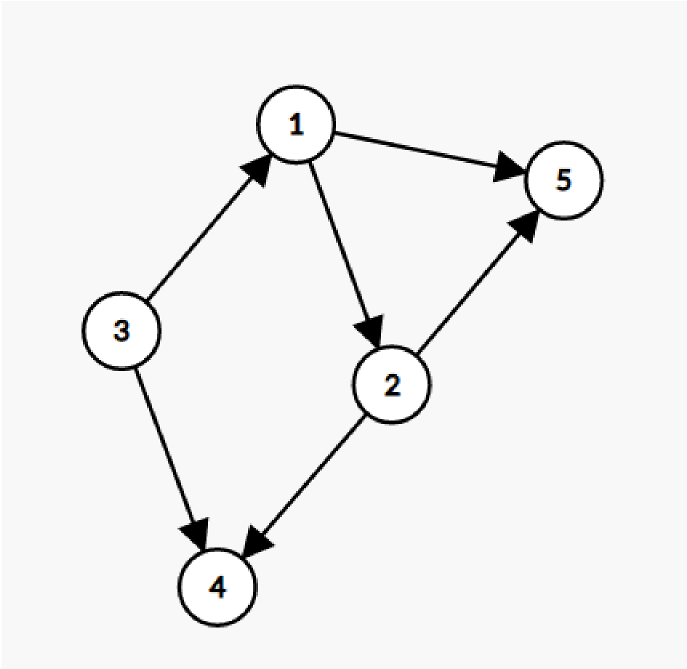
问题引入

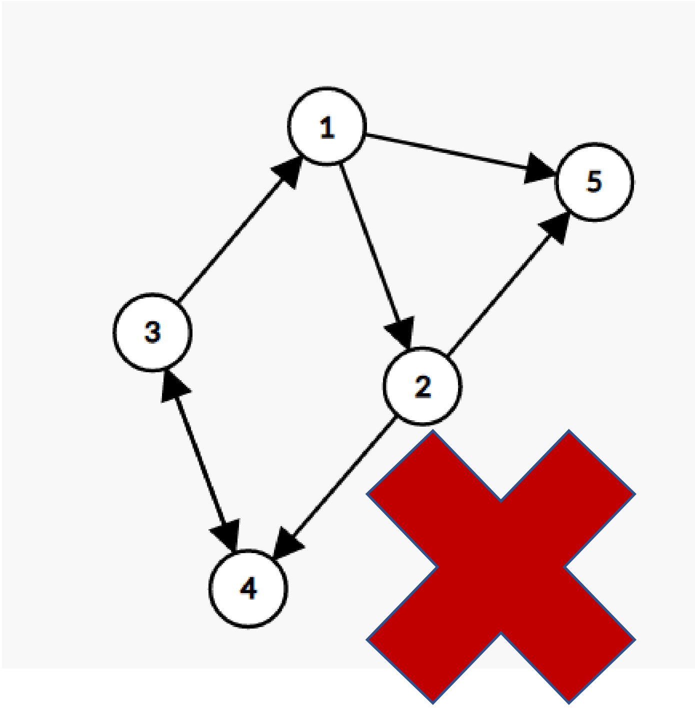
- 工程上，一些工程需要子工程完成以后才能开工；
- 要获得道具需要先完成系列任务；
- 一些课程有先修课程要求；

DAG

- 有向无环图
- 无自环，可以有重边
- 可以不连通



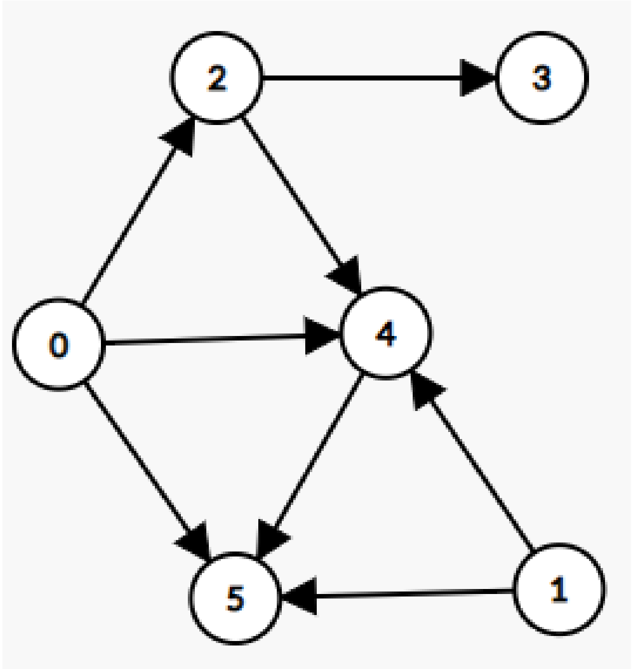




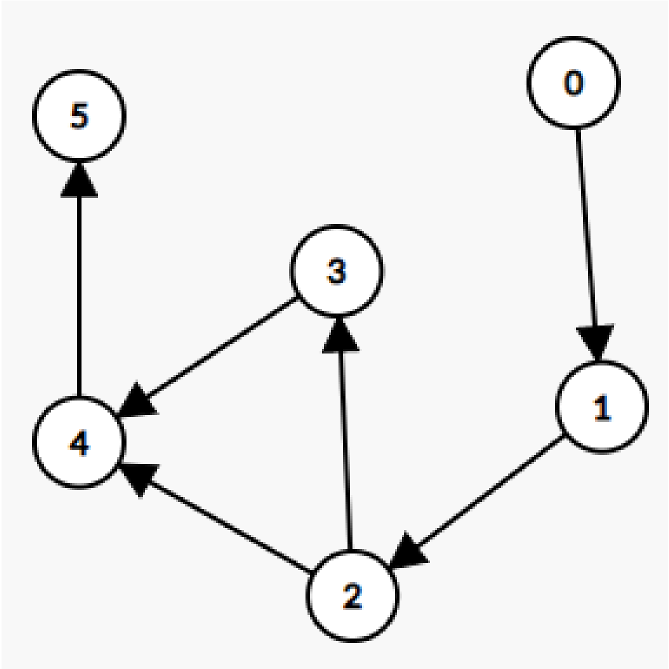
拓扑序

- 将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边<u,v>∈E(G)，则u在线性序列中出现在v之前。
- 对于线性序列A，$\forall A_i \neq A_j \in A, \exists < A_i, A_j > \in E(G) \Rightarrow i < j$，称A为G的一个拓扑序
- 对于线性序列A，$\forall A_i \neq A_j \in A, j < i \Rightarrow \neg\exists < A_i, A_j > \in E(G)$，称A为G的一个拓扑序
- 只有有向无环图才存在拓扑序。
- 有向无环图一定存在拓扑序。
- 一张图的拓扑序一般不唯一。
- 求DAG中的一个拓扑序，就是拓扑排序的过程。

023145



012345



拓扑排序

- 没有入边（入度为0）的点，其一定可以排在最前面
- 这些没有入边的点之间也一定没有边，故可以任意顺序排列。
- 若有点的入边只与上述点相邻，则该点可以紧挨上述点排在后面。
- 以此类推。

若序列中u在v之前，则不存在v->u的边

1. 将没有入边的点加入队列。
2. 将队头u出队，删除u的所有出边，若有点在删除之后入度为0，将其加入队列
3. 重复2，直至队空
4. 出队序列即为拓扑序。

代码

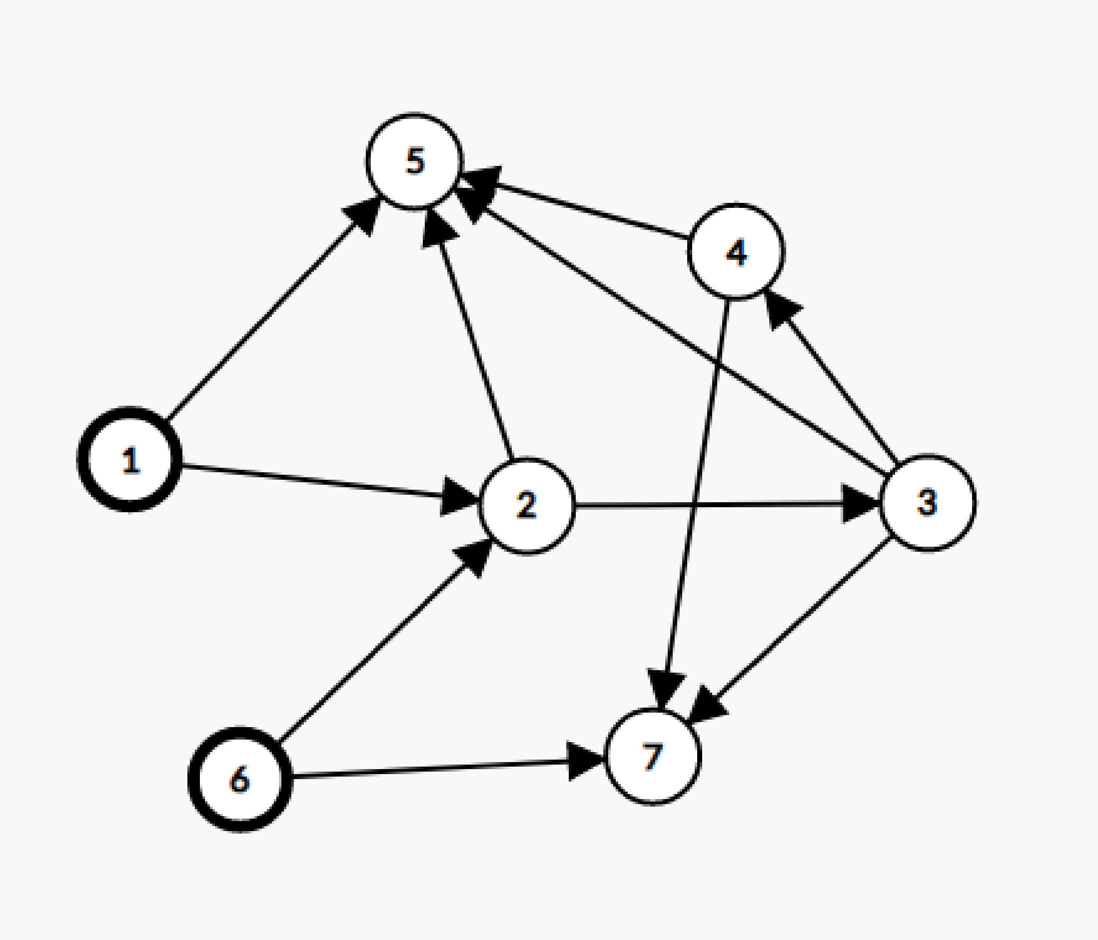
给定点数n，边数m，所有边；求拓扑序

不需要进行标记，因为对于数组in，每个点只会有一次变到0。

时间复杂度O(n+m)

```
void bfs(){
    queue<int>q;
    For(i,1,n)if(!in[i])q.push(i);
    while(!q.empty()){
        int u=q.front();q.pop();
        a[++tot]=u;
        rep(i,u,e,head){
            int v=e[i].v;
            --in[v];
            if(!in[v])q.push(v);
        }
    }
}
```

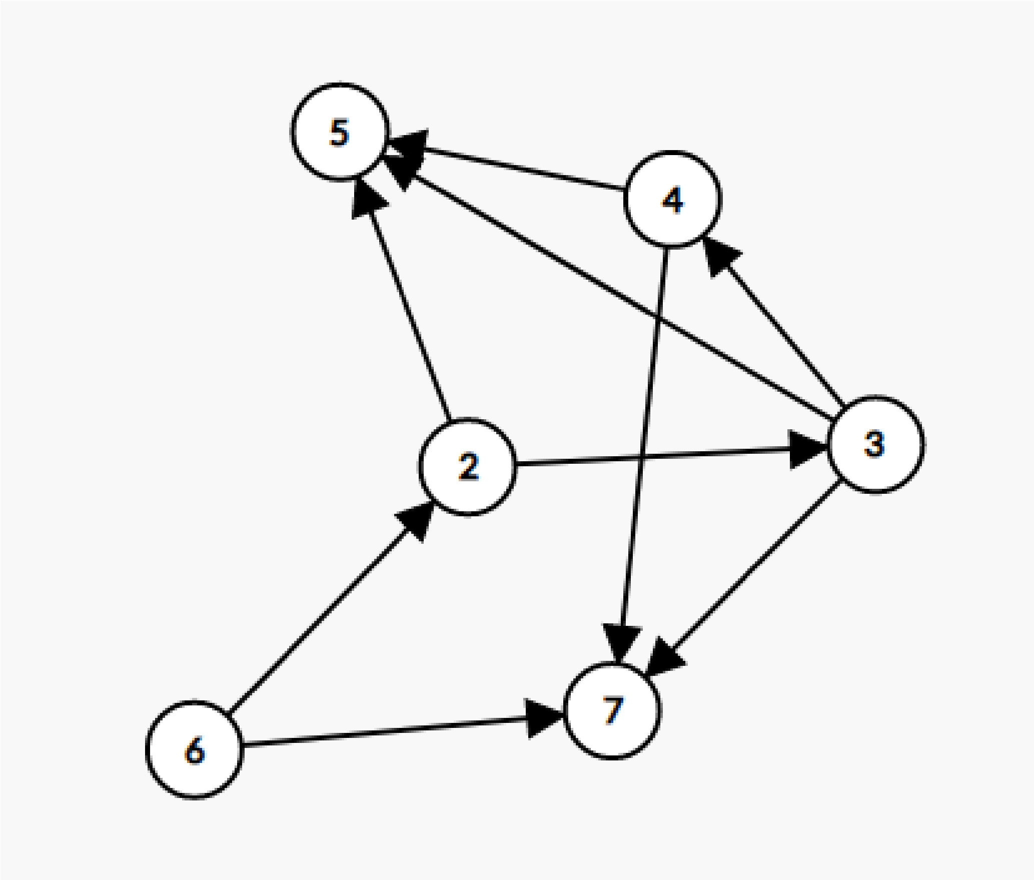
样例



1、6入队

队列:
1 6
拓扑序:

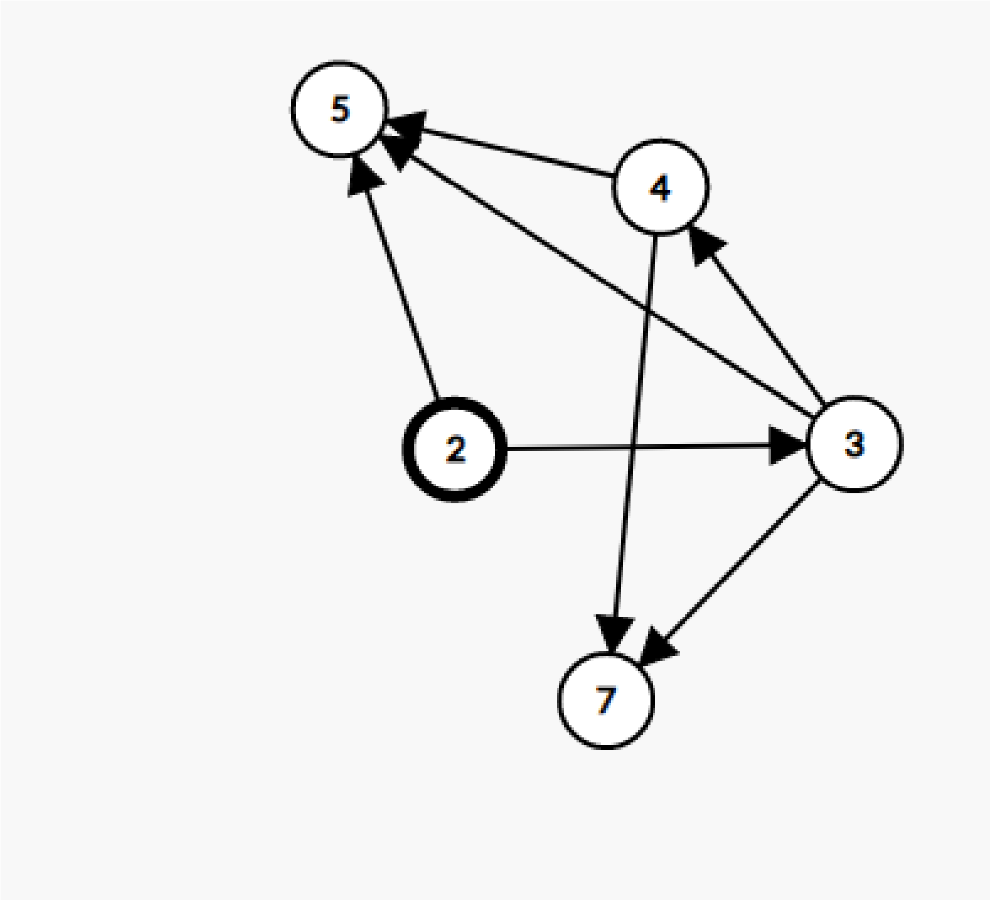
样例



1出队
删除<1,2>,<1,3>

队列:
6
拓扑序:
1

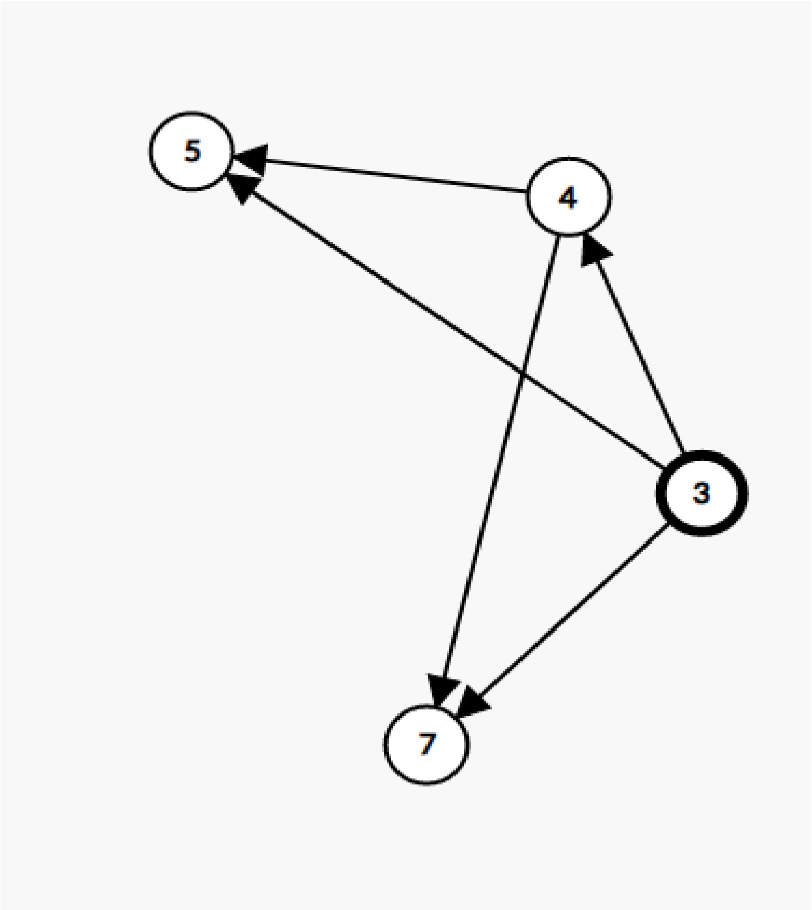
样例



6出队
删除<6,2>，此时2入度为0，2入队
删除<6,7>

队列:
2
拓扑序:
1 6

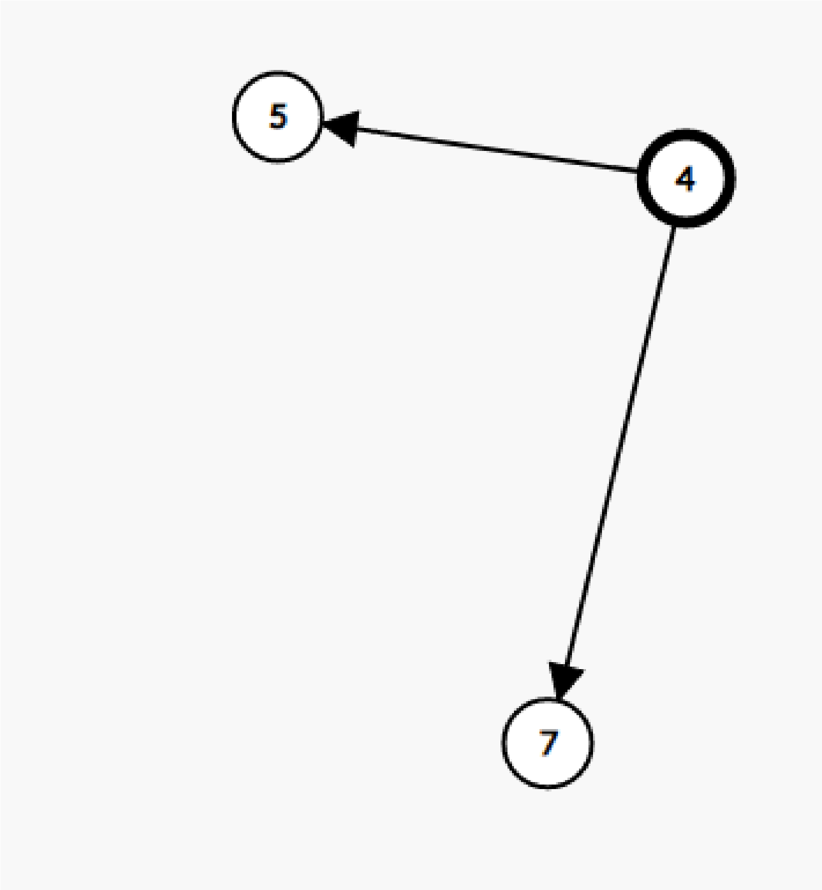
样例



2出队
删除<2,5>
删除<2,3>，3入度为0，3入队

队列:
3
拓扑序:
1 6 2

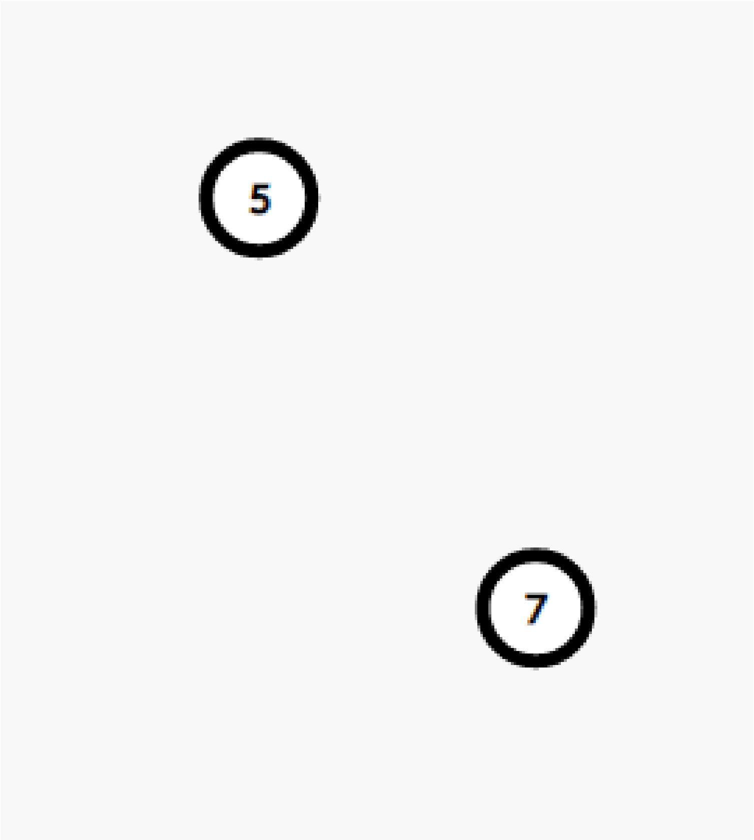
样例



3出队
删除<3,4>，4入度为0，4入队
删除<3,5>

队列：
4
拓扑序：
1 6 2 3

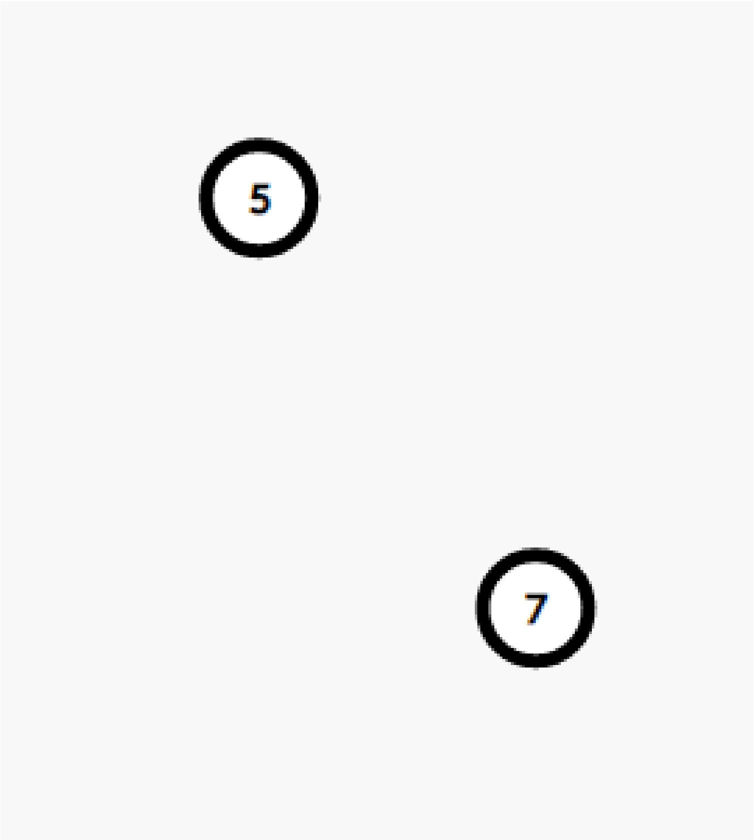
样例



4出队
删除<4,7>，7入度为0，7入队
删除<4,5>，5入度为0，5入队

队列:
7 5
拓扑序:
1 6 2 3 4

样例



7 5出队

队列:

拓扑序:
1 6 2 3 4 7 5

例题

- 食物链
- 给定食物网上的捕食关系，求最大食物链条数

- 声明一个数组f，f[u]表示从生产者开始，以u结束的食物链条数
- 按拓扑序更新答案。

多源最短路

多源最短路径

- 图G中有n个点，m条边，每条边有边权。
- 求任意两点间的最短路径。

floyd

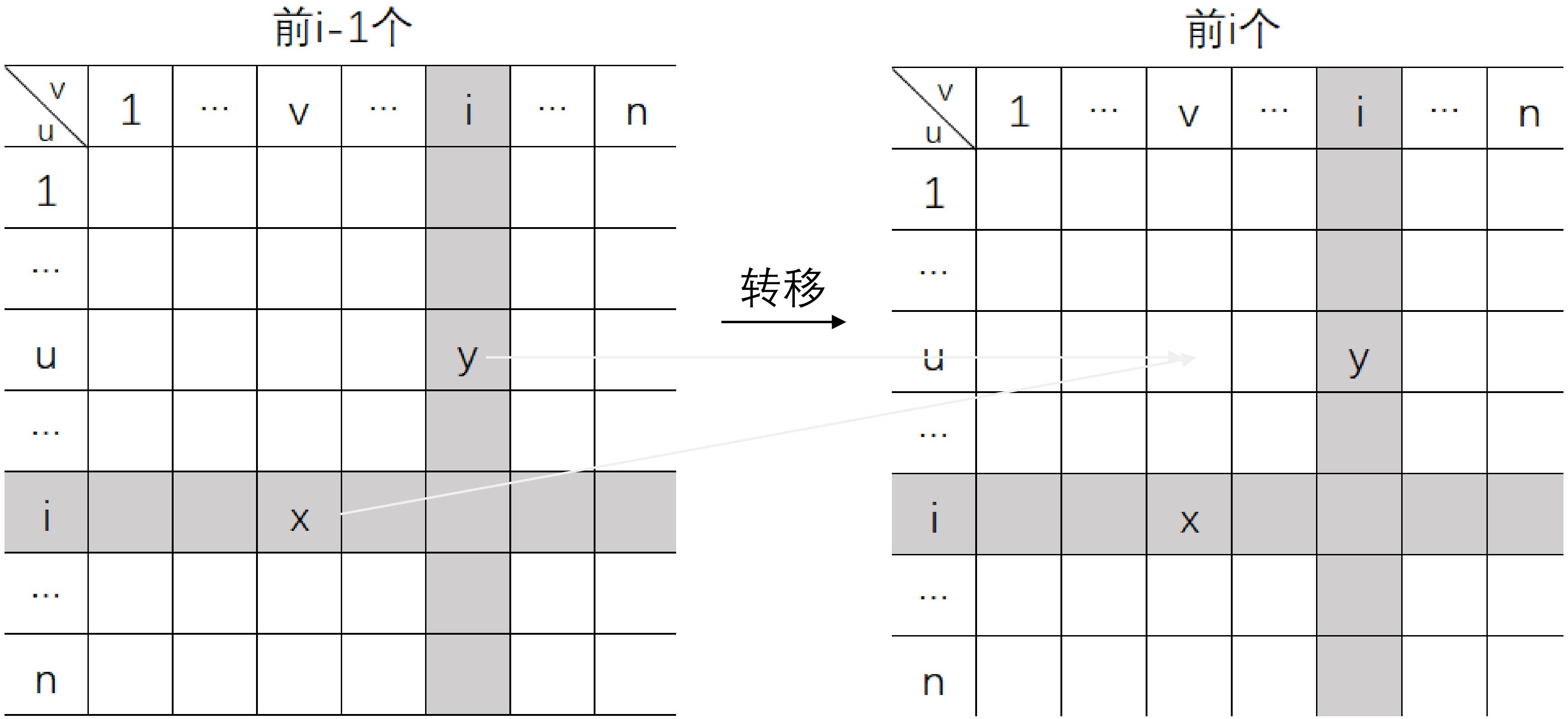
- 一条从u到v的最短路径，它要么是直接从u到v连的边，要么是u经过某些节点后到v。
- 可以声明一个数组，dp[i][u][v] 含义为只经过前i个点（或直达），u到v的最短路径长度。
- 这时转移就很方便了，dp[i][u][v]的值:
  - 要么是u到v经过前i-1个点的最短路径长度。
  - 要么是u到v中间一定经过第i个点的最短长度。
  - 即dp[i] [u][v] =min(dp[i-1][u][v],dp[i-1][u][i]+dp[i-1][i][v])

floyd

- 这样我们建立了从i-1到i的转移方程。
- 外层循环i，内层循环u和v即可得到任意两点间最短路径长dp[n][u][v]。
- 时间复杂度$O(n^3)$
- 空间复杂度$O(n^3)$

floyd

- 事实上，这里空间复杂度可以进一步压缩到$O(n^2)$。



代码

```
For(k,1,n){
    For(i,1,n)
        For(j,1,n)
            dp[i][j]=Min(dp[i][j],dp[i][k]+dp[k][j]);
}
```

- 细节：初始化、重边、负权？

单源最短路

单源最短路

从某个固定起点（源点）开始，到其他所有点的最短路径。

用dis[v]表示从源点到v的最短路径长度。

- Bellman-Ford、SPFA
- Dijkstra

Bellman–Ford

一种基于松弛操作的最短路算法。

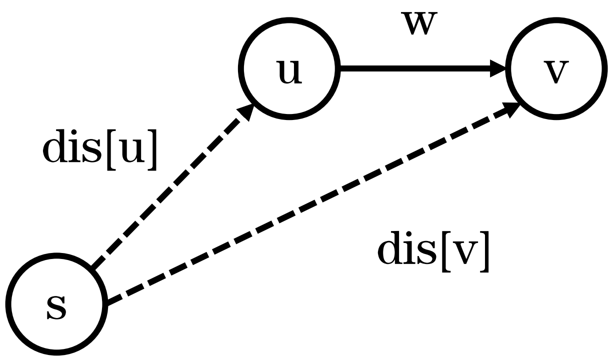
时间复杂度：O(nm)
空间复杂度：O(m)（邻接表）、O(n^2)（邻接矩阵）

优点：可以处理负权图、可以处理有边数限制的最短路（走不超过k条边的最短路）。
缺点：复杂度高。

Bellman–Ford

松弛：对于一条边<u,v,w>，若满足dis[v]>dis[u]+w，则说明存在一条更短的路径，所以更新dis[v]=dis[u]+w。

松弛完后，v的最短路径上多了一条边。



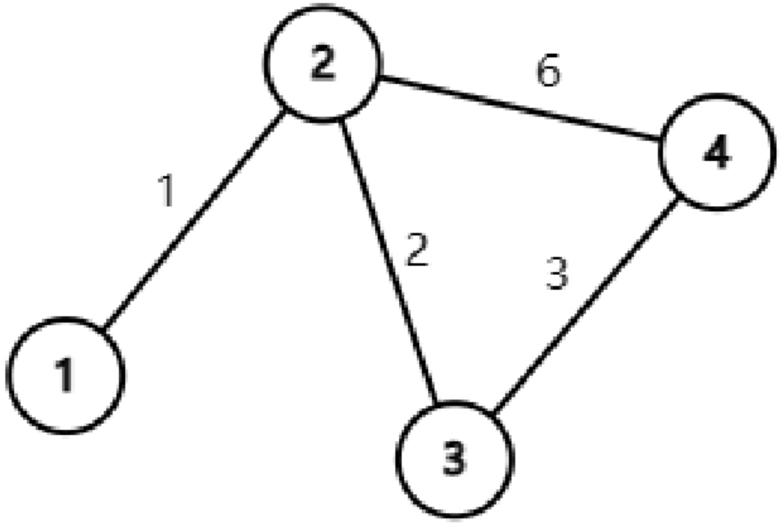
Bellman–Ford

算法流程：

1. 初始化dis数组，dis[s]=0，其他均为INF。
2. 遍历所有边，进行松弛操作。
3. 重复2操作，直到不能进行松弛操作为止。

每次松弛都会让最短路上的边数+1，而在没有负环的情况下，最短路上最多有n-1条边，初始最短路上边数为0，所以至多进行n轮松弛操作，所以复杂度为O(nm)。

Bellman–Ford



dis

0

1	2	3	4
0	INF	INF	INF

1

1	2	3	4
0	1	INF	INF

2

1	2	3	4
0	1	3	7

3

1	2	3	4
0	1	3	6

4

1	2	3	4
0	1	3	6

Bellman–Ford

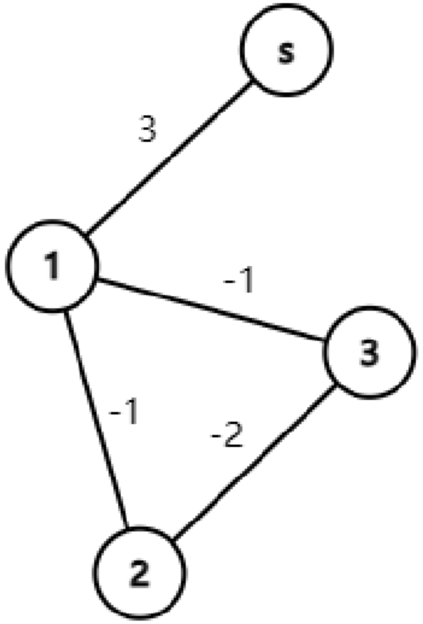
有边数限制的最短路 参考代码：

```
void bellman_ford(int s, int k) { //s表示源点，k表示限制的边的数目，限制k条边就要做k轮松弛
    memset(dis, 0x3f, sizeof dis);
    dis[s] = 0; //初始化dis数组
    while(k--) {
        //为了避免出现串联效应（一轮松弛中某些点更新后又再次更新其他点），需要用另外一个数组backup备份前一次操作后的结果。
        //举个例子，1->2->3，1->2边权为1，2->3边权为2。
        //第1次松弛后的结果应该是{0, 1, INF}。
        //但若不备份，那么在第一次松弛过程中，dis[3]会被更新成dis[2]+2。
        for(int i = 1; i <= n; ++i) backup[i] = dis[i];
        for(int i = 1; i <= m; ++i)
            if(dis[e[i].v] > backup[e[i].u] + e[i].w)
                dis[e[i].v] = backup[e[i].u] + e[i].w;
    }
}
```

Bellman–Ford

如果存在负环，那么会无限进行松弛操作，不存在最短路。
所以，若进行了n轮松弛后仍然可以再次进行松弛，则存在负环。

可以判负环。



SPFA

队列优化的Bellman–Ford算法。

在Bellman–Ford算法中，有些点不会被松弛，导致浪费。

只有一个点在上一轮被成功松弛时，它才可能在这一轮成功松弛与它相连的点。
基于这样的思想，SPFA将所有被成功松弛的点储存在队列中，是Bellman–Ford算法的一种优化。

时间复杂度：一般O(km)（k为常数），最坏O(nm)
空间复杂度：O(m)（邻接表）、O(n^2)（邻接矩阵）

优点：可以处理负权图，比Dijkstra好写一点。
缺点：容易被卡。

SPFA

算法流程：

1. 初始化dis数组，dis[s]=0，其他均为INF。
2. 将源点s加入队列，同时标记s为在队列中，其他均为不在队列中。
3. 重复以下操作，直到队列为空：
   a. 将队首u出队，标记u为不在队列中。
   b. 枚举所有u为起点的边<u,v,w>，若可以松弛，则看v是否被标记，若被未标记，则将v入队。

SPFA　　最短路 参考代码：

```
void spfa(int s) {
    memset(dis, 0x3f, sizeof dis);
    dis[s] = 0; //初始化dis数组。
    queue<int> q;
    q.push(s), flag[s] = true; //将源点s加入队列中并标记它在队列中。
    while(!q.empty()) {
        int x = q.front(); q.pop();
        flag[x] = false; //取队首，并标记它已经不在队列中。
        for(int i = fr[x], y; i; i = nxt[i]) { //枚举与x相邻的边。
            y = to[i];
            if(dis[y] > dis[x] + w[i]) { //能够松弛y。
                dis[y] = dis[x] + w[i];
                if(!flag[y]) q.push(y), flag[y] = true; //若y不在队列中，则将y加入队列中
并标记一下。
            }
        }
    }
}
```

SPFA

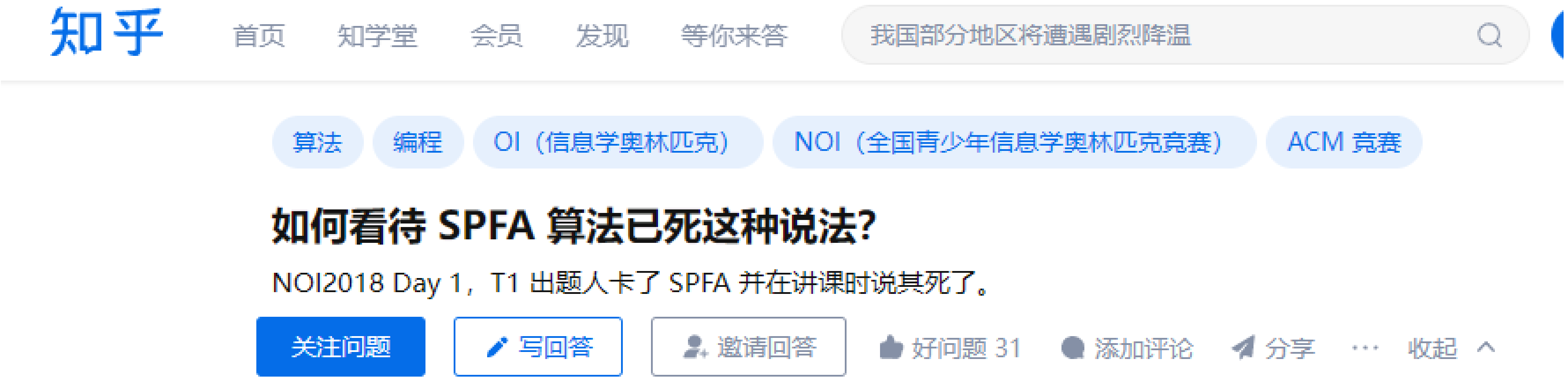
同样，它也可以判负环，基本思路是：

用一个cnt数组记录点被松弛的次数，如果存在cnt[x]>n，那么存在负环

SPFA　　判负环 参考代码：

```
bool spfa(int s) {
    memset(dis, 0x3f, sizeof dis);
    dis[s] = 0;
    queue<int> q;
    q.push(s), flag[s] = true;
    while(!q.empty()) {
        int x = q.front(); q.pop();
        flag[x] = false;
        for(int i = fr[x], y; i; i = nxt[i]) {
            y = to[i];
            if(dis[y] > dis[x] + w[i]) {
                ++cnt[y];
                if(cnt[y] > n) return true; //用cnt数组记录每个点被松弛的次数，如果被松弛了超过n次，则说明有负环。
                dis[y] = dis[x] + w[i];
                if(!flag[y]) q.push(y), flag[y] = true;
            }
        }
    }
    return false;
}
```

SPFA



https://www.zhihu.com/question/292283275

SPFA非常容易被卡

Dijkstra

朴素的Dijkstra：
时间复杂度：O(n^2)
空间复杂度：O(n^2)

堆优化的Dijkstra：
时间复杂度：O((n+m)logn)
空间复杂度：O(m)

优点：效率稳定，不会被卡，基本可以解决正权图所有问题。
缺点：不适用于负权图。

Dijkstra

算法思想：

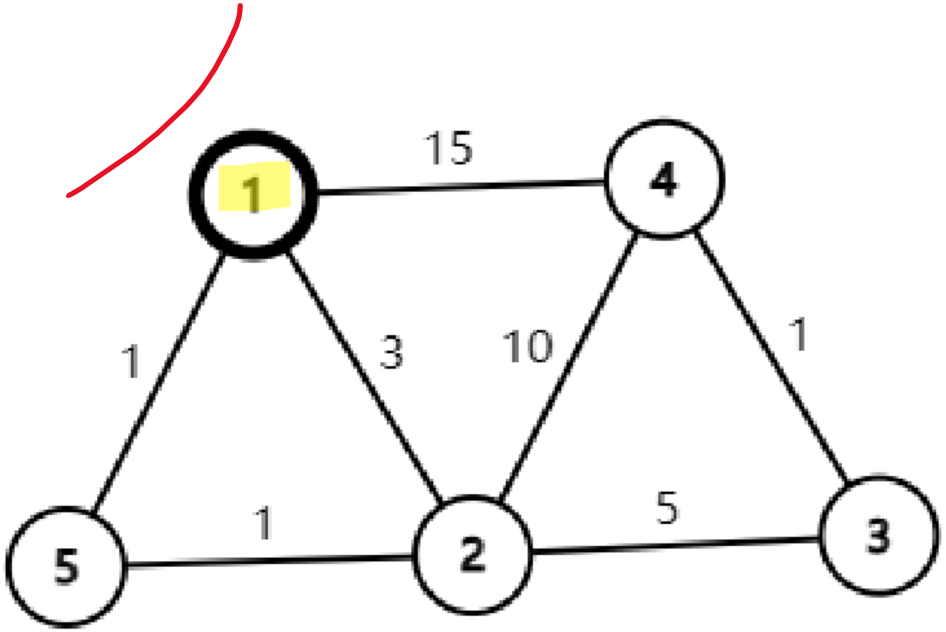
核心思想是贪心。

将所有点分为两个集合S、T。S中的点最短路已经确定，T中的点最短路未确定。（初始状态为S为空，T包含所有点。第一次操作将源点放入S）

每次在T中寻找与集合S相邻（与集合S中的某个点有边相连）的点中dis值最小的点，将它放入S集合中并用这个点来松弛与它相邻的点，直到最后T集合为空。

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点

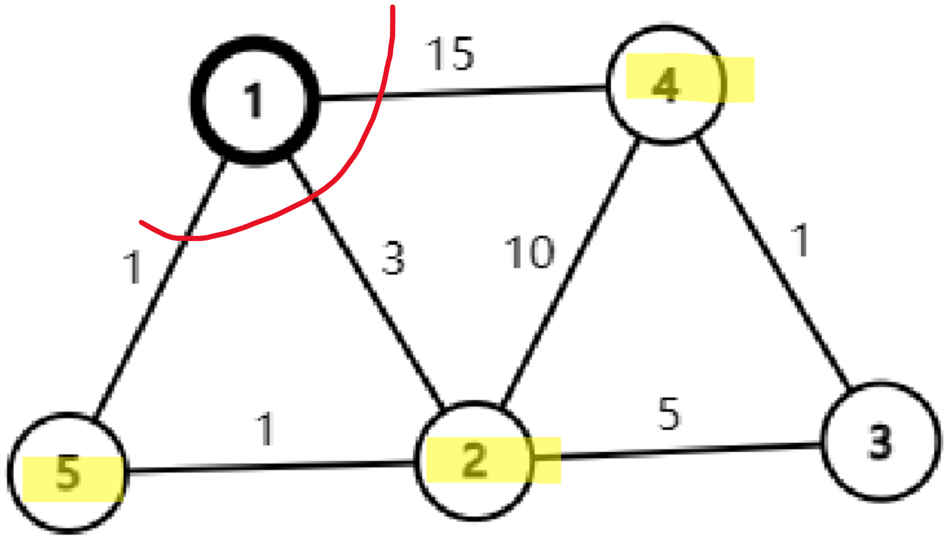


S	空
T	{1,2,3,4,5}
dis[1]	0
dis[2]	INF
dis[3]	INF
dis[4]	INF
dis[5]	INF

放入源点并用它松弛2、4、5

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点

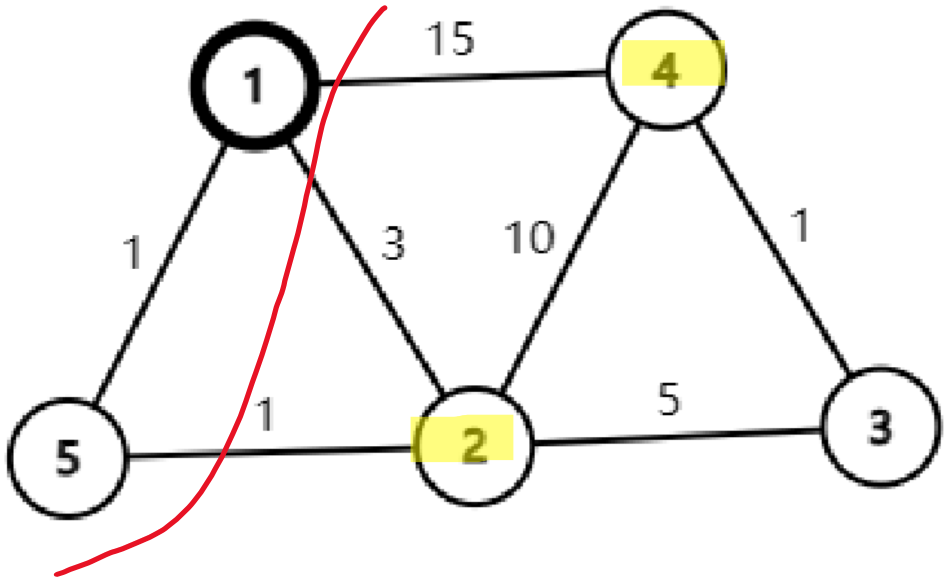


S	{1}
T	{2,3,4,5}
dis[1]	0
dis[2]	3
dis[3]	INF
dis[4]	15
dis[5]	1

dis值最小的是dis[5]=1，所以5放入S中，并用5松弛2

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点

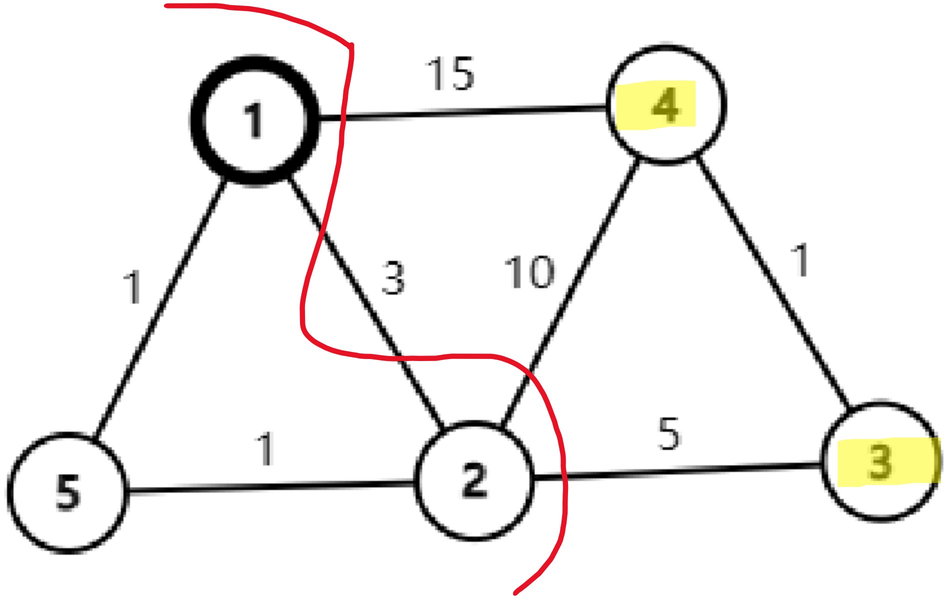


S	{1,5}
T	{2,3,4}
dis[1]	0
dis[2]	2
dis[3]	INF
dis[4]	15
dis[5]	1

dis值最小的是dis[2]=2，所以2放入S中，并用2松弛3、4

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点

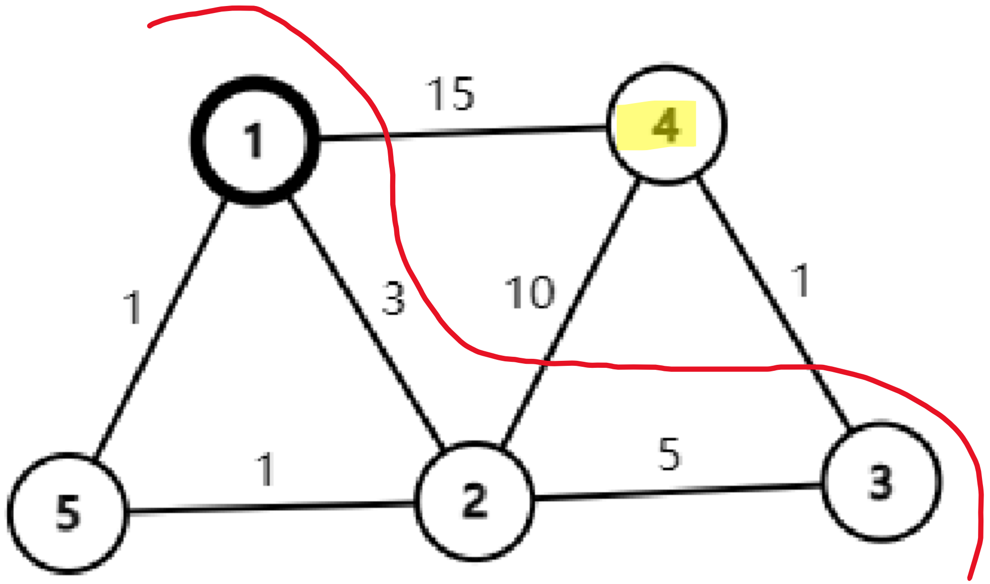


S	{1,5,2}
T	{3,4}
dis[1]	0
dis[2]	2
dis[3]	7
dis[4]	12
dis[5]	1

dis值最小的是dis[3]=7，所以3放入S中，并用3松弛4

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点

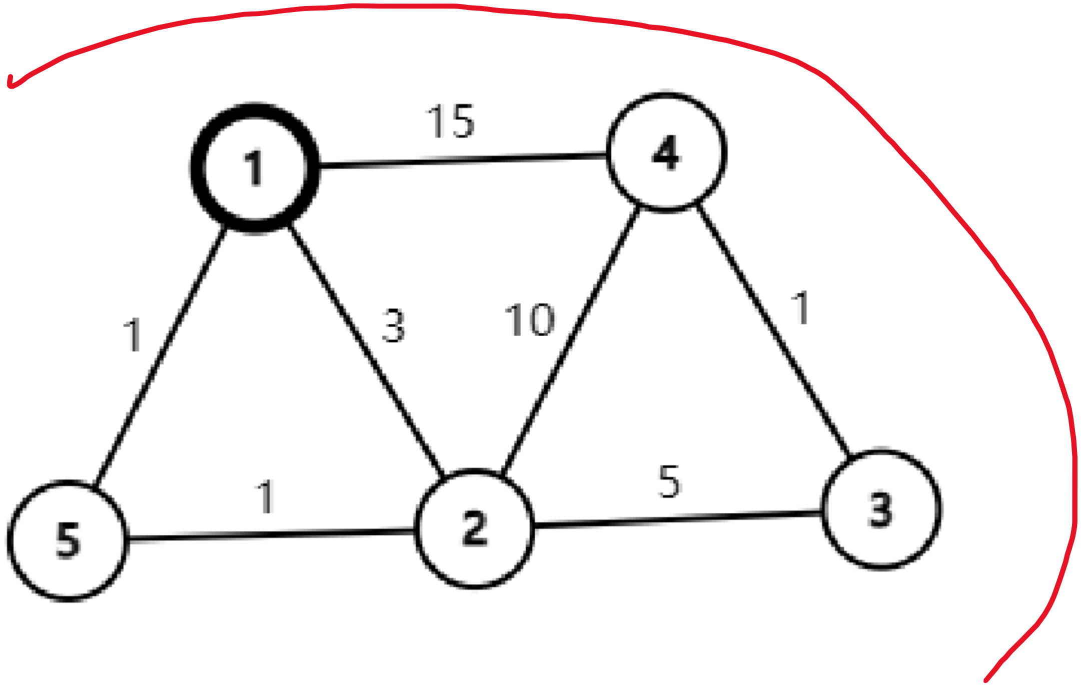


S	{1,5,2,3}
T	{4}
dis[1]	0
dis[2]	2
dis[3]	7
dis[4]	8
dis[5]	1

将4放入S中

Dijkstra

红圈围起来的点：S中的点
黄色荧光笔标记的点：与S相邻的点
表格中绿色的部分：这一轮操作中应该被松弛的点



S	{1,5,2,3,4}
T	空
dis[1]	0
dis[2]	2
dis[3]	7
dis[4]	8
dis[5]	1

结束

Dijkstra

复杂度的关键在于找dis值最小的点。

朴素的Dijkstra直接暴力枚举所有的点，然后松弛，时间复杂度为O(n^2)，这一般适用于稠密图。

堆优化的Dijkstra用一个堆维护dis值，一般适用于稀疏图。
最简单的写法是用优先队列维护。
大概流程为：每成功松弛一个点u，就把二元组(dis[u],u)放入优先队列中。每次取堆顶，若u不在S中，则将u加入S并进行松弛操作。

Dijkstra

朴素的Dijkstra 参考代码：

```
void dij(int s) {
    memset(dis, 0x3f, sizeof dis);
    dis[s] = 0; //初始化dis数组
    for(int k = 1, t; k <= n; ++k) { //一共n轮，每次找不在S集合中的点中dis值最小的点，将其放入S集合中并用它去松弛其他点
        t = -1;
        for(int i = 1; i <= n; ++i)
            if(!flag[i] && (t == -1 || dis[i] < dis[t])) t = i; //如果（不在S中 && （ t 还没有赋值 || i的dis值比t更小））, 那么t = i
        flag[t] = true; //标记t在S集合中
        for(int i = 1; i <= n; ++i) //松弛操作
            dis[i] = min(dis[i], dis[t] + a[t][i]);
    }
}
```

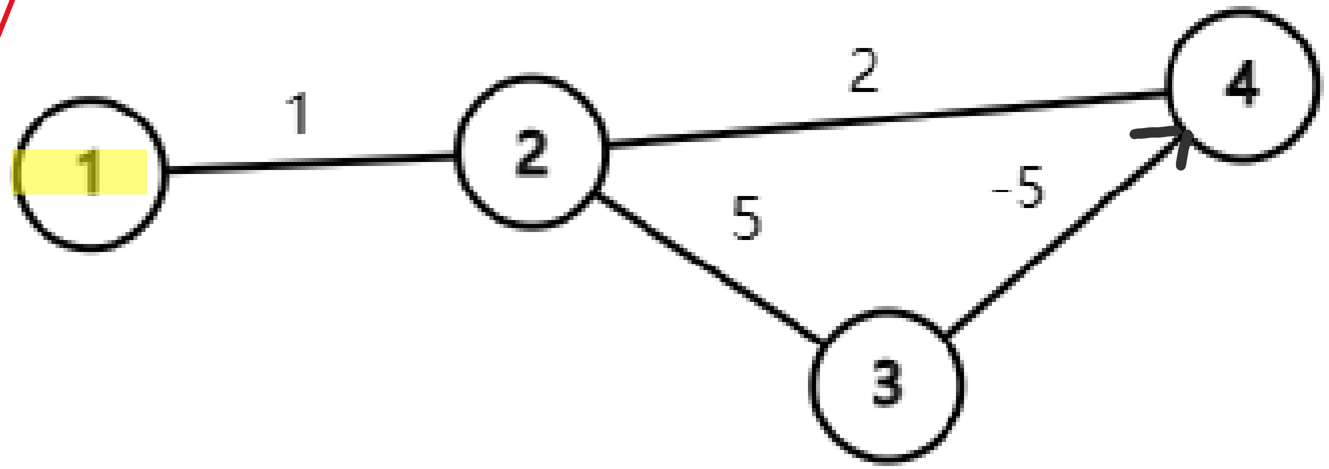
Dijkstra

堆优化的Dijkstra 参考代码：

```
void dij(int s) {
    memset(dis, 0x3f, sizeof dis);
    dis[s] = 0; //初始化dis数组
    priority_queue<pair<int, int>> q; //一个优先队列
    q.push({0, s}); //c++中的pair<int, int>默认先按第一维比较，所以把dis值放在第一维
    while(!q.empty()) {
        int x = q.top().second; q.pop(); //取堆顶
        if(flag[x]) continue; //判断是否已经在S集合中，如果已经在S集合中，那么直接continue
        flag[x] = true; //标记x在S集合中
        for(int i = fr[x], y; i; i = nxt[i]) { //遍历所有与x相连的边
            y = to[i];
            if(dis[y] > dis[x] + w[i]) { //判断是否可以松弛。
                dis[y] = dis[x] + w[i];
                if(!flag[y]) q.push({-dis[y], y}); //c++中的优先队列默认为大根堆，而我们需要小根堆，为了方便可以直接取相反数
            }
        }
    }
}
```

Dijkstra

为何不能有负权边?

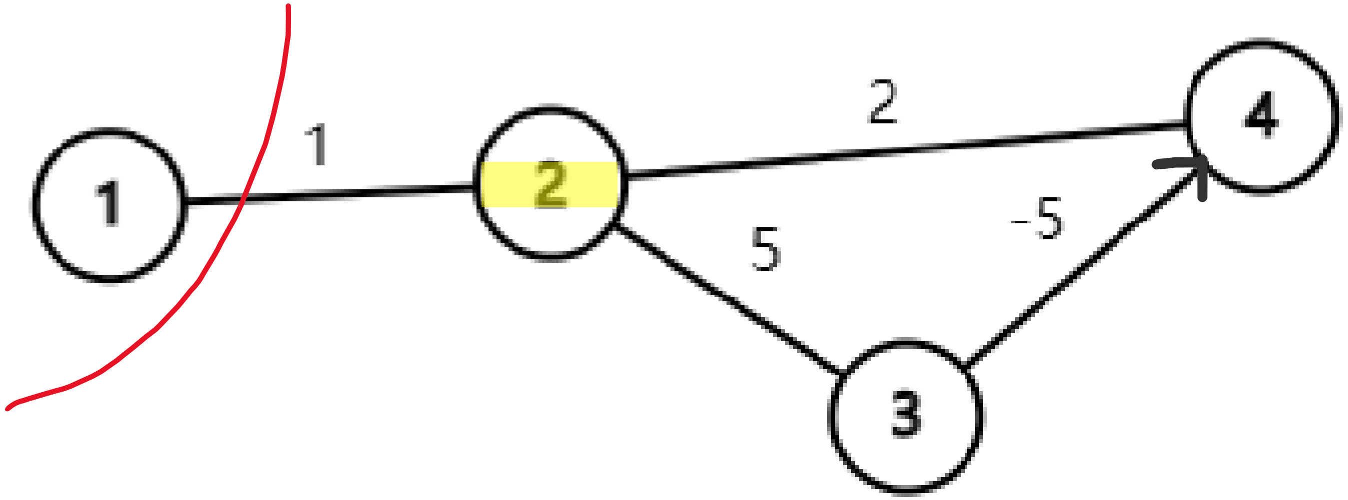


S	空
T	{1,2,3,4}
dis[1]	0
dis[2]	INF
dis[3]	INF
dis[4]	INF

将1加入S，更新2

Dijkstra

为何不能有负权边?

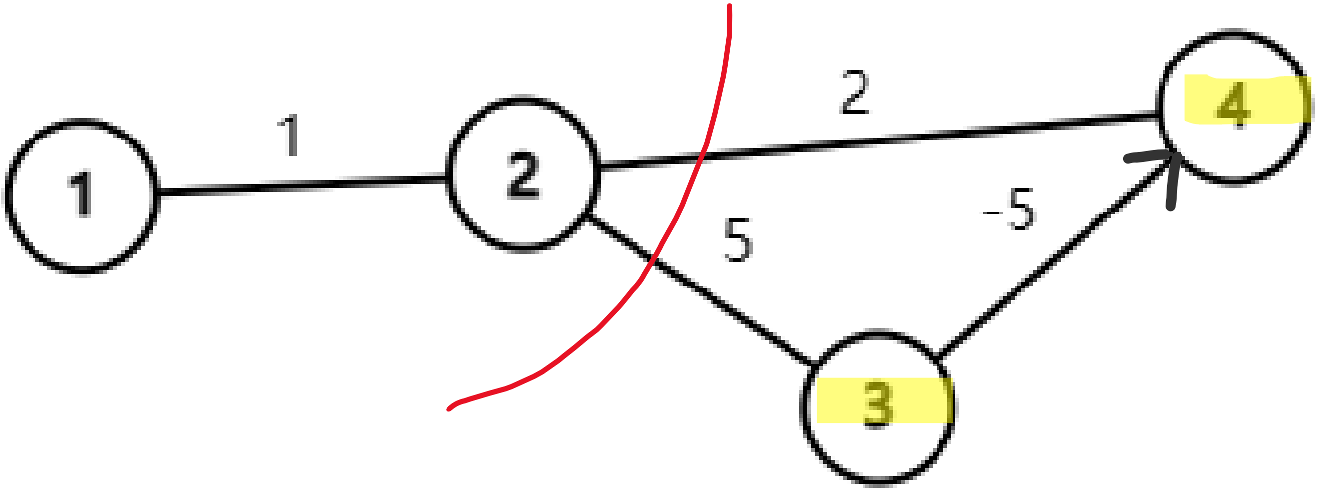


S	{1}
T	{2,3,4}
dis[1]	0
dis[2]	1
dis[3]	INF
dis[4]	INF

将2加入S，更新3、4

Dijkstra

为何不能有负权边?

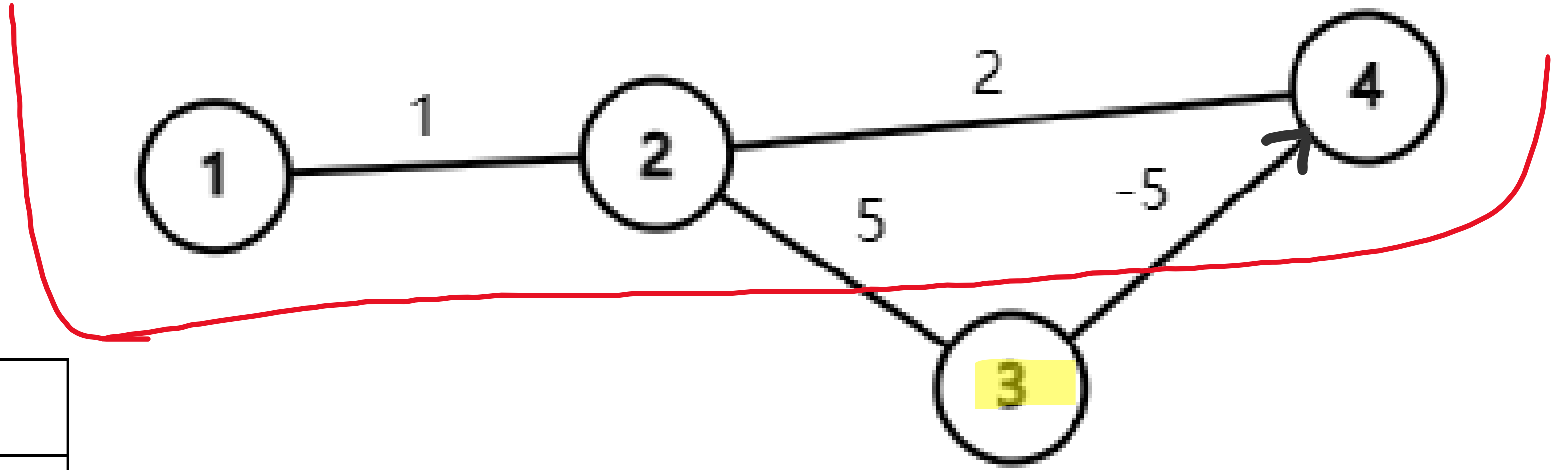


S	{1,2}
T	{3,4}
dis[1]	0
dis[2]	1
dis[3]	6
dis[4]	3

dis最小的是4，将4加入S

Dijkstra

为何不能有负权边?

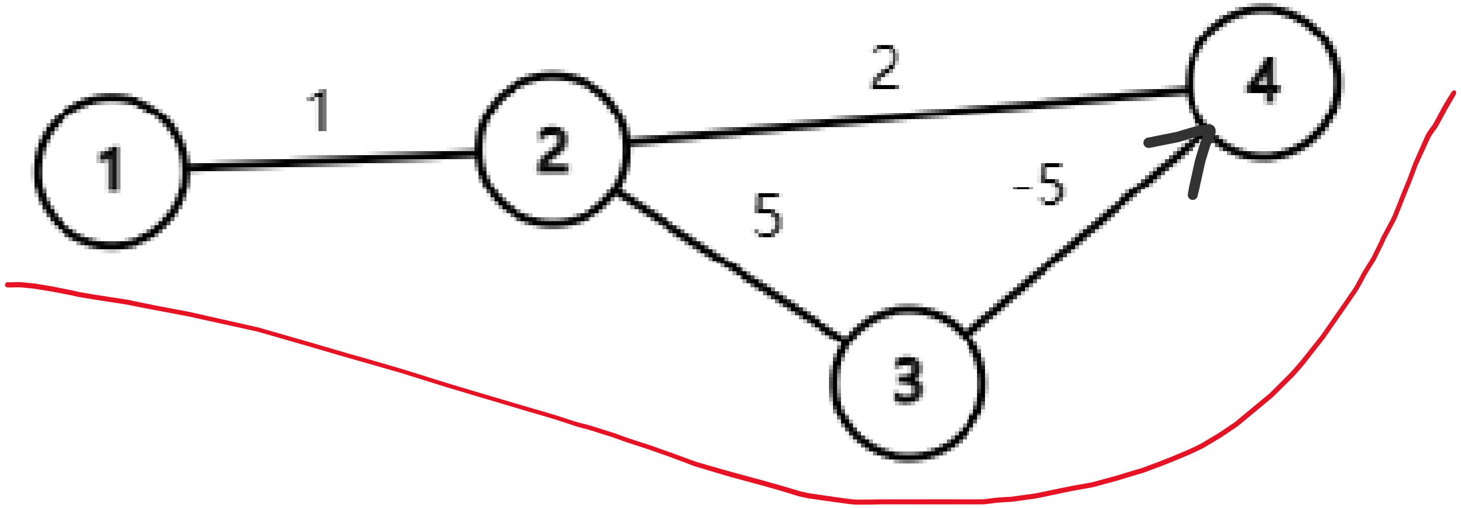


S	{1,2,4}
T	{3}
dis[1]	0
dis[2]	1
dis[3]	6
dis[4]	3

将3加入S

Dijkstra

为何不能有负权边?



S	{1,2,4,3}
T	空
dis[1]	0
dis[2]	1
dis[3]	6
dis[4]	3

实际上应该为：

dis[1]	0
dis[2]	1
dis[3]	–2
dis[4]	1

在Dijkstra的过程中，加入S的点是已经确定dis的，这在正权图中没有问题。
但如果存在负权边，S中的点的dis还可能被之后的负权边更新。

总结

- 若限制了边数，用Bellman–Ford。
- 若图为负权图，用Bellman–Ford或SPFA。
- 若图为正权图，用Dijkstra。
  - 若为稠密图，用朴素的Dijkstra 。
  - 若为稀疏图，用堆优化的Dijkstra 。
- 谨慎使用SPFA。

无权图、边权只有0和1的图

无权图只需要做BFS就可以求出最短路

边权只有0和1的图使用0–1 BFS就可以在O(n)复杂度内求出最短路，而不需要再加个log。

一般情况下，我们把没有权值的边扩展到的点放到队首，有权值的边扩展到的点放到队尾。这样即可保证像普通 BFS 一样整个队列队首到队尾权值单调不下降。

下面是伪代码：

```
while (队列不为空) {
  int u = 队首;
  弹出队首;
  for (枚举 u 的邻居) {
    更新数据
    if (...)
      添加到队首;
    else
      添加到队尾;
  }
}
```

参考dij的过程

记录路径

只需记录每一个点的前驱结点。

加上一句

```
if(dis[y] > dis[x] + w[i]) pre[y] = x;
```

输出的时候递归输出

```
void print(int x) {
    if(pre[x]) print(pre[x]);
    printf("%d ", x);
}
```

最短路计数

Dijkstra或者SPFA加上计数操作

```
if(dis[y] > dis[x] + w[i]) cnt[y] = cnt[x];
else if(dis[y] == dis[x] + w[i]) cnt[y] += cnt[x];
```

或者先求出最短路，然后记忆化搜索

```
int dfs(int x) {
    if(cnt[x]) return cnt[x];
    for(int i = fr[x], y; i; i = nxt[i]) {
        y = to[i];
        if(dis[x] == dis[y] + w[i]) cnt[x] += dfs(y);
    }
    return cnt[x];
}
```

次短路计数

在Dijkstra的基础上修改：

记录dist[x][0]、dist[x][1]分别为终点为x的最短路长度、次短路长度；
cnt[x][0]、cnt[x][1]分别为终点为x的最短路长度、次短路数量。

首先，我们要将是最短路还是次短路也纳入到Dijkstra的状态中：

```
struct Node {
    int id, type, distance;
    // 之前只需要id和distance，现在多一个type表示是最短路还是次短路
}
```

然后修改更新部分的代码，当前点为t，可以到达的点为j。分为四种情况讨论：

1. dist[j][0]>dist[v][type]+w[i]：当前最短路变成次短路，更新最短路，将最短路和次短路加入优先队列。其中，到达j的最短路个数和到达t是一样的 : cnt[j][0]=cnt[t][type] 。
2. dist[j][0]==dist[v][type]+w[i]：找到一条新的最短路，更新最短路条数到达j的最短路个数应该加上到达t的最短路个数，从t经过的最短路，在j上经过的时候也是最短路：cnt[j][0]+=cnt[t][type]
3. dist[j][1]>dist[v][type]+w[i]：找到一条更短的次短路，覆盖掉当前次短路，加入优先队列到达j的最短路个数和到达t是一样的 : cnt[j][1]=cnt[t][type]
4. dist[j][1]=dist[v][type]+w[i]：找到一条新的次短路，更新次短路条数到达j的最短路个数应该加上到达t的最短路个数，从t经过的最短路，在j上经过的时候也是最短路：cnt[j][1]+=cnt[t][type]

次短路计数

示例代码：

```
int dijkstra(){

    memset(st, 0, sizeof st);
    memset(dist, 0x3f, sizeof dist);
    memset(cnt, 0, sizeof cnt);

    priority_queue<node,vector<node>,greater<node>> heap;
    dist[S][0] = 0;
    cnt[S][0] = 1;
    heap.push({S,0,0});

    while(heap.size()){

        node t = heap.top();
        heap.pop();

        int ver = t.id , type = t.type , distance = t.distance;
        if(st[ver][type]) continue;
        st[ver][type] = true;


        for(int i = h[ver];i != -1;i = ne[i]){
            int j = e[i];

            //先考虑最短的情况(大于、等于)
            if(dist[j][0] > dist[ver][type] + w[i]){
                //dist[j][0]成为次小,先要赋值给dist[j][]中次小的状态
                dist[j][1] = dist[j][0]; cnt[j][1] = cnt[j][0];
                heap.push({j, 1, dist[j][1]});  //发生改变就要入队
                dist[j][0] = dist[ver][type] + w[i];  cnt[j][0] = cnt[ver][type];  //直接转移
                heap.push({j,0,dist[j][0]});

            }else if(dist[j][0] == dist[ver][type] + w[i]){

                cnt[j][0] += cnt[ver][type]; //从t经过的最短路，在j上经过的时候也是最短路

            //轮到枚举次小
            }else if(dist[j][1] > dist[ver][type] + w[i]){

                dist[j][1] = dist[ver][type] + w[i];
                cnt[j][1] = cnt[ver][type];
                 heap.push({j, 1, dist[j][1]});

            }else if(dist[j][1] == dist[ver][type] + w[i]){
                cnt[j][1] += cnt[ver][type];  //从t经过的最短路，在j上经过的时候也是最短路
            }
        }
    }
    int res = cnt[T][0];
    //最后还要特判以下最小和次小的路径之间是否相差1符合要求
    if (dist[T][0] + 1 == dist[T][1]) res += cnt[T][1];
    return res;
}
```

例题1. 通信线路

在郊区有 N 座通信基站，P 条 **双向** 电缆，第 i 条电缆连接基站 A_i 和 B_i。

特别地，1 号基站是通信公司的总站，N 号基站位于一座农场中。

现在，农场主希望对通信线路进行升级，其中升级第 i 条电缆需要花费 L_i。

电话公司正在举行优惠活动。

农产主可以指定一条从 1 号基站到 N 号基站的路径，并指定路径上不超过 K 条电缆，由电话公司免费提供升级服务。

农场主只需要支付在该路径上剩余的电缆中，升级价格最贵的那条电缆的花费即可。

求至少用多少钱可以完成升级。

数据范围

$0 \le K < N \le 1000$,
$1 \le P \le 10000$,
$1 \le L_i \le 1000000$

求**所有从** $1 \to n$ **的路径中第** $k+1$ **大的边**的最小值。

例题1. 通信线路 题解

求所有从 $1 \to n$ 的路径中第 $k+1$ 大的边的最小值。

二分答案+最短路

怎么求一个边权x的排名？
把边权大于x的边变成1，小于等于x的变成0，跑一遍最短路，求出在所有路径中最少有多少条边的长度大于x，然后+1就是x的排名。

显然这个排名具有单调性，我们可以二分答案。

例题2. 洛谷P5304 旅行者

题目描述

复制Markdown 展开

J 国有 n 座城市，这些城市之间通过 m 条单向道路相连，已知每条道路的长度。

一次，居住在 J 国的 Rainbow 邀请 Vani 来作客。不过，作为一名资深的旅行者，Vani 只对 J 国的 k 座历史悠久、自然风景独特的城市感兴趣。
为了提升旅行的体验，Vani 想要知道他感兴趣的城市之间「两两最短路」的最小值（即在他感兴趣的城市中，最近的一对的最短距离）。

也许下面的剧情你已经猜到了—— Vani 这几天还要忙着去其他地方游山玩水，就请你帮他解决这个问题吧。

$2 \le k \le n$，$1 \le x, y \le n$，$1 \le z \le 2 \times 10^9$，$T \le 5$。

测试点编号	n 的规模	m 的规模	约定
1	$\le 1,000$	$\le 5,000$	无
2	$\le 1,000$	$\le 5,000$	无
3	$\le 100,000$	$\le 500,000$	保证数据为有向无环图
4	$\le 100,000$	$\le 500,000$	保证数据为有向无环图
5	$\le 100,000$	$\le 500,000$	保证数据为有向无环图
6	$\le 100,000$	$\le 500,000$	无
7	$\le 100,000$	$\le 500,000$	无
8	$\le 100,000$	$\le 500,000$	无
9	$\le 100,000$	$\le 500,000$	无
10	$\le 100,000$	$\le 500,000$	无

一个图 n 点 m 条边，里面有 k 个特殊点，问这 k 个点之间两两最短路的最小值是多少？ $n \le 10^5, m \le 5 * 10^5$

例题2. 洛谷P5304 旅行者 题解

一个图 n 点 m 条边，里面有 k 个特殊点，问这 k 个点之间两两最短路的最小值是多少？ $n \leq 10^5, m \leq 5*10^5$

二进制+最短路

如果我们把k个特殊点分成两个集合A、B，我们如何求A、B之间的点的两两最短路的最小值呢?
添加S、T两个点，S与A中所有点连一个边权为0的边，T与B所有点连一个边权为0的边。
最后S和T之间的最短路即A、B之间的点的两两最短路的最小值。

题解是这样构造的：
枚举第i个二进制位，将该位是0的点放在A集合，该位是1的点放在B集合。用上面的方法跑一个最短路，最后取个min即可。

为啥这样是正确的呢?
假设最后答案在x和y取，那么x肯定不等于y，即x与y的二进制中必然会有至少一位不同，所以x与y必然会在某一次中被分到两个集合中。这样就保证了一定能够取到x、y。

例题3. 道路和航线

给定一个图，有两种边：一种是有向，权值可正可负；另一种无向，权值恒为正；

图满足：若a到b有一条有向边，不存在b到a的路（不能从b开始回到a）。

给一个点，求这个点到所有点的最短路。

例题3. 道路和航线 题解

给定一个图，有两种边：一种是有向，权值可正可负；另一种无向，权值恒为正；

图满足：若a到b有一条有向边，不存在b到a的路（不能从b开始回到a）。

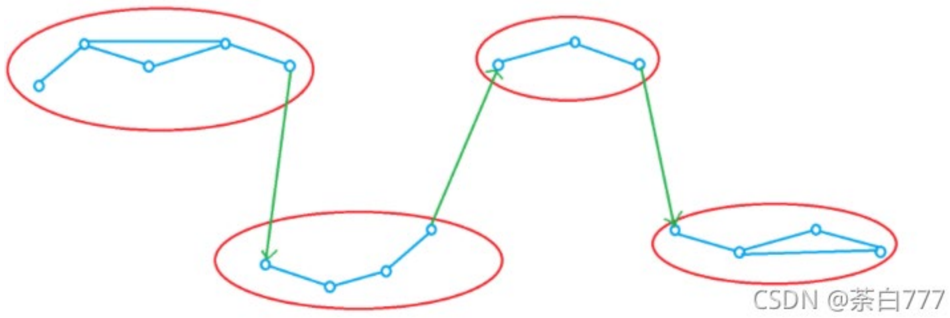
给一个点，求这个点到所有点的最短路。

首先，这个图存在负边，所以不能直接使用Dijkstra。

观察题目，若只考虑有向边，我们发现它是一个DAG。所以我们考虑缩点，整个图就变成了一个DAG，我们可以在上面跑拓扑排序。

而无向边的权值为正，所以可以直接跑Dijkstra。

细节可能较多。



例题4. 拯救大兵瑞恩

1944 年，特种兵麦克接到国防部的命令，要求立即赶赴太平洋上的一个孤岛，营救被敌军俘虏的大兵瑞恩。

瑞恩被关押在一个迷宫里，迷宫地形复杂，但幸好麦克得到了迷宫的地形图。

迷宫的外形是一个长方形，其南北方向被划分为 N 行，东西方向被划分为 M 列， 于是整个迷宫被划分为 $N \times M$ 个单元。

每一个单元的位置可用一个有序数对 (单元的行号, 单元的列号) 来表示。

南北或东西方向相邻的 2 个单元之间可能互通，也可能有一扇锁着的门，或者是一堵不可逾越的墙。

注意： 门可以从两个方向穿过，即可以看成一条无向边。

迷宫中有一些单元存放着钥匙，同一个单元可能存放 **多把钥匙**，并且所有的门被分成 P 类，打开同一类的门的钥匙相同，不同类门的钥匙不同。

大兵瑞恩被关押在迷宫的东南角，即 (N, M) 单元里，并已经昏迷。

迷宫只有一个入口，在西北角。

也就是说，麦克可以直接进入 $(1, 1)$ 单元。

另外，麦克从一个单元移动到另一个相邻单元的时间为 1，拿取所在单元的钥匙的时间以及用钥匙开门的时间可忽略不计。

试设计一个算法，帮助麦克以最快的方式到达瑞恩所在单元，营救大兵瑞恩。

数据范围

$|X_{i1} - X_{i2}| + |Y_{i1} - Y_{i2}| = 1$,
$0 \le G_i \le P$,
$1 \le Q_i \le P$,
$1 \le N, M, P \le 10$,
$1 \le k \le 150$

例题4. 拯救大兵瑞恩 题解

对于每个点，它们的属性不光有(x,y)坐标这个属性，还有是否有钥匙这个属性，所以一个点(x,y),可以根据其属性
来写为三维状态$f(x,y,state)$，表示从起点到(x,y)这个点且当前拥有个要是状态是state的路线最短路
看似可以用dp(动态规划来做这个题),但是用dp做，计算当前状态就需要之前的所有状态，但是这是一个网格状的棋盘，是可以走回路的
比如说，一个人在(x,y)先去拿钥匙，再回(x,y)，出现环形结构，那么计算当前状态反而需要当前状态的值，出现矛盾，不太好计算
所以可以把$f(x,y,state)$，当成一个三维坐标，但和正常的三维坐标不同的是，第三个状态不是位置，而是一种钥匙的状态，但不妨碍可以借鉴三维坐标。
先用dp的思想来想状态来如何转移
key来表示当前位置的钥匙存在状态
此时这是拿当前位置钥匙的方程转移$f(x,y,state)=min(f(x,y,state),f(x,y,state|key))$，拿钥匙不消耗时间
如果不拿钥匙，往四个方向走，需要消耗1个时间点
此时$f(x,y,state)=min(f(a,b,state)+1,f(x,y,state))$,$(a,b)$点是$(x,y)$的邻点
Dp需要的结构是拓扑序，导致不能正常转移
有环形依赖的结构则需要最短路解决
通过上述的状态转移方程，发现当前位置有钥匙的话，捡起钥匙，不需要消耗体力，所以就相当于边权为0
如果不捡位置，取周围4个方向的位置(能走过去的话)，则需要消耗1个体力，相当于边权为1
此时问题就简化为了，一个三维状态点，且边权是0和1的最短路问题

差分约束

我们先来看一个简单的数学问题，如下给定 4 个变量和 5 个不等式约束条件，求 $x_3 - x_0$ 的最大值。

$$
\begin{aligned}
x_1 - x_0 &\le 2 && (1)\\
x_2 - x_0 &\le 7 && (2)\\
x_3 - x_0 &\le 8 && (3)\\
x_2 - x_1 &\le 3 && (4)\\
x_3 - x_2 &\le 2 && (5)
\end{aligned}
$$

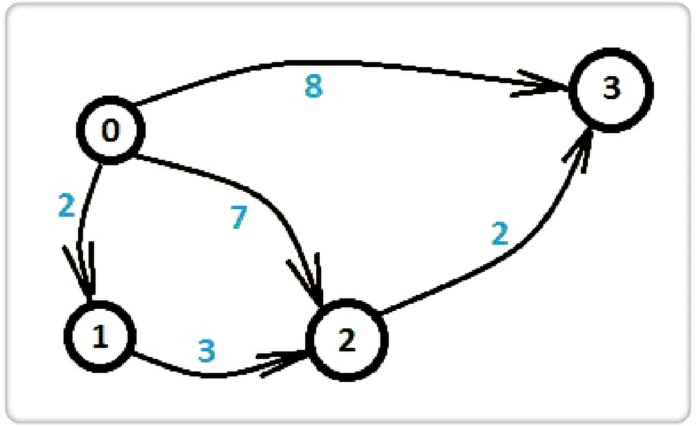
我们可以通过不等式的两两加得到三个结果，

$$
\begin{aligned}
x_3 - x_0 &\le 8 && (\text{式}3)\\
x_3 - x_0 &\le 9 && (\text{式}2+\text{式}5)\\
x_3 - x_0 &\le 7 && (\text{式}1+\text{式}4+\text{式}5)
\end{aligned}
$$

差分约束

这个例子很简单，只有 4 个变量和 5 个不等式约束条件，那如果有上百变量上千约束条件呢？仅凭肉眼手工计算效率太差，因此我们需要一个较为系统的解决办法。

我们先来看一幅图，如下，给定四个小岛以及小岛之间的有向距离，问从 0 号岛到 3 号岛的最短距离。箭头指向的线代表两个小岛之间的有向边，蓝色数字代表距离权值。



这个问题就是经典的最短路问题。由于这个图比较简单，我们可以枚举所有的路线，发现总共三条路线，如下：

1. 0 -> 3，长度为 8
2. 0 -> 2 -> 3，长度为 7 + 2 = 9
3. 0 -> 1 -> 2 -> 3，长度为 2 + 3 + 2 = 7

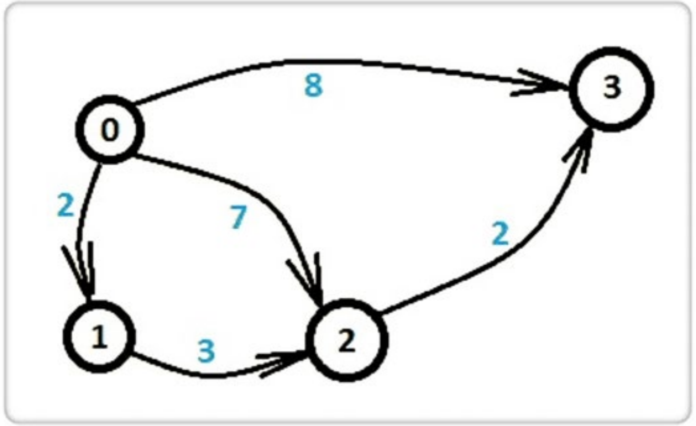
最短路为三条线路中的长度的最小值，即 7，所以最短路的长度就是 7。细心的读者会发现，这幅图和最上方的五个不等式约束条件是有所关联的，但这个关联并不是巧合，而正是我们接下来要讲的那个 "系统的解决办法"。

差分约束

差分约束系统中的每个约束条件 $x_i - x_j <= c_k$ 都可以变形成 $x_i <= x_j + c_k$ ，这与单源最短路中的三角形不等式 $dist[y] <= dist[x] + z$ 非常相似。

因此，我们可以把每个变量 x_i 看做图中的一个结点，对于每个约束条件 $x_i - x_j <= c_k$ ，看成是从结点 j 向结点 i 的一条权值为 c_k 的有向边，于是我们就可以把一个差分约束系统转化成图的最短路问题。

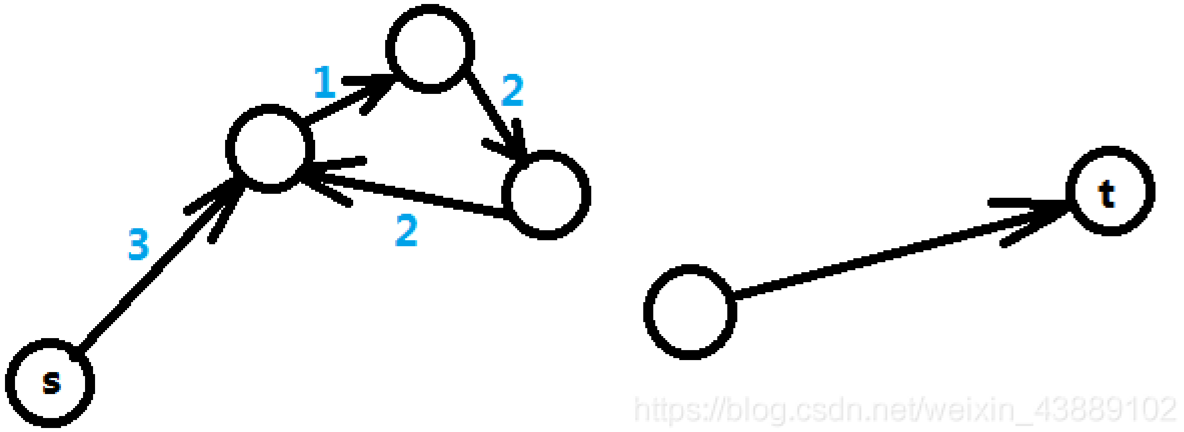
$$x_1 - x_0 \leq 2$$
$$x_2 - x_0 \leq 7$$
$$x_3 - x_0 \leq 8$$
$$x_2 - x_1 \leq 3$$
$$x_3 - x_2 \leq 2$$



差分约束

不连通：

图中s和t不连通，说明s和t没有约束关系，即xs−xt可以取任意值，有无穷多组可行解。

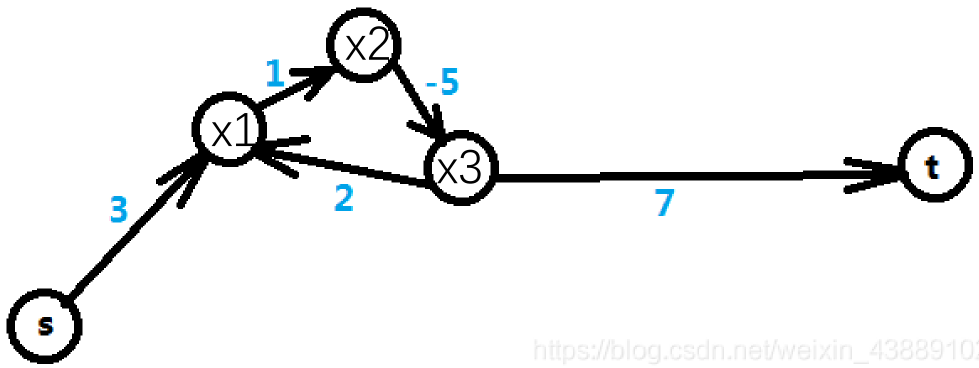


差分约束

存在负环：

如图，负环部分对应的不等式为：x2<=x1+1、x3<=x2-5、x1<=x3+2。
三个式子加起来得到0<=1-5+2，矛盾，所以不可能存在可行解

所以，存在负环一定没有可行解。



差分约束

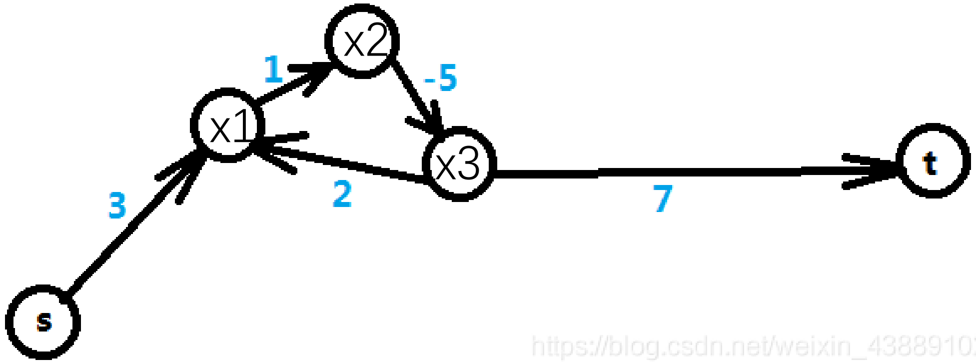
如何求解可行解?

如果不等式为<=，那么求最大可行解，要用到最短路。
如果不等式为>=，那么求最小可行解，要用到最长路。

考虑<=的情况。
首先建立一个超级源点S，对于所有的点i都建立一条S->i的边权为0的边，表示S<=xi。

然后以S为源点跑最短路，dis[i]即为xi的最大值。

如果是<，则可以把xi<xj+c改成xi<=xj+c-1或者xi<=xj+c-eps



差分约束 例题 洛谷P4926

题目描述

复制Markdown 收起

今天 Scarlet 在机房有幸目睹了一场别开生面的 OI 训练。因为一些奇妙的 SPJ，比赛中所有选手的得分都是正实数（甚至没有上限）。

当一位选手 A 的分数不小于选手 B 的分数 k（$k>0$）倍时，我们称选手 A k **倍杀** 了选手 B，选手 B **被** 选手 A k **倍杀** 了。

更奇妙也更激动人心的是，训练前有不少选手立下了诸如 "我没 k 倍杀选手 X，我就女装"，"选手 Y 把我 k 倍杀，我就女装" 的 Flag。

知道真相的良心教练 Patchouli 为了维持机房秩序，放宽了选手们的 Flag 限制。Patchouli 设定了一个 **正** 常数 T，立下 "我没 k 倍杀选手 X 就女装" 的选手只要成功 $k-T$ 倍杀了选手 X，就不需要女装。同样的，立下 "选手 Y 把我 k 倍杀我就女装" 的选手只要没有成功被选手 Y $k+T$ 倍杀，也不需要女装。

提前知道了某些选手分数和具体 Flag 的 Scarlet 实在不忍心看到这么一次精彩比赛却没人女装，为了方便和 Patchouli 交易，Scarlet 想要确定最大的实数 T 使得赛后一定有选手收 Flag 女装。

给出一系列不等式：

$$x_{a_i} \geq (k_i - t) \times x_{b_i} \quad \& \quad (k_i + t) \times x_{a_i} > x_{b_i}$$

以及一些 x_i 的值 。

求出最大的 t 使得不等式无解 。

差分约束 例题 洛谷P4926 题解

给出一系列不等式：

$$x_{a_i} \geq (k_i - t) \times x_{b_i} \quad \& \quad (k_i + t) \times x_{a_i} > x_{b_i}$$

以及一些 x_i 的值。

求出最大的 t 使得不等式无解。

算法：差分约束 + 二分答案。

1. 连边

首先对不等式进行拆分化简：

$$x_{a_i} \geq (k_i - t) \times x_{b_i}$$

$$\log_2(x_{a_i}) \geq \log_2(x_{b_i}) + \log_2(k_i - t)$$

连边 `add(b,a,log2(k-t))`

$$(k_i + t) \times x_{a_i} > x_{b_i}$$

$$\log_2(x_{a_i}) + \log_2(k_i + t) > \log_2(x_{b_i})$$

$$\log_2(x_{a_i}) > \log_2(x_{b_i}) - \log_2(k_i + t)$$

由于本题有精度 10^{-5} 的容量范围，所以我们可以连边 `add(b,a,-log2(k+t))`

（具体实现连边操作时只用记录 k 的值，t 根据边的种类在差分的时候分类讨论。）

2. 判断无解

输出 `-1` 仅当 $t = 0$ 时不等式仍旧有解。

3. 二分答案

二分一个 t，判断这个时候不等式是否有解。输出答案。

其他

Johnson全源最短路（可以有负权边）：
参考博客：https://studyingfather.blog.luogu.org/johnson-algorithm

同余最短路：
参考博客：https://www.cnblogs.com/guanlexiangfan/p/15502326.html
https://oi-wiki.org/graph/mod-shortest-path/

一些习题

P3956 [NOIP2017 普及组] 棋盘
P1126 机器人搬重物
P1141 01迷宫
P1162 填涂颜色
P1331 海战
P1332 血色先锋队
P1443 马的遍历