北京大学
PEKING UNIVERSITY

北京大学暑期课《ACM/ICPC竞赛训练》

北京大学信息学院 郭炜
guo_wei@PKU.EDU.CN
http://weibo.com/guoweiofpku

课程网页：http://acm.pku.edu.cn/summerschool/pku_acm_train.htm

广度优先搜索

入门：抓住那头牛

抓住那头牛(POJ3278)

农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点N(0<=N<=100000)，牛位于点K(0<=K<=100000)。农夫有两种移动方式：

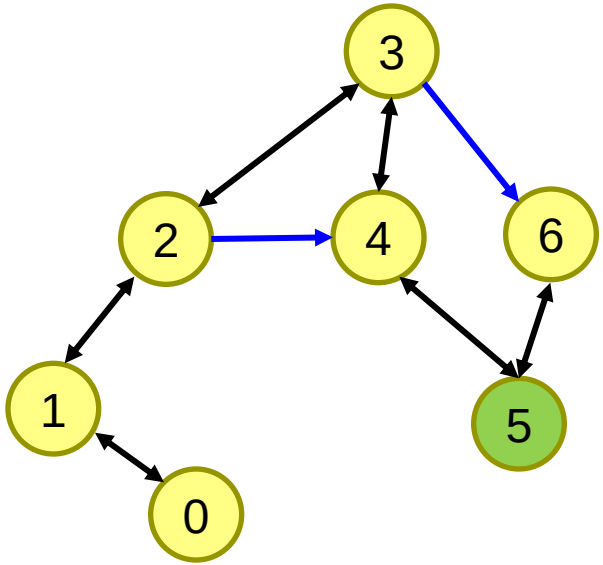
1、从X移动到X-1或X+1，每次移动花费一分钟
2、从X移动到2*X，每次移动花费一分钟

假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

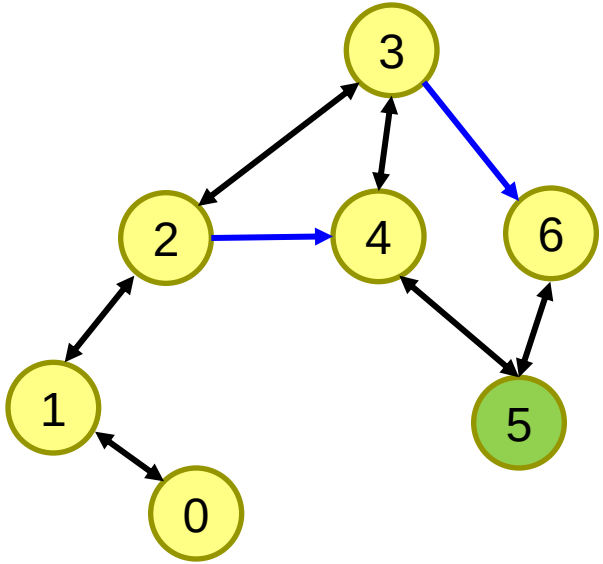
假设农夫起始位于点3，牛位于5

N=3, K=5，最右边是6。

如何搜索到一条走到5的路径？

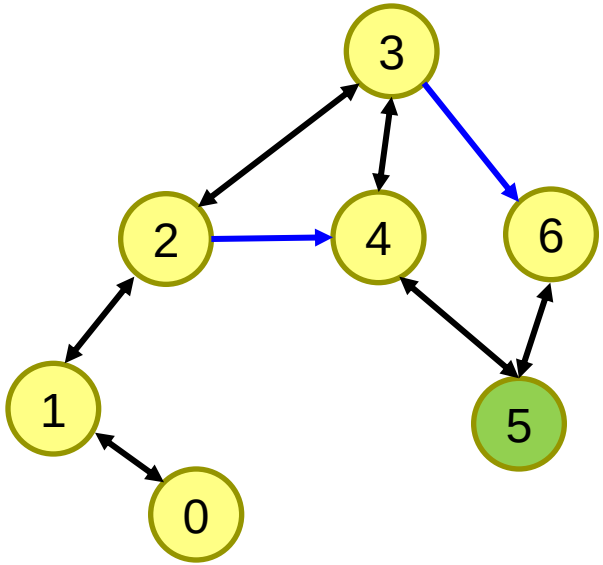


假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



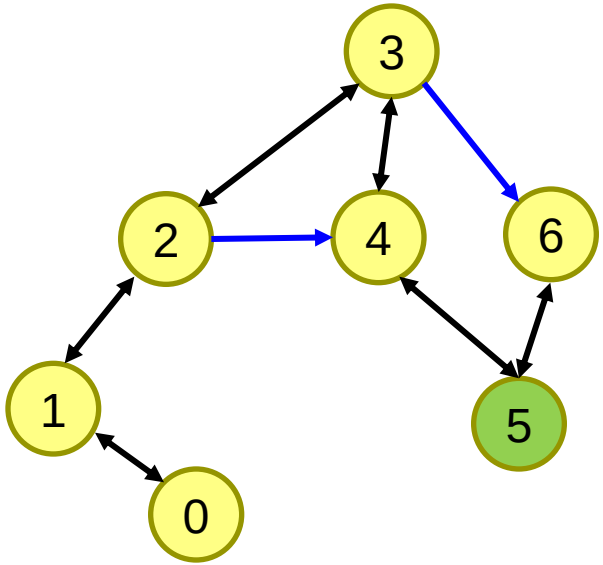
策略1）深度优先搜索：从起点出发，随机挑一个方向，能往前走就往前走(扩展)，走不动了则回溯。不能走已经走过的点(要判重)。

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



运气好的话：
3->4->5
或
3->6->5
问题解决！

# 假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？

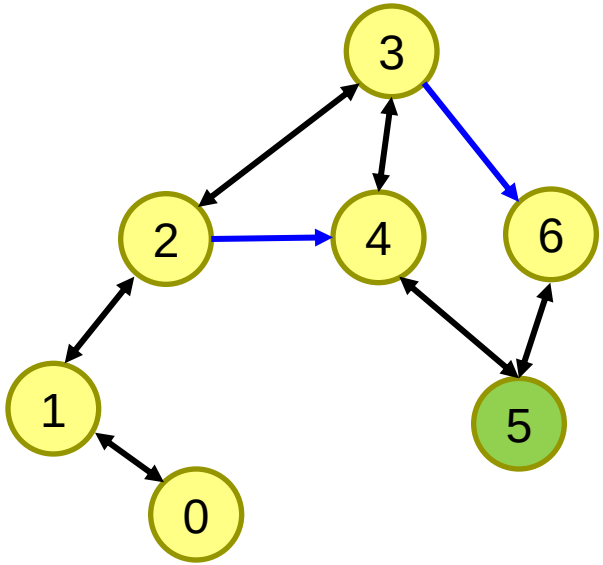


运气不太好的话：
3->2->4->5
运气最坏的话：
3->2->1->0->4->5

要想求最优(短）解，则要遍历所有走法。可以用各种手段优化，比如，若已经找到路径长度为n的解，则所有长度大于n的走法就不必尝试。

运算过程中需要存储路径上的节点，数量较少。用栈存节点。

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？

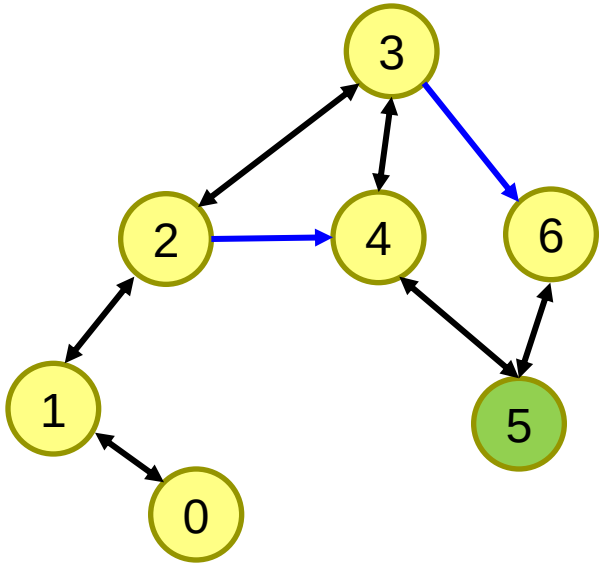


策略2）广度优先搜索：

给节点分层。起点是第0层。从起点最少需n步就能到达的点属于第n层。

第1层：2, 4, 6
第2层：1, 5
第3层：0

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？

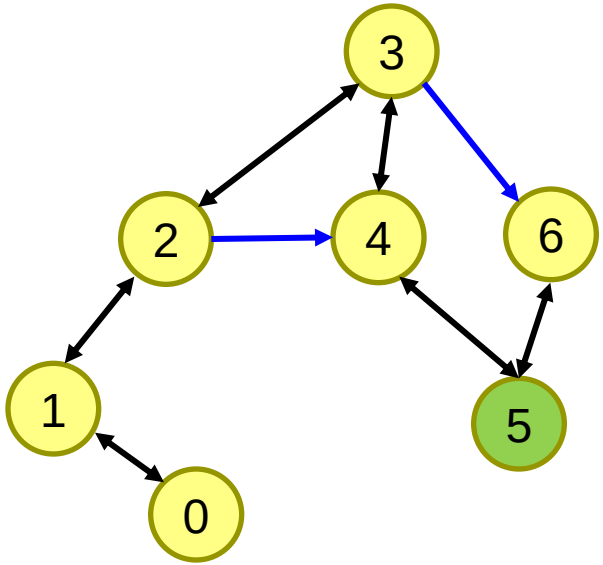


策略2）广度优先搜索：

给节点分层。起点是第0层。从起点最少需n步就能到达的点属于第n层。

依层次顺序，从小到大扩展节点。把层次低的点全部扩展出来后，才会扩展层次高的点。

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



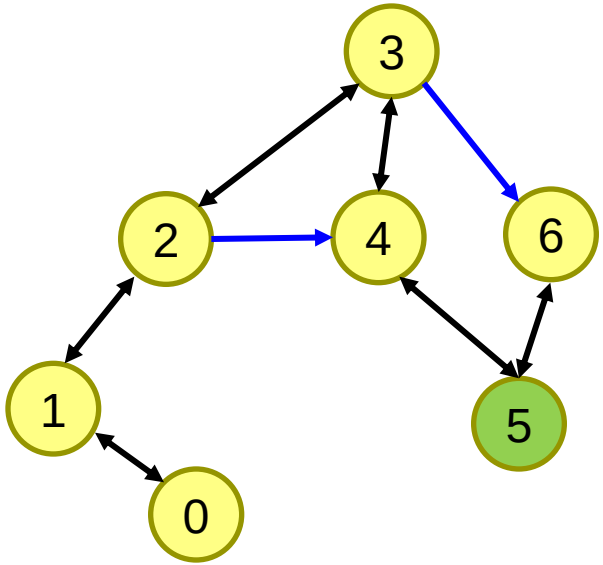
策略2）广度优先搜索：

搜索过程（节点扩展过程）：

3
2 4 6
1 5

问题解决。
扩展时，不能扩展出已经走过的节点(要判重) 。

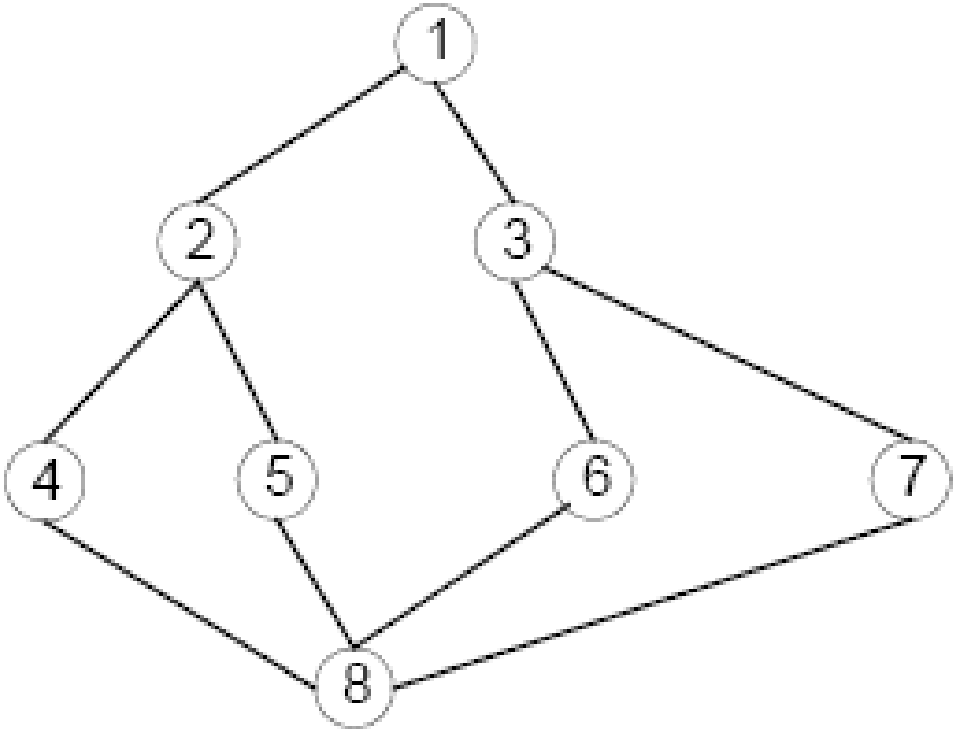
假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



策略2）广度优先搜索：

可确保找到最优解，但是因扩展出来的节点较多，且多数节点都需要保存，因此需要的存储空间较大。
用队列存节点。

深搜 vs. 广搜



若要遍历所有节点：

- 深搜

1-2-4-8-5-6-3-7

- 广搜

1-2-3-4-5-6-7-8

广搜算法

□广度优先搜索算法如下：（用QUEUE）

(1) 把初始节点S0放入Open表中；

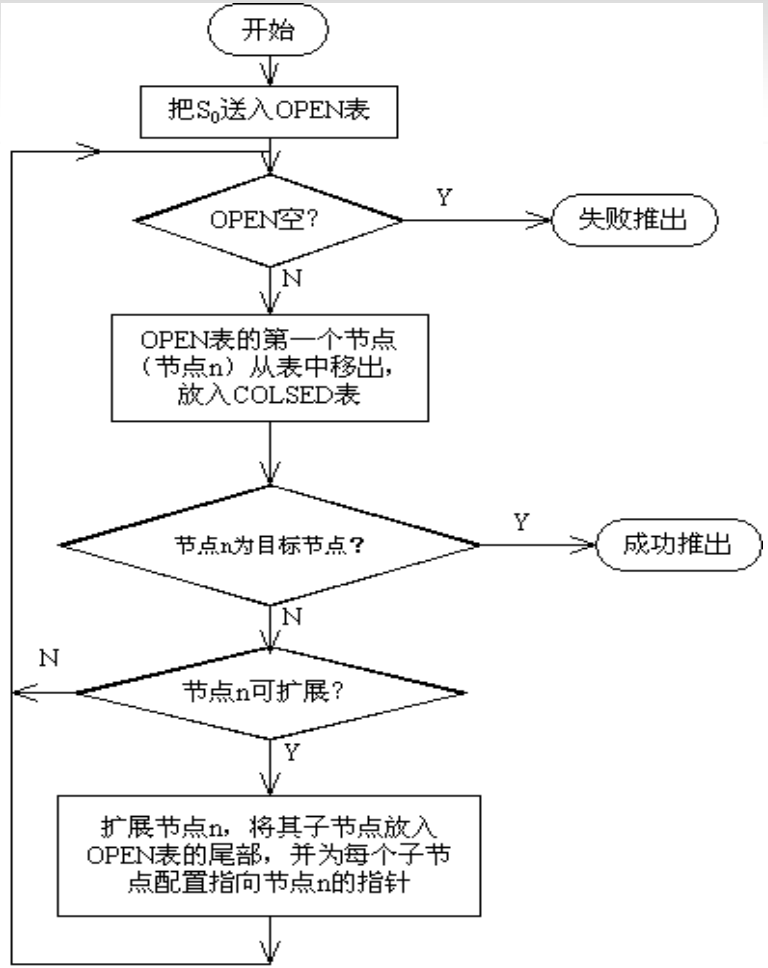
(2) 如果Open表为空，则问题无解，失败退出；

(3) 把Open表的第一个节点取出放入Closed表，并记该节点为n；

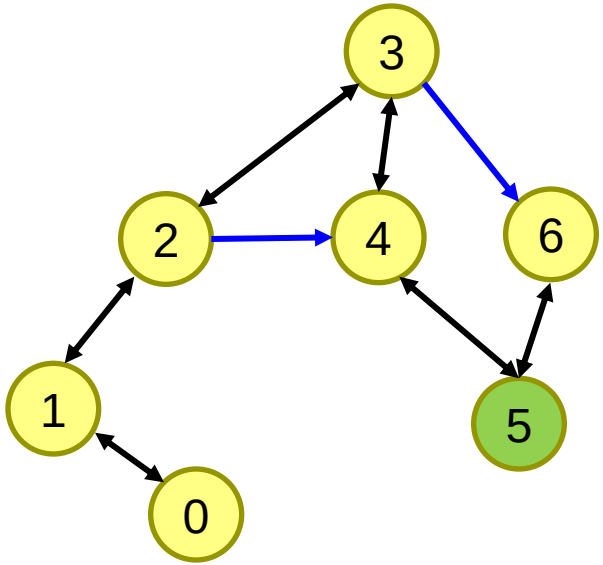
(4) 考察节点n是否为目标节点。若是，则得到问题的解，成功退出；

(5) 若节点n不可扩展，则转第(2)步；

(6) 扩展节点n，将其不在Closed表和Open表中的子节点(判重）放入Open表的尾部，并为每一个子节点设置指向父节点的指针(或记录节点的层次），然后转第(2)步。



假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



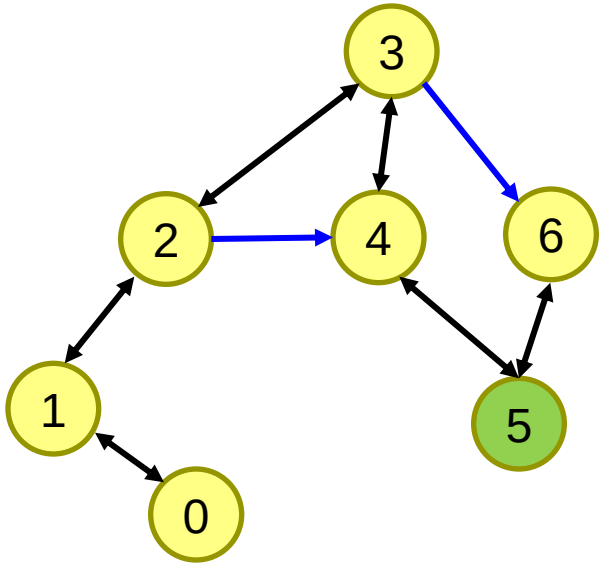
广度优先搜索队列变化过程：

Closed

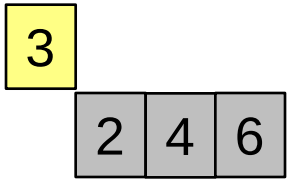
3

Open

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



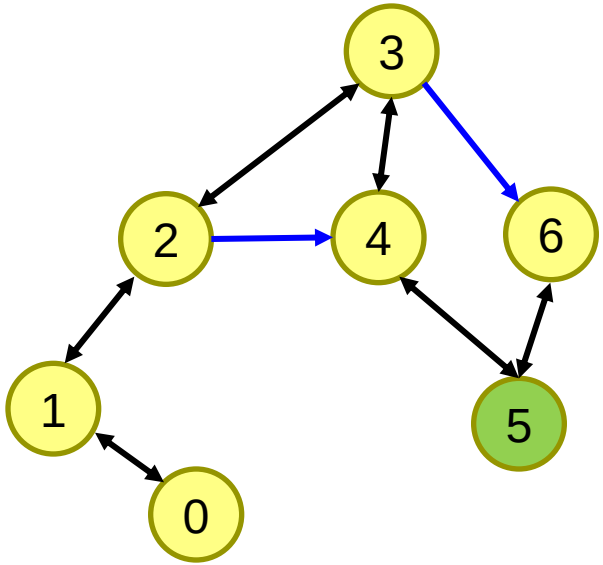
广度优先搜索队列变化过程：



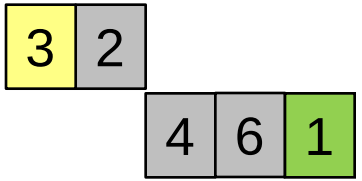
Closed

Open

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



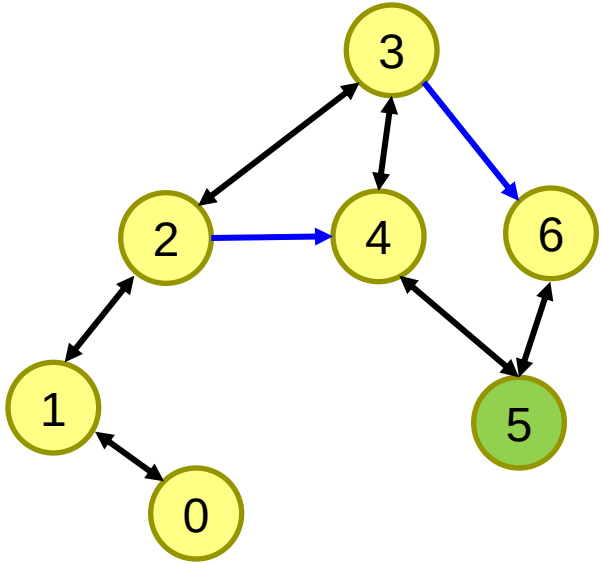
广度优先搜索队列变化过程：



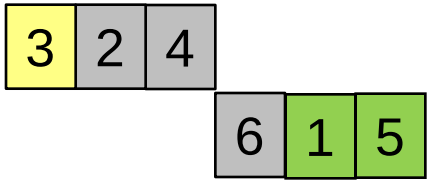
Closed

Open

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



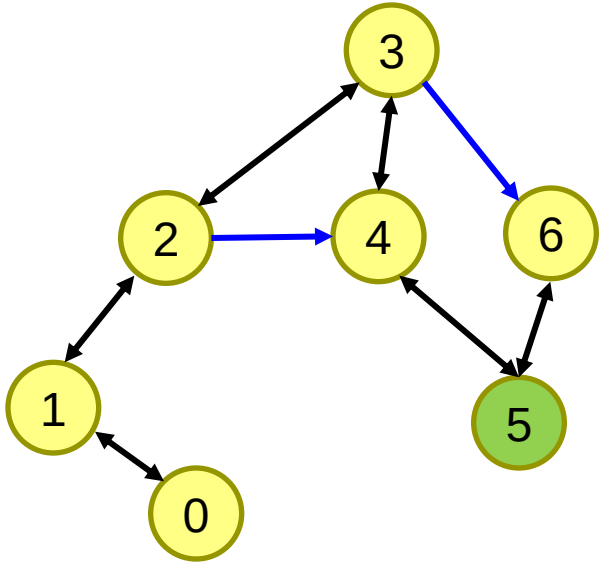
广度优先搜索队列变化过程：



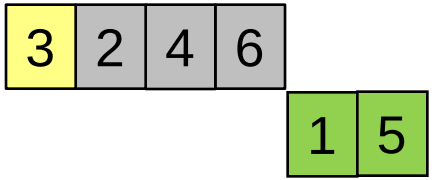
Closed

Open

假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



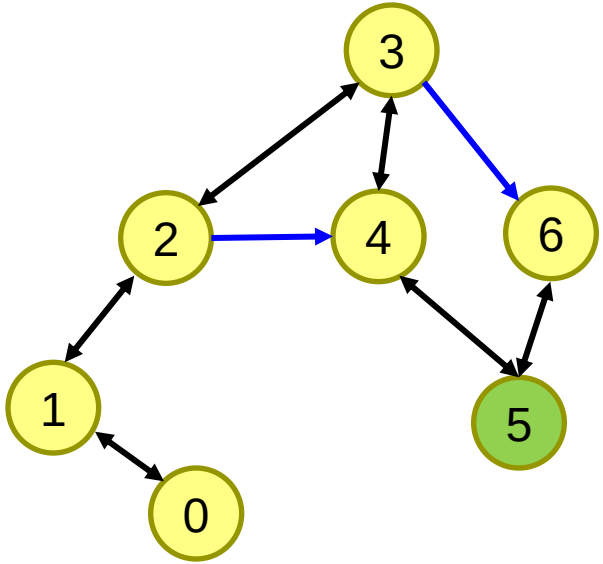
广度优先搜索队列变化过程：



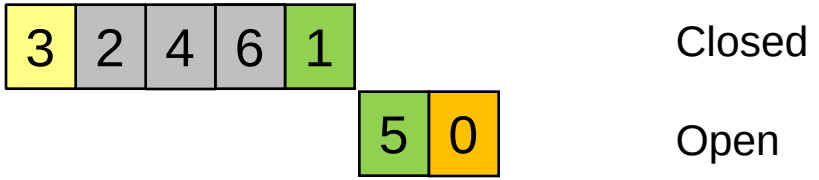
Closed

Open

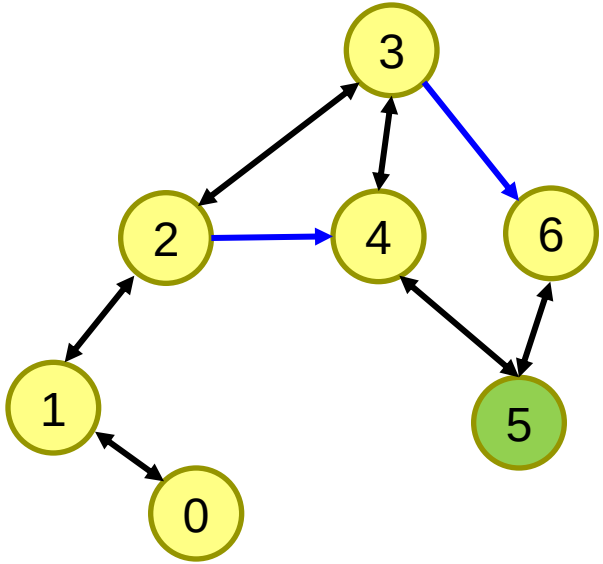
假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



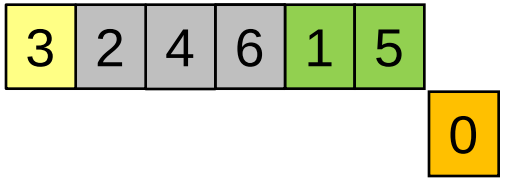
广度优先搜索队列变化过程：



假设农夫起始位于点3，牛位于5
N=3, K=5，最右边是6。
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：



Closed

Open

目标节点5出队列，问题解决！

//poj3278 Catch That Cow

```
#include <iostream>
#include <cstring>
#include <queue>
using namespace std;
int N,K;
const int MAXN = 100000;
int visited[MAXN+10];  //判重标记,visited[i] = true表示i已经扩展过
struct Step{
        int x;  //位置
        int steps;  //到达x所需的步数
        Step(int xx,int s):x(xx),steps(s) { }
};
queue<Step> q;  //队列,即Open表
int main()   {
        cin >> N >> K;
        memset(visited,0,sizeof(visited));
        q.push(Step(N,0));
        visited[N] = 1;
```

```
while(!q.empty()) {
        Step s = q.front();
        if( s.x == K ) { //找到目标
                cout << s.steps <<endl;
                return 0;
        }
        else {
                if( s.x - 1 >= 0 && !visited[s.x-1] ) {
                        q.push(Step(s.x-1,s.steps+1));
                        visited[s.x-1] = 1;
                }
                if( s.x + 1 <= MAXN && !visited[s.x+1] ) {
                        q.push(Step(s.x+1,s.steps+1));
                        visited[s.x+1] = 1;
                }
```

```
                    if( s.x * 2 <= MAXN &&!visited[s.x*2]  ) {
                        q.push(Step(s.x*2,s.steps+1));
                        visited[s.x*2]  = 1;
                    }
                    q.pop();
                }
            }
            return 0;
}
```

迷宫问题（POJ3984）

定义一个矩阵：

0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

迷宫问题（POJ3984）

基础广搜。先将起始位置入队列

每次从队列拿出一个元素，扩展其相邻的4个元素入队列(要判重），
直到队头元素为终点为止。队列里的元素记录了指向父节点（上一步）的指针

队列不能用STL的queue或deque，要自己写。用一维数组实现，维护一个
队头指针和队尾指针

百练4980 拯救行动

公主被恶人抓走，被关押在牢房的某个地方。牢房用N*M (N, M <= 200)的矩阵来表示。矩阵中的每项可以代表道路（@）、墙壁（#）、和守卫（x）。

英勇的骑士（r）决定孤身一人去拯救公主（a）。我们假设拯救成功的表示是“骑士到达了公主所在的位置”。由于在通往公主所在位置的道路中可能遇到守卫，骑士一旦遇到守卫，必须杀死守卫才能继续前进。

现假设骑士可以向上、下、左、右四个方向移动，每移动一个位置需要1个单位时间，杀死一个守卫需要花费额外的1个单位时间。同时假设骑士足够强壮，有能力杀死所有的守卫。

给定牢房矩阵，公主、骑士和守卫在矩阵中的位置，请你计算拯救行动成功需要花费最短时间。

```
2
7 8
#@######@
#@a#@@r@
#@@#x@@@
@@#@@#@#
#@@@##@@
@#@@@@@
@@@@@@@@
```

百练4980 拯救行动

队列里放以下结构：

```
struct Position
{
        int r,c,L; //L是标记，对于‘@’,都是1,对于‘x’,开始是0，然后由0变成1，算走一步 等于把x 拆成两个点，进入第一点就只能走到第二点，不能往别处走
        int steps;
        Position(int rr,int cc,int LL,int ss):
                r(rr),c(cc),L(LL),steps(ss) {}
        Position() { }
};
```

对于‘@’点，扩展时直接扩展和其相邻的4个点；
对于(‘x’,0)点，扩展时只能扩展出(‘x’,1)
对于(‘x’,1)点，扩展和其相邻的4个点；

```
2
7 8
#@######@
#@a#@@r@
#@@#x@@@
@@#@@#@#
#@@@##@@
@#@@@@@@
@@@@@@@@
```

百练4980 拯救行动

队列里放以下结构：

```
struct Position
{
        int r,c,L; //L是标记，对于‘@’,都是1,对于‘x’,开始是0，然后由0变成1，算走一步 等于把x 拆成两个点，进入第一点就只能走到第二点，不能往别处走
        int steps;
        Position(int rr,int cc,int LL,int ss):
                r(rr),c(cc),L(LL),steps(ss) {}
        Position() { }
};
```

对于‘@’点，扩展时直接扩展和其相邻的4个点；
对于(‘x’,0)点，扩展时只能扩展出(‘x’,1)
对于(‘x’,1)点，扩展和其相邻的4个点；

```
2
7 8
#@######@
#@a#@@r@
#@@#x@@@
@@#@@#@#
#@@@##@@
@#@@@@@@
@@@@@@@@
```

百练6044 鸣人和佐助

已知一张地图（以二维矩阵的形式表示）以及佐助和鸣人的位置。地图上的每个位置都可以走到，只不过有些位置上有大蛇丸的手下(#)，需要先打败大蛇丸的手下才能到这些位置。

鸣人有一定数量的查克拉，每一个单位的查克拉可以打败一个大蛇丸的手下。假设鸣人可以往上下左右四个方向移动，每移动一个距离需要花费1个单位时间，打败大蛇丸的手下不需要时间。如果鸣人查克拉消耗完了，则只可以走到没有大蛇丸手下的位置，不可以再移动到有大蛇丸手下的位置。

佐助在此期间不移动，大蛇丸的手下也不移动。请问，鸣人要追上佐助最少需要花费多少时间？

```
4 4 1
#@##
**##
###+
****
```

百练6044 鸣人和佐助

状态定义为：

（r, c, k），鸣人所在的行，列和查克拉数量

如果队头节点扩展出来的节点是有大蛇手下的节点，则其 k 值比对头的k要减掉 1。如果队头节点的查克拉数量为0, 则不能扩展出有大蛇手下的节点。

```
4 4 1
#@##
**##
###+
****
```

求钥匙的鸣人

不再有大蛇丸的手下。

但是佐助被关在一个格子里，需要集齐k种钥匙才能打开格子里的门门救出他。

K种钥匙散落在迷宫里。有的格子里放有一把钥匙。一个格子最多放一把钥匙。走到放钥匙的格子，即得到钥匙。

鸣人最少要走多少步才能救出佐助。

求钥匙的鸣人

状态：

（r, c, keys） ： 鸣人的行，列，已经拥有的钥匙种数

目标状态 （x, y, K） （x, y）是佐助呆的地方

如果队头节点扩展出来的节点上面有不曾拥有的某种钥匙，则该节点的 keys比队头节点的 keys要加1

Jack and Jill（POJ1729）

- Jack和Jill要从各自的家走到各自的学校，但是他们互相不喜欢对方，因此希望你找到一个行走方案，使得在行走的过程中他们之间的直线最近距离最远。
- 单位时间内每个人都可以从一个格子转移到相邻的（上下左右）四个格子之一，。
- 计算距离的时候只算单位时间移动结束之后两人所在位置的直线距离，不考虑移动过程中的距离。
- 走过的路可以来回走，到达学校后不能再离开学校
- 图例：
  - H：Jack的家；S:Jack的学校
  - h:Jill的家；s:Jill的学校
  - *:不可进入的区域
  - . :可进入的区域

```
..........
...H......
.**...s...
.**.......
.**.......
.**.......
.**.......
.**.......
.**.......
...S..h..*
..........
```

Jack and Jill：状态和状态转移规则

- 将当前Jack和Jill某一时刻各自所在的格子对作为状态。
- 确定搜索顺序：
  - 将所有格子对按照格子对的直线距离从大到小排序。
- 状态转移：从一个格子对，变到另一个格子对

Jack and Jill：状态转移规则

- 从大到小枚举最近距离k，维护一个有效状态集合T，T中包含所有直线距离大于等于k的格子对。（前面的排序起作用了）。
- 对于当前的T，采用广搜，判断是否能从初始状态到达目标状态。
- 问题：
  - 为什么采用广搜？
  - K的初始值？K真的要从大到小枚举吗？

Jack and Jill：状态转移规则

- 从大到小枚举最近距离k，维护一个有效状态集合T，T中包含所有直线距离大于等于k的格子对。（前面的排序起作用了）。
- 对于当前的T，采用广搜，判断是否能从初始状态到达目标状态。
- 问题：
  - 为什么采用广搜？
  - K的初始值？K真的要从大到小枚举吗？(二分即可)

广度优先搜索

八数码问题

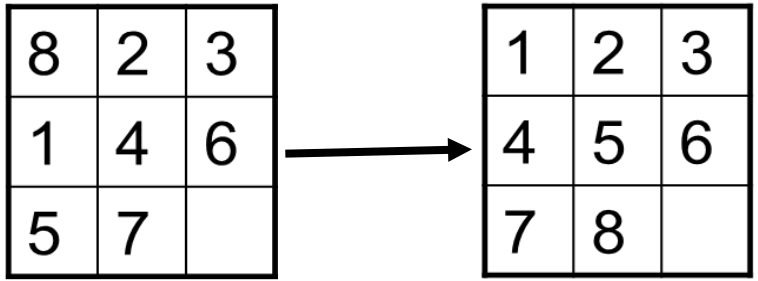
八数码（POJ1077）

- 八数码问题是人工智能中的经典问题

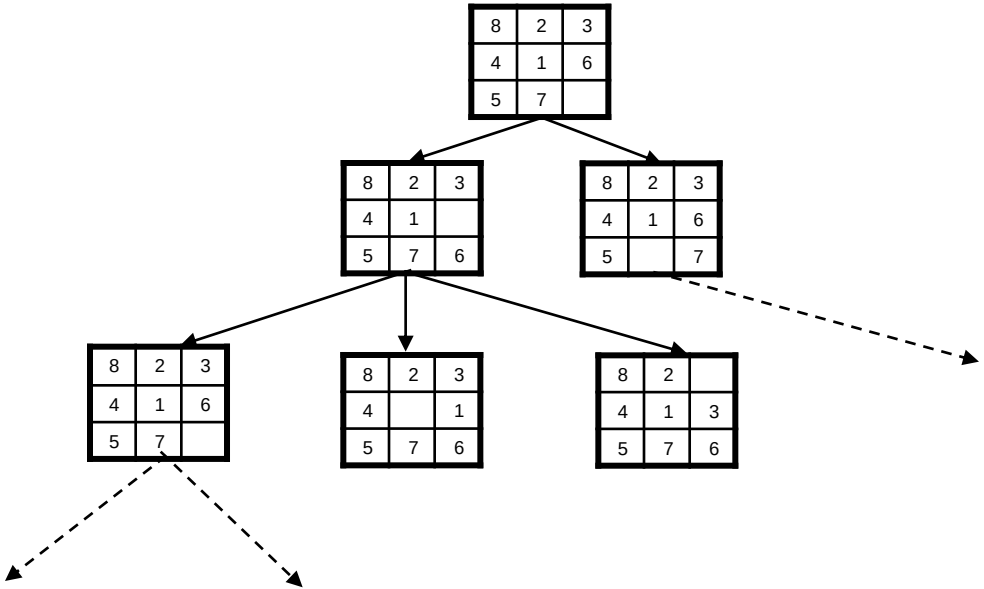
有一个3*3的棋盘，其中有0-8共9个数字，0表示空格，其他的数字可以和0交换位置。求由初始状态到达目标状态

1 2 3
4 5 6
7 8 0

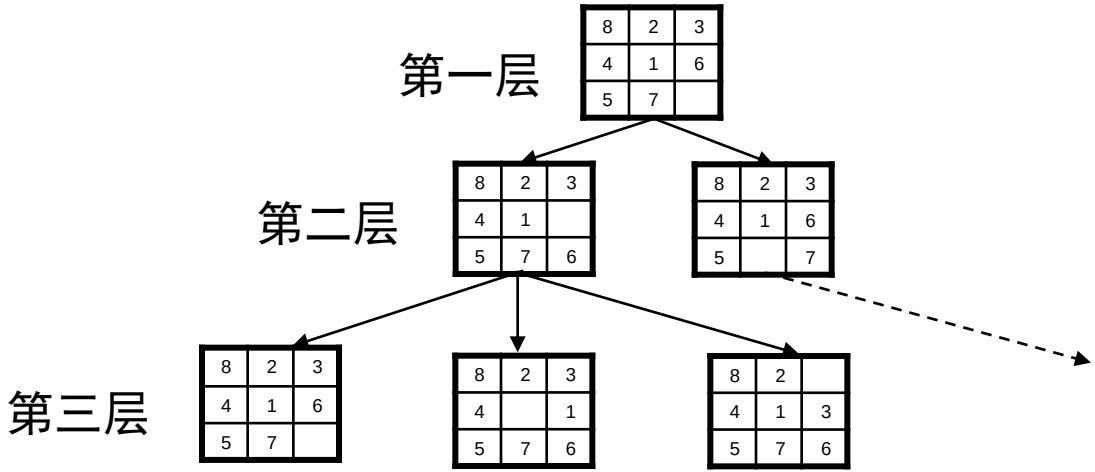
的步数最少的解。



- 状态空间

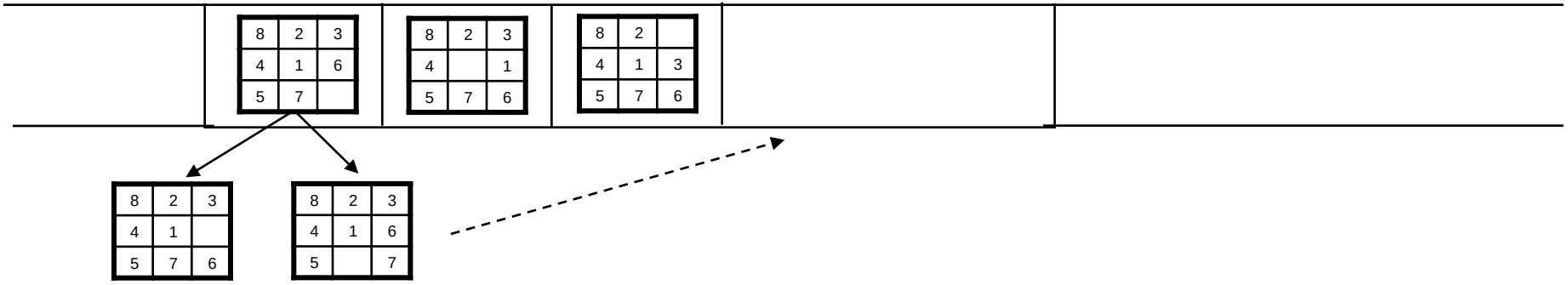


- 广度优先搜索（bfs）
  - 优先扩展浅层节点(状态)，逐渐深入



广度优先搜索

- 用队列保存待扩展的节点
- 从队首队取出节点，扩展出的新节点放入队尾，直到队首出现目标节点（问题的解）

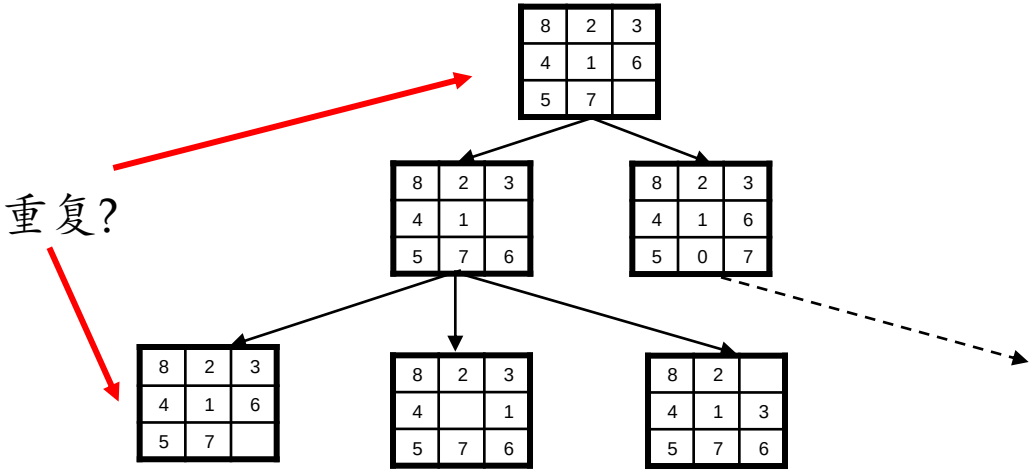


- 广度优先搜索的代码框架

```
BFS()
{
        初始化队列
        while(队列不为空且未找到目标节点)
        {
          取队首节点扩展，并将扩展出的非重复节点放入队尾；
                必要时要记住每个节点的父节点；
        }
}
```

关键问题：判重

- 新扩展出的节点如果和以前扩展出的节点相同，则则个新节点就不必再考虑
- 如何判重？



关键问题：判重

- 状态(节点）数目巨大，如何存储？
- 怎样才能较快判断一个状态是否重复？

用合理的编码表示“状态”，减小存储代价

- 方案一：

8	2	3
4	1	6
5	7	

每个状态用一个字符串存储,

要9个字节, 太浪费了！！！

用合理的编码表示“状态”，减小存储代价

- 方案二：

8	2	3
4	1	6
5	7	

- 每个状态对应于一个9位数，则该9位数最大为876,543,210，小于2^{31}，则int 就能表示一个状态。
- 判重需要一个标志位序列，每个状态对应于标志位序列中的1位，标志位为0表示该状态尚未扩展，为1则说明已经扩展过了
- 标志位序列可以用字符数组a存放。a的每个元素存放8个状态的标志位。最多需要876,543,210位，因此a数组需要876,543,210 /8 + 1个元素，即 109,567,902 字节
- 如果某个状态对应于数x ,则其标志位就是a[x/8]的第x%8位
- 空间要求还是太大!!!!

用合理的编码表示“状态”，减小存储代价

- 方案三：

8	2	3
4	1	6
5	7	

- 将每个状态的字符串形式看作一个9位九进制数，则该9位数最大为$876543210_{(9)}$, 即 $381367044_{(10)}$ 需要的标志位数目也降为$381367044_{(10)}$ 比特,即47,670,881字节。
- 如果某个状态对应于数x ,则其标志位就是a[x/8]的第x%8位
- 空间要求还是有点大！！！！

用合理的编码表示“状态”，减小存储代价

- 方案三：

8	2	3
4	1	6
5	7	

- 状态数目一共只有9! 个，即$362880_{(10)}$个，怎么会需要 $876543210_{(9)}$ 即 $381367044_{(10)}$ 个标志位呢？

用合理的编码表示“状态”，减小存储代价

- 方案三：

8	2	3
4	1	6
5	7	

- 状态数目一共只有9! 个，即$362880_{(10)}$个，怎么会需要 $876543210_{(9)}$ 即 $381367044_{(10)}$ 个标志位呢？
- 如果某个状态对应于数x ,则其标志位就是a[x/8]的第x%8位
- 因为有浪费！例如，$666666666_{(9)}$ 根本不对应于任何状态，也为其准备了标志位！

用合理的编码表示“状态”，减小存储代价

- 方案四：

8	2	3
4	1	6
5	7	

- 把每个状态都看做’0’-’8’的一个排列，以此排列在全部排列中的位置作为其序号。状态用其排列序号来表示
- 012345678是第0个排列，876543210是第9!-1个
- 状态总数即排列总数：9!=362880
- 判重用的标志数组a只需要362,880比特即可。
- 如果某个状态的序号是x ,则其标志位就是a[x/8]的第x%8位

用合理的编码表示“状态”，减小存储代价

- 方案四：

8	2	3
4	1	6
5	7	

- 在进行状态间转移，即一个状态通过某个移动变化到另一个状态时，需要先把int形式的状态（排列序号），转变成字符串形式的状态，然后在字符串形式的状态上进行移动，得到字符串形式的新状态，再把新状态转换成int形式（排列序号）。

用合理的编码表示“状态”，减小存储代价

- 方案四：

8	2	3
4	1	6
5	7	

- 需要编写给定排列（字符串形式）求序号的函数
- 需要编写给定序号，求该序号的排列（字符串形式）的函数

给定排列求序号：

整数 1,2…k的一个排列：
a1 a2 a3 …ak
求其序号
基本思想：算出有多少个排列比给定排列小。
先算1到a1-1放在第1位，会有多少个排列：(a1-1)*((k-1)!)
再算a1不变，1到a2-1 放在第2位(左边出现过的不能再用)，会有多少个排列：(a2-1)*((k-2)!)
再算a1，a2不变，1到a3-1 放在第3位，会有多少个排列
….全加起来。 时间复杂度：$O(n^2)$

3241
1,2放在第一位，有 2*3! = 12 种
3在第一位，1放在第2位，有 2! = 2种
32？ 1放在第3位，有 1种
=>前面共 12+2+1 = 15种。所以 3241是第16个排列

给定序号n求排列：

1234的排列的第9号

第一位假定是1，共有3!种，没有到达9，所以第一位至少是2

第一位是2，一共能数到 3!+3!号，>= 9，所以第一位是2

第二位是1，21??，一共能数到 3!+2! = 8 不到9，所以第二位至少是 3

第二位是3，23??，一共能数到 3!+2!+2! >= 9，因此第二位是3

第三位是1，一共能数到3!+2!+1 = 9，所以第三位是1，第四位是 4

答案：2314

时间复杂度：$O(n^2)$

- 时间与空间的权衡
  - 对于状态数较小的问题，可以用最直接的方式编码以空间换时间
  - 对于状态数太大的问题，需要利用好的编码方法以时间换空间
  - 具体问题具体分析

用广搜解决八数码问题(POJ1077)

输入数据：
2 3 4 1 5 0 7 6 8
输出结果：
ulldrurdllurdruldr

输出数据是一个移动序列，使得移动后结果变成

1 2 3
4 5 6
7 8

输入样例：

2 3 4
1 5
7 6 8

移动序列中
u 表示使空格上移
d 表示使空格下移
r 表示使空格右移
l 表示使空格左移

八数码问题有解性的判定

- 八数码问题的一个状态实际上是0~8的一个排列，对于任意给定的初始状态和目标，不一定有解，即从初始状态不一定能到达目标状态。
  - 因为排列有奇排列和偶排列两类，从奇排列不能转化成偶排列或相反。
- 如果一个数字0~8的随机排列，用F(X) (X!=0)表示数字X前面比它小的数的个数，全部数字的F(X)之和为Y=∑(F(X))，如果Y为奇数则称该排列是奇排列，如果Y为偶数则称该排列是偶排列。
  - 871526340排列的 Y=0+0+0+1+1+3+2+3=10，10是偶数，所以是偶排列。
  - 871625340排列的Y=0+0+0+1+1+2+2+3=9 9是奇数，所以是奇排列。
  - 因此，可以在运行程序前检查初始状态和目标状态的奇偶性是否相同，相同则问题可解，应当能搜索到路径。否则无解。

八数码问题有解性的判定

证明：移动0的位置，不改变排列的奇偶性

a1 a2 a3 a4 0 a5 a6 a7 a8 a9

0向上移动：

a1 0 a3 a4 a2 a5 a6 a7 a8 a9

八数码问题 ，单向广搜，最简单做法，POJ 891MS HDU TLE

```
#include <iostream>
#include <bitset>
#include <cstring>
using namespace std;
int goalStatus;  //目标状态
bitset<362880> Flags; //节点是否扩展的标记
const int MAXS = 400000;
char result[MAXS];   //结果
struct Node {
        int status; //状态，即排列的编号
        int father; //父节点指针
        char move;   //父节点到本节点的移动方式 u/d/r/l
        Node(int s,int f,char m):status(s), father(f),move(m) { }
        Node() { }
};
Node myQueue[MAXS];  //状态队列，状态总数362880
int qHead;    int qTail; //队头指针和队尾指针
char sz4Moves[] = "udrl"; //四种动作
unsigned int factorial[21];  //存放0-20的阶乘。21的阶乘unsigned放不下了
```

```
unsigned int GetPermutationNumForInt(int * perInt,int len)
//perInt里放着整数 0 到 len-1 的一个排列，求它是第几个排列
//len不能超过21
{
        unsigned int num = 0;
        bool used[21];
        memset(used,0,sizeof(bool)*len);
        for( int i = 0;i < len; ++ i ) {
                unsigned int n = 0;
                for( int j = 0; j < perInt[i]; ++ j) {
                        if(! used[j] )
                                ++n;
                }
                num += n * factorial[len-i-1];
                used[perInt[i]] = true;
        }
        return num;
}
```

```
template< class T>
unsigned int GetPermutationNum( T s1, T s2,int len)
//给定排列，求序号 。[s1,s1+len)里面放着第0号排列，[s2,s2+len)是要求序号的排列
//两者必须一样长，len不能超过21
//排列的每个元素都不一样。返回排列的编号
{
        int perInt[21]; //要转换成 [0,len-1] 的整数的排列
        for( int i = 0;i < len; ++i )
                for( int j = 0; j < len; ++j )  {
                        if( * ( s2 + i )  == * (s1+j)) {
                                perInt[i] = j;
                                break;
                        }
                }
        unsigned int num = GetPermutationNumForInt(perInt,len);
        return num;
}
```

```
template <class T>
void GenPermutationByNum(T   s1, T  s2,int len, unsigned int No)
//根据排列编号，生成排列 len不能超过21
{ //[s1,s1+len) 里面放着第0号 permutation,，排列的每个元素都不一样
        int perInt[21]; //要转换成 [0,len-1] 的整数的排列
        bool used[21];
        memset(used,0,sizeof(bool)*len);
        for(int i = 0;i < len; ++ i ) {
                unsigned int tmp;  int n = 0;int j;
                for( j = 0; j < len; ++j ) {
                        if( !used[j] ) {
                                if( factorial[len - i - 1] >= No+1)  break;
                                else No -= factorial[len - i - 1];
                        }
                }
                perInt[i] = j;
                used[j] = true;
        }
```

```
        for( int i = 0;i < len; ++i )
                * ( s2 + i ) = * ( s1 + perInt[i]);

}
int StrStatusToIntStatus( const char * strStatus)
{//字符串形式的状态，转换为整数形式的状态(排列序号）
        return GetPermutationNum( "012345678",strStatus,9);
}
void IntStatusToStrStatus( int n, char * strStatus)
{//整数形式的状态(排列序号），转换为字符串形式的状态
        GenPermutationByNum((char*)"012345678",strStatus,9,n);
}
```

```
int NewStatus( int nStatus, char cMove) {
//求从nStatus经过 cMove 移动后得到的新状态。若移动不可行则返回-1
        char szTmp[20];    int nZeroPos;
        IntStatusToStrStatus(nStatus,szTmp);
        for( int i = 0;i < 9; ++ i )
                if( szTmp[i] == '0' ) {
                        nZeroPos = i;
                        break;
                } //返回空格的位置
        switch( cMove) {
                case 'u': if( nZeroPos - 3 < 0 )  return -1; //空格在第一行
                        else {     szTmp[nZeroPos] = szTmp[nZeroPos - 3];
                                szTmp[nZeroPos - 3] = '0'; }
                        break;
                case 'd': if( nZeroPos + 3 > 8 )  return -1; //空格在第三行
                        else {     szTmp[nZeroPos] = szTmp[nZeroPos + 3];
                                szTmp[nZeroPos + 3] = '0';}
                        break;
```

```
            case 'l':   if( nZeroPos % 3 == 0)  return -1; //空格在第一列
                        else {      szTmp[nZeroPos] = szTmp[nZeroPos -1];
                                    szTmp[nZeroPos -1 ] = '0'; }
                        break;
            case 'r':   if( nZeroPos % 3 == 2)  return -1; //空格在第三列
                        else {      szTmp[nZeroPos] = szTmp[nZeroPos + 1];
                                    szTmp[nZeroPos + 1 ] = '0';             }
                        break;
        }
        return StrStatusToIntStatus(szTmp);
}
```

```
bool Bfs(int nStatus) { //寻找从初始状态nStatus到目标的路径
        int nNewStatus;      Flags.reset(); //清除所有扩展标记
        qHead = 0;           qTail = 1;
        myQueue[qHead] = Node(nStatus,-1,0);
        while ( qHead != qTail) { //队列不为空
                nStatus = myQueue[qHead].status;
                if( nStatus == goalStatus ) //找到目标状态
                        return true;
                for( int i = 0;i < 4;i ++ ) { //尝试4种移动
                        nNewStatus = NewStatus(nStatus,sz4Moves[i]);
                        if( nNewStatus == -1 )   continue; //不可移，试下一种
                        if( Flags[nNewStatus] )  continue; //扩展标记已经存在，则不入队
                        Flags.set(nNewStatus,true); //设上已扩展标记
                        myQueue[qTail++] =
                              Node(nNewStatus,qHead,sz4Moves[i]); //新节点入队列
                }
                qHead ++;
        }
        return false;
}
```

```
int main(){
        factorial[0] = factorial[1] =1;
        for(int i = 2;i < 21; ++i )
                factorial[i] = i * factorial[i-1];
        goalStatus = StrStatusToIntStatus("123456780");
        char szLine[50];  char szLine2[20];
        while( cin.getline(szLine,48)) {
                int i,j;
                //将输入的原始字符串变为数字字符串
                for( i = 0, j = 0; szLine[i]; i ++ ) {
                        if( szLine[i] != ' ' ) {
                                if( szLine[i] == 'x' )  szLine2[j++] = '0';
                                else  szLine2[j++] = szLine[i];
                        }
                }
                szLine2[j] = 0; //字符串形式的初始状态
                int sumGoal = 0;  //从此往后用奇偶性判断是否有解
```

```
for( int i = 0;i < 8; ++i )
        sumGoal += i -1;
int sumOri = 0;
for( int i = 0;i < 9 ; ++i ) {
        if( szLine2[i] == '0')
                continue;
        for( int j = 0; j < i; ++j ) {
                if( szLine2[j] < szLine2[i] && szLine2[j] != '0' )
                        sumOri ++;
        }
}

if( sumOri %2 != sumGoal %2 ) {
        cout << "unsolvable" << endl;
        continue;
}
//上面用奇偶性判断是否有解
```

```
                if( Bfs(StrStatusToIntStatus(szLine2))) {
                        int nMoves = 0;
                        int nPos = qHead;
                        do {  //通过father找到成功的状态序列，输出相应步骤
                                result[nMoves++] = myQueue[nPos].move;
                                nPos = myQueue[nPos].father;
                        } while( nPos); //nPos = 0 说明已经回退到初始状态了
                        for( int i = nMoves -1; i >= 0; i -- )
                                cout << result[i];
                }
                else
                        cout << "unsolvable" << endl;
        }
}
```

时间复杂度，就是状态总数

广度优先搜索

八数码问题进一步讨论

广搜与深搜的比较

- 广搜一般用于状态表示比较简单、求最优策略的问题
  - 优点：是一种完备策略，即只要问题有解，它就一定可以找到解。并且，广度优先搜索找到的解，还一定是路径最短的解。
  - 缺点：盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用的节点，因此其搜索效率较低。需要保存所有扩展出的状态，占用的空间大
- 深搜几乎可以用于任何问题
  - 只需要保存从起始状态到当前状态路径上的节点
- 根据题目要求凭借自己的经验和对两个搜索的熟练程度做出选择

71

八数码问题 ： 如何加快速度

POJ 1077为单组数据

HDU 1043 为多组数据

裸的广搜在POJ 能过，在HDU会超时

八数码问题 ： 如何加快速度

1. 双向广搜(HDU能过）

从两个方向以广度优先的顺序同时扩展

2. 针对本题预处理 (HDU能过）

因为状态总数不多，只有不到40万种，因此可以从目标节点开始，进行一遍彻底的广搜，找出全部有解状态到目标节点的路径。

3. A* 算法(HDU能过）

双向广度优先搜索(DBFS)

- DBFS算法是对BFS算法的一种扩展。

  - BFS算法从起始节点以广度优先的顺序不断扩展，直到遇到目的节点
  - DBFS算法从两个方向以广度优先的顺序同时扩展，一个是从起始节点开始扩展，另一个是从目的节点扩展，直到一个扩展队列中出现另外一个队列中已经扩展的节点，也就相当于两个扩展方向出现了交点，那么可以认为我们找到了一条路径。

双向广度优先搜索(DBFS)

- 比较
  - DBFS算法相对于BFS算法来说，由于采用了双向扩展的方式，搜索树的宽度得到了明显的减少，时间复杂度和空间复杂度上都有提高！

双向广度优先搜索(DBFS)

- 比较
  - DBFS算法相对于BFS算法来说，由于采用了双向扩展的方式，搜索树的宽度得到了明显的减少，时间复杂度和空间复杂度上都有提高！
  - 假设1个结点能扩展出n个结点，单向搜索要m层能找到答案，那么扩展出来的节点数目就是: $(1-n^m)/(1-n)$

双向广度优先搜索(DBFS)

- 比较
  - DBFS算法相对于BFS算法来说，由于采用了双向扩展的方式，搜索树的宽度得到了明显的减少，时间复杂度和空间复杂度上都有提高！
  - 假设1个结点能扩展出n个结点，单向搜索要m层能找到答案，那么扩展出来的节点数目就是: $(1-n^m)/(1-n)$
  - 双向广搜，同样是一共扩展m层，假定两边各扩展出m/2层，则总结点数目 $2 * (1-n^{m/2})/(1-n)$

双向广度优先搜索(DBFS)

- 比较
  - DBFS算法相对于BFS算法来说，由于采用了双向扩展的方式，搜索树的宽度得到了明显的减少，所以在算法的时间复杂度和空间复杂度上都有较大的优势！
  - 假设1个结点能扩展出n个结点，单向搜索要m层能找到答案，那么扩展出来的节点数目就是: $(1-n^m)/(1-n)$
  - 双向广搜，同样是一共扩展m层，假定两边各扩展出m/2层，则总结点数目 $2 * (1-n^{m/2})/(1-n)$
  - 每次扩展结点总是选择结点比较少的那边进行扩展，并不是机械的两边交替。

DBFS的框架(1)

一、双向广搜函数:

```
void dbfs()
{
    1. 将起始节点放入队列q0,将目标节点放入队列q1;
    2. 当两个队列都未空时，作如下循环:
        1) 如果队列q0里的节点比q1中的少,则扩展队列q0;
        2) 否则扩展队列q1
    3. 如果队列q0未空，不断扩展q0直到为空;
    4. 如果队列q1未空，不断扩展q1直到为空;
}
```

DBFS的框架(2)

二、扩展函数

int expand(i) //其中i为队列的编号，0或1

{

取队列qi的头结点H;

对H的每一个相邻节点adj:

1 如果adj已经在队列q_i之中出现过，则抛弃adj;

2 如果adj在队列q_i中未出现过，则:

1） 将adj放入队列q_i;

2) 如果adj 曾在队列q_{1-i}中出现过, 则：输出找到的路径

}

需要两个标志序列，分别记录节点是否出现在两个队列中

八数码问题，单向广搜POJ 891MS 双向广搜 POJ 63MS HDU 通过!

//下面程序是1号字，所以看不见。请全选，拷贝粘贴到别处后再看

广搜与深搜的比较

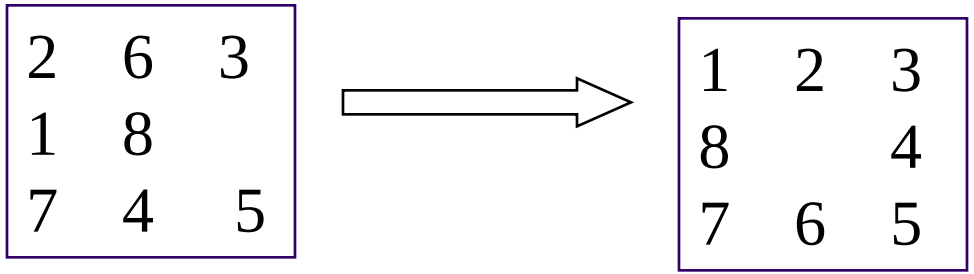
- 广搜一般用于状态表示比较简单、求最优策略（步数）的问题
  - 优点：是一种完备策略，即只要问题有解，它就一定可以找到解。并且，广度优先搜索找到的解，还一定是步数最少的解。
  - 缺点：盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用的节点，因此其搜索效率较低。需要保存所有扩展出的状态，占用的空间大
- 深搜几乎可以用于任何问题
  - 只需要保存从起始状态到当前状态路径上的节点
- 根据题目要求凭借自己的经验和对两个搜索的熟练程度做出选择

82

启发式搜索算法(A算法)

- 在BFS算法中，若对每个状态n都设定估价函数f(n)=g(n)+h(n)，并且每次从Open表中选节点进行扩展时，都选取f值最小的节点，则该搜索算法为启发式搜索算法，又称A算法。
- g(n)：从起始状态到当前状态n的代价
- h(n)：从当前状态n到目标状态的估计代价

A算法的例子--八数码



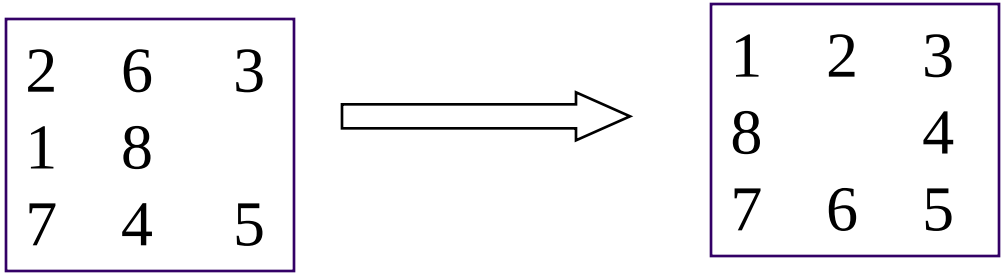
定义估价函数：

f(n) = g(n) + h(n)

g(n)为从初始节点到当前节点的步数

h(n)为当前节点“不在位”的方块数

h计算举例



2,6,1,8,4,5都不在位，因此h(n) = 6

A*算法

- A算法中的估价函数若选取不当，则可能找不到解，或找到的解也不是最优解。因此，需要对估价函数做一些限制，使得算法确保找到最优解（步数，即状态转移次数最少的解）。A*算法即为对估价函数做了特定限制，且确保找到最优解的A算法。

A*算法

- f*(n)=g*(n) +h*(n)

  f*(n)：从初始节点S0出发，经过节点n到达目标节点的最小步数 (真实值）。

  g*(n): 从S0出发，到达n的最少步数（真实值）

  h*(n): 从n出发，到达目标节点的最少步数（真实值）

  估价函数f(n)则是f*(n)的估计值。

A*算法

- f (n)=g(n) +h(n) ,且满足以下限制：

g(n)是从s0到n的真实步数（未必是最优的），因此：

g(n)>0 且g(n)>=g*(n)

h(n)是从n到目标的估计步数。估计总是过于乐观的，即

h(n)<= h*(n)

且h(n)相容，则A算法转变为A*算法。A*正确性证明略。

A*算法

h(n)的相容:

如果h函数对任意状态s1和s2还满足：

h(s1) <= h(s2) + c(s1,s2)

c(s1,s2）是s1转移到s2的步数，则称h是相容的。

h相容能确保f递增，这样A*能更高效找到最优解。

h相容 =>

g(s1) + h(s1) <= g(s1) + h(s2) +c(s1,s2) = g(s2)+h(S2)

=>

f(s1) <= f(s2)

即f是递增的。

A * 算法伪代码(在节点信息中记录了其父节点)：

```
open=[Start]
closed=[]
while open不为空 {
    从open中取出估价值f最小的结点n
    if n == Target then
        return 从Start到n的路径 // 找到了！！！
    else {
        for n的每个子结点x {
            if x in open {
                计算新的f(x)
                比较open表中的旧f(x)和新f(x)
                if 新f(x) < 旧f(x) {
                    删掉open表里的旧f(x)，加入新f(x)
                    }
            }
            else if x in closed {
                计算新的f(x)
                比较closed表中的旧f(x)和新f(x)
                if 新f(x) < 旧f(x) {
                    remove x from closed
                    add x to open
                }
            }  // 比较新f(x)和旧f(x) 实际上比的就是新旧g(x),因h(x)相等
            else {
                // x不在open，也不在close，是遇到的新结点
                计算f(x) add x to open
                }
        }
        add n to closed
    }
}
//open表为空表示搜索结束了，那就意味着无解！
```

A*算法

A*算法的搜索效率很大程度上取决于估价函数h(n)。一般说来，在满足h(n)≤h*(n)的前提下，h(n)的值越大越好。

八数码问题：

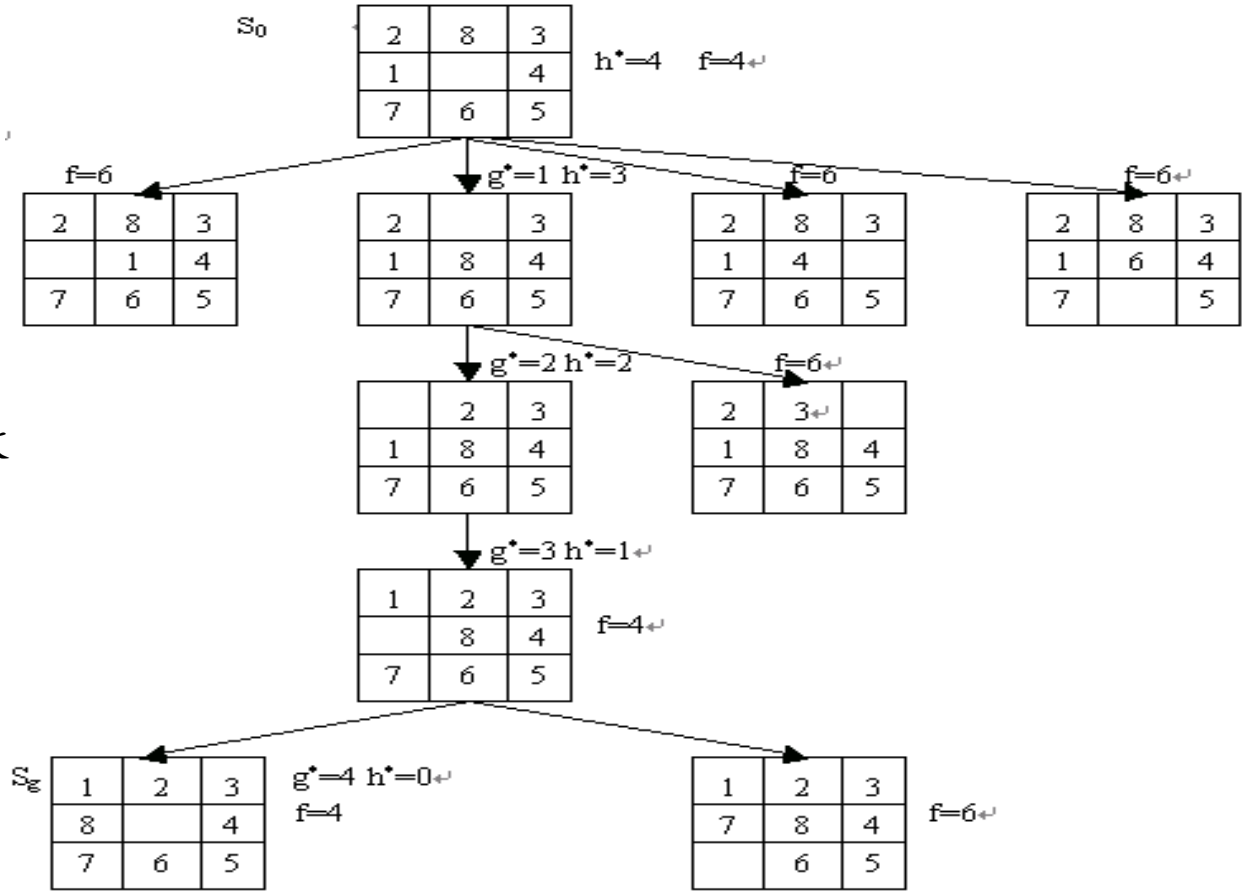
方案一. h(n) 为不在位的数字个数

方案二. h(n) 为不在位的数字到其该呆的位置的曼哈顿距离和

后者优于前者

A*算法解决八数码问题

h(n) 为不在位的数字到其该呆的位置的曼哈顿距离和



```
//八数码问题， AStar POJ 250MS HDU 3734MS
#include <iostream>
#include <bitset>
#include <cstring>
#include <set>
using namespace std;
const int DIGIT_NUM = 9;
int const NODES =  362880 + 20;
int nGoalStatus;  //目标状态
struct Node {
        int status;     int f;          int g;          int h;
        int parent; // parent代表一个状态，而不是状态的索引
        char move; //经由何种动作到达本状态
};
bool operator < ( const Node & n1,const Node & n2) {          return n1.f < n2.f;  }
multiset<Node > open;
multiset<Node > closed;//也许用vector也可以。用multiset只是在从 closed表里删除元素时快些。其实closed表就是一个 362880元素的数组，可能更快
bitset<NODES> inOpen;
bitset<NODES> inClosed;
multiset<Node>::iterator openIdx[NODES]; ' openIdx[i] 就是状态 i 在open表里的地址
multiset<Node>::iterator closedIdx[NODES];
char szResult[NODES];   //结果
char szMoves[NODES]; //移动步骤 ：  u/d/r/l
char sz4Moves[] = "udrl";//四种动作
```

```
template< class T>
unsigned int GetPermutationNum(T * first, T * permutation,int len)
{ //permutation编号从0开始算, [first,first+len) 里面放着第0号 permutation，排列的每个元素都不一样
  //返回排列的编号
        unsigned int factorial[21] = {
1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,1932053504,1278945280,
2004310016,2004189184,4006445056,3396534272,109641728,2192834560 };
        bool used[21] = {0};
        int perInt[21]; //要转换成 [0,len-1] 的整数的排列
        for( int i = 0;i < len; ++i )
                for( int j = 0; j < len; ++j )  {
                        if( * ( permutation + i )  == * (first+j)) {
                                perInt[i] = j;
                                break;
                        }
                }
        unsigned int num = 0;
        for( int i = 0;i < len; ++ i ) {
                unsigned int n = 0;
                for( int j = 0; j < perInt[i]; ++ j) {
                        if(! used[j] )
                                ++n;
                }
```

```
                        num += n * factorial[len-i-1];
                        used[perInt[i]] = true;
                }
                return num;

}
template <class T>
void GenPermutationByNum(T * first, T * permutation,int len, unsigned int No)
//根据排列编号，生成排列
{ //[first,first+len) 里面放着第0号 permutation,，排列的每个元素都不一样
                unsigned int factorial[21] = {
1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,1932053504,1278945280,
                                        2004310016,2004189184,4006445056,3396534272,109641728,2192834560
};
                bool used[21] = {0};
                int perInt[21]; //要转换成 [0,len-1] 的整数的排列
                for(int i = 0;i < len; ++ i ) {
                        unsigned int tmp;
                        int n = 0;
                        int j;
                        for( j = 0; j < len; ++j ) {
                                if( !used[j] ) {
                                        if( factorial[len - i - 1] >= No+1)
                                                break;
```

```
                                                    else
                                                                No -= factorial[len - i - 1];
                                                    }
                                        }
                                        perInt[i] = j;
                                        used[j] = true;
                    }
                    for( int i = 0;i < len; ++i )
                                        * ( permutation + i ) = * ( first + perInt[i]);
}
int StrStatusToIntStatus( const char * strStatus)
{
                    return GetPermutationNum( "012345678",strStatus,9);
}
void IntStatusToStrStatus( int n, char * strStatus)
{
                    GenPermutationByNum((char*)"012345678",strStatus,9,n);
}
int MyAbs(int a) {
                    return a >= 0?a:-a;
}
```

```
int h(char * status)
{//h的值是不在位的数字到它该呆的位置的曼哈顿距离之和
        int sum  = 0;
        for( int i = 0;i < 9; ++i ) {
                if( status[i] != '0') {
                        int rightx = ( status[i] - '1' )/3;
                        int righty = ( status[i] - '1' )%3;
                int nowx = i / 3;
                        int nowy = i % 3;
                        sum += MyAbs(rightx - nowx) + MyAbs(righty-nowy);
                }
        }
        return sum;
}
int NewStatus( int nStatus, char cMove) {
//求从nStatus经过 cMove 移动后得到的新状态。若移动不可行则返回-1
        char szTmp[20];
        int nZeroPos;
        IntStatusToStrStatus(nStatus,szTmp);
        for( int i = 0;i < 9; ++ i )
                if( szTmp[i] == '0' ) {
                        nZeroPos = i;
                        break;
                } //返回空格的位置
```

```
    switch( cMove) {
                case 'u': if( nZeroPos - 3 < 0 )  return -1; //空格在第一行
                          else {          szTmp[nZeroPos] = szTmp[nZeroPos - 3];
                                          szTmp[nZeroPos - 3] = '0';              }
                          break;
                case 'd':       if( nZeroPos + 3 > 8 )  return -1; //空格在第三行
                          else {     szTmp[nZeroPos] = szTmp[nZeroPos + 3];
                                          szTmp[nZeroPos + 3] = '0';              }
                          break;
                case 'l':       if( nZeroPos % 3 == 0)  return -1; //空格在第一列
                          else {          szTmp[nZeroPos] = szTmp[nZeroPos -1];
                                          szTmp[nZeroPos -1 ] = '0';              }
                          break;
                case 'r':       if( nZeroPos % 3 == 2)  return -1; //空格在第三列
                          else {          szTmp[nZeroPos] = szTmp[nZeroPos + 1];
                                          szTmp[nZeroPos + 1 ] = '0';             }
                          break;
    }
    return StrStatusToIntStatus(szTmp);
}
```

```
Node AStar(int nStatus){
        open.clear();        closed.clear();        inOpen.reset();        inClosed.reset();
        char strStatus[20];
        IntStatusToStrStatus(nStatus,strStatus);
        Node nd;    nd.status = nStatus;    nd.parent = -1;
        nd.g = 0;    nd.move = -1;        nd.h = h(strStatus);
        nd.f = nd.g + nd.h;
        open.insert(nd);
        inOpen.set(nStatus,true);
        openIdx[nStatus] = open.begin();
        while ( !open.empty()) { //队列不为空
                nd = * open.begin();
                open.erase(open.begin());
                inOpen.set(nd.status,false);
                multiset<Node>::iterator p = closed.insert(nd);
                inClosed.set(nd.status,true);
                closedIdx[nd.status] = p;
                if( nd.status == nGoalStatus )
                        return nd;
                for( int i = 0;i < 4;i ++ ) { //尝试4种移动
                        Node newNd;
                        newNd.status = NewStatus(nd.status,sz4Moves[i]);
                        if( newNd.status == -1 )  continue; //不可移，试下一种
                        IntStatusToStrStatus(newNd.status,strStatus);
```

```
newNd.g = nd.g + 1;
newNd.h = h(strStatus);
newNd.f = newNd.g + newNd.h;
newNd.parent = nd.status;
newNd.move = sz4Moves[i];
if( inOpen[newNd.status ]) {
        Node tmp = * openIdx[newNd.status];
        if( tmp.f > newNd.f ) {
                open.erase(openIdx[newNd.status]);
                p = open.insert(newNd);
                openIdx[newNd.status] = p;
        }
}
else if( inClosed[newNd.status]) {
        Node tmp = * closedIdx[newNd.status];
        if( tmp.f > newNd.f ) {
                closed.erase(closedIdx[newNd.status]);
                inClosed.set(newNd.status,false);
                p = open.insert(newNd);
                openIdx[newNd.status] = p;
                inOpen.set(newNd.status,true);
        }
}
```

```
                                        else {
                                                p = open.insert(newNd);
                                                openIdx[newNd.status] = p;
                                                inOpen.set(newNd.status,true);
                                        }
                                }
                        }
                        nd.status = -1;
                        return nd;
}
int main(){
                        nGoalStatus = StrStatusToIntStatus("123456780");
                        char szLine[50];  char szLine2[20];
                        while( cin.getline(szLine,48) ) {
                                int i,j;
                                for( i = 0, j = 0; szLine[i]; i ++ ) {
                                        if( szLine[i] != ' ' ) {
                                                if( szLine[i] == 'x' )  szLine2[j++] = '0';
                                                else  szLine2[j++] = szLine[i];
                                        }
                                }
                                szLine2[j] = 0;
                                int sumGoal = 0;
```

```
for( int i = 0;i < 8; ++i )
		sumGoal += i -1;
int sumOri = 0;
for( int i = 0;i < 9 ; ++i ) {
		if( szLine2[i] == '0‘)			continue;
		for( int j = 0; j < i; ++j ) {
			if( szLine2[j] < szLine2[i] && szLine2[j] != '0' )
				sumOri ++;
		}
}
if( sumOri %2 != sumGoal %2 ) {
		cout << "unsolvable" << endl;
		continue;
}
Node nd = AStar(StrStatusToIntStatus(szLine2));
if( nd.status != -1 ) {
		int nMoves = 0;
		multiset<Node>::iterator pos;
		pos = closedIdx[nd.status];
		do {		if( pos->move != -1 ) {
				szResult[nMoves++] = pos->move;
				pos = closedIdx[ pos->parent];
			}
		} while(pos->move != -1);
```

```
                                        for( int i = nMoves -1; i >= 0; i -- )
                                                cout << szResult[i];
                                        cout << endl;
                                }
                                else
                                        cout << "unsolvable" << endl;
                }
                return 0;
}

/*
open 队列是set，按f值排序，用set存放每个节点.删除队头 元素就是删除begin
拿出来以后，放到closed表，更新 inopen标志和inclosed标志，还有closeIdx数组
closed队列 也是set, 按f值排序，
建立一个数组，数组元素就是status在open队列里面的迭代器(假设迭代器不因为队列元素的增删而改变）
,status直接作为数组下标
建立一个数组，数组元素就是status在closed队列里面的迭代器(假设迭代器不因为队列元素的增删而改变）
,status直接作为数组下标

open表里表头拿出来的元素是 goalStatus,然后根据里面记录的 parent,(就是状态），到 closed里面找到其对应
元素，然后根据其记录的 parent 和 move，找出问题的解
*/
```

A*算法示例

估价函数选取的另一例子：
（ACM/ICPC Regional Contest Kanpur 2001）：

有一些打乱的段落，比如：
p2,p3,p4,p7, p1, p5,p6
每次只能拷贝若干个连续的段落，然后剪切到某个段落前面，最后要形成
p1,p2,p3,p4,p5,p6,p7
问最少要拷贝粘贴多少次。

A*算法

h(s)选为在状态s下，未“就位”的段落数目？

h(s)选为在状态s下，未“就位”的段落到其该呆的位置的距离之和？

A*算法

h(s)选为在状态s下，未“就位”的段落数目？

h(s)选为在状态s下，未“就位”的段落到其该呆的位置的距离之和？

都不行，h会减少太快，即从某个状态s到s’，可能导致 g的值加1，然而h的值减少很多，这样f(s’）< f(s),f的递增性被破坏，搜索效率降低。

A*算法

考虑以下h(s):

H(s) = s的排列方式下，后继段落错误的段落数目

H(1,3,2) = 2 (1的后继是3错,3的后继是2错）
H(3,2,1) = 2
H(4,2,3,1) = 2 (2的后继对)

A*算法

将连续段落集合 S 从p1后面剪切粘贴到p2后面，后继正确性受到影响的段落最多3个：

p1,p2和S中最后的一个段落

因此从s->s’，h的值最多减少3.但是g的值只增加1，h还是不相容，怎么办？

A*算法

将连续段落集合 S 从p1后面剪切粘贴到p2后面，后继正确性受到影响的段落最多3个：

p1,p2和S中最后的一个段落，

因此从s->s’，h的值最多减少3.但是g的值只增加1，h还是不相容，怎么办？

f(s) = 3 * g(s) + h(s)

POJ上可用A*算法解决的题：

1376
1324
1084
2449
1475

迭代加深搜索算法

- 算法思路
  - 总体上按照深度优先算法方法进行
  - 对搜索深度需要给出一个深度限制dm，当深度达到了dm的时候，如果还没有找到解答，就停止对该分支的搜索，换到另外一个分支继续进行搜索。
  - dm从1开始，从小到大依次增大（因此称为迭代加深）
- 迭代加深搜索是最优的，也是完备的
- 重复搜索深度较小的节点，是否浪费？

IDA*：迭代加深的A*算法

- IDA*的基本思路是：首先将初始状态结点的估价函数H值设为阈值maxH，然后进行深度优先搜索，搜索过程中忽略所有H值大于maxH的结点；如果没有找到解，则加大阈值maxH，再重复上述搜索，直到找到一个解。在保证H值的计算满足A*算法的要求(相容，递增）下，可以证明找到的这个解一定是最优解。在程序实现上，IDA* 要比 A* 方便，因为不需要保存结点，不需要判重复，也不需要根据 H值对结点排序，占用空间小。
  在一般的问题中是这样使用IDA*算法的，当前局面的估价函数值+当前的搜索深度 > 预定义的最大搜索深度时，就停止继续往下搜索。

适用题目： poj2286 the rotation game

IDA*：迭代加深的A*算法

- IDA*算法如下：

```
Procedure  IDA*
Begin
    初始化当前的深度限制 c = 1;
2:  把初始节点压入栈; 并假定 c'= ∝;
    While 栈不空do
        Begin
          弹出栈顶元素n
                If  n= goal, then结束, 返回n以及从初始节点到n的路径。
                Else  do
             Begin
                For n 的每个子节点n'
                    If   f ( n' ) ≤ c, then 把n'压入栈
                    Else c'=min( c', f ( n' ) )
                End for
              End
    End While
    If 栈为空并且c' 未变化 , then 停止并退出;
    If 栈为空并且c'≠ ∝, then c = c', 并返回2。
End
```

c'是本次搜索中发现的f值大于c的节点中，最小的f值

IDA*与A*

- IDA*与A*算法相比，主要的优点是对于内存的需求
  - A*算法需要指数级数量的存储空间，因为没有深度方面的限制
  - 而IDA*算法只有当节点n的所有子节点n'的f(n')小于限制值c时才扩展它，这样就可以节省大量的内存。
- 另外，IDA*不需要对扩展出的节点按启发值排序。