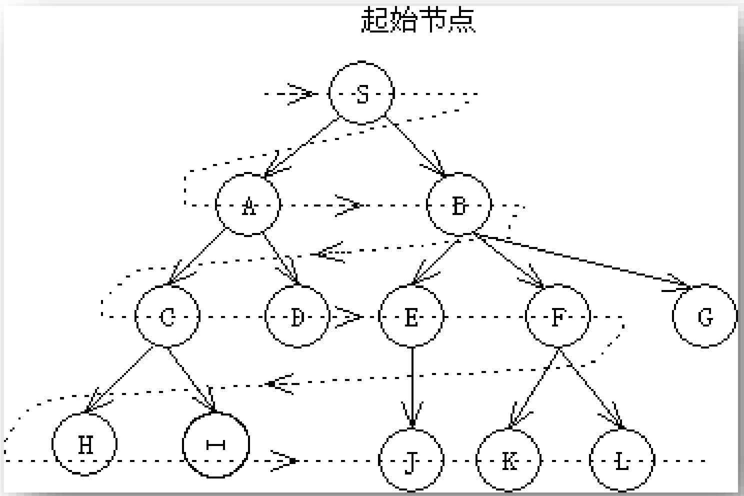
广度优先搜索

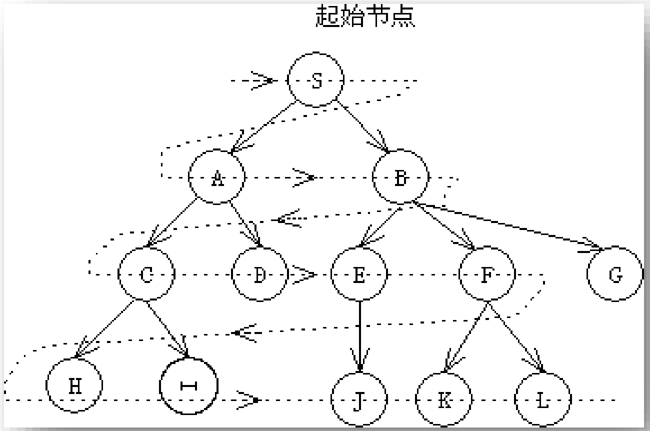
广度优先搜索

- 又称为**宽度优先搜索 (BFS)**
- 一种**先生成**的节点**先扩展**的策略
  - **搜索过程:**
    - 从 **初始节点*S*** 开始**逐层**向下扩展
    - 在**第*n*层节点**还没有全部搜索完之前
    - 不进入**第*n*+1层节点**的搜索



广度优先搜索

- **具体做法**
  - 假设有**两个表**：
    - **Open表**存放待处理节点
    - **Closed表**存放处理完节点
  - Open表中的节点总是**按进入的先后排序**
    - 先进入Open表的节点排在前面
    - 后进入Open表的节点排在后面



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：

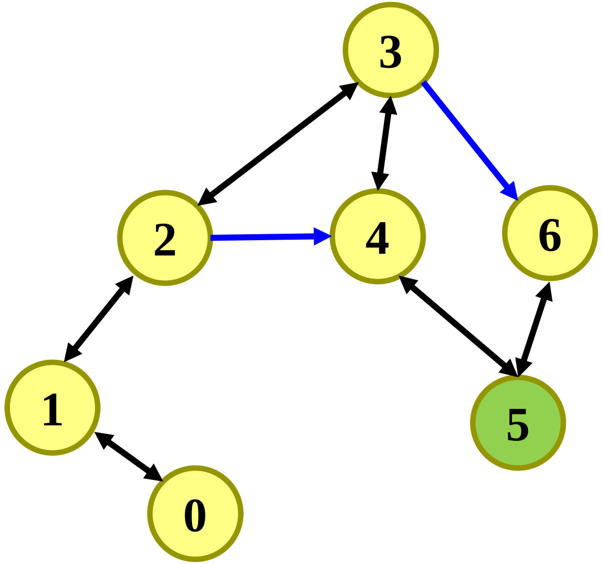
1) 从X移动到X-1或X+1，每次移动花费一分钟

2) 从X移动到2*X，每次移动花费一分钟

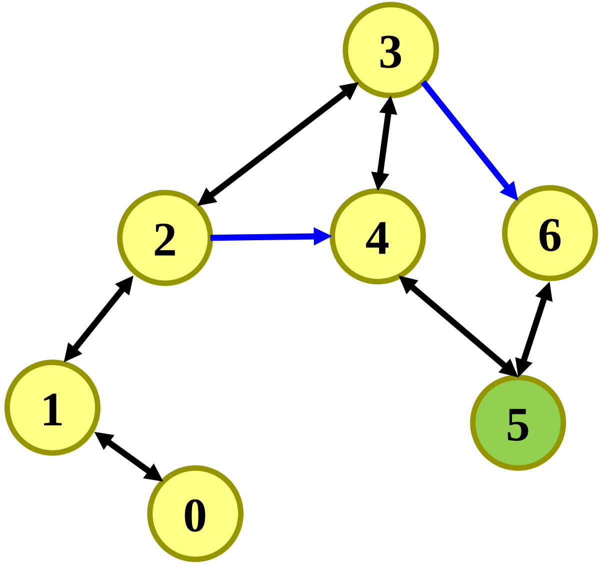
假设牛没有意识到农夫的行动，站在原地不动。

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?



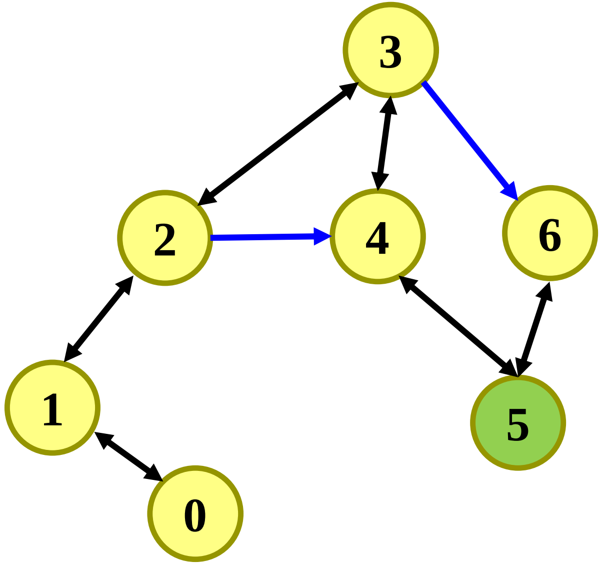
假设农夫起始位于点3, 牛位于5
N=3, K=5, 最右边是6
如何搜索到一条走到5的路径?



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：

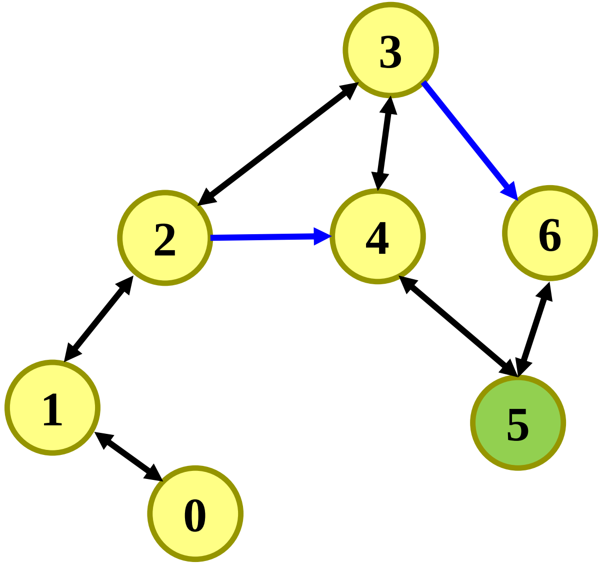
3 → 4 → 5

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。
如何搜索到一条走到5的路径?



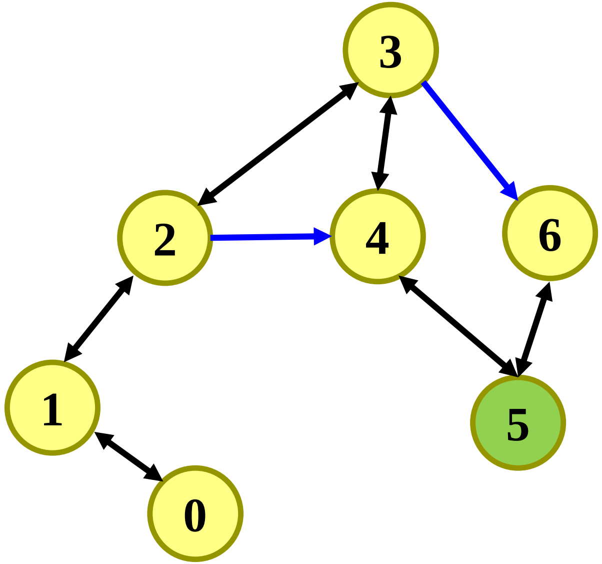
运气不太好的话:
3 → 2 → 4 → 5
运气最坏的话:
3 → 2 → 1 → 0 → 4 → 5

- 要想求最优(短)解, 则要遍历所有走法
- 可以用各种手段优化:
- 例, 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

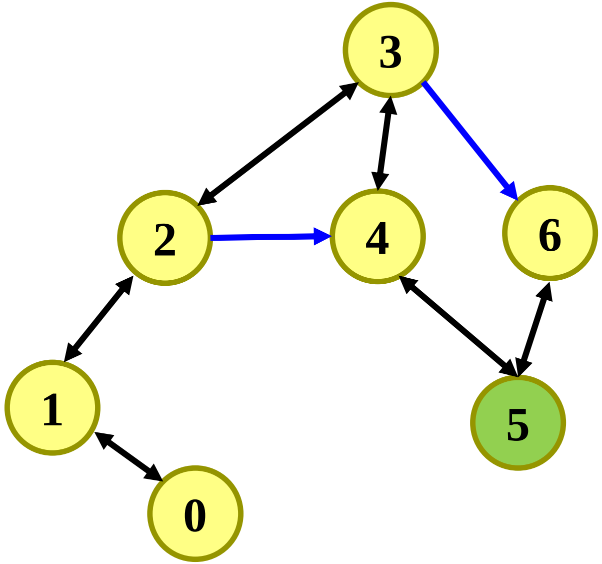
从起点最少需n步就能到达的点属于第n层

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？

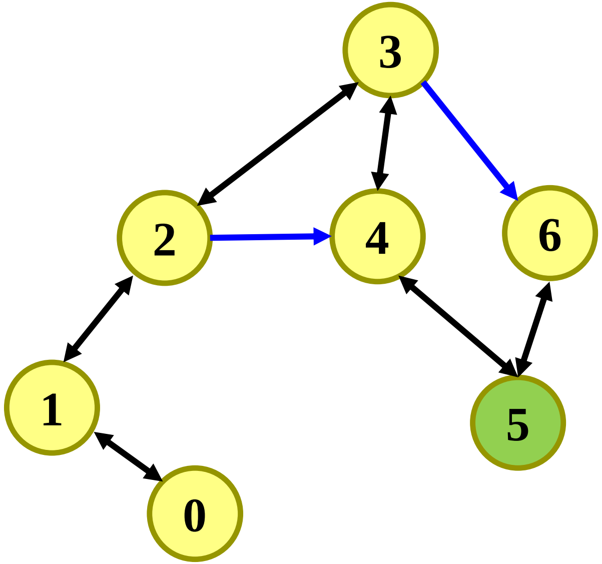


策略2) 广度优先搜索：

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

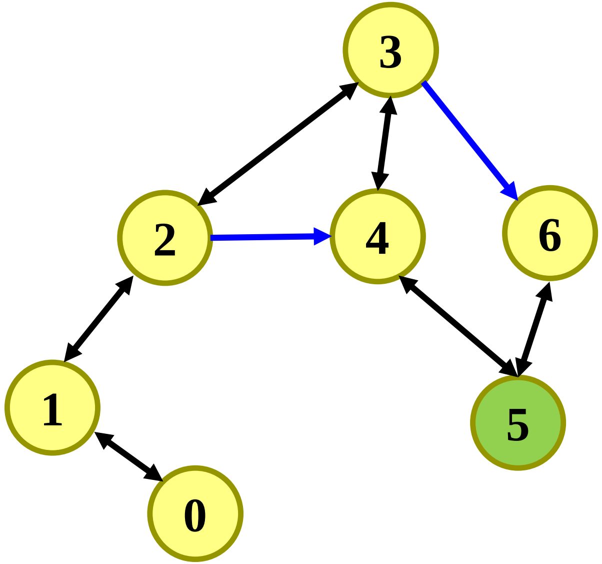
2 4 6

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?

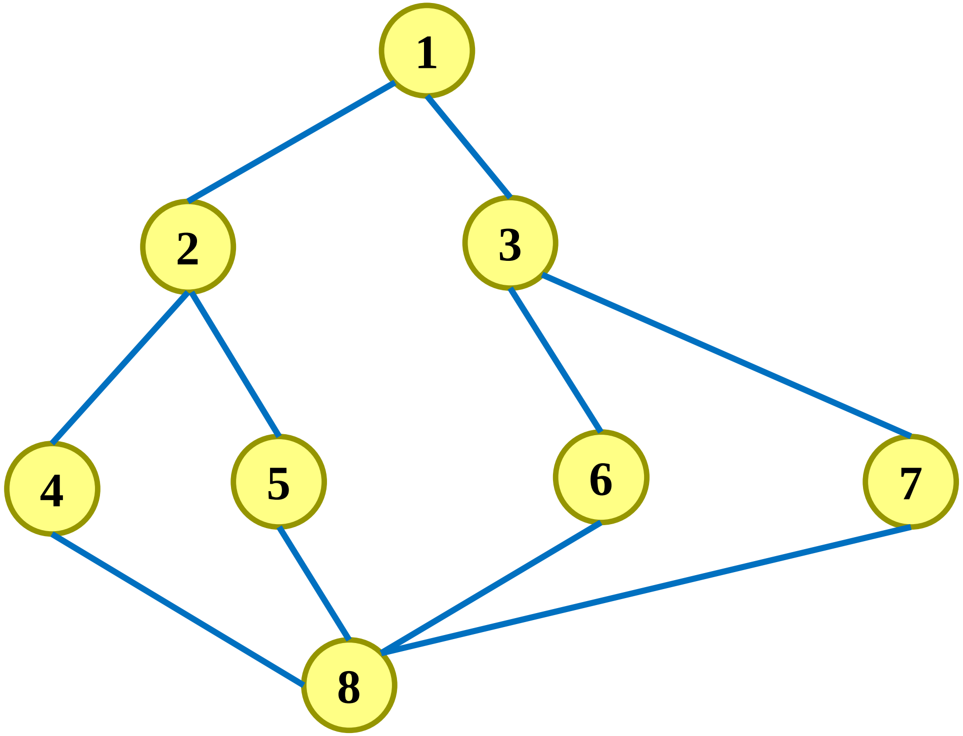


策略2) 广度优先搜索:

- 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S_0放入**Open表**中；

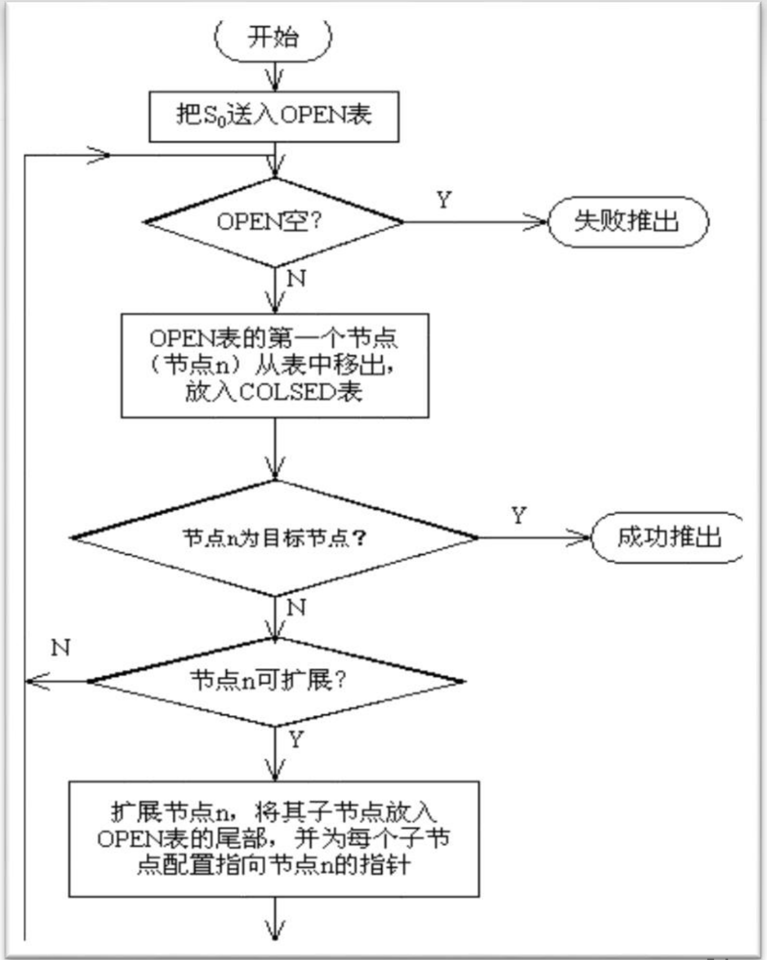
(2) 如果**Open表**为空，则问题无解，失败退出；

(3) 把**Open表**的第一个节点取出放入**Closed表**，并记该节点为n；

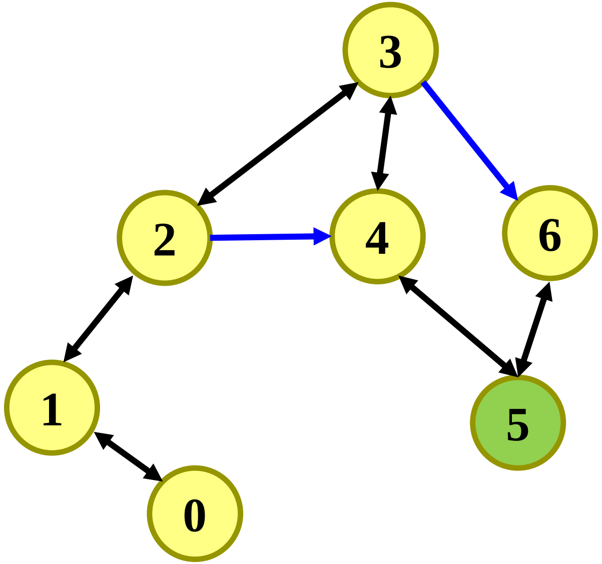
(4) 考察节点n是否为目标节点。若是，则得到问题的解，成功退出；

(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?



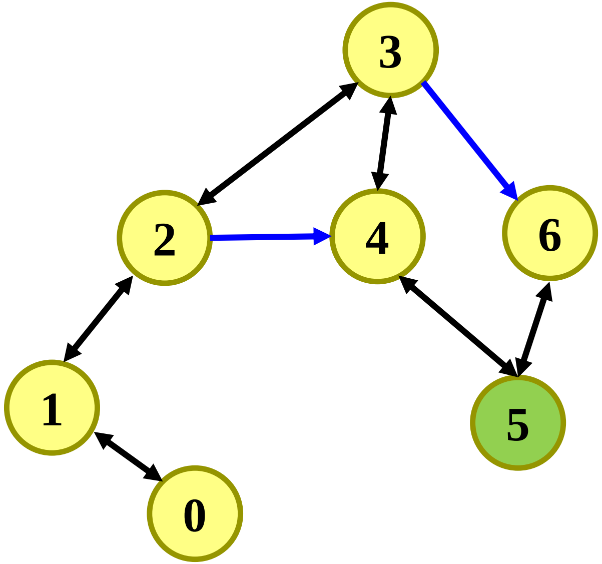
广度优先搜索队列变化过程:

Closed

3

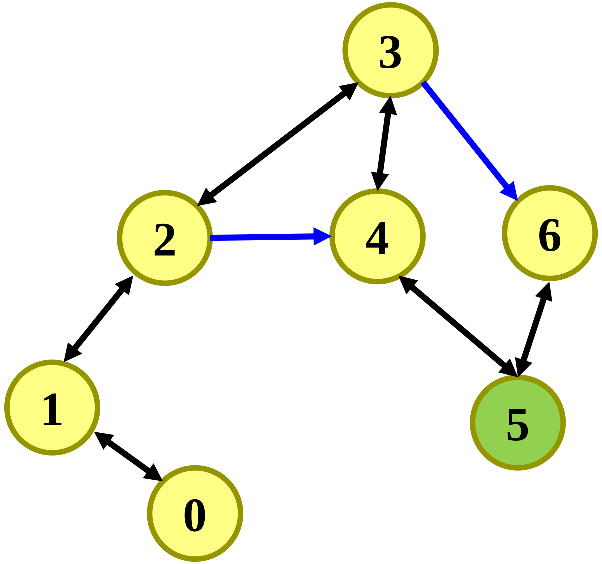
Open

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：

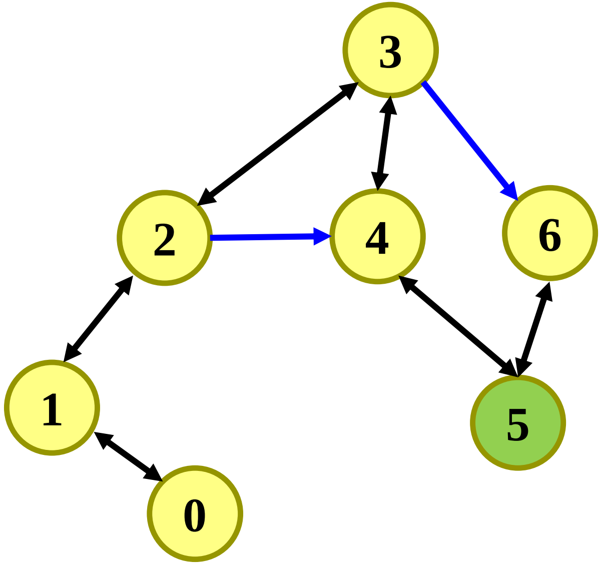
假设农夫起始位于点3, 牛位于5
N=3, K=5, 最右边是6。
如何搜索到一条走到5的路径?



广度优先搜索队列变化过程：

3	2				Closed
		4	6	1	Open

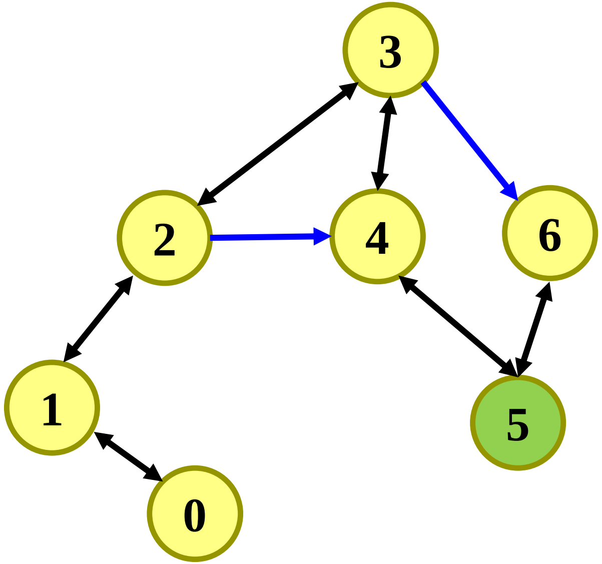
假设农夫起始位于点3, 牛位于5
N=3, K=5, 最右边是6。
如何搜索到一条走到5的路径?



广度优先搜索队列变化过程:

Closed	3	2	4
Open	6	1	5

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:

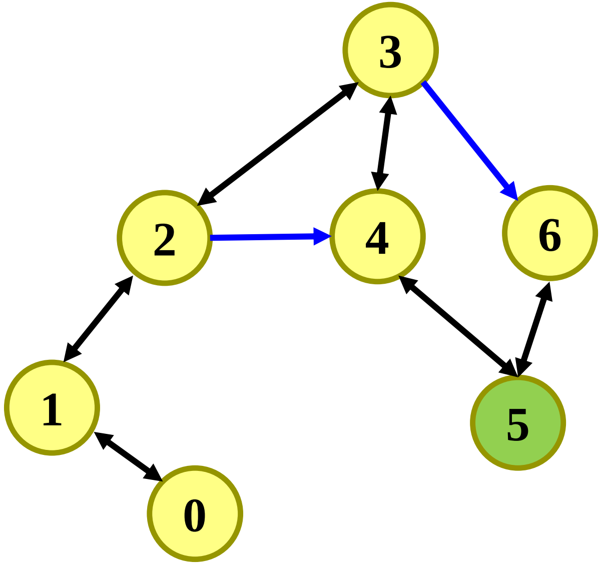
Closed

3	2	4	6

Open

1	5

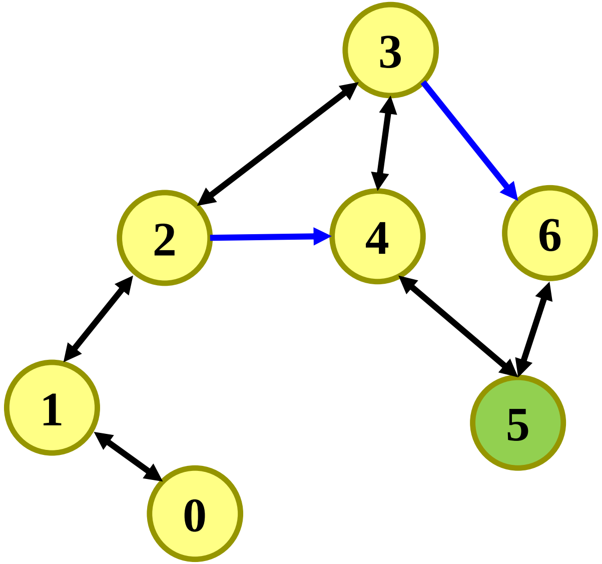
假设农夫起始位于点3, 牛位于5
N=3, K=5, 最右边是6。
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

3	2	4	6	1			Closed
					5	0	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：

3	2	4	6	1	5	Closed
					0	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;
}
```

广度优先搜索

八数码问题

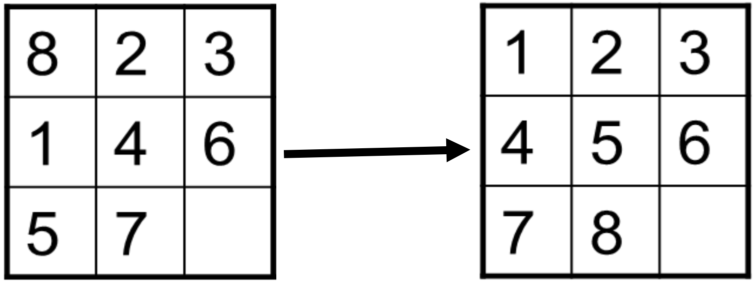
八数码 (POJ1077)

- 八数码问题是人工智能中的经典问题

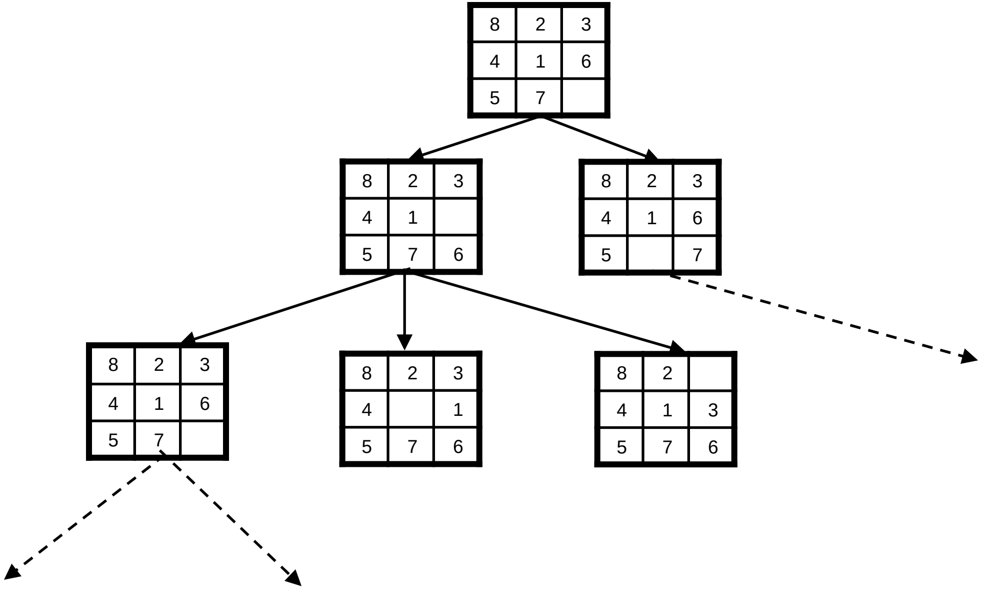
有一个3*3的棋盘，其中有0-8共9个数字，0表示空格，其他的数字可以和0交换位置。求由初始状态到达目标状态

1 2 3
4 5 6
7 8 0

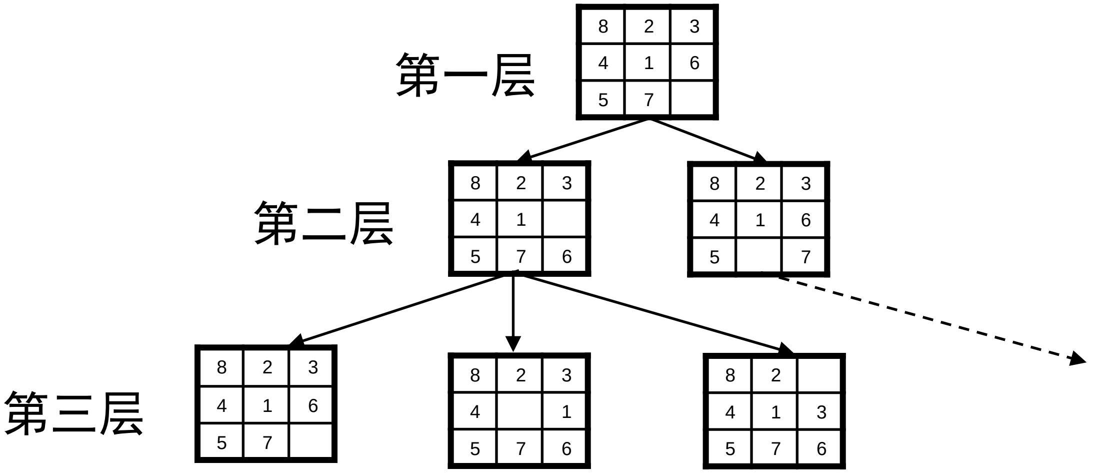
的步数最少的解。



- 状态空间

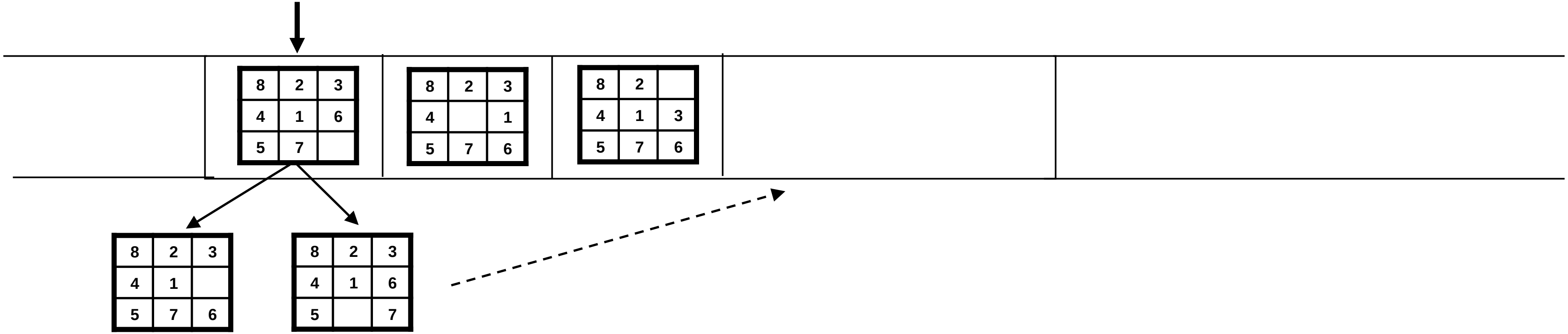


- 广度优先搜索 (BFS)
  - 优先扩展浅层节点 (状态), 逐渐深入



• 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）

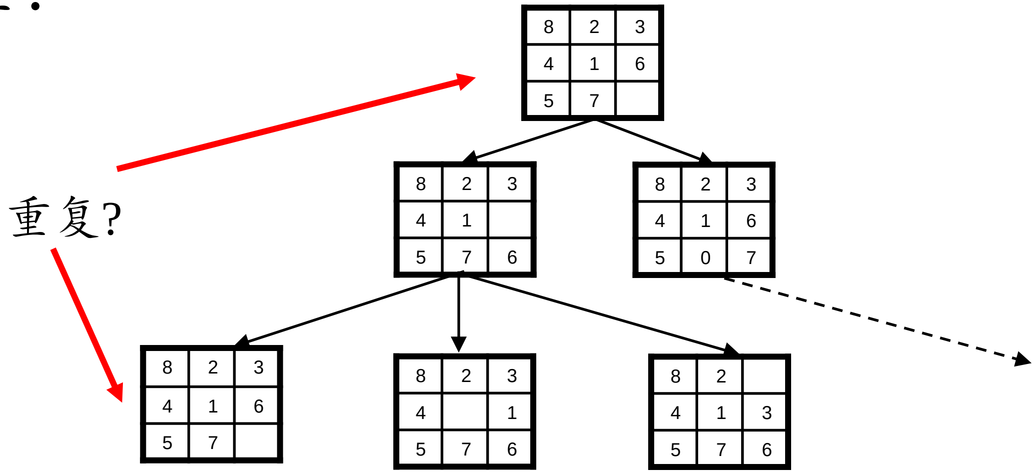


广度优先搜索的代码框架

```
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，则该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	

  - 把每个状态都看作‘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

1) 先算1到**(a1-1)**放在第1位, 会有多少个排列: **(a1-1)* ((k-1)!)**

2) 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)!)

3) 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: $O(n^2)$

3 2 4 1
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
32XX 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:

1 2 3 4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!) >= 9, 所以第一位是2

第二位是1, **2 1 ? ?**, 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, **2 3 ? ?**, 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：2 3 1 4

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：

ulldd rurdllurdruldr

- 输入样例：

2	3	4
1	5	
7	6	8

- 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	

- 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

  - 排列可分为
    - 奇排列
    - 偶排列
  - **从奇排列不能转化成偶排列或相反**

八数码问题有解性的判定

- 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
  - 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;   //>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
```

```
//给定排列,求序号
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 ){
//[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
{ //[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]);

}

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

//整数形式的状态(排列序号)，转换为字符串形式的状态
void IntStatusToStrStatus( int n,  char * strStatus){
       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<u>**最优策略**</u>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

    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

    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

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，相当于两个扩展方向出现了交点，那么可以认为找到了一条路径

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**
  - 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!