



广度优先搜索

入门：抓住那头牛

抓住那头牛 (POJ3278)

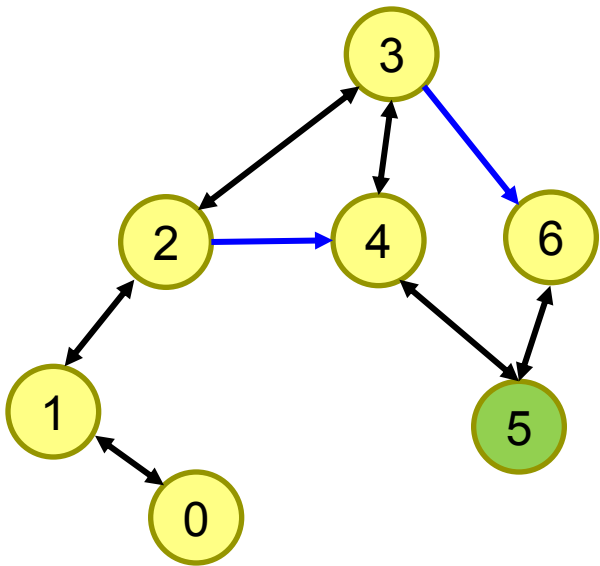
农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 N ($0 \leq N \leq 100000$)，牛位于点 K ($0 \leq K \leq 100000$)。农夫有两种移动方式：

- 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

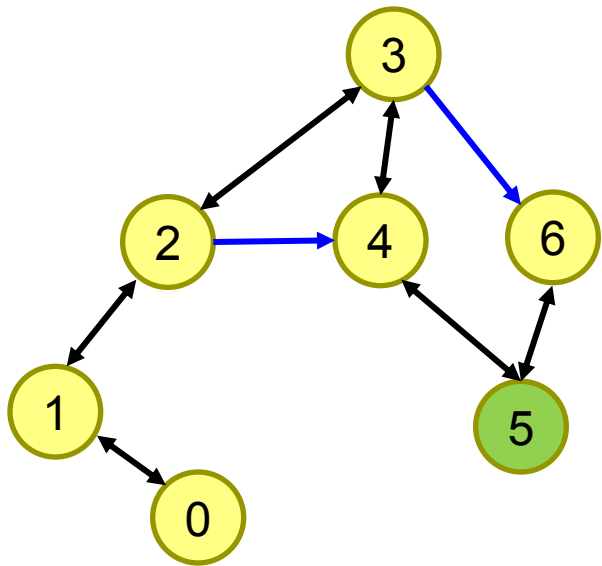


假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

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？

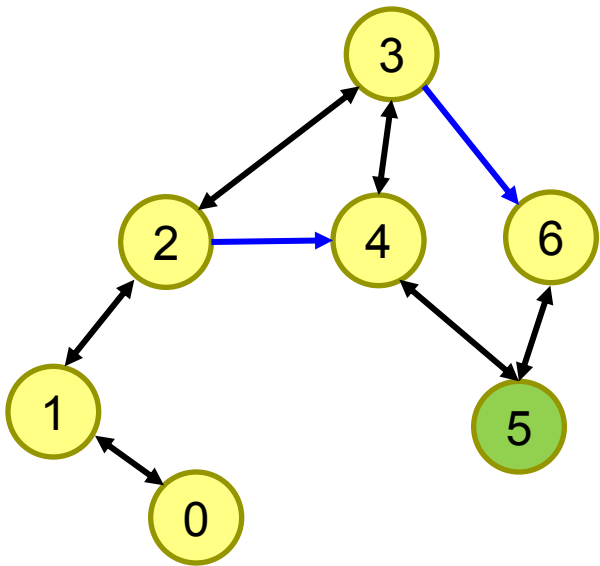


假设农夫起始位于点3，牛位于5
 $N=3$, $K=5$ ，最右边是6。
如何搜索到一条走到5的路径？



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

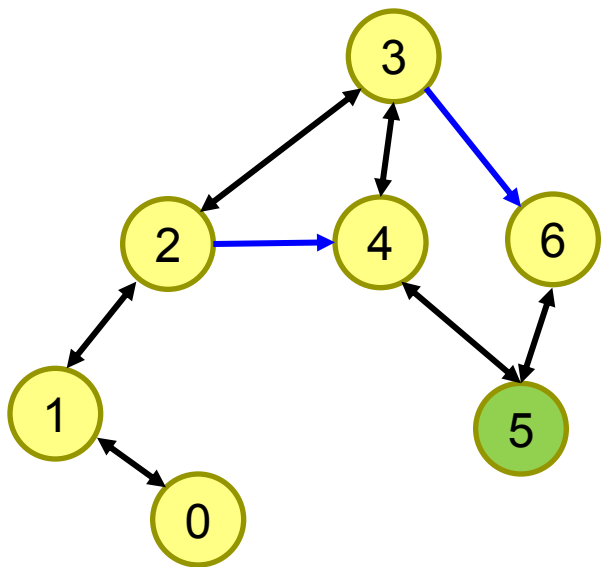
3-→6-→5

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：

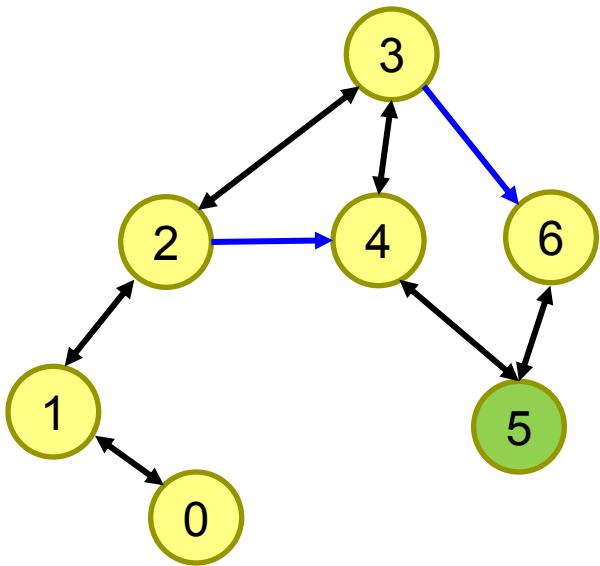
3→2→1→0→4→5

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:

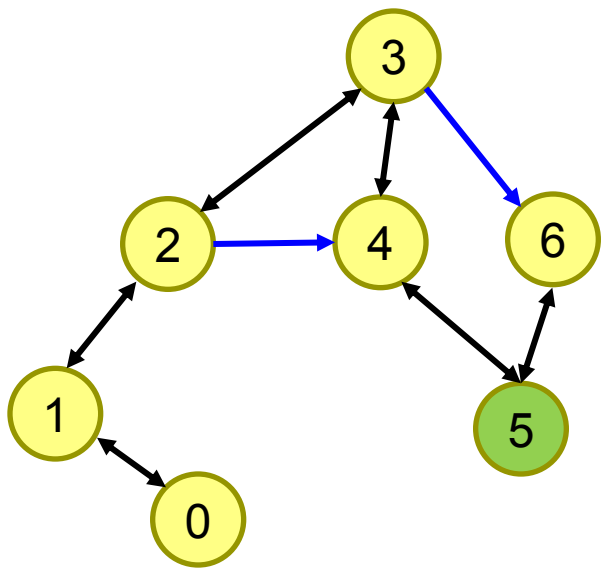
给节点分层。起点是第0层。从起点最少需 n 步就能到达的点属于第 n 层。

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？

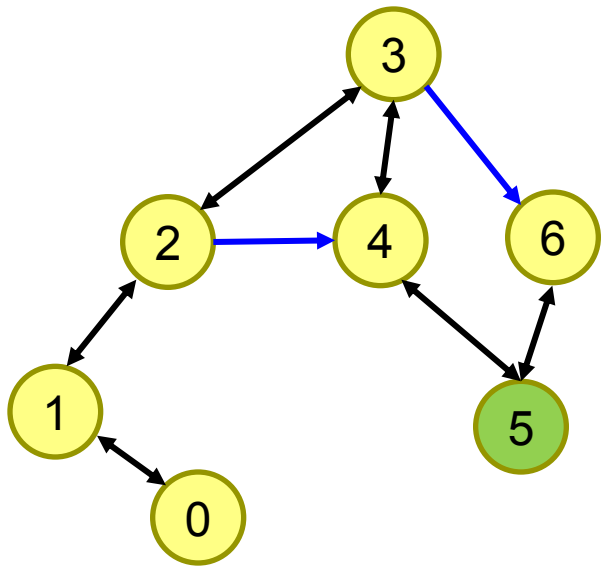


策略2) 广度优先搜索:

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：

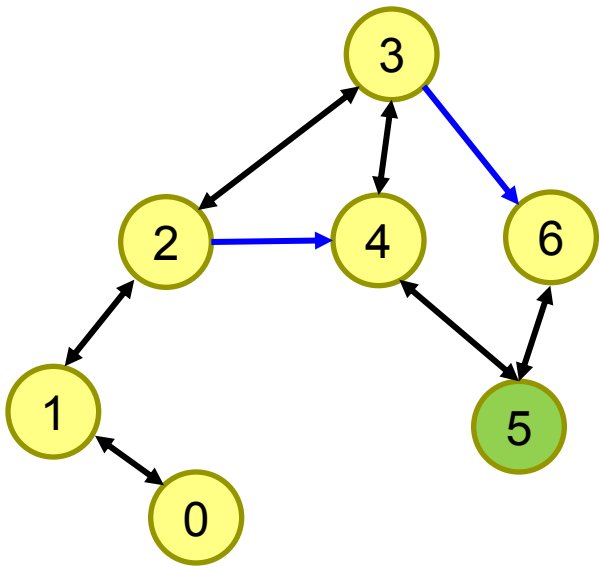
搜索过程（节点扩展过程）：

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？

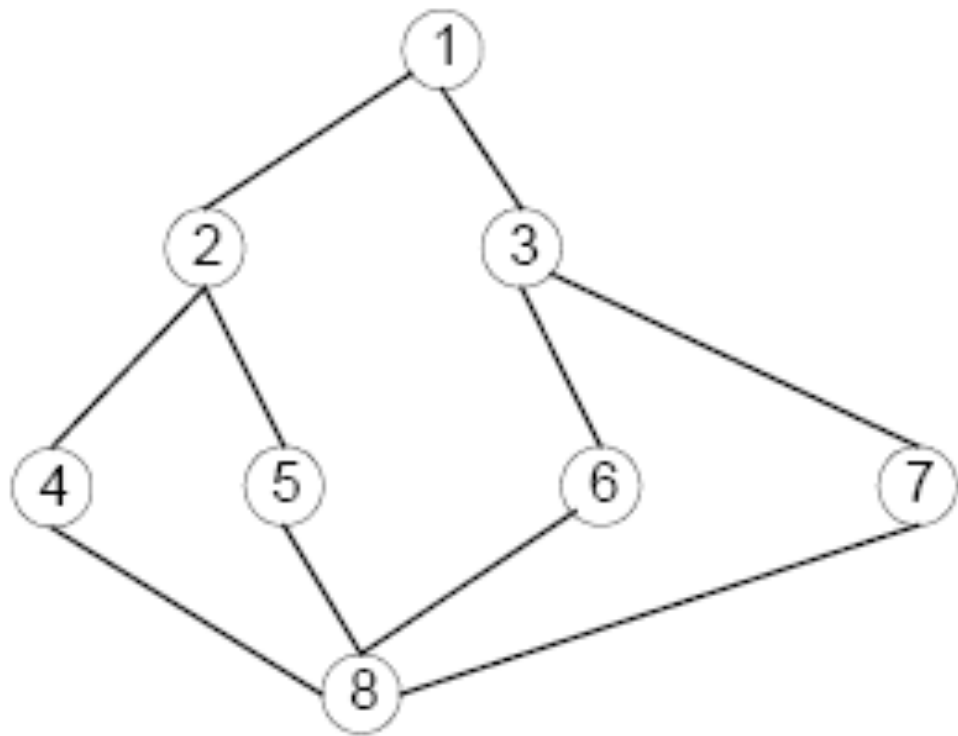


策略2) 广度优先搜索:

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

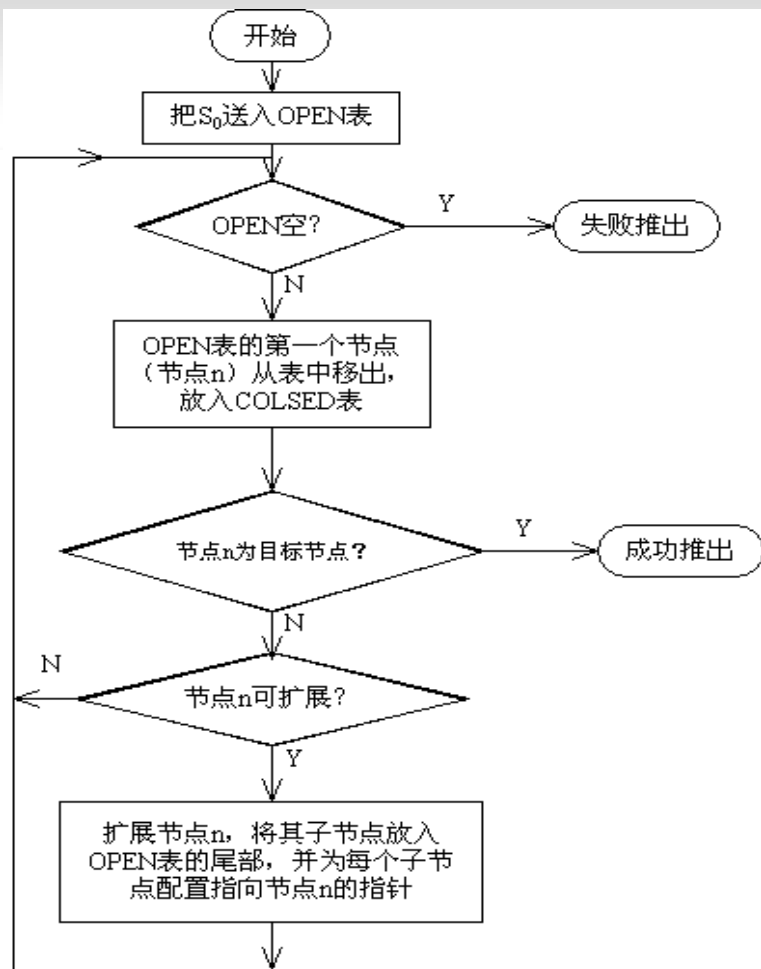
□ 广搜

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

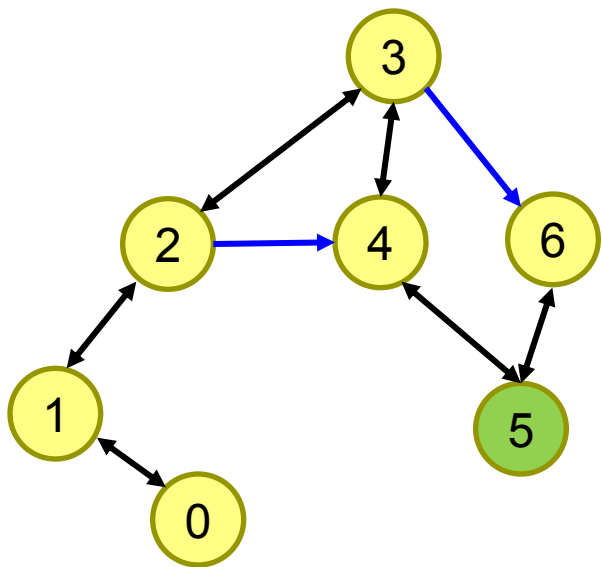
广搜算法

□ 广度优先搜索算法如下：（用QUEUE）

- （1）把初始节点 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？



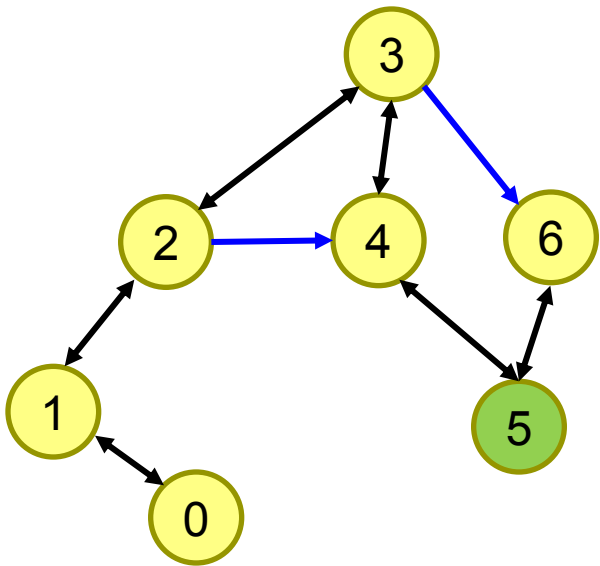
广度优先搜索队列变化过程：

3

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：

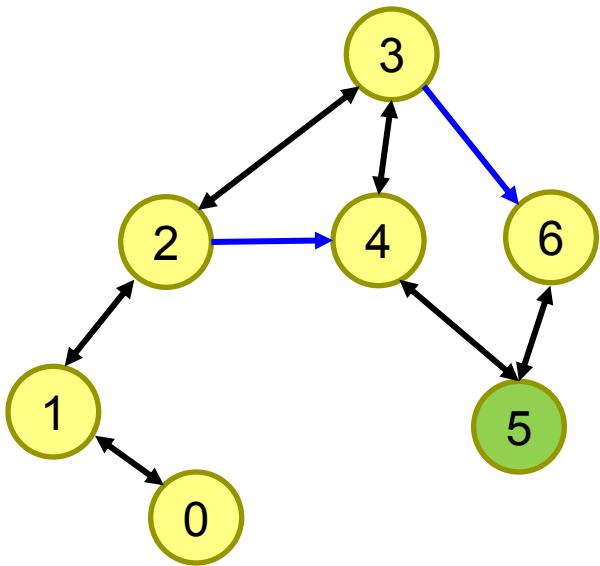
3

2 4 6

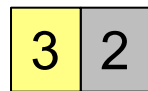
Closed

Open

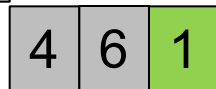
假设农夫起始位于点3，牛位于5
 $N=3$, $K=5$ ，最右边是6。
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

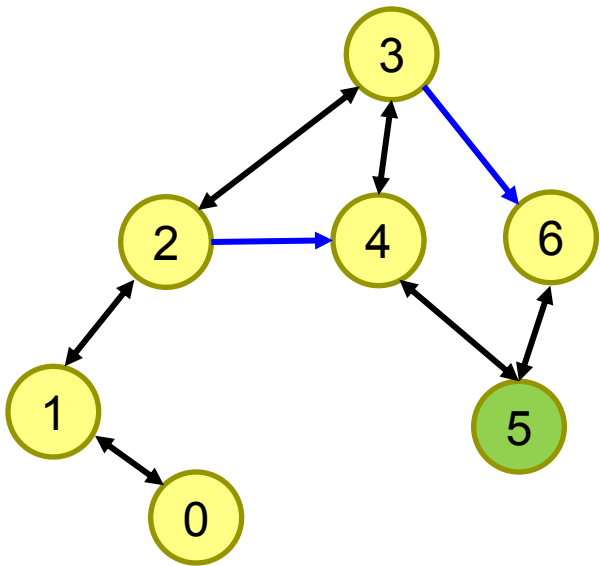


Closed



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：

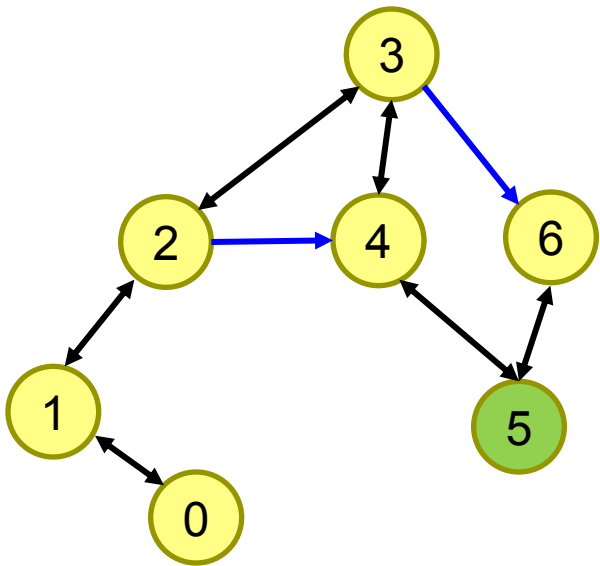


Closed

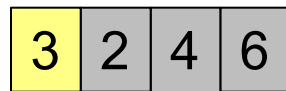


Open

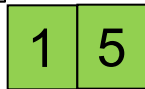
假设农夫起始位于点3，牛位于5
 $N=3$, $K=5$ ，最右边是6。
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

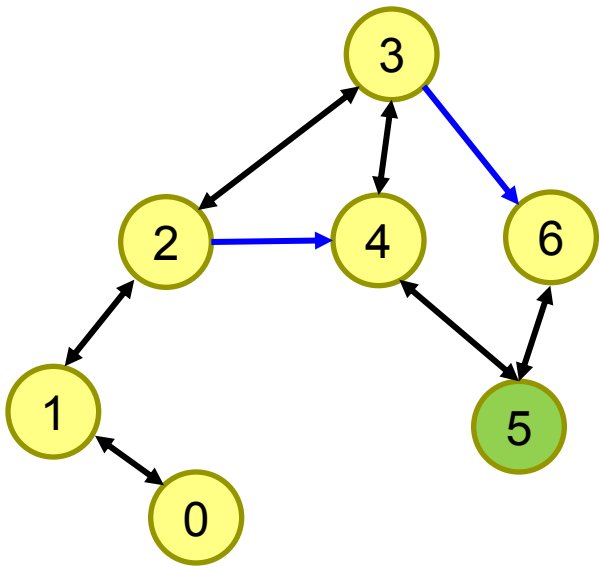


Closed

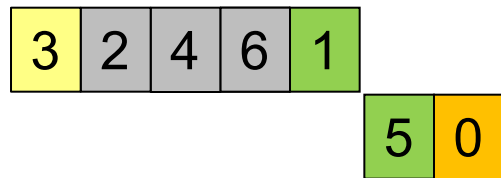


Open

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？



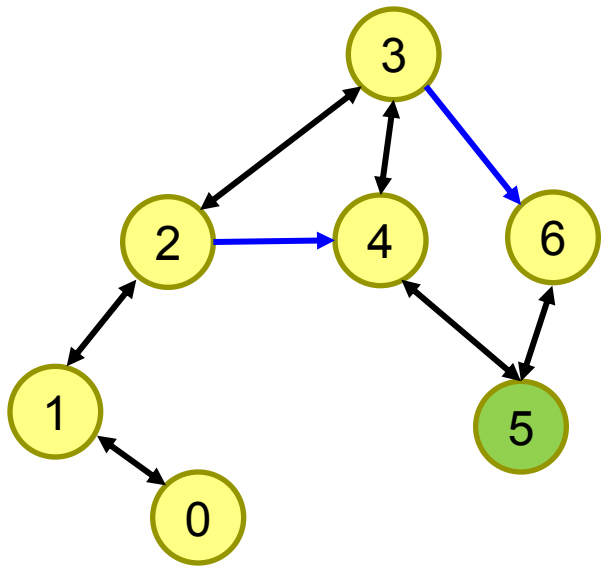
广度优先搜索队列变化过程:



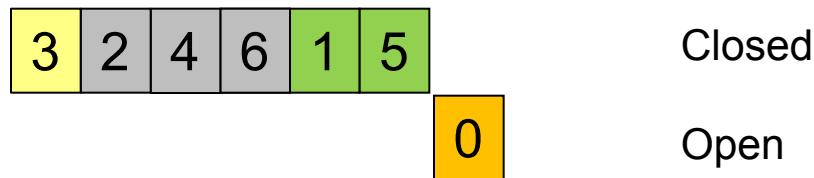
Closed

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:



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;  
}
```

P0J3984 迷宫问题

定义一个二维数组：

```
int maze[5][5] = {  
    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。

P0J3984 迷宫问题

- 广搜
- 不能使用STL的queue，要自己用数组来实现队列
- 新的节点要进入队列时，需在该节点内部用father指针记录其父节点在队列中的下标（到达迷宫中某个位置的时候，要记录刚才才是从哪个位置过来的）
- 当目标节点出队列时，沿着节点的father指针链，就能倒着找到从起点到目标的路径


```

#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

struct Pos {
    int r,c;
    int father; //父节点在队列中的下标,-1表示本节点是起点
    Pos(int rr=0,int cc=0,int ff=0):r(rr),c(cc),father(ff) { }
};

int maze[8][8];
Pos que[100];
int head,tail; //队列头尾指针
Pos dir[4] = {Pos(-1,0),Pos(1,0),Pos(0,-1),Pos(0,1)}; //移动方向
int main() {
    memset(maze,0xff,sizeof(maze));
    for( int i = 1;i <= 5; ++i)
        for (int j = 1; j <= 5; ++j )
            cin >> maze[i][j];

    head = 0;
    tail = 1;
    que[0] = Pos(1,1,-1);

```

```

while( head != tail ) { //队列不为空
    Pos ps = que[head];
    if( ps.r == 5 && ps.c == 5 ) { //目标节点出队列
        vector<Pos> vt;
        while(true) {
            vt.push_back( Pos( ps.r, ps.c, 0 ) );
            if( ps.father == -1 ) //起点
                break;
            ps = que[ps.father];
        };
        for( int i = vt.size()-1; i >= 0; -- i )
            cout << "(" << vt[i].r-1 << ", " <<
                vt[i].c-1 << ")" << endl;
        return 0;
    }
}

```

```

else { //队头节点不是目标节点
    int r = ps.r, c = ps.c;
    for( int i = 0; i < 4; ++i)
        if(! maze[r+dir[i].r][c+dir[i].c]) {
            que[tail++] =
                Pos(r+dir[i].r, c+dir[i].c, head);
            //新扩展出来的节点的父节点在队列里的下标是head
            maze[r+dir[i].r][c+dir[i].c] = 1;
        }
    ++head;
}
}
return 0;
}

```

迷宫问题变形一(百练4980, 拯救行动)

要从迷宫中的起点 r 走到终点 a , 迷宫中各个字符代表道路 ($@$)、墙壁 ($\#$)、和守卫 (x)。

能向上下左右四个方向走。不能走到墙壁。
每走一步需要花费1分钟

行走过程中一旦遇到守卫, 必须杀死守卫才能继续前进。 杀死一个守卫需要花费额外的1分钟

求到达目的地最少用时

```
# @ # # # # @  
# @ a # @ @ r @  
# @ @ # x @ @ @  
@ @ # @ @ # @ #  
# @ @ @ # # @ @  
@ # @ @ @ @ @ @  
@ @ @ @ @ @ @ @
```

百练4980 拯救行动

解法一：

队列里放以下结构：

```
struct Position
{
    int r,c;
    int steps;
};
```

将‘x’对应的节点放入队列时，直接将其steps多加1

要求队列是 **steps**最小的在队头的优先队列！

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

迷宫问题变形一(百练4980, 拯救行动)

解法二:

●状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫  
    int t; //走到本节点花的时间  
};
```

迷宫问题变形一(百练4980, 拯救行动)

- 状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫  
    int t; //走到本节点花的时间  
};
```

- 若 (r, c) 处没有守卫, 则由状态 $(r, c, 0, t)$ 可以扩展出 $(r+1, c, 0, t+1)$, $(r-1, c, 0, t+1)$, $(r, c+1, 0, t+1)$, $(r, c-1, 0, t+1)$

迷宫问题变形一

- 状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫  
    int t; //走到本节点花的时间  
};
```

- 若 (r, c) 处没有守卫, 则由状态 $(r, c, 0, t)$ 可以扩展出 $(r+1, c, 0, t+1)$, $(r-1, c, 0, t+1)$, $(r, c+1, 0, t+1)$, $(r, c-1, 0, t+1)$
- 若 (r, c) 处有守卫, 则由状态 $(r, c, 0, t)$ 只能扩展出 $(r, c, 1, t+1)$

迷宫问题变形一

- 状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫  
    int t; //走到本节点花的时间  
};
```

- 若 (r, c) 处没有守卫, 则由状态 $(r, c, 0, t)$ 可以扩展出
 $(r+1, c, 0, t+1)$, $(r-1, c, 0, t+1)$, $(r, c+1, 0, t+1)$, $(r, c-1, 0, t+1)$

- 若 (r, c) 处有守卫, 则由状态 $(r, c, 0, t)$ 只能扩展出 $(r, c, 1, t+1)$

- 由状态 $(r, c, 1, t)$ 可以扩展出:

$(r+1, c, 0, t+1)$, $(r-1, c, 0, t+1)$, $(r, c+1, 0, t+1)$, $(r, c-1, 0, t+1)$

迷宫问题变形一(百练4980, 拯救行动)

判重数组:

```
int flag[M][N][2];
```

`flag[r][c][0]`表示在坐标 (r, c) , 尚未杀死守卫的情况

`flag[r][c][1]`表示在坐标 (r, c) , 已经杀死守卫的情况

其实只要 `int flag[M][N];` 也可以.

迷宫问题变形二(百练6044, 鸣人和佐助)

要从迷宫中的起点 r 走到终点 a , 迷宫中各个字符代表道路 (@)、墙壁 (#)、和守卫 (x)。

能向上下左右四个方向走。不能走到墙壁。
每走一步需要花费1分钟

行走过程中一旦遇到守卫, 必须杀死守卫才能继续前进。杀死一个守卫需要花费1块钱, 最开始有 n 块钱。

求到达目的地最少用时

```
# @ # # # # @  
# @ a # @ @ r @  
# @ @ # x @ @ @  
@ @ # @ @ # @ #  
# @ @ @ # # @ @  
@ # @ @ @ @ @ @  
@ @ @ @ @ @ @ @
```

迷宫问题变形二(百练6044, 鸣人和佐助)

●状态表示:

```
struct Pos {  
    int r, c; //本节点的位置  
    int m; //钱数  
    int t; //走到本节点花的时间  
};
```

●判重数组

`flag[r][c][m]`表示到达 (r, c) 时, 钱数为 m 这种状态是否扩展过。

```
######@  
#@a#@@r@  
###x@@@  
@@#@@#@#  
#####@  
@#@@@@  
@@@@@@@
```

迷宫问题变形三(百练8436, Saving Tang Monk)

要从迷宫中的起点 r 走到终点 a , 迷宫中各个字符代表道路 (@)、墙壁 (#)、和守卫 (x), 放有钥匙的道路 (1--9, 表示有9种钥匙)

行走过程中一旦遇到守卫, 必须杀死守卫才能继续前进。杀死一个守卫需要花费额外1分钟。最多5个守卫。

走到终点时, 必须要每种钥匙至少有一把才算完成任务。钥匙不全, 也可以经过终点。

想拿第 k 种钥匙, 必须手里已经有第 $k-1$ 种钥匙。拿不了钥匙, 也可以经过放钥匙的地方

求完成任务最少用时

```
#####@  
#@a#@@r@  
#@@#x@@@  
@@#@@#1#  
#@@2##@@  
@#@@5@@@  
@@@@@@@@
```

```
struct Status
{
    short r,c;
    short keys;
    short foughted;           //守卫是否打过
    int steps;
    short layout;             //守卫的局面(哪些被杀, 哪些还没被杀)
};

char flags[100][100][10][33]; //判重
```

`flags[r][c][k][x]` 对应的状态是:

在位置 (r,c) , 手里有 k 把钥匙, 守卫的局面是 x

一共只有5个守卫, 他们被杀或没被杀的情况一共有32种, 可以用5个bit表示



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

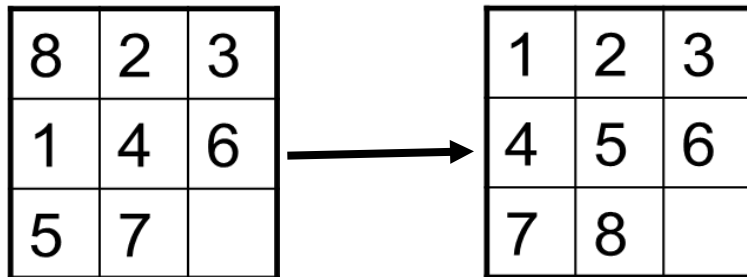
有一个3*3的棋盘，其中有0-8共9个数字，0表示空格，其他的数字可以和0交换位置。求由初始状态到达目标状态

1 2 3

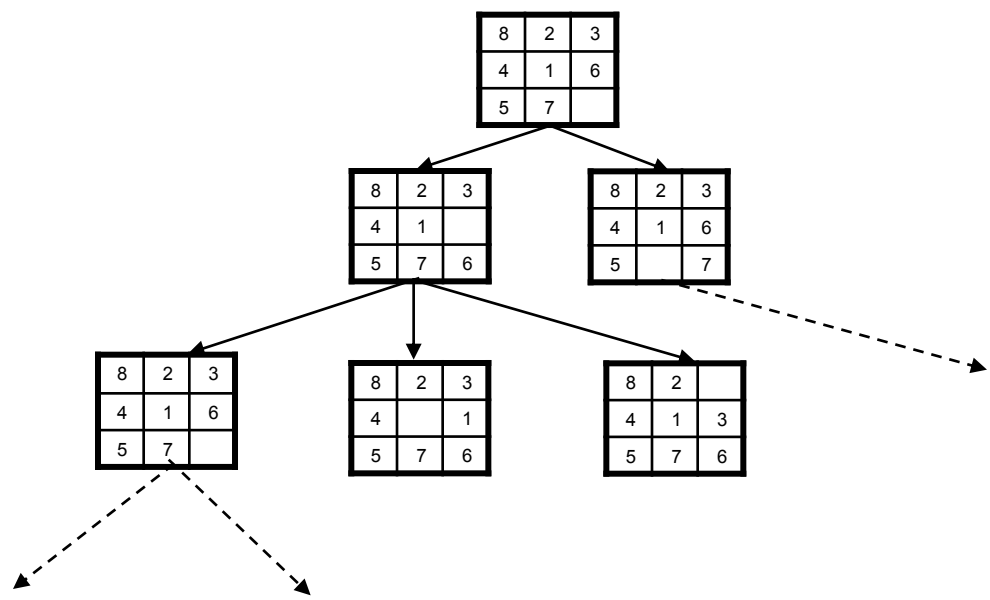
4 5 6

7 8 0

的步数最少的解。

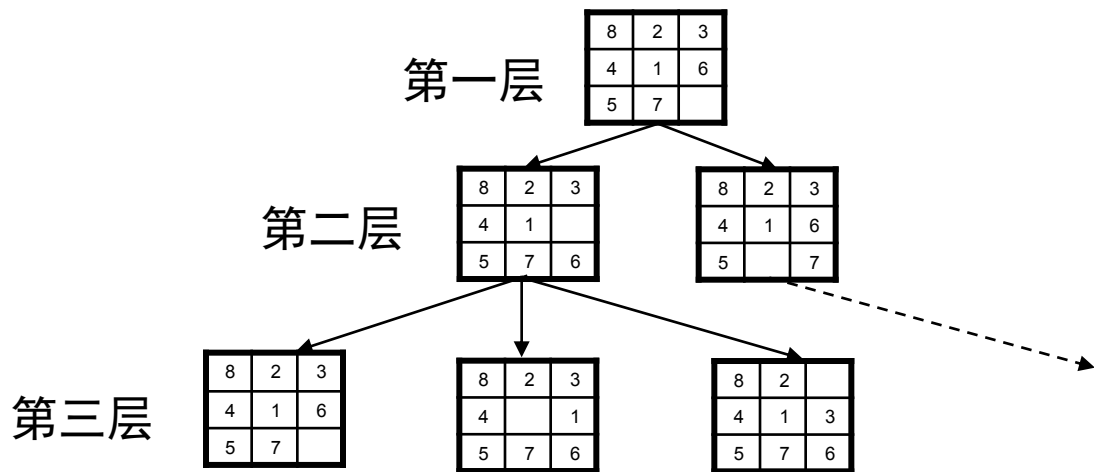


● 状态空间



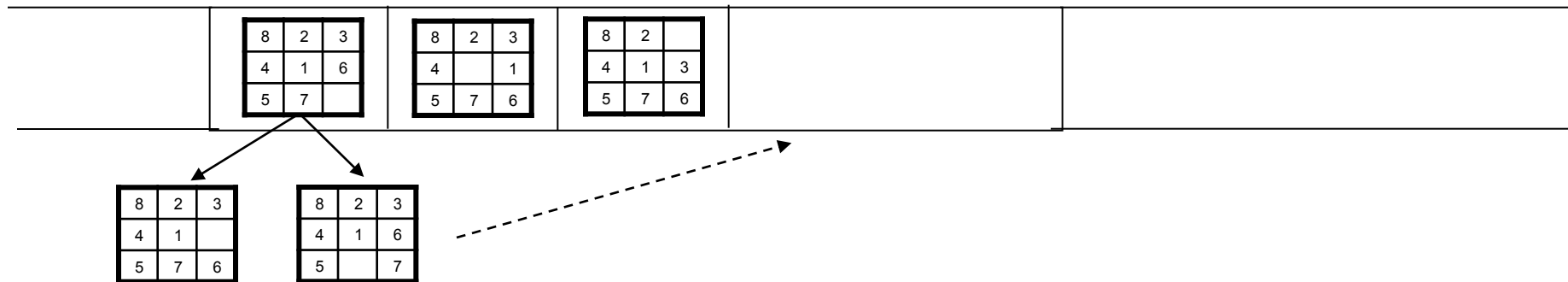
● 广度优先搜索 (bfs)

- 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；
```

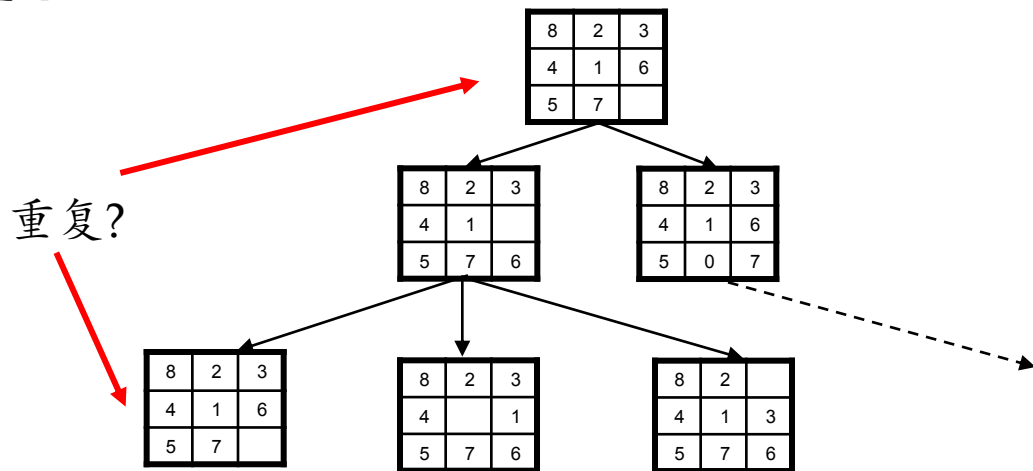
```
        必要时要记住每个节点的父节点；
```

```
    }
```

```
}
```

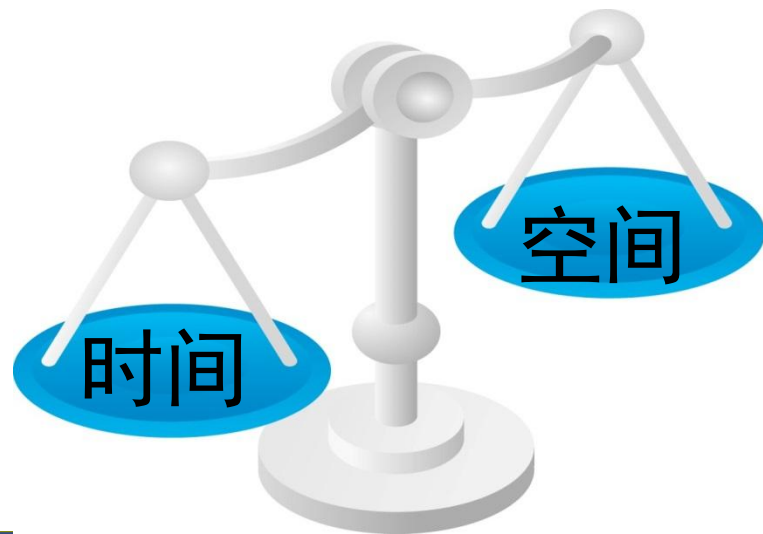
关键问题：判重

- 新扩展出的节点如果和以前扩展出的节点相同，则这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 $1, 2 \cdots k$ 的一个排列：

$a_1 \ a_2 \ a_3 \ \cdots a_k$

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 a_1-1 放在第1位，会有多少个排列： $(a_1-1) * ((k-1)!)$

再算 a_1 不变，1到 a_2-1 放在第2位(左边出现过的不能再用)，会有多少个排列： $(a_2-1) * ((k-2)!)$

再算 a_1, a_2 不变，1到 a_3-1 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 $3!+2!+2! \geq 9$, 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:

ulddrurdllurdruldr

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 $F(X)$ ($X \neq 0$) 表示数字X前面比它小的数（不包括0）的个数，全部数字的 $F(X)$ 之和为 $Y = \sum (F(X))$ ，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，用set判重，

```
#include <iostream>
#include <bitset>
#include <cstring>
#include <cstdio>
#include <cstdlib>
#include <set>
```

```
using namespace std;
```

```
int goalStatus = 123456780; //目标状态
```

```
const int MAXS = 400000;
```

```
char result[MAXS]; //要输出的移动方案
```

```
struct Node {
```

```
    int status; //状态
```

```
    int father; //父节点指针，即myQueue的下标
```

```
    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 = 0; //队头指针
```

```
int qTail = 1; //队尾指针
```

```
char moves[] = "udr1"; //四种移动
```

```
int NewStatus( int status, char move) {  
    //求从status经过 move 移动后得到的新状态。若移动不可行则返回-1  
    char tmp[20];  
    int zeroPos; //字符'0'的位置  
    sprintf(tmp,"%09d",status); //需要保留前导0  
    for( int i = 0;i < 9; ++ i )  
        if( tmp[i] == '0' ) {  
            zeroPos = i;  
            break;  
        } //返回空格的位置  
    switch( move) {  
        case 'u':  
            if( zeroPos - 3 < 0 )  
                return -1; //空格在第一行  
            else {  
                tmp[zeroPos] = tmp[zeroPos - 3];  
                tmp[zeroPos - 3] = '0';  
            }  
            break;  
    }  
}
```

```
case 'd':
    if( zeroPos + 3 > 8 )
        return -1; //空格在第三行
    else {
        tmp[zeroPos] = tmp[zeroPos + 3];
        tmp[zeroPos + 3] = '0';
    }
    break;
case 'l':
    if( zeroPos % 3 == 0)
        return -1; //空格在第一列
    else {
        tmp[zeroPos] = tmp[zeroPos -1];
        tmp[zeroPos -1 ] = '0';
    }
    break;
```

```
case 'r':
    if( zeroPos % 3 == 2)
        return -1; //空格在第三列
    else {
        tmp[zeroPos] = tmp[zeroPos + 1];
        tmp[zeroPos + 1 ] = '0';
    }
    break;
}
return atoi(tmp);
}
```



```
bool Bfs(int status) {  
    //寻找从初始状态status到目标的路径，找不到则返回false  
    int newStatus;  
    set<int> expanded;  
    myQueue[qHead] = Node(status,-1,0);  
    expanded.insert(status);  
    while ( qHead != qTail) { //队列不为空  
        status = myQueue[qHead].status;  
        if( status == goalStatus ) //找到目标状态  
            return true;  
        for( int i = 0;i < 4;i ++ ) { //尝试4种移动  
            newStatus = NewStatus(status,moves[i]);  
            if( newStatus == -1 )  
                continue; //不可移，试下一种  
            if(expanded.find(newStatus)!=expanded.end())  
                continue; //已扩展过，试下一种  
            expanded.insert(newStatus);  
        }  
    }  
}
```

```
        myQueue[qTail++] =  
            Node(newStatus, qHead, moves[i]);  
            //新节点入队列  
    }  
    qHead ++;  
}  
return false;  
}
```

```
int main() {
    char line1[50]; char line2[20];
    while( cin.getline(line1,48)) {
        int i,j;
        //将输入的原始字符串变为数字字符串
        for( i = 0, j = 0; line1[i]; i ++ ) {
            if( line1[i] != ' ' ) {
                if( line1[i] == 'x' )
                    line2[j++] = '0';
                else
                    line2[j++] = line1[i];
            }
        }
        line2[j] = 0; //字符串形式的初始状态
    }
}
```

```

if( Bfs(atoi(line2))) {
    int moves = 0;
    int pos = qHead;
    do { //通过father找到成功的状态序列，输出相应步骤
        result[moves++] = myQueue[pos].move;
        pos = myQueue[pos].father;
    } while( pos); //pos = 0 说明已经回退到初始状态了
    for( int i = moves -1; i >= 0; i -- )
        cout << result[i];
}
else
    cout << "unsolvable" << endl;
}
}

```



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

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 q_0 ,将目标节点放入队列 q_1 ;
2. 当两个队列都未空时, 作如下循环:
 - 1) 如果队列 q_0 里的节点比 q_1 中的少,则扩展队列 q_0 ;
 - 2) 否则扩展队列 q_1
3. 如果队列 q_0 未空, 不断扩展 q_0 直到为空;
4. 如果队列 q_1 未空, 不断扩展 q_1 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 q_i 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 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!

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：
 - 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）