程序设计实习

算法基础

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2024 年 5 月 29 日

例题 01: 抓住那头牛

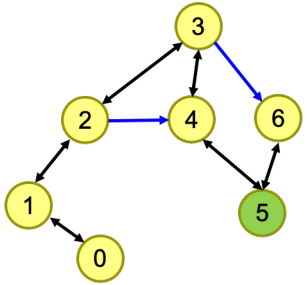
农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 $N(0 \leq N \leq 100000)$，牛位于点 $K(0 \leq K \leq 100000)$。农夫有两种移动方式：

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?

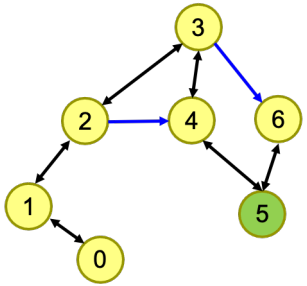
例题 01: 抓住那头牛

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?



例题 01: 抓住那头牛

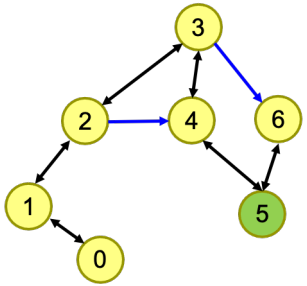
假设农夫起始位于点 3，牛位于 5($N = 3, K = 5$)，最右边是 6。如何搜索到一条走到 5 的路径?



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)。

例题 01: 抓住那头牛

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?

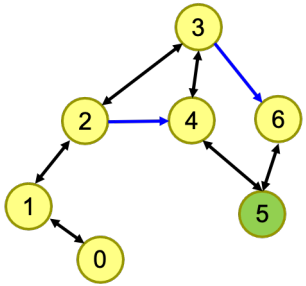


策略 1）深度优先搜索：从起点出发，随机挑一个方向，能往前走就往前走 (扩展)，走不动了则回溯。不能走已经走过的点 (要判重)。

运气好的话：3->4->5 或 3->6->5 问题解决!

例题 01: 抓住那头牛

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?



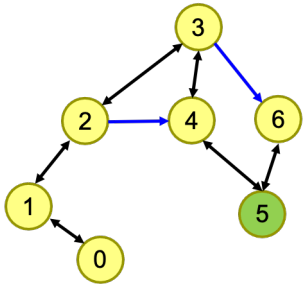
策略 1）深度优先搜索：从起点出发，随机挑一个方向，能往前走就往前走 (扩展)，走不动了则回溯。不能走已经走过的点 (要判重)。

运气好的话：3->4->5 或 3->6->5 问题解决!

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

例题 01: 抓住那头牛

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?

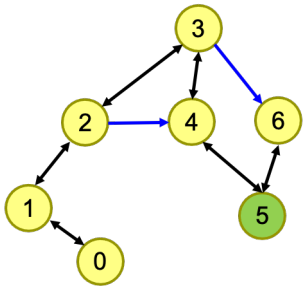


策略 1）深度优先搜索：从起点出发，随机挑一个方向，能往前走就往前走 (扩展)，走不动了则回溯。不能走已经走过的点 (要判重)。

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。

例题 01: 抓住那头牛

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?



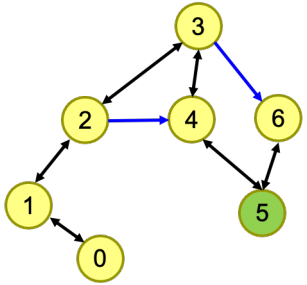
策略 2）广度优先搜索：给节点分层。起点是第 0 层。从起点最少需 n 步就能到达的点属于第 n 层。

第 1 层：2,4,6
第 2 层：1,5
第 3 层：0

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。扩展时，不能扩展出已经走过的节点 (要判重)。

例题 01: 抓住那头牛

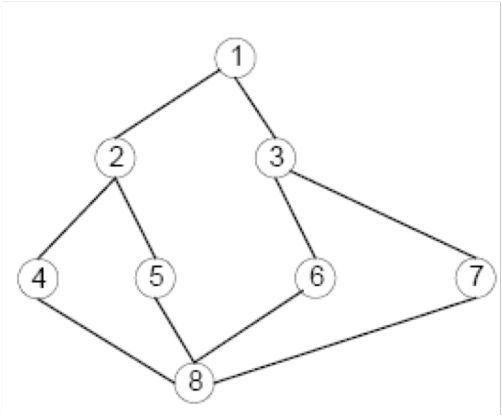
假设农夫起始位于点 3，牛位于 5($N = 3, K = 5$)，最右边是 6。如何搜索到一条走到 5 的路径?



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。

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

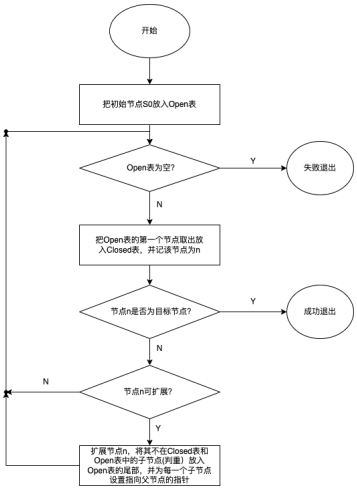
广搜

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。

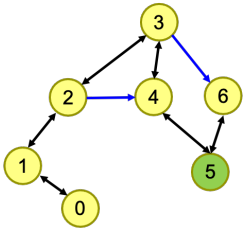


广度优先搜索的代码框架

```
1  BFS() {
2    初始化队列
3    while(队列不为空且未找到目标节点) {
4      取队首节点扩展，并将扩展出的非重复节点放入队尾;
5      必要时要记住每个节点的父节点;
6    }
7  }
```

例题 01: 抓住那头牛

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?



广度优先搜索队列变化过程：

Closed

3

Open

例题 01: 抓住那头牛

```cpp
#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] = 1 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;
```

例题 01: 抓住那头牛

```
19    while (!q.empty()) {
20      Step s = q.front();
21      q.pop();
22      if (s.x == K) { //找到目标
23        cout << s.steps << endl;
24        return 0;
25      }
26      if (s.x - 1 >= 0 && !visited[s.x - 1]) {
27        q.push(Step(s.x - 1, s.steps + 1));
28        visited[s.x - 1] = 1;
29      }
30      if (s.x + 1 <= MAXN && !visited[s.x + 1]) {
31        q.push(Step(s.x + 1, s.steps + 1));
32        visited[s.x + 1] = 1;
33      }
34      if (s.x * 2 <= MAXN && !visited[s.x * 2]) {
35        q.push(Step(s.x * 2, s.steps + 1));
36        visited[s.x * 2] = 1;
37      }
38    }
39    return 0;
40  }
```

例题 02: 迷宫问题

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。

例题 02: 迷宫问题

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

例题 02: 迷宫问题

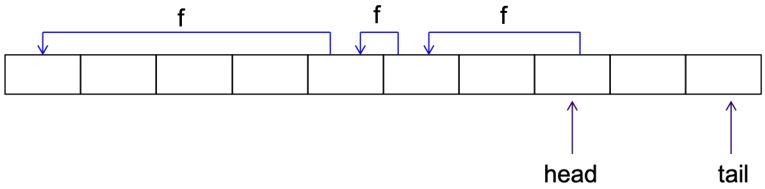
基础广搜。

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
队列元素：

```
struct {
  int r,c;
  int f; //父节点在队列中的下标
};
```

例题 02: 迷宫问题

队列不能直接用 STL 的 queue，要自己写。可以用一维数组实现，维护一个队头指针和队尾指针



例题 02: 迷宫问题-变形 1

鸣人要从迷宫中的起点 r 走到终点 a, 去营救困在那里的佐助。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@@@
@@#@@#@#
#@@@##@@
@#@@@@@
```

例题 02: 迷宫问题-变形 1

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

例题 02: 迷宫问题-变形 1

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
队列是**优先队列**，steps 最小的在队头

例题 02: 迷宫问题-变形 1

解法二：
队列里放以下结构：

```
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
};
```

例题 02: 迷宫问题-变形 1

解法二：
队列里放以下结构：

```
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)

例题 02: 迷宫问题-变形 2

鸣人要从迷宫中的起点 r 走到终点 a 去救佐助，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 个查克拉，最开始有 n 个查克拉。

求到达目的地最少用时。

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

例题 02: 迷宫问题-变形 2

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 值比队头的 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。

例题 02: 迷宫问题-变形 3

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。

例题 02: 迷宫问题-变形 3

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) 是佐助待的地方

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

例题 02: 迷宫问题-变形 4

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@@@
@@@@@@@@
```

例题 02: 迷宫问题-变形 4

```
struct Status {
  short r,c;
  short keys;
  short fighted;//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

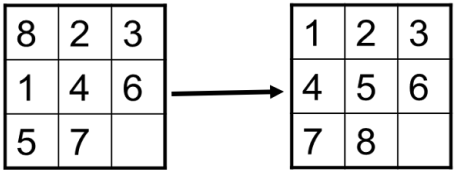
例题 03: 八数码问题 (Eight)

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

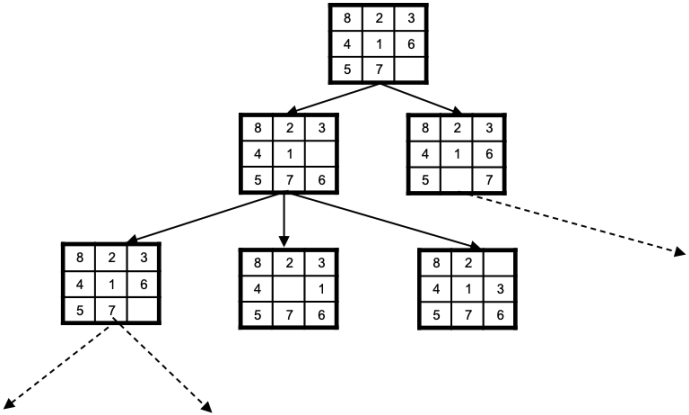
```
1 2 3
4 5 6
7 8 0
```

的步数最少的解。



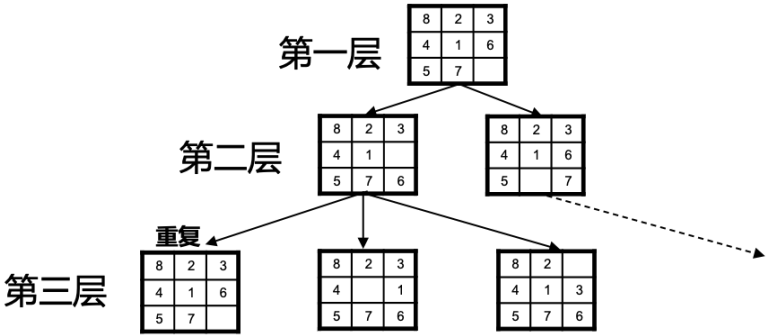
例题 03: 八数码问题 (Eight)

状态空间：



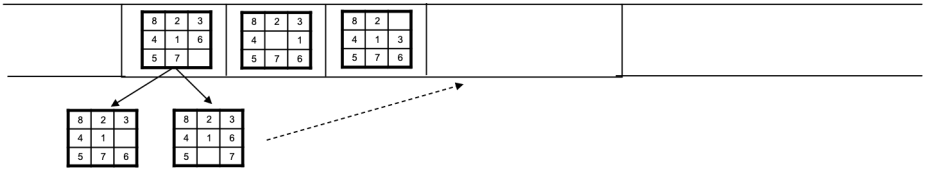
例题 03: 八数码问题 (Eight)

广度优先搜索（bfs)：优先扩展浅层节点 (状态)，逐渐深入



例题 03: 八数码问题 (Eight)

广度优先搜索（bfs）
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）

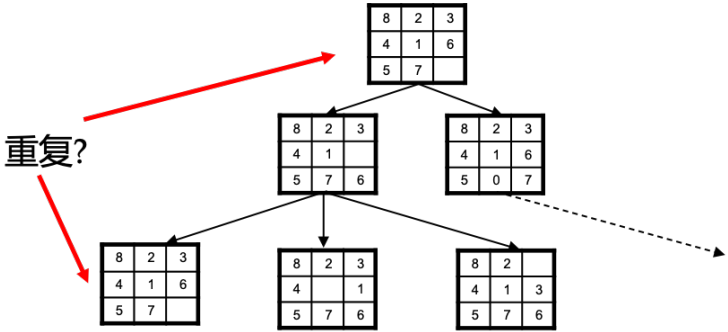


例题 03: 八数码问题 (Eight)

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?



例题 03: 八数码问题 (Eight)

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?

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案一：
每个状态用一个字符串存储，

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案一：
每个状态用一个字符串存储，
要 9 个字节，太浪费了！！！

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案二：

- 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。

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案二：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案二：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案二：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案二：

- 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!!!!

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案三：

- 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。

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案三：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

方案三：

- 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！！！！

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

- 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?

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

- 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！

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

方案四：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

方案四：

- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价
方案四：

- 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）。
- 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

例题 03: 八数码问题 (Eight)

用合理的编码表示“状态”，减小存储代价

方案五：

- 还是把一个状态看作一个数的 10 进制表示形式
- 用set<int>/unordered_set<int>进行判重。每入队一个状态，就将其加到set/unordered_set<int>里面，判重时，查找该set/unordered_set<int>，看能否找到状态

例题 03: 八数码问题 (Eight)

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 (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 $Y = 0 + 0 + 0 + 1 + 1 + 3 + 2 + 3 = 10,$，所以是偶排列。
  - 871625340 排列的 $Y = 0 + 0 + 0 + 1 + 1 + 2 + 2 + 3 = 9$，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。

例题 03: 八数码问题 (Eight)

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

例题 03: 八数码问题 (Eight)

```cpp
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cstdlib>
#include <unordered_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[] = "udrl"; //四种移动
```

例题 03: 八数码问题 (Eight)

```
22  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
23    char tmp[20];
24    int zeroPos;                    //字符'0'的位置
25    sprintf(tmp, "%09d", status); //需要保留前导 0
26    for (int i = 0; i < 9; ++i) {
27      if (tmp[i] != '0')
28        continue;
29      zeroPos = i;
30      break;
31    }
32    switch (move) {
33    case 'u':
34      if (zeroPos - 3 < 0) return -1; //空格在第一行
35      tmp[zeroPos] = tmp[zeroPos - 3];
36      tmp[zeroPos - 3] = '0';
37      break;
38    case 'd':
39      if (zeroPos + 3 > 8) return -1; //空格在第三行
40      tmp[zeroPos] = tmp[zeroPos + 3];
41      tmp[zeroPos + 3] = '0';
42      break;
```

例题 03: 八数码问题 (Eight)

```
43      case 'l':
44        if (zeroPos % 3 == 0) return -1; //空格在第一列
45        tmp[zeroPos] = tmp[zeroPos - 1];
46        tmp[zeroPos - 1] = '0';
47        break;
48      case 'r':
49        if (zeroPos % 3 == 2) return -1; //空格在第三列
50        tmp[zeroPos] = tmp[zeroPos + 1];
51        tmp[zeroPos + 1] = '0';
52        break;
53      }
54      return atoi(tmp);
55  }
```

例题 03: 八数码问题 (Eight)

```
57  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
58    int newStatus;
59    unordered_set<int> expanded;
60    myQueue[qHead] = Node(status, -1, 0);
61    expanded.insert(status);
62    while (qHead != qTail) { //队列不为空
63      status = myQueue[qHead].status;
64      if (status == goalStatus) //找到目标状态
65        return true;
66      for (int i = 0; i < 4; i++) { //尝试 4 种移动
67        newStatus = NewStatus(status, moves[i]);
68        if (newStatus == -1)
69          continue; //不可移，试下一种
70        if (expanded.find(newStatus) != expanded.end())
71          continue; //已扩展过，试下一种
72        expanded.insert(newStatus);
73        myQueue[qTail++] = Node(newStatus, qHead, moves[i]); //新节点入队列
74      }
75      qHead++;
76    }
77    return false;
78  }
```

例题 03: 八数码问题 (Eight)

```
80  int main() {
81    char line1[50], line2[20];
82    while (cin.getline(line1, 48)) {
83      int i, j;
84      for (i = 0, j = 0; line1[i]; i++) { //将输入的原始字符串变为数字字符串
85        if (line1[i] == ' ')
86          continue;
87        if (line1[i] == 'x')
88          line2[j++] = '0';
89        else
90          line2[j++] = line1[i];
91      }
92      line2[j] = 0; //字符串形式的初始状态
93      if (bfs(atoi(line2))) {
94        int moves = 0, pos = qHead;
95        do { //通过 father 找到成功的状态序列，输出相应步骤
96          result[moves++] = myQueue[pos].move;
97          pos = myQueue[pos].father;
98        } while (pos); // pos = 0 说明已经回退到初始状态了
99        for (int i = moves - 1; i >= 0; i--)
100         cout << result[i];
101     }
102     else {
103       cout << "unsolvable" << endl;
104     }
105   }
106   return 0;
107 }
```

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。空间换时间
- 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，时间换空间
  - 对于最优性问题，可能需要整个搜索完才能得到解

例题 03: 八数码问题 (Eight) 进阶

如何加快速度?

- 双向广搜
  从两个方向以广度优先的顺序同时扩展
- 针对本题预处理
  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。
- A* 算法

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。

DBFS vs BFS

- 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！

DBFS vs BFS

- 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: $(1-n^m)/(1-n)$

DBFS vs BFS

- 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 $2 * (1 - n^{m/2})/(1 - n)$

DBFS vs BFS

- 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。

双向广度优先搜索 (DBFS)

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

```
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:
    如果 adj 已经在队列 qi 之中出现过，则抛弃 adj;
    如果 adj 在队列 qi 中未出现过，则:
      将 adj 放入队列 qi;
      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

例题 03: 八数码问题 (Eight)

```cpp
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <unordered_set>
#include <algorithm>
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[2][MAXS]; //两个方向的状态队列，状态总数 362880
int matchingStatus;    //双向碰到的那个状态
int matchingQ;         // 队列 matchingQ 的队头元素是双向碰到的那个状态
int qHead[2];          //队头指针
int qTail[2];          //队尾指针
char moves[] = "udrl"; //四种移动
```

例题 03: 八数码问题 (Eight)

```
23  inline char ReverseMove(char c) {
24    switch (c) {
25    case 'u':
26      return 'd';
27    case 'l':
28      return 'r';
29    case 'r':
30      return 'l';
31    case 'd':
32      return 'u';
33    }
34    return 0;
35  }
36  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
37    char tmp[20];
38    int zeroPos;                    //字符'0'的位置
39    sprintf(tmp, "%09d", status); //需要保留前导 0
40    for (int i = 0; i < 9; ++i) {
41      if (tmp[i] != '0') continue;
42      zeroPos = i;
43      break;
44    }
```

例题 03: 八数码问题 (Eight)

```
45      switch (move) {
46      case 'u':
47        if (zeroPos - 3 < 0)
48          return -1; //空格在第一行
49        tmp[zeroPos] = tmp[zeroPos - 3];
50        tmp[zeroPos - 3] = '0';
51        break;
52      case 'd':
53        if (zeroPos + 3 > 8)
54          return -1; //空格在第三行
55        tmp[zeroPos] = tmp[zeroPos + 3];
56        tmp[zeroPos + 3] = '0';
57        break;
58      case 'l':
59        if (zeroPos % 3 == 0)
60          return -1; //空格在第一列
61        tmp[zeroPos] = tmp[zeroPos - 1];
62        tmp[zeroPos - 1] = '0';
63        break;
64      case 'r':
65        if (zeroPos % 3 == 2)
66          return -1; //空格在第三列
67        tmp[zeroPos] = tmp[zeroPos + 1];
68        tmp[zeroPos + 1] = '0';
69        break;
70      }
71      return atoi(tmp);
72    }
```

例题 03: 八数码问题 (Eight)

```
73  bool dbfs(int status) { //寻找从初始状态 status 到目标的路径，找不到则返回 false
74    int newStatus;
75    unordered_set<int> expanded[2];
76    qHead[0] = 0, qTail[0] = 1, qHead[1] = 0, qTail[1] = 1;
77    myQueue[0][0] = Node(status, -1, 0);
78    expanded[0].insert(status);
79    myQueue[1][0] = Node(goalStatus, -1, 0);
80    expanded[1].insert(goalStatus);
81    while (qHead[0] != qTail[0] && qHead[1] != qTail[1]) {          //两个队列不都为空
82      int qNo; //本次要扩展的队列
83      if (qHead[0] == qTail[0])
84        qNo = 1;
85      else if (qHead[1] == qTail[1])
86        qNo = 0;
87      else if (qTail[0] - qHead[0] < qTail[1] - qHead[1])
88        qNo = 0; //比较两个队列元素个数
89      else
90        qNo = 1;
91      int vqNo = 1 - qNo; //另一队列
92      status = myQueue[qNo][qHead[qNo]].status;
93      if (expanded[vqNo].find(status) != expanded[vqNo].end()) {
94        // status 在另一队列扩展过，路径找到
95        matchingStatus = status;
96        matchingQ = qNo;
97        return true;
98      }
```

例题 03: 八数码问题 (Eight)

```
99      for (int i = 0; i < 4; i++) { //尝试 4 种移动
100       newStatus = NewStatus(status, moves[i]);
101       if (newStatus == -1)
102         continue; //不可移，试下一种
103       if (expanded[qNo].find(newStatus) != expanded[qNo].end())
104         continue; //如果已经扩展过，则不能入队
105       expanded[qNo].insert(newStatus);
106       myQueue[qNo][qTail[qNo]] = Node(newStatus, qHead[qNo], moves[i]);
107       qTail[qNo]++;
108     }
109     qHead[qNo]++;
110   }
111   return false;
112 }
113 int main() {
114   char line1[50], line2[20];
115   while (cin.getline(line1, 48)) {
116     int i, j;
117     //将输入的原始字符串变为数字字符串
118     for (i = 0, j = 0; line1[i]; i++) {
119       if (line1[i] == ' ')
120         continue;
121       if (line1[i] == 'x')
122         line2[j++] = '0';
123       else
124         line2[j++] = line1[i];
125     }
126     line2[j] = 0; //字符串形式的初始状态
```

例题 03: 八数码问题 (Eight)

```
127     if (dbfs(atoi(line2))) {
128       int moves = 0, pos;
129       if (matchingQ == 0) {
130         pos = qHead[0];
131       } else {
132         for (int i = 0; i < qTail[0]; ++i) {
133           if (myQueue[0][i].status == matchingStatus) {
134             pos = i;
135             break;
136           }
137         }
138       }
139       while (pos) {
140         result[moves++] = myQueue[0][pos].move;
141         pos = myQueue[0][pos].father;
142       }
143       reverse(result, result + moves);
```

例题 03: 八数码问题 (Eight)

```
144       if (matchingQ == 0) {
145         for (int i = 0; i < qTail[1]; ++i) {
146           if (myQueue[1][i].status == matchingStatus) {
147             pos = i;
148             break;
149           }
150         }
151       } else {
152         pos = qHead[1];
153       }
154       while (pos) {
155         result[moves++] = ReverseMove(myQueue[1][pos].move);
156         pos = myQueue[1][pos].father;
157       }
158       for (int i = 0; i < moves; ++i)
159         cout << result[i];
160       cout << endl;
161     } else {
162       cout << "unsolvable" << endl;
163     }
164   }
165   return 0;
}
```