# 《算法设计与分析》

# 第七章 分枝—限界法

马丙鹏

2023年11月13日

中国科学院大学
University of Chinese Academy of Sciences

# 第七章 分枝－限界法

- 7.1 一般方法
- 7.2 LC-检索
- 7.3 15-谜问题
- 7.4 布线问题
- 7.5 LC-检索(续)
- 7.6 分枝-限界算法
- 7.7 0/1背包问题
- 7.8 货郎担问题

# 7.3 15-谜问题

## ■ 问题描述

- □在一个分成16格的方形棋盘上放有15块编了号的牌。对于这些牌给定的一种初始排列，要求通过一系列的合法移动将初始排列转换成目标排列。
- □合法移动：每次将一个邻接于空格的牌移动到空格位置。

(a) 初始排列

| | | | |
|---|---|---|---|
| 1 | 3 | 4 | 15 |
| 2 | | 5 | 12 |
| 7 | 6 | 11 | 14 |
| 8 | 9 | 10 | 13 |

(b) 目标排列

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

# 7.3 15-谜问题

- 问题状态：
  - 15块牌在棋盘上的任一种排列。
  - 初始状态：初始排列(任意给定的)
  - 目标状态：目标排列(确定的)
  - 若由初始状态到某状态存在一系列合法移动，则称该状态可由初始状态到达。

# 7.3 15-谜问题

- 问题状态：
  - 目标状态是否可由初始状态到达？
    - 在求解问题前，首先需要判定目标状态是否在初始状态的状态空间中。
      - ✓并不是所有的初始状态都能变换成目标状态。
      - ✓棋盘存在16！种(约20万亿种)不同排列，而对于任一给定的初始状态，可到达的状态为这些排列中的一半。

# 7.3 15-谜问题

- 如何判定目标状态在初始状态的状态空间中？
  - 判定条件由以下4步给出：

① 给棋盘的方格位置编号：按照目标状态中各块牌在棋盘上的排列给对应方格编号，空格为16。

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

目标排列

# 7.3 15-谜问题

- 如何判定目标状态在初始状态的状态空间中？
  - 判定条件由以下4步给出：

② 记POSITION(i)为编号为i的牌在初始状态中的位置；POSITION(16)表示空格的位置。

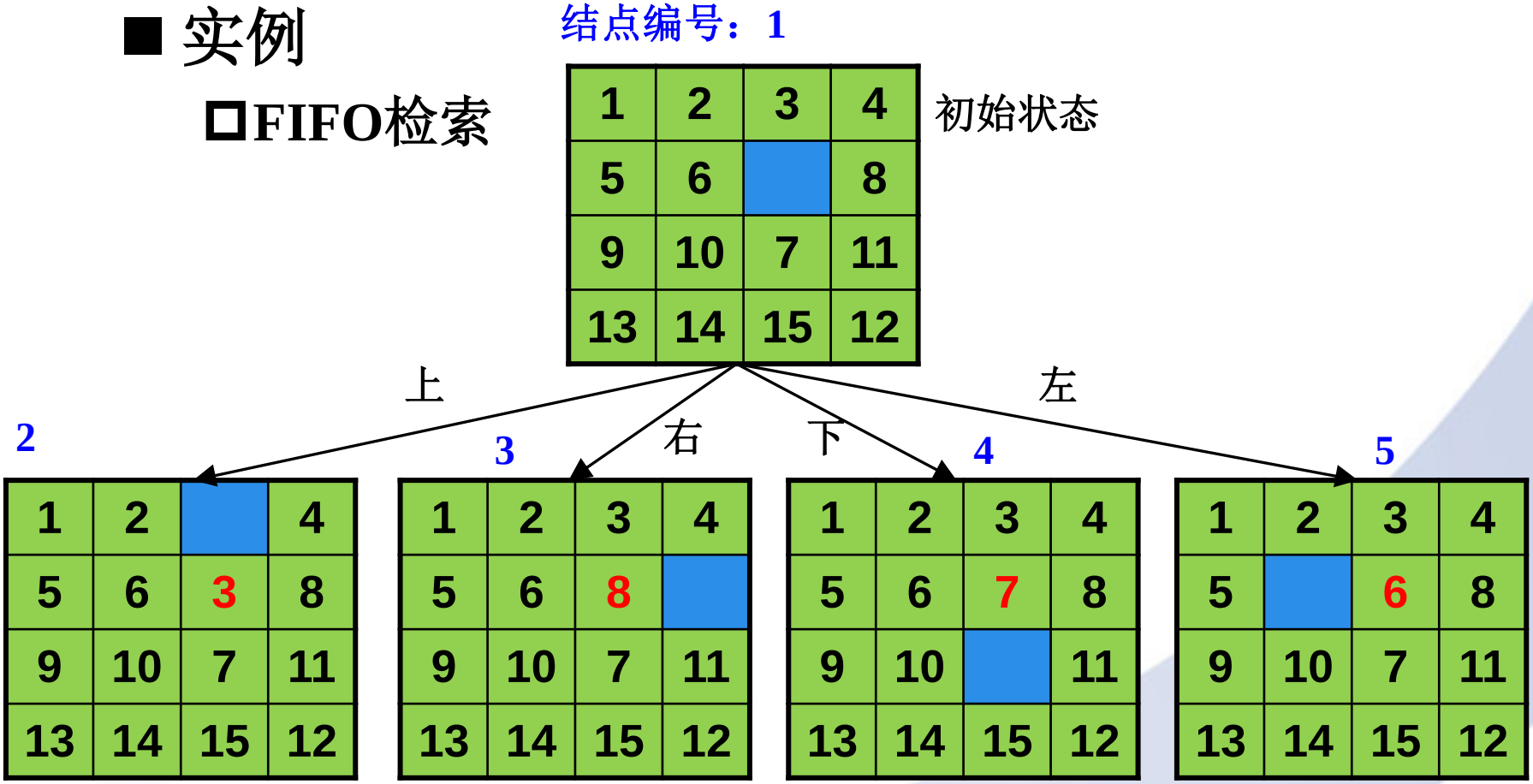
| | | | |
|---|---|---|---|
| 1 | 3 | 4 | 15 |
| 2 | | 5 | 12 |
| 7 | 6 | 11 | 14 |
| 8 | 9 | 10 | 13 |

**POSITION(1:16) = (1, 5, 2, 3, 7, 10, 9, 13, 14, 15, 11, 8, 16, 12, 4, 6)**

# 7.3 15-谜问题

■ 实例

□FIFO检索



# 7.3 15-谜问题

- 如何判定目标状态在初始状态的状态空间中？
  - 判定条件由以下4步给出：

③ 记LESS(i)是这样牌j的数目：

j<i，但POSITION(j) > POSITION(i)，

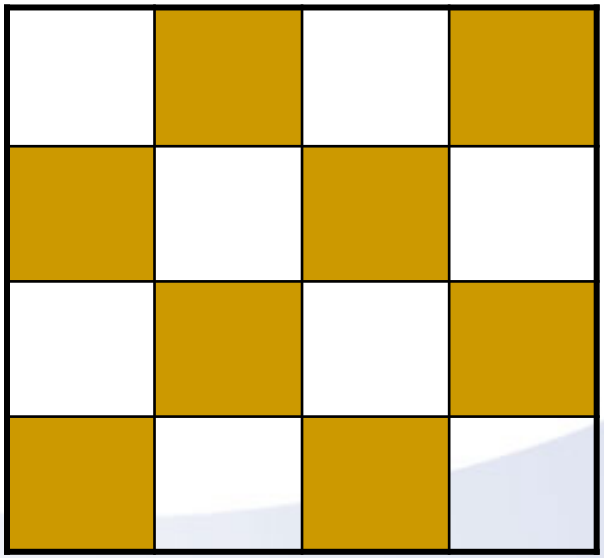
即：编号小于i但初始位置在i之后的牌的数目。

例：LESS(1)=0; LESS(4)=1; LESS(12)=6

| | | | |
|---|---|---|---|
| 1 | 3 | 4 | 15 |
| 2 | | 5 | 12 |
| 7 | 6 | 11 | 14 |
| 8 | 9 | 10 | 13 |

# 7.3 15-谜问题

- 如何判定目标状态在初始状态的状态空间中？
  - 判定条件由以下4步给出：
    - ④ 引入一个量X如图所示，初始状态时，
      - ✓若空格落在橙色方格上，则X=1；
      - ✓若空格落在白色方格上，则X=0。

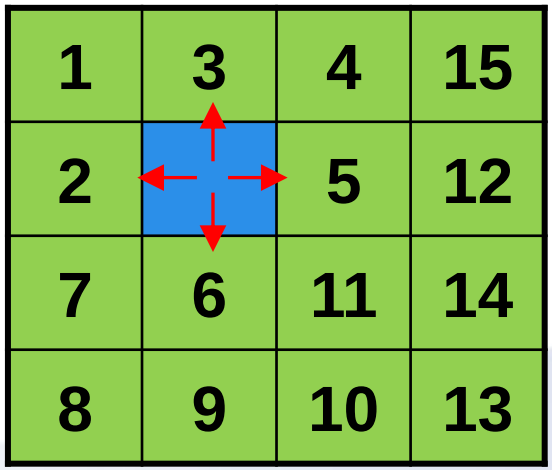


# 7.3 15-谜问题

- 如何判定目标状态在初始状态的状态空间中？
  - 目标状态是否在初始状态的状态空间中的判别条件由定理7.1给出：
  - 定理7.1 当且仅当 $\sum_{i=1}^{16} LESS(i) + X$是偶数时，目标状态可由此初始状态到达。
    证明 (略)

# 7.3 15-谜问题

- 15迷问题状态空间树的构造
  - 从初始状态出发，每个结点的儿子结点表示由该状态通过一次合法的移动而到达的状态。
  - 注：移动牌与移动空格是等效的。状态空间树中标注空格的一次合法移动。
  - 空格的一次合法移动有四个可能的方向：上、下、左、右。



# 7.3 15-谜问题

■ 实例

□初始状态

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

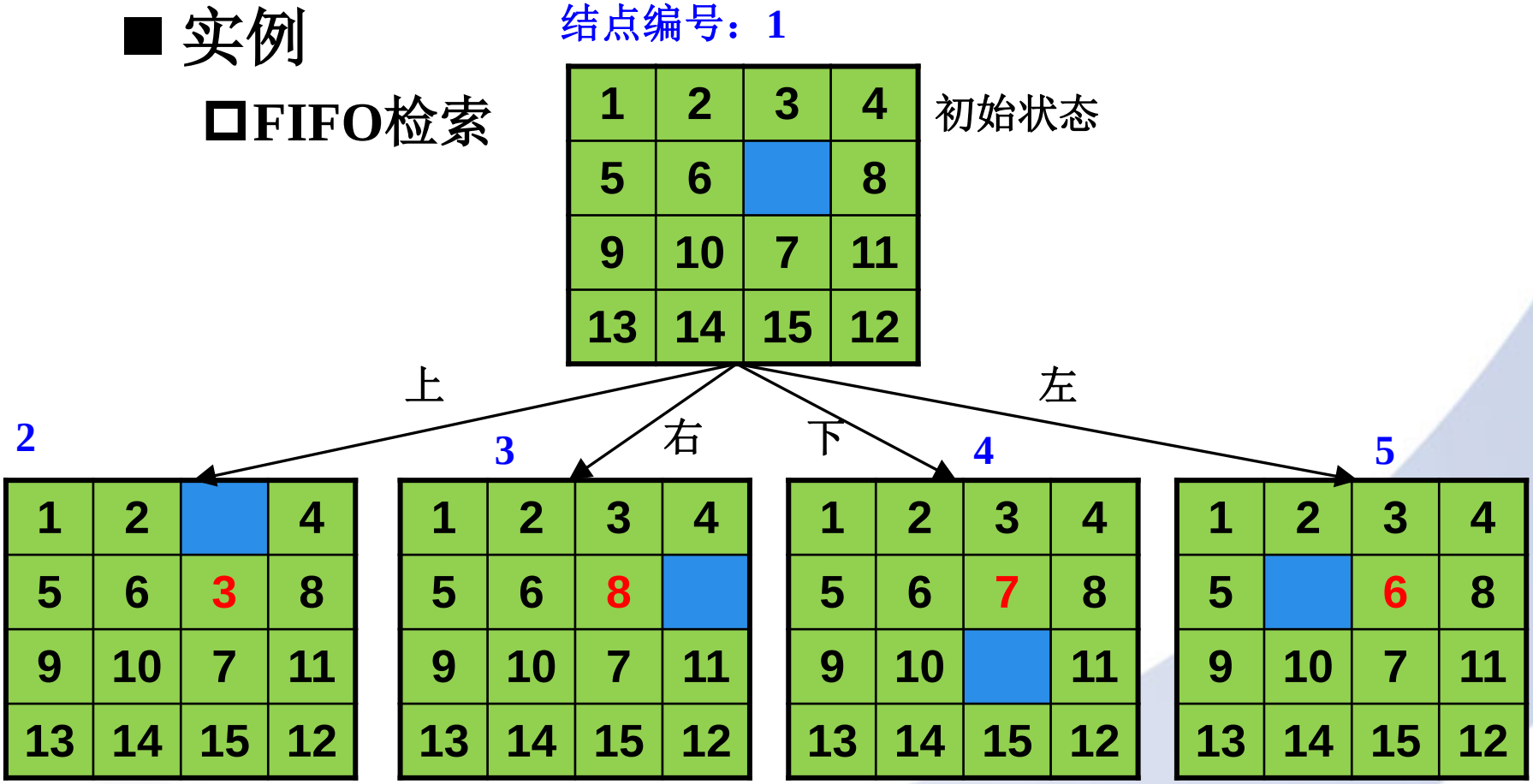
□剪枝策略：

➢若P的儿子中有与P的父结点重复的，则剪去该分枝。

# 7.3 15-谜问题

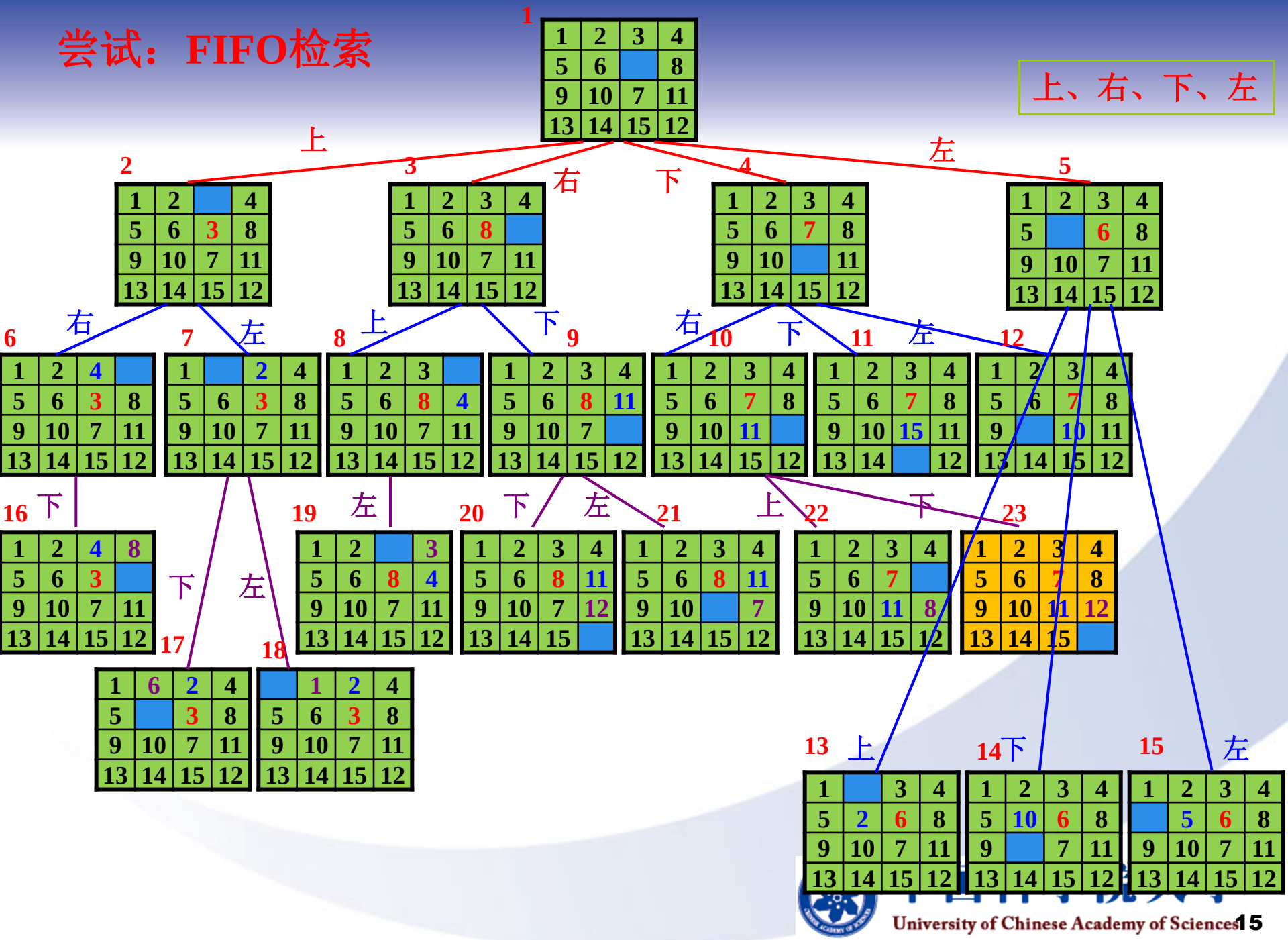
■ 实例

□FIFO检索



# 尝试：FIFO检索

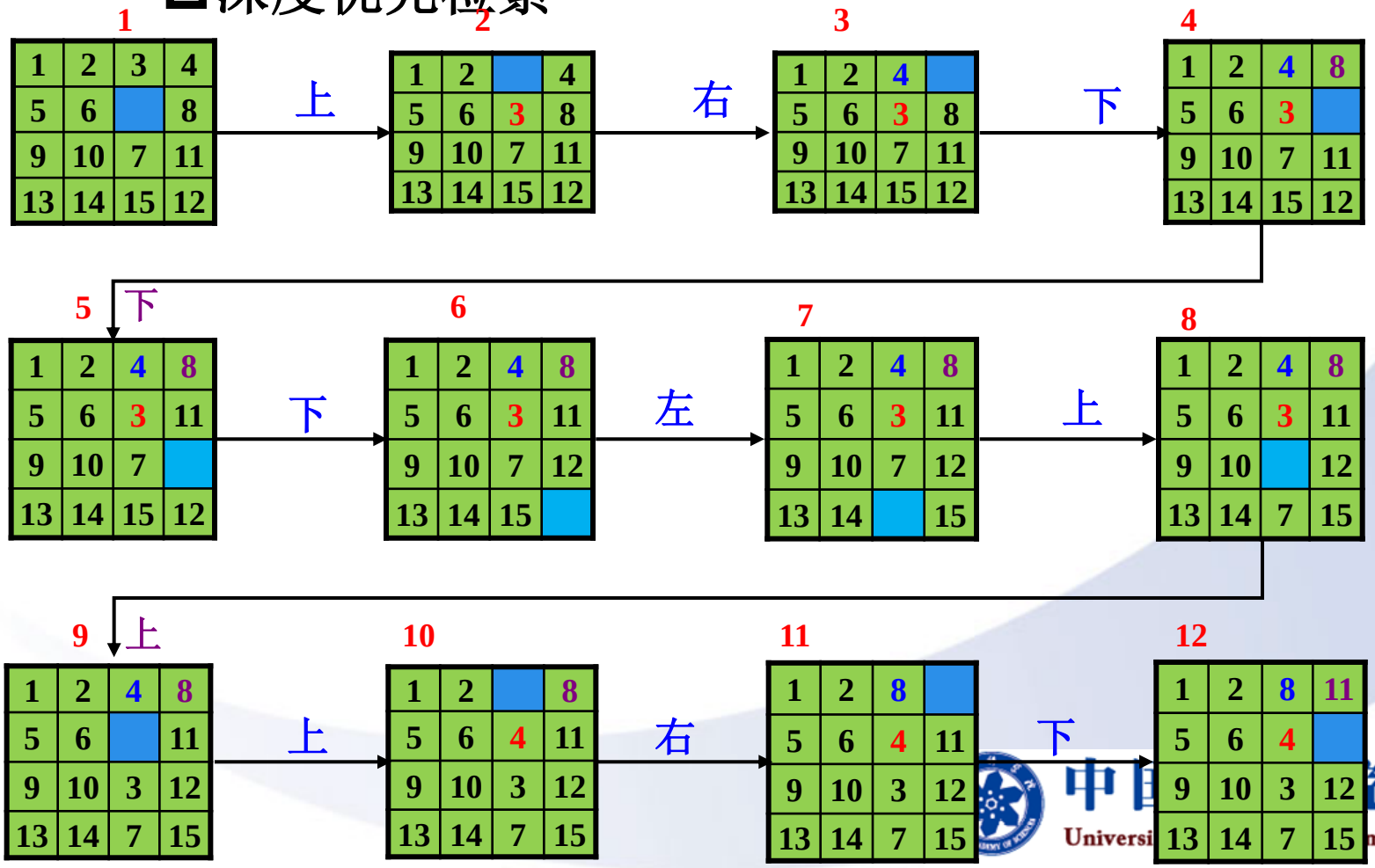
上、右、下、左



# 7.3 15-谜问题

## ■ 实例

### □深度优先检索



# 7.3 15-谜问题

- 实例
  - 简单检索存在的问题：呆板和盲目
    - 是否能够找到一种“智能”方法，对不同的实例做不同的处理？
    - 策略：给状态空间树中的每个结点赋予成本值c(X)

# 7.3 15-谜问题

- 实例
  - 如何定义c(X)?
    - 如果实例有解，则将由根出发到最近目标结点路径上的每个结点赋以这条路径的长度作为它们的成本。
    - 例：
      - ✓c(1)=c(4)=c(10)=c(23)=3
      - ✓其余结点的成本均为∞
      - ✓检索时杀死成本为∞的结点
    - 该方法实际上是不可操作的——c(X)不可能通过简单的方法求出。精确计算c(X)的工作量与求解原始问题相同。

# 7.3 15-谜问题

- 实例
  - 如何定义c(X)?
    - 估算法：定义成本估计函数$\hat{c}(X)$

$$\hat{c}(X) = f(X) + \hat{g}(X)$$

其中，

① f(X)是由根到结点X的路径长度

② $\hat{g}(X)$是以X为根的子树中由X到目标状态的一条最短路径长度的估计值—— $\hat{g}(X)$ 至少应是能把状态X转换成目标状态所需的最小移动数。故，令

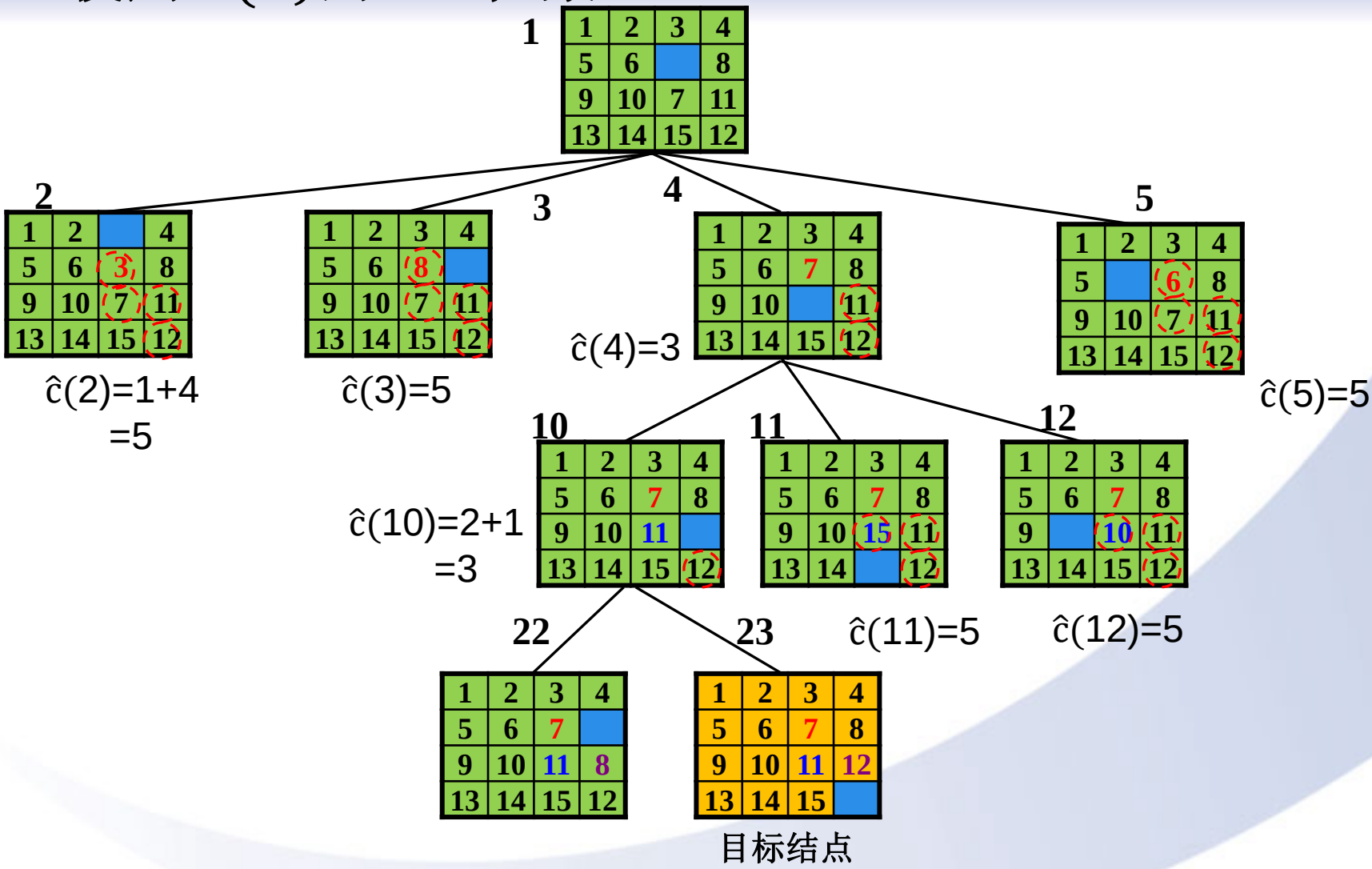
$\hat{g}(X)$ = 不在其目标位置的非空白牌数目

# 7.3 15-谜问题

- 实例
  - 如何定义c(X)?
    - 估算法：
      - ✓例：$\hat{g}(X) = 3$
        注：为达到目标状态所需的实际移动数>>$\hat{g}(X)$
      - ✓$\hat{c}(X)$是c(X)的下界

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

# 例：使用 ĉ(X)的LC-检索




使用ĉ(X)的LC-检索可以使检索过程很快地定位到结点23

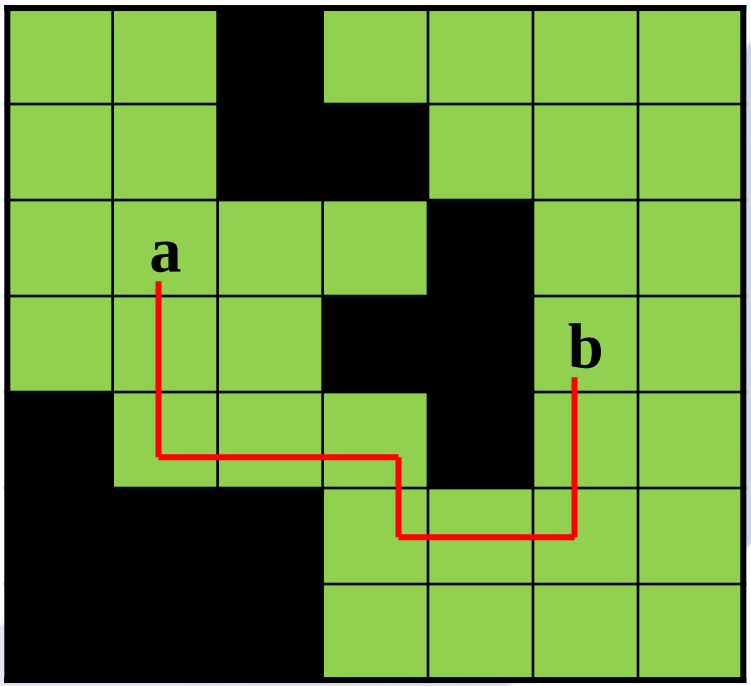
# 第七章 分枝－限界法

- 7.1 一般方法
- 7.2 LC-检索
- 7.3 15-谜问题
- 7.4 布线问题
- 7.5 LC-检索(续)
- 7.6 分枝-限界算法
- 7.7 0/1背包问题
- 7.8 货郎担问题

# 7.4 布线问题

## ■ 问题描述

- 印刷电路板将布线区域划分成m×n的方格阵列。精确的布线问题要求确定连接方格a的中点到方格b的中点的最短布线方案。
- 要求
  - 在布线时，电路只能沿直线或直角布线，
  - 为了避免线路相交，已经布线了的方格要做封锁标记，其它线路不允许穿过被封锁了的方格。



# 7.4 布线问题

## ■ 算法思想

- 解此问题的队列式分支限界法从起始位置a开始将它作为第一个扩展结点。
  与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列中，并且将这些方格标记为1，即从起始方格a到这些方格的距离为1。
- 接着，算法从活结点队列中取出队首结点作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格标记为2，并存入活结点队列。
- 这个过程一直继续到算法搜索到目标方格b或活结点队列为空时为止。即加入剪枝的广度优先搜索。

# 7.4 布线问题

## ■ 算法实现

- 方格位置：定义一个表示电路板上方格位置的类Position类，成员row和col表示行和列
- 移动方向：在方格处，布线可沿右、下、左、上4个方向进行。沿这4个方向的移动分别记为0，1，2，3。下表中，offset[i].row和offset[i].col(i= 0,1,2,3)分别给出沿这4个方向前进1步相对于当前方格的相对位移。

| 移动i | 方向 | offset[i].row | offset[i].col |
|---|---|---|---|
| 0 | 右 | 0 | 1 |
| 1 | 下 | 1 | 0 |
| 2 | 左 | 0 | -1 |
| 3 | 上 | -1 | 0 |

# 7.4 布线问题

- 算法实现
  - 用二维数组grid表示所给的方格阵。
    - 初始时，grid[i][j] = 0, 表示该方格允许布线，而grid[i][j] = 1表示该方格被封锁，不允许布线。
    - 为了便于处理方格边界的情况，算法在所给方格阵列的四周设置一道“围墙”，即增设标记为“1”的附加方格。

# 7.4 布线问题

- 算法实现
  - 初始化
    - 算法开始时，测试初始方格与目标方格是否相同。如果相同，则不必计算，直接返回最短距离0，否则，算法设置方格阵列的边界，初始化位移矩阵offset。
    - 因为数字0和1用于表示方格的开放或封锁状态，所以在表示距离时不用这两个数字。算法将起始位置的距离标记为2，实际距离应为标记距离-2。

# 7.4 布线问题

- 算法实现
  - 算法搜索步骤：
    - 算法从起始位置start开始，标记所有标记距离为3的方格并存入活节点队列，
      然后依次标记所有标记距离为4, 5, 6, …的方格，
      直到目标方格finish或活节点队列为空时为止。

# 7.4 布线问题

## ■ 算法实现

```
static class Position
{
   private int row; //方格所在的行
   private int col;  //方格所在的列
static  int[][] grid; //方格阵列,存储距离值
static int size;       //方格阵列大小
static int pathLen;  //最短线路长度
static ArrayQueue  q; //扩展结点队列
static Position statrt, finish; //起点，终点
static  Position [] path; //最短路径
```

# 7.4 布线问题

## ■ 算法实现

```
Position []offset = new Position[4];
  offset[0]=new Position(0,1);  // 右
  offset[1]=new Position(1,0);  // 下
  offset[2]=new Position(0,-1); // 左
  offset[3]=new Position(-1,0); // 上
```

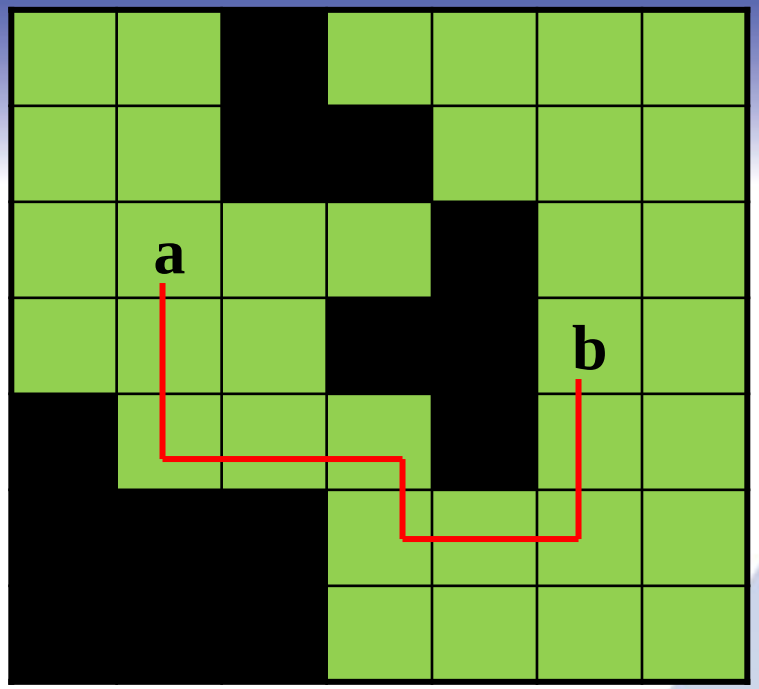
定义移动方向的相对位移

```
 for (int i = 0; i <= size+1; i++){
     grid[0][i] = grid[size+1][i] = 1; // 顶部和底部
     grid[i][0] = grid[i][size+1] = 1; // 左翼和右翼
}
```

设置边界的围墙

# 7.4 布线问题

■ 算法实现




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | | | 0 | 0 | 0 | 1 |
| 1 | 0 | 2 | 0 | | | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | | | 0 | 0 | 1 |
| 1 | | 0 | 0 | 0 | | 0 | 0 | 1 |
| 1 | | | | 0 | 0 | 0 | 0 | 1 |
| 1 | | | | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| R | | | | | | | | 3 |
| C | | | | | | | | 2 |

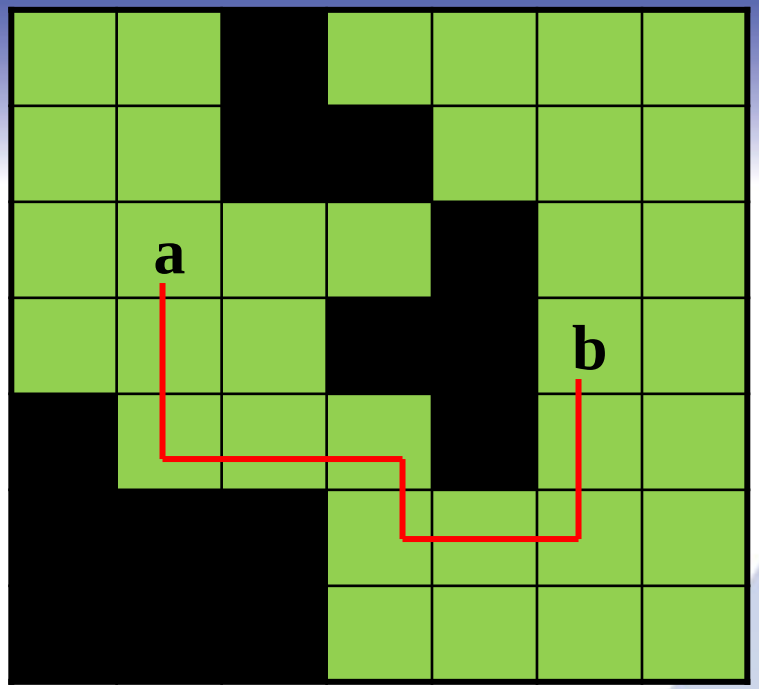
# 7.4 布线问题

## ■ 算法实现

```
for (int i = 0; i < 4; i++) { //四周方格数
        nbr.row = here.row + offset[i].row;
        nbr.col = here.col + offset[i].col;
        if (grid[nbr.row][nbr.col] == 0)
         { // 该方格未标记
         grid[nbr.row][nbr.col]= grid[here.row][here.col] + 1;
           if ((nbr.row == finish.row) &&(nbr.col == finish.col))
              break; // 完成布线
              q.put(new Position(nbr.row,nbr.col));
           }
  }
```

# 7.4 布线问题

■ 算法实现

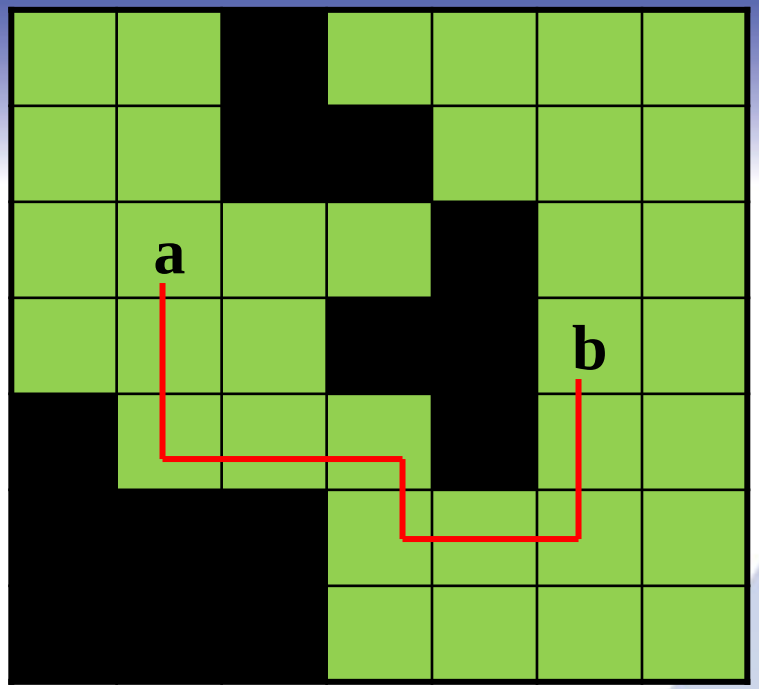



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | | 0 | | | | 1 |
| 1 | 0 | 3 | | | | | | 1 |
| 1 | 3 | 2 | 3 | | | | | 1 |
| 1 | 0 | 3 | 0 | | | b | | 1 |
| 1 | | 0 | 0 | 0 | | 0 | | 1 |
| 1 | | | | 0 | 0 | 0 | 0 | 1 |
| 1 | | | | 0 | 0 | 0 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| R | | | | | | | 2 | 3 | 4 | 3 |
| C | | | | | | | 2 | 1 | 2 | 3 |
| g | | | | | | | 3 | 3 | 3 | 3 |

# 7.4 布线问题

■ 算法实现

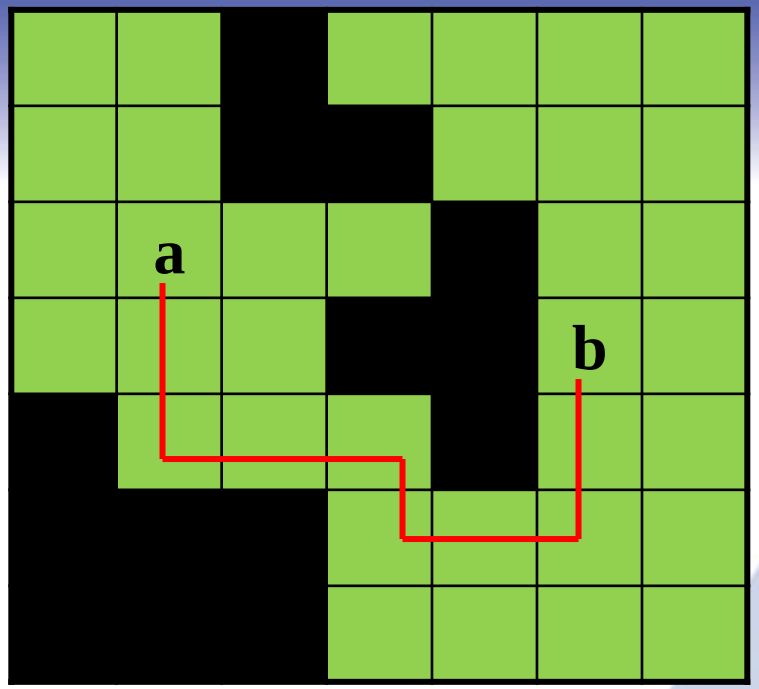



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | | 0 | | | | 1 |
| 1 | 0 | 3 | | | | | | 1 |
| 1 | 3 | 2 | 3 | 4 | | | | 1 |
| 1 | 0 | 3 | 4 | | | b | | 1 |
| 1 | | 0 | 0 | 0 | | 0 | | 1 |
| 1 | | | | 0 | 0 | 0 | 0 | 1 |
| 1 | | | | 0 | 0 | 0 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| R | | | | | | 4 | 3 | 2 | 3 | 4 |
| C | | | | | | 3 | 4 | 2 | 1 | 2 |
| g | | | | | | 4 | 4 | 3 | 3 | 3 |

# 7.4 布线问题

■ 算法实现

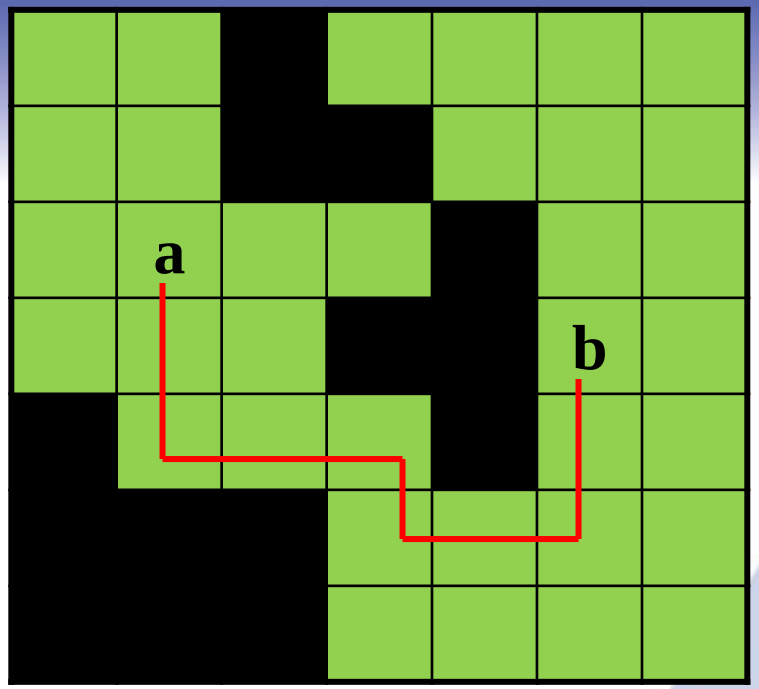



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | | 0 | | | | 1 |
| 1 | 0 | 3 | | | | | | 1 |
| 1 | 3 | 2 | 3 | 4 | | | | 1 |
| 1 | 4 | 3 | 4 | | | b | | 1 |
| 1 | | 4 | 0 | 0 | | | | 1 |
| 1 | | | | 0 | 0 | 0 | 0 | 1 |
| 1 | | | | 0 | 0 | 0 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| R | | | | 4 | 5 | 4 | 3 | 2 | 3 |
| C | | | | 1 | 2 | 3 | 4 | 2 | 1 |
| g | | | | 4 | 4 | 4 | 4 | 3 | 3 |

# 7.4 布线问题

■ 算法实现

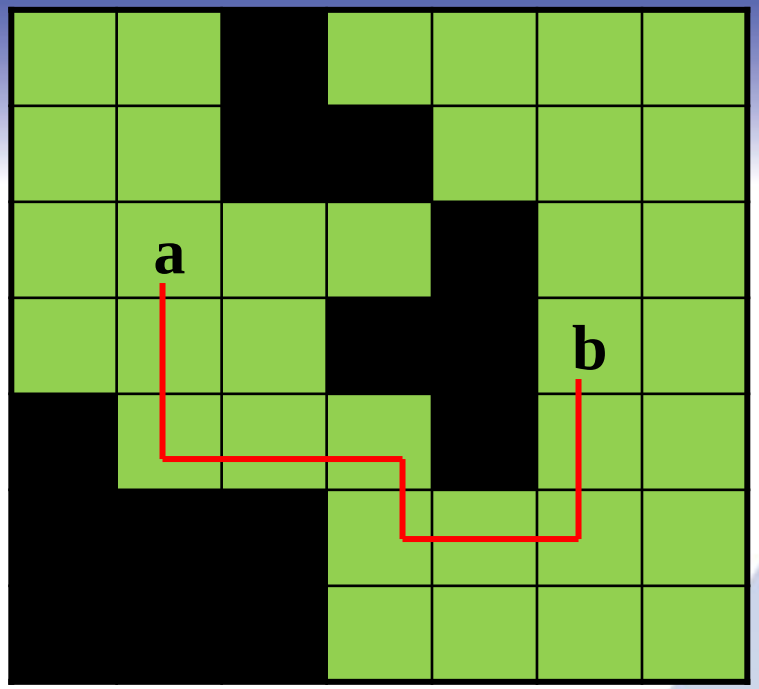



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 5 | 4 | | 0 | | | | 1 |
| 1 | 4 | 3 | | | | | | 1 |
| 1 | 3 | 2 | 3 | 4 | | | | 1 |
| 1 | 4 | 3 | 4 | | | b | | 1 |
| 1 | | 4 | 5 | 6 | | 10 | | 1 |
| 1 | | | | 7 | 8 | 9 | 10 | 1 |
| 1 | | | | 8 | 9 | 10 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| R | | | 2 | 4 | 4 | 5 | 4 | 3 | 2 |
| C | | | 1 | 1 | 1 | 2 | 3 | 4 | 2 |
| g | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 |

# 7.4 布线问题

■ 算法实现




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 3 | 2 | | | | | | 1 |
| 1 | 2 | 1 | | | | | | 1 |
| 1 | 1 | a | 1 | | | | | 1 |
| 1 | 2 | 1 | 2 | | | b | | 1 |
| 1 | | 2 | 3 | 4 | | 8 | | 1 |
| 1 | | | | 5 | 6 | 7 | 8 | 1 |
| 1 | | | | 6 | 7 | 8 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# 7.4 布线问题

找到目标位置后，可以通过回溯方法找到这条最短路径。

```
pathLen=grid[finish.row][finish.col]-2;
path= new Position[pathLen];
here=finish; // 从目标位置向起始位置回溯
for (int j = pathLen-1; i>=0; j--) {
        path[j]= here;
       for(int i=0; i<4; i++){ // 找前驱位置
           nbr.row = here.row + offset[i].row;
           nbr.col = here.col + offset[i].col;
           if (grid[nbr.row][nbr.col] == j+2)break;
        }
        here= new Position (nbr.row,nbr.col); //向前移动
}
```

# 7.4 布线问题

pathLen=9， j=8

b:(4, 6),

右:(4, 7)

下:(5, 6), 10=8+2 √

左:(4, 5)

上:(3, 6)

:

:

:

(5, 3)

右:(5, 4)

下:(6, 3)

左:(5, 2), 4=2+2 √

上:(4, 3)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 5 | 4 | | 0 | | | | 1 |
| 1 | 4 | 3 | | | | | | 1 |
| 1 | 3 | 2 | 3 | 4 | | | | 1 |
| 1 | 4 | 3 | 4 | | | b | | 1 |
| 1 | | 4 | 5 | 6 | | 10 | | 1 |
| 1 | | | | 7 | 8 | 9 | 10 | 1 |
| 1 | | | | 8 | 9 | 10 | | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# 7.4 布线问题

- 复杂度分析
  - 计算时间为O(mn)+O(L)
    - 由于每个方格成为活结点进入活结点队列最多1次，因此活结点队列中最多只处理O(mn)。
    - 每个扩展结点需O(1)时间，因此算法共耗时O(mn)。
    - 构造相应的最短距离需要O(L)时间，其中L是最短布线路径的长度。
  - 需要空间为 O(mn)。

# 7.4 布线问题

- 讨论：
  - 为什么这条路径一定就是最短的呢？
    - 这是由于我们这个搜索过程的特点所决定的，
    - 假设存在一条由a至b的更短的路径，b结点一定会更早地被加入到活节点队列中并得到处理。

# 7.4 布线问题

- 讨论：
  - 为什么用回溯法来处理就是相当低效的？
    - 回溯法的搜索是依据深度优先的原则进行的，
    - 如果把上下左右四个方向规定一个固定的优先顺序去进行搜索，搜索会沿着某个路径一直进行下去直到碰壁才换到另一个子路径。
    - 但是最开始根本无法判断正确的路径方向是什么？这就造成了搜索的盲目和浪费。
    - 即使搜索到了一条由a至b的路径，根本无法保证它就是所有路径中最短的。
    - 这要求必须把整个区域的所有路径逐一搜索后才能得到最优解。
  - 所以，布线问题不适合用回溯法解决。

# 第七章 分枝－限界法

- 7.1 一般方法
- 7.2 LC-检索
- 7.3 15-谜问题
- 7.4 布线问题
- 7.5 LC-检索(续)
- 7.6 分枝-限界算法
- 7.7 0/1背包问题
- 7.8 货郎担问题

# 7.5 LC-检索(Least Cost)

- LC-检索的抽象化控制
  - 成本估计函数的动机
    - 设T是一棵状态空间树，c( ·)是T的结点成本函数。
    - c(X)是根为X的子树中任一答案结点的最小成本。
    - c(T)是T中最小成本答案结点。
    - c( ·)难以找到 => 成本估计函数ĉ(X)
  - 成本估计函数的性质
    - 如果X是一个答案结点或者是一个叶结点，则c(X) = ĉ(X)。
  - LC-检索的抽象化控制：
    - 过程LC用ĉ(•)去寻找一个答案结点。

```
procedure LC(T, ĉ)
 //为找答案结点检索T, ĉ为结点成本估计函数//
  if T是答案结点 then 输出T; return endif   //T为答案结点，输出T//
  E ← T   //E-结点//
  将活结点表初始化为空
  loop
      for E的每个儿子X do
          if X是答案结点 then 输出从X到T的路径；return endif
          call ADD(X)  //X是新的活结点，ADD将X加入活结点表中//
          PARENT(X) ← E //指示到根的路径//
      repeat
      if 不再有活结点 then    print("no  answer code")   stop  endif
      call LEAST(E)  //从活结点表中找ĉ最小的活结点，赋给E，并从活结点表
      中删除//
  repeat
end LC
```

找到答案结点，输出到根的路径

# 7.5 LC-检索(Least Cost)

- LC-检索的抽象化控制
  - 说明：
    - **LEAST(X)：**
      在活结点表中找一个具有最小值的活结点，从活结点表中删除这个结点，并将此结点放在变量X中返回。
    - **ADD(X)：**
      将新的活结点X加到活结点表中。
    - 活结点表：
      以min-堆结构存放。

# 7.5 LC-检索(Least Cost)

- LC的正确性证明
  - 算法LC是正确的。
    - 满足确定性、能行性、有输入、输出。
    - 满足终止性
      - ✓对于有限状态空间树，在找到一个答案结点或者在生成并检索了整棵状态空间树后算法终止。
      - ✓对于无限状态空间树，在找到一个答案结点或者对成本估计函数做出适当限制后也能使得算法终止。
    - 因此LC算法正确。
  - (详细证明略)

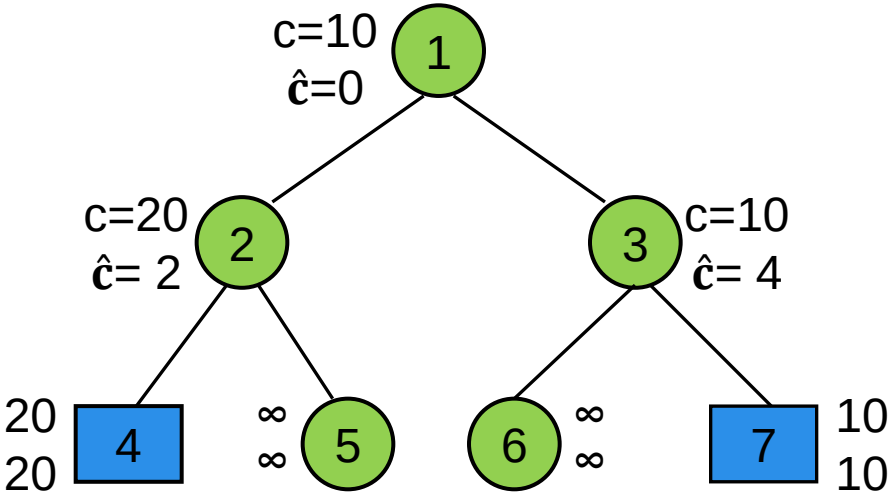
# 7.5 LC-检索(Least Cost)

- LC-检索与FIFO-检索和D-检索的关系
  - LC算法与FIFO-检索及D-检索基本相同：
    - 若活结点表采用队列，用LEAST(X)和ADD(X)从队列中删除或加入元素，则LC就变成了 FIFO-检索
    - 若活节点表采用栈，用LEAST(X)和ADD(X)从栈中删除或加入元素，则LC就变成了 D-检索
  - 不同：
    - 对下一个E-结点的选择规则不同。

# 7.5 LC-检索(Least Cost)

## ■ LC-检索的特性

- □当有多个答案结点时，LC是否一定找得到具有最小成本的答案结点呢？否




首先扩展结点1，得结点2，3；$\hat{c}(2) = 2 < \hat{c}(3) = 4$；

然后扩展结点2，得答案结点4，$c(4) = 20$；

实际最小成本的答案结点是7，$c(7) = 10$。

# 7.5 LC-检索(Least Cost)

- LC-检索的特性
  - 当有多个答案结点时，LC是否一定找得到具有最小成本的答案结点呢？否
  - 原因：
    - 存在这样的结点X和Y：
      当$c(X) > c(Y)$时，$\hat{c}(X) < \hat{c}(Y)$
  - 改进策略1：
    - 约定：
      对每一对$c(X) < c(Y)$的结点X和Y，有$\hat{c}(X) < \hat{c}(Y)$
    - 目标：使得LC总会找到一个最小成本的答案结点(如果状态空间树中有答案结点的话)
    - 找到这样的$\hat{c}(X)$通常是不可能的。

# 7.5 LC-检索(Least Cost)

- LC-检索的特性
  - 定理7.2 在有限状态空间树T中，对于每一个结点X，令是c(X)的估计值且具有以下性质：
    对于每一对结点Y、Z，当且仅当c(Y)<c(Z)时，有 $\hat{c}(Y) < \hat{c}(Z)$。
    那么在使$\hat{c}(\cdot)$作为c( )的估计值时，算法LC到达一个最小的成本答案结点终止。
  - 证明：(略)

# 7.5 LC-检索(Least Cost)

- LC-检索的特性
  - 改进策略2：
    - 对结点成本函数做如下限定：
      对于每一个结点X有$\hat{c}(X) \leq c(X)$且对于答案结点X有 $\hat{c}(X) = c(X)$ 。
  - 算法LC：
    - 仍不能保证算法LC找到最小成本答案结点
  - 算法LC1：
    - 找最小成本答案结点的改进算法，该算法可以找到成本最小的答案结点。

# 7.5 LC-检索(Least Cost)

```
procedure LC1(T, ĉ)
 //为找出最小成本答案结点检索T, ĉ为具有上述性质的结点成本估计函数//
  E ← T //第一个E-结点//
  置活结点表为空
  loop
      if E是答案结点 then 输出从E到T的路径；return endif
      for E的每个儿子X do
          call ADD(X)  //X是新的活结点，ADD将X加入活结点表中//
          PARENT(X) ← E //指示到根的路径//
      repeat
      if 不再有活结点 then    print("no  answer code")；stop  endif
      call LEAST(E)  //从活结点表中找ĉ 最小的活结点，赋给X，并从活结点表
      中删除//
  repeat
end LC1
```

生成结点后立即加入活结点表

# 7.5 LC-检索(Least Cost)

- 算法LC1正确性的证明
  - 定理7.3 令$\hat{c}(\bullet)$是满足如下条件的函数，
    在状态空间树T中，对于每一个结点X，有
    $\hat{c}(X) \le c(X)$，
    而对于T中的每一个答案结点X，有 $\hat{c}(X) = c(X)$。
    如果算法在第一个return处终止，则所找到的答案结点是具有最小成本的答案结点。
  - 证明：设此时E-结点是答案结点，L为活结点表中的任意一个结点。

① $\hat{c}(E) \le \hat{c}(L)$

② $\hat{c}(E) = c(E)$

③ $\hat{c}(L) \le c(L)$

$c(E) \le c(L)$

# End