# Foundation of Artificial Intelligence

**Prof. Fangshi Wang**

Beijing Jiaotong University

Email: fshwang@bjtu.edu.cn

# 第3章　搜索策略

# 第3章 搜索策略

- 在求解一个问题时涉及两个方面：
  1. 该**问题的表示**，如果一个问题找不到一个合适的表示方法，就谈不上对它进行求解;
  2. 选择一种相对合适的**求解方法**。
- 在人工智能领域,问题求解的基本方法有**搜索法**、**演绎法**、**归纳法**、**推理法**等。
- 由于绝大多数需要用人工智能方法求解的问题**缺乏直接求解的方法**，因此，**搜索不失为一种求解问题的一般方法**。搜索求解的应用非常广泛。
- 首先介绍搜索的基本概念和问题的表示，然后着重介绍搜索策略。

◆例如，8皇后问题和8数码问题的解决方案不止一个，究竟哪种方案才能在**满足**问题所规定的**约束条件下达到目标状态**呢?这就是**搜索问题**。

◆你经过反复努力和试探，终于找到了一种解决办法。在高兴之余，你可能马上又会想到这个方案所用的**步骤是否最少**?也就是说它是**最优的方案**吗?

◆如果不是，如何才能找到**最优方案**?

◆这显然是一个最优化问题，但却无法用以前的优化方法求解。以前求解的问题通常都是先用代数方程或者微分方程等数学表达式得到最优化问题的数学模型，然后求解这些代数方程或者微分方程，得到问题的解。这类问题要靠**搜索**发现（最优）解决方案。

◆另一个问题：**在计算机上又如何实现这样的搜索**?

◆这些问题就是本章我们要介绍的**搜索问题**，而求解这类搜索问题的技术称之为**搜索技术**。

# 3.1 图搜索策略

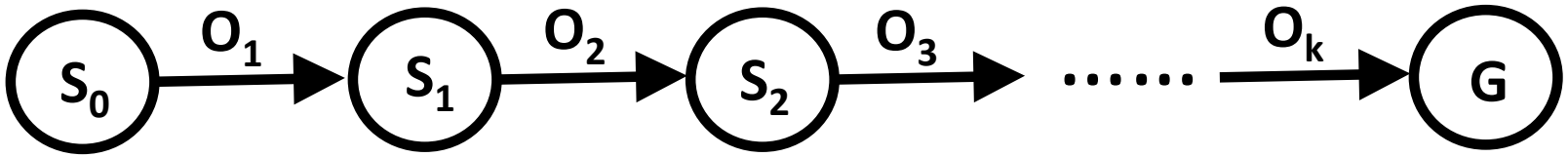
针对搜索问题，可以采用**状态空间知识表示法**来表示问题。

首先，回顾相关术语：

- **状态**（state）就是用来描述在问题求解过程中**某一个时刻进展情况**等**陈述性知识**的一组变量或数组，是某种结构的符号或数据。
- **操作**也称为**运算**，可以是一个动作（如棋子的移动）、过程、规则、数学算子等，使问题由一个具体状态转换到另一个具体状态。 。
- **状态空间**是采用**状态变量**和**操作符号**表示系统或问题的有关知识的**符号体系**。
- 状态空间通常用**有向图**来表示，其中，**结点**表示问题的**状态**，结点之间的**有向边**表示引起状态变换的**操作**，有时边上还赋有**权值**，表示变换所需的**代价**，称为**状态空间图**。

# 3.1 图搜索策略

- 在状态空间图中，求解一个问题就是从初始状态出发，不断运用可使用的操作，在满足约束的条件下达到目标状态。搜索技术又称为“状态图搜索”方法。
- **解（solution）：解是**一个从初始状态到达目标状态的有限的操作序列。
- **搜索（search）：**为达到目标，寻找这样的行动序列的过程被称为**搜索**。
- 搜索算法的**输入是问题**，**输出的**是问题的**解**，以操作序列 $\{O_1, O_2, \ldots O_k\}$ 的形式返回问题的解。
- **路径：**状态空间的一条路径是通过操作连接起来的一个状态序列，其中第一个状态是初始状态，最后一个状态是目标状态。

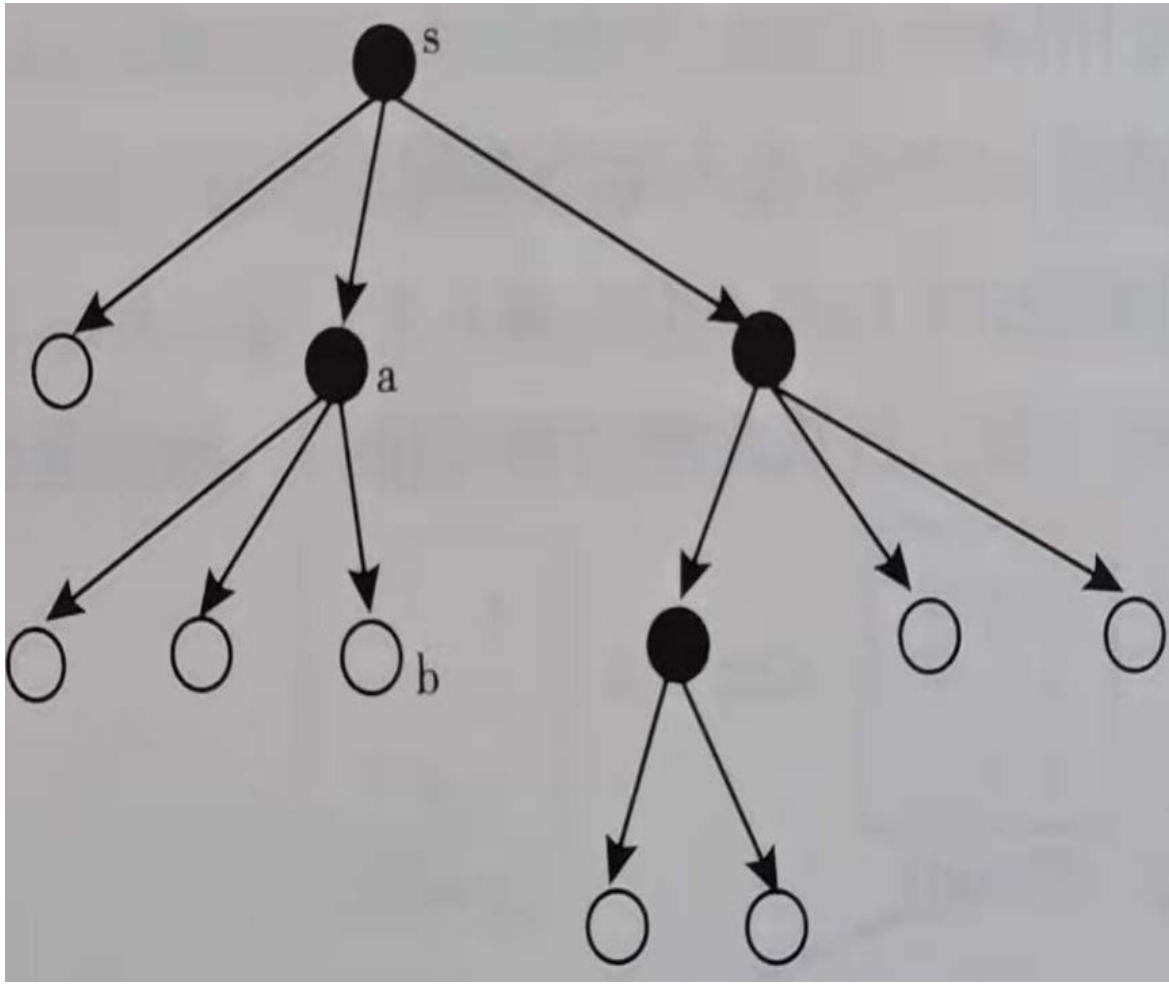


# 3.1 图搜索策略

- **图搜索策略是一种在图中寻找解路径的方法**。
- 为了提高搜索效率，图搜索并不是先生成所有状态的连接图、再进行搜索，而是**边搜索边生成图**，直到找到一个符合条件的解，即路径为止。
- 在搜索的过程中，**生成的无用状态越少**--即非路径上的状态越少，搜索的效率就越高，所对应的**搜索策略就越好**。

# 3.1 图搜索策略

- 图中结点表示状态，**实心圆**表示**已扩展**的结点(即已生成了连接该结点的所有后继结点)，**空心圆**表示还**未被扩展**的结点。
- **图搜索策略**，就是**选择下一个被扩展结点的规则。**
- **即**如何从某实心圆出发，**选择**一个空心圆来作为下一个**被扩展的结点**，以便尽快地找到一条满足条件的路径。



# 图搜索算法

1. $G=G_0$ ($G_0$=s), OPEN:=(s);

// G是生成的搜索图，**OPEN表**用来存储**待扩展的结点**，每次循环从OPEN表中取出一个结点加以扩展，并把新生成的结点加入OPEN表；

2. CLOSED:=( );

// **CLOSED表**用来存储**已扩展的结点**，其用途是检查新生成的结点是否已被扩展过。

3. **LOOP**: IF OPEN=( ) THEN EXIT(FAIL);

4. *n*:=FIRST(OPEN), REMOVE(*n*, OPEN), ADD(*n*, CLOSED);

// 将n从OPEN中删除，加入CLOSED中，即将 n 归入已被扩展的结点

5. IF GOAL(*n*) THEN EXIT(SUCCESS);

6. EXPAND(*n*)→{$m_i$}, G:=ADD ($m_i$, G);

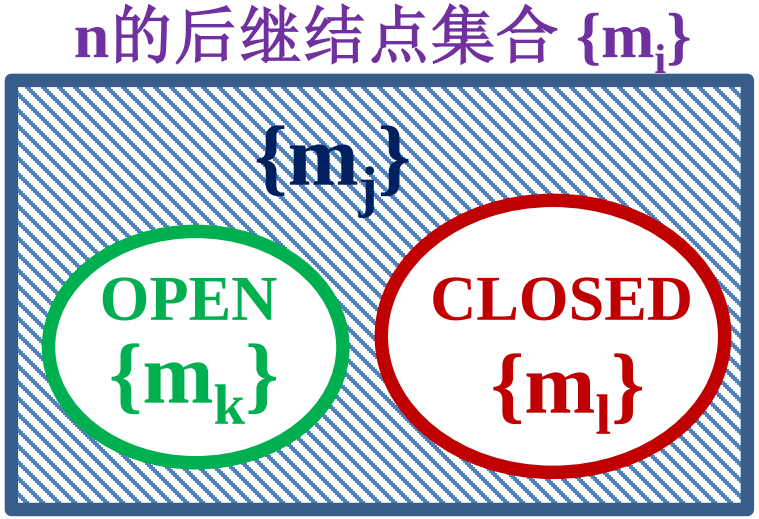
// 扩展结点 n，建立集合{$m_i$}, 使其包含n 的后继结点，而不包含n 的祖先，并将这些后继结点加入G中。

**注：*n* 的后继结点有三类： $\{m_i\} = \{m_j\} \cup \{m_k\} \cup \{m_l\}$,**

**（1）*n*的后继结点$m_j$既不包含于OPEN，也不包含于CLOSED；**

**（2）*n*的后继结点$m_k$包含在OPEN中；**

**（3）*n*的后继结点$m_l$包含在CLOSED中；**



7. For all nodes in $\{m_i\}$;

//对$\{m_i\}$中所有结点，标记和修改指针：

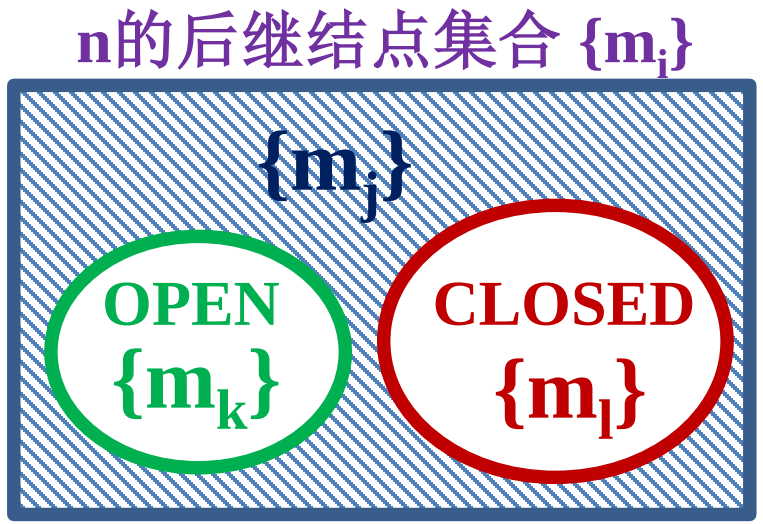
(1) ADD ($m_j$, OPEN），并标记从$m_j$ 到n 的指针；

(2) 判断是否需要修改 **$m_k$ or $m_l$** 的前驱指针（已有前驱），使其指向指向***n***；

(3) 判断是否需要修改 **$m_l$** 后继结点的前驱指针，使其指向**$m_l$**本身；

8. 对OPEN中的结点按**某种原则**重新排序；

9. GO LOOP（语句3）；



# 3.1 图搜索策略

◆针对待扩展结点，不同的选择方法就构成了**不同的图搜索策略**。

使用不同的搜索策略，找到解的**搜索空间范围也不同**。

◆对于大空间问题，搜索策略需要解决**组合爆**炸的问题。

◆**搜索策略的主要任务**是确定选取将被扩展结点的方式，有两种基本方式:

- 若在选择结点时，利用了与问题相关的知识或者启发式信息，则称之为**启发式搜索策略**或**有信息引导**的搜索策略，
- 否则就称之为**盲目搜索策略**或**无信息引导**的搜索策略。

# 3.2 盲目搜索

◆ **盲目搜索**也被称为**无信息搜索**、**通用搜索**。

即该搜索策略不使用**超出问题定义**提供的状态之外的附加信息，只使用问题定义中可用的信息。

◆**常用的两种盲目搜索方法：**

**（1）Depth-first search** 深度优先搜索

**（2）Breadth-first search** 宽度优先搜索

# 问题的求解方法

## ◆完备性：

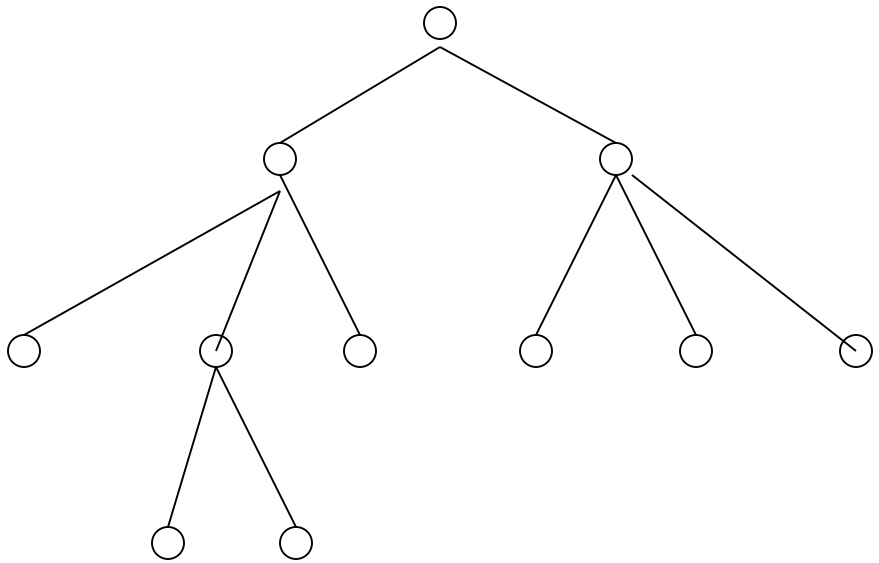
- 当问题有解时，保证能找到一个解。
- 当问题有解，却找不到，就不具有完备性。

## ◆最优性：

- 当问题有最优解时，保证能找到最优解（最小损耗路径）。
- 当问题有最优解，但找不到，找到的只是次优解，则不具有最优性。

# （1）深度优先搜索---DFS

- **深度优先搜索**(Depth-first Search Algorithm) 是从**图搜索算法**变化来的。
- 深度优先搜索的**基本思想**是**优先扩展深度最深的结点**。
- 在一个图中，**初始结点的深度定义为0**，其他结点的深度定义为其父结点的深度加 1。



# （1）深度优先搜索---DFS

- **DFS**是将OPEN表中的结点按搜索树中结点**深度**的**降序**排序，**深度最大的结点排在最前面，深度相同的结点可以任意排列。**
- DFS每次选择一个**深度最深的结点**进行扩展；
- 如果有相同深度的**多个**结点，则按照事先的约定从中选择一个。
- 如果该结点**没有子结点**，即是叶子结点，则选择一个除了该结点以外的深度最深的结点进行扩展。
- 依次进行下去，直到**找到问题的解**，则结束；
- 若**再也没有结点可扩展**，则结束，这种情况下表示没有找到问题的解。

# （1）深度优先搜索---DFS

◆DFS 的实现方法

使用 **LIFO** (Last-In First-Out)的**栈**存储OPEN表，把后继结点放在**栈顶**。

◆ **DFS**是将OPEN表中的结点按搜索树中结点**深度**的**降序**排序，深度最大的结点排在**栈顶**，深度相同的结点可以任意排列。

◆ DFS总是扩展搜索树中当前OPEN表中**最深**的结点（**即栈顶元素**）。

◆ 搜索很快推进到搜索树的最深层，那里的结点没有后继。当那些结点被扩展完之后，就从表OPEN中去掉（**出栈**），然后搜索算法回溯到下一个还有未扩展后继的深度稍浅的结点。

# Depth-first Search Algorithm

1. G:=$G_0$ ($G_0$=s), OPEN:=(s), CLOSED:=( );
2. **LOOP**: IF OPEN=( ) THEN EXIT (FAIL);
3. n:=FIRST(OPEN);
4. IF GOAL(n) THEN EXIT (SUCCESS);
5. REMOVE(n, OPEN), ADD(n, CLOSED);
6. IF DEPTH (n) ≥ Dm GO **LOOP**;

   // Dm 是一个阈值，用于控制深度，以免陷入“深渊”

7. EXPAND(n) →{$m_i$}, G:=ADD($m_i$, G);

   // 扩展结点n，建立集合 **{$m_i$}**, 使其包含n的**后继结点**，而不包含n 的祖先，并将这些后继结点加入G中。

# Depth-first Search Algorithm

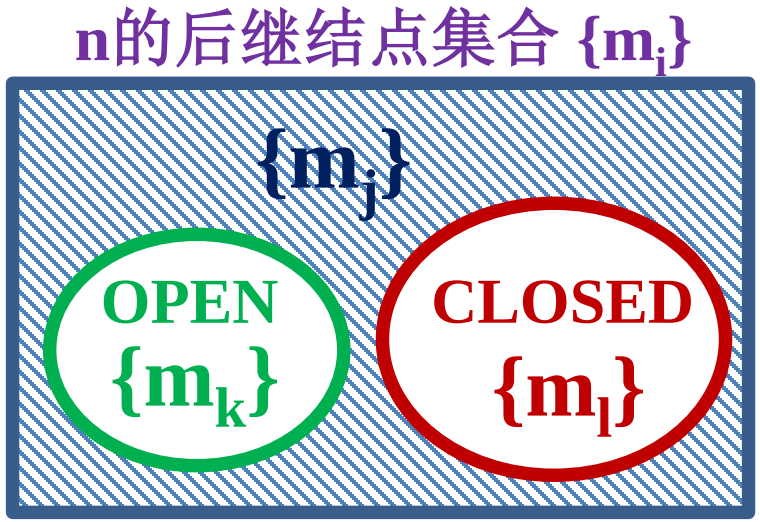
8. IF Goal state is in{$m_i$}, THEN EXIT(SUCCESS);

   //IF 目标在{$m_i$}中，则成功退出;

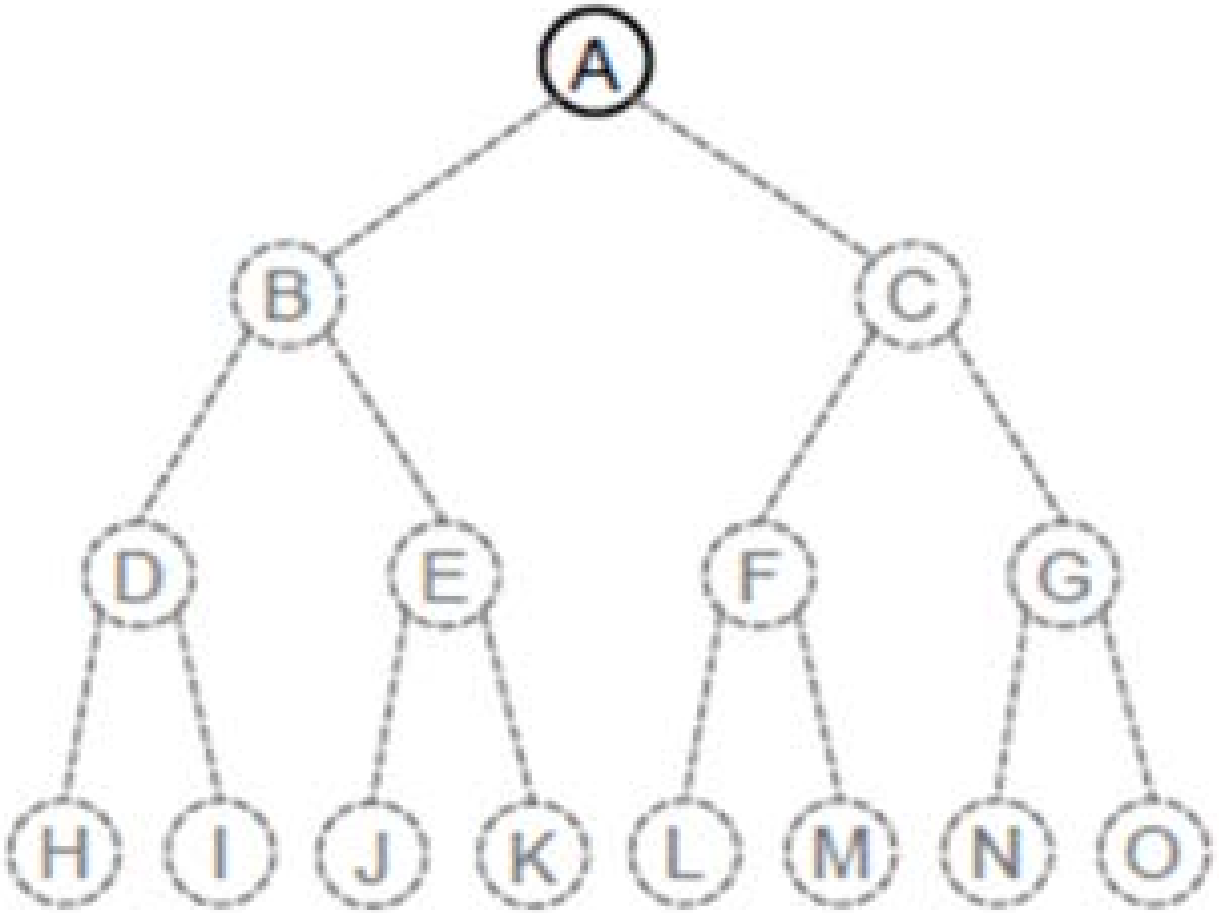
9. **ADD ($m_j$ ,OPEN),** and mark the pointer from $m_j$ to n;

   // **$m_j$** 表示不在CLOSED和OPEN中的结点，ADD 用于把**$m_j$** 放进OPEN表顶端，使深度最大的结点可优先扩展。

10. GO LOOP;

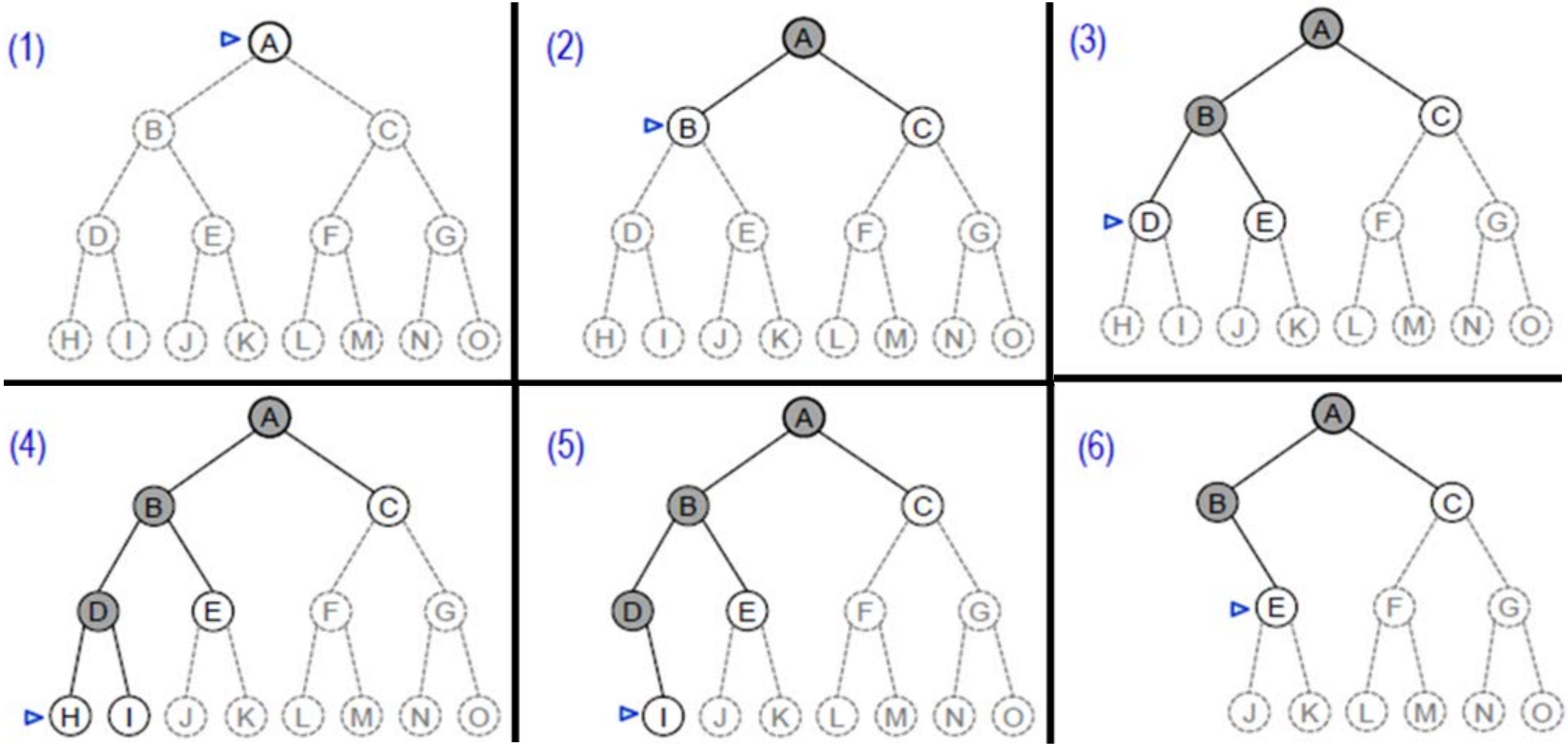


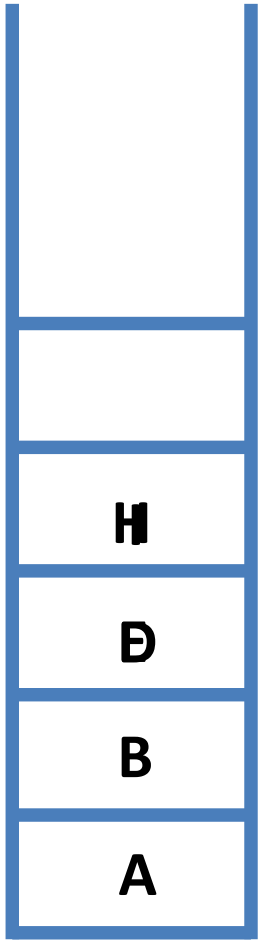
# Depth-first Search on a Simple Binary Tree




DFS 访问的顺序：即扩展顺序，为 {H, I, D, J, K, E, B, L, M, F, N, O,G, C, A}.

# Depth-first Search on a Simple Binary Tree

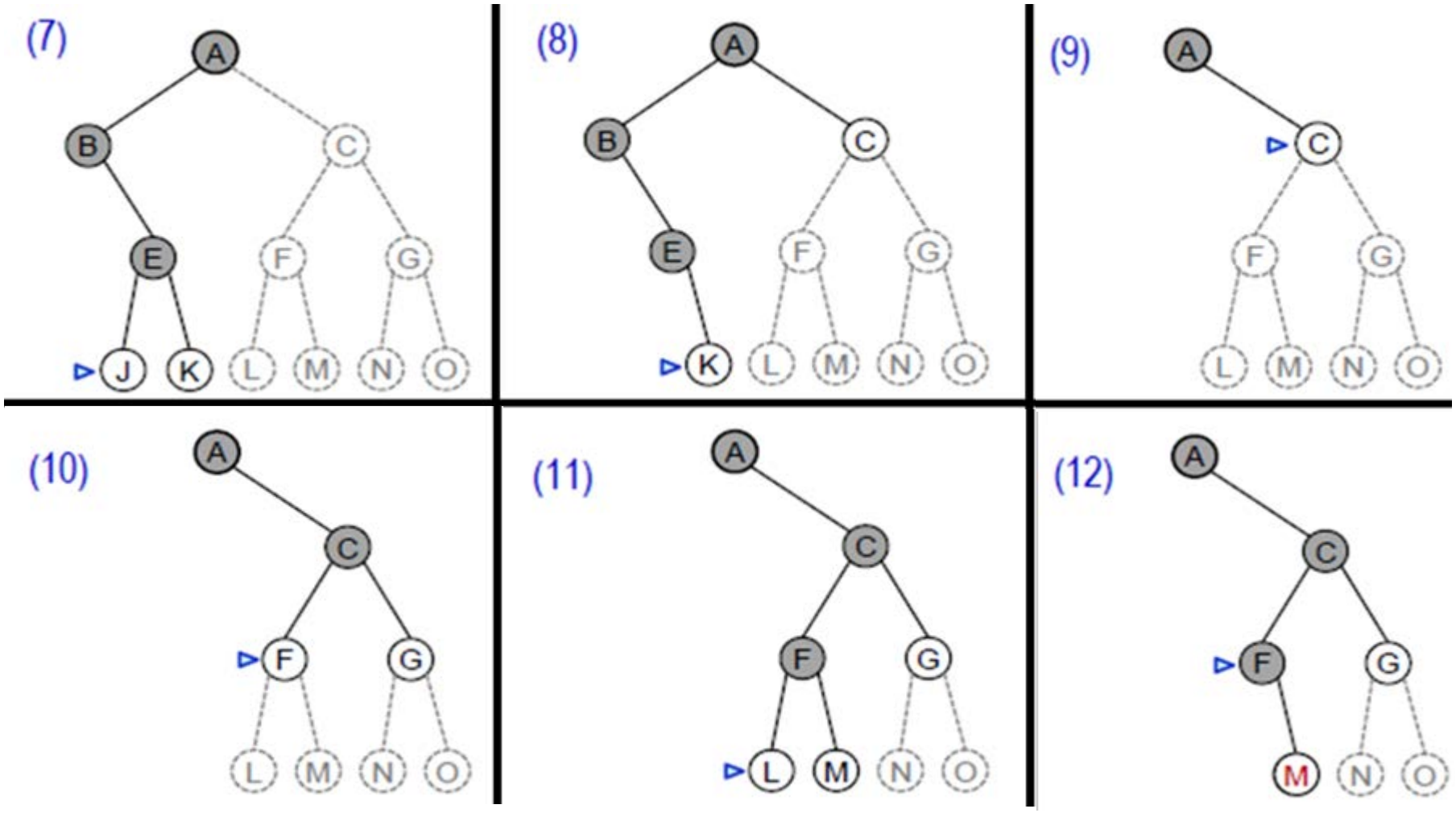


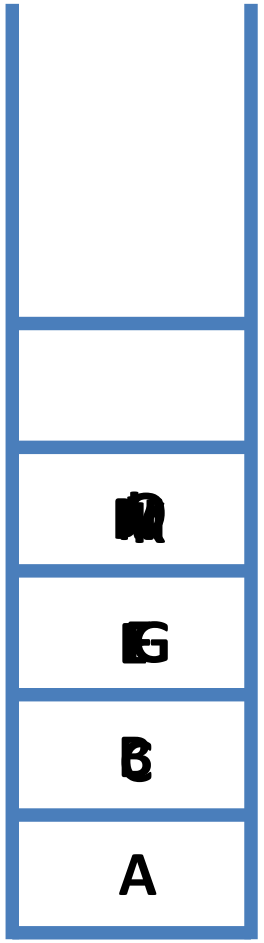




OPEN

CLOSED

# Depth-first Search on a Simple Binary Tree




CLOSED H I D

扩展顺序(即出栈顺序) 为 {H, I, D, J, K, E, B, L, M, F, N, O,G, C, A}.

# 深度优先搜索的性质

◆一般不能保证找到解，更不能保证找到最优解，　即DFS是**不完备的、也不是最优的**。

◆对于很多问题，DFS会可能沿着一个“错误”的路线搜索下去而陷入“深渊”。为避免此情况，在DFS中往往会加上一个**深度限制**，即若一个节点的深度达到了深度限制，则强制回溯到浅一点的节点，进行扩展。

◆当深度限制**过深**时，会陷入“**深渊**”，**求解效率低**；

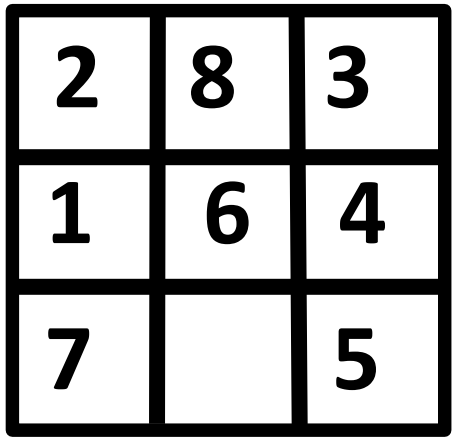
若深度限制**过浅**，可能找不到解，即**不完备**。

◆**最坏情况时，搜索空间等同于穷举**。

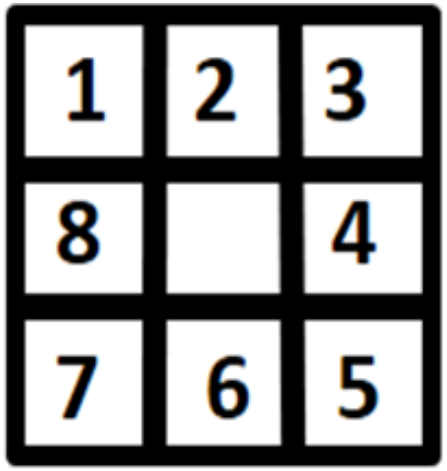
◆DFS是一个通用的、与问题无关的方法。

# 例2.1 八数码问题

假设八数码问题的初始状态和目标状态如下图所示：

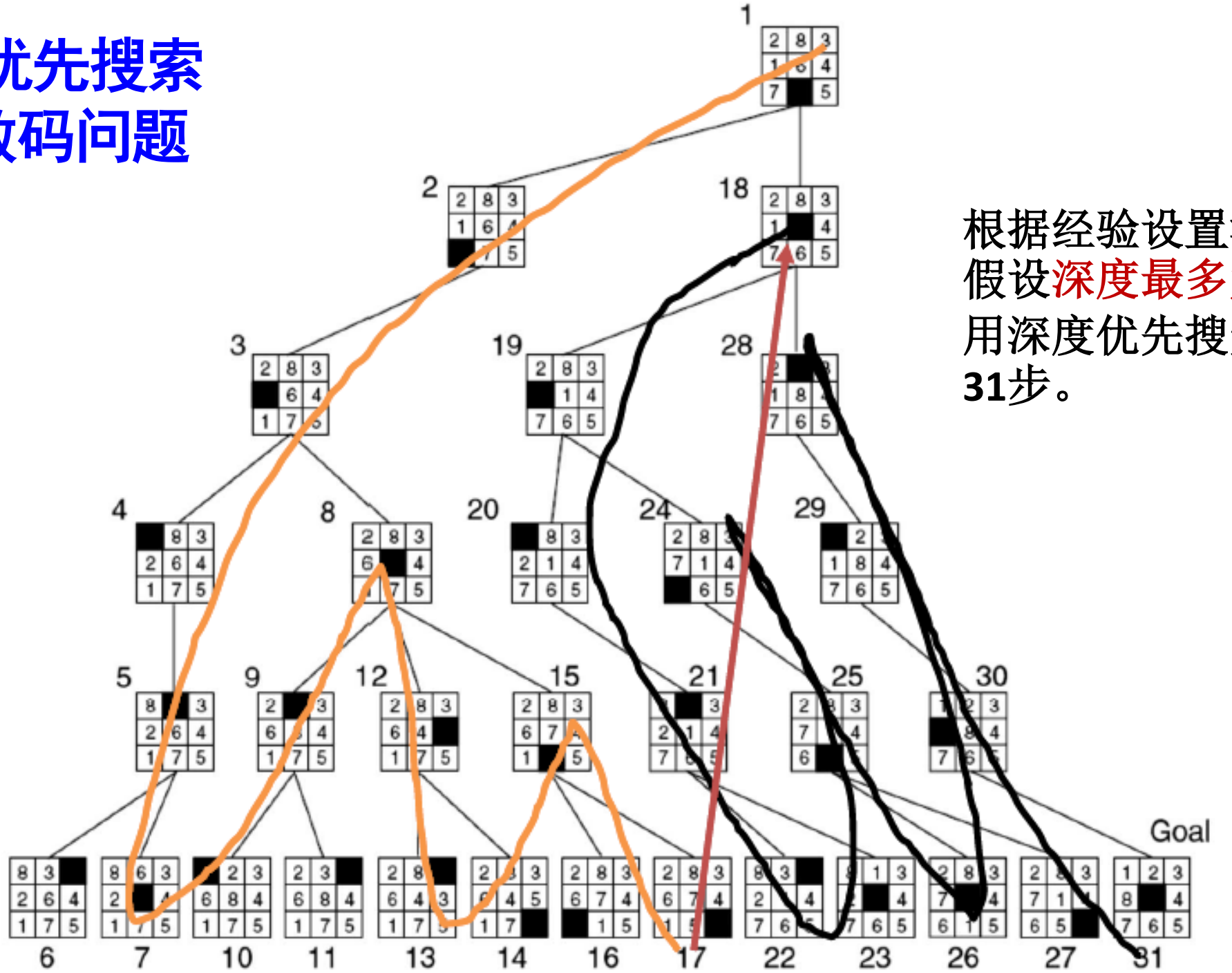



（a）初始状态




（b）目标状态

# 用深度优先搜索解决8数码问题




根据经验设置深度限制，假设深度最多为5，则采用深度优先搜索需要了31步。

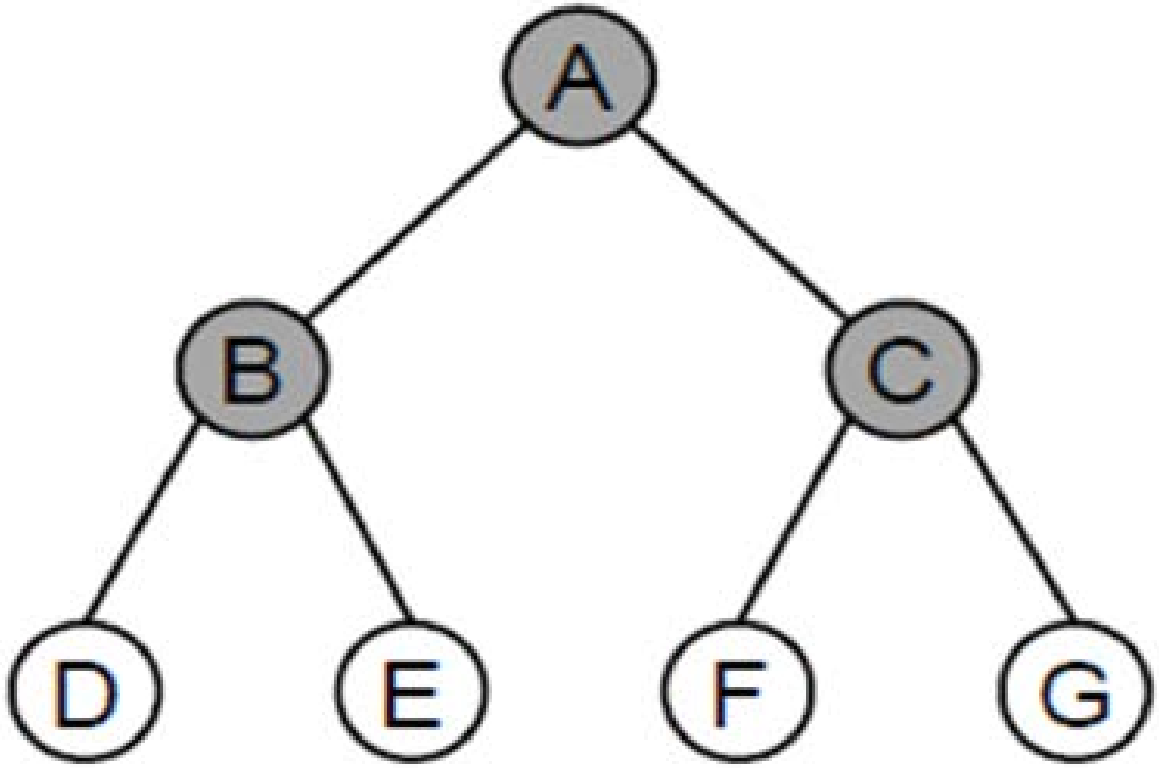
# （2）宽度优先搜索---BFS

◆**宽度优先搜索**（Breadth-first Search, BFS）也是从**图搜索算法**变化来的.

◆宽度优先的**搜索策略**:

- 先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推。
- BFS每次总是**优先**扩展**深度最浅**的结点。
- 如果有多个结点深度是相同的，则按照事先约定的规则从深度最浅的几个结点中选择一个。
- 一般地，在下一层的任何结点扩展之前，搜索树上本层深度的所有结点都应该已经扩展过。

# Breadth-first Search on a Simple Binary Tree
## 简单二叉树的宽度优先搜索




**BFS 访问的顺序：** 即扩展顺序，为 **{A, B, C, D, E, F, G}.**

# （2）宽度优先搜索---BFS

◆实现方法

- 使用**FIFO** (First-In First-Out)**队列**存储OPEN表。
- **BFS**是将OPEN表中的结点按搜索树中结点**深度的增序**排序，**深度最浅**的结点排在最前面（**队头**），深度相同的结点可以任意排列。
- 新结点（结点比其父结点深）总是加入到**队尾**，这意味着浅层的老结点会在深层的新结点之前被扩展。

# Breadth-First Search Algorithm

1. G:=G0 (G0=s), OPEN:=(s), CLOSED:=( );

// G是生成的搜索图，**OPEN表（队列，FIFO）**用来存储**待扩展的结点，**
CLOSED用于存放已被扩展的结点。

2. **LOOP**: IF OPEN=( ) THEN EXIT (FAIL);
3. *n*:=FIRST(OPEN); //从OPEN表中取队头元素
4. IF GOAL(*n*) THEN EXIT (SUCCESS);
5. REMOVE(*n*, OPEN), ADD(*n*, CLOSED);
6. EXPAND(*n*) $\rightarrow\{m_i\}$, G:=ADD($m_i$, G);

// 扩展结点***n***，建立集合**$\{m_i\}$**，使其包含***n***的后继结点，而不包含***n***的祖先，并将这些后继结点加入G中。

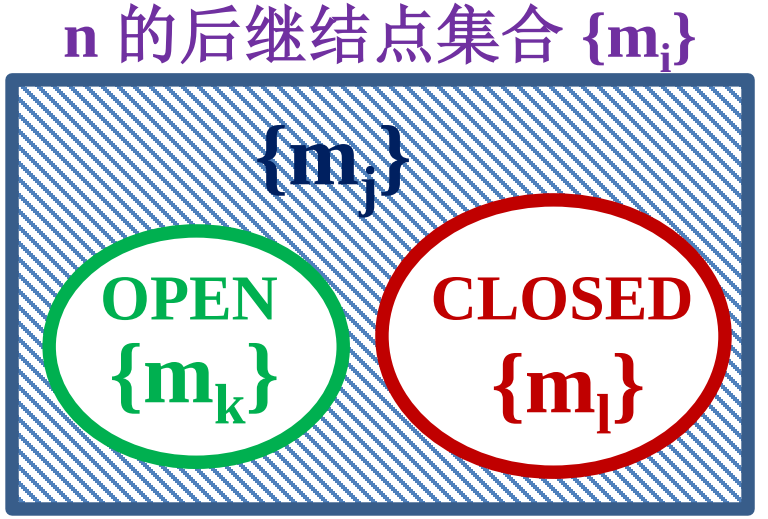
# Breadth-First Search Algorithm

7. IF 目标状态已在{$m_i$}中, THEN EXIT(SUCCESS);

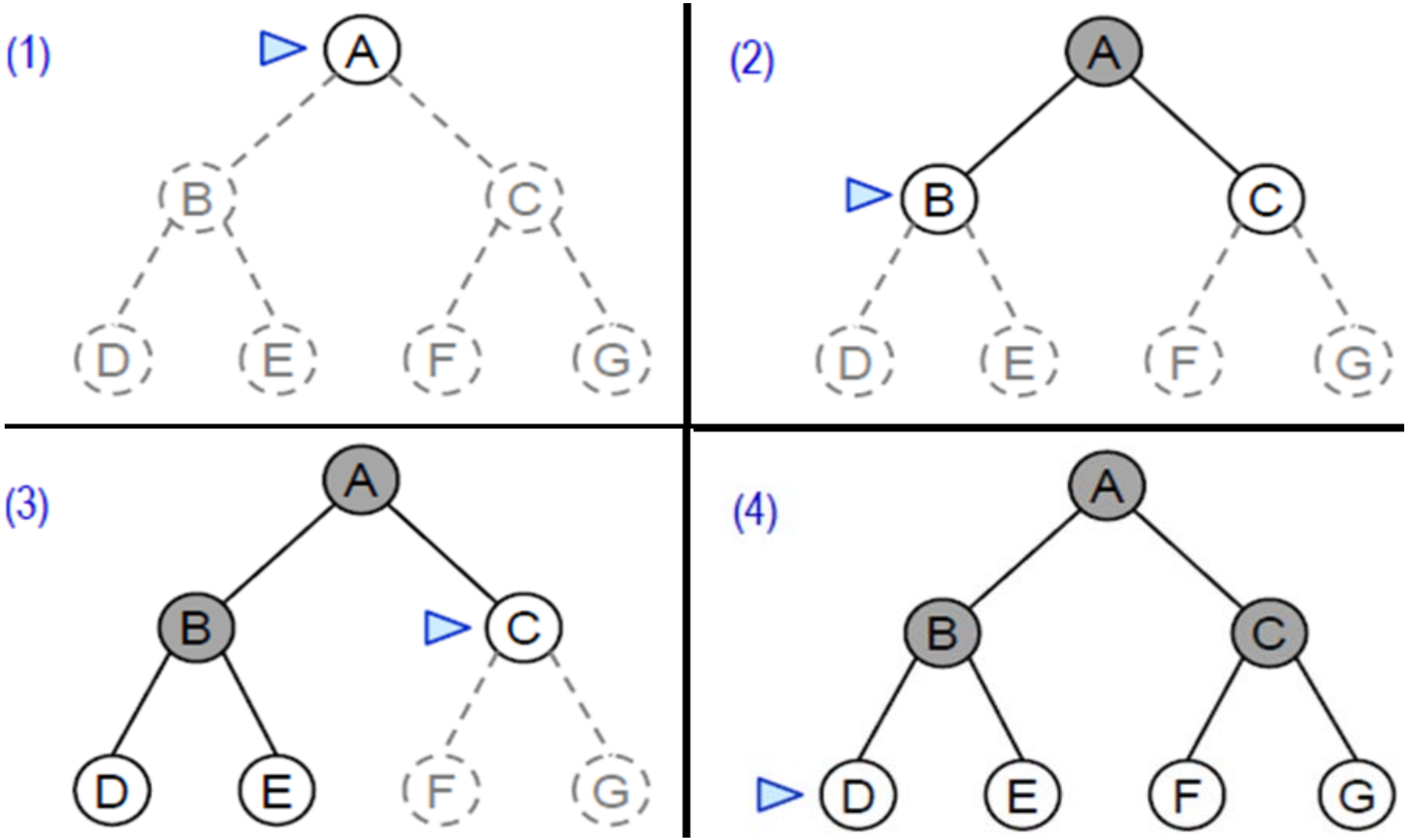
8. **ADD($m_j$ ,OPEN),** and mark the pointer from $m_j$ to *n*;

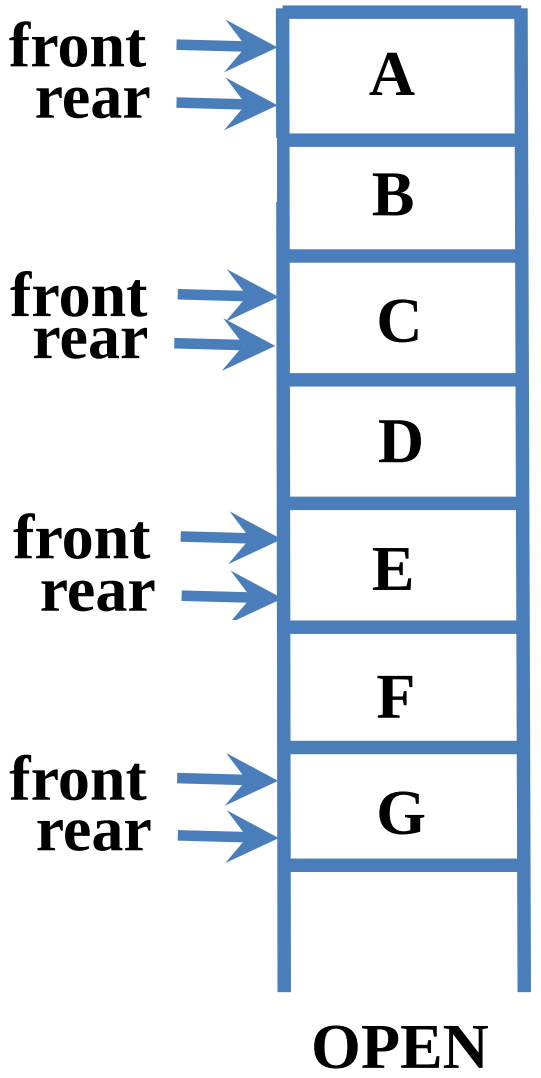
// **$m_j$**表示不在CLOSED和OPEN中的结点，ADD用于把**$m_j$**放进OPEN表**队尾**，使深度最浅的结点可优先扩展。

9. GO LOOP;



# Breadth-first Search on a Simple Binary Tree



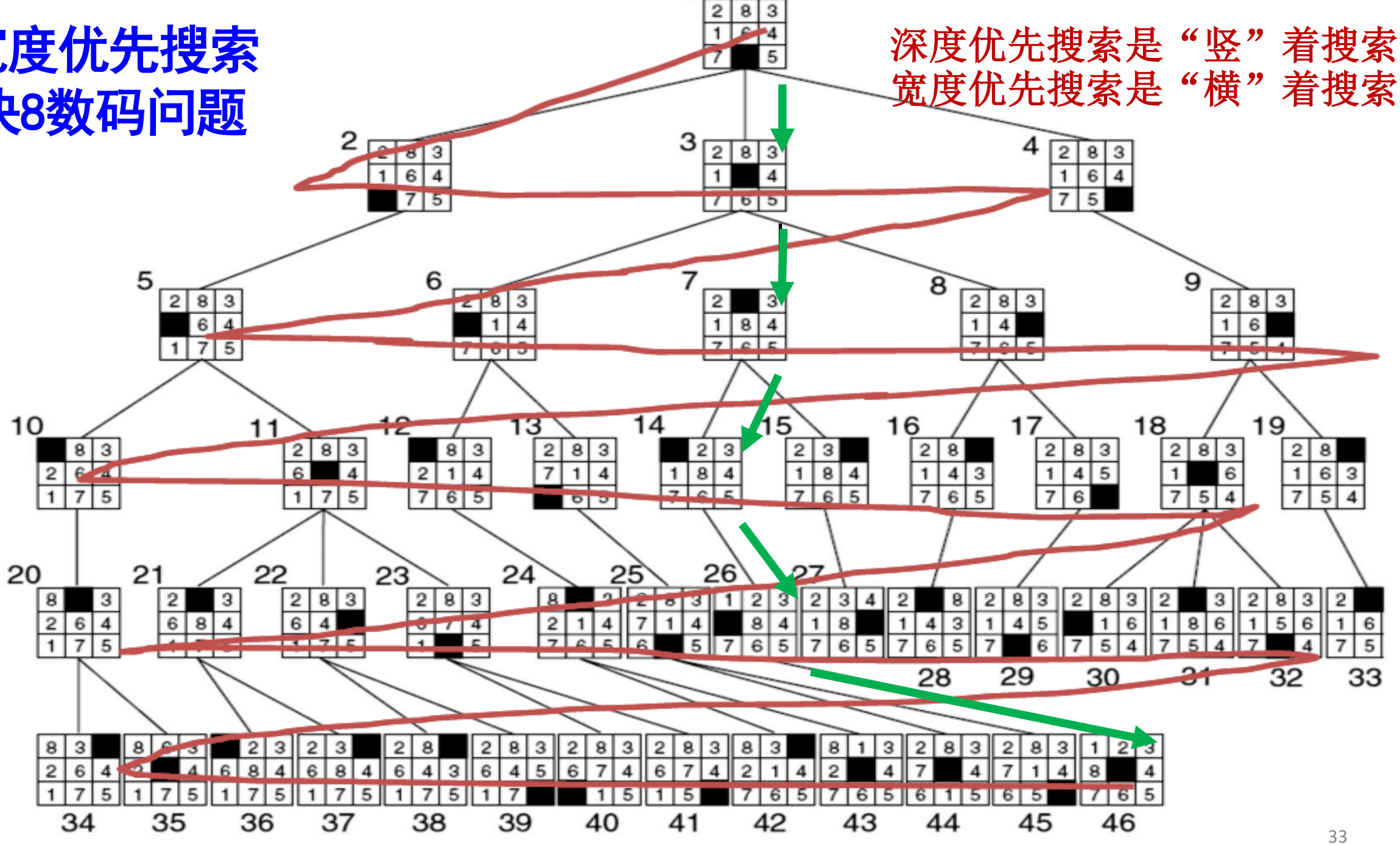




CLOSED

搜索顺序为 {A, B, C, D, E, F, G}.

# 用宽度优先搜索解决8数码问题

深度优先搜索是“竖”着搜索，
宽度优先搜索是“横”着搜索。



# 宽度优先搜索的性质

◆当问题有解时，一定能找到解. 即**BFS是完备的**（**complete**）。

◆当问题为单位代价，且问题有解时，一定能找到最优解，即**BFS具有最优性**。

◆ BFS是一个通用的、与问题无关的方法.

◆**缺点**：求解问题的**效率较低**。

# BFS 与 DFS 的比较

- DFS总是首先扩展**最深**的未扩展结点；BFS总是首先扩展**最浅**的未扩展结点。
- BFS 效率低，但却一定能求得问题的最优解，即**BFS是完备的，且是最优的**；
- **DFS**不能保证找到最优解，即**DFS既不完备，也不最优；**
- 与BFS相比，**DFS优势**在于：空间复杂度低，因为只存储一条从根到叶子的路径。
- 在**不要求求解速度**且目标结点的层次**较深**的情况下，**BFS优于DFS**，因为BFS一定能够求得问题的解，而DFS在一个扩展得很深但又没有解的分支上进行搜索，是一种无效搜索，降低了求解的效率，有时甚至不一定能找到问题的解；
- 在**要求求解速度**且目标结点的层次**较浅**的情况下，**DFS优于BFS**。因为DFS可快速深入较浅的分支，找到解。

# 盲目搜索的特点

◆盲目搜索策略采用“固定”的搜索模式，不针对具体问题。

◆优点是：适用性强，几乎所有问题都能通过深度优先或者宽度优先搜索来求得全局最优解。

◆缺点是：**搜索范围比较大，效率比较低**

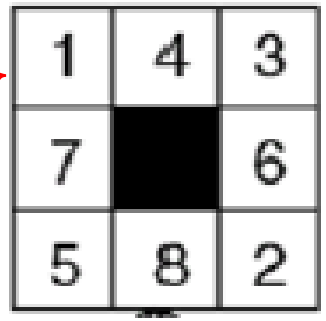
◆在许多不太复杂的情况下，使用盲目搜索策略也能够取得很好的效果。

# 3.3 Informed Search Strategy

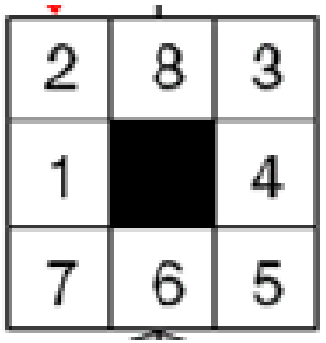
## （启发式搜索 Heuristic Search）

- 盲目搜索策略在搜索过程中，不对状态优劣进行判断，仅按照固定方式搜索。
- 但很多时候我们人类对两个状态的优劣是有判断的。

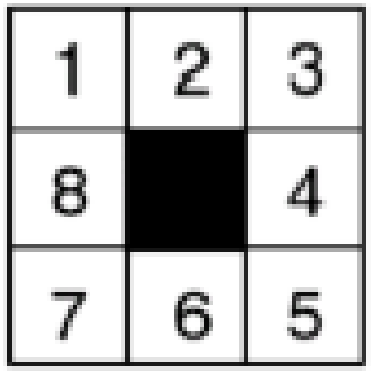
A. 错位6个




VS




B. 错位3个




Goal

**人解决问题的“启发性”：**

对两个可能的状态A和B，选择“从目前状态到最终状态”更好的一个作为搜索方向。

# 如何更聪明地搜索？

◆盲目搜索太“无知”。

◆**启发式搜索，亦被称为有信息搜索.**

- 将人解决问题的“知识”告诉机器，使得搜索算法能够更“聪明”地实现搜索，缩小搜索范围，降低尝试的次数，避免大海捞针。
- 每一步都尽可能选择“最优”动作，以最快的速度找到问题的解。
- **希望**引入启发知识，在保证找到最佳解的情况下，**尽可能减少搜索范围，尽快找到解，提高搜索效率**。

◆ **启发式搜索策略**采用**超出问题本身定义**的、问题特有的知识，因此能够找到比无信息搜索更有效的解。

# 评价函数（evaluation function）

为了尽快找到从初始结点到目标结点的一条代价比较小的路径，我们希望所选择的结点尽可能在最佳路径上。如何评价一个结点在最佳路径上的可能性呢? 我们采用**评价函数**来进行估计:

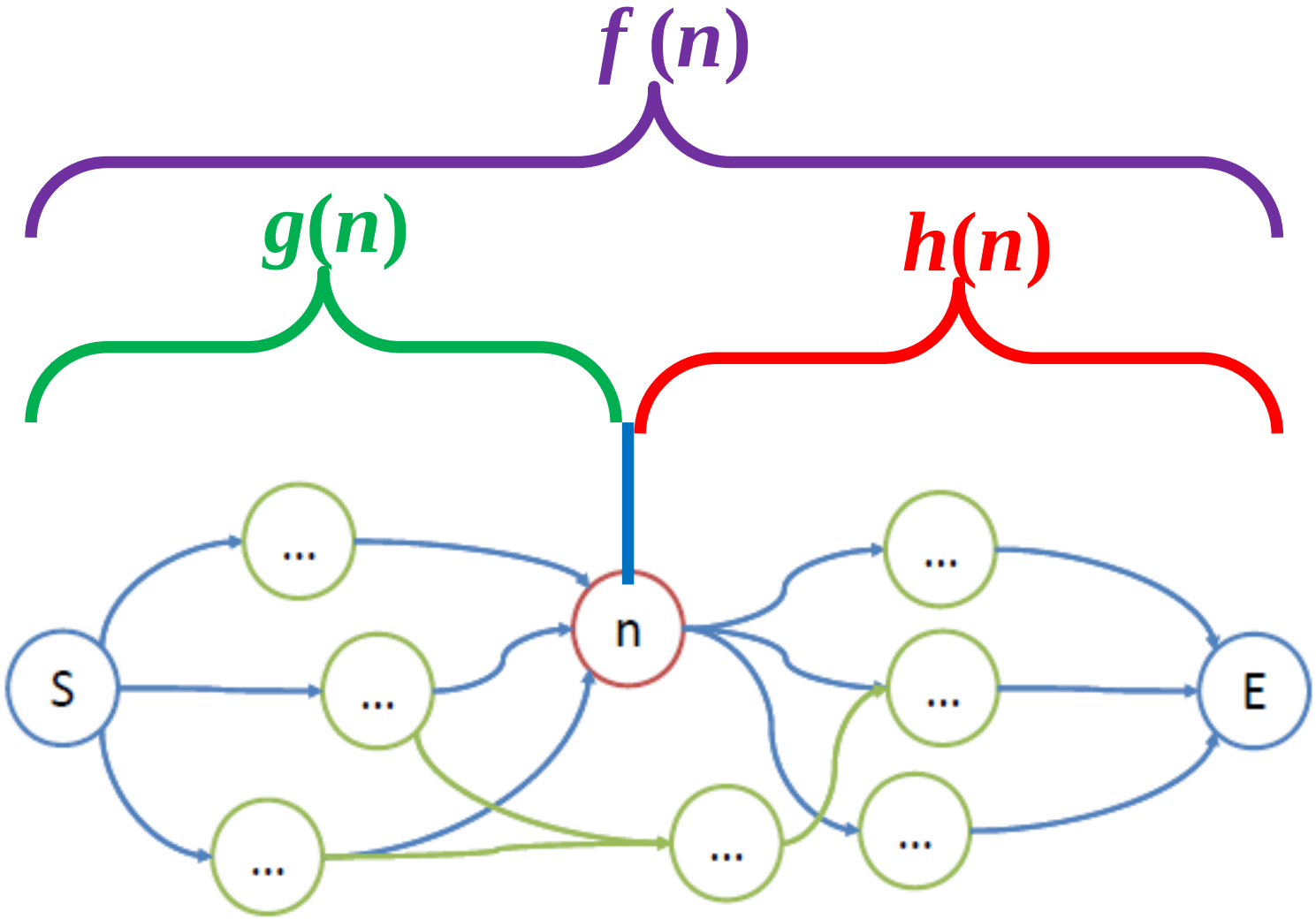
$$f(n) = g(n) + h(n).$$ 其中，$n$为当前结点，即待评价结点。

$f(n)$ 是从初始结点出发、经过结点 $n$ 、到达目标结点的**最佳路径**代价值的**估计值.**

**(1)** $g(n)$ 为从初始结点到结点$n$ 的**最佳路径**代价值的**估计值;**

**(2)** $h(n)$ 为从结点 $n$ 到目标结点的**最佳路径**代价值的估计值，称为**启发式函数**

# 评价函数 $f(n) = g(n) + h(n)$



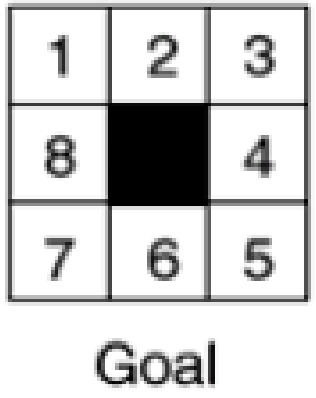

- $g(n)$：为从初始状态到达结点 n 的路径（已消耗）的代价
- $h(n)$：从结点 $n$ 到目标状态的最短路径上的代价的**估计值**
- $f(n)$ 为从初始状态经过结点$n$ 到达目标状态的最短路径上的代价的**估计值**

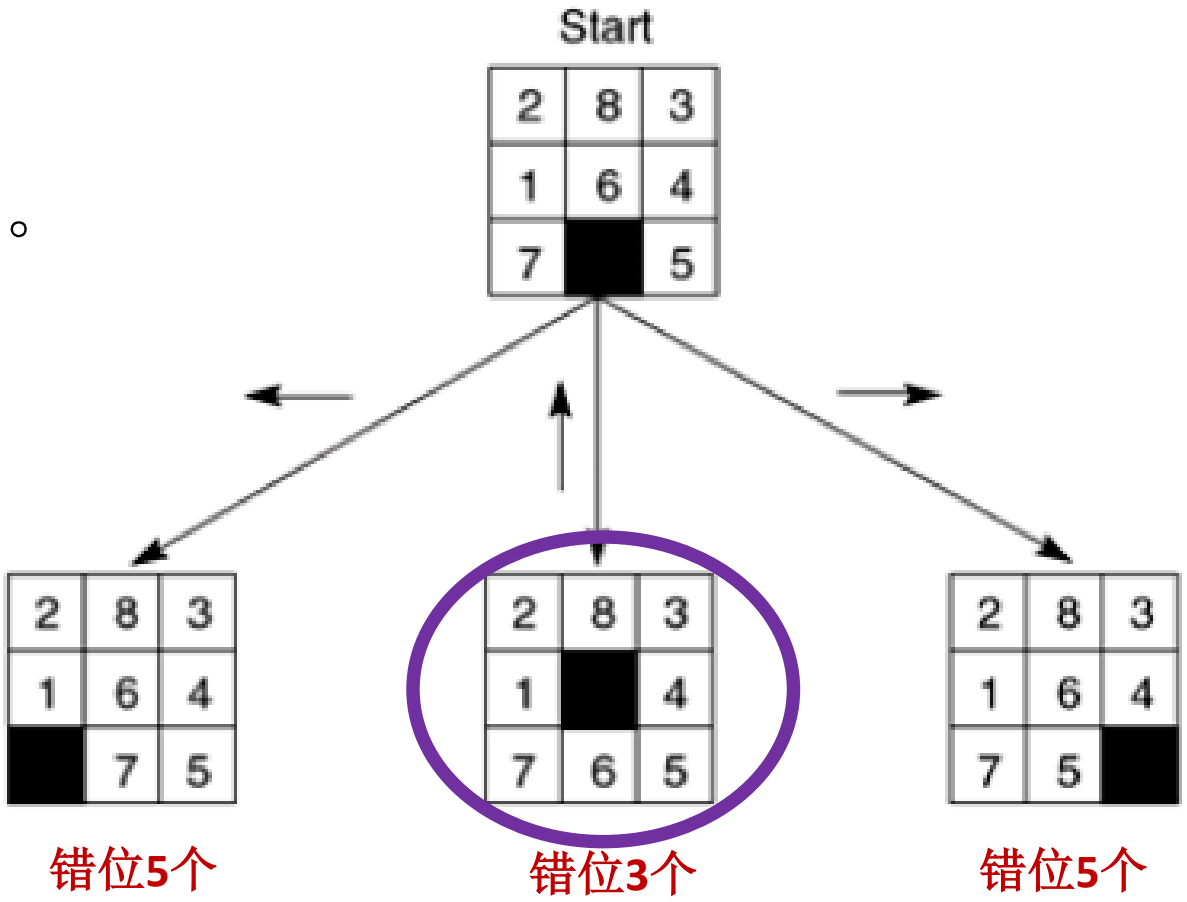
# 如何设计启发函数？ Heuristic function

以8 数码问题为例

◆在当前状态下，共有三种可能的选择。

◆如何评判三种走法的优劣？



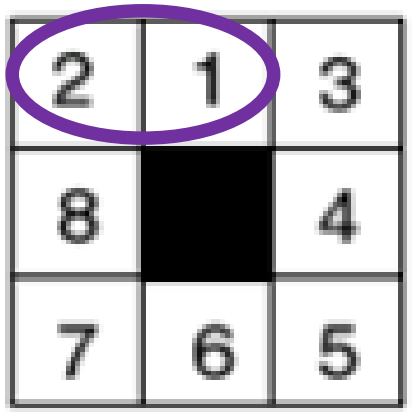
# 如何设计启发函数？ Heuristic function

◆**Method A**：

- 当前棋局与目标棋局之间错位的牌的数量，**错数最少者为最优**。
- **缺点**：这个启发方法**没有考虑到距离因素**，

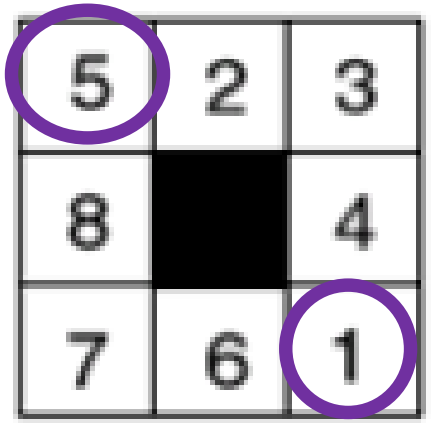
  棋局中“1”“2”颠倒，与“1”“5”颠倒，虽然错数是一样的，

  但是移动难度显然不同。




相对容易移动

VS

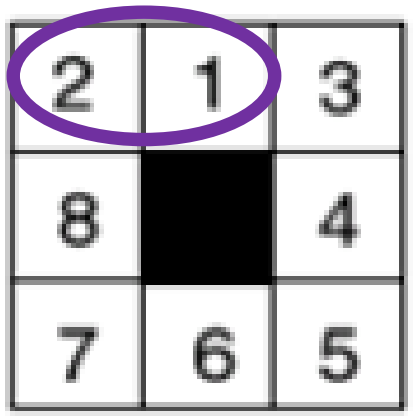


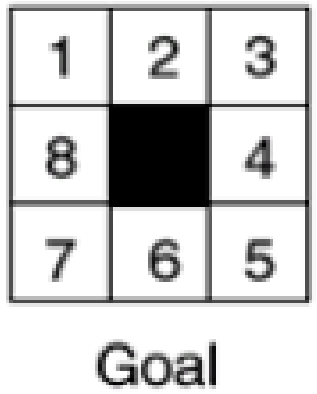

相对难移动

# 如何设计启发函数？ Heuristic function

◆**Method B**：

- 改进：比A更好的启发方法是“错位的牌 距离目标位置的距离和最小”。
- 缺点：仍然存在很大的问题：没有考虑到牌移动的难度。
- 两张牌即使相差一格，如“1”“2”颠倒，将其移动至目标状态依然不容易。




根本无解！

# 如何设计启发函数？ Heuristic function

◆**Method C**：

**改进：**在遇到需要颠倒两张相邻牌的时候，认为其需要的步数为一个固定的数字。

◆**Method D**：

**改进：**将B与C的组合，考虑距离，同时再加上需要颠倒的数量。

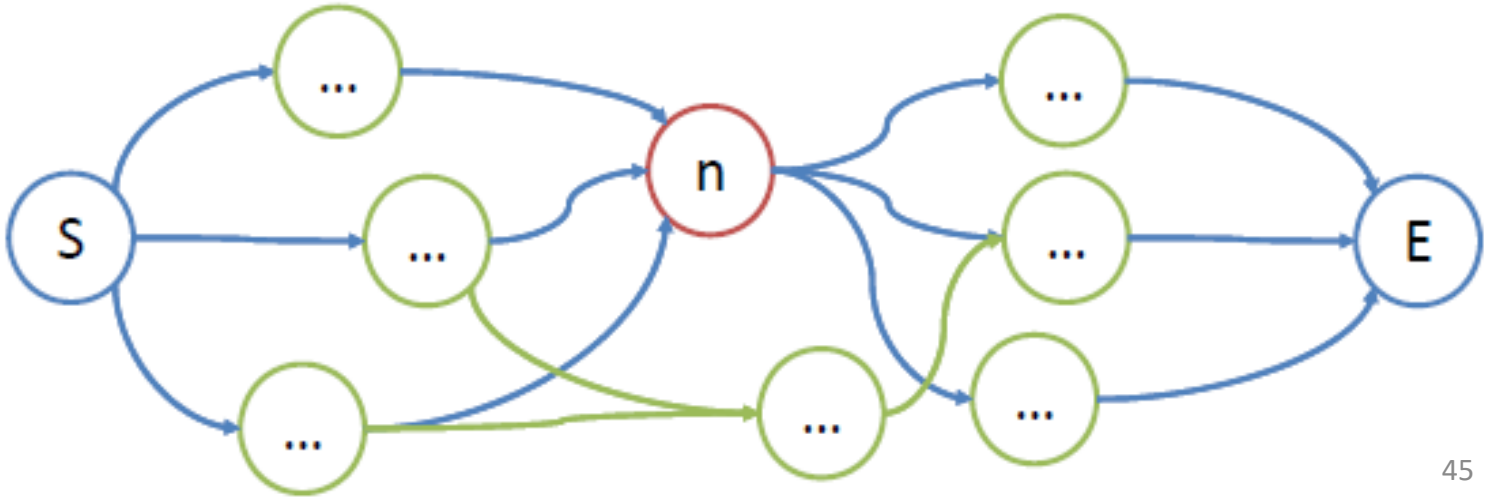
# 如何设计启发函数？ Heuristic function

单纯依靠启发函数搜索是否可行？

◆对于一个具体问题，可以定义**最优路线**：“从**初始结点**出发，以最优路线经过**当前结点**，并以最优路线达到**目标结点**”。

- **盲目搜索**，只考虑了前半部分，能计算出从初始结点走到当前结点的优劣。
- **启发函数**则只考虑了后半部分，只“估计”了当前结点到目标结点的优劣。

◆两者相结合，就是**启发式搜索策略**。



# 常用的启发式搜索算法

## 3.3.1 A Search

（亦称为最佳优先搜索， Best-first Search ）

## 3.3.2 A* Search

（亦称为最佳图搜索算法）

# 3.3.1 A搜索算法

◆**A搜索**又称为**最佳优先搜索**（Best-First Search）。

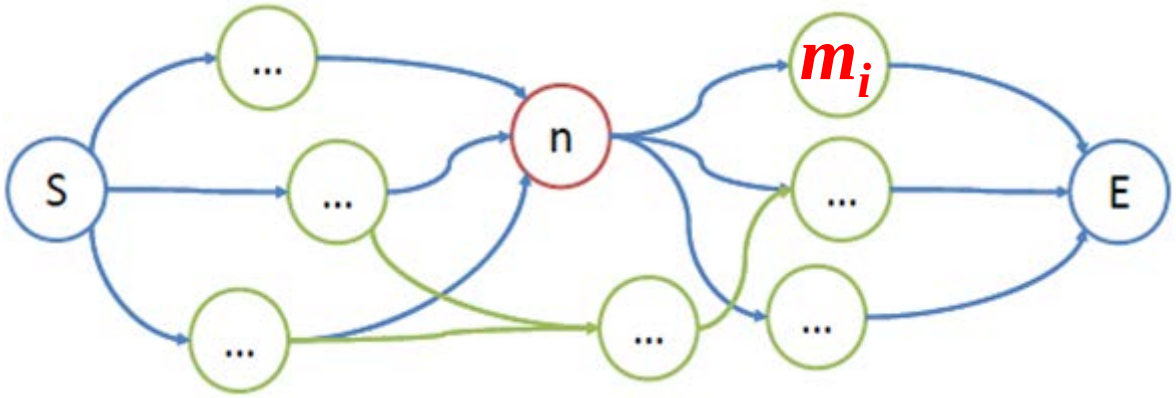
◆**搜索策略：**选择**评价函数 *f*(n) 值最低**的结点作为下一个将要被扩展的结点。

◆实现方法

- **A搜索**采用**队列**存放OPEN表，其中所有结点按照**评价函数值**进行**升序**排列，最佳结点排在最前面，因此称为“**最佳优先搜索**”。

# A search Algorithm

1. OPEN:=(s), f(s):=g(s)+h(s); // *s* 是初始结点/状态.
2. LOOP: IF OPEN=( ) THEN EXIT(FAIL);
3. *n*:=FIRST(OPEN);
4. IF GOAL(*n*) THEN EXIT(SUCCESS);
5. REMOVE(*n*, OPEN), ADD(*n*, CLOSED);
6. EXPAND(*n*) →{$m_i$}, compute ***f* (*n*, $m_i$)** := ***g*(*n*, $m_i$)** + *h*($m_i$);




**//扩展结点*n*，建立集合{$m_i$}，使其包含*n*的所有后继结点$m_i$，并计算每个$m_i$的 *f* 值；**

// $m_i$ 是*n* 的后继结点； ***g*(n, $m_i$)** 是从s 开始，经过 n 到达$m_i$ 的代价，

// h($m_i$) 是从$m_i$ 到目标结点的最短路径的代价值.

// ***f* (*n*, $m_i$)** 从s 开始，经过n和$m_i$ 到达目标结点的代价的估计，是扩展 *n* 后的代价估计；

# A search Algorithm（cont）

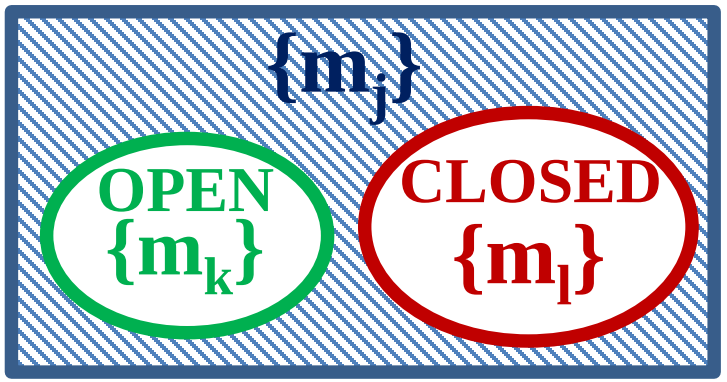
ADD($m_j$, OPEN), and mark the pointer from $m_j$ to n; // 标记$m_j$ 到n的指针；

// 将*n* 的后继中既不在OPEN中又不在CLOSED中的结点**$m_j$**放入OPEN表中，并令**$m_j$** 的指针指向n;

**(1) IF f(*n*, $m_k$)<f($m_k$)** THEN f($m_k$):=f(*n*, $m_k$), and mark the pointer from $m_k$ to n;

// f(**$m_k$**)是扩展 *n* 之前计算的代价，若**条件1**成立，说明扩展 *n* 之后，从**$m_k$**到目标结点的代价比扩展 *n* 之前的代价小，应修改**$m_k$**的代价，并使**$m_k$**的指针指向*n*。

**n 的后继结点集合 {$m_i$}**



# A search Algorithm（cont）

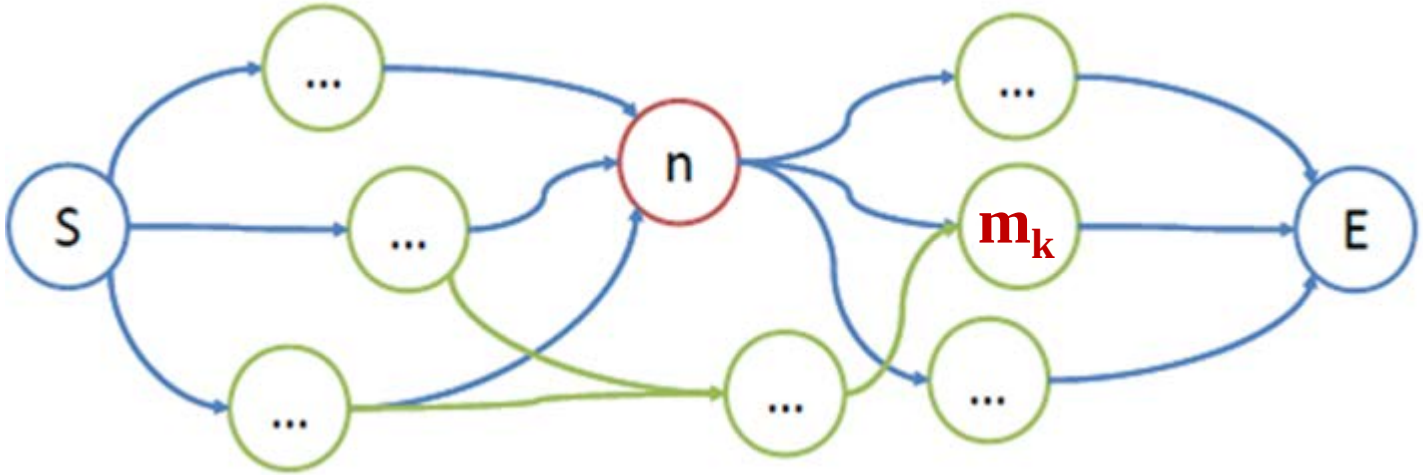
IF $f(n, m_k) < f(m_k)$, 则令 $f(m_k) := f(n, m_k)$ 且令 $m_k$的指针指向 $n$

- $f(n, m_k) = Dist(S \ldots n, m_k \ldots E)$

  是：从s出发，经过 $n$ 和 $m_k$ ,到达 E的最短路径；

- $f(m_k) = Dist(S \ldots m_k \ldots E)$

  是：从s出发，不经过 $n$ 只经过 $m_k$ , 到达 E的最短路径。




$m_k$已在OPEN表中

# A search Algorithm（cont）

**(2) IF $f(n, m_l)<f(m_l,)$** THEN

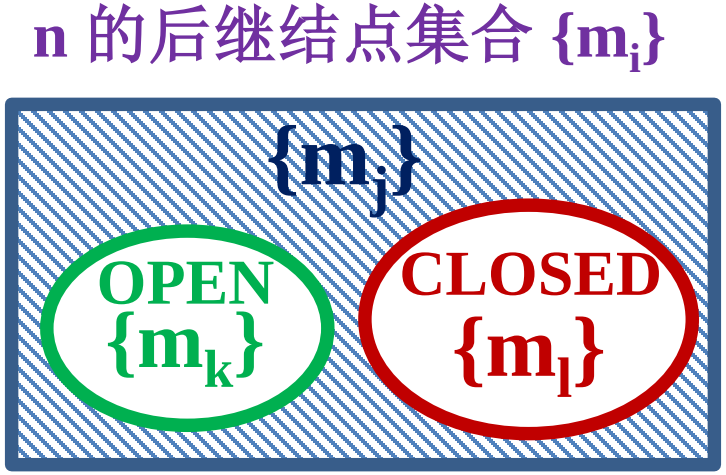
{$f(m_l):=f(n, m_l)$； mark the pointer from $m_l$ to n; ADD($m_l$, OPEN); }

// 若**条件2**成立，则修改**$m_l$** 的代价，且修改**$m_l$** 的指针，使之指向*n*。

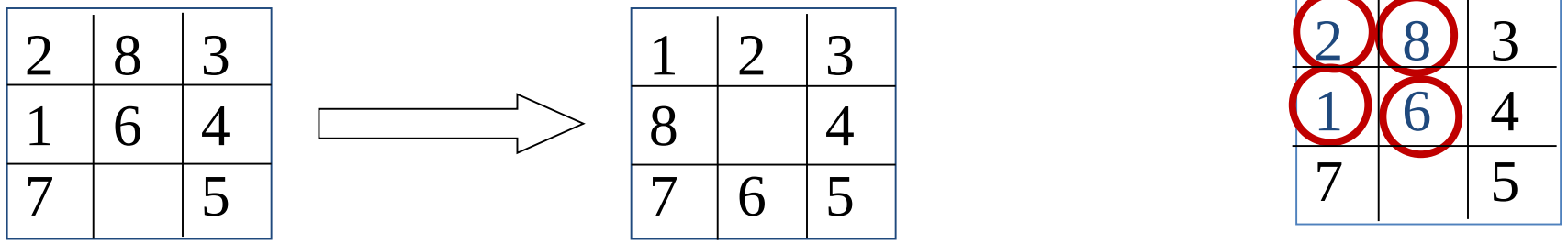
// 把**$m_l$** 重新放回OPEN队列中，不必考虑改**$m_l$** 后继结点的指针。

7. OPEN中的结点按 ***f*** 值从小到大的**升序排序**；

8. GO LOOP；



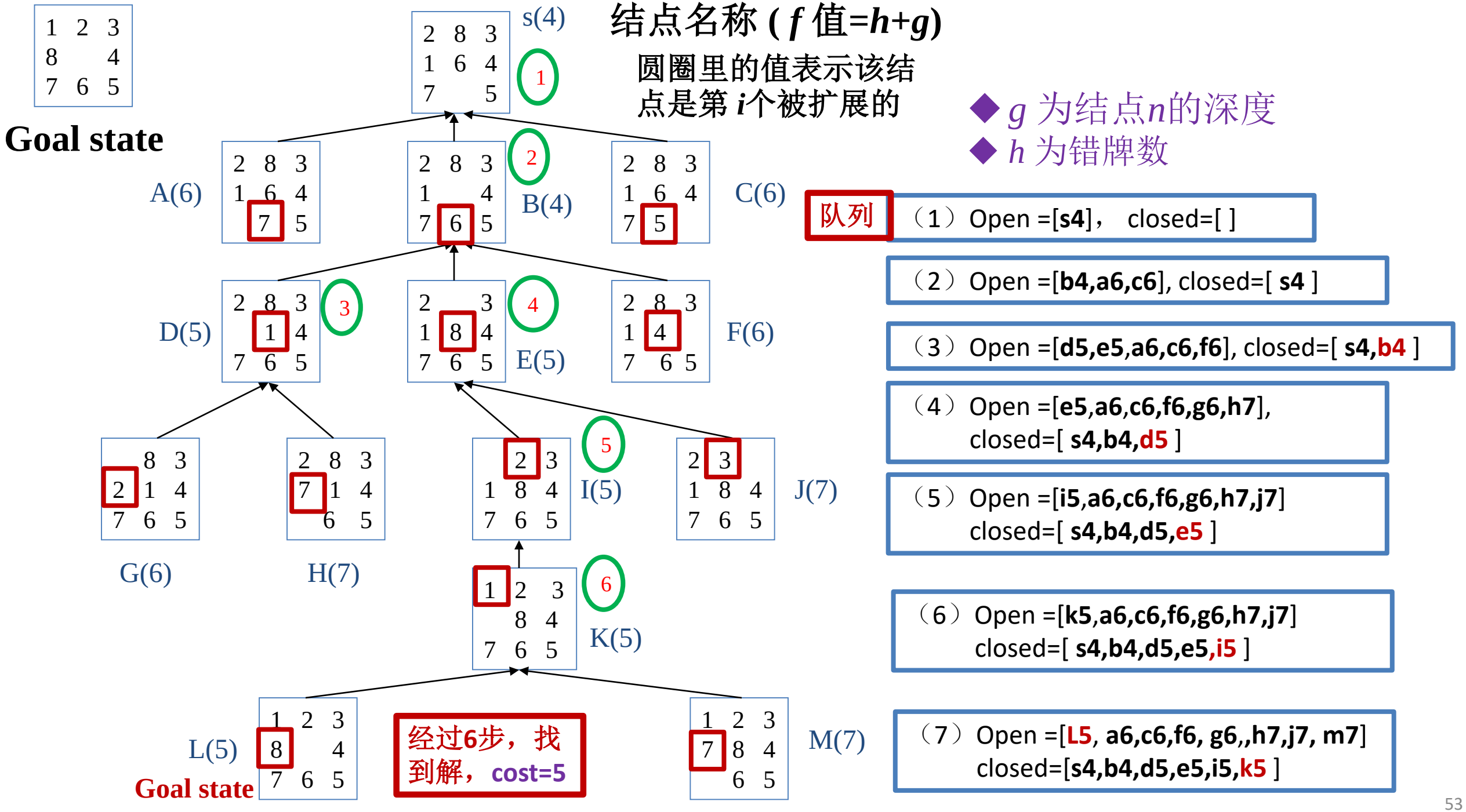
# 用 A 算法解决8数码问题



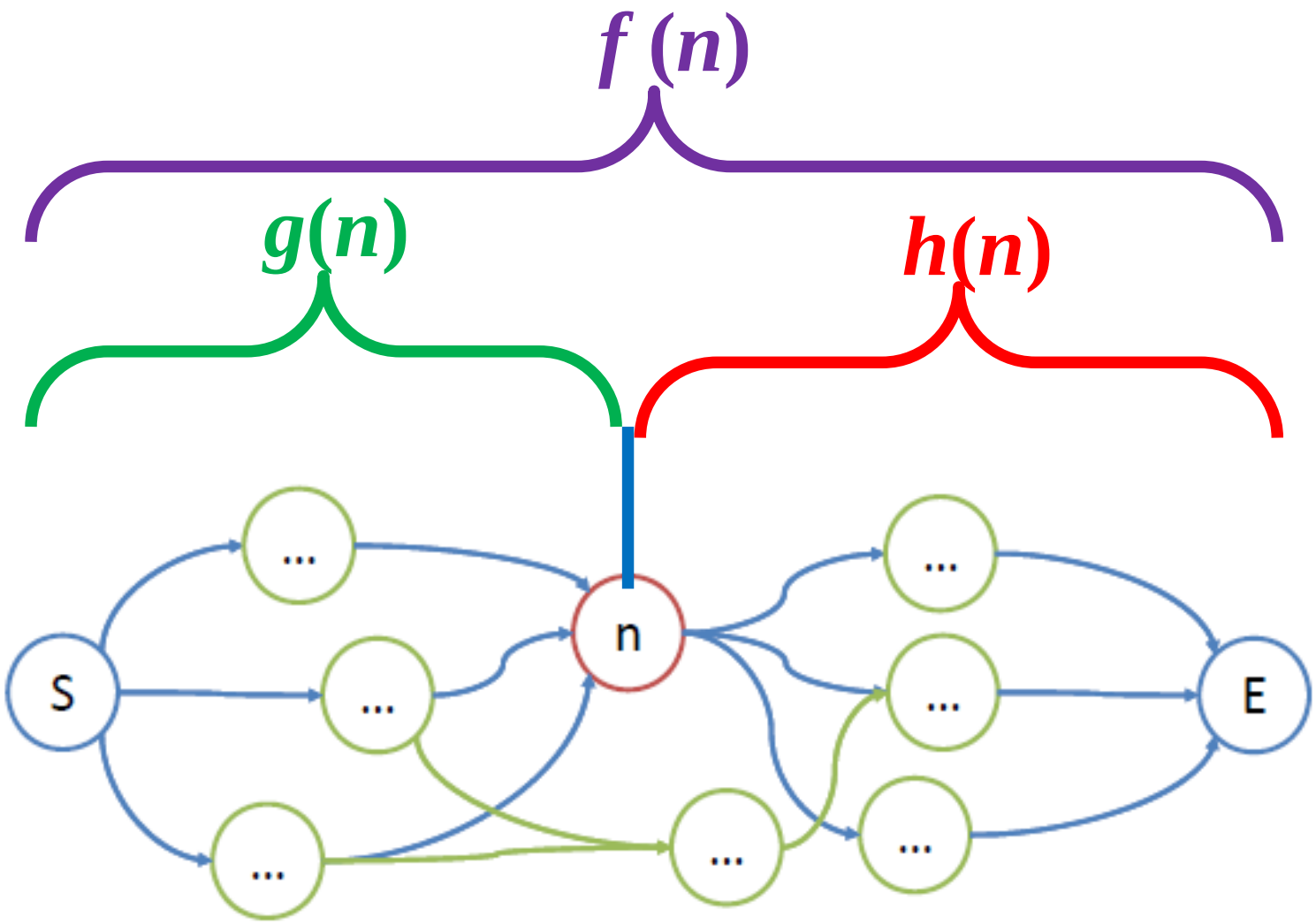

**(a) Initial state *s*** **(b) goal state** h(s) =4, g(s)=0

定义评价函数：$f(n) = g(n) + h(n)$

- g(*n*)为从初始状态到当前状态的代价值，定义为**移动将牌的步数**，即**结点*n*的深度**.
- h(*n*) 是从n 到目标结点的最短路径的代价值，定义为当前状态中“不在位”的将牌数。


Goal state
结点名称 ( f 值=h+g)
圆圈里的值表示该结点是第 i个被扩展的
◆ g 为结点n的深度
◆ h 为错牌数
s(4)
A(6)
B(4)
C(6)
D(5)
E(5)
F(6)
G(6)
H(7)
I(5)
J(7)
K(5)
L(5)
Goal state
M(7)
经过6步，找到解，cost=5
队列
（1）Open =[s4]， closed=[ ]
（2）Open =[b4,a6,c6], closed=[ s4 ]
（3）Open =[d5,e5,a6,c6,f6], closed=[ s4,b4 ]
（4）Open =[e5,a6,c6,f6,g6,h7],
closed=[ s4,b4,d5 ]
（5）Open =[i5,a6,c6,f6,g6,h7,j7]
closed=[ s4,b4,d5,e5 ]
（6）Open =[k5,a6,c6,f6,g6,h7,j7]
closed=[ s4,b4,d5,e5,i5 ]
（7）Open =[L5, a6,c6,f6, g6,,h7,j7, m7]
closed=[s4,b4,d5,e5,i5,k5 ]

# 评价函数 $f(n) = g(n) + h(n)$




- $g(n)$：为从初始状态到达结点 n 的路径（已消耗）的代价
- $h(n)$：从结点 $n$ 到目标状态的最短路径上的代价的**估计值**
- $f(n)$ 为从初始状态经过结点$n$ 到达目标状态的最短路径上的代价的估计值

# 评价函数的情况

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

◆If $f(n) = g(n)$, i.e. $h(n)=0$

当 $h(n)=0$ 时，A搜索退化为盲目搜索；

- If $f(n) = g(n)$, 称为一致代价搜索（Uniform-cost Search，UCS，盲目搜索，如:Dijkstra 算法 ）
- If $f(n) = g(n) = d(n)$, 即为 BFS (盲目搜索).

◆BFS 与 UCS (如:Dijkstra 算法 ) 的区别：

BFS只关心路径上的步数（即结点深度，相当于每个边上的权重均为1），UCS则关心路径上的权重之和。

◆If $f(n) = h(n)$, 即$g(n)=0$，称为贪婪最佳优先搜索（Greedy Best-First Search，GBFS），简称贪婪搜索.

# 贪婪搜索

◆**贪婪搜索**是最佳优先搜索的特例, 即 f(n) = h(n)，相当于 g(*n*)=0

◆评价函数**仅使用启发式函数**对结点进行评价， *h*(*n*)为从 *n* 到目标结点的最小估计代价。

◆**搜索策略**：试图扩展最接近目标的结点。

◆为什么称为“贪婪”？在每一步，它都试图得到能够最接近目标的结点。

◆贪婪搜索策略不考虑整体最优，仅求取**局部最优**。

◆贪婪搜索**不能保证得到最优解**，所以，它**不是最优的**. 但其搜索**速度非常快**。

◆贪婪搜索**是不完备的**。

# 贪婪搜索 与 Dijkstra的区别

- **贪婪搜索：** 属于启发式搜索， $f(n) = h(n)$，不考虑 $g(n)$，相当于$g(n)=0$ .

  在贪婪搜索中，用每个顶点到目标顶点的代价预估值进行升序排序。

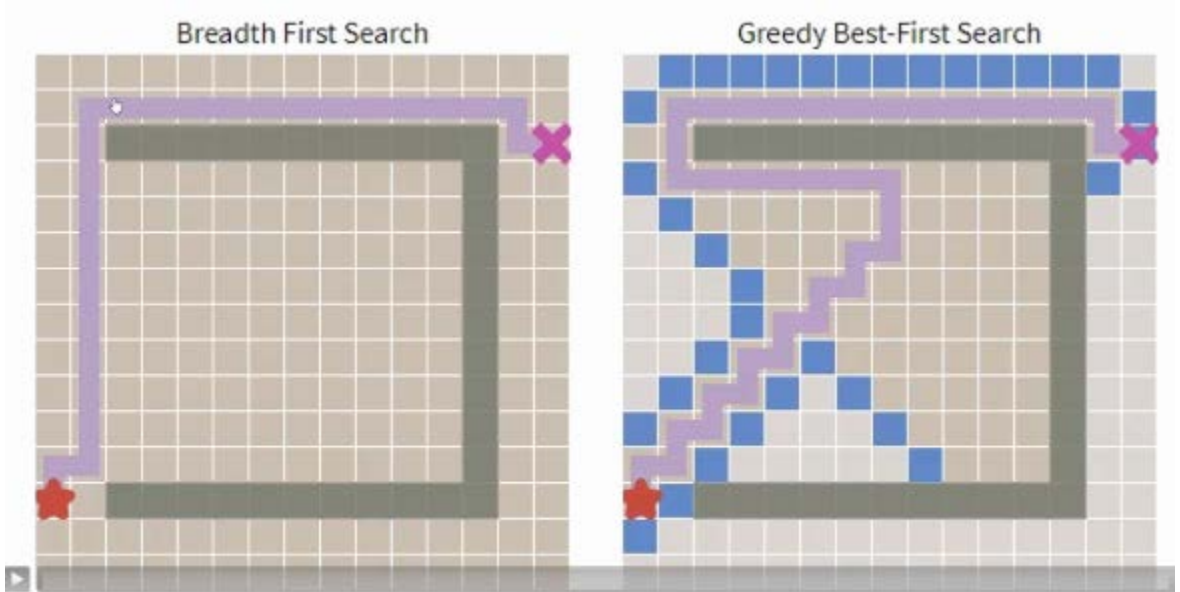
- **Dijkstra：** 属于盲目搜索， $f(n) = g(n)$，不考虑 $h(n)$ ，相当于$h(n)=0$

  在Dijkstra算法中，队列采用每个顶点到起始顶点的代价预估值进行升序排序。

# 对比 BFS 和 贪婪搜索 算法

**贪婪搜索**是最佳优先搜索的特例, 亦称为**贪婪最佳优先搜索**（**GBFS**）

- 在没有障碍物的情况，显然 GBFS 搜索速度更快，且两种算法都能找到最短路径。
- 但如果有障碍，GBFS算法会过分贪心地想尽快接近目标结点，而有可能导致得到的不是最优解，而是**次优解**或**局部最优解**。



# 3.3.2 A* 搜索算法

◆A搜索算法没有对启发函数 $f(n)$做任何限制。

◆实际上，启发函数对于搜索过程十分重要的，如果选择不当，则**有可能找不到问题的解**（**A搜索是不完备**），或者找到的**不是问题的最优解**。

◆如果启发函数h(n)满足如下条件：

$$h(n) \leqslant h^*(n)$$

则可以证明当问题有解时，A算法一定可以找到一个代价值最小的结果，即**最优解**。满足该条件的A算法称作**A*算法**。

◆A*搜索是最佳优先搜索的最广为人知的形式，也称为**最佳图搜索算法**。

# 启发函数的上限问题

**A*算法与A算法没有本质区别，**只是规定了启发函数的上限，

即 $h(n) \leq h^*(n)$。

- 如果令f(n) = g(n) + 0，此时启发函数为0，退化为盲目搜索，必定能找到最优解（BFS），但效率最低。
- 如果添加“一点点”启发，搜索效率提高，并仍然能找到最优解。
- 如果启发函数 h(n) 过大，高于h*(n)，会忽略 g(n)，导致脱离实际情况，反而不能保证总能找到最优解了。

# 3.3.2 A* 搜索算法

- 可证明：若问题有解，则利用A*算法一定能搜索到解，并且一定能搜索到最优解。因此，**A*算法比A算法好**。
- **A搜索既不是完备，也不是最优的。**
- **A *搜索既是完备的，也是最优的。**

# 用 A* 算法解决8数码问题

要用A*算法找到最短距离的解，需要一个**启发式函数**，通常**有两个候选函数。**

- h1(n) = “不在位”的将牌数
- h2(n) = 所有将牌与其目标位置之间的曼哈顿距离之和。

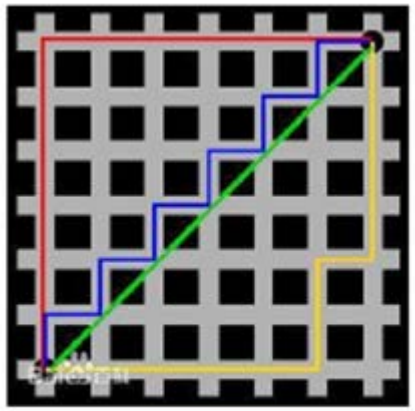
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**goal state**

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

**Initial state *s***

**h1(s)=3**



Manhattan distance

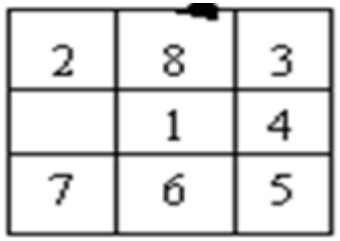
Distance：
Tile1： 1
Tile2： 1
Tile8： 2

**h2(s)=4**

# (a)为目标状态，分别计算图（b）中的 *h1* and *h2*

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

**(a) goal state**



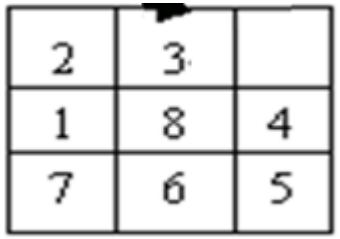

**(b)**

g=1

**h1=3， h2=5**

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

**(a) goal state**




**(b)**

g=2, f=6

**h1=4, h2=4**

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

**(a) goal state**


**(b)**

g=1, f=4

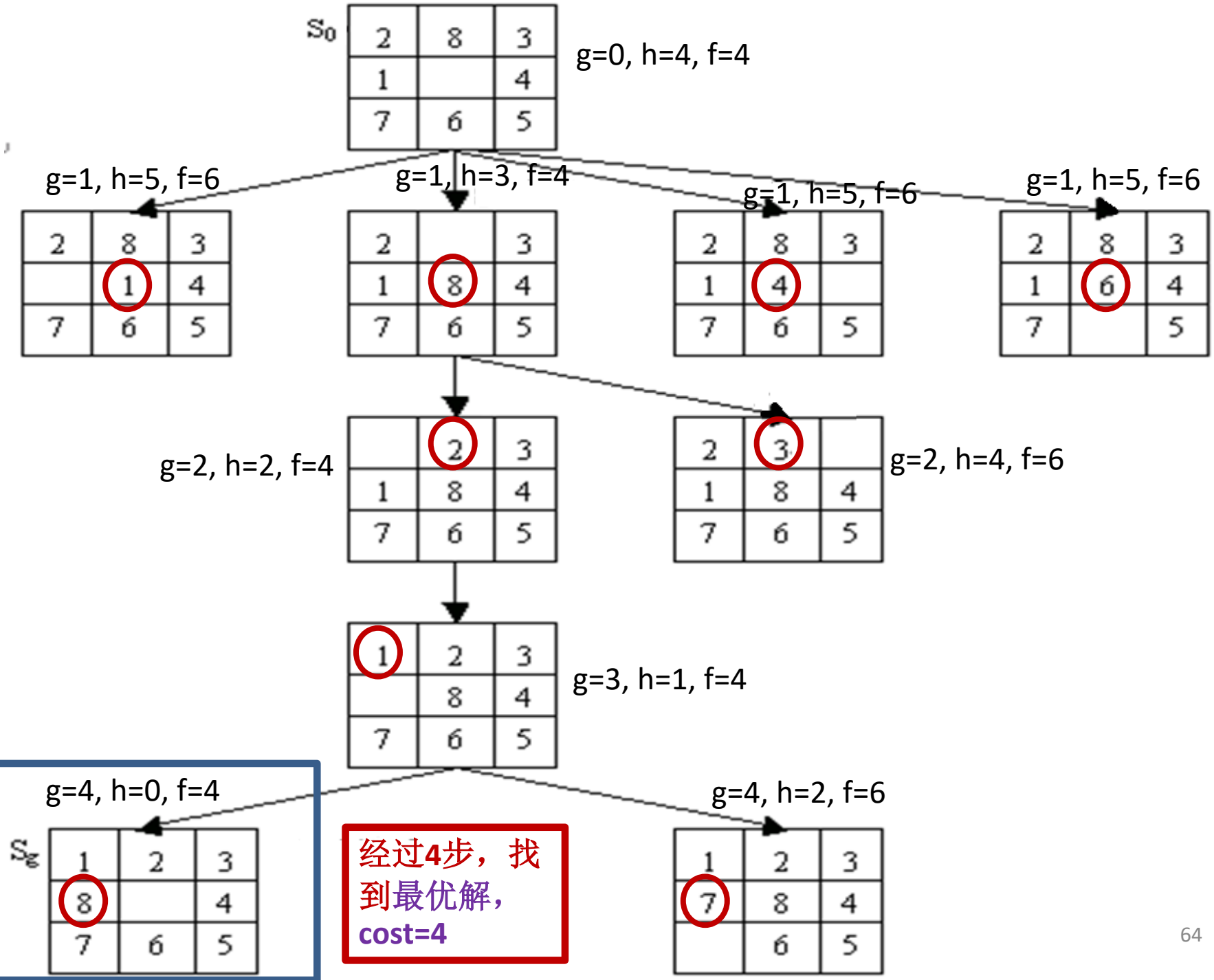
**h1=3, h2=3**

- h1(n) = “不在位”的将牌数
- h2(n) = 所有将牌与其目标位置之间的曼哈顿距离之和。

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

**goal state**

**h(n)=**所有将牌与其目标位置之间的曼哈顿距离之和



# 如何判断h(n)≤h*(n)是否成立

◆一般来说，我们并不知道h*(n) 的值，那么如何判断h(n)≤h*(n)是否成立呢?

◆这就要根据具体问题具体分析了。比如说，问题是在地图上找到一条从地点A到地点B的距离最短的路径，我们可以用当前结点到目标结点的**欧氏距离作为启发函数h(n)**。

◆虽然我们不知道h*(n)是多少，但由于两点间直线距离最近，所以**肯定有 h(n)≤h*(n)**。这样用A*算法就可以找到该问题距离最短的一条路径。

# 八数码问题的启发函数

◆h*(n)的意义是：“把当前错误的牌数移动到正确位置所需要的最小步数”。

◆可以定义的几种启发函数：

- Method A: 当前错位的牌的数量
- Method B: 错位的牌距离目标位置的距离和
- Method C: 在B的基础上，添加颠倒两张相邻牌的固定步数

◆以上启发函数h(n）均**满足： h(n) ≤ h*(n) ，均为A*算法**。

# 采用Method A，证明其为 A* 算法

- 令 **d(*n*) =已移动将牌的步数，即结点n在搜索树中的深度**

  **w(*n*) =结点n所表示的状态中“不在位”的将牌数**

- 将w(n)个**“不在位”的将牌放在其各自的目标位置上，至少需要移动**w(n)步。
- **h*(*n*)** 是结点n从当前位置移动到目标结点的最少实际步数，显然 w(n) ≤ h*(n).
- 以w(*n*)作为启发式函数h(*n*)，可以满足对h(*n*) 的下界的要求，即有 **h(n)= w(n) ≤ h*(n).**
- 因此，当选择w(*n*)作为启发式函数解决8数码问题时，A算法就是A*算法。

# 用A* 算法解决“修道士与野人”问题

有k个修道士和k个野人，一条船。修道士们要用船将所有人从左岸运到右岸。但有下面条件和限制：

（a）所有人都会划船，船一次载重不超过3个人

（b）任何时刻，在河的两岸以及船上的野人数目不能超过修道士的数目，否则野人将吃掉修道士；

（c）但允许在河的某一岸或者船上只有野人而没有传教士；

（d）野人会服从修道士的任何过河安排。

求合理的过河方案。

# “用A* 算法解决“修道士与野人”问题

Step1 设计**状态空间**表示，令


- **未过河**的修道士数量为 m
- **未过河**的野人数量为 c，
- **未过河**的船的数量为 b（=1表示船未过河，=0表示船已过河）。

◆则状态空间中的状态可以用向量：S = { m , c , b } 表示。

◆假设K=5，理论上，过河的所有状态数量为：6*6*2=72个

◆**初始状态**为(5,5,1)，**目标状态**为(0,0,0)。

# 用A* 算法解决“修道士与野人”问题

Step2 设计**操作集合，**即**过河操作**。

- 用P 代表船从左岸到右岸；用Q 代表船从右岸到左岸。
- **船上**：修道士人数为 i， 野人人数为 j ，满足:

(1) $1 \leqslant i + j \leqslant 3$;　　(2) $i \neq 0$时， $i \geqslant j$（P21表示船上有2个道士，1个野人）

根据以上限制，可能的操作共有16种（船一次载重不超过3个人）：

- 船从左岸到右岸： P01; P02; P03; P10; P11; P20; P21; P30;
- 船从右岸到左岸： Q01; Q02; Q03; Q10; Q11; Q20; Q21; Q30;

# 用A* 算法解决“修道士与野人”问题

Step3 设计满足A* 算法的启发函数

◆在初始状态(5,5,1)下，若不考虑限制(b：野人吃人)，则至少要操作9次

（初始时，**船与人在河同侧**，每次运3人过去，然后1人回来，重复4.5个来回）

◆相当于：每操作一次，只运1个人过河。

初始状态：10人　河　0人

摆渡1：7人 —3人→ 0人

摆渡2：7人 ←1人— 2人

摆渡3：5人 —3人→ 2人

摆渡4：5人 ←1人— 4人

摆渡5：3人 —3人→ 4人

摆渡6：3人 ←1人— 6人

摆渡7：1人 —3人→ 6人

摆渡8：1人 ←1人— 8人

摆渡9：0人 —2人→ 8人

# 用A* 算法解决“修道士与野人”问题

Step3 设计满足A* 算法的启发函数

- ◆相当于：每操作一次，只运1个人过河。**是否可以令h(x)=m+c？**
- ◆稍分析就可以发现，**不可以**，因为 h(x)=m+c 不满足 h(x)≤h*(x) 。

  如: 对状态x=(1,1,1)，h(x)=m+c=2，而此时最短路径上的代价h*(x) =1，即只需1步就可完成。

- ➢ 按要求，**应该： h(n) ≤ h*(n)**
- ➢ 但现在，**h*(x) =1< h(x) =2**，不满足A*算法的条件，
- ➢ 实际上：h*(x)应该有一个上限：h*(x) < m+c，即过河次数不高于10次，因为一共才10个人，不可能超过10次。

# 用A* 算法解决“修道士与野人”问题

Step3 设计满足A* 算法的启发函数

分情况讨论：

- 假设船在左岸（船与人同侧），b=1（船未过河），状态为(m,c,1)。
- 不考虑约束条件，当最后一次恰好3人同船过河时，效率最高。

  **单独算1次**。
- 剩下m+c-3个人运过河，需要运送 $\frac{(m+c-3)}{2} * 2 = m + c - 3$次，

故：**一共需要运送/摆渡 m+c-3+1 = m+c-2 次**（单向摆渡次数）。

# 用A* 算法解决“修道士与野人”问题

Step3 设计满足A* 算法的启发函数

分情况讨论：

- 假设船在右岸（船与人不同侧），b=0， 初始状态为(m,c,0)。
- 则首先需要额外有一个人把船划回左岸，消耗1次，
- 同时左岸人数增多1（总人数变为： m+c+1 ），
- 转变成第一种情况，即： (m+c,0) ↔ ( m+c+1, 1)
- 第一种情况的初始状态为（ m+c,1），一共需要运送 m+c-2 次
- 现在，用m+c+1代替上式中的 m+c，则一共需要运送 m+c-1次
- 再加上最开始“消耗1次”，则共需要运送(m+c-1) **+1** = m+c 次。

# 用A* 算法解决“修道士与野人”问题

Step3 设计满足A* 算法的启发函数

两者结合，得到：

$$h(n)=\begin{cases} m+c-2, b=1 \\ m+c \quad , b=0 \end{cases}$$

综合即，

$$h(n)=m+c-2b$$

此时满足A*条件，即 h(n)=m+c-2b ≤ h*(x)=m+c。

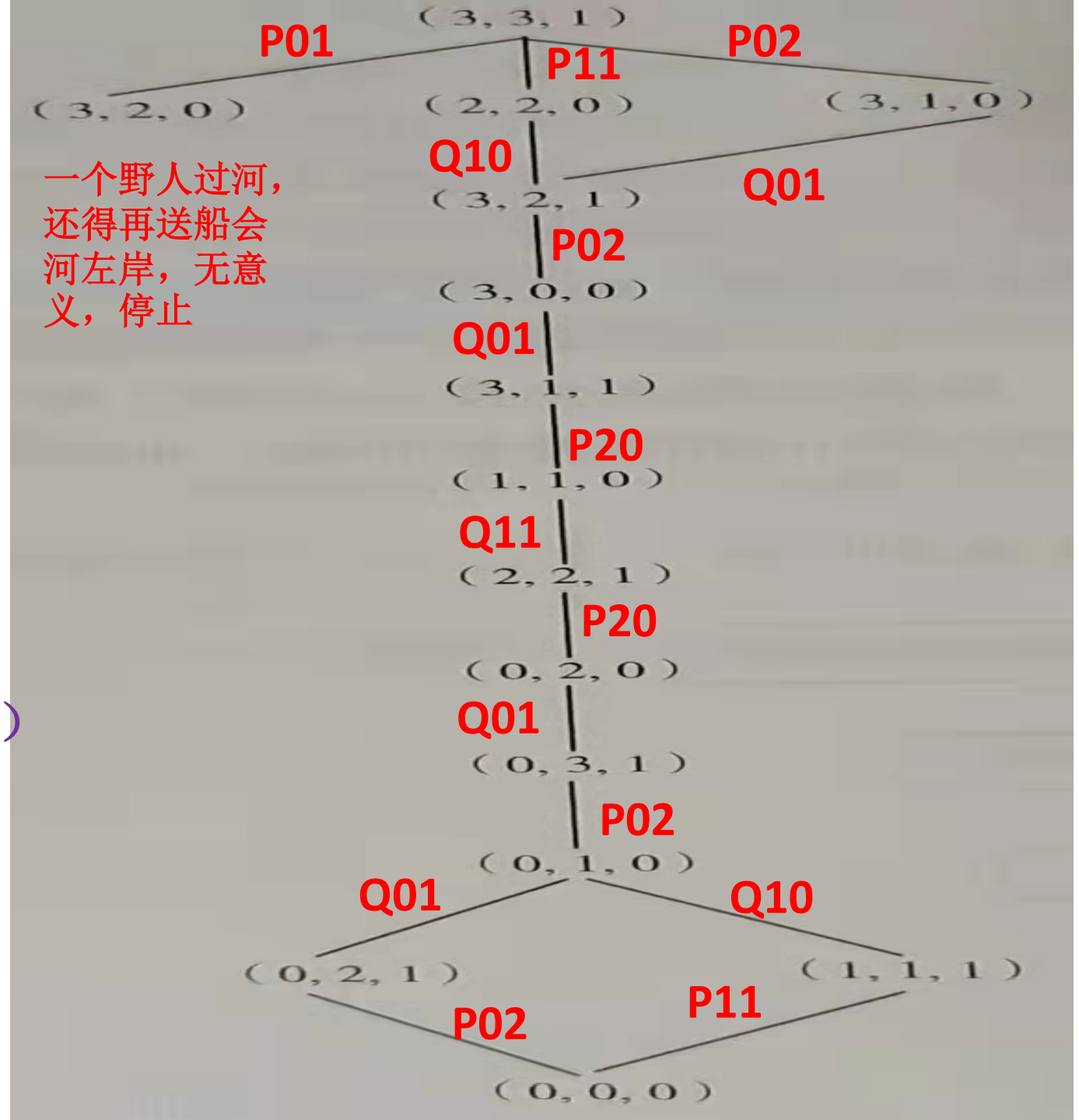
# 用A* 算法解决“修道士与野人”问题的搜索图


假设k=3， b=1（初始时，船与人同侧）

则有3个修道士和3个野人，

初始状态为（3，3，1）

目标状态为（0，0，0）




一个野人过河，还得再送船会河左岸，无意义，停止