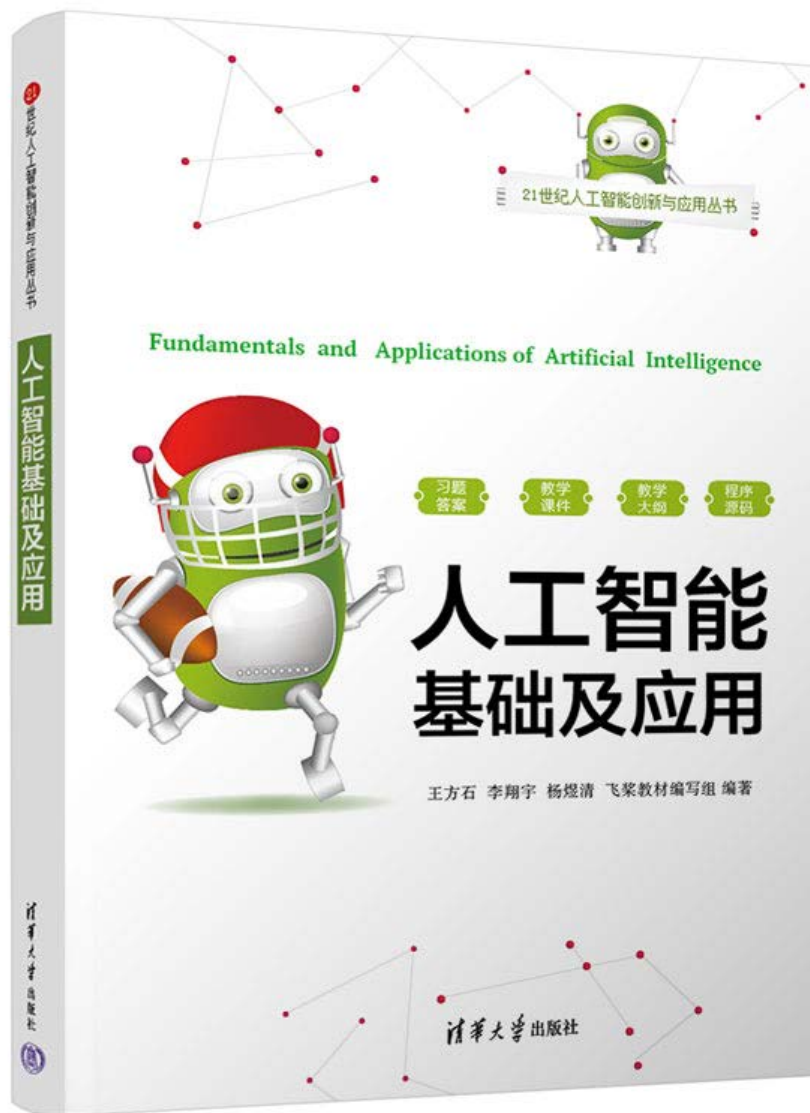


人工智能基础



北京交通大学 软件学院
王方石

Email: fshwang@bjtu.edu.cn

第3章 搜索策略

3.1 图搜索策略

3.2 盲目的图搜索策略

3.2.1 深度优先搜索 (DFS)

3.2.2 宽度优先搜索 (BFS)

3.3 启发式图搜索策略

3.3.1 A Search, 即最佳优先搜索

3.3.2 A* Search

3.4 局部搜索：爬山法、模拟退火法、遗传算法

本章学习目标

- ◆ 理解搜索的基本概念。
- ◆ 掌握盲目搜索算法：深度优先搜索和宽度优先搜索。
- ◆ 掌握启发式搜索算法：A搜索和A*搜索。
- ◆ 掌握局部搜索算法：爬山法、模拟退火法和遗传算法。

第3章 搜索策略

- ◆ 在求解许多问题时都采用试探的搜索方法，**搜索**就是一个不断尝试和探索的过程。
- ◆ 当我们终于找到了一种解决办法，又会想：这个方案是不是最优方案。
- ◆ 若不是，怎样才能找到最优方案?如何用计算机代替人类完成这样的搜索？
- ◆ 为**模拟**这些试探性的**问题求解过程**而发展的一种技术，就称为**搜索技术**。
- ◆ **搜索技术**是人工智能中的一个核心技术，它直接关系到智能系统的性能和运行效率。
- ◆ 以**状态空间表示法**描述问题空间，已知**初始状态和目标状态**，**搜索问题**就是求解**一个操作序列**使得智能体能从初始状态转移到目标状态。
- ◆ **搜索问题的主要任务**是找到正确的搜索策略。
- ◆ **搜索策略**是指在搜索过程中确定扩展状态顺序的规则。
- ◆ 所求得的从初始状态转移到目标状态的**一个操作序列**就是问题的一个**解**，若某操作序列可以使总代价最低，则该方案称为**最优解**。

3.1 图搜索策略

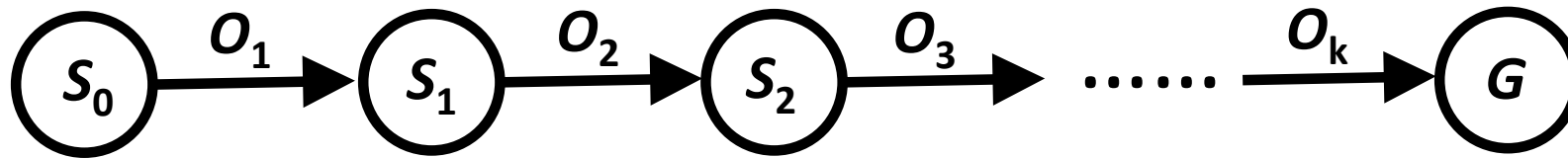
- ◆ 很多搜索问题都可以转化为图搜索问题，即在**图结构**中搜索该问题的解决方案。
- ◆ 为了进行搜索，首先需要采用某种形式表示所要求解的问题，**表示方法**是否适当，将直接影响搜索效率。
- ◆ 一般常用**状态空间表示法**描述所求解的问题空间。
- ◆ 然后在状态空间中搜索，以求得一个从初始状态到目标状态的操作算子序列，即**问题的解**。
- ◆ 对于一个确定的问题，与求解有关的状态空间往往只是整个状态空间的一部分，只要能生成并存储这部分状态空间，就可求得问题的解。
- ◆ 在状态空间图中，求解一个问题就是从初始状态出发，不断运用可使用的操作，在**满足约束的条件下**达到目标状态，故**搜索技术**又称为“**状态图搜索**”方法。

回顾2.4节的概念

- ◆ **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，**节点**表示问题的**状态**，节点之间的**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**。

回顾2.4节的概念

- ◆ **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 $\{O_1, O_2, \dots, O_k\}$ 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

通过图搜索求解问题的基本过程如下：

- (1) 采用状态空间表示法描述所有状态，并给出问题的初始状态和目标状态。
- (2) 规定一组操作（算子），每个操作（算子）都能够将一个状态转换到另一个状态。
- (3) 选择一种搜索策略，用于遍历或搜索该问题的状态图空间。
- (4) 将问题的初始状态（即初始节点）作为当前状态。
- (5) 按已确定的搜索策略选择适用的操作（算子），对当前状态进行操作，生成一组后继状态（或称为后继节点、子节点）。
- (6) 检查新生成的后继状态中是否包含目标状态。若包含，则搜索到了问题的解，从初始状态到达目标状态的操作序列即为解，算法结束；若不包含，则按已确定的搜索策略，从已生成的状态中选择一个状态作为当前状态，若已无可操作的状态，则未搜索到解，算法结束。
- (7) 返回至第（5）步。

3.1 图搜索策略

- ◆ 在实现图搜索的算法中，需建立两个数据结构：**OPEN表**和**CLOSED表**。
- ◆ **OPEN表**用于存放待扩展的节点，**CLOSED表**用于存放已扩展的节点。
- ◆ 所谓**扩展节点**，是指用合适的操作（算子）对该节点进行操作，生成一组后继节点。
- ◆ 一个节点经过一个算子操作后，一般只生成一个后继节点，但对于一个节点，适用的算子可能有多个，故此时会生成**一组后继节点**。
- ◆ **需要注意的是**：在这些后继节点中，可能包含当前扩展节点的父节点、祖父节点，则这些**祖先节点不能作为当前扩展节点的子节点**。
- ◆ 图搜索**不允许重复访问节点**，即**OPEN表和CLOSED表的交集为空**。
- ◆ **图搜索策略**就是**选择下一个被扩展节点的规则**。

基于状态空间图的搜索算法的步骤

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节点, 每次循环从OPEN表中取出一个节点加以扩展, 并把新生成的节点加入OPEN表;
2. $CLOSED:=()$;
// **CLOSED**表用来存储已扩展的节点, 其用途是检查新生成的节点是否已被扩展过。
// 图搜索不允许重复访问节点, 即 **OPEN表** \cap **CLOSED表** = \emptyset .
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 归入已被扩展的节点
5. IF $GOAL(n)$ THEN **EXIT**(SUCCESS);

6. EXPAND(n) \rightarrow $\{m_i\}$, $G := \text{ADD}(m_i, G)$;

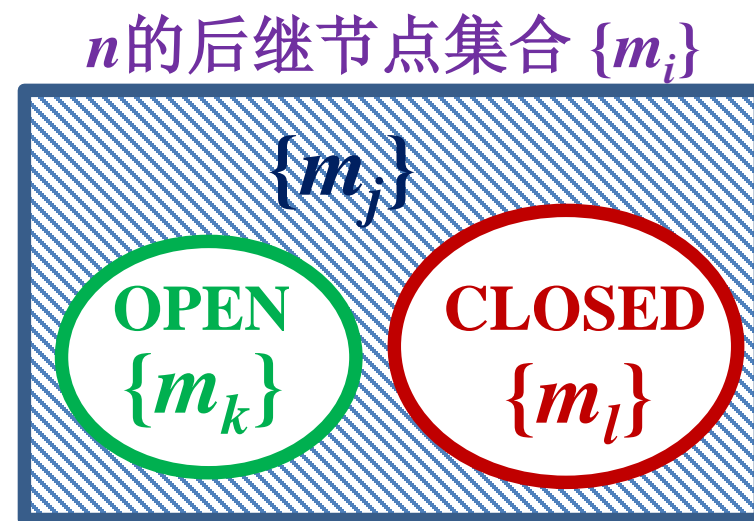
// 扩展节点 n , 建立集合 $\{m_i\}$, 使其包含 n 的后继节点, 而不包含 n 的祖先, 并将这些后继节点加入 G 中。

注: n 的后继节点有三类: $\{m_i\} = \{m_j\} \cup \{m_k\} \cup \{m_l\}$,

(1) n 的后继节点 m_j 既不包含于 OPEN, 也不包含于 CLOSED;

(2) n 的后继节点 m_k 包含在 OPEN 中;

(3) n 的后继节点 m_l 包含在 CLOSED 中;



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

//对 $\{m_i\}$ 中所有节点，标记和修改其前驱指针：

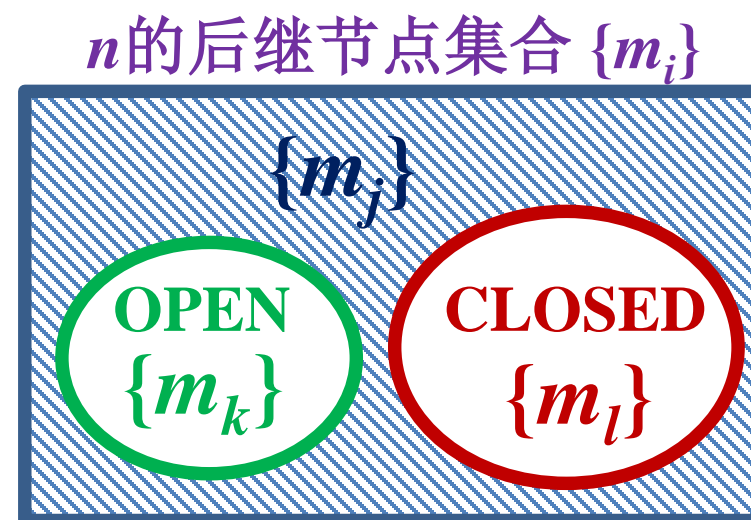
(1) ADD (m_j , OPEN) , 并令 m_j 的前驱指针指向 n ;

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

(3) 判断是否需要修改 m_l 后继节点的前驱指针，使其指向 m_l 本身;

8. 对OPEN中的节点按某种原则重新排序;

9. GO LOOP（语句3）;



两种方式的搜索策略

- (1) 在不具备任何与给定问题有关的知识或信息的情况下，系统按照某种固定的规则依次或随机地调用操作算子，这种搜索方法称为**盲目搜索策略**（Blind Search Strategy），又称为**无信息引导的搜索策略**（Uninformed Search Strategy）；
- (2) 可应用与给定问题有关的领域知识，动态地优先选择当前最合适的操作算子，这种搜索方法称为**启发式搜索策略**（Heuristic Search Strategy）或**有信息引导的搜索策略**（Informed Search Strategy）。

不同搜索策略的搜索性能也不同。

搜索策略性能的评价标准

◆完备性

- 当问题有解时，保证能找到一个解，具有完备性。
- 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），具有最优性。
- 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。

3.2 盲目的图搜索策略

◆ 盲目搜索也被称为**无信息搜索**、**通用搜索**。

在盲目搜索的过程中，没有任何与问题有关的**先验知识**或者**启发信息**可以利用，算法**只能判断当前状态是否为目标状态**，而**无法比较两个非目标状态的好坏**。

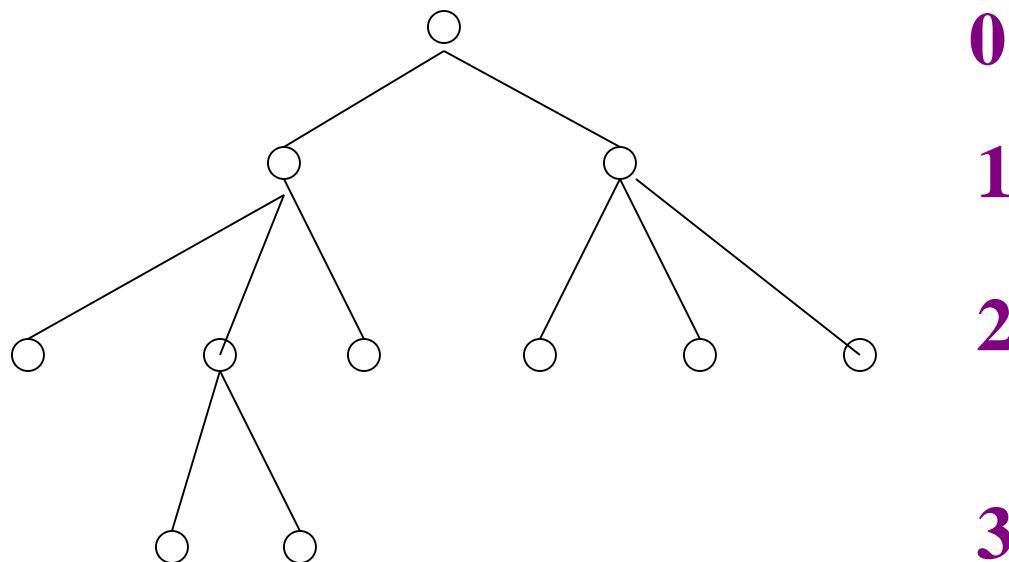
◆ 常用的两种盲目搜索方法：

(1) **深度优先搜索 (Depth-first search, DFS)**

(2) **宽度优先搜索 (Breadth-first search, BFS)**

(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**优先扩展当前深度最深的节点**，是树的**先根遍历**。
- ◆ 在一个图中，**初始节点的深度定义为0**，其他节点的深度定义为其父节点的深度加 1。



(1) 深度优先搜索---DFS

- ◆DFS总是选择**深度最深**的节点进行扩展，
- ◆若有**多个**相同深度的节点，则按照指定的规则从中选择一个；
- ◆若该节点**没有子节点**，则选择一个除了该节点之外的深度最深的节点进行扩展。
- ◆以此类推，直到**找到问题的解**为止；或者直到**找不到可扩展的节点**，结束搜索，此种情况说明没有找到问题的解。

(1) 深度优先搜索---DFS

◆DFS 的实现方法

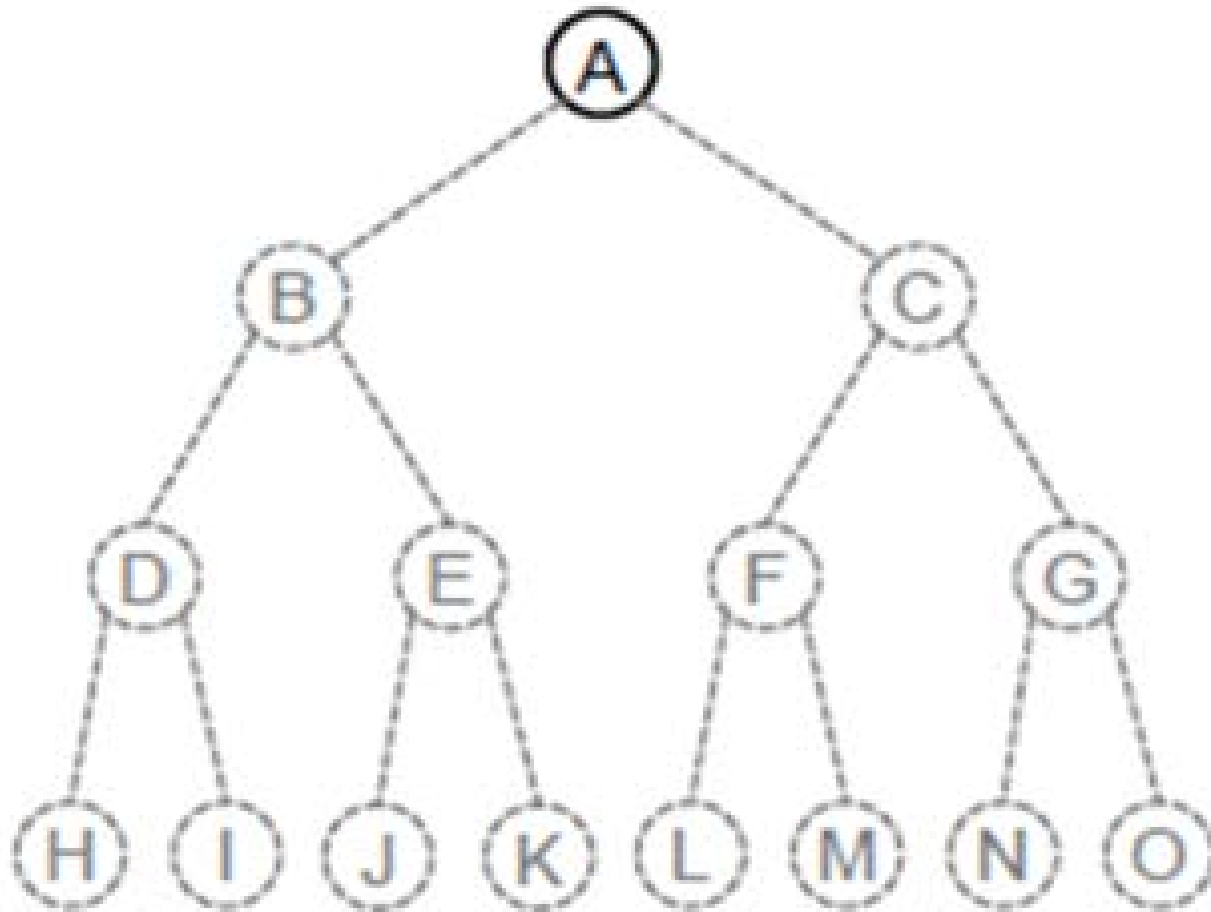
使用 **LIFO** (Last-In First-Out)的**栈**存储OPEN表，把后继节点放在**栈顶**。

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

◆ **DFS**总是扩展搜索树中当前OPEN表中**最深**的节点（即**栈顶元素**）。

◆ 搜索很快推进到搜索树的最深层，那里的节点没有后继。当那些节点被扩展完之后，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节点。

DFS: 类似于树的先根遍历



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

DFS算法

深度优先搜索算法的过程如下：

- (1) 将初始节点 S 放入OPEN表的栈顶；
- (2) 若OPEN表为空，表示再也没有可扩展的节点，即未能找到问题的解，则算法结束；
- (3) 将OPEN表的**栈顶元素**（记为节点 n ）取出，放入CLOSED表中；
- (4) 若节点 n 是目标节点，则已求得问题的解，算法结束；
- (5) 若节点 n 不可扩展，即 n 没有后继节点，则转至步骤（2）；
- (6) 扩展节点 n ，将其**所有未被访问过的子节点**依次**放入OPEN表的栈顶**，并将这些子节点的前驱指针设为指向父节点 n ，然后转至步骤（2）。

深度优先搜索的性质

- ◆ 若**状态空间有限**，**DFS是完备的**，因为它最多扩展所有节点，直到找到一个解。
- ◆ 但在**无限状态空间**中，若沿着一个“错误”的路径搜索下去而陷入“深渊”，则会导致无法到达目标节点，在这种情况下，DFS**是不完备的**。
- ◆ 为避免此情况发生，在DFS中往往会加上一个**深度限制**，称为**深度受限的深度优先搜索**，即若一个节点的深度达到了事先指定的深度阈值 k ，强制进行回溯，选择一个比它浅的节点进行扩展，而不是沿着当前节点继续扩展。
- ◆ 当深度限制**过深**时，会陷入“**深渊**”，**求解效率低，未必会找到最优解**；
若深度限制**过浅**，可能找不到解，即**不完备**。
- ◆ 所以，应该根据具体问题**合理地设定深度限制值**，或在**搜索过程中逐步加大深度限制值**，反复搜索，直到找到解。
- ◆ **DFS不一定是完备的，也未必能找到最优解**。
- ◆ **最坏情况时，DFS的搜索空间等同于穷举**。
- ◆ **DFS是一个通用的、与问题无关的方法**。

DFS无法避免冗余路径，不具有最优性。

深度=0

深度=1

深度=2

深度=3

深度=4

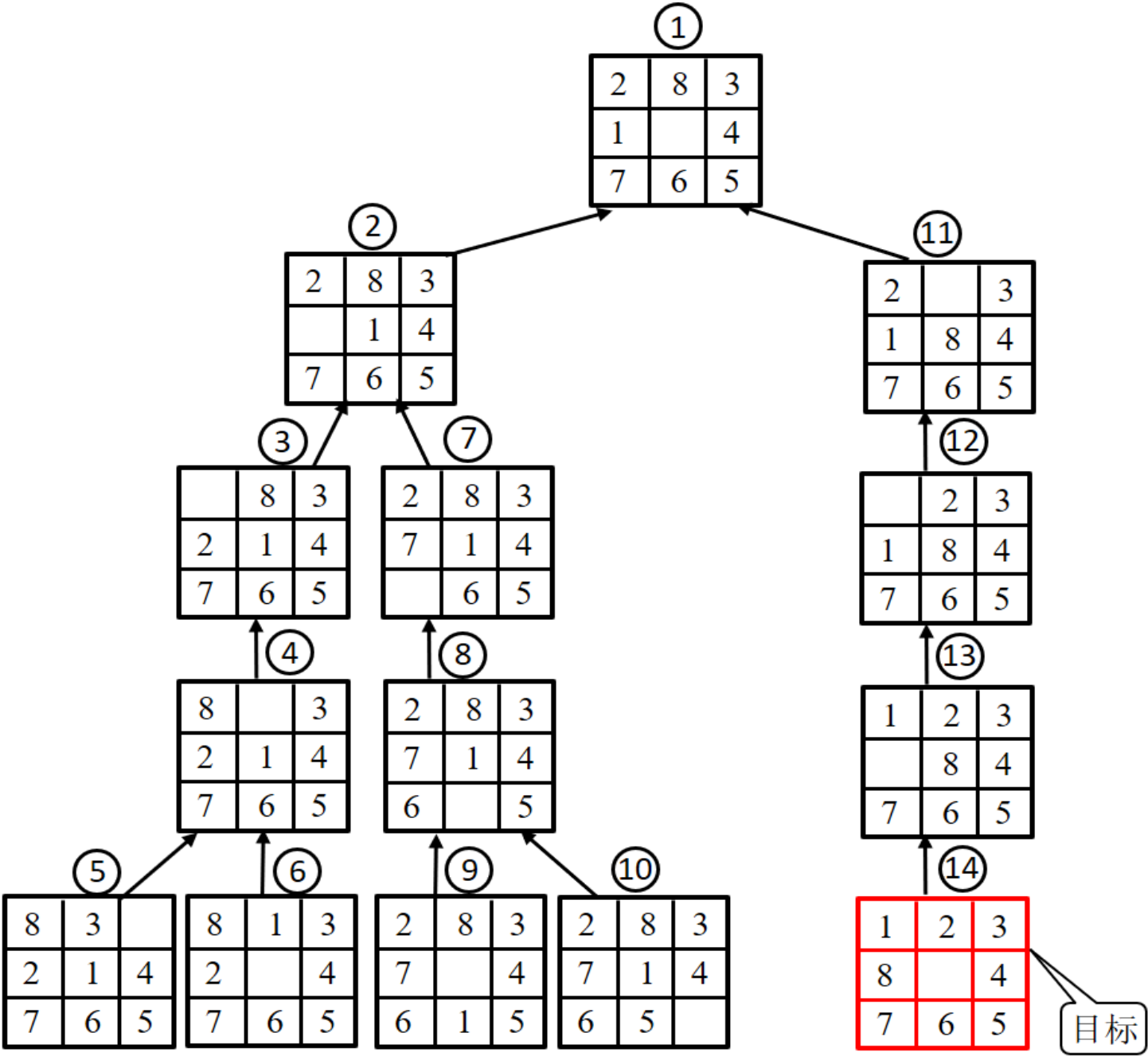


图3.2 采用深度限制为4的深度优先搜索算法求解八数码问题的搜索图

例3.1 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：

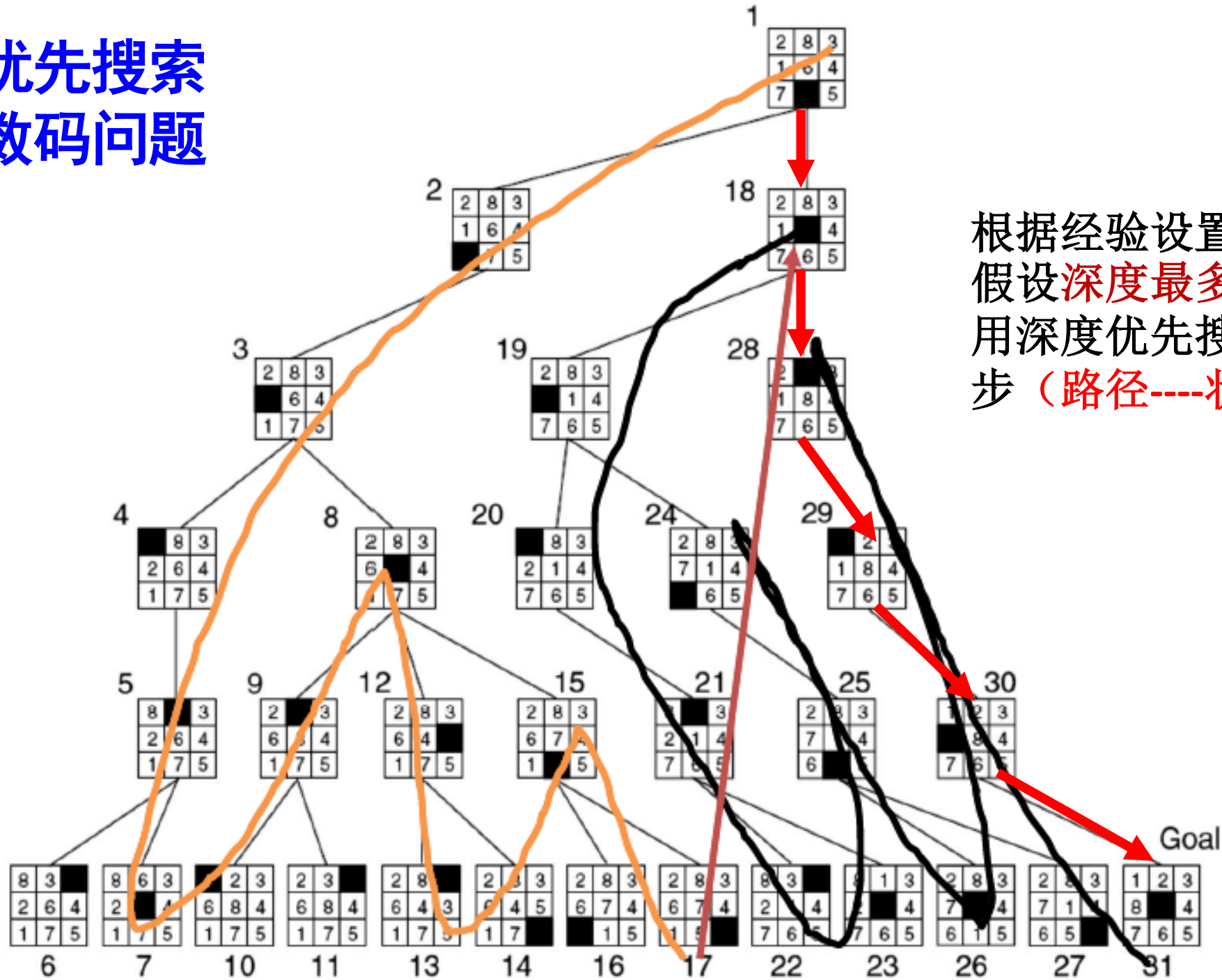
2	8	3
1	6	4
7		5

(a) 初始状态

1	2	3
8		4
7	6	5

(b) 目标状态

用深度优先搜索 解决八数码问题



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30步（路径----状态序列）。

(2) 宽度优先搜索---BFS

◆宽度优先搜索也称为广度优先搜索。

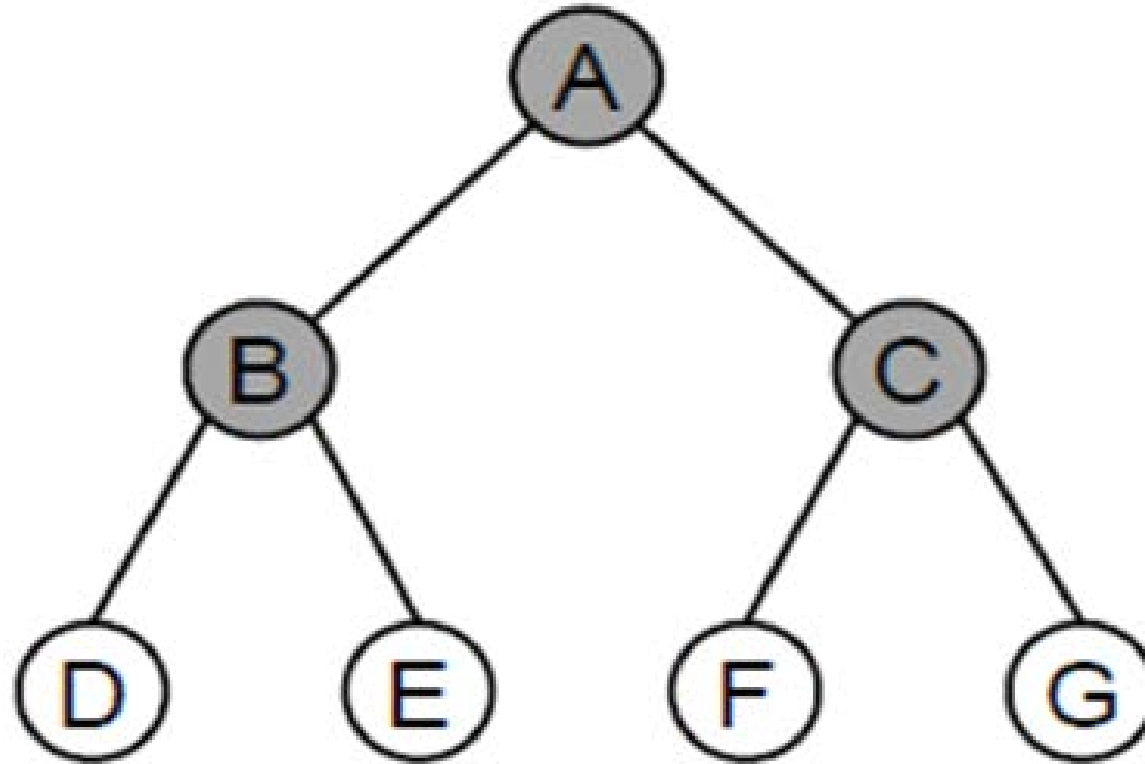
◆基本思想是：优先扩展深度最浅的节点。

➤ 先扩展根节点，再扩展根节点的所有后继，然后再扩展它们的后继，依此类推。

➤ 如果有多个节点深度是相同的，则按照事先约定的规则，从深度最浅的几个节点中选择一个，进行扩展。

◆一般地，在下一层的任何节点扩展之前，搜索树上本层深度的所有节点都应该已经扩展过。

宽度优先搜索：类似于树的层次遍历



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

宽度优先搜索的实现方法

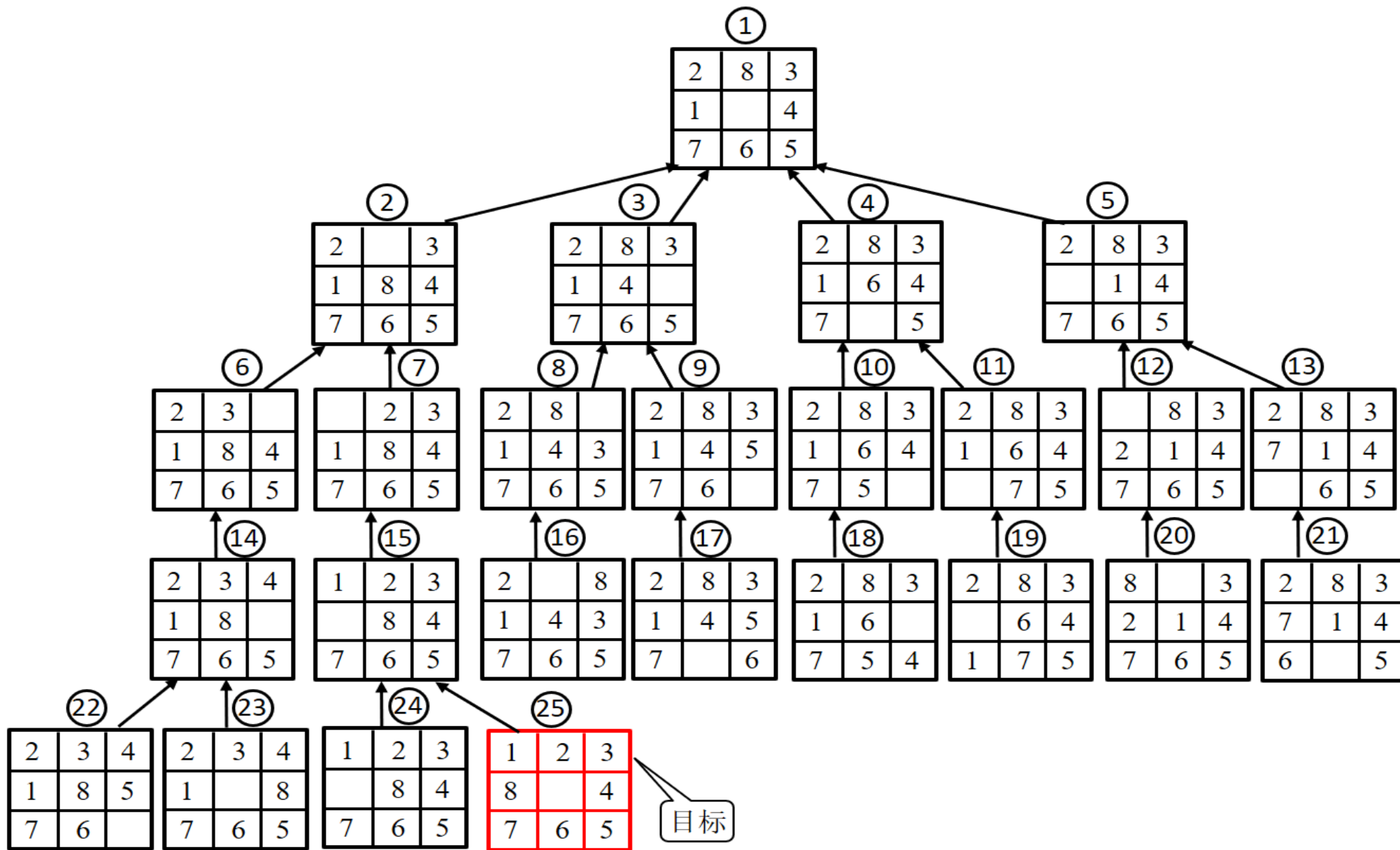
- 采用**FIFO** (First-In First-Out)的**队列**存储OPEN表。
- **BFS**是将OPEN表中的节点按搜索树中节点**深度的增序**排序，**深度最浅**的节点排在最前面（**队头**），深度相同的节点可以任意排列。
- 新节点（深度比其父节点深）总是加入到**队尾**，深度相同的节点可按某种事先约定的规则排列，这意味着浅层的老节点会在深层的新节点之前被扩展。

BFS算法

宽度优先搜索算法的过程如下：

- (1) 将初始节点S放入OPEN表的队头；
- (2) 若OPEN表为空，表示再也没有可扩展的节点，即未能找到问题的解，则算法结束；
- (3) 将OPEN表的**队头元素**（记为节点n）取出，放入CLOSED表中；
- (4) 若节点n是目标节点，则已求得问题的解，算法结束；
- (5) 若节点n不可扩展，即n没有后继节点，则转至步骤（2）。
- (6) 扩展节点n，将其**所有未被访问过的子节点**依次放入**OPEN表的队尾**，并将这些子节点的前驱指针设为指向父节点n，然后转至步骤（2）。

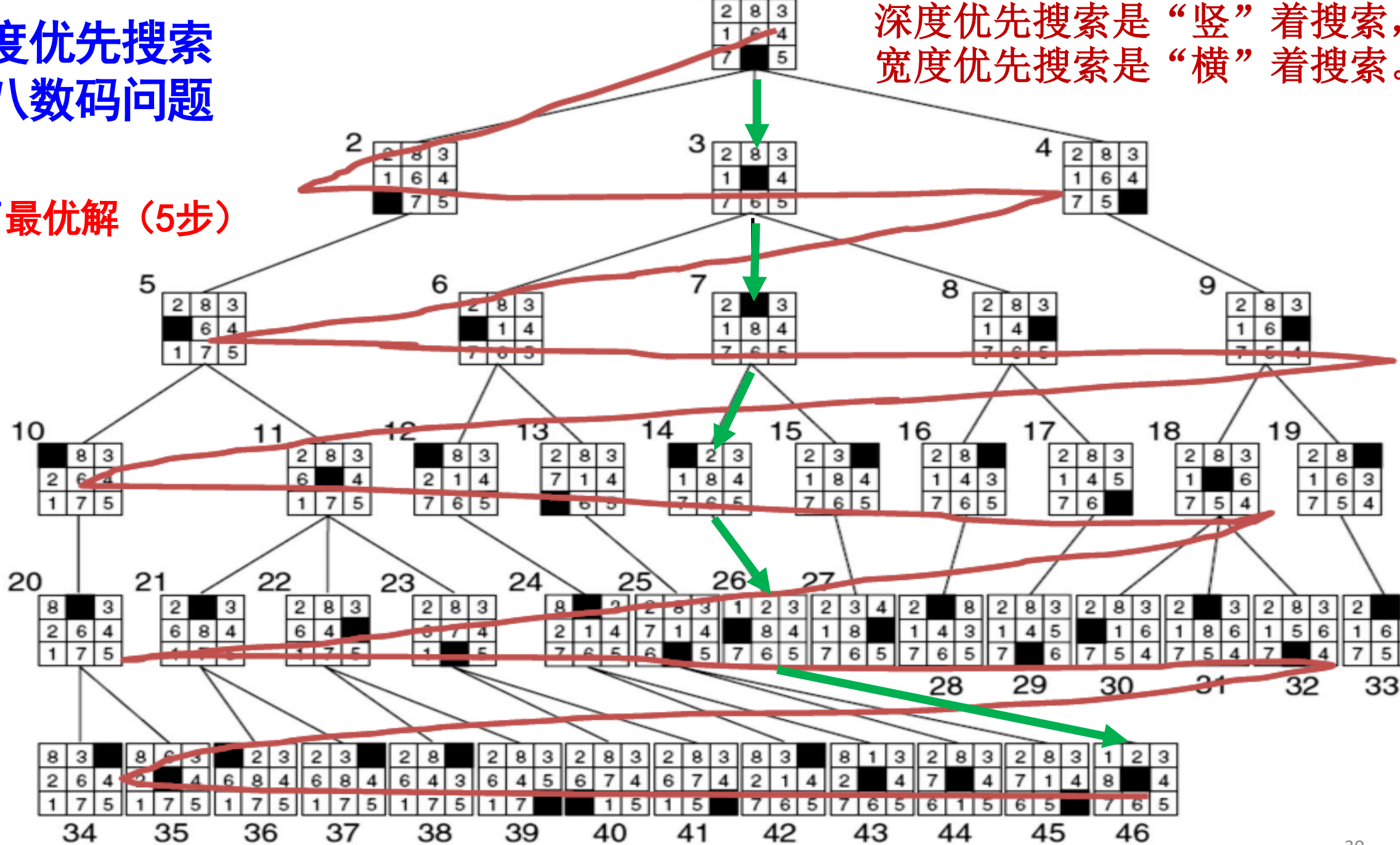
采用宽度优先搜索算法求解八数码问题的搜索图



用宽度优先搜索
解决八数码问题

找到了最优解 (5步)

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

- ◆ **BFS是完备的**（complete），即当问题有解时，一定能找到解。
- ◆ 若路径代价是节点深度的非递减函数，或者每步代价都相等，则宽度优先搜索一定能找到最优解，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**的未扩展节点；BFS总是首先扩展**最浅**的未扩展节点。
- ◆**BFS是完备的，且是最优的**；**DFS既不完备，也不最优**。
- ◆ **DFS**节省大量时间和空间。BFS需占用较大的存储空间。
- ◆在**不要求求解速度**且目标节点的层次**较深**的情况下，**BFS优于DFS**，因为BFS一定能求得问题的解，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**且目标节点的层次**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**：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**：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 启发式图搜索策略

- ◆ 盲目搜索策略在搜索过程中，仅采用**固定搜索模式**，不针对具体问题。
- ◆ 没有利用所求解问题的任何先验信息，既不对待扩展的状态的优劣进行判断，也不考虑所求的解是否为最优解。
- ◆ 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

1	4	3
7		6
5	8	2

VS

2	8	3
1		4
7	6	5

B. 错位3个



1	2	3
8		4
7	6	5

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。

盲目搜索策略会导致所需扩展的节点数很多，产生很多无用节点，效率较低。

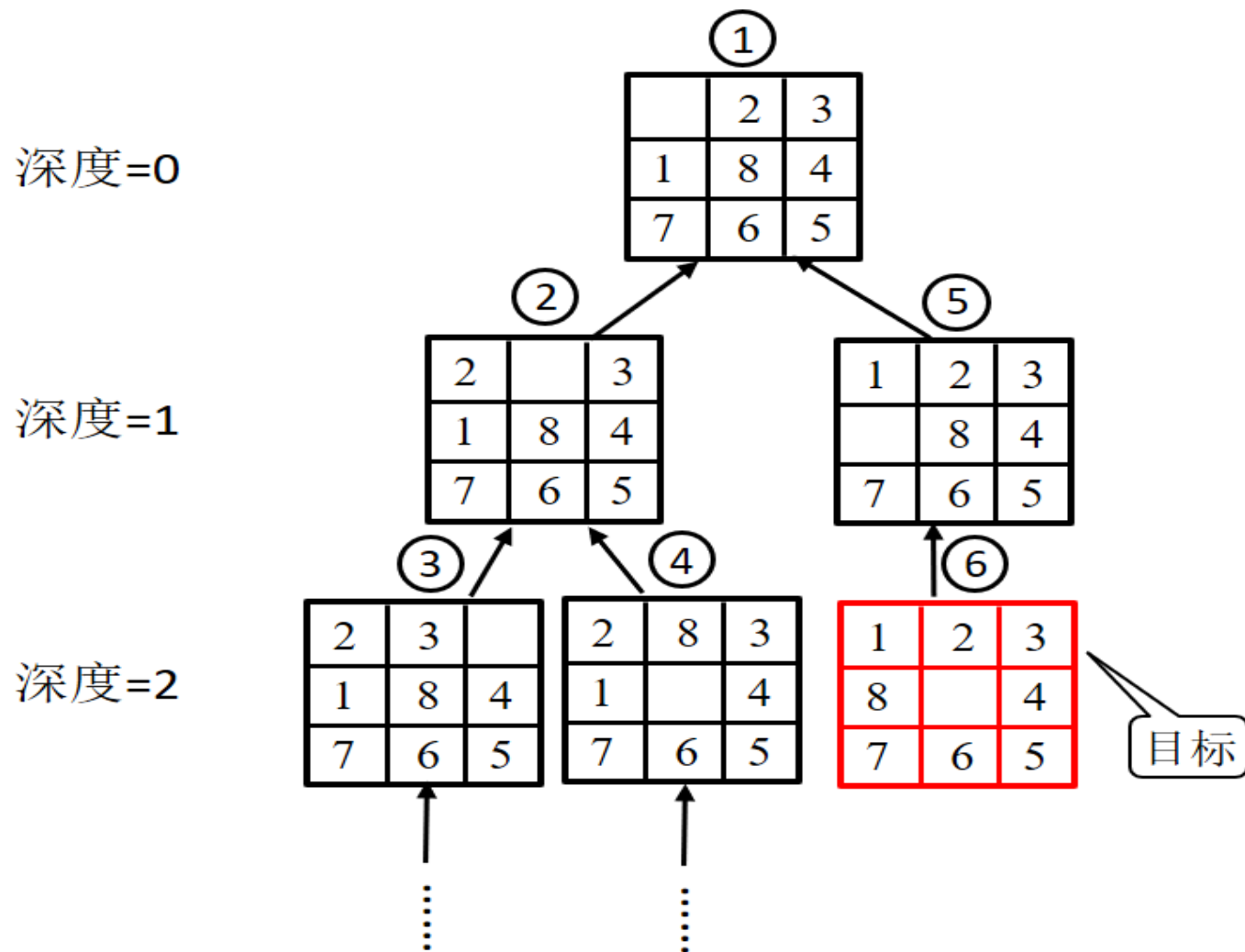


图3.4八数码问题中节点⑤比节点①的状态好

如何高效率地搜索？

- ◆ **启发式搜索**将人解决问题的“知识”告诉机器，使得搜索算法能够利用启发式信息，更“聪明”地进行搜索，尽可能地缩小搜索范围，减少试探的次数，提高搜索效率，避免大海捞针。
- ◆ **启发式搜索策略**的**基本思想**：在搜索过程中利用与所求解问题有关的特征信息，指导搜索**向最有希望到达目标节点的方向前进**。启发式搜索的**每一步都选择最优的操作**，以最快速度找到问题的解。
- ◆ 为了尽快找到从初始节点到目标节点的一条代价比较小的路径，在搜索的每一步，我们都希望选择在最佳路径（即代价最小的路径）上的节点进行扩展。

评价函数 (evaluation function)

如何评价一个节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:

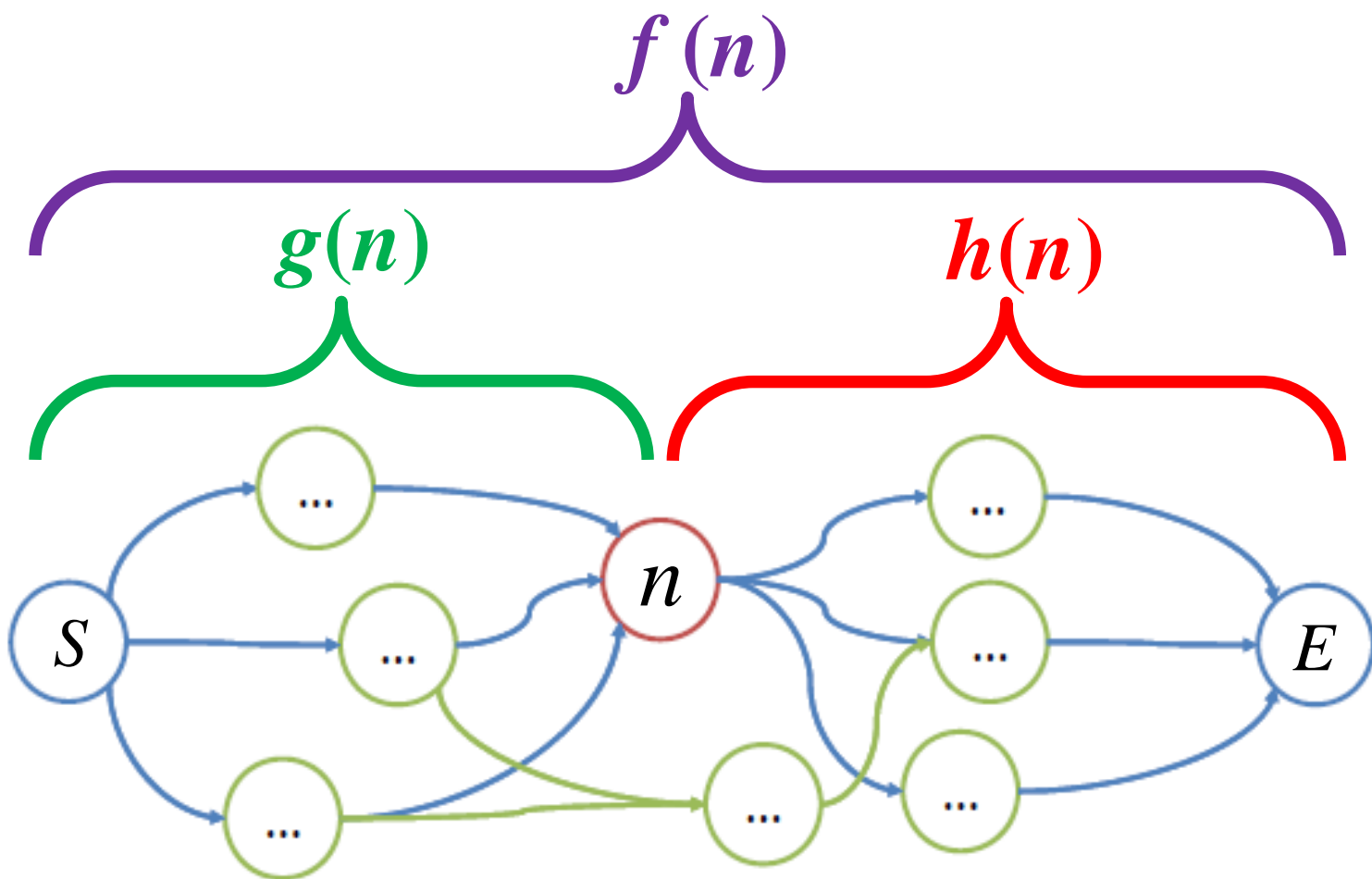
$f(n) = g(n) + h(n)$. 其中, n 为当前节点, 即待评价节点。

$f(n)$ 是从初始节点 s 出发、经过节点 n 、到达目标节点的**最佳路径代价值**的估计值.

(1) $g(n)$ 为从初始节点到节点 n 的**最佳路径代价值**的**实际值**;

(2) $h(n)$ 为从节点 n 到目标节点的**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 S 到达节点 n 的最佳路径上代价的**实际值**
- $h(n)$: 从节点 n 到目标状态 E 的最佳路径上代价的**估计值**, 称为**启发函数**。
- $f(n)$ 为从初始状态 S 经过节点 n 到达目标状态 E 的最佳路径上代价的**估计值**, 称为**评价函数**。

如何设计启发函数? Heuristic function

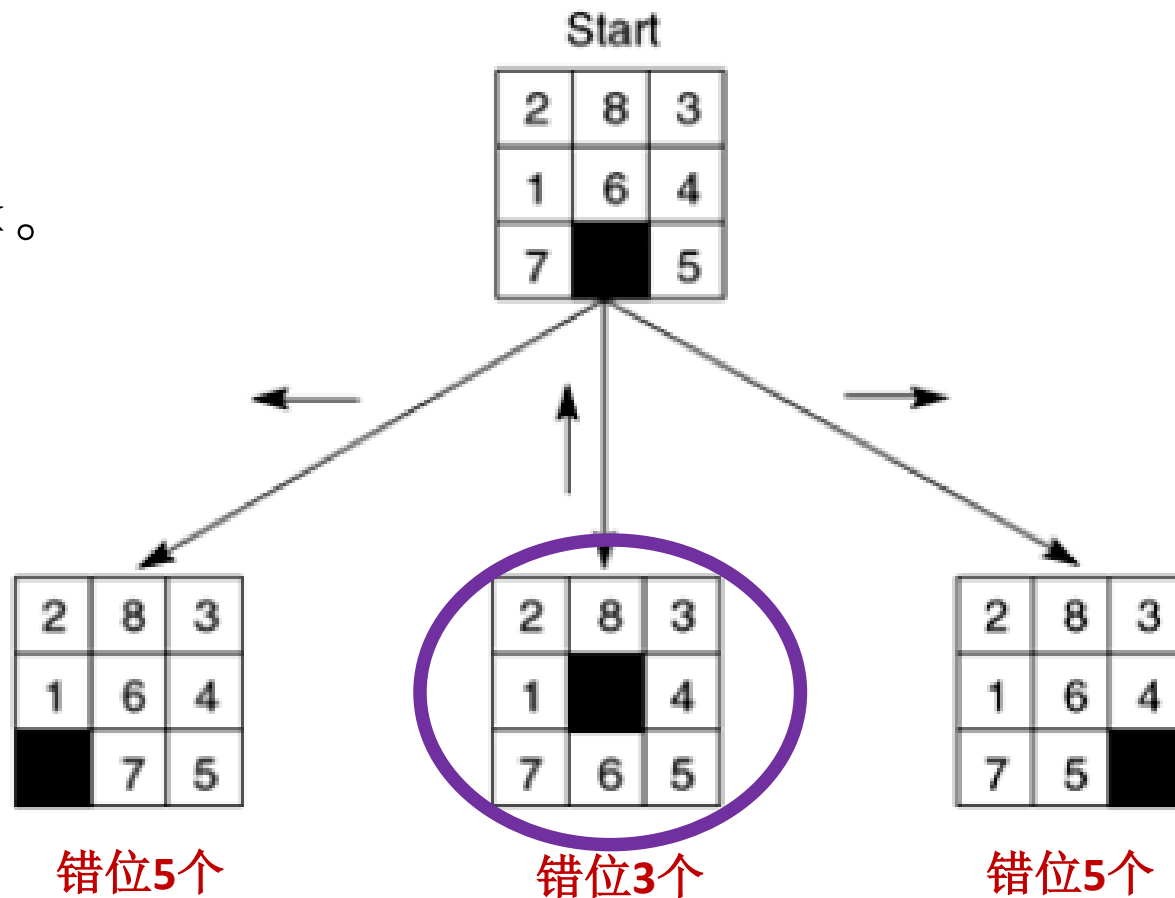
以8数码问题为例

◆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？

1	2	3
8		4
7	6	5

Goal



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数码牌的数量，**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，
但是移动距离显然不同。

2	1	3
8		4
7	6	5

相对容易移动

VS

5	2	3
8		4
7	6	1

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“错位数码牌与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。

2	1	3
8		4
7	6	5

1	2	3
8		4
7	6	5

Goal

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，在考虑距离的同时，再加上需要颠倒的数量。

如何设计启发函数？ Heuristic function

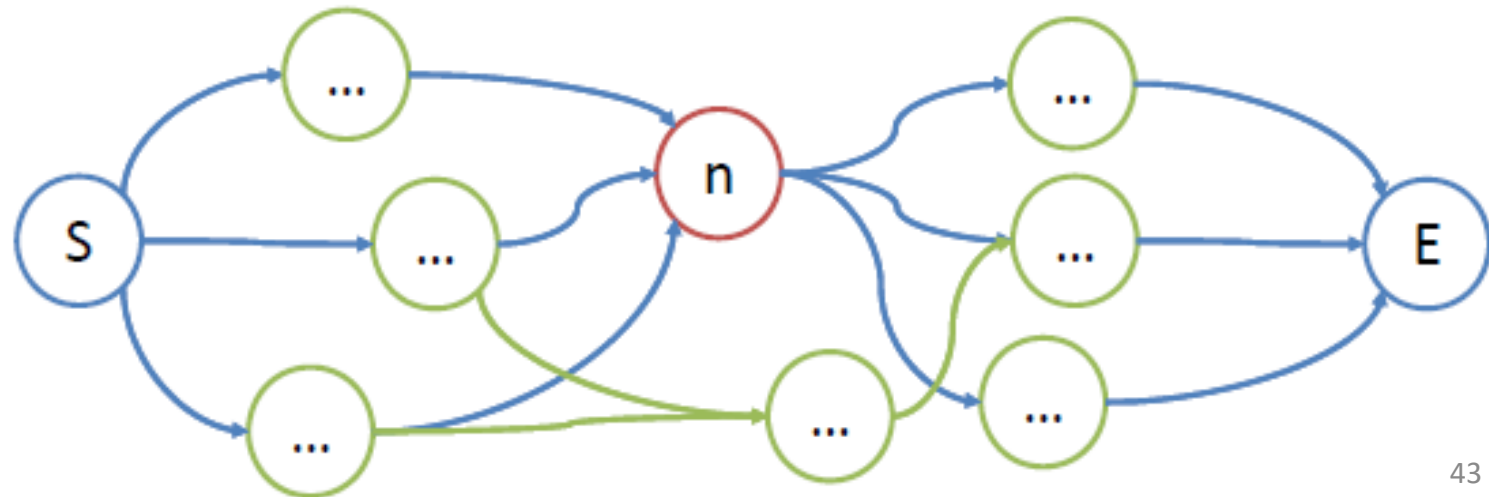
单纯依靠启发函数搜索是否可行？

◆ 对于一个具体问题，可以定义**最优路线**：“从**初始节点**出发，以**最优路线**经过**当前节点**，并以**最优路线**达到**目标节点**”。

➤ **盲目搜索**，只考虑了前半部分，能计算出从初始节点走到当前节点的优劣。

➤ **启发函数**，只考虑了后半部分，只“估计”了当前节点到目标节点的优劣。

◆ 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），是一种贪心算法。

◆ **核心思想**是：每一步都选择距离目标最近的节点进行扩展。

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

◆ **实现方法**

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

评价函数的情况

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

- ◆ If $f(n) = g(n)$, i.e. $h(n)=0$ 时, A搜索退化为盲目搜索;
- ◆ If $f(n) = g(n) = d(n)$, 即 n 的深度, 则为 **BFS** (盲目搜索).
- ◆ 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**.
- ◆ 贪婪最佳优先搜索的搜索策略: 在每一步, 它总是优先扩展与目标最接近的节点。
- ◆ 贪婪搜索策略不考虑整体最优, 仅求取局部最优。
- ◆ 贪婪搜索是不完备的, 也不具有最优性, 但其搜索速度非常快。

用 A 算法解决八数码问题

首先，需要设计评价函数： $f(n) = g(n) + h(n)$ ，令：

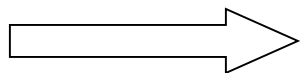
◆ $g(n)$ 为移动数码牌的步数，即节点 n 的深度。

◆ $h(n)$ 为当前状态中“错位数码牌”的个数。

例3.3

8		3
2	1	4
7	6	5

(a) 初始状态 s



1	2	3
8		4
7	6	5

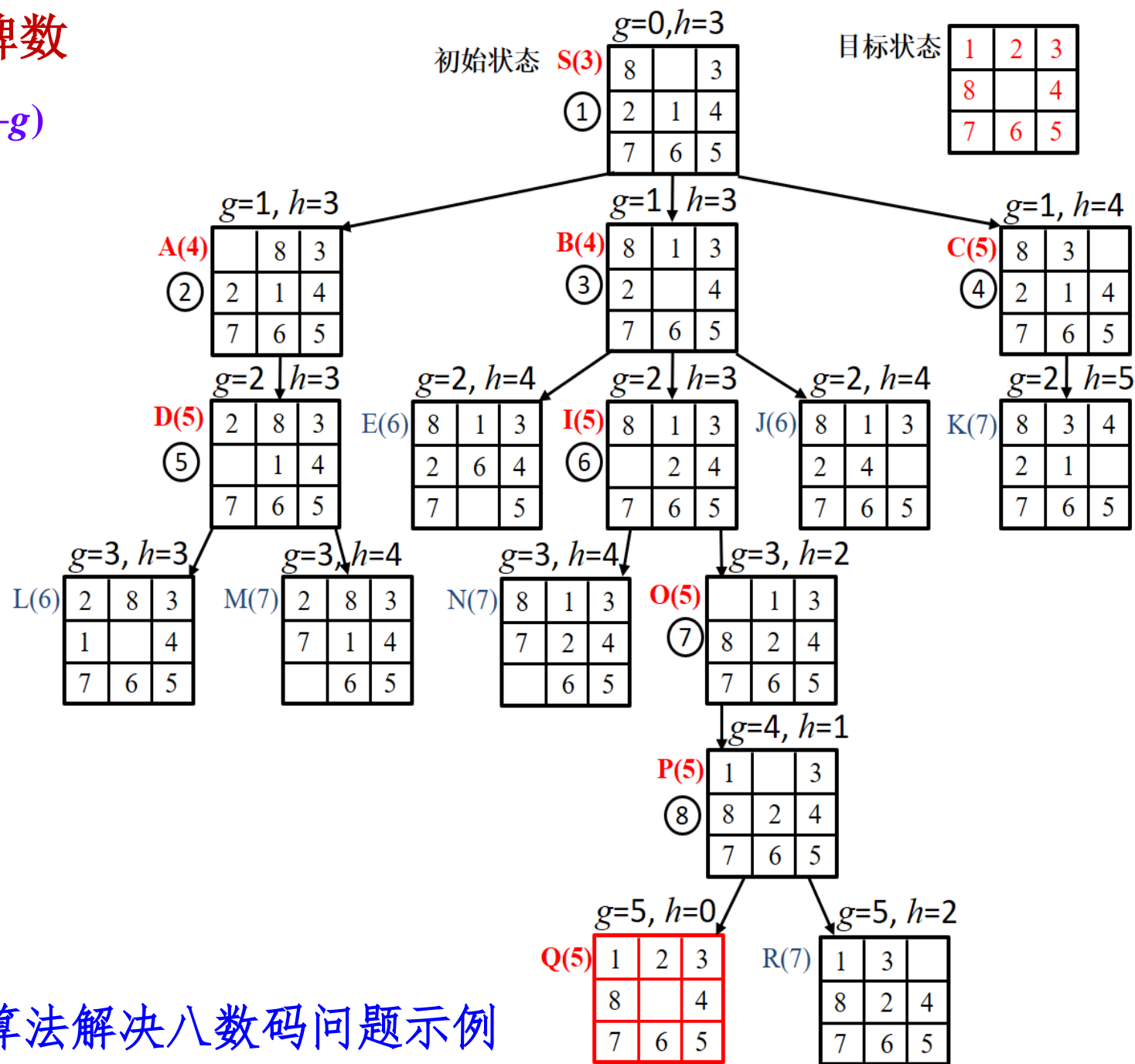
(b) 目标状态

8		3
2	1	4
7	6	5

$$g(s)=0, h(s)=3$$

$h(n)$ = 错位数码牌数

节点名称 (f 值= $h+g$)

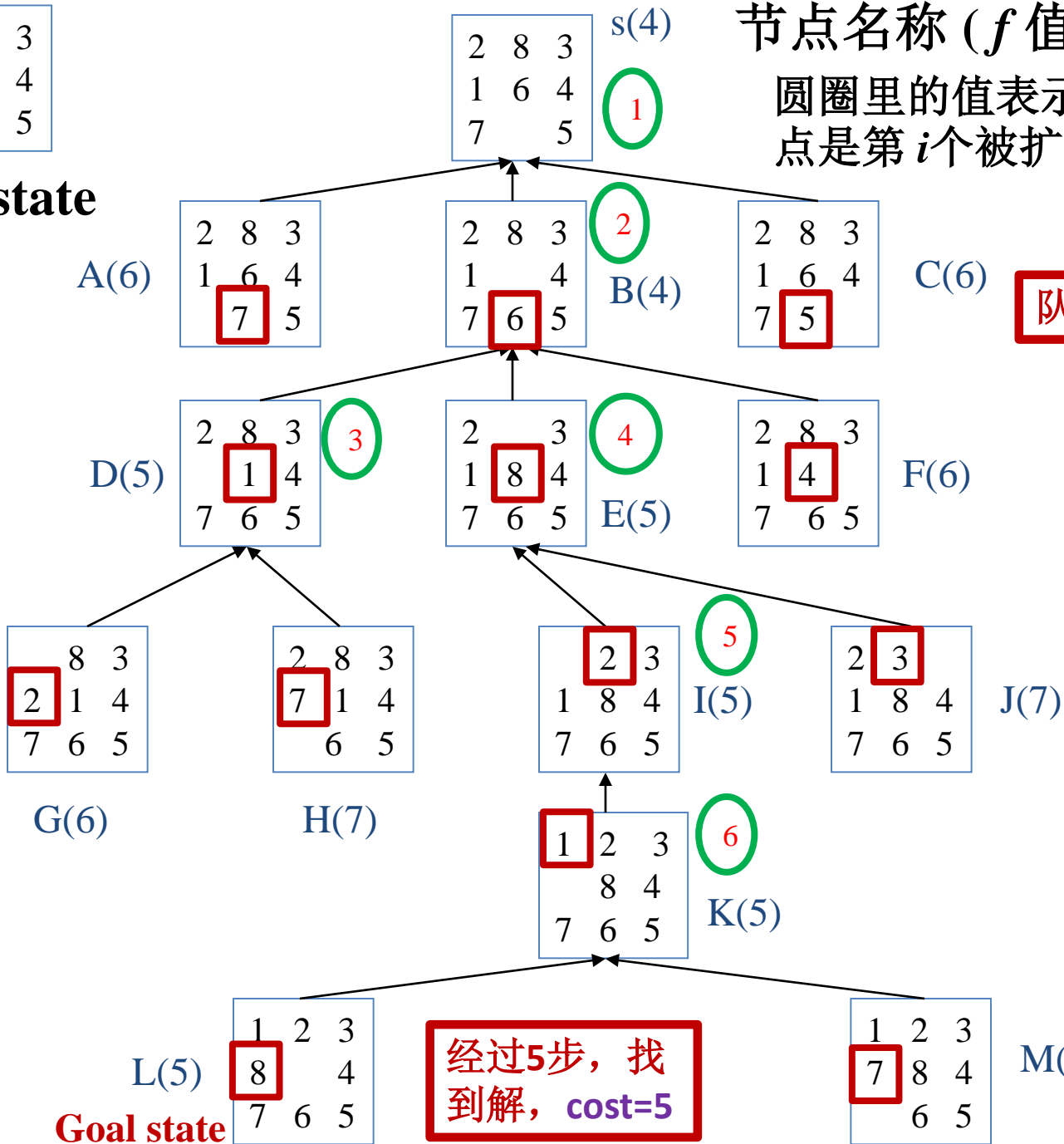


空白格的操作顺序
不固定，圆圈里的
数字是访问的顺序

图3.6 采用A算法解决八数码问题示例

1	2	3
8		4
7	6	5

Goal state



3.3.2 A* 搜索算法

- ◆ A搜索算法没有对启发函数 $h(n)$ 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搜索不完备），或者找到的不是问题的最优解（A搜索不具有最优性）。
 - 如果 $f(n) = g(n) + 0$ ，此时启发函数为0，退化为盲目搜索。
 - 如果在 f 中添加“一点点”启发，搜索效率就会提高；
 - 如果启发函数 $h(n)$ 过大，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。
- ◆ 因此需要对启发函数加以限制，这就是A*搜索算法。

A* 算法

◆ $h^*(n)$ 定义为从当前状态 n 到目标状态的最佳路径上的**实际代价**，即最小代价。

◆ 若启发函数 $h(n)$ 满足如下条件：

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

则可以证明当问题有解时，A算法一定能找到一个代价值最小的解，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**。

如何判断 $h(n) \leq h^*(n)$ 是否成立

- ◆ 一般来说， $h^*(n)$ 是未知的，那么如何判断 $h(n) \leq h^*(n)$ 是否成立呢？
- ◆ 这就要根据具体问题具体分析了。例如，所求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，可以采用当前节点到目标节点的欧氏距离作为启发函数 $h(n)$ 。
- ◆ 虽然不知道 $h^*(n)$ 是什么，但由于两点间直线距离最近，所以无论怎样定义 $h^*(n)$ ，肯定有 $h(n) \leq h^*(n)$ 。只要满足此限定条件，就可以用A*算法找到该问题的一条最优路径。
- ◆ 可证明：若问题有解，则利用A*算法一定能找到一个代价值最小的解，即最优解。因此，A*算法比A算法好。
- ◆ 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，
- ◆ 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* 算法解决八数码问题

◆ 要用A*算法解决八数码问题，需要一个启发式函数，通常有两个候选的启发式函数。

➤ $h1(n)$ = “错位数码牌” 的个数

➤ $h2(n)$ = 所有数码牌与其目标位置之间的曼哈顿距离之和。

➤ $h3(n)$ = $h2(n)$ + 颠倒两张相邻牌的固定步数

◆ $h^*(n)$ 的意义是：“把当前错位的数码牌移动到正确的位置上所需要的最小步数”。

◆ 可以证明：以上3个启发函数均满足： $h(n) \leq h^*(n)$ ，均为A*算法。

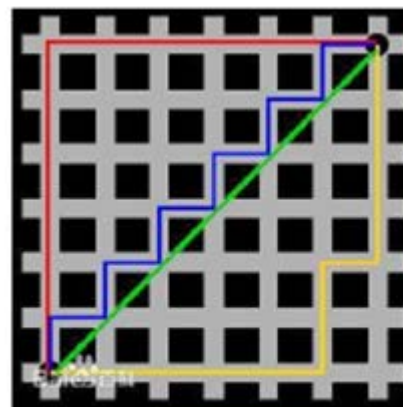
1	2	3
8		4
7	6	5

目标状态

2	8	3
1		4
7	6	5

初始状态 s

$h1(s)=3$



Manhattan distance

Distance:

“1” : 1

“2” : 1

“8” : 2

$h2(s)=4$

采用启发函数 $h_1(n)$ ，证明A算法为 A* 算法

- ◆ 令 $d(n)$ = 已移动数码牌的步数，即节点 n 在搜索树中的深度
- $w(n)$ = 节点 n 所表示的状态中“错位数码牌”的个数
- ◆ 将 $w(n)$ 个“错位”的数码牌放在其各自的目标位置上，至少需要移动 $w(n)$ 步。
- ◆ $h^*(n)$ 是节点 n 从当前状态移动到目标状态的最少需要的实际步数，
显然 $w(n) \leq h^*(n)$
- ◆ 以 $w(n)$ 作为启发式函数 $h(n)$ ，可以满足对 $h(n)$ 的上界的要求，
即有 $h(n) = w(n) \leq h^*(n)$.
- ◆ 因此，当选择 $w(n)$ 作为启发式函数解决八数码问题时，A算法就是A*算法。

思考题：采用启发函数 $h_2(n)$ ，证明A算法为 A* 算法。

(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

2	8	3
	1	4
7	6	5

(b)

$g=1$

$h1=3, h2=5$

1	2	3
8		4
7	6	5

(a) goal state

2	3	
1	8	4
7	6	5

(b)

$g=2, f=6$

$h1=4, h2=4$

1	2	3
8		4
7	6	5

(a) goal state

2		3
1	8	4
7	6	5

(b)

$g=1, f=4$

$h1=3, h2=3$

◆ $h1(n)$ = “错位数码牌” 的个数

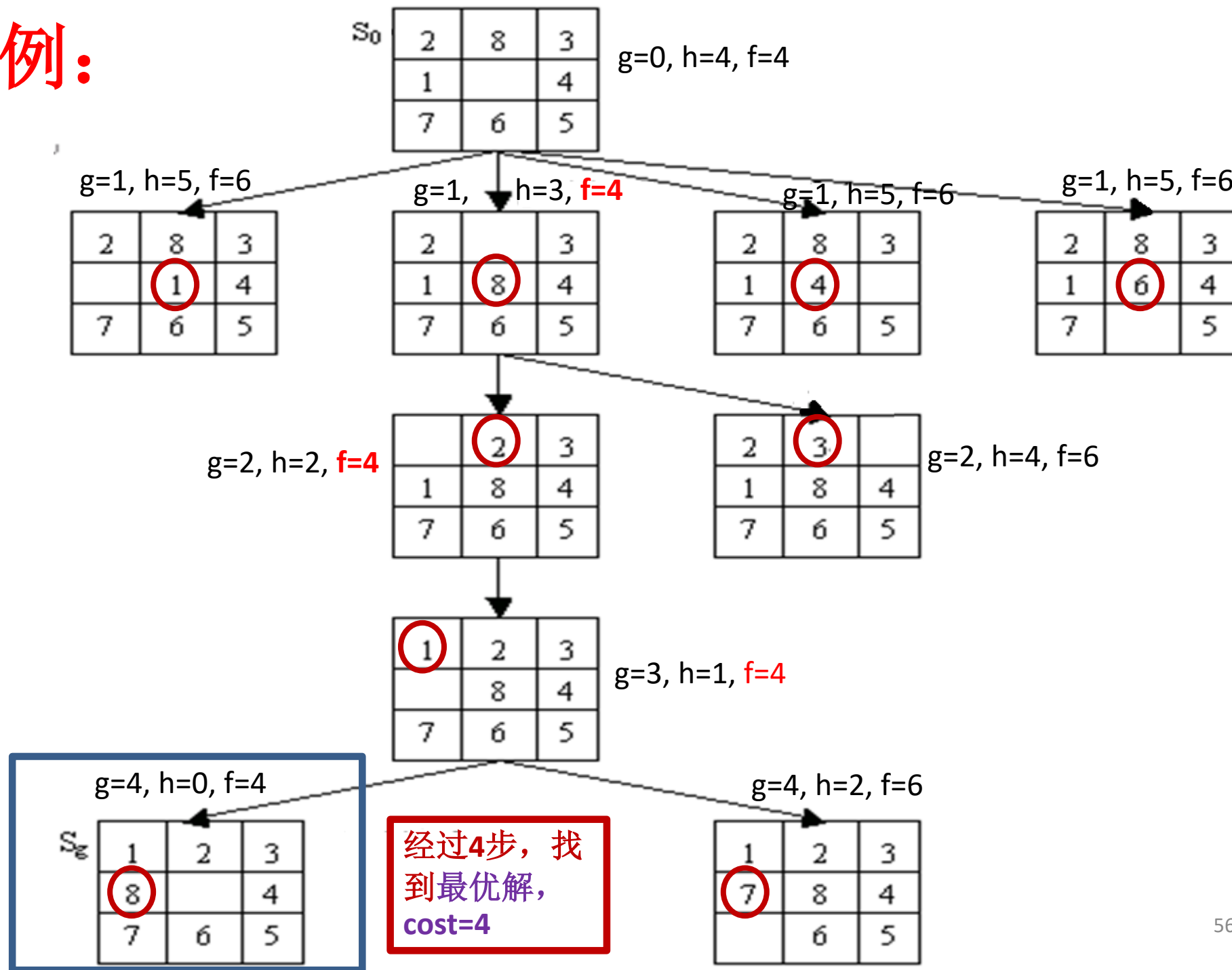
◆ $h2(n)$ = 所有数码牌与其目标位置之间的曼哈顿距离之和。

1	2	3
8		4
7	6	5

goal state

$h(n)$ =所有数码牌
与其目标位置之间的
曼哈顿距离之和

例:



用A* 算法解决 “修道士与野人渡河” 问题

在河的左岸有 K 个传教士、 K 个野人和1条船，传教士们想用这条船将所有成员都从河左岸运到河右岸去，但有下面条件和限制：

- (1) 所有传教士和野人都会划船。
- (2) 船的容量为 r ，即一次最多运送 r 个人过河。
- (3) 任何时刻，在河的两岸以及船上的野人数目不能超过传教士的数目，
否则野人将吃掉传教士。
- (4) 允许在河的某一岸或船上只有野人而没有传教士。
- (5) 野人会服从传教士的任何过河安排。

请采用A*算法搜索出一个确保全部成员安全过河的合理方案。

用A* 算法解决“修道士与野人渡河”问题



Step1 设计状态空间表示，采用三元组形式

表示一个状态，令 $S=(m, c, b)$ ，其中：

- m 为未过河的传教士人数， $m \in [0, K]$ ，已过河的传教士人数为 $K-m$ 。
- c 为未过河的野人数， $c \in [0, K]$ ，已过河的野人数为 $K-c$ 。
- b 为未过河的船数， $b \in [0, 1]$ ，已过河的船数为 $1-b$ 。
- 初始状态为 $S_0 = (K, K, 1)$ ，表示**全部成员及船**都在河的**左岸**，
目标状态为 $S_g = (0, 0, 0)$ ，表示全部成员及船都已到达了河**右岸**。

用A* 算法解决 “修道士与野人渡河” 问题

Step2 设计操作集合，即过河操作。

◆ 设计两类操作算子：

➤ L_{ij} 操作表示将船从左岸划向右岸，第一下标 i 表示船载的传教士人数，第二下标 j 表示船载的野人数；

➤ R_{ij} 操作表示将船从右岸划回左岸，下标的定义同前。

➤ 这两类操作需满足如下限制：(1) $1 \leq i + j \leq r$ ； (2) $i \neq 0$ 时， $i \geq j$ 。

◆ 假设 $K=5$ ， $r=3$ ，则合理的操作共有16种，其中

➤ 船从左岸到右岸的操作有： L_{01} 、 L_{02} 、 L_{03} 、 L_{10} 、 L_{11} 、 L_{20} 、 L_{21} 、 L_{30} ；

➤ 船从右岸到左岸的操作有： R_{01} 、 R_{02} 、 R_{03} 、 R_{10} 、 R_{11} 、 R_{20} 、 R_{21} 、 R_{30} 。

用A* 算法解决“修道士与野人渡河”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(3)--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：1个人固定作为船夫,每摆渡一次,只运1个人过河（即往返一趟，运送2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 “修道士与野人渡河” 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 $h(x) \leq h^*(x)$ 。

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： $h(n) \leq h^*(n)$

➤ 但此刻， $h^*(x)=1 < h(x)=2$ ，不满足A*算法的条件，

➤ 实际上： $h^*(x)$ 应该有一个上限： $h^*(x) \leq m+c$ ，即过河次数不高于2K次，因为一共才2K个人，摆渡次数不可能超过2K次，取 $h^*(n) = m+c$ 。

用A* 算法解决 “修道士与野人渡河” 问题

Step3 设计满足A* 算法的启发函数

分情况讨论 ($r=3$) :

- 假设船与人同在左岸, $b=1$ (船未过河), 状态为 $(m,c,1)$ 。
- 不考虑“野人会吃人”的约束条件, 当最后一次恰好3人同船过河时, 效率最高, 单独算1次摆渡。
- 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“修道士与野人渡河”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，即船与人在河的不同侧， $b=0$ ，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： $(m+c,0) \leftrightarrow (m+c+1, 1)$
- 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”，则第二种情况共需要运送 $(m+c-1) + 1 = m+c$ 次。

用A* 算法解决 “修道士与野人渡河” 问题

Step3 设计满足A* 算法的启发函数

两种情况结合，得到：

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

可写作： **$h(n) = m + c - 2b$**

此时， $h(n)=m+c-2b \leq h^*(n)=m+c$ ，满足A*算法的条件。

“修道士与野人渡河” 问题：
K=5, r=3, 11次摆渡

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

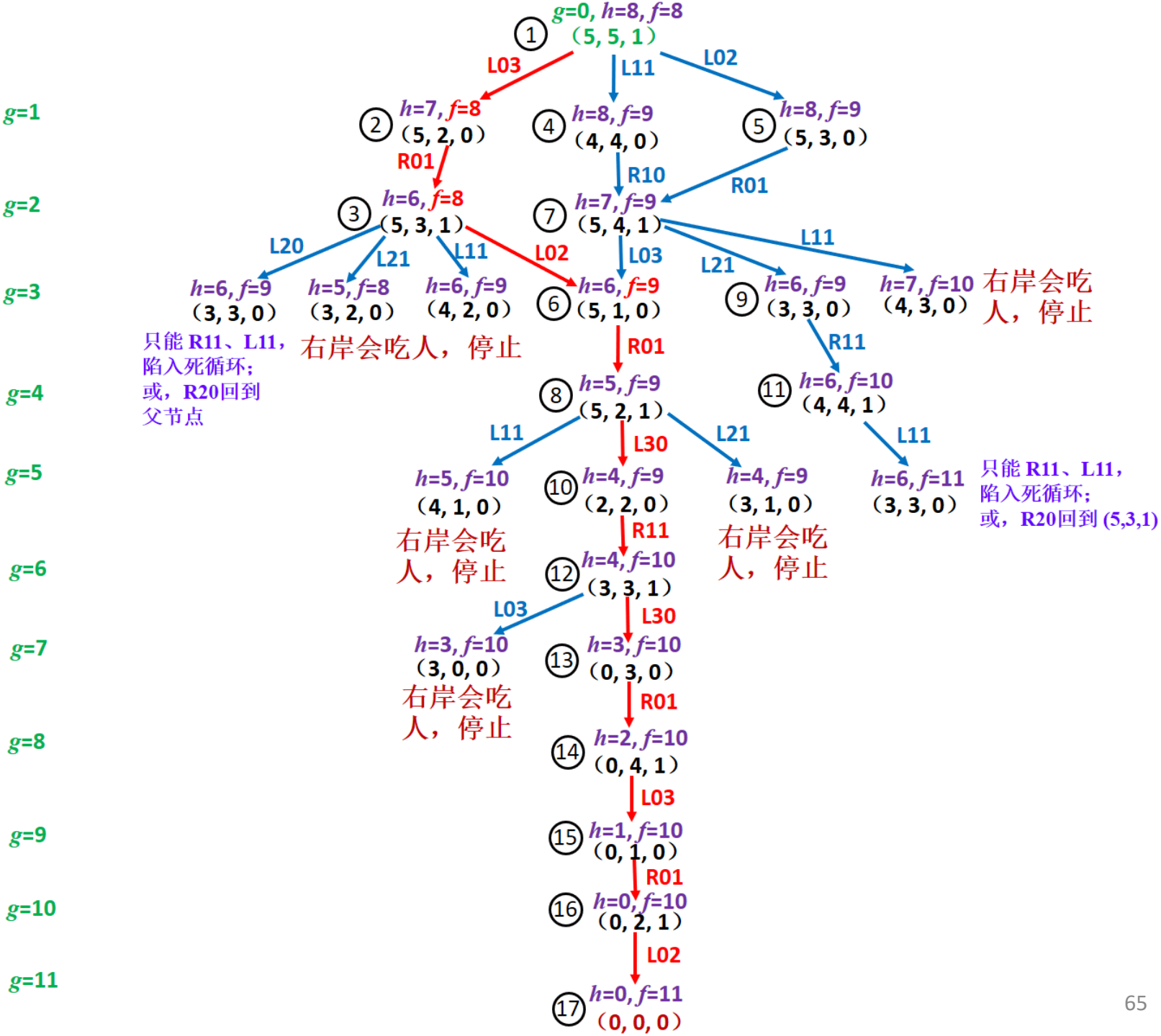
将待扩展节点存储于队列；

圆圈中的数字表示被扩展的顺序。

合法操作共有16种：

- 左->右： L01; L02; L03; L10;
L11; L20; L21; L30;
- 右->左： R01; R02; R03; R10;
R11; R20; R21; R30;

图3.7 采用A*算法解决传教士与野人渡河问题示例



3.4 局部搜索算法

- ◆ 前面介绍的搜索算法都在内存中保留一条或多条路径，记录路径中在每个节点处的扩展选择。
- ◆ 当找到目标时，从初始节点到达此目标的**路径**就是这个问题的**一个解**。
- ◆ 但在许多问题中，人们**并不关注到达目标的路径**。例如，在八皇后问题中，重要的是最终皇后在棋盘上的布局，而不是摆放皇后的先后次序。
- ◆ 许多重要的应用都具有这样的性质，例如集成电路设计、工厂场地布局等。
- ◆ 因此，考虑另外一类算法，**它不关心从初始状态到达目标状态的路径**，只对一个（当前状态）或多个（邻近）状态进行评价和修改，称为**局部搜索算法**。
- ◆ 局部搜索算法适用于那些**只关注解状态而不关注路径代价的问题**，该类算法从单个当前节点（而不是多条路径）出发，通常只移动到它的邻近状态。一般情况下，不保留搜索路径。

局部搜索

- ◆ **局部搜索的基本思想**：在搜索过程中，始终向着**离目标最接近的方向**搜索。
- ◆ **目标**可以是最大值，也可以是最小值
- ◆ 局部搜索算法有如下两个主要**优点**：
 - 使用很少的内存；
 - 在大的或无限（连续）状态空间中，能发现合理的解。
- ◆ 局部搜索算法：
 - 爬山法
 - 模拟退火法
 - 遗传算法

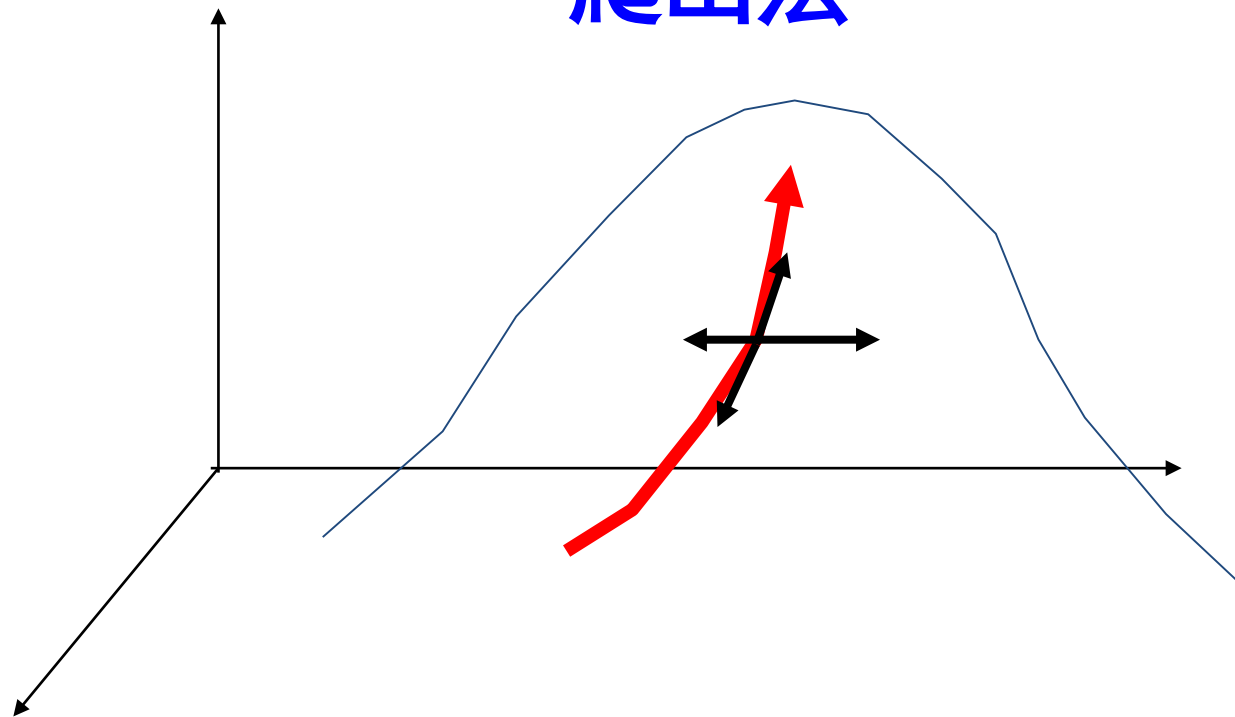
3.4.1 爬山法

- ◆ **爬山法是最基本的局部搜索技术**。最陡上升版的爬山法是简单的循环过程，在每个状态，都不断向值增加的方向持续移动，即**登高**。
- ◆ **爬山法的过程**如下：
 - 算法**从指定的初始状态开始**，或任意选择问题的一个初始状态。
 - 然后，在每一步，爬山法都将当前状态 n 与周围相邻节点的值进行比较，若当前节点值最大，则返回当前节点，作为最大值，即山峰最高点；
 - 否则，从 n 的所有相邻状态中**找到 n 的最佳邻接节点**，用以代替节点 n ，成为新的当前状态（此处，**最佳邻接节点是启发函数 h 值最低的相邻节点**）。
 - 重复上述过程，直到找到目标为止；或者无法找到进一步改善的状态，算法结束。

3.4.1 爬山法

- ◆ 在爬山法中，当前节点都会被它的最佳邻接节点所代替。
- ◆ 爬山法**不保存搜索树**，当前节点的数据结构**只记录当前状态和目标函数值**。
- ◆ 爬山法有时被称为**贪婪局部搜索**，因为它不考虑与当前状态不相邻的状态，总是在相邻节点（局部范围内）中选择状态**最好**的一个，也**不考虑这个最好状态是否是全局最优的**。
- ◆ 爬山法往往很有效，它能很快地朝着解（目标状态）的方向进展，因为它可以很容易地改善一个不良状态。

爬山法

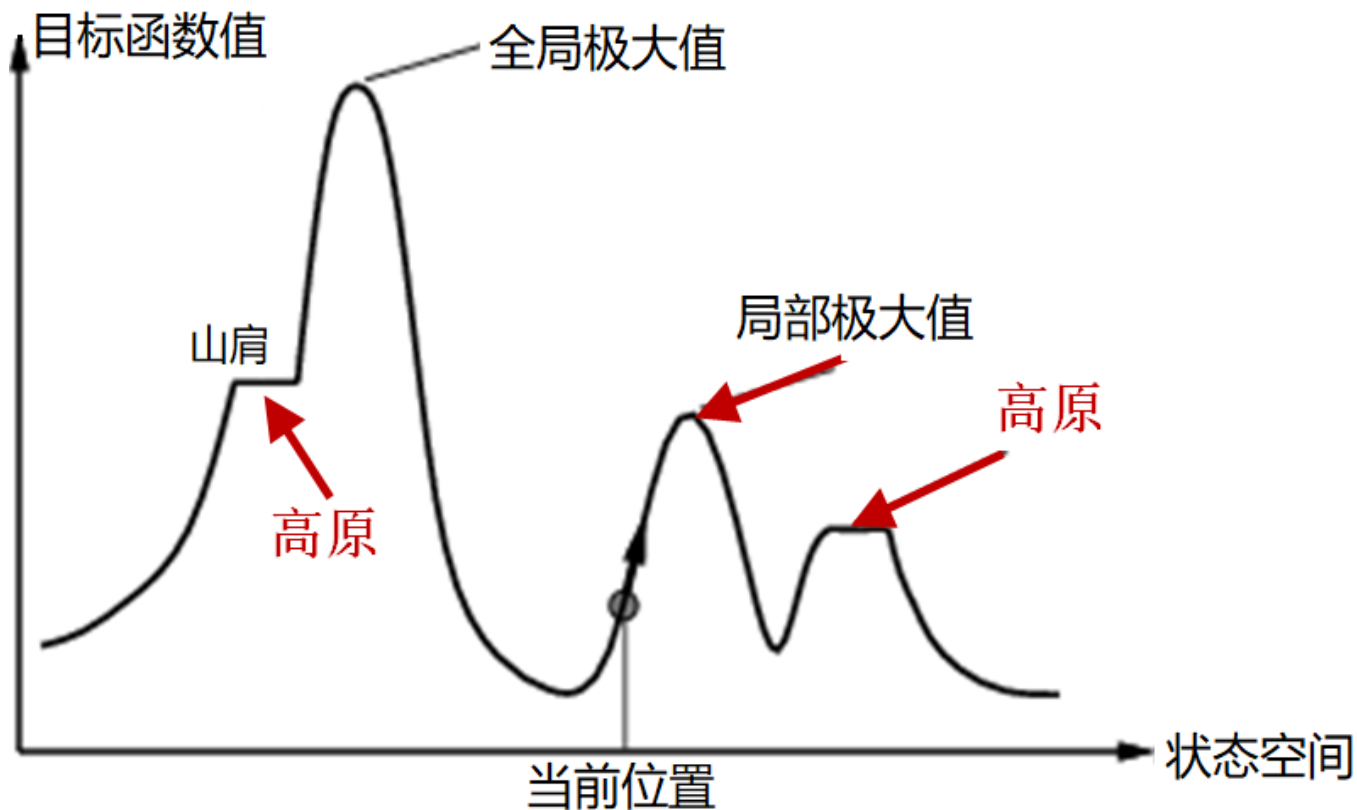


如果把山顶作为目标， $h(n)$ 表示当前位置 n 与山顶之间的高度差，则该算法相当于总是登向山顶。在单峰的条件下，必能到达山顶。

爬山法的三种困境（1）

在多个山峰的情况下，爬山法经常会陷入如下三种困境：

（1）**局部极大值**：是指一个比所有相邻状态值都要高、但却比全局最大值要小的状态。爬山法到达局部极大值附近，就会被拉向峰顶，然后就卡在局部极大值处，无处可走。



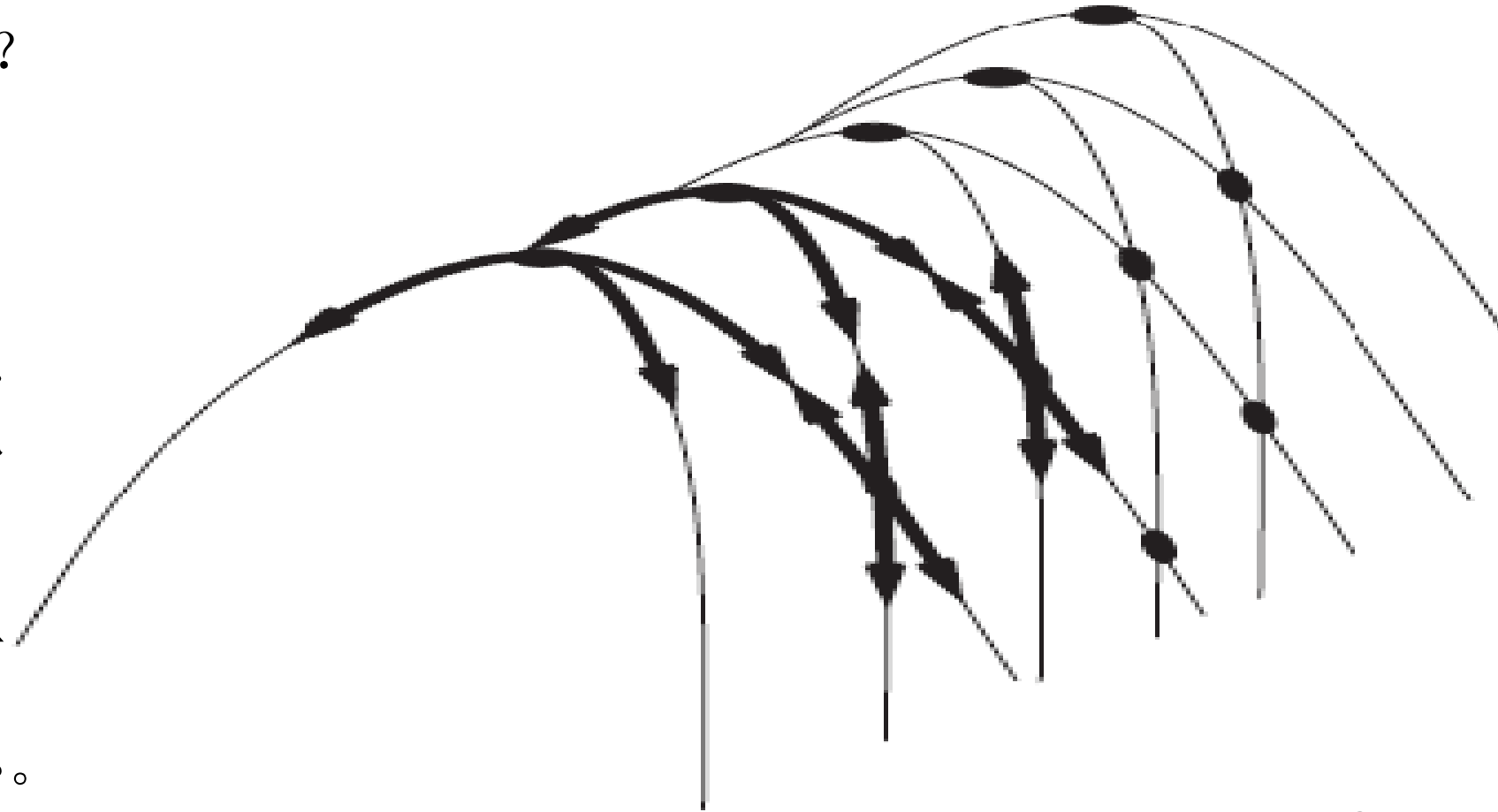
（2）**高原**：是一块平原区域，是平的局部极大值，不存在上山的出口；或者是山肩，从山肩还有可能取得进展（见图3.8）。爬山法在高原处可能会迷路。

爬山法的三种困境（2）

（3）**山脊**：是由一系列局部极大值构成的，形成了一个不直接相连的局部极大值序列。
在这样的情况下非常难爬行。

◆为什么山脊会使爬山法困难？

- 图中的状态（黑色圆点）叠加在从左到右上升的山脊上。从每个局部极大点出发，可能的行动都是指向**下山方向**的。
- 搜索可能会在山脊的两面来回震荡，前进步伐很小。



爬山法算法

function HILL-CLIMBING(*problem*) returns a state that is a local maximum

inputs: *problem*, a problem

local variables: *current*, a node

neighbor, a node

current \leftarrow MAKE-NODE(INITIAL-STATE[*problem*]);

loop do

neighbor \leftarrow a highest-valued successor of *current*; // 找到值最大的相邻节点

if VALUE[*neighbor*] \leq VALUE[*current*] then return STATE[*current*];

// 若当前节点的值大于所有的相邻节点，说明已到达最高峰，当前节点即目标，则返回当前节点。

current \leftarrow *neighbor*; // 否则，说明该相邻节点比当前节点好，用它代替当前节点，进行下一次循环。

注：Value 就是启发函数 h

采用爬山法解决 n 皇后问题

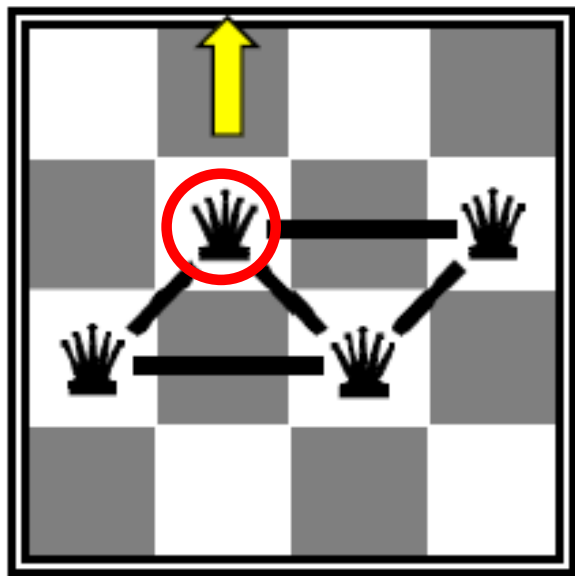
- ◆ 将 n 个皇后放在 $n \times n$ 的棋盘上。使得任意两个皇后不能互相攻击，即任意两个皇后都不在同一行、同一列或同一斜线上。
- ◆ 设任意两个皇后的坐标分别是 (i, j) 和 (k, l) ，
 - 为使得任意两个皇后不在同一行上，要求 $j \neq l$ ；
 - 为使得任意两个皇后不在同一列上，要求 $i \neq k$ ，也可以规定在棋盘的每一列上只能放置一个皇后；
 - 任意两个皇后在同一斜线上的充要条件是 $|i-k|=|j-l|$ ，即两个皇后的行号之差与列号之差的绝对值相等，则为使得任意两皇后不在同一斜线上，只需要求 $|i-k| \neq |j-l|$ 。

例：四皇后问题

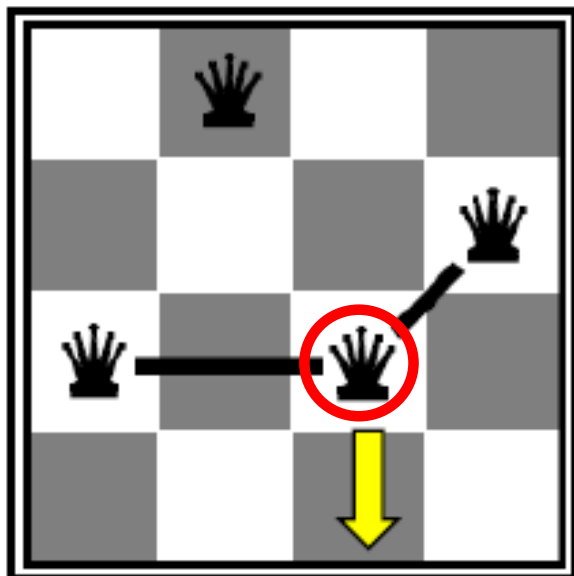
- ◆ 采用爬山法解决N皇后问题，首先需定义启发函数，令

$h(n)$ = 相互攻击的皇后对的数量

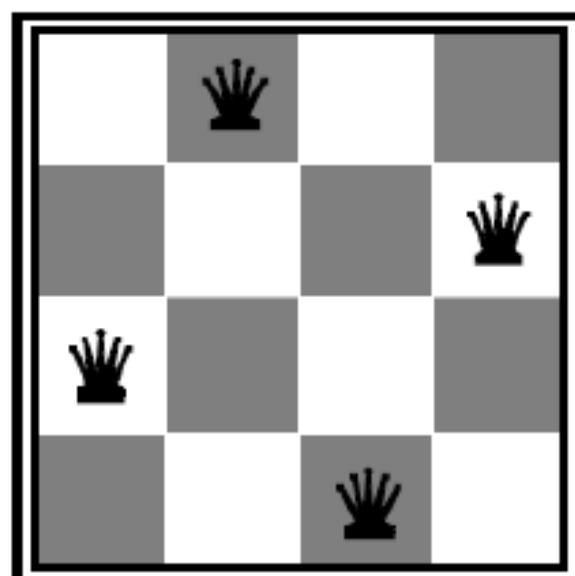
- ◆ 该函数的全局最小值是 $h=0$ ，即没有任意两个皇后是互相攻击的，仅在找到解时， h 值才会等于零。
- ◆ 如果有多个最佳后继，爬山算法通常会从一组最佳后继中随机选择一个。



(a) $h = 5$



(b) $h = 2$



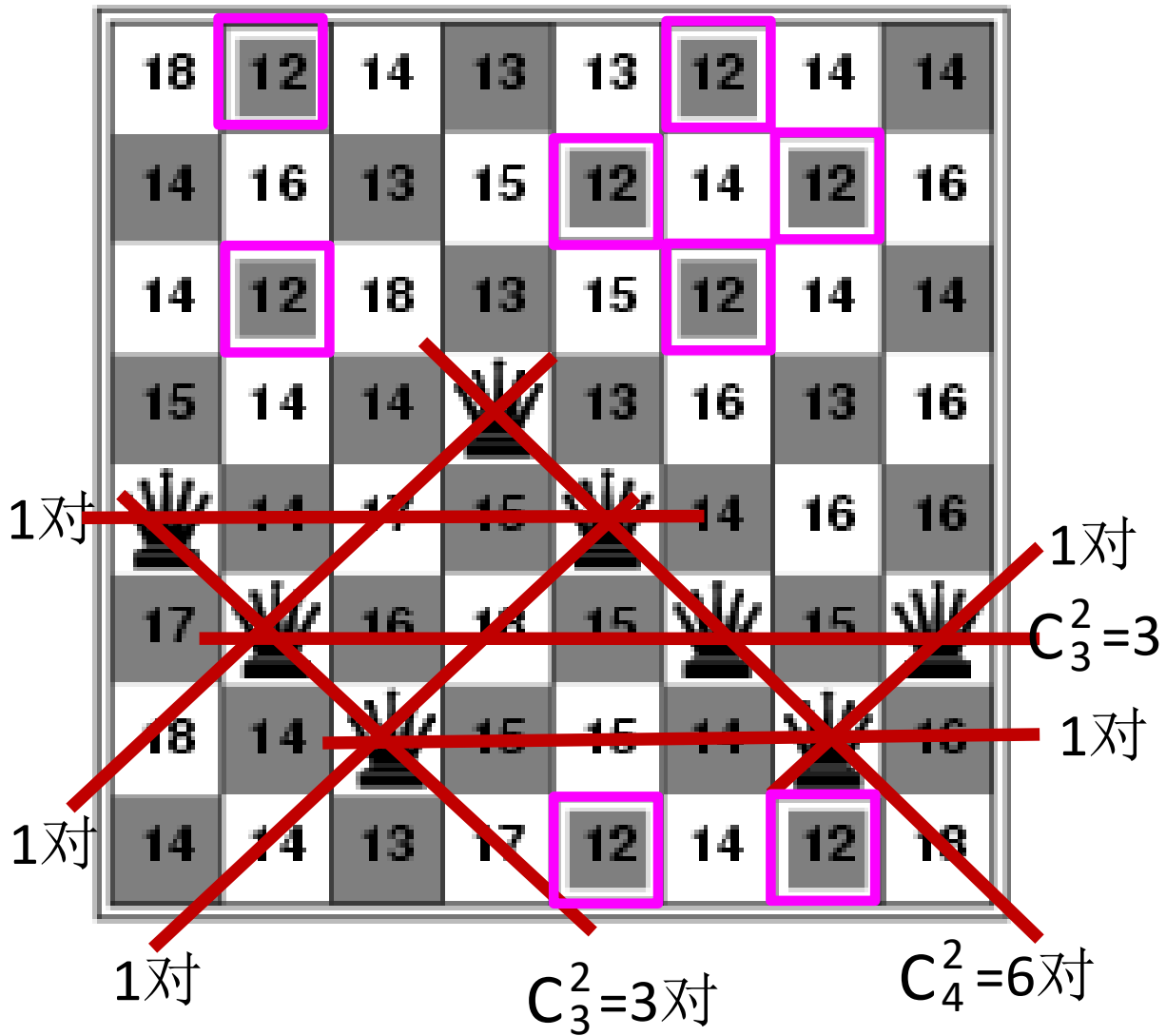
(c) $h = 0$ 解

采用爬山法解决 n 皇后问题的步骤

假设在初始状态中每列只摆放一个皇后。

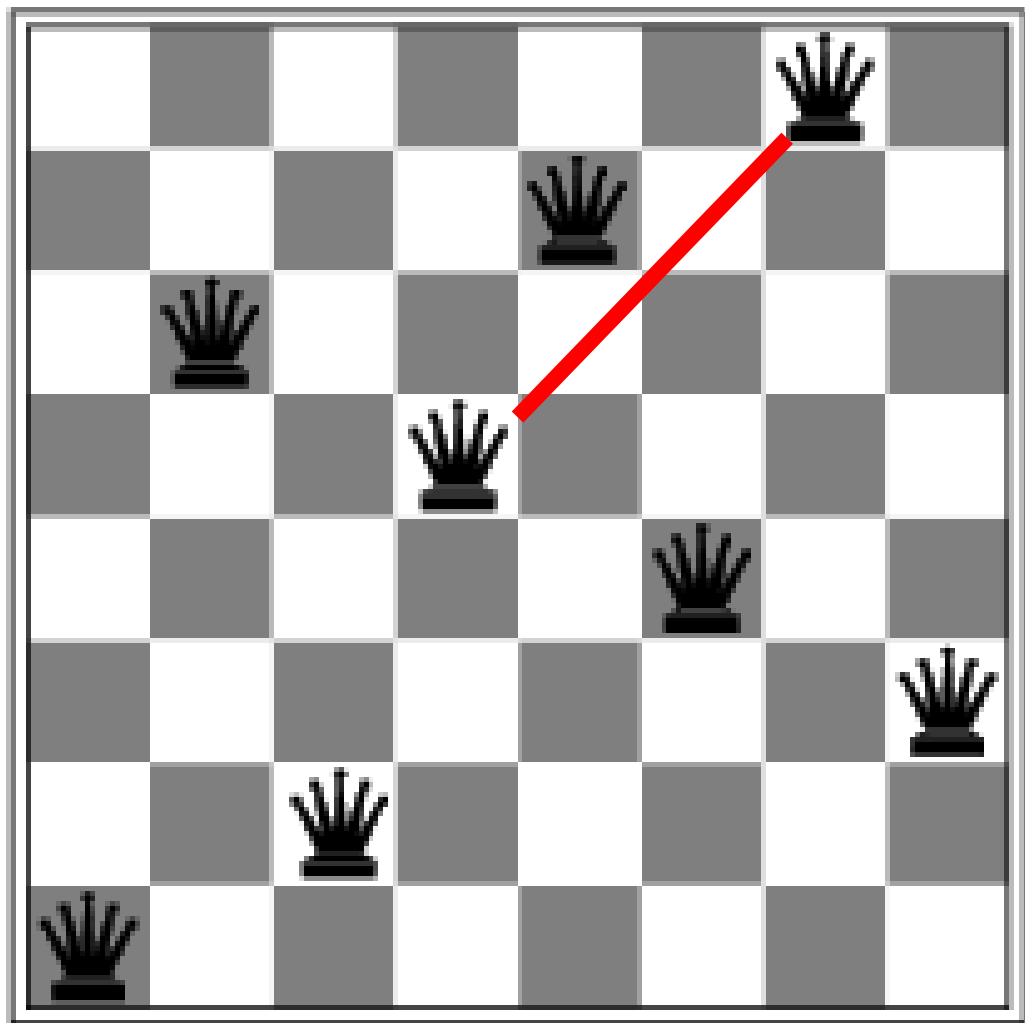
- (1) 针对当前状态，计算启发函数 h 的值。
- (2) 若 $h=0$ ，即找到一个解，算法结束。否则，计算各个方格里的 h 值。
- (3) 若无法找到比当前状态的 h 值更小的相邻状态，说明已陷入局部极值，找不到解，则算法结束。
否则，从若干个小于当前 h 值的最佳后继中随机挑选一个，将该列的皇后移到此位置，并转到步骤(2)。

例：已知八皇后的一个状态图，计算其启发函数h的值



- ◆ 当前状态的启发函数 h = 形成相互攻击的皇后对的数量, $h = 17$
- ◆ 每个方格中显示的数字表示: 将这一列中的皇后移到该方格后得到的后继的 h 值。
- ◆ 当前棋盘中最小的 h 值为12, 一共有8个, 均用粉色方框圈起来了, 表示是最佳移动。
- ◆ 如果有多个最佳移动, 即多个最小值, 爬山法会从中随机选择一个后继进行扩展。

例：八皇后问题的一个局部极小值



- ◆ 八皇后问题状态空间中的一个**局部极小值**：该状态的 $h=1$, 但是它的每个后继的 h 值都比它高.
- ◆ 此时，爬山法无法找到全局的最优解（即 $h=0$ ），即**爬山法是不完备的**。

图3.11八皇后问题陷入局部极小值的示例

爬山法

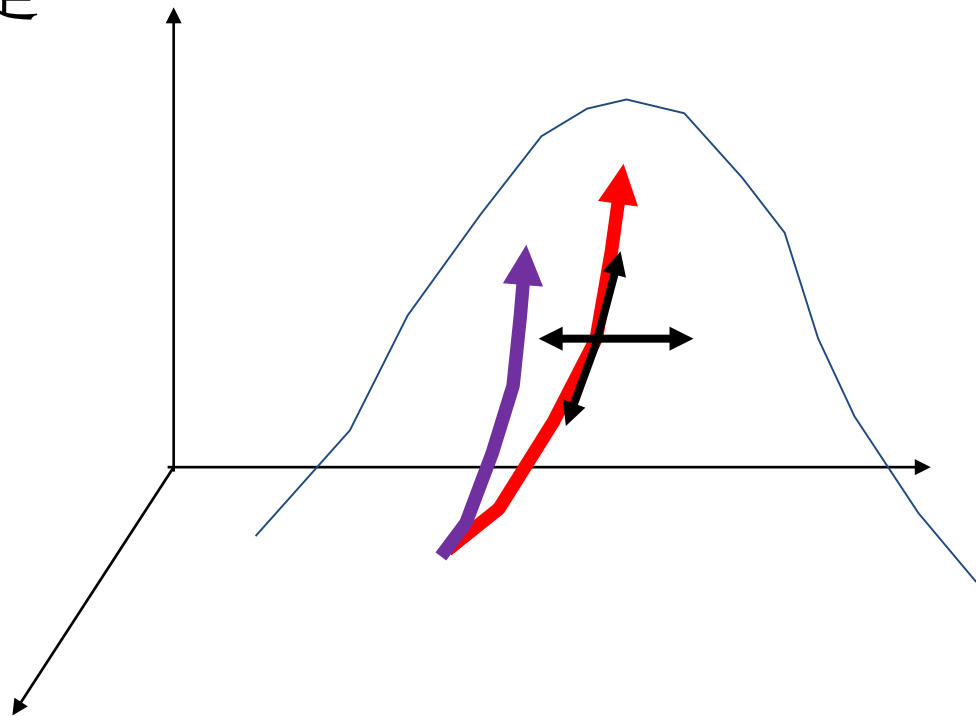
- ◆ 采用最陡上升版爬山法求解八皇后问题，从随机生成的初始状态开始搜索，有实验证明：在86%的情况下会被卡住，只有14%的问题实例能求得解。算法求解速度快，成功找到解的平均步数是4步，被卡住的平均步数是3步。
- ◆ 到现在为止，我们描述的爬山法是不完备的——它们经常会在存在目标的情况下，因为被局部极（大、小）值卡住而找不到目标。

随机爬山法

◆ **随机爬山法**是最陡上升版爬山法的变种，是一种局部贪心的最优算法。

◆ 该算法的主要思想是：

- 在向上移动的过程中，**随机地选择下一步（不一定是陡峭的，但一定是向上的）**，每个状态被选中的概率可能随向上移动陡峭程度的不同而发生变化。
- 与最陡上升算法相比，**收敛速度通常较慢**。
- **随机爬山法仍然不完备**，还会被局部极大值卡住。



随机重启爬山法

◆ 随机重启爬山法

- 随机生成一个初始状态，开始搜索，执行一系列这样的爬山搜索，直到找到目标为止；若找不到目标，则再随机生成一个初始状态，开始新一轮搜索，.....
- 随机重启爬山法依然不完备，但以它能以逼近1的概率接近完备，因为它最终将生成一个目标状态作为初始状态。
- 如果每次爬山搜索成功的概率为 p ，则重启需要的期望值是 $1/p$ ，即成功的概率越高，需要重启的概率越小。
- 对于八皇后问题，随机重启爬山法实际上是有效的。即使有300万个皇后，这个方法找到解的时间不超过1分钟。
- 爬山法成功与否严重依赖于状态空间地形图的形状：如果在图中几乎没有局部极大值和高原，随机重启爬山法会很快找到一个好的解。

3.4.2 模拟退火法

- ◆爬山法搜索从来不“下山”，即不会向值比当前节点低的（或代价高的）方向搜索，它肯定是不完备的，因为可能卡在局部极大值上。
- ◆与之相反，纯粹的随机行走是从后继集合中完全等概率地随机选取后继，即可能选择向比当前节点差的方向搜索。随机行走法是完备的，但是效率极低。
- ◆因此，将爬山法和随机行走法以某种方式结合，希望兼顾高效率和完备性。模拟退火就是这样的算法。

退火

- ◆ 在冶金中，**退火**是将材料加热后再经特定速率冷却，目的是增大晶粒的体积，并减少晶格中的缺陷。
- ◆ 为了更好地理解模拟退火，有一个形象的比喻：
 - 在高低不平的平面上有个乒乓球，我们希望乒乓球掉到最深的凹陷处，但它现在却处于某一个浅凹陷处，相当于局部极小值点。
 - 如果只允许乒乓球滚动，那么它只能停留在该浅凹陷中，出不来。
 - 如果晃动平面，就可以使乒乓球弹出浅凹陷处。
 - 关键是晃动的力度要适当，既能使得乒乓球从局部极小值处弹出来，又不能将它从全局最小值处弹出来。
 - 模拟退火的解决方法就是开始使劲摇晃（即先高温加热），然后慢慢降低摇晃的力度（即逐渐降温）。

模拟退火法

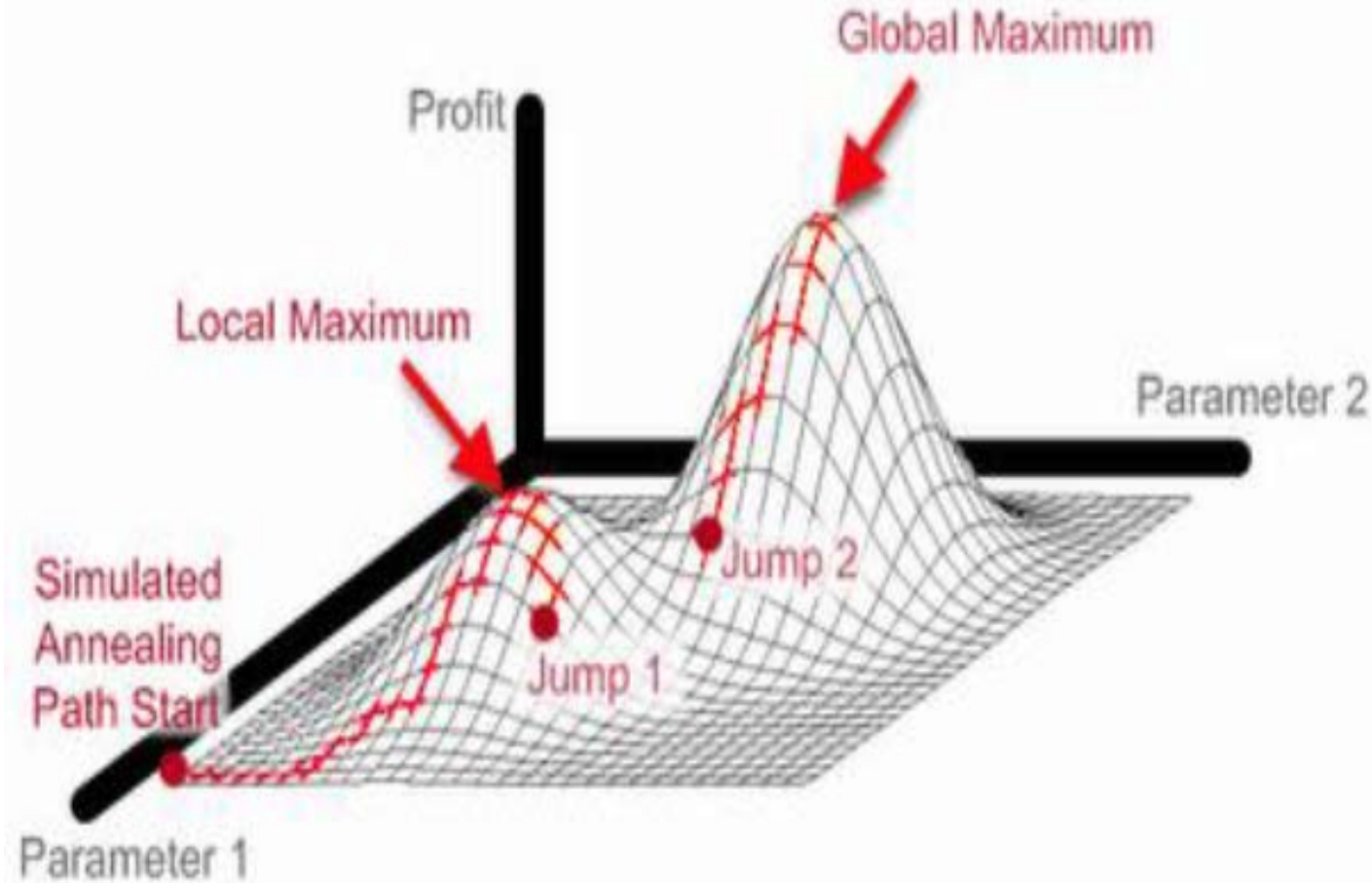
◆ 模拟退火是一种**逼近全局最优解**的概率方法，发表于1953年。

◆ 模拟退火算法是**允许“下山”**的随机爬山法。

◆ **基本思路：**

➤ 在退火初期，“下山”（即“变坏”）移动容易被采纳，以便摆脱局部极值；

➤ 随时间推移，“下山”的次数越来越少，即逐渐减少向“坏”的方向移动的频率。



模拟退火法的本质

- ◆ 模拟退火法本质上也是一种**贪心算法**，只不过是**以一定的概率来接受更差的状态**，这种**概率**会随着时间的推移变得**越来越小**；
- ◆ 其优点是可能会让算法**跳出局部最优解，最终找到全局最优解**。

模拟退火算法的过程

function SIMULATED-ANNEALING (*problem*, *schedule*) returns a solution state

inputs: *problem*, a problem

schedule, a mapping from time to “temperature”

current \leftarrow MAKE-NODE (*problem*.INITIAL-STATE)

FOR $t = 1$ TO ∞ DO

$T \leftarrow \text{schedule}(t)$ // T 是温度， t 是时间，*schedule*是将时间 t 映射到温度 T 的一个函数。
// $T=0$ 时，说明温度已经最低，退火结束，则返回当前节点。

if $T = 0$ then return *current*

next \leftarrow a randomly selected successor of *current* // 随机选取当前节点的一个后继
作为下一个节点。

$\Delta E \leftarrow \text{next.Value} - \text{current.Value}$ // 计算当前节点和下一个节点之间的能量差值 ΔE 。

if $\Delta E > 0$ then *current* \leftarrow *next* // 若 $\Delta E > 0$ ，说明是上山，状态在变好，则将下
一节点作为当前节点，进行下一次循环。

else *current* \leftarrow *next* only with probability $e^{\Delta E / (rT)}$

// 若 $\Delta E < 0$ ，说明是下山，状态在变坏。此时，可以以此概率将下一节点作为当前节点，进行下一次循环。

模拟退火法

- ◆ 模拟退火算法的内层循环与爬山法类似，只是它**没有选择最佳移动**，而是进行**随机移动**。
- ◆ 如果该移动能改善情况，该移动则被接受；否则，算法以某个小于1 的概率接受该（变坏的）移动。
- ◆ 如果移动导致状态“变坏”，**接受概率会呈指数级下降**——根据**能量差值 ΔE** 判断。
- ◆ 这个**概率也随“温度” T 的降低而下降**：开始 T 高的时候，可能允许“坏的”移动；当 T 降低时，则不可能允许“坏的”移动。
- ◆ 如果调度让温度 T 下降得足够慢，算法找到全局最优解的概率接近于1。
- ◆ 采用模拟退火算法**求解八皇后问题**，关键是设计启发函数。令启发函数

$h(n)$ =相互攻击的皇后对的数量

$h(n)$ 值越小，说明状态越好。用 $h(n)$ 代替 ΔE 计算公式中的Value即可。

3.4.3 遗传算法

- ◆ 1960年代末，受达尔文“物竞天择，适者生存”进化论思想的启发，提出了**模拟自然选择和遗传学机理的生物进化过程**的计算模型，通过模仿自然进化过程来搜索复杂问题的最优解，这就是求解优化问题的**遗传算法**。
- ◆ 遗传算法是一种**启发式随机搜索**算法，它通过数学的方式，利用计算机仿真运算，模仿生物遗传和进化过程中的染色体基因**选择、交叉、变异**机理，来完成自适应搜索问题最优解的过程。
- ◆ 在遗传算法中，后继节点是由两个父辈状态组合而成的，而不是对单一状态修改而得到的。其处理过程是**有性繁殖**，而**不是无性繁殖**。
- ◆ 遗传算法借鉴生物进化中“适者生存”的理论，定义了一些术语。

3.4.3.1 遗传算法的基本概念

- (1) **个体** (Individual)：就是遗传算法要处理的**染色体** (Chromosome)，组成染色体的元素称为**基因**。染色体中的每一位就是一个基因，基因的位置称为**基因座**，基因的取值称为**等位基因**。例如：**11011**为一个染色体或个体，每一位上的0或1表示基因。
- (2) **种群** (Population)：是由若干个个体（即染色体）组成的集合。一个种群中个体的个数称为该**种群的规模**。
- 种群规模会影响遗传优化的结果和效率。大的种群中含有丰富的个体模式，可以改进遗传算法的搜索质量，防止**早熟收敛**（算法较早地收敛于局部最优解，称为**早熟收敛**）。
 - 但大的种群也增加了个体适应度函数的计算量，从而降低了收敛速度。
 - 一般种群规模选取在[20, 100]区间的值。

3.4.3.1 遗传算法中的基本概念

(3) **适应度** (Fitness) **函数**: 适应度是指个体对环境的适应程度。

- 在优化问题中, 用一个估计函数来度量个体的适应度, 这个函数称为**适应度函数**。
- 其函数值是遗传算法实现**优胜劣汰**的主要依据。
- 个体的值越大, 说明该个体的状态越好, 竞争能力越强, 被选择参与遗传操作来产生新个体的可能性就越大, 以此体现生物遗传中适者生存的原理。

3.4.3.2 编码

(1) 二进制编码

- 二进制编码就是用一个二进制的字符串表示一个个体，其中每个0或1为等位基因（即基因的取值）。染色体上由若干个基因构成的一个有效信息段称为基因组。

例如：一个染色体11011，0/1表示基因，前3个基因就是一个基因组110。

- 二进制编码使得交叉、变异等遗传操作易于实现，但在求解高维优化问题时，二进制编码串会很长，将导致遗传算法的搜索效率很低。

(2) 实数编码

为了克服二进制编码的缺点，在问题变量是实向量的情况下，可直接采用十进制编码，即为实数编码。实数编码就是用一个十进制的字符串表示一个个体，然后在实数空间上进行遗传操作。近年来，遗传算法在求解高维或复杂优化问题时，一般都采用实数编码。

3.4.3.2 编码

(3) 有序编码

- 有序编码也叫序列编码、排列编码，是针对一些特殊问题的特定编码方式。**该编码方式排列有限集合内的元素。**
- 若集合内包含 m 个元素，则存在 $m!$ 种排列方法，当 m 不大时， $m!$ 也不会太大，可采用穷举法解决问题。当 m 比较大时， $m!$ 就会非常大，穷举法失效，遗传算法在解决这类问题上具有优势。
- 针对很多组合优化问题，目标函数的值不仅与表示解的字符串中各字符的值有关，而且与其所在字符串中的位置有关。这样的问题称为**有序问题**。
- 若目标函数的值只与表示解的字符串中各字符的位置有关，而与具体的字符值无关，则称为**纯有序问题**，如八皇后问题（ **$\{1, \dots, 8\}$ 集合中的元素**）。
- 有序编码的优点是使问题简洁，易于理解，编码自然、合理。

3.4.3.3 种群设定

- ◆ 由于遗传算法是对种群进行操作的，因此需要为遗传操作构造一个由若干个个体组成的初始种群。
- ◆ 初始种群中的个体一般是随机产生的。
- ◆ 假设设定种群规模为 M ，首先随机生成一定数目（通常为 $2M$ ）的个体，然后从中挑选较好的 M 个个体，构成初始种群。

3.4.3.4 适应度函数的设计

- ◆ 适应度函数的设计直接影响遗传算法的收敛速度以及能否找到最优解，因为遗传算法仅根据适应度函数来评价种群中每个个体适应性的优劣。
- ◆ 在遗传算法中，适应度函数值规定为非负，并且在任何情况下都希望其值越大越好。
- ◆ 在具体应用中，适应度函数的设计要结合待求解问题本身的要求而定。一般而言，适应度函数是由待求解优化问题的目标函数变换得到的。
- ◆ 若问题的目标函数 $f(x)$ 为最大化问题，则适应度函数可以取为 $\text{Fit}(f(x)) = f(x)$
- ◆ 若问题的目标函数 $f(x)$ 为最小化问题，则适应度函数可以取为 $\text{Fit}(f(x)) = 1/f(x)$

3.4.3.5 遗传操作

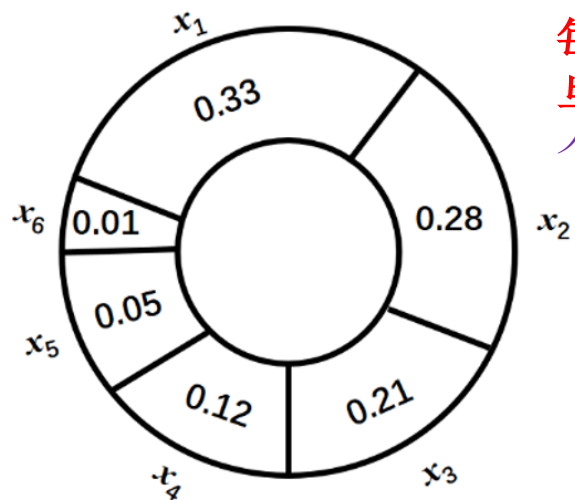
- ◆ 遗传操作（Genetic Operator）：可作用于种群，用于产生新的种群。
- ◆ 标准的遗传操作包括以下三种基本形式：
 - 选择（Selection）
 - 交叉（Crossover）
 - 变异（Mutation）

(1) 选择操作

- ◆ **选择（Selection）操作**，也称作复制（Reproduction），它是从当前种群中按照一定概率选出的优良个体，使它们有机会作为父代繁殖下一代。
- ◆ 选择操作的**目的**是使种群优胜劣汰、不断进化，并且提高种群的收敛速度和搜索效率。
- ◆ **根据个体的适应度值来判断其优劣**，适应度值越高，越具有优良性，该个体被选择的机会就越大，显然这一操作借鉴了达尔文“**适者生存**”的进化原则。
- ◆ 实现选择操作的方法有很多，不同的选择策略对算法的性能也有较大的影响。最常用的选择方法称为“**轮盘赌**”方法。

	个体编号					
	x1	x2	x3	x4	x5	x6
适应度值	3.6	3.1	2.3	1.35	0.6	0.12
选择概率	0.33	0.28	0.21	0.12	0.05	0.01
累计概率	0.33	0.61	0.82	0.94	0.99	1.00

11.07



每个个体被选择的概率与其适应度值成正比。
个体被选择的概率为

$$P_i = \frac{f_i}{\sum_{j=1}^M f_j}$$

(2) 交叉操作

- ◆ 交叉（Crossover）算子，也称为**重组**操作，是遗传算法中的核心操作。
- ◆ 交叉是分别用**两个父代个体的部分基因片段**重组为新的子代的操作，使父代的优良特征能传递给子代，并产生新的特性。
- ◆ 由交叉操作得到的**子代个体，构成了新种群**，其中个体适应度的平均值和最大值均比父代有明显提高。
- ◆ 交叉是遗传算法中获得比父代更优秀的个体的最重要手段。
- ◆ 最常见的是**单点交叉**。

例如，对于两个父代染色体：x1=**10110011**、x2=**01100101**，随机产生交配位“3”，则交叉产生两个子代染色体：y1=**10100101**、y2=**01110011**。

(2) 交叉操作

- ◆ 在进化过程中，交叉并非百分之百地发生，而是以某一概率发生，这个概率称为**交叉概率 P_c** 。一般 P_c 在 $[0.6, 1.00]$ 取值，实验表明 P_c 取0.7左右时，搜索结果比较理想。
- ◆ P_c 控制着交叉操作的应用频率，在每代种群中，都需要对 $M \times P_c$ 个个体的染色体结构进行交叉操作。
- ◆ 交叉概率越大，在种群中引入新的染色体结构的速度就越快，但优良基因结构遭到破坏的可能性也相应增大。
- ◆ 若交叉概率太低，则可能导致早熟收敛。
- ◆ 当交叉操作产生的**后代的适应度不再比其父代高、且未找到全局最优解**时，算法会较早地收敛于局部最优解，称为**早熟收敛**。
- ◆ **早熟收敛的根源**是发生了有效等位基因的缺失，即缺失了最优解位串上的等位基因。
- ◆ 若想要跳出局部最优，只有进行变异操作，才能改变这种情况。

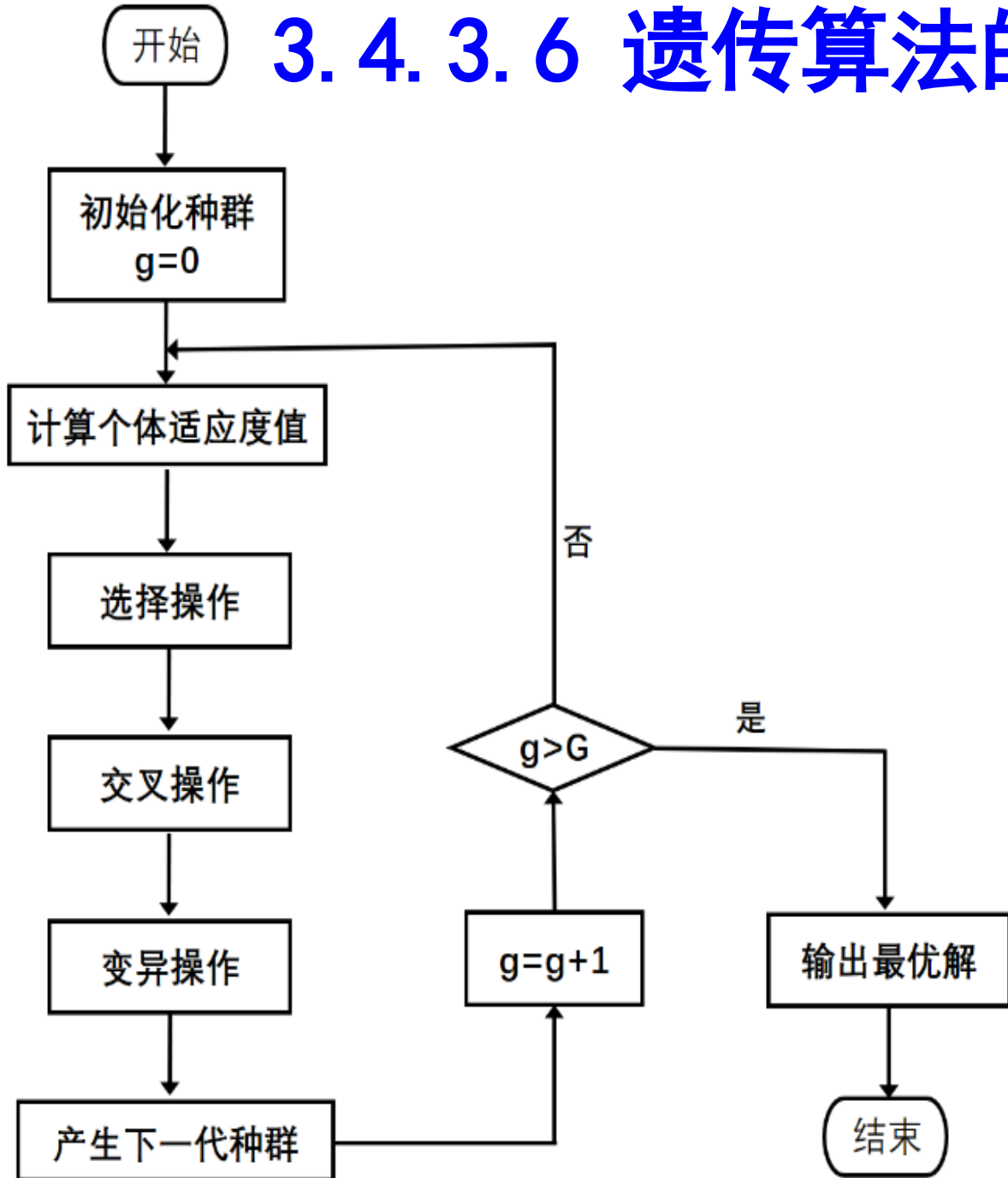
(3) 变异操作

- ◆ **变异**是随机改变个体编码中某些位的基因值的操作，从而产生新一代的个体。
- ◆ 变异操作是**按位进行**的，即对某一位的内容进行变异。
- ◆ 变异的主要**目的**是保持种群的**多样性**，对选择、交叉过程中可能丢失的某些遗传基因进行修复和补充，**当发生早熟收敛时，可利用变异跳出局部最优解的陷阱。**
- ◆ 变异操作不仅可以保证实现搜索的目标，而且可以提高搜索的效率。
- ◆ 变异操作是在交叉操作后进行的，在子代某个个体中**随机选择基因座，然后按变异概率 P_m 进行变异。**
- ◆ P_m 不能取很大的值。若变异概率太大，则种群中重要的、单一的基因可能会丢失，且使遗传算法趋于纯粹的随机搜索。一般 P_m 在 $[0.001, 0.01]$ 取值。
- ◆ 变异与选择、交叉结合在一起，保证了遗传算法的有效性，使遗传算法具有局部的随机搜索能力，同时使得遗传算法保持种群的多样性，以防止早熟收敛。

5种变异方法

- (1) **位点变异**。在个体位串中随机择选一个或多个基因座，并对这些基因座的基因值按变异概率 P_m 作变动。对于以二进制位串表示的染色体而言，若某位原来的值为1，变异操作就是将其值变为0，反之亦然。对于实数编码，可将被选择的基因座的值变为按概率选择的其他基因值。为了消除非法性，再将其他基因所在的基因座上的基因变为被选择的基因。
- (2) **互换变异**。随机择选某个个体的两个基因进行简单互换。
- (3) **插入变异**。在某个个体中随机选择一个基因，然后将此基因插入随机选择的基因座。
- (4) **移动变异**。在某个个体中随机择选一个基因，向左或者向右移动随机的位数。
- (5) **逆转变异**。在某个个体中随机选择两点（称为逆转点），然后将两点之间的基因值以逆序插入原位置。

3.4.3.6 遗传算法的基本过程



- (1) 编码：根据所要求解的具体问题确定合适的表示个体的编码方法。
- (2) 随机产生 M 个个体，构成一个初始种群 $P(0)$ 。从初始种群开始迭代，令 g 表示进化的代数，初始化 $g=0$ ，设置进化的最大代数为 G ，进化后的各代种群表示为 $P(g)$ ， $g \in [0, G]$ 。
- (3) 若 $g > G$ ，则输出当前种群中具有最大适应度值的个体，作为最优解，算法结束。
- (4)-(7) 计算适应度值、选择、交叉、变异；
- (8) 由种群 $P(g)$ 得到下一代种群 $P(g+1)$ ，作为当前种群，令 $g=g+1$ ，转到步骤(3)。

3.4.3.7 采用遗传算法求解八皇后问题

采用**有序编码表示每个棋盘布局**：某八皇后状态需要指明8个皇后的位置，每列有一个皇后，每个状态可用8个数字表示，每个数字表示该列中皇后所在的行号，其值在1到8之间。令：***fitness*** = 不相互攻击的皇后对的数目。最优解的适应度是28。

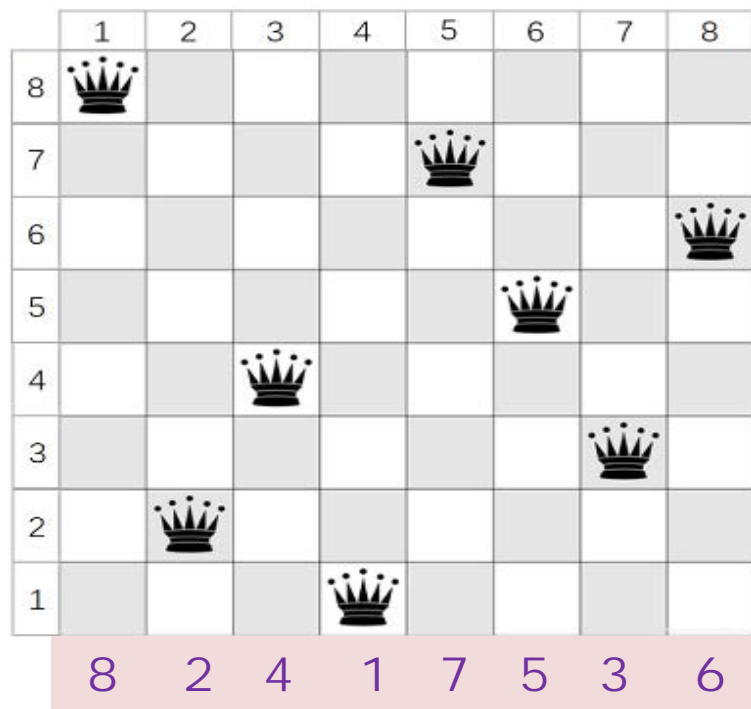


图3.15八皇后问题的一个解， $fitness=28$

$$C_8^2=28$$

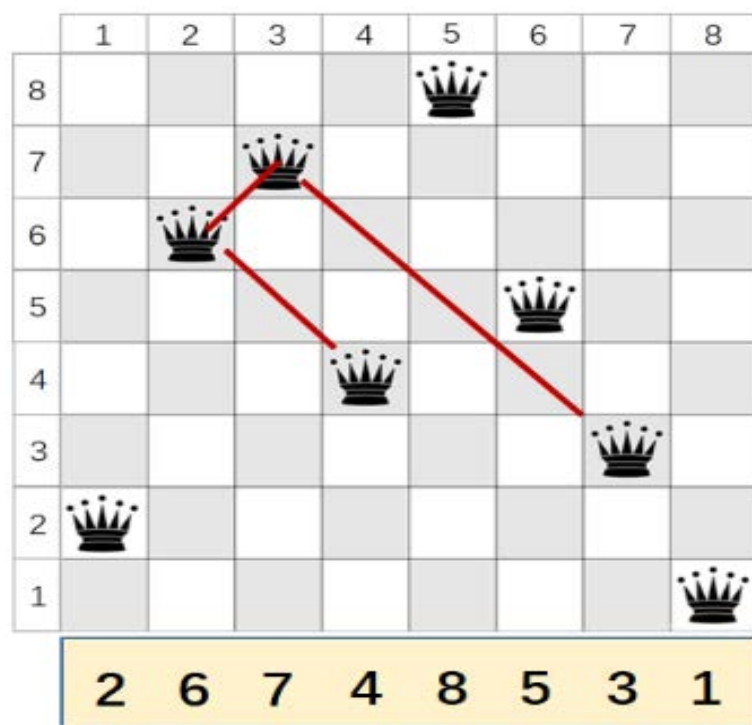


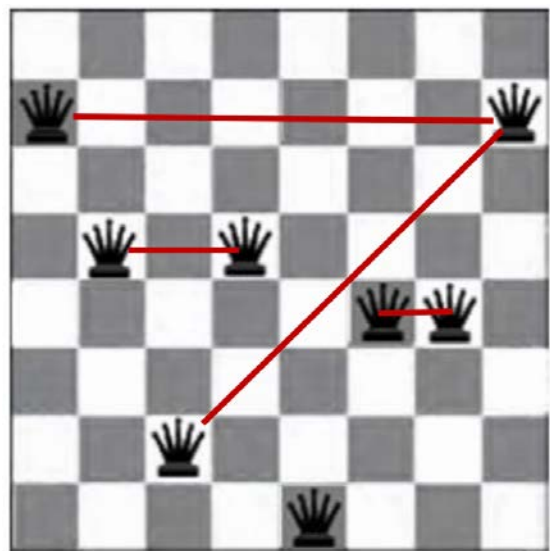
图3.16八皇后问题的一个布局， $fitness=25$

◆ 当前状态的 $h = 3$

◆ 适应度 = $28 - 3 = 25$

设计适应度函数

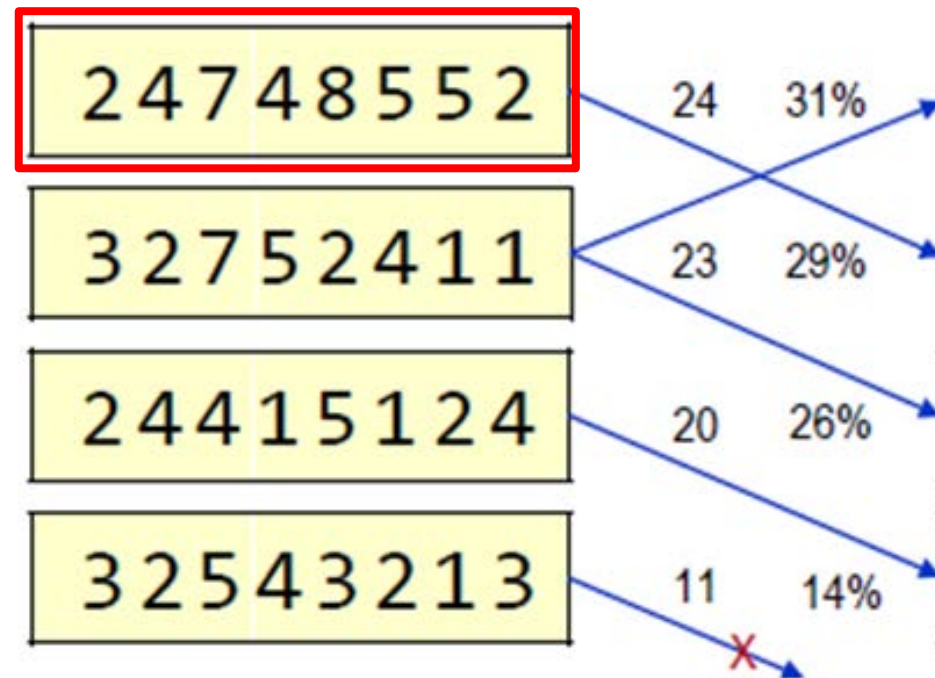
- ◆ 右图中，这四个状态的适应度分别是24 (4对攻击)、23 (5对攻击)、20 和11。



2	4	7	4	8	5	5	2
---	---	---	---	---	---	---	---

◆ 当前状态的 $h = 4$

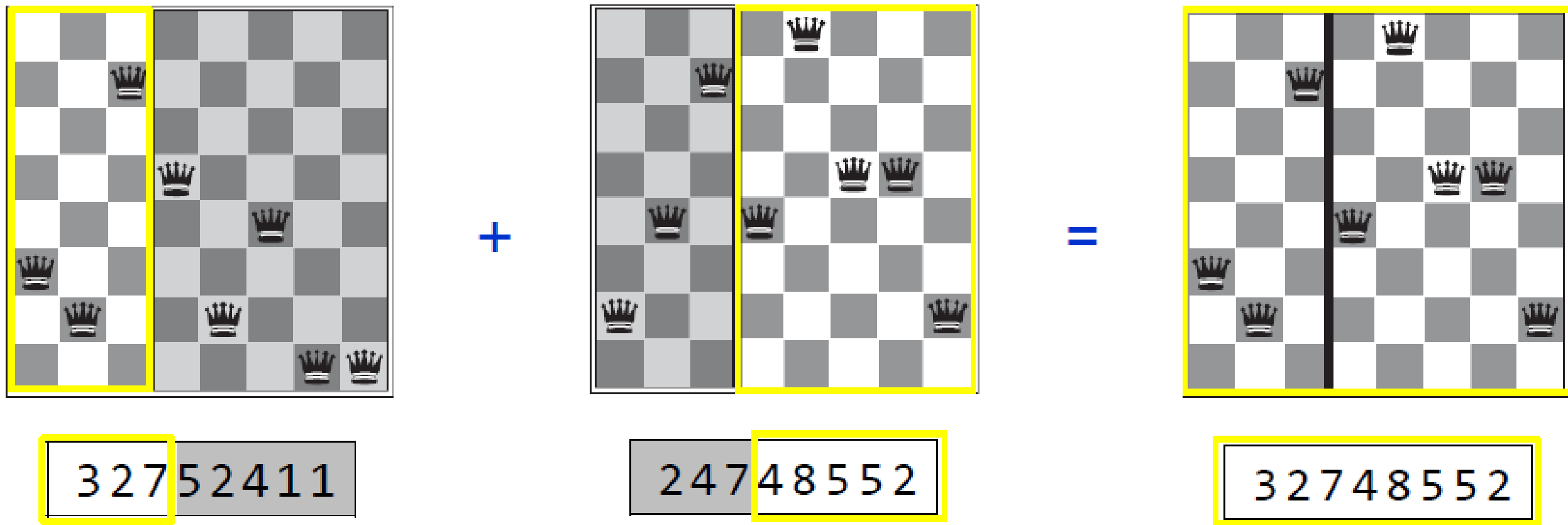
◆ 适应度 = $28 - 4 = 24$



- ◆ 在这个特定的遗传算法实现中，被选择进行繁殖的概率直接与个体的适应度成正比，其百分比标在旁边。（即适应度越高，越好。）

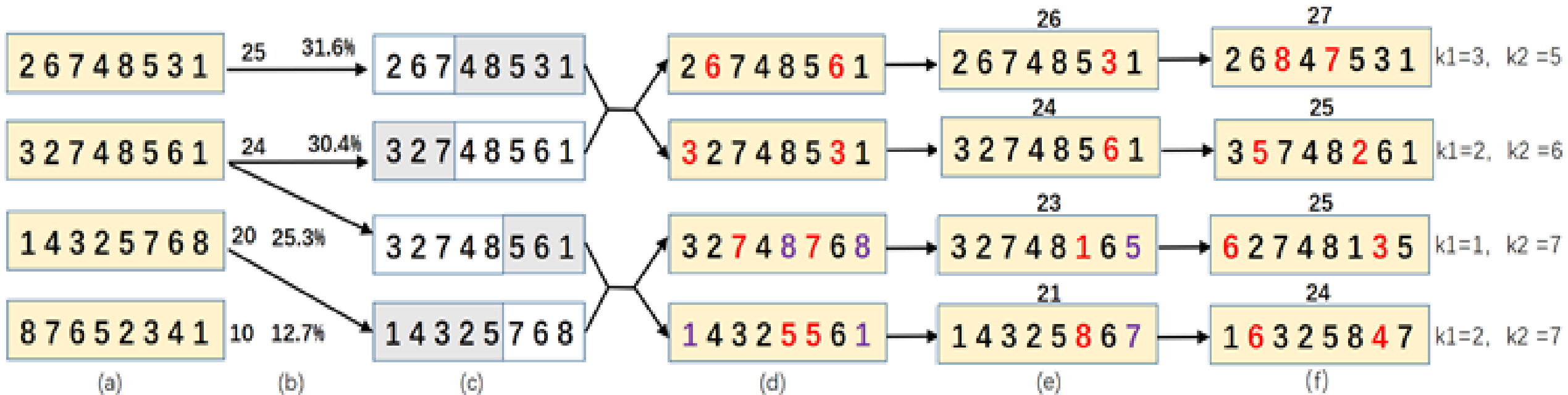
第一个状态被选择的概率为： $24 / (24 + 23 + 20 + 11) = 24 / 78 = 30.8\%$

交叉示意图



这三个8皇后的状态分别对应于“选择”中的两个父辈和“杂交”中它们的后代。

阴影的若干列在杂交步骤中被丢掉，而无阴影的若干列则被保留下来。



(a)为初始种群 (b)计算适应度函数 (c)选择 (d)交叉产生的后代 (e)解决冲突 (f)变异

- ◆ 由于子代个体中存在重复的基因，需要**解决冲突 (e)**，即把子代中重复的基因按照某种映射关系（例如， $1 \leftrightarrow 4 \leftrightarrow 7$ ， $3 \leftrightarrow 6$ ， $5 \leftrightarrow 8$ ）处理成不含重复基因的个体。
- ◆ **图 (f)** 中每个位置都会按照某个小的独立概率随机**变异**。
- ◆ 在变异过程中，需要保证不造成任何基因的缺失和重复，即每个编码都是一个**没有重复基因值的8位数字的排列**。
- ◆ 例如，采用**交换变异**方法，随机生成两个随机数 $k1$ 和 $k2$ ($1 \leq k1 < k2 \leq 8$)，交换该个体编码中第 $k1$ 位和第 $k2$ 位上的值，完成变异。

The Genetic Algorithm

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals // 输入: 种群, 是若干个体的集合
        FITNESS-FN, a function that measures the fitness of an individual
repeat // 个体适应度函数
    new_population  $\leftarrow$  empty set // 初始时, 设新种群为空集
    for  $i = 1$  to SIZE(population) do // 取每一个个体
         $x \leftarrow$  RANDOM-SELECTION(population, FITNESS-FN) // 计算个体的适应度, 按
         $y \leftarrow$  RANDOM-SELECTION(population, FITNESS-FN) // 选择算子选择2个个体
        child  $\leftarrow$  REPRODUCE( $x, y$ ) // 将随机选择的2个个体进行杂交, 生成新个体child
        if (small random probability) then child  $\leftarrow$  MUTATE(child)
        add child to new_population // 若随机概率小, 则将新个体child进行变异。
        // 将新个体child加入新种群。
    population  $\leftarrow$  new_population // 用新种群替换原来的种群。
until some individual is fit enough, or enough time has elapsed // 某些新个体足够
// 健康, 或超时
return the best individual in population, according to FITNESS-FN // 根据适应度,
// 返回最佳个体
```

第3章小结

◆ 搜索的基本概念:

搜索、搜索策略、搜索技术、解、最优解、完备性、最优性

◆ 盲目的图搜索策略

- 深度优先搜索 (DFS)

- 宽度优先搜索 (BFS)

◆ 启发式图搜索策略

- A Search, 即最佳优先搜索

- A* Search

◆ 局部搜索: 爬山法、模拟退火法、遗传算法