

Foundation of Artificial Intelligence

人工智能基础

李翔宇

软件学院

Email: lixiangyu@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

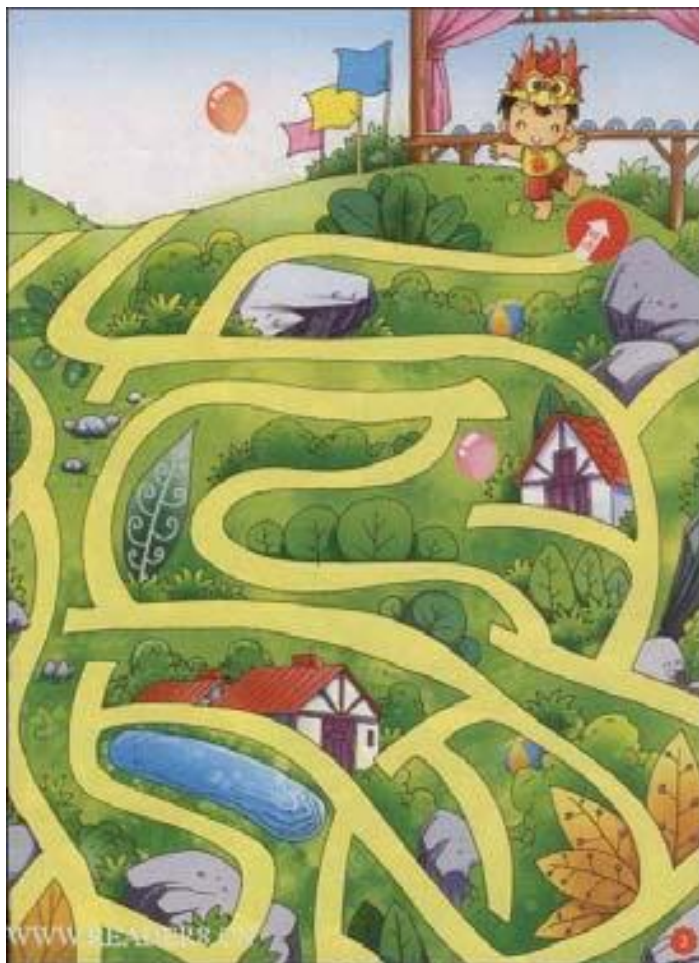
3.3.1 A Search, 即最佳优先搜索

3.3.2 A* Search

3.4 局部搜索：爬山法、模拟退火法、遗传算法

走迷宫

如何走迷宫？



Problem solving in AI 人工智能中的问题求解

- 问题求解 (Problem Solving) 是人工智能第一个大的成就。
- 人工智能问题求解技术主要涉及两个方面：
问题的表示、求解的方法。
- 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。
- 状态空间表示法：用来表示问题及其搜索过程的一种方法。

3.1 图搜索策略

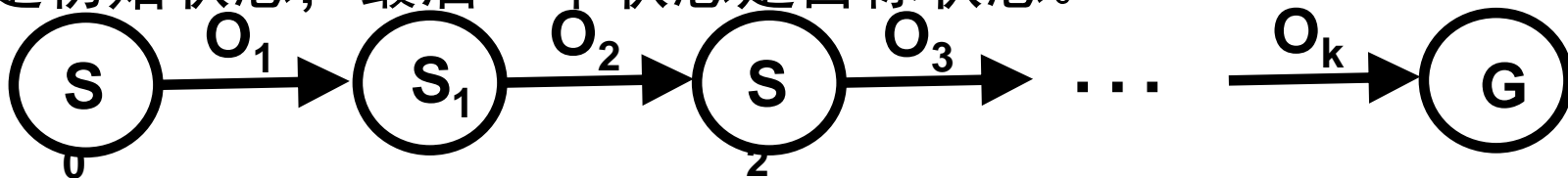
针对搜索问题，可以采用**状态空间知识表示法**来表示问题。

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**。

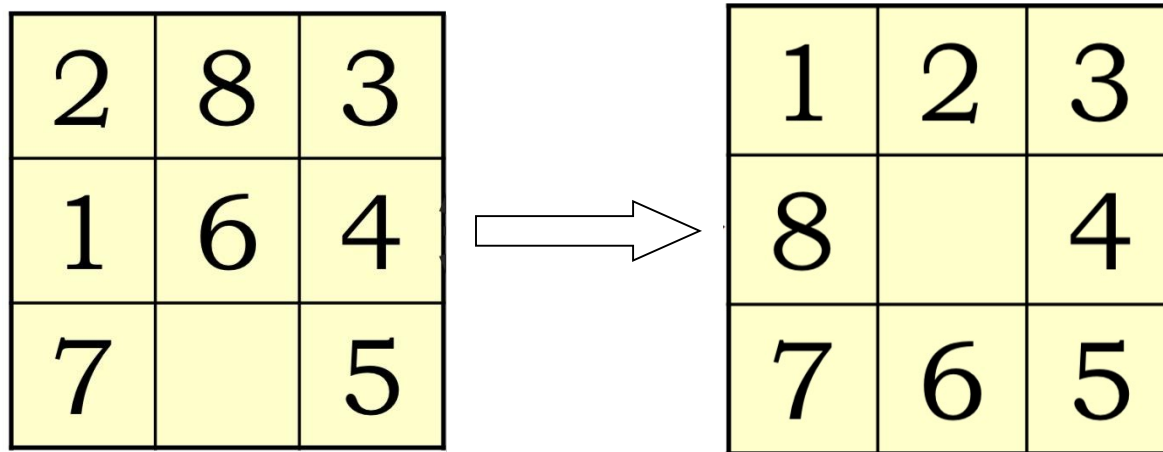
3.1 图搜索策略

- ◆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 $\{O_1, O_2, \dots, O_k\}$ 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。



例： 八数码难题 8-puzzle

在 3×3 的棋盘，摆有八个棋子，每个棋子上标有1至8的某一 数字。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。如何将棋盘从某一初始状态变成最后的目标状态？



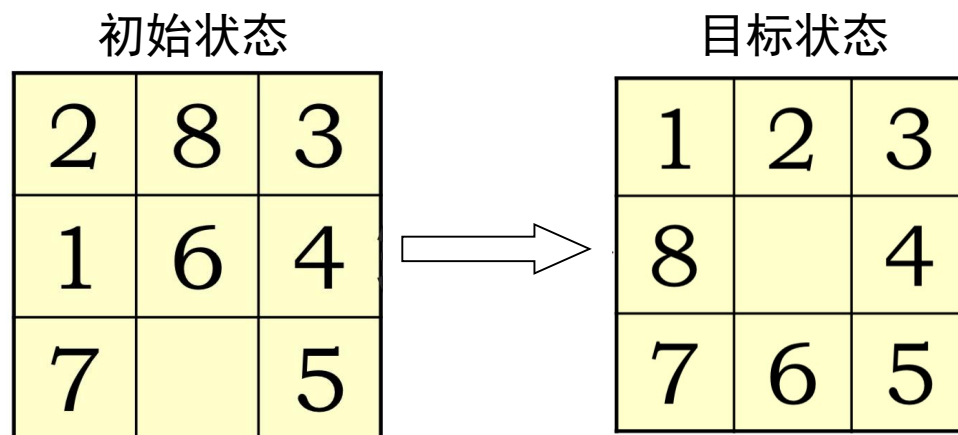
(a) Initial state

(b) goal state

例：八数码难题 8-puzzle

- 八个数码的任何一种摆法就是一个**状态**。
- 八数码的所有摆法构成了状态集合S，它们构成了一个**状态空间**。
- **操作集合**：将移动空格作为操作，即在方格盘上移动数码等价于移动空格。

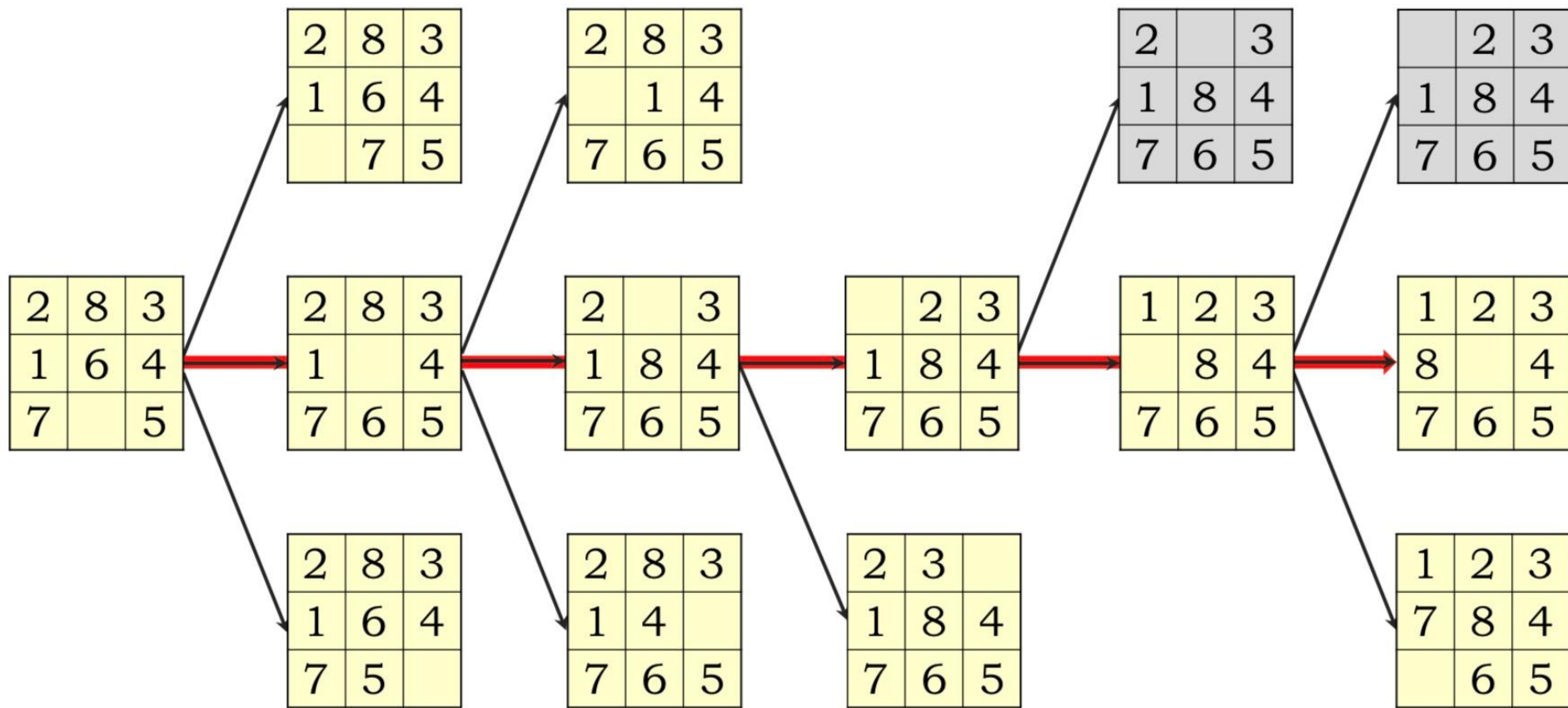
Up: 将空格向上移, if 空格不在最上一行
Down: 将空格向下移, if 空格不在最下一行
Left: 将空格向左移, if 空格不在最左一列
Right: 将空格向右移, if 空格不在最右一列



Searching for Solutions 搜索解

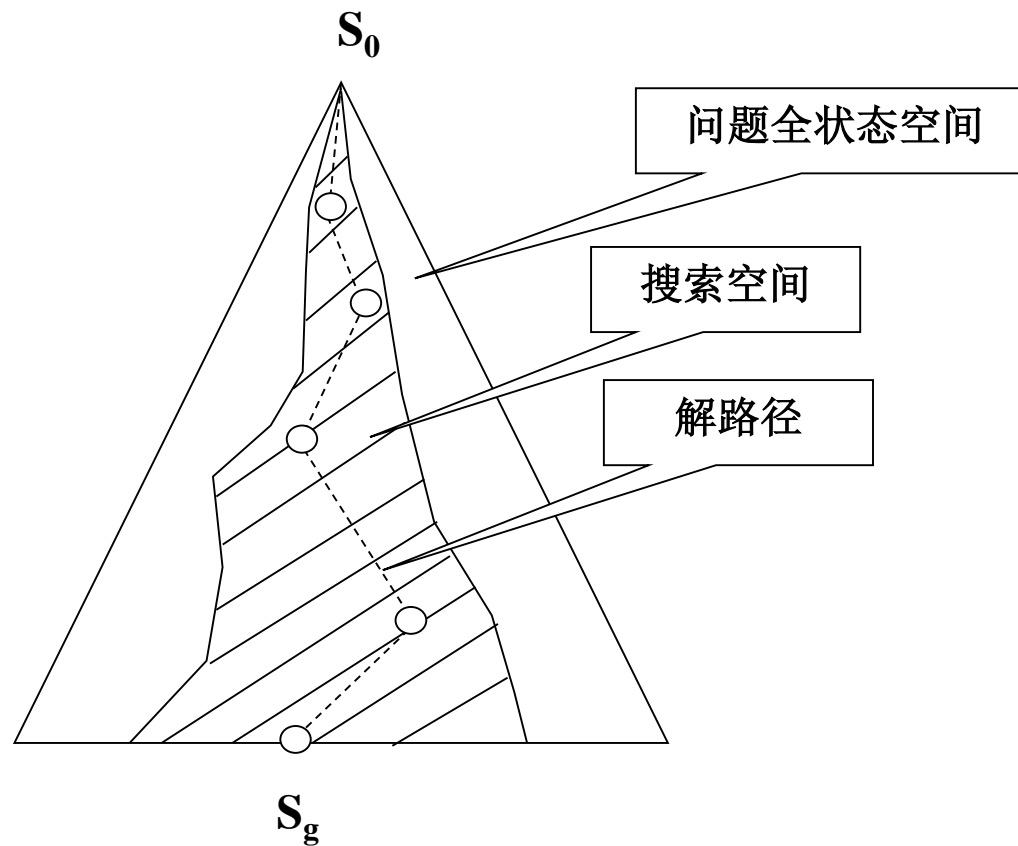
- 从初始状态到目标状态的路径就是问题的解
- 问题求解就是在状态图中搜索这样一条路径

怎么做？



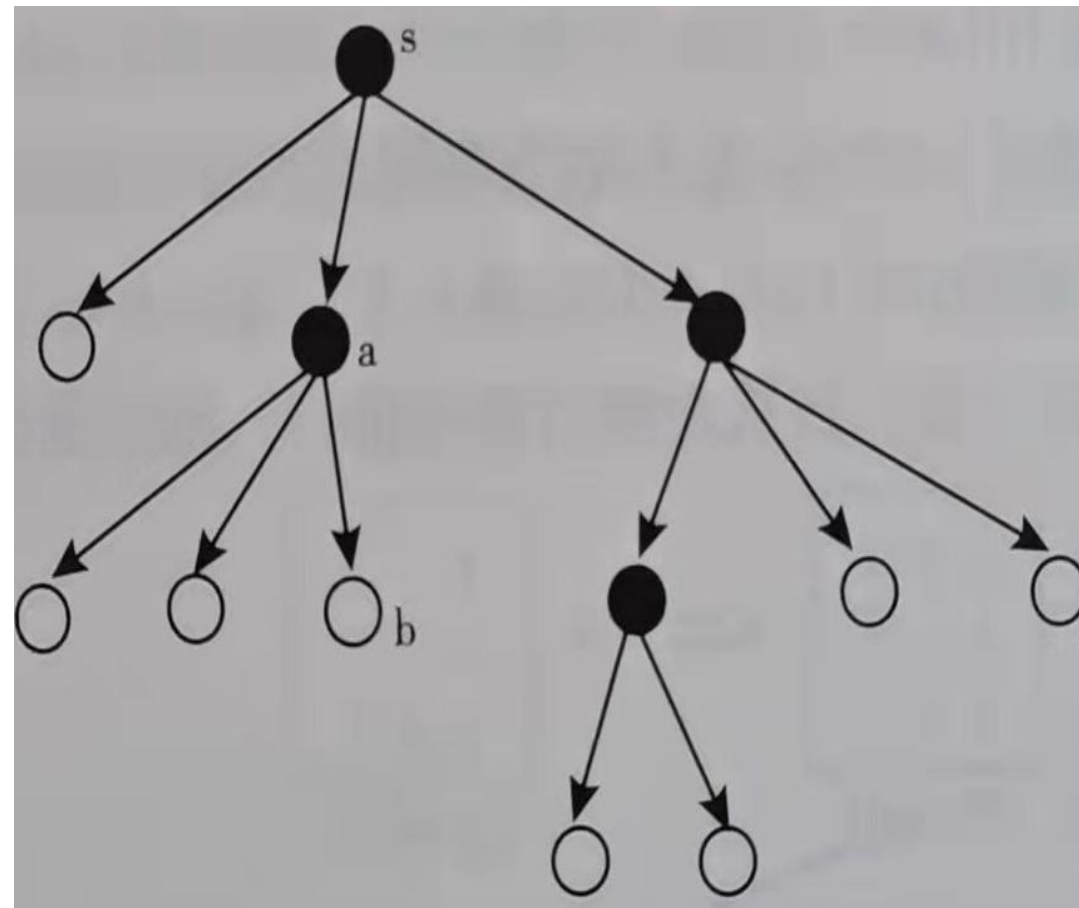
3.1 图搜索策略

- ◆ 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

- ◆ 右图中结点表示状态，**实心圆**表示**已扩展**的结点(即已经生成出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**一个叶结点(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。



Open list & Closed list

□Open 表：存储刚生成的待扩展节点，有进有出

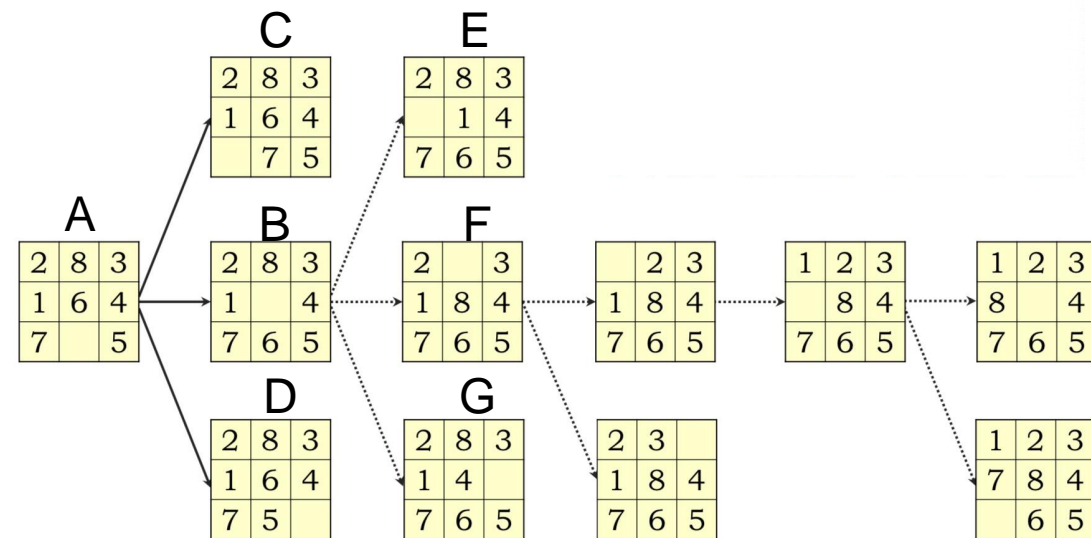
Open list: a list of nodes that is generated but yet not expanded

□Closed 表：存储已扩展节点，有进无出（检查新生成的结点是否已被扩展过） Closed list: a list of nodes that have been expanded

每个结点 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;

n 的后继结点集合 $\{m_i\}$



Open list & Closed list

□Open 表：存储刚生成的待扩展节点，有进有出

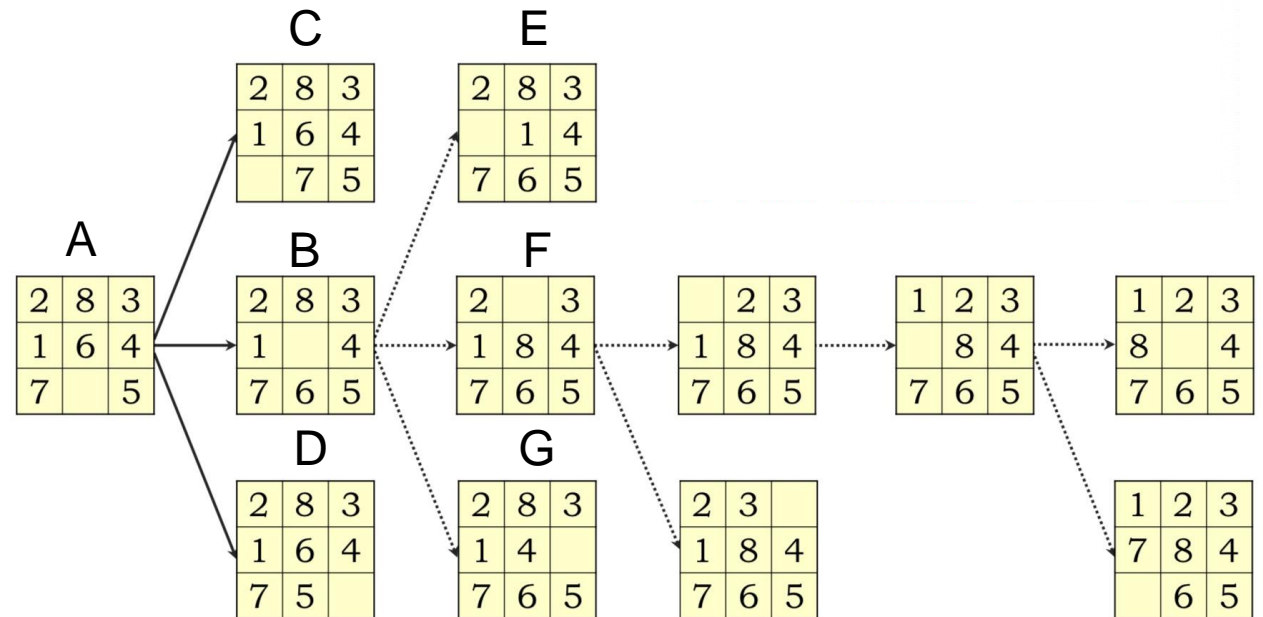
Open list: a list of nodes that is generated but yet not expanded

□Closed 表：存储已扩展节点，有进无出（检查新生成的结点是否已被扩展过） Closed list: a list of nodes that have been expanded

Open=[A]; Closed=[]

Open=[B, C, D]; Closed=[A]

Open=[C, D, E, F, G]; Closed=[A, B]



图搜索算法

ALGORITHM **GRAPHSEARCH**(problem)

INPUT: A problem

OUTPUT: A solution, or failure

Initialize the open list using the initial state of the problem

Initialize the closed list to be empty

WHILE TRUE DO

IF the open list is empty **THEN**

RETURN failure

 Choose a leaf node and remove it from the open list

IF the node contains a goal state **THEN**

RETURN the corresponding solution

Add the node to the closed list

 Expand the chosen node and add the resulting nodes to the open list

only if not in the open nor the closed list

3.1 图搜索策略

◆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会有所不同。

◆**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：

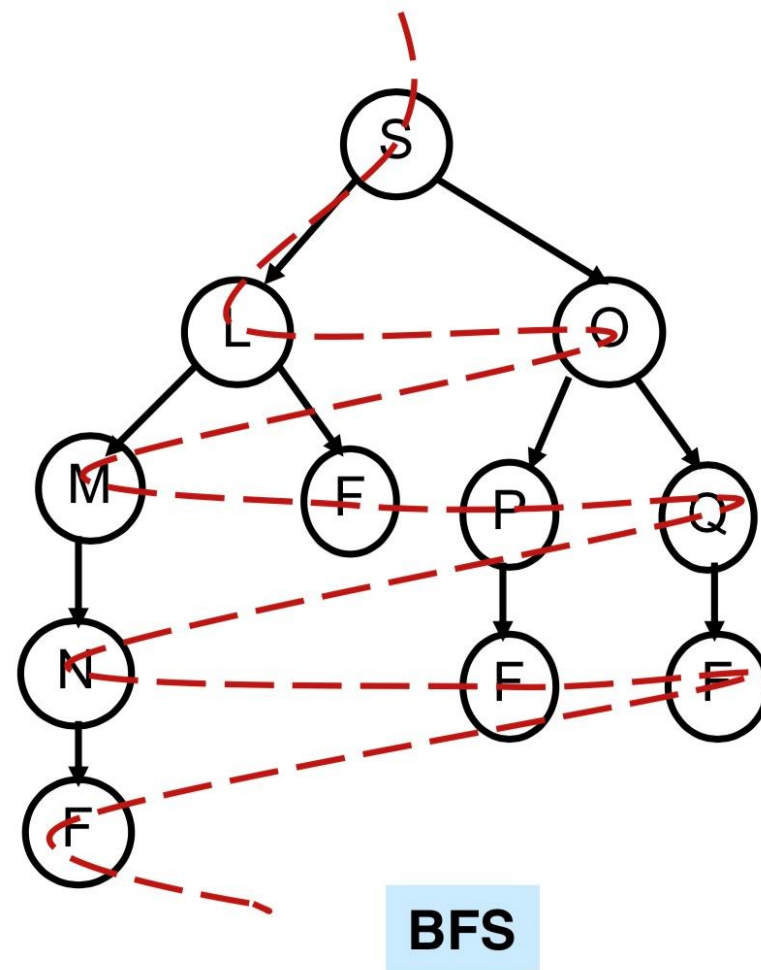
(1) **Breadth-first search** 宽度优先搜索

(2) **Depth-first search** 深度优先搜索

宽度优先搜索(BFS)

Search Strategy 搜索策略

- 先扩展初始状态，然后扩展它的所有后继节点，依此类推
- BFS每次总是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 (BFS)

Implementation 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，深度最小的结点排在最前面（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。



BFS on a Graph 图的宽度优先搜索算法

ALGORITHM	BREADTH-FIRST-SEARCH(problem)
INPUT:	A problem
OUTPUT:	A solution, or failure

Initialize the open list using the initial state of the problem /*FIFO queue*/

Initialize the closed list to be empty

WHILE TRUE DO

IF the open list is empty **THEN RETURN** failure

 Choose the the shallowest node and remove it from the open list

IF the node contains a goal state **THEN**

RETURN the corresponding solution

 Add the node to the closed list

FOR each action **DO**

 Create a child node n

IF n is not in the open or closed lists **THEN**

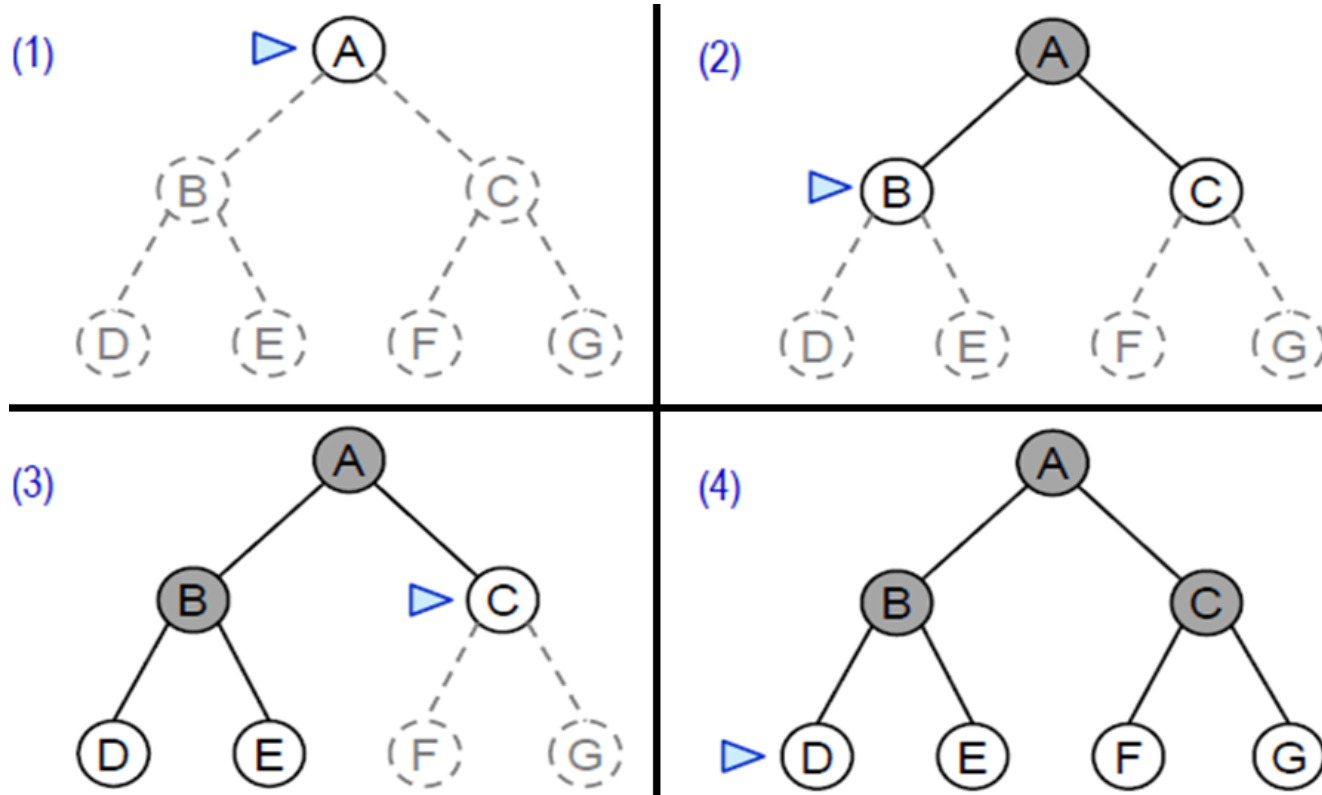
IF the child node contains a goal state

THEN RETURN the corresponding solution

 Insert n to the open list

Breadth-first Search on a Simple Binary Tree

简单二叉树的宽度优先搜索

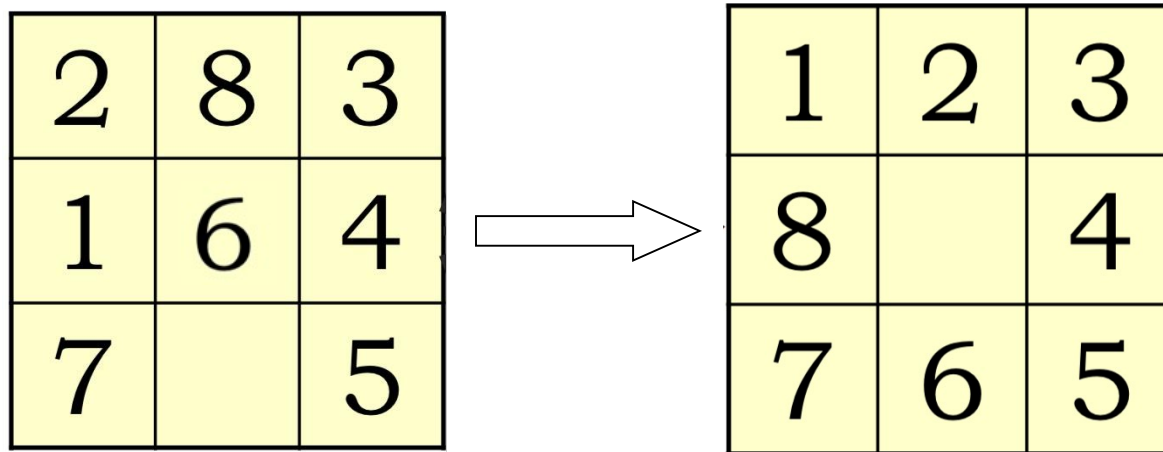


Open表	Close 表
[A]	[]
[B,C]	[A]
[C,D,E]	[A, B]
[D,E,F,G]	[A, B, C]
[E,F,G]	[A, B, C, D]
[F,G]	[A, B, C, D, E]
[G]	[A, B, C, D, E, F]
[]	[A, B, C, D, E, F, G]

The searching order is {A, B, C, D, E, F, G}.

8-puzzle Problem BFS

在 3×3 的棋盘，摆有八个棋子，每个棋子上标有1至8的某一 数字。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。如何将棋盘从某一初始状态变成最后的目标状态？

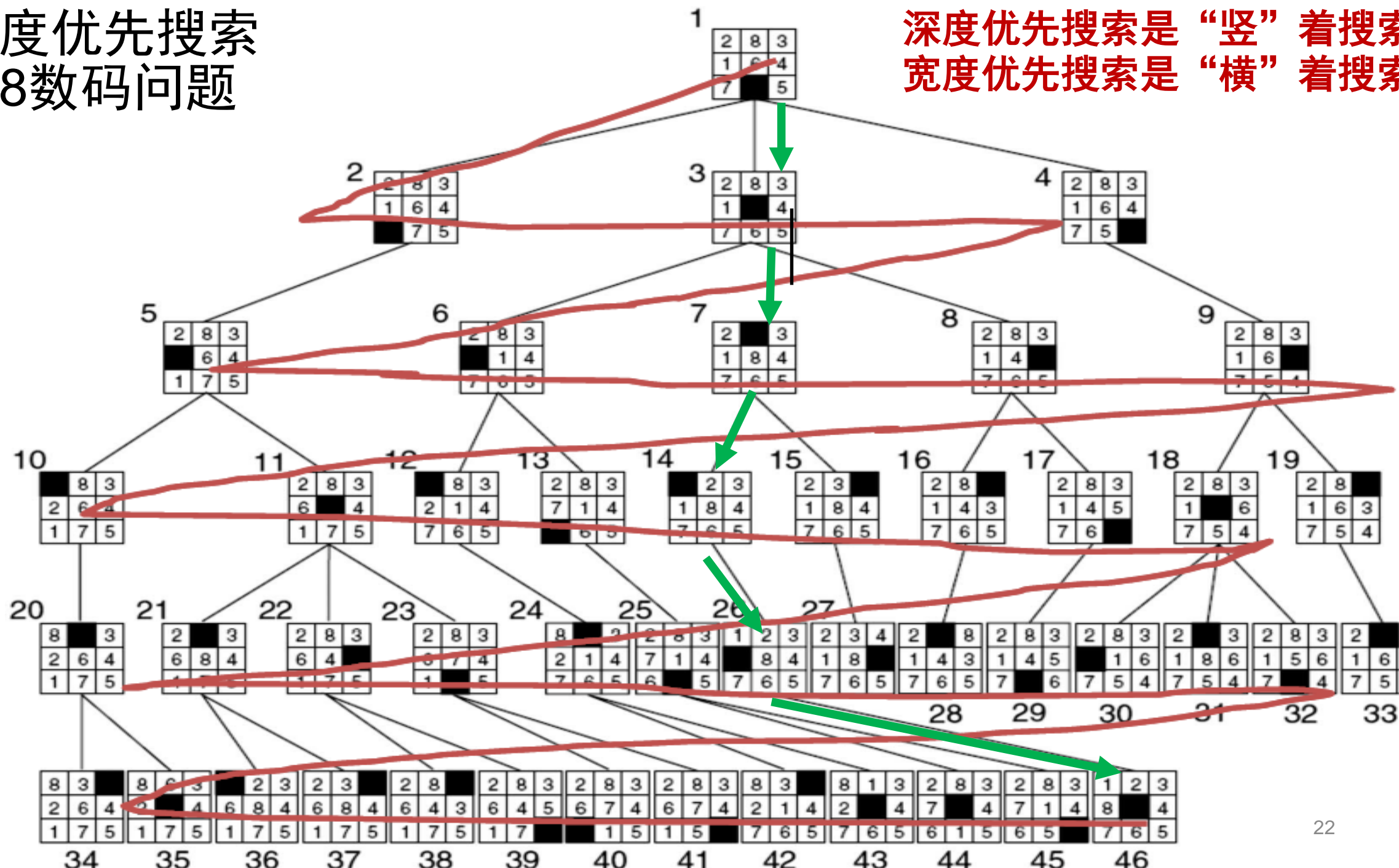


(a) Initial state

(b) goal state

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

◆**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。

Features of BFS 宽度优先搜索的性质

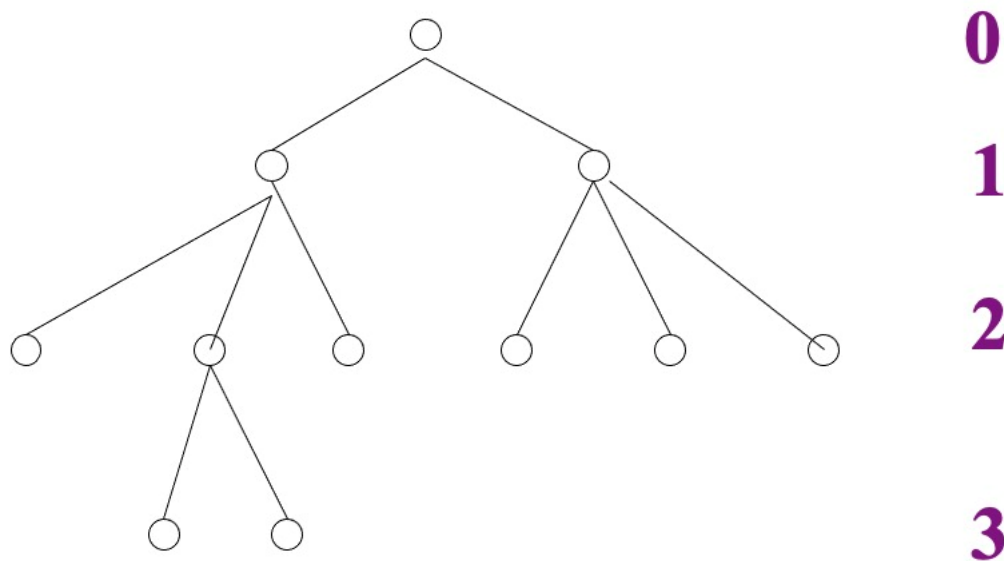
- 完备性：当问题有解时，一定能找到解.
- 最优性：当问题为单位代价，且问题有解时，一定能找到最优解
- BFS是一个通用的与问题无关的方法.
- 求解问题的效率较低

宽度优先搜索不适合求解大规模问题

深度优先搜索Depth-first search (DFS)

Search Strategy 搜索策略

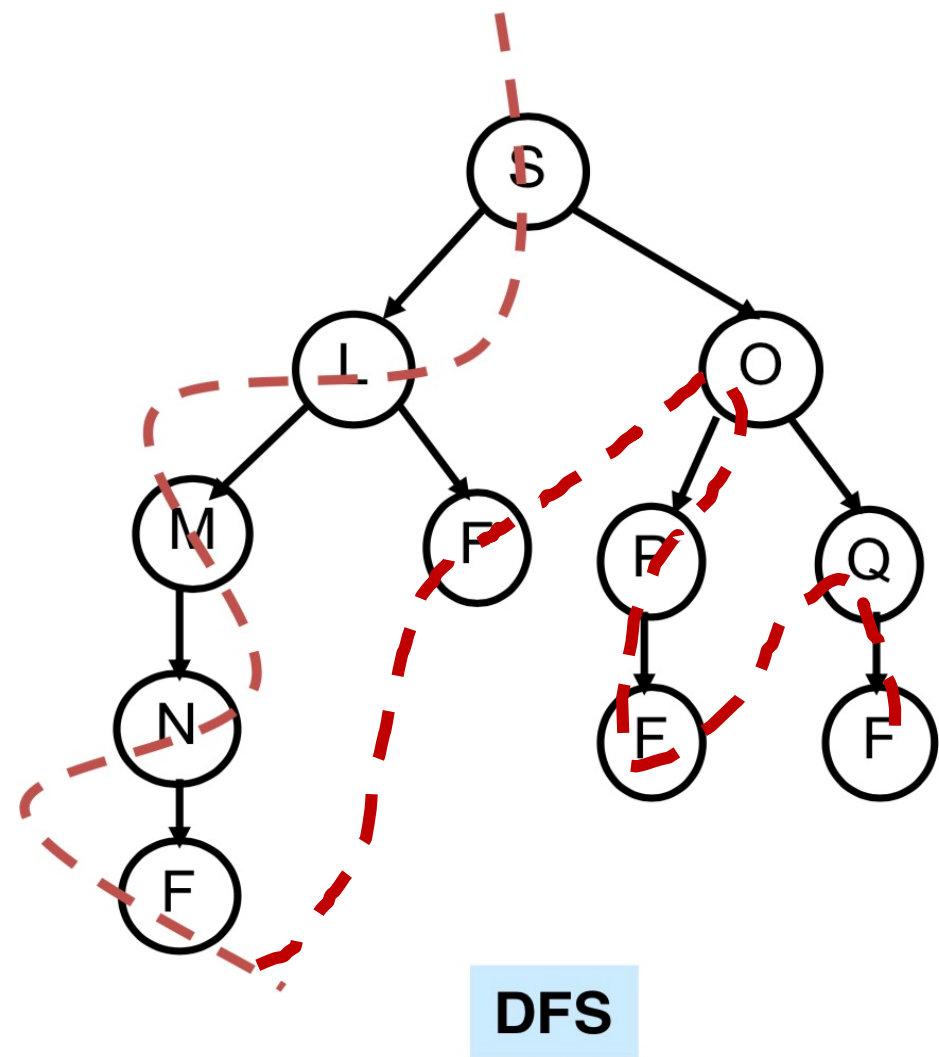
- 深度优先搜索的**基本思想**是**优先扩展深度最深的结点**。
- 在一个图中，**初始结点的深度定义为0**，其他结点的深度定义为其父结点的深度加 1。



深度优先搜索---DFS

Search Strategy 搜索策略

- ◆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。
- ◆搜索直接伸展到搜索树的最深层，直到那里的结点**没有后继结点**；
- ◆然后搜索算法回退到下一个还有未扩展后继结点的上层结点继续扩展。
- ◆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。



Depth-first search (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。



DFS on a Graph 图的深度优先搜索算法

ALGORITHM DEPTH-FIRST-SEARCH(problem)

INPUT: A problem

OUTPUT: A solution, or failure

Initialize the open list using the initial state of the problem /*LIFO stack*/

Initialize the closed list to be empty

WHILE TRUE DO

IF the open list is empty **THEN RETURN** failure

 Choose the first node in the open list

IF the node contains a goal state **THEN**

RETURN the corresponding solution

 Add the node to the closed list

FOR each action **DO**

 Create a child node *n*

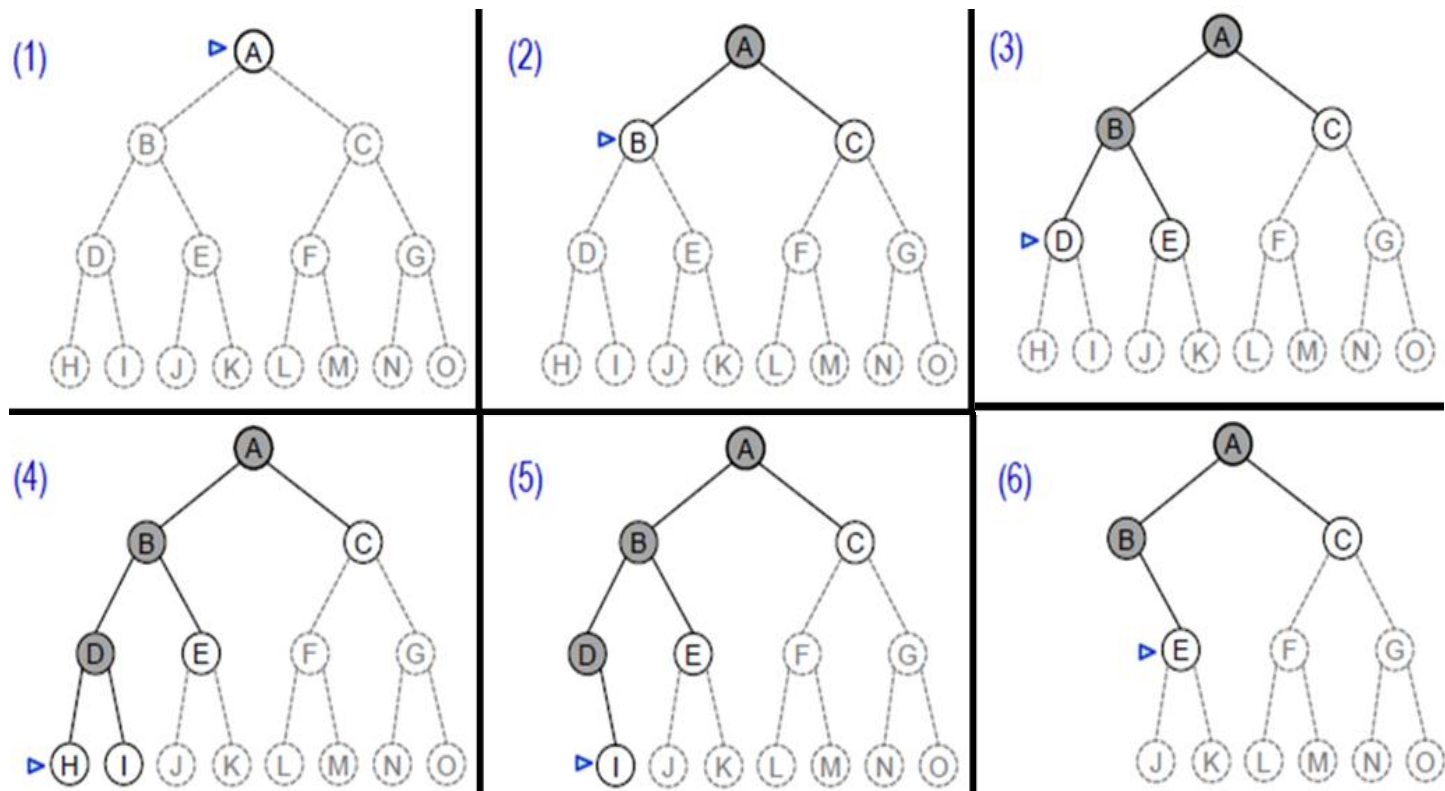
IF *n* is not in the open or closed lists **THEN**

IF the child node contains a goal state

THEN RETURN the corresponding solution

 Insert *n* to the open list

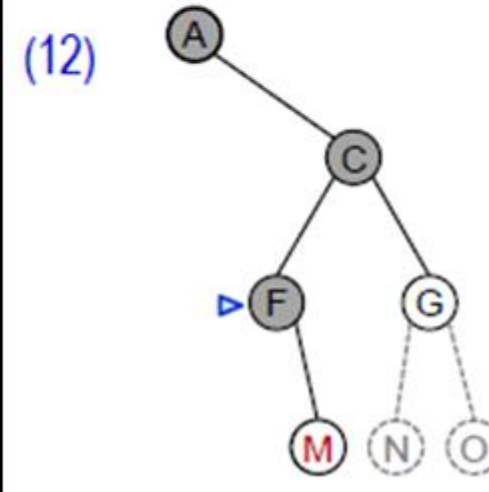
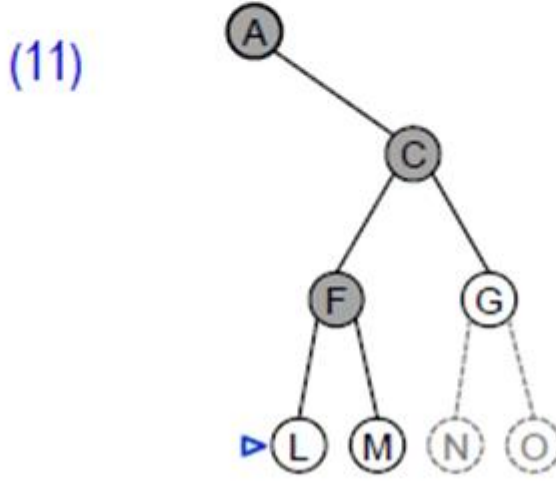
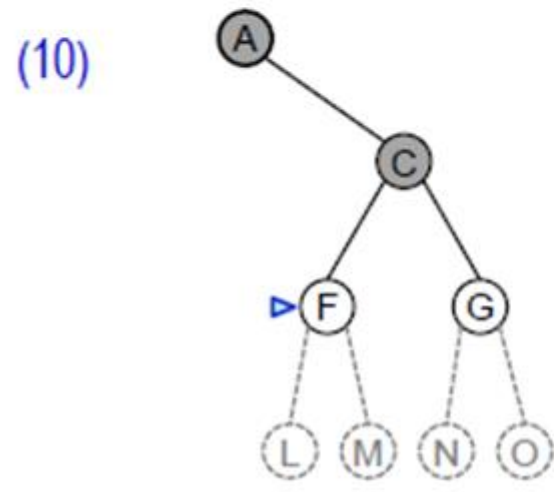
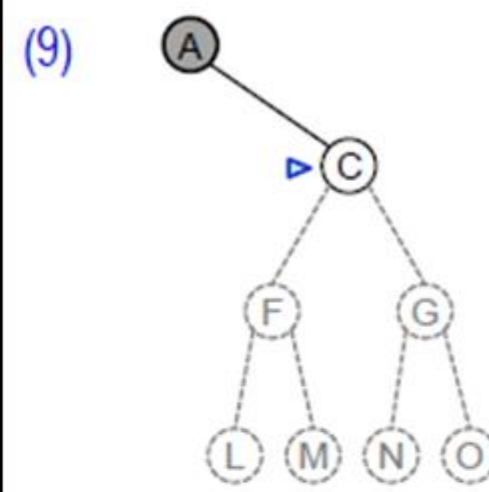
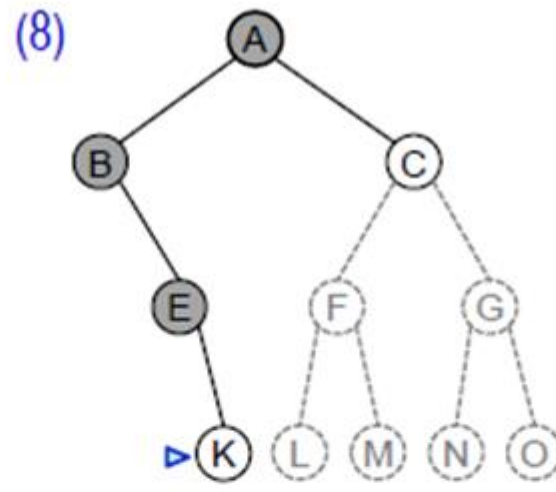
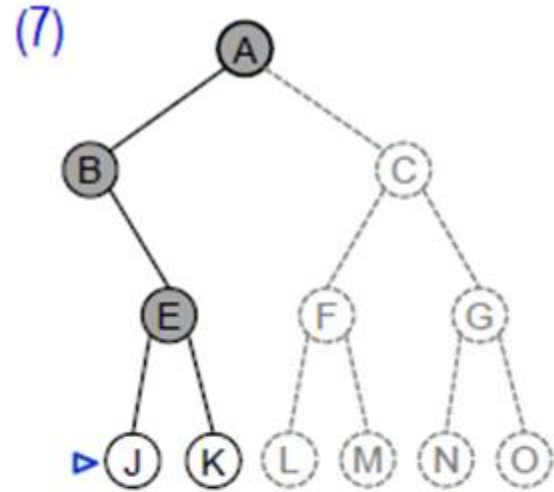
Depth-first Search on a Simple Binary Tree



Open表	Close 表
[A]	[]
[B,C]	[A]
[D,E,C]	[A, B]
[H,I,E,C]	[A, B, D]
[I,E,C]	[A, B, D, H]
[E,C]	[A, B, D, H,I]
[J, K, C]	[A, B, D, H, I, E]
[K, C]	[A, B, D, H, I, E, J]
[C]	[A, B, D, H, I, E, J, K]
[F, G]	[A, B, D, H, I, E, J, K, C]
[L, M, G]	[A, B, D, H, I, E, J, K, C, F]

依次类推，直到找到目标函数或open=[]

Depth-first Search on a Simple Binary Tree

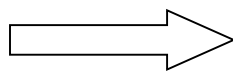


8-puzzle Problem DFS

在 3×3 的棋盘，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。如何将棋盘从某一初始状态变成最后的目标状态？

2	8	3
1	6	4
7		5

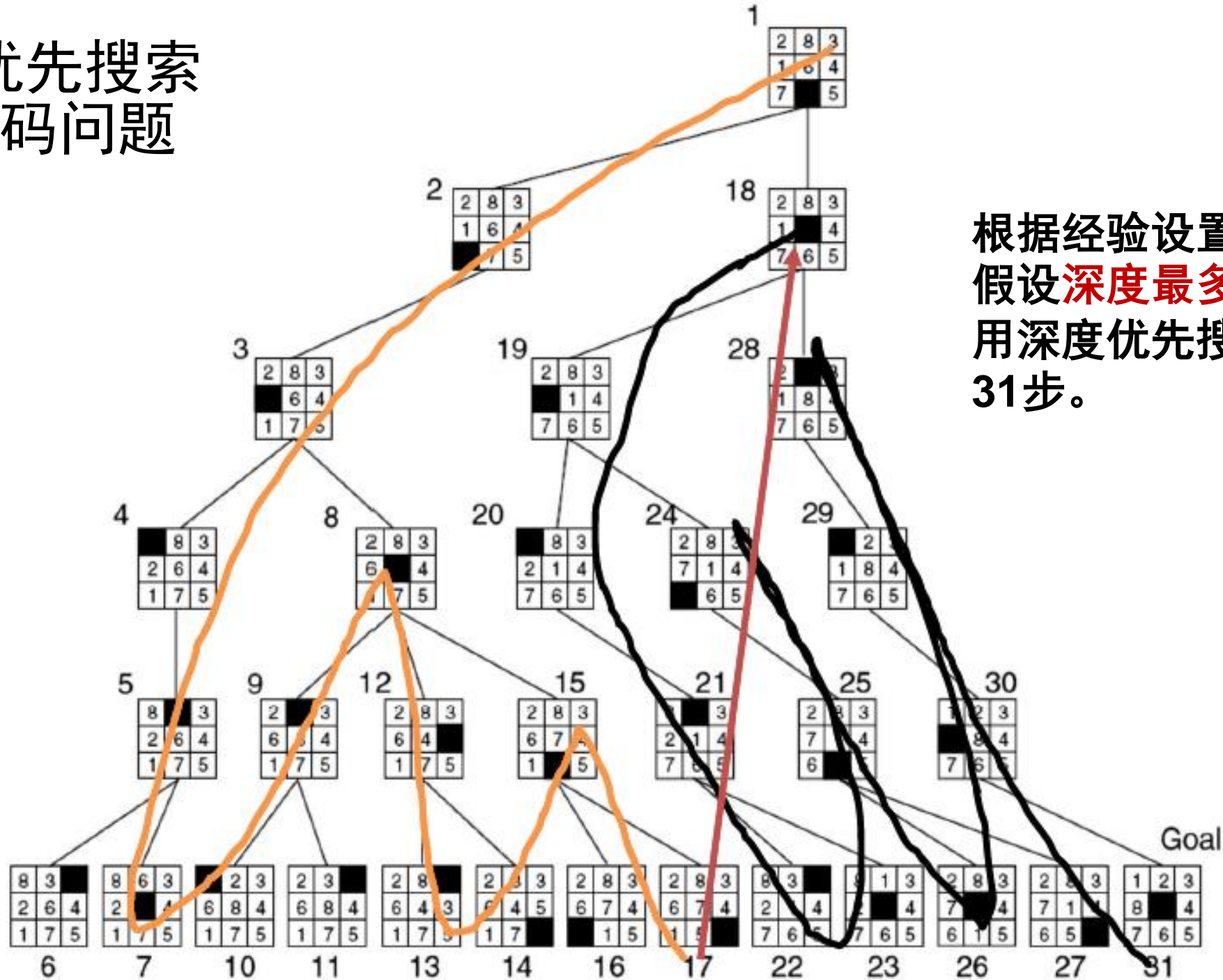
(a) Initial state



1	2	3
8		4
7	6	5

(b) goal state

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。

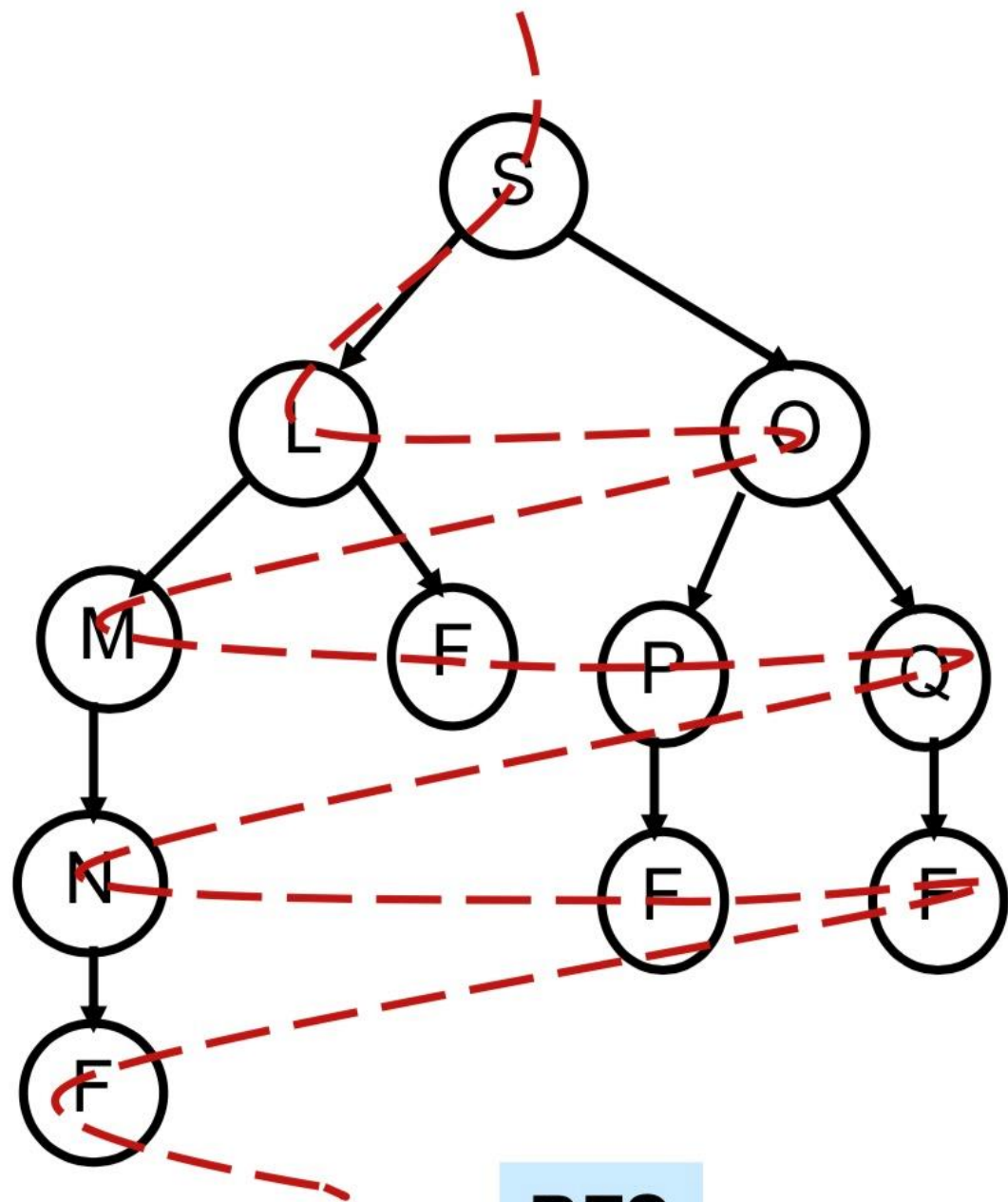
Features of DFS 深度优先搜索的性质

- 最优性：非最优，一般不能保证找到最优解
- 完备性：当深度限制不合理时（浅于目标节点时），找不到解.
- 最坏情况时，搜索空间等同于穷举

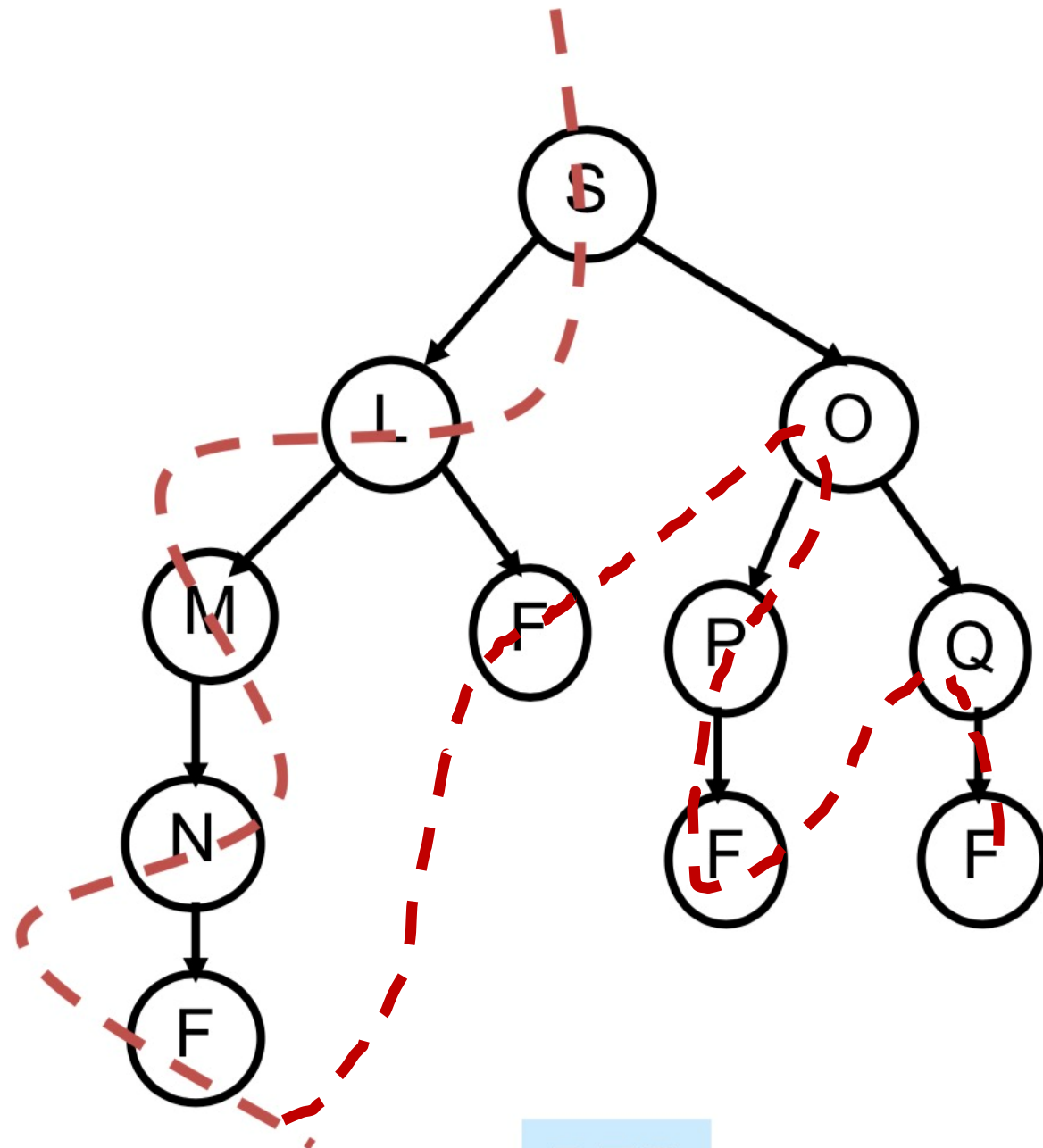
与BFS相比，其优势在于：空间复杂度低，因为只存储一条从根到叶子的路径。

- DFS是一个通用的与问题无关的方法.

广泛使用，多花点时间，牺牲最优性，使问题可解



BFS



DFS

BFS 与 DFS 的比较

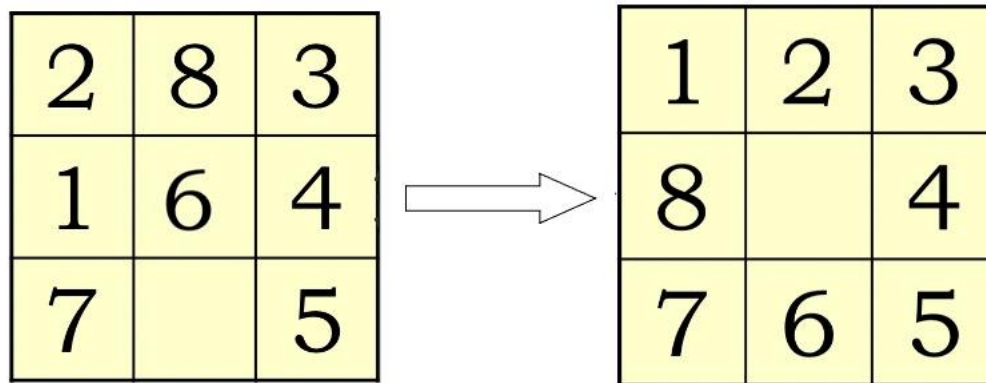
- ◆DFS总是首先扩展**最深**的未扩展结点；BFS总是首先扩展**最浅**的未扩展结点。
- ◆BFS 效率低， 但却一定能够求得问题的最优解， 即**BFS是完备的， 且是最优的**；
- ◆DFS不能保证找到最优解， **即DFS既不完备， 也不最优**；
- ◆在**不要求求解速度**且目标结点的层次**较深**的情况下， **BFS优于DFS**， 因为BFS一定能够求得问题的解， 而DFS在一个扩展的很深但又没有解的分支上进行搜索， 是一种无效搜索， 降低了求解的效率， 有时甚至不一定能找到问题的解；
- ◆在**要求求解速度**且目标结点的层次**较浅**的情况下， **DFS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 有信息搜索策略

- ◆ 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。



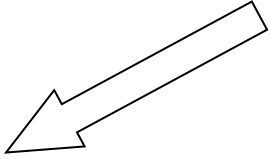
(a) **Initial state**

(b) **goal state**

Solve 8-Puzzle problem

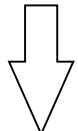
(a) Initial state s

2	8	3
1	6	4
7		5



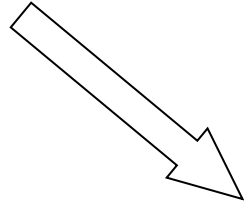
2	8	3
	1	4
7	6	5

A (错位3个)



1	2	3
	8	4
7	6	5

B (错位1个)



2	8	3
1	4	5
7		6

C(错位6个)



1	2	3
8		4
7	6	5

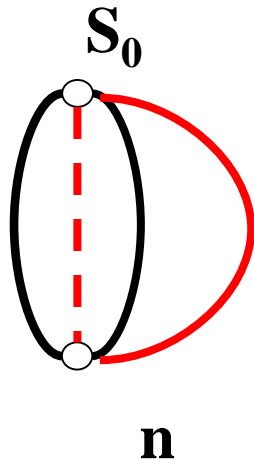
(b) goal state

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。

Uninformed (Blind) Search v.s. Informed Search Strategy

Uninformed (Blind) Search

盲目搜索



?

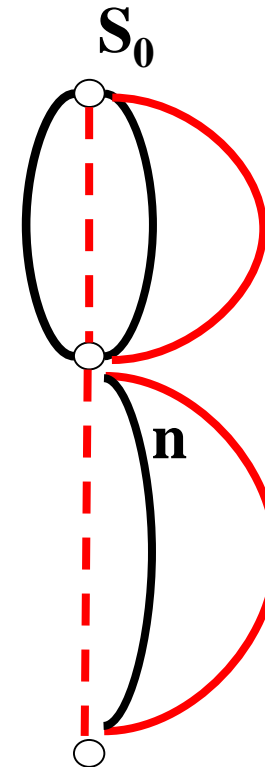


S_g

BFS, DFS

Informed(Heuristic) Search Strategy

启发式搜索



S_g

评价函数 (evaluation function)

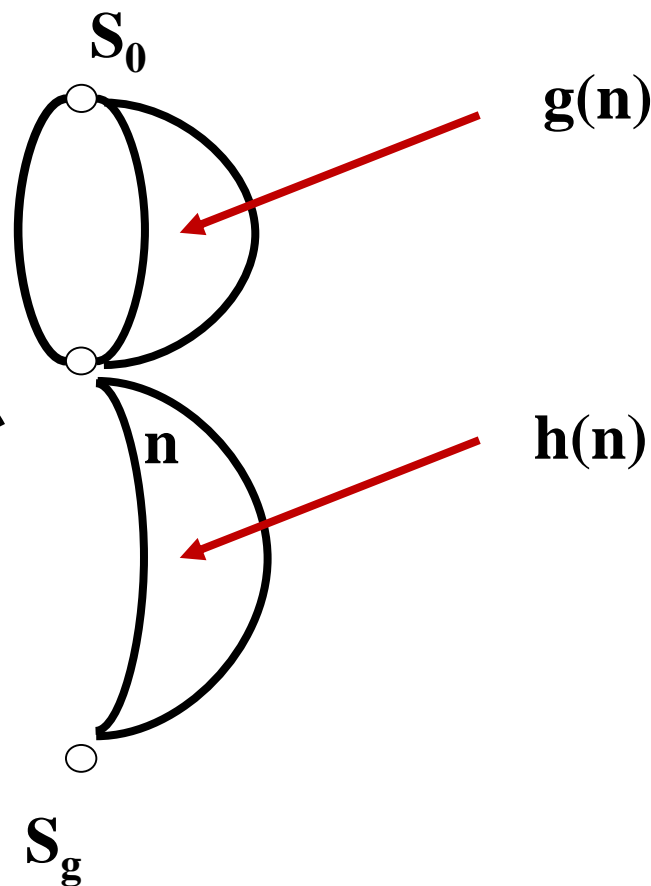
为了尽快找到从初始结点到目标结点的一条代价比较小的路径，我们希望所选择的结点尽可能在最佳路径上。如何评价一个结点在最佳路径上的可能性呢？

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 实际发生的路径代价值；
- (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**



如何设计启发函数？ Heuristic function

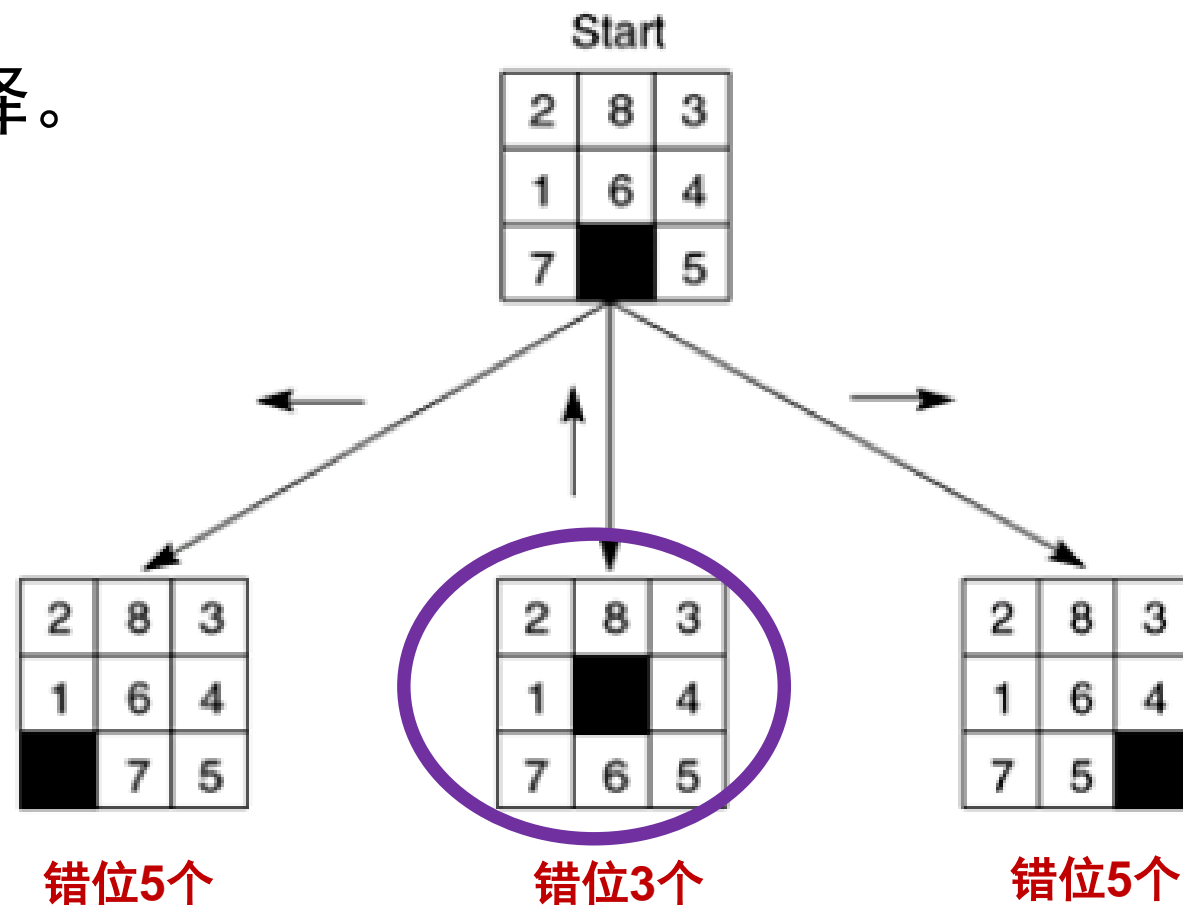
以8 数码问题为例

◆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。

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

1	2	3
8		4
7	6	5

Goal

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。

2	1	3
8		4
7	6	5

1	2	3
8		4
7	6	5

Goal

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

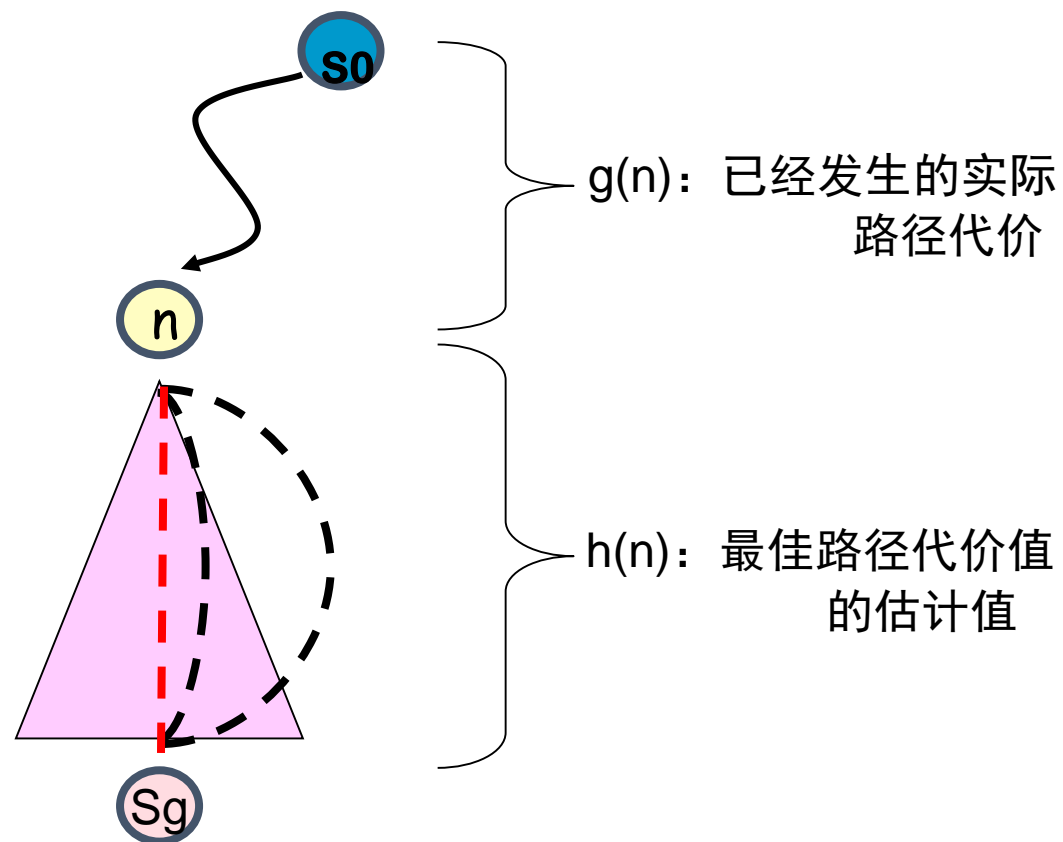
单纯依靠启发函数搜索是否可行？

◆ 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）

A搜索

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

◆ **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。

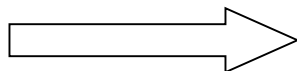
◆ **实现方法**

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**”。

用A 算法解决8数码问题

2	8	3
1	6	4
7		5

(a) Initial state s



1	2	3
8		4
7	6	5

(b) goal state

2	8	3
1	6	4
7		5

$$h(s) = 4, g(s) = 0$$

定义评价函数： $f(n) = g(n) + h(n)$

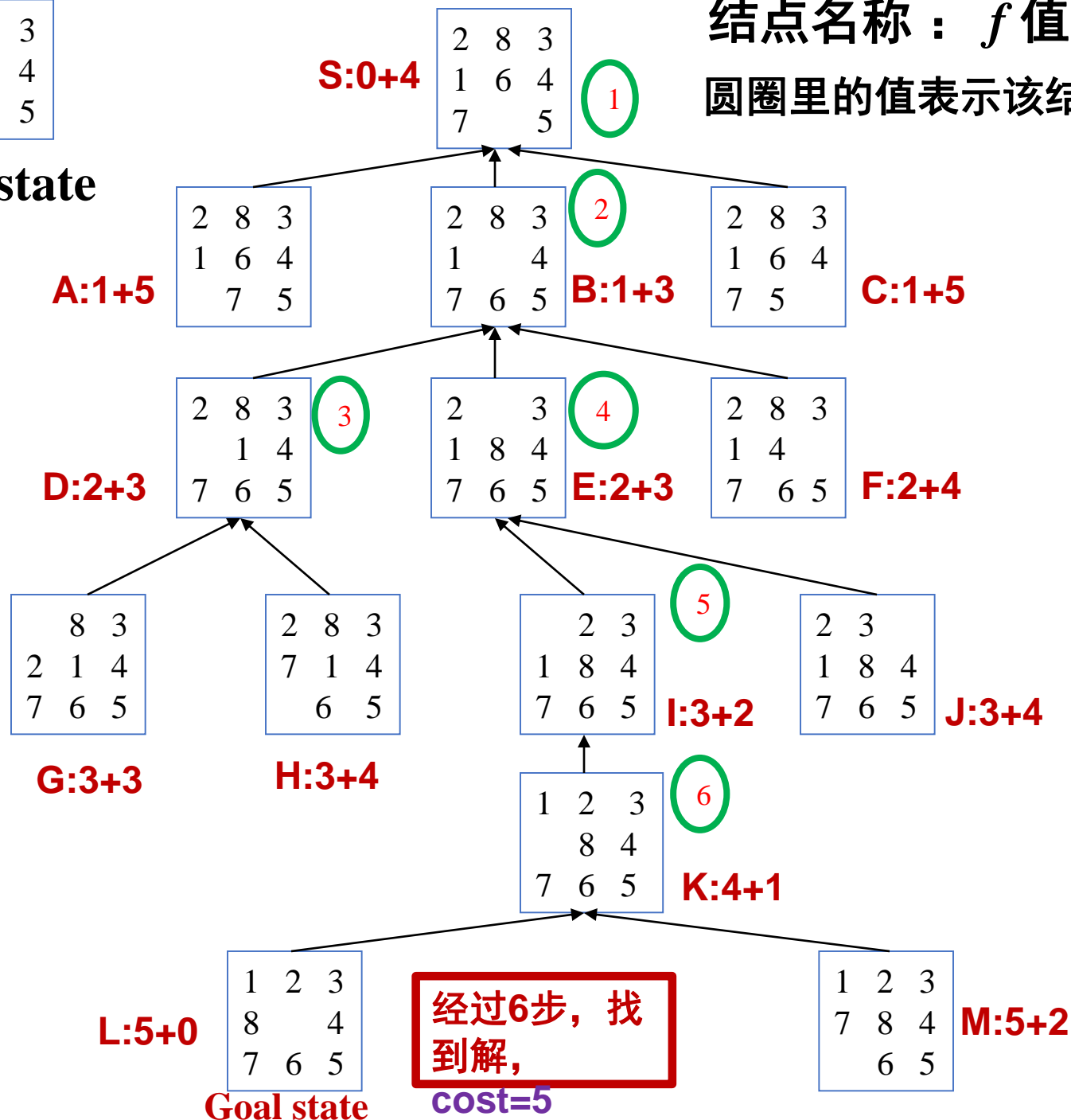
- ◆ $g(n)$ 为从初始状态到当前状态的代价值，定义为移动数码牌的步数，即结点 n 的深度。
- ◆ $h(n)$ 是从 n 到目标结点的最短路径的代价值，定义为当前状态中“错位”数码牌的个数。

1	2	3
8		4
7	6	5

结点名称： f 值= $g+h$

圆圈里的值表示该结点是第 i 个被扩展的

Goal state



◆ g 为结点 n 的深度, h 为错位数码牌个数

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]

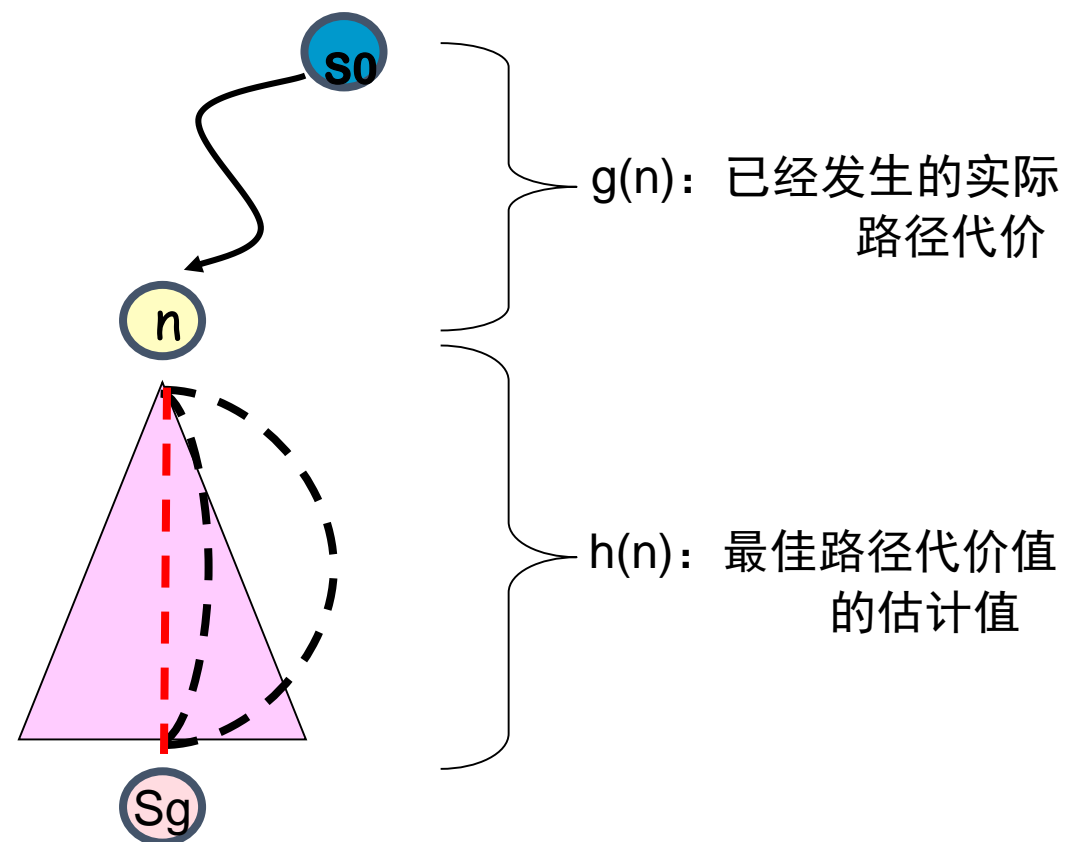
7. Open = [L5, A6, C6, F6, G6, H7, J7, M7]
Closed = [S4, B4, D5, E5, I5, K5]

扩展节点: 6 生成节点: 13

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



启发式搜索评价函数的情况

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

◆ If $f(n) = g(n)$, i.e. $h(n)=0$

当 $h(n)=0$ 时，退化为盲目搜索；

➤ If $h(n)=0$, $f(n) = g(n) = d(n)$, $d(n)$ 代表节点n的深度

即为 **BFS (盲目搜索)**

➤ If $h(n) = 0, f(n) = g(n)$ 满足m是n的儿子节点，则 $g(m)<g(n)$

即为 **DFS (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。

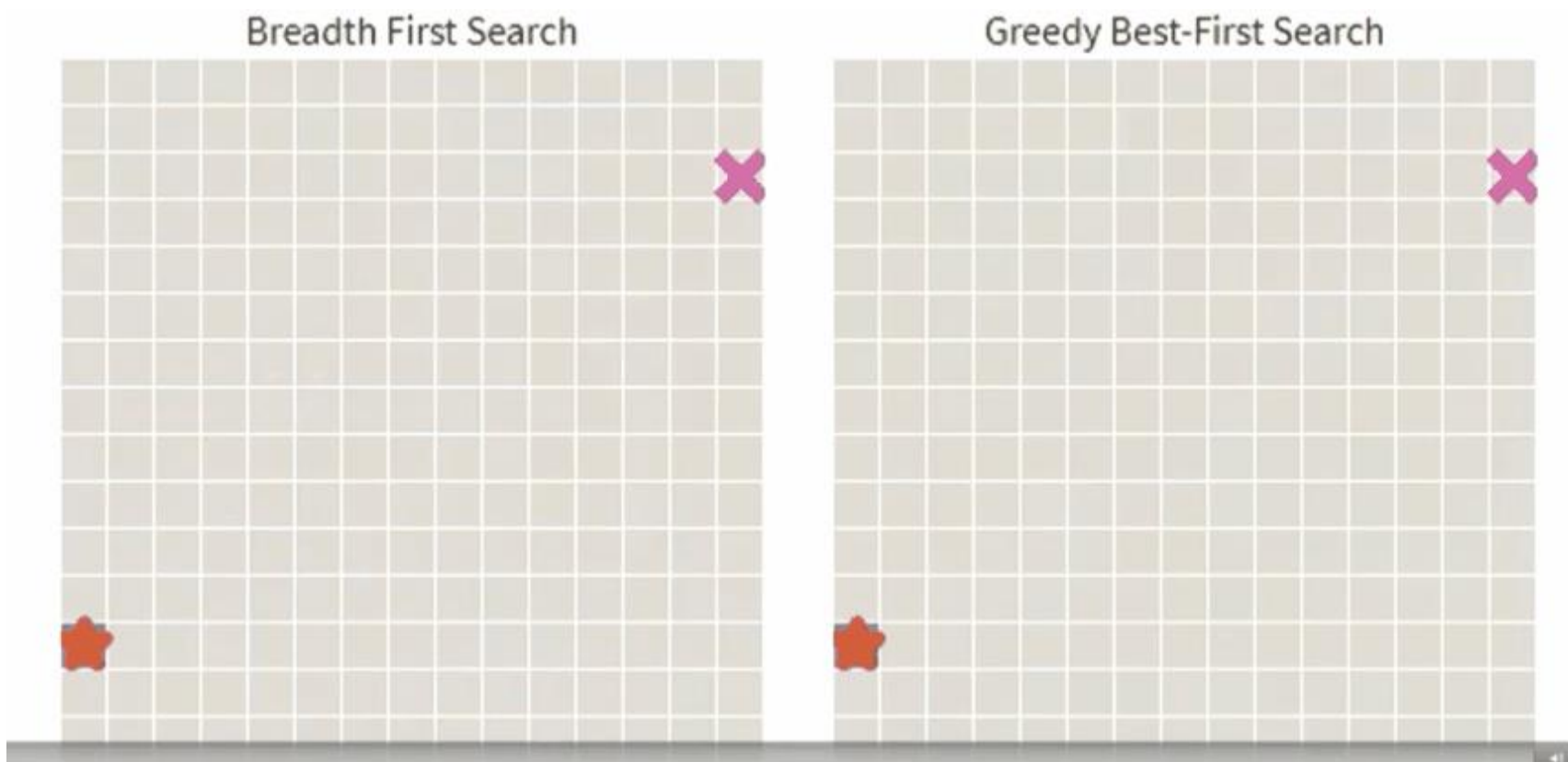
贪婪搜索

- ◆ **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**。

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）

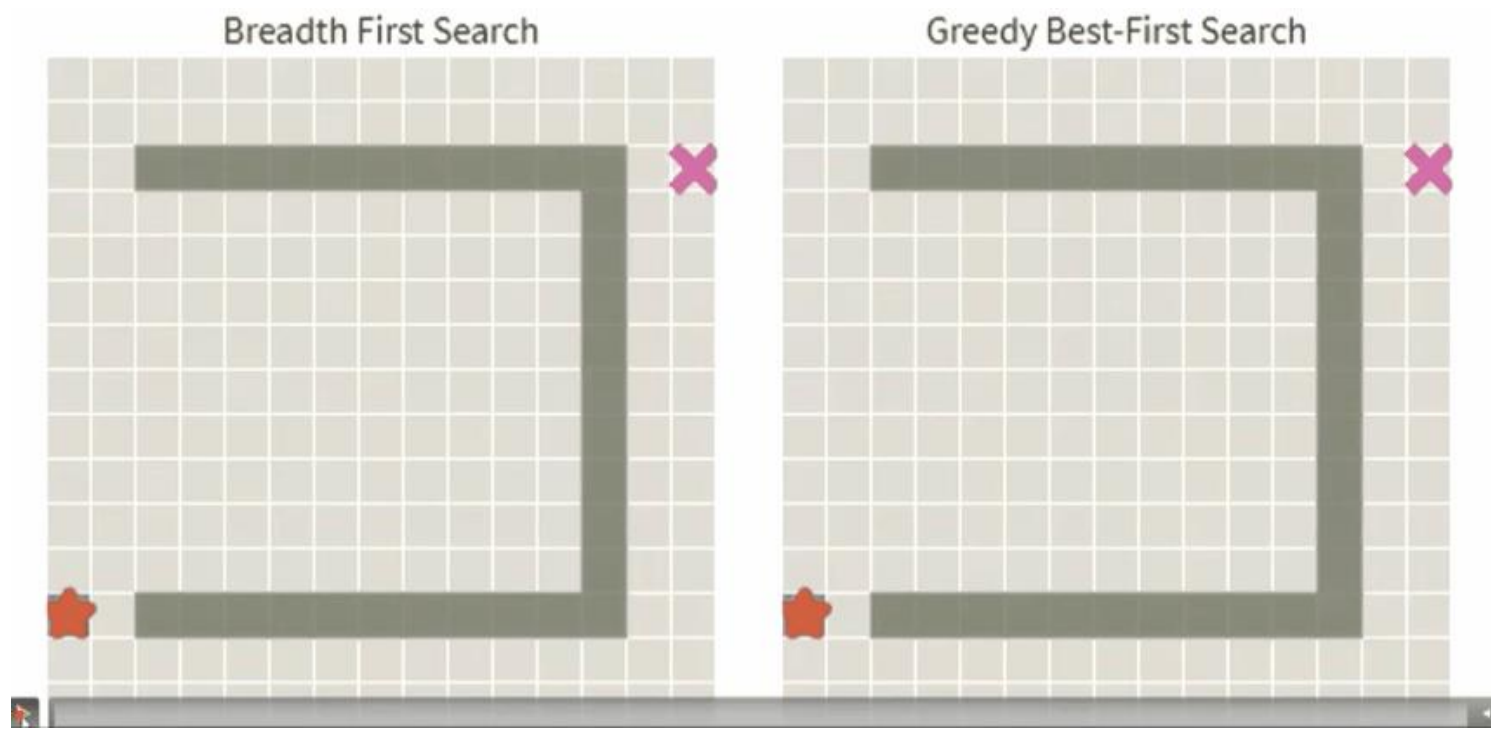
◆ 在没有障碍物的情况，显然 GBFS 搜索速度更快，且两种算法都能找到最短路径。



对比 BFS 和 贪婪搜索 算法

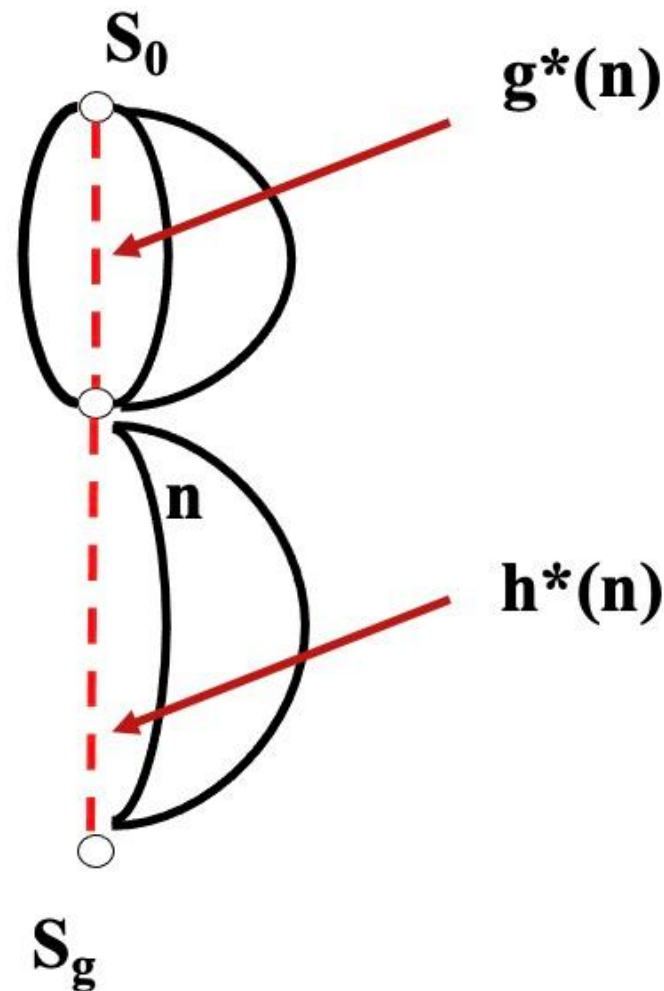
贪婪搜索是最佳优先搜索的特例，亦称为贪婪最佳优先搜索（GBFS）

- ◆ 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。
- ◆ 但如果存在障碍物，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搜索算法没有对评价函数 $f(n)$ 做任何限制。
- ◆ 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**。
- $g^*(n)$: 从 s （初始状态/结点）到结点 n 的最短路径的代价值。可知： $g(n) \geq g^*(n)$
- $h^*(n)$: 从结点 n 到目标结点的最短路径的代价值。
- $f^*(n) = g^*(n) + h^*(n)$: 从 s 经过 n 到目标结点的最短路径的代价值。
- ◆ 若 $h(n) \leq h^*(n)$ ，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**。



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。**
- ◆ **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，我们并不知道 $h^*(n)$ 的值，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。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 $h(n) \leq h^*(n)$** 。这样用A*算法就可以找到该问题距离最短的一条路径。

用 A* 算法解决8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。

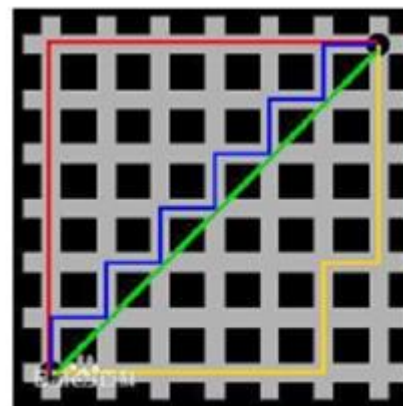
1	2	3
8		4
7	6	5

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

$h2(s)=4$

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。

2	8	3
1		4
7	6	5

Initial state

1	2	3
8		4
7	6	5

(a) goal state

2	8	3
	1	4
7	6	5

(b)

$g=1$
 $h1=3,$
 $h2=5$

(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	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)

$g=1, f=4$
 $h1=3, h2=3$

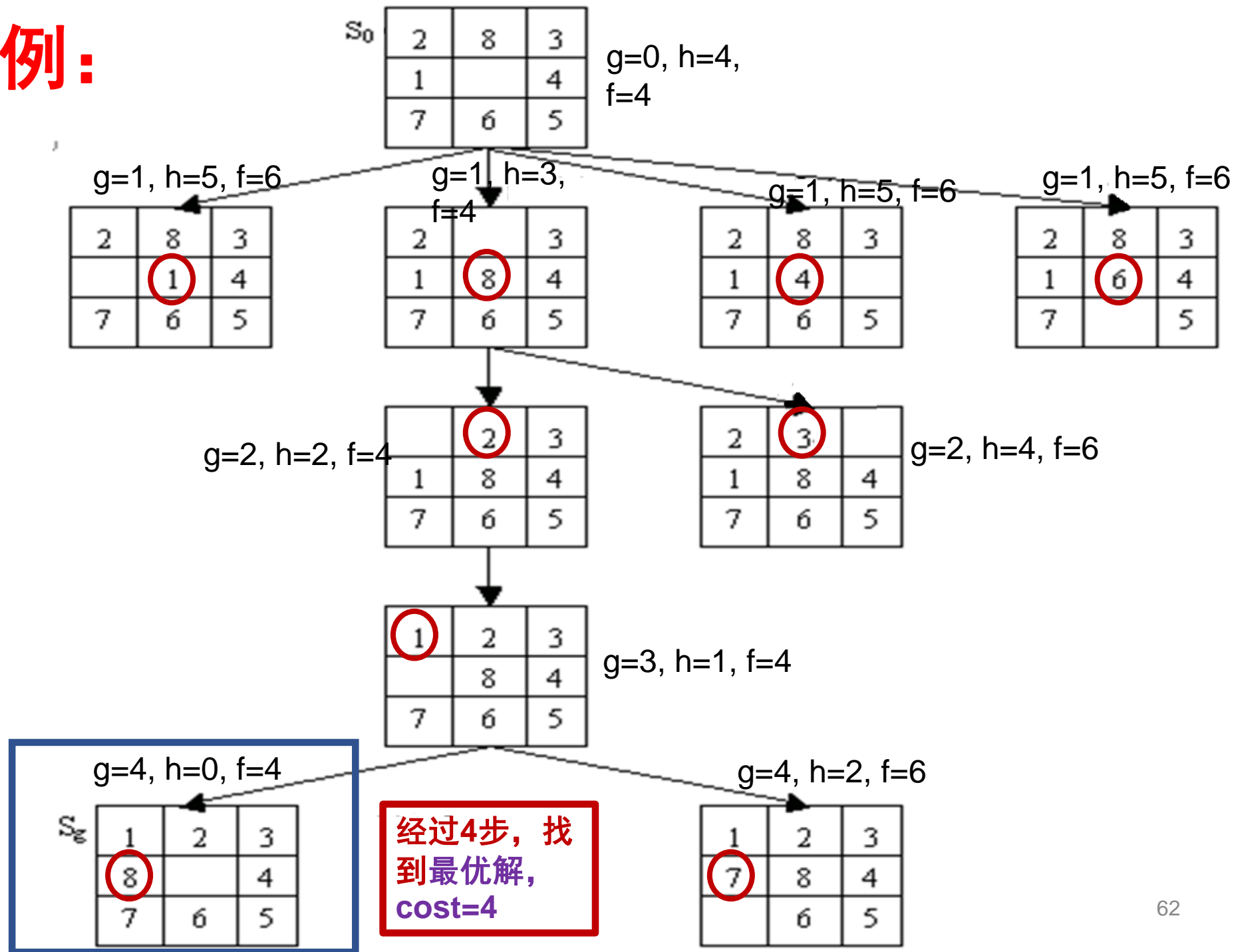
- ◆ $h1(n)$ = “错位数码牌” 的个数
- ◆ $h2(n)$ = 所有数码牌与其目标位置之间的曼哈顿距离之和。

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”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--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” 问题

Step3 设计满足A* 算法的启发函数

◆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” 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” 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1个人把船划从右岸划回左岸，消耗1次，
- 同时左岸人数增多1，即总人数变为： $m+c+1$
- 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” 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， $h(n)=m+c-2b \leq h^*(n) \leq m+c$ ，满足A*算法的条件。

“修道士与野人”问题：

$K=5, r=3$, 11次摆渡

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;

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

L01, L02, L03, L11,
L10, L20, L21, L30

采用A*算法解决传教士与野人渡河问题示例

$g=1$

$g=2$

$g=3$

$g=4$

$g=5$

$g=6$

$g=7$

$g=8$

$g=9$

$g=10$

$g=11$

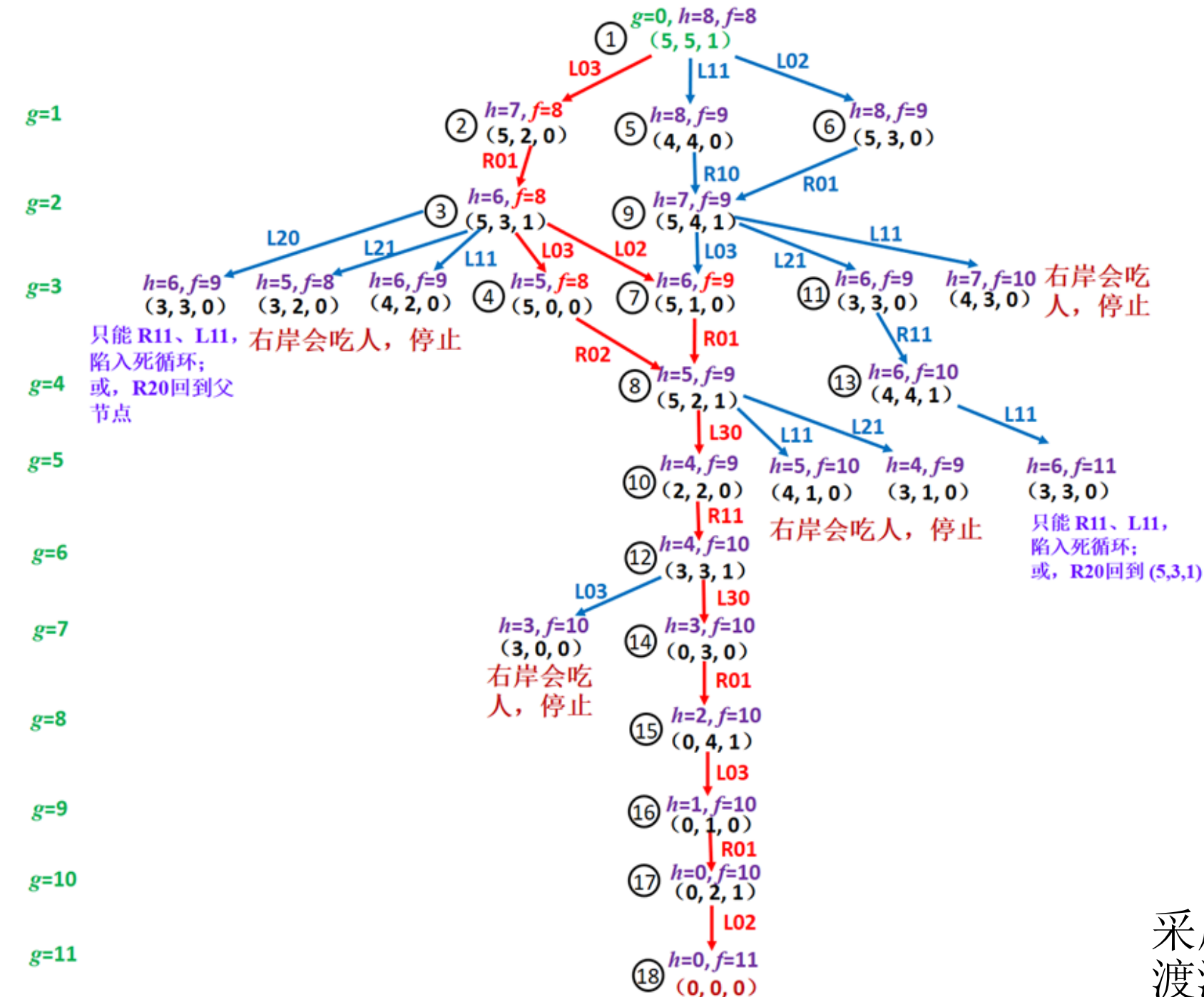
只能 R11、L11，
右岸会吃人，停止
陷入死循环；
或，R20回到父
节点

右岸会吃
人，停止

右岸会吃人，停止

只能 R11、L11，
陷入死循环；
或，R20回到 (5,3,1)

右岸会吃
人，停止



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

Classical Search 经典搜索

- 我们前面介绍过的搜索算法都系统地探索空间，称为**经典搜索算法**。
- 当找到目标时，到达此目标的路径就是这个问题的一個解。
- 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。
- **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。
- 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

例：八皇后问题

在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。



在八皇后问题中，重要的是最终皇后在棋盘上的布局，而不是皇后加入的先后次序。

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、作业车间调度、自动程序设计，电信网络优化、车辆寻径和文件夹管理。

局部搜索算法

◆ **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：

- Hill-climbing search（爬山搜索）

- Simulated annealing search（模拟退火搜索）

- Genetic algorithms（遗传算法）

Hill-climbing Search 爬山搜索

- ◆ **爬山法是最基本的局部搜索技术。**爬山法(最陡上升版本)搜索，是简单的循环过程，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。



Hill-climbing Search 爬山搜索

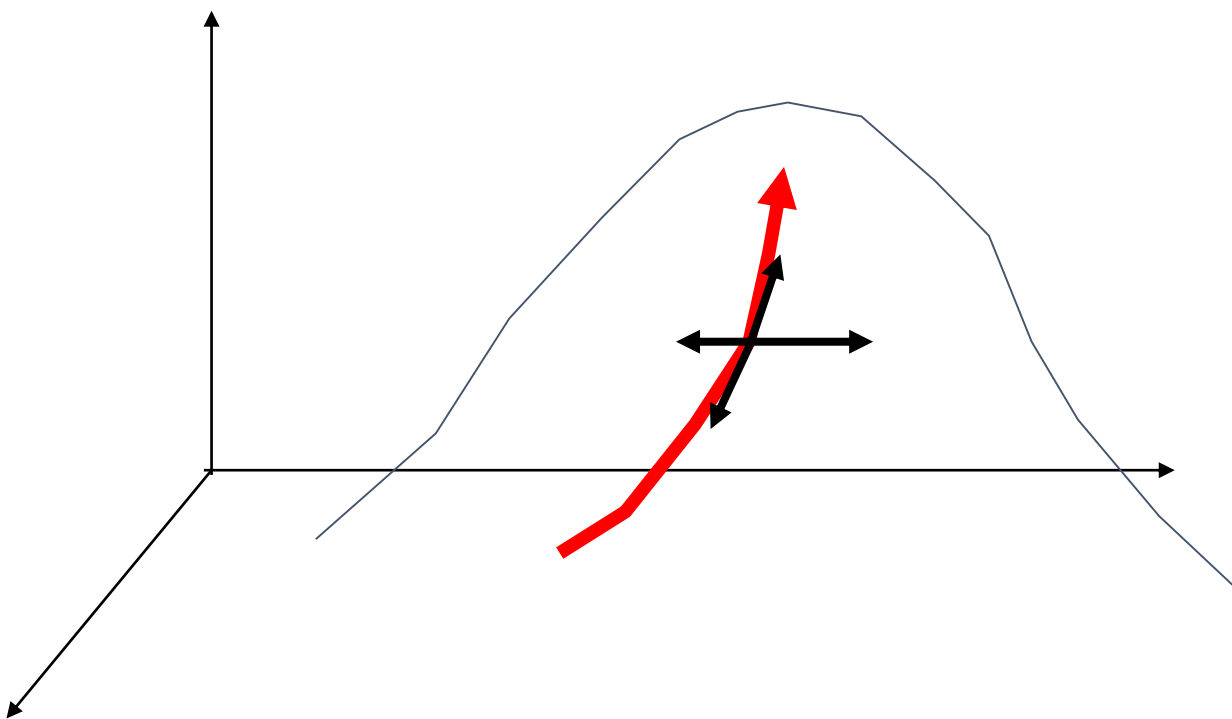
- ◆ **爬山法是最基本的局部搜索技术。**爬山法（最陡上升版本）搜索，是简单的循环过程，不断向值增加的方向持续移动——即，登高。
- ◆ 算法在到达一个“峰顶”时终止，邻接状态中没有比它值更高的。
- ◆ 算法不维护搜索树，因此当前结点的数据结构只需要记录当前状态和目标函数值。
- ◆ 爬山法不会考虑与当前状态不相邻的状态。这就像健忘的人在大雾中试图登顶珠穆朗玛峰一样。



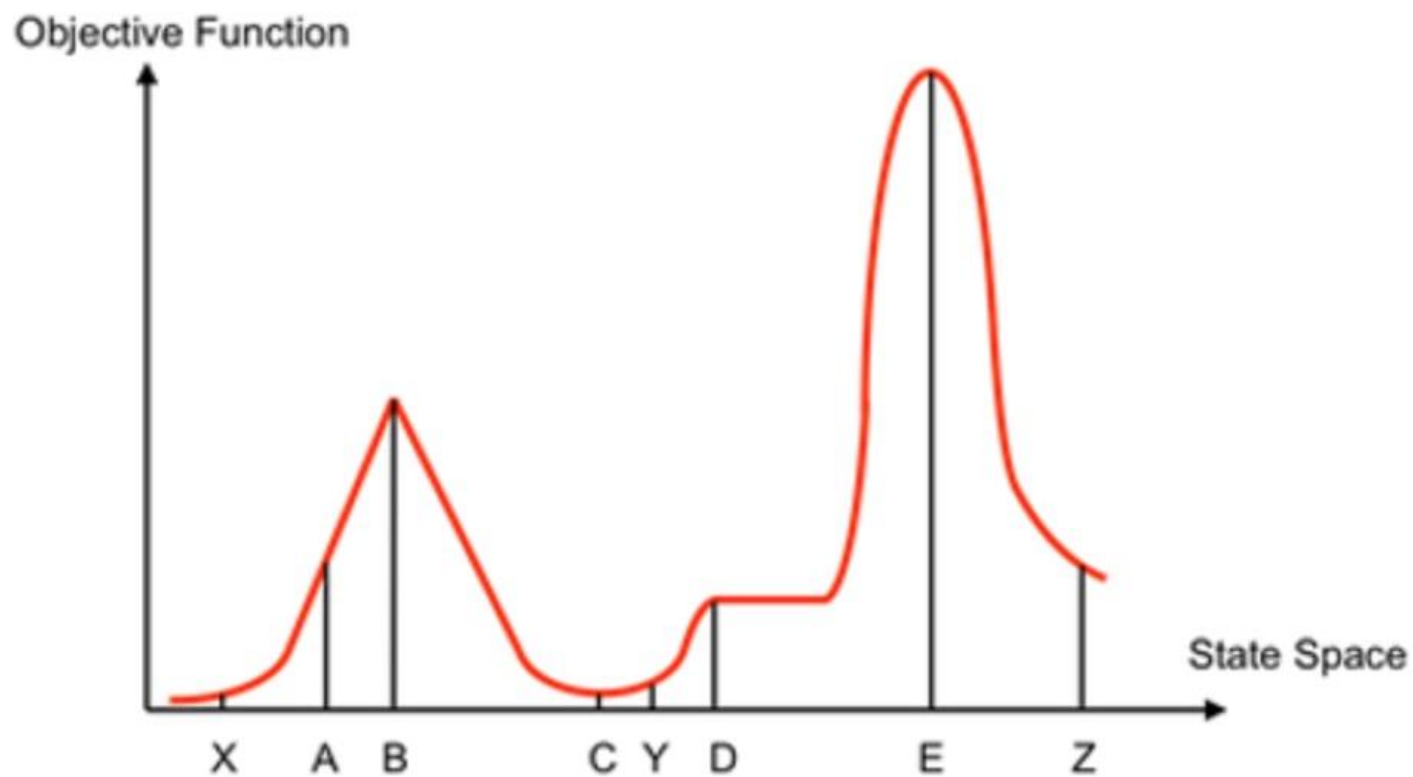
- ◆ 在每一步，当前结点都会被它的最佳邻接结点所代替
- ◆ 这里，最佳邻接结点就是**启发式代价h值**最低的邻接结点。

Hill-climbing Search 爬山搜索

- ◆ 爬山法是一种**迭代**算法：开始时选择问题的一个任意解，然后递增地修改该解的一个元素，若得到一个更好的解，则将该修改作为新的解；重复上述步骤直到无法找到进一步的改善。



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。



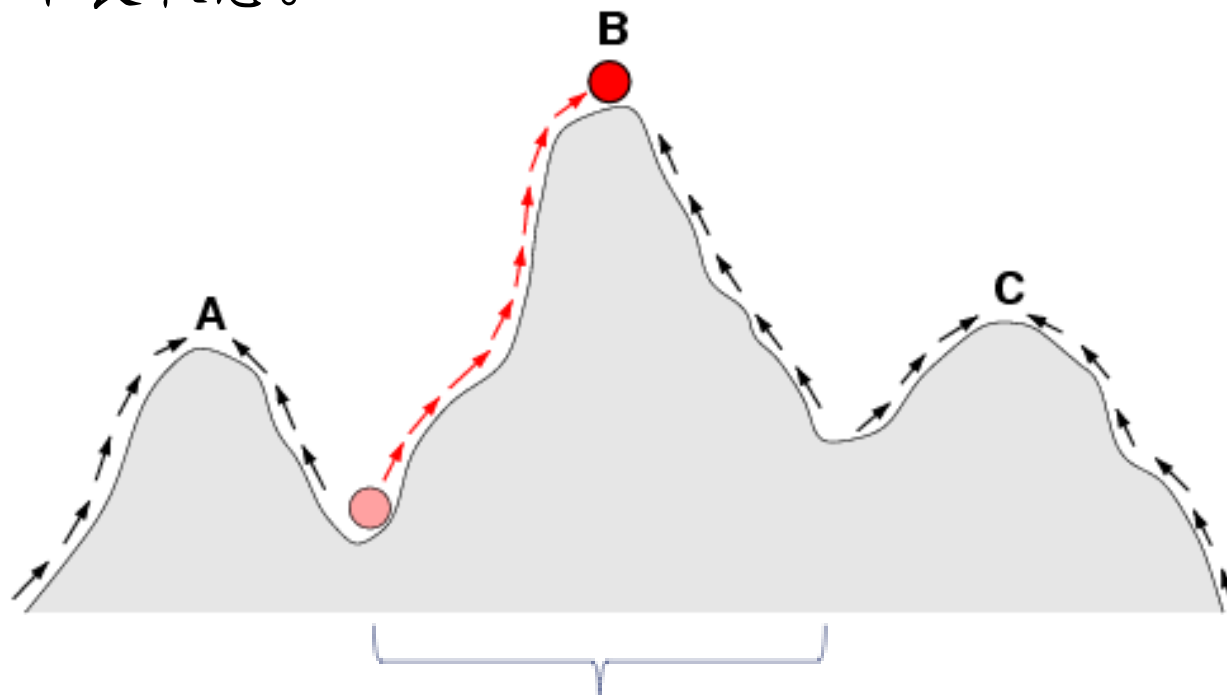
Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Hill-climbing Search 爬山搜索

- ◆ 爬山法有时被称为**贪婪局部搜索**，因为它总是选择邻居中状态**最好**的一个，而不考虑下一步该如何走。
- ◆ 爬山法往往很有效，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。

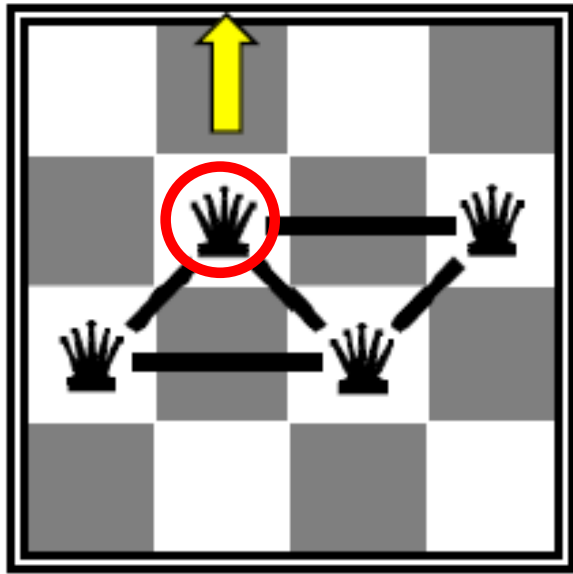


Optimal when starting in one of these states

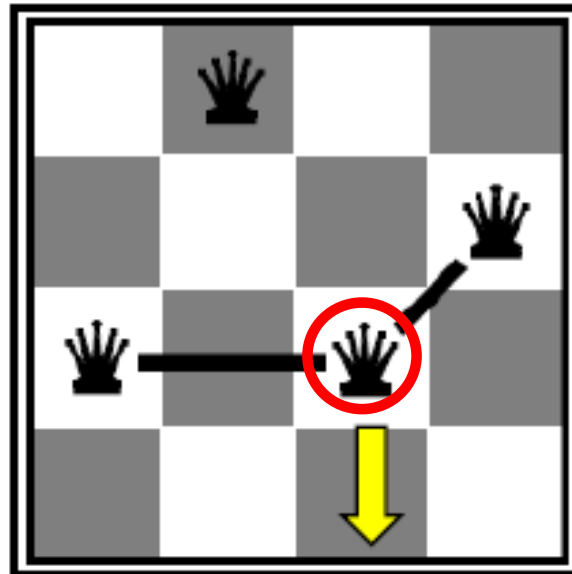
采用爬山法解决 n 皇后问题

- ◆ 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 $|i-k| \neq |j-l|$ 。

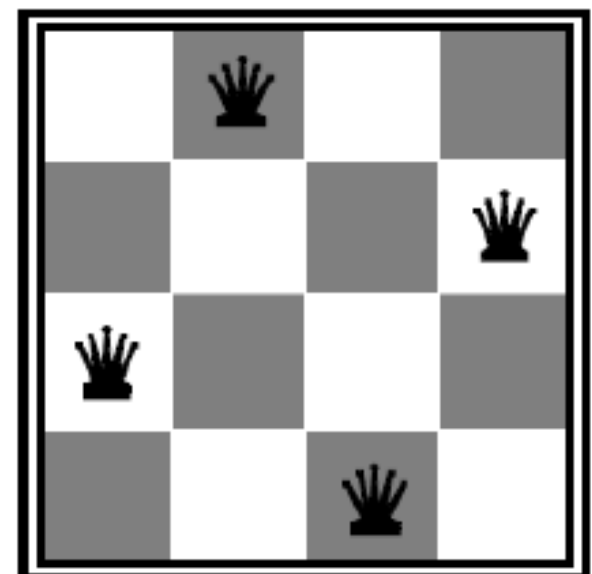
Example: 4-queens problems 4皇后问题



(a) $h = 5$



(b) $h = 2$

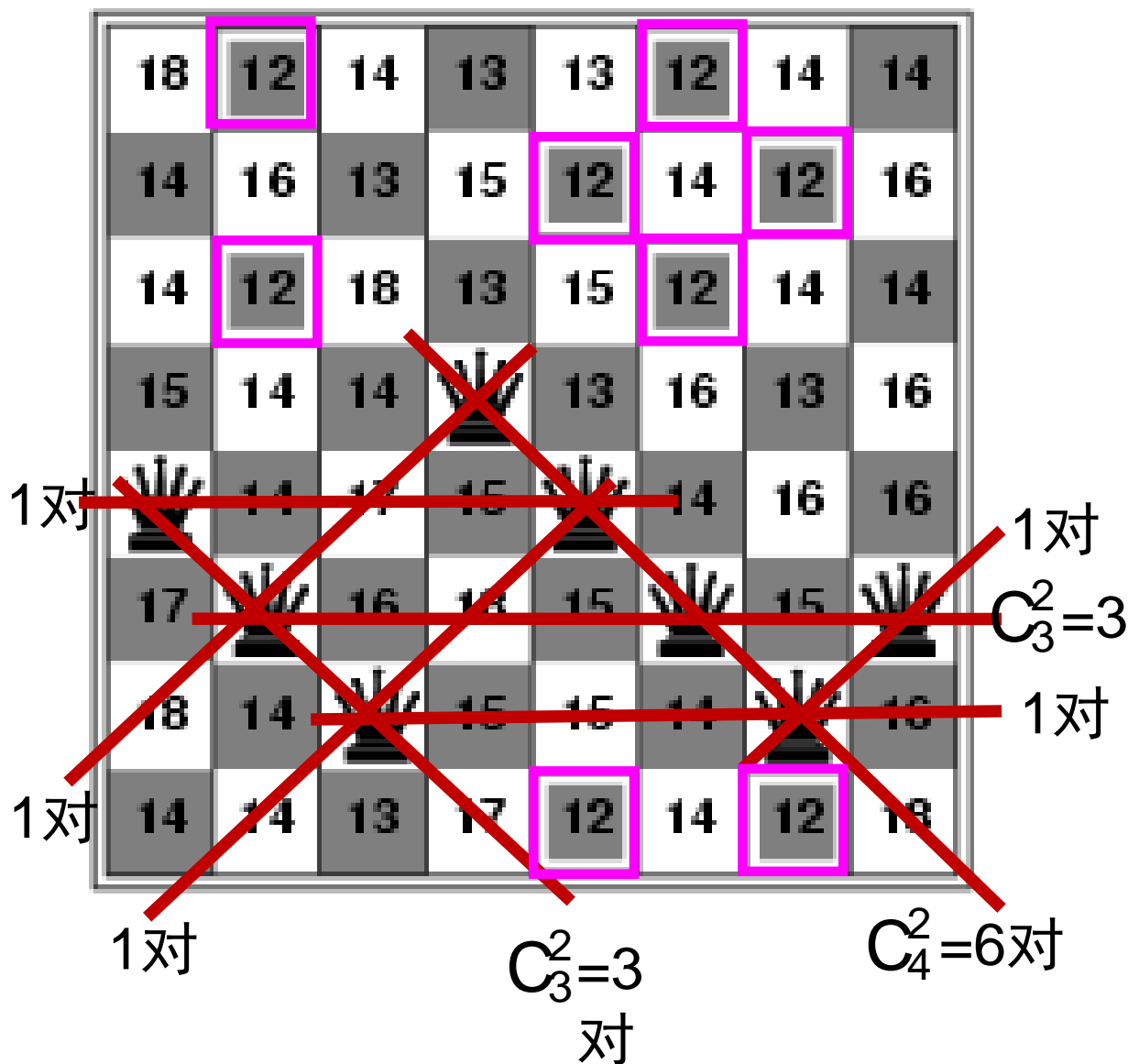


(c) $h = 0$ 解

- ◆ 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 $h=0$ ，即没有任何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。

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。

8皇后的状态图，计算启发函数h的值



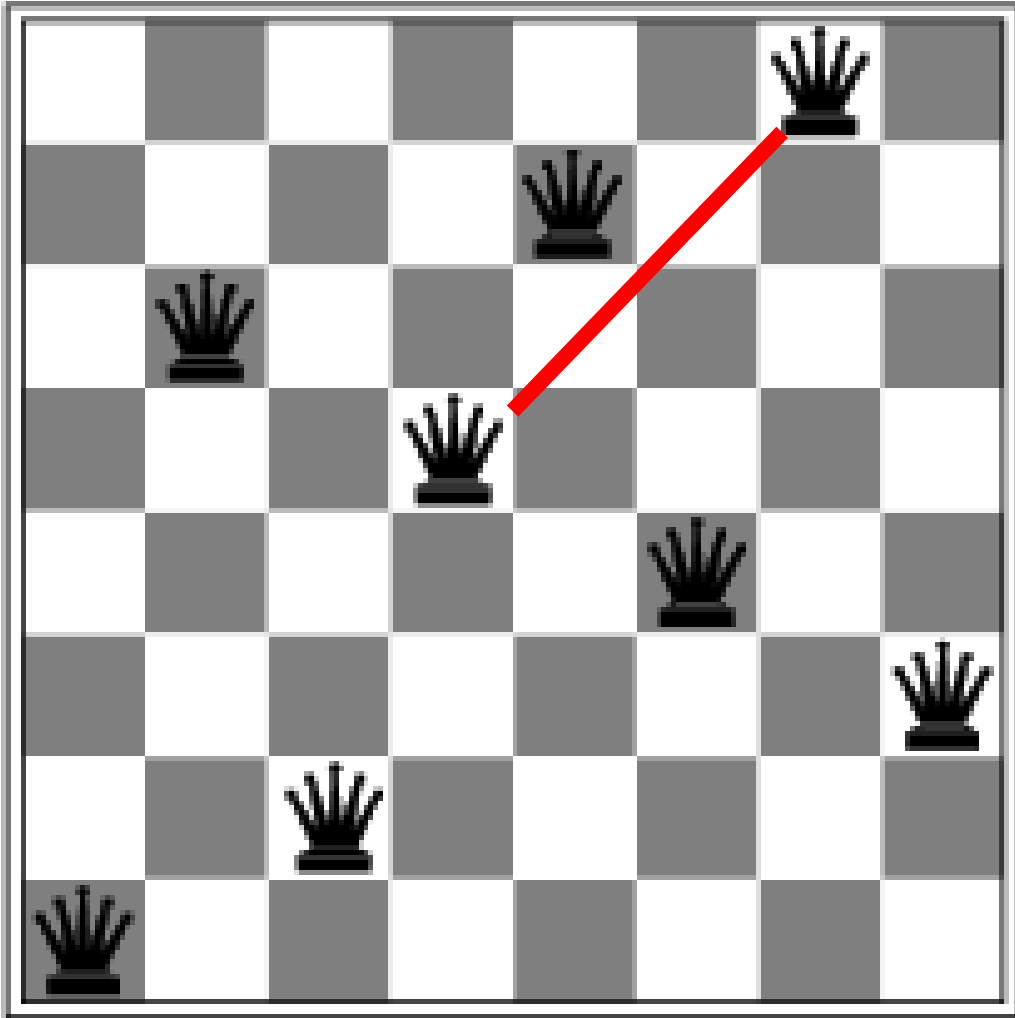
- ◆ h = 形成相互攻击的皇后对的数量, $h = 17$
- ◆ 当前状态的启发式代价评估 $h = 17$, 方格中显示的数字表示将这一列中的皇后移到该方格而得到的后继的 h 值。
- ◆ 最佳移动在图中做了标记, 即粉色框格.
- ◆ 如果有多个后继同是最小值, 爬山法会在最佳后继集合中随机选择一个进行扩展。

求解8皇后的爬山法思路

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

- (1) 在当前状态中，从若干个最佳后继中随机挑选一个，将该列的皇后移到此位置，
- (2) 再重新计算各个方格里的h值，
- (3) 转到步骤(1)，直到得到**最优解** ($h=0$) 或无法找到比当前状态h值更小的相邻状态 (**陷入局部极值，找不到解**)。

Hill-climbing search: 8-queens problem



A local minimum with $h = 1$

- ◆ 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，爬山法陷入了局部最大值 ($h=1$)，无法找到全局的最优解（即 $h=0$ ），即爬山法是不完备的。

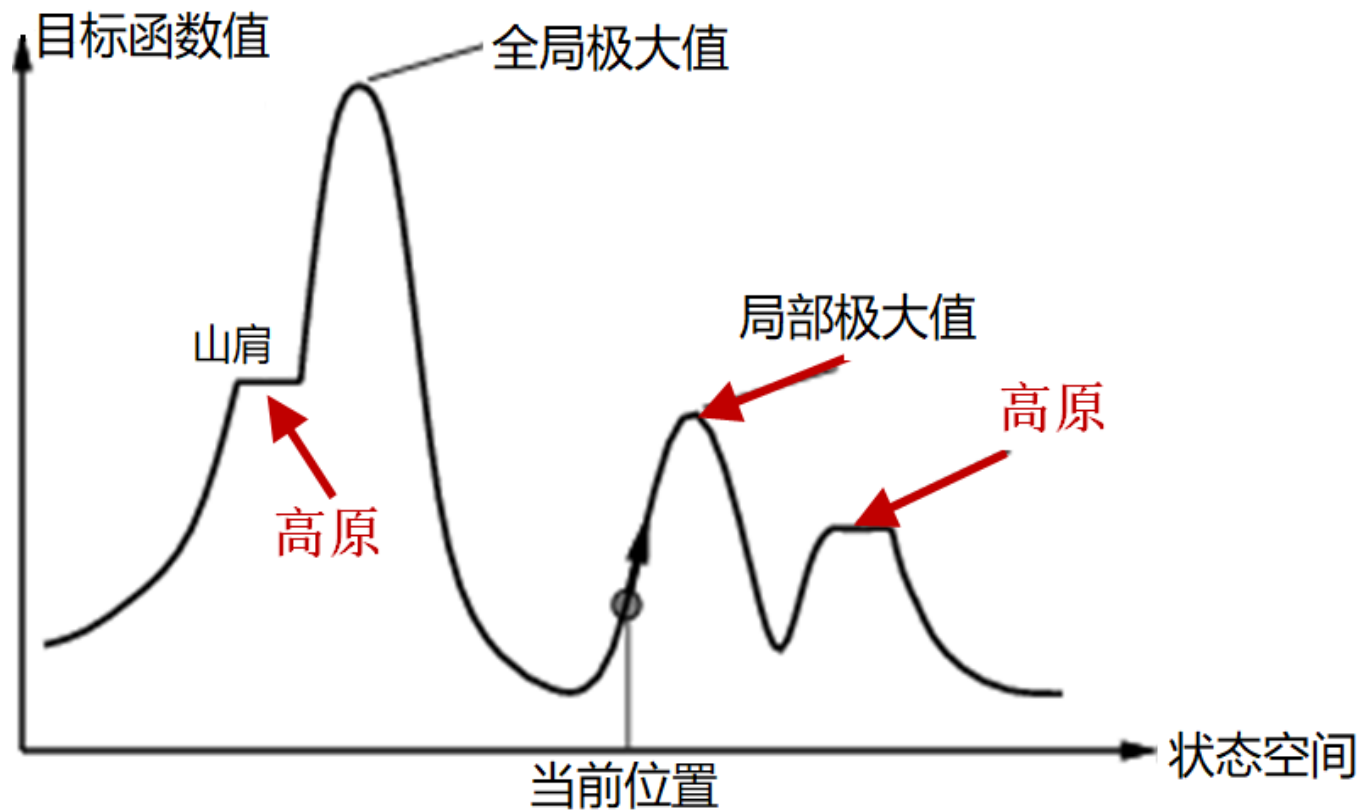
Problem of Hill-Climbing

- ◆ 贪婪算法**很难处理陷入局部极大值**的情况。
- ◆ 在这种情况下，爬山法都会到达无法再取得进展的地点。
- ◆ 从随机生成的八皇后问题开始，采用最陡上升的爬山法，其中**86%**的情况下会被卡住，只有14%的问题实例能求得解。
- ◆ 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，因为被局部极大值卡住而**找不到目标**。

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) **局部最大值**：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。

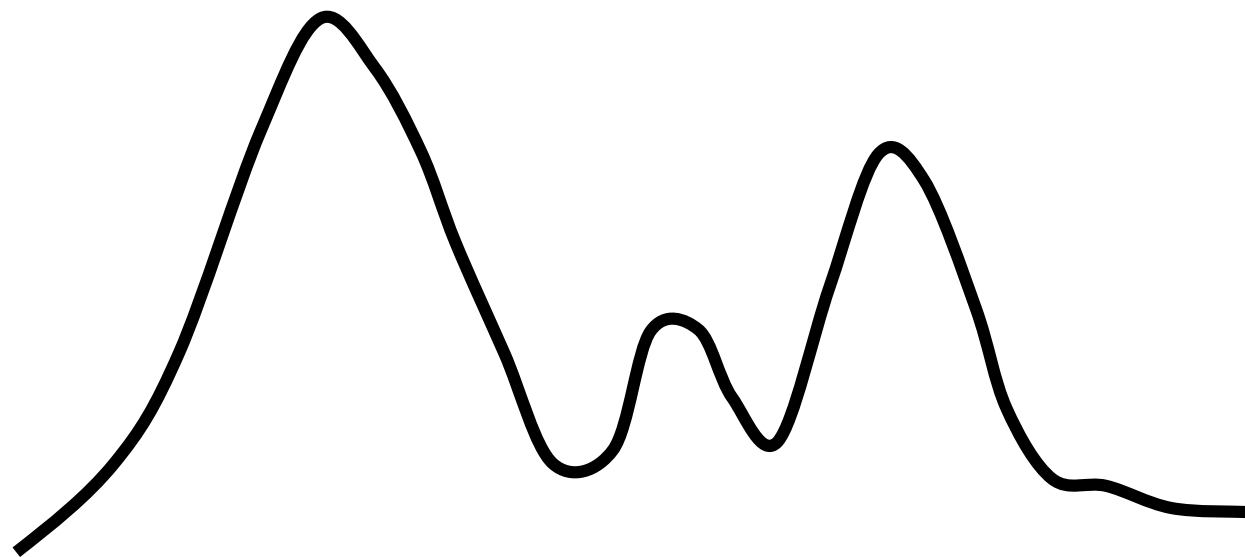
◆为什么山脊会使爬山法困难？

- 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。



存在的问题

- 局部最优问题

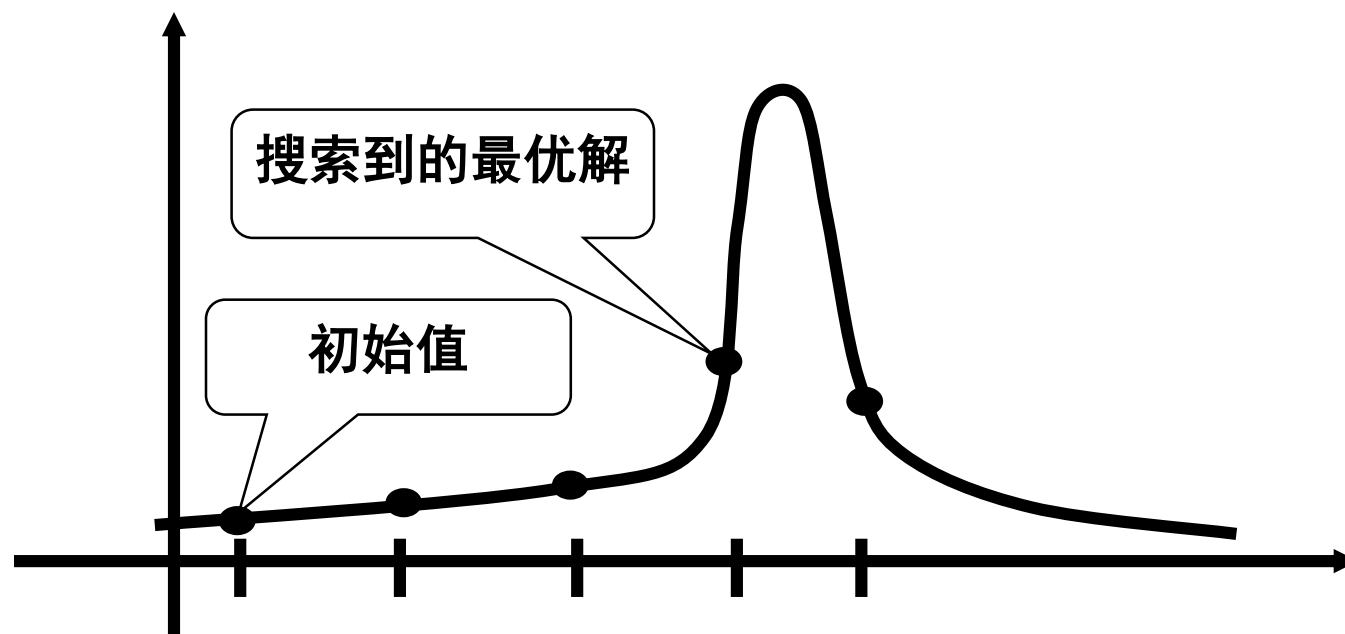


解决方法

- 每次并不一定选择邻域内最优的点，而是依据一定的概率，从邻域内选择一个点，指标函数优的点，被选中的概率比较大，而指标函数差的点，被选中的概率比较小。

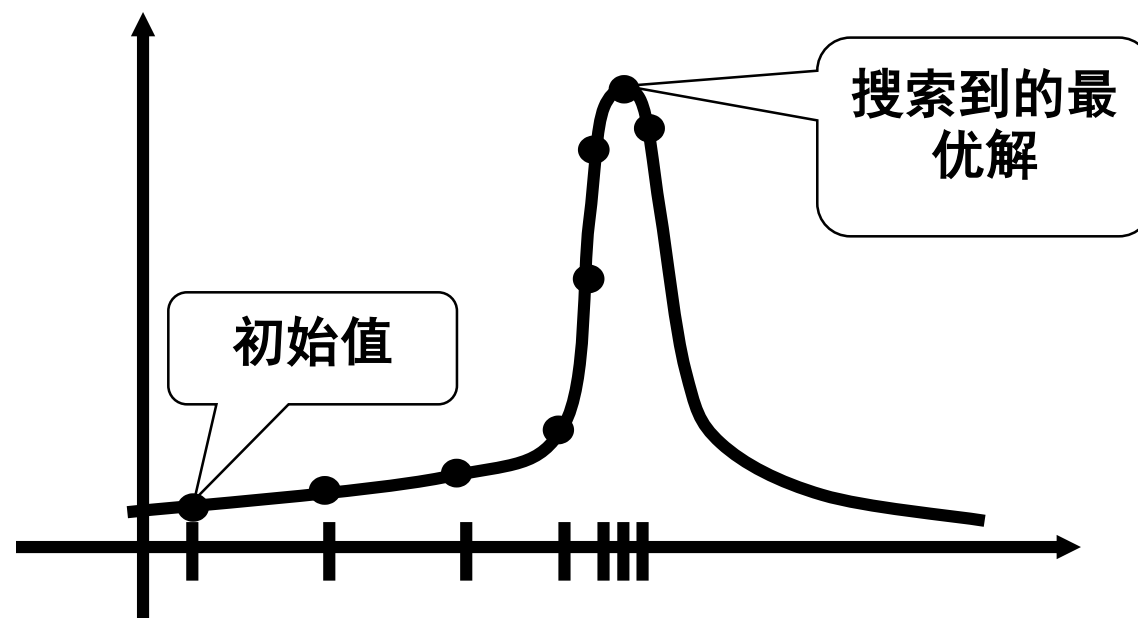
存在的问题

- 步长问题



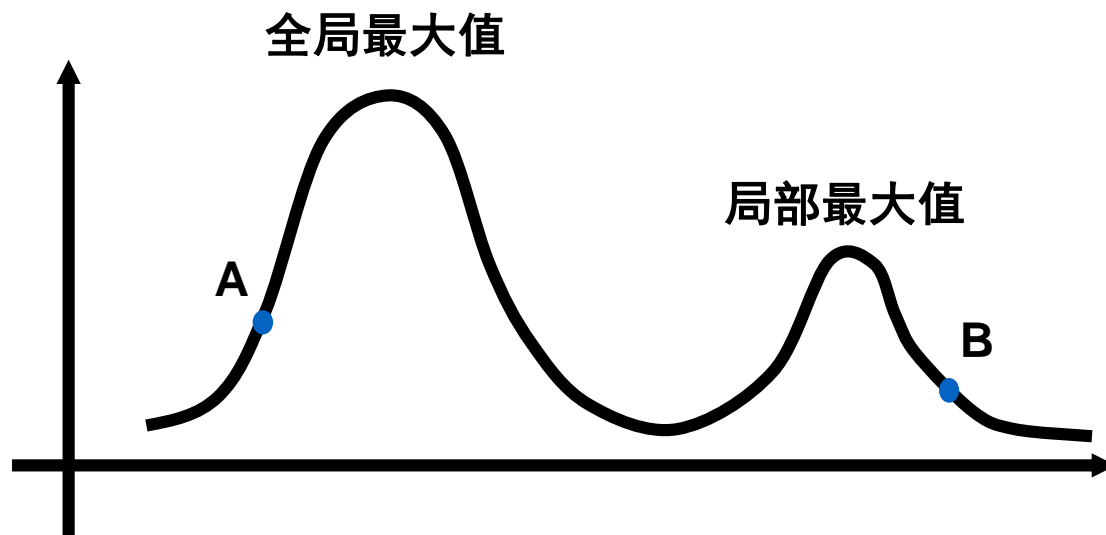
解决方法

- 变步长



存在问题

- 起始点问题



解决方法

- 随机的生成一些初始点，从每个初始点出发进行搜索，找到各自的最优解。再从这些最优解中选择一个最好的结果作为最终的结果。

Variants of Hill-climbing 爬山法的变型

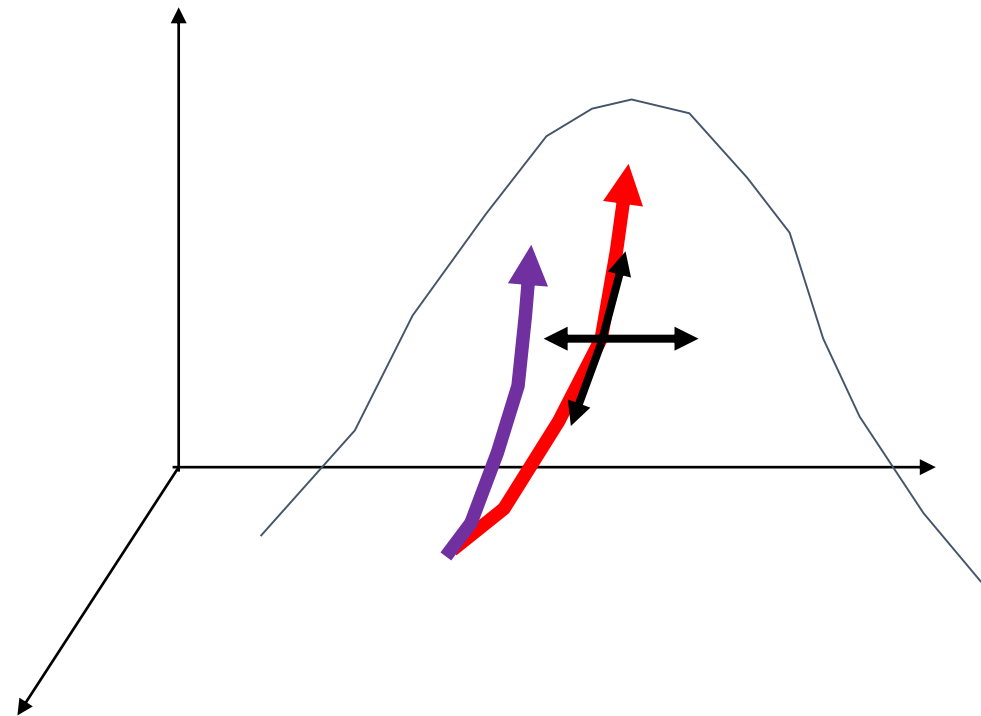
- ✓ **随机爬山法(Stochastic hill climbing)** :在向上移动的过程中, 随机地选择下一步, 被选中的概率可能随向上移动的陡峭程度的不同而变化。收敛速度通常更慢。仍然不完备, 还会被局部极值卡住。
- ✓ **随机重启爬山法(Random-restart hill climbing)**: 随机生成一个初始状态, 开始搜索, 执行一系列这样的爬山搜索, 直到找到目标为止。它几乎是完备, 概率逼近1, 因为最终它将生成一个目标状态作为初始状态。
 - 如果每次爬山搜索成功的概率为 p , 则重启需要的期望值是 $1/p$ 。
 - 对于八皇后问题, 随机重启爬山法是有效的。有300 万个皇后, 此方法找到解的时间不超过1分钟。
 - 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。

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**，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。



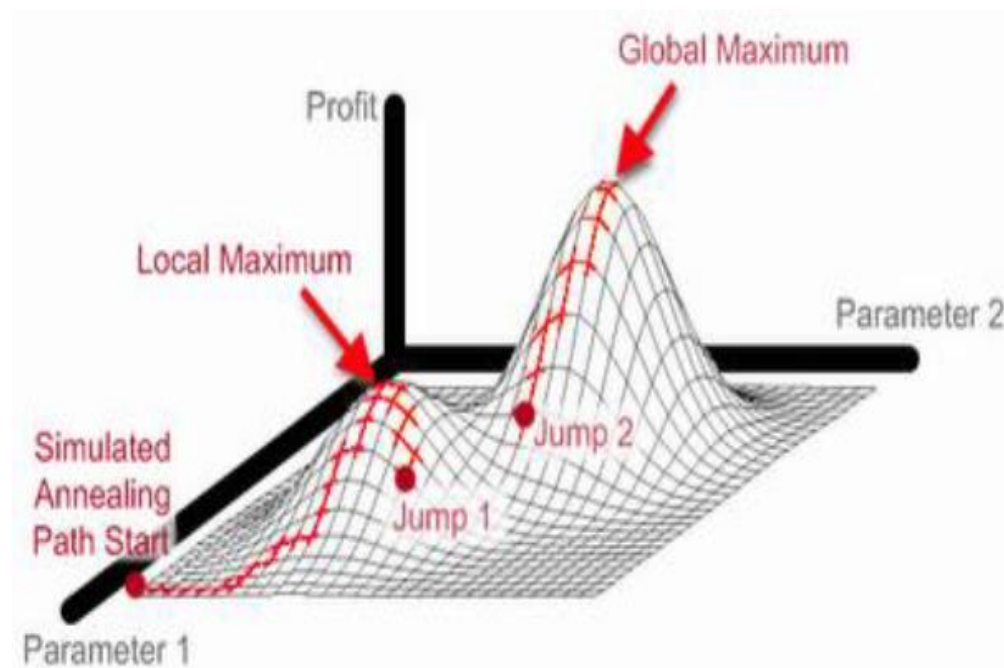
随机重启爬山法

◆ 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。

Simulated Annealing 模拟退火

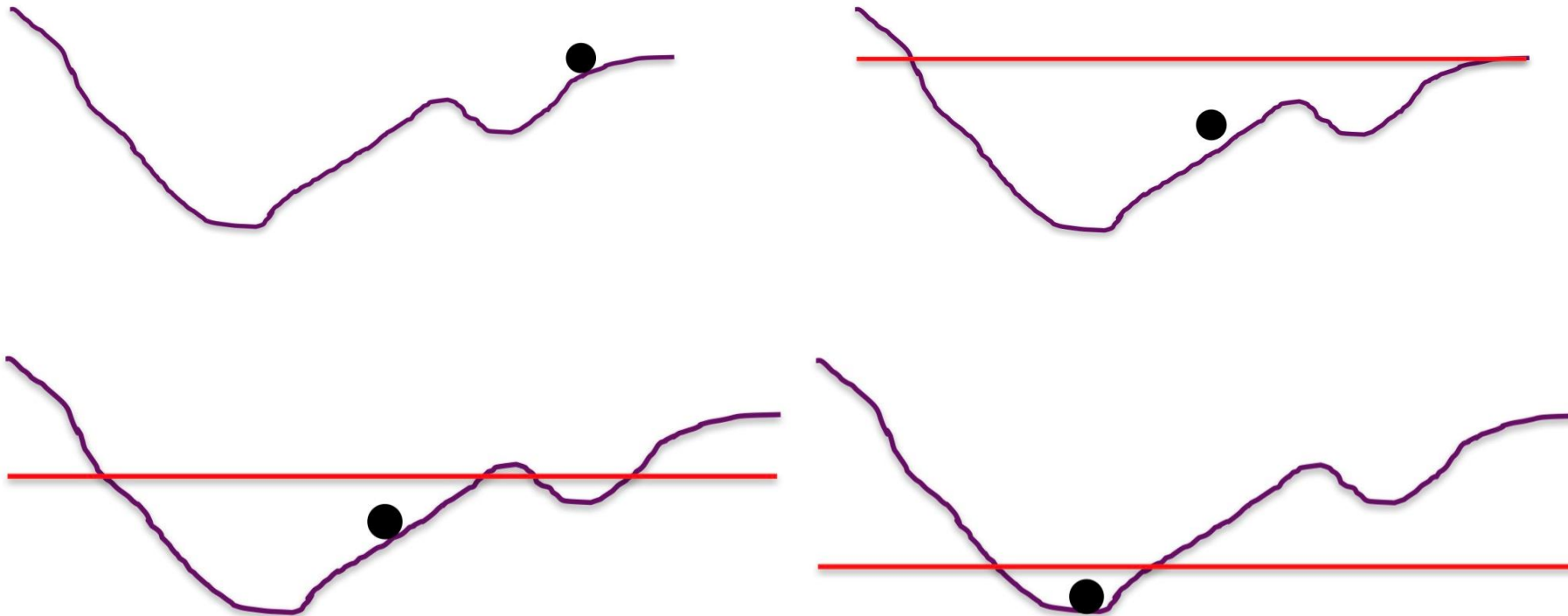
- ◆ 爬山法搜索从来不“下山”，即不会向值比当前结点低的（或代价高的）方向搜索，它肯定是不完备的，理由是可能卡在局部极大值上。
- ◆ 与之相反，纯粹的随机行走是完备的，但是效率极低。随机行走就是从后继集合中完全等概率的随机选取后继。
- ◆ 因此，将爬山法和随机行走以某种方式结合，同时得到效率和完备性的想法是合理的。模拟退火就是这样的算法。
- ◆ 在冶金中，退火是通过将金属和玻璃加热到高温，然后逐渐冷却，使材料达到低能结晶状态，从而使金属和玻璃回火或硬化的过程。



模拟退火

A Physical Analogy(minimization not max):

- Imagine letting a ball roll downhill on the function surface
- Now shake the surface, while the ball rolls,
- Gradually reducing the amount of shaking



To avoid being stuck in a local maxima, it tries randomly (using a probability function) to move to another state, if this new state is better it moves into it, otherwise try another move... and so on.

模拟退火的基本思路

- ◆ 避免局部极大值，允许一些“坏”的移动，但逐渐减少他们（“坏”的移动）的频率。
- ◆ 以**概率**突破局部最优，走向全局最优。

Simulated Annealing 模拟退火

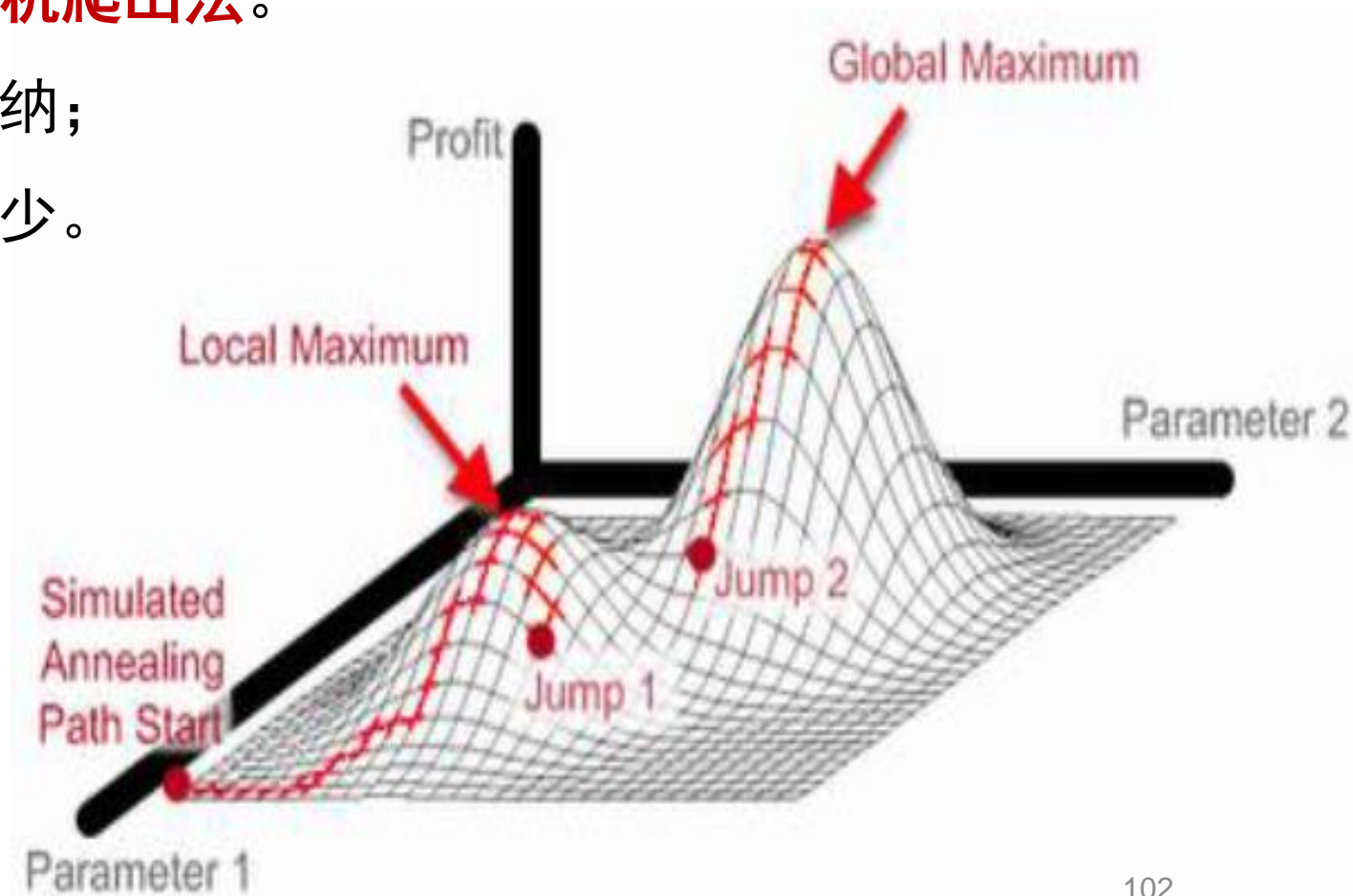
- ◆ 模拟退火算法的内层循环与爬山法类似，只是它**不选择最佳移动**，而是进行**随机移动**。
- ◆ 如果该移动能改善情况，则接受该移动；否则，算法以某个小于1 的概率接受该（变坏的）移动。
- ◆ 如果移动导致状态“变坏”，**接受概率会呈指数级下降**——根据**能量差值** ΔE 判断。
- ◆ 这个**概率也随“温度” T 的降低而下降**：开始 T 高的时候，可能允许“坏的”移动；当 T 降低时，则不可能允许“坏的”移动。
- ◆ 如果调度让温度 T 下降得足够慢，算法找到全局最优解的概率接近于1。
- ◆ 采用模拟退火算法**求解八皇后问题**，关键是设计启发函数。令启发函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

$h(n)$ = 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。

Simulated Annealing 模拟退火

- ◆ 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**。
- ◆ 在退火初期，下山移动容易被采纳；
- ◆ 随时间推移，下山的次数越来越少。



Simulated annealing search

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 的一个调度算法。

// $T=0$ 时, 说明温度已经最低, 退火结束, 则返回当前结点。
if $T = 0$ **then return** *current*

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

$\Delta E \leftarrow \text{next.VALUE} - \text{current.VALUE}$ // 计算当前结点和下一个结点之间的能量差值 ΔE 。

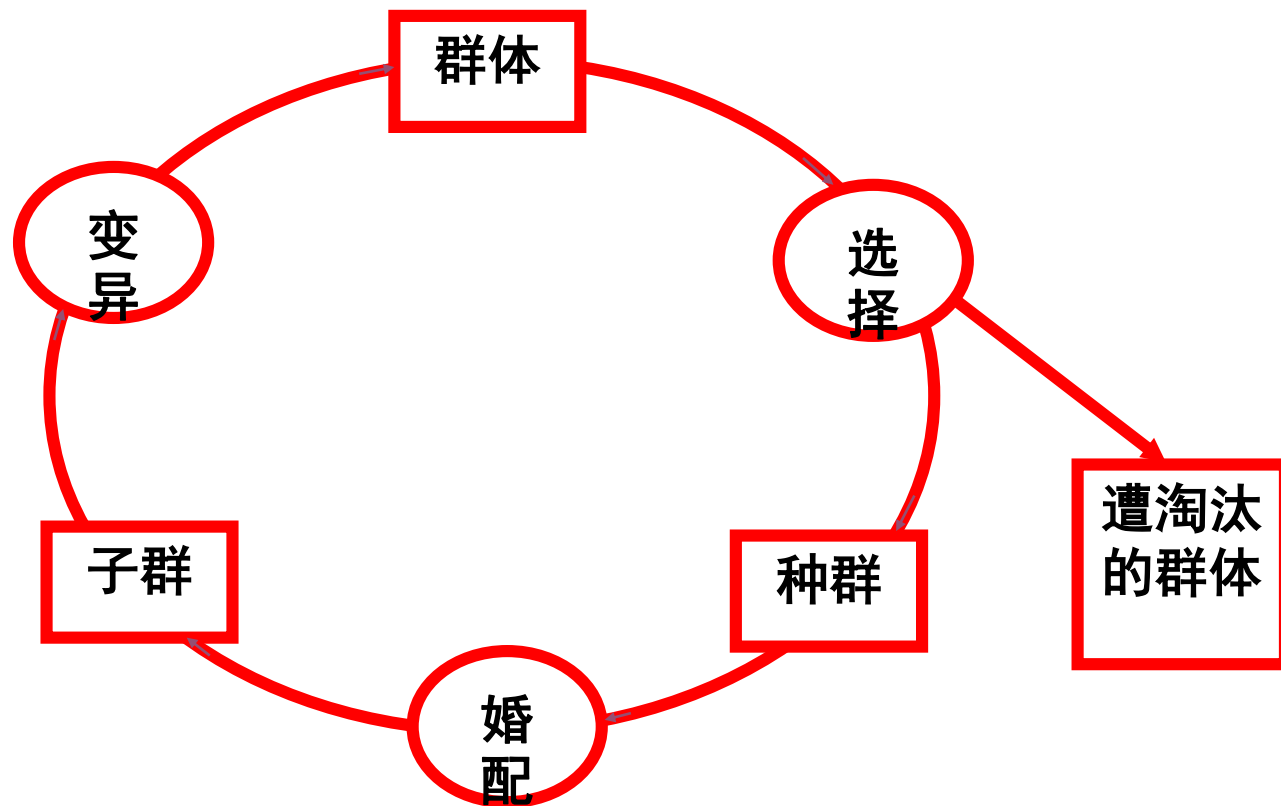
if $\Delta E > 0$ **then** *current* \leftarrow *next* // 若 $\Delta E > 0$, 说明是上山, 状态在变好, 则将
下一结点作为当前结点, 进行下一次循环。

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

// 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结点作为当前结点, 进行下一次循环。

遗传算法

- ✓ 达尔文进化论：“物竞天择、适者生存”
- ✓ 70年代由美国的密执根大学的Holland教授首先提出
- ✓ 遗传算法作为一种有效的工具，已广泛地应用于最优化问题求解之中



Genetic Algorithms 遗传算法

- ◆遗传算法是一种模仿自然选择过程的启发式搜索算法。
- ◆在遗传算法中，后继节点是由两个父辈状态的组合、而不是修改单一状态生成的。其处理过程是有性繁殖，而不是无性繁殖。
- ◆遗传算法属于**进化算法**这个大分类。
- ◆遗传算法采用自然进化所派生的技术来生成优化问题的解，例如：遗传、变异、选择、以及杂交。

遗传算法的基本概念

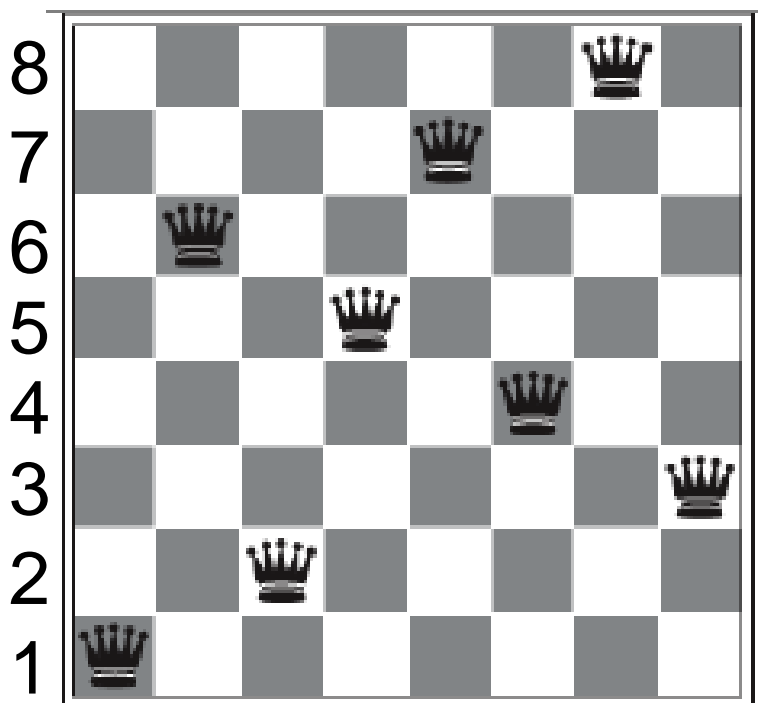
- ◆ **种群** (Population) : 是初始时给定的多个解的集合, 其中有一组 k 个随机生成的状态, 称其为**种群**。
- ◆ **个体** (Individual) : 指种群中的单个状态, 用于描述其基本遗传结构的数据结构, 表示为有限字母表上的一个字符串, 通常是0和1的字符串。
- ◆ **染色体** (Chromosome) : 指对个体进行编码后所得到的编码串。染色体中的每一位称为基因, 染色体上由若干个基因构成的一个有效信息段称为**基因组**。例如: 11011为一个染色体, 每一位上的0或1表示基因。

遗传算法的基本概念

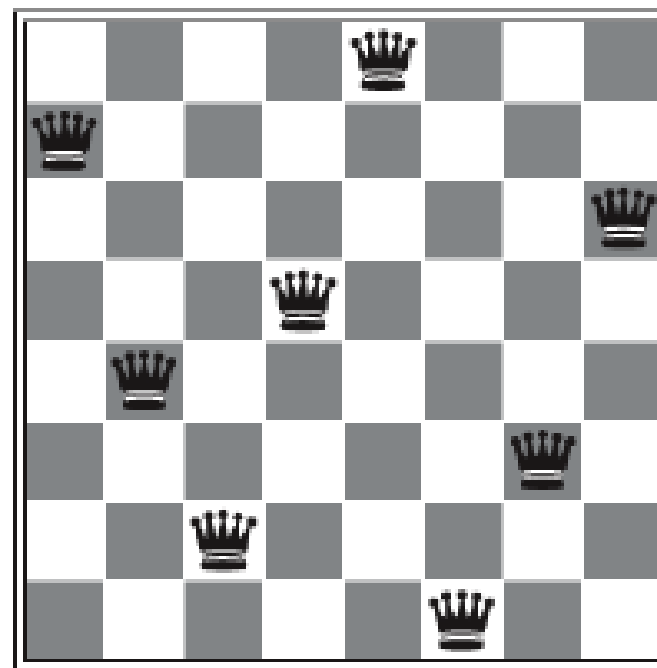
- ◆ **适应度**（Fitness） **函数**：一种用来对种群中各个个体的**环境适应性**进行度量的**函数**。其函数值是遗传算法实现**优胜劣汰**的主要依据。
- ◆ 对于好的状态，适应度函数应返回较高的值，即：**适应度越高，越好**。
- ◆ **遗传操作**（Genetic Operator）：指作用于种群而产生新的种群的操作。
- ◆ 标准的遗传操作包括以下三种基本形式：
 - 选择（Selection）
 - 交叉（Crossover）
 - 变异（Mutation）

Example: 8-queens problem 8皇后问题

某8皇后状态需要指明8个皇后的位置，每列有一个皇后，其状态可用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。



1 6 2 5 7 4 8 3



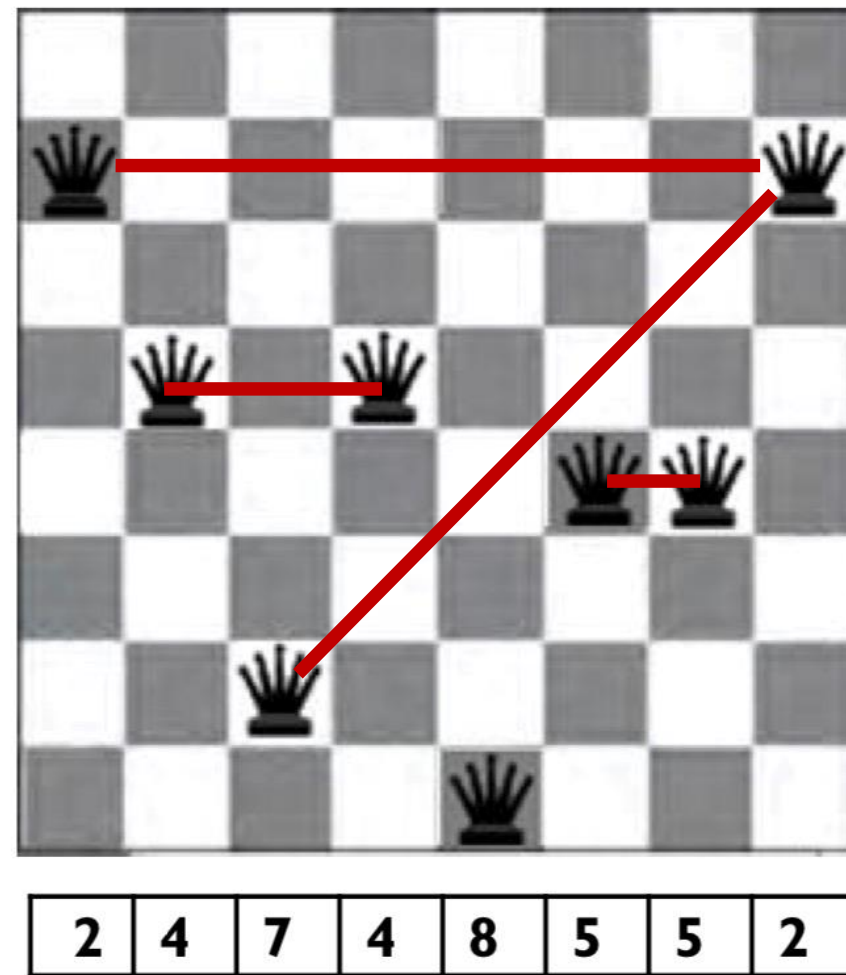
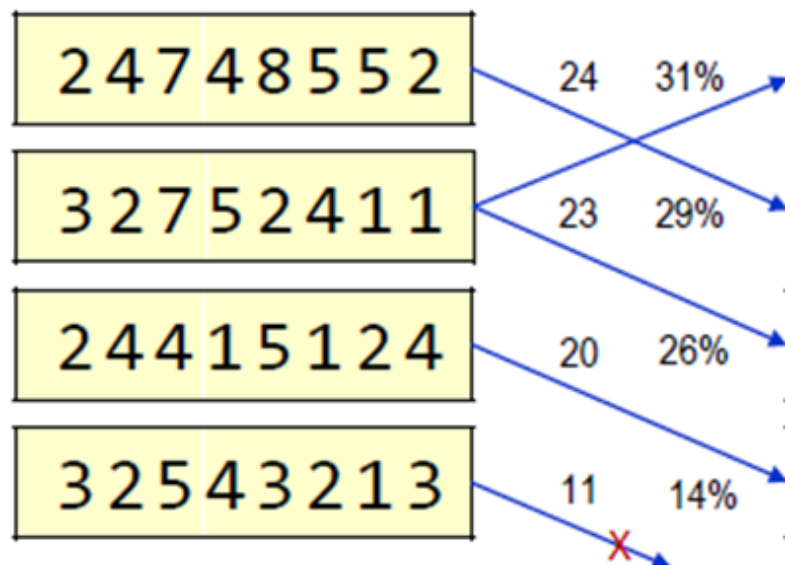
7 4 2 5 8 1 3 6

Example: 8-queens problem 8皇后问题

- ◆在八皇后问题中，我们用**不相互攻击的皇后对的数目来表示适应度**。最优解的适应度是28。

$$C_8^2=28$$

- ◆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。

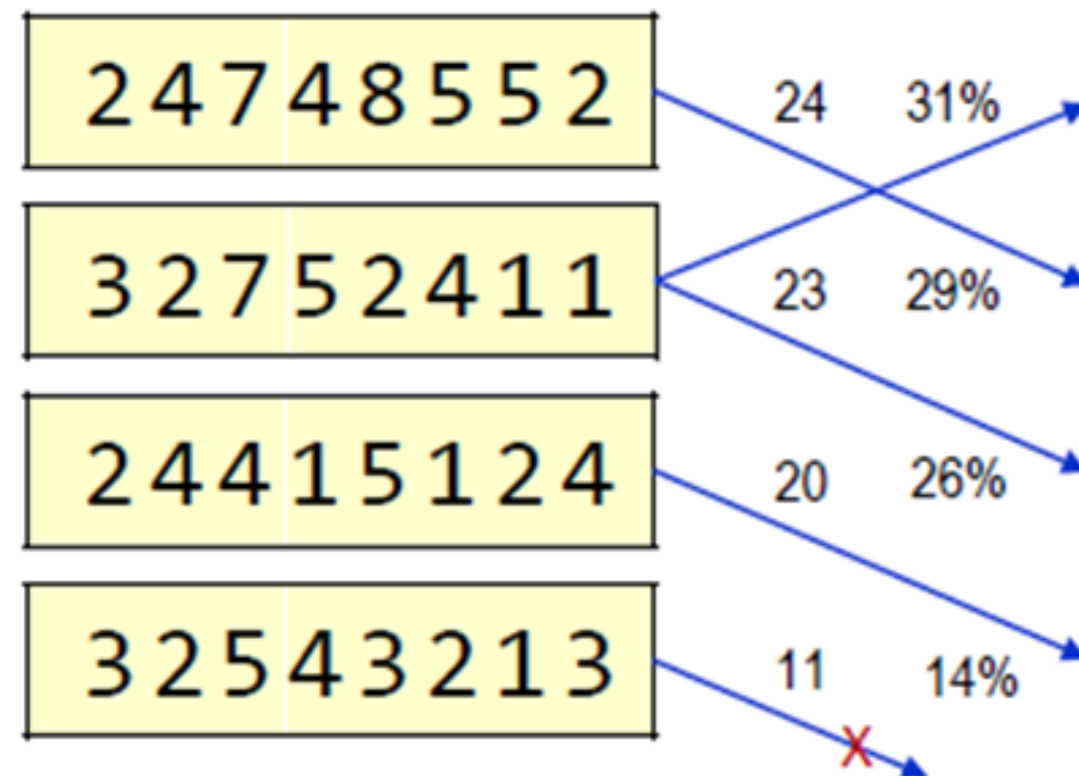


Example: 8-queens problem 8皇后问题

- ◆在八皇后问题中，我们用**不相互攻击的皇后对的数目来表示适应度**。最优解的适应度是28。

$$C_8^2=28$$

- ◆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。



- ◆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%

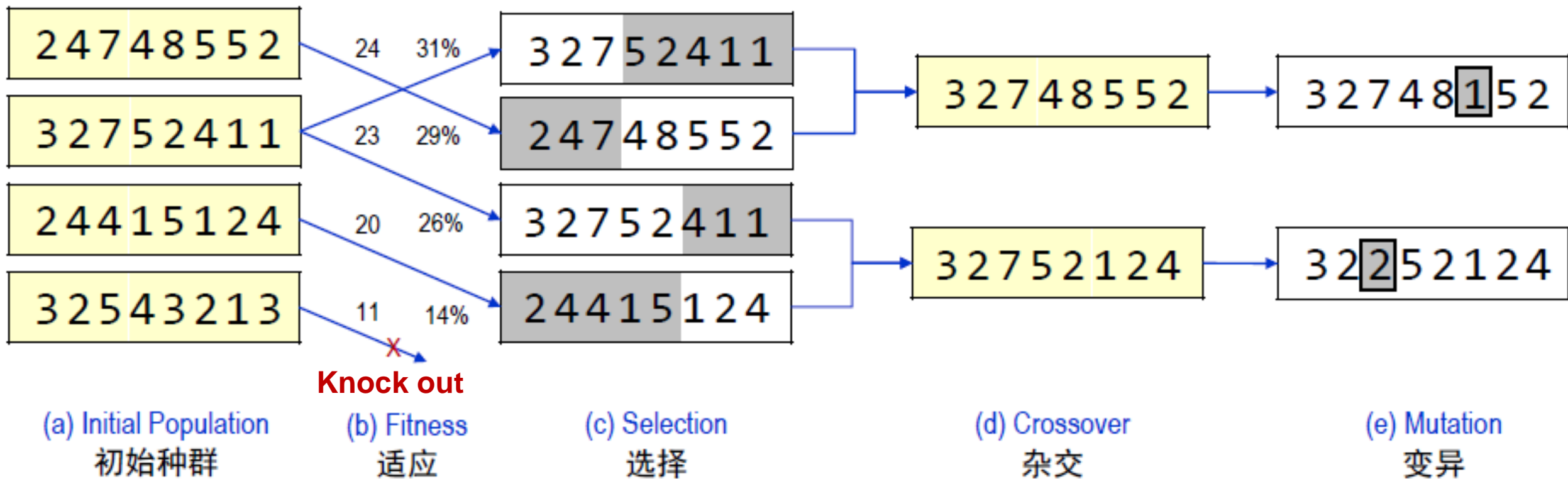


图1. 数字串表示8皇后的状态(a)为初始种群 (b)计算适应度函数 (c) 选择并配对的结果
(d) 杂交产生的后代 (e) 变异的结果

- ◆ 按概率随机地选择两对进行繁殖，第2个个体被选中两次，而第4个个体一次也没被选中。
- ◆ 随机选择一个位置作为**杂交点**，第一对的在第3位数字之后，第二对的在第5位数字之后。
- ◆ 图 (e) 中每个位置都会按照某个小的独立概率随机**变异**。
- ◆ 在八皇后问题中，这相当于**随机地选取一个皇后并把它随机地放到该列的某一个方格里**。

Example: 8-queens problem 8皇后问题

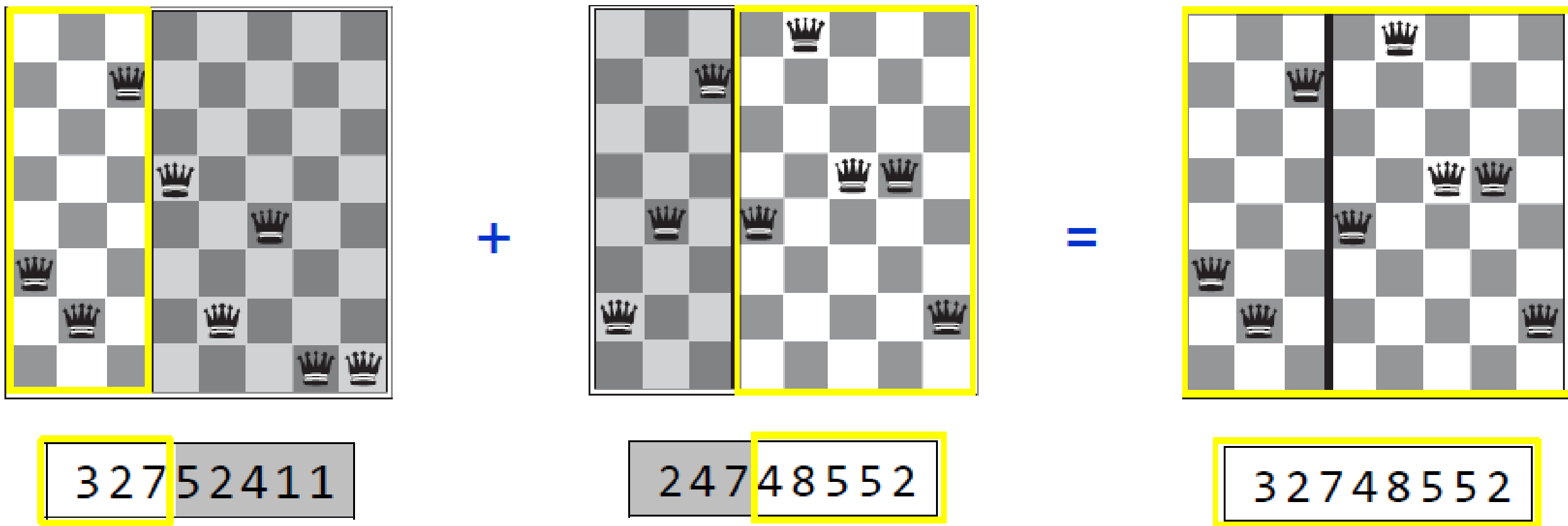


图2. 这三个8皇后的状态分别对应于“选择”中的两个父辈和“杂交”中它们的后代。
阴影的若干列在杂交步骤中被丢掉，而无阴影的若干列则被保留下来。

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
           // 个体适应度函数
  repeat
    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
```

Thank you !