

Foundation of Artificial Intelligence

人工智能基础

李翔宇

软件学院

Email: lixiangyu@bjtu.edu.cn

人工智能第三次课2025



长按识别二维码

用产生式表示：如果一种微生物的染色斑是革兰氏阴性，其形状呈杆状，病人是中间宿主，那么该生物是绿杆菌的可能性有6成。

IF 本微生物的染色斑是革兰氏阴性 \wedge 本微生物的形状呈杆状 \wedge 病人是中间宿主, THEN 该生物是绿杆菌 (0. 6) 。

使用 RDF 三元组表示事实知识：葡萄牙籍的C罗效力于西班牙的皇家马德里足球队。

（皇家马德里足球队，属于，西班牙）

（C罗，效力，皇家马德里足球队）

（C罗，国籍，葡萄牙）

搜索策略

3.1 图搜索策略

3.2 盲目搜索

3.2.1 深度优先搜索 (DFS)

3.2.2 宽度优先搜索 (BFS)

3.3 启发式搜索

3.3.1 A Search, 即最佳优先搜索

3.3.2 A* Search

Problem solving in AI 人工智能中的问题求解

- 现实中很多智能问题有不同表现，其背后共同的问题是什么？
- 问题求解 (Problem Solving) 是人工智能第一个大的成就。许多的问题求解工作可以通过搜索得到解决。

走迷宫

如何走迷宫？



表示一个求解问题

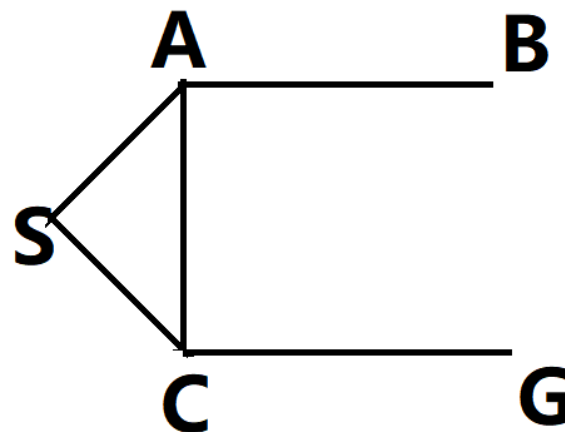
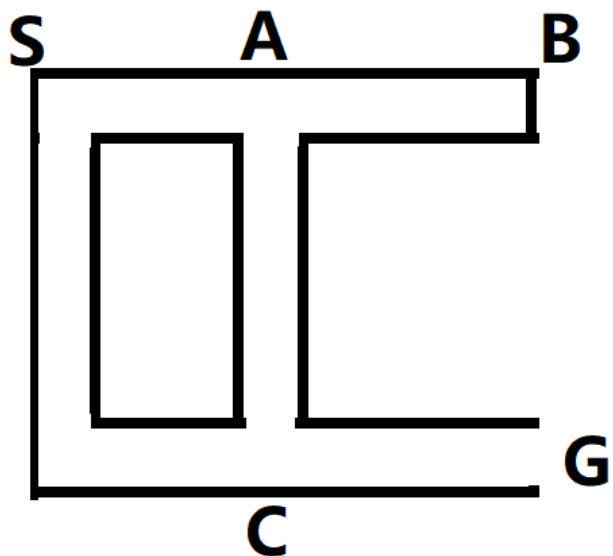
搜索技术的思路

- 表示一个求解问题
- 给出一个搜索算法

为什么研究走迷宫？

简单、直观

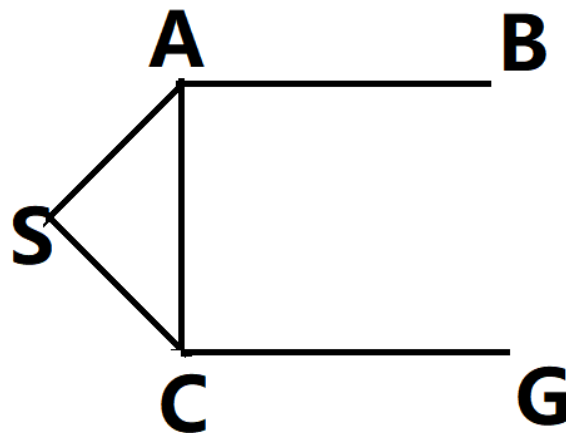
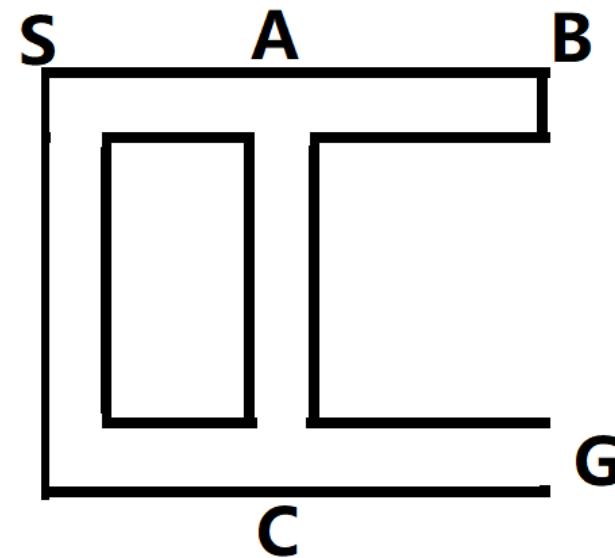
具有代表性



表示一个求解问题

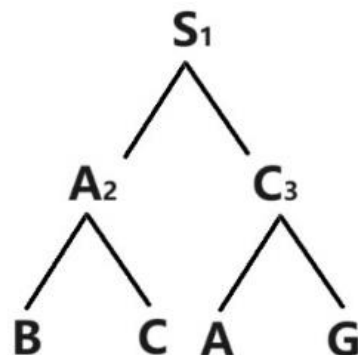
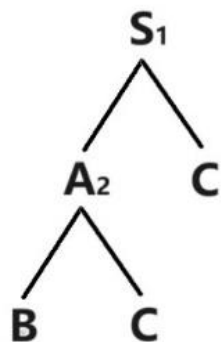
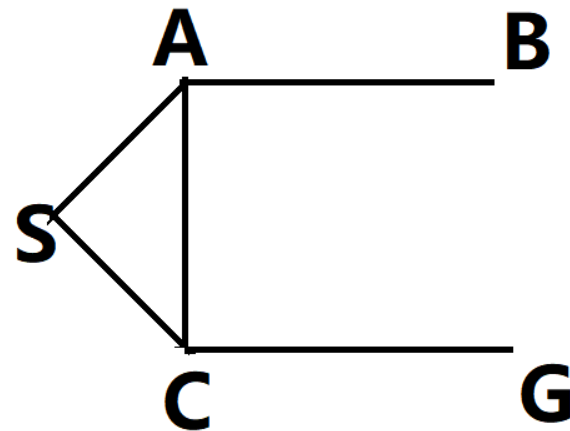
搜索技术的思路

- 每个节点
- 节点之间的连线
- 用图 (graph) (V, E) 表示
- 节点集合 $V = \{S, A, B, C, G\}$
- 边的集合是 $E = \{(S, A), (S, C), (A, B), (A, C), (C, G)\}$



给出一个搜索算法

- 算法：在什么情况下该做什么
- 一个具体的算法：搜索过程，节点顺序号
- 树结构
- 宽度优先搜索算法



3.1 图搜索策略

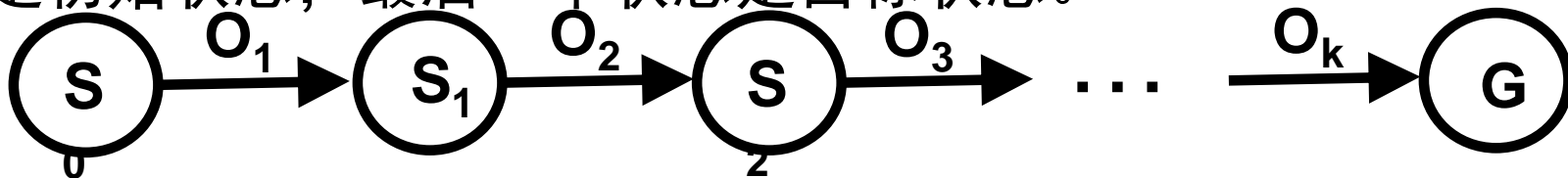
针对搜索问题，可以采用**状态空间知识表示法**来表示问题。

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**。

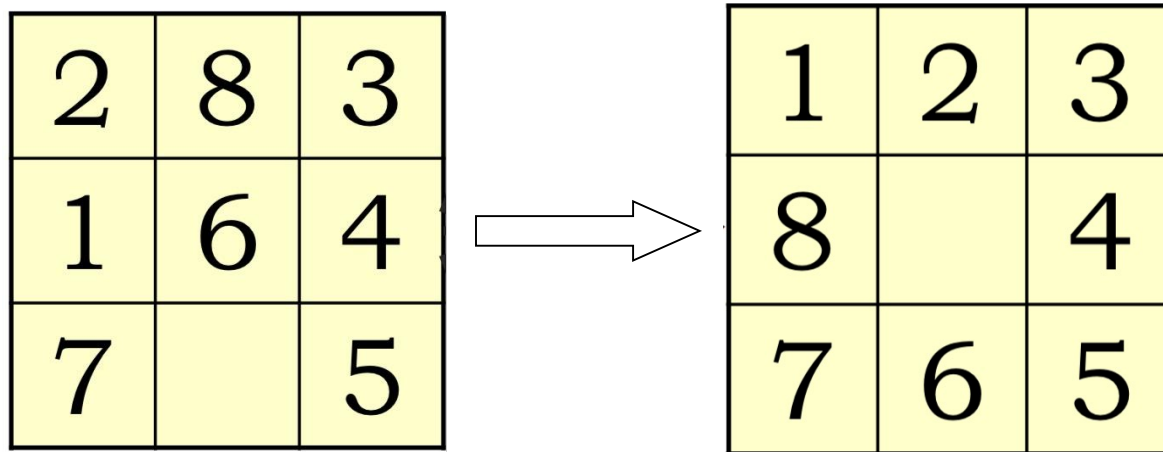
3.1 图搜索策略

- ◆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？



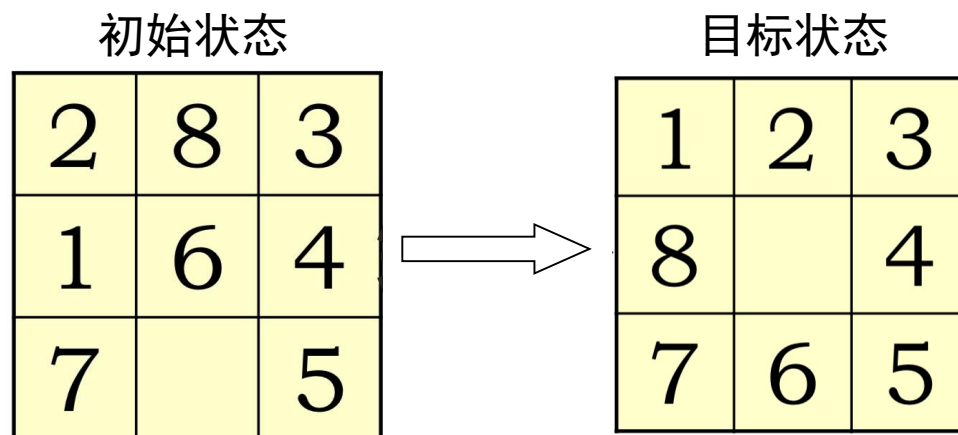
(a) Initial state

(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。

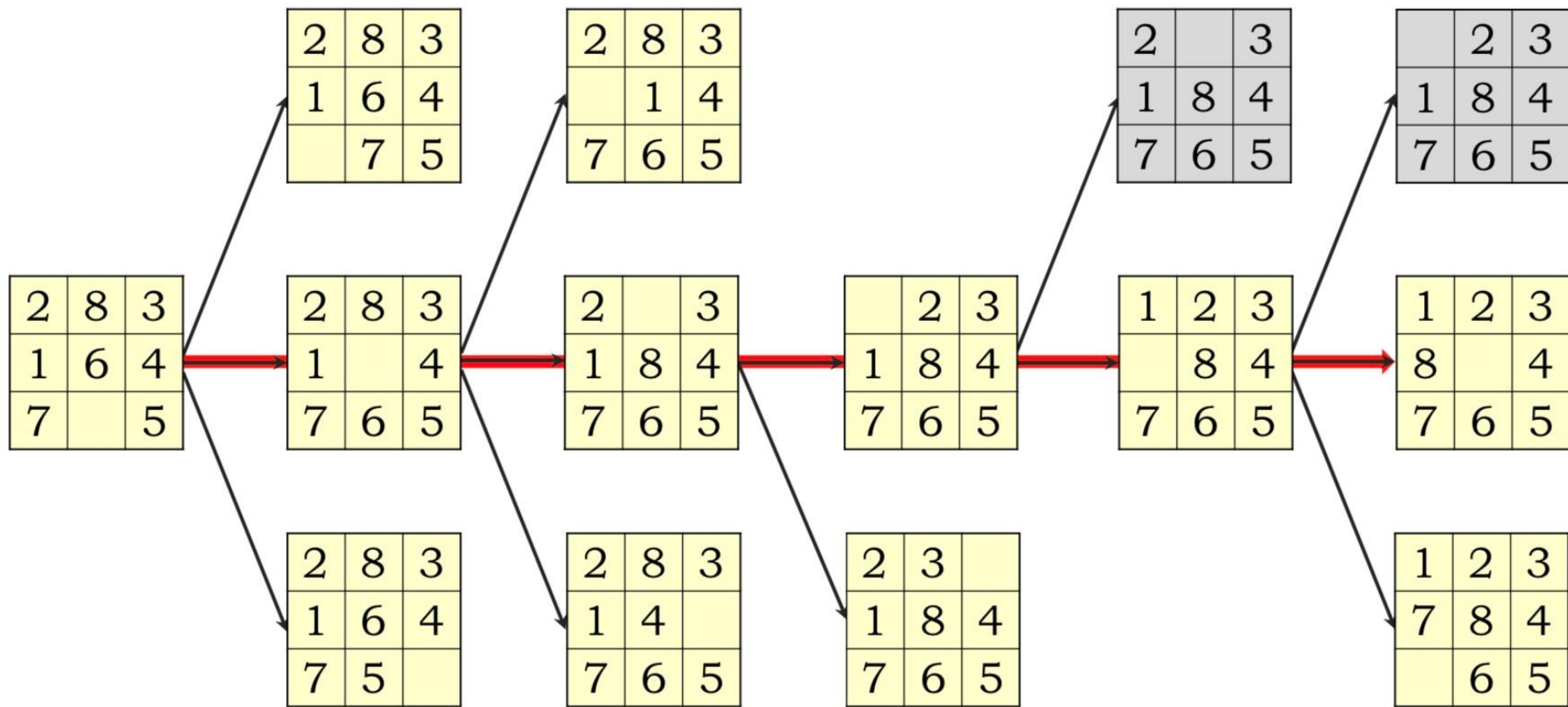
Up: 将空格向上移, if 空格不在最上一行
Down: 将空格向下移, if 空格不在最下一行
Left: 将空格向左移, if 空格不在最左一列
Right: 将空格向右移, if 空格不在最右一列



Searching for Solutions 搜索解

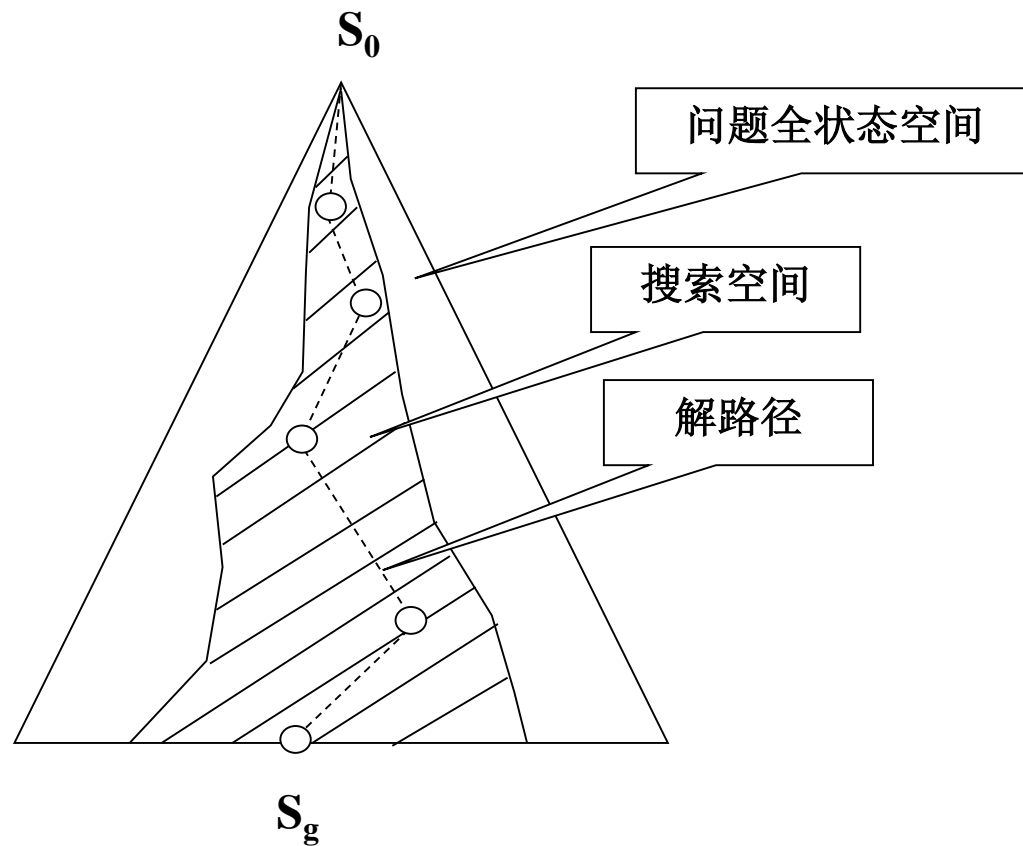
- 从初始状态到目标状态的路径就是问题的解
- 问题求解就是在状态图中搜索这样一条路径

怎么做？



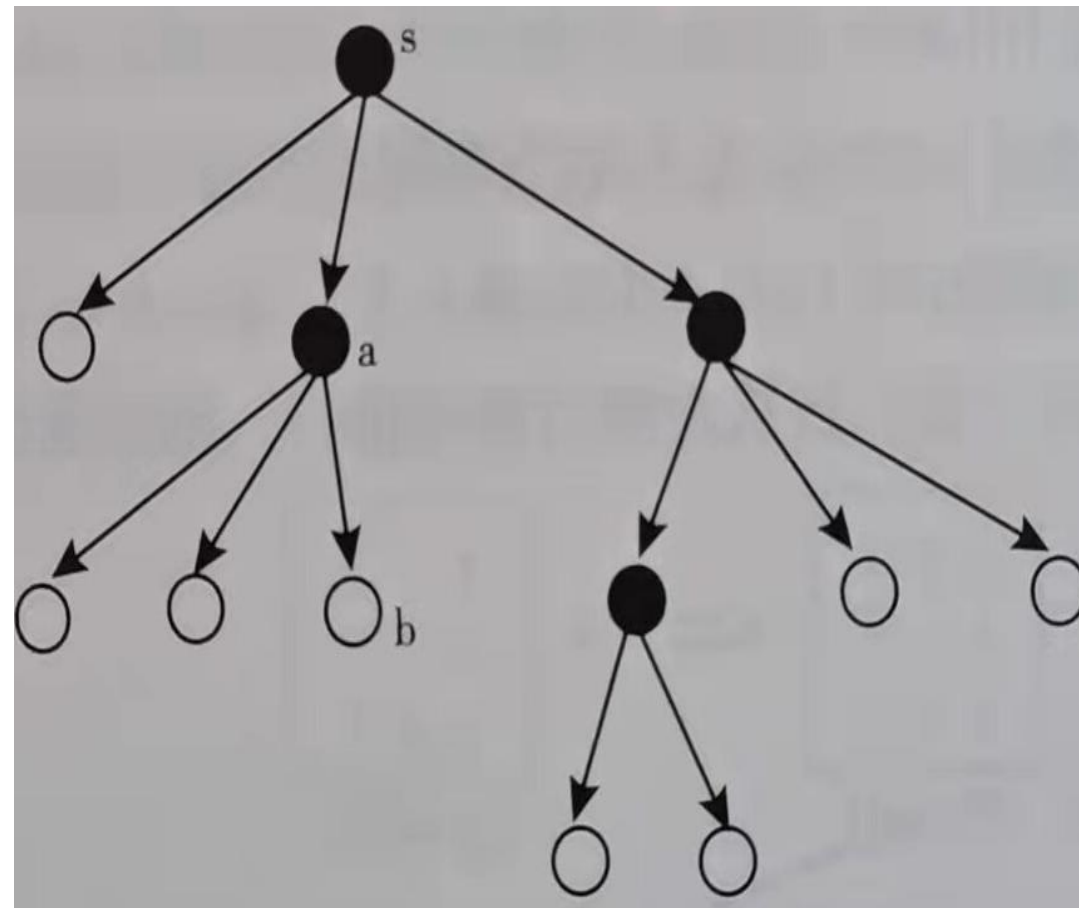
3.1 图搜索策略

- ◆ 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

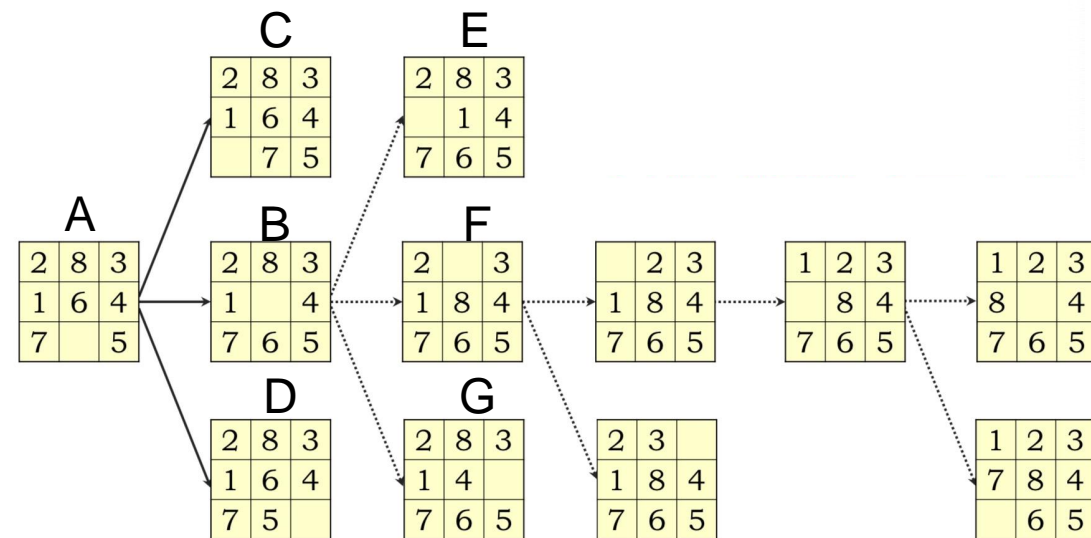
Open list: a list of nodes that is generated but yet not expanded

□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

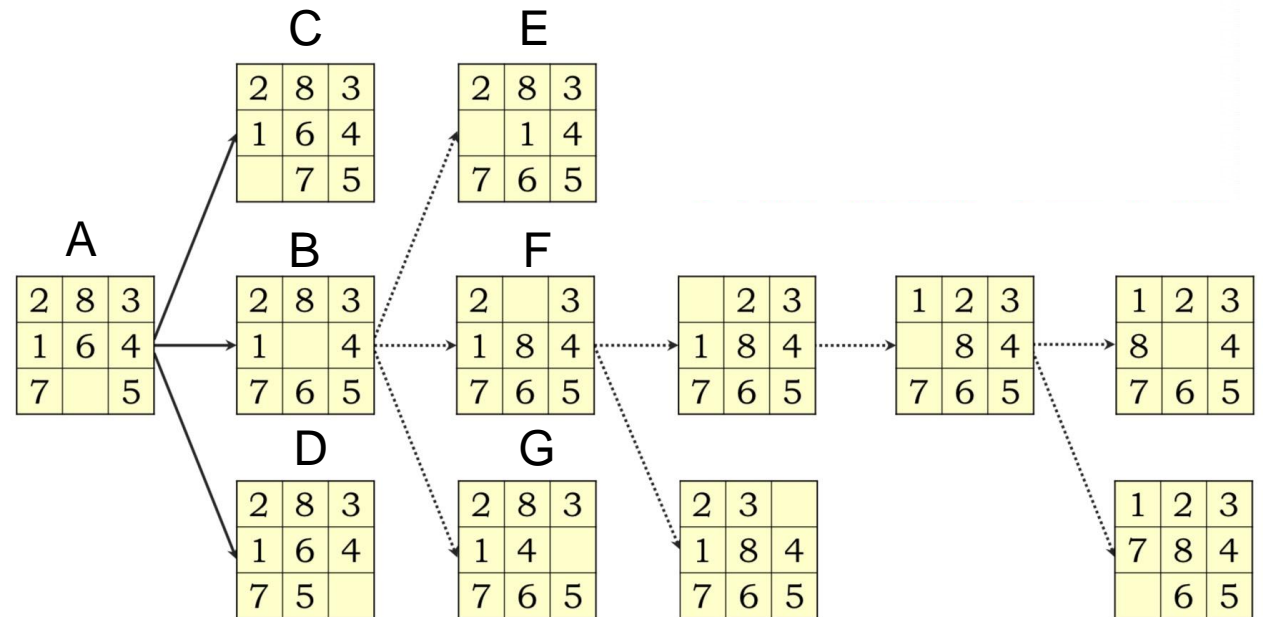
Open list: a list of nodes that is generated but yet not expanded

□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：

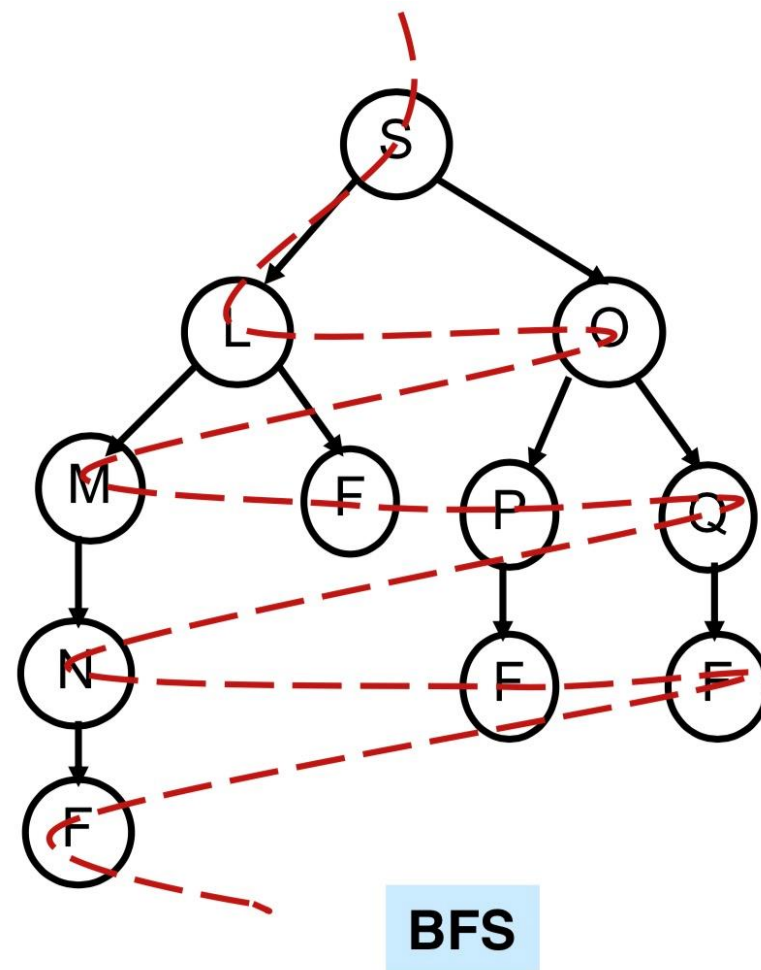
(1) **Breadth-first search** 宽度优先搜索

(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**

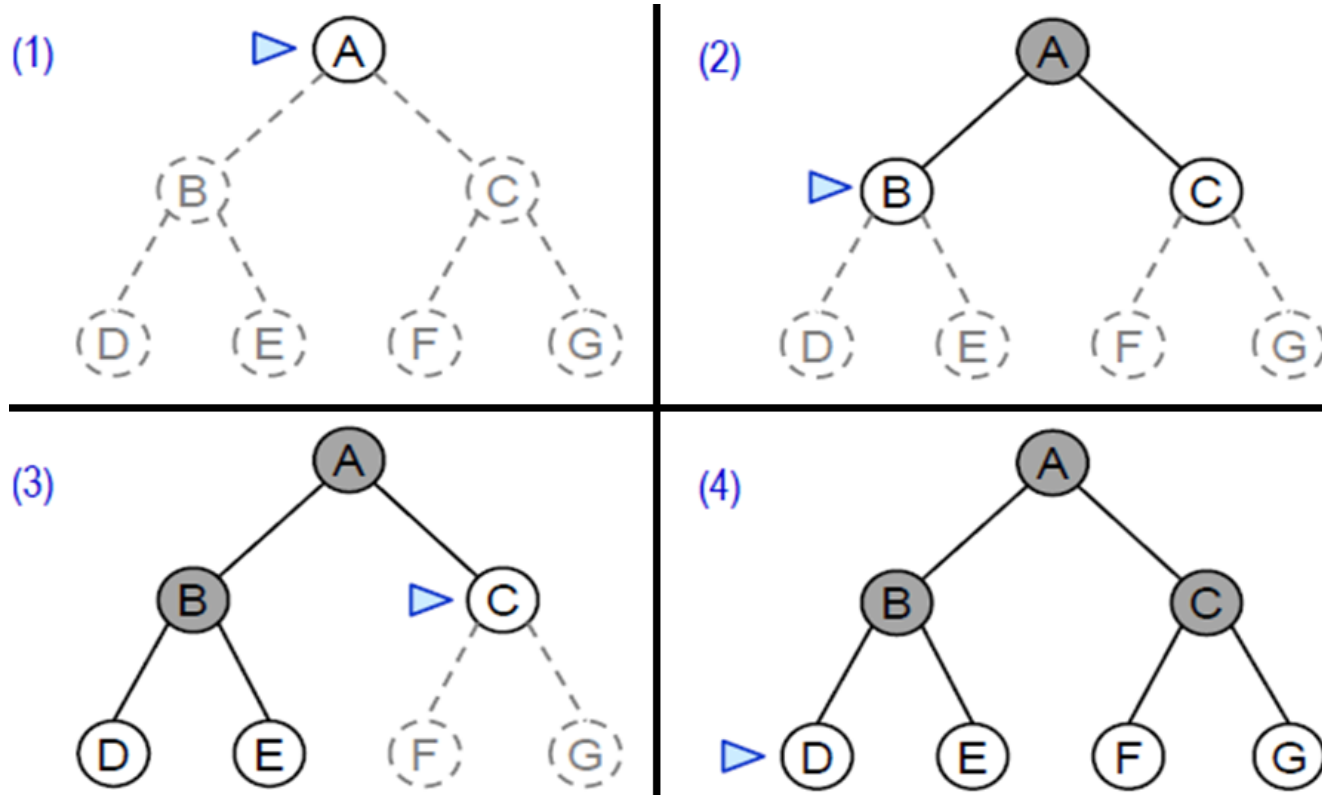
IF the child node contains a goal state

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

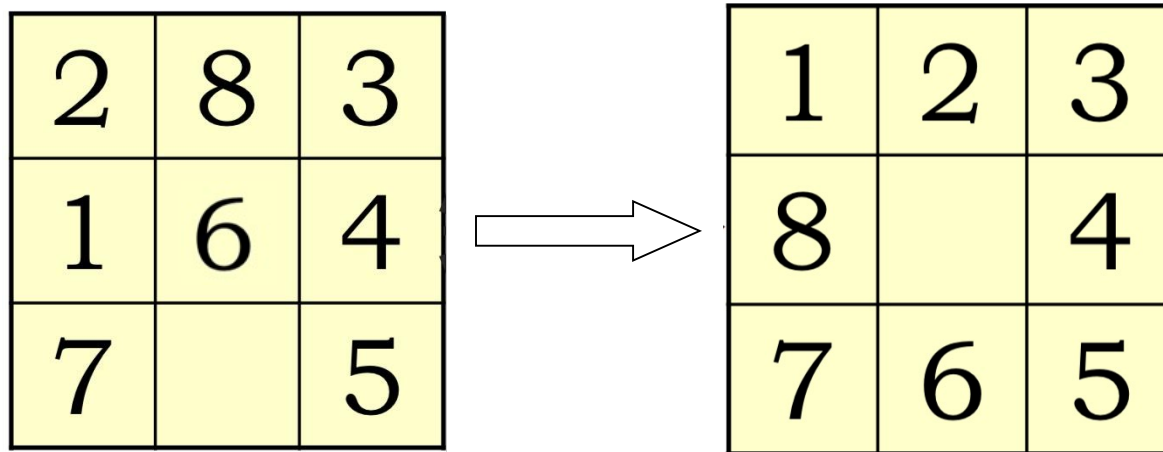


Open表	Close 表
[A]	[]
[B,C]	[A]
[C,D,E]	[A, B]
[D,E,F, G]	[A, B, C]
[E,F,G]	[A, B, C, D]
[F,G]	[A, B, C, D, E]
[G]	[A, B, C, D, E, F]
[]	[A, B, C, D, E, F, G]

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？

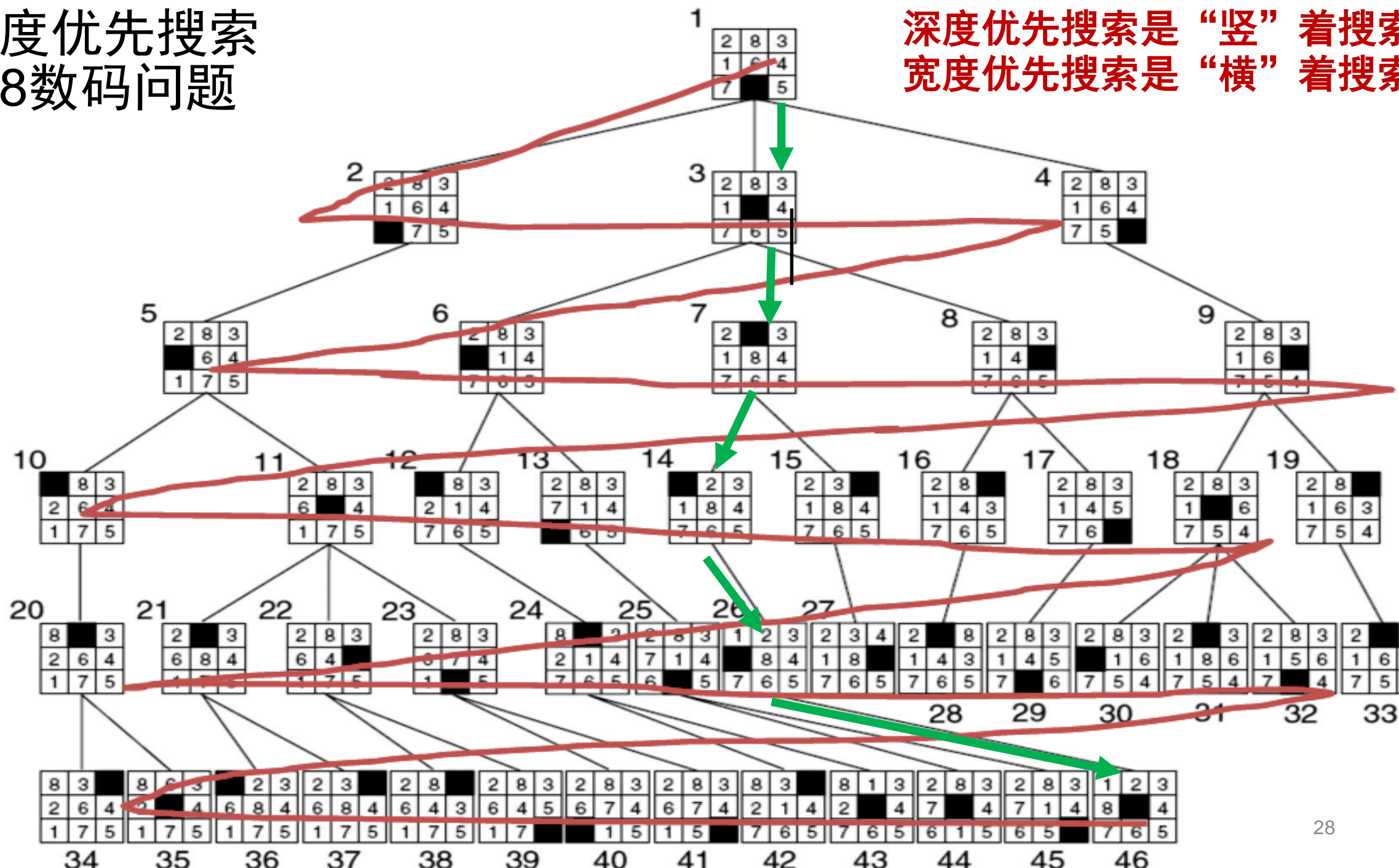


(a) Initial state

(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

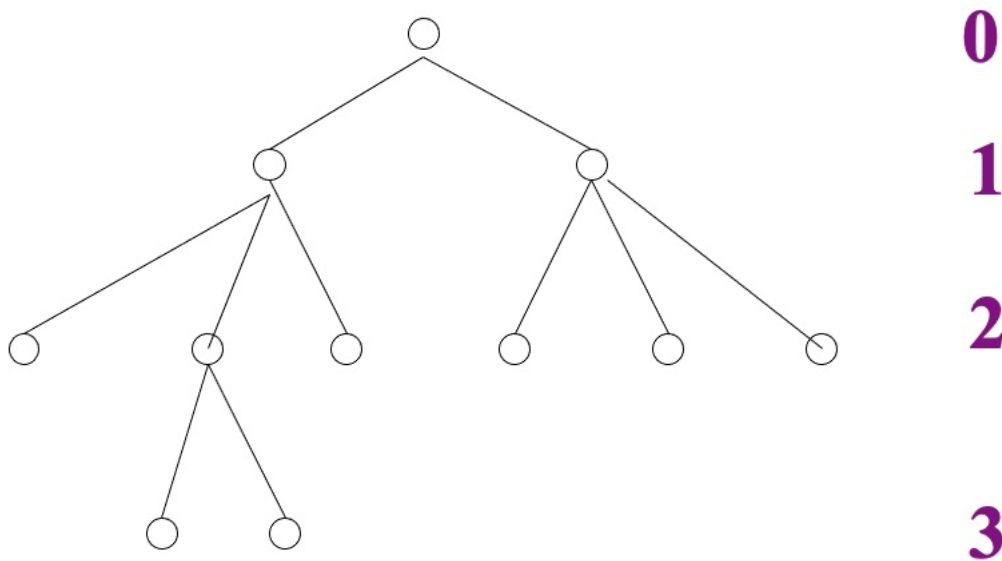
- 完备性：当问题有解时，一定能找到解.
- 最优性：当问题为单位代价，且问题有解时，一定能找到最优解
- BFS是一个通用的与问题无关的方法.
- 求解问题的效率较低

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

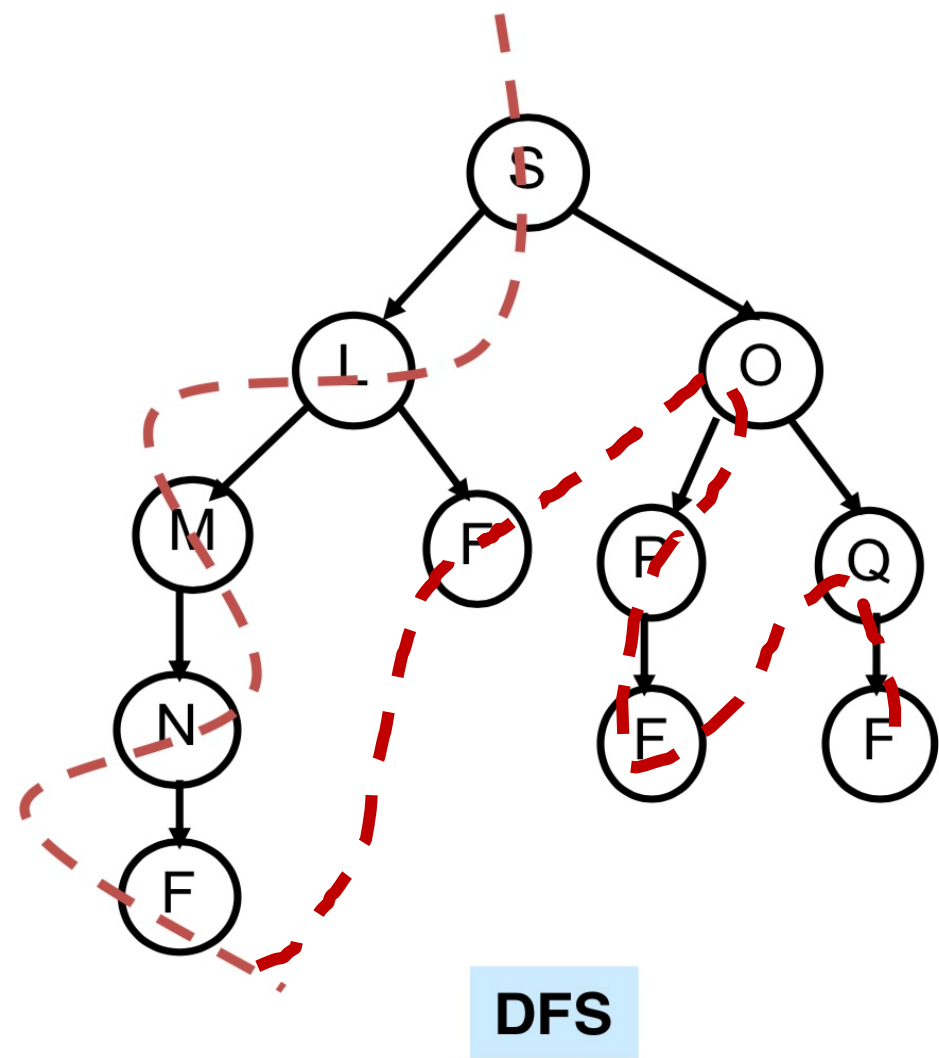
- 深度优先搜索的**基本思想**是**优先扩展深度最深的结点**。
- 在一个图中，**初始结点的深度定义为0**，其他结点的深度定义为其父结点的深度加 1。



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*

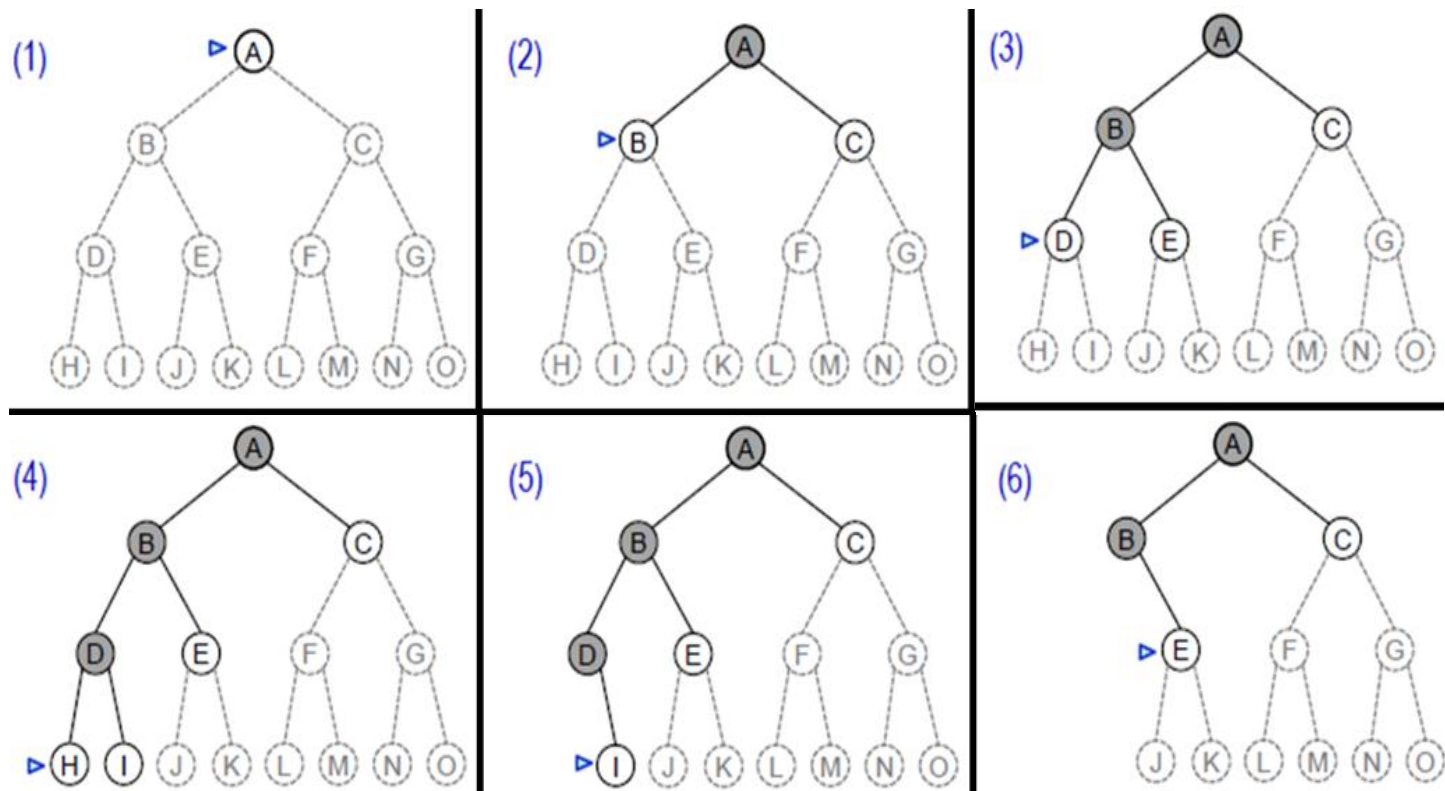
IF *n* is not in the open or closed lists **THEN**

IF the child node contains a goal state

THEN RETURN the corresponding solution

 Insert *n* to the open list

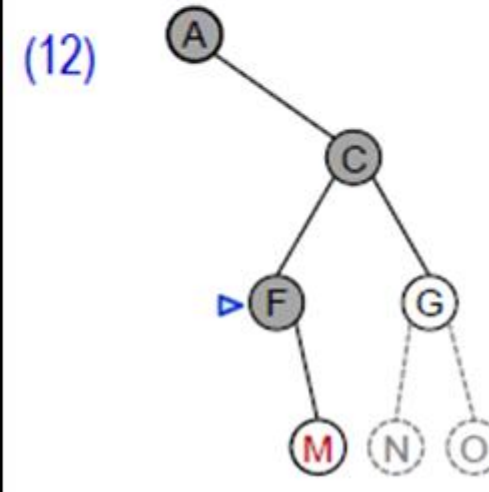
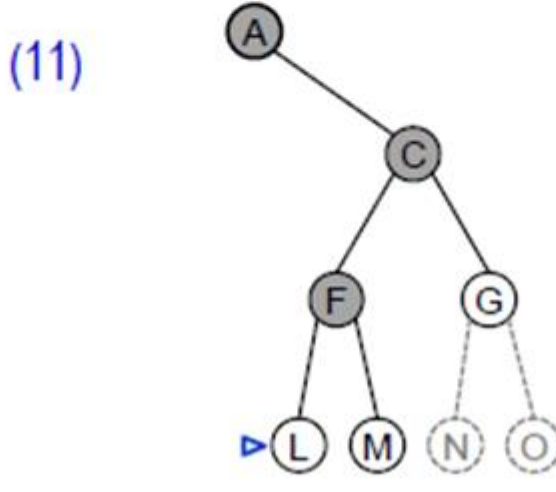
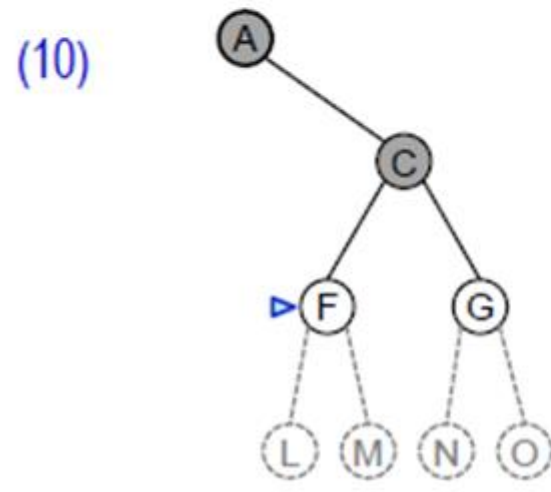
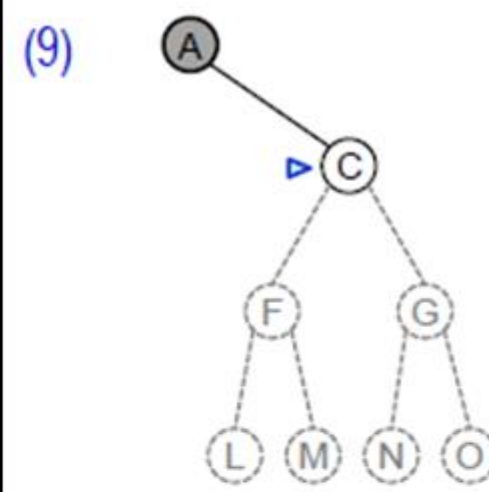
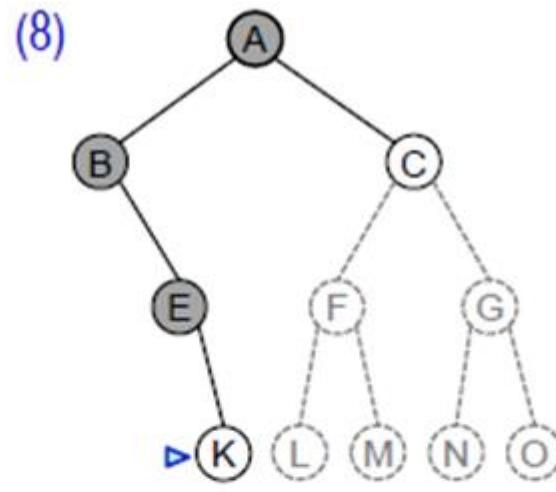
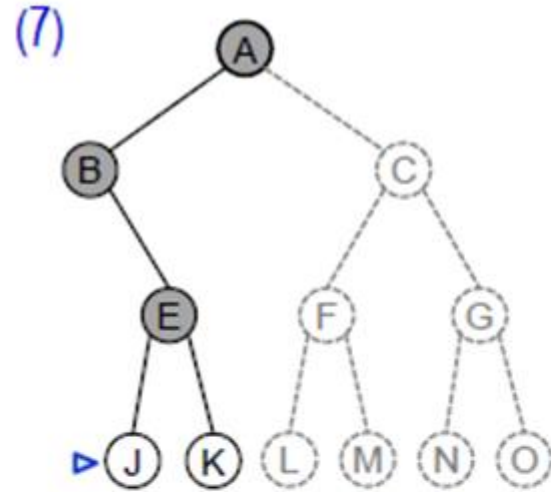
Depth-first Search on a Simple Binary Tree



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]

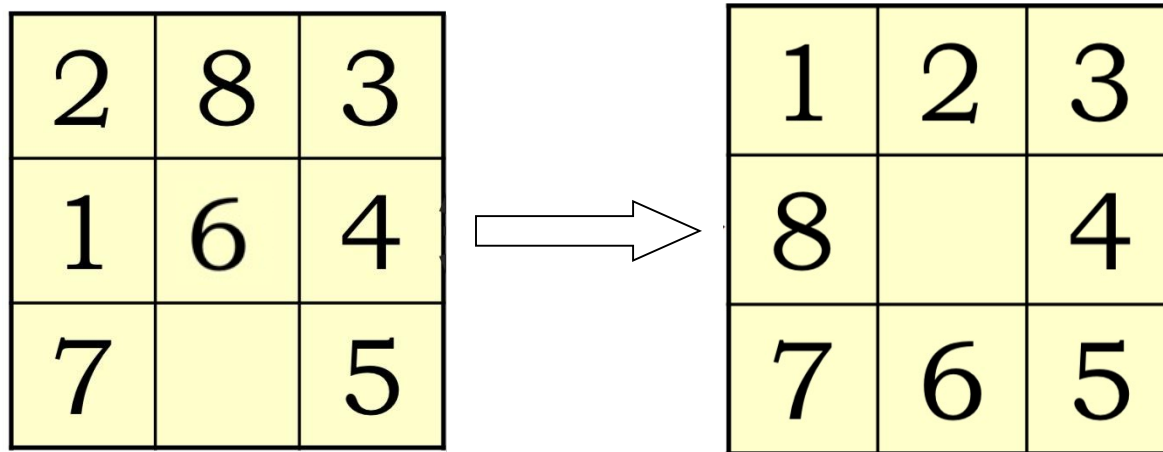
依次类推，直到找到目标函数或open=[]

Depth-first Search on a Simple Binary Tree



8-puzzle Problem DFS

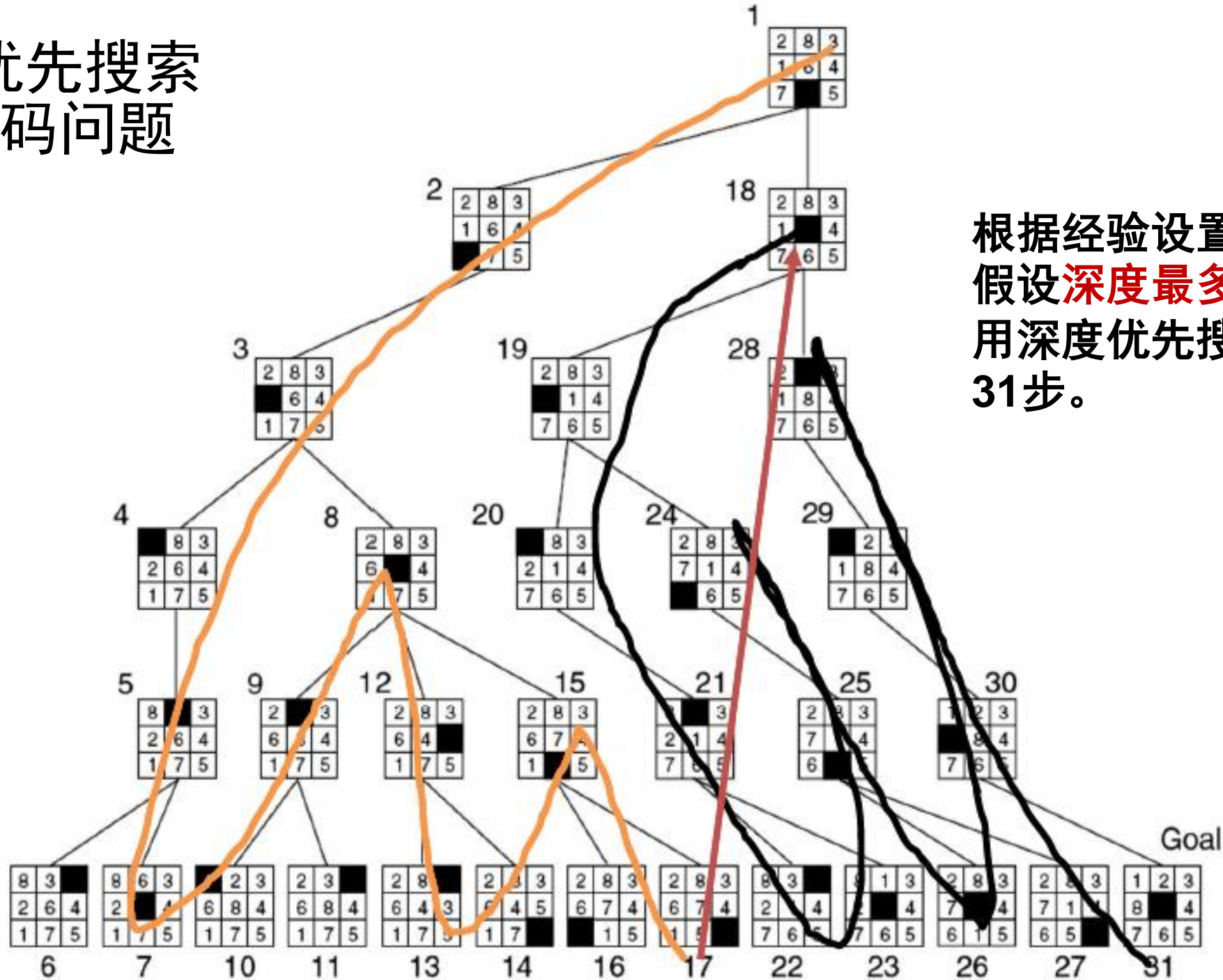
在 3×3 的棋盘，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。如何将棋盘从某一初始状态变成最后的目标状态？



(a) Initial state

(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。

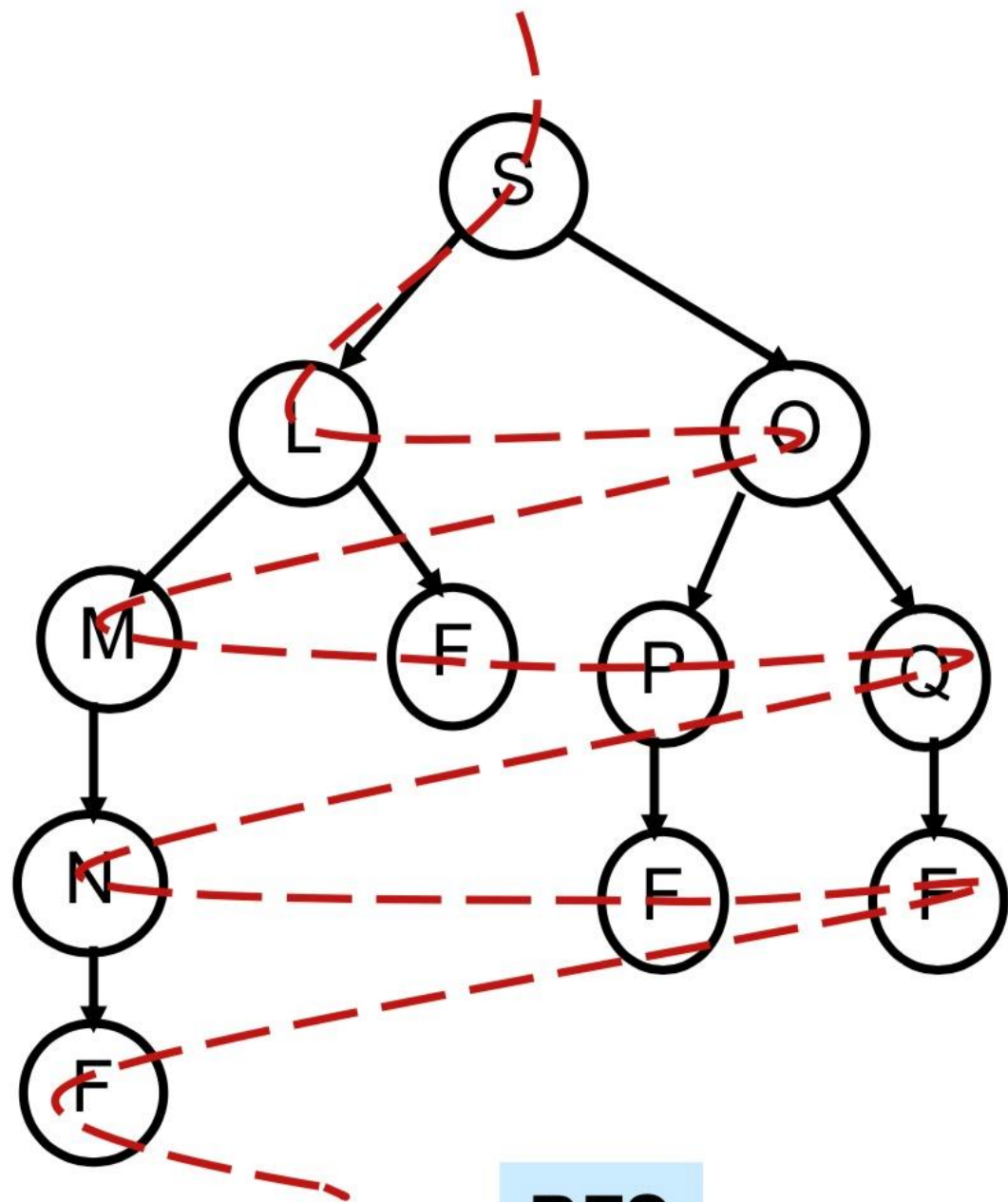
Features of DFS 深度优先搜索的性质

- 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

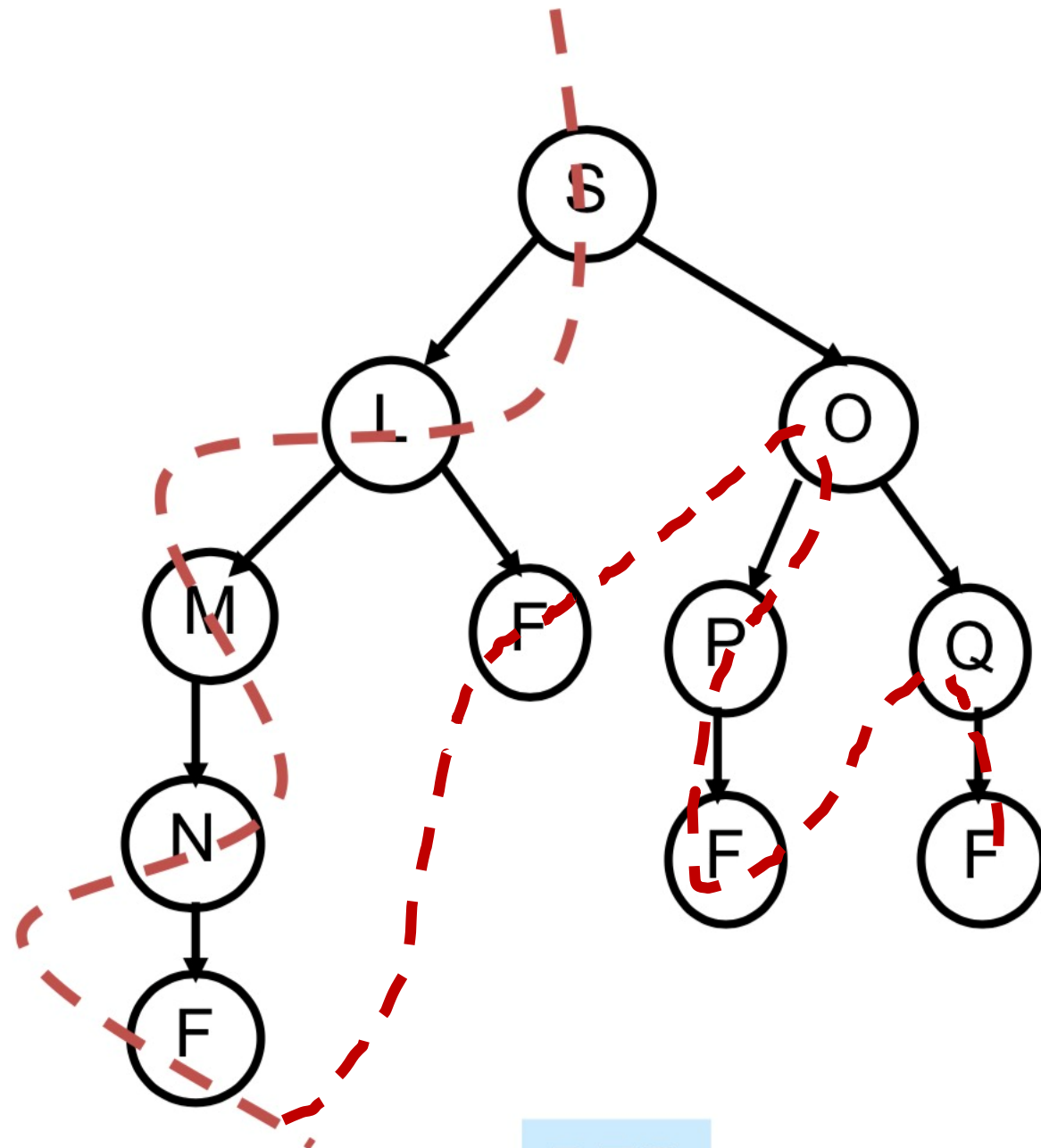
与BFS相比，其优势在于：空间复杂度低，因为只存储一条从根到叶子的路径。

- 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

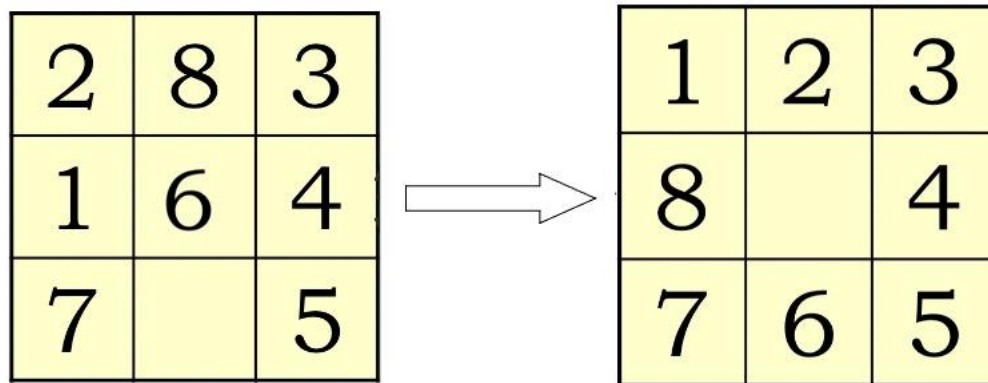
- ◆DFS总是首先扩展**最深**的未扩展结点；BFS总是首先扩展**最浅**的未扩展结点。
- ◆BFS 效率低， 但却一定能够求得问题的最优解， 即**BFS是完备的， 且是最优的**；
- ◆DFS不能保证找到最优解， **即DFS既不完备， 也不最优**；
- ◆在**不要求求解速度**且目标结点的层次**较深**的情况下， **BFS优于DFS**， 因为BFS一定能够求得问题的解， 而DFS在一个扩展的很深但又没有解的分支上进行搜索， 是一种无效搜索， 降低了求解的效率， 有时甚至不一定能找到问题的解；
- ◆在**要求求解速度**且目标结点的层次**较浅**的情况下， **DFS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



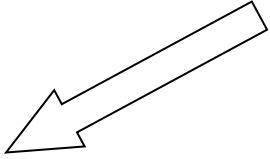
(a) Initial state

(b) goal state

Solve 8-Puzzle problem

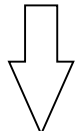
(a) Initial state s

2	8	3
1	6	4
7		5



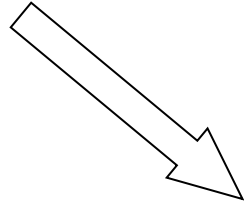
2	8	3
	1	4
7	6	5

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

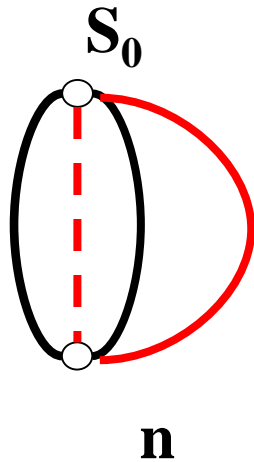
(b) goal state

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



?

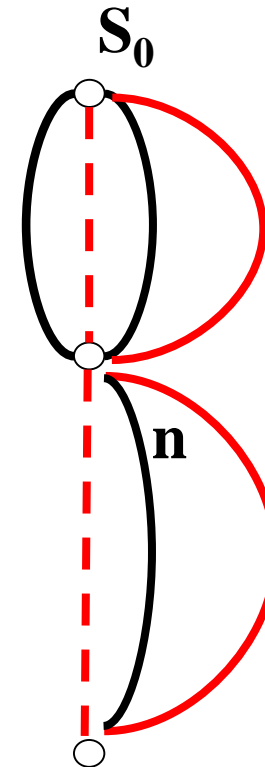


S_g

BFS, DFS

Informed(Heuristic) Search Strategy

启发式搜索



S_g

评价函数 (evaluation function)

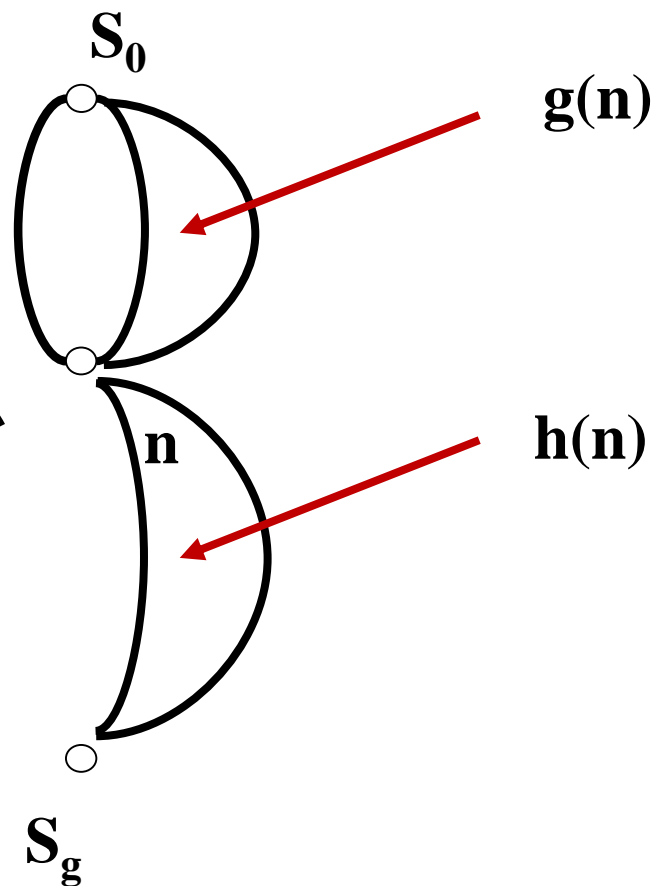
为了尽快找到从初始结点到目标结点的一条代价比较小的路径，我们希望所选择的结点尽可能在最佳路径上。如何评价一个结点在最佳路径上的可能性呢？

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

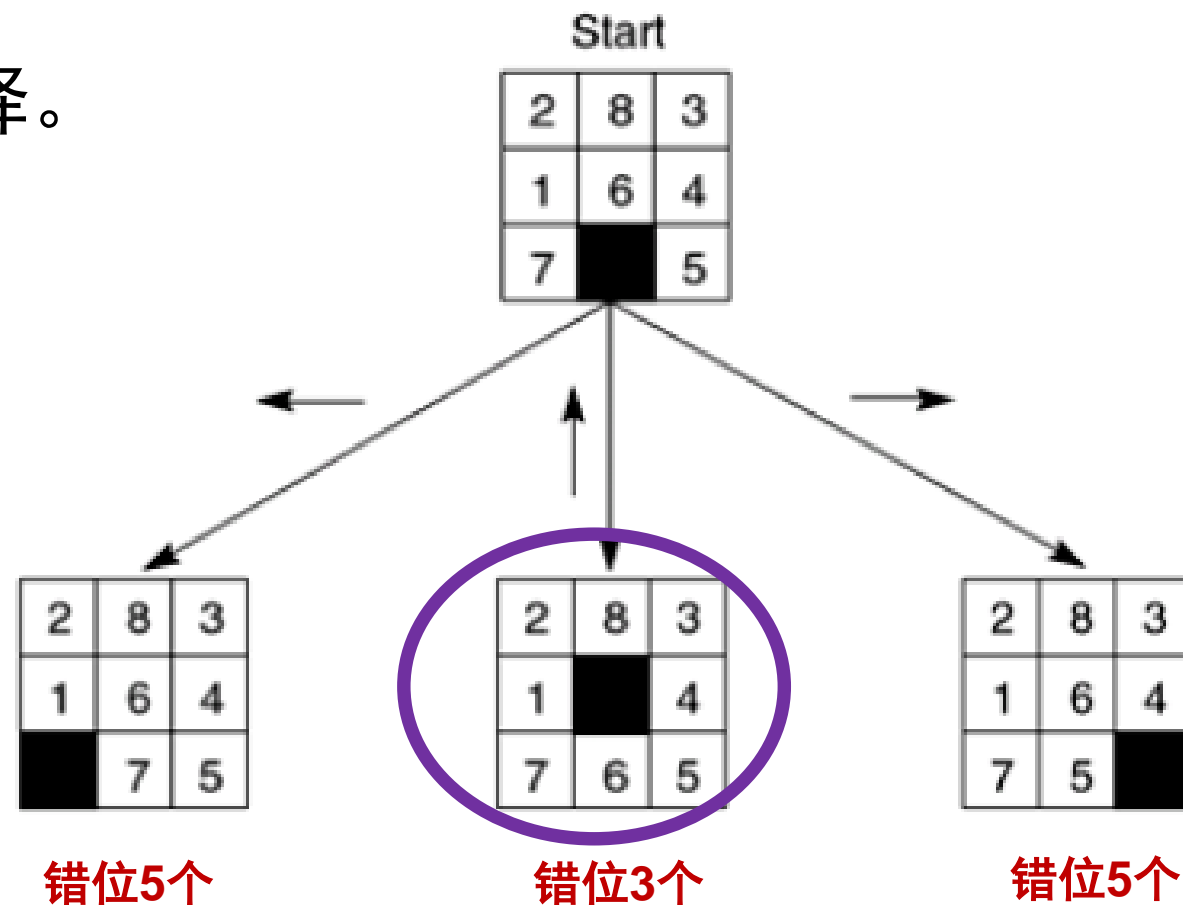
以8 数码问题为例

◆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。

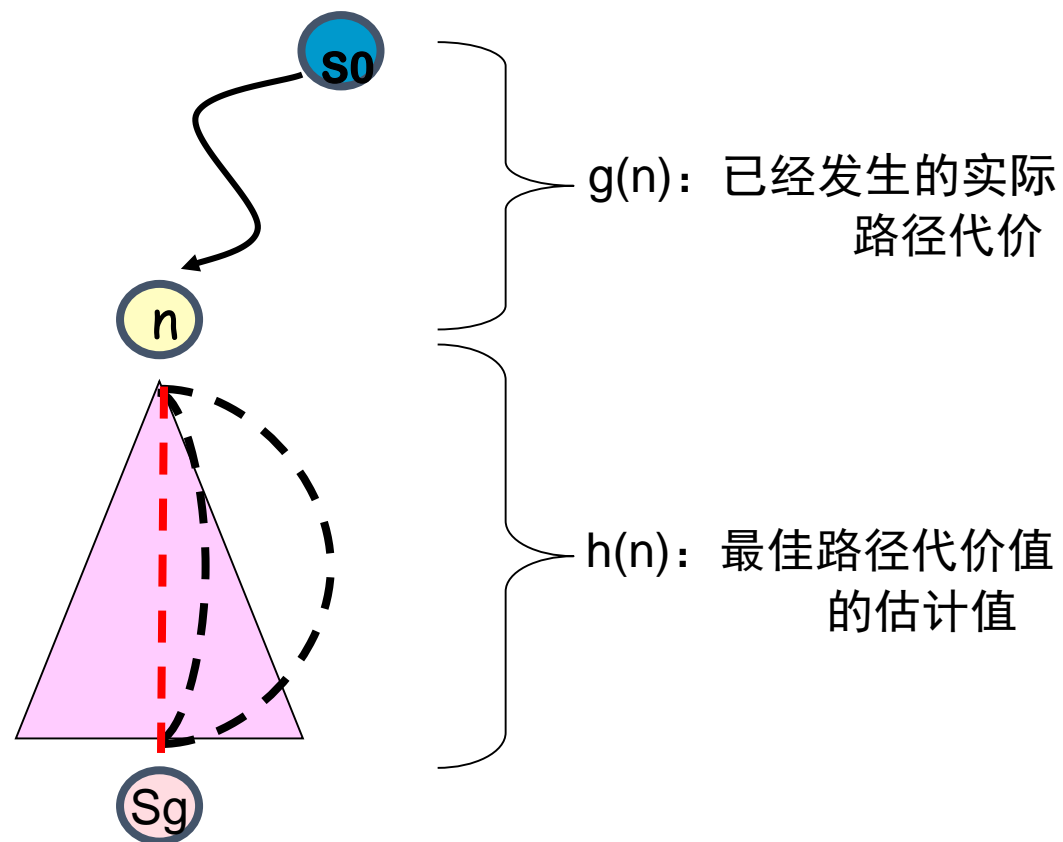
◆ Method D:

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)

◆ **Search Strategy** 搜索策略

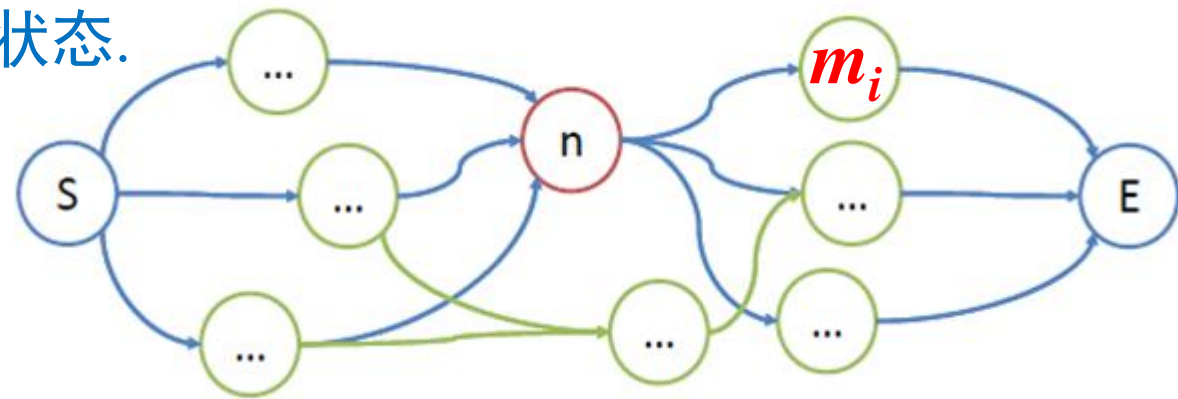
评估函数被看作是 **代价估计**，**评估值** $f(n)$ **最低** 的结点被选择首先进行扩展。

◆ **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) $\rightarrow \{m_i\}$, 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.

// $f(n, m_i)$ 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；

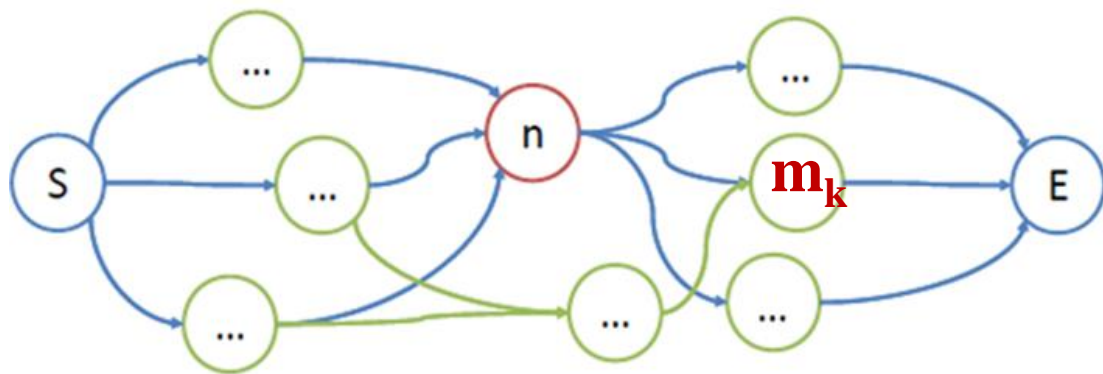
A search Algorithm (cont)

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

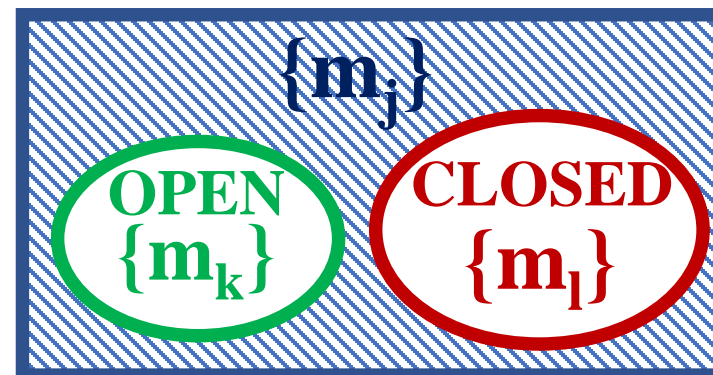
// 将 n 的后继中既不在OPEN中又不在CLOSED中的结点 m_j 放入OPEN表中, 并令 m_j 的指针指向 n ;

(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 $\{m_j\}$



A search Algorithm (cont)

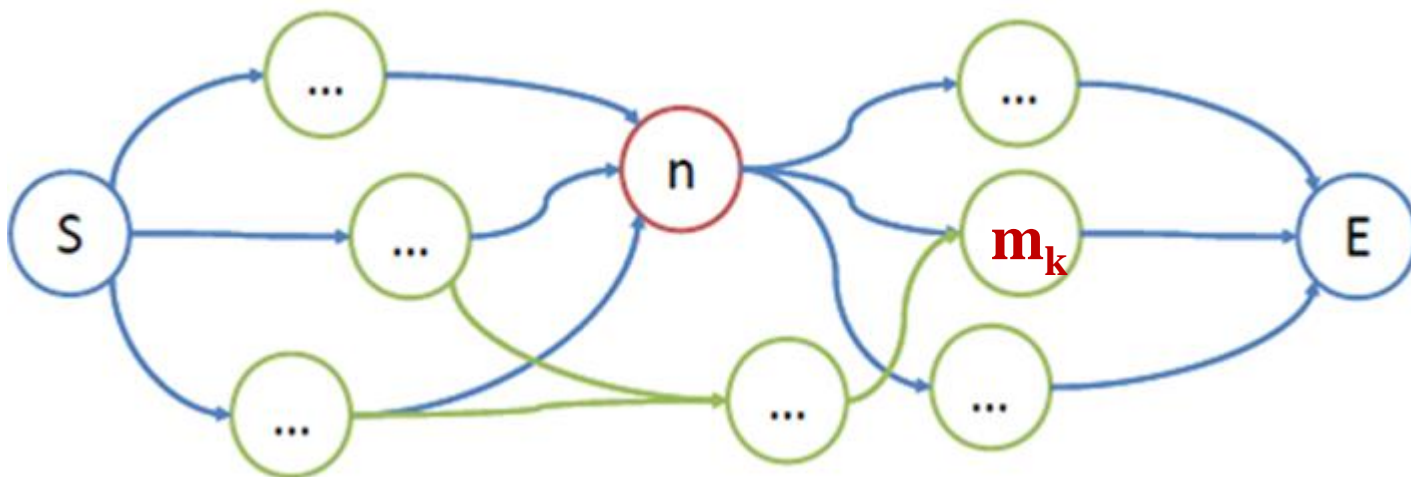
IF $f(n, m_k) < f(m_k)$, **则令** $f(m_k) := f(n, m_k)$ **且令** m_k **的指针指向** n

◆ $f(n, m_k) = \text{Dist} (S \dots n, m_k \dots E)$

是：从s出发，经过 n 和 m_k ，到达 E的最短路径；

◆ $f(m_k) = \text{Dist}(S \dots m_k \dots E)$

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 $f(n, m_1) < f(m_1)$ THEN

$\{f(m_1) := f(n, m_1); \text{ mark the pointer from } m_1 \text{ to } n; \text{ ADD}(m_1, \text{OPEN}); \}$

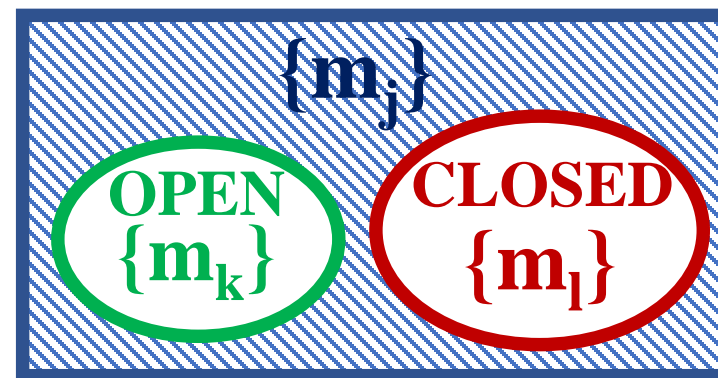
// 若条件2成立，则修改 m_1 的代价，且修改 m_1 的指针，使之指向 n 。

// 把 m_1 重新放回OPEN队列中，不必考虑改 m_1 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;

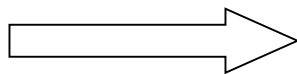
n 的后继结点集合 $\{m_i\}$



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

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

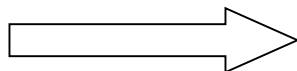
定义评价函数: $f(n) = g(n) + h(n)$

- ◆ $g(n)$ 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。

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

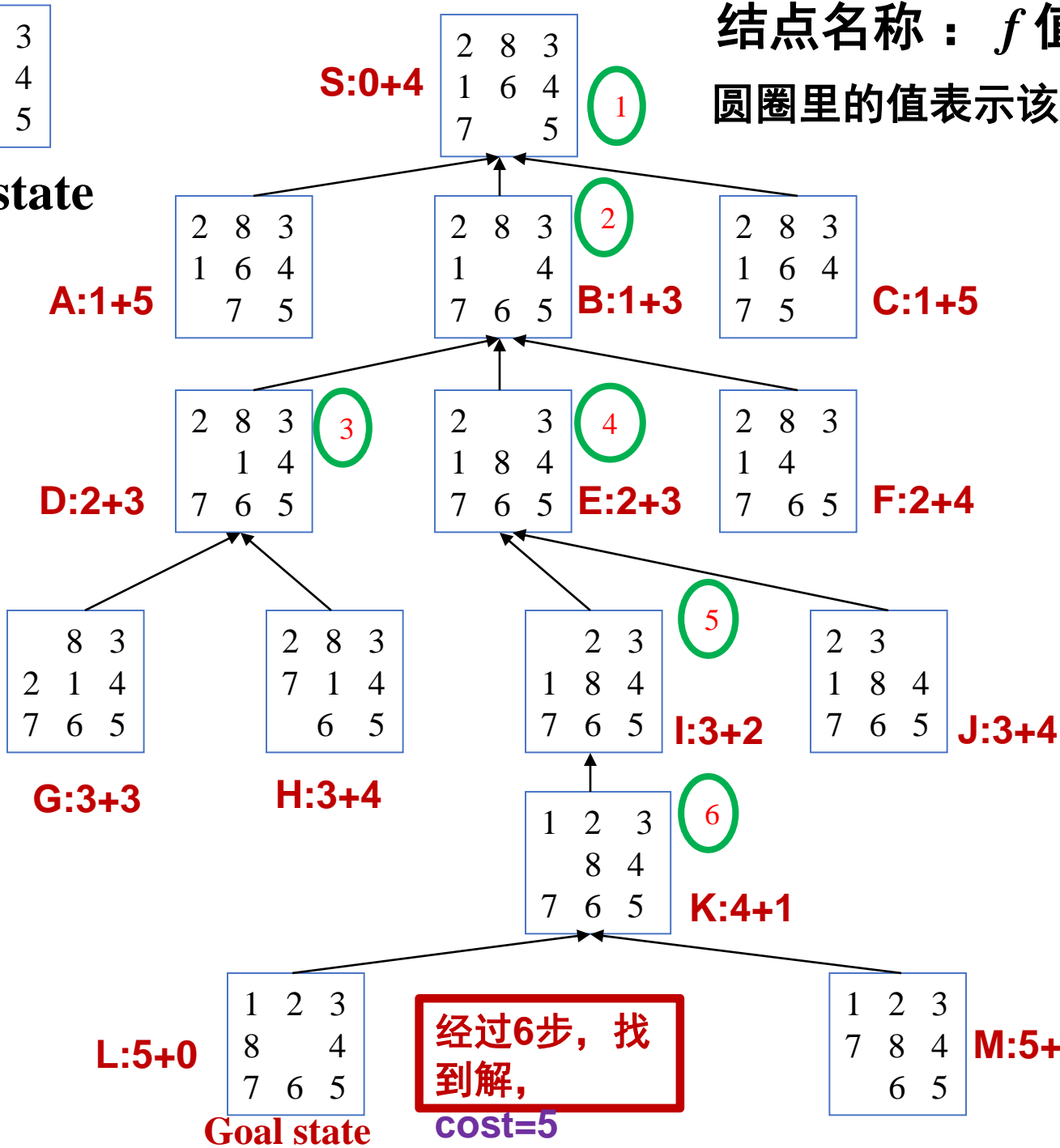
$h(s) = 4, g(s) = 0$

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，定义为移动将牌的步数，即结点的深度。
- ◆ $h(n)$ 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。

1	2	3
8		4
7	6	5

Goal state



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

◆ g 为结点 n 的深度, h 为错牌数

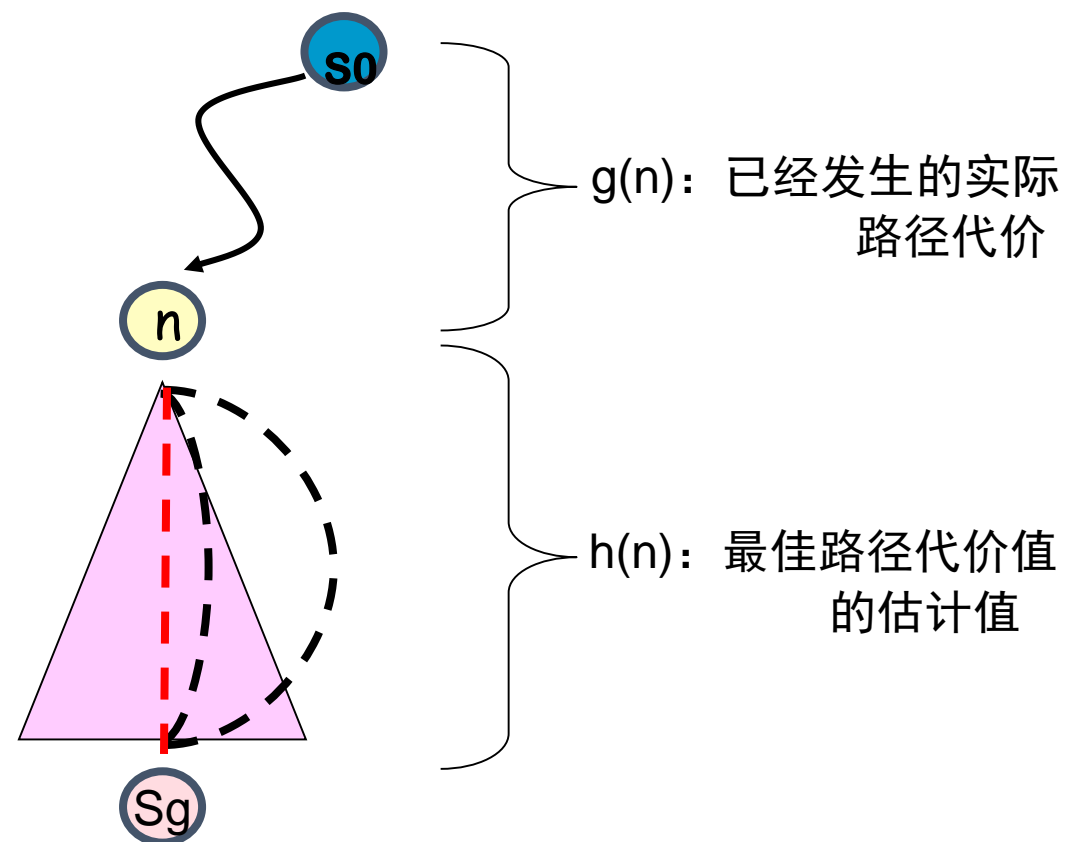
1. Open =[**S4**], Closed=[]
2. Open =[**B4,A6,C6**], Closed=[**S4**]
3. Open =[**D5,E5,A6,C6,F6**], Closed=[**S4,B4**]
4. Open =[**E5,A6,C6,F6,G6,H7**], Closed=[**S4,B4,D5**]
5. Open =[**I5,A6,C6,F6,G6,H7,J7**], Closed=[**S4,B4,D5,E5**]
6. Open =[**K5,A6,C6,F6,G6,H7,J7**], Closed=[**S4,B4,D5,E5,I5**]
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**。

Thank you !

课下思考题

Dijkstra算法是启发式搜索 or 盲目搜索, 为什么?