第4章 搜索策略

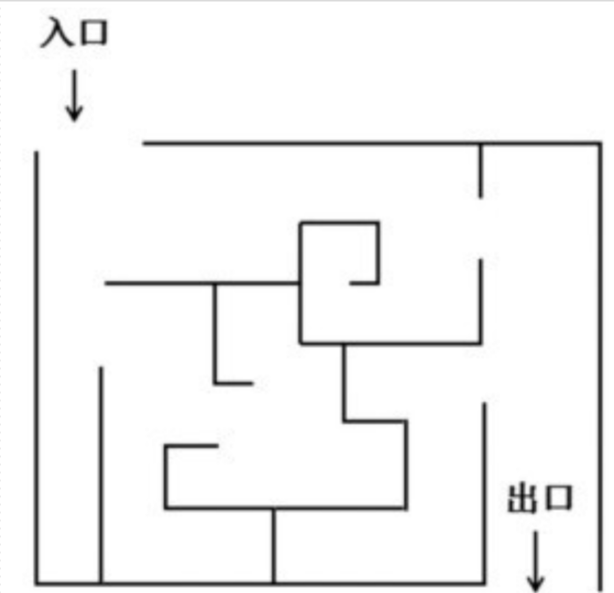
- 问题求解系统划分为两大类
  - 知识贫乏系统
    - 依靠搜索技术解决问题
    - 知识贫乏、缺乏针对性
    - 效率相对低
  - 知识丰富系统
    - 依靠推理技术解决问题
    - 基于丰富知识的推理技术，直截了当
    - 效率相对高

第4章 搜索策略

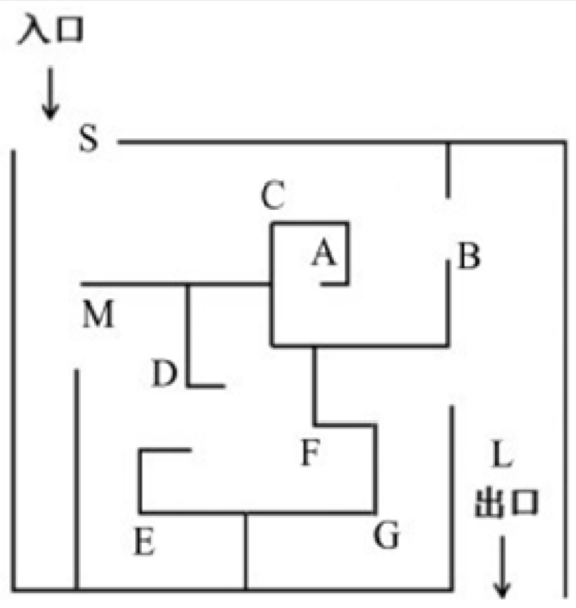
- 两大类搜索技术:
  - 1、一般图搜索、启发式搜索
  - 2、基于问题归约的与或图搜索
- 两种典型的推理技术:
  - 1、基于归结的演绎推理
    - 归结反演
  - 2、基于规则的演绎推理
    - 正向演绎推理
    - 逆向演绎推理

计算机搜索 VS 人类搜索

- 相对于人的思维，计算机的搜索更加深刻体现了符号主义思想。

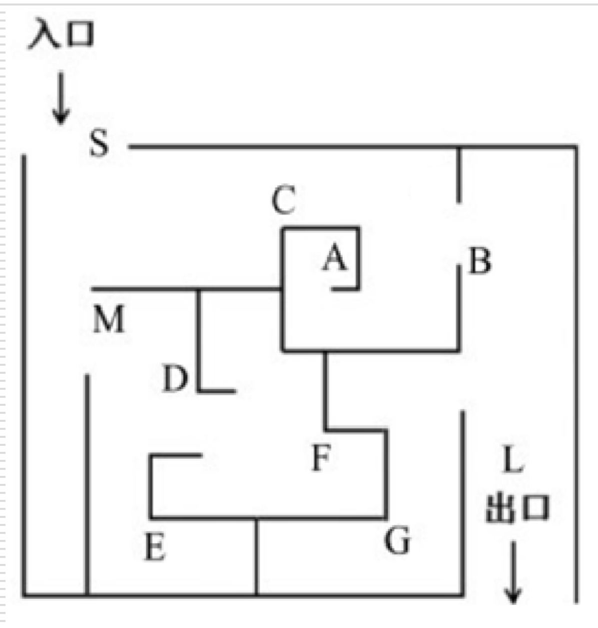


迷宫图示例

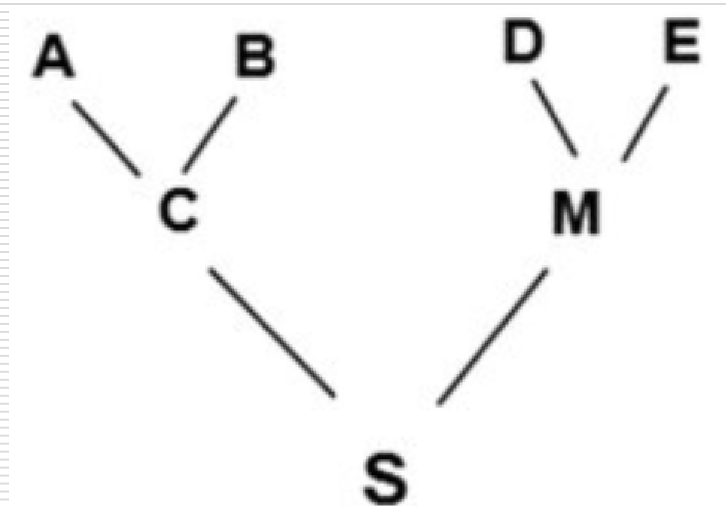


字母标记后的迷宫图

计算机搜索 VS 人类搜索



字母标记后的迷宫图



解决迷宫问题的搜索图

4.1 搜索概述

- 对于给定的问题，智能系统的行为一般是找到能够达到所希望目标的动作序列，并使其所付出的代价最小、性能最好。
- 基于给定的问题，问题求解的第一步是目标的表示。
- 搜索就是找到智能系统的动作序列的过程。

- 搜索算法的输入是给定的问题，输出时表示为动作序列的方案。
- 一旦有了方案，就可以执行该方案所给出的动作了。（执行阶段）
- 因此，求解问题包括：

  - 目标表示
  - 搜索
  - 执行

一般给定问题就是确定该问题的一些基本信息，一个问题由4部分组成：

（1）初始状态集合：定义了初始的环境。

（2）操作符集合：把一个问题从一个状态变换为另一个状态的动作集合。

（3）目标检测函数：用来确定一个状态是不是目标。

（4）路径费用函数：对每条路径赋予一定费用的函数。

其中，初始状态集合和操作符集合定义了问题的搜索空间。

在人工智能中，搜索问题一般包括两个重要的问题：

- **搜索什么**

搜索什么通常指的就是目标。

- **在哪里搜索**

在哪里搜索就是“搜索空间”。搜索空间通常是指一系列状态的汇集，因此称为状态空间。

- 和通常的搜索空间不同，人工智能中大多数问题的**状态空间在问题求解之前不是全部知道的**。

- 所以，人工智能中的搜索可以分成两个阶段：
  - 状态空间的生成阶段
  - 在该状态空间中对所求问题状态的搜索

- **搜索可以根据是否使用启发式信息分为**
  - **盲目搜索**
  - **启发式搜索**

□ 盲目搜索

只是可以区分出哪个是目标状态。

一般是按预定的搜索策略进行搜索。

没有考虑到问题本身的特性，这种搜索具有很大的盲目性，效率不高，不便于复杂问题的求解。

□启发式搜索

是在搜索过程中加入了与问题有关的启发式信息，用于指导搜索朝着最有希望的方向前进，加速问题的求解并找到最优解。

- 根据问题的表示方式分为
  - 状态空间搜索
  - 与或图搜索

状态空间搜索是用状态空间法来求解问题所进行的搜索

与/或图搜索是指用问题规约方法来求解问题时所进行的搜索。

- 考虑一个问题的状态空间为一棵树的形式。
- 宽度优先搜索
- 深度优先搜索

如果根节点首先扩展，然后是扩展根节点生成的所有节点，然后是这些节点的后继，如此反复下去。

在树的最深一层的节点中扩展一个节点。只有当搜索遇到一个死亡节点（非目标节点并且是无法扩展的节点）的时候，才返回上一层选择其他的节点搜索。

- 无论是宽度优先搜索还是深度优先搜索，遍历节点的顺序一般都是固定的，即一旦搜索空间给定，节点遍历的顺序就固定了。这种类型的遍历称为“确定性”的，也就是盲目搜索。
- 对于启发式搜索，在计算每个节点的参数之前无法确定先选择哪个节点扩展，这种搜索一般也称为非确定的。

搜索策略评价标准：

- 完备性：
  - 如果存在一个解答，该策略是否保证能够找到？
- 最优性：
  - 如果存在不同的几个解答，该策略是否可以发现最高质量的解答？
- 时间复杂性：
  - 需要多长时间可以找到解答？
- 空间复杂性：
  - 执行搜索需要多少存储空间？

4.2 一般图搜索

有许多智力问题(如梵塔问题、旅行商问题、八皇后问题、农夫过河问题等)和实际问题（如路径规划、机器人行动规划等）都可以归结为状态空间搜索。

用状态空间搜索来求解问题的系统均定义一个状态空间，并通过适当的搜索算法在状态空间中搜索解答路径。

状态空间搜索

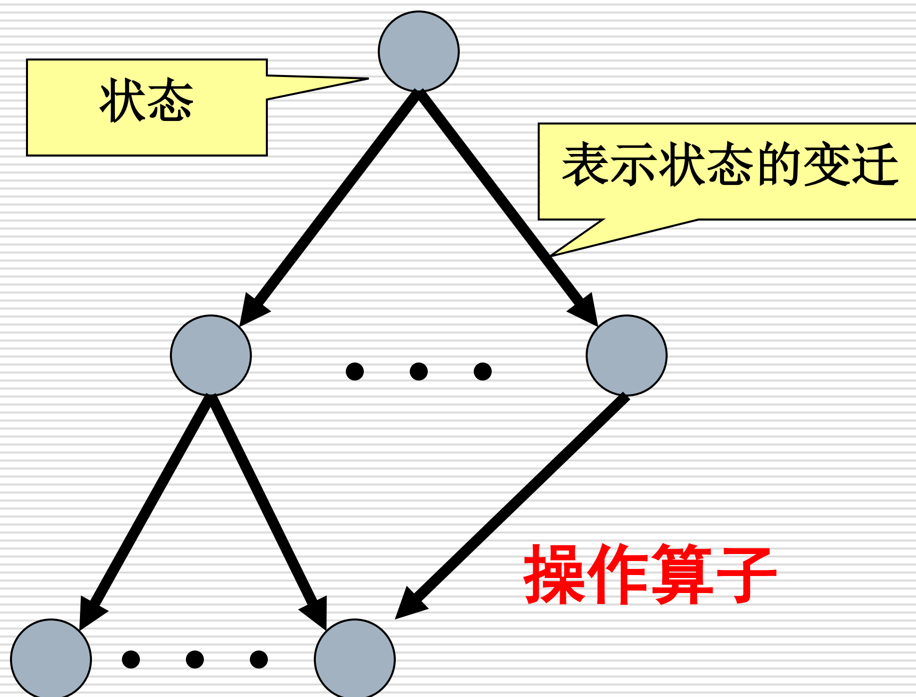
——1.状态空间及其搜索的表示

(1)状态空间的表示★

- 状态空间记为SP，可表示为二元组：
  - SP=(S,O)
  - S——问题求解（即搜索）过程中所有可能到达的合法状态构成的集合；
  - O——操作算子的集合，操作算子的执行会导致问题状态的变迁；
  - 状态——用于记载问题求解（即搜索）过程中某一时刻问题现状的快照；
    - 抽象为矢量形式 $Q=[q_0,q_1,\ldots,q_n]^T$
    - 每个元素q_i称为状态分量
    - 给定每个状态分量q_i的值就得到一个具体的状态

$$Q_k=[q_{0k},q_{1k},\ldots,q_{nk}]^T$$

用状态空间搜索来求解问题的系统均定义一个状态空间，并通过适当的搜索算法在状态空间中搜索解答路径。

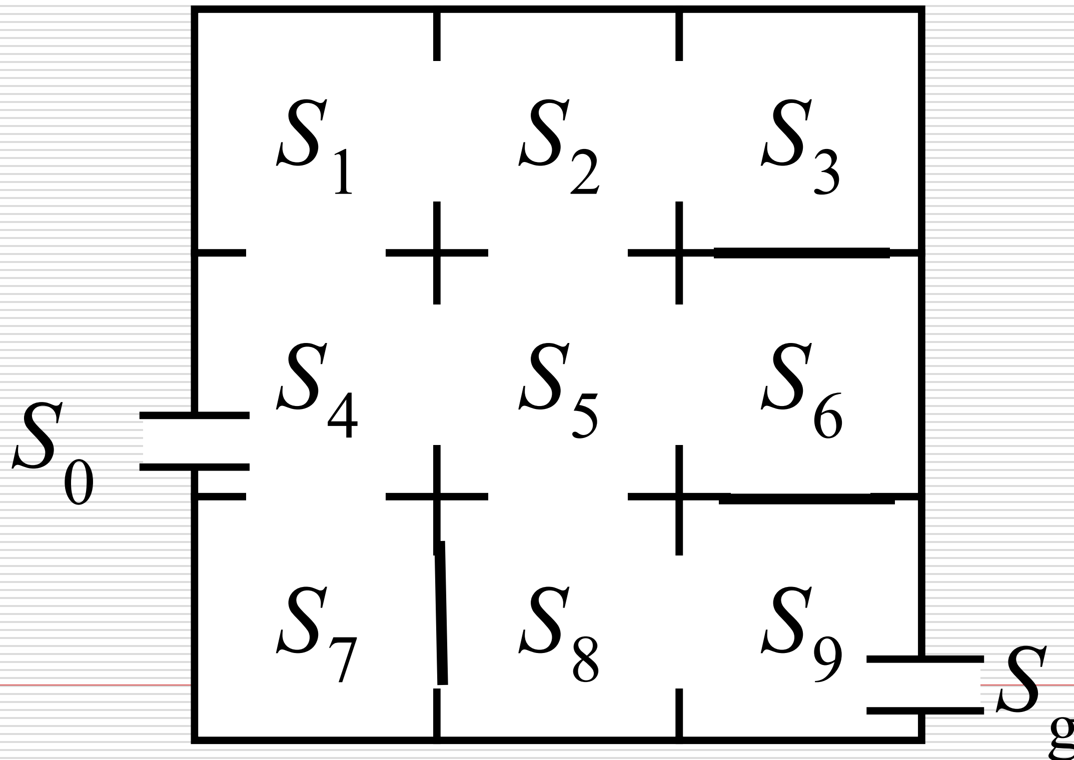


- 节点
  - 状态
- 有向弧
  - 状态的变迁
- 弧上的标签
  - 导致状态变迁的操作算子

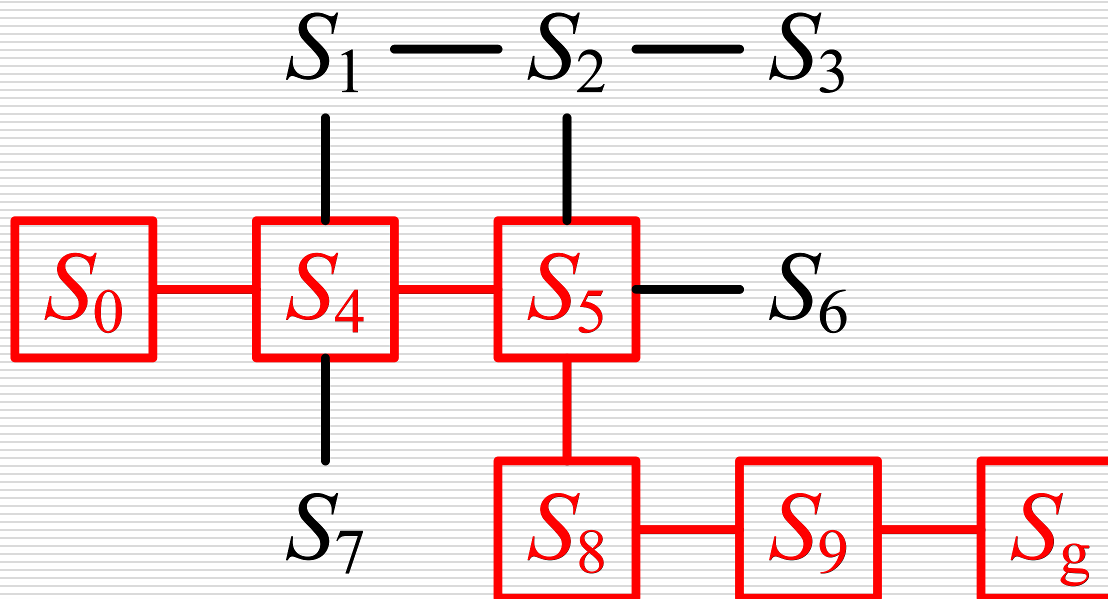
问题的状态空间是一个表示该问题的全部可能状态及其变迁的有向图。

状态空间搜索

例1：走迷宫是人们熟悉的一种游戏。



- 格子、入口和出口——节点——状态
- 通道——有向弧——操作算子
- 迷宫可以由一个有向图表示

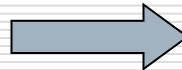


例2:在一个3×3的方格棋盘上放置着1,2,3,4,5,6,7,8八个数码，每个数码占一格，且有一个空格。这些数码可在棋盘上移动，其移动规则是：与空格相邻的数码方可移入空格。

现在的问题是：对于指定的初始棋局和目标棋局，给出数码的移动序列。该问题称为八数码问题。

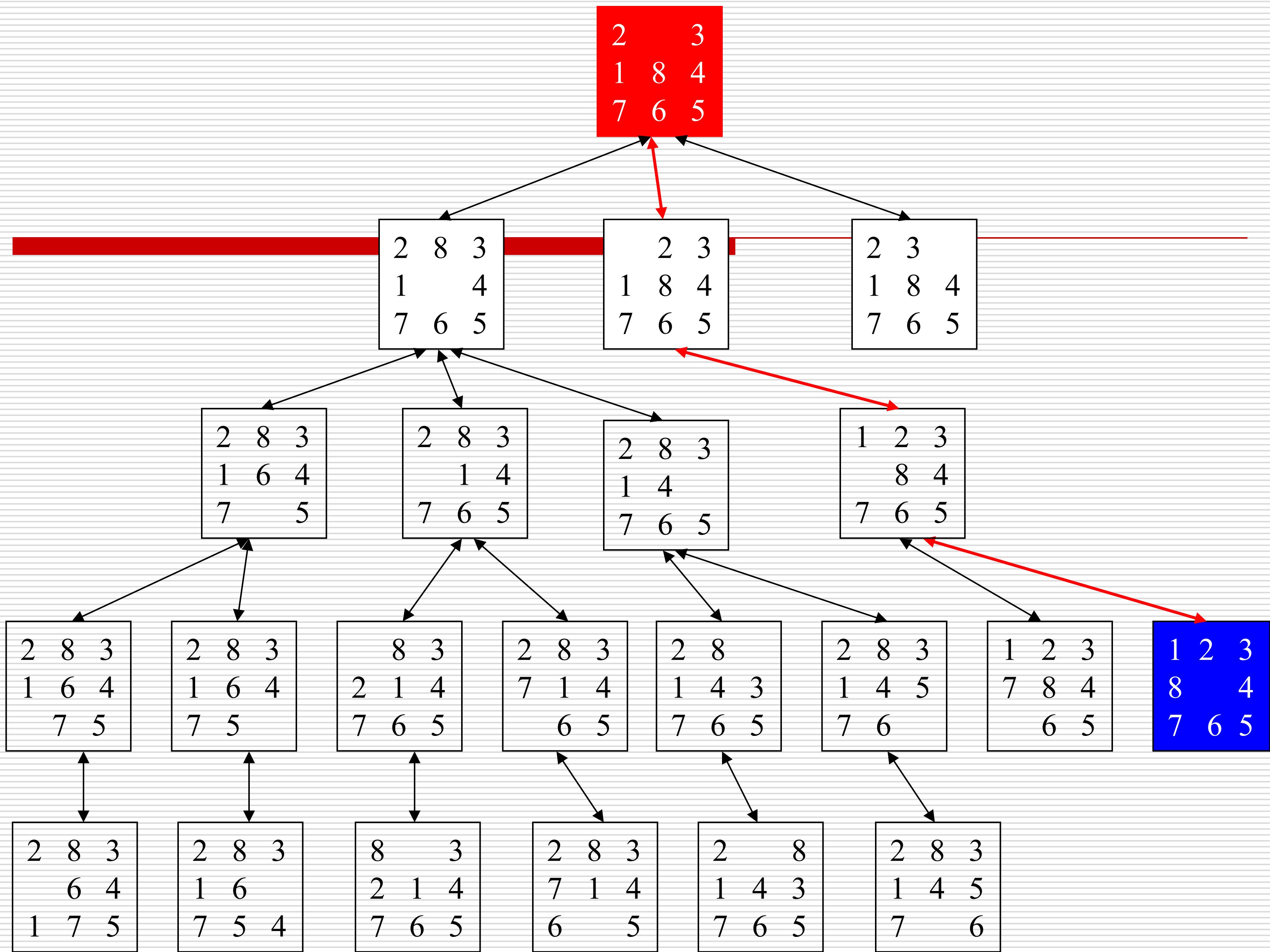
2		3
1	8	4
7	6	5

初始棋局



1	2	3
8		4
7	6	5

目标棋局



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

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

初始棋局

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

目标棋局

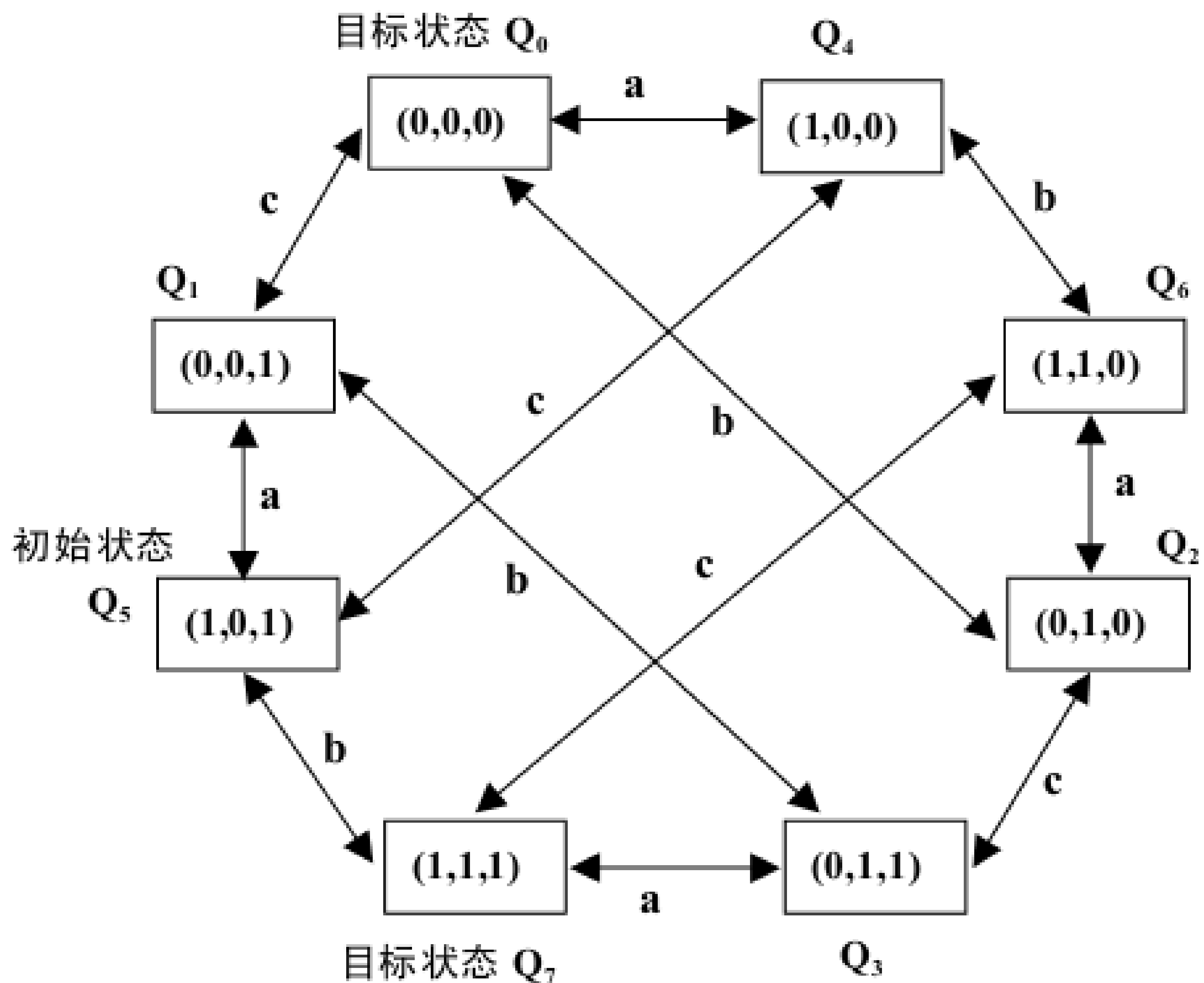
十五数码难题

例3:钱币翻转问题。设有3个钱币，其初始状态为(正、反、正),欲得的目标状态为(正、正、正)或(反、反、反)。问题是允许每次只能且必须翻转一个钱币，连翻三次。问能否达到目标状态。

分析：通过引入一个**三维变量**将问题表示出来。设三维变量为：$Q=[q_1, q_2, q_3]$，式中q_i（$i=1,2,3$)=1表示钱币为正面，q_i（$i=1,2,3$)=0表示钱币为反面。则三个钱币可能出现的状态有8种组合：

$Q_0=(0,0,0)$, $Q_1=(0,0,1)$, $Q_2=(0,1,0)$, $Q_3=(0,1,1)$, $Q_4=(1,0,0)$, $Q_5=(1,0,1)$, $Q_6=(1,1,0)$, $Q_7=(1,1,1)$。

即初始状态为Q_5, 目标状态为Q_0或Q_7, 要求步数为3。



钱币问题的状态空间图

钱币问题的状态空间图，图中可知：从Q_5不可能经过三步到达Q_0，即不存在从Q_5到达Q_0的解。但从Q_5出发到达Q_7的解有7个，它们是aab，aba，baa，bbb，bcc，cbc和ccb。

状态空间搜索

——1.状态空间及其搜索的表示

(2)状态空间表示的经典例子“传教士和野人问题”

- 问题的描述：
  - N个传教士带领N个野人划船过河；
  - 3个安全约束条件：
    - 1)船上人数不得超过载重限量，设为K个人；
    - 2)任何时刻（包括两岸、船上）野人数目不得超过传教士；
    - 3)允许在河的某一岸或者在船上只有野人而没有传教士；

状态空间搜索

——1.状态空间及其搜索的表示

(2)状态空间表示的经典例子“传教士和野人问题”

- 特例：N=3，K=2
  - 状态分量m——传教士在左岸的实际人数
  - 状态分量c——野人在左岸的实际人数
  - 状态分量b——指示船是否在左岸(1、0)
  - 3个安全约束条件
    - m ≧ c (左岸安全)且 N-m ≧ N-c(右岸安全);(想一想，为什么不考虑船安全？)
    - m=0且0≤c ≤N (左岸没有传道士，右岸一定安全);
    - N-m=0且0≤N-c≤N (右岸没有传道士，左岸一定安全);

设初始状态下传教士、野人和船都在左岸，目标状态下这三者均在右岸，问题状态以（m,c,b）表示。

- 问题的求解任务可描述为：(3,3,1) → (0,0,0)
- 该简单问题可能到达的合法状态：
  - 可能状态总数：4×4×2=32
  - 根据条件得出合法状态：20
    - m ≧ c 且 N-m ≧ N-c (左岸安全且右岸安全)
      - m=1,c=1；m=2,c=2
    - m=0且0≤c ≤N(左岸没有传道士)
      - m=0,c=0,1,2,3
    - N-m=0且0≤N-c≤N(右岸没有传道士)
      - m=3,c=0,1,2,3
  - 不可能到达的合法状态：
    - (0,0,1),(0,3,0),(3,0,1),(3,3,0)
  - 可能到达的合法状态共16个

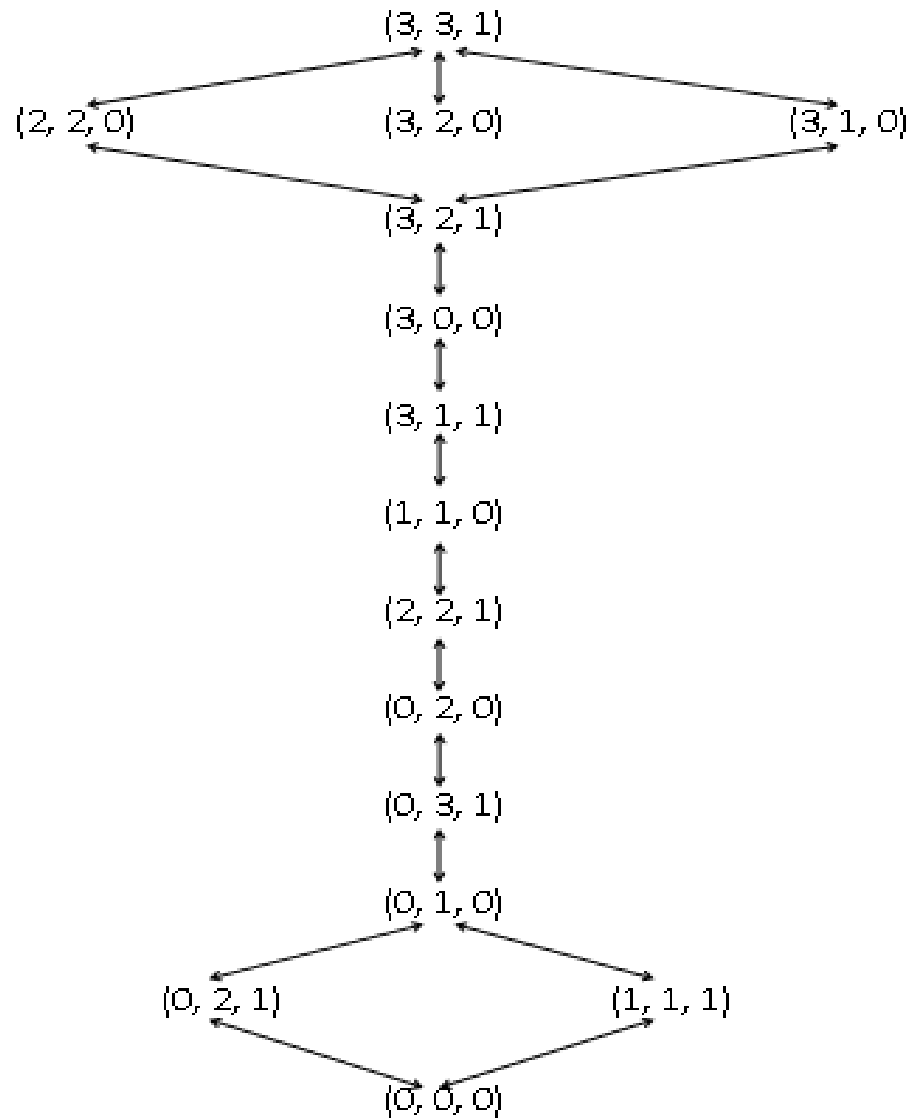
状态空间搜索

——1.状态空间及其搜索的表示

(2)状态空间表示的经典例子“传教士和野人问题”

- 定义2类操作算子：
  - L(x,y)——指示从左岸到右岸的划船操作
  - R(x,y)——指示从右岸到左岸的划船操作
    - x + y ≤K=2(船的载重限制)；
    - x和y取值的可能组合只有5个
    - 10，20，11，01，02
    - （允许在船上只有野人而没有传教士）
  - 共有10个操作算子

渡河问题的状态空间有向图



状态空间搜索

——1.状态空间及其搜索的表示

- 总结状态空间表示方法：★
  - 用状态空间方法表示问题时，首先必须定义状态的描述形式，通过使用这种描述形式可把问题的一切状态都表示出来。另外，还要定义一组操作，通过使用这些操作可把问题由一种状态转变为另一种状态。
  - 问题的求解过程是一个不断把操作作用于状态的过程。如果在使用某个操作后得到的新状态是目标状态，就得到了问题的一个解。这个解是从初始状态到目标状态所用操作构成的序列。

状态空间搜索

——1.状态空间及其搜索的表示

- 总结状态空间表示方法：★
  - 要使问题由一种状态转变到另一种状态时，就必须使用一次操作。这样，在从初始状态转变到目标状态时，就可能存在多个操作序列(即得到多个解)。那么，其中**使用操作最少或较少的解才为最优解**(因为只有在使用操作时所付出的代价为最小的解才是最优解)。
  - 对其中的某一个状态，可能存在多个操作．使该状态变到几个不同的后继状态．那么到底用哪个操作进行搜索呢?这就有赖于**搜索策略了．不同的搜索策略有不同的顺序**，这就是本章后面要讨论的问题。

主观题 10分

设置

课堂练习

- **有一农夫带一只狐狸、一只小羊和一篮菜过河（从左岸到右岸）。假设船太小，农夫每次只能带一样东西过河；考虑到安全，无农夫看管时，狐狸和小羊不能在一起，小羊和那篮菜也不能在一起。请为该问题的解决设计状态空间，并画出状态空间图。**

正常使用主观题需2.0以上版本雨课堂

作答

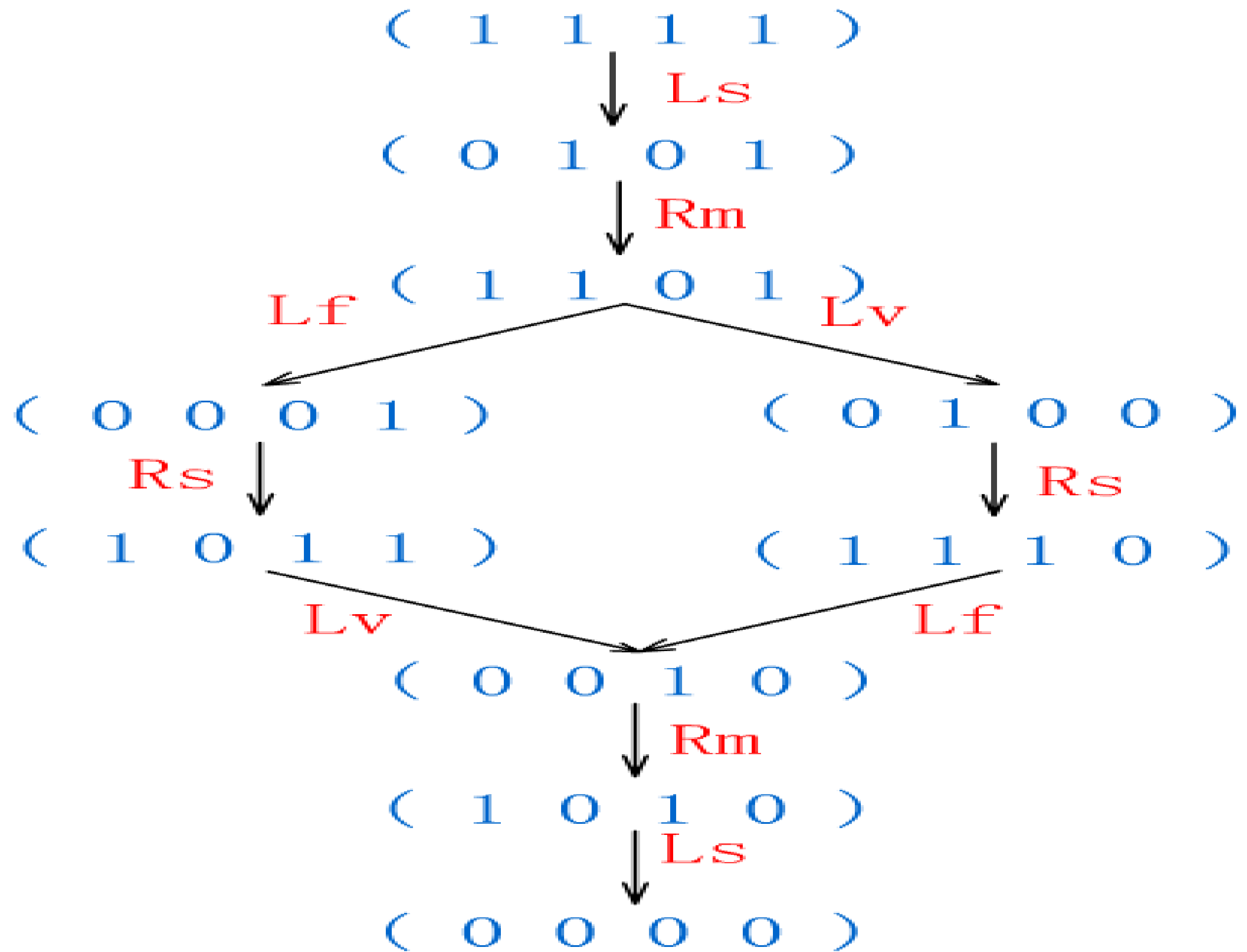
课堂练习

- 有一农夫带一只狐狸、一只小羊和一篮菜过河（从左岸到右岸）。假设船太小，农夫每次只能带一样东西过河；考虑到安全，无农夫看管时，狐狸和小羊不能在一起，小羊和那篮菜也不能在一起。请为该问题的解决设计状态空间，并画出状态空间图。

解析

- 以变量m、f、s、v分别指示农夫、狐狸、小羊、菜,且每个变量只可取值1(表示在左岸)或0(表示在右岸)。问题状态可以四元组(m、f、s、v)描述,设初始状态下均在左岸,目标状态下都到达右岸。从而,问题求解任务可描述为
  (1，1，1，1)->(0，0，0，0)
- 思考：为什么不把船的状态放到状态空间中去?
- 由于问题简单,状态空间中可能的状态总数为2×2×2×2 = 16,由于要遵从安全限制,合法的状态只有(除初、目状态外)：
  1110，1101，1011，1010，0101，0001，0010，0100;
  不合法状态有：0111,1000,1100,0011,0110,1001
- 设计二类操作算子:Lx、Rx,x为m、f、s、v时分别指示农夫独自,带狐狸,带小羊,带菜过河；状态空间图如下所示.由于Lx和Rx是互逆操作,故而解答路径可有无数条,但最近的只有二条;都是7个操作步.

解析:四元组(m、f、s、v)代表（农夫、狐狸、小羊、菜）



状态空间搜索

——1.状态空间及其搜索的表示

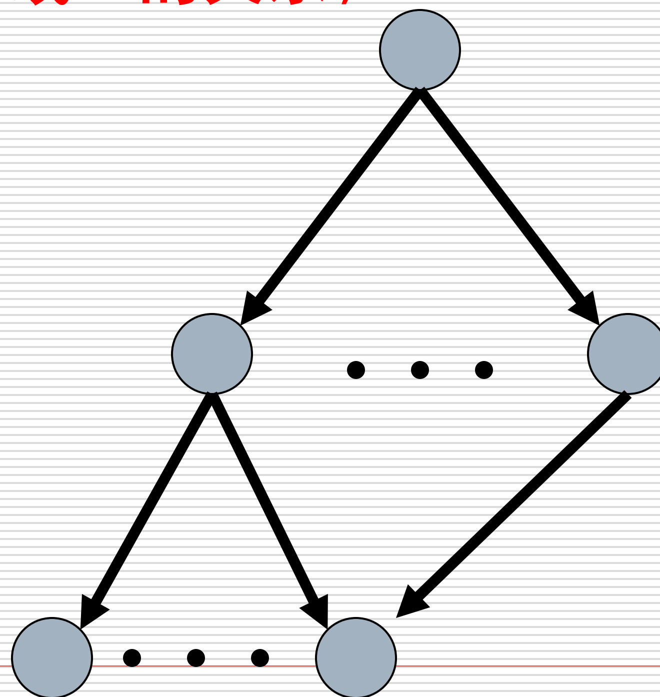
(3)状态空间的搜索

- 状态空间的搜索记为SE，可表示为五元组：
  - SE=(S,O,E,I,G)；
  - E——搜索引擎；
  - I——问题的初始状态，I ∈ S；
  - G——问题的目标状态集合，G ⊂ S。
- 搜索引擎E——可以设计为实现任何搜索算法的控制系统。
- 基本思想——通过搜索引擎E寻找一个操作算子的调用序列，使问题从初始状态I变迁到目标状态G之一。
- 解答路径——初-目变迁过程中的状态序列或相应的操作算子调用序列。

状态空间搜索

——1.状态空间及其搜索的表示

- 或图（一般图）
  - 一个状态可以有多个可供选择的操作算子；
  - 操作算子间的选择是一种“或”的关系；
  - 这样的有向图称为或图。

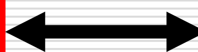


状态空间搜索

——1.状态空间及其搜索的表示

(3) 状态空间的搜索

- 或图（一般图）
  - 一个状态可以有多个可供选择的操作算子；
  - 操作算子间的选择是一种“或”的关系，这样的有向图称为或图。
- 状态空间一般都表示为或图（一般图）
- 搜索图——在搜索解答路径的过程中画出搜索时涉及到的节点和弧线，构成所谓的搜索图。

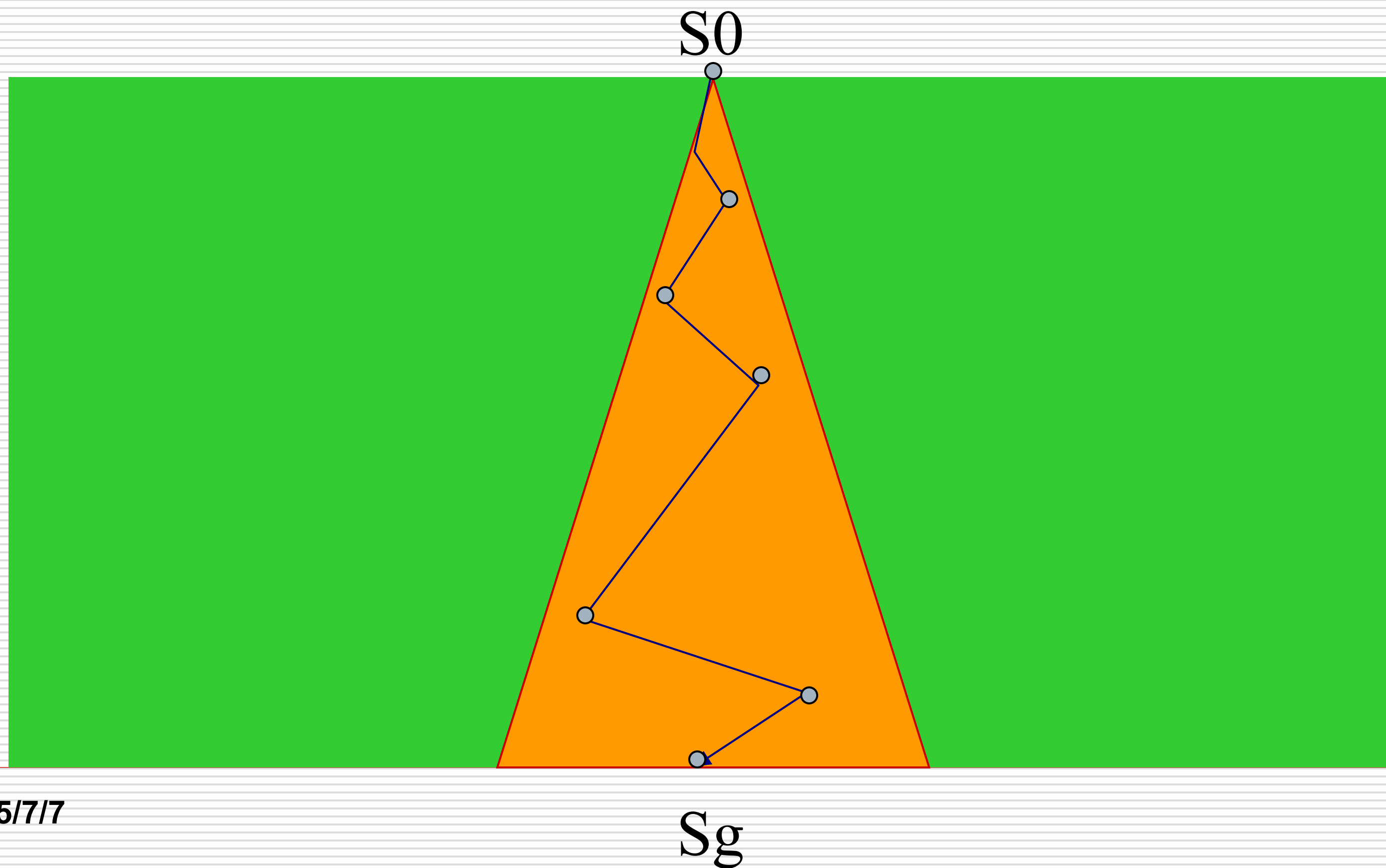
状态空间搜索  一般图搜索

状态空间搜索

——1.状态空间及其搜索的表示

(3)状态空间的搜索

- 状态空间、搜索图和解答路径之间的关系



状态空间搜索

——1.状态空间及其搜索的表示

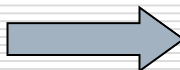
(4)一般图搜索例子——八数码游戏

- 求解的问题：
  - 给定初始布局(即**初始状态**)和目标布局(即**目标状态**)，
  - 如何移动数码才能从初始布局到达目标布局？
- 解答
  - 就是一个合法的**棋牌走步序列**。

1		3
7	2	4
6	8	5

初始布局

移动数码



1	2	3
8		4
7	6	5

目标布局

状态空间搜索

——1.状态空间及其搜索的表示

(4)一般图搜索例子——八数码游戏

- 用一般图搜索方法解决该问题的步骤：
  - 1、为**问题状态**的表示建立数据结构
  - 2、制定**操作算子**集合
  - 3、设计**搜索引擎**
- 第一步：为问题状态的表示建立数据结构
  - 3×3的一个矩阵S
  - 矩阵元素Sij∈{0,1,2,3,4,5,6,7,8},其中1≤i,j≤3
  - 数字0表示空格
  - 数字1-8表示相应的棋牌
- 八数码问题就可表示为：

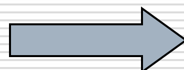
状态空间搜索

——1.状态空间及其搜索的表示

(4)一般图搜索例子——八数码游戏

1		3
7	2	4
6	8	5

移动数码



1	2	3
8		4
7	6	5

初始布局　　　　目标布局

$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{Bmatrix}$$

状态空间搜索

——1.状态空间及其搜索的表示

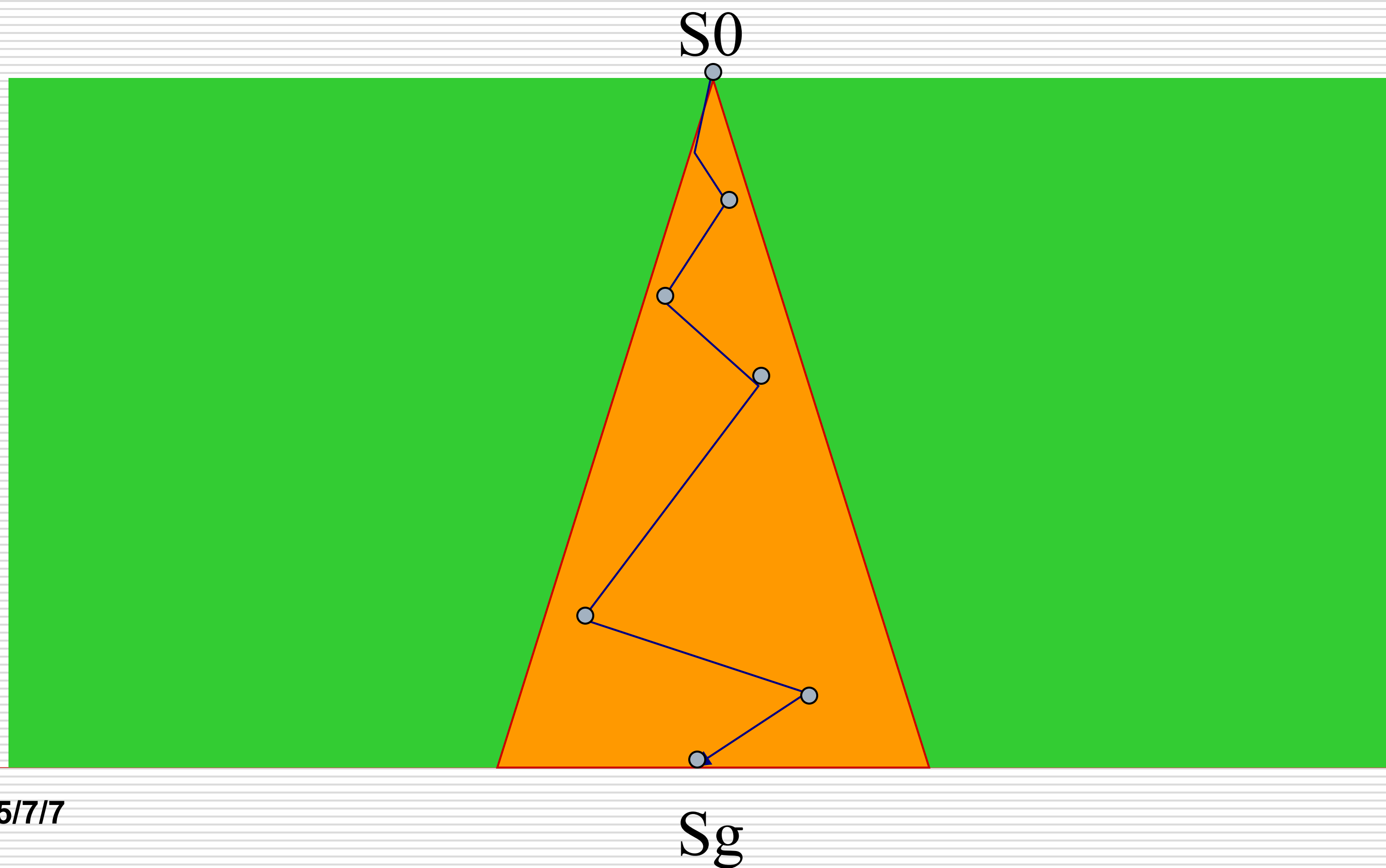
(4)一般图搜索例子——八数码游戏

- 第二步：制定操作算子集
  - 直观方法——为每个棋牌制定一套可能的走步：左、上、右、下四种移动。
    - 需要32个操作算子
  - 简易方法——仅为空格制定这4种走步。
    - 仅需4个操作算子
- 第三步：设计搜索引擎
  - 问题状态空间的大小与问题涉及的元素个数是程指数级爆炸式增长（即，组合爆炸问题）
    - 如，棋盘布局（问题状态）总共有9!=362880个
  - 研究焦点是解决组合爆炸问题，通过压缩搜索空间，提高搜索效率。

状态空间搜索

——1.状态空间及其搜索的表示

状态空间、搜索图和解答路径之间的关系

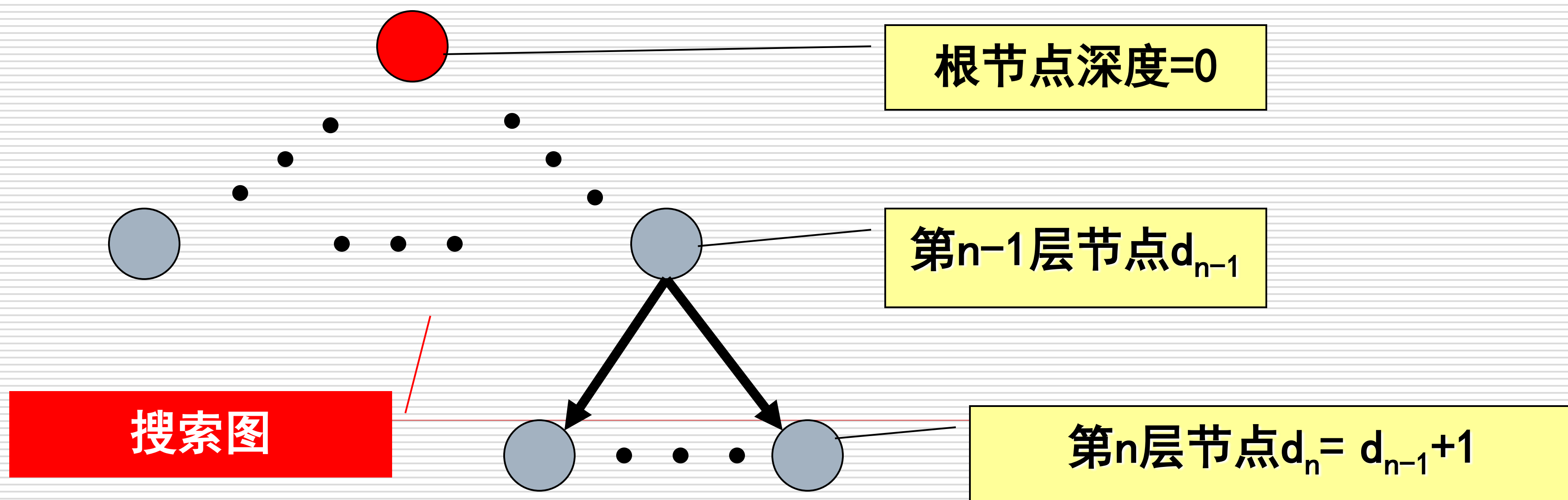


状态空间搜索

——2.一般图搜索策略

（1）搜索术语★

- 1、节点深度
  - **根节点**指示**初始状态**，令其深度为0；
  - 搜索图中的其他节点的**深度**d_n就可以递归地定义为其**父节点深度**d_{n-1}加1：$d_n = d_{n-1}+1$。

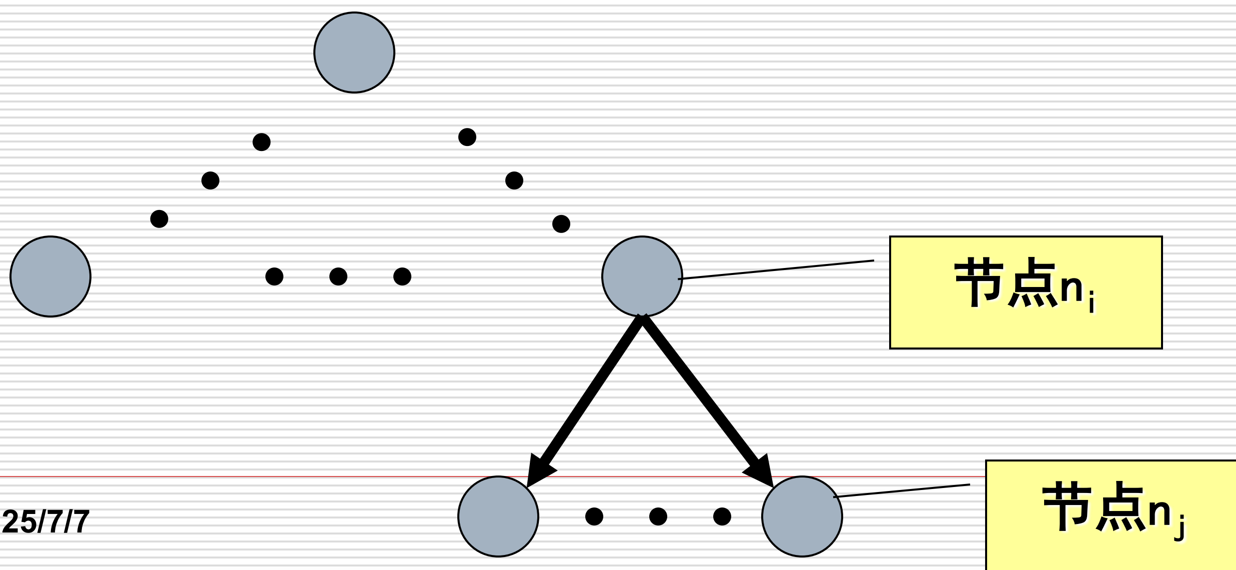


状态空间搜索

——2.一般图搜索策略

（1）搜索术语★

- 2、节点扩展
  - 应用操作算子将上一状态（节点n_i）变迁到下一状态（节点n_j）
  - 节点n_i称为被扩展节点（父节点）
  - 节点n_j称为n_i的子节点

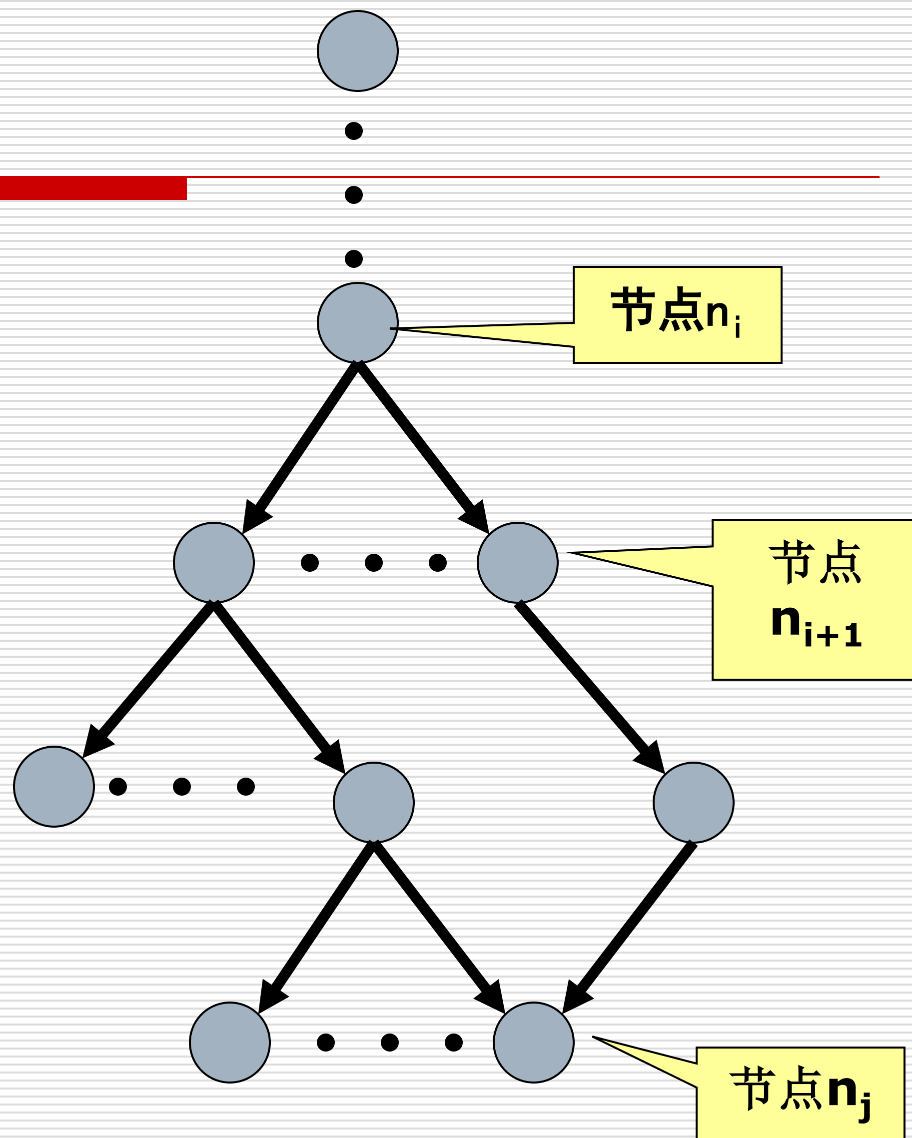


状态空间搜索

——2.一般图搜索策略

（1）搜索术语★

- 3、路径
  - 节点n_i到n_j的路径是由相邻节点间的**弧线构成的折线**
  - 要求路径是无环的
- 4、路径代价——**相邻节点n_i和n_{i+1}间的路径代价**记为$C(n_i, n_{i+1})$
  - 通常令**任意相邻节点间的路径代价相同**，并以路径长度1指示。
  - 即$C(n_i, n_{i+1})=1$ 。



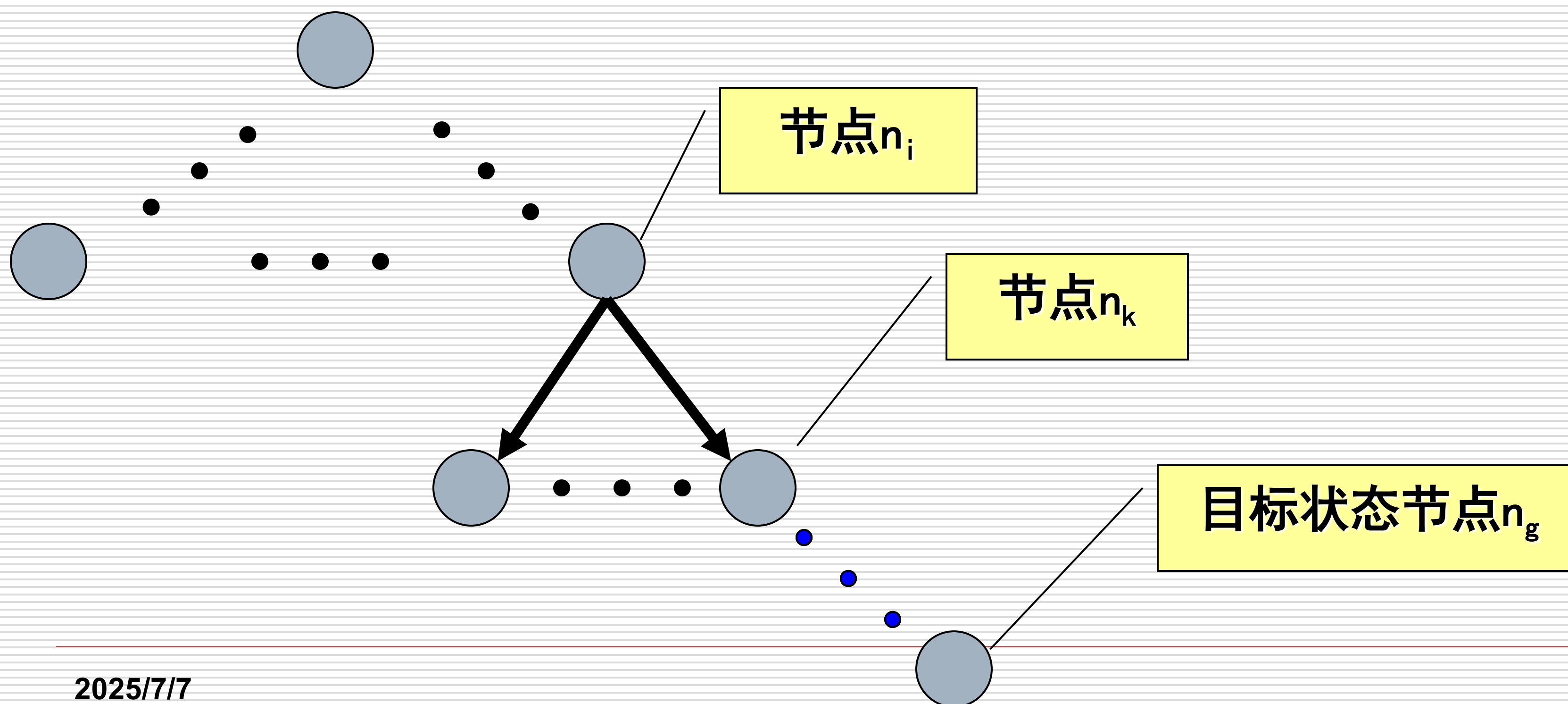
状态空间搜索

——2.一般图搜索策略

（1）搜索术语★

- 4、路径代价

$C(n_i, n_g) = C(n_i, n_k) + C(n_k, n_g)$



状态空间搜索

——2.一般图搜索策略

（2）一般图搜索算法★

- 符号说明：
  - s-初始状态节点
  - G-搜索图
  - OPEN-存放待扩展节点的表
  - CLOSE-存放已被扩展的节点的表
  - MOVE-FIRST(OPEN)-取OPEN表首的节点作为当前要被扩展的节点n，同时将节点n移至CLOSE表
- 一般图搜索算法划分为二个阶段：
  - 1、初始化
  - 2、搜索循环

状态空间搜索
——2.一般图搜索策略

（2）一般图搜索算法★

- 算法划分为二个阶段：
  - 1、初始化
    - 建立只包含初始状态节点s的搜索图G:={s}
    - OPEN:={s}
    - CLOSE:={}
  - 2、搜索循环
    - MOVE-FIRST(OPEN)-取出OPEN表首的节点n作为扩展的节点，同时将其移到close表
    - 扩展出n的子节点,插入搜索图G和OPEN表
    - 适当的标记和修改指针
    - 排序OPEN表
  - 通过循环地执行该算法，搜索图G会因不断有新节点加入而逐步长大，直到搜索到目标节点。

状态空间搜索

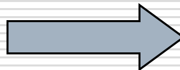
——2.一般图搜索策略

以下面的八数码为例，看一般图的搜索算法

1		3
7	2	4
6	8	5

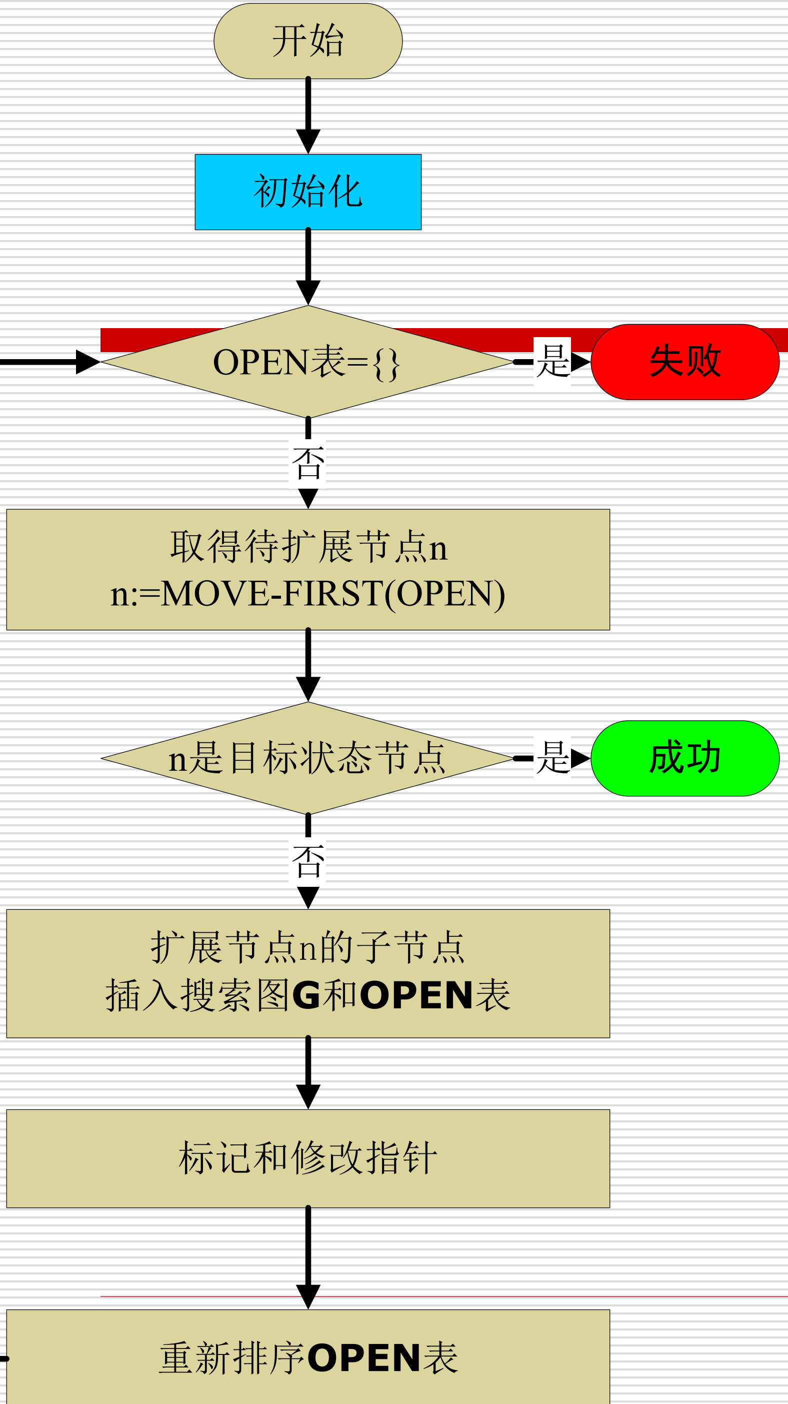
初始布局

移动数码

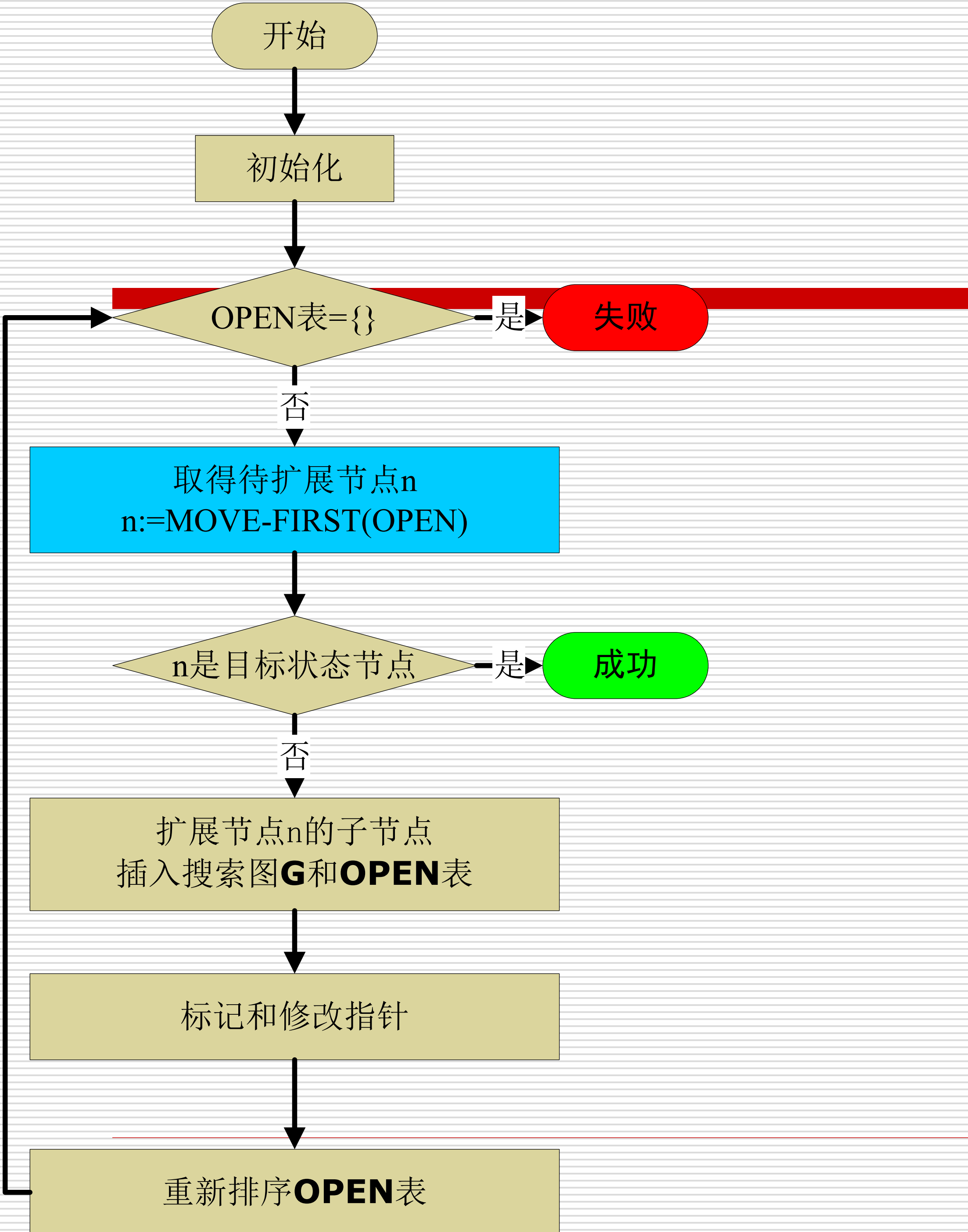


1	2	3
8		4
7	6	5

目标布局



$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$



$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

开始

初始化

OPEN表={}

是 → 失败

否

取得待扩展节点n
n:=MOVE-FIRST(OPEN)

n是目标状态节点

是 → 成功

否

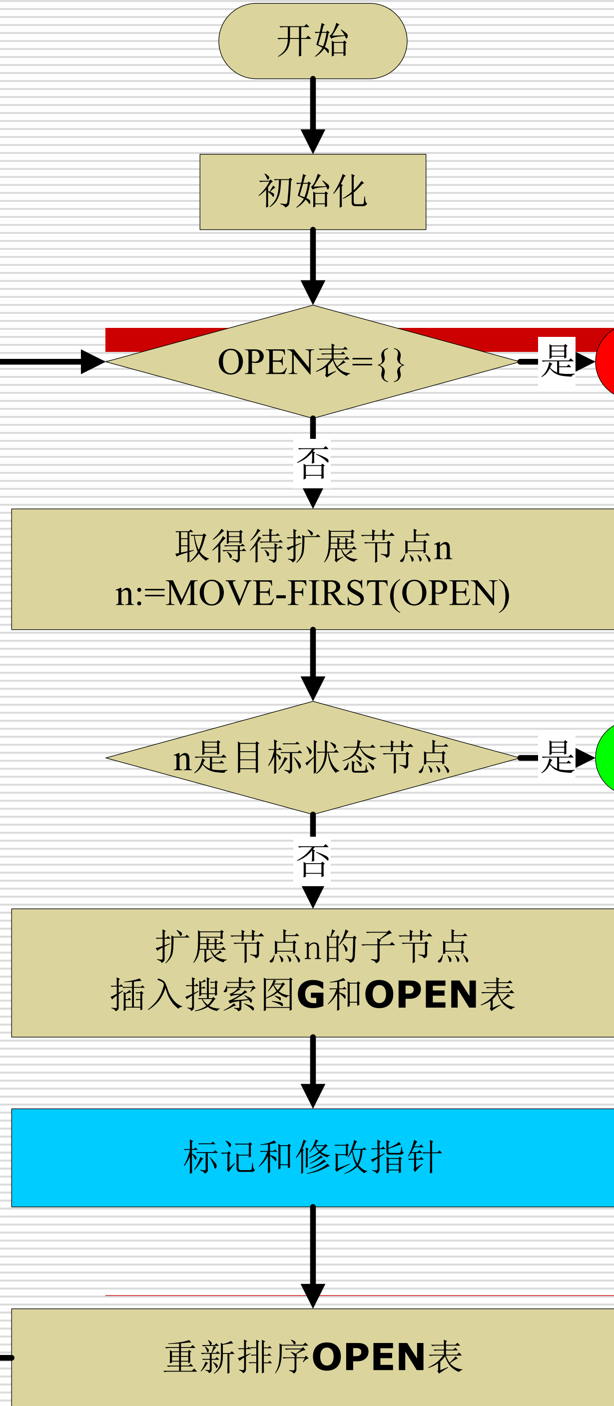
扩展节点n的子节点
插入搜索图**G**和**OPEN**表

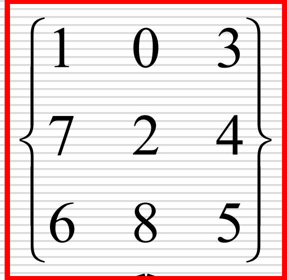
标记和修改指针

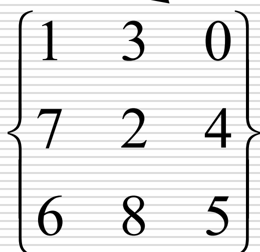
重新排序**OPEN**表

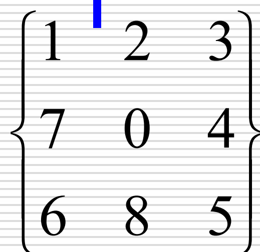
$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

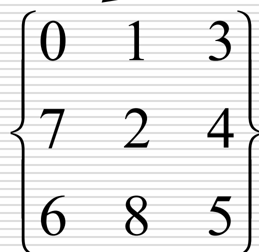
$$\begin{Bmatrix} 1 & 3 & 0 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 0 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 0 & 1 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

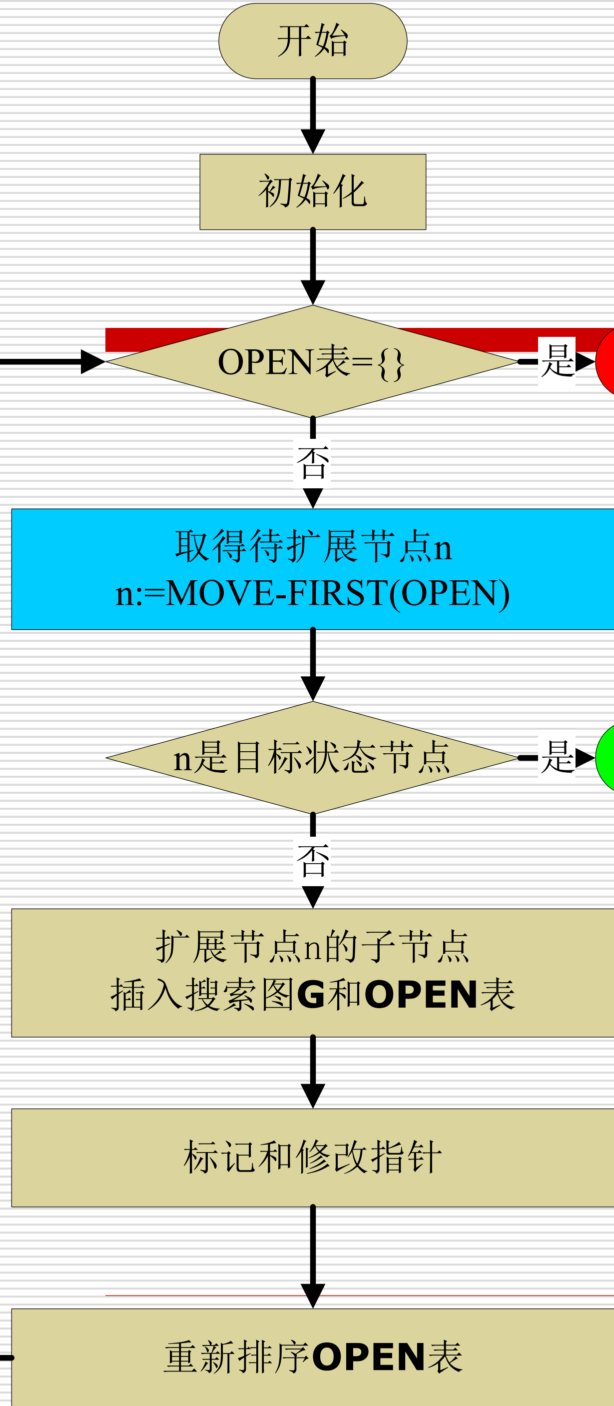

开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表


1 0 3
7 2 4
6 8 5

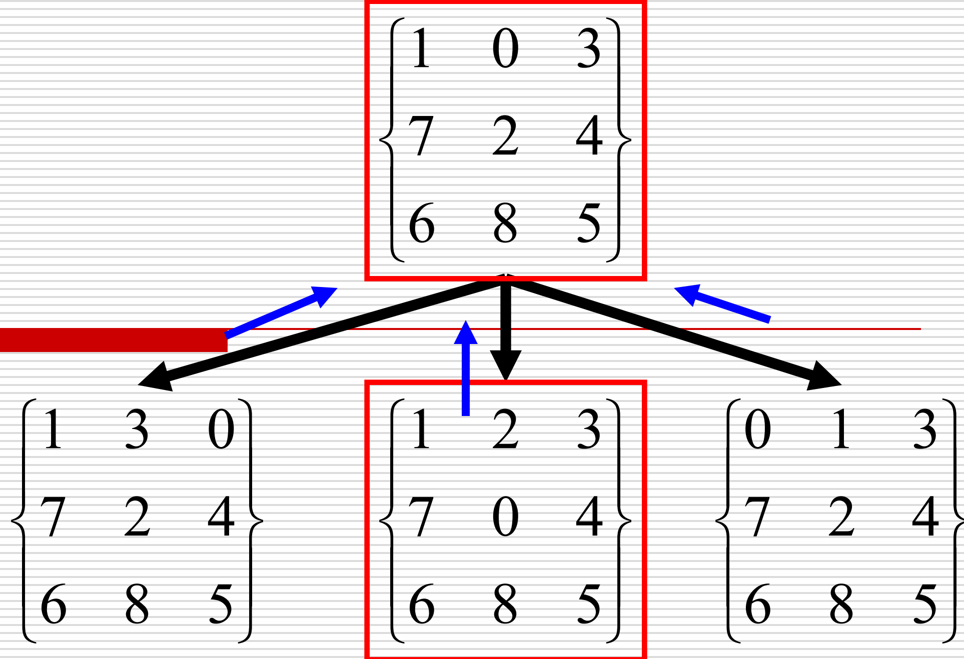

1 3 0
7 2 4
6 8 5


1 2 3
7 0 4
6 8 5

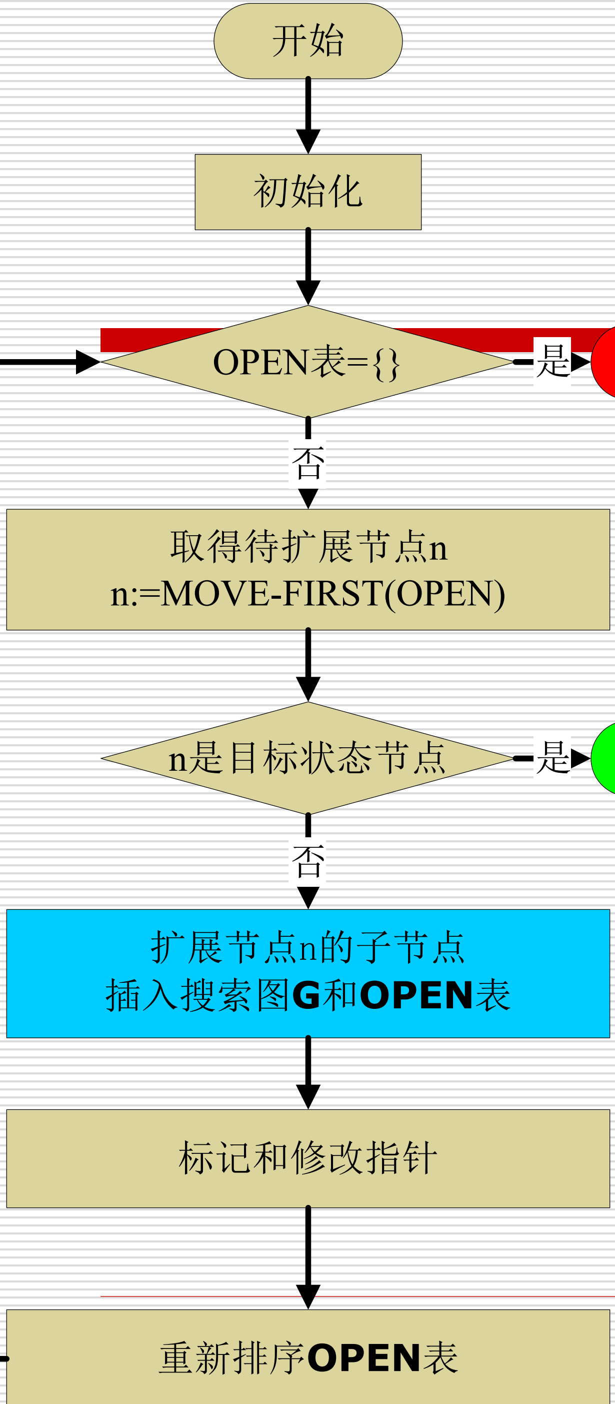

0 1 3
7 2 4
6 8 5

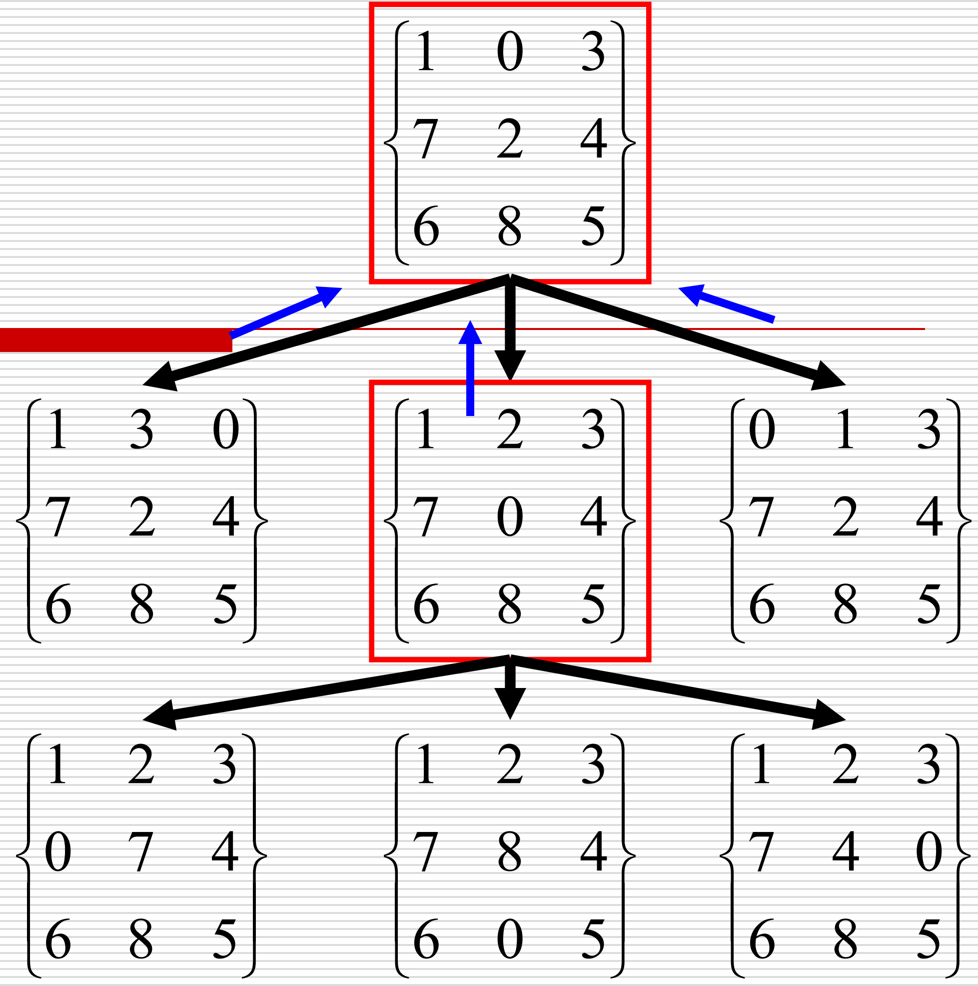


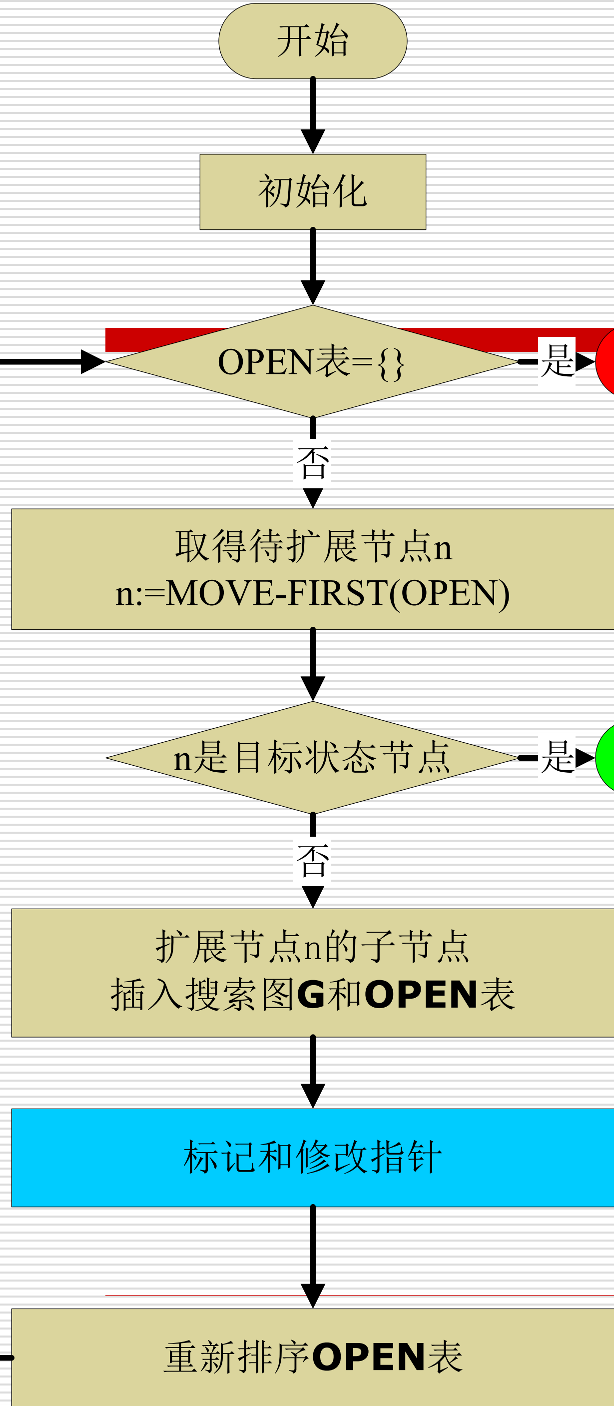
开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表

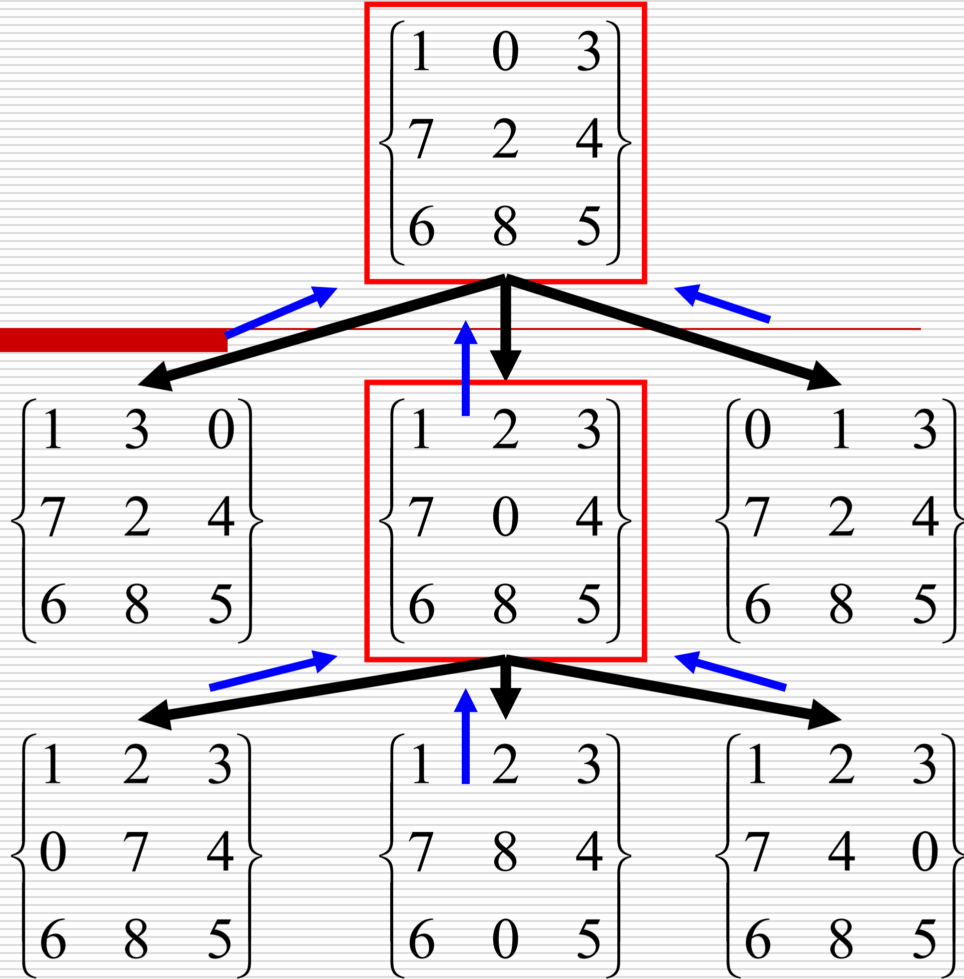


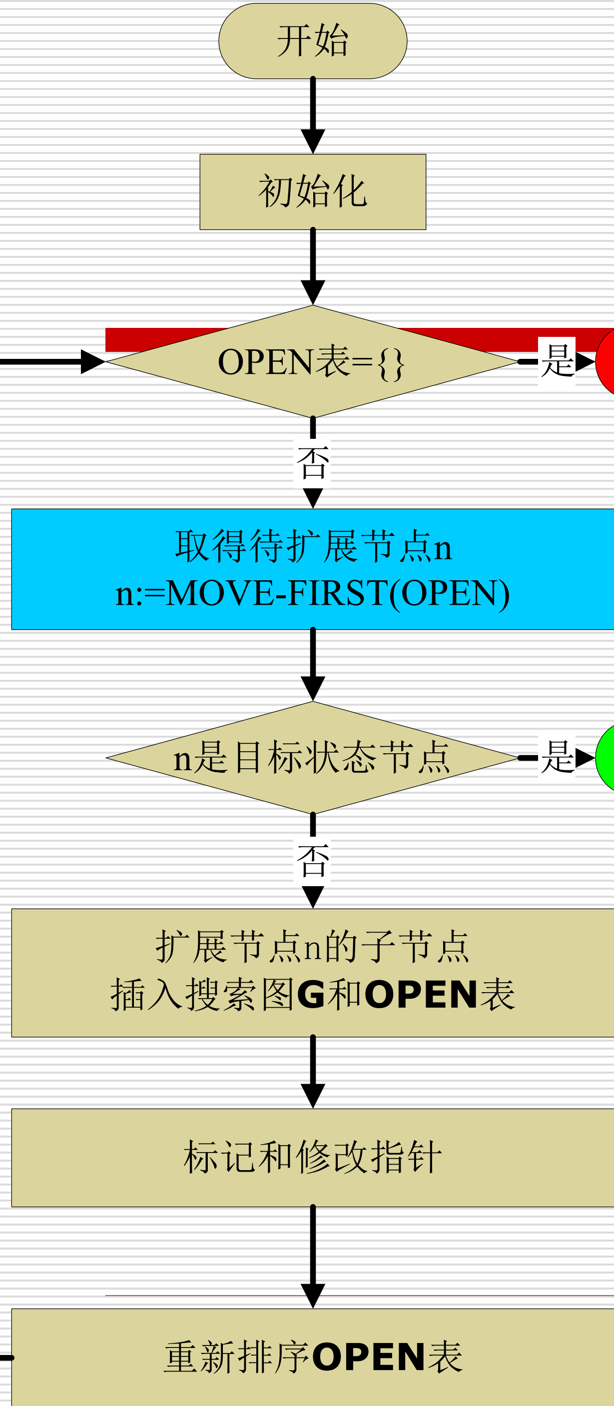
1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5

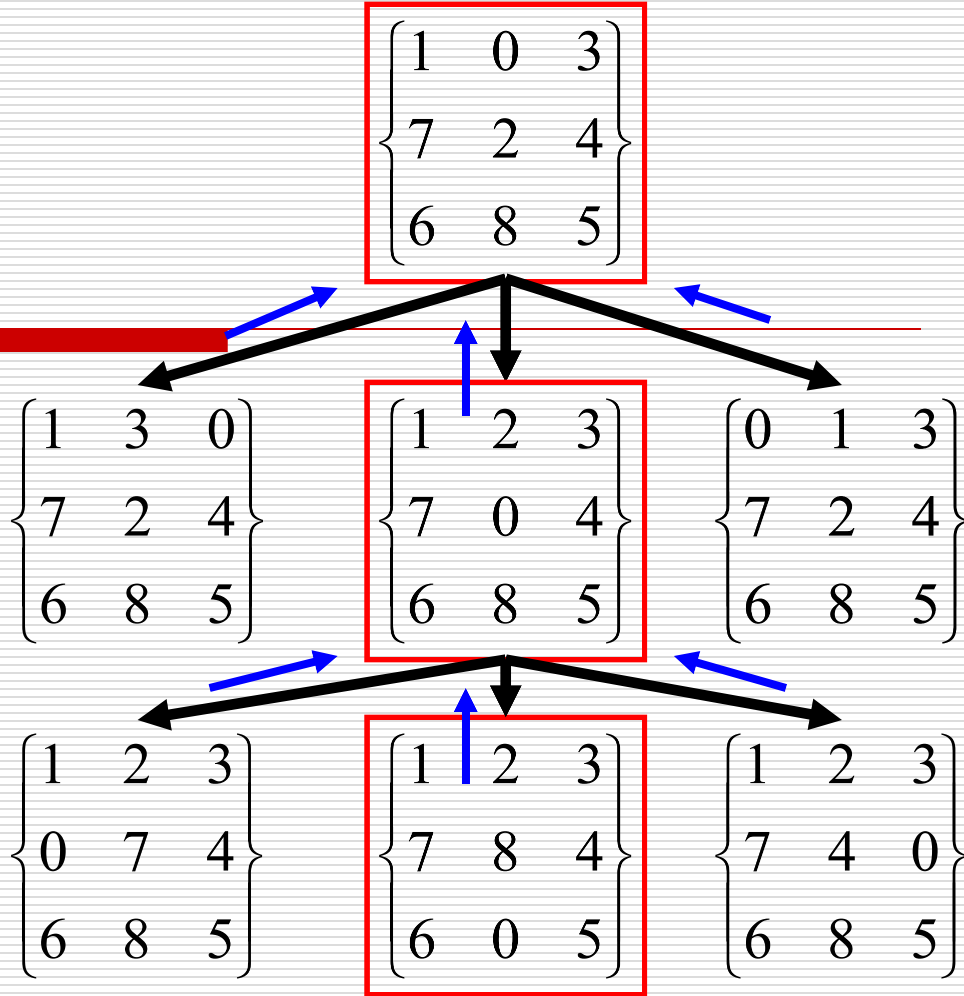

开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表

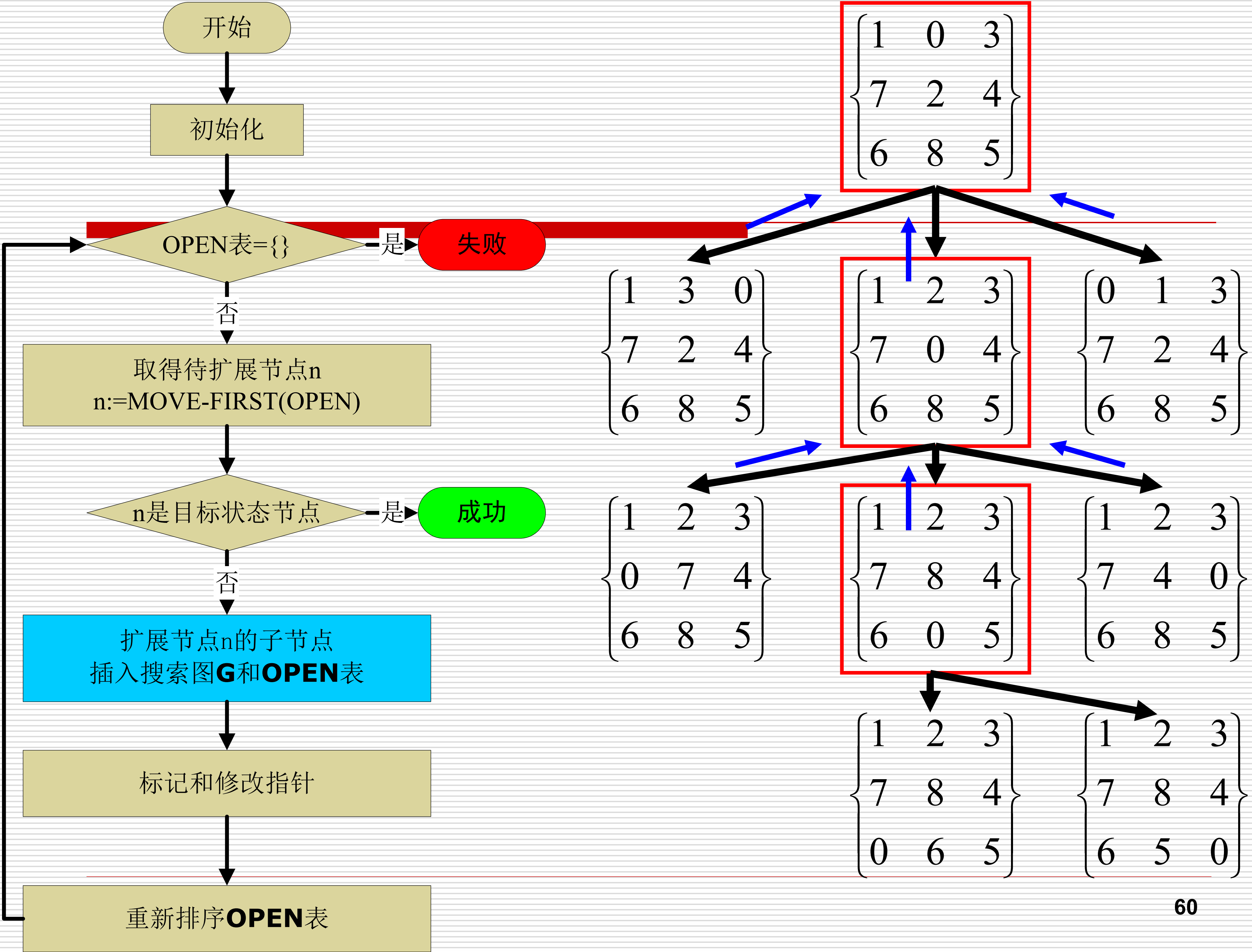

1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表

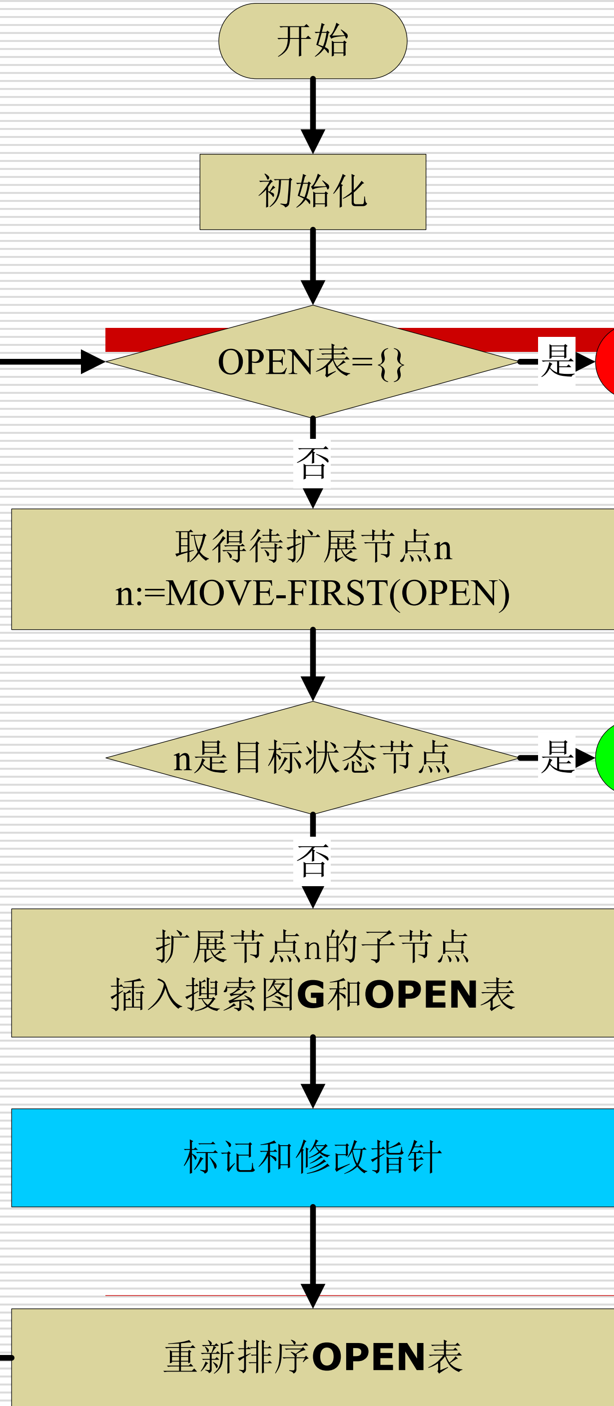

1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表


1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5



开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表
1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5
1 2 3
7 8 4
0 6 5
1 2 3
7 8 4
6 5 0

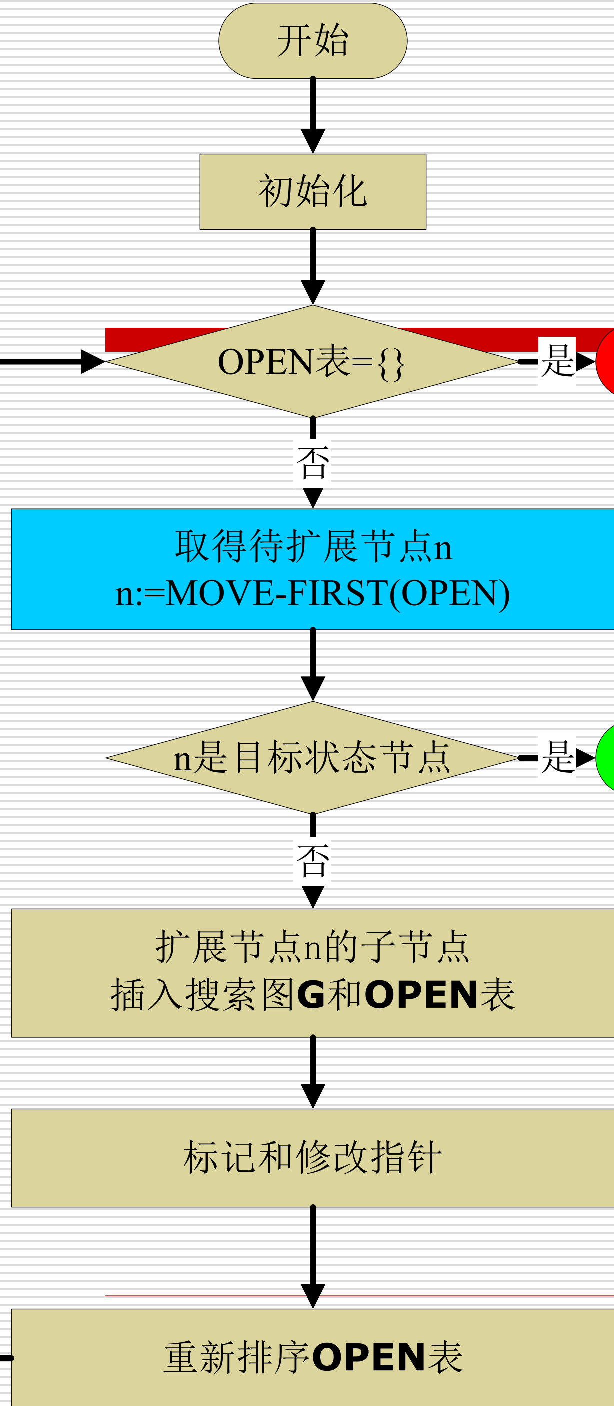


$$\begin{Bmatrix}1 & 0 & 3\\ 7 & 2 & 4\\ 6 & 8 & 5\end{Bmatrix}$$

$$\begin{Bmatrix}1 & 3 & 0\\ 7 & 2 & 4\\ 6 & 8 & 5\end{Bmatrix} \quad \begin{Bmatrix}1 & 2 & 3\\ 7 & 0 & 4\\ 6 & 8 & 5\end{Bmatrix} \quad \begin{Bmatrix}0 & 1 & 3\\ 7 & 2 & 4\\ 6 & 8 & 5\end{Bmatrix}$$

$$\begin{Bmatrix}1 & 2 & 3\\ 0 & 7 & 4\\ 6 & 8 & 5\end{Bmatrix} \quad \begin{Bmatrix}1 & 2 & 3\\ 7 & 8 & 4\\ 6 & 0 & 5\end{Bmatrix} \quad \begin{Bmatrix}1 & 2 & 3\\ 7 & 4 & 0\\ 6 & 8 & 5\end{Bmatrix}$$

$$\begin{Bmatrix}1 & 2 & 3\\ 7 & 8 & 4\\ 0 & 6 & 5\end{Bmatrix} \quad \begin{Bmatrix}1 & 2 & 3\\ 7 & 8 & 4\\ 6 & 5 & 0\end{Bmatrix}$$

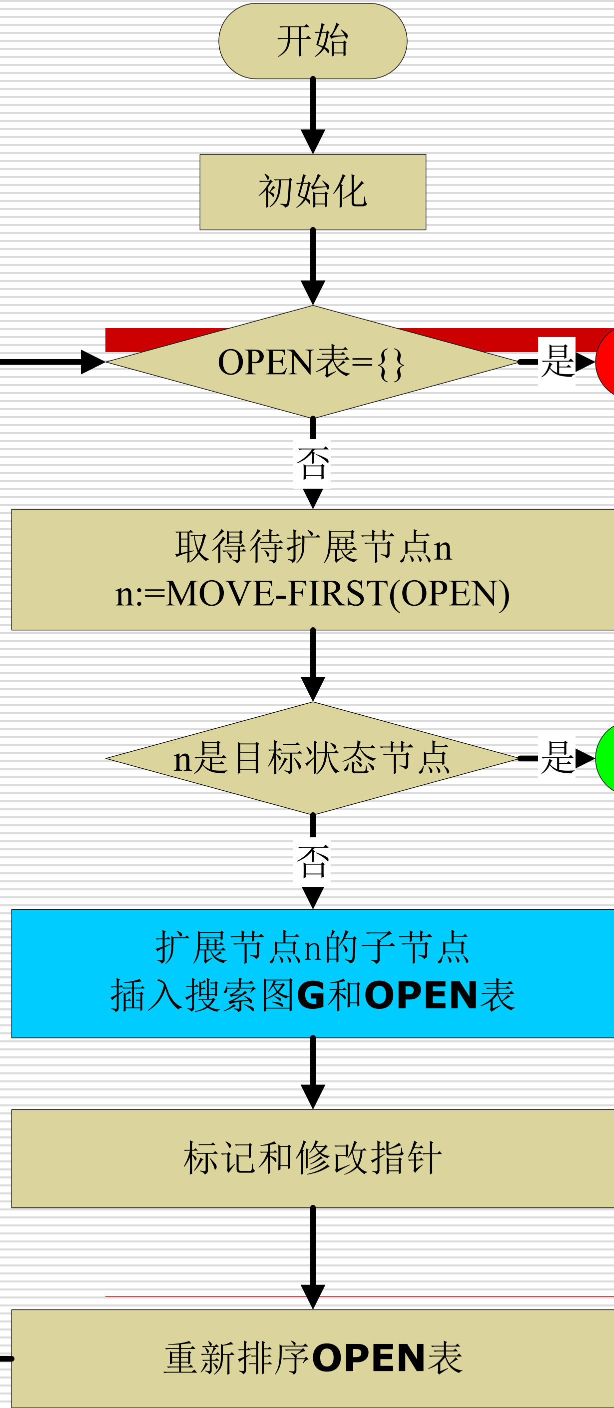


$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 3 & 0 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 0 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 0 & 1 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 7 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 0 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 4 & 0 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 0 & 6 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 5 & 0 \end{Bmatrix}$$



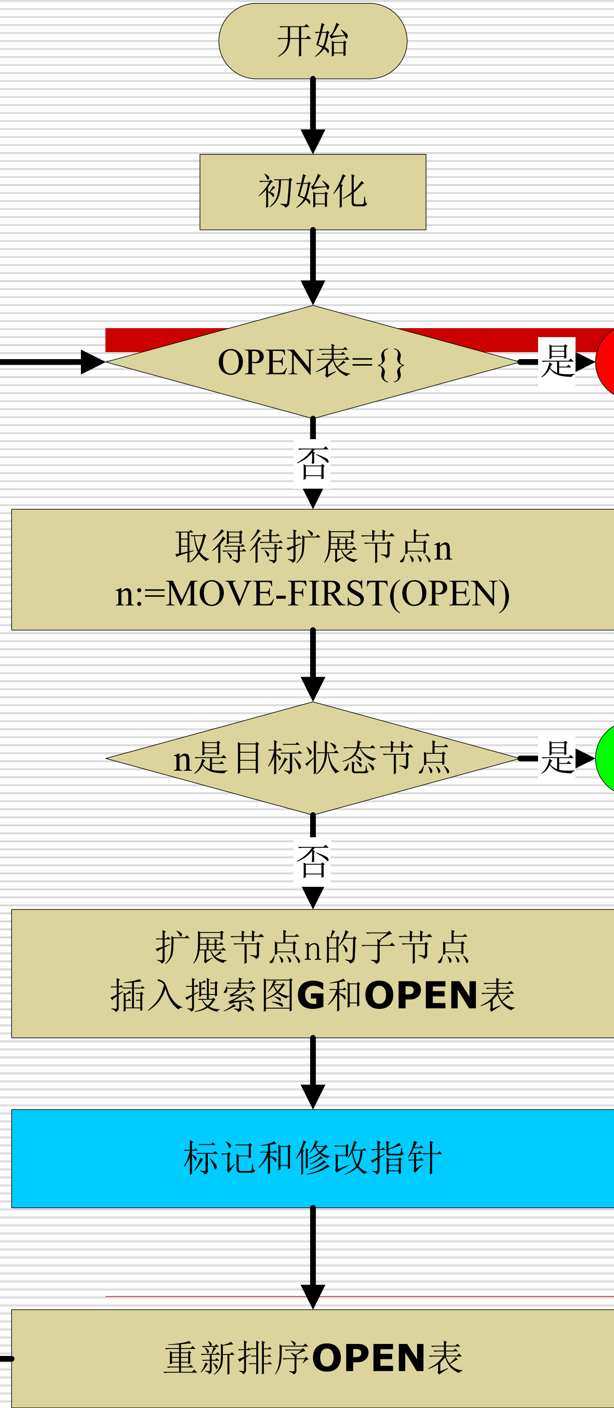
$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 3 & 0 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 0 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 0 & 1 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 7 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 0 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 4 & 0 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 0 & 6 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 5 & 0 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{Bmatrix}$$



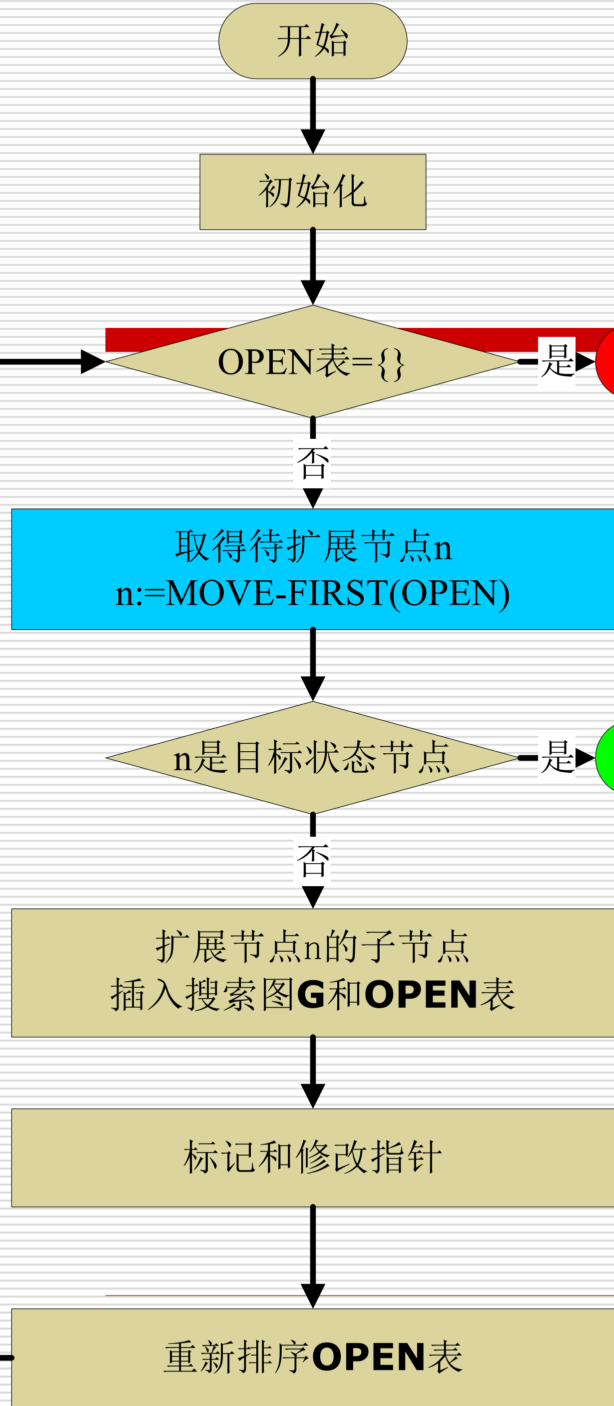
$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$

$\begin{Bmatrix} 1 & 3 & 0 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$ $\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 0 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$ $\begin{Bmatrix} 0 & 1 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$

$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 7 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$ $\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 0 & 5 \end{Bmatrix}$ $\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 4 & 0 \\ 6 & 8 & 5 \end{Bmatrix}$

$\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 0 & 6 & 5 \end{Bmatrix}$ $\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 5 & 0 \end{Bmatrix}$

$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{Bmatrix}$



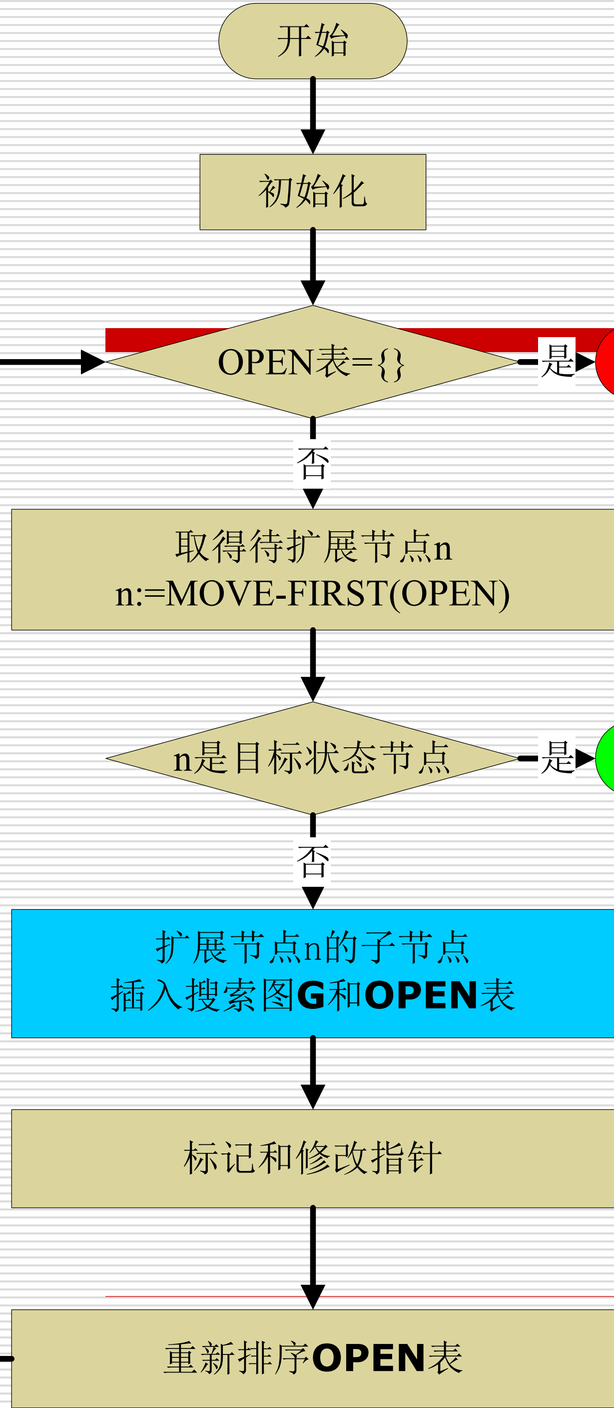
$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 3 & 0 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 0 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 0 & 1 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 7 & 4 \\ 6 & 8 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 0 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 4 & 0 \\ 6 & 8 & 5 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 0 & 6 & 5 \end{Bmatrix} \quad \begin{Bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 6 & 5 & 0 \end{Bmatrix}$$

$$\begin{Bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{Bmatrix}$$



$$\begin{Bmatrix}1 & 0 & 3\\7 & 2 & 4\\6 & 8 & 5\end{Bmatrix}$$

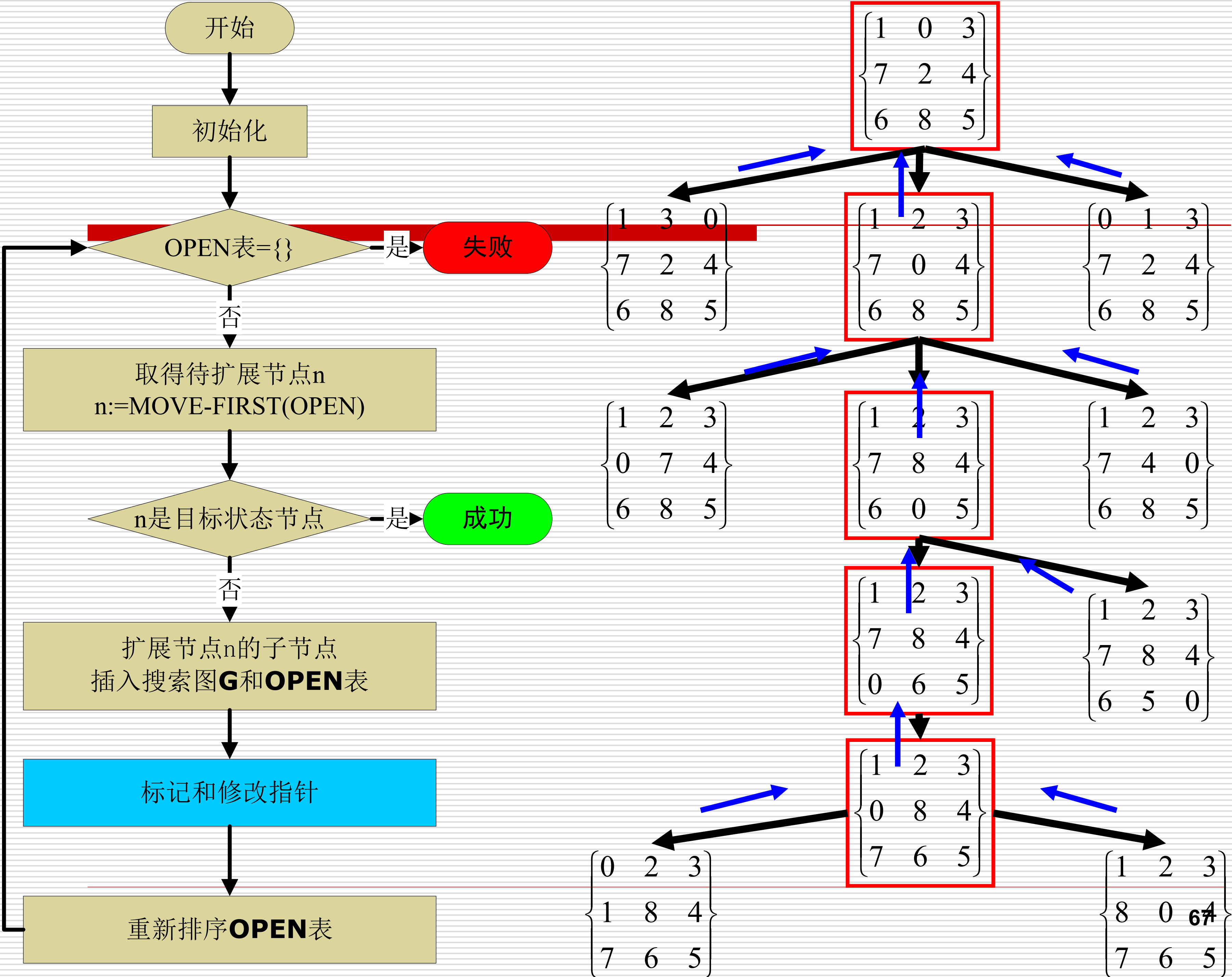
$$\begin{Bmatrix}1 & 3 & 0\\7 & 2 & 4\\6 & 8 & 5\end{Bmatrix}\quad\begin{Bmatrix}1 & 2 & 3\\7 & 0 & 4\\6 & 8 & 5\end{Bmatrix}\quad\begin{Bmatrix}0 & 1 & 3\\7 & 2 & 4\\6 & 8 & 5\end{Bmatrix}$$

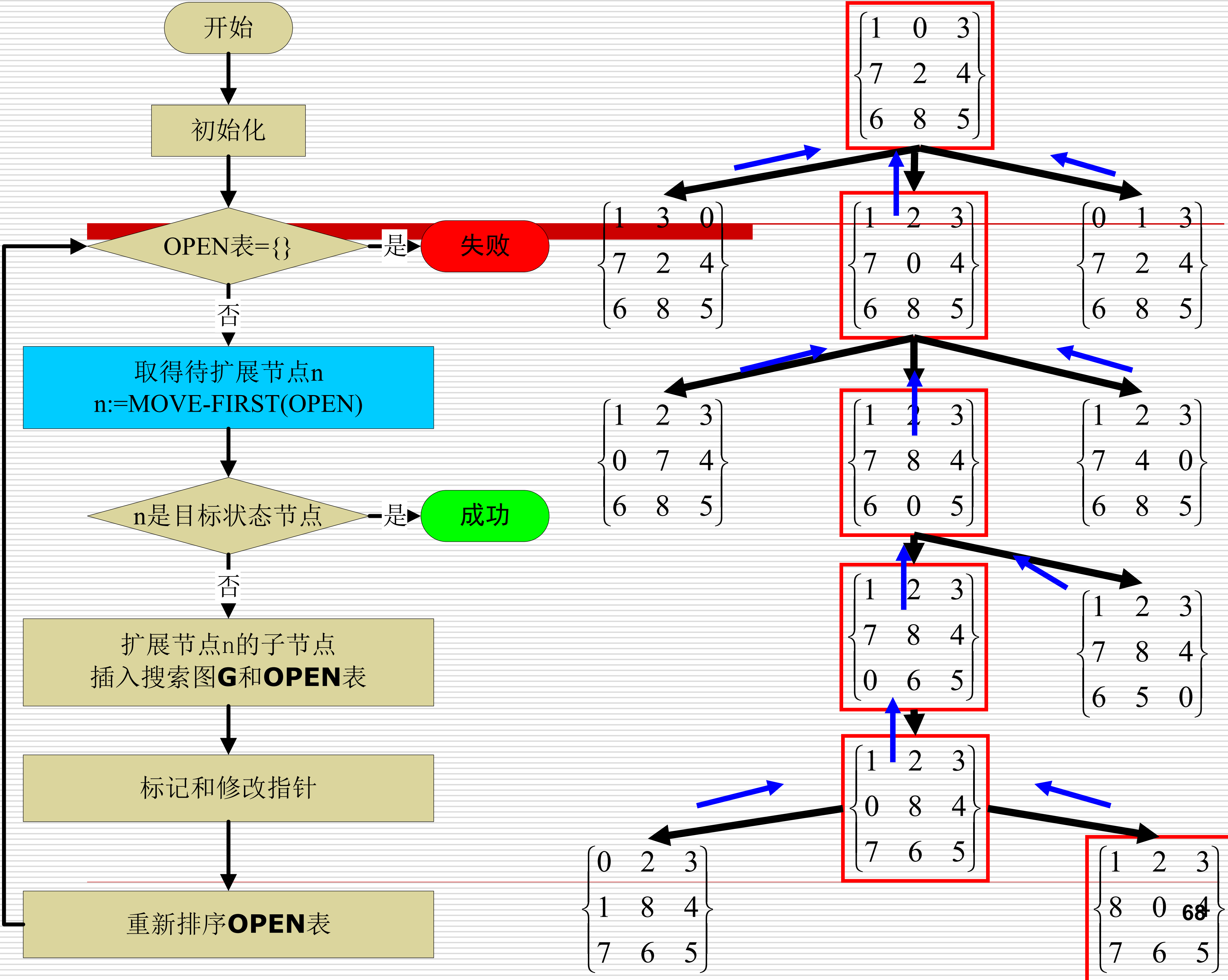
$$\begin{Bmatrix}1 & 2 & 3\\0 & 7 & 4\\6 & 8 & 5\end{Bmatrix}\quad\begin{Bmatrix}1 & 2 & 3\\7 & 8 & 4\\6 & 0 & 5\end{Bmatrix}\quad\begin{Bmatrix}1 & 2 & 3\\7 & 4 & 0\\6 & 8 & 5\end{Bmatrix}$$

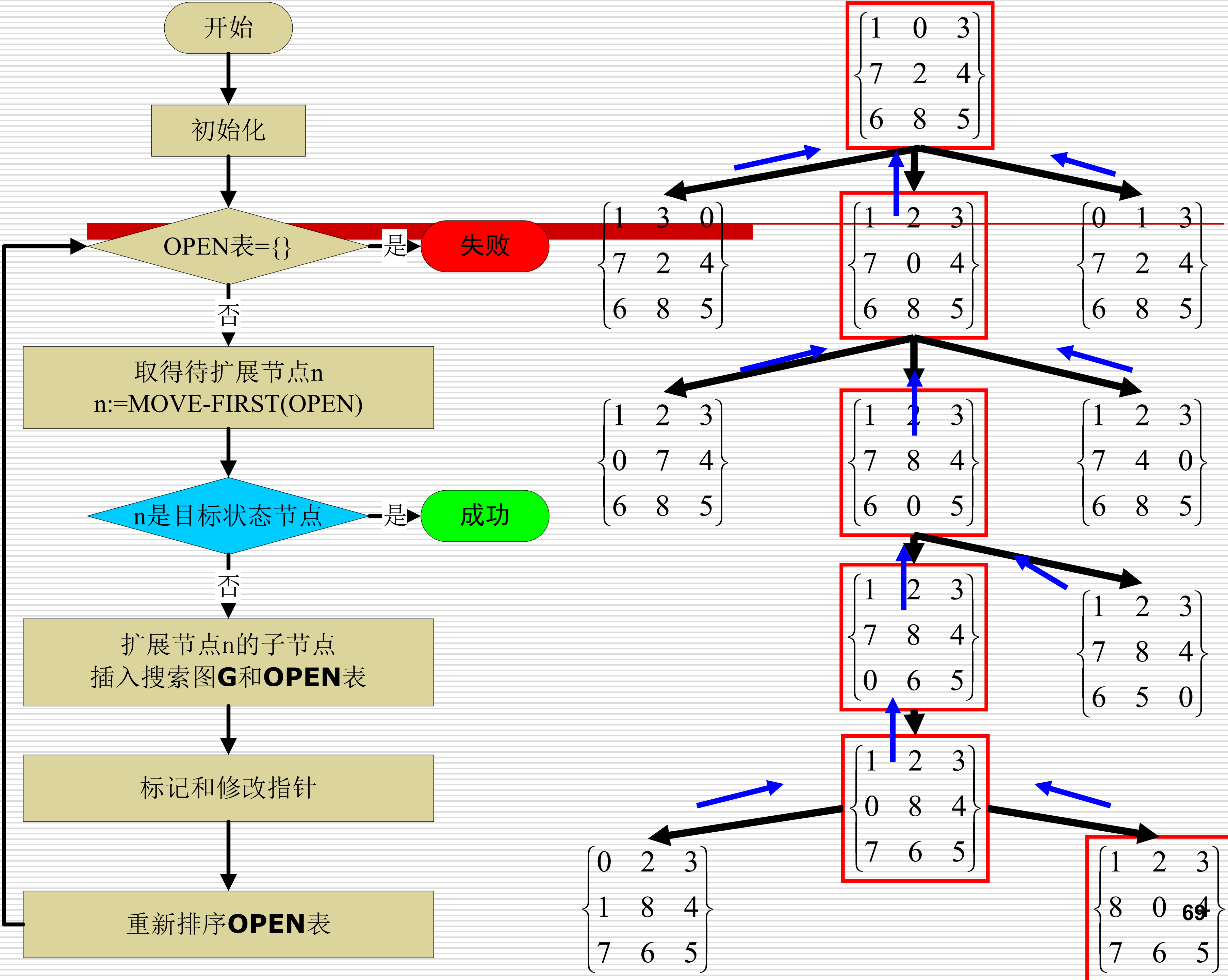
$$\begin{Bmatrix}1 & 2 & 3\\7 & 8 & 4\\0 & 6 & 5\end{Bmatrix}\quad\begin{Bmatrix}1 & 2 & 3\\7 & 8 & 4\\6 & 5 & 0\end{Bmatrix}$$

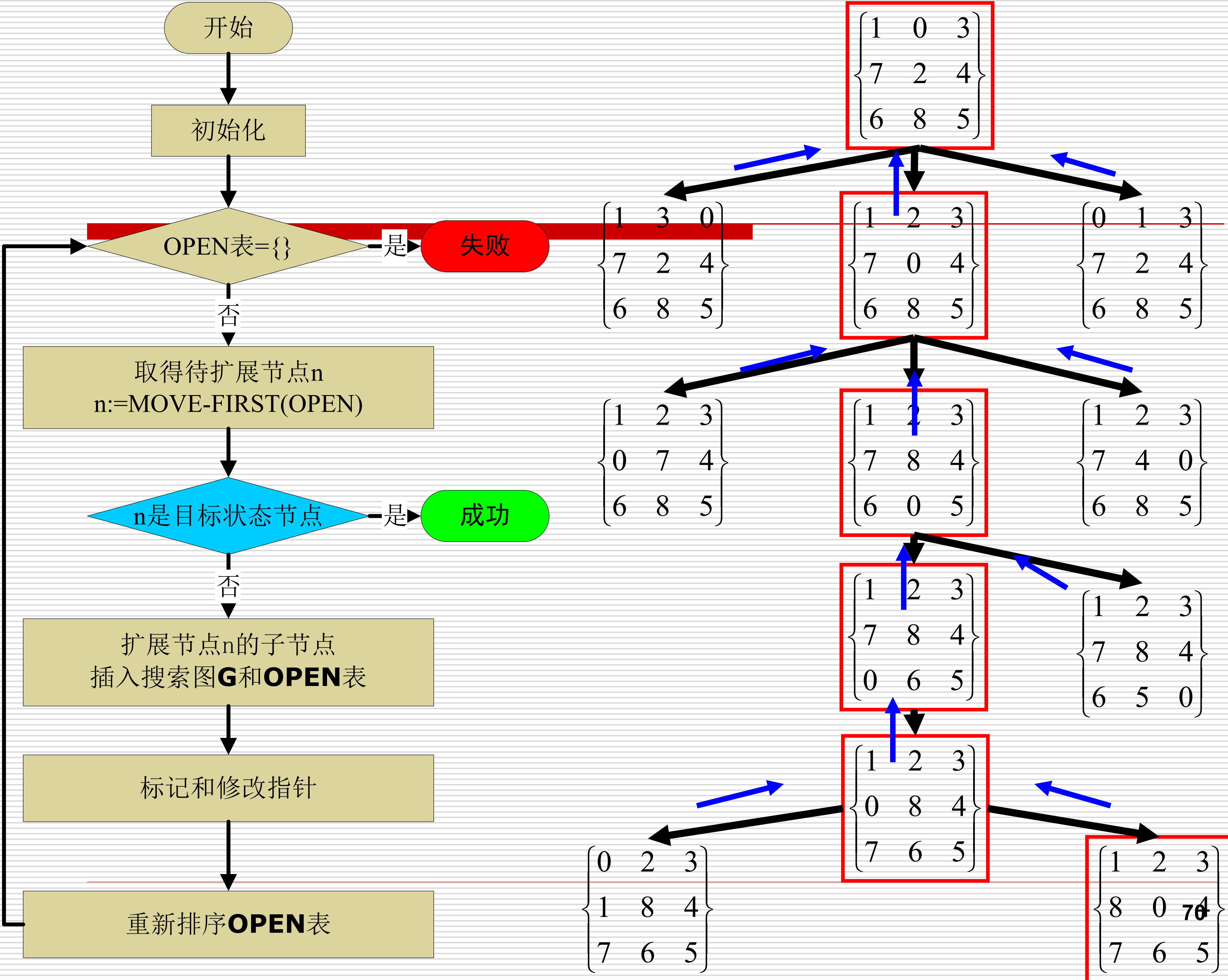
$$\begin{Bmatrix}1 & 2 & 3\\0 & 8 & 4\\7 & 6 & 5\end{Bmatrix}$$

$$\begin{Bmatrix}0 & 2 & 3\\1 & 8 & 4\\7 & 6 & 5\end{Bmatrix}\quad\begin{Bmatrix}1 & 2 & 3\\8 & 0 & 4\\7 & 6 & 5\end{Bmatrix}$$


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表
1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5
1 2 3
7 8 4
0 6 5
1 2 3
7 8 4
6 5 0
1 2 3
0 8 4
7 6 5
0 2 3
1 8 4
7 6 5
1 2 3
8 0 4
7 6 5


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表
1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5
1 2 3
7 8 4
0 6 5
1 2 3
7 8 4
6 5 0
1 2 3
0 8 4
7 6 5
0 2 3
1 8 4
7 6 5
1 2 3
8 0 4
7 6 5


开始
初始化
OPEN表={}
是
失败
否
取得待扩展节点n
n:=MOVE-FIRST(OPEN)
n是目标状态节点
是
成功
否
扩展节点n的子节点
插入搜索图G和OPEN表
标记和修改指针
重新排序OPEN表
1 0 3
7 2 4
6 8 5
1 3 0
7 2 4
6 8 5
1 2 3
7 0 4
6 8 5
0 1 3
7 2 4
6 8 5
1 2 3
0 7 4
6 8 5
1 2 3
7 8 4
6 0 5
1 2 3
7 4 0
6 8 5
1 2 3
7 8 4
0 6 5
1 2 3
7 8 4
6 5 0
1 2 3
0 8 4
7 6 5
0 2 3
1 8 4
7 6 5
1 2 3
8 0 4
7 6 5

状态空间搜索

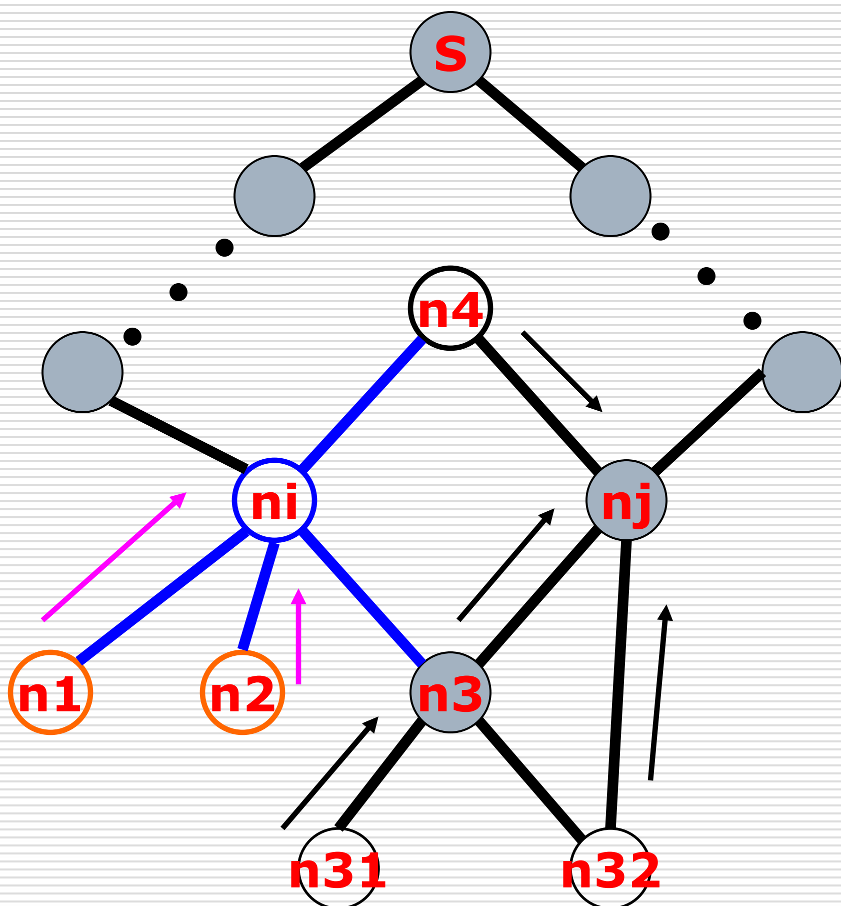
——2.一般图搜索策略

（2）一般图搜索算法——搜索过程中的指针修改★

- 节点n扩展后的子节点分为3类：
  - (i)全新节点
  - (ii)已出现在OPEN表中的节点
  - (iii)已出现的CLOSE表中的节点
- 指针标记和修改的方法：
  - (i)类节点：加入OPEN表，建立从子节点到父节点n的指针
  - (ii)类节点、(iii)类节点
    - 比较经由老父节点、新父节点n到达初始状态节点的路径代价
    - 经由节点n的代价比较小，则移动子节点指向老父节点的指针，改为指向新父节点n
    - (iii)类节点还得从CLOSE表中移出，重新加入OPEN表。

状态空间搜索

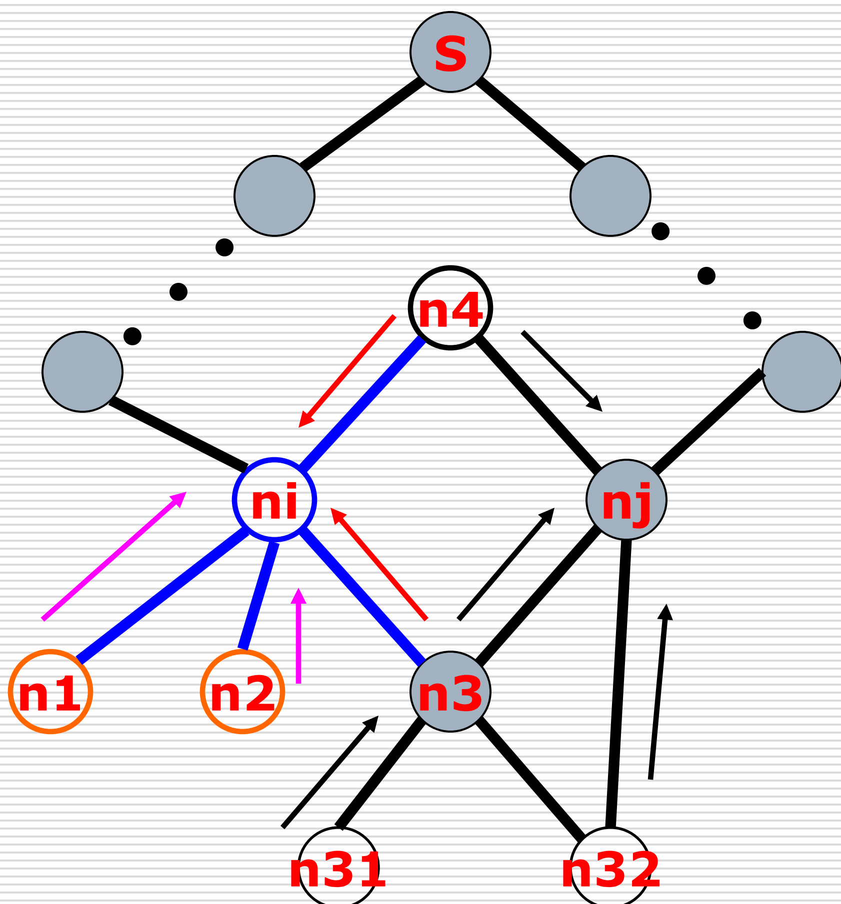
——2.一般图搜索策略



- 节点ni是当前扩展的节点；
- 扩展出4个后续节点；
- n1、n2是新节点，只需建立指向父节点的指针，并加入OPEN表；

状态空间搜索

——2.一般图搜索策略



- n4已经存在于OPEN表，并且已有父节点nj
  - n4经nj的路径代价大
  - 取消n4指向nj的指针
  - 改为建立n4指向新父节点ni的指针
- n3已经存在于CLOSE表，并且已有父节点nj
  - 需要做和n4同样的比较和指针修改工作。并且要重新加入open表。

4.3 盲目搜索

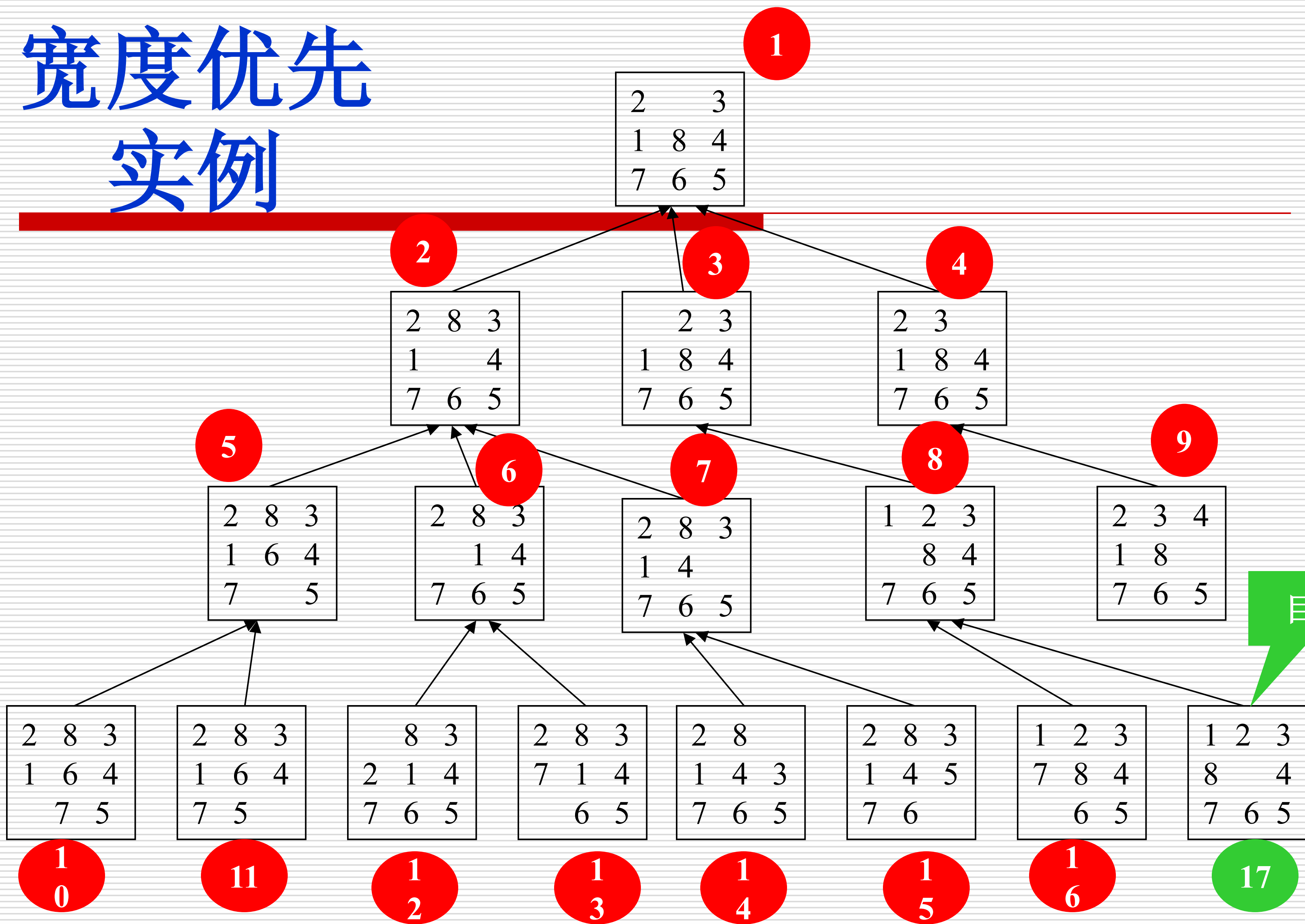
- 提高搜索效率的关键——优化OPEN表中节点的排序方式。
- 简单的排序策略——按预先确定的顺序或随机地排序新加入到OPEN表中的节点。
- 常用的简单方式：

  - 宽度优先——扩展当前节点后生成的子节点总是置于OPEN表的后端，即OPEN表作为队列使用，先进先出，使搜索优先向横广方向发展。

  - 深度优先——扩展当前节点后生成的子节点总是置于OPEN表的前端，即OPEN表作为栈使用，后进先出，使搜索优先向纵深方向发展。

宽度优先

- 宽度优先——扩展当前节点后生成的子节点总是置于OPEN表的后端，即OPEN表作为队列，先进先出，使搜索优先向横向方向发展。
- 过程：指从初始节点S_0开始，向下逐层搜索，在n层节点未搜索完之前，不进入n+1层搜索。同层节点的搜索次序可以任意。即先按生成规则生成第一层节点，在该层全部节点中沿宽(广)度进行横向扫描，检查目标节点Sg是否在这些子节点中。若没有,则再将所有第一层节点逐一扩展，得到第二层节点。并逐一检查第二层节点中是否包含有S_g，如此依次按照先生成、先检查、先扩展的原则进行下去，直到发现S_g为止


宽度优先
实例
1
2 3
1 8 4
7 6 5
2
2 8 3
1 4
7 6 5
3
2 3
1 8 4
7 6 5
4
2 3
1 8 4
7 6 5
5
2 8 3
1 6 4
7 5
6
2 8 3
1 4
7 6 5
7
2 8 3
1 4
7 6 5
8
1 2 3
8 4
7 6 5
9
2 3 4
1 8
7 6 5
目标
2 8 3
1 6 4
7 5
10
2 8 3
1 6 4
7 5
11
8 3
2 1 4
7 6 5
12
2 8 3
7 1 4
6 5
13
2 8
1 4 3
7 6 5
14
2 8 3
1 4 5
7 6
15
1 2 3
7 8 4
6 5
16
1 2 3
8 4
7 6 5
17

宽度优先搜索

- 如果搜索是以接近起始节点的程度依次扩展节点的，那么这种搜索就叫做宽度优先搜索。
- 这种搜索是逐层进行的，在对下一层的任意节点进行搜索之前，必须搜索完本层的所有节点。
- “先产生的节点先扩展”

采用队列结构，宽度优先算法可以表示如下：

(1)把初始节点S_0，放入OPEN表。

(2)如果OPEN表为空。则问题无解，失败并退出。

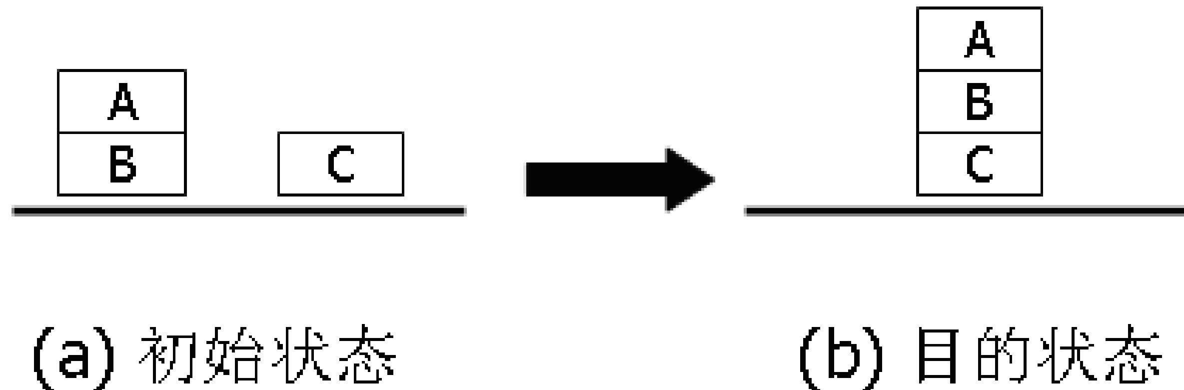
(3)把OPEN表中的第一个节点取出放入CLOSE表中，并按顺序冠以编号n；

(4)考察节点n是否为目标节点。若是，则求得了问题的解，成功并退出。

(5)若节点n不可扩展，则转第(2)步；

(6)扩展节点n，将其子节点放到OPEN表的尾部，并为每一个子节点都配置指向父节点的指针，然后转第(2)步。

例 **通过挪动积木块，希望从初始状态达到一个目的状态，即三块积木堆叠在一起。积木A在顶部，积木B在顶中间，积木C在底部。请画出按照宽度优先搜索策略所产生的搜索树。**



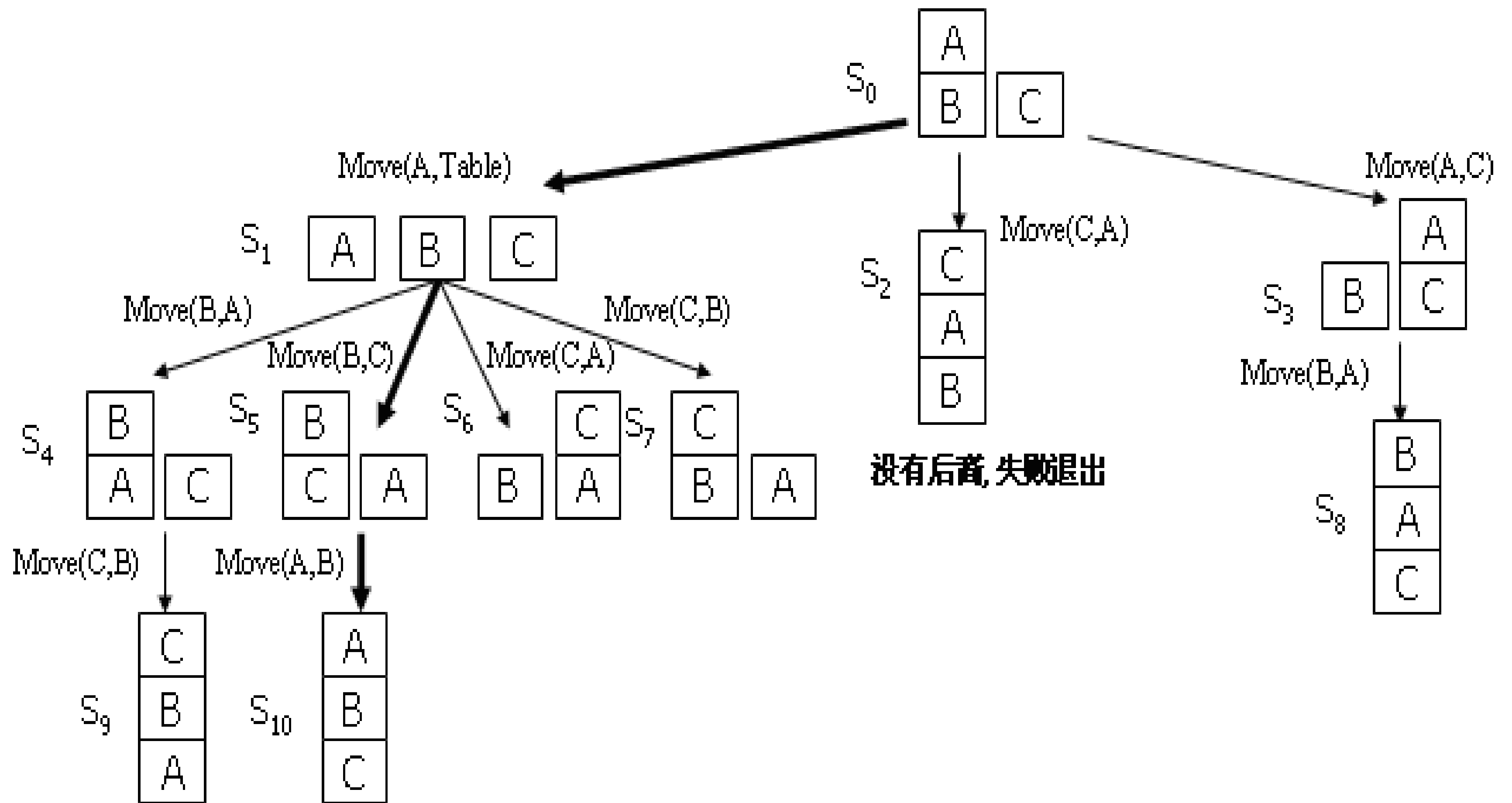
(a) 初始状态　　(b) 目的状态

这个问题的唯一操作算子为MOVE(X,Y)，即积木X搬到Y（积木或桌面）上面。如“挪动积木A到桌面上表示为MOVE(A,Table)。该操作算子可运用的先决条件是：

（1）被挪动积木的顶部必须为空。

（2）如Y是积木（不是桌面），则积木Y的顶部也必须为空。

（3）同一状态下，运用操作算子的次数不得多于一次。



积木问题的宽度优先搜索树

宽度优先搜索的性质

- 当问题有解时，一定能找到解(完备性)
- 当问题为单位代价时，且问题有解时，一定能找到最优解（最优性）
- 方法与问题无关，具有通用性
- 效率较低
- 属于图搜索方法

宽度优先搜索的优点和缺点

- 宽度优先搜索是一种盲目搜索，时间和空间复杂度都比较高，当目标节点距离初始节点较远时会产生许多无用的节点，搜索效率低。
- 宽度优先搜索中，时间需求是一个很大的问题，特别是当搜索的深度比较大时，尤为严重，但是空间需求是比执行时间更严重的问题。

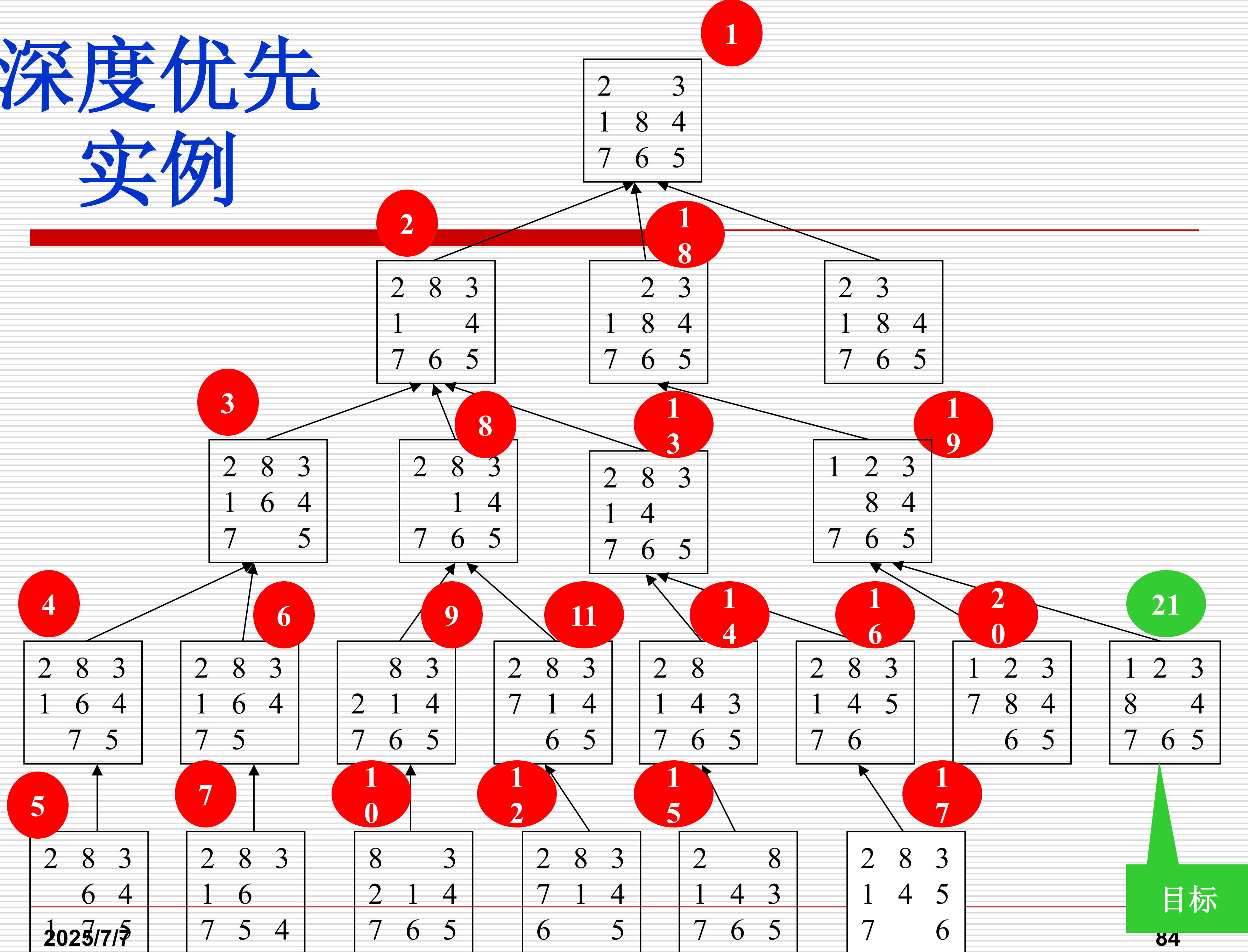
宽度优先搜索优点：

目标节点如果存在，用宽度优先搜索算法总可以找到该目标节点，而且是最小（即最短路径）的节点。

深度优先

- 深度优先——扩展当前节点后生成的子节点总是置于OPEN表的前端，即OPEN表作为栈，后进先出，使搜索优先向纵深方向发展。
- 过程：从初始节点S_0开始，按生成规则生成下一级各子节点，检查是否出现目标节点S_g；若未出现，则按“最晚生成的子节点优先扩展”的原则，再用生成规则生成再下一级的子节点，再检查是否出现S_g；若仍未出现，则再扩展最晚生成的子节点。如此下去，沿着最晚生成的子节点分枝，逐级“纵向”深入搜索。

深度优先实例



深度优先搜索

- 在深度优先搜索中，首先扩展最新产生的(最深的)节点，深度 相等的节点可以任意排列。
- “最晚产生的节点最先扩展”
  - 起始节点深度为0
  - 任何其他节点的深度等于其父辈节点深度加上1 :$d_n=d_{n-1}+1$

深度优先搜索

•很明显这样做，不一定找到最佳解，而且由于深度的限制，可能找不到解，然而，若不加深度限制，可能沿着一条路线无限制地扩展下去，这当然是不允许的。

•为保证找到解，应选择适当的深度界限，或者采取不断加大深度界限的办法，反复搜索，直到找到解。这种改进的方法叫做迭代加深搜索。

基于栈实现的深度优先搜索算法：

(1)把初始节点S_0放入OPEN表；

(2)如果OPEN表为空，则问题无解，失败并退出。

(3)把OPEN表中的第一个节点取出放入CLOSE表中，并按顺序冠以编号n；

(4)考察节点n是否为目标节点。若是，则求得了问题的解，成功并退出。

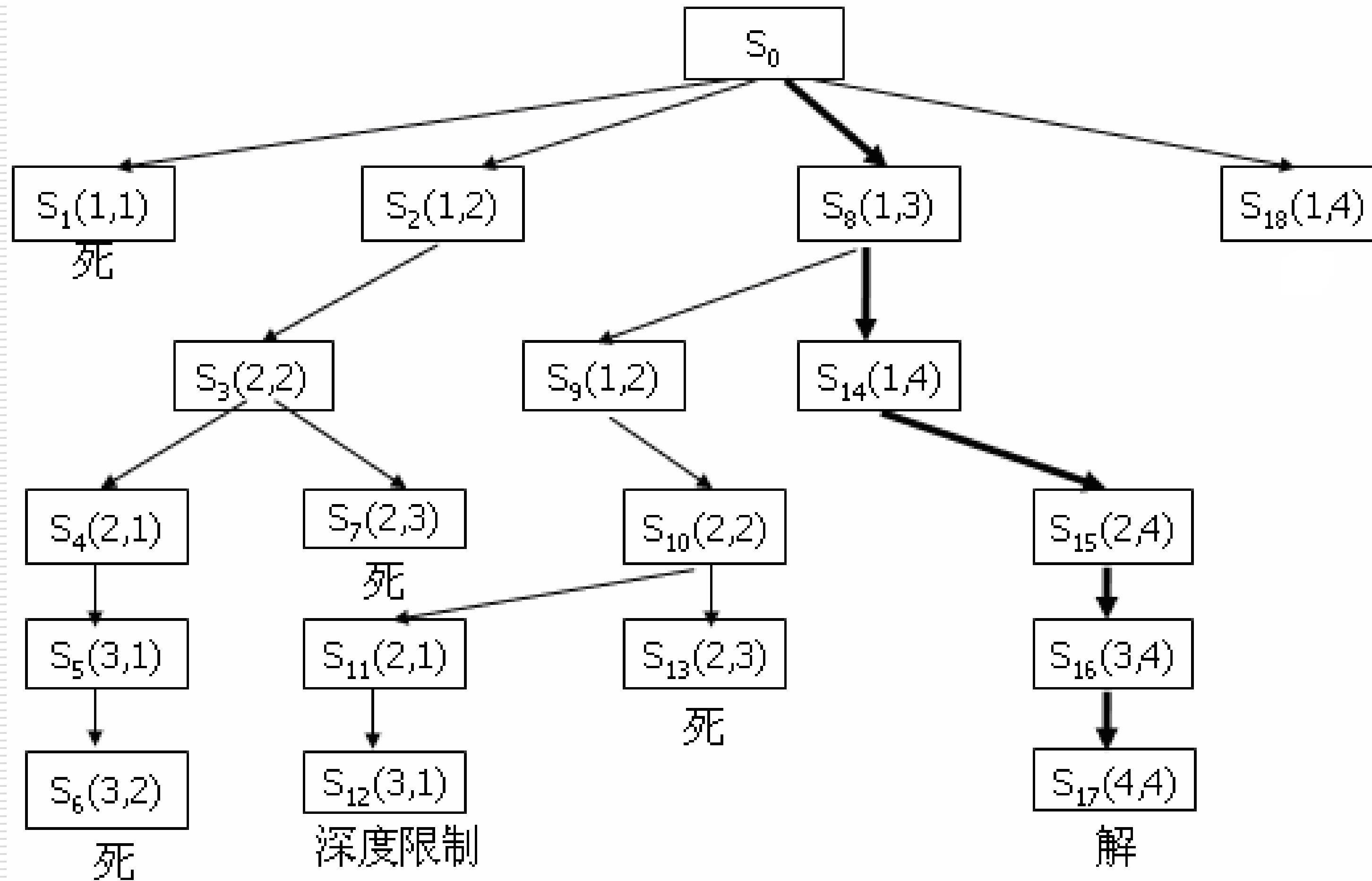
(5)若节点n不可扩展，则转第(2)步；

(6)扩展节点n，将其子节点放到OPEN表的首部，并为其配置指向父节点的指针，然后转第(2)步。

例 卒子穿阵问题: 要求一卒子从顶部通过图所示的列阵到达底部。要求卒子行进中不可进入到代表敌兵驻守的区域内（标注1），并不准后退。假定限制值为5。请画出按照深度优先搜索策略所产生的搜索树

行 \ 列	1	2	3	4
1	1	0	0	0
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0

行 \ 列	1	2	3	4
1	1	0	0	0
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0



卒子穿阵的深度优先搜索树

深度优先搜索的性质

- 一般不能保证找到最优解
- 当深度限制不合理时，可能找不到解，可以将算法改为可变深度限制
- 最坏情况时，搜索空间等同于穷举
- 是一个通用的与问题无关的方法

深度优先搜索的优点

- 深度优先搜索的优点是比宽度优先搜索算法需要较少的空间，该算法只需要保存搜索树的一部分，它由当前正在搜索的路径和该路径上还没有完全展开的节点标志所组成。
- 深度优先搜索的存储器要求是深度约束的线性函数。

深度优先搜索的缺点

既不是完备的，也不是最优的。

主要问题是可能搜索到了错误的路径上。很多问题可能具有很深甚至是无限的搜索树，如果不幸选择了一个错误的路径，则深度优先搜索会一直搜索下去，而不会回到正确的路径上。这样一来对于这些问题，深度优先搜索要么陷入无限的循环而不能给出一个答案，要么最后找到一个答案，但路径很长而且不是最优的答案。

比较

- 适用场合

  •深度优先——当一个问题有多个解答或多条解答路径，且只须找到其中一个时；往往应对搜索深度加以限制。

  •宽度优先——确保搜索到最短的解答路径。

- 共同优缺点：

  •可直接应用一般图搜索算法实现，不需要设计特别的节点排序方法，从而简单易行，适合于许多复杂度不高的问题求解任务。

  •节点排序的盲目性，由于不采用领域专门知识去指导排序，往往会在白白搜索了大量无关的状态节点后才碰到解答，所以也称为盲目搜索。

有界深度搜索和迭代加深搜索

有界深度优先搜索过程总体上按深度优先算法方法进行，但对搜索深度需要给出一个深度限制*dm*，当深度达到了*dm*的时候，如果还没有找到解答，就停止对该分支的搜索，换到另外一个分支进行搜索。

有界深度搜索算法

(1)把初始节点S_0放入OPEN表中，置S_0的深度$d(S_0)=0$。

(2)如果OPEN表为空，则问题无解，失败并退出。

(3)把OPEN表中的第一个节点取出放入CLOSE表中。并按顺序冠以编号n。

(4)考察节点n是否为目标节点。若是，则求得了问题的解，成功并退出。

(5)如果节点n的深度d(n)= dm，则转第(2)步。

(6)如果节点n不可扩展，则转第(2)步。

(7)扩展节点n。将其子节点放入OPEN表的首部，并为其配置指向父节点的指针。然后转第(2)步。

策略说明：

- （1）深度限制*dm*很重要。当问题有解，且解的路径长度小于或等于*dm*时，则搜索过程一定能够找到解，但是和深度优先搜索一样这并不能保证最先找到的是最优解。
- 但是当*dm*取得太小，解的路径长度大于*dm*时，则搜索过程中就找不到解，即这时搜索过程甚至是不完备的。

（2）深度限制*dm*不能太大。当*dm*太大时，搜索过程会产生过多的无用节点，既浪费了计算机资源，又降低了搜索效率。

（3）有界深度搜索的主要问题是深度限制值*dm*的选取。

改进方法：（迭代加深搜索）

先任意给定一个较小的数作为*dm*，然后按有界深度算法搜索，若在此深度限制内找到了解，则算法结束；如在此限制内没有找到问题的解，则增大深度限制*dm*，继续搜索。

- 迭代加深搜索，试图尝试所有可能的深度限制：
  - 首先深度为0，
  - 然后深度为1，
  - 然后为2，等等。
- 如果初始深度为0，则该算法只生成根节点，并检测它。
- 如果根节点不是目标，则深度加1，通过典型的深度优先算法，生成深度为1的树。
- 当深度限制为m时，树的深度为m。

- 迭代加深搜索看起来会很浪费，因为很多节点都可能扩展多次。
- 然而对于很多问题，这种多次的扩展负担实际上很小，直觉上可以想象，如果一棵树的分支系数很大，几乎所有的节点都在最底层上，则对于上面各层节点扩展多次对整个系统来说影响不是很大。

迭代加深搜索算法

```
Procedure Iterative-deepening
Begin
    (1)设置当前深度限制=1;
    (2)把初始节点压入栈,并设置栈顶指针;
    (3)While栈不空并且深度在给定的深度限制之内do
        Begin
        弹出栈顶元素;
        If栈顶元素=goal,返回并结束;
        Else以任意的顺序把栈顶元素的子女压入栈中;
        End
      End whild
        (4)深度限制加1,并返回2;
End.
```

总结

- 宽度优先搜索、深度优先搜索和迭代加深搜索都可以用于生成和测试算法。
- 宽度优先搜索需要指数数量的空间，深度优先搜索的空间复杂度和最大搜索深度呈线性关系。

- **迭代加深搜索**对一棵深度受控的树采用深度优先的搜索。它结合了宽度优先和深度优先搜索的优点。
- 和宽度优先搜索一样，它是最优的，也是完备的。但对空间要求和深度优先搜索一样是适中的。

搜索最优策略的比较

标准	宽度优先	深度优先	有界深度	迭代加深
时间	b^d	b^m	b^l	b^d
空间	b^d	bm	bl	bd
最优	是	否	否	是
完备	是	否	如果 l>d,是	是

- 表注：b是分支系数，d是解答的深度，m是搜索树的最大深度，l是深度限制。

单选题 1分

设置

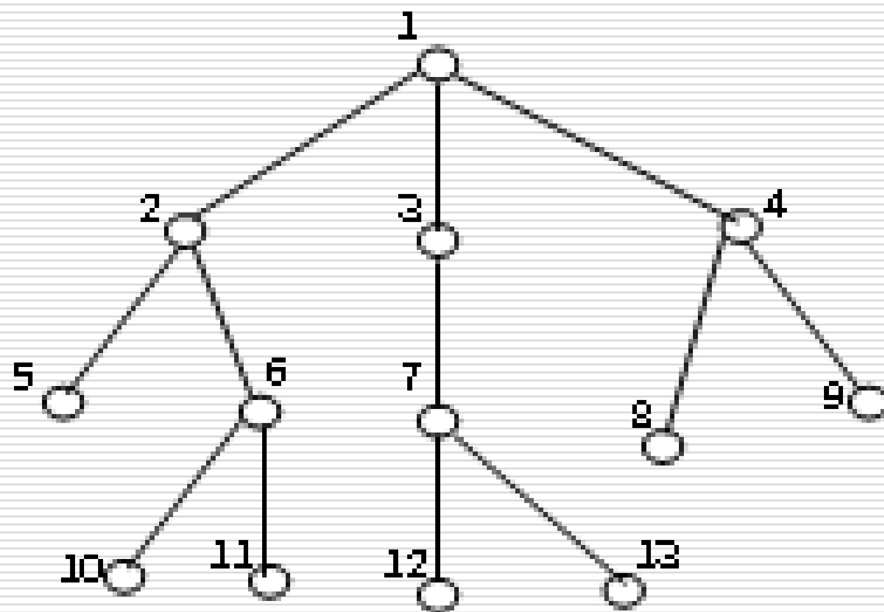
1、对于下列搜索图中，采用宽度优先搜索，其节点的扩展顺序**不可能**的是。

A 1-2-3-4-5-6

B 1-2-3-7-4-5

C 1-3-4-2-7-8

D 1-4-2-3-9-8



提交

单选题 1分 设置

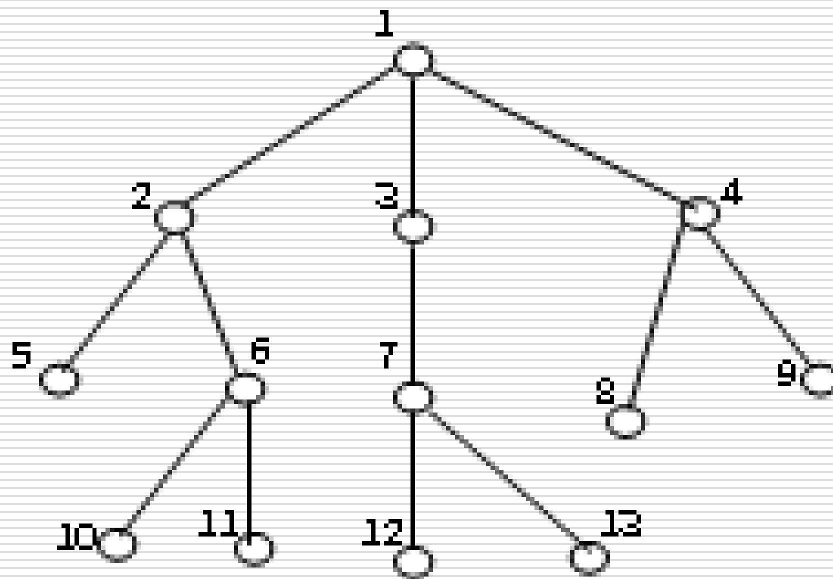
2、对于下列搜索图中，采用深度优先搜索，其节点的扩展顺序可能的是。

A 1-2-6-10-5-11

B 1-4-2-3-9-8

C 1-3-7-13-12-2

D 1-2-3-4-5-6



提交

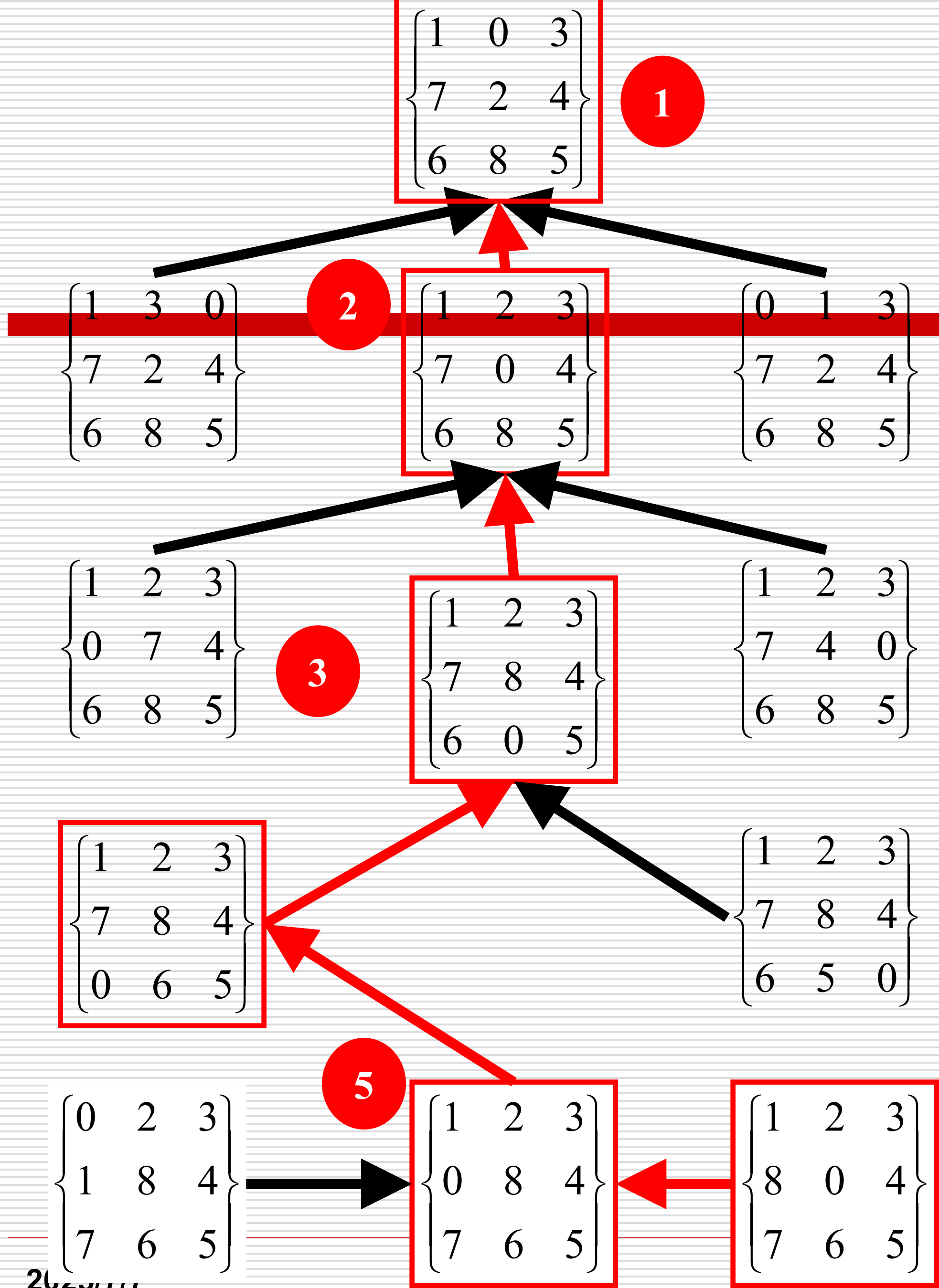
4.4 启发式搜索★

一般图搜索算法

- 常用的简单方式：
  - 深度优先
  - 宽度优先
  - 【缺点：节点排序的盲目性】
    - 在白白搜索了大量无关的状态节点后才碰到解答，效率低

盲目搜索

- 提高一般图搜索效率的关键★
  - 优化OPEN表中节点的排序方式



最理想情况：

每次排序后OPEN表

表首元素n

总在解答路径上

启发式搜索

- 启发式知识指导OPEN表排序的一般图搜索：
  - 全局排序——对OPEN表中的所有节点排序，使最有希望的节点排在表首。
    - A算法， A*算法（重点掌握！）
  - 局部排序——仅对新扩展出来的子节点排序，使这些新节点中最有希望者能优先取出考察和扩展；
    - 爬山法（了解，对深度优先法的改进）；

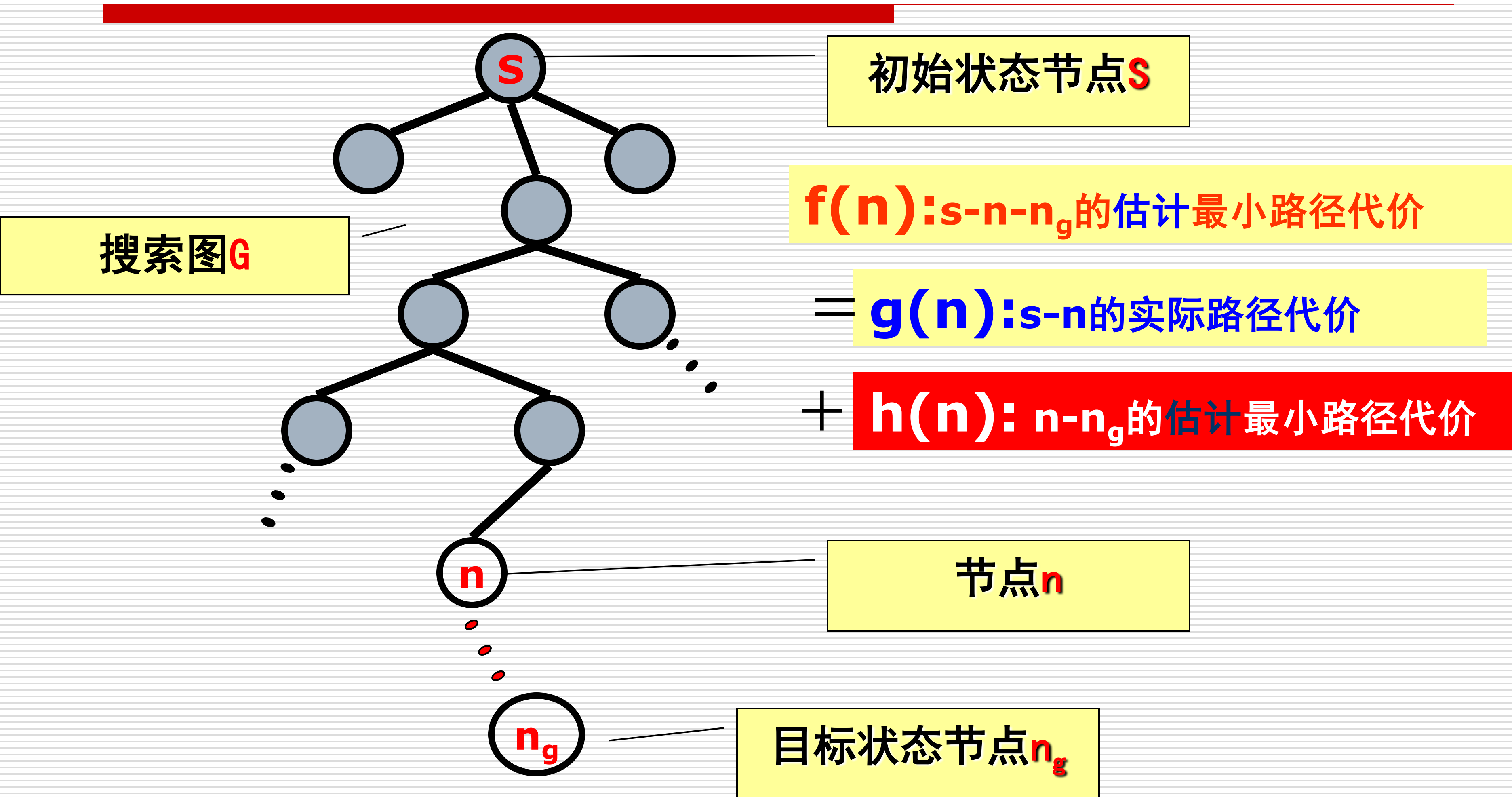
启发式搜索

——1.A算法（掌握）

- 【基本思想】
  - 设计体现启发式知识的评价函数f(n)；
  - 指导一般图搜索中OPEN表待扩展节点的排序；
- 【评价函数f(n)=g(n)+h(n) 】 ★
  - n-搜索图G中的节点；
  - f(n)- G中从初始状态节点s，经由节点n到达目标节点n_g，估计的最小路径代价；
  - g(n)- G中从s到n，目前实际的路径代价；
  - h(n)-从n到n_g，估计的最小路径代价；

启发式搜索

——1.A算法（掌握）



启发式搜索

——1.A算法（掌握）

- 【

价；

- 如

启发式搜索
——1.A算法（掌握）

- A算法的设计与一般图搜索相同，划分为二个阶段★：
  - 1、初始化
    - 建立只包含初始状态节点s的搜索图G:={s}
    - OPEN:={s}
    - CLOSE:={}
  - 2、搜索循环
    - MOVE-FIRST(OPEN)-取出OPEN表首的节点n
    - ⑥扩展出n的子节点,插入搜索图G和OPEN表
    - ⑦适当的标记和修改指针（子节点→父节点）
    - ⑧排序OPEN表（评价函数f(n)的值排序）
  - 通过循环地执行该算法，搜索图会因不断有新节点加入而逐步长大，直到搜索到目标节点。

启发式搜索
——1.A算法（掌握）

- 算法A的设计与一般图搜索类似，划分为二个阶段★：
  - 1、初始化
  - 2、搜索循环
    - MOVE-FIRST(OPEN)-取出OPEN表首的节点n
    - ⑥扩展出n的子节点,插入搜索图G和OPEN表
      - 对每个子节点ni,计算f(n,ni)=g(n,ni)+h(ni)
    - ⑦适当的标记和修改指针（子节点→父节点）
    - ⑧排序OPEN表（评价函数f(n)的值排序）

启发式搜索

——1.A算法（掌握）

- ⑥扩展出n的子节点,插入搜索图G和OPEN表
  - 对每个子节点ni,计算f(n,ni)=g(n,ni)+h(ni)
- ⑦适当的标记和修改指针（子节点→父节点）
  - (i)全新节点：f(ni)=f(n,ni)
  - (ii)已出现在OPEN表中的节点
  - (iii)已出现的CLOSE表中的节点
    - IF f(ni)>f(n,ni) THEN
    - 修改指针指向新父结点n
    - f(ni)=f(n,ni)
- ⑧排序OPEN表（f(n)值从小到大排序）

启发式搜索

——1.A算法（掌握）

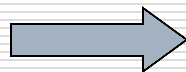
A算法实例——八数码游戏

- 1)设计评价函数f(n)
  - f(n)=d(n)+w(n),其中
  - d(n)-节点n在搜索图中的节点深度，对g(n)的度量；
  - w(n)-代表启发式函数h(n),其值是节点n与目标状态节点n_g相比较，不考虑空格，错位的棋牌个数；

1		3
7	2	4
6	8	5

初始布局

移动数码



1	2	3
8		4
7	6	5

目标布局

启发式搜索

——1.A算法（掌握）

启发式算法A实例——八数码游戏

- 1)设计评价函数f(n)
  - f(n)计算实例

1		3
7	2	4
6	8	5

初始布局s

1	2	3
8		4
7	6	5

目标布局n_g

f(n):s-n-n_g的最小路径代价

＝ g(n):s-n的最小路径代价

＋ h(n): n-ng的最小路径代价

f(s) ＝ d(s):当前节点深度

＋ w(s):错位的棋牌个数

注：W(S)不考虑空格

启发式搜索

——1.A算法（掌握）

启发式算法A实例——八数码游戏

- 1)设计评价函数f(n)
  - f(n)计算实例

1		3
7	2	4
6	8	5

初始布局s

1	2	3
8		4
7	6	5

目标布局n_g

f(n):s-n-n_g的最小路径代价

＝ g(n):s-n的最小路径代价

＋ h(n): n-ng的最小路径代价

f(s) ＝ d(s):当前节点深度

＋ w(s):错位的棋牌个数

＝ 0 ＋ 4

＝ 4

注：W(S)不考虑空格

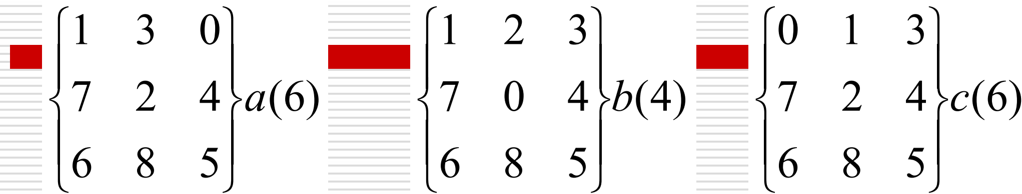
$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} s(4)$$

初始化

OPEN:={s4}

CLOSE:={}

$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} s(4)$$



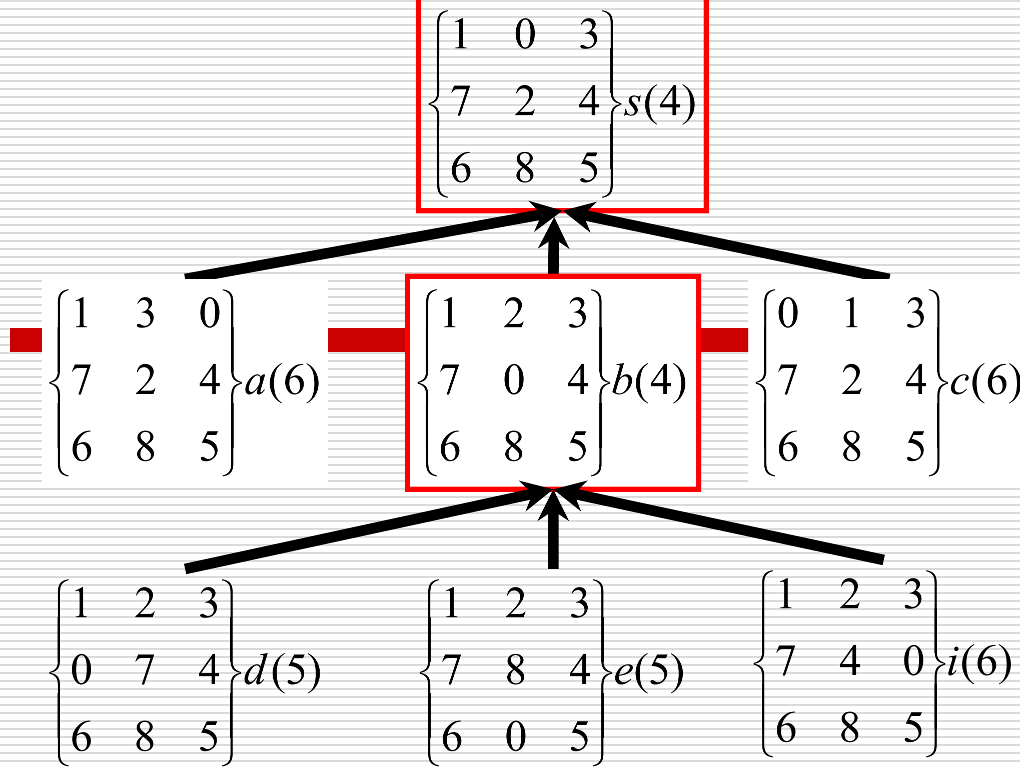
循环1

OPEN:={b4

a6

c6}

CLOSE:={s4}



循环2

OPEN:={d5

e5

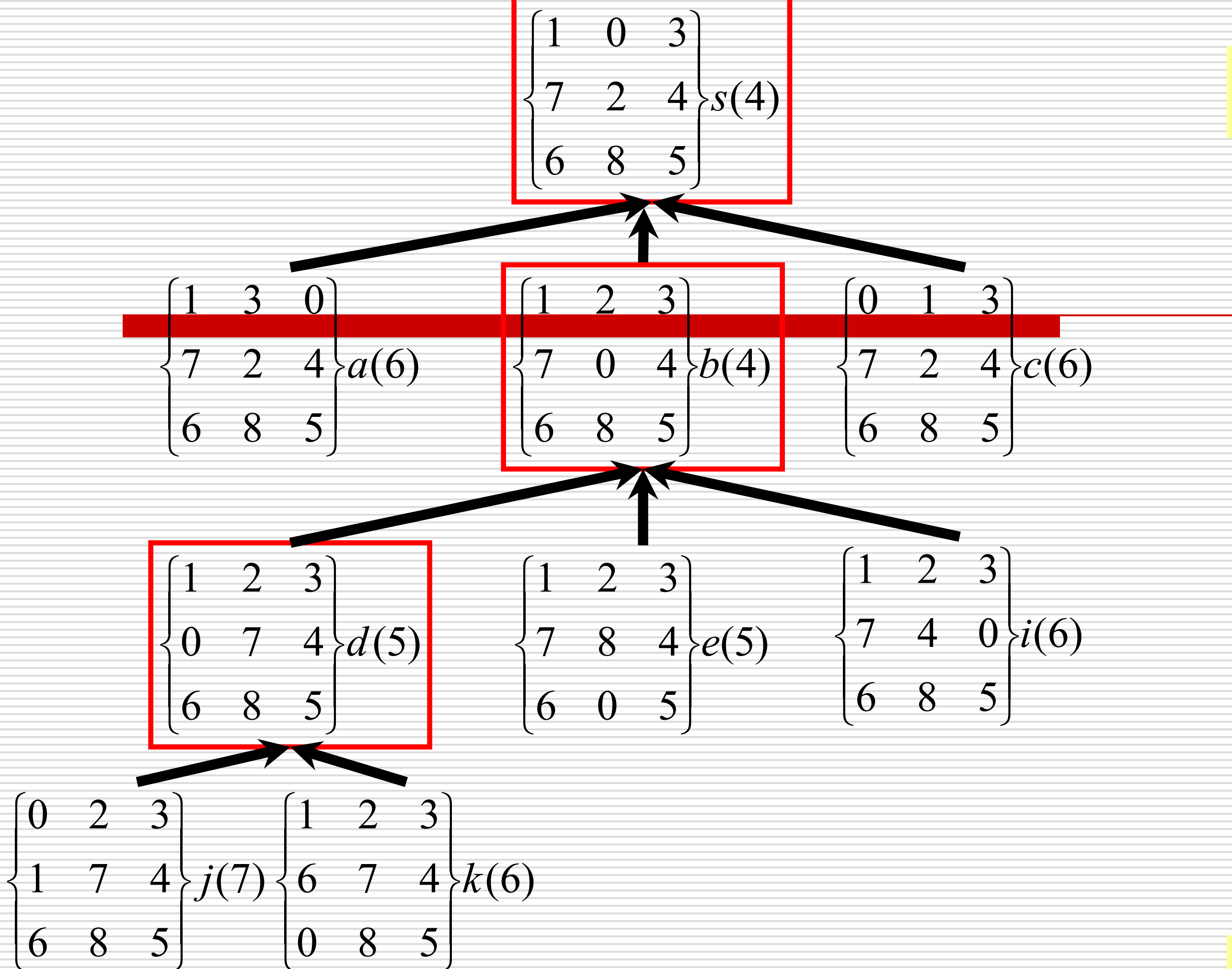
a6

c6

i6}

CLOSE:={s4

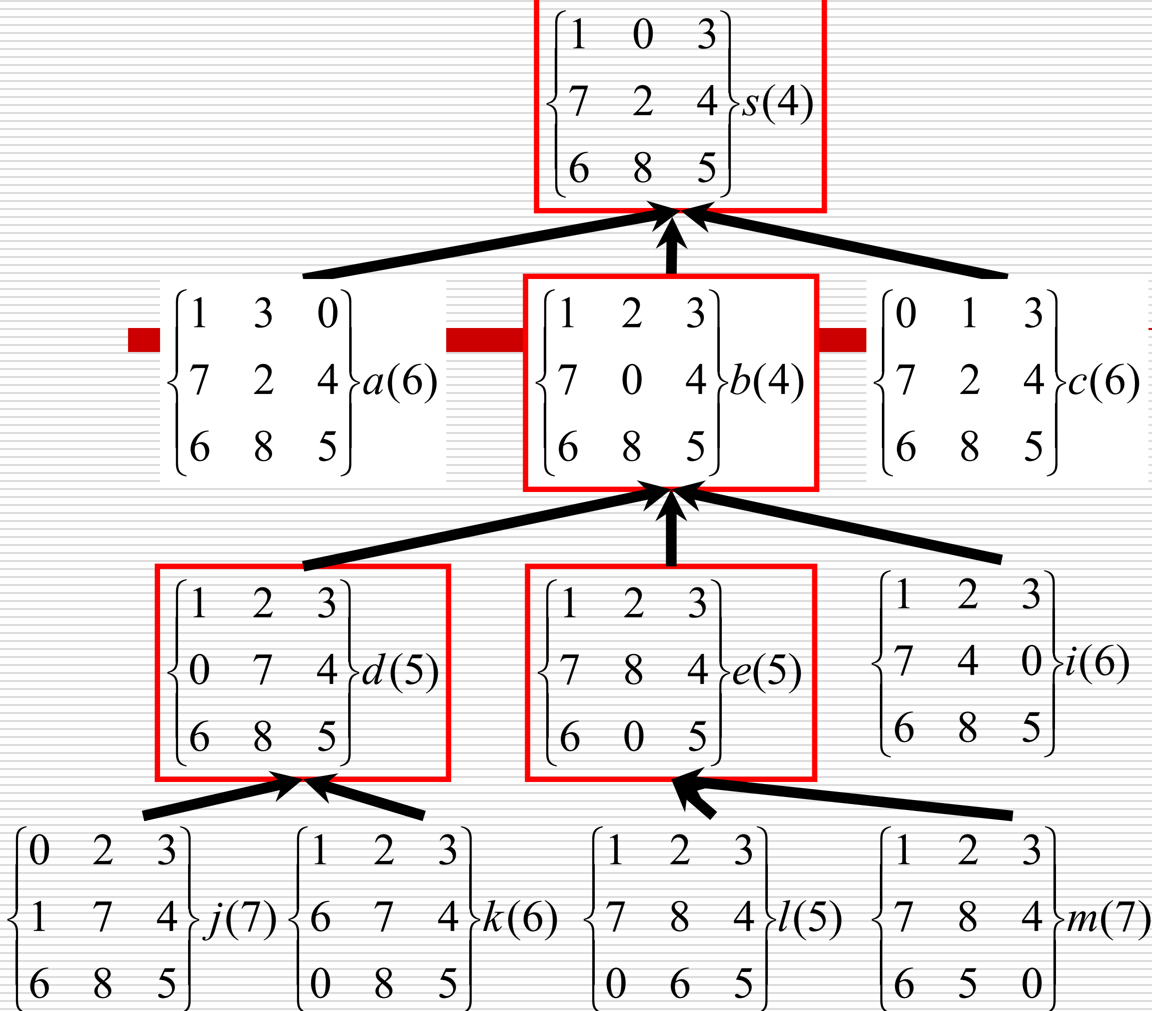
b4}



循环3

OPEN:={e5
a6
c6
i6
k6
j7}

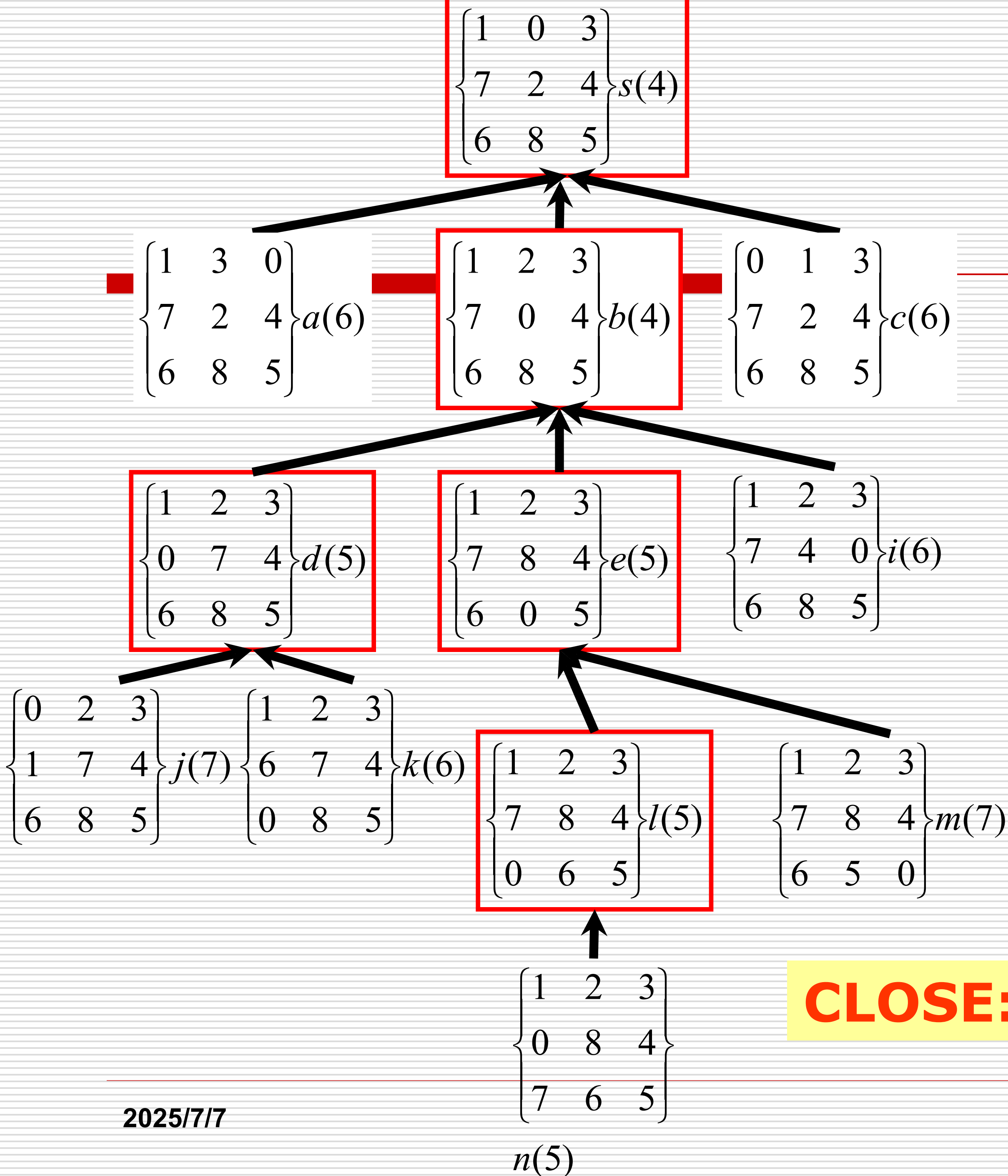
CLOSE:={s4
b4
d5}



循环4

OPEN:={l5
a6
c6
i6
k6
j7
m7}

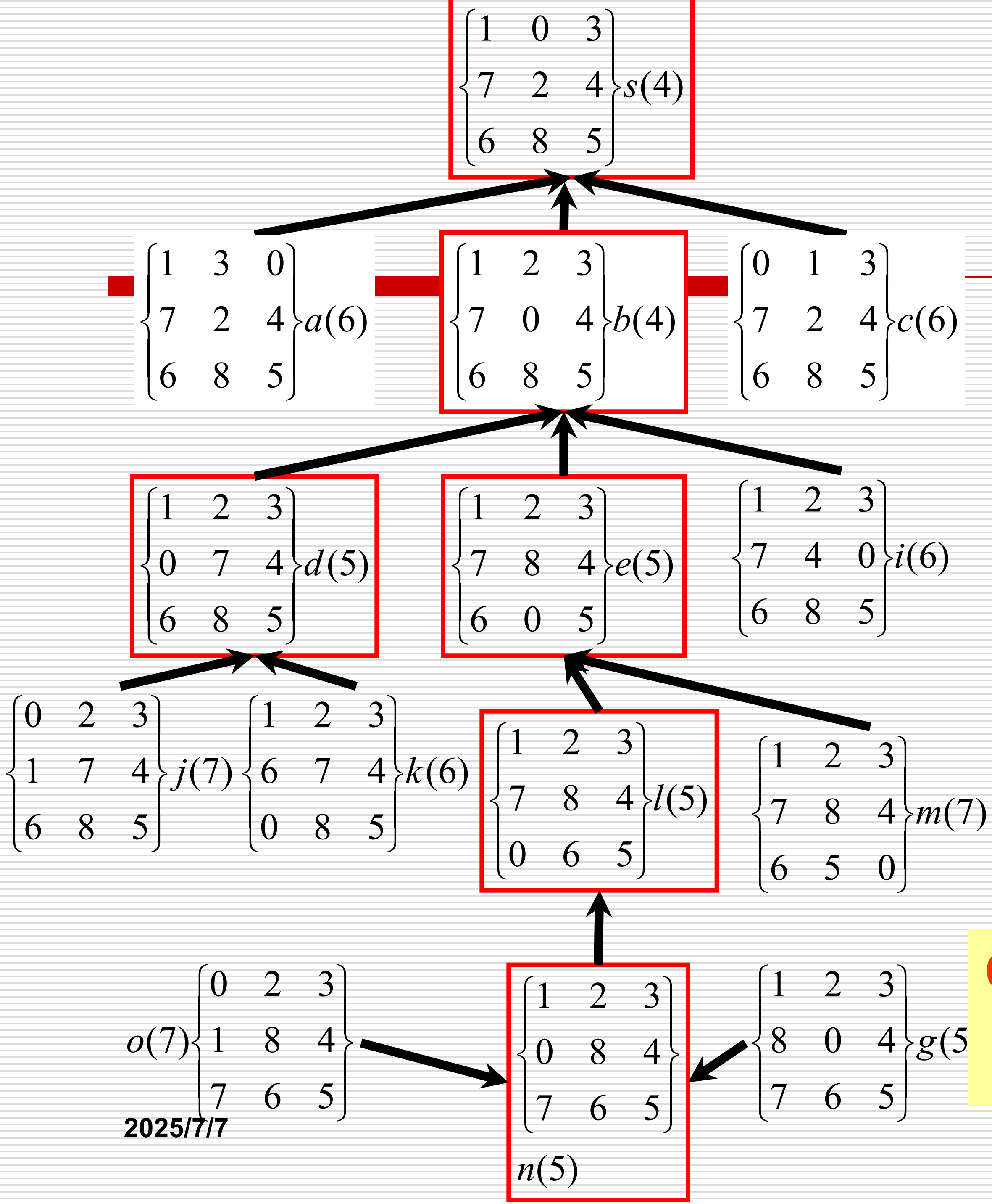
CLOSE:={s4,b4,d5,e5}



循环5

OPEN:={n5
a6
c6
i6
k6
j7
m7}

CLOSE:={s4,b4,d5,e5,l5}



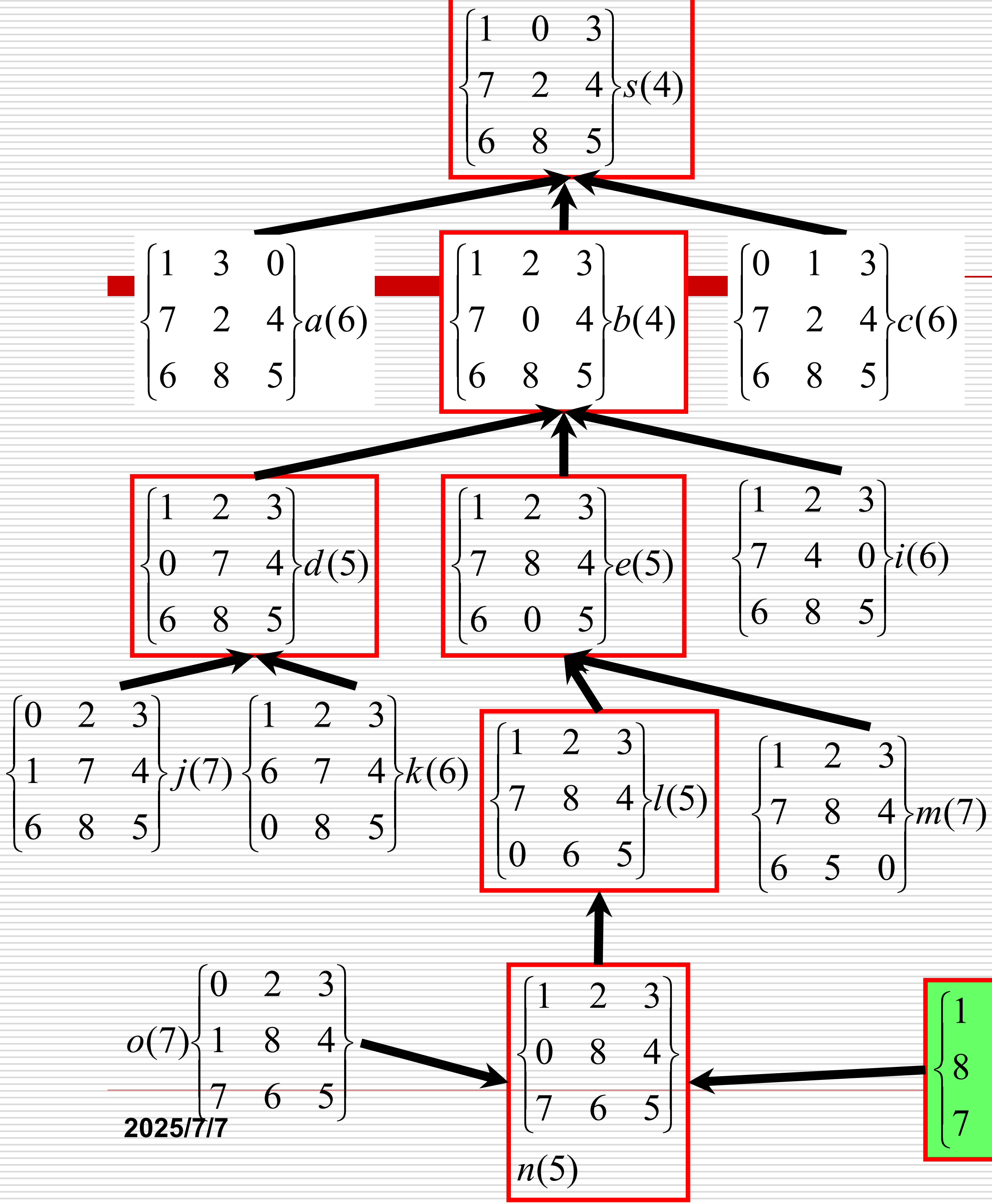
循环6

OPEN:={g5
a6
c6
i6
k6
j7
m7
o7}

CLOSE:={s4,b4,d5,
e5,l5,n5}

循环7

成功结束



最理想搜索图G

$$\begin{Bmatrix}1&0&3\\7&2&4\\6&8&5\end{Bmatrix}$$

$$\begin{Bmatrix}1&3&0\\7&2&4\\6&8&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\7&0&4\\6&8&5\end{Bmatrix}\quad\begin{Bmatrix}0&1&3\\7&2&4\\6&8&5\end{Bmatrix}$$

$$\begin{Bmatrix}1&2&3\\0&7&4\\6&8&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\7&8&4\\6&0&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\7&4&0\\6&8&5\end{Bmatrix}$$

$$\begin{Bmatrix}1&2&3\\7&8&4\\0&6&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\7&8&4\\6&5&0\end{Bmatrix}$$

$$\begin{Bmatrix}0&2&3\\1&8&4\\7&6&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\0&8&4\\7&6&5\end{Bmatrix}\quad\begin{Bmatrix}1&2&3\\8&0&4\\7&6&5\end{Bmatrix}$$

$$s(4)=\begin{Bmatrix}1&0&3\\7&2&4\\6&8&5\end{Bmatrix}$$

$$a(6)=\begin{Bmatrix}1&3&0\\7&2&4\\6&8&5\end{Bmatrix}\quad b(4)=\begin{Bmatrix}1&2&3\\7&0&4\\6&8&5\end{Bmatrix}\quad c(6)=\begin{Bmatrix}0&1&3\\7&2&4\\6&8&5\end{Bmatrix}$$

$$d(5)=\begin{Bmatrix}1&2&3\\0&7&4\\6&8&5\end{Bmatrix}\quad e(5)=\begin{Bmatrix}1&2&3\\7&8&4\\6&0&5\end{Bmatrix}\quad i(6)=\begin{Bmatrix}1&2&3\\7&4&0\\6&8&5\end{Bmatrix}$$

$$j(7)=\begin{Bmatrix}0&2&3\\1&7&4\\6&8&5\end{Bmatrix}\quad k(6)=\begin{Bmatrix}1&2&3\\6&7&4\\0&8&5\end{Bmatrix}\quad l(5)=\begin{Bmatrix}1&2&3\\7&8&4\\0&6&5\end{Bmatrix}\quad m(7)=\begin{Bmatrix}1&2&3\\7&8&4\\6&5&0\end{Bmatrix}$$

判断失误

$$o(7)\begin{Bmatrix}0&2&3\\1&8&4\\7&6&5\end{Bmatrix}\quad n(5)=\begin{Bmatrix}1&2&3\\0&8&4\\7&6&5\end{Bmatrix}\quad g(5)=\begin{Bmatrix}1&2&3\\8&0&4\\7&6&5\end{Bmatrix}$$

例 2 给定4L和3L的水壶各一个，水壶上没有刻度，可以向水壶中加水。如何在4L的壶中准确地得到2L水？

例 2 给定4L和3L的水壶各一个，水壶上没有刻度，可以向水壶中加水。

如何在4L的壶中准确地得到2L水？

（x, y）——4L壶里的水有xL，3L壶里的水有yL，

n表示搜索空间中的任一节点。

则给出下面的启发式函数：

- $h(n)=2$ 如果$0<x<4$并且$0<y<3$

  $=4$ 如果$0<x<4$或者$0<y<3$

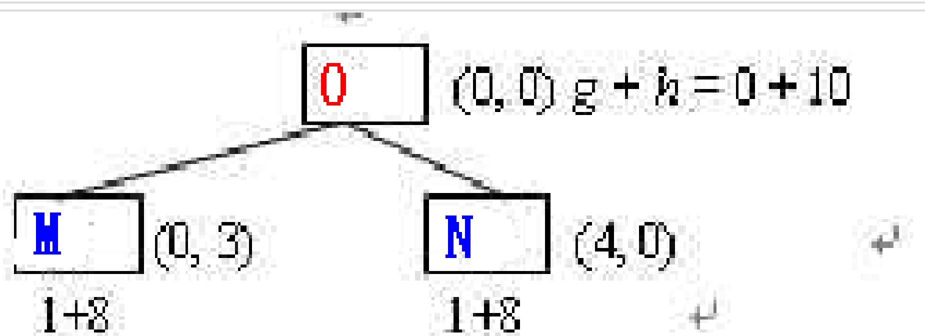
  $=8$ 如果 $x=0$并且 $y=3$

  或者 $x=4$ 并且 $y=0$

  $=10$ 如果 $x=0$ 并且 $y=0$

  或者 $x=4$并且 $y=3$

- 假定$g(n)$表示搜索树中搜索的深度，则根据图搜索策略得下图的搜索空间。



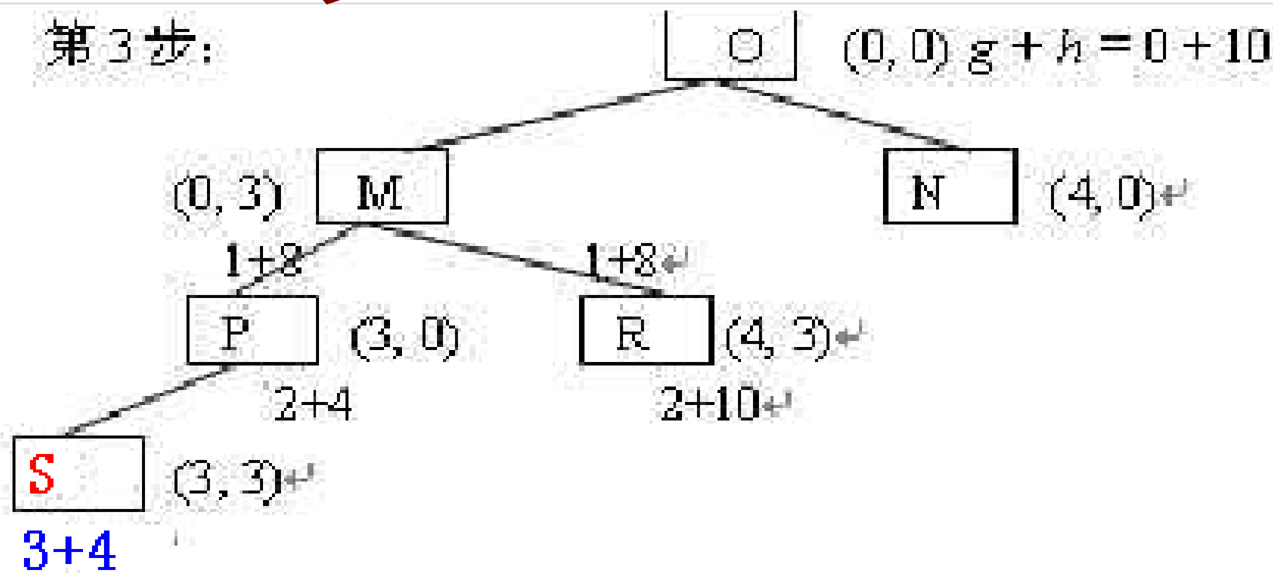
- 水壶问题的状态空间扩展图

在第0步，由节点0可以得到g + h =10。

在第1步，得到两个节点M和N，其估价函数值都为1+8=9，因此可以任选一个节点扩展。

假定选择了节点M，在第2步扩展M得到两个后继了点P和R，对于P有2+4=6，对于R有2+10=12。现在，在节点P、R、N中，节点P具有最小的估价函数值，所以选择节点P扩展。

在第3步，可以得到节点S，其中3+4=7。现在，在节点S、R、N中，节点S的估价函数值最小，所以下一步就会选择S节点扩展。该过程一直进行下去，直到到达目标节点。



- 水壶问题的状态空间扩展图

启发式搜索

——2.实现启发式搜索的关键因素（理解）

- 实现启发式搜索应考虑的关键因素★：
  - （1）搜索算法的可采纳性(Admissibility);
  - （2）启发式函数h(n)的强弱及其影响；

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)-1968 ★

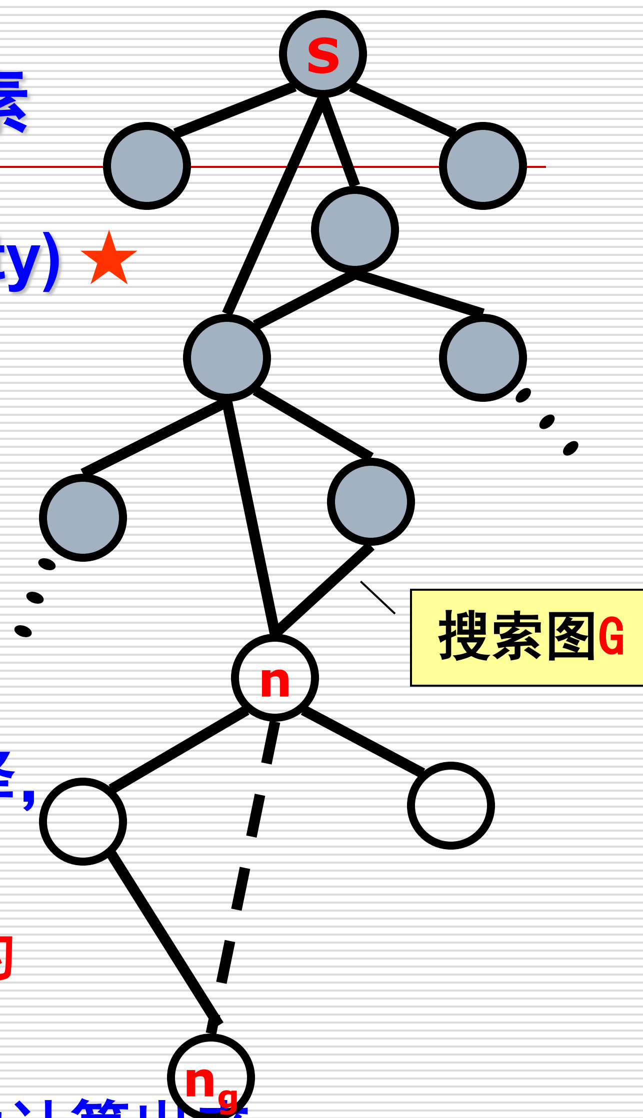
- 1)定义
  - 在存在从初始状态节点到目标状态节点解答路径的情况下，若一个搜索法总能找到最短（代价最小）的解答路径，则称该状态空间中的搜索算法具有可采纳性，也叫最优性。
  - 如，宽度优先的搜索算法是可采纳的，只是搜索效率不高。
- 2) A算法的可采纳性——定义$f^*(n)=g^*(n)+h^*(n)$
  - n-搜索图G中最短解答路径的节点；
  - $f^*(n)$- s经节点n到n_g的实际最短解答路径的路径代价；
  - $g^*(n)$-该路径前段（从s到n）的路径代价；
  - $h^*(n)$-该路径后段（从n到n_g）的路径代价；

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility) ★

- 3) 评价函数f与f^*的比较
  - f(n)、g(n)、h(n)分别是 $f^*(n)$、$g^*(n)$、$h^*(n)$的近似值（估计值）
  - 理想情况下：
    - 若g(n)=g*(n)、h(n)=h*(n)，
    - 则搜索过程中，每次都正确选择，
    - 不扩展任何无关的节点。
  - 实际情况：设计接近f*的f是很困难的
    - 在算法执行过程中，
    - g(n)容易从已经生成的搜索树中计算出来



启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility) ★

- 3) 评价函数f与f*的比较
  - 理想情况下：
    - 若g(n)=g*(n)、h(n)=h*(n)，不扩展无关的节点
  - 实际情况：
    - 设计接近f*的f是很困难的
      - 在算法执行过程中，
      - g(n)容易从已经生成的搜索树中计算出来，比如就以节点深度d(n)当做g(n)
      - h(n)尽可能靠近h*(n) ——A算法的关键。

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)

- 4)改进启发式函数——八数码游戏
  - f(n)=d(n)+w(n),其中
  - w(n)-表示错位的棋牌个数，不够贴切，错误的扩展了节点d；
  - p(n)-节点n与目标状态节点比较，错位棋牌在不受阻拦的情况下，移动到目标状态相应位置所需走步（移动次数）的总和；
  - p(n)比w(n)更接近于h*(n)-p(n)不仅考虑了棋牌的错位因素，还考虑了错位的距离（移动距离）

启发式搜索

——2.实现启发式搜索的关键因素

4)改进启发式函数——八数码游戏

- f(s)计算实例

初始布局s

1		3
7	2	4
6	8	5

(错位移动距离标注: 1, 1, 1, 2)

目标布局n_g

1	2	3
8		4
7	6	5

f(s) = d(s):当前节点深度 + w(s):错位的棋牌个数 = 0 + 4 = 4

f(s) = d(s):当前节点深度 + p(s):错位棋牌移动距离 = 0 + 5 = 5

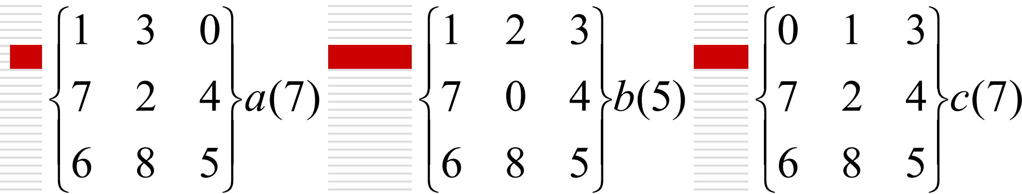
$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} s(5)$$

初始化

OPEN:={s5}

CLOSE:={}

$$\begin{Bmatrix} 1 & 0 & 3 \\ 7 & 2 & 4 \\ 6 & 8 & 5 \end{Bmatrix} s(5)$$



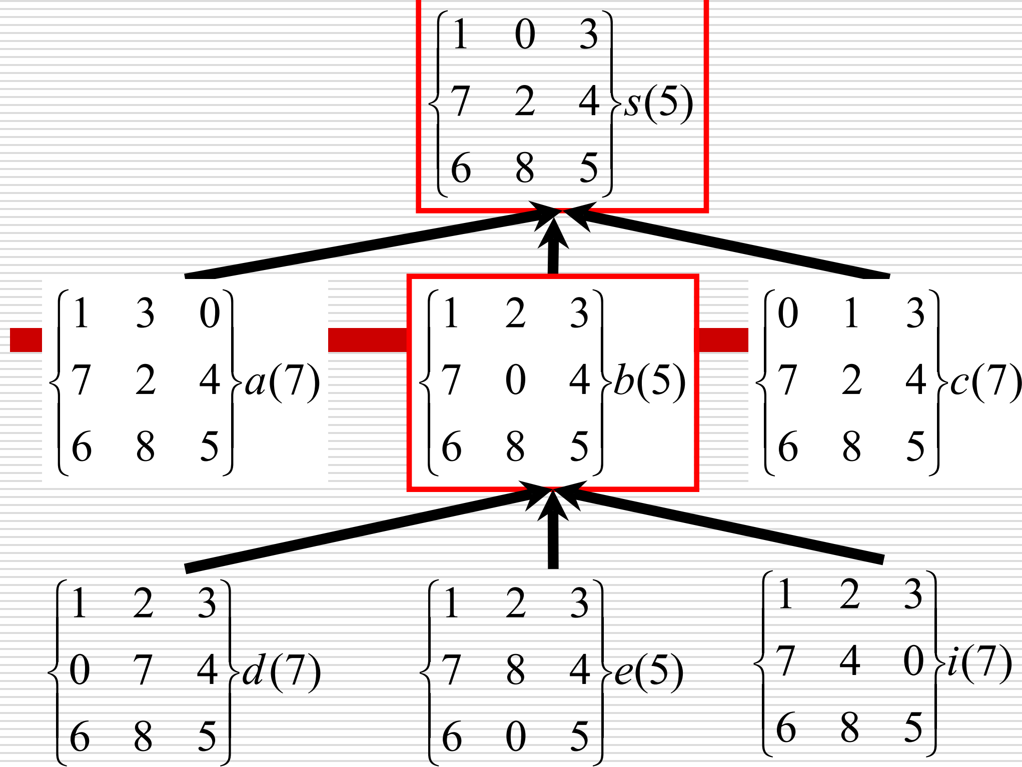
循环1

OPEN:={b5

a7

c7}

CLOSE:={s5}



循环2

OPEN:={e5

a7

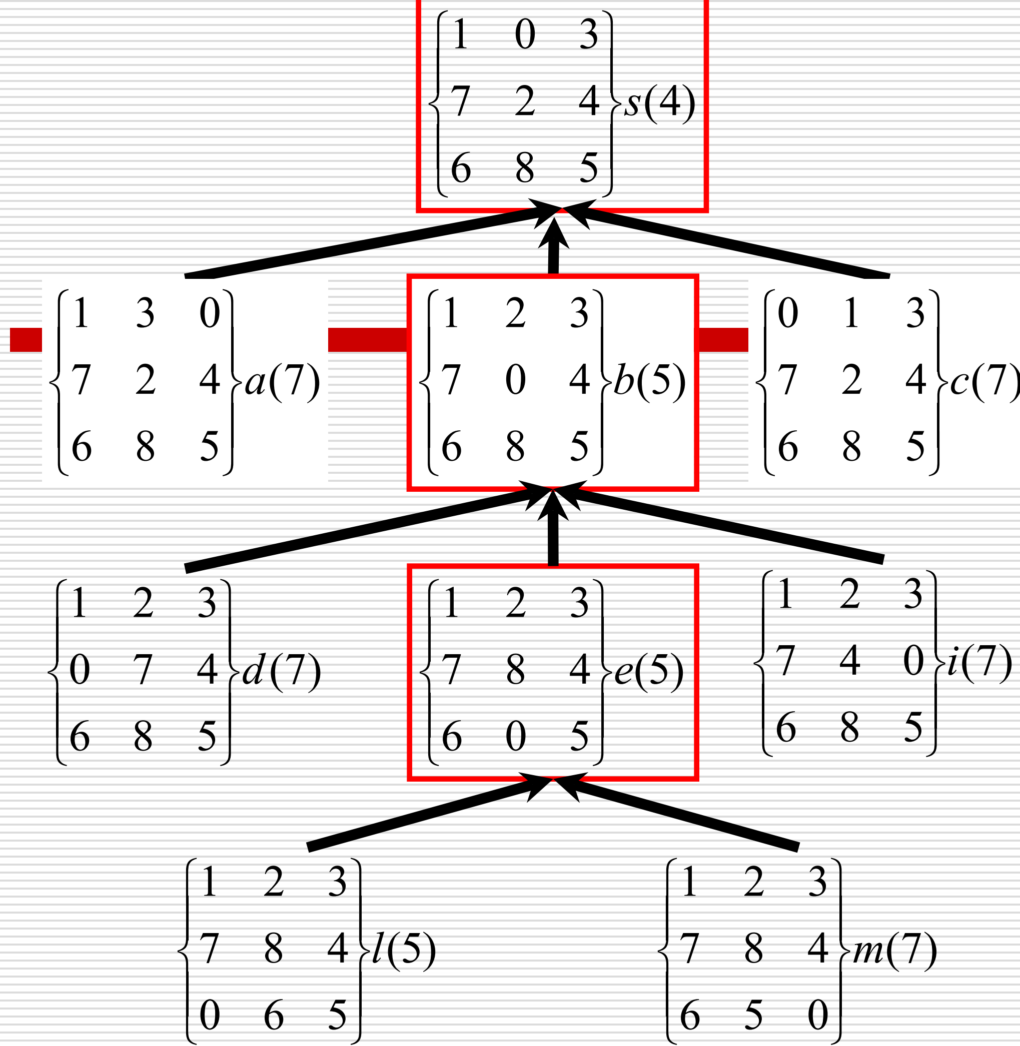
c7

d7

i7}

CLOSE:={s5

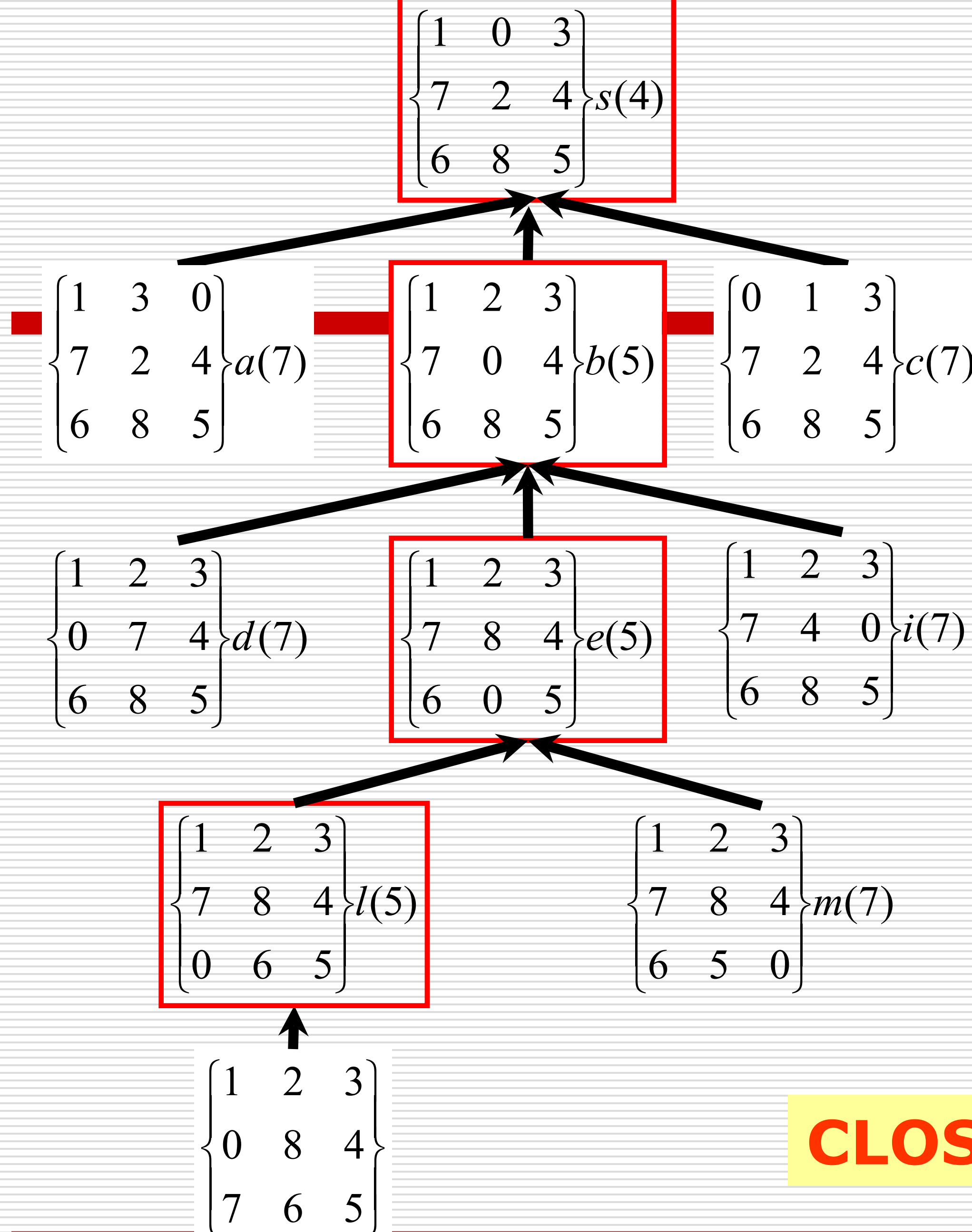
b5}



循环3

OPEN:={l5
a7
c7
d7
i7
m7}

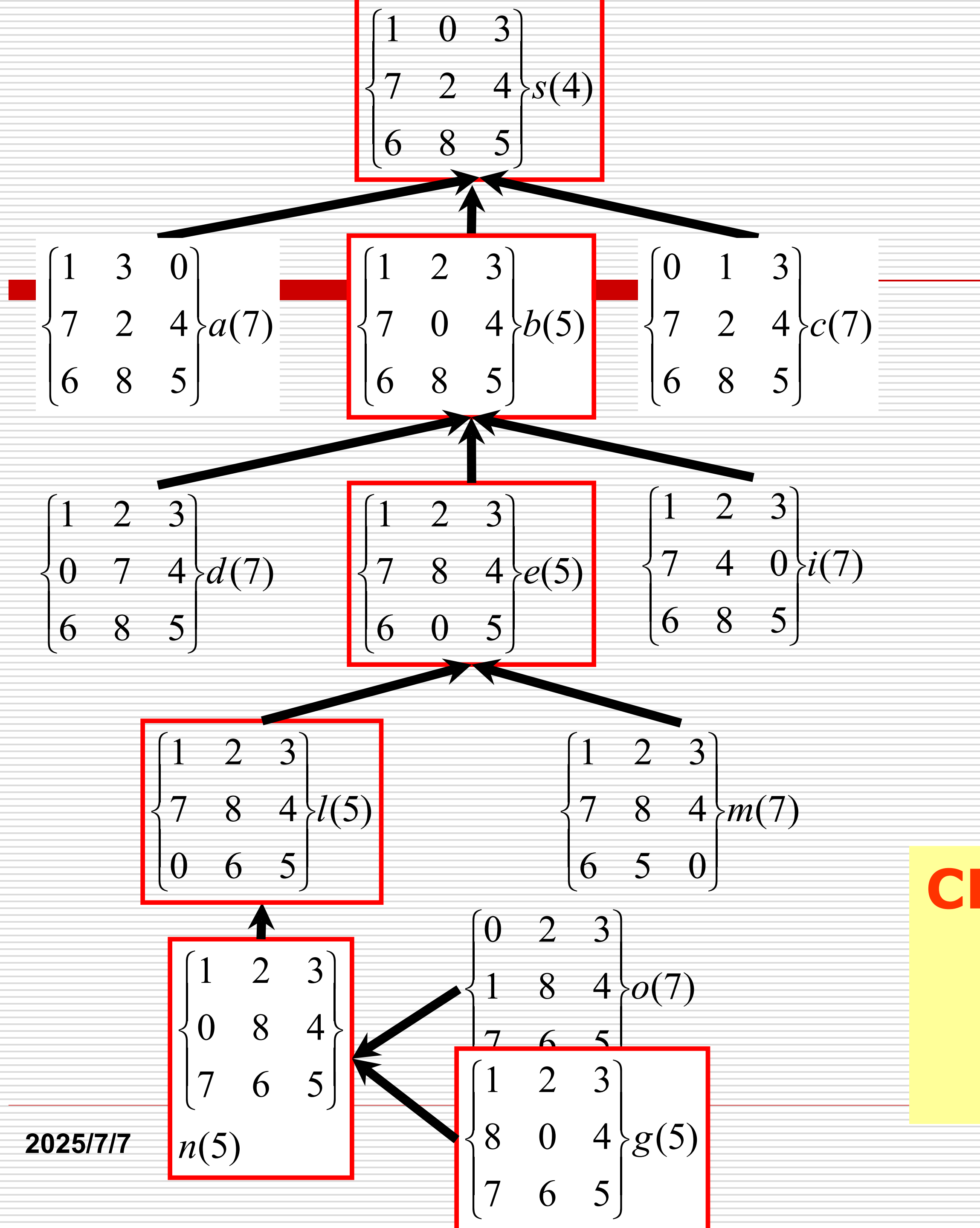
CLOSE:={s5
b5
e5}



循环4

OPEN:={n5
a7
c7
d7
i7
m7}

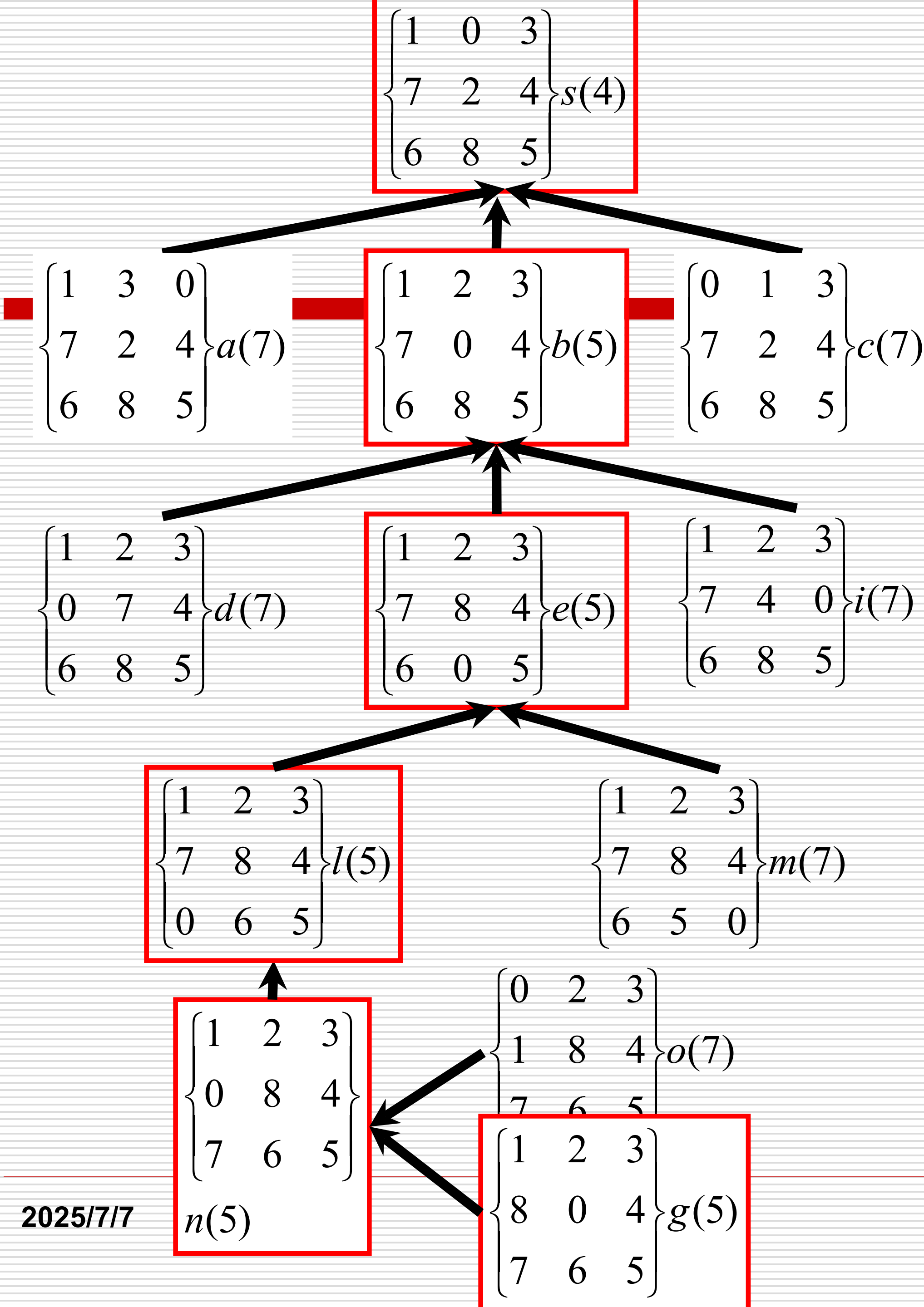
CLOSE:={s5,b5,e5,l5}



循环5

OPEN:={g5
a7
c7
d7
i7
m7
o7}

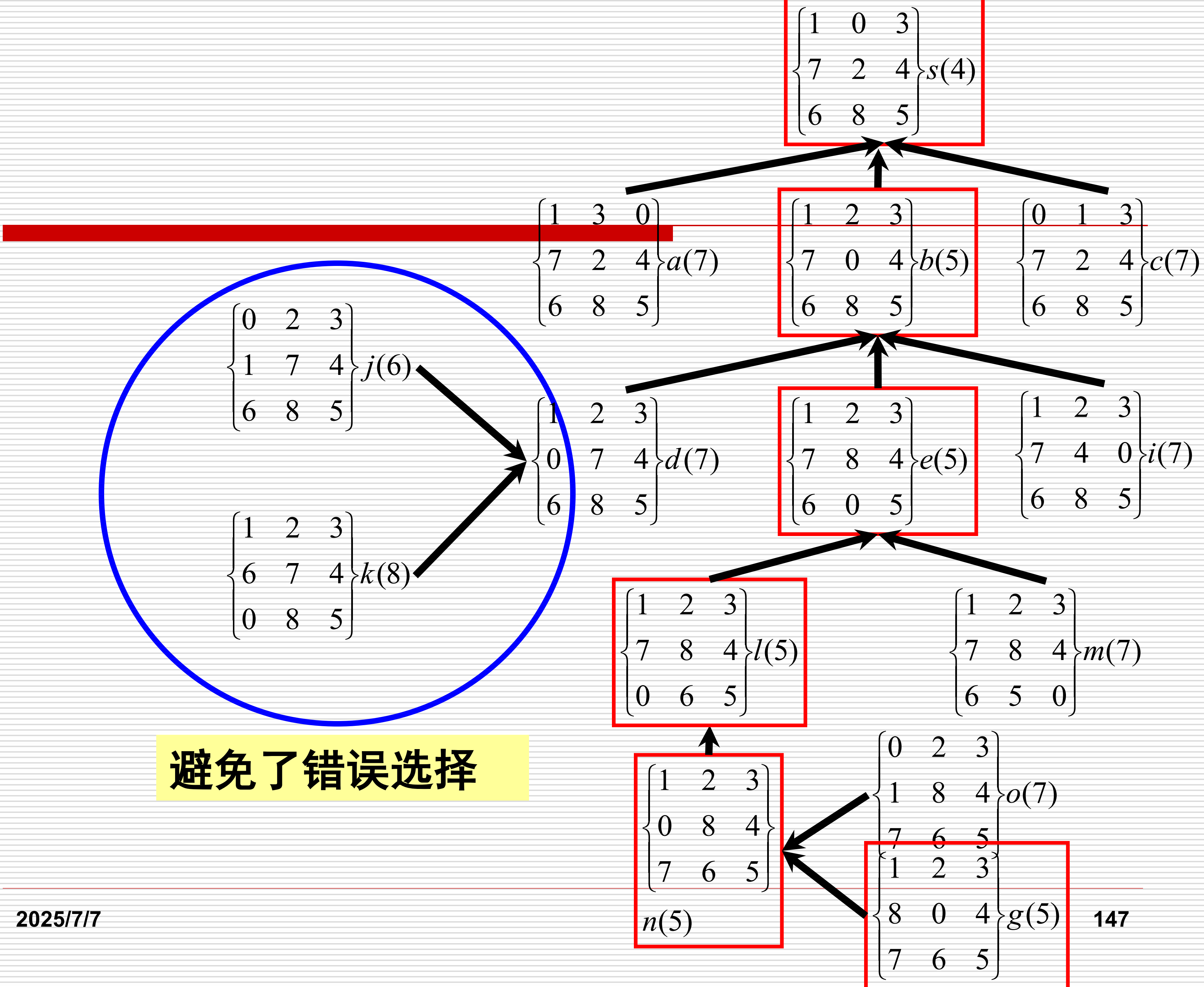
CLOSE:={s5,b5,
e5,l5,
n5}



循环6

成功结束

最理想搜索图G


s(4)
a(7)
b(5)
c(7)
d(7)
e(5)
i(7)
j(6)
k(8)
l(5)
m(7)
n(5)
o(7)
g(5)
避免了错误选择

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)

- 5) A*算法定义★：
  - 1、在A算法中，规定h(n)≤h*(n);
  - 2、经如此限制以后的A算法就是A*算法。
- A*算法是可采纳的，即总能搜索到最短解答路径
- 【回顾：八数码游戏的h(n)】
  - w(n)-错位的棋牌个数
  - p(n)-错位棋牌在不受阻拦的情况下，移动到目标状态相应位置所需走步（移动次数）的总和；
  - 上述两者均是可采纳的。(想想为什么)

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)

- 5) A*算法定义：
  - 1、在A算法中，规定$h(n) \le h^*(n)$;
  - 2、经如此限制以后的A算法就是A*算法。
- A*算法是可采纳的，即总能搜索到最短解答路径
- 证明：【《人工智能 上册》陆汝钤P_{248}】
  - 1）如果存在一条从初始状态到目标状态的解答路径，则一定存在一条最短解答通路；
  - 2）设状态n′是最短解答路径上的一个状态，那么经过有限步后，n′必然会成为OPEN表的第一个节点；
  - 3）因为最短解答路径只有有限个节点n′，所以有限步后算法必然因到达目标状态n_g。这就是最优解。
  - 证明完毕。

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)

- 5)满足可采纳性条件的算法——A*算法
  - 证明：
    - 2）设状态n′是最短解答路径上的一个状态，那么经过有限步后，n′必然会成为OPEN表的第一个节点；
    - $f(n')=g(n')+h(n')$
    - ∵根据假设，n′在最短解答路径上
    - ∴ 经过有限步骤后，$g(n')= g^*(n')$
    - ∴$f(n')=g^*(n')+h(n')$
    - ∵ $h(n)\le h^*(n)$
    - ∴$f(n')=g^*(n')+h(n') \le g^*(n')+h^*(n')=f^*(n')$
    - ∵ $f^*(n')= f^*(ng)$
    - ∴ $f(n') \le f^*(ng)$

启发式搜索

——2.实现启发式搜索的关键因素

（1）搜索算法的可采纳性(Admissibility)

- 5)满足可采纳性条件的算法——A*算法
  - 证明：
    - 2）设状态n′是最短解答路径上的一个状态，那么经过有限步后，n′必然会成为OPEN表的第一个节点；
    - 设OPEN表中n′之前的节点只有有限个，设为N个，其中估计值最小者为a1，并称之为第一代节点；由第一代节点生成的节点称为第二代节点，其中估计值最小者为a2；
    - a2≥a1+e(其中，e>0，表示每扩展一次起码的代价)
    - 扩展j代后， aj ≥a1+(j-1)e
    - 当j足够大时一定有aj> f*(ng)
    - ∵f(n′) ≤f*(ng)且OPEN表中n′之前的节点经过j次扩展后的最小估计值aj> f*(ng) ≥f(n′)
    - ∴经过有限步后，n′必然会成为OPEN表的第一个节点

启发式搜索

——2.实现启发式搜索的关键因素

（2）启发式函数的强弱及其影响★

- h(n)接近h*(n)的程度——衡量启发式函数的强弱
  - h(n)<h*(n)且差距较大时，OPEN表中节点排序的误差较大，h(n)过弱，产生较大的搜索图；
  - h(n)>h*(n)，h(n)过强，A算法失去可采纳性，不能确保找到最短解答路径；
  - h(n)=h*(n)是最理想的， OPEN表中节点排序没有误差，可以确保产生最小的搜索图，搜索到最短解答路径；
    - 无法设计
  - A*算法搜索问题解答的关键
    - h(n)在满足h(n) ≤ h*(n)的条件下，越大越好！

启发式搜索

——2.实现启发式搜索的关键因素

（2）启发式函数的强弱及其影响★

- 定理：解决同一问题的两个A*算法A1和A2，
  - 若$h1(n) \le h2(n) \le h^*(n)$且$g1(n)=g2(n)$
  - 则$t(A1) \ge t(A2)$
  - 其中，h1、h2分别是算法A1、A2的启发式函数，t指示相应算法到达目标状态时**搜索图含的节点总数**。
  - 【证明：《人工智能 上册》陆汝钤 P250）】
- 八数码游戏：$w(n) \le p(n) \le h^*(n)$
- p(n)扩展出的节点总数$\le t(w(n))$

启发式搜索

——2.实现启发式搜索的关键因素

（2）设计h(n)的实用考虑★

- 和宽度优先和深度优先的关系
  - 若h(n)≡0, 则意味着先进入OPEN表的节点会优先被考察和扩展，因为即使不以d(n)作为g(n), 通常先进入OPEN表的节点n也具有较小的g(n)值
  - 若g(n)≡0,则导致后进入OPEN表的节点会优先被考察和扩展，因为后进入OPEN表的节点n往往更接近于目标状态，即h(n)值较小，从而使搜索过程接近于深度优先的搜索策略

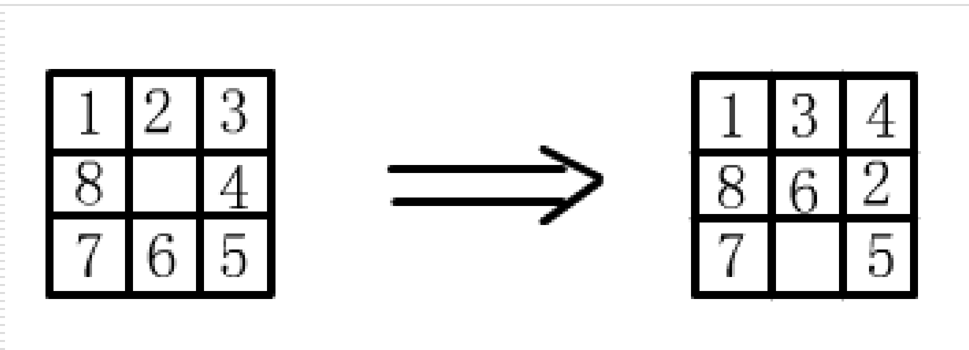
启发式搜索

——2.实现启发式搜索的关键因素

（2）设计h(n)的实用考虑

- 为更有效地搜索解答，可使用评价函数$f(n) = g(n) + wh(n)$，w用作加权
  - 在搜索图的浅层（上部），可让w取较大值，以使g(n)所占比例很小，从而突出启发式函数的作用，加速向纵深方向搜索
  - 一旦搜索到较深的层次，又让w取较小值，以使g(n)所占比例很大，并确保$wh(n) \le h^*(n)$,从而引导搜索向横广方向发展，寻找到较短的解答路径。

应用启发式搜索算法A解决以下八数码问题：



设评价函数f(n) = d(n) + p(n)，画出搜索图，并给出各搜索循环结束时Open和Close表的内容。

正常使用主观题需2.0以上版本雨课堂

作答

迭代加深A*算法

- 由于A*算法把所有生成的节点保存在内存中，所以A*算法在耗尽计算时间之前一般早已经把空间耗尽了。
- 目前开发了一些新的算法，它们的目的是为了克服空间问题。
- 但一般不满足最优性或完备性，如迭代加深A*算法IDA*、简化内存受限A*算法SMA*等。
- 下面简单介绍IDA*算法。

- 迭代加深搜索算法，它以深度优先的方式在有限制的深度内搜索目标节点。
- 在每个深度上，该算法在每个深度上检查目标节点是否出现，如果出现则停止，否则深度加1继续搜索。
- 而A*算法是选择具有最小估价函数值的节点扩展。

- 迭代加深A* 搜索算法IDA*是上述两种算法的结合。
- 这里启发式函数用做深度的限制，而不是选择扩展节点的排序。

迭代加深A*算法

```
Procedure IDA*算法
  Begin
  (1) 初始化当前的深度限制c=1
  (2) 把初始节点压入栈; 并假定 c' = ∞
  (3) While 栈不空桥do
    Begin
      弹出栈顶元素n
      If n=goal, Then 结束, 返回n以及从初始节点到n的路径
      Else do
      Begin
        For n 的每个子节点 n'
          If f(n') ≤ c , Then 把 n' 压入栈
            Else c' = min(c', f(n'))
          End for
      End
    End While
  (4) If 栈为空并且 c' = ∞, Then 停止并退出
  (5) If 栈为空并且 c' ≠ ∞ , Then c = c' , 并返回2
End
```

特点

- IDA*算法和A*算法相比，主要优点是对于内存的需求。A*算法需要指数级数量的存储空间，因为没有深度方面的限制。而IDA*算法只有当节点n的所有子节点 n' 的 $f(n')$ 小于限制值c时才扩展它,这样就可以节省大量的内存。
- 另一问题是当启发式函数是最优的时候，IDA*算法和A*算法扩展相同的节点，并且可以找到最优路径。

启发式搜索

- 启发式知识指导OPEN表排序的一般图搜索：
  - 全局排序——对OPEN表中的所有节点排序，使最有希望的节点排在表首。
    - A算法， A*算法（掌握！）
  - 局部排序——仅对新扩展出来的子节点排序，使这些新节点中最有希望者能优先取出考察和扩展；
    - 爬山法（了解，对深度优先法的改进）；

4.5 回溯策略和爬山法

- **简单的搜索策略：**
- $g(n) \equiv 0$， $f(n)= h(n)$，
- **局部排序——只排序新扩展出来的子节点，即局部排序**
  - **简单易行，适用于不要求最优解答的问题求解任务。**

1）爬山法——实现启发式搜索的最简单方法。

- **类似于人爬山——只要好爬，总是选取最陡处，以求快速登顶。**
- **求函数极大值问题——非数值解法，依赖于启发式知识，试探性地逐步向顶峰逼近**
- **适用于能逐步求精的问题。**
- **爬山法特点：**
- **只能向上，不准后退，从而简化了搜索算法；体现在：**

＊ 从当前状态节点扩展出的子节点（相当于找到上爬的路径）中，将h(n)最小的子节点（对应于到顶峰最近的上爬路径）作为下一次考察和扩展的节点，其余子节点全部丢弃。

- **不需设置OPEN和CLOSE表，因为没有必要保存任何待扩展节点；**
- **爬山法对于单一极值问题（登单一山峰）十分有效而又简便，对于具有多极值的问题无能为力——会错登上次高峰而失败：不能到达最高峰。**

2）回溯策略

- 可以有效地克服爬山法面临的困难——保存了每次扩展出的子节点，并按h(n)值从小到大排列。
- 相当于爬山的过程中记住了途经的岔路口——路径搜索失败时回溯（后退），向另一路径方向搜索

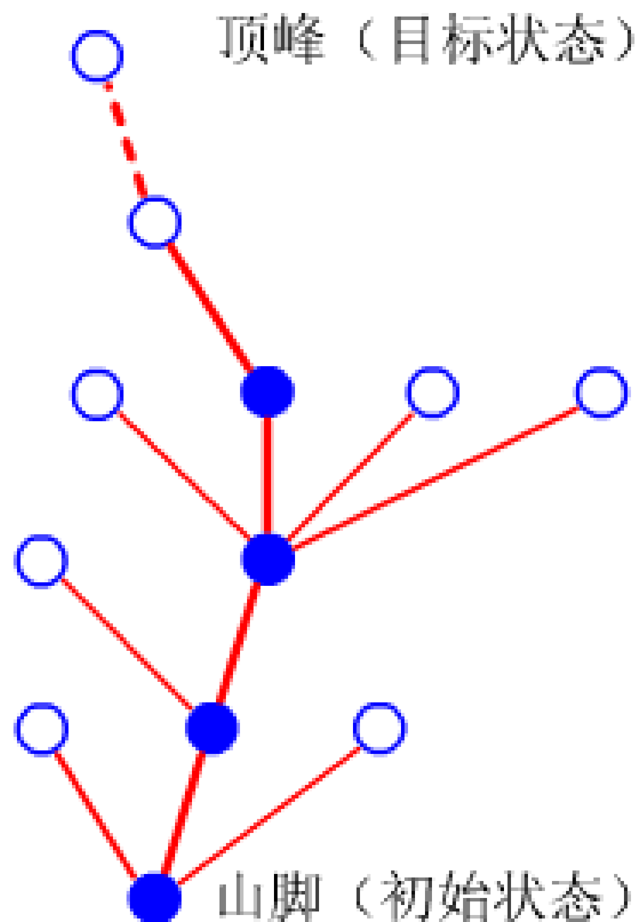


图2.11 应用回溯策略的爬山

2）回溯策略

- 递归过程——实现回溯策略的有效方式
- 算法就取名为BACKTRACK(n)，参数n为当前被扩展的节点，
- 初次调用时n即为初始状态节点s；
- 分二个部分：

* 判断当前节点n的状态，
* 作搜索工作——扩展节点n，递归调用该算法，处理返回结果。

令PATH、SNL、n、n’为局部变量：

- PATH--节点列表，指示解答路径；
- SNL--当前节点扩展出的子节点列表；
- MOVE-FIRST(SNL)--把SNL表首的节点移出，作为下一次要加以扩展的节点；
- n、n‘--分别指示当前考察和下一次考察的节点。

该递归过程的算法就取名为BACKTRACK(n),参数n为当前被扩展的节点，算法的初次调用式是BACKTRACK(s),s即为初始状态节点。算法的步骤如下：

（1） 若n是目标状态节点，则算法的本次调用成功结束，返回空表；

（2） 若n是失败状态，则算法的本次调用失败结束，返回’FAIL’；

（3） 扩展节点n，将生成的子节点置于列表SNL，并按评价函数f(k) = h(k)的值从小到大排序（k指示子节点）；

（4） 若SNL为空，则算法的本次调用失败结束，返回’FAIL’；

（5） n’= MOVE-FIRST(SNL)；

（6） PATH = BACKTRACK(n’)；

（7） 若PATH =’FAIL’, 返回到语句（4）；

（8） 将n’加到PATH表首，算法的本次调用成功结束，返回PATH。

2）回溯策略

- 三种失败状态：
  - 不合法状态（如传教士和野人问题中所述的那样）
  - 旧状态重现（如八数码游戏中某一棋盘布局的重现，会导致搜索算法死循环），
  - 状态节点深度超过预定限度（例如八数码游戏中，指示解答路径不超过6步）。
- 回溯条件
  - 失败状态，由算法第（2）句指示；
  - 搜索进入“死胡同”，由该算法的第(4)句定义。

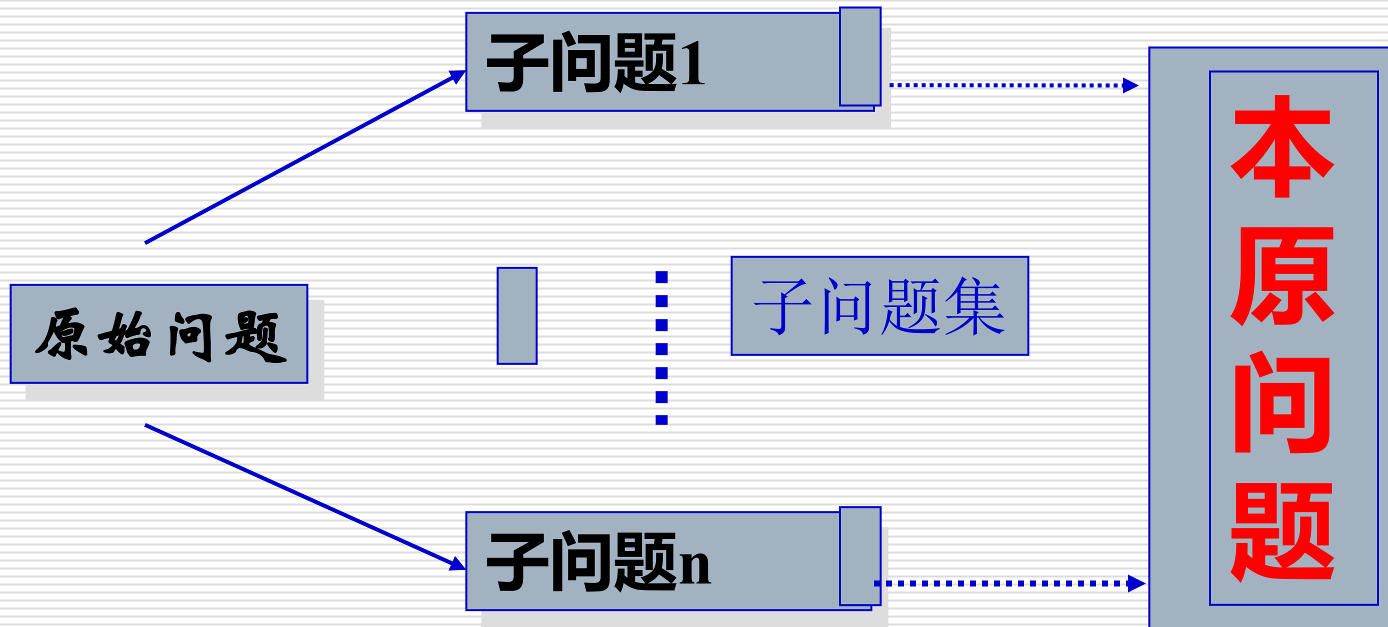
2）回溯策略

- 解答路径的生成——从相应于目标状态节点的空表开始，递归返回PATH。
- 影响回溯算法效率的关键因素——回溯次数。
  - 回溯——搜索到失败状态时的一种弥补行为，
  - 准确选择下一步搜索考察的节点——大幅度减少甚至避免回溯。
  - 设计好的启发式函数h(n)是至关重要的。

4.6 问题归约

- 问题归约是人求解问题常用的策略：
  - 把复杂的问题变换为若干需要同时处理的较为简单的子问题后再加以分别求解
  - 只有子问题全部解决时，问题才算解决；
  - 问题的解答由子问题的解答联合构成。

问题归约法
（Problem Reduction Representation）



- 问题归约可以用三元组表示：（S_0，O，P），其中
  - S_0是初始问题，即要求解的问题；
  - P是本原问题集，其中的每一个问题是不用证明的，自然成立的，如公理、已知事实等，或已证明过的问题；
  - O是操作算子集，它是一组变换规则，通过一个操作算子把一个问题化成若干个子问题。

- 问题归约表示方法就是由初始问题出发，运用操作算子产生一些子问题，对子问题再运用操作算子产生子问题的子问题，这样一直进行到产生的问题均为本原问题，则问题得解。

- 看如下符号积分问题：

  初始问题——$\int f(x)\,dx$

  变换规则——积分规则

  本原问题——可直接求原函数和积分，如 $\int \sin(x)\,dx$，$\int \cos(x)\,dx$等。

所有问题归约的最终目的是产生本原问题。

$$\int (\sin^3 x + x^4/(x^2 + 1))\, dx$$

$$= \int \sin^3 x dx + \int (x^4/(x^2 + 1))dx$$

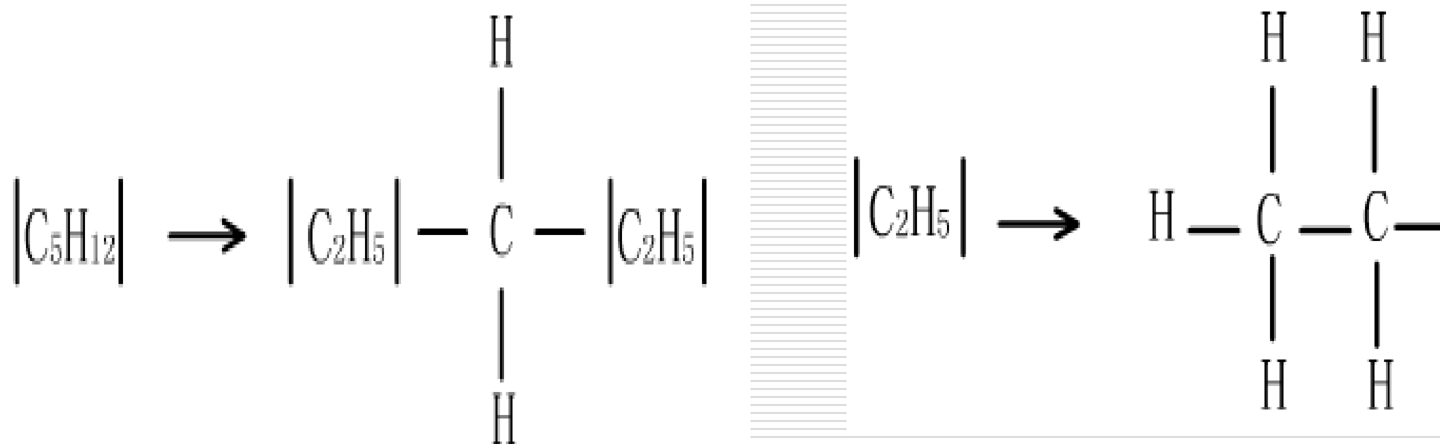
$$= \int -(1 - \cos^2 x)d\cos x + \int (x^2 - 1 + 1/(1 + x^2))dx$$

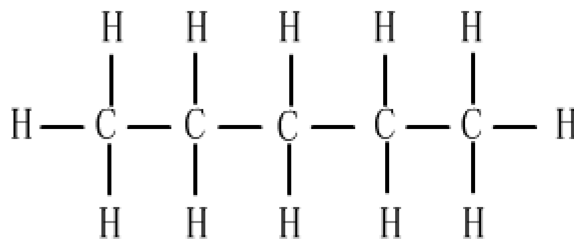
$$= (\int -d\cos x + \cos^2 x d\cos x) + (\int x^2 dx - \int dx + \int (1/(1 + x^2))dx)$$

$$= -\cos x + \cos^3 x/3 + x^3/3 - x + \mathrm{arctg} x$$

分子结构识别问题

如何区分分子式相同但分子结构不同的有机化合物成为重要而又困难的问题。著名的专家系统 DENDRAL能用于有效地识别分子结构，该系统建立了一套重写规则去把分子式重写为原子数较少的分子式和原子间结合关系的混合结构





□ 问题归约的实质：★

从目标(要解决的问题)出发逆向推理，建立子问题以及子问题的子问题，直至最后把初始问题归约为一个平凡的本原问题集合。

4.7 与/或图搜索

- 应用问题归约策略得到的状态空间图，也称为“与或图”
- 逻辑“与”关系——用圆弧将几条节点间关联弧连接在一起
  - 表示问题分解为子问题；
  - 子问题的状态联合起来构成问题状态。
  - 子问题全部解决才会导致问题的解决；
- 逻辑“或”关系：
  - ①问题可以有多种分解方式；
  - ②问题（子问题）可能激活多个状态变迁操作；
  - 只要一种分解方式或状态变迁操作能导致最终的解答成功即可；
  - 导致多个可能的解答。

与或图

- 用AND-OR图把问题归约为子问题替换集合。
- 如，假设问题A既可通过问题$C1$与$C2$，也可通过问题$C3$、$C4$和$C5$，或者由单独求解问题$C6$来解决，如下图所示。图中各节点表示要求解的问题或子问题。

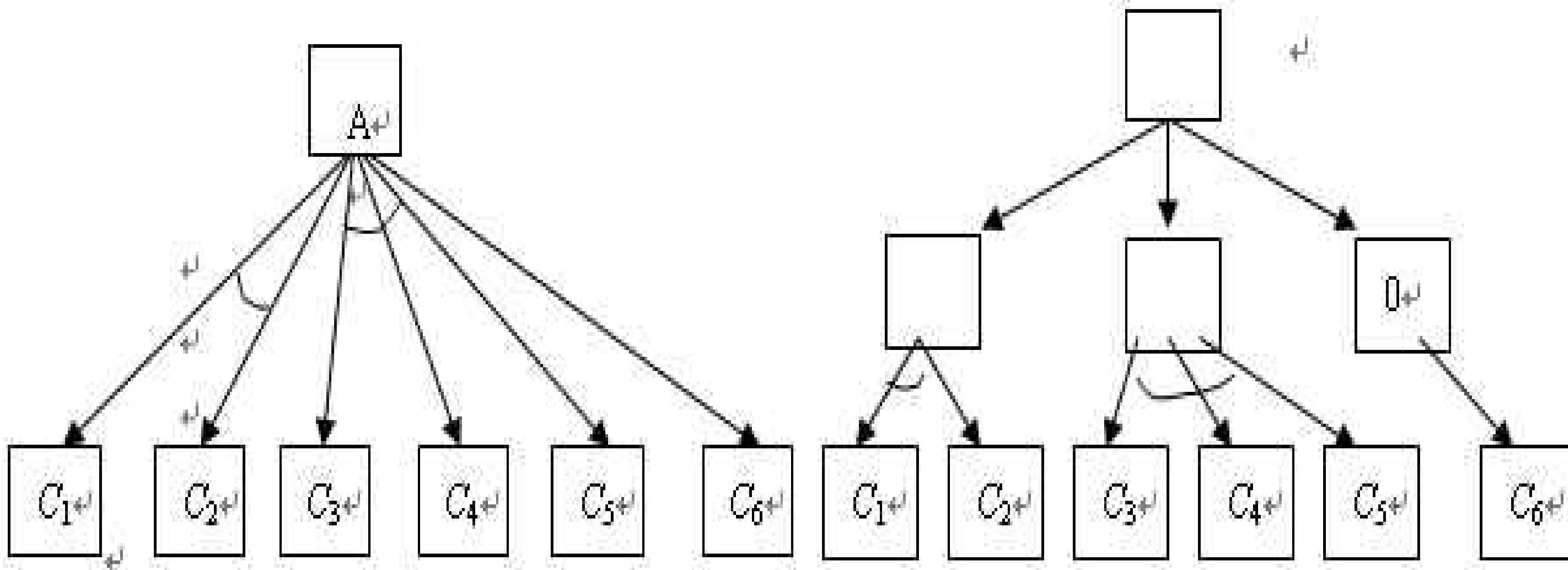


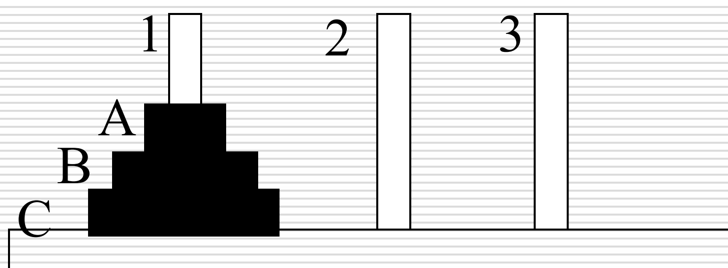
图 3-7 子问题替换集合的结构

图 3-8 各节点后继只含一个 K 连接弧的 AND-OR 图

与或图

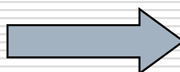
梵塔问题

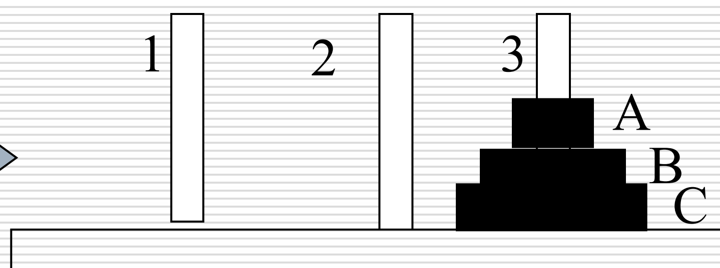
- **问题描述：**
- 初始状态下三个盘按A、B、C顺序堆放在1号柱子上；
- 目标状态下三个盘以同样次序顺序堆放在3号柱子上；
- 盘子的**搬移规则**：
  - 每次只能搬一个盘子；
  - 较大盘不能压放在较小盘之上；



初始状态

(1 1 1)

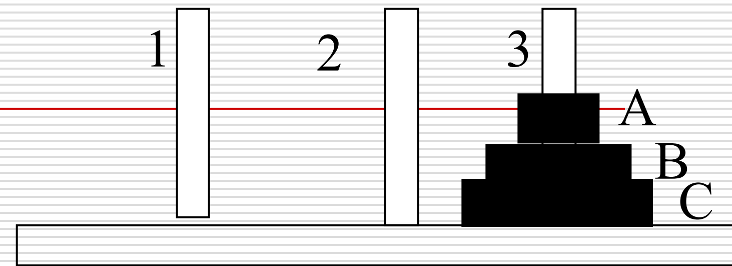




目标状态

(3 3 3)

梵塔问题——状态空间图



目标状态

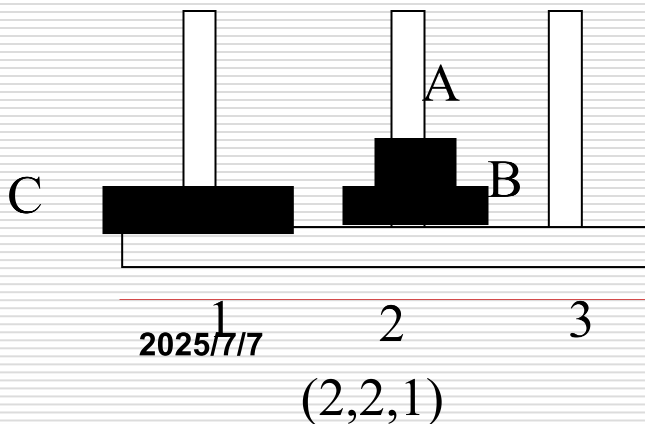
(3 3 3)

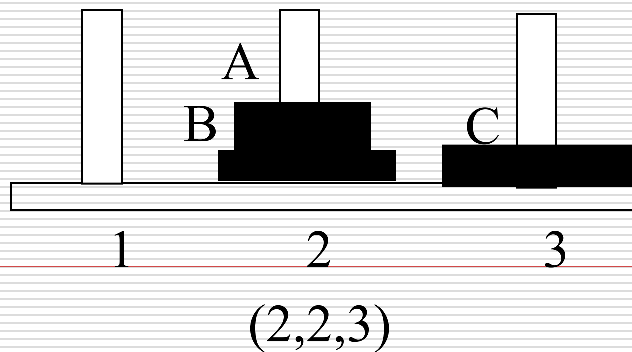
(1,1,1)→(3,3,3)

(1,1,1)→(2,2,1)

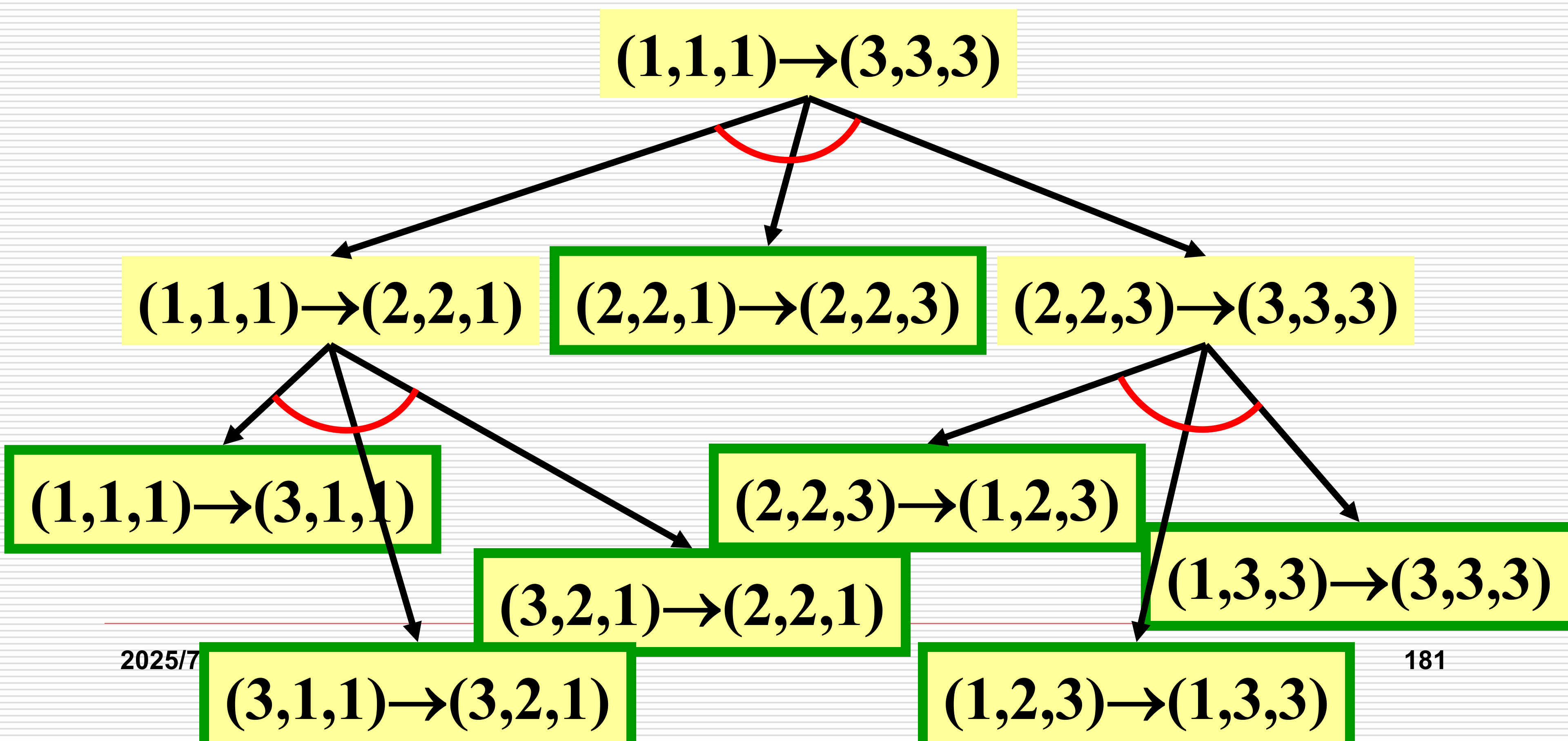
(2,2,1)→(2,2,3)

(2,2,3)→(3,3,3)



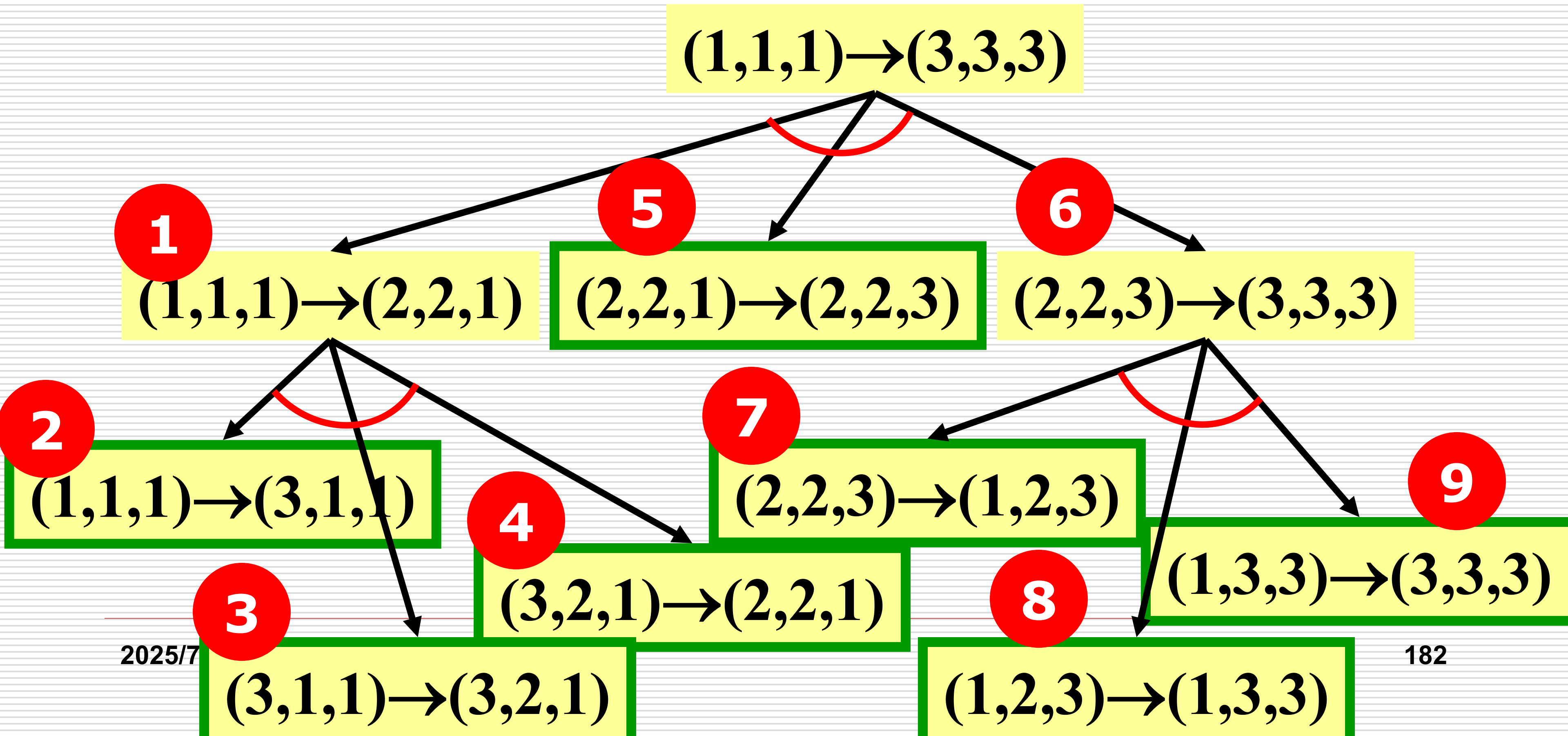


梵塔问题——状态空间图



梵塔问题

子问题间有交互作用，
问题分解注意正确的顺序



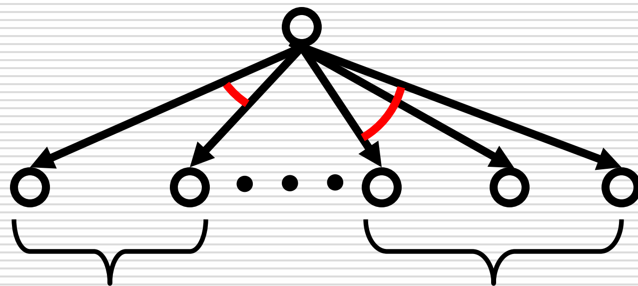
与或图搜索

问题归约求解问题的过程
表示为与或图搜索

- 与或图视为对一般图(或图)的扩展；★
  - 引入K-连接
    - 父子节点间可以存在“与”关系
  - 结果——解图。
    - 解答路径往往不复存在，代之以广义的解路径——解图。

与或图搜索

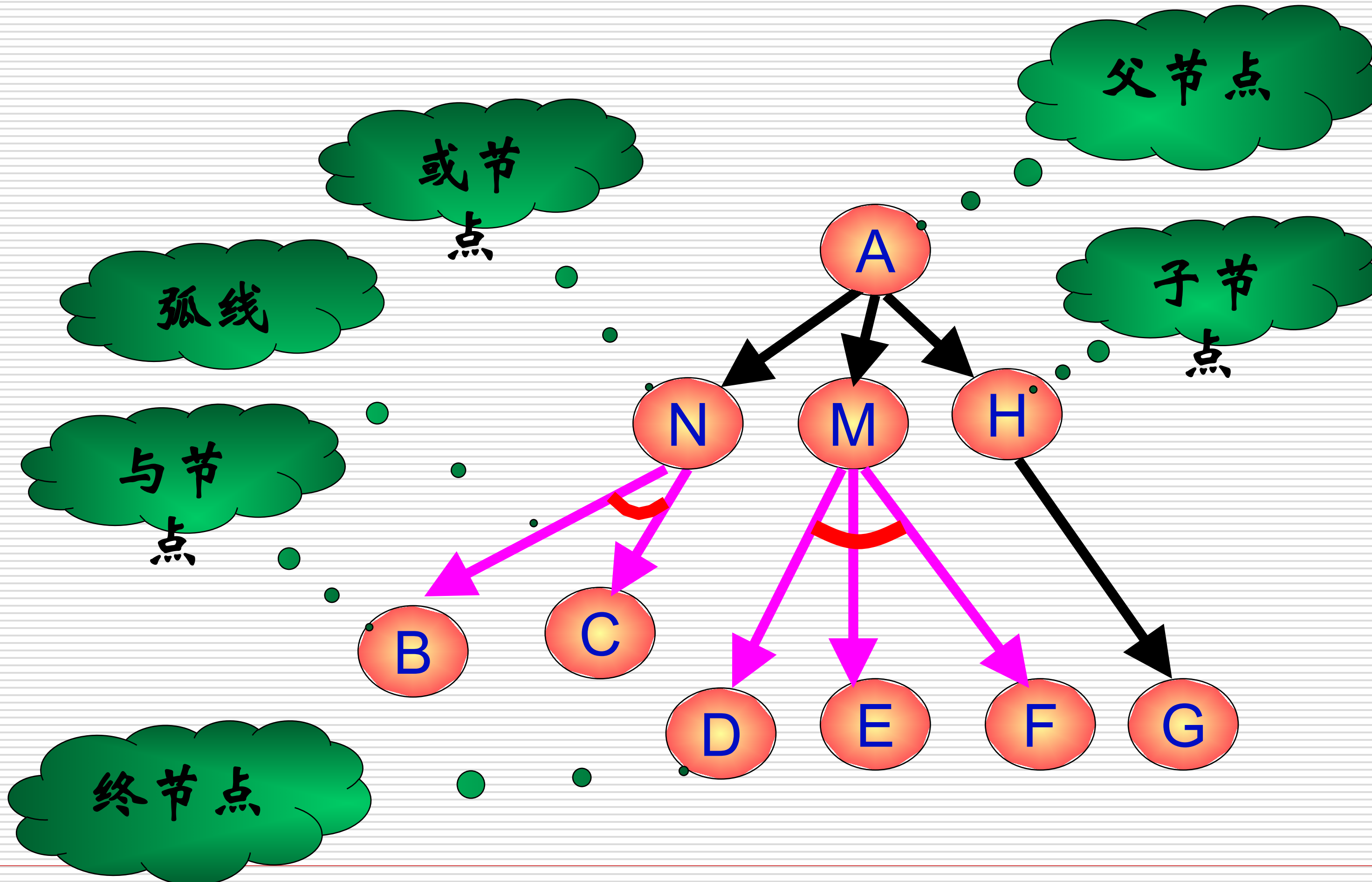
- 1)与或图搜索的基本概念
  - 1、K-连接★
    - 从父节点到K个子节点的连接，子节点间有“与”关系；
    - 以圆弧指示这些子节点间的“与”关系；
    - 一个父节点可以有多个K-连接
      - K-连接间——”或”关系
    - 当所有的K都等于1时，与或图蜕化为一般图（或图）。



与或图搜索

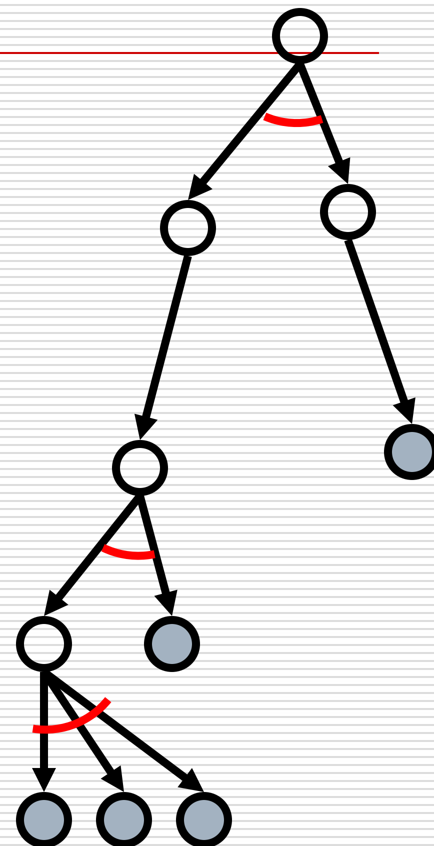
- 1)与或图搜索的基本概念
  - 2、根、叶、终节点★
    - 根节点——无父节点的节点
      - 用于指示问题初始状态（和一般图一样）
    - 叶节点——无子节点的节点
    - 终节点——能用于联合表示目标状态的节点
      - 终节点必定是叶节点，反之不然；
      - 目标状态——终结点的集合。

一些关于与或图的术语



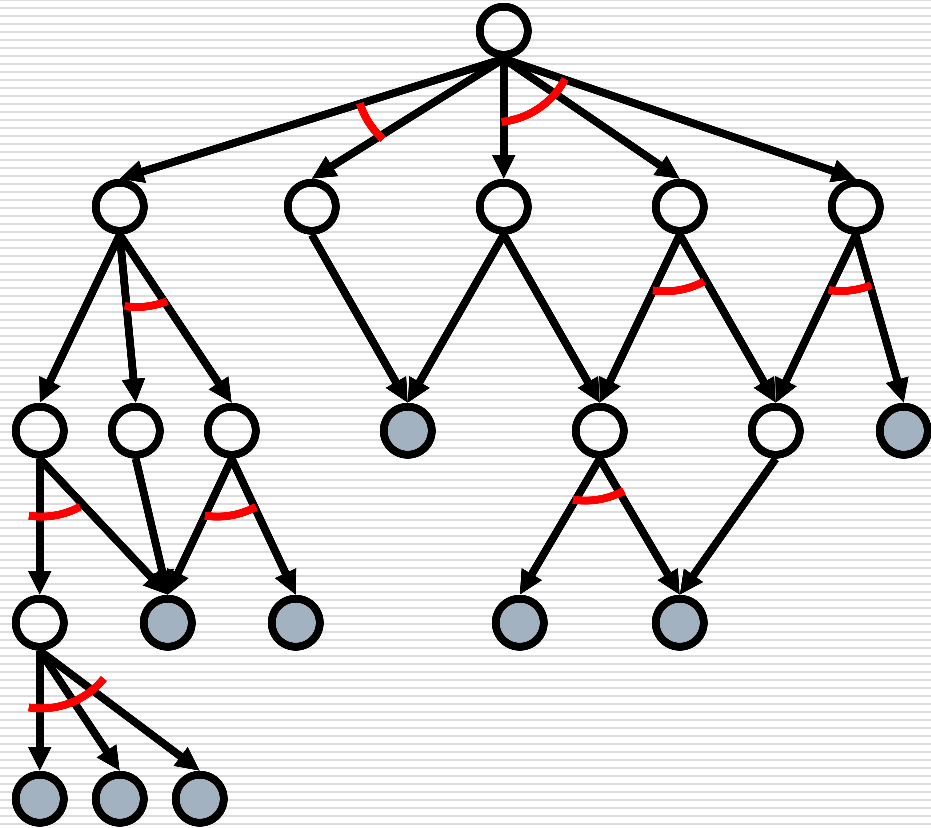
与或图搜索

- 1)与或图搜索的基本概念
  - 3、解图的生成★
    - 解图纯粹是一种“与”图
    - 解图中，节点或节点组间不存在“或”关系；
    - 所有叶节点都是终节点



与或图搜索

- 1)与或图搜索的基本概念
  - 3、解图的生成★
  - 自根节点开始选K-连接;
  - 从该K-连接指向的每个子节点出发，再选一K-连接;
  - 如此反复进行，直到所有K-连接都指向终节点为止.



单选题 1分

设置

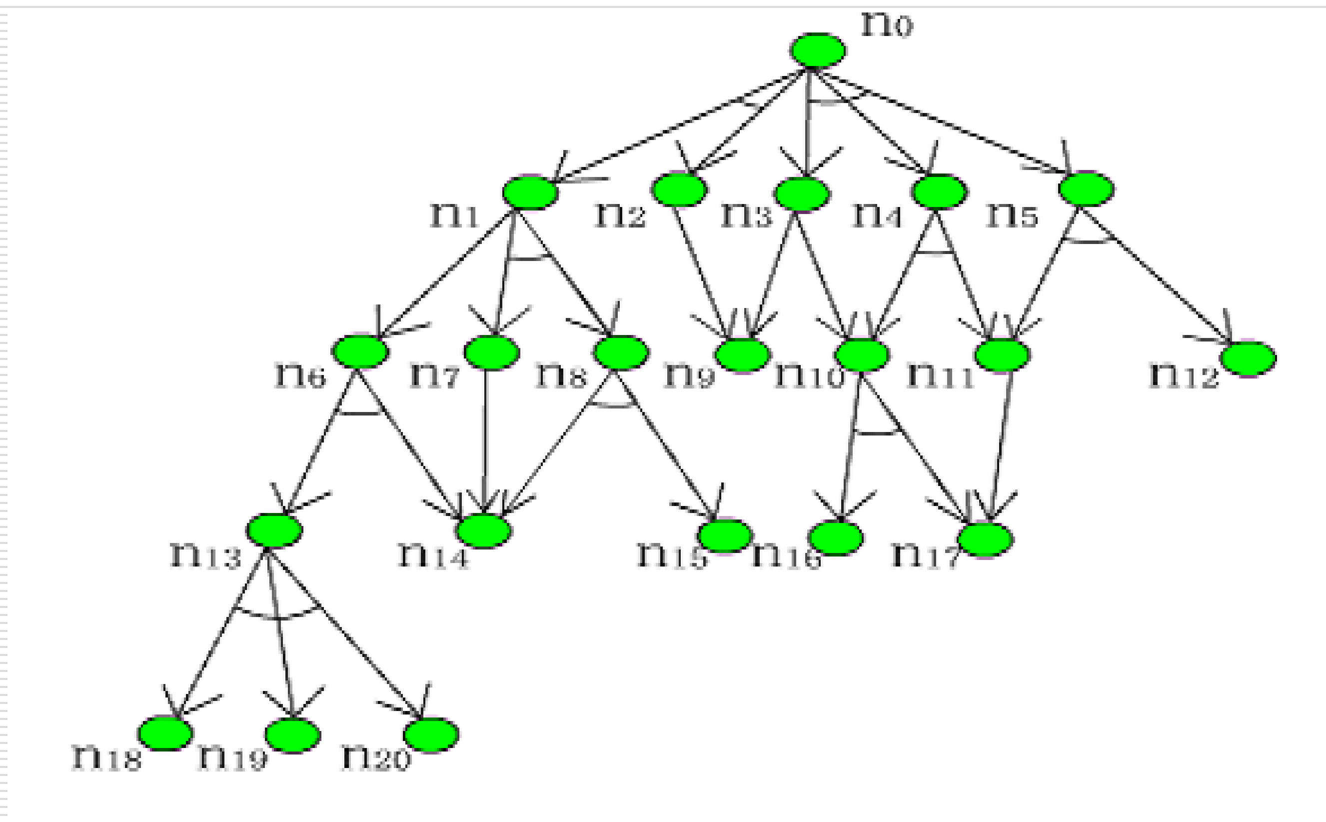
1、下列与或图中有多少个解图。

A 2

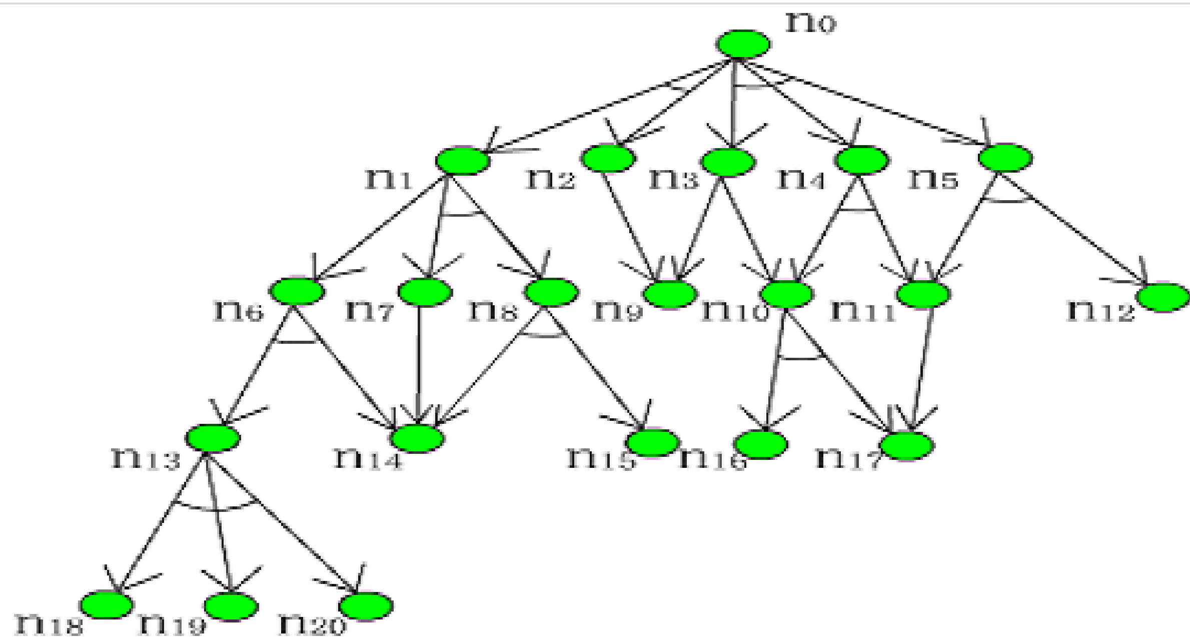
B 3

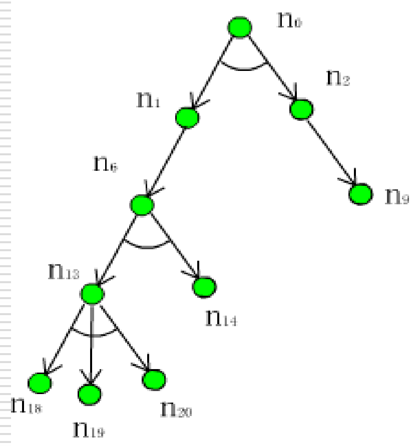
C 4

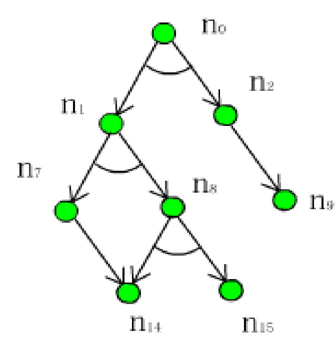
D 5

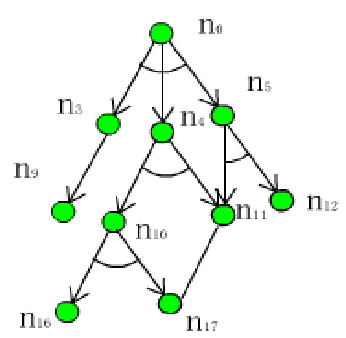


提交









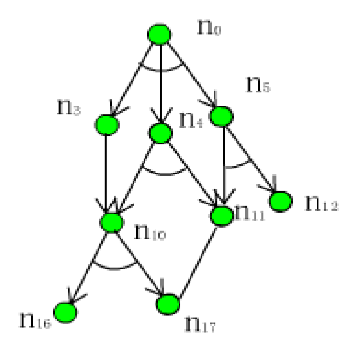


图2.20 四个可能的图解

与或图搜索

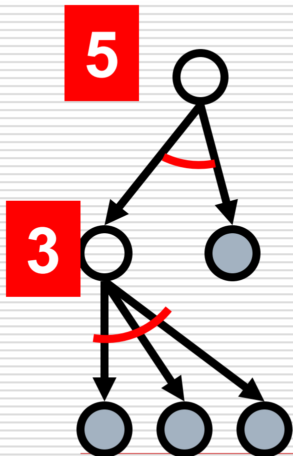
- 1)与或图搜索的基本概念
  - 3、解图的生成★
    - 解图纯粹是一种“与”图
      - 解图中，节点或节点组间不存在“或”关系；
      - 所有叶节点都是终节点
    - 与或图中存在“或”关系，搜索到多个解图；

与或图搜索

- 2) 解图、解图代价、能解节点和不能解节点的定义
  - (1)解图——与或图（记为G）任一节点（记为n）到终节点集合的解图（记为G‘）是G的子图。
    - (1)若n是终节点，则G‘就由单一节点n构成；
    - (2)若n有一K-连接指向子节点n1，n2，…nk,且每个子节点都有到终节点集合的解图，则G‘由该k-连接和所有这些相应于子节点的解图构成;
    - (3)否则不存在n到终节点集合的解图。

与或图搜索

- 2) 解图、解图代价、能解节点和不能解节点的定义
  - (2)解图代价★ ——以C(n)指示节点n到终节点集合解图的代价，并令K-连接的代价就为K, 则
    - (1)若n是终节点，则C(n) = 0；
    - (2)若n有一K-连接指向子节点n1，n2，…nk,且这些子节点每个都有到终节点集合的解图,则
      C(n) = K + C(n1) + C(n2) + … + C(nk)

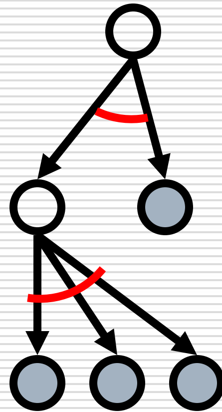


与或图搜索

- 2) 解图、解图代价、能解节点和不能解节点的定义
  - (3)能解节点 ★
    - (1) 终节点是能解节点；
    - (2) 若节点n有一K-连接指向子节点n1，n2，…nk, 且这些子节点都是能解节点，则n是能解节点；

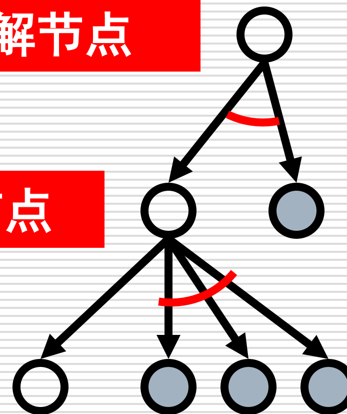
能解节点

能解节点



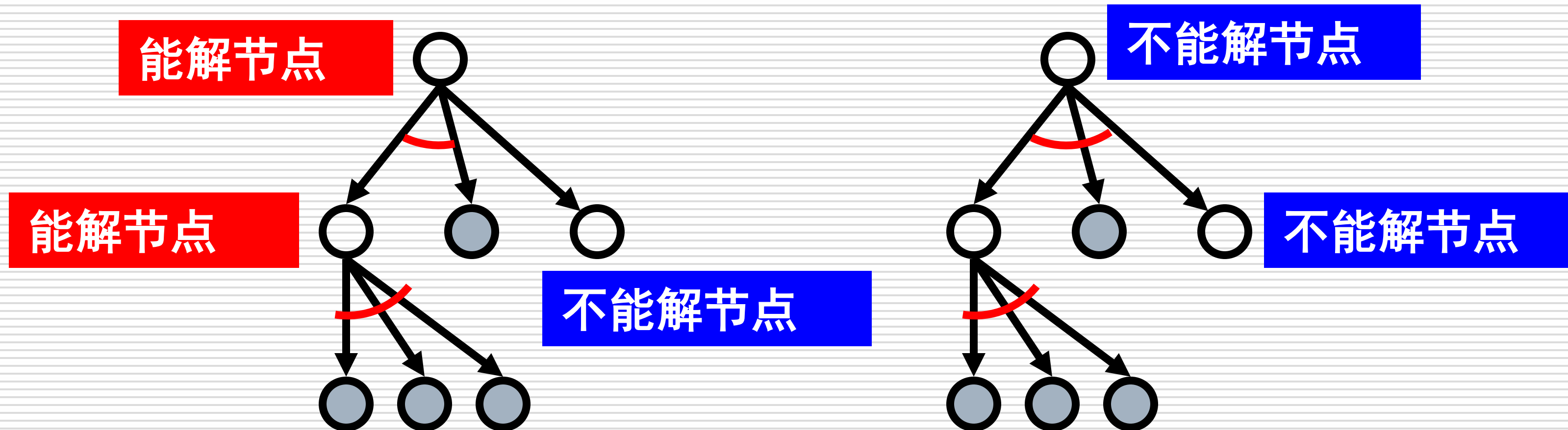
能解节点

能解节点



与或图搜索

- 2) 解图、解图代价、能解节点和不能解节点的定义
  - (4)不能解节点★
    - (1)非终节点的叶节点是不能解节点；
    - (2)若节点n的每一个K-连接都至少指向一个不能解节点，则n是不能解节点。



与或图的启发式搜索

- 与或图中搜索的是解图，不是解答路径；
  - 评价函数f(n)=h(n)
  - h(n)是对n到终节点集合解图最小解图代价的估计；
- 与或图中存在“或”关系，会有多个候选的局部解图；
  - 选择局部解图中可能代价最小的用于下一步搜索。
- 1)（局部）解图代价——f(n0) ★
  - n0——初始状态节点
  - 递归地计算出f(n0),比直接用h(n0)估算更为准确。
  - 父节点n的K-连接指向的子节点：n1，n2，…nk
  - f(n) = K + h(n1) + h(n2) + … + h(nk)，代替h(n)

与或图的启发式搜索

- 2) AO*算法★
- 符号说明：
  - G-搜索图；
  - G’-被选中的待扩展局部解图；
  - LGS-待扩展局部解图集；
  - n_0-根节点，即初始状态节点；
  - n-被选中的待扩展节点；
  - $f_i(n_0)$-第i个待扩展局部解图的可能代价。

与或图的启发式搜索

- 2) AO*算法★
- 算法划分二个阶段：
  - 1、初始化
    - 建立只包含初始状态节点n_0的搜索图G:={n_0}；
    - 待扩展局部解图集LGS:={}；
  - 2、搜索循环
    - 选择和扩展LGS中的局部解图；
    - 精化新局部解图代价的估计；
    - 传递节点的能解性。

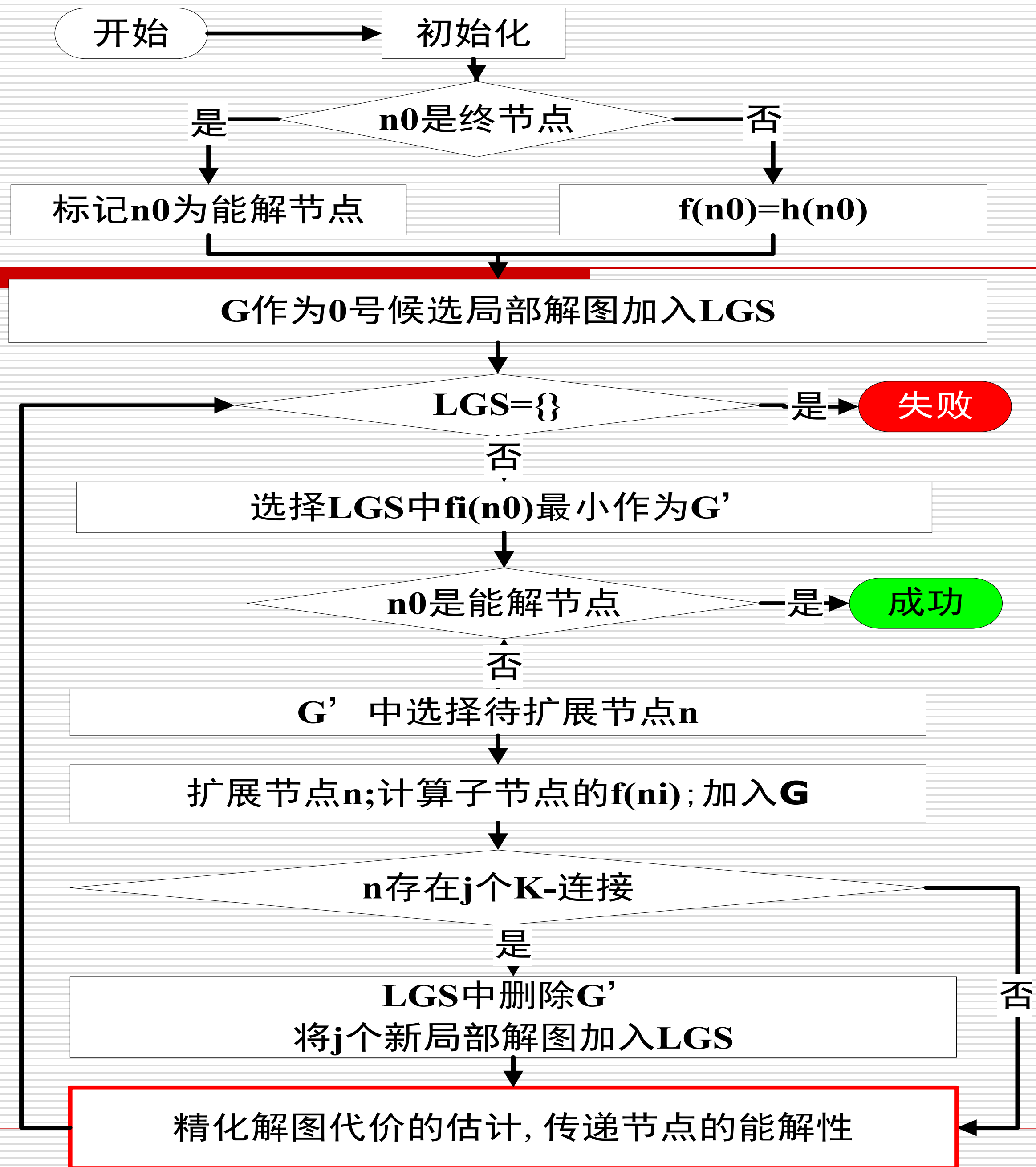
与或图的启发式搜索

- 2、搜索循环
  - 选择和扩展LGS中的局部解图；
    - ④选择LGS中$f_i(n_0)$最小的待扩展解图G’；
    - ⑤随机选择G’中一个非终节点的叶节点作为n；
    - ⑥扩展n
      - 建立K-连接，子节点n_i并加入G；
      - 计算子节点n_i的$f(n_i)=h(n_i)$
    - ⑦若n存在j个K-连接
      - LGS中删除G’
      - 将j个新的局部解图加入LGS;

与或图的启发式搜索

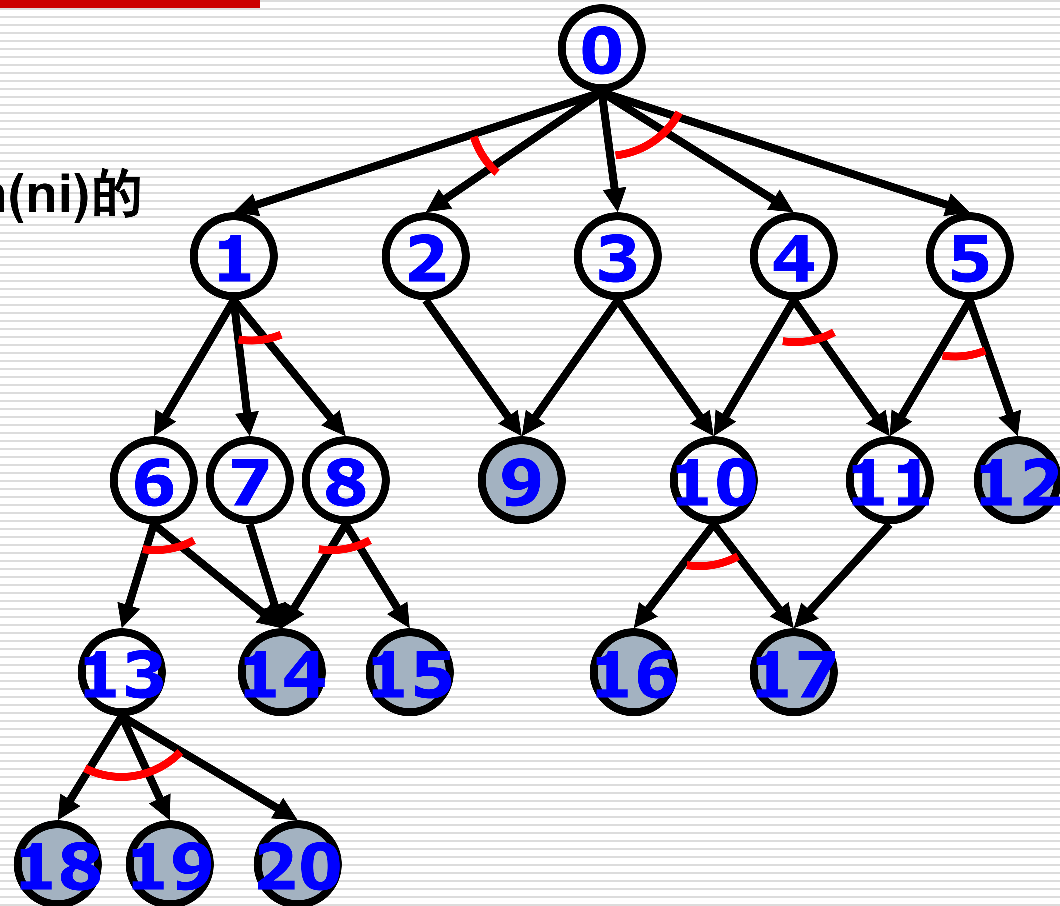
- 2、搜索循环
  - 选择和扩展LGS中的局部解图；
  - ⑧精化新局部解图代价的估计
    - 用公式 $f(n) = K + h(n1) + h(n2) + \cdots + h(nk)$ 取代原先的 $f(n)$；
    - 递归地作用到初始节点 n_0；
  - ⑨传递新局部解图中节点的能解性
    - 标记作为终节点的子节点为能解节点；
    - 递归地传递节点的能解性到初始节点 n_0。

$f(n)=h(n)$

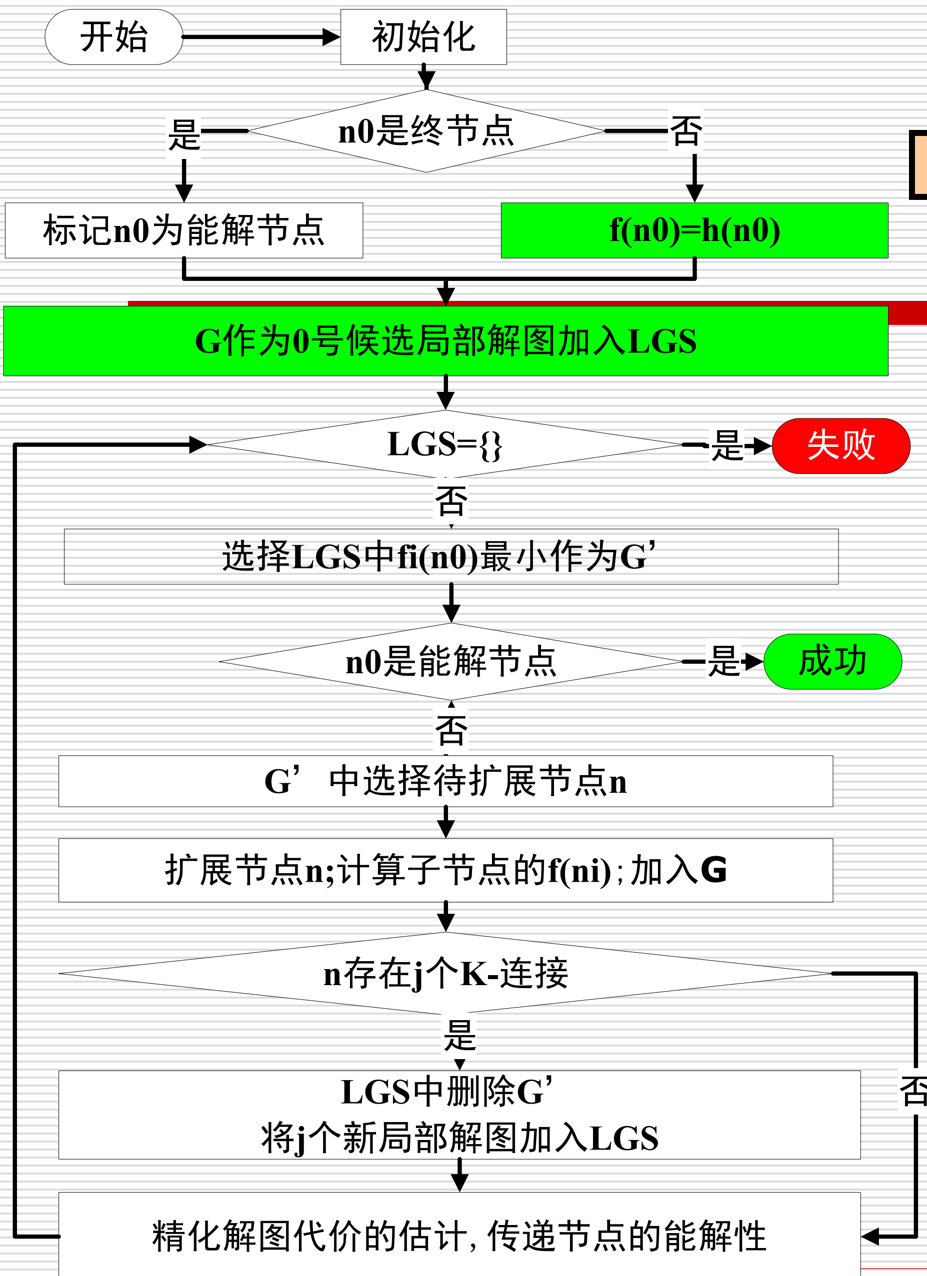

开始
初始化
n0是终节点
是
否
标记n0为能解节点
f(n0)=h(n0)
G作为0号候选局部解图加入LGS
LGS={}
是
失败
否
选择LGS中fi(n0)最小作为G’
n0是能解节点
是
成功
否
G’ 中选择待扩展节点n
扩展节点n;计算子节点的f(ni);加入G
n存在j个K-连接
是
否
LGS中删除G’
将j个新局部解图加入LGS
精化解图代价的估计，传递节点的能解性

与或图的启发式搜索

- **2) AO*算法**
- AO*算法应用例
- 搜索过程中，启发式函数h(ni)的估算如下：
- h(n0)=3
- h(n1)=2
- h(n2)=1
- h(n3)=1
- h(n4)=4
- h(n5)=2
- h(n6)=2
- h(n7)=1
- h(n8)=1
- h(n13)=3



候选的待扩展局部解图集LGS:

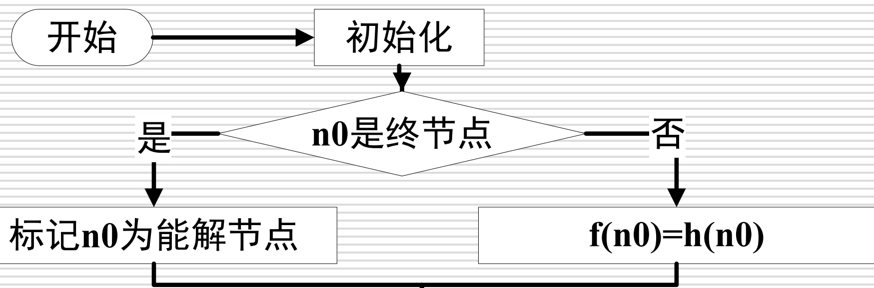


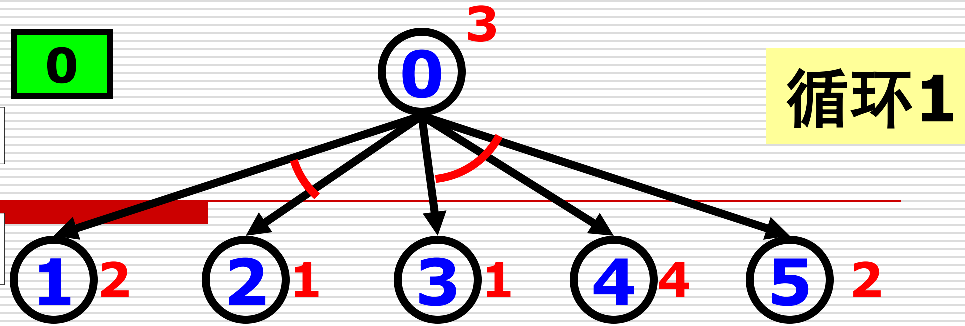
0

0^3

初始化

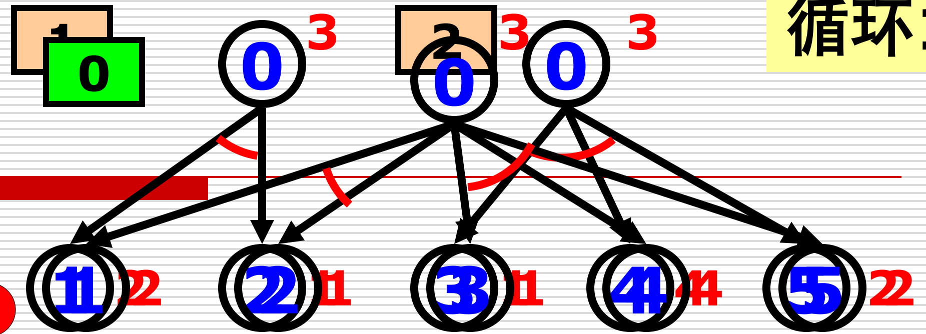
候选的待扩展局部解图集LGS:





候选的待扩展局部解图集LGS:

循环1



开始 → 初始化

n0是终节点
- 是 → 标记n0为能解节点
- 否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} — 是 → 失败

否

选择LGS中fi(n0)最小作为G’

n0是能解节点 — 是 → 成功

否

G’ 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

n存在j个K-连接

是

LGS中删除G’
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性

候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} —是→ 失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点 —是→ 成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

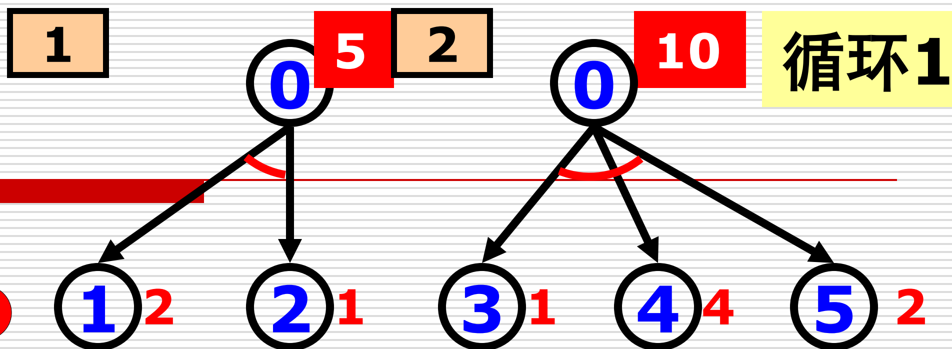
n存在j个K-连接

是

LGS中删除G'
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性



$f(n) = K + h(n1) + h(n2) + \ldots + h(nk)$

取代原先的f(n);

$f_1(n_0) = 2 + h(n1) + h(n2)=5$

$f_2(n_0) = 3 + h(n3) + h(n4)+h(n5)$
$=10$

候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} 是 → 失败

否

选择LGS中fi(n0)最小作为G’

n0是能解节点 是 → 成功

否

G’ 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

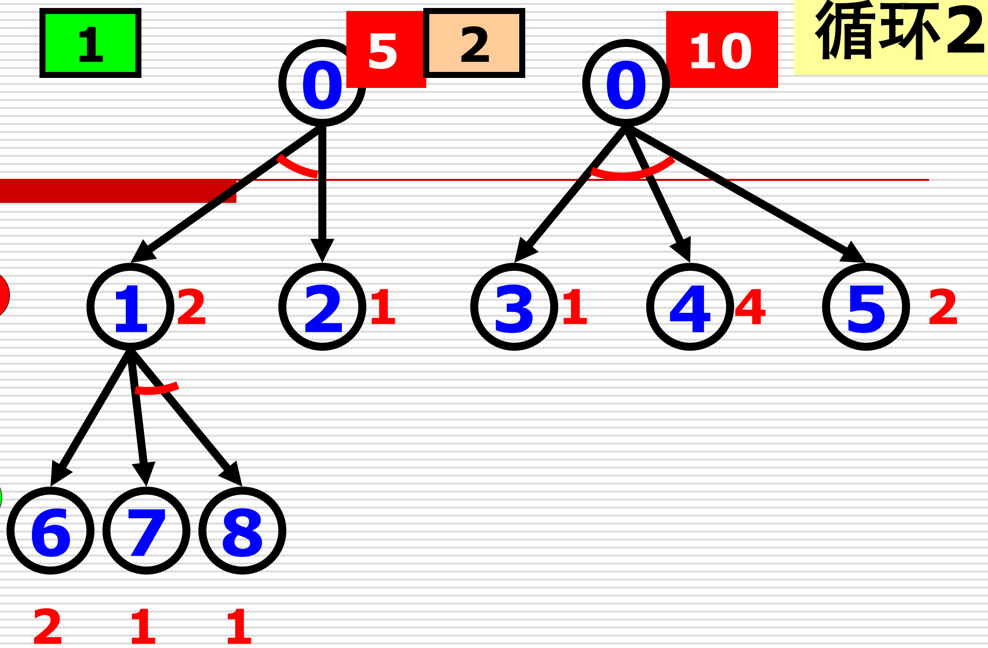
n存在j个K-连接

是

LGS中删除G’
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性



候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} —是→ 失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点 —是→ 成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

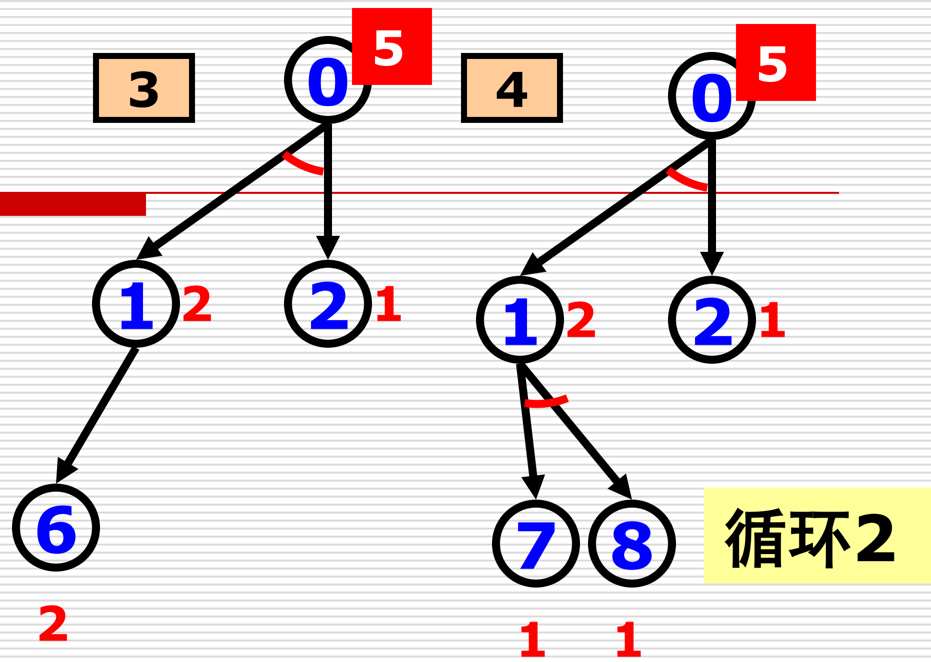
n存在j个K-连接

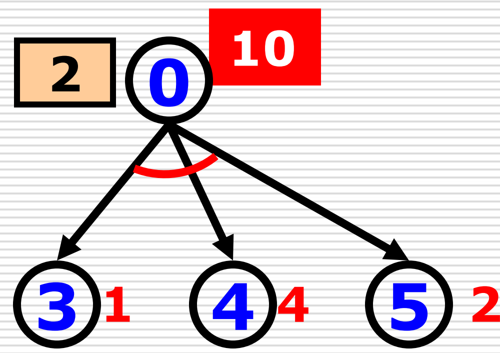
是

LGS中删除G'
将j个新局部解图加入LGS

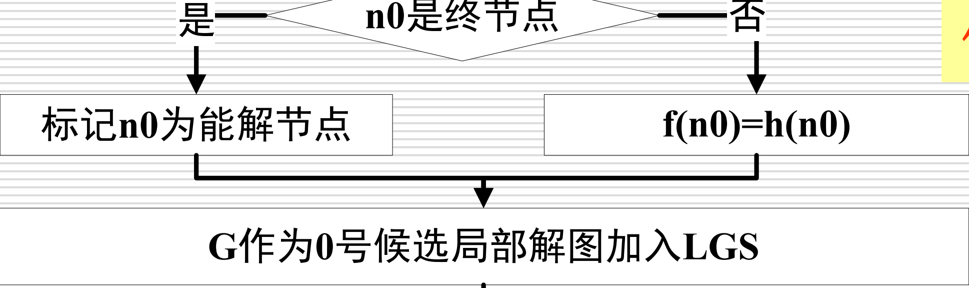
否

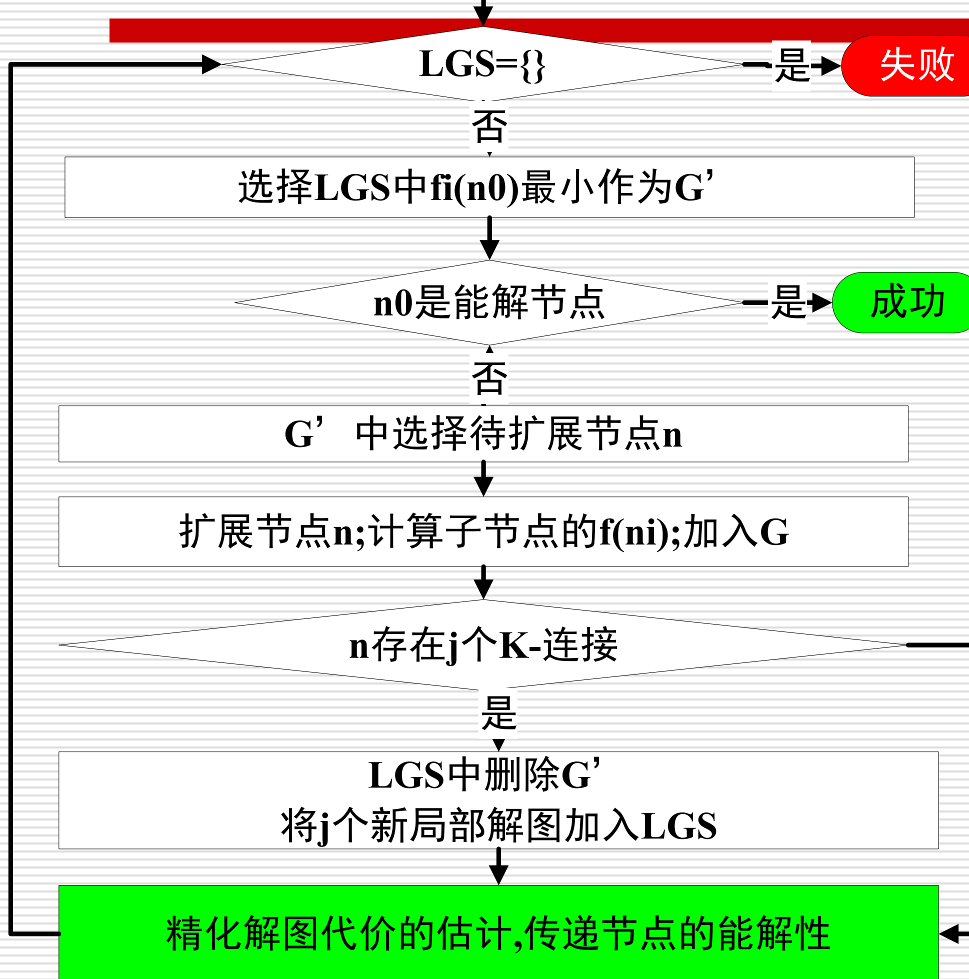
精化解图代价的估计,传递节点的能解性

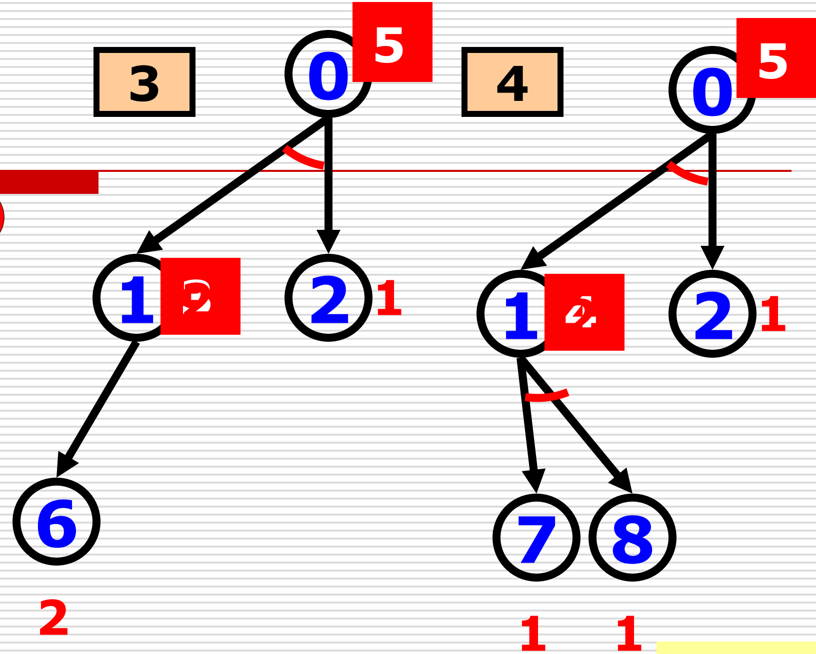




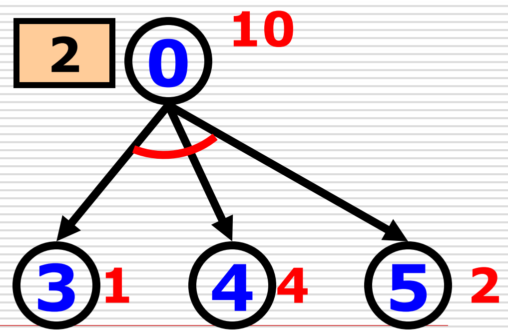
候选的待扩展局部解图集LGS:







循环2



标记n0为能解节点

f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={}

是

失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点

是

成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

n存在j个K-连接

是

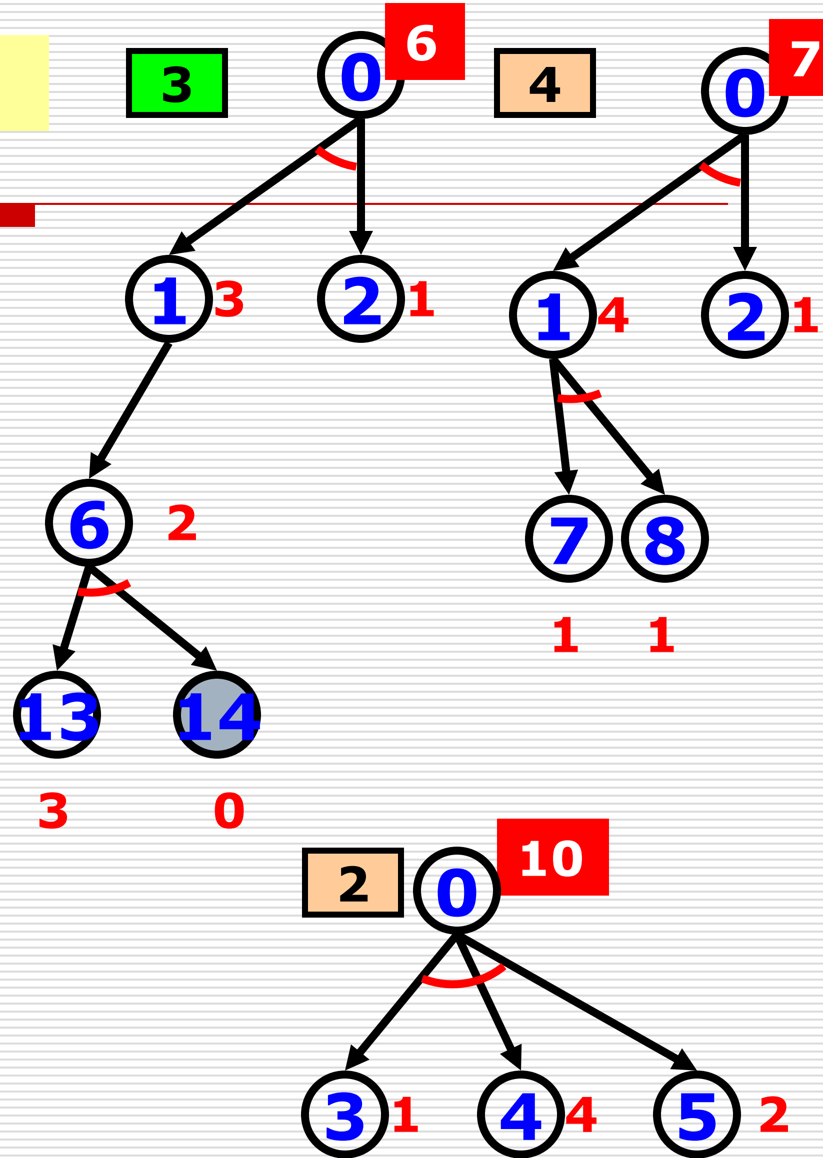
LGS中删除G'
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性

循环3

候选的待扩展局部解图集LGS:



候选的待扩展局部解图集LGS:

循环3

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} —是→ 失败

否

选择LGS中fi(n0)最小作为G’

n0是能解节点 —是→ 成功

否

G’ 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

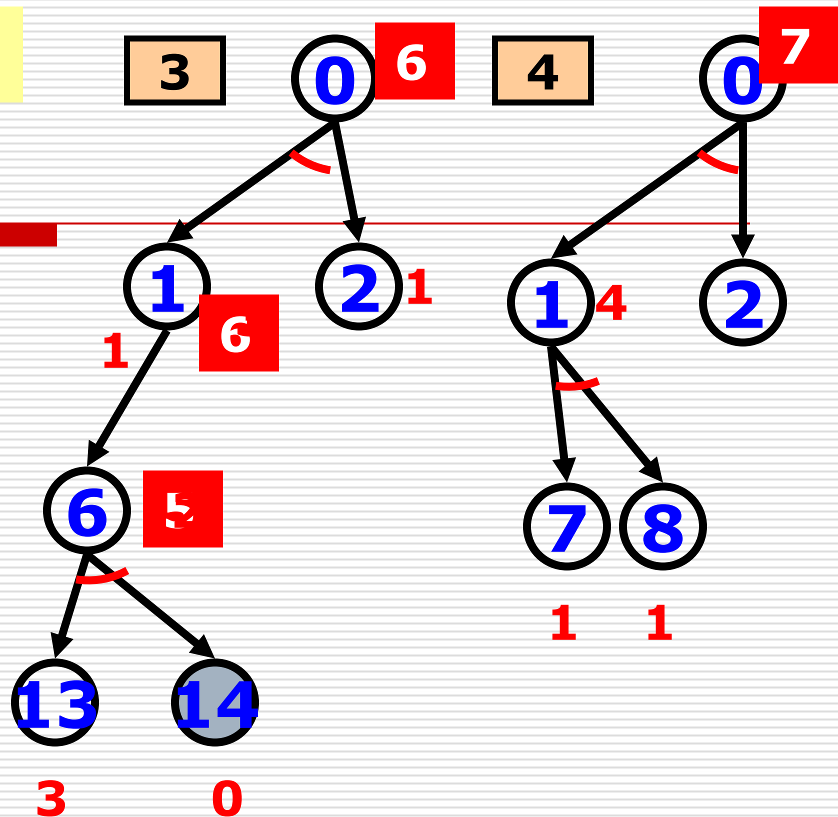
n存在j个K-连接

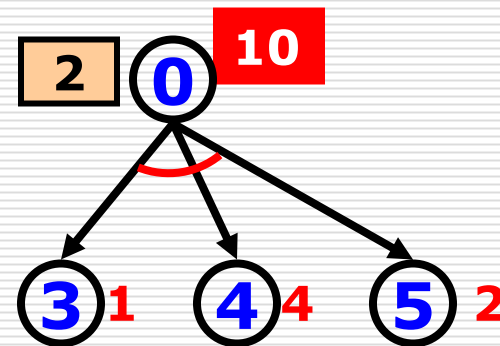
是

LGS中删除G’
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性





候选的待扩展局部解图集LGS:

循环4

开始
初始化
n0是终节点
是
标记n0为能解节点
否
f(n0)=h(n0)
G作为0号候选局部解图加入LGS
LGS={}
是 失败
否
选择LGS中fi(n0)最小作为G'
n0是能解节点
是 成功
否
G' 中选择待扩展节点n
扩展节点n;计算子节点的f(ni);加入G
n存在j个K-连接
是
LGS中删除G'
将j个新局部解图加入LGS
否
精化解图代价的估计,传递节点的能解性

3 0 9
1 6
2 1
6 5
13 3
14 0

4 0 7
1 4
2 1
7 1
8 1
14 0

2 0 10
3 1
4 4
5 2

循环5

候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={}

是 → 失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点

是 → 成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

n存在j个K-连接

是

LGS中删除G'
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性

3 0 9 1 6 2 1 6 5 13 3 14 0

4 0 7 1 4 2 1 7 1 8 1 14 0 15 0

2 0 10 3 1 4 4 5 2

循环5

候选的待扩展局部解图集LGS:

开始

初始化

n0是终节点

是

否

标记n0为能解节点

f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={}

是

失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点

是

成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

n存在j个K-连接

是

否

LGS中删除G'
将j个新局部解图加入LGS

精化解图代价的估计,传递节点的能解性

3 0 9 1 6 2 1 6 5 13 14 3 0

4 0 7 1 5 2 1 7 1 8 2 14 15 0 0

2 0 10 3 1 4 4 5 2

循环6

候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} —是→ 失败

否

选择LGS中fi(n0)最小作为G'

n0是能解节点 —是→ 成功

否

G' 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

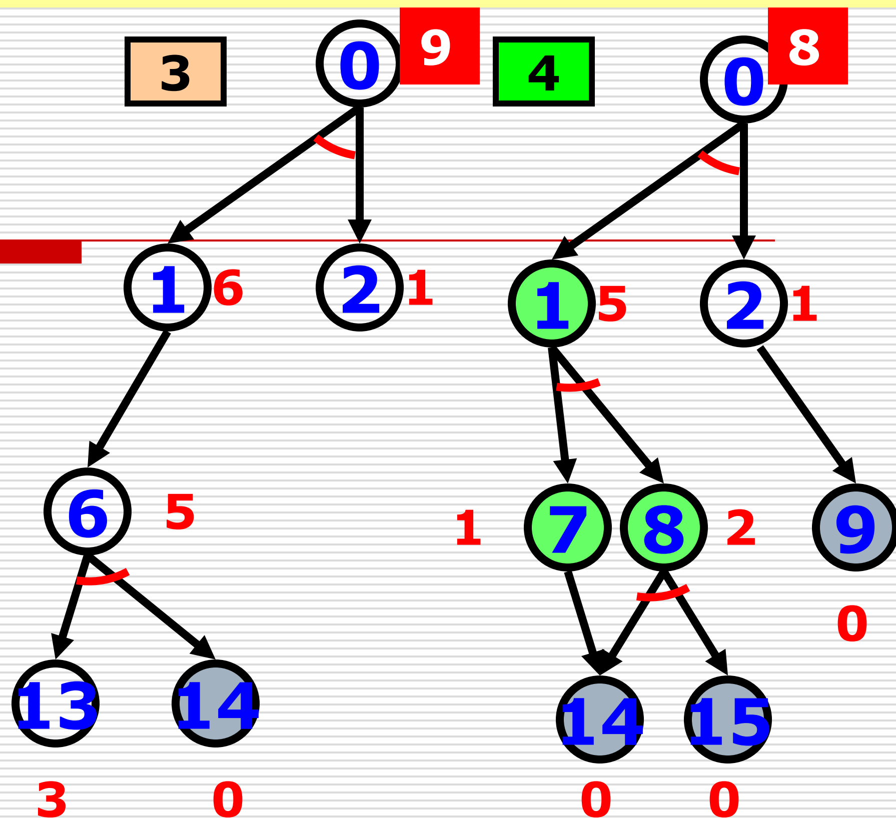
n存在j个K-连接

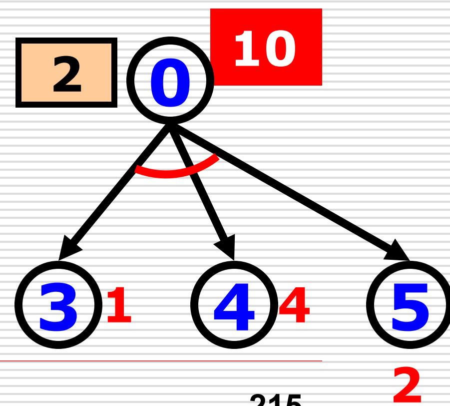
是

LGS中删除G'
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性





搜索成功！ 候选的待扩展局部解图集LGS:

开始 → 初始化

n0是终节点

是 → 标记n0为能解节点

否 → f(n0)=h(n0)

G作为0号候选局部解图加入LGS

LGS={} 是 → 失败

否

选择LGS中fi(n0)最小作为G’

n0是能解节点 是 → 成功

否

G’ 中选择待扩展节点n

扩展节点n;计算子节点的f(ni);加入G

n存在j个K-连接

是

LGS中删除G’
将j个新局部解图加入LGS

否

精化解图代价的估计,传递节点的能解性

3 0 9 1 6 2 1 6 5 13 14 3 0

4 0 8 1 5 2 1 7 1 8 2 9 0 14 15 0 0

2 0 10 3 1 4 4 5 2

与或图的启发式搜索

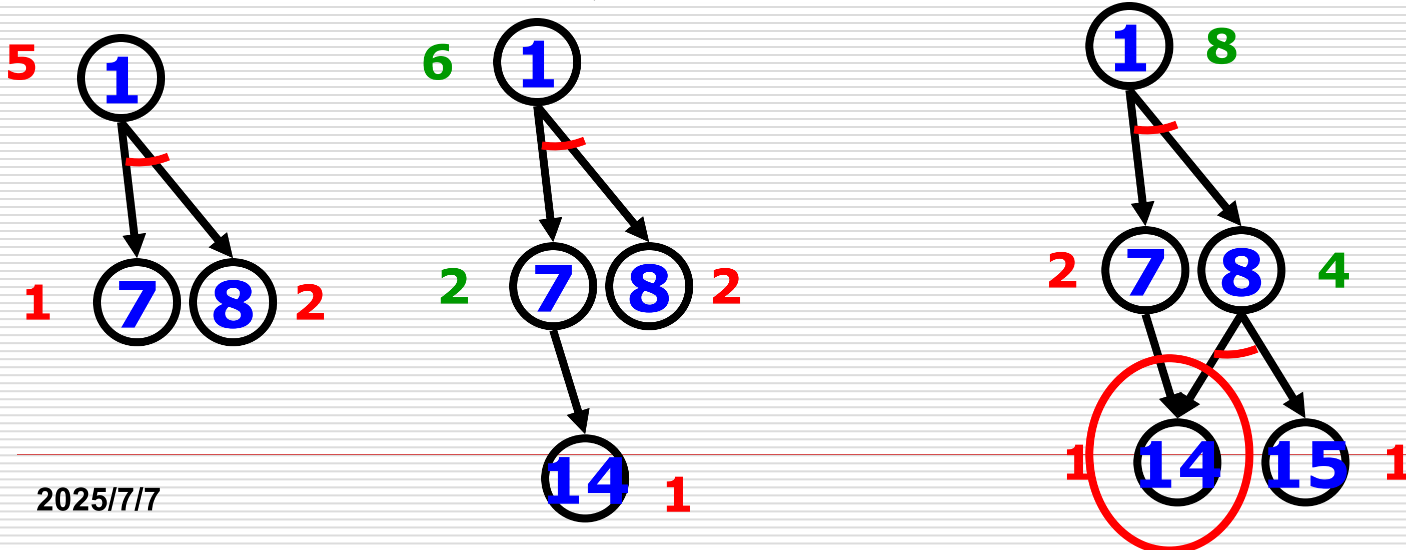
- 4）算法应用的若干问题
- 1、从局部解图G'中选择加以扩展的节点n
  - 与或图搜索的是解图而非解路径；
  - 选择f(n) = h(n)的值最小的节点n加以扩展并不一定会加速搜索过程；
  - 应选择导致解图代价发生较大变化的节点n优先加以扩展；
    - 使搜索的注意力快速地聚焦到实际代价较小的候选解图上；
  - 简单情况下，可随机选择加以扩展的节点。

与或图的启发式搜索

- 4）算法应用的若干问题
- 2、算法AO*与A*的比较★
  - 解图——解答路径，
  - 估计代价最小的局部解图加以优先扩展——OPEN表中f(n)最小的节点；
  - 只考虑评价函数f(n)=h(n)——同时计算分量g(n)和h(n)，
  - 应用LGS存放待扩展局部解图，并依据fi(n0)值排序——应用OPEN表和CLOSE表分别存放待扩展节点和已扩展节点，并依据f(n)值排序OPEN表。

与或图的启发式搜索

- 4）算法应用的若干问题
- 3、解图代价的重复计算
  - 某些子节点可能会有多个父节点；
  - 这种子节点到终节点集合的解图代价在计算自根节点n0出发的解图时被重复累计。



约束满足搜索

- 约束满足问题（CSP）就是为一组变量寻找满足约束的赋值。
- 如，N-皇后问题就是一个约束满足问题。这里的问题就是为N个变量赋值，每个变量的值表示每行上皇后的问题，值域均为[1，N]，约束就是N个皇后谁也“吃”不到谁。

- 定义 一个约束满足问题表述为一个三元组（V，D，C），V为n个变量的集合$V=\{v_1, ..., v_n\}$，D为变量vi（i=1, 2, ..., n)相应的取值集合$D=\{D_1, ..., D_n\}$，C为约束的有限集合，其中每个约束对若干变量同时可取的值做出限制。问题的解是对所有变量，满足所有约束的赋值。

- 说明：

（1）在上述定义中，限定每个相应于变量v_i的取值集合D_i（$0\leqslant i\leqslant n$）都是**离散**的和**有限**的。

（2）C为约束的有限集合，约束是一个或多个对象属性间的数学或逻辑关系。如，把10元钱换成1、2、5元钱，1、2、5元钱为对象，对象的个数为各自的属性，数学关系可以表示为：c1 + 2c2 + 5c3 = 10。如果问题的每个约束仅涉及两个变量，则称为二元约束问题（BCSP）。

（3） 如果至少存在一个解答满足某个约束，则称该约束是可满足的。例如，对于约束{（x + y≤2）∧(0≤x, y≤2)}(x和y为整数)是可满足的，这里x=y=1。如果x和y为整数，则约束{（x + y≤2）∧(x＞2, y＞2)}是不可满足的。

（4）如果找不到一个变量值的组合，使之满足所有的约束，则可以找到一个满足最大数目约束的解，这种情况称为最大约束满足问题。

- 目前约束推理的研究主要集中在两个方面：
  - 约束搜索
  - 约束语言
- 约束搜索主要研究有限域上的约束满足，对有限域而言，约束满足问题一般是一个NP难问题，因此不可能存在一个线性时间的算法能够找到所有的解答。

- 例 地图着色问题：对于下图所示的地图，从{ 红（R），绿（G），黄（Y）}中选择一种颜色赋予图中的国家，使得相邻的国家具有不同的色彩。

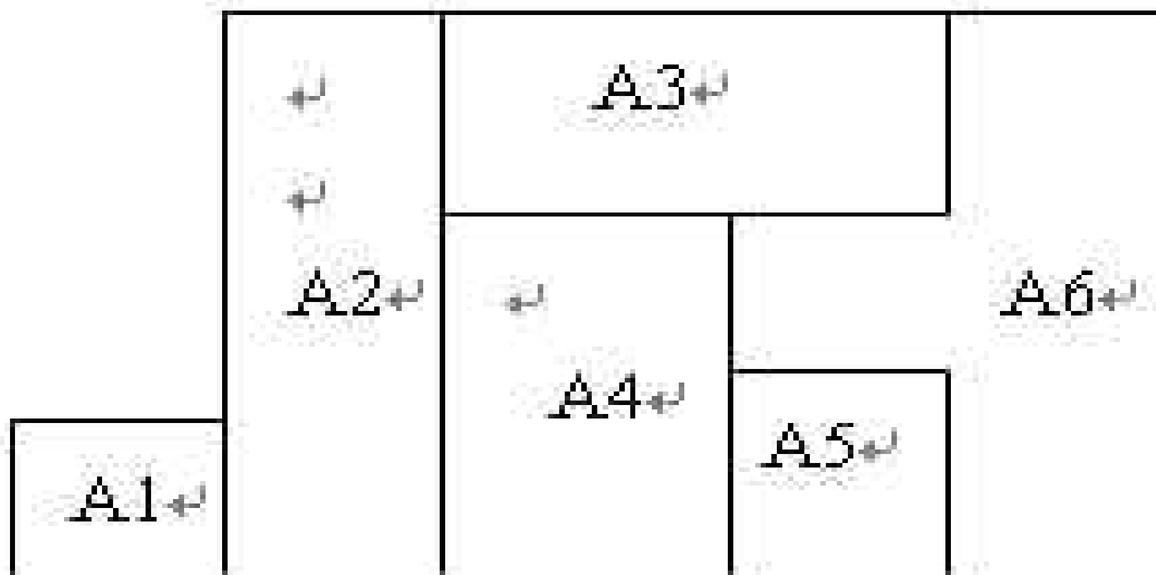


图 3-24 假设的地图

- 用下图来表示相邻关系，其中结点表示国家，连线表示结点之间的邻接关系，约束用≠表示。
- 首先，对于任意一个国家，赋予任意一个色彩。假定首先A1赋予R。用Ai≠R表示Ai不能被赋予R色彩。这样，可以有下面的步骤：

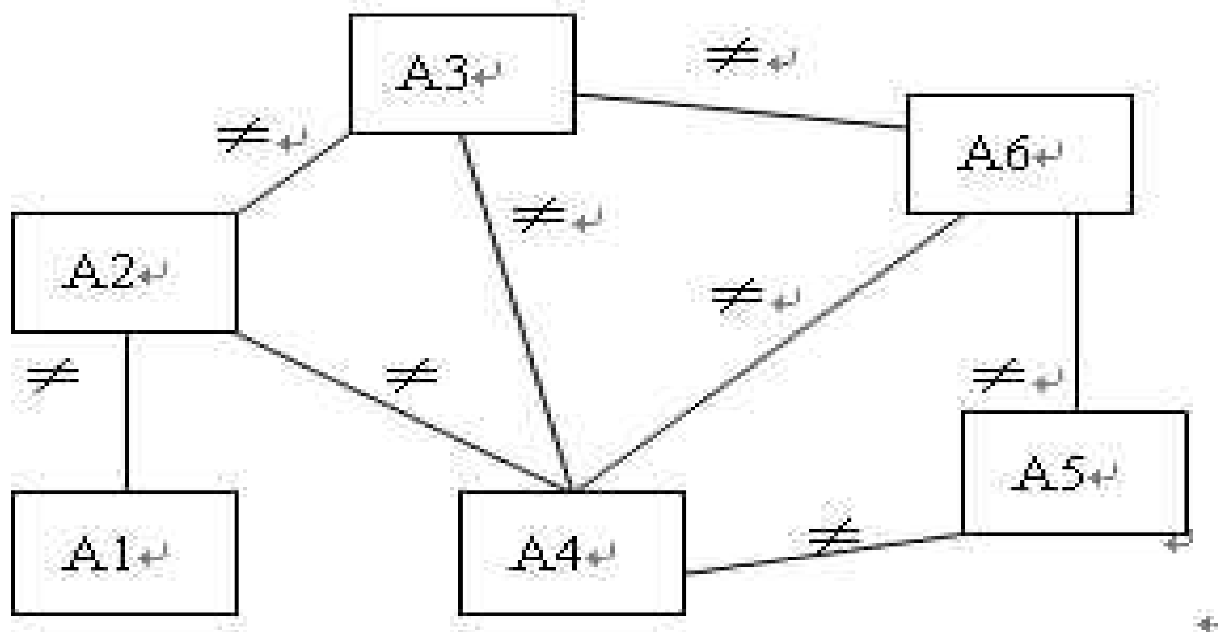


图 3.25 约束网络

（“≠”表示连线两端的国家不能使用同一种色彩）

Step1:{A1←R, A2≠R}

Step2:{A1←R, (A2←G, A3≠G, A4≠G)}

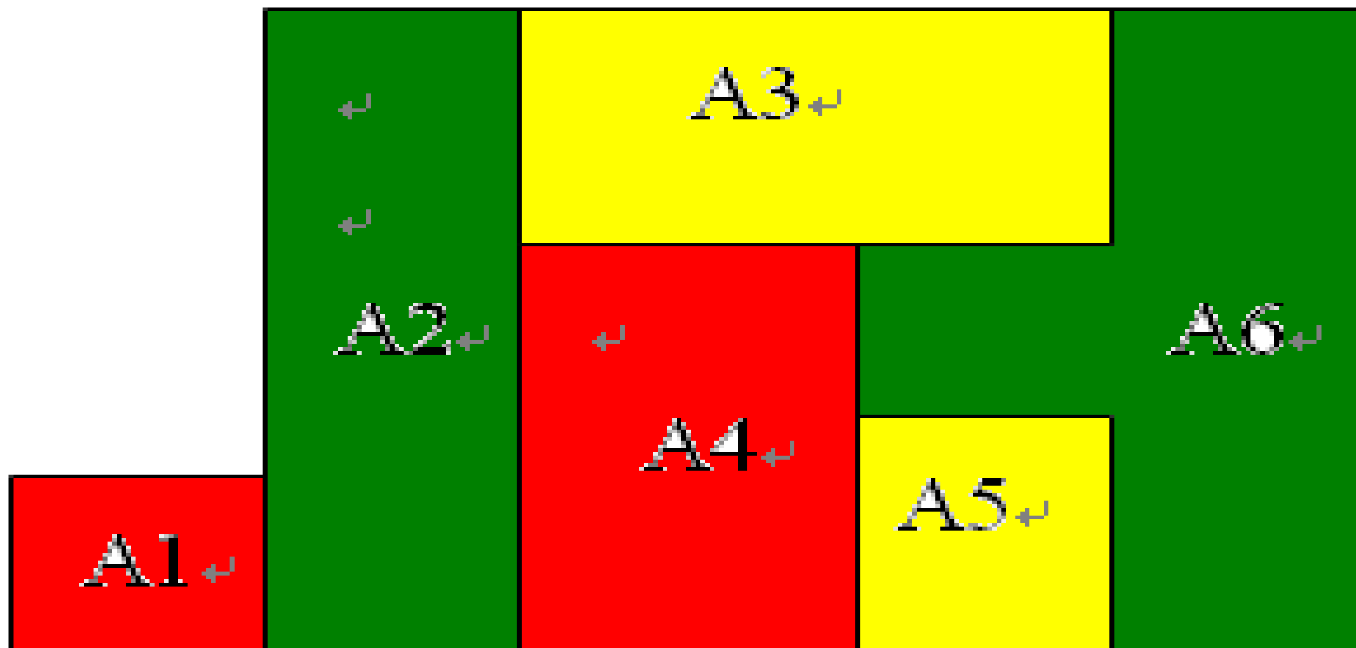
Step3:{A1←R, A2←G, (A3←Y, A4≠Y, A6≠Y), A4≠G}

Step4:{A1←R, A2←G, A3←Y, (A4←R, A5≠R, A6≠R), A6≠Y}

Step5:{A1←R, A2←G, A3←Y, A4←R, (A5←Y, A6≠Y), A6≠R}

Step6:{A1←R, A2←G, A3←Y, A4←R, A5←Y, A6←G}

- 在步骤2中，用A2←G代替步骤1中的A2≠R，并增加一些约束，用括号表示。这里步骤6得到一个可能的解答。

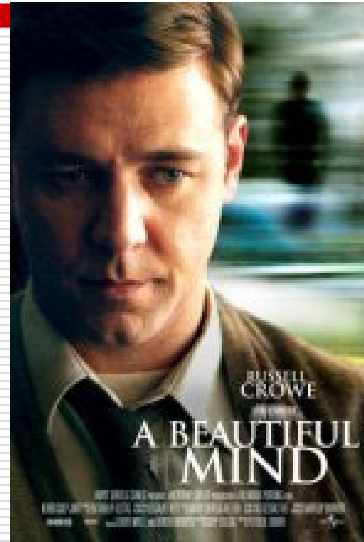


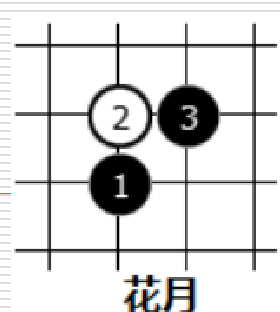
4.8 博弈

- 博弈提供了一个可构造的任务领域，在这个领域中，具有明确的胜利和失败；
- 博弈问题对人工智能研究提出了严峻的挑战。例如，如何表示博弈问题的状态、博弈过程和博弈知识等。

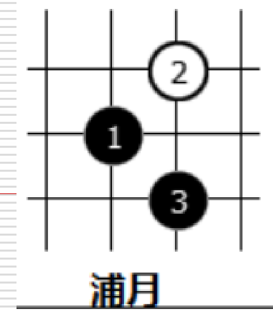
4.8 博弈

- 约翰•纳什的博弈论
- 业界的“棋类不败算法”
- 计算机的优势：
  - 计算力和攻击力
  - 在时间方面绝对占优
- 传统的计算机的劣势：
  - 没有大局观，不能针对不同的人采取不同的策略，不能根据不同的时期判断同一子的不同价值
  - 贪吃，怕死

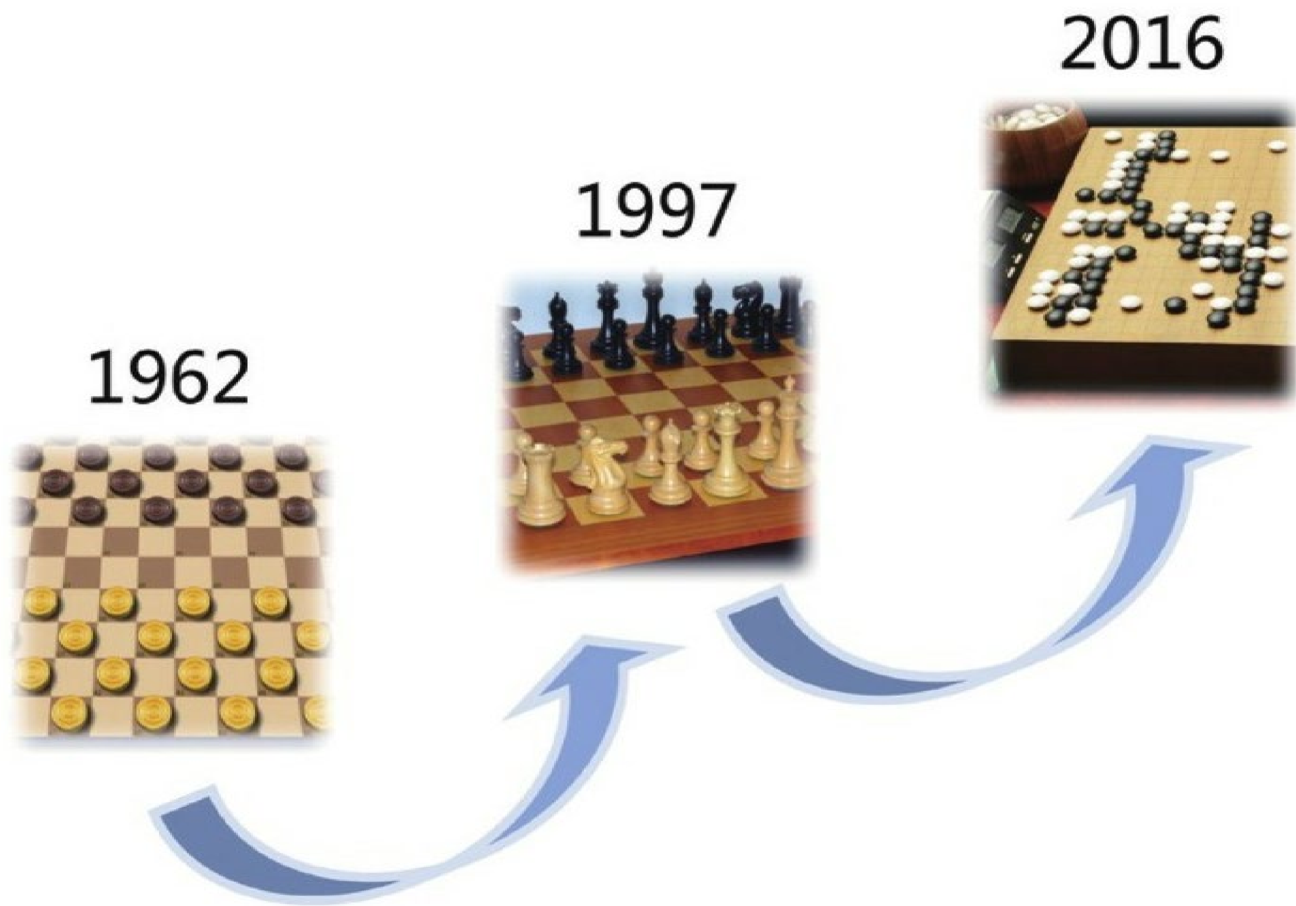




花月



浦月



三个时代，三场人机对弈

IBM的“深蓝”

北京时间1997年5月12日凌晨4点50分，美国纽约公平大厦，当IBM公司的“深蓝”超级电脑将棋盘上的一个兵走到C4的位置上时，国际象棋世界冠军卡斯帕罗夫对“深蓝”的人机大战落下帷幕，“深蓝”以3.5：2.5的总比分战胜卡斯帕罗夫。

正在与深蓝下棋的卡斯帕罗夫

一鸣惊人的“Alpha-Go”

2016年3月15日，谷歌围棋人工智能Alpha-Go与韩国棋手李世石进行最后一轮较量，AlphaGo获得本场比赛胜利，最终人机大战总比分定格在1:4。

Alpha-Go的执棋者—黄士杰

围棋和国际象棋的比较

	国际象棋	围棋
步数方面	每盘棋的走步大约是20～40步，每步的合法走法也在32～38之间	每盘棋的走步大约是250～300步，布局和中盘时每步的合法走法也有200种以上
整体与局部的关系	如果忽略将丢失重要棋子的走步，国际象棋可以剪掉大量的分支，并且使得战术大为简化	存在局部与整体之间的复杂关系，在每个局部都走出合理的走法，并不能导致最后的胜利
评价函数	以国王为主，胜负全集中在国王身上，这导致了很多算法上的简化，特别是导致了评价函数的简化	而围棋没有一个中心，各个子功能相同，子子平等，处处平等，没有一个简单而又精确的评价函数
状态空间	10^{120}	10^{360}
可选状态数	从多到少	从少到多

□这里讲的博弈是二人博弈，二人零和、全信息、非偶然博弈，博弈双方的利益是完全对立的。

- 所谓“二人零和”，是指在博弈中只有“敌、我”二方。且双方的利益完全对立，其赢得函数之和为零，即

$\varphi 1+\varphi 2=0$

式中，φ1为我方赢得(利益)；φ2为敌方赢得(利益)。

即：博弈的双方有三种结局：

(1)我胜：$\varphi 1>0$；敌负：$\varphi 2= -\varphi 1<0$。

(2)我负：$\varphi 1= -\varphi 2<0$；敌胜：$\varphi 2>0$。

(3)平局：$\varphi 1=0$，$\varphi 2=0$。

- 所谓“全信息”，是指博弈双方都了解当前的格局及过去的历史。
- 所谓“非偶然"，是指博弈双方都可根据得失大小进行分析，选取我方赢得最大，敌方赢得最小的对策，而不是偶然的随机对策。

博弈的特点

（1）对垒的双方MAX和MIN轮流采取行动，博弈的结果只能有3种情况：MAX胜、MIN败；MAX败，MIN胜；和局。

（2）在对垒过程中，任何一方都了解当前的格局和过去的历史。

（3）任何一方在采取行动前都要根据当前的实际情况，进行得失分析，选择对自己最为有利而对对方最不利的对策，在不存在"碰运气"的偶然因素，即双方都很理智地决定自己的行动。

➢这类博弈如一字棋、象棋、围棋等。

- 另外一种博弈是机遇性博弈，是指不可预测性的博弈，如掷硬币游戏等。

例：

- 假设有七枚钱币，任一选手只能将已分好的一堆钱币分成两堆个数不等的钱币，两位选手轮流进行，直到每一堆都只有一个或两个钱币，不能再分为止，哪个选手遇到不能再分的情况，则为输。

- 用数字序列加上一个说明表示一个状态，其中数字表示不同堆中钱币的个数，说明表示下一步由谁来分，
- 如（7，MIN）表示只有一个由七枚钱币组成的堆，由MIN走，MIN有3种可供选择的分法，即
  - （6，1，MAX），（5，2，MAX），（4，3，MAX），
- 其中MAX表示另一选手，不论哪一种方法，MAX在它的基础上再作符合要求的划分。

- 下图已将双方可能的方案完全表示出来了，无论MIN开始时怎么走法，MAX总可以获胜，取胜的策略用双线箭头表示。

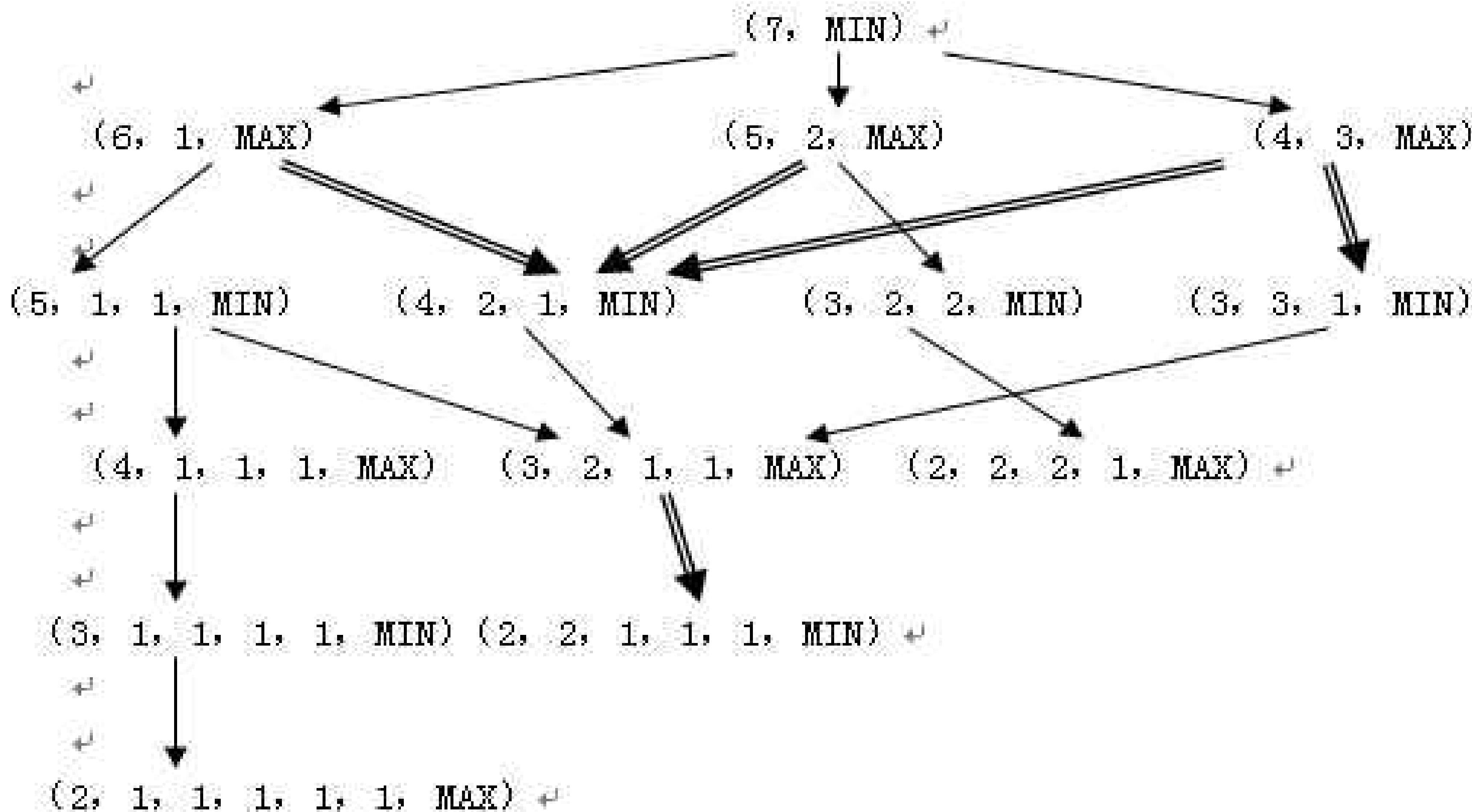


图 3-17 分钱币的博弈

- 实际的情况没有这么简单，任何一种棋都不可能将所有情况列尽，因此，只能模拟人“向前看几步”，然后做出决策，决定自己走哪一步最有利。
- 只能给出几层走法，然后按照一定的估算方法，决定走哪一步棋。
- 在双人完备信息博弈过程中，双方都希望自己能够获胜。因此当一方走步时，**都是选择对自己最有利，而对对方最不利的走法**。

- 假设博弈双方为MAX和MIN。在博弈的每一步，可供他们选择的方案都有很多种。
- 从MAX的观点看，可供自己选择的方案之间是“或”的关系，原因是主动权在自己手里，选择哪个方案完全由自己决定，可供自己选择的方案之间是“或”的关系，而对那些可供MIN选择的方案之间是“与”的关系，这是因为主动权在MIN手中，任何一个方案都可能被MIN选中，MAX必须防止那种对自己最不利的情况出现。

- 下图是把双人博弈过程用图的形式表示出来，这样就可以得到一棵AND-OR树，这种AND-OR树称为博弈树。
- 在博弈树中，那些下一步该MAX走的节点称为MAX节点，而下一步该MIN走的节点称为MIN节点。

- 下图 所示是向前看两步，共四层的博弈树，用□表示MAX，用○表示MIN，端节点上的数字表示它对应的估价函数的值。在MIN处用圆弧连接，用0表示其子节点取估值最小的格局。

- 图中节点处的数字，在端节点是估价函数的值，称它为静态值，在MIN处取最小值，在MAX处取最大值，最后MAX选择箭头方向的走步。

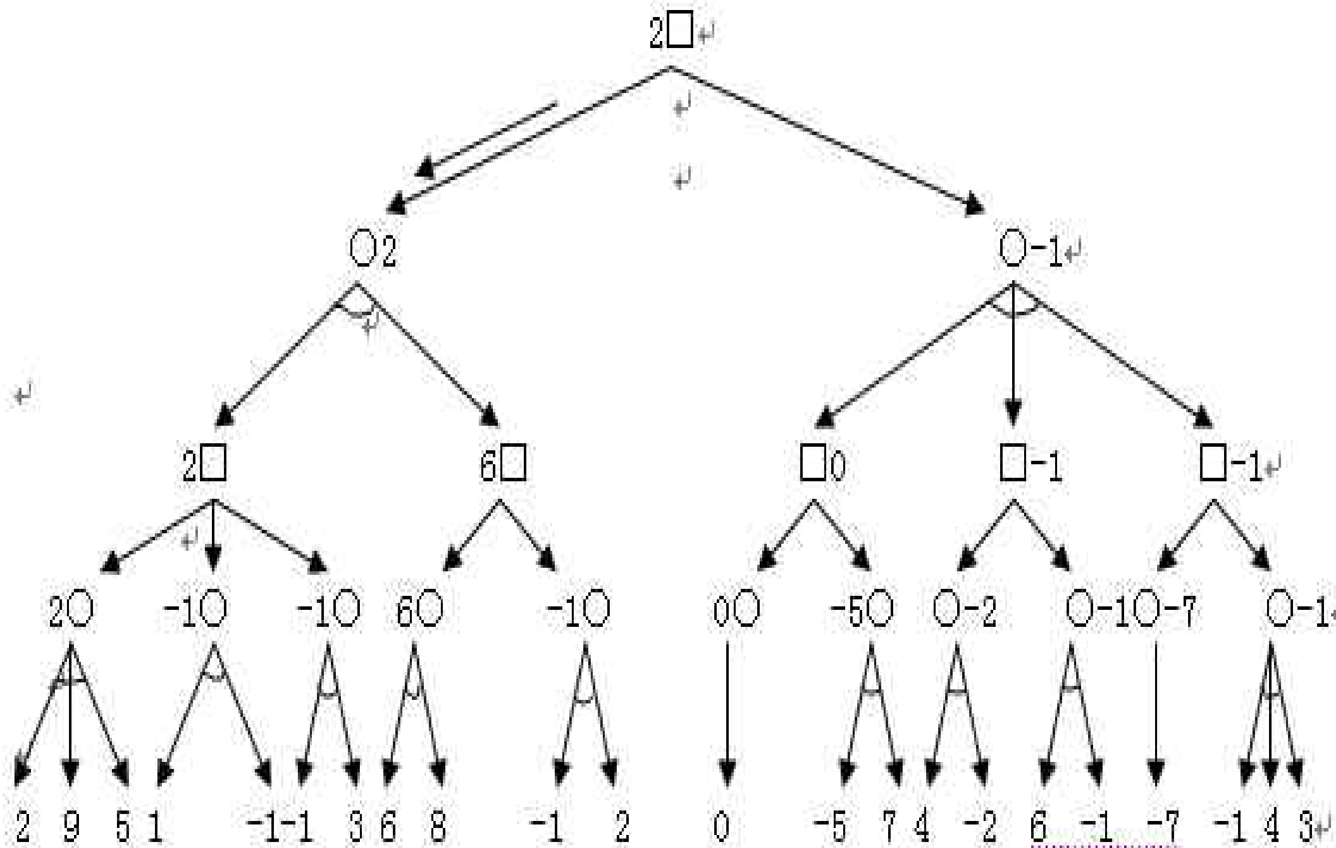


图3-18 四层博弈树

博弈树特点：★

(1) 博弈的初始状态是初始节点；

(2) 博弈树的“与”节点和“或”节点是逐层交替出现的；

(3) 整个博弈过程始终站在某一方的立场上，所以能使自己一方获胜的终局都是本原问题，相应的节点也是可解节点，所有使对方获胜的节点都是不可解节点。

在人工智能中可以采用搜索方法来求解博弈问题，下面就来讨论博弈中两中最基本的搜索方法。

极大极小过程★

- 极大极小过程是考虑双方对弈若干步之后，从可能的走法中选一步相对好的走法来走，即在有限的搜索深度范围内进行求解。
- 需要定义一个静态估价函数f，以便对棋局的态势做出评估。

- 这个函数可以根据棋局的态势特征进行定义。假定对弈双方分别为MAX和MIN，规定：

  有利于MAX方的态势：f(p)取正值

  有利于MIN方的态势：f(p)取负值

  态势均衡的时候：f(p)取零

其中p代表棋局。

- MINMAX基本思想：

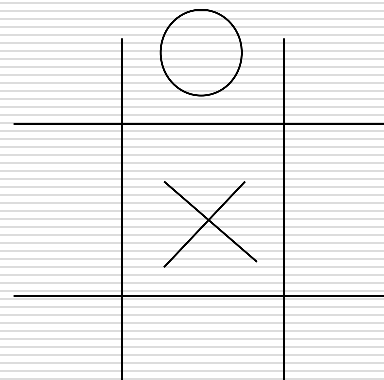
(1)当轮到MIN走步的节点时（取与时），MAX应考虑最坏的情况（即f(p)取极小值）。

(2)当轮到MAX走步的节点时（取或时），MAX应考虑最好的情况（即f(p)取极大值）。

(3)评价往回倒推时，相应于两位棋手的对抗策略，交替使用（1）和（2）两种方法传递倒推值。

- 所以这种方法称为极大极小过程。

- 用一字棋说明极大极小过程，设只进行两层，即每方只走一步。
- 一字棋游戏规则如下：设有一个三行三列的棋盘，如图所示，两个棋手轮流走步，每个棋手走步时往空格上摆一个自己的棋子，谁先使自己的棋子成三子一线为赢。设MAX方的棋子用×标记，MIN方的棋子用〇标记，并规定MAX方先走步。



为了不致于生成太大的博弈树，假设每次仅扩展两层。估价函数定义如下：

设棋局为P，估价函数为e(P)。

- (1)若格局p对任何一方都不是获胜的，则

  e(p) =（所有空格都放上 MAX 的棋子之后三子成一线的总数）-（所有空格都放上 MIN 的棋子后三子成一线的总数）

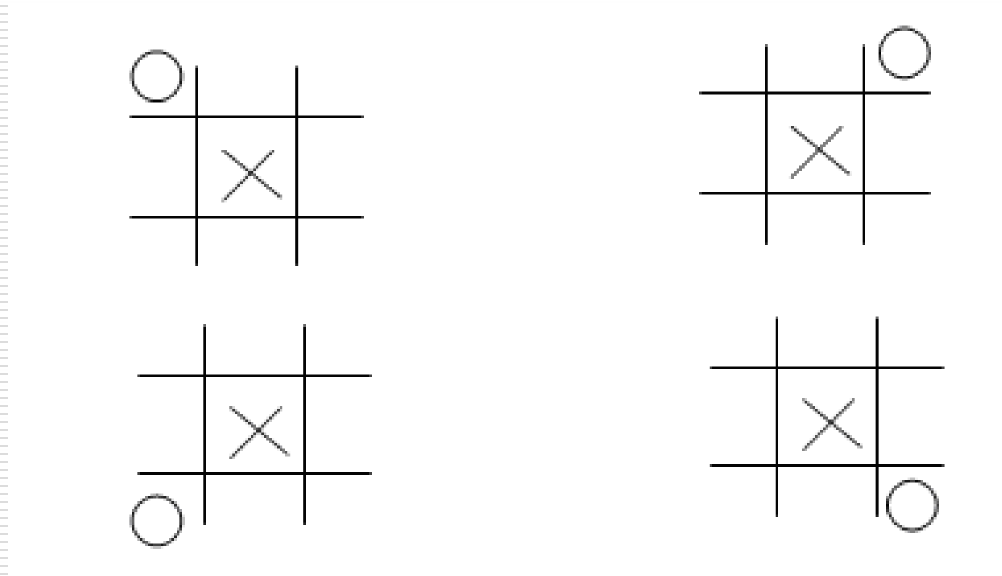
- (2)若p是MAX获胜，则

  e(p) = +∞

- （3） 若p是MIN获胜，则

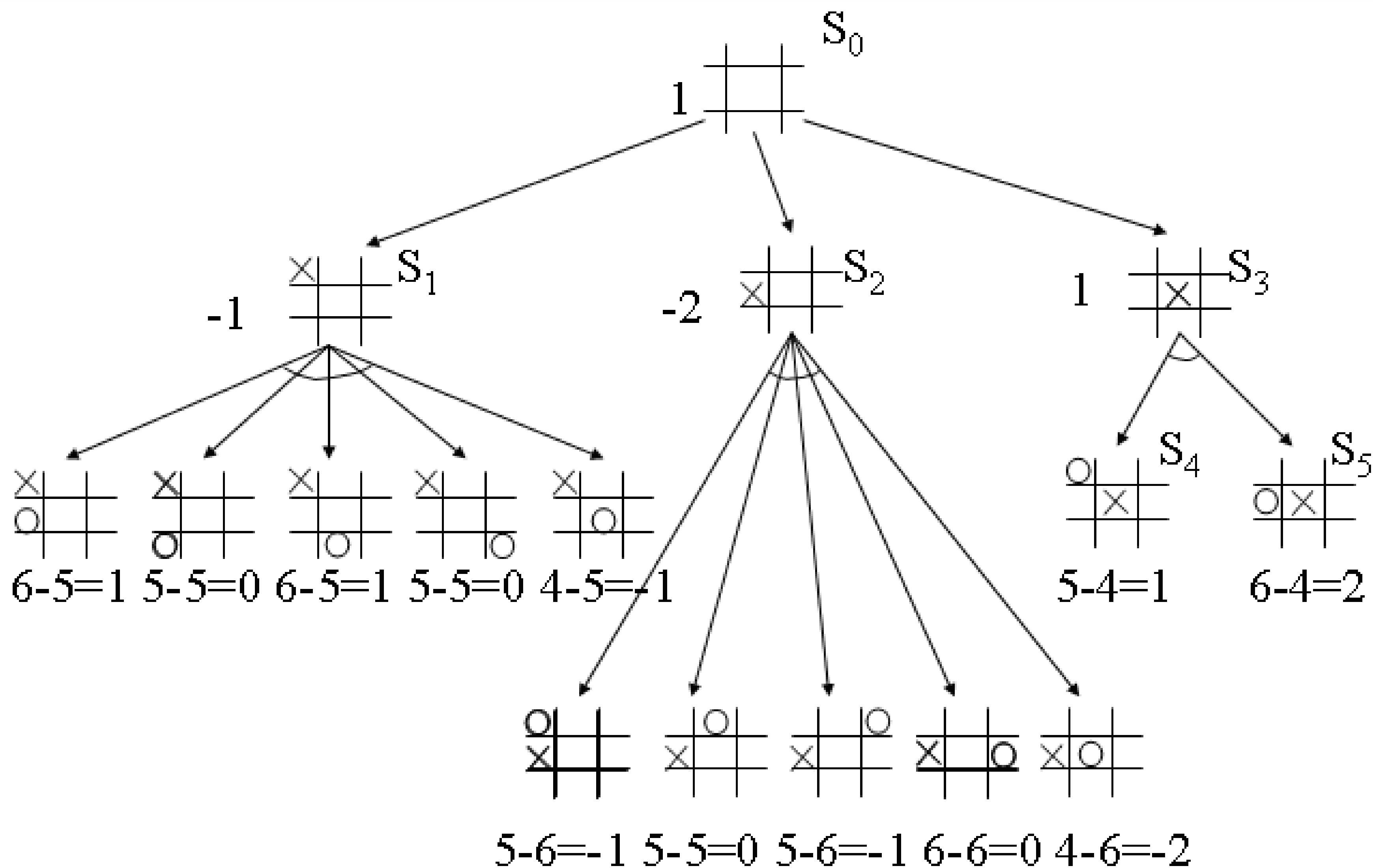
  e(p) = -∞

- 若p为下图所示，且



- e(P)=e(+P)-e(-P)=5-4=1

- 假设由MAX先走棋，且我们站在MAX立场上。下图给出了MAX的第一着走棋生成的博弈树。图中节点旁的数字分别表示相应节点的静态估值或倒推值。由图可以看出，对于MAX来说最好的一着棋是S3，因为S3比S1和S2有较大的估值。



- 下图 所示是向前看两步，共四层的博弈树，用□表示MAX，用○表示MIN，端节点上的数字表示它对应的估价函数的值。在MIN处用圆弧连接，用0表示其子节点取估值最小的格局。

- 图中节点处的数字，在端节点是估价函数的值，称它为静态值，在MIN处取最小值，在MAX处取最大值，最后MAX选择箭头方向的走步。

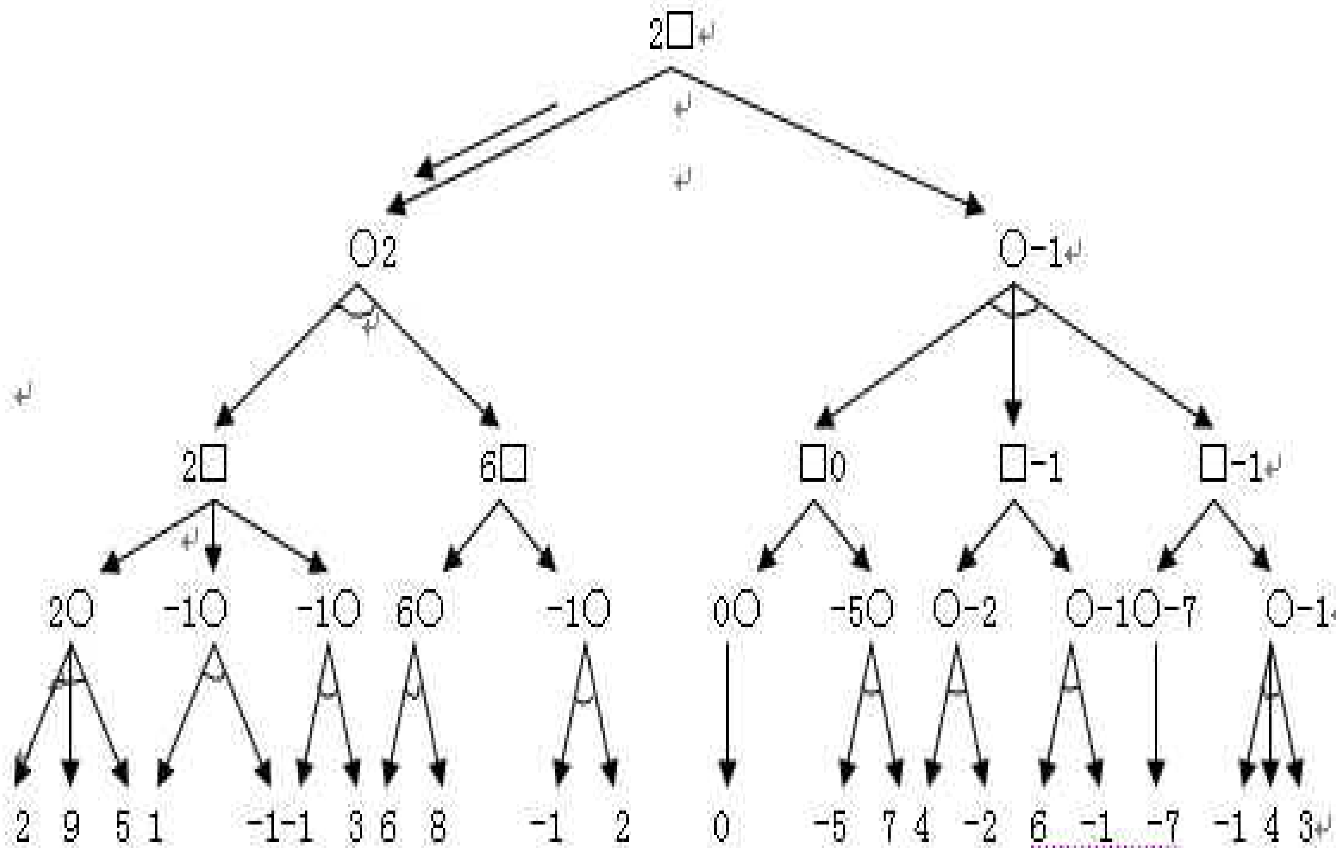
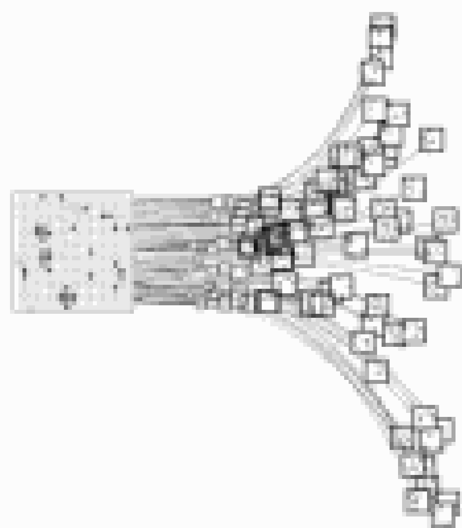


图3-18 四层博弈树

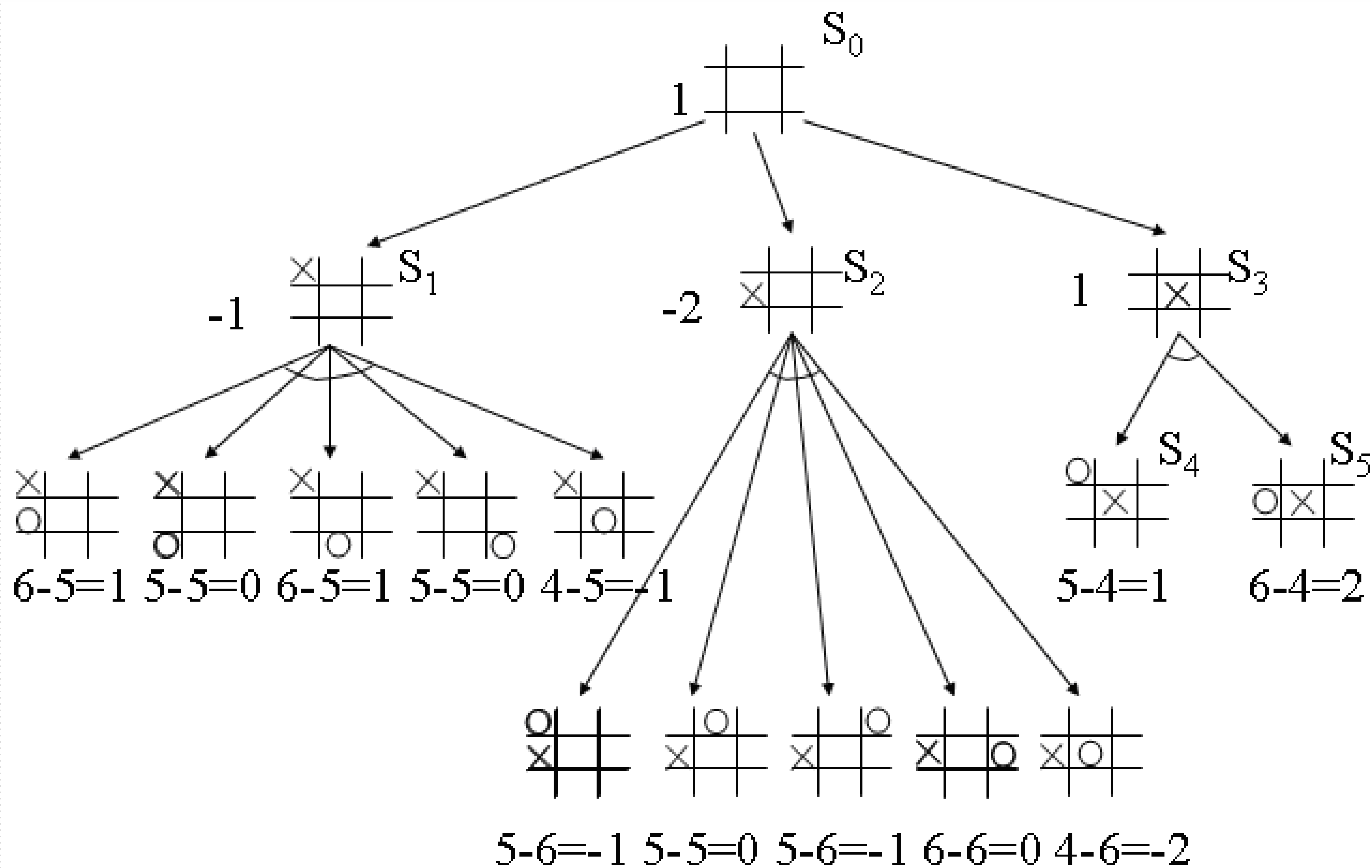


博弈树面临的最大问题—组合爆炸

α－β 过程

- 上面讨论的极大极小过程先生成一棵博弈搜索树，而且会生成规定深度内的所有节点，然后再进行估值的倒推计算，这样使得生成博弈树和估计值的倒推计算两个过程完全分离，因此搜索效率较低。
- 如果能边生成博弈树，边进行估值的计算，则可能不必生成规定深度内的所有节点，以减少搜索的次数，这就是下面要讨论的 α－β 过程。

- 考虑：如果边生成博弈树，边进行估值的计算会带来什么好处。



- α－β 过程就是把生成后继和倒推值估计结合起来，及时剪掉一些无用分支，以此来提高算法的效率。
- 下面仍然用一字棋进行说明。现将原图左边所示的一部分重画在中。

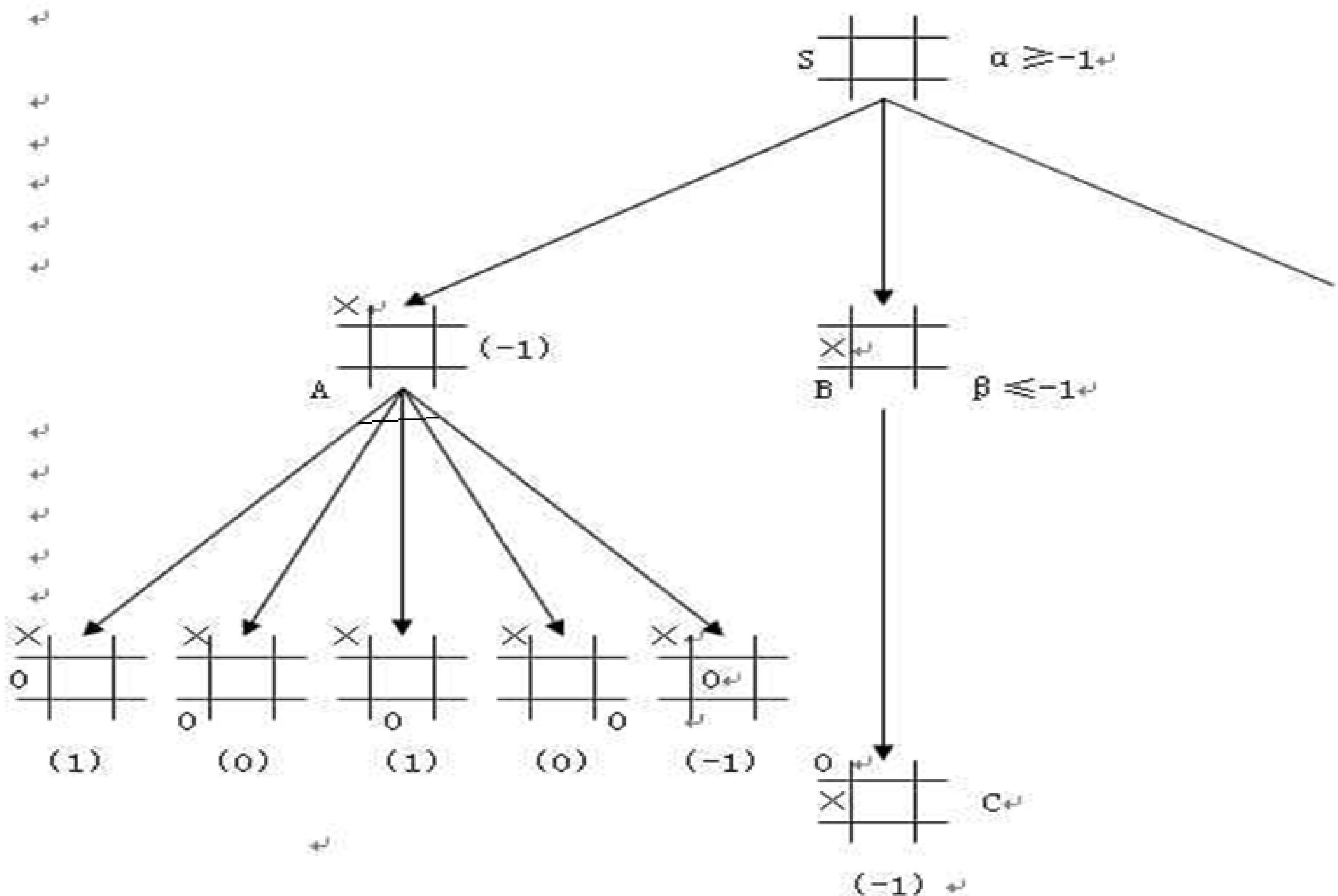


图 3.20 一字棋博弈的 α－β 过程

- 前面的过程实际上类似于宽度优先搜索，将每层格局均生成，现在用深度优先搜索来处理。
- 比如在节点A处，若已生成5个子节点，并且A处的倒推值等于-1，我们将此下界叫做MAX节点的α值，即α≥-1。

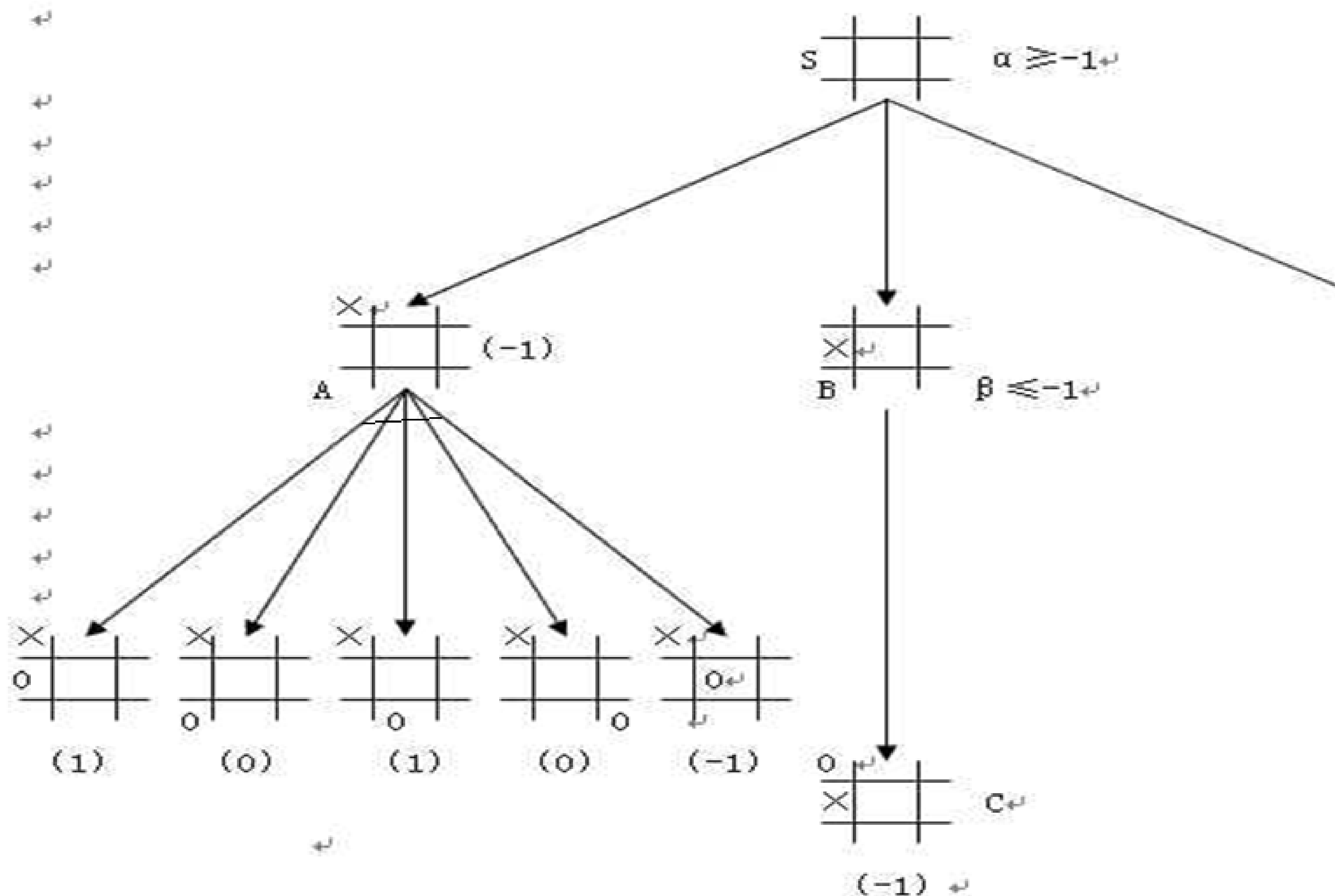


图 3.20 一字棋博弈的α-β过程

- 现在轮到节点B，产生它的第一后继节点C，C的静态值为-1，可知B处的倒推值≤-1，此为上界MIN节点的β值，即B处β≤-1，这样B节点最终的倒推值可能小于-1，但绝不可能大于-1，
- 因此，B节点的其他后继节点的静态值不必计算，自然不必再生成，反正B决不会比A好．所以通过倒推值的比较．就可以减少搜索的工作量

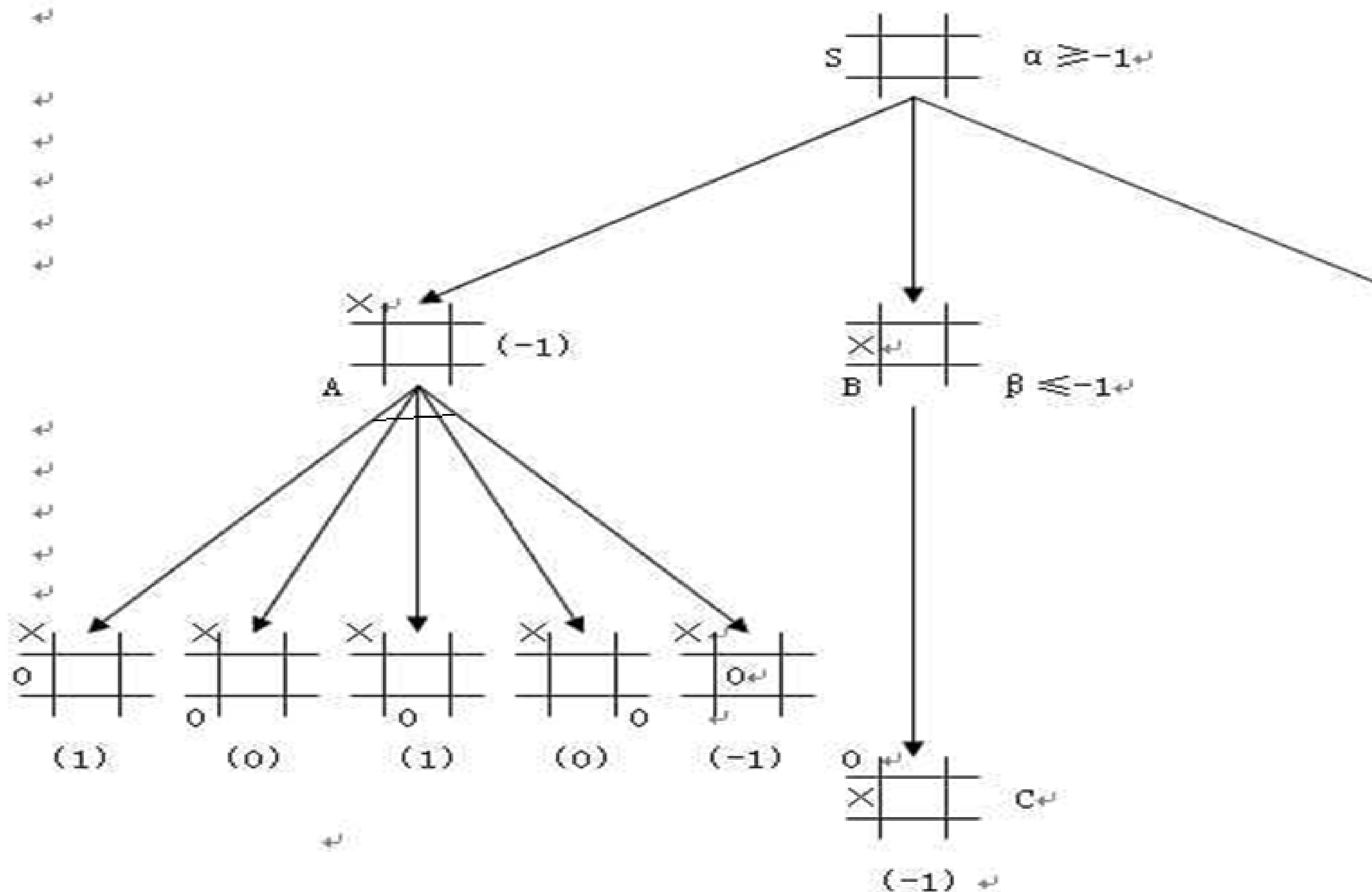
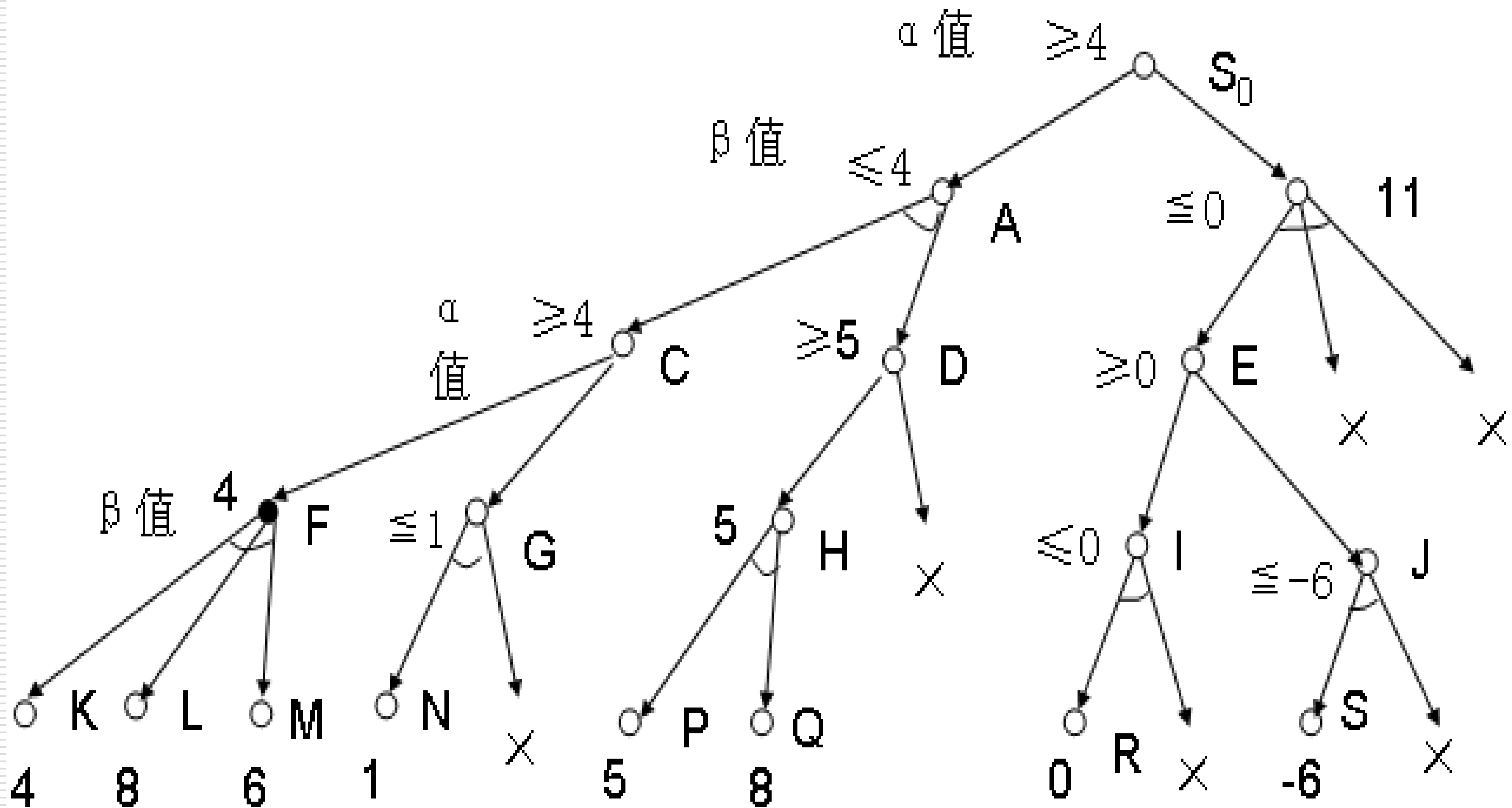


图 3.20 一字棋博弈的α-β过程

- 上图表示了β值小于等于父节点的α值时的情况，实际上当某个MIN节点的β值不大于它的先辈的MAX节点（不一定是父节点）的α值时，则MIN节点就可以终止向下搜索。
- 同样，当某个节点的α值大于等于它的先辈MIN节点的β值时，则该MAX节点就可以终止向下搜索。

再看一个例子，如下图所示。其中最下面一层端节点旁边的数字是假设的估值。



- 通过上面的讨论可以看出，α－β过程首先使搜索树的某一部分达到最大深度，这时计算出某些MAX节点的α值，或者是某些MIN节点的β值。
- 随着搜索的继续，不断修改个别节点的α或β值。对任一节点，当其某一后继节点的最终值给定时，就可以确定该节点的α或β值。

- 当该节点的其他后继节点的最终值给定时，就可以对该节点的α或β值进行修正。
- 注意α、β值修改有如下规律：(1)MAX节点的α值永不下降；(2)MIN节点的β值永不增加。

因此可以利用上述规律进行剪枝，一般可以停止对某个节点搜索，即剪枝的规则表述如下：

- (1) 若任何MIN节点的β值小于或等于任何它的先辈MAX节点的α值，则可停止该MIN节点以下的搜索，然后这个MIN节点的最终倒推值即为它已得到的β值。

  该值与真正的极大极小值的搜索结果的倒推值可能不相同，但是对开始节点而言，倒推值是相同的，使用它选择的走步也是相同的。

- (2) 若任何MAX节点的α值大于或等于它的MIN先辈节点的β值，则可以停止该MAX节点以下的搜索，然后这个MAX节点处的倒推值即为它已得到的α值。

- 当满足规则1而减少了搜索时，进行了α剪枝；而当满足规则2而减少了搜索时，进行了β剪枝。
- 保存α和β值，并且一旦可能就进行剪枝的整个过程通常称为α-β过程，当初始节点的全体后继节点的最终倒推值全部给出时，上述过程便结束。
- 在搜索深度相同的条件下，采用这个过程所获得的走步总跟简单的极大极小过程的结果是相同的，区别只在于α-β过程通常只用少得多的搜索便可以找到一个理想的走步。

设置

课堂测试

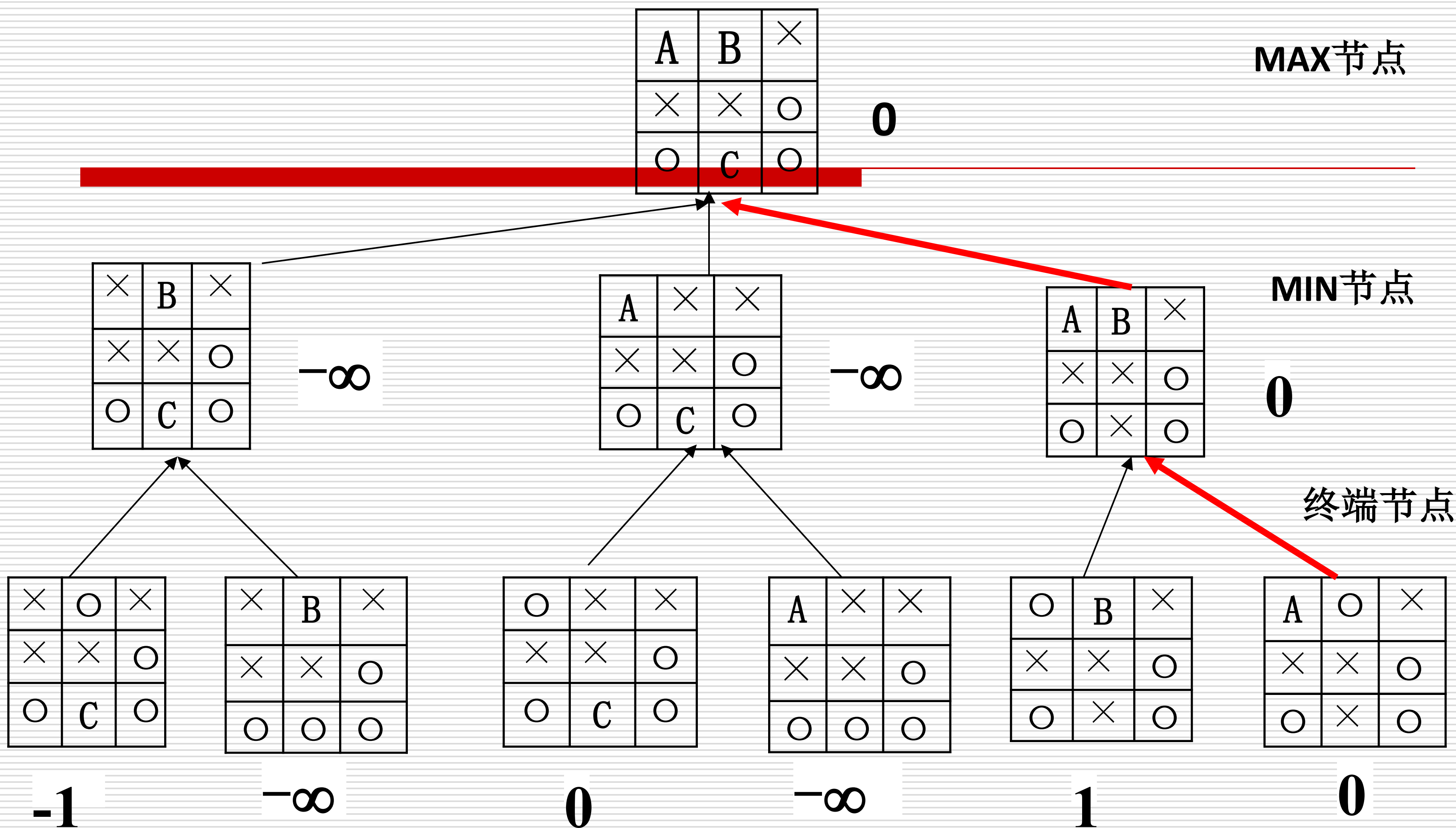
有一个一字棋残局如下图所示，○和×在结束前有三步棋可以走，而且下一步走棋者是× 。这时存在着三个空格A，B，C，请站在×的立场画出相应的二层博弈树，并判断应该把棋子放到哪一格内。

A	B	×
×	×	○
○	C	○

正常使用主观题需2.0以上版本雨课堂

作答

解析



对于棋盘残局中的×来说，最好的选择，是将×放在C的位置上，这时可以导致平局局面。