Artificial Intelligence
人工智能

第3章 搜索技术

第3章 搜索技术

3.1 概述

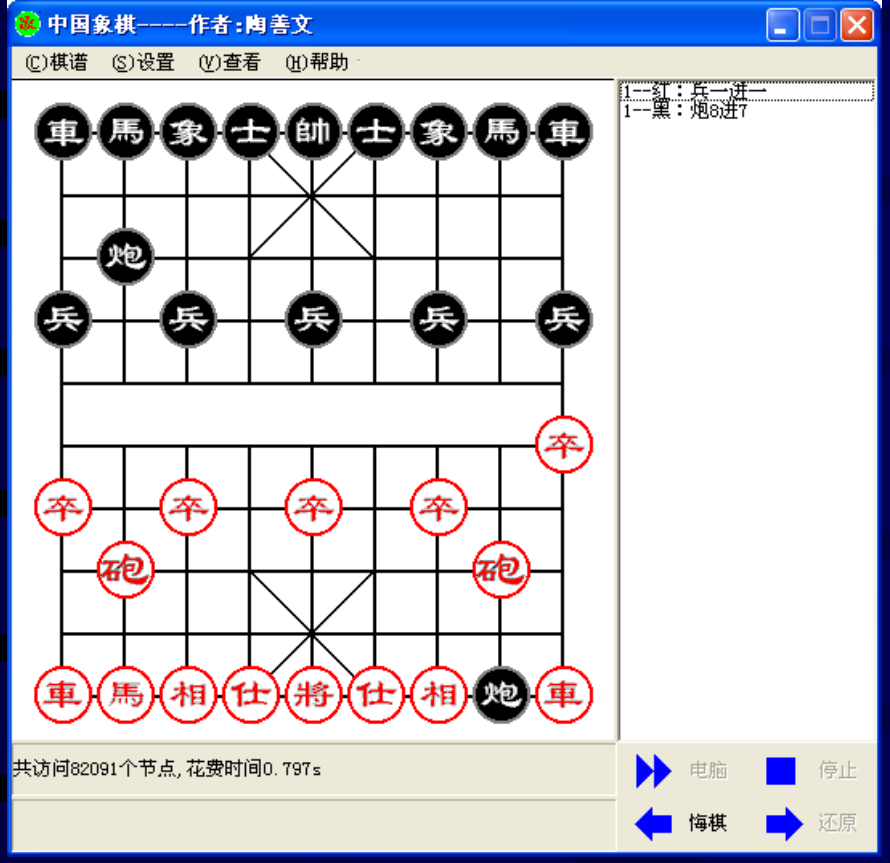
3.2 盲目搜索方法

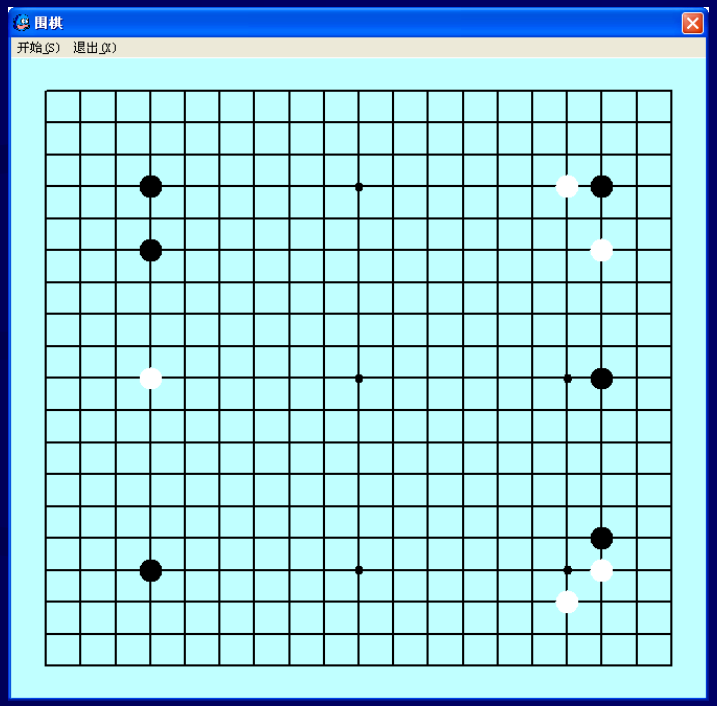
3.3 启发式搜索

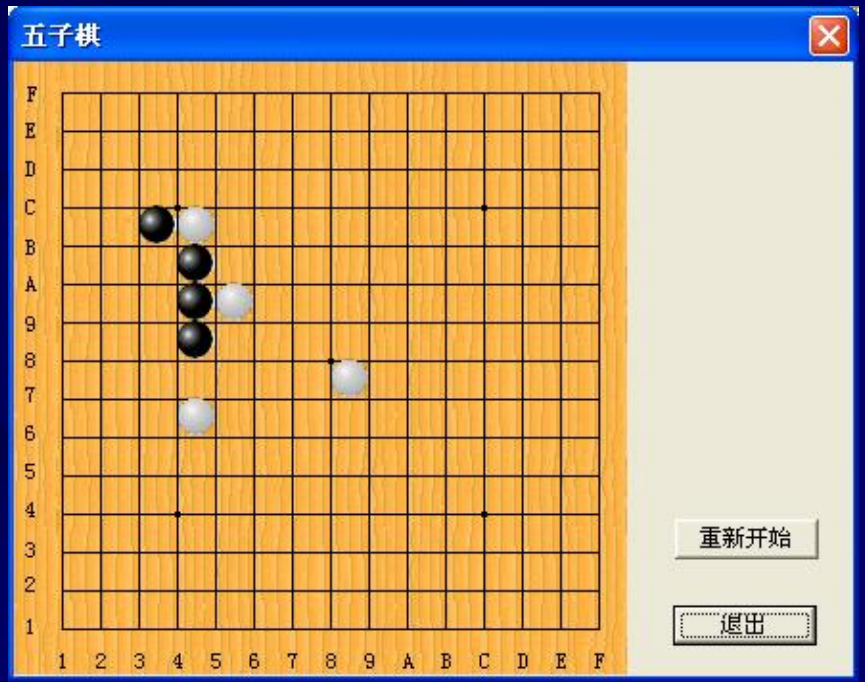
3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析


中国象棋----作者:陶善文
(C)棋谱 (S)设置 (V)查看 (H)帮助
1--红：兵一进一
1--黑：炮8进7
車 馬 象 士 帥 士 象 馬 車
炮 炮
兵 兵 兵 兵 兵
卒 卒 卒 卒 卒
炮 炮
車 馬 相 仕 將 仕 相 炮 車
共访问82091个节点，花费时间0.797s
电脑
停止
悔棋
还原


围棋
开始(S) 退出(X)


五子棋
F E D C B A 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 A B C D E F
重新开始
退出

3.1 概述

- 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）
- 求解问题包括：
  - 目标表示
  - 搜索
  - 执行

- 给定问题就是确定该问题的基本信息：

（1）初始状态集合：定义了问题的初始状态。

（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。

- 搜索问题包括：
  - 搜索什么（目标）
  - 在哪里搜索（搜索空间）

- 搜索分成：
  - 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

- 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。

- 按表示方式分
  - 状态空间搜索
  - 与或树搜索

用状态空间法来求解问题所进行的搜索

用问题规约方法来求解问题时所进行的搜索

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？
- 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？
- 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？

- 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

根节点首先扩展，接着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

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。

- 这种类型的遍历称为“确定性”的，也就是盲目搜索。

搜索控制策略（1）

- 搜索控制策略
  - 不可撤回的控制策略
  - 试探性控制策略
    - ✓回溯型
    - ✓图搜索

搜索控制策略（2）

不可撤回的控制策略

例：八数码问题

评价函数：f：　　（规定：评价函数非增）

2	8	3
1	6	4
7		5

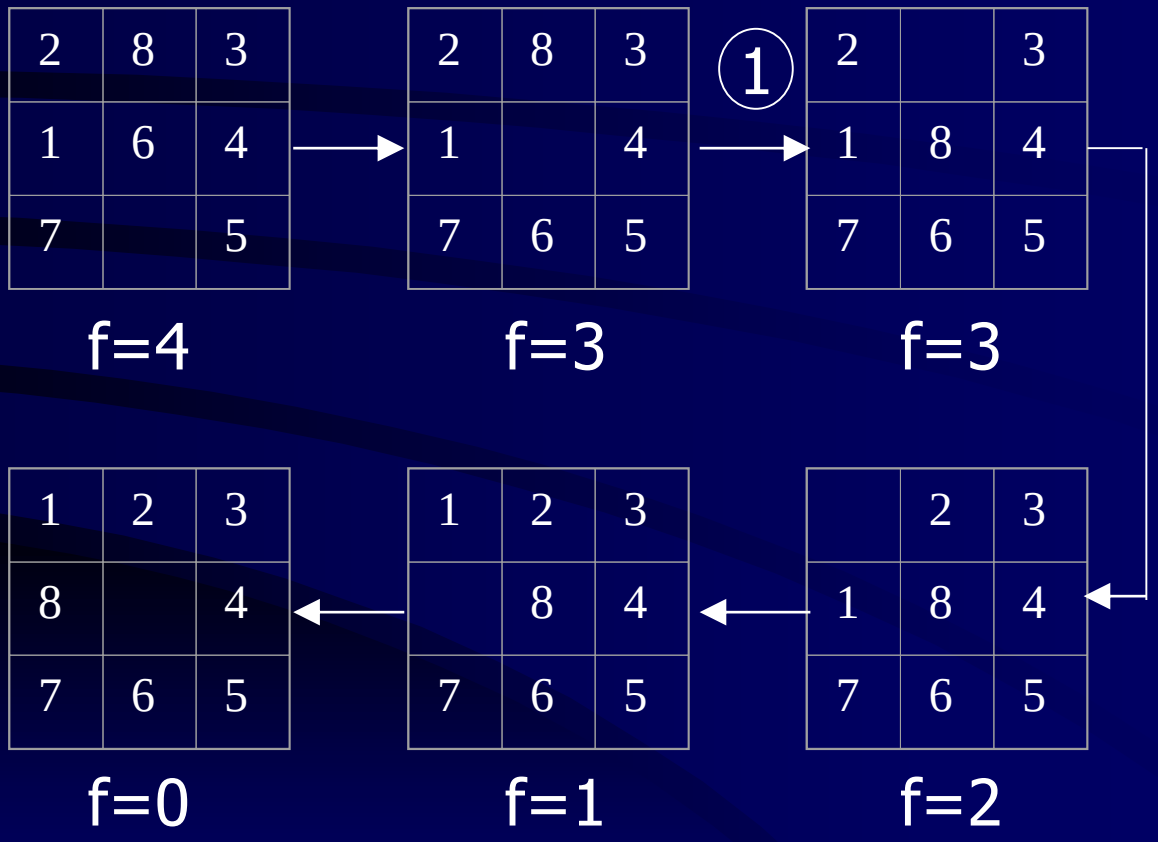
与

1	2	3
8		4
7	6	5

的差异为4

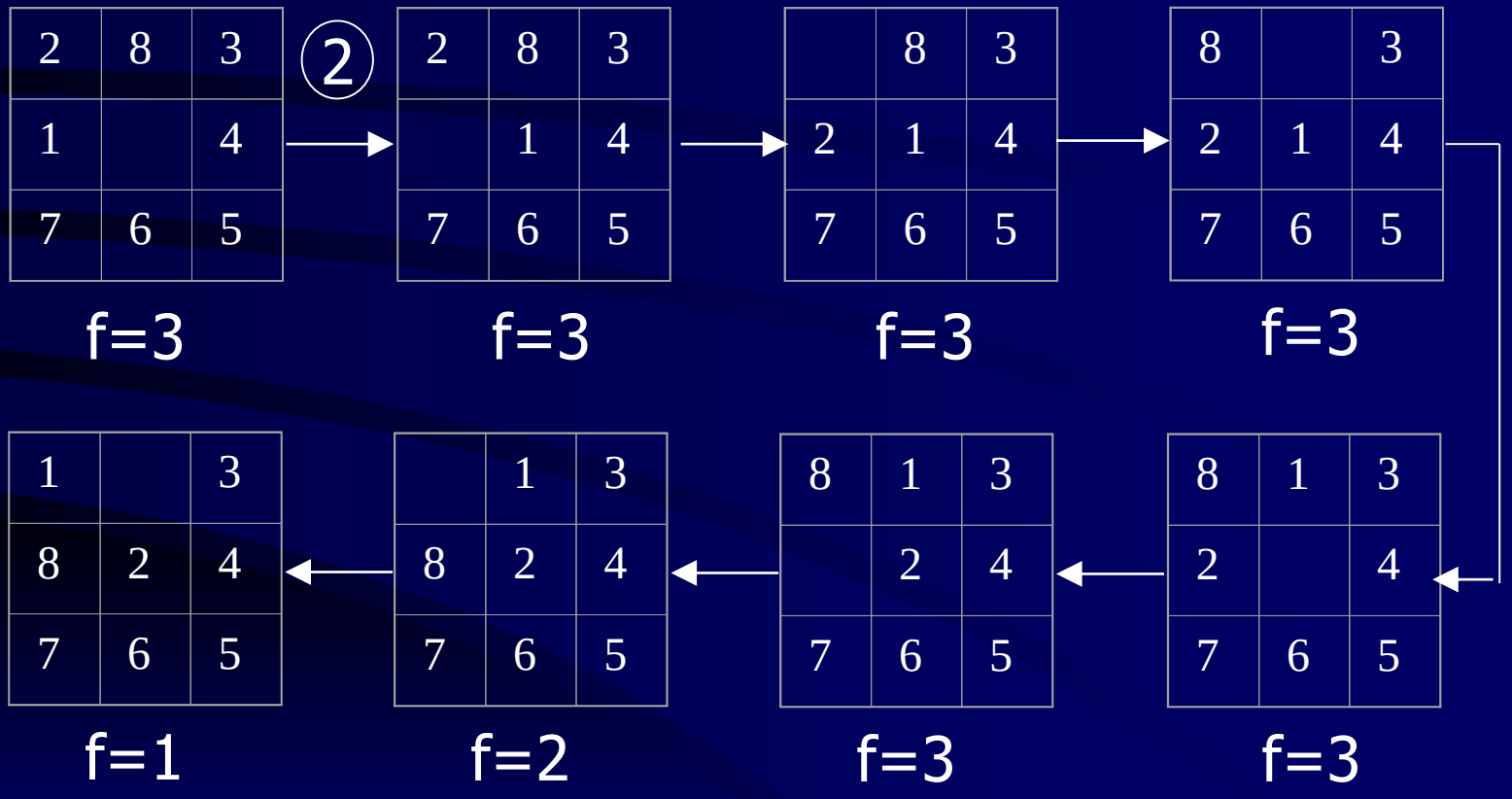
搜索控制策略（3）

不可撤回的控制策略



搜索控制策略（4）

不可撤回的控制策略



搜索控制策略（5）

不可撤回的控制策略

可能无解

1	2	5
	8	4
7	6	3

f=2

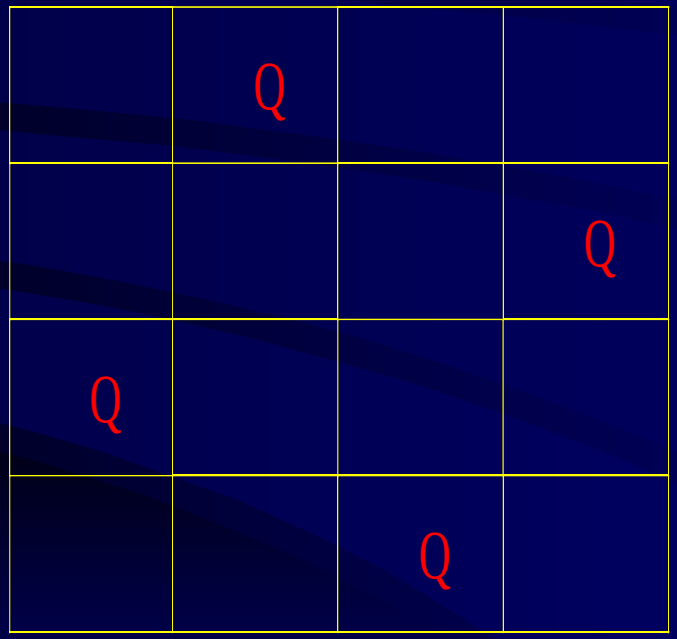
1	2	3
	8	4
7	6	5

目标

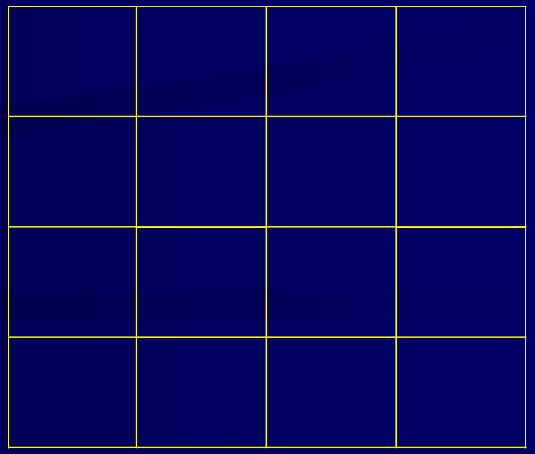
搜索控制策略（6）

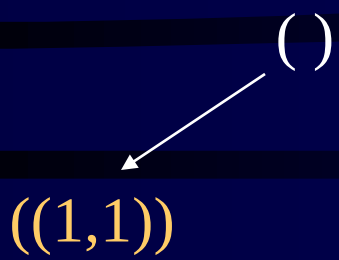
回溯策略

例：四皇后问题

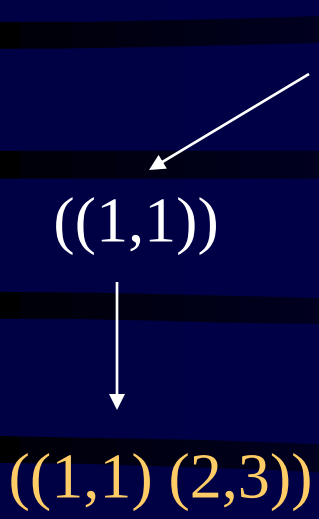


( )

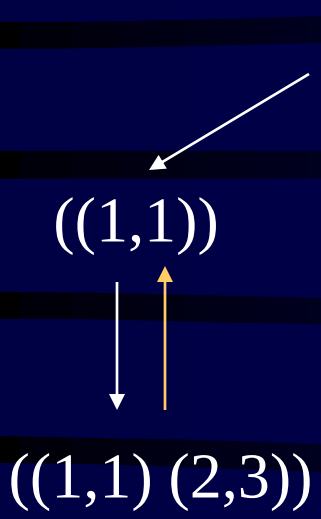




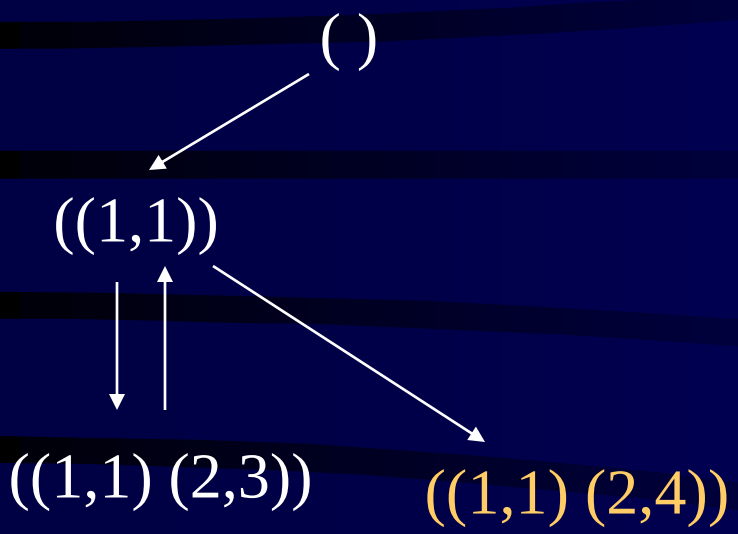
Q			



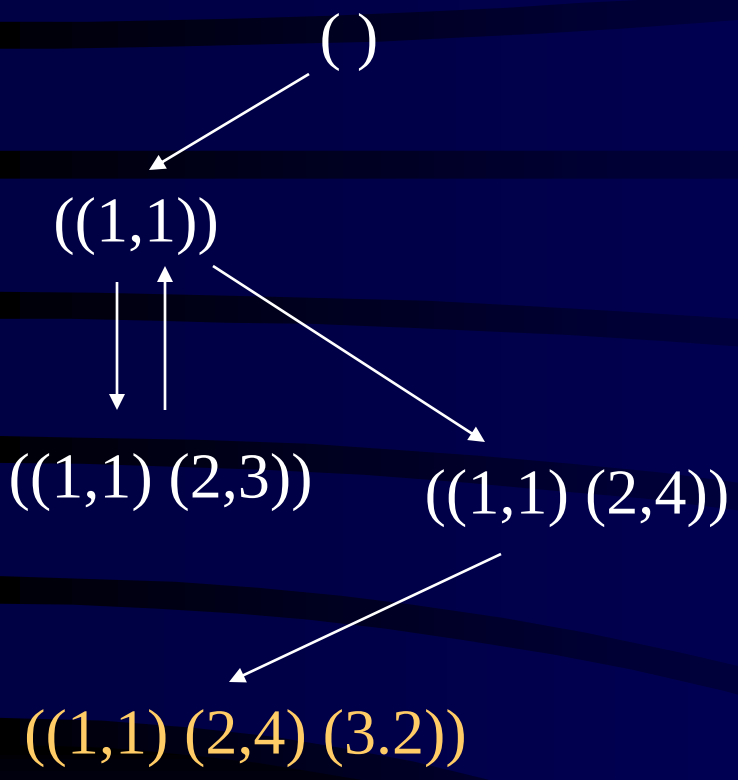
Q			
		Q	



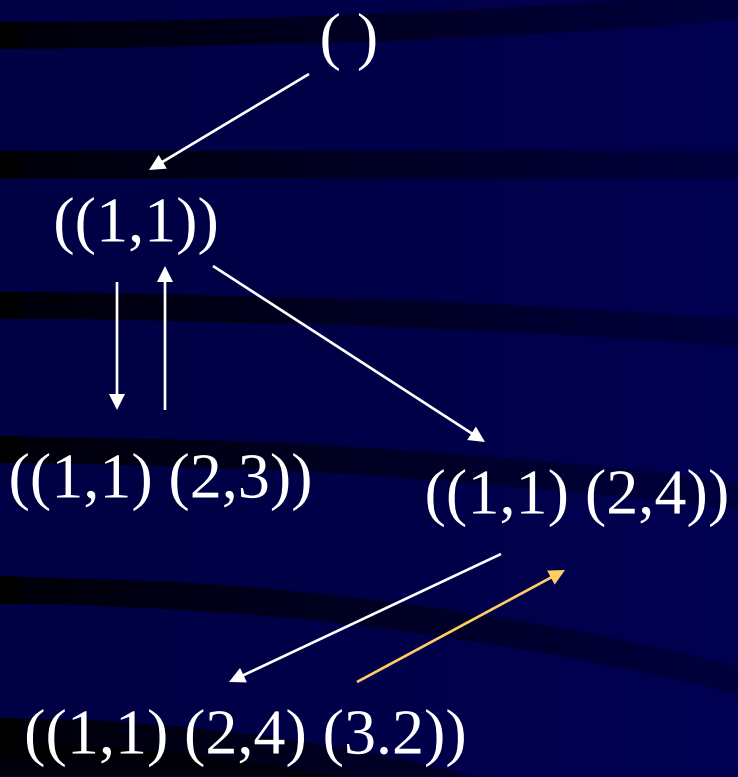
Q			



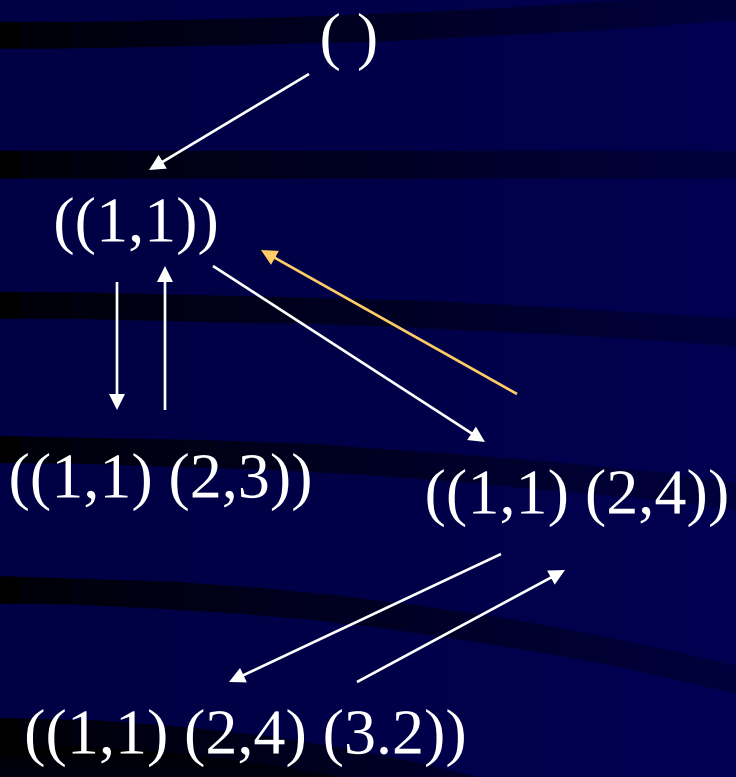
Q			
			Q



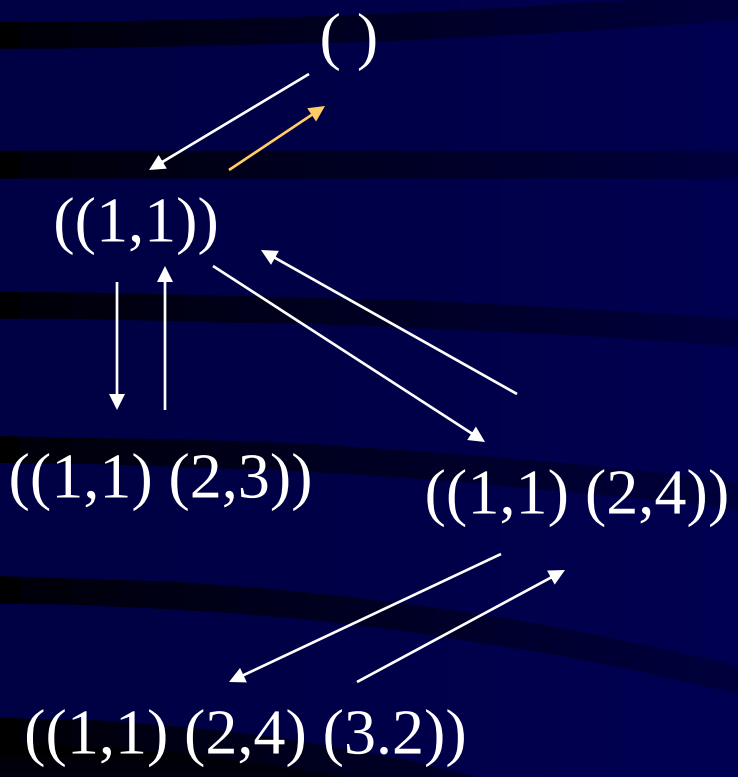
Q			
			Q
	Q		

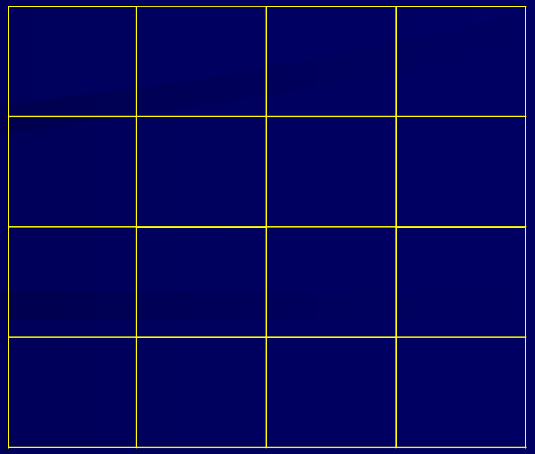


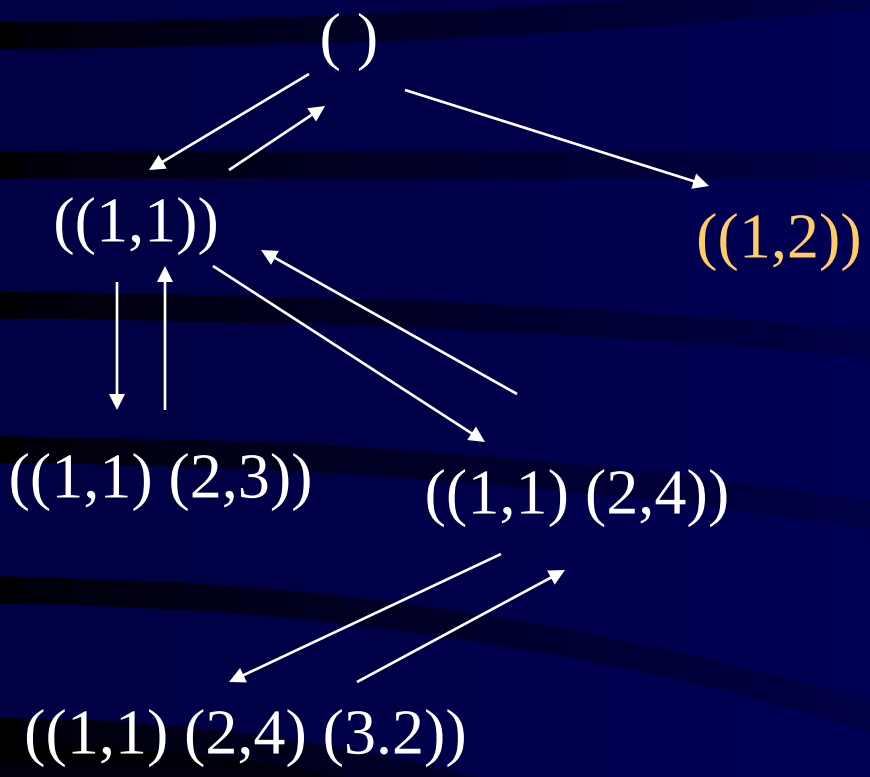
Q			
			Q



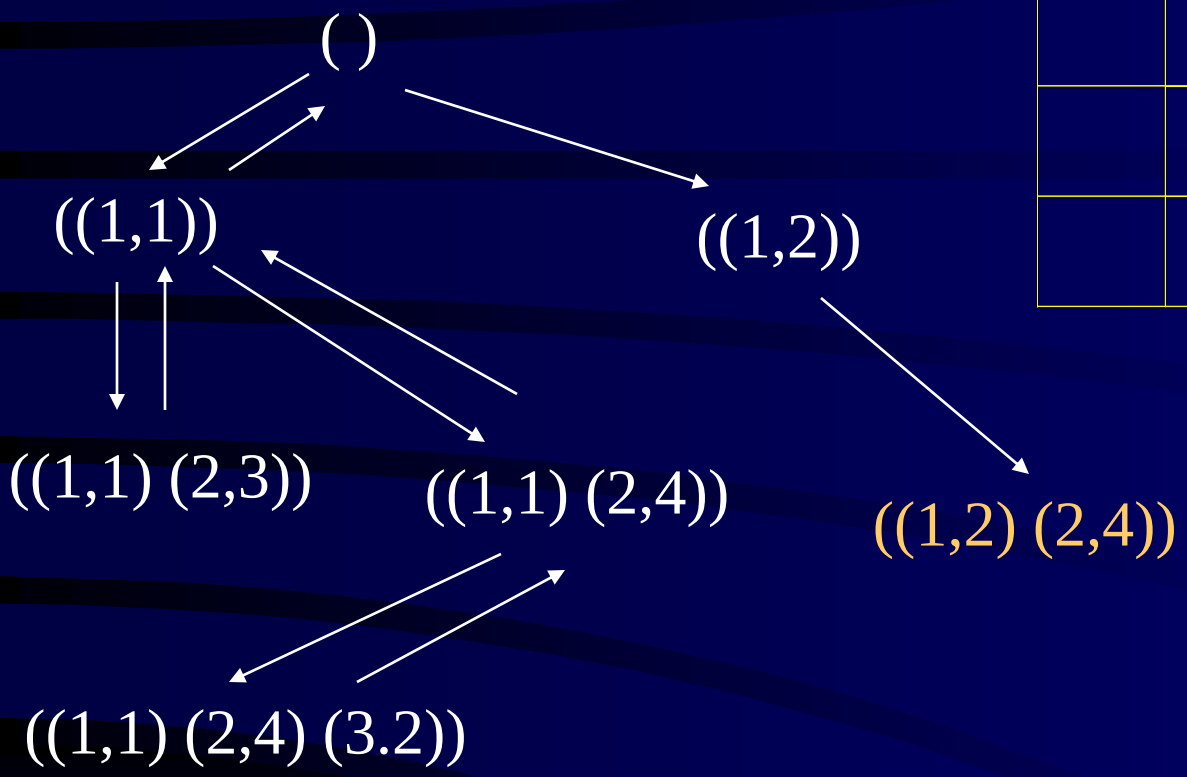
Q			


( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))

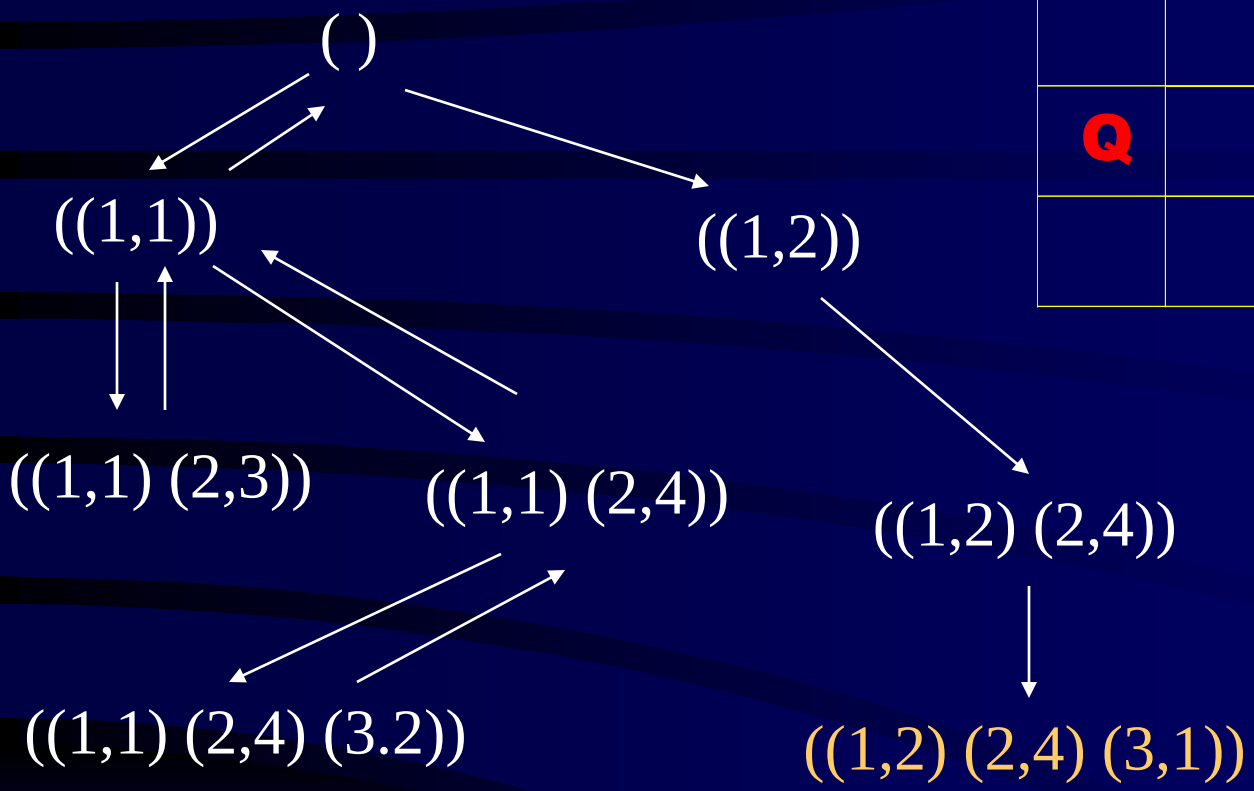




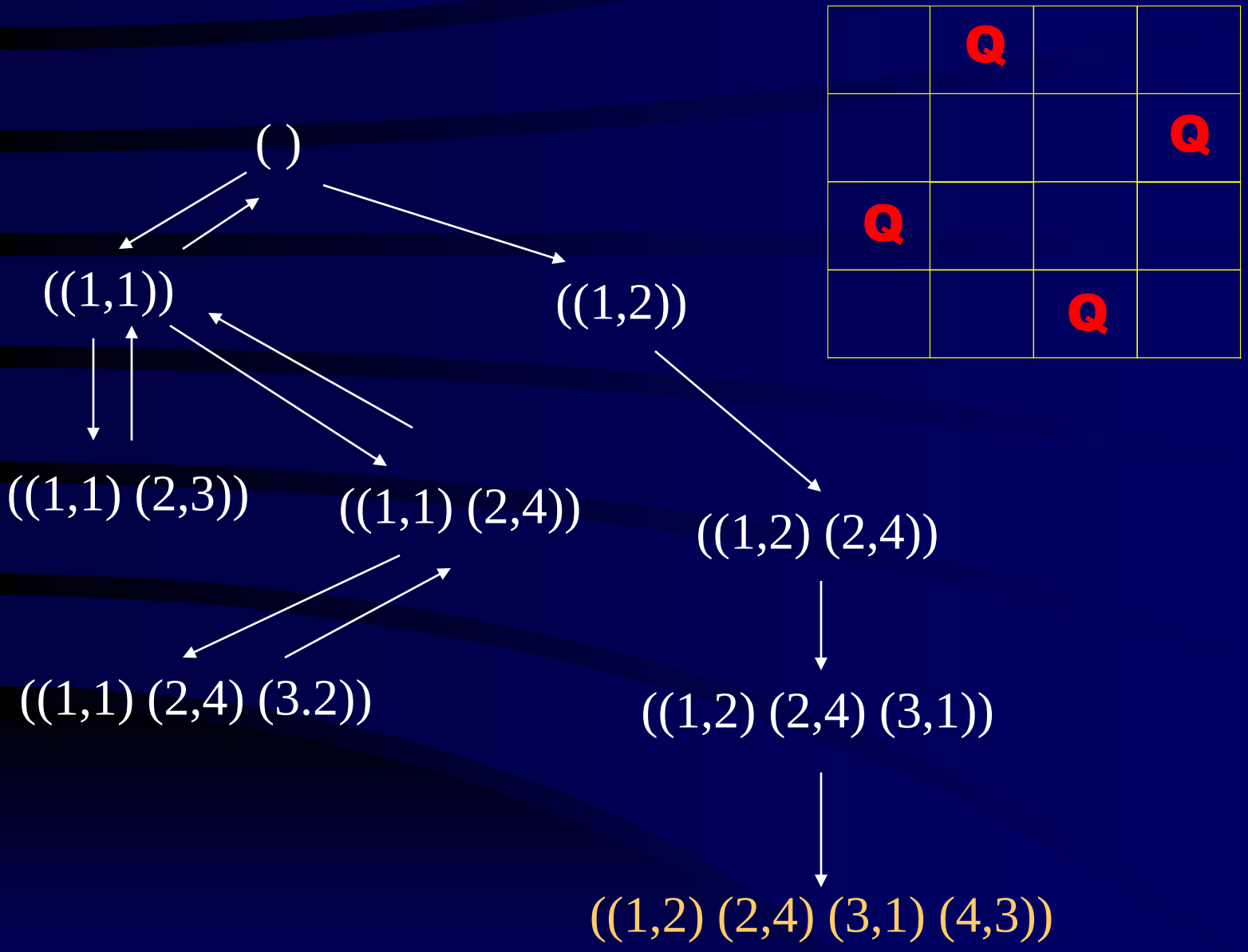
	Q		



	Q		
			Q



	Q		
			Q
Q			


( )
((1,1))
((1,2))
((1,1) (2,3))
((1,1) (2,4))
((1,2) (2,4))
((1,1) (2,4) (3.2))
((1,2) (2,4) (3,1))
((1,2) (2,4) (3,1) (4,3))
Q
Q
Q
Q

第3章 搜索技术

3.1 概述

3.2 盲目搜索方法

3.3 启发式搜索

3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析

3.2 盲目搜索方法

3.2.1 宽度优先搜索

3.2.2 深度优先搜索

3.2.3 迭代加深搜索

3.2 盲目搜索方法

Procedure Generate & Test（生成与测试）

Begin

Repeat

生成一个新的状态，称为当前状态；

Until 当前状态=目标；

End.

上述算法在每次Repeat-Until循环中都生成一个新的状态，并且只有当新的状态等于目标状态的时候才退出。在该算法中最重要的部分是新状态的生成。

如果生成的新状态不可使用，则该算法应该终止。

- **对于给定问题，如何生成新状态呢？**

定义一个四元组，以此来表示状态空间：

{ nodes , arc , goal , current }

nodes 当前搜索空间中现有状态的集合

arc 表示可应用与当前状态的操作符，把当前状态转换为另一个状态

goal表示需要到达的状态

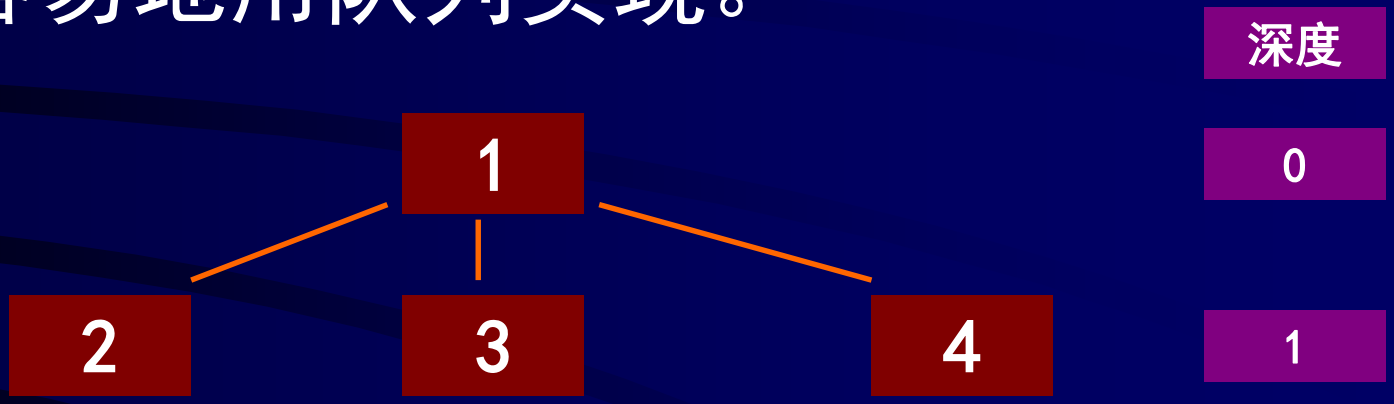
current 现在生成的用于和目标状态比较的状态

- 典型的用于搜索的状态生成方法：

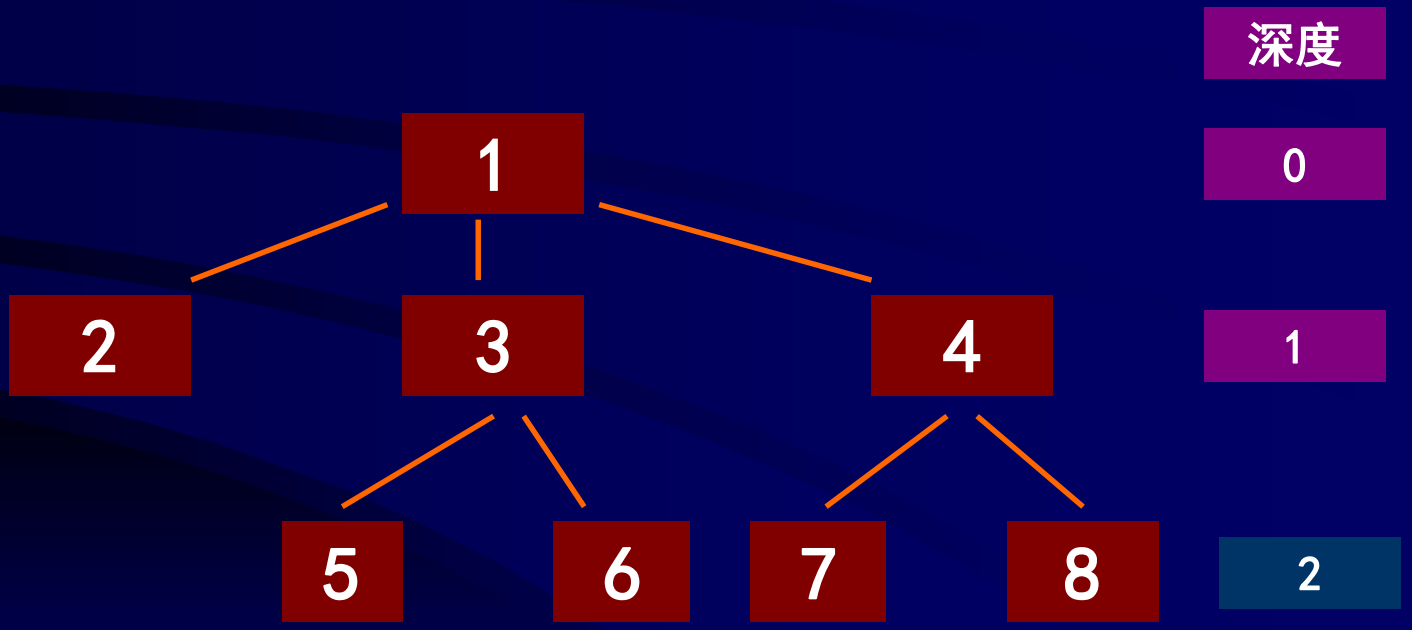
宽度优先；深度优先

3.2.1 宽度优先搜索

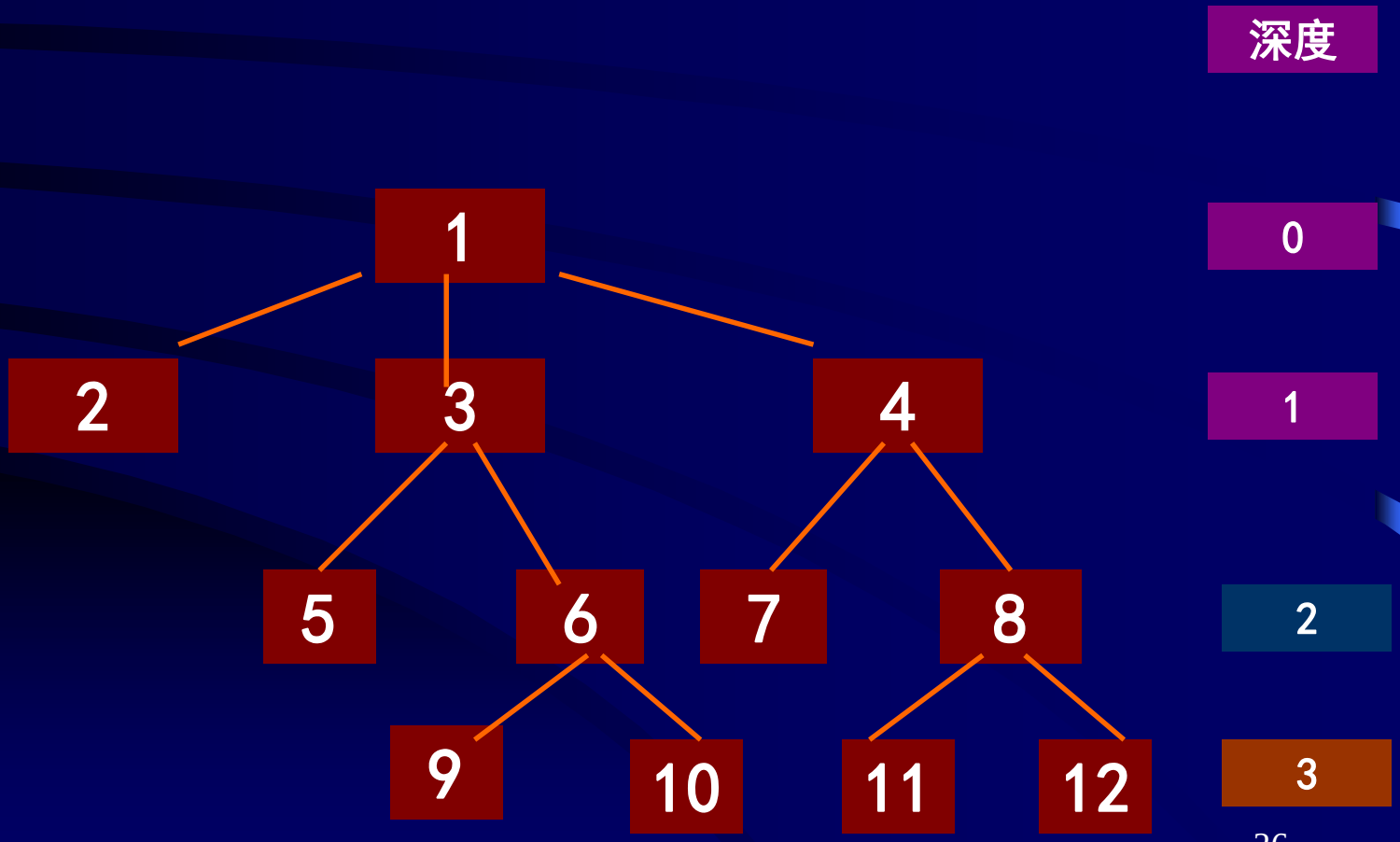
- 沿着树的宽度遍历树的节点，它从深度为0的层开始，直到最深的层次。它可以很容易地用队列实现。

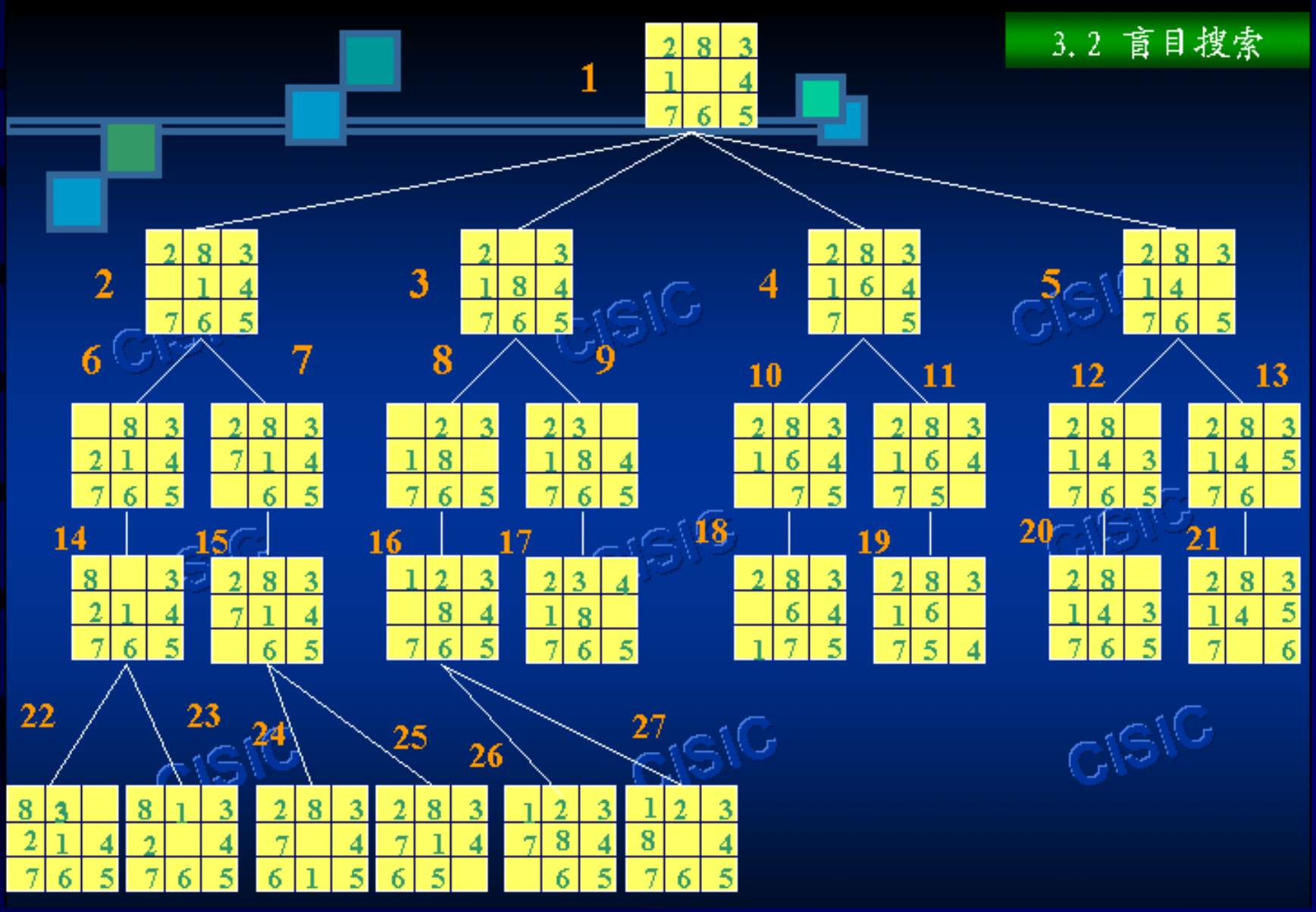


3.2.1 宽度优先搜索



3.2.1 宽度优先搜索





宽度优先搜索

采用队列结构，宽度优先算法可以表示如下：

```
Procedure Breadth-first-search
  Begin
      把初始节点放入队列;
   Repeat
      取得队列最前面的元素为current;
     If current=goal
       成功返回并结束;
      Else do
       Begin
            如果current有子女，则current的子女
            以任意次序添加到队列的尾部;
       End
    Until 队列为空
  End
```

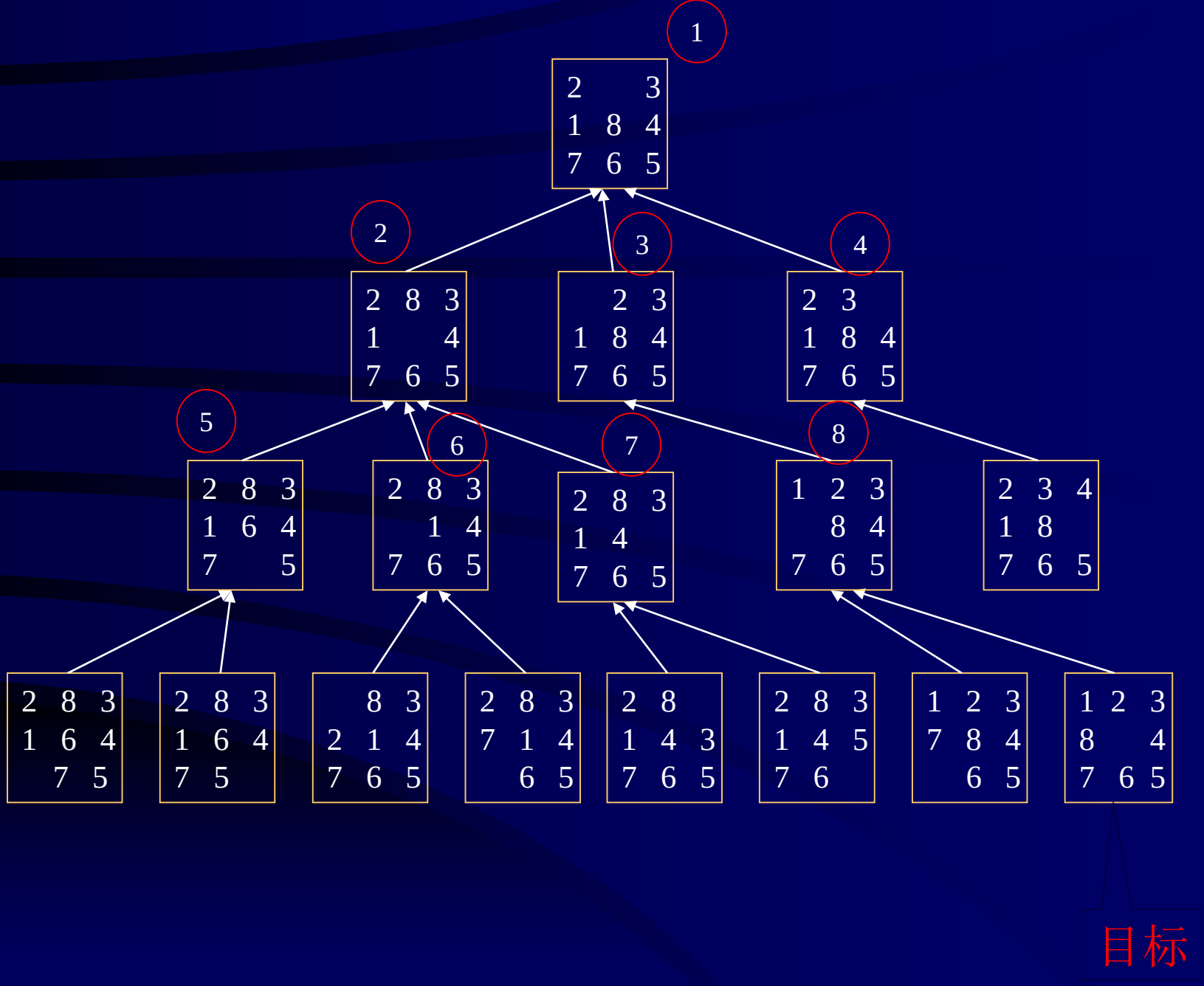
- 宽度优先搜索算法原理：
- 如果当前的节点不是目标节点，则把当点节点的子孙以任意顺序增加到队列的后面，并把队列的前端元素定义为current。
- 如果目标发现，则算法终止。

搜索最优策略的比较

深　　度	节点数	时间	空间
0	1	1 μs	100 B
2	111	1 s	11 KB
4	11111	11 s	1 MB
6	10^6	18 min	111 MB
8	10^8	31 h	11 GB
10	10^{10}	128 d	1 TB
12	10^{12}	35 y	111 TB
14	10^{14}	3500 y	11111 TB

- 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。

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。


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

3.2.2 深度优先搜索

深度优先搜索生成节点并与目标节点进行比较是沿着树的最大深度方向进行的，只有当上次访问的节点不是目标节点，而且没有其他节点可以生成的时候，才转到上次访问节点的父节点。

转移到父节点后，该算法会搜索父节点的其他的子节点。

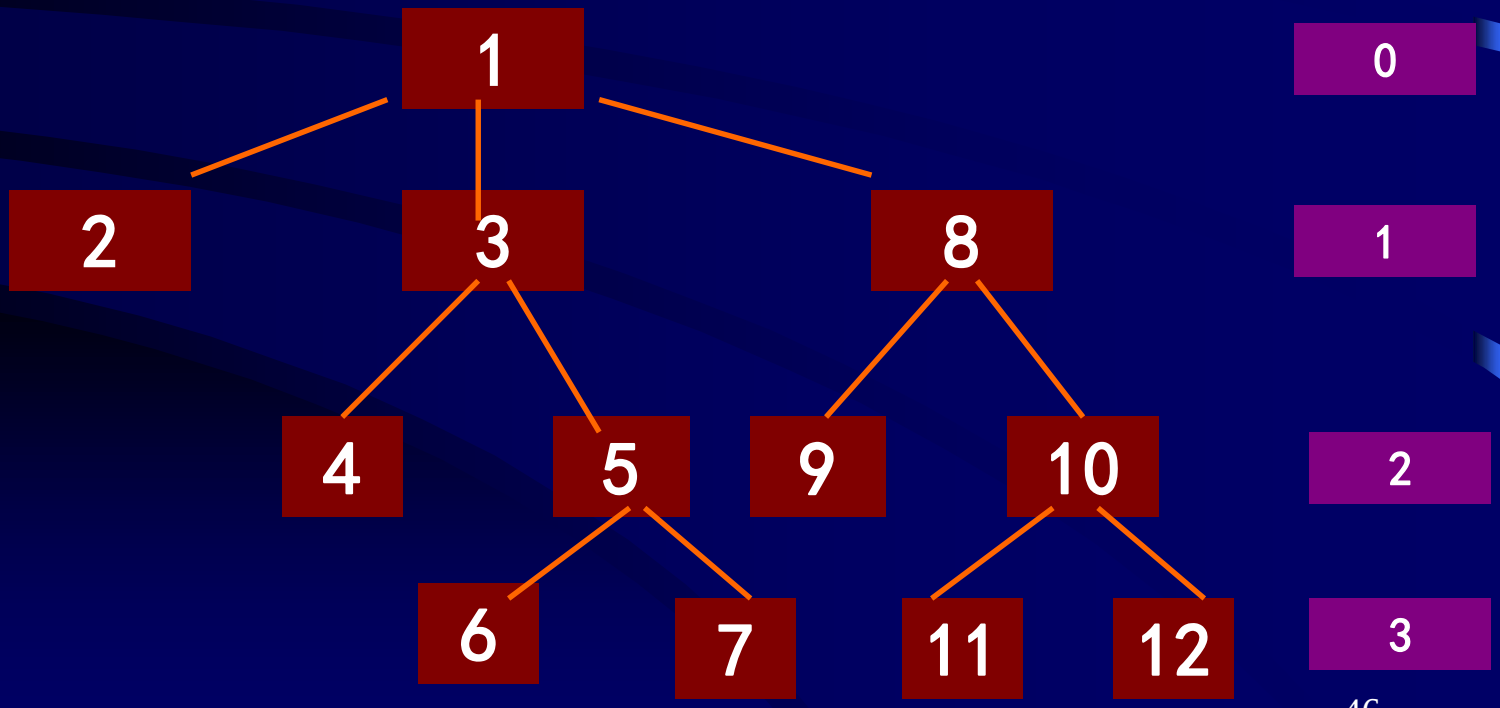
深度优先搜索也称为回溯搜索，它总是首先扩展树的最深层次上的某个节点，只是当搜索遇到一个死亡节点（非目标节点而且不可扩展），搜索方法才会返回并扩展浅层次的节点。

上述原理对树中的每一节点是递归实现的(实现该递归用栈)。

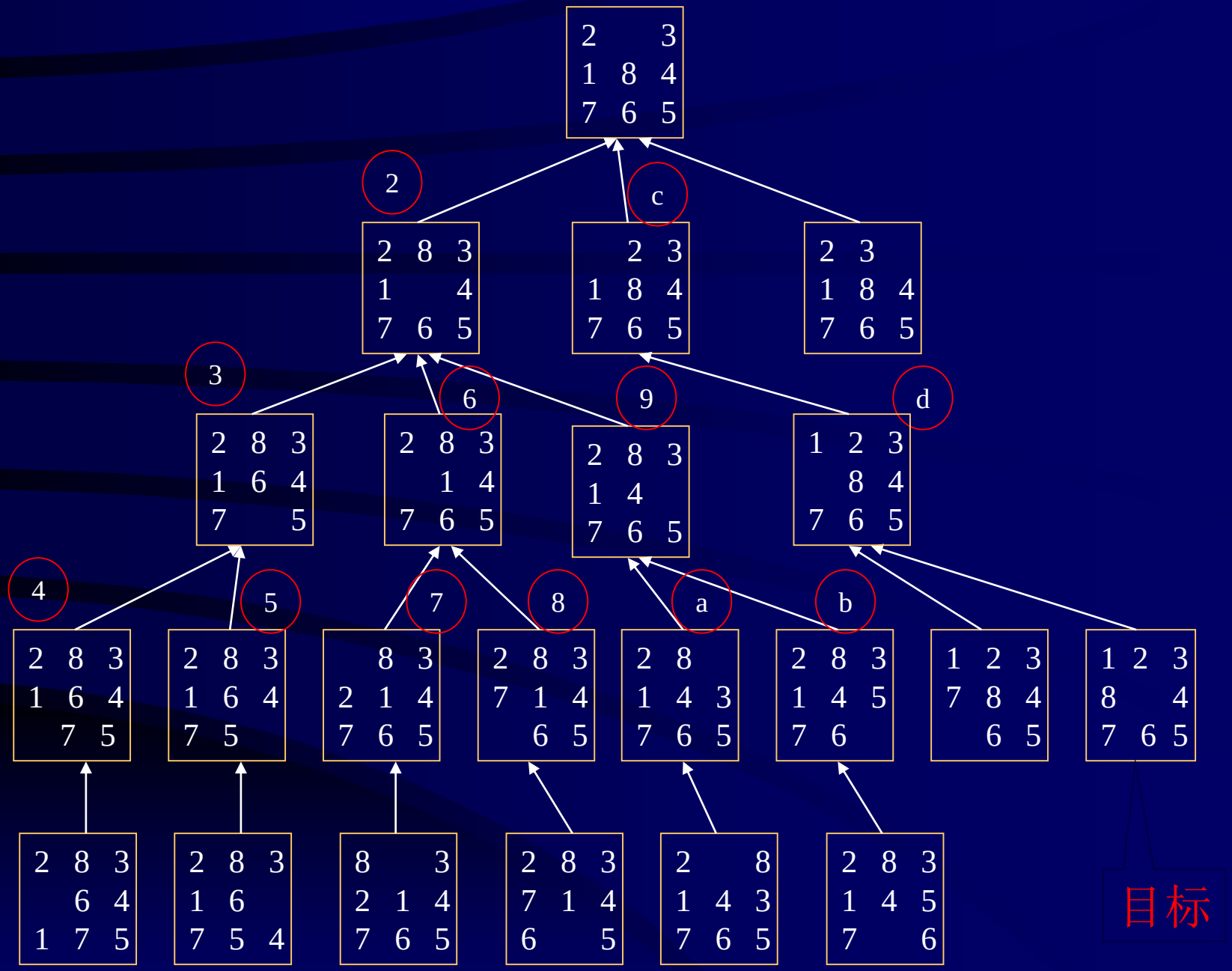
基于栈实现的深度优先搜索算法：

```
Procedure Depth First Search
  Begin
      把初始节点压入栈，并设置栈顶指针；
      While 栈不空 do
            Begin
            弹出栈顶元素；
            If 栈顶元素=goal，成功返回并结束；
            Else 以任意次序把栈顶元素的子女压入栈中；
      End While
  End
```

- 初始节点放到栈中，栈指针指向栈的最上边的元素。
- 为了对该节点进行检测，需要从栈中弹出该节点，如果是目标，该算法结束，否则把其子节点以任何顺序压入栈中。该过程直到栈变成为空。
- 遍历一棵树的过程（下图）。



- 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。


2 3
1 8 4
7 6 5
2
c
2 8 3
1 4
7 6 5
2 3
1 8 4
7 6 5
2 3
1 8 4
7 6 5
3
6
9
d
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
4
5
7
8
a
b
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
目标

3.2.3 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。

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：
  - 深度为0
  - 深度为1
  - 深度为2，…
- 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*。

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

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。

- 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。

第3章 搜索技术

3.1 概述

3.2 盲目搜索方法

3.3 启发式搜索

3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析

3.3 启发式搜索

3.3.1 启发性信息和评估函数
3.3.2 最好优先搜索算法
3.3.3 通用图搜索算法
3.3.4 A*算法
3.3.5 迭代加深A*算法

3.3 启发式搜索

- 启发式搜索用于两种不同类型的问题：
  - 前向推理
  - 反向推理
- 前向推理一般用于状态空间的搜索。在前向推理中，推理是从预选定义的初始状态出发向目标状态方向执行。
- 反向推理一般用于问题规约中。在反向推理中，推理是从给定的目标状态向初始状态执行。

A*算法

AND-OR图算法

3.3.1 启发性信息和评估函数

（启发式搜索）如果在选择节点时能充分利用与问题有关的特征信息，估计出节点的重要性，就能在搜索时选择重要性较高的节点，以便求得最优解。

- 用来评估节点重要性的函数称为评估函数。
- 评估函数为：

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

- 评估函数f(x)定义为从初始节点S0出发，约束地经过节点x到达目标节点Sg的所有路径中最小路径代价的估计值。

$g(x)$——从初始节点S0到节点x的实际代价；

$h(x)$——从x到目标节点Sg的最优路径的评估代价，它体现了问题的启发式信息，其形式要根据问题的特性确定，$h(x)$称为启发式函数。

- 启发式方法把问题状态的描述转换成了对问题解决程度的描述，这一程度用评估函数的值来表示。

- 如对八数码问题S_0和Sg

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

- 评估函数：

$$f(x)=d(x)+w(x)$$

$d(x)$ 表示节点在x搜索树中的深度，

$w(x)$ 表示节点x中不在目标状态中相应位置的数码个数，$w(x)$就包含了问题的启发式信息。

- 一般来说某节点的$w(x)$越大，即“不在目标位”的数码个数越少，说明它离目标节点越远。

- 对初始节点S_0，由于$d(S_0)=0$，

$w(S_0)=5$，因此$f(S_0)=5$。

- 在搜索过程中除了需要计算初始节点的评估函数外，更多的是需要计算新生节点的评估函数。

3.3.2 最好优先搜索算法

（最好优先）搜索是从最有希望的节点开始，并且生成其所有的子节点。

- 计算每个节点的性能（合适性），
- 选择最有希望的节点进行扩展，而不是仅仅从当前节点所生成的子节点中进行选择。
- 如果在早期选择了一个错误的节点，最好优先搜索就提供了一个修改的机会。

- 最好优先搜索算法并没有显式地给出如何定义启发式函数，
- 它不能保证当从起始节点到目标节点的最短路径存在时，一定能够找到它。

- A*算法就是对启发式函数加上限制后得到的一种启发式搜索算法。
- 在讨论A*算法之前，首先讨论通用的图搜索算法。

3.3.3 通用图搜索算法

- 图搜索算法只记录状态空间中那些被搜索过的状态，它们组成一个搜索图G。
- G由两种节点组成：
  - Open节点，如果该节点已经生成，而且启发式函数值$h(x)$ 已经计算出来，但是它没有扩展。这些节点也称为未考察节点。
  - Closed节点，如果该节点已经扩展并生成了其子节点。Closed节点是已经考察过的节点。
- 可以给出两个数据结构OPEN和CLOSED表，分别存放了Open 节点和Closed节点。

- 节点x 总的费用函数f（x）是g（x）和h（x）之和。
- 生成费用 g（x）可以比较容易地得到，如，如果节点 x 是从初始节点经过m步得到，则g（x）应该和 m 成正比（或者就是m）。
- h（x）只是一个预测值。
- 上述图搜索算法生成一个明确的图G（称为搜索图）和一个G的子集T（称为搜索树），图G中的每一个节点也在树T上。
- 搜索树是由返回指针来确定的。
- G中的每一个节点（除了初始节点$S0$）都有一个指向G中一个父辈节点的指针。该父辈节点就是树中那个节点的唯一父辈节点。

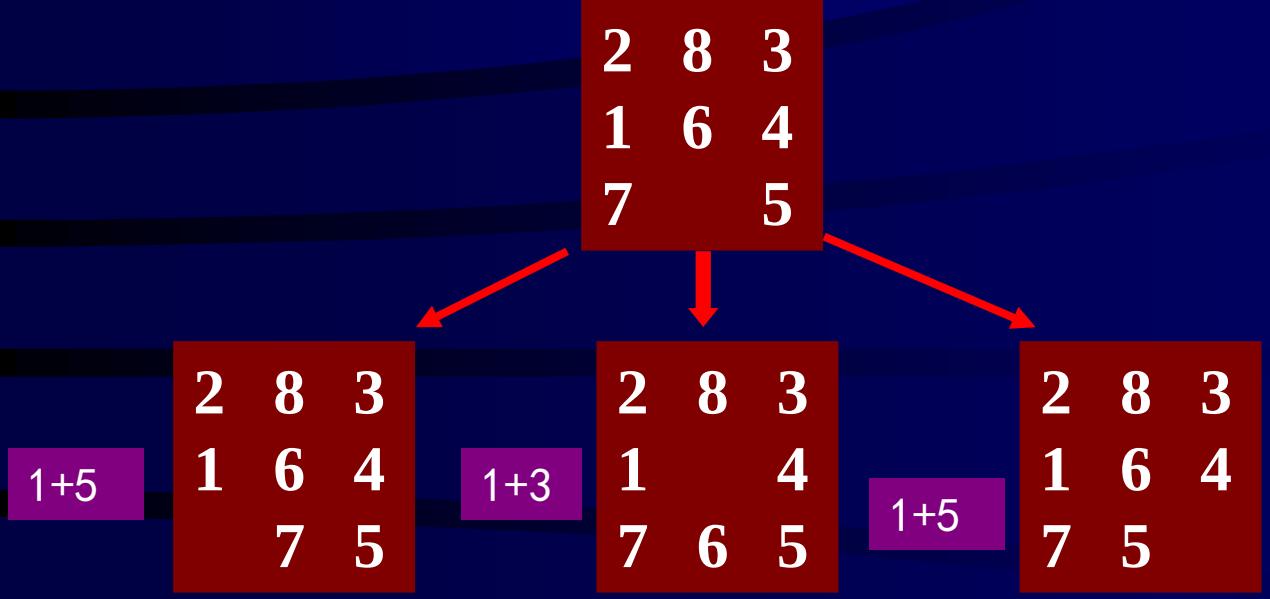


图3-5八数码问题的全局择优搜索树

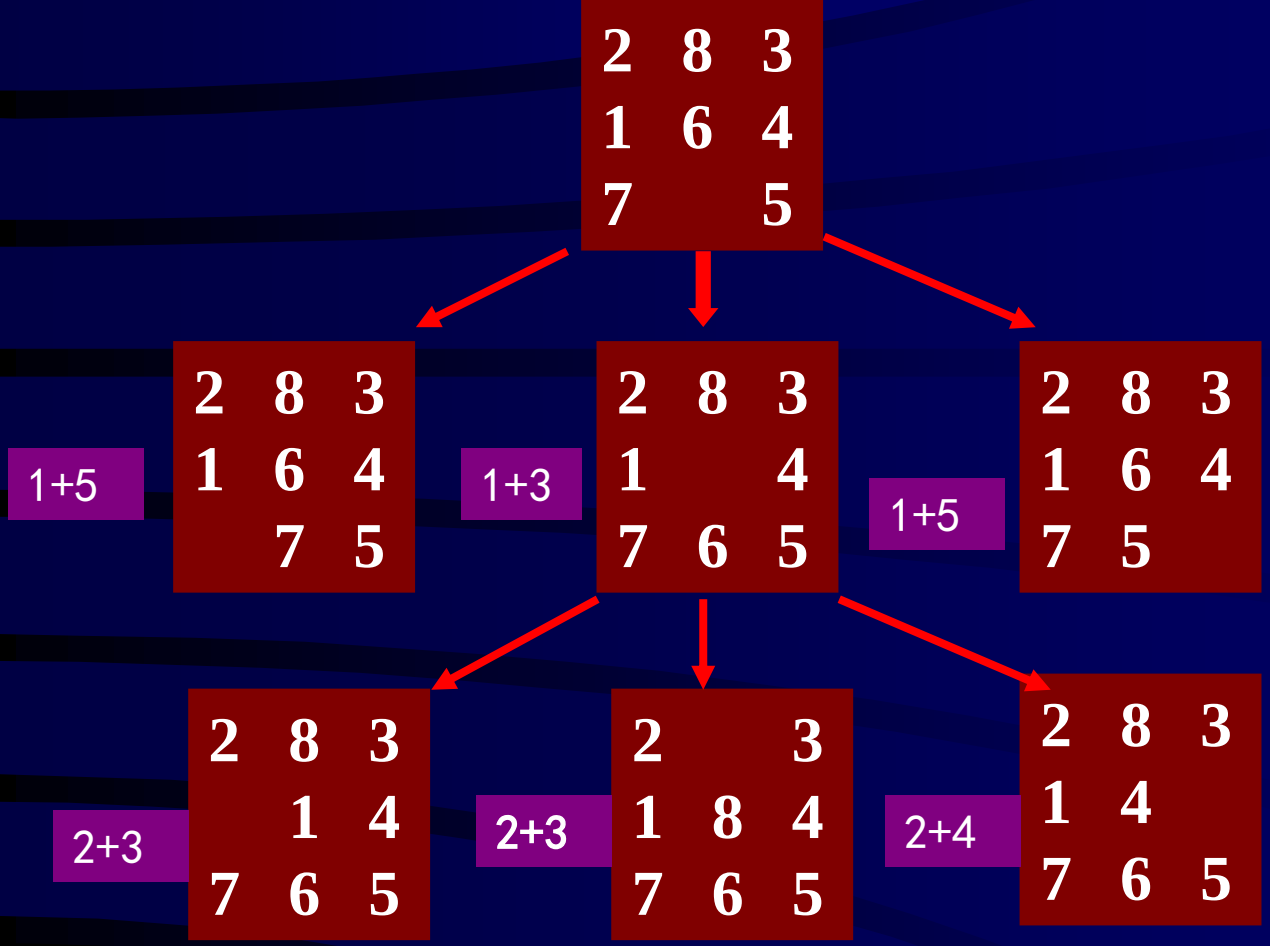


图3-5八数码问题的全局择优搜索树

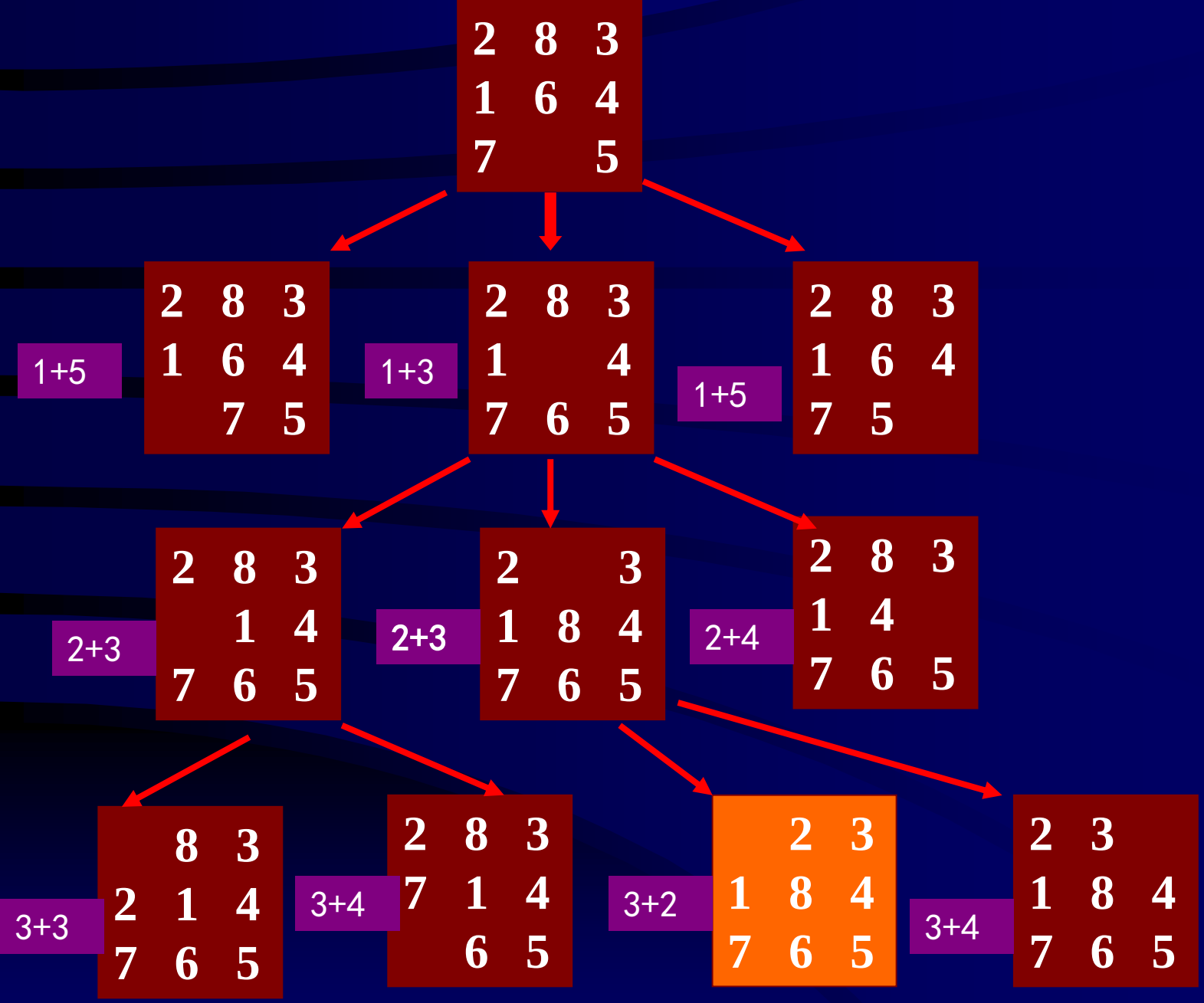


图3-5八数码问题的全局择优搜索树

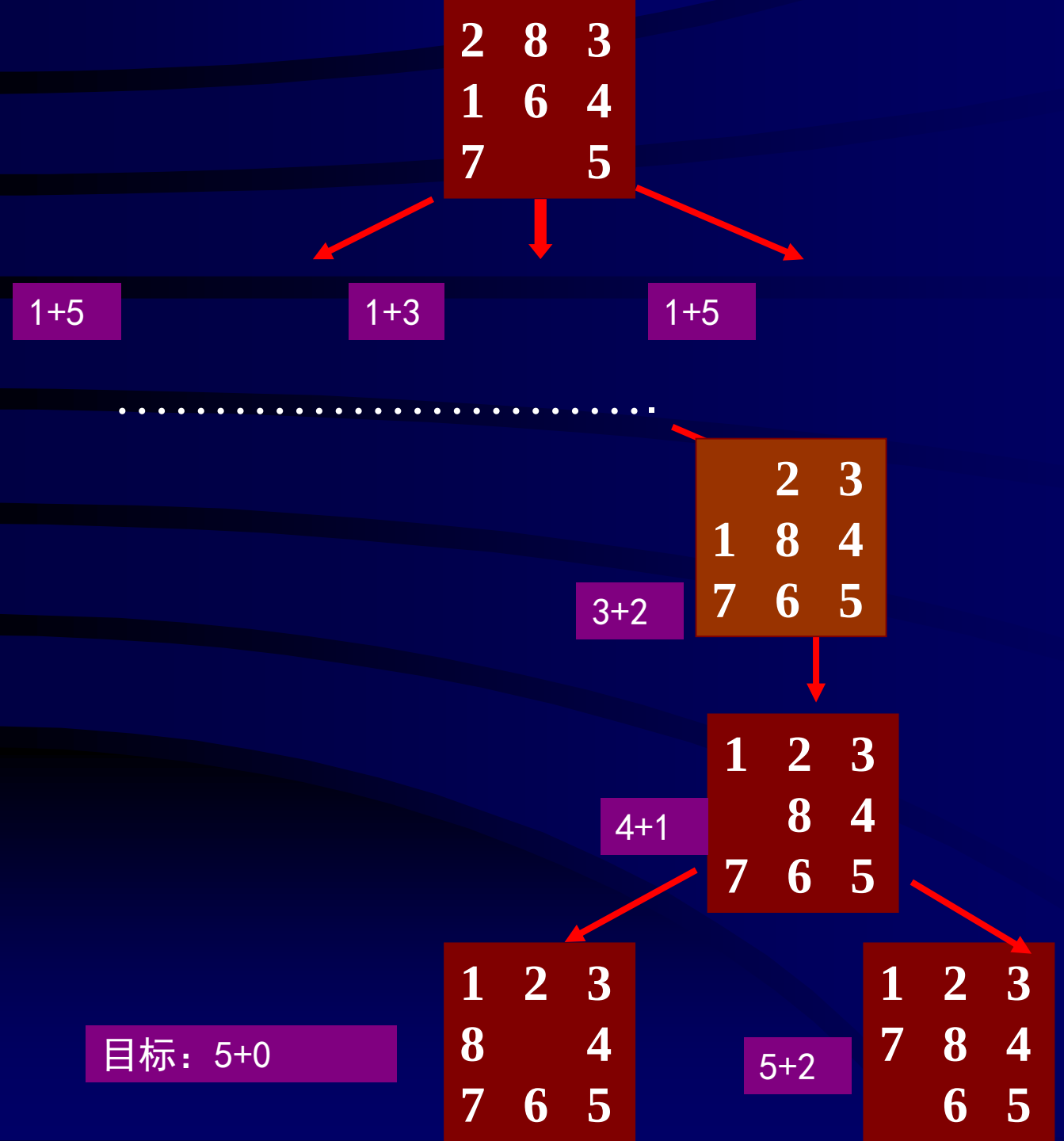
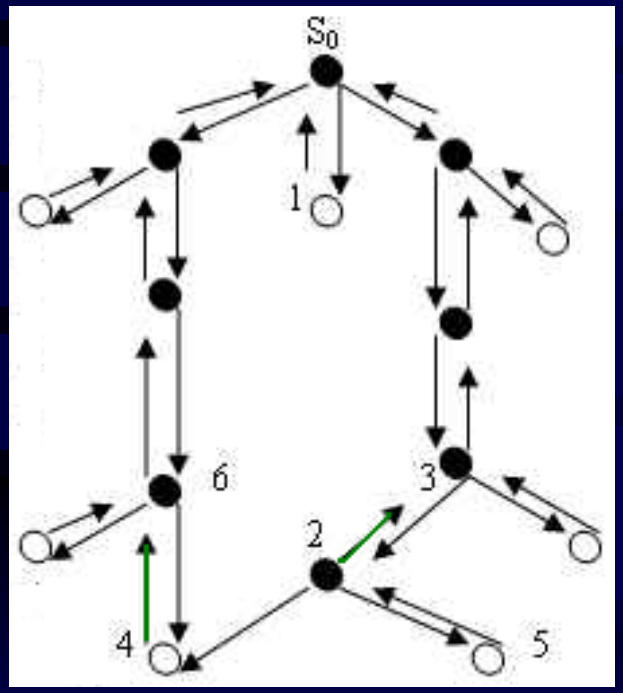


图3-5八数码问题的全局择优搜索树

最好优先搜索算法

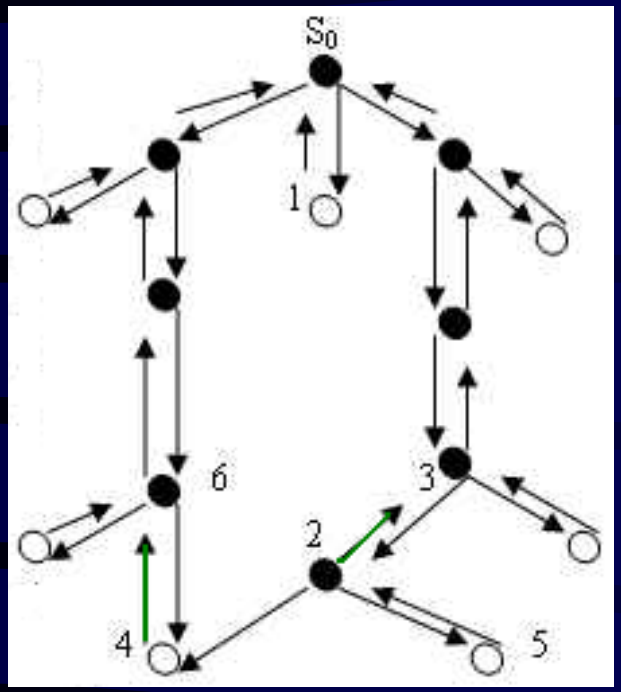
```
Procedure Graph-Search
Begin
    建立只含初始结点 S0 的搜索图 G,计算 f(S0);将 S0 放入 OPEN 表;将 CLOSED 表初始化为空
    While OPEN 表不空 Do
    Begin
        从 OPEN 表中取出 f(n)值最小的结点 n,将 n 从 OPEN 表中删除并放入 CLOSED 表
        If n 是目标结点 Then 根据 n 的父指针指出从 S0 到 n 的路径,算法停止
        Else
        Begin
            扩展结点 n
            If 结点 n 有子结点
             Then
            Begin
                (1)生成 n 的子结点集合{mi} 把 mi 作为 n 的子结点加入到 G 中,并计算 f(mi)
                (2)If mi 未曾在 OPEN 和 CLOSED 表中出现,Then 将它们配上刚计算过的 f
                   值,将 mi 的父指针指向 n,并把它们放入 OPEN 表
                (3)If mi 已经在 OPEN 表中,Then 该结点一定有多个父结点,在这种情况下,
                   比较 mi 相对于 n 的 f 值和 mi 相对于其原父指针指向的结点的 f 值,若前
                   者不小于后者,则不做任何更改,否则将 mi 的 f 值更改为 mi 相对于 n 的 f
                   值,mi 的父指针更改为 n
                (4)If mi 已经在 CLOSE 表中,Then 该结点同样也有多个父结点。在这种情况
                   下,比较 mi 相对于 n 的 f 值和 mi 相对于其原父指针指向的结点的 f 值。
                   如果前者不小于后者,则不作任何更改,否则将 mi 从 CLOSED 表移到 OPEN
                   表,置 mi 的父指针指向 n
                (5)按 f 值从小到大的次序对 OPEN 表中的结点重新排序
            End
        End
    End
End
```

- 设搜索算法已生成如下图（a）所示的搜索图和搜索树（带返回指针的部分）。图中实心圆点（●）表示在 CLOSED 表中的节点，空心圆点（○）表示在OPEN表中还未扩展的节点。

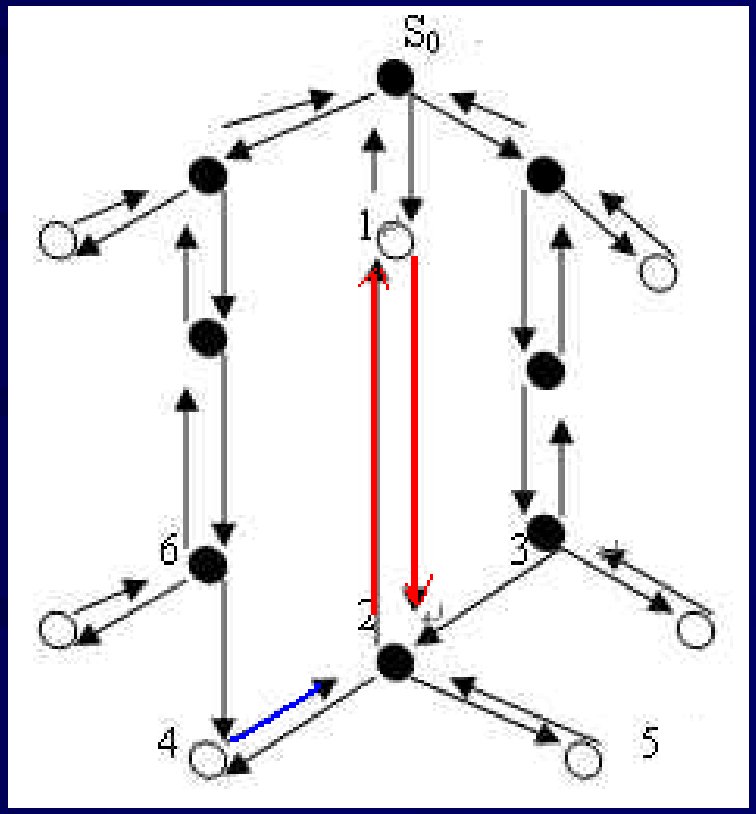


（a）是扩展节点1之前的搜索图和搜索树

- 当节点1扩展时，产生节点2。
- 而节点2是CLOSED 表中的节点，节点2的新$g(2)$值为2（设相邻两节点之间的代价均为1），而原来$g(2)$ 值为4，故将节点2的指针指向节点1，将原来指向节点3的指针断开。
- 同时要考虑修改节点2的后继节点4的返回指针和g（4）的值，将节点4指向节点2，并把原来指向节点6的指针断开。



（a）是扩展节点1之前的搜索图和搜索树



（b）是扩展节点1之后的搜索树和搜索图

例子（说明该算法的使用）

例 3.1 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*x*L，

3L壶里的水有*y*L，

*n*表示搜索空间中的任一节点。

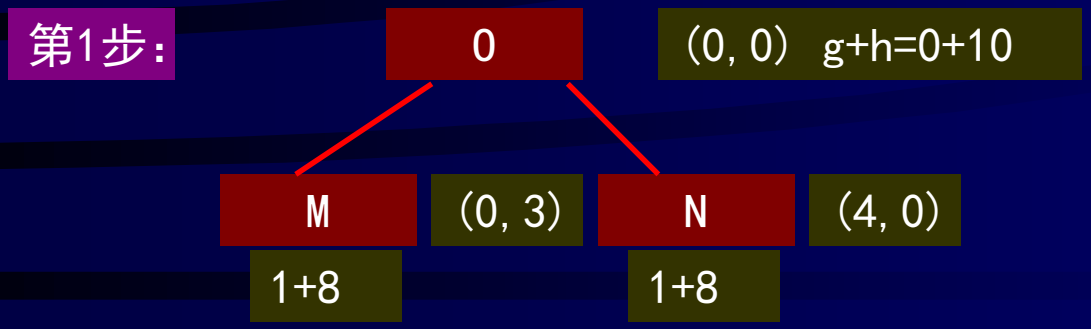
则给出下面的启发式函数：

- $h(n)$ = 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。

第0步：　　0　　（0, 0） g+h=0+10

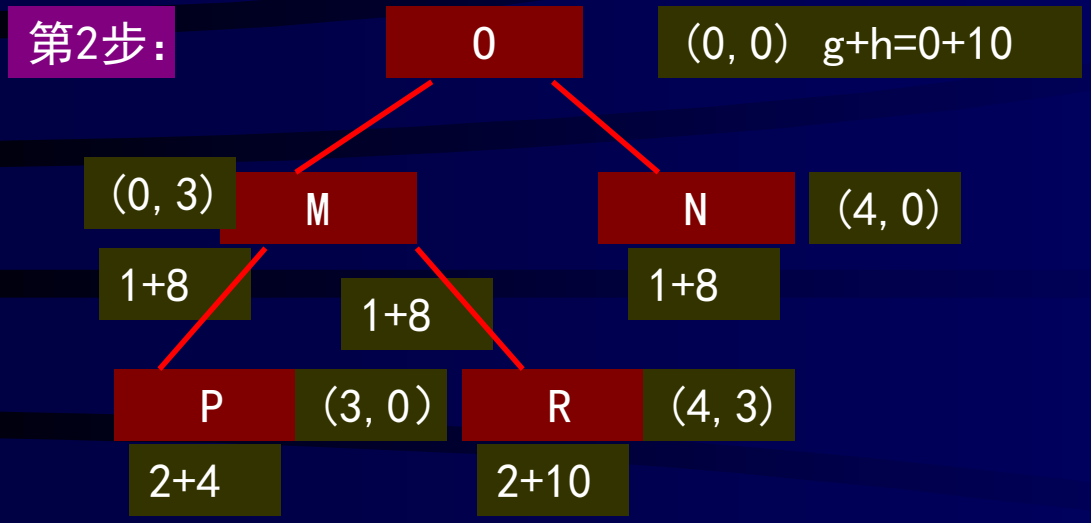
- 水壶问题的状态空间扩展图

在第0步，由节点0可以得到 g + h =10。



- 水壶问题的状态空间扩展图

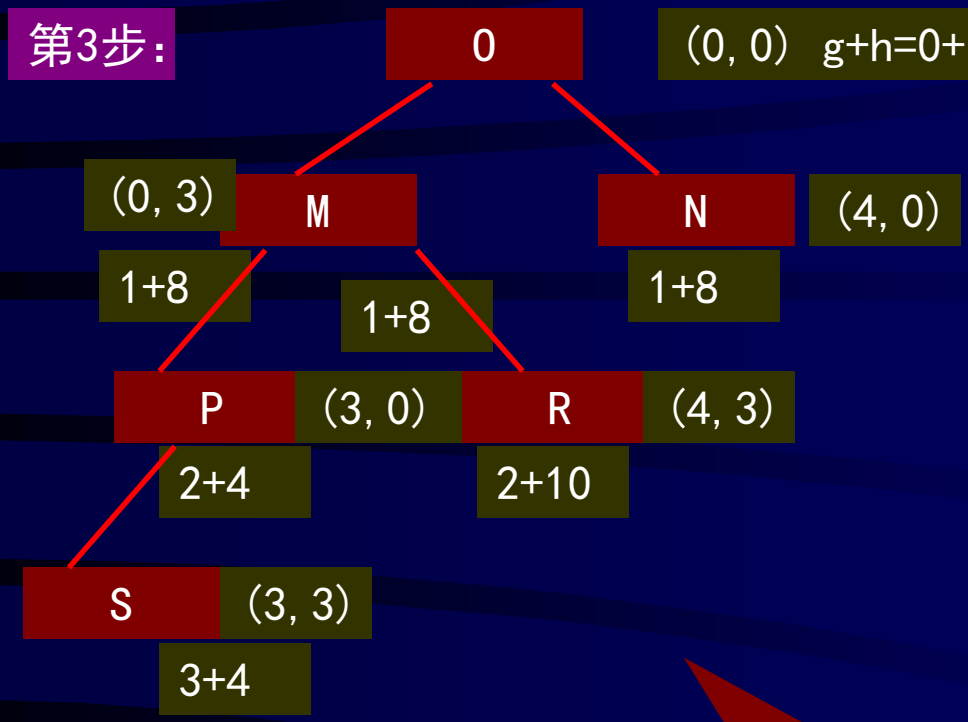
在第1步，得到两个节点M和N，其估价函数值都为1+8=9，因此可以任选一个节点扩展。



- 水壶问题的状态空间扩展图

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。



- 水壶问题的状态空间扩展图

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。

3.3.4 A*算法

- 在图搜索策略的基础上，给出A*算法。
- 评估函数f^*:

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

  - $g^*(n)$为起始节点到节点n的最短路径的代价，
  - $h^*(n)$是从n到目标节点的最短路径的代价。
- 这样$f^*(n)$就是从起始节点出发通过节点n到达目标节点的最佳路径的总代价的估值。
- 把估价函数$f(n)$和 $f^*(n)$相比较，$g(n)$是对$g^*(n)$的估价。$h(n)$是对$h^*(n)$的估价。

- 在这两个估价中，尽管g（n）容易计算，但它不一定就是从起始节点$S0$到节点n的真正的最短路径的代价，很可能从初始节点$S0$到节点n的真正最短路径还没有找到，所以一般都有：

$$g(n) \geqslant g^*(n)。$$

- 有了$g^*(n)$ 和$h^*(n)$ 的定义，如果对最好优先的启发式搜索算法中的$g(n)$ 和$h(n)$做如下的限制：

（1）g（n）是对g^*（n） 估计，且g（n）>0

（2）h（n）是$h^*(n)$的下界，即对任意节点n均有$h(n) \leqslant h^*(n)$

则称这样得到的算法为A*算法。

- Admissible (可采纳的) 性$h(n) \leqslant h^*$（n）的限制十分重要，它保证A*算法能够找到最优解。

在图3-5所示的八数码问题中，假定$h(n) = w(n)$。

尽管我们并不知道$h^*(n)$具体为多少，但当采用单位代价时，通过对“不在目标状态中相应位置的数码个数”的估计，可以得出至少需要移动$w(n)$步才能够到达目标，显然$w(n) \leqslant w^*(n)$。

因此它满足A*算法的要求，所以图3-5所示的路径是最短路径。

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

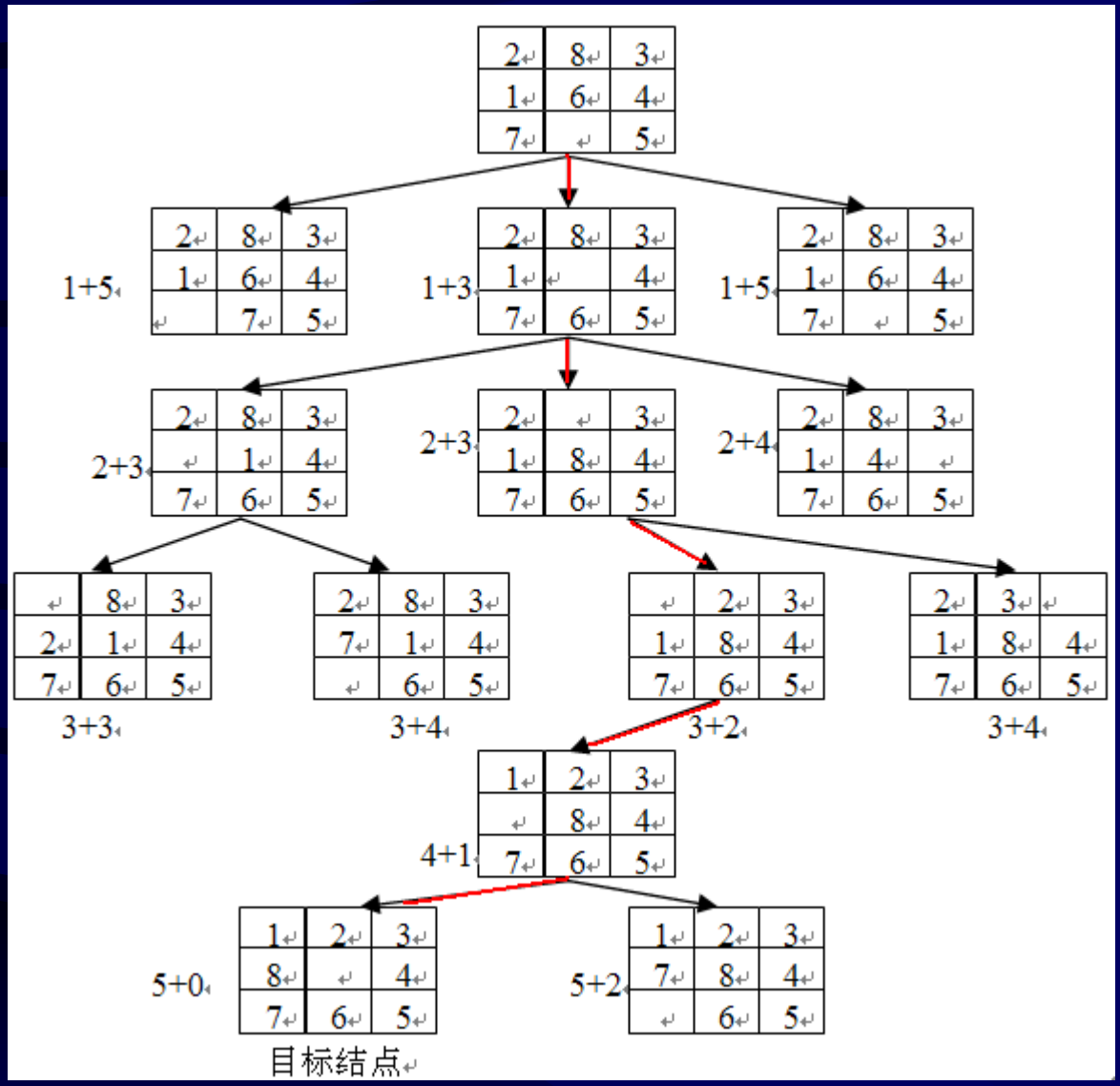
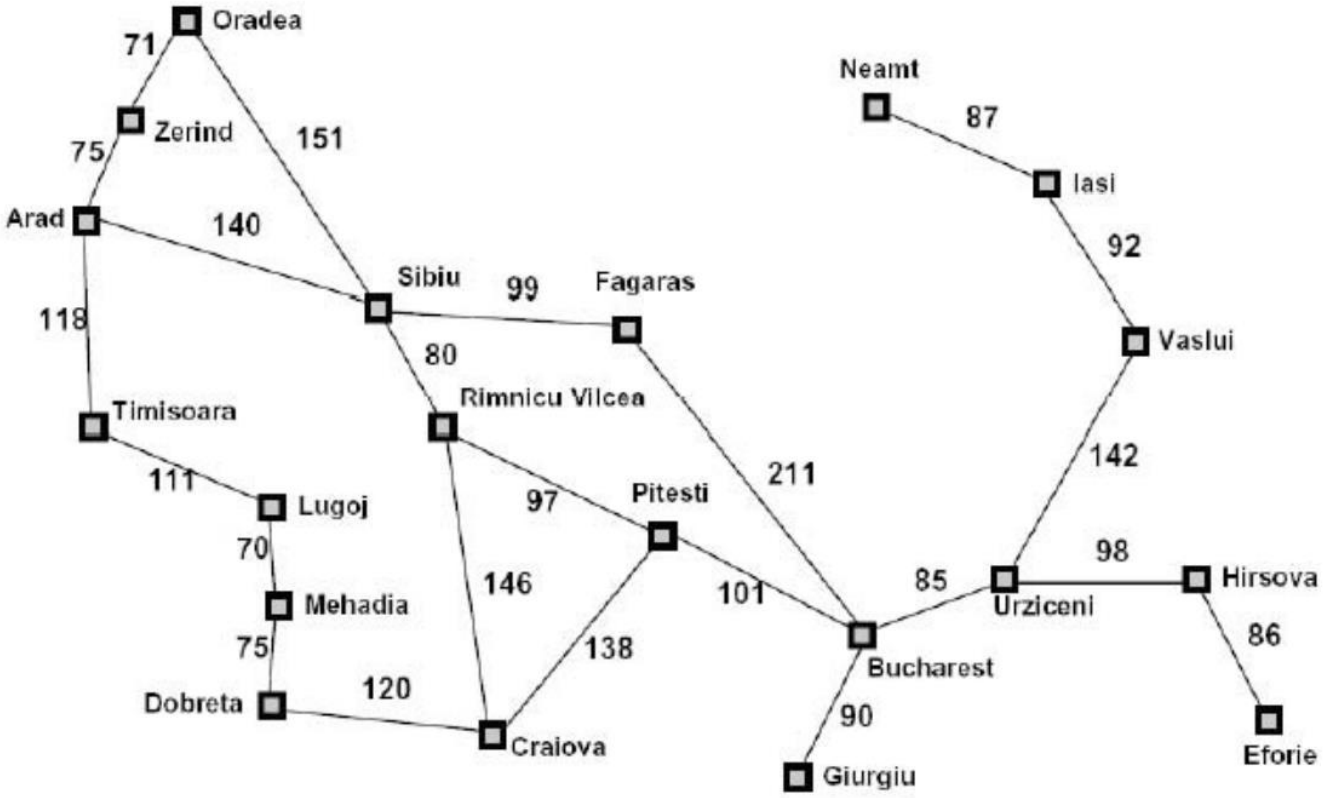
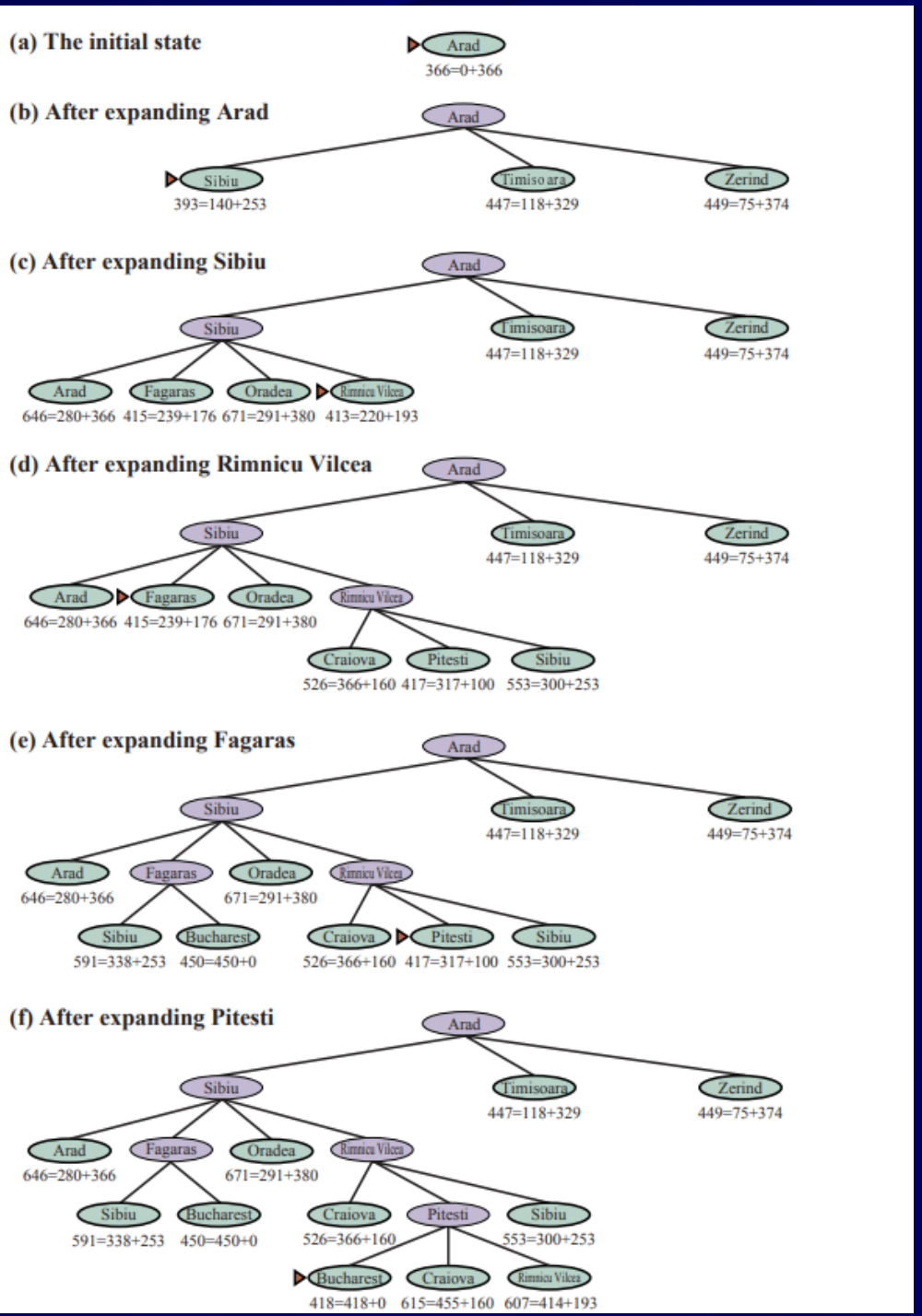


图3-5八数码问题的全局择优搜索树



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Arad-Sibiu-RV-Pitesli-Bucharest: 418


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
393=140+253
447=118+329
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
671=291+380
Sibiu
Bucharest
Craiova
Pitesti
Sibiu
591=338+253
450=450+0
526=366+160
417=317+100
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
671=291+380
Sibiu
Bucharest
Craiova
Pitesti
Sibiu
591=338+253
450=450+0
526=366+160
553=300+253
Bucharest
Craiova
Rimnicu Vilcea
418=418+0
615=455+160
607=414+193

同一问题启发函数$h(n)$可以有多种设计方法。

在八数码问题中，还可以定义启发函数$h(n) = p(n)$为节点n的每一数码与其目标位置之间的距离总和。显然有$w(n) \leqslant p(n) \leqslant w^*(n)$，相应的搜索过程也是A*算法。

然而$p(n)$比$w(n)$有更强的启发性信息，因为由$h(n) = p(n)$构造的启发式搜索树，比$h(n) = w(n)$构造的启发式搜索树节点数要少。

Depth	IDS	A*(Misplaced) **h1**	A*(Manhattan) **h2**
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

3.3.5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。

IDA*算法

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

第3章 搜索技术

3.1 概述

3.2 盲目搜索方法

3.3 启发式搜索

3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析

3.4 问题归约和AND-OR图启发式搜索

3.4.1 问题归约的描述

3.4.2 AND-OR图表示

3.4.3 AO*算法

- 启发式搜索可以应用的第二个问题是AND-OR图的反向推理问题。
- AND-OR图的反向推理过程可以看作是一个问题归约过程，
- 即是说在问题求解过程中，将一个大的问题变换成若干个子问题，子问题又可以分解成更小的子问题，这样一直分解到可以直接求解为止，全部子问题的解就是原问题的解；并称原问题为初始问题，可直接求解的问题为本原问题。
- 问题归约是不同于状态空间法的另一种问题描述和求解的方法。

3.4.1 问题归约的描述

- 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
  - P是本原问题集(不用证明的，公理、已知事实，或已证明过的问题)
  - O是操作算子集(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。

3.4.2 AND-OR图表示

- 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。

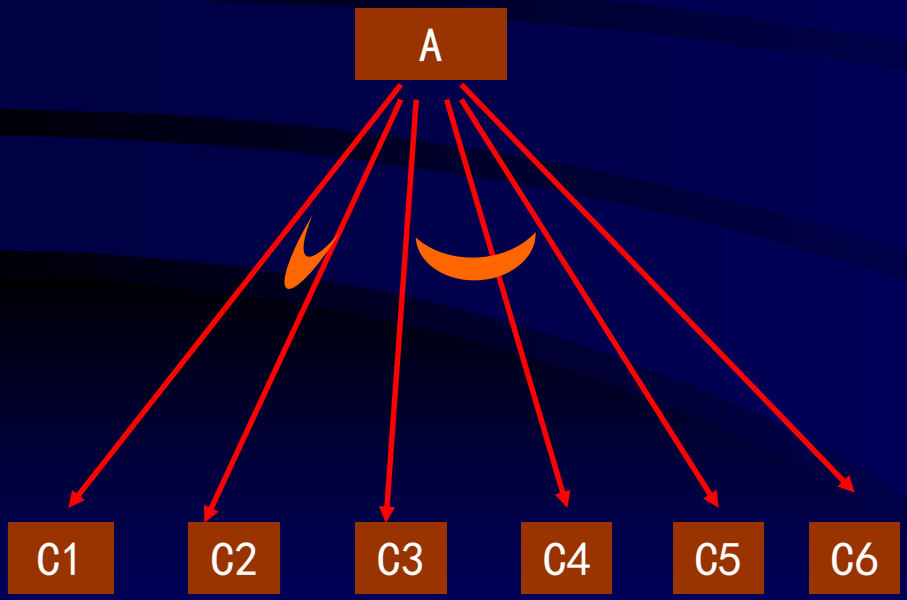


图3-7 子问题替换集合的结构

- 问题*C*1和*C*2 构成后继问题的一个集合，问题*C*3、*C*4和*C*5 构成另一后继问题集合；而问题*C*6 则为第三个集合。
- 对应于某个给定集合的各节点，用一个连接它们的圆弧来标记。
- 图3-7中连接*C*1与*C*2 和*C*3、*C*4、*C*5 的圆弧分别叫2连接弧和3连接弧。
- 一般而言，这种弧叫*K*连接弧，表示对问题*A*由某个操作算子作用后产生*K*个问题。

- 由节点及*K*连接弧组成的图，称为AND-OR图，当所有*K*均为1时，就变为普通的OR图。
- 可以对如图3-7所示的AND-OR图进行变换，引进某些附加节点，以便使含有一个以上后继问题的每个集合能够聚集在它们各自的父辈节点之下。

- 这样图3-7就变为图3-8所示的结构了，每个节点的后继只包含一个*K*连接弧。
- 弧连接的子节点叫与节点，如*C*1 、*C*2及*C*3、*C*4、*C*5 。
- *K*=1的连接弧连接的子节点叫做或节点。

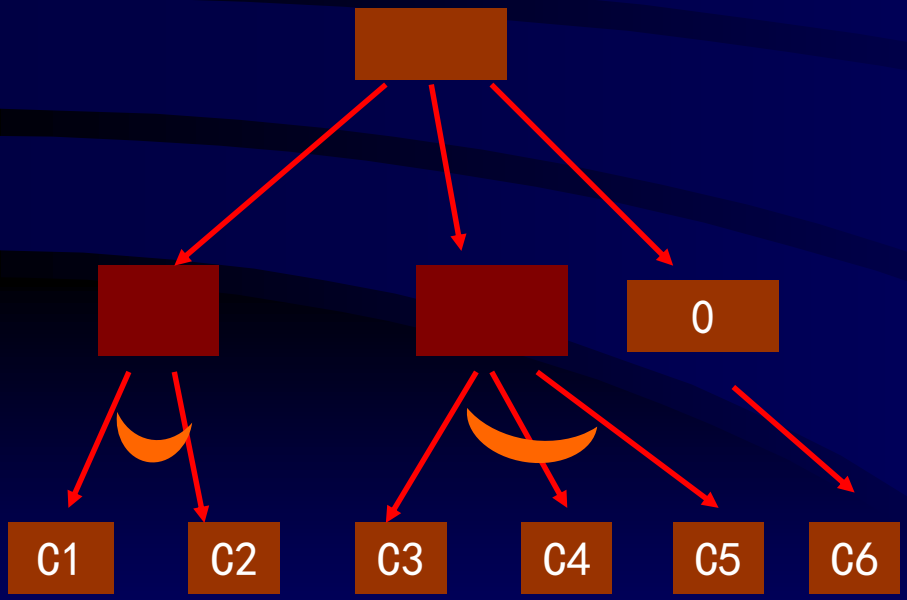


图3-8 各节点后继只含一个K连接弧的AND-OR图

- 假设AND-OR图中每个节点，只包含一个K连接的子节点。
- 将问题求解归约为AND-OR图搜索时，将初始节点表示初始问题描述，对应于本原问题的节点称为叶节点。

- 在AND-OR图上执行的搜索过程，其目的在于表明起始节点是有解的，AND-OR图中的一个可解节点可递归地定义如下：

（1）叶节点是可解节点；

（2）如果某节点为或子节点，那么该节点可解当且仅当至少有一个子节点为可解节点；

（3）如果某节点为与子节点，那么该节点可解当且仅当所有子节点均为可解节点。

- 不可解节点可递归定义如下：

(1) 有后裔节点的非叶节点是不可解节点；

(2) 或节点是不可解节点，当且仅当它的所有子节点都是不可解节点；

(3) 与节点是不可解节点，当且仅当它的子节点中至少有一个是不可解节点。

- 能导致初始节点可解的那些可解节点及有关连线组成的子图称为该AND-OR图的解图。
- 对OR图进行搜索，若搜索到某个节点*n*时，不论*n*是否生成了后继节点，*n*的费用是由其本身状态决定的，但对于AND-OR图不同，其费用计算规则如下：

  （1）*n*未生成后继节点，则费用由*n*的费用决定。

  （2）*n*已经生成了后继节点，则费用由*n*的后继节点的费用决定。

- 因为后继节点代表了分解的子问题，子问题的难易程度决定原问题求解的难易程度，所以不再考虑*n*本身的难易程度。
- 因此当决定了某个路径时，要将后继节点的估计值往回传送。
- 假设当前节点*n*到目标集Sg的费用估计为*h*（*n*）。

- 节点 n 的费用可以按下面的方法计算。

(1) 如果$n \in Sg$，则$h(n)=0$。

(2) 若n有一组由“与”弧连接的后继节点{ $n1, n2, \ldots, n_m$ }，则

$$h(n)=c1+c2+\ldots+c_m+h(n1)+h(n2)+\ldots h(n_m)$$

(3) 若n有一组由“或”弧连接的后继节点{ $n1, n2, \ldots, n_m$ }，则$h(n)$以其后继节点中费用最小者为其费用。

(4) 若n是既有“与”弧又有“或”弧连接的后继节点，则整个“与”弧作为一个“或”弧后继来考虑。

第3章 搜索技术

3.1 概述

3.2 盲目搜索方法

3.3 启发式搜索

3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析

3.5 博弈

3.5.1 极大极小过程

3.5.2 α－β 过程

3.5 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。

- 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。

（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。

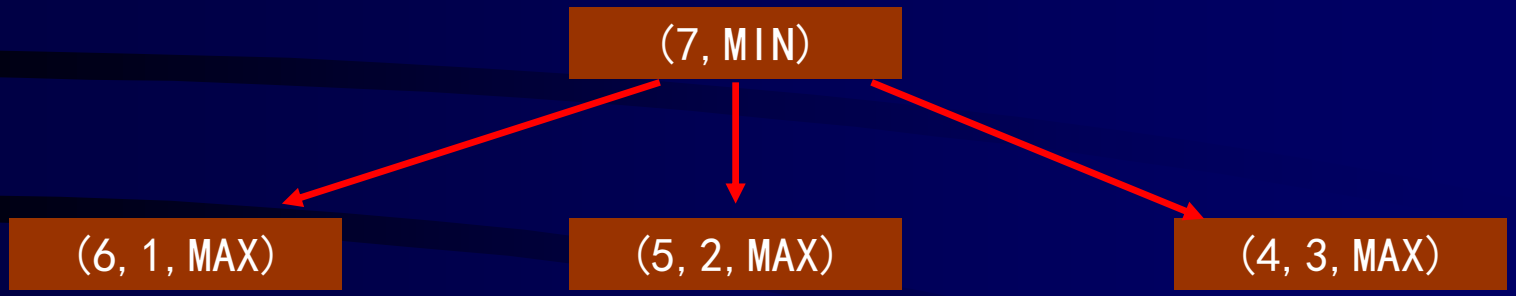
例：假设有七枚钱币，任一选手只能将已分好的一堆钱币分成两堆个数不等的钱币，两位选手轮流进行，直到每一堆都只有一个或两个钱币，不能再分为止，哪个选手遇到不能再分的情况，则为输。

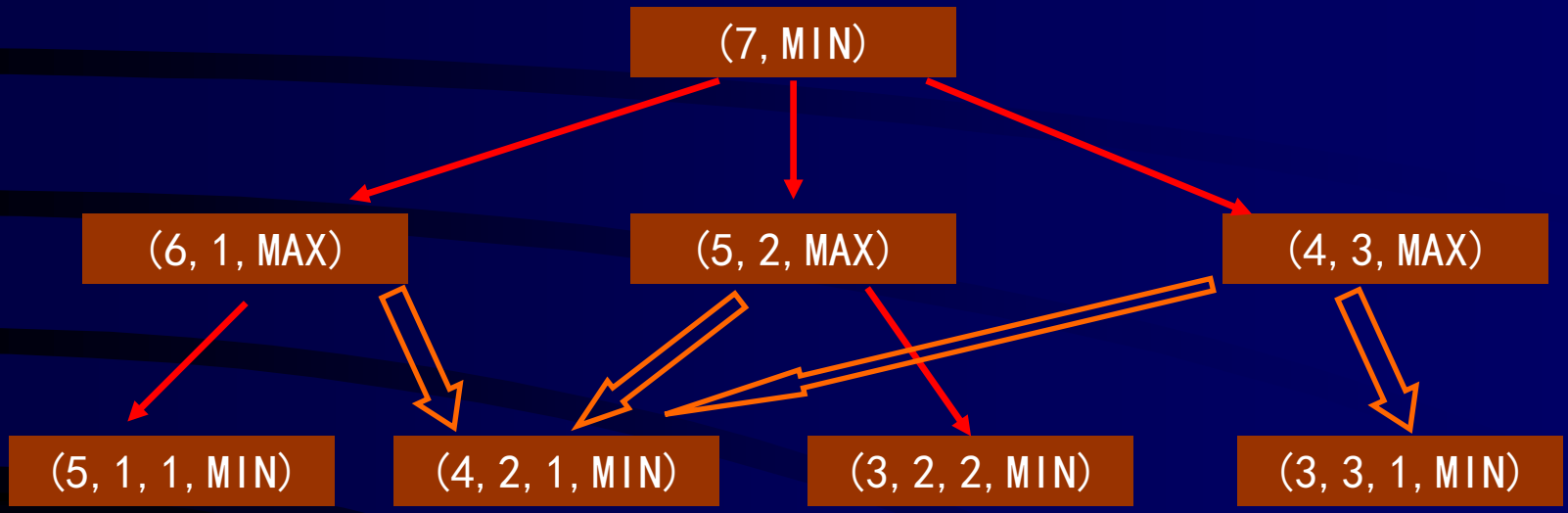
- 用数字序列加上一个说明表示一个状态，其中数字表示不同堆中钱币的个数，说明表示下一步由谁来分，
- 如（7，MIN）表示只有一个由七枚钱币组成的堆，由MIN走，MIN有3种可供选择的分法，即

  （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。

- 下图将双方可能方案表示出来，无论MIN怎么走，MAX总可获胜，取胜的策略用粗箭头表示。




(7, MIN)
(6, 1, MAX)
(5, 2, MAX)
(4, 3, MAX)
(5, 1, 1, MIN)
(4, 2, 1, MIN)
(3, 2, 2, MIN)
(3, 3, 1, MIN)

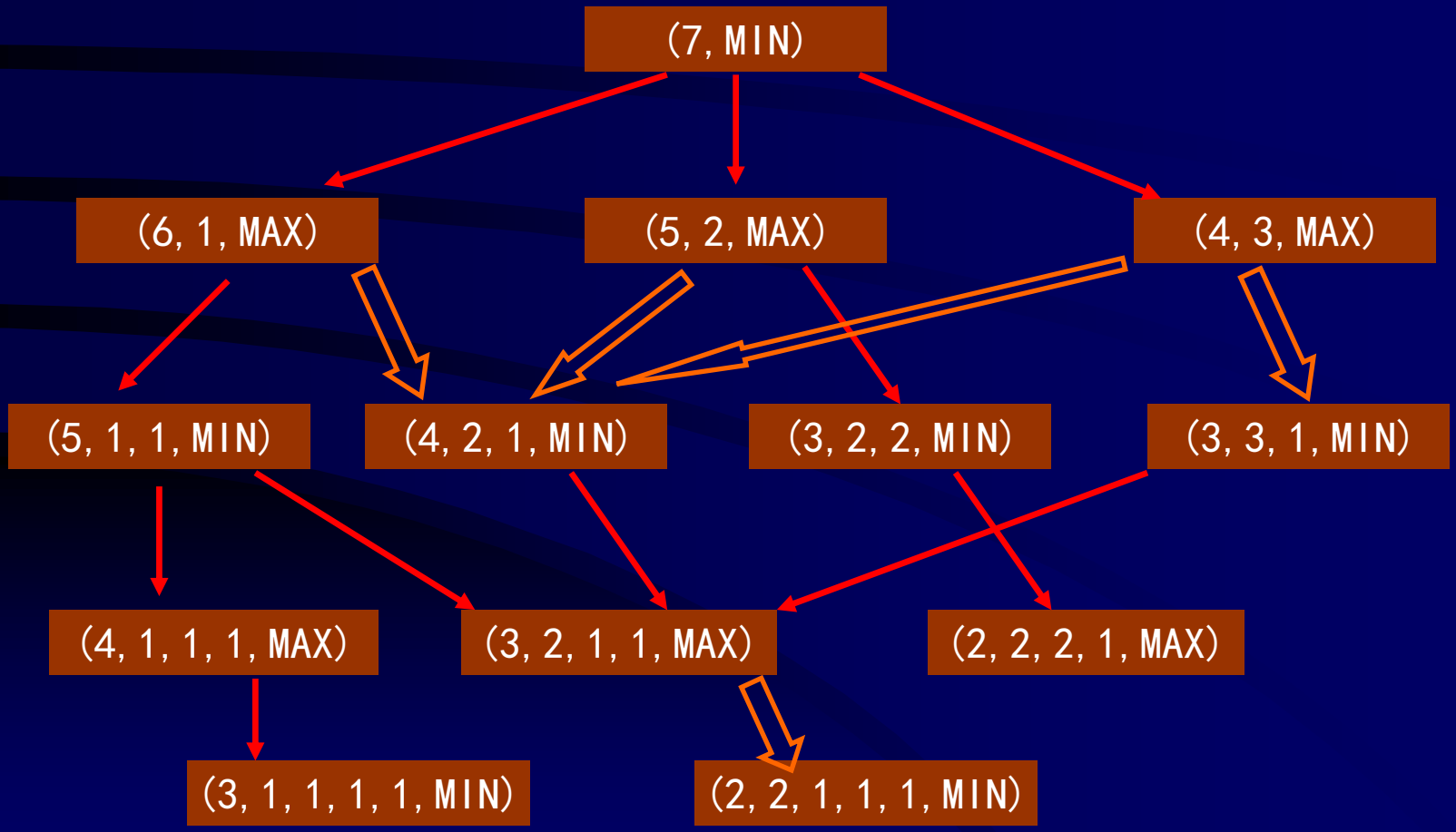
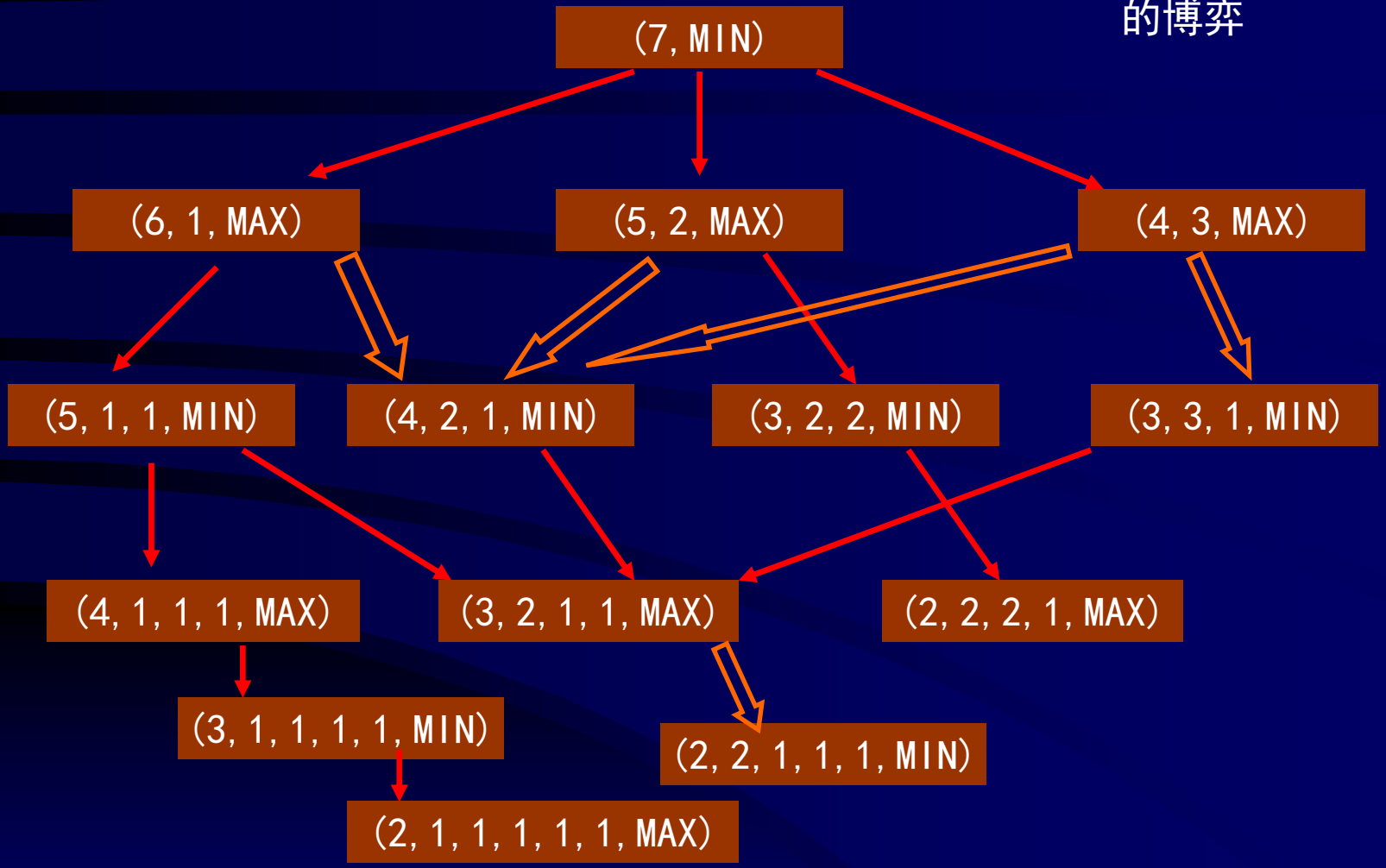

(7, MIN)
(6, 1, MAX)
(5, 2, MAX)
(4, 3, MAX)
(5, 1, 1, MIN)
(4, 2, 1, MIN)
(3, 2, 2, MIN)
(3, 3, 1, MIN)
(4, 1, 1, 1, MAX)
(3, 2, 1, 1, MAX)
(2, 2, 2, 1, MAX)
(3, 1, 1, 1, 1, MIN)
(2, 2, 1, 1, 1, MIN)

图3-17 分钱币的博弈


(7, MIN)
(6, 1, MAX)
(5, 2, MAX)
(4, 3, MAX)
(5, 1, 1, MIN)
(4, 2, 1, MIN)
(3, 2, 2, MIN)
(3, 3, 1, MIN)
(4, 1, 1, 1, MAX)
(3, 2, 1, 1, MAX)
(2, 2, 2, 1, MAX)
(3, 1, 1, 1, 1, MIN)
(2, 2, 1, 1, 1, MIN)
(2, 1, 1, 1, 1, 1, MAX)

- 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，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。

- 图3-17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。

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。

- AI中可以采用搜索方法来求解博弈问题，下面讨论博弈中两种最基本的搜索方法。

3.5.1 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，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，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。

- 图3-18 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。

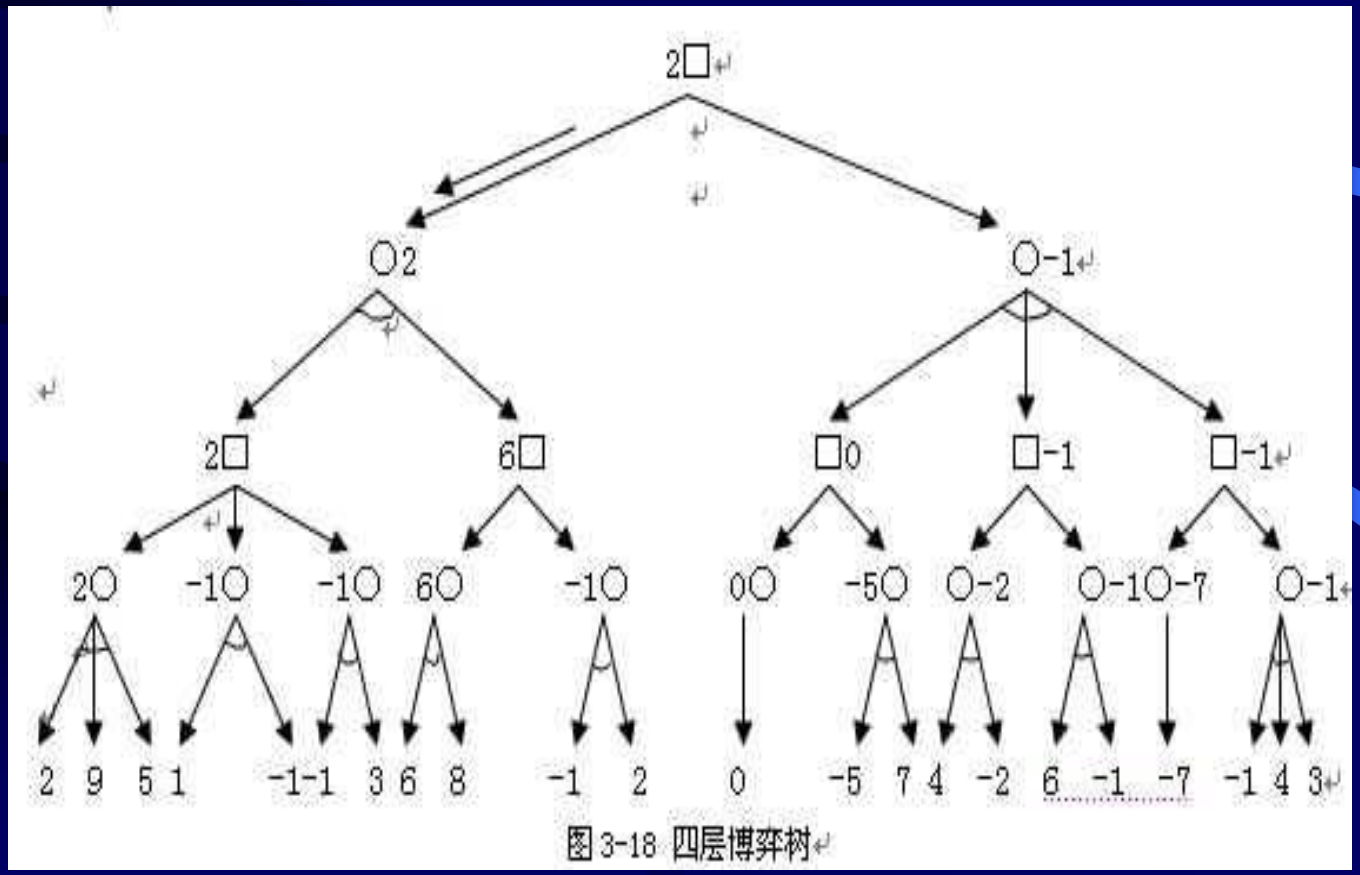


图3-18 四层博弈树

- 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。

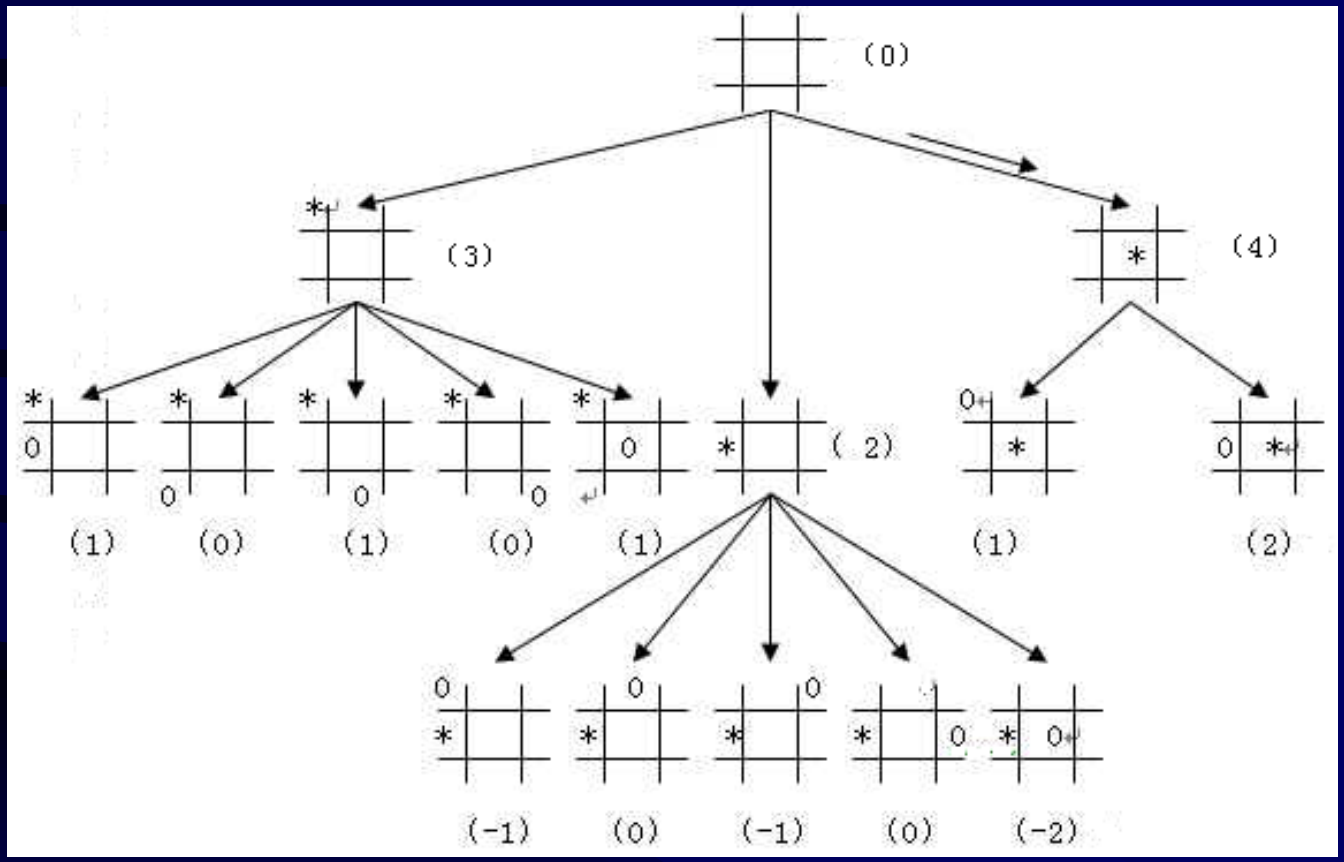


图3-19 一字棋博弈的极大极小过程

估价函数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）

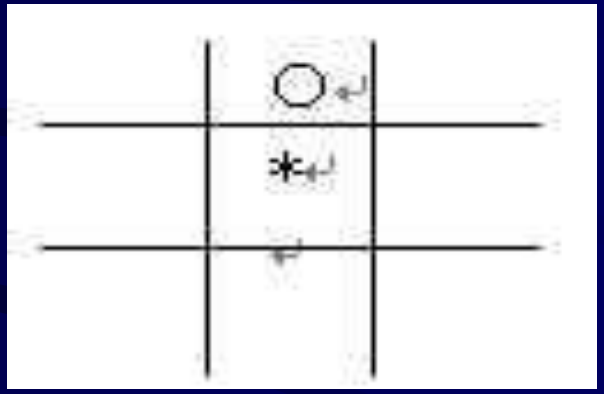
(2)若p是MAX获胜，则

$e(p) = +\infty$

（3） 若p是MIN获胜，则

$e(p) = -\infty$

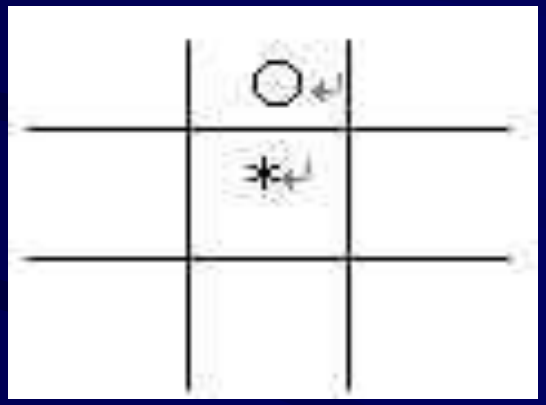
- 若p为



- 就有e(p) = 6-4 =2，其中* 表示MAX 方，〇表示MIN方。

- 在生成后继节点时，可以利用棋盘的对称性，省略了从对称上看是相同的格局。
- 图3-19 给出了MAX 最初一步走法的搜索树，由于* 放在中间位置有最大的倒推值，故MAX 第一步就选择它。

- MAX走了箭头指向的一步，例如MIN将棋子走在*的上方，得到 ：



- 下面MAX就从这个格局出发选择一步，做法与图3-19类似，直到某方取胜为止。

- 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。

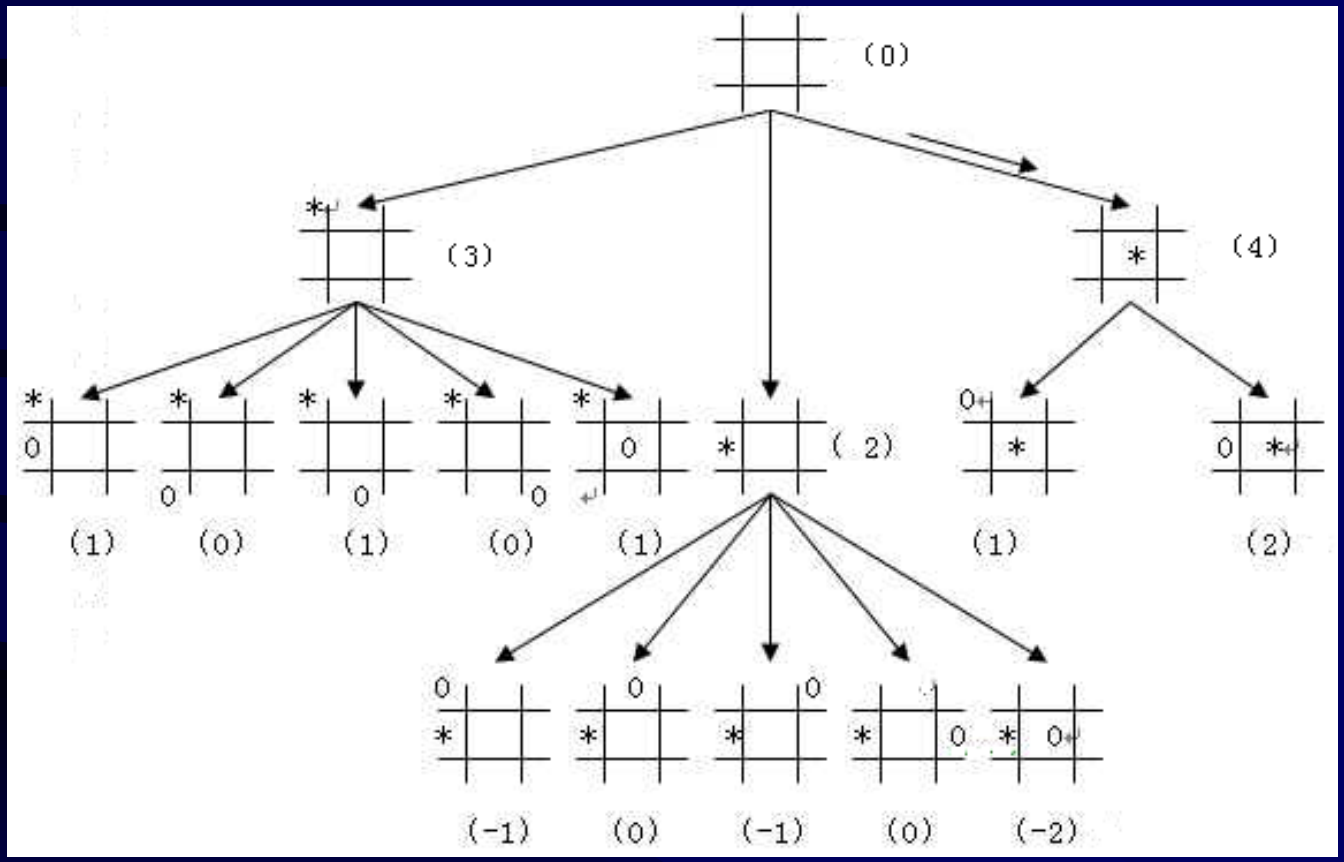


图3-19 一字棋博弈的极大极小过程

MINMAX算法：

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

3.5.2 α－β过程 （1）

- 极大极小方法是把搜索树的生成和估值这两个过程完全分开. 只有在已经生成树之后才开始进行估值, 这一分离导致了低效率的策略
  - 游戏状态随游戏招数成指数增长

中国象棋

一盘棋平均走50步，总状态数约为10^{160}

假设每秒搜索10^4个节点，约需10^{145}年

结论：不能搜索整个状态空间

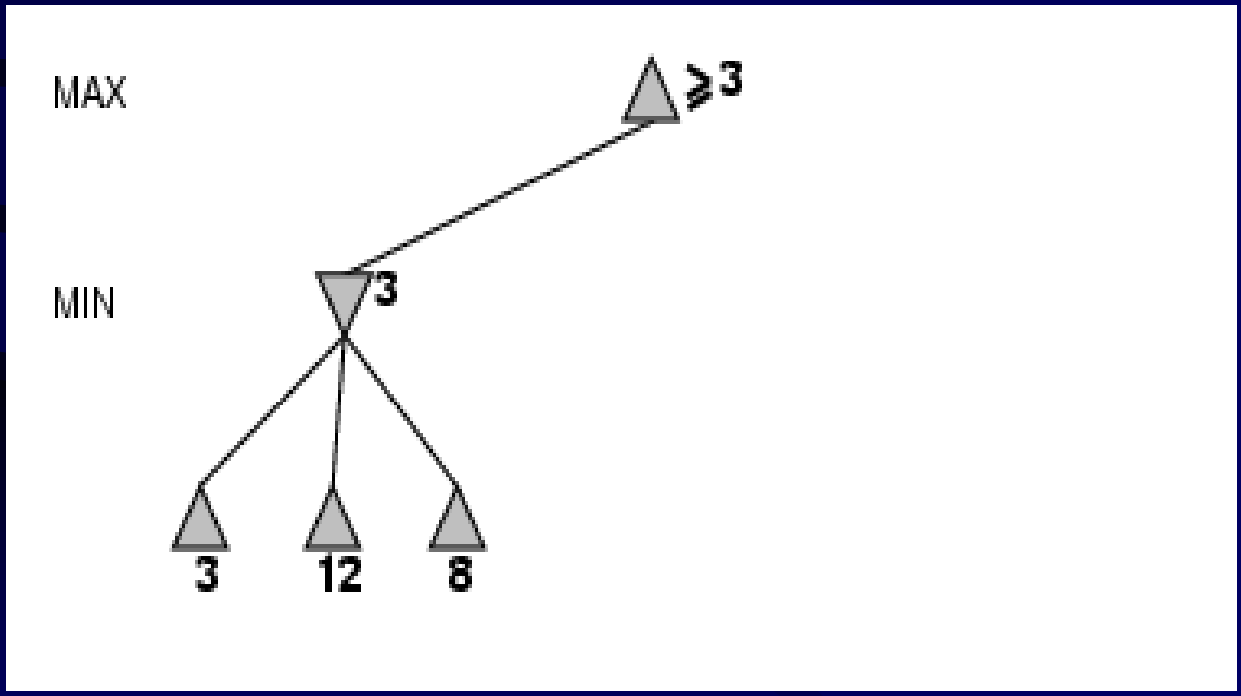
- α-β剪枝技术是极大极小方法的改进, 是一种提高博弈树搜索效率的方法
- α-β剪枝技术是一种边生成节点, 边计算估值和倒推值的方法, 从而剪去某些分枝.

α-β过程（2）

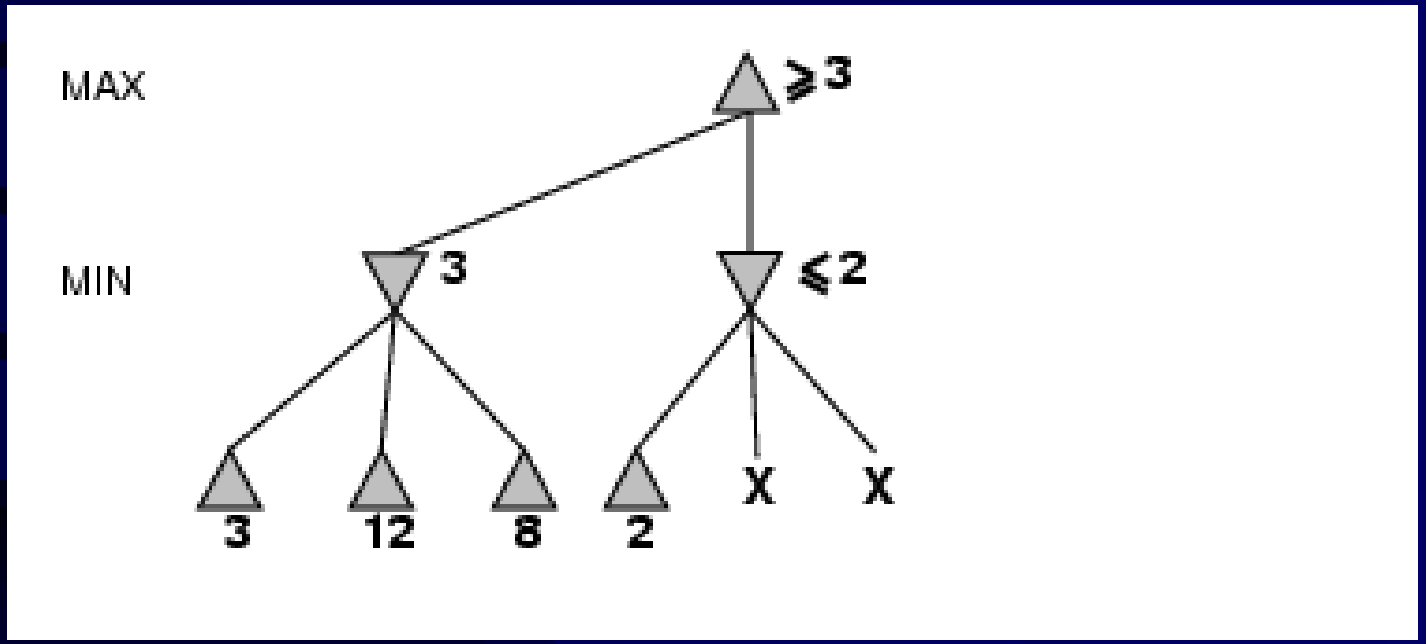
- α和β值的定义和计算
  - 对于一个“与”（MIN）节点, 它取当前子节点中的最小的倒推值作为它的倒推值的上界, 称此值为β值. 也就是说β值可以等于其后继节点当前最小的最终倒推值. “与”节点的β值是永远不会增加的
  - 对于一个“或”（MAX）节点, 它取当前子节点中得最大的倒推值作为它的倒推值的下界, 称此值为α值. 也就是说α值可以等于其后继节点当前最大的最终倒推值. “或”节点的α值是永远不会减少的.

α-β过程（3）

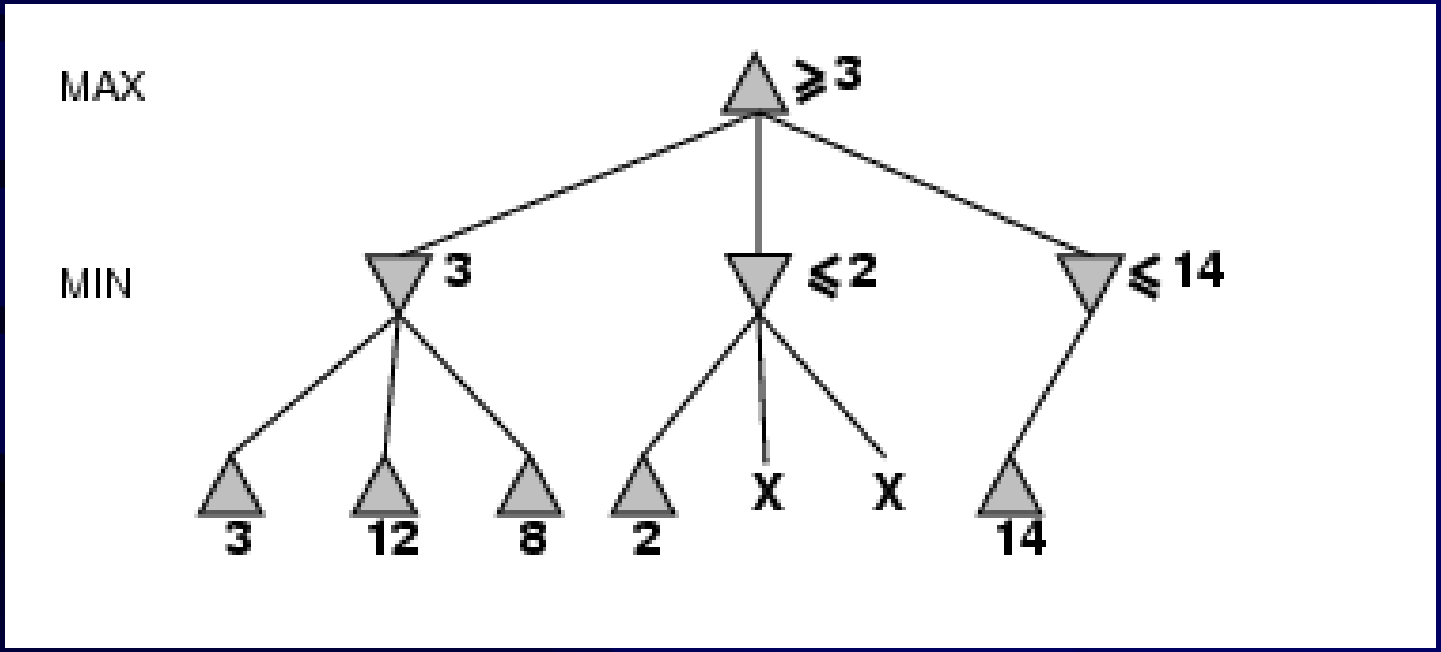
执行深度优先搜索直到第一个叶节点



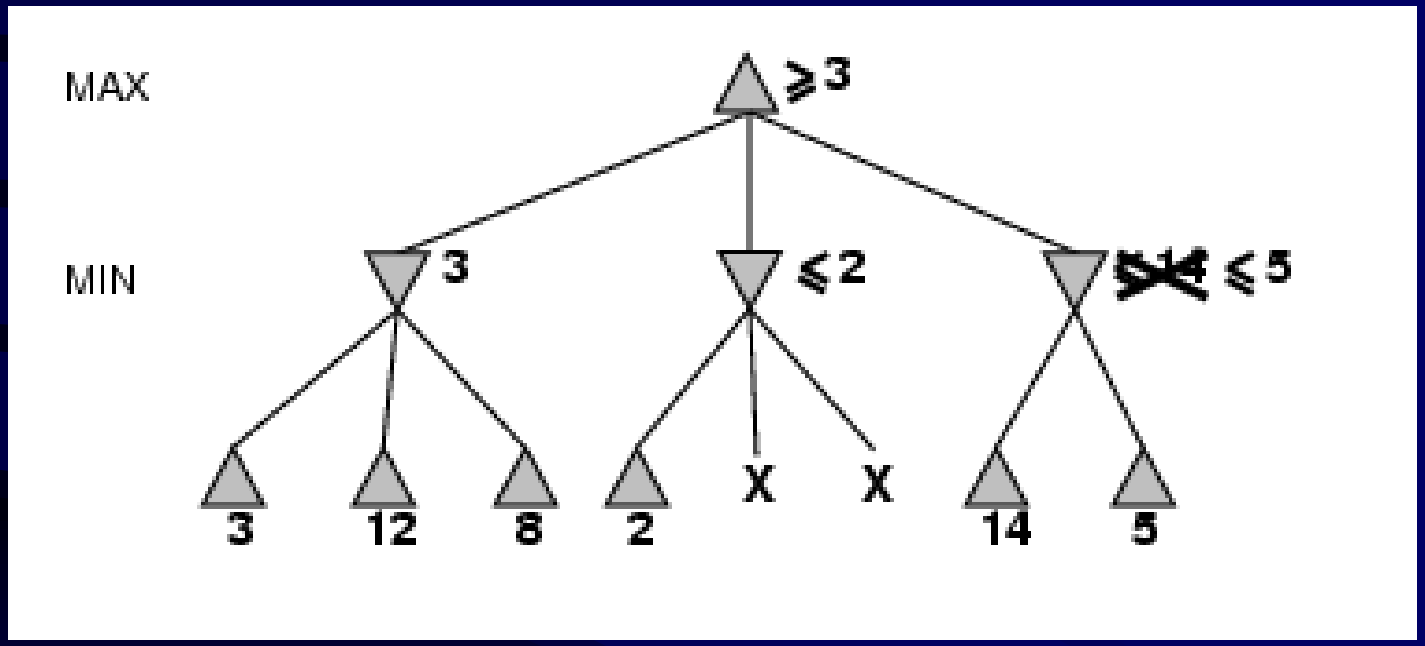
α–β过程（4）



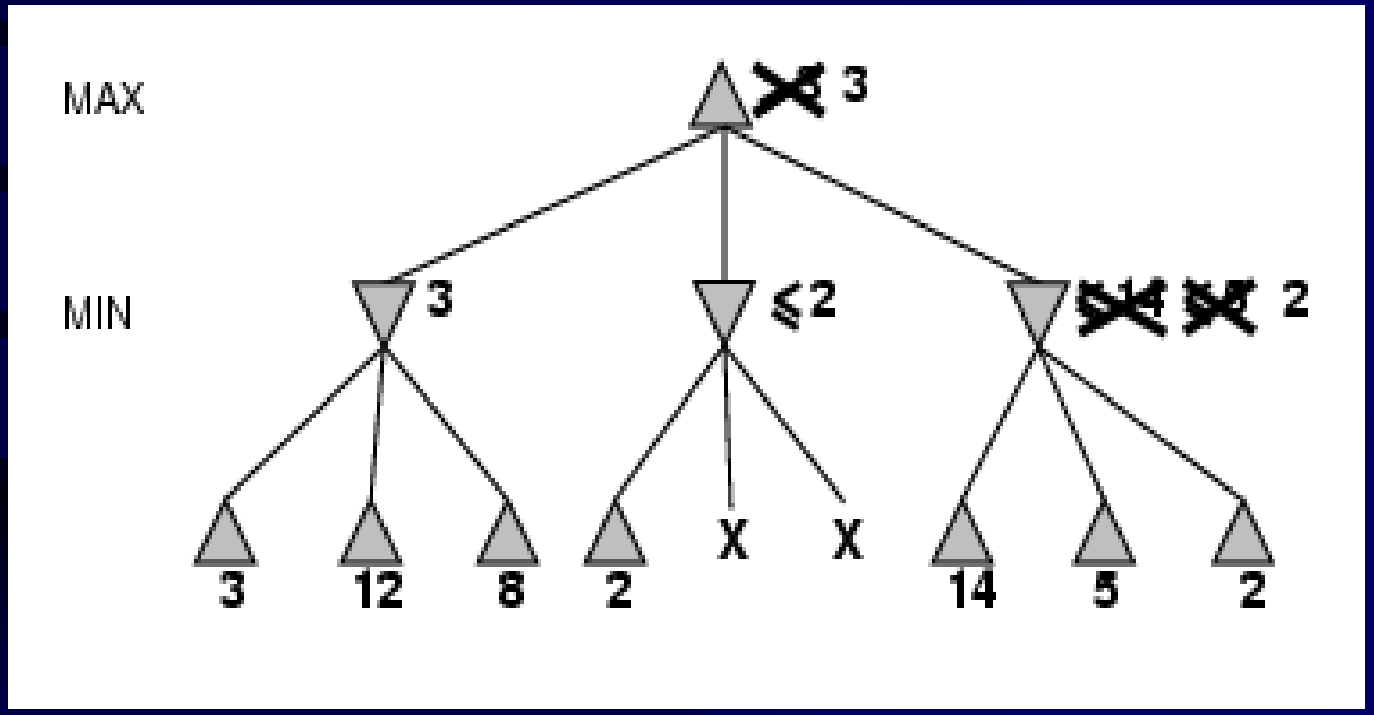
α-β过程（5）



α-β过程（6）

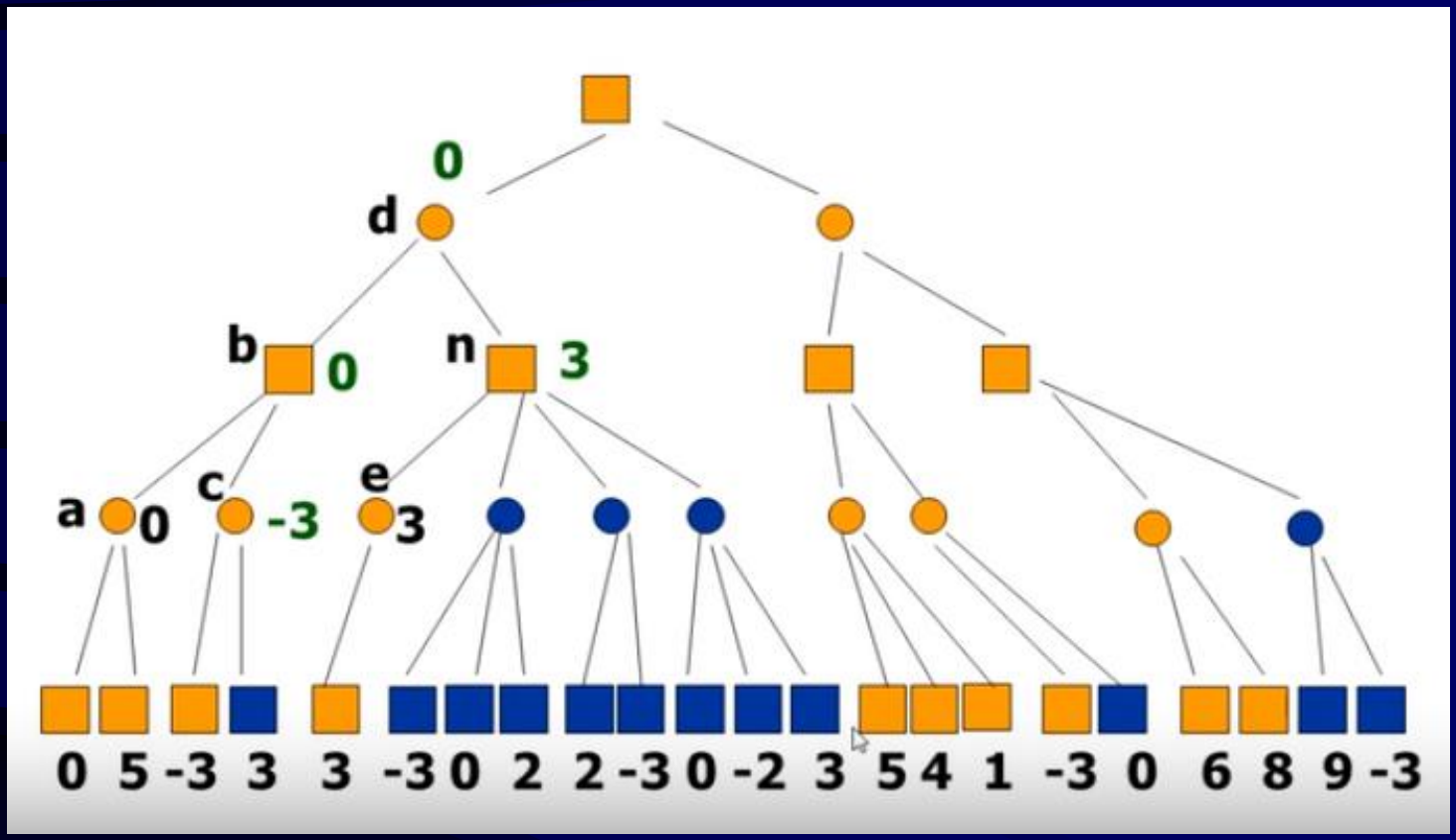


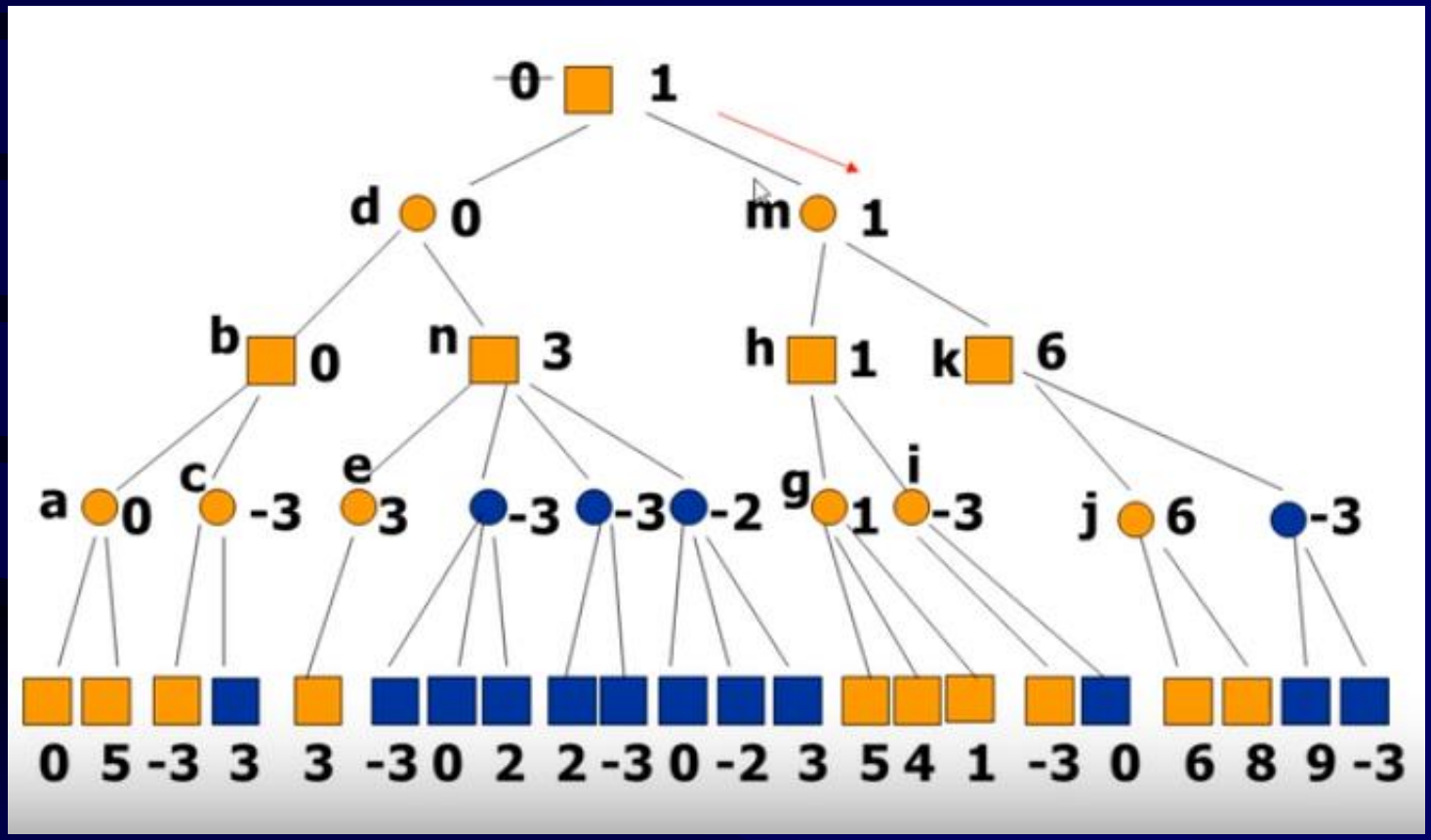
α–β过程（7）

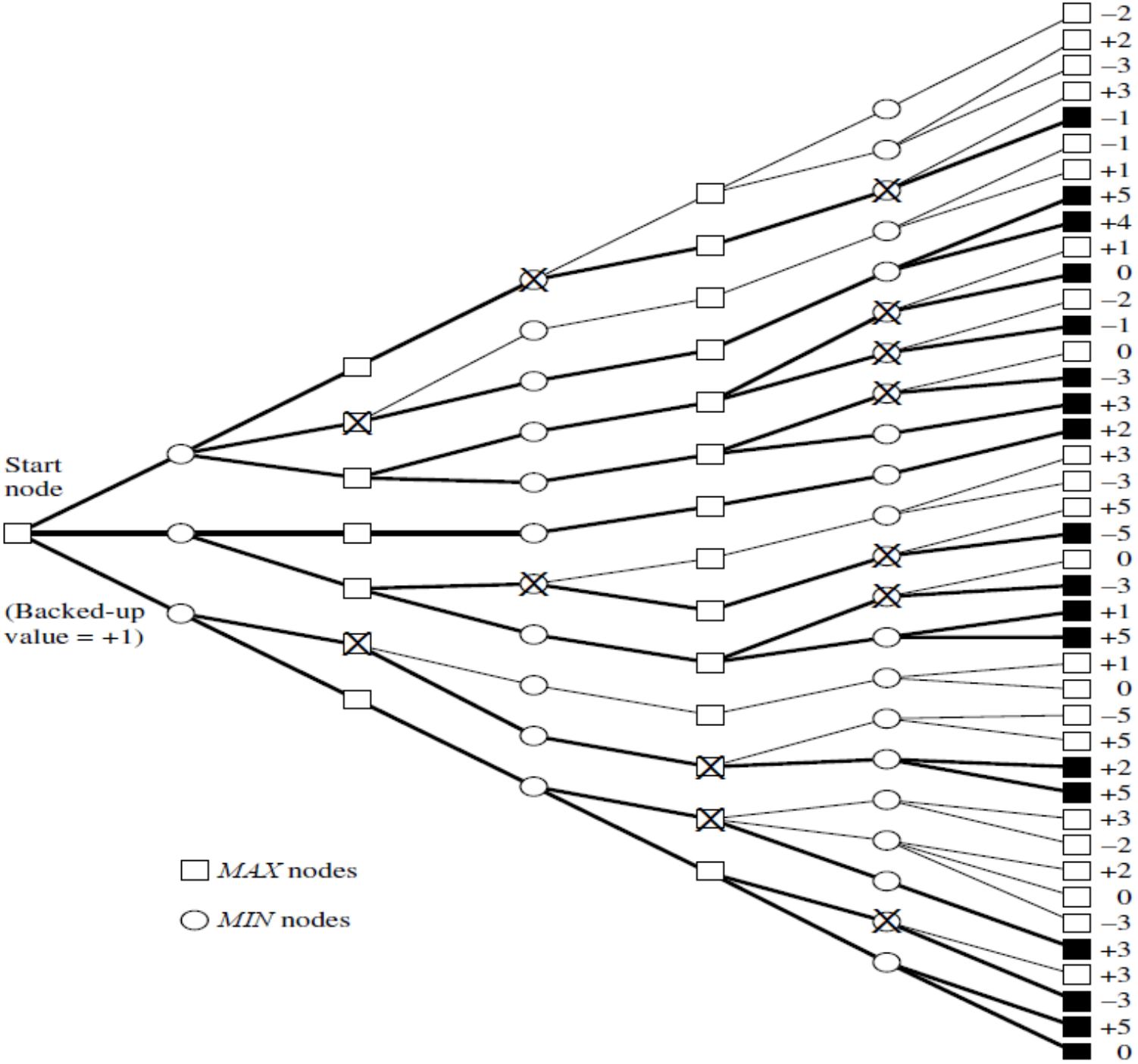


α-β过程（8）

- α-β剪枝技术的一般规律：
  - 任何“或”（MAX）节点x的α值如果不能降低（大于等于）其父节点的β值，则对节点x以下的分枝可以停止搜索，并使x的倒推值为α. 这种技术称为β剪枝
  - 任何“与”（MIN）节点x的β值如果不能升高（小于等于）其父节点的α值，则对节点x以下的分枝可以停止搜索，并使x的倒推值为β. 这种技术称为α剪枝
  - 要进行α-β剪枝，至少必须使某一部分的搜索树生长到最大深度. 因为α和β值必须以某个端节点的静态估值为依据. 因此采用α-β过程都要使用某种深度优先的搜索方法.


0
d
b
0
n
3
a
0
c
-3
e
3
0 5 -3 3 3 -3 0 2 2 -3 0 -2 3 5 4 1 -3 0 6 8 9 -3


0 1
d 0
m 1
b 0
n 3
h 1
k 6
a 0
c -3
e 3
-3
-3
-2
g 1
i -3
j 6
-3
0 5 -3 3 3 -3 0 2 2 -3 0 -2 3 5 4 1 -3 0 6 8 9 -3


Start
node
(Backed-up
value = +1)
MAX nodes
MIN nodes
−2
+2
−3
+3
−1
−1
+1
+5
+4
+1
0
−2
−1
0
−3
+3
+2
+3
−3
+5
−5
0
−3
+1
+5
+1
0
−5
+5
+2
+5
+3
−2
+2
0
−3
+3
+3
−3
+5
0

α-β过程（9）

- 特性
  - 剪枝并不影响最后的结果 $\sqrt{b}$
  - 好的招数序列可以改进剪枝的效率
  - 如果可以找到最好的剪枝，则时间复杂度为： $O(b^{m/2})$
    - 有效的分支系数为 $\sqrt{b}$
    - α-β搜索向前看的走步数是极大极小搜索的两倍

- 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。
- 现将图3-19左边所示的一部分重画在图3-20中。

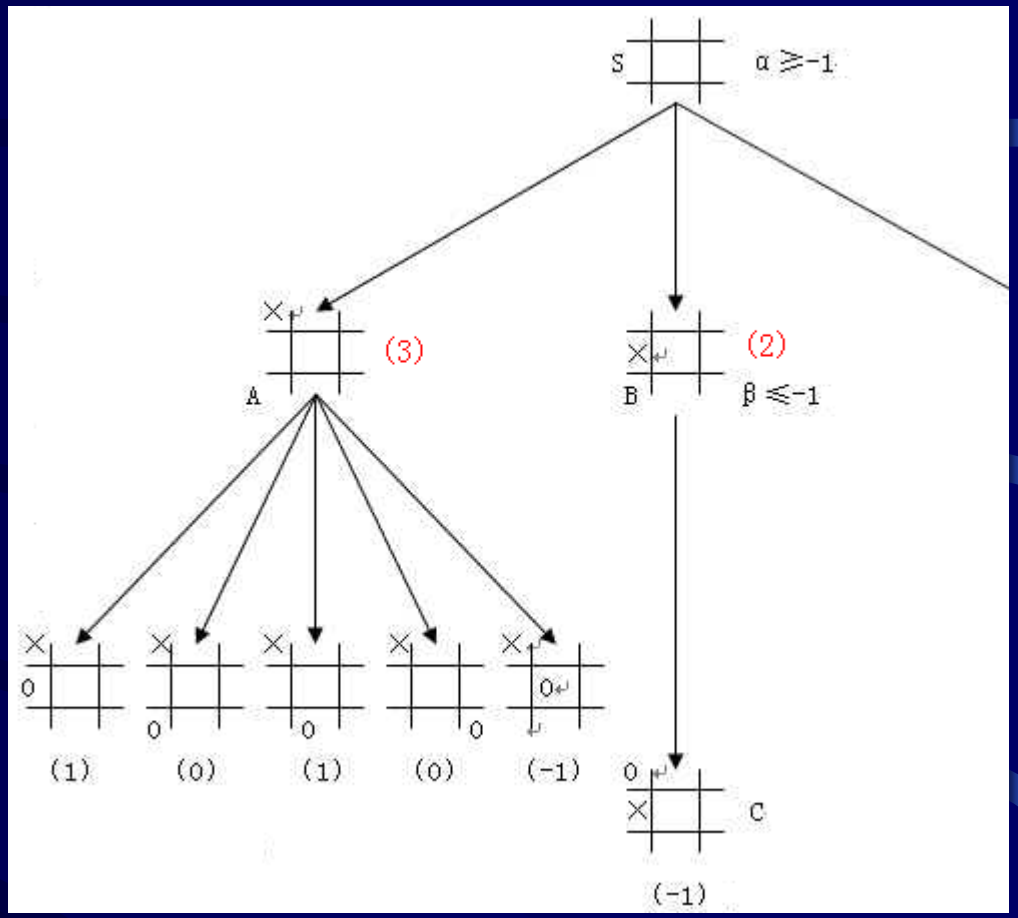


图3-20 一字棋博弈的α－β过程

- 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。

- 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
- 在图3-20中作为MIN节点B的β值小于等于B的前辈MAX节点S的α值，从而B的其他后继节点可以不必再生成。

- 图3-20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。

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3.5 一个约束满足问题表述为一个三元组（V，D，C），

V——n个变量的集合V={v1，……，vn}，

D——变量vi （i=1，2，……，n）相应的取值集合D={D1，……，Dn}，

C——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vi的取值集合Di（0≤i≤n）都是离散的和有限的。

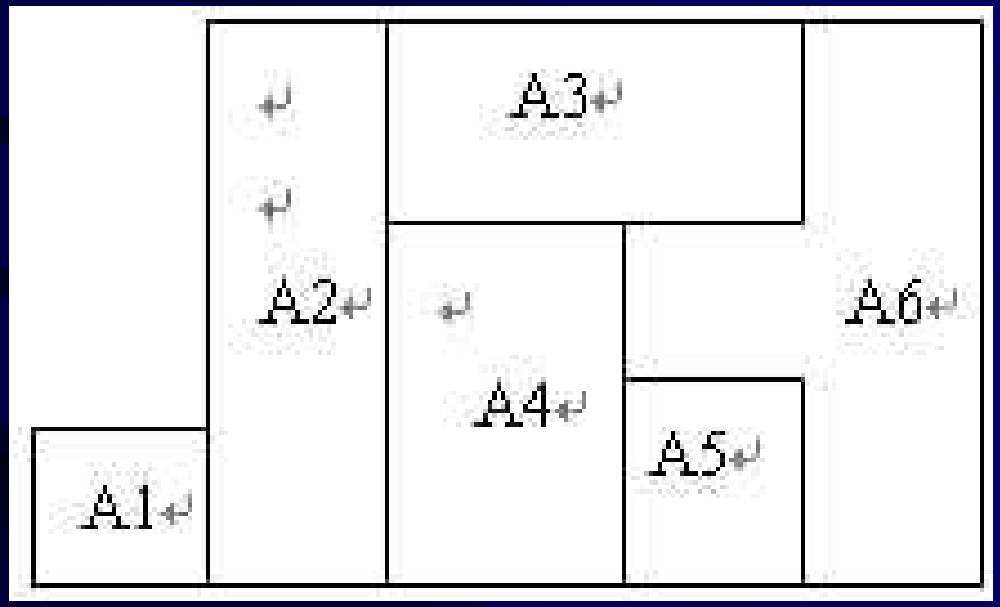
（2）C为约束的有限集合，约束是一个或多个对象属性间的数学或逻辑关系。如，把10元钱换成1、2、5元钱，1、2、5元钱为对象，对象的个数为各自的属性，数学关系可以表示为：c1 + 2c2 + 5c3 = 10。如果问题的每个约束仅涉及两个变量，则称为二元约束问题（BCSP）。

（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。

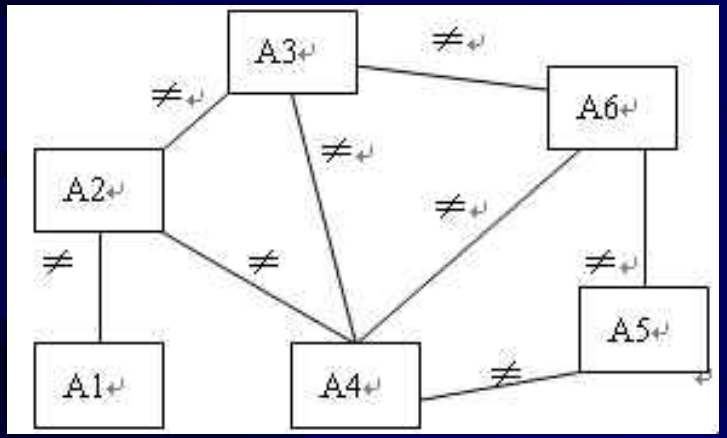
- 目前约束推理的研究主要集中在两个方面：
  - 约束搜索
  - 约束语言
- 约束搜索主要研究有限域上的约束满足，对有限域而言，约束满足问题一般是一个NP难问题，因此不可能存在一个线性时间的算法能够找到所有的解答。

例3 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。



假设的地图

- 用图3-25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）}

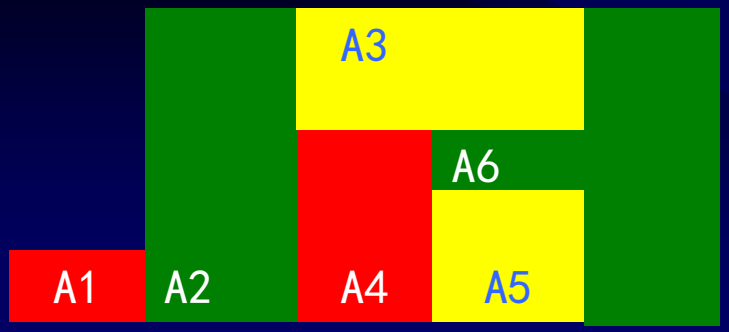
Step3：{A1←R，A2←G，（A3←Y，A4≠Y，A6≠Y），A4≠G}

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

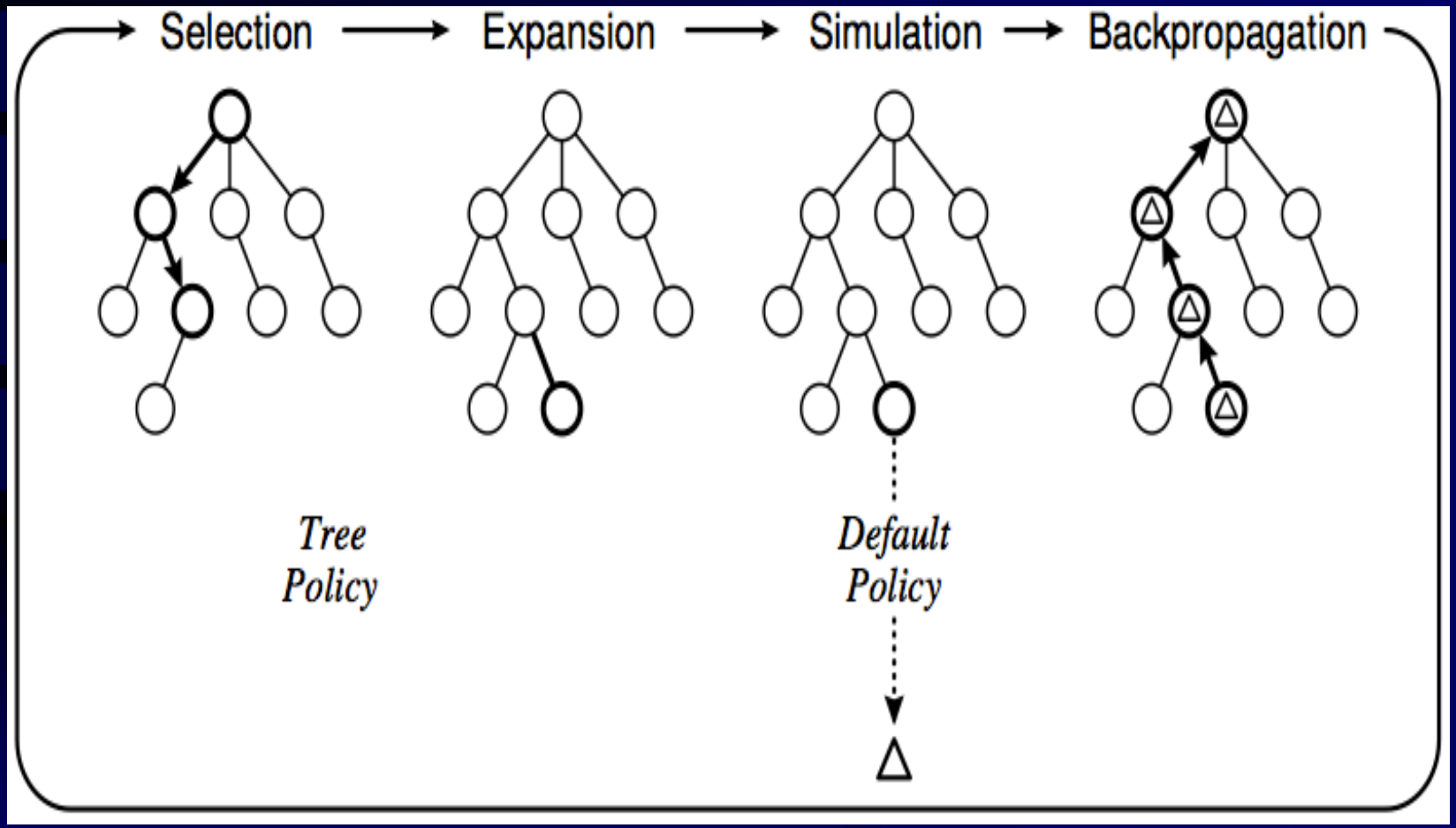
Step5：{A1←R，A2←G，A3←Y，A4←R，（A5←Y，A6≠Y），A6≠R}

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。



蒙特卡洛树搜索


Selection
Expansion
Simulation
Backpropagation
Tree Policy
Default Policy

选择(Selection)

- 在选择阶段，需要从根节点，也就是要做决策的局面R出发向下选择出一个最急迫需要被拓展的节点N，局面R是是每一次迭代中第一个被检查的节点；对于被检查的局面而言，他可能有三种可能：
  - 该节点所有可行动作都已经被拓展过
  - 该节点有可行动作还未被拓展过
  - 这个节点游戏已经结束了

选择(Selection)

- 对于这三种可能：
  - 如果所有可行动作都已经被拓展过了，那么我们将使用UCB公式计算该节点所有子节点的UCB值，并找到值最大的一个子节点继续检查。反复向下迭代。
  - 如果被检查的局面依然存在没有被拓展的子节点，那么我们认为这个节点就是本次迭代的的目标节点N，并找出N还未被拓展的动作A。执行模拟。
  - 如果被检查到的节点是一个游戏已经结束的节点。那么从该节点直接执行反向传播。

选择(Selection)

- 信任度上限树(Upper Confidence bound applied to Trees(UCT))
  - N表示总模拟次数，W表示胜局次数。每次都选择胜率最大的节点进行模拟。但是这样会导致新节点无法被探索到。为了在最大胜率和新节点探索上保持平衡，UCT（Upper Confidence Bound，上限置信区间算法）被引入。

$$\frac{w_i}{n_i} + c * \sqrt{\frac{lnN_i}{n_i}}$$

拓展(Expansion)

- 在选择阶段结束时候，我们查找到了一个最迫切被拓展的节点N，以及他一个尚未拓展的动作A。在搜索树中创建一个新的节点Nn作为N的一个新子节点。Nn的局面就是节点N在执行了动作A之后的局面。

模拟(Simulation)

- 为了让Nn得到一个初始的评分。我们从Nn开始，让游戏随机进行，直到得到一个游戏结局，这个结局将作为Nn的初始评分。一般使用胜利/失败来作为评分，只有1或者0。

反向传播(Back Propagation)

- 在Nn的模拟结束之后，它的父节点N以及从根节点到N的路径上的所有节点都会根据本次模拟的结果来添加自己的累计评分。

- 分油问题
- 一个一斤的瓶子装满油，另有一个七两和一个三两的空瓶，再没有其他工具。只用这三个瓶子怎样精确地把一斤油分成两个半斤油。

（10，7，3）三个瓶子装满油的情况

- 初始状态（10，0，0）
- 目标状态（5，5，0）

第3章 搜索技术

3.1 概述

3.2 盲目搜索方法

3.3 启发式搜索

3.4 问题归约和AND-OR图启发式搜索

3.5 博弈

3.6 案例分析

3.6 案例分析

3.6.1 八皇后问题

3.6.2 洞穴探宝

3.6.3 五子棋

八皇后问题

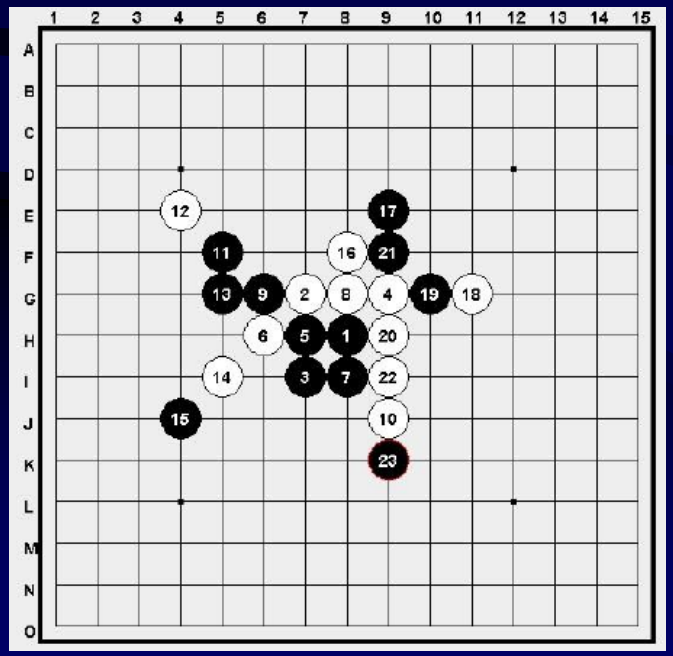
- 在皇后问题中，要把N个皇后放入一个N×N的方格棋盘中，并保证任意两个皇后都不在同一行，同一列或同一对角线上。

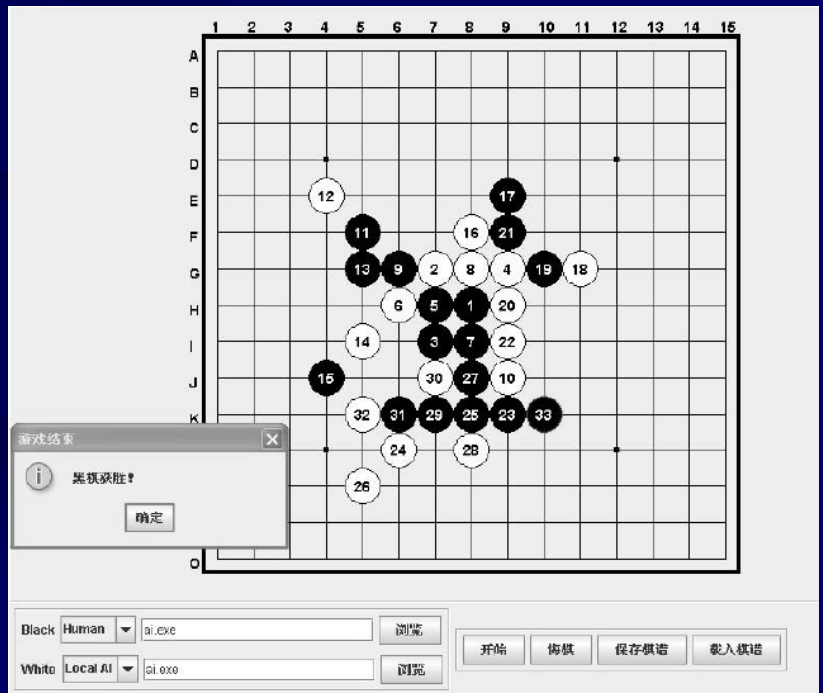
洞穴探宝

- 传说有一位探险家听人说在一个洞穴中藏着大量金银财宝，以前曾有许多人试图找出这些财宝，可劳而无功。该洞穴是一个地下长廊的迷宫，连接着各种不同的洞穴，其中有些洞穴住有鬼怪和山盗。幸好财宝都在同一洞穴中。现问：哪条路既能使那位探险家不受到伤害又能找到财宝？采用图搜索技术，编程完成这个任务。

五子棋

- 五子棋一方执黑子，一方执白子，轮流行棋，哪方无论横线、竖线还是斜线方向先连成五者为胜。这里规定：采用国际上标准的 15×15路线的正方形棋盘；两人（机）分别执黑白两色棋子，轮流在棋盘上选择一个无子的交叉点走子，无子的交叉点又称为空点；由黑方先行走棋。给出五子棋算法设计及主要实现技术。





小结

介绍了深度优先和宽度优先算法，
分析了它们的复杂性。
当状态空间比较大的时候，宽度优先是不合适的。
深度优先有着很多的应用。
深度优先不是一种完备的方法。
迭代加深搜索在这种情况下更适合一些。
和A*算法结合，迭代加深搜索得到IDA*算法。

最流行的是A*和AO*算法。

A*算法用于或（OR）图，

AO*算法用于与或（AND-OR）图。

A*算法用于状态空间中寻找目标，以及从起始结点到目标结点的最优路径问题。

AO*算法用于确定实现目标的最优路径。并且已经有人通过机器学习的方法增强状态空间中结点的启发式信息，对A*算法进行扩展。

博弈问题，看作是一种特殊的与或搜索问题。

给出了极大极小方法和α－β剪枝技术。

讨论了约束满足问题。

3.7思考题

1. 编写程序，输入为两个网页的URL，找出从一个网页到另一个网页的链接路径。用哪种搜索策略最适合？双向搜索适用吗？能用搜索引擎实现一个前驱函数吗？

2. 中国象棋。查阅资料，并与中国象棋计算机程序对弈，阐述程序实现的主要技术。

3. 国际象棋。扩展阅读资料，阐述国际象棋中的关键技术。

4. 围棋。扩展阅读资料，阐述围棋中的关键技术。

5. 桥牌。扩展阅读资料，阐述桥牌中的关键技术。

扩展阅读

- 2016年3月15日，由Google DeepMind开发的人工智能围棋程序AlphaGo（阿法狗）以4:1战胜围棋世界冠军李世石。在五番棋的对抗过程中，从观战的超一流棋手讨论和反应可以发现，AlphaGo的着法有些超出了他们的预料，但事后评估又认为是好棋。有棋手就认为，人类真的可以向计算机学习围棋，加深对围棋规律的理解，觉得“它可能比我们更接近围棋之神”。这意味着AlphaGo的深度学习算法，甚至可以从大数据中发现人类千百年来还未发现的规律和知识，为人类扩展自己知识体系开辟了新的认识通道。

- AlphaGo的关键技术包括：
- **（1）局部落子和整体棋局着法的平衡能力。**AlphaGo通过深度学习产生出来的策略网络（或称走棋网络），在对抗过程中可以实现局部着法的优化；通过估值网络实现对全局的不间断的评估，用于判定每一步棋对全局棋胜负的影响。而且，还可以通过快速走子算法和蒙特卡洛树搜索机制，加快走棋速度，实现对弈质量和速度保证的合理折衷。这些技术使得计算机初步可以既考虑局部得失，又考虑全局整体胜负。而这种全局性“直觉”平衡能力，正是过去我们认为是人类独有的。
- 蒙特卡洛树搜索的方法，边模拟边建立一个搜索树，父节点可以共享子节点的模拟结果，以提高搜索的效率。包括以下4个过程：选择——以当前棋局为根节点，自上而下地选择一个落子点；扩展——向选定的节点添加一个或多个子节点；模拟——对扩展出的节点用蒙特卡洛方法进行模拟；回传——根据模拟结果依次向上更新祖先节点的估计值。

- **（2）全新的学习能力。**AlphaGo的核心技术是深度学习方法，它的围棋知识不是像“深蓝”那样是编在程序里的，而是它通过大量棋谱和自我对弈自己学会掌握的。而且，这种学习能力具有相当大的通用性。AlphaGo将深度学习方法引入到蒙特卡洛树搜索中，主要设计了策略网络和估值网络等2个深度学习网络。这样就可以在规定的时间内，实现更多的搜索和模拟，从而达到提高围棋程序下棋水平的目的。此外，AlphaGo还把增强学习引入到计算机围棋中，通过不断的自我学习，提高其下棋水平。自己学习的能力，使得计算机有了进化的可能；而通用性，则使其不再局限在围棋领域。

THE END