基于树的搜索

深圳大学计算机与软件学院
卢亚辉

主要内容

- 搜索介绍
- 广度优先搜索求解八数码问题
- 回溯法求解N皇后问题
- 分支限界法求解背包问题
- 其他方法

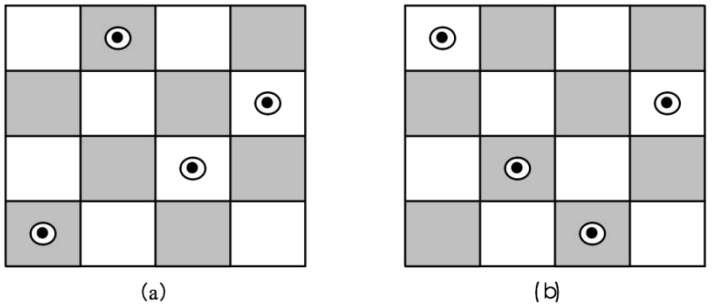
搜索与问题求解

- 问题求解过程是搜索答案(目标)的过程
- 所以问题求解技术也叫搜索技术—通过对状态空间的搜索而求解问题的技术
  - 是一种基于目标的搜索
  - 在寻找到达目标的过程中，当面对多个未知的选项时，首先检验各个不同的导致已知评价的状态的可能行动序列，然后选择最佳序列—这个过程就是搜索

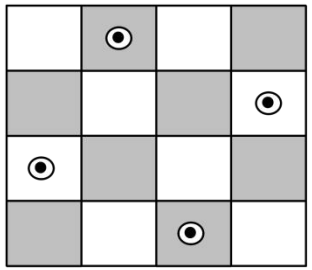
N皇后问题

在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

n＝1 显而易见。n＝2、3，问题无解。n>=4 时，以4后为例



4后问题的两种无效布局



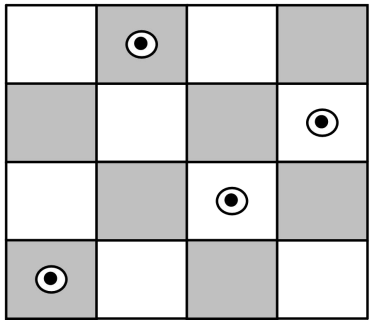
4后问题的一个有效布局

解空间

- 解空间的表示方法：图形表示，集合表示，向量空间表示，状态空间树表示

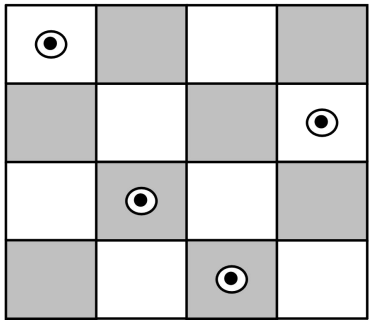
向量 $x=(x_1,x_2,x_3,x_4)$ 表示皇后的布局。分量x_i 表示第 i 行皇后的列位置。

x_i的取值范围 $S_i=\{1,2,3,4\}$ ， 有 4^4 个可能解，（2,4,1,3）是一个可行解（c图）



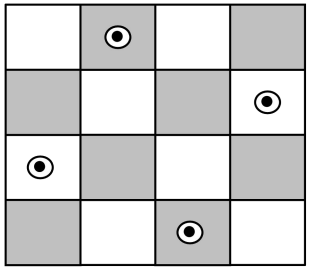
（a）

(2,4,3,1)



(b)

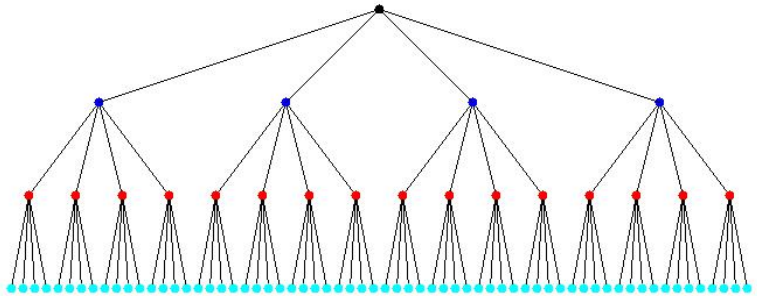
(1,4,2,3)



(2,4,1,3)

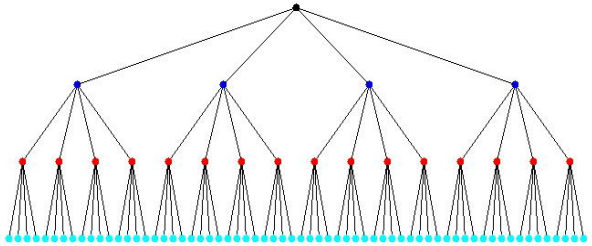
状态空间树

- 状态空间树看成为一棵高度为 n 的树，
- 第0层有 $|S_0|=m_0$ 个分支结点，构成 m_0 棵子树，每一棵子树都有 $|S_1|=m_1$ 个分支结点。
- 第1层，有 $m_0 \times m_1$ 个分支结点，构成 $m_0 \times m_1$ 棵子树。
- 第 $n-1$ 层，有 $m_0 \times m_1 \cdots \times m_{n-1}$ 个结点，它们都是叶子结点。
- 所有叶子节点构成一个解空间



可能解，可行解，最优解

4后问题状态空间树：4叉完全树
完全四叉树的所有叶子节点构成一个解空间，
每个叶子节点就是一个可能解，
满足约束要求的叶子节点就是可行解，
最优解就是所有解中最符合目标函数的那个叶子节点
约束方程：
　不在同一列　$x_i \neq x_j \quad 1 \le i \le 4, 1 \le j \le 4, i \neq j$
　不在同一个斜线上　$|x_i - x_j| \neq |i - j|$
目标函数：
数字最小：min（x1+x2+x3+x4）



求解过程—树搜索

- 求解问题的过程使用搜索树形式
  - 每个状态对应搜索树中一个节点
  - 根节点对应于初始状态
  - 每次从搜索树的上层节点出发，根据约束条件进入下一个可能的状态，即展开新的一层树节点—节点扩展
  - 节点展开的顺序即代表了不同的搜索策略
  - 当展开的节点为目标状态时，就找到了问题的一个解
  - 如果一个节点不满足约束，或者不可能存在目标状态，那么我们不再搜索这个节点以下的子树，这叫做剪枝

哲学思想

- 每个人的解决问题的方法。
- 1）碰到问题，思考所有的解决方法（思考的周密性——全面）
- 2）所有的这些解决方法的组织，关系等（思考的系统性——条理）
- 3）排除不可行，不实际的方法（思考的选择性——优化）
- 解决问题！

搜索的分类

- 搜索过程的分类：状态空间搜索(图搜索方式)，问题空间搜索(层次方法)，博弈空间搜索
- 搜索策略分类：无信息搜索与启发式搜索
- 启发式：利用中间信息改进控制策略
  - (1) 任何有助于找到问题的解，但不能保证找到解的方法是启发式方法
  - (2) 有助于加速求解过程和找到较优解的方法是启发式方法

广度优先搜索

- 广度优先搜索过程：
  - 首先扩展根节点
  - 接着扩展根节点的所有后继节点
  - 然后再扩展后继节点的后继，依此类推
  - 在下一层任何节点扩展之前搜索树上的本层深度的所有节点都已经被扩展

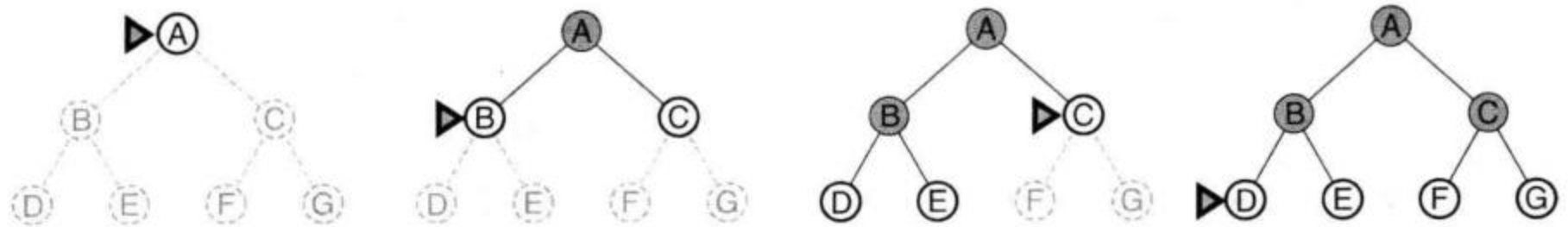
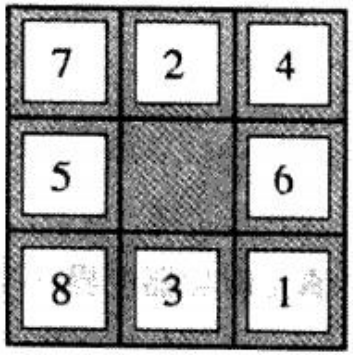


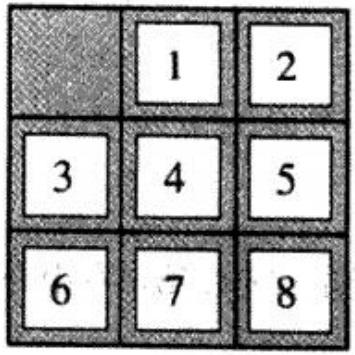
图 3.12　一棵简单二叉树上的宽度优先搜索。在每个阶段，用一个记号指出下一个将要扩展的结点

八数码游戏

- ◇ **状态**：状态描述指定了 8 个棋子中的每一个以及空位在棋盘的 9 个方格上的分布。
- ◇ **初始状态**：任何状态都可以被指定为初始状态。注意要到达任何一个给定的目标，可能的初始状态中恰好只有一半可以作为开始（习题 3.4）。
- ◇ **后继函数**：用来产生通过四个行动（把空位向 *Left*、*Right*、*Up* 或 *Down* 移动）能够达到的合法状态。
- ◇ **目标测试**：用来检测状态是否能匹配图 3.4 中所示的目标布局。（其它目标布局也是可能的。）
- ◇ **路径耗散**：每一步的耗散值为 1，因此整个路径的耗散值是路径中的步数。



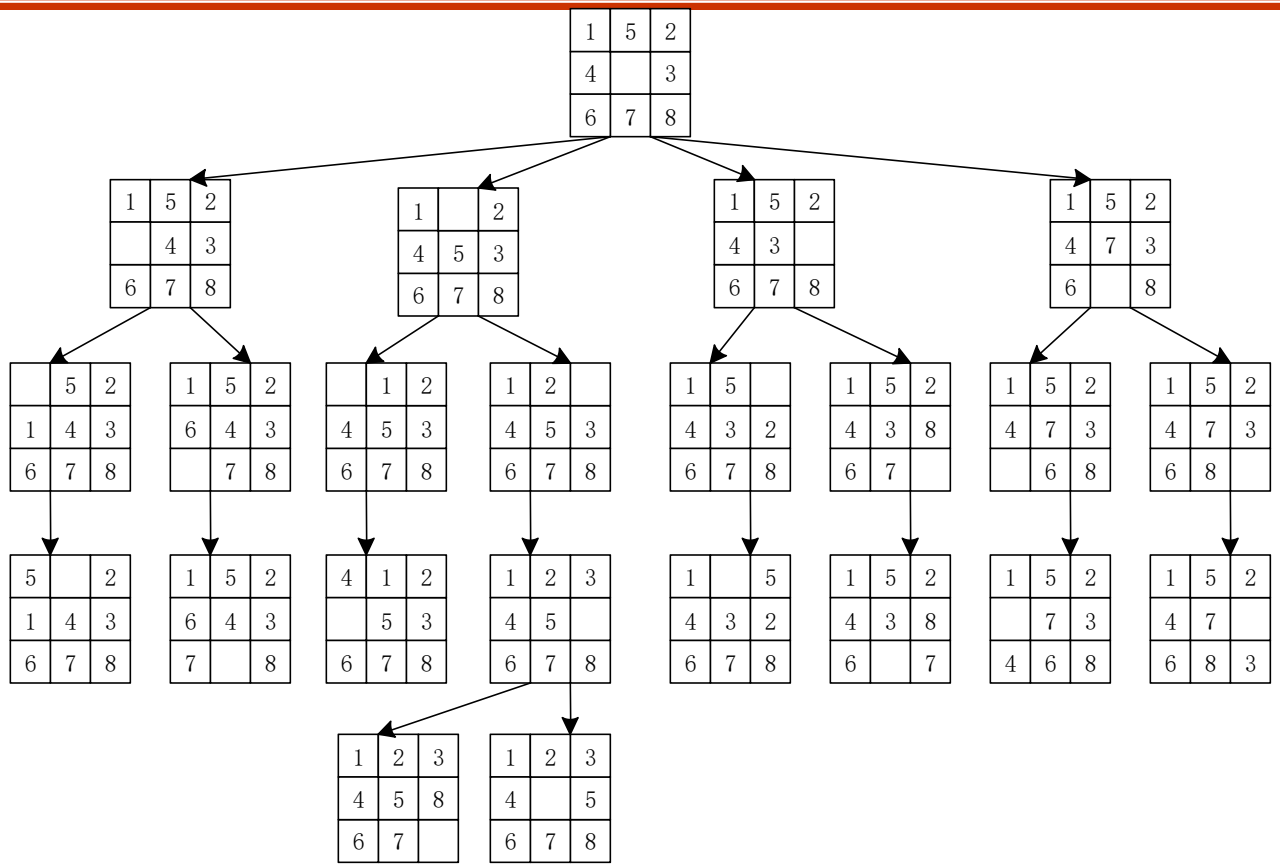
起始状态



目标状态

八数码问题属于**滑块问题**家族，这类问题经常被用作 AI 中新的搜索算法的测试用例。滑块问题为 NP 完全问题，因此不要期望能找到在最坏情况下明显好于本章和下一章所描述的搜索算法的方法。八数码问题共有 9!/2 = 181 440 个可达到的状态，并且很容易求解。15 数码问题（在 4×4 的棋盘上）有大约 1.3 万亿个状态，用最好的搜索算法求解一个随机的实例的最优解需要几毫秒。24 数码问题（在 5×5 的棋盘上）的状态数可达 10^{25} 个，求解随机实例的最优解可能需要几个小时。

八数码问题的广度优先搜索



广度优先搜索的算法

```
public void breadthFirstSearch() {
    ArrayDeque<Map<String, Object>> nodeDeque =  new ArrayDeque<Map<String, Object>>();
    Map<String, Object> node =  new HashMap<String, Object>();
     /**
      * 第一个节点放入
      */
    nodeDeque.add(node);
     while (!nodeDeque.isEmpty()) {
       node = nodeDeque.pop();
       log.info(node);
       List<Map<String, Object>> list = getChildren(node);
        if (!CollectionUtil.isBlank(list)) {
            for (Map<String, Object> child : list) {
              nodeDeque.add(child);
            }

        }
    }
}
```

可以参考书上对于图的BFS算法
核心思想是使用队列Queue实现FIFO

广度优先搜索的性能

- 从4种度量来评价广度优先搜索：
  - 完备性—总能找到一个解
  - 如果每步扩展的代价相同时，广度优先搜索能找到最优解
  - 内存代价是比执行时间代价更大的问题($O(b^{d+1})$)
  - 指数级的时间代价

深度	节点数	时间	内存
2	1100	0.11 秒钟	1 M字节
4	111,100	11 秒钟	106 M字节
6	10^7	19 分钟	10 G字节(KM字节)
8	10^9	31 小时	1 T字节(KG字节)
10	10^{11}	129 天	101 T字节(KG字节)
12	10^{13}	35 年	10 P字节(KT字节)
14	10^{15}	3,523 年	1 E字节(KP字节)

图 3.11 广度优先搜索的时间和内存需求。显示的数字在如下假设基础上得到：搜索树的分支因子为 $b = 10$；10 000 个节点/秒；1000 字节/节点

深度优先搜索

- 深度优先搜索过程：
  - 总是扩展搜索树的当前扩展分支(边缘)中最深的节点
  - 搜索直接伸展到搜索树的最深层，直到那里的节点没有后继节点
  - 那些没有后继节点的节点扩展完毕就从边缘中去掉
  - 然后搜索算法回退下一个还有未扩展后继节点的上层节点继续扩展
- 回溯搜索：每次只产生一个后继节点而不是所有的后继节点

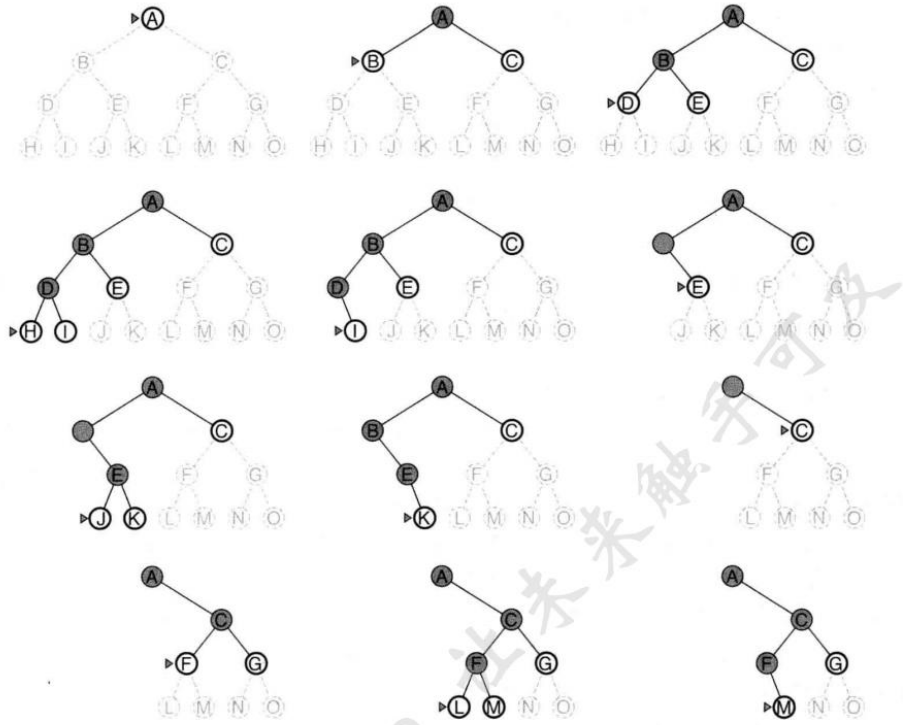


图 3.16　二叉树的深度优先搜索。未探索区域用浅灰色表示。已经被探索并且在边缘中没有后代的结点可以从内存中删除。第三层的结点没有后继并且 M 是唯一的目标结点

DFS算法

```
public void depthFirstSearch() {

    Stack<Map<String, Object>> nodeStack =  new Stack<Map<String, Object>>();
    Map<String, Object> node =  new HashMap<String, Object>();
     /**
      * 第一个节点放入
      */
    nodeStack.add(node);
     while (!nodeStack.isEmpty()) {
       node = nodeStack.pop();
       log.info(node);
       List<Map<String, Object>> list = getChildren(node);
        if (!CollectionUtil.isBlank(list)) {
            for (Map<String, Object> child : list) {
              nodeStack.add(child);
            }
        }
    }

}
```

可以参考书上对于图的DFS算法
核心思想是使用Stack实现FILO

深度优先搜索的性能

- 深度优先搜索通过后进先出队列LIFO(栈)调用Tree-Search实现 / 通常使用递归函数实现，依次对当前节点的子节点调用该函数
- 性能：
  - 内存需求少—如分支因子=b/最大深度=m的状态空间深度优先搜索只需要存储bm+1个节点(比较广度优先$O(b^{d+1})$)
  - 最坏情况下时间复杂性也很高$O(b^m)$

回溯法简介

有“通用解题法”之称，将所有的解（问题的解空间）按照一定结构排列，再进行搜索。

- 一般解空间构造成为为树状结构，用深度优先的策略搜索
- 两种方式：
  - 只需要一个解的话，找到解就停止
  - 需要求所有解，则需做“树的遍历”找到所有解。
- 通常用排除法，减少搜索空间

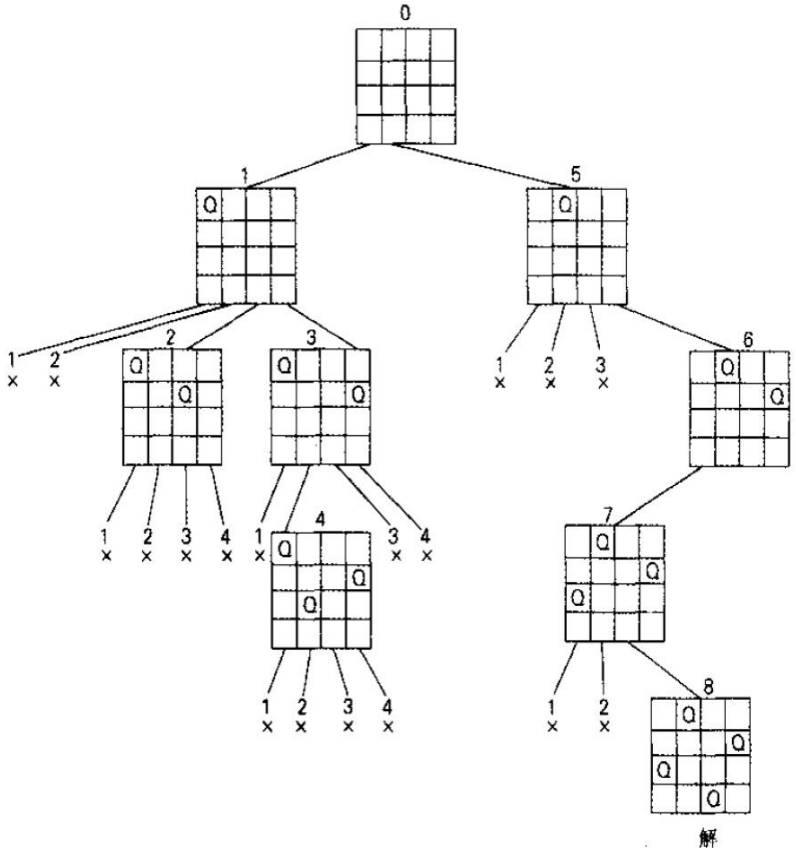
回溯法的基本思想

(1)针对所给问题，定义问题的解空间；
(2)确定易于搜索的解空间结构；
(3)以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

常用剪枝函数：
用约束函数在扩展结点处剪去不满足约束的子树；
用限界函数剪去得不到最优解的子树。

用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为h(n)，则回溯法所需的计算空间通常为O(h(n))。而显式地存储整个解空间则需要$O(2^{h(n)})$或O(h(n)!)内存空间。

求解过程图示



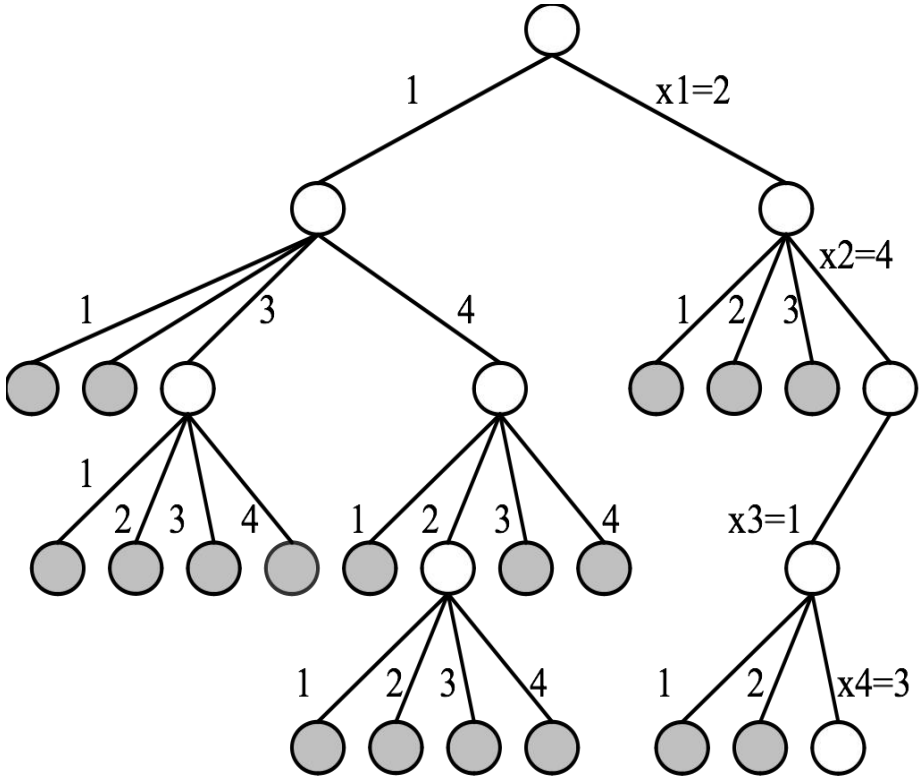


图 11.2 用回溯法解 4 皇后问题的状态空间树。x 表示一个试图把皇后放在指定列的不成功的尝试。节点上方的数字指出了节点被生成的次序

扩展节点的方法

- 扩展结点:一个正在产生儿子的结点称为扩展结点
- 活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
- 死结点:一个所有儿子已经产生的结点称做死结点
- 深度优先的问题状态生成法：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）
- 宽度优先的问题状态生成法：在一个扩展结点变成死结点之前，它一直是扩展结点
- 为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数(bounding function)来处死那些实际上不可能产生所需解的活结点，以减少问题的计算量。

- 通过应用范例学习回溯法的设计策略。

- （1）装载问题；
- （2）批处理作业调度；
- （3）符号三角形问题
- （4）n后问题；
- （5）0-1背包问题；
- （6）最大团问题；
- （7）图的m着色问题
- （8）旅行售货员问题
- （9）圆排列问题
- （10）电路板排列问题
- （11）连续邮资问题

深度有限搜索

- 深度优先搜索的无边界问题可以通过提供一个预先设定的深度限制l来解决
  - 深度=l的节点当作无后继节点看待
  - 虽然解决了无限路径问题，但如果l<d则找不到解
  - 如果选择l>d则深度有限搜索也不是最优的
  - 时间复杂度O(b^l)
  - 空间复杂度O(bl)
  - 深度优先搜索可看作是一种特例即$l=\infty$

迭代深入搜索（或迭代深入深度优先搜索）是一个用来寻找最合适的深度限制的通用策略，它经常和深度优先搜索结合使用。它的做法是不断地增大深度限制——首先为 0，接着为 1，然后为 2，依此类推——直到找到目标节点。当深度限制达到 d，即最浅的目标节点的深度时，就能找到目标节点。图 3.14 给出了这个算法。迭代深入搜索算法结合了深度优先搜索和广度优先搜索的优点。它的空间需求和深度优先搜索一样小，为 $O(bd)$。它和广度优先搜索一样当分支因子有限时是完备的，当路径耗散是节点深度的非递增函数时则是最优的。图 3.15 表示了在一个二叉搜索树上 ITERATIVE-DEEPENING-SEARCH 函数 4 次迭代的情

一般来讲，当搜索空间很大而且解的深度未知时，迭代深入搜索是一个首选的无信息搜索方法。

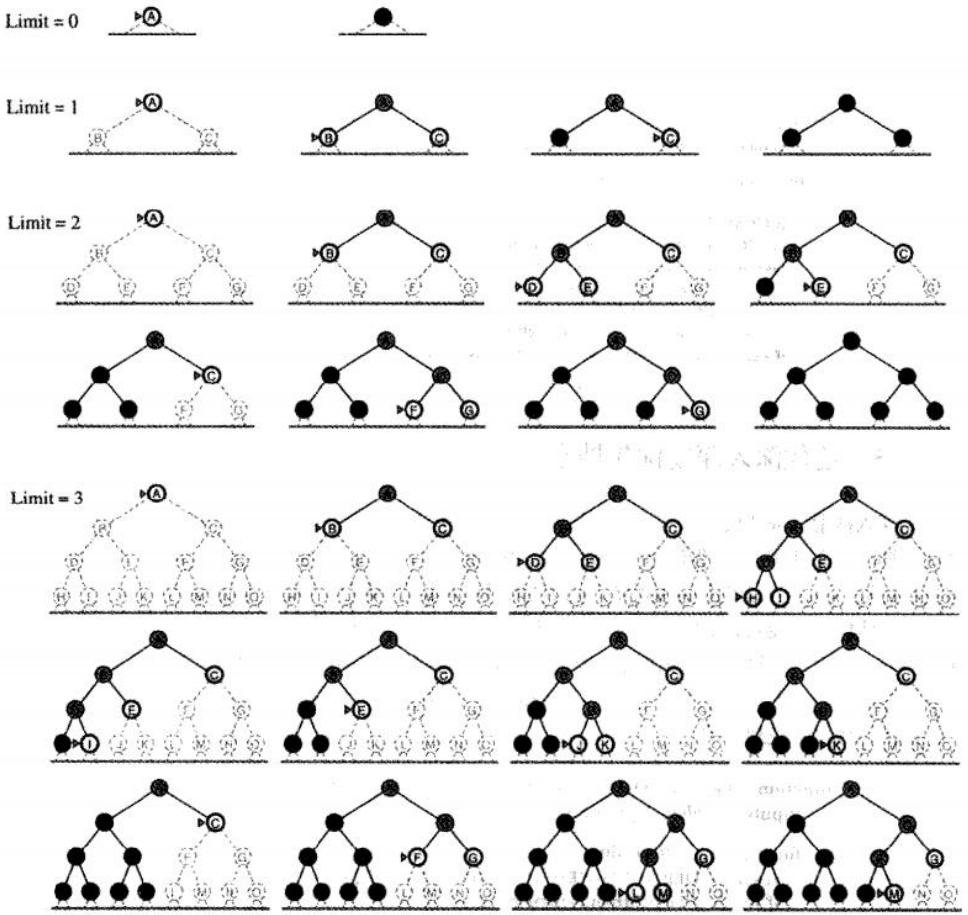


图 3.15　迭代深入搜索在一棵二叉树上的 4 次迭代

启发式搜索

- 盲目搜索的不足：效率低，耗费过多的计算空间与时间。原因在于——待扩展的节点顺序固定，系统不会对有希望达到目标的节点特别关照。
- 启发式搜索：利用相关领域特征的启发信息来排列待扩展节点的顺序，从而选择“最有希望”的节点作为下一个被扩展的节点，这种搜索策略称之为启发式搜索。

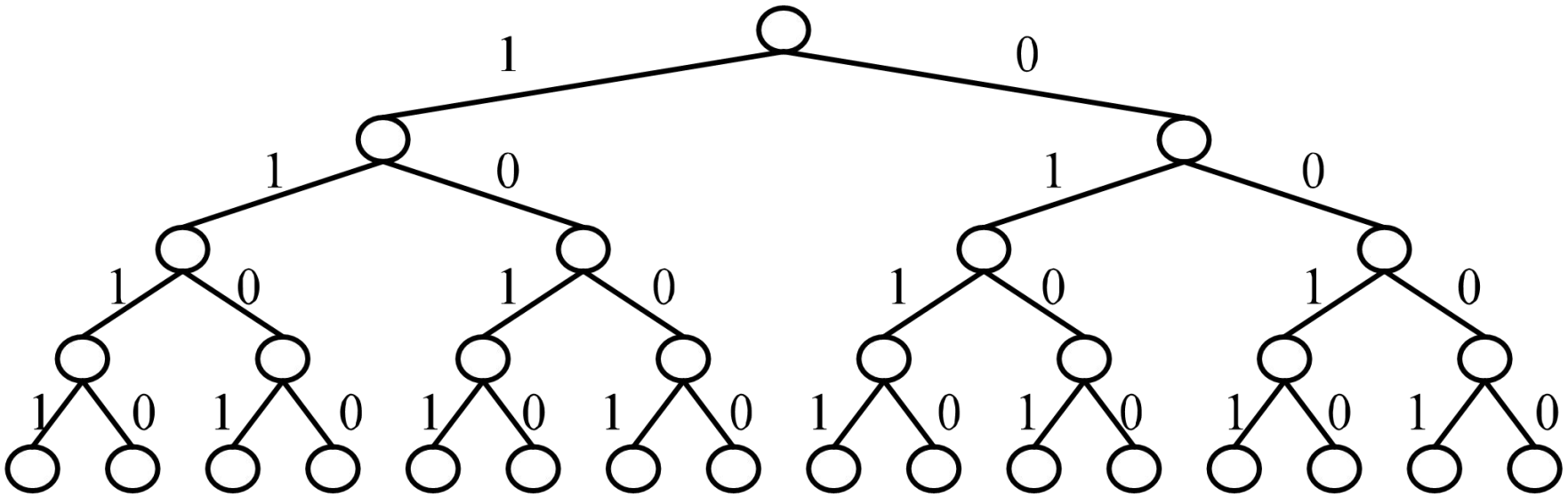
启发式搜索策略基本思想

- 假定有一个估值函数f，可以帮助确定下一个要扩展的最优节点。
- 把当前新状态按估计值从小到大的顺序插入SS中。
- 当下一个节点是目标节点时过程终止。
- 几点说明：
  - 状态的比较是非常费时的，可以考虑一些优化算法来提高效率。
  - 给定估值函数 f 的意义，则有序搜索算法就确定为已知的搜索算法：f=状态节点的深度，则为广度优先搜索；f=状态节点的深度的负数，则为深度优先搜索；
  - 有序搜索算法不一定保证找到解，即使解是存在的；另外，算法找到的解无法保证是最优解——估值函数的选取至关重要！！

- 例：0/1背包问题，$S=\{0,1\}$，当时 $n=3$，0/1背包问题的解空间是：

  {(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)}

  当输入规模为 n 时，有 2^n 种可能的解。



分支限界法

和回溯法相比，分支界限法需要两个额外的条件：

- 对于一棵状态空间树的每一个节点所代表的部分解，我们要提供一种方法，计算出通过这个部分解繁衍出的任何解在目标函数上的最佳值边界[①]。
- 目前求得的最佳解的值。

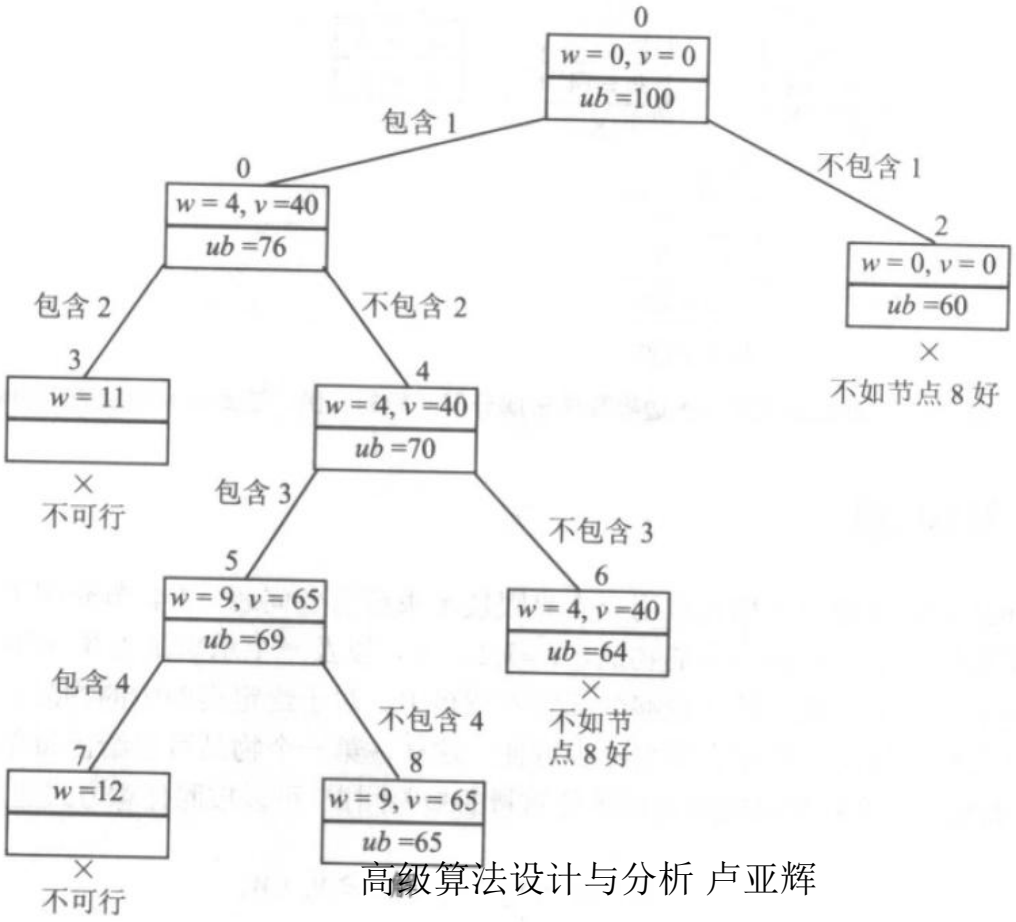
如果可以得到这些信息，我们可以拿某个节点的边界值和目前求得的最佳解进行比较：如果边界值不能超越（也就是说，在最小化问题中不小于，在最大化问题中不大于）目前的最佳解，这个节点就是一个没有希望的节点，需要立即中止（也有人说把树枝剪掉），因为从这个节点生成的解，没有一个能比目前已经得到的解更好。这就是分支界限技术的主要思想。

一般来说，对于一个分支界限算法的状态空间树来说，只要符合下面三种中的一种原因，我们就会中止掉它的在当前节点上的查找路径：

- 该节点的边界值不能超越目前最佳解的值。
- 该节点无法代表任何可行解，因为它已经违反了问题的约束。
- 该节点代表的可行解的子集只包含一个单独的点（因此无法给出更多的选择）。在这种情况下，我们拿这个可行解在目标函数上的值和目前求得的最佳解进行比较，如果新的解更好一些的话，就用前者替换后者。

物品	重量	价值	价值/重量
1	4	40 美元	10
2	7	42 美元	6
3	5	25 美元	5
4	3	12 美元	4

背包的承重量 W 等于 10



现在让我们讨论一下如何应用分支界限技术求解背包问题。3.4 节介绍了背包问题：给定 n 个重量为 w_i、价值为 v_i 的物品，$i=1,2,...,n$，以及一个承重量为 W 的背包，找出其中最有价值的物品子集，并且能够全部装入背包中。对于给定实例中的物品，按照降序对它们的“价值重量比”排序会带来不少方便。这样，第一个物品可以给出每单位重量的最佳回报，而最后一个物品只能给出每单位重量的最差回报，可以按照任意方式建立这种次序：

$$v_1 / w_1 \geq v_2 / w_2 \geq ... \geq v_n / w_n$$

我们可以很自然地用图 11.8 中的一棵二叉树来构造这个问题的状态空间树。树中第 i（$0 \leq i \leq n$）层的每一个节点，都代表了 n 个物品中所有符合以下特征的子集，其中每个子集都包含了根据序列中前 i 个物品所作出的特定选择。这个特定选择是根据从根到该节点的一条路径所惟一确定的：向左的分支表示包含下一个物品，而向右的分支表示不包含下一个物品。我们把这个选择的总重量 w 和总价值 v，记录在节点中，有必要的话，任何向这个选择添加 0 个或者多个物品之后得到的子集的上界 ub 也会被记录下来。

计算上界 ub 的一个简单方法是，把已经选择物品的总价值 v，加上背包的剩余承重量 $W-w$ 与剩下物品的最佳单位回报 v_{i+1} / w_{i+1} 的积：

$$ub = v + (W - w)(v_{i+1} / w_{i+1}) \qquad (11.1)$$

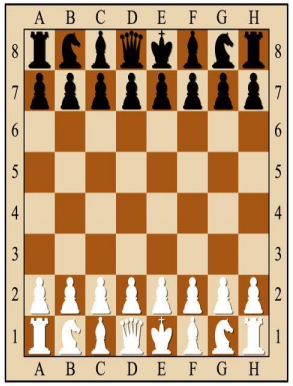
对于它的状态空间树的根来说（参见图 11.8），还没有选择任何物品。因此，已选物品的总重量 w 和它们的总价值 v 都等于 0。由公式（11.1）计算出来的上界值等于\$100。根的左子女节点 1，代表了包含物品 1 的子集。已包含物品的总重量和总价值分别是 4 和\$40；上界的值等于 40+(10−4)*6=\$76。节点 2 代表了不包含物品 1 的子集。相应地，$w=0$，$v=\$0$，而 $ub=0+(10-0)*6=\$60$。对于这个最大化问题来说，由于节点 1 的上界要比节点 2 的上界大，所以它的希望也较大，因此我们首先从节点 1 开始扩展分支。它的子女节点 3 和节点 4，分别代表包含物品 1、包含或者不包含物品 2 的子集。由于节点 3 所代表的每一个子集的总重量超过了背包的承重量，我们可以立即中止节点 3。节点 4 的 w 和 v 与它的父母是相同的；它的上界 ub 等于 40+(10−4)*5 = \$70。我们选择节点 4 而不是节点 2 来开始下一次扩展（为什么？），由此得到了节点 5 和节点 6，它们分别包含和不包含物品 3。我们使用和前面的节点相同的方法，计算这两个节点的总重量、总价值以及上界。从节点 5 扩展分支生成了节点 7 和节点 8，节点 7 代表无可行解，节点 8 只代表一个子集{1,3}。（由于没有其他的物品可供考虑，节点 8[1]的上界就简单地等于这两个物品的总重量。）剩下的活节点 2 和 6 的上界值小于节点 8 所代表的解的值。因此，这两个节点都可以中止掉，而节点 8 所代表的子集{1,3}就是该问题的最优解。

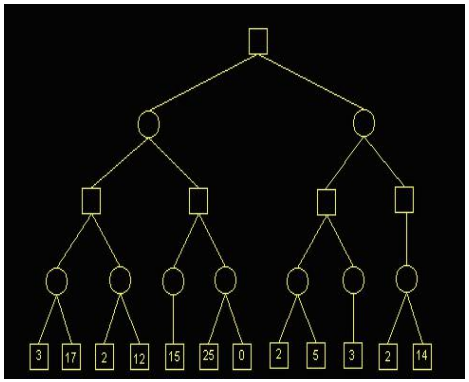
用分支界限算法求解背包问题具有相当不寻常的特性。一般来说，一棵状态空间树的中间节点并不能确定问题的查找空间中的一个点，因为解的某些分量还未明确（作为一个例子，可以看看前面小节中所讨论的分配问题的分支界限树）。然而，对于背包问题来说，树的每一个节点都可以代表给定物品的一个子集。在生成了树中的每一个新节点之后，我们可以利用这个事实，来更新目前为止的最佳子集信息。如果对上面研究的实例采用这种做法，我们就可以在节点 8 生成之前中止节点 2 和节点 6，因为它们都小于节点 5 所代表的子集的价值。

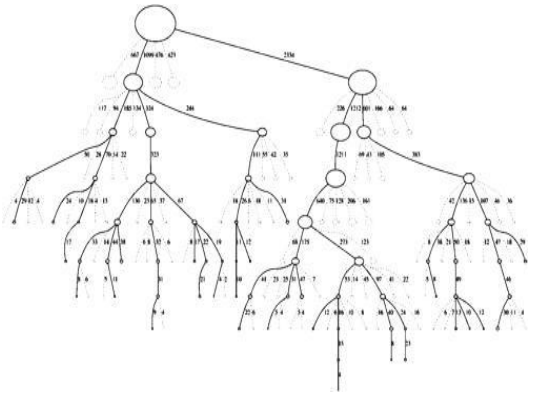
博弈搜索问题与方法

- 从智能体角度看，博弈是多智能体之间的竞争和对抗 / 在竞争的环境中，每个智能体的目的是冲突的，由此引出对抗搜索问题—称为博弈
- 两个游戏者的博弈可以定义为一类搜索问题，其中包括：
  - 初始状态—棋盘局面和哪个游戏者出招
  - 后继函数—返回(招数, 状态)对的一个列表，其中每对表示一个合法招数和相应的结果状态
  - 终止测试—判断游戏是否结束
  - 效用函数—或称目标函数，对终止状态给出一个数值如输赢和平局(以-1/+1/0表示)
  - 双方的初始状态和合法招数定义了游戏的博弈树—此为博弈搜索

国际象棋的求解







- 国际象棋AI战胜人类棋手：符合人类直觉的**传统算法**
  - 国际象棋AI的算法和人类弈棋思维差不多，但算得快、算得深
- 国际象棋等传统棋类算法的核心概念：**局面评估函数**
  - 局面评估函数可以静态计算（仅依赖当前局面），不用往前搜索。
  - 局面评估函数和真实的局面情况，存在误差，如将黑胜的结果误判为白胜。好的函数误差小，适应性强（没有明显bug，常见局面不出错）。
  - 从一个节点往下搜索是改善这个节点评分的办法：多算胜，少算不胜。
  - 利用局面评估函数进行搜索优化，是棋类博弈算法的核心工作。如Alpha-Beta剪枝算法，意思是损失巨大的分支就不用搜索了。
- 局面评估剪枝，搜索优化。将搜索树规模降到计算机算力之内。
- 搜索代码+人类知识库开局库。硬件工程师+软件算法工程师+专业棋手。

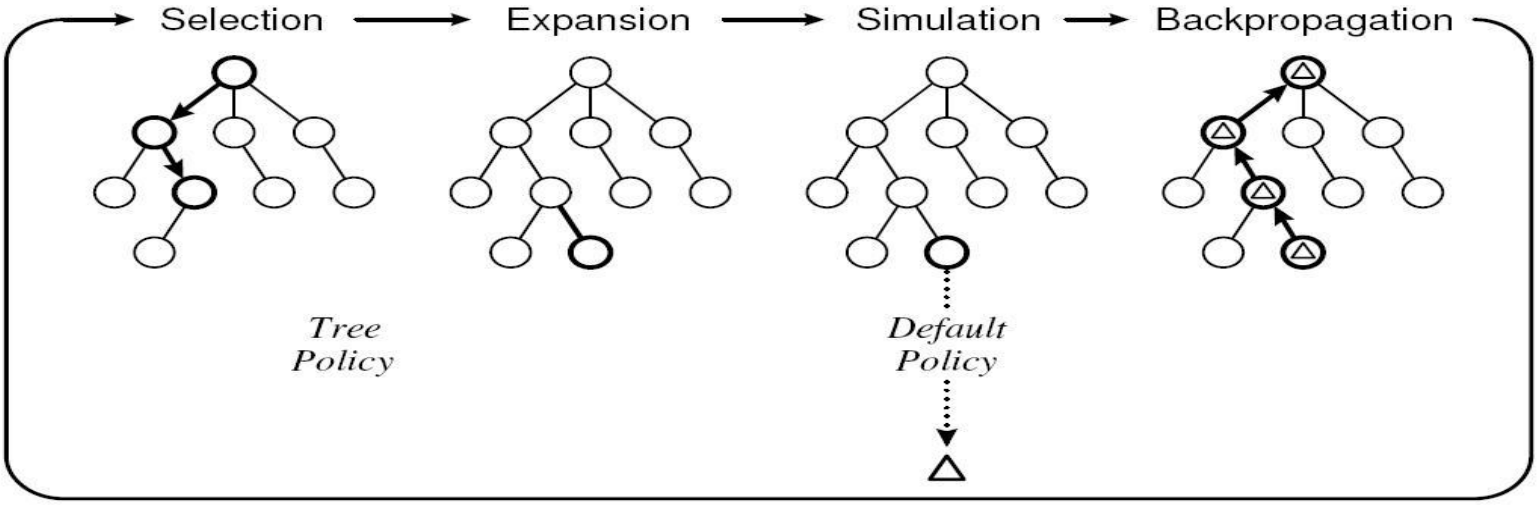
局部剪枝搜索

- 基本思想—与只从一个单独的起始状态出发不同，局部剪枝搜索从k个随机生成的状态开始，每步生成全部k个状态的所有后继状态 / 如果其中之一是目标状态，算法停止；否则从全部后继状态中选择最佳的k个状态继续搜索
- 在局部剪枝搜索过程中，有用的信息在k个并行的搜索线程之间传递—算法会很快放弃没有成果的搜索而把资源放在取得最大进展的搜索上

随机剪枝搜索

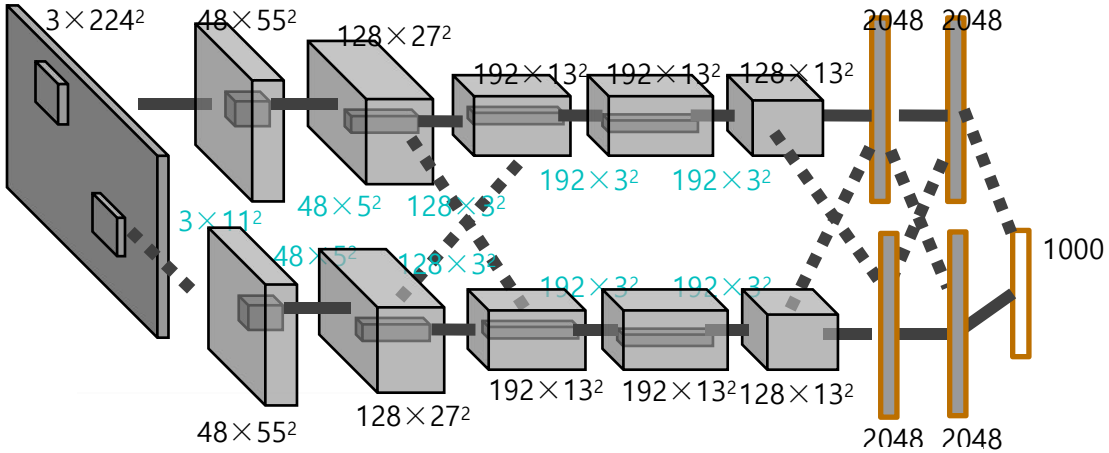
- 如果k个状态缺乏多样性，则局部剪枝搜索会受其影响，性能变差
- 算法的变种—随机剪枝搜索帮助缓解这一问题—随机剪枝搜索不是选择最好的k个后代，而是按照一定概率随机地选择k个后继状态 / 选择给定后继状态的概率是状态值的递增函数
  - 类似于自然选择过程—状态对应生物体，其值对应于适应性，后代就是后继状态

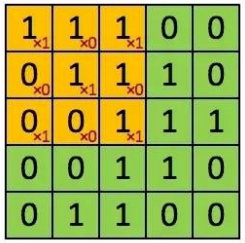
MCTS蒙特卡洛树搜索

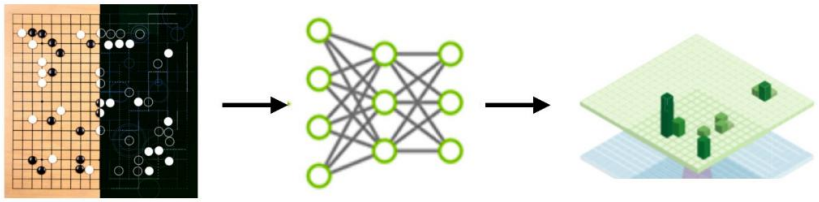


- 蒙特卡洛树形搜索（MCTS，Monte-Carlo Tree Search，2006）
- 给胜率高的点分配更多的计算力
- 任意时间算法，计算越多越精确
- 到叶子结点就roll out（黑白轮流快速下完数子出胜负结果）。
- 终局精确数子+多次随机模拟 = 局势评估（胜率）
- 人不可能用这种方法下棋，计算机特有的优势。但快速下完也易出bug。
- 全局思维、多次模拟统计结果，大局观已经强于人类不可靠的直觉。

深度学习的训练方法：卷积神经网络







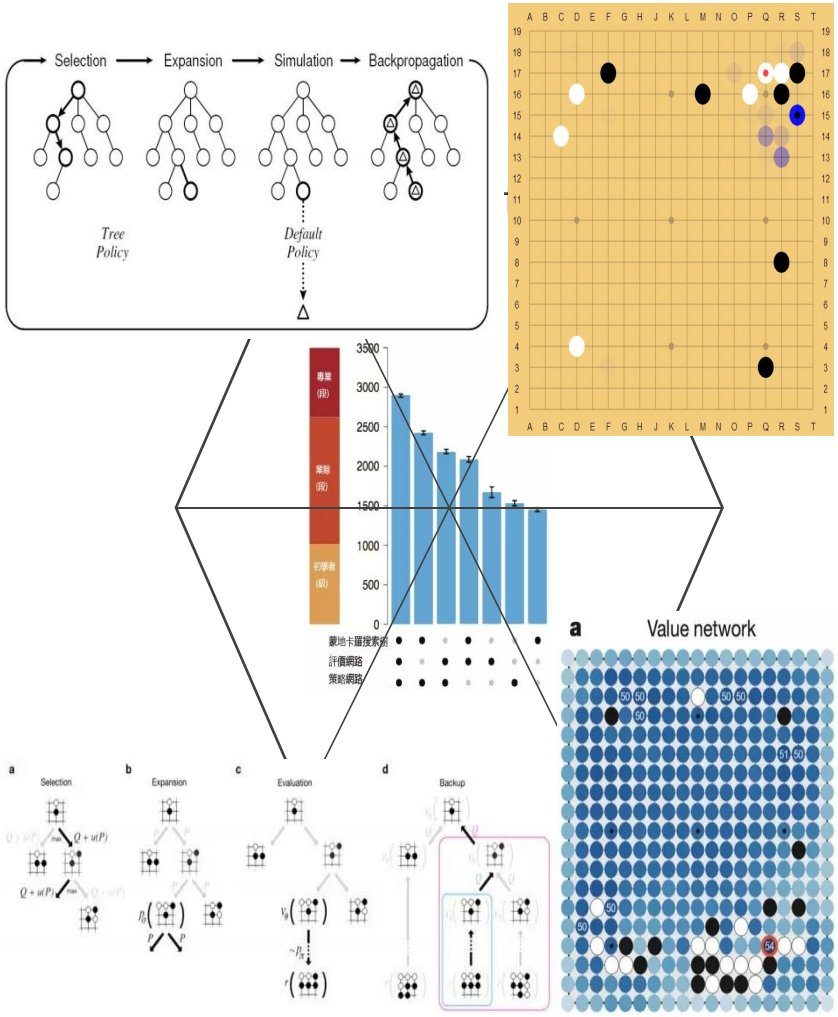
棋盘信息　　神经网络　　落子选择

- 局部感知域
- 权重共享
- 特征训练
- 卷积层+池化层

AlphaGo的实现原理

MCTS（蒙特卡洛树搜索）

给胜率高的点分配更多的计算力
任意时间算法，计算越多越精确
1、选取 2、展开 3、评估 4、倒传



Policy Network（策略网络）

利用神经网络预测下一步最可能的概率
有监督学习：人类棋谱
13个卷积层，每层192个卷积核，每个卷积核3*3，参数个数800万+
GPU 3ms/步
预测准确率 57%

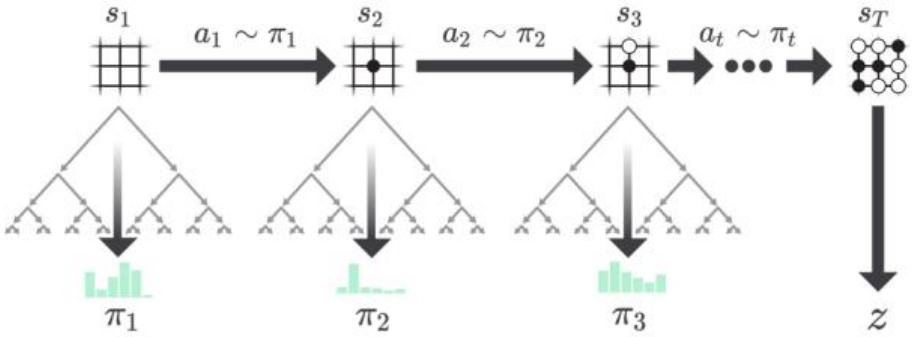
Rollout（随机模拟走子）

通过随机模拟走子胜率来判定形势
速度很快（1ms/盘）
随机性与合理性的平衡

Value Network（价值网络）

在每个分支节点直接判断形势
与Rollout随机模拟相结合，互为补充

强化学习



AlphaGo开发过程中的最强力武器：强化学习

- 机器海量自我对弈，根据结果修改自己的神经网络系数：机器独特的学习优势。
- 强化学习是学习过程，在下棋的时候没有用到。背后海量的资源用于学习。
- Deepmind的特点与思维方法是依靠机器自己改进，而非人工写代码解决问题。
- 强化学习如何进行很自由，新领域能作出非常多改进，效率提升潜力很大。

CFR虚拟遗憾最小化

- 黑格尔：人类从历史中学到的唯一的教训，就是没有从历史中吸取到任何教训
- 定义了一个Regrets 值，其含义是，在当前状态下，我选择行为A，而不是行为B，后悔的值是多少。
- 如果在之前的游戏情况中，我们没有选取某一个行为的后悔值最大，那么在下一次我们就更偏向于选择该行为。
- 如果我们进行了n次游戏，然后把每次游戏没有选取某个行为的后悔值加起来，然后做一个归一化，就可以得到一个概率分布作为我们的策略。而在每一步的后悔值由对手的行为所决定。
- 对于多次决策问题，就需要使用Counterfactual Regrets。给出每一步的Counterfactual Value。


SHENZHEN UNIVERSITY
1983