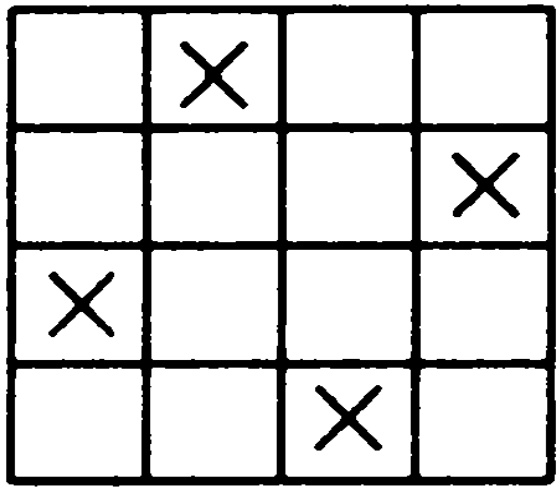
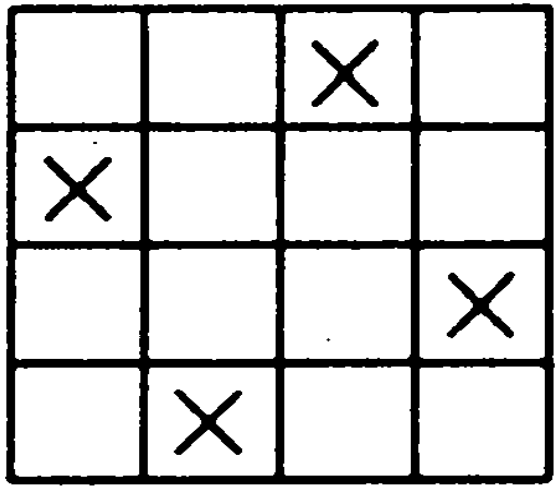
5.8 N后问题

问题描述:n*n棋盘上放置n个皇后使得每个皇后互不受攻击. 即任二皇后不能位于同行同列和同一斜线上.

如四后问题的两个解

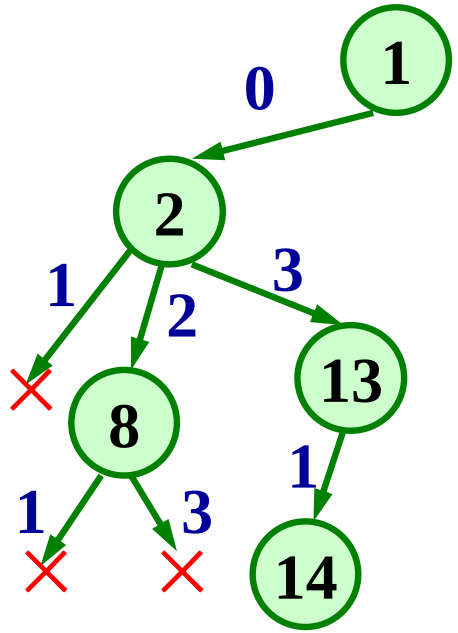


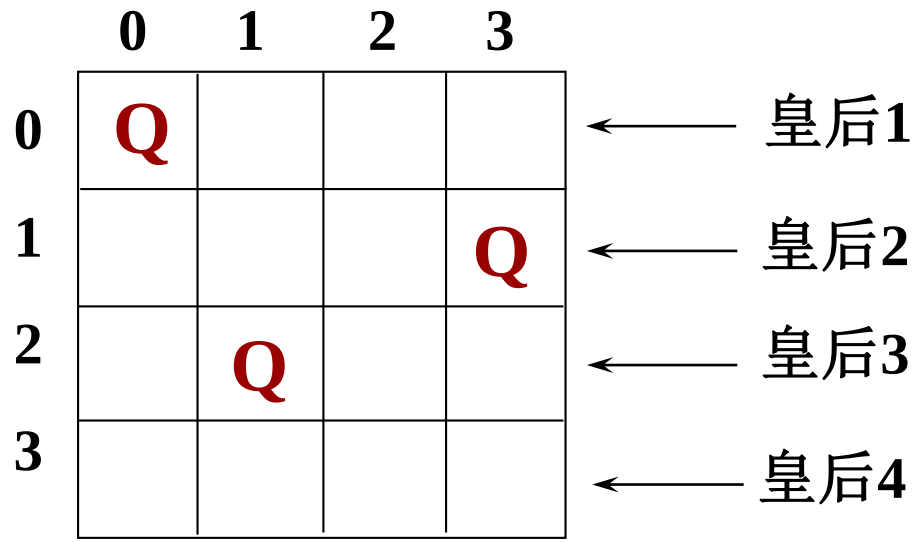


5.8 N后问题

4皇后问题

四皇后问题的解空间树是一个完全4叉树，树的根结点表示搜索的初始状态，从根结点到第1层结点对应皇后1在棋盘中第0行的可能摆放位置，从第1层结点到第2层结点对应皇后2在棋盘中第1行的可能摆放位置，依此类推。

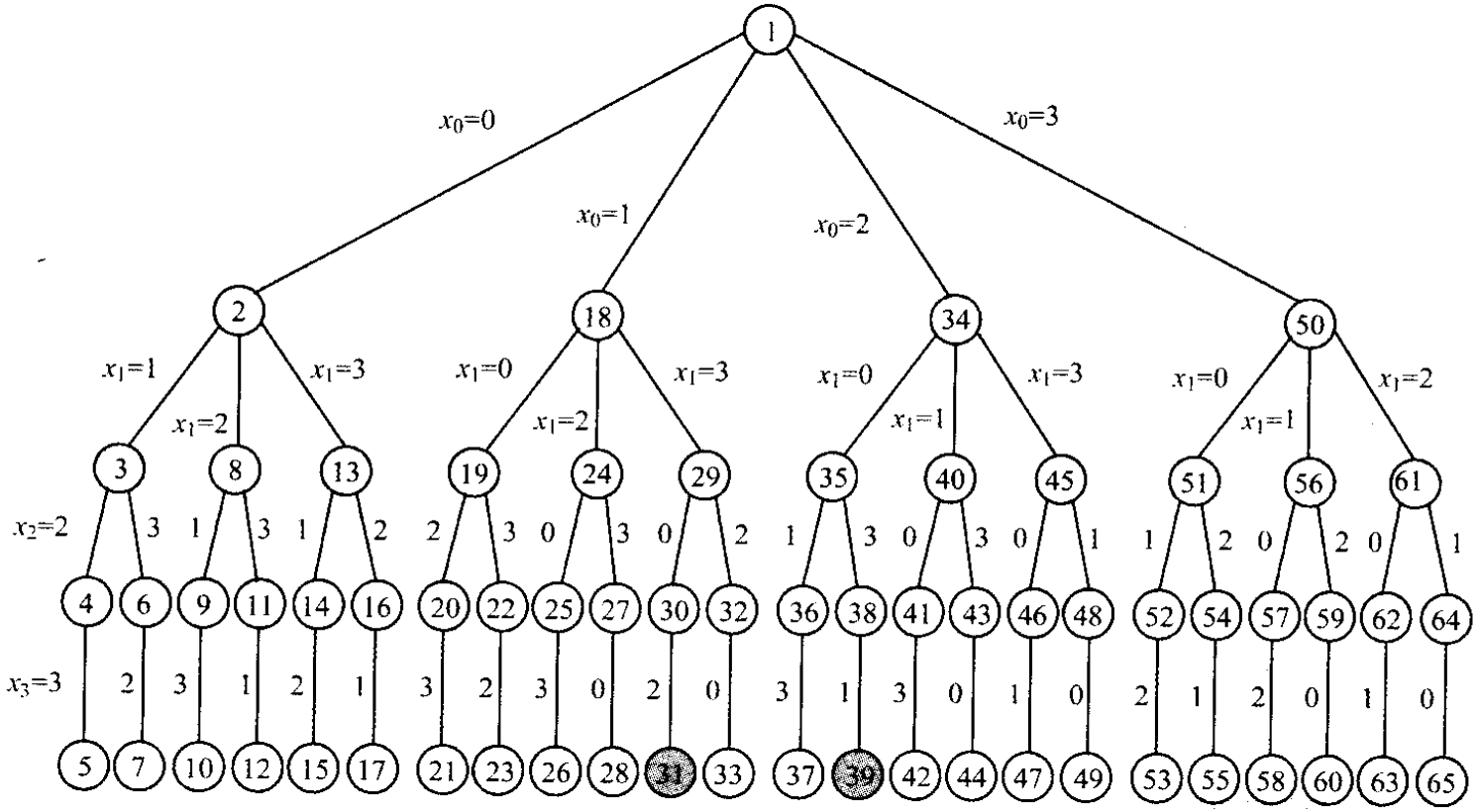




解空间树是子集树还是排列树？

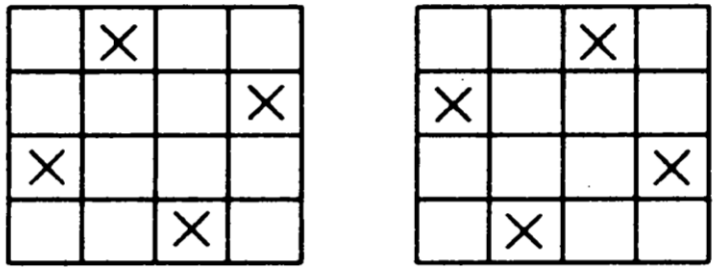
5.8 N后问题

4皇后问题的解空间树



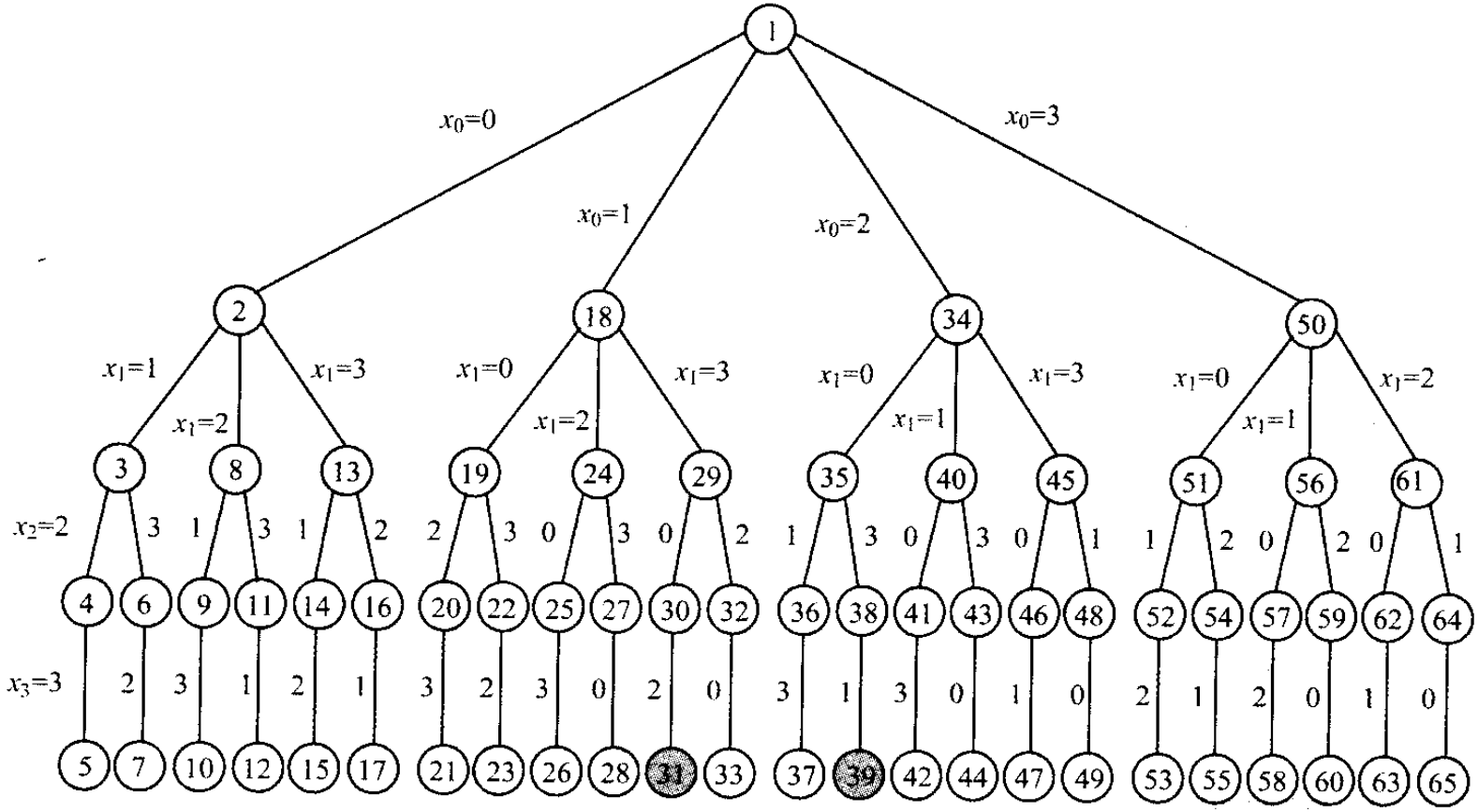
图是4-皇后问题的解空间树。

解状态中包含答案状态31和39。



5.8 N后问题

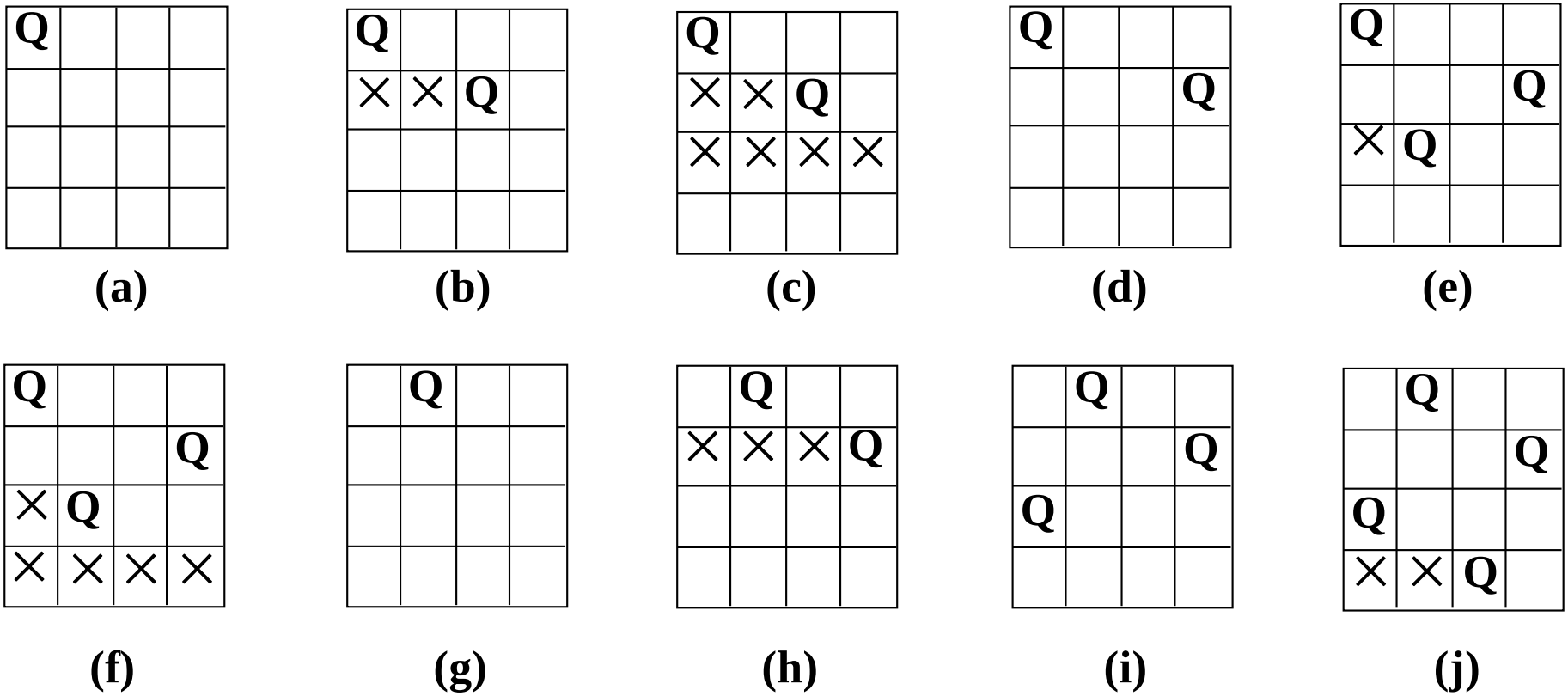
4皇后问题的解空间树



一般称这种用于确定n个元素的**排列满足某些性质**的解空间树为排列树（permutationtree）。排列树有n!个叶子结点，遍历排列树的时间为O(n!)。

5.8 N后问题

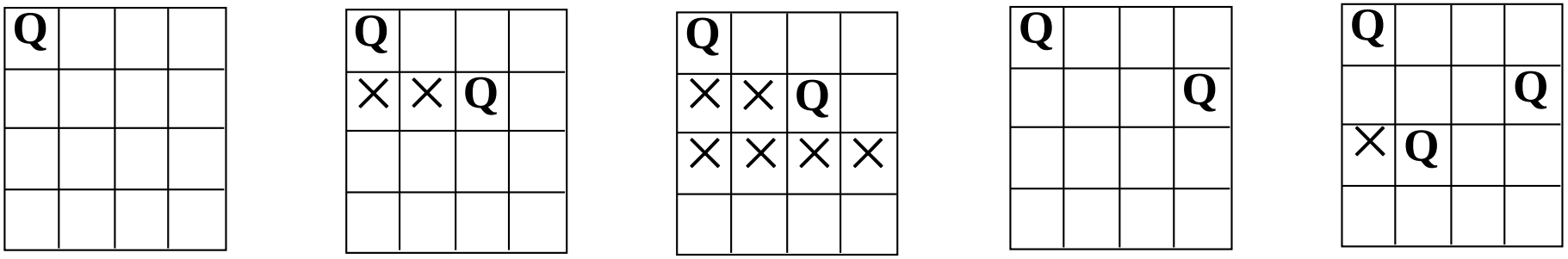
回溯法求解4皇后问题的搜索过程（一个可行解）

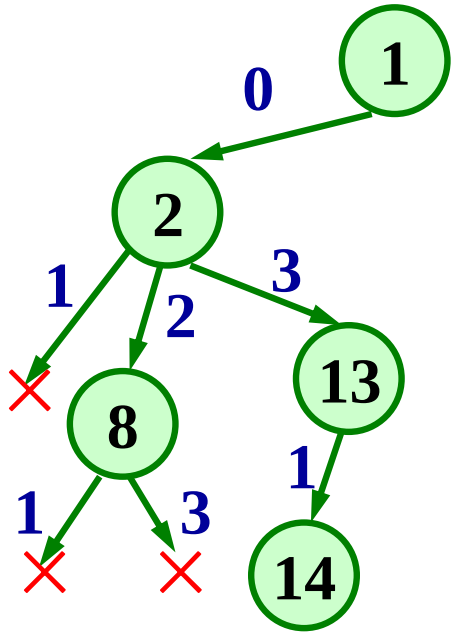


哪几个子图做了回溯操作？

5.8 N后问题

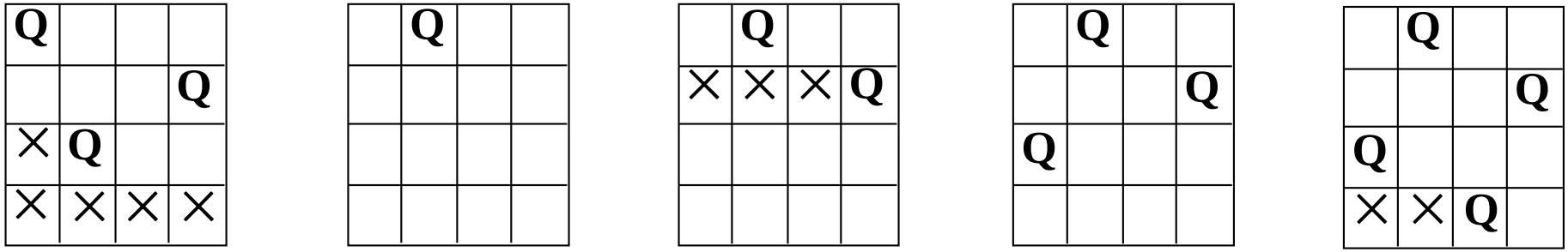
4皇后问题的解空间树搜索详细过程

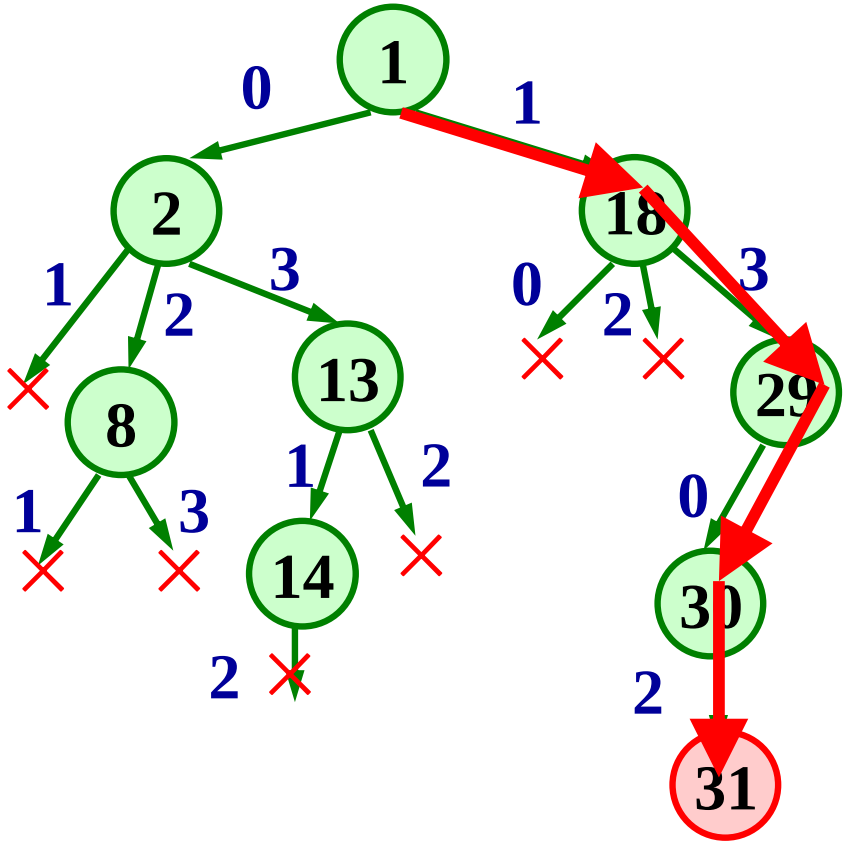




5.8 N后问题

4皇后问题的解空间树搜索详细过程





一个解（布局）为：

(1, 3, 0, 2)

一个布局！

5.8 N后问题

4皇后问题的解空间搜索状态

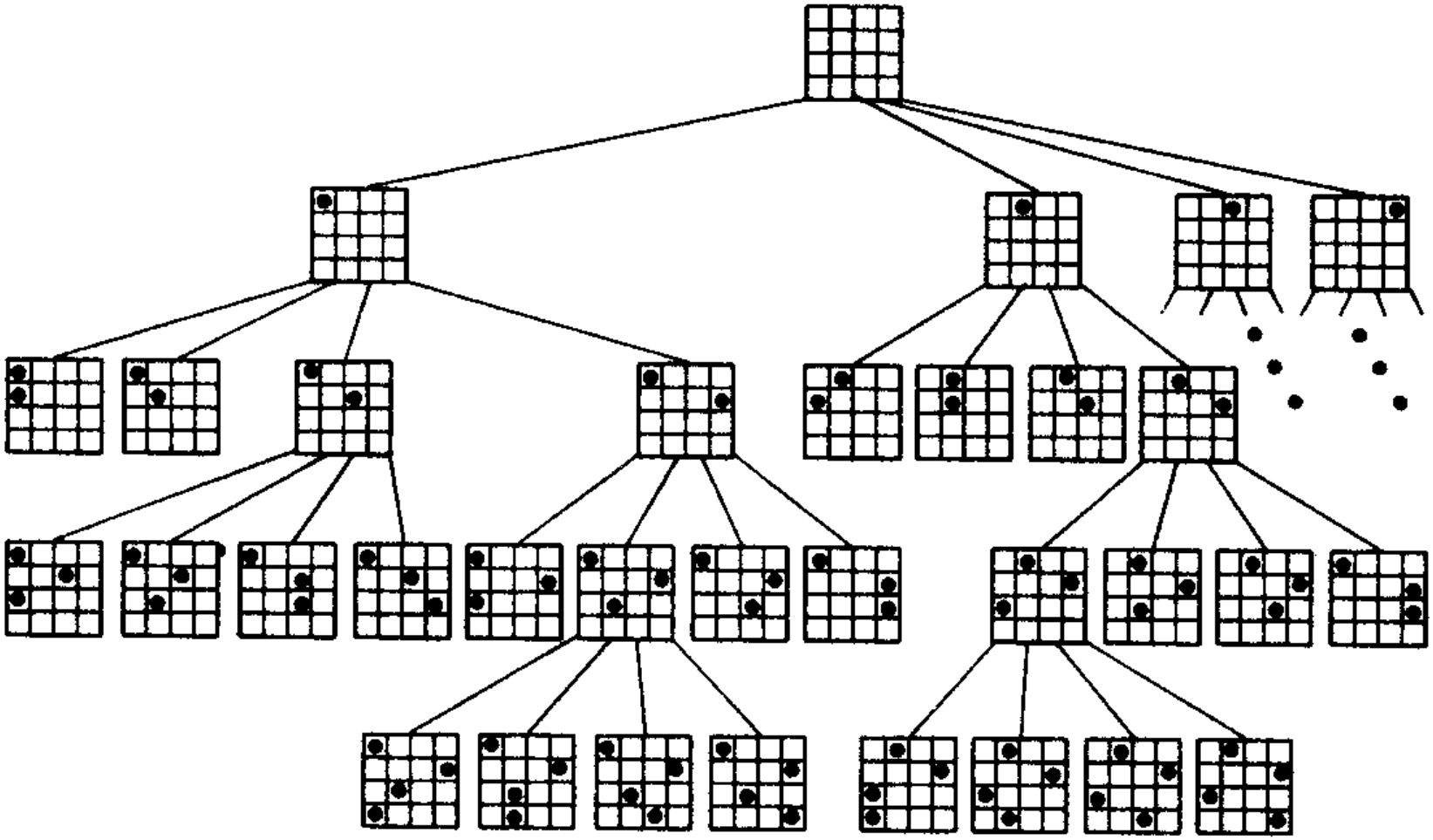


图 6.29　四皇后问题的棋盘状态树

5.8 N后问题　　算法分析-8后为例

设八个皇后为x_i，分别在第i行(i=1，2，3，4……，8)；

✓ 问题的解状态：可以用$(1, x_1)$，$(2, x_2)$，……，$(8, x_8)$表示8个皇后的位置；

由于行号固定，可简单记为：$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$；

✓ 问题的解空间：$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$，$1 \leqslant x_i \leqslant 8$(i=1，2，3，4……，8)，共$8^8$个状态；

✓ 约束条件：八个$(1, x_1)$, $(2, x_2)$，$(3, x_3)$, $(4, x_4)$，$(5, x_5)$, $(6, x_6)$，$(7, x_7)$，$(8, x_8)$不在同一行、同一列和同一对角线上

- **原问题即：在解空间中寻找符合约束条件的解状态。**

- **按什么顺序去搜？**
  - **目标是没有漏网之鱼，尽量速度快。**

- 穷举法
  - 通过8重循环模拟搜索空间中的8^8个状态；
  - 按枚举思想，以DFS的方式，从第1个皇后在第1列开始搜索，枚举出所有的“解状态”：
    - 从中找出满足约束条件的“答案状态”。

- **约束条件？**
  - **不在同一列：$x_i \neq x_j$；**
  - **不在同一主对角线上：$x_i - i \neq x_j - j$；**
  - **不在同一负对角线上：$x_i + i \neq x_j + j$。**
  - **违规的情况可以整合改写为：**
    - **$abs(x_i - x_j) = abs(i - j)$ or $(x_i = x_j)$**

算法描述-8后问题穷举法

```
queen1( )
{
  int a[9];
  for (a[1]=1;a[1]<=8;a[1]++)
    for (a[2]=1;a[2]<=8;a[2]++)
      for (a[3]=1;a[3]<=8;a[3]++)
        for (a[4]=1;a[4]<=8;a[4]++)
          for (a[5]=1;a[5]<=8;a[5]++)
            for (a[6]=1;a[6]<=8;a[6]++)
              for(a[7]=1;a[7]<=8;a[7]++)
                for(a[8]=1;a[8]<=8;a[8]++){
                  if (check(a,8)=0)   continue;
                  else
                    for(i=1;i<=8;i++)
                      print(a[i]);
                }
}
```

用a[1]~a[8]存储x_1~x_8

```
check(a[],n)
{int i,j;
 for(i=2;i<=n;i++)
    for(j=1;j<=i-1;j++)
      if (a[i]=a[j]) or
       (abs(a[i]-a[j])=abs(i-j))
          return(0);
 return(1);
}
```

双重循环，任意两个皇后之间都必须检查。

算法描述-8后问题递归回溯法

- n：皇后个数
- x：当前解
- sum：当前已找到的可行方案数

```
void backtrack (int t)  {
    if (t>n) sum++;
      else
        for (int i=1;i<=n;i++) {
          x[t]=i;
          if (place(t)) backtrack(t+1);
         }
 }
```

调用：backtrack(1);

本问题：排列树
约束条件：由place(t)控制

约束函数：
place(t)

算法描述-8后问题递归回溯法

```
bool Queen::Place(int k)
{//判定两个皇后是否在同一斜线或在同一列上
  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
  return true;
}
```

用函数Place(k) 测试待确定的第k皇后是否和前面已确定的k -1个皇后是否在同一条斜角线或同一列上。

第k个皇后只要与任一皇后在同一条斜角线，就返回false，当第k个皇后能放置于X(k)的当前值处时，这个返回值为true。

算法描述---迭代回溯

```
void    N-queens(n){
   x[1]= 0; sum=0;   k = 1;      //k为层次
   while  (k > 0)   {
        x[k] = x[k] + 1;
        while(  (x[k] <= n) &&!place(k))  // //第k个皇后位置不合适
                x[k]= x[k] + 1;     //尝试下一个位置
          if  (x[k]  <=  n)
              if  (k == n) sum++;//找到一组解
              else {     //继续搜索第k+1个皇后位置，从头开始；
                    k= k + 1;    x[k]= 0
                    }
          else k --;  //回溯 }
}
```

5.10 回溯法小结

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

(1)产生x[k]的时间；

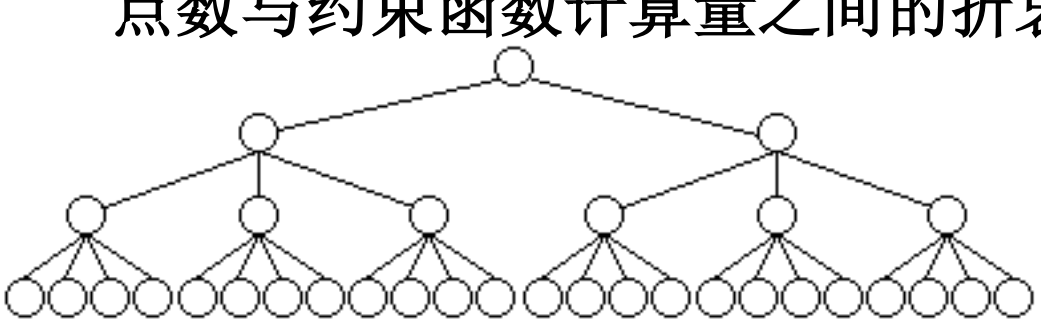
(2)满足显约束的x[k]值的个数；

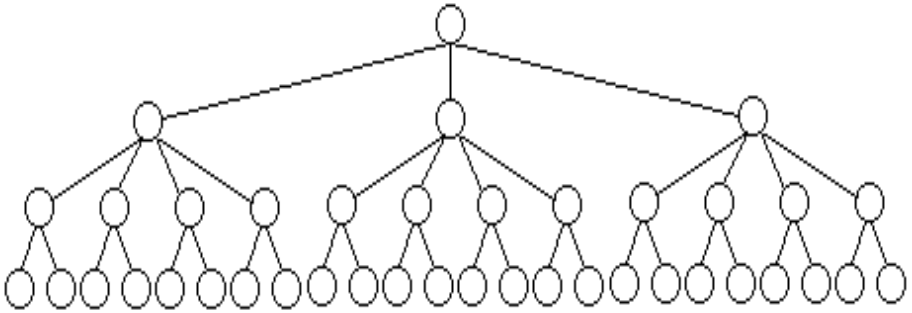
(3)计算约束函数constraint的时间；

(4)计算上界函数bound的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷

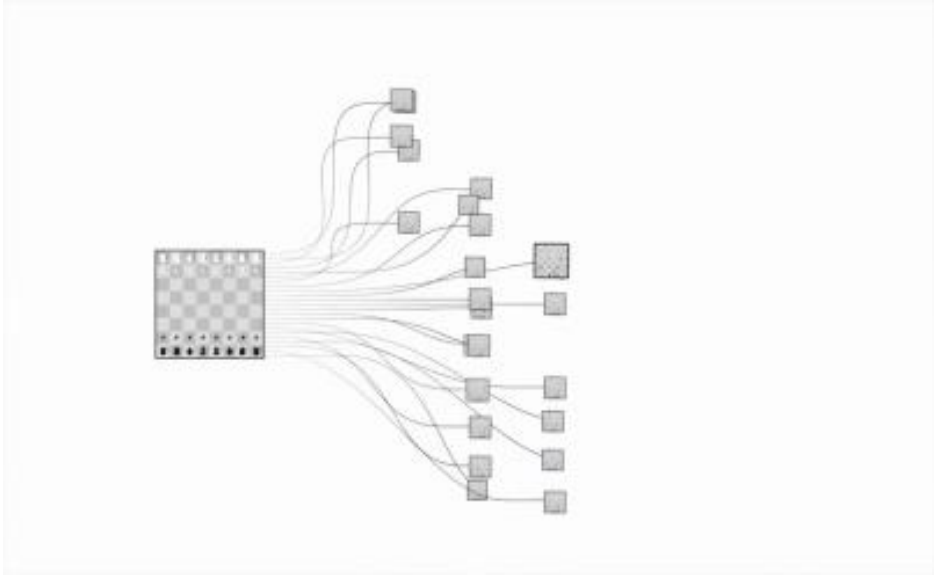


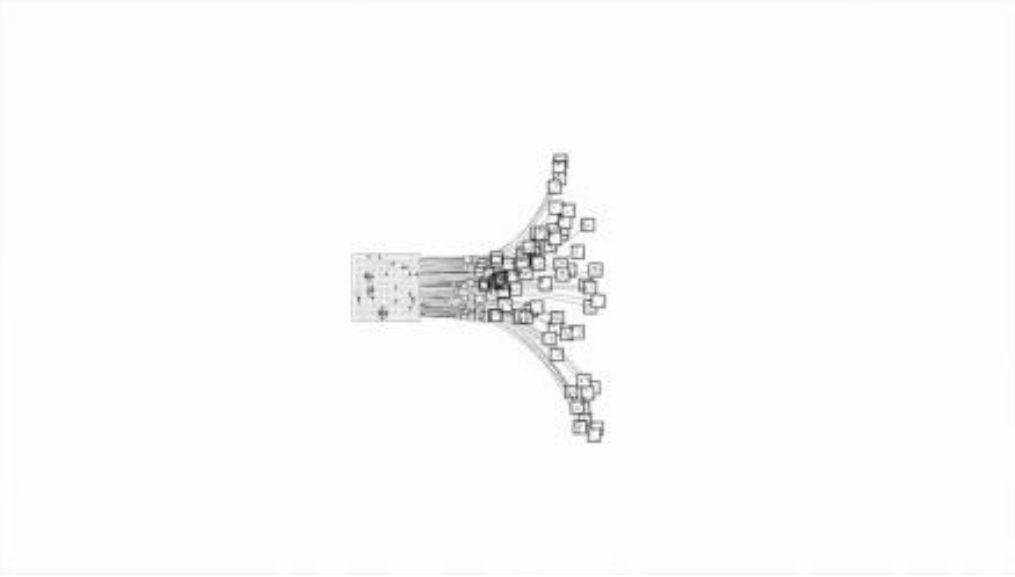


策略：能进则进，不能进则换，不能换则退。

*5.9 拓展学习 AlphaGo中的回溯法

机器下棋的算法本质都是搜索树，围棋难在它的树宽可以达到好几百（国际象棋只有几十）。在有限时间内要遍历这么宽的树，就只能牺牲深度（俗称“往后看几步”），但围棋又是依赖远见的游戏，甚至不仅是看“几步”的问题。所以，要想保证搜索深度，就只能放弃遍历，改为随机采样——这就是为什么在没有MCTS（蒙特卡罗搜树）类的方法之前，机器围棋的水平几乎是笑话，而采用了MCTS方法后博弈水平能超过人类。



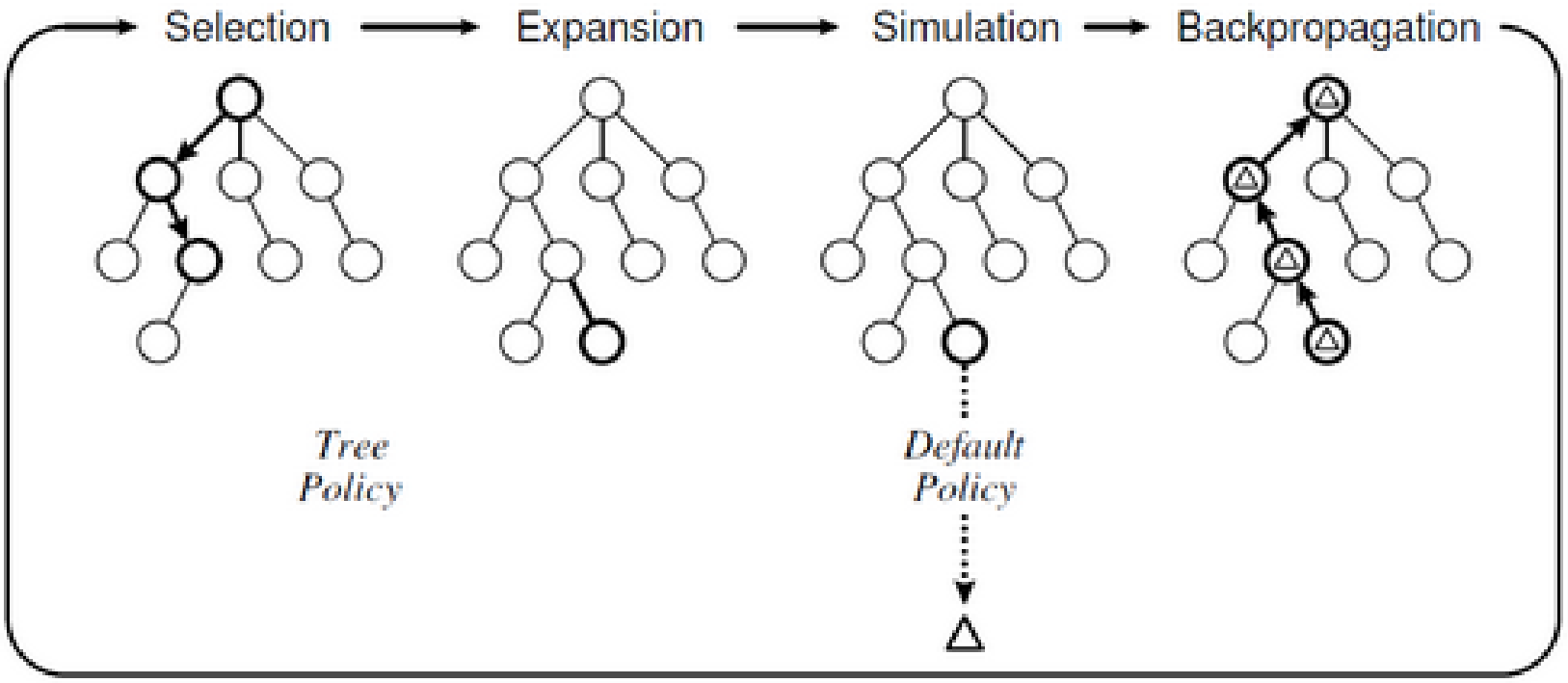


传统AI技术还是暴力搜索，将游戏中所有的可能性表示成一棵树，树的第N层就代表着游戏中的第N步。树的node数是随着树的深度成指数增长的，不考虑剪枝，每个node都是需进行估值。

*5.9 拓展学习 AlphaGo中的回溯法

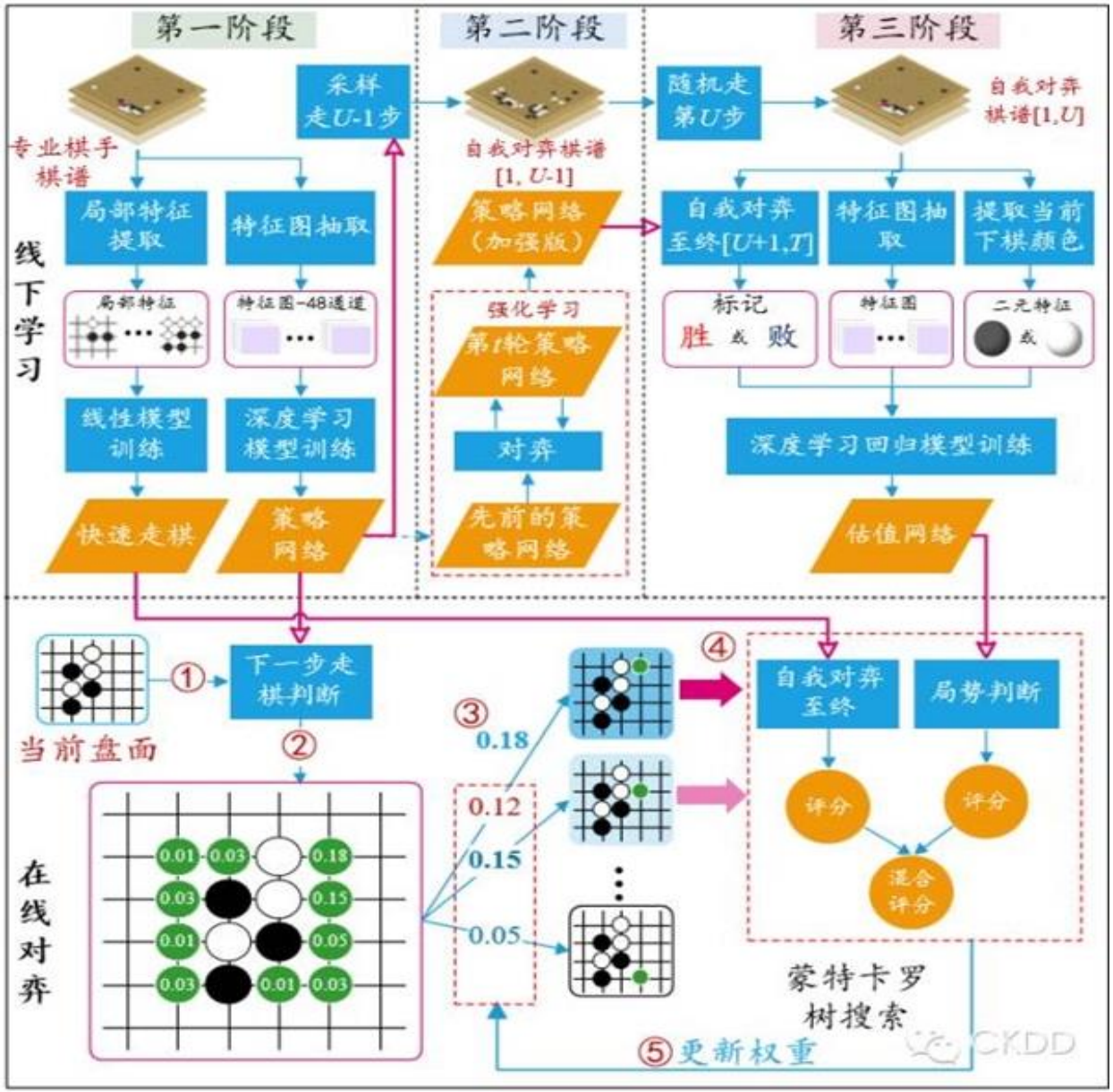
蒙特卡洛树搜索(Monte Calro Tree Search, MCTS)

对于一个具体的问题来说，回溯法的有效性往往就体现在当问题实例的规模n较大时，它能够用很少的时间就求出问题的解。而对于一个问题的具体实例我们又很难预测回溯法的算法行为，特别是我们很难估计出回溯法在解这一具体事例时所产生的结点数，这是分析回溯法效率的主要困难。蒙特卡洛树搜索算法能有效克服这一困难：



*5.9 拓展学习 AlphaGo中的回溯法

阿尔法狗计算框架及原理



补充学习资料

http://tech.sina.com.cn/d/2014-07-28/10299520957.shtml

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

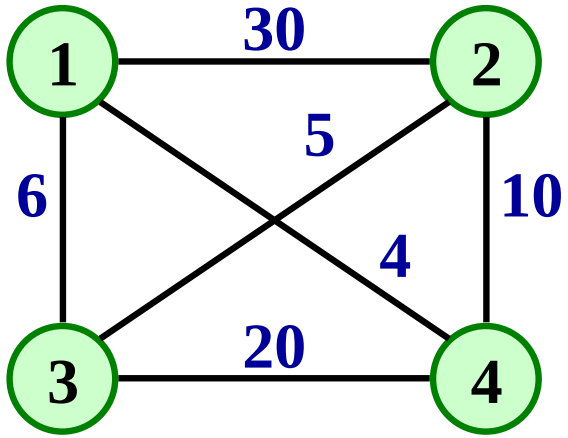
http://wkee.net/post/google-alphago-weakness.html

http://blog.csdn.net/natsu1211/article/details/50986810

http://www.longgaming.com/archives/214

课堂练习4：回溯法求解旅行售货员问题

某售货员要到若干城市去推销商品，已知各城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程（或总旅费）最小。



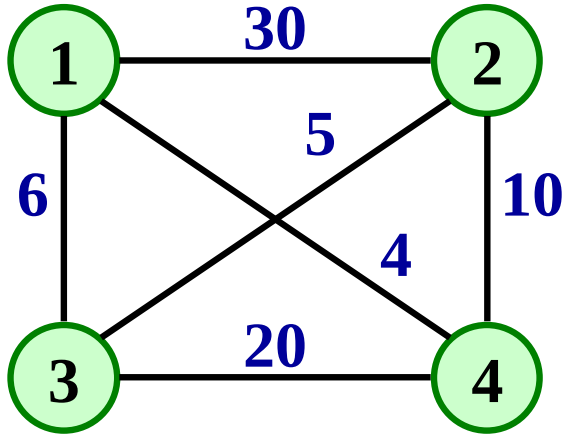
4顶点带权图

课堂练习4：回溯法求解旅行售货员问题

- 设G = (V, E)是一个带权图。图中各边的费用（权）为正数。
- 图中的一条周游路线是包括V中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。
- **旅行售货员问题：**

✓ ***在下图G中找出费用最小的周游路线。***

✓ ***给出解空间树，写出搜索过程。***

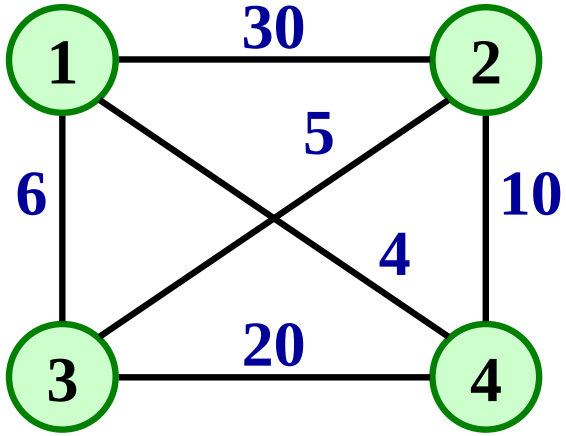


4顶点带权图

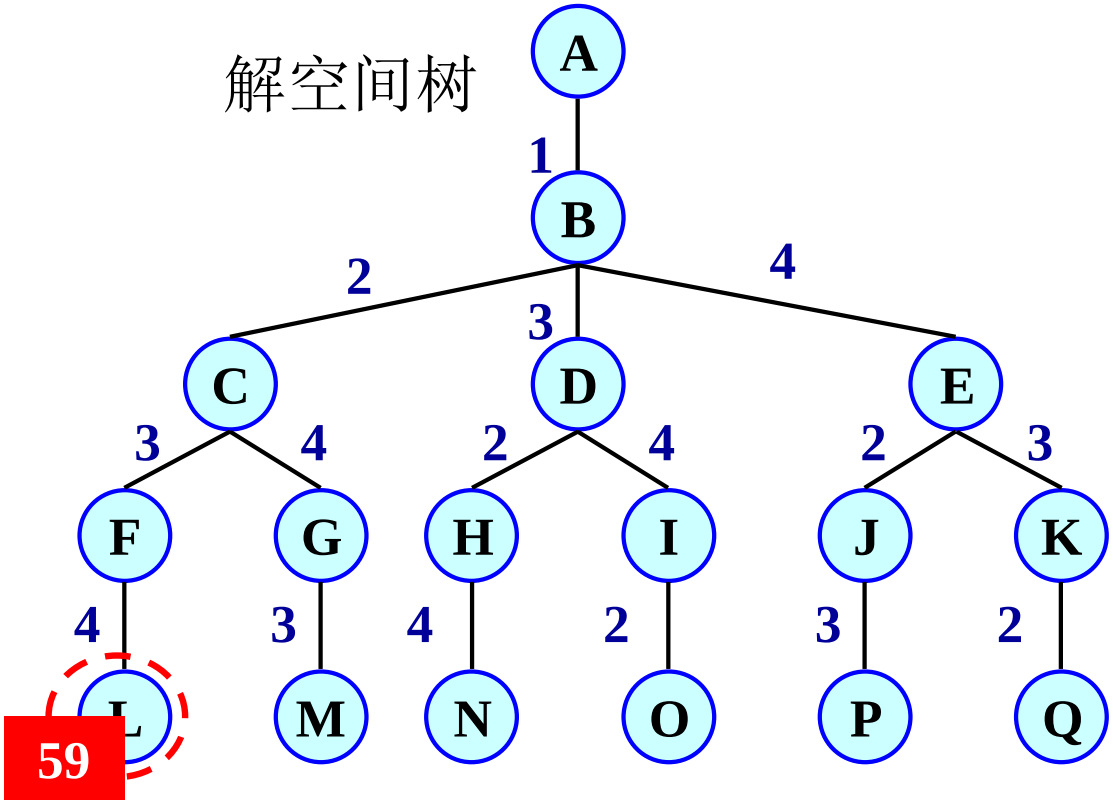
- 针对所给问题，定义问题的解空间；
- 确定易于搜索的解空间结构；
- 以深度优先搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

课堂练习4：回溯法求解旅行售货员问题

求解过程



4顶点带权图

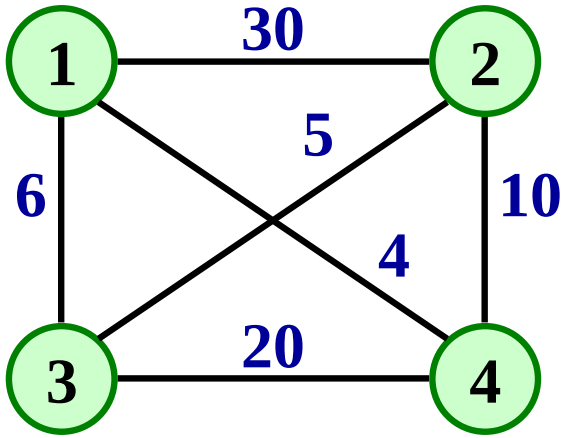


旅行售货员问题的解空间树

- 用回溯法找最小费用周游路线时，从解空间树的根结点A出发，搜索至B，C，F，L。在叶子结点L处记录找到的周游路线1,2,3,4,1，该周游路线的费用为59。

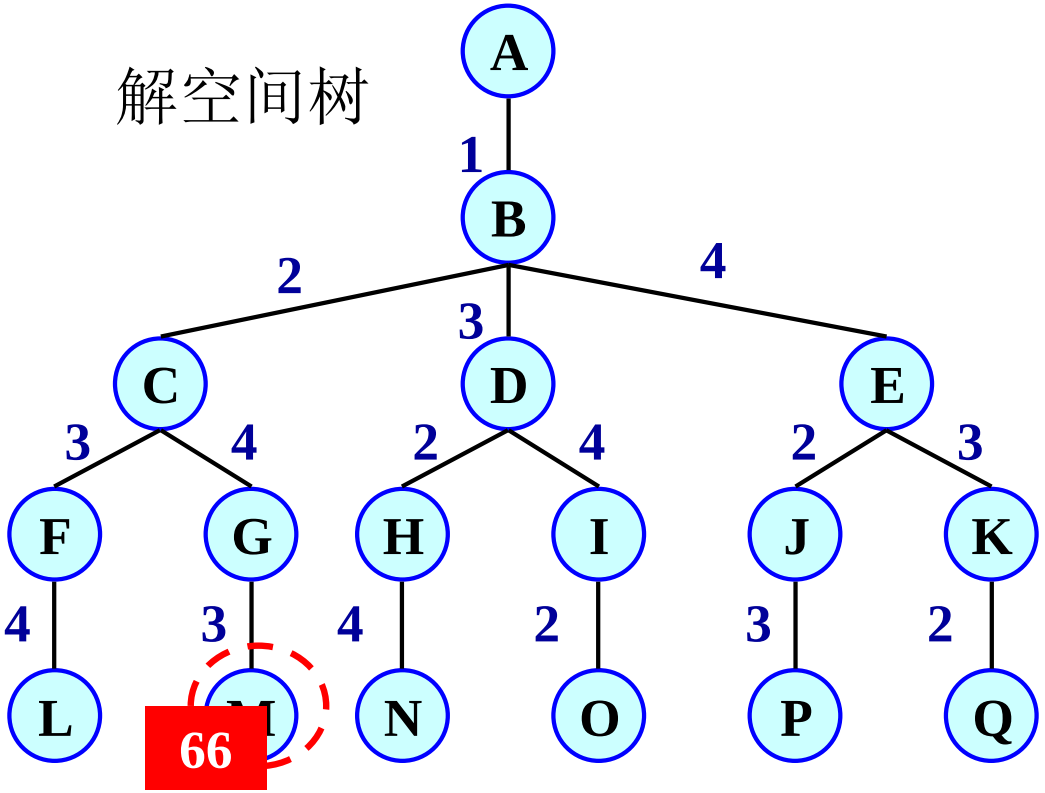
课堂练习4：回溯法求解旅行售货员问题

求解过程



4顶点带权图

解空间树

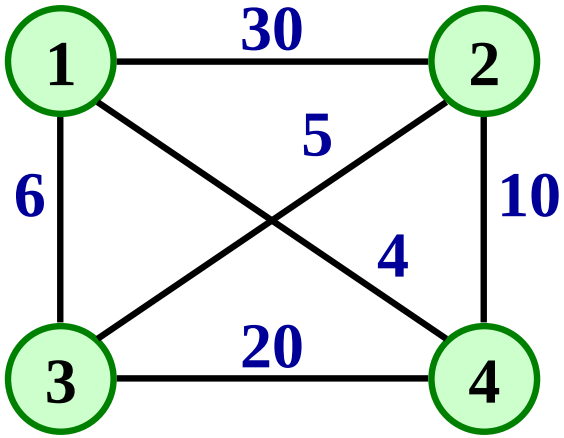


旅行售货员问题的解空间树

- 从叶子结点L返回最近活结点F处。由于F已没有可扩展结点，算法又返回到结点C处。结点C成为新扩展结点，由新扩展结点，算法再移动到结点G又移至结点M，得到新的周游路线1,2,4,3,1，其费用为66。费用比前一个大，舍弃！

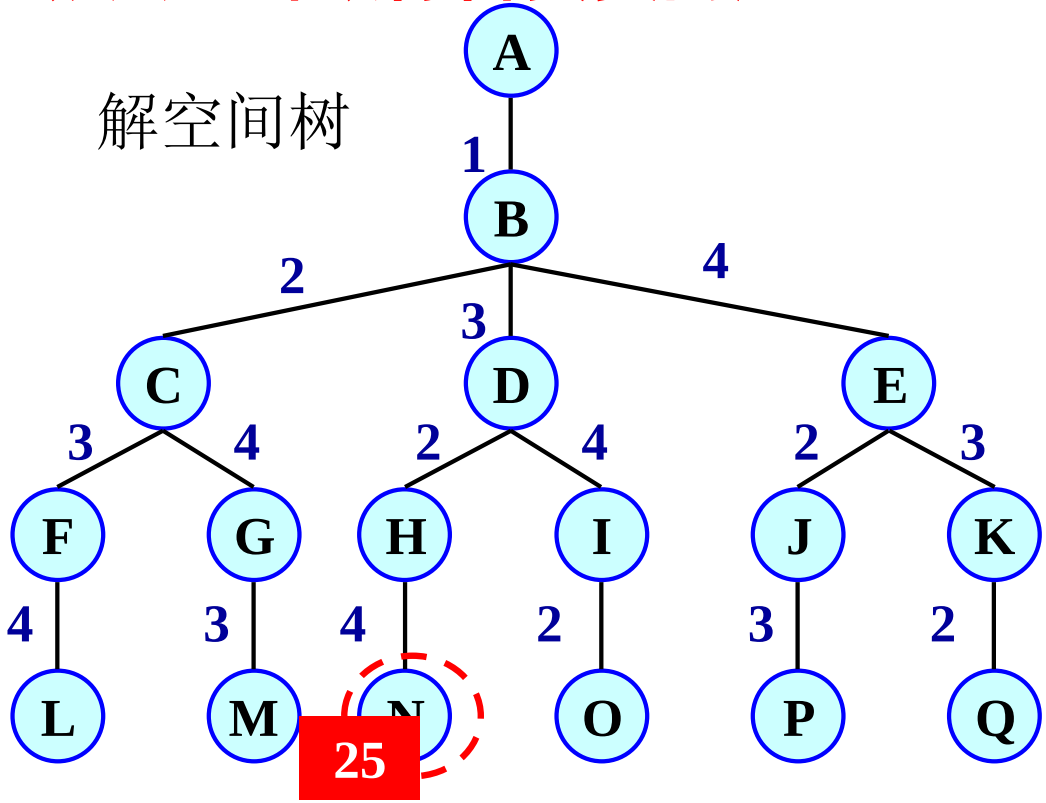
课堂练习4：回溯法求解旅行售货员问题

求解过程



4顶点带权图

解空间树

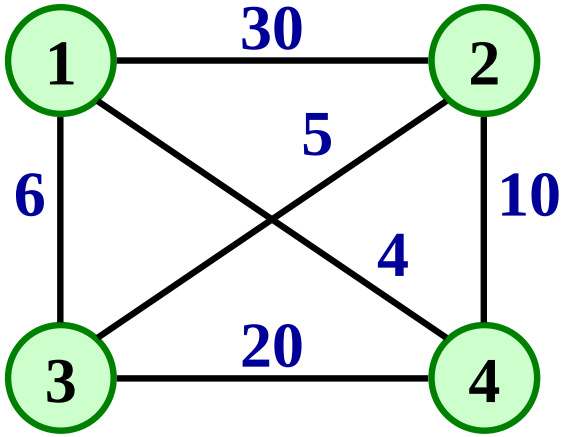


旅行售货员问题的解空间树

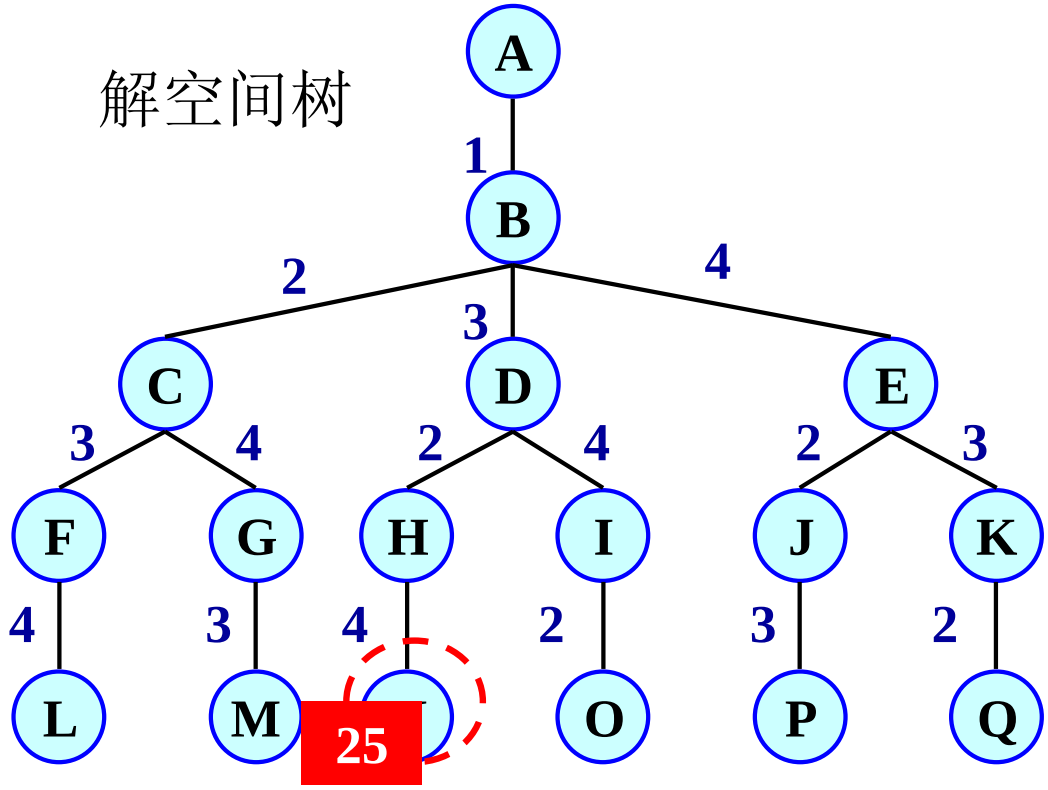
- 算法又依次返回到结点G，C，B。从结点B，算法继续搜索到结点D，H，N。在叶子结点N处，相应的周游路线1,3,2,4,1，其费用为25。它是当前找到的最好的一条周游路线。

课堂练习4：回溯法求解旅行售货员问题

求解过程



4顶点带权图



旅行售货员问题的解空间树

- 从结点N算法返回至结点H，D，然后再从结点D开始继续向纵深搜索至结点O。依此方式继续搜索整个解空间树，最终得到最小费用周游路线1,3,2,4,1。