方法概述: 搜索算法介绍

- 搜索的重要性和提高效率的途径
  - 搜索是人工智能的基本手段. 推理可以看成一种搜索，联想、回忆、灵感也是一种搜索过程;
  - 搜索效率: ***Time Complexity, Space Complexity, Solution Quality; Completeness***
  - 提高搜索的基本途径: 一是积累并恰当使用有助于减少搜索的信息和知识(启发式方法); 二是采用并行处理技术;

方法概述: 搜索算法介绍（续）

- 搜索算法

(1)穷举搜索(*Exhaustive Search*)

(2)盲目搜索(*Blind Search*)

- 深度优先(*DFS*)或回溯搜索(*Backtracking*);
- 广度优先搜索(*BFS*);
- 分枝限界法(*Branch & Bound*);
- 博弈树搜索(*α-β Search*)

(3)启发式搜索(*Heuristic Search*)

方法概述: 搜索算法介绍（续）

- 搜索空间的三种表示
  - 表序表示: 搜索对象用线性表数据结构表示;
  - 显式图表示: 搜索对象在搜索前就用图(树)的数据结构表示;
  - 隐式图表示: 除了初始结点, 其他结点在搜索过程中动态生成. 缘于搜索空间大, 难以全部存储.

方法概述: 搜索算法介绍（续）

- 提高搜索效率的思考：随机搜索
  - 上世纪70年代中期开始, 国外一些学者致力于研究随机搜索求解困难的组合问题, 将随机过程引入搜索;
  - 选择规则是随机地从可选结点中取一个, 从而可以从统计角度分析搜索的平均性能;
  - 随机搜索的一个成功例子: 判定一个很大的数是不是素数, 获得了第一个多项式时间的算法;

组合爆炸

一个问题有n个变量，k个选择，
则遍历空间的大小为k^n

一张充分大的普通纸对折40次：
高度超过5千座珠穆朗玛峰

一张充分大的普通纸对折50次：
月地距离的128倍

并行处理也不可能克服和绕过组合爆炸；

但并行处理可能是当前提高求解规模的最好和现实的途径。

一、宽度优先搜索法

Breadth First Search (BFS)

宽度优搜索法：逐层地对结点进行扩展。

•例：重排九宫问题：在3X3的方格棋盘上放置分别标有数字1，2，3，···，8的八张牌，初始状态为S_0，目标状态为S_t

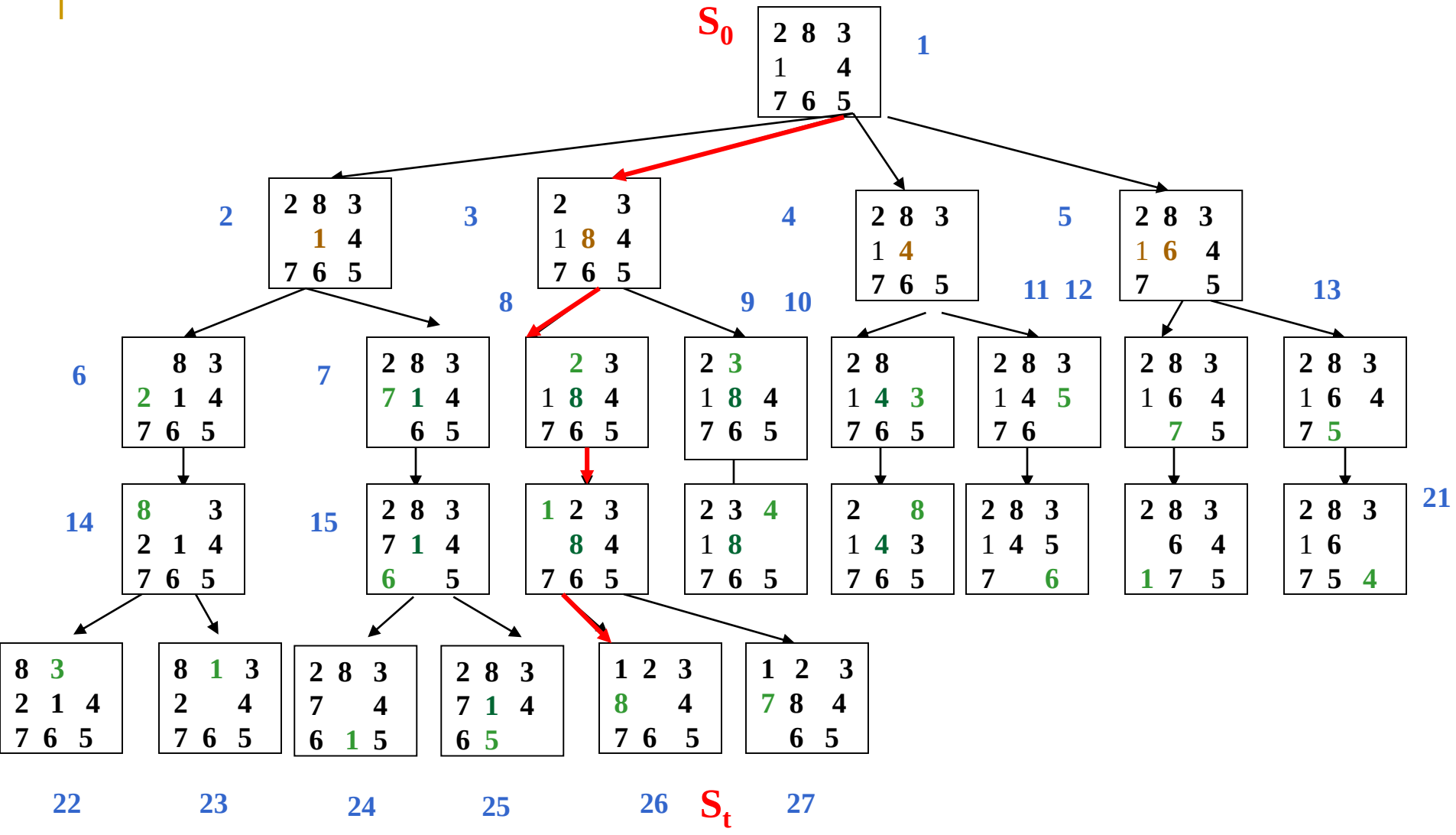
S_0

2	8	3
1		4
7	6	5

→

S_t

1	2	3
8		4
7	6	5



Route: $S_0 \rightarrow 3 \rightarrow 8 \rightarrow 16 \rightarrow 26(S_t)$

- 优点：

只要问题有解，用宽度优先搜索法一定可以得到解，而且得到的是路径最短的解。

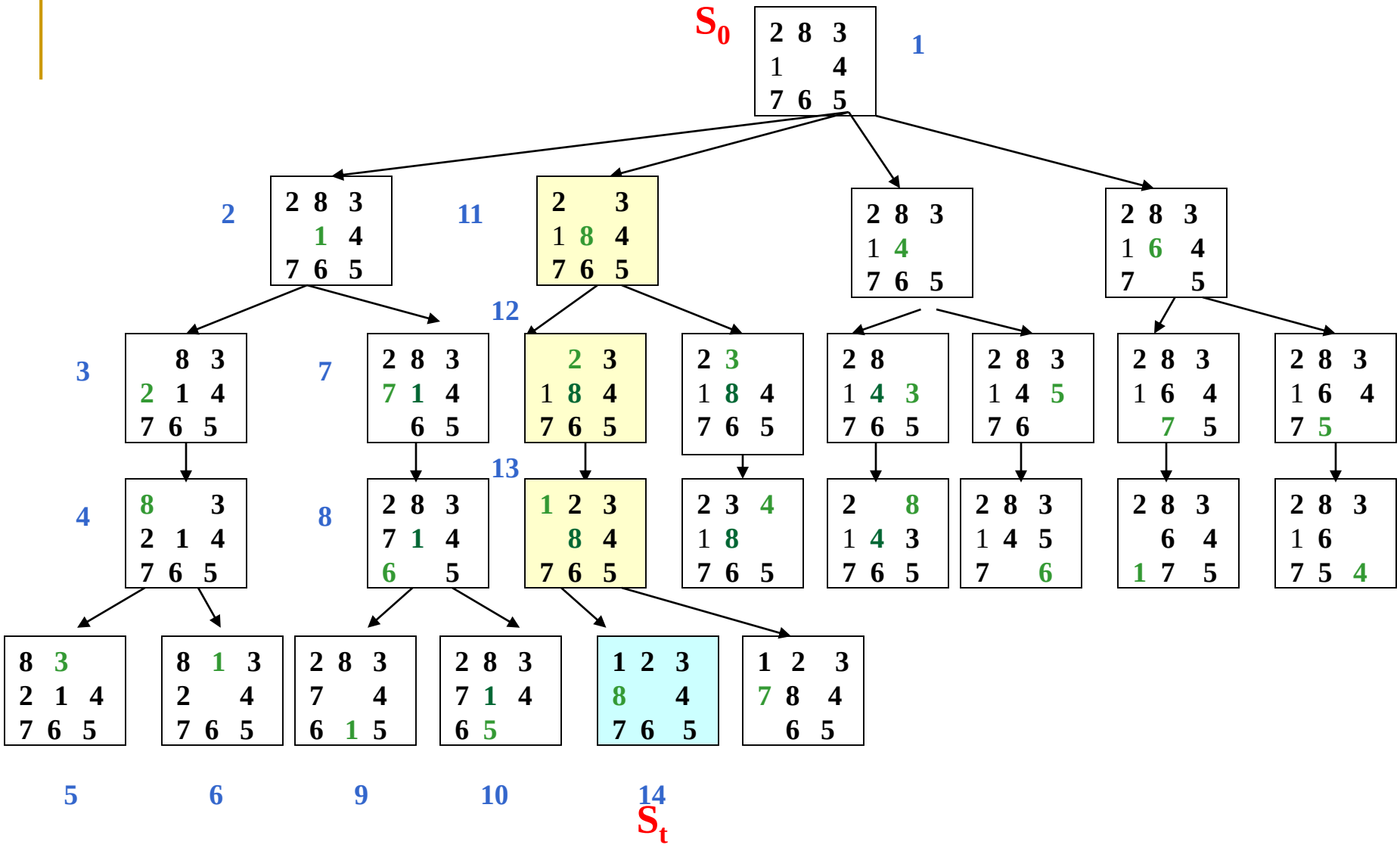
缺点：

盲目性较大。当目标结点距离初始结点较远时将会产生许多无用结点，搜索效率低。

二、深度优先搜索法

Depth First Search (DFS)

深度优搜索法：纵深地对结点进行扩展。



S_o:l→11 → 12 →13 → 14:S_t

搜索一旦进入某个分支，就将沿着该分支一直向下搜索。如果目标结点恰好在此分支上，则可较快地得到解。

但如果目标结点不在此分支上，而该分支又是一个无穷分支，则就不可能得到解。

所以深度优先搜索是不完备的。

（二）、有界深度优先搜索法

解决深度优先搜索不完备的问题，引入搜索深度的界限（设为d_m），当搜索深度达到了深度界限，而尚未出现目标结点，就换一个分支进行搜索。

三、启发式搜索法

估价函数：用于估价结点重要性的函数。

·一般形式：$f(x)=g(x)+h(x)$

其中：$g(x)$为从初始结点S_i到结点x已经付出的代价；

·$h(x)$是从结点x到目标结点S_t的最优路径的估计代价，它体现了问题的启发性信息。

$g(x)$：表示结点x的深度；

$h(x)$：表示结点x的格局与目标结点格局不相同的牌数。

博弈论

囚徒困境

智猪博弈

“囚徒困境”

你和同伙落入法网，被分开关押，无法串供。你们事先约定，大家都打死不招，然而地方检察官给出了下面几个选择：

1. 如果你坦白交待而同伙拒不认罪，那么你将被释放，而他将蹲三年监狱；

2. 如果他坦白交待而你拒不认罪，那就轮到你蹲三年大牢，而他获得自由；

3. 如果两人都坦白，每人各处两年有期徒刑；

4. 如果两人都不认罪，每人各处一年有期徒刑。


囚徒困境
对手的对策
合作
（保持沉默）
背叛
（坦白交代）
我的对策
合作
（保持沉默）
一年监禁
（高收益）
三年监禁
（傻瓜收益）
背叛
（坦白交代）
获得释放
（诱惑收益）
两年监禁
（低收益）

结果：都招了

“智猪博弈”

猪圈里有两头猪，一头大猪，一头小猪。猪圈的一边有个踏板，每踩一下踏板，在远离踏板的猪圈的另一边的投食口就会落下少量的食物。如果有一只猪去踩踏板，另一只猪就有机会抢先吃到另一边落下的食物。当小猪踩动踏板时，大猪会在小猪跑到食槽之前刚好吃光所有的食物；若是大猪踩动了踏板，则还有机会在小猪吃完落下的食物之前跑到食槽，争吃到另一半残羹。

小猪将选择舒舒服服地等在食槽边；而大猪则为一点残羹不知疲倦地奔忙于踏板和食槽之间。

博弈树搜索

- 诸如下棋、打牌、竞技、战争等一类竞争性智能活动称为博弈。其中最简单的一种称为“二人零和、全信息、非偶然”博弈。

- 所谓“二人零和、全信息、非偶然”博弈是指：
  - (1)对垒的A，B双方轮流采取行动，博弈的结果只有三种情况：A方胜，B方败；B方胜，A方败；双方战成平局。
  - (2)在对垒过程中，任何一方都了解当前的格局及过去的历史。
  - (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，选取对自己最为有利而对对方最为不利的对策，不存在“碰运气”的偶然因素。即双方都是很理智地决定自己的行动。

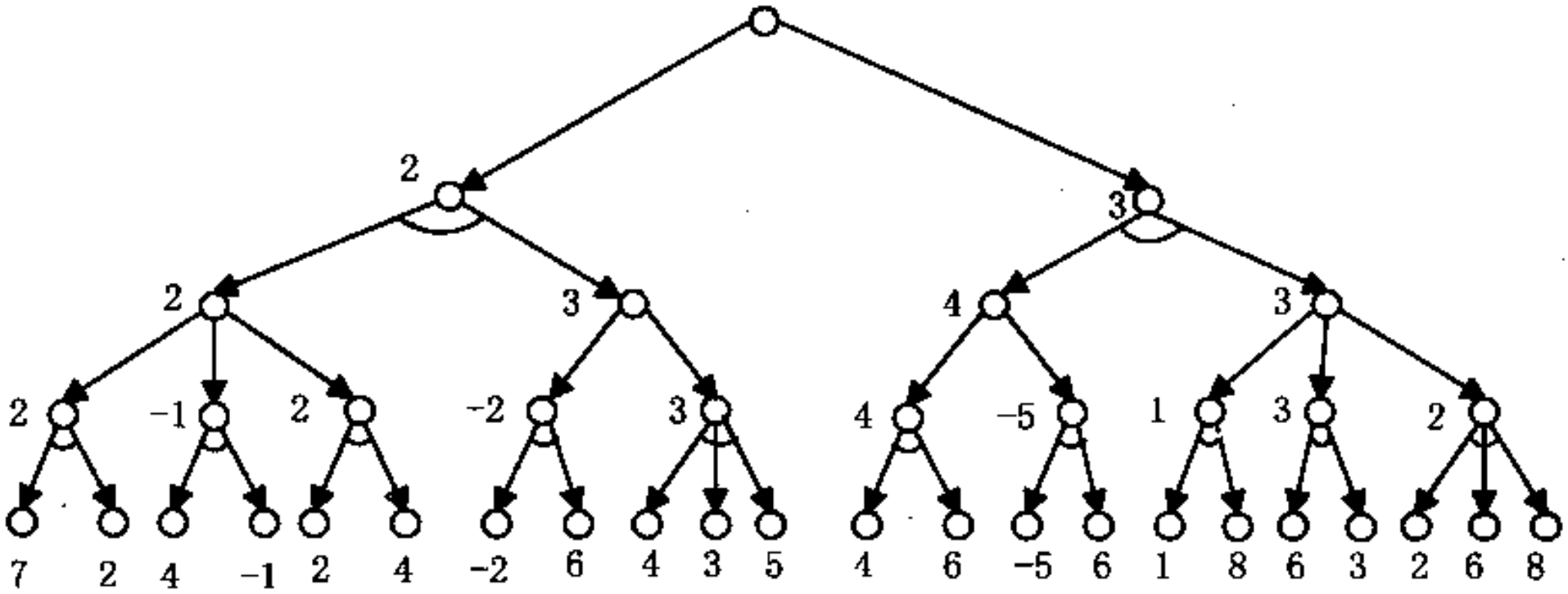
- 博弈树的概念
- 在博弈过程中，任何一方都希望自己取得胜利。因此，当某一方当前有多个行动方案可供选择时，他总是挑选对自己最为有利而对对方最为不利的那个行动方案。
- 这样，如果站在某一方（如A方，即在A要取胜的意义下)，把上述博弈过程用图表示出来，则得到的是一棵“与或树”。描述博弈过程的与或树称为博弈树 ，它有如下特点：

- 博弈树特点：
  - (1)博弈的初始格局是初始节点。
  - (2)在博弈树中，“或”节点和“与”节点是逐层交替出现的。自己一方扩展的节点之间是“或”关系，对方扩展的节点之间是“与”关系。双方轮流地扩展节点。
  - (3)所有自己一方获胜的终局都是本原问题，相应的节点是可解节点；所有使对方获胜的终局都是不可解节点。

- 极小极大分析法
  - 在二人博弈问题中，为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案，就需要对当前的情况以及将要发生的情况进行分析，从中选出最优的走步。最常使用的分析方法是极小极大分析法。
  -

- 其基本思想是：
  - (1)设博弈的双方中一方为A，另一方为B。然后为其中的一方(例如A)寻找一个最优行动方案。
  - (2)为了找到当前的最优行动方案，需要对各个可能的方案所产生的后果进行比较。
  - (3)为计算得分，需要根据问题的特性信息定义一个估价函数，用来估算当前博弈树端节点的得分。

- (4)当端节点的估值计算出来后，再推算出父节点的得分，推算的方法是：对“或”节点，选其子节点中一个最大的得分作为父节点的得分，这是为了使自己在可供选择的方案中选一个对自己最有利的方案；对“与”节点，选其子节点中一个最小的得分作为父节点的得分，这是为了立足于最坏的情况。
- (5)如果一个行动方案能获得较大的倒推值，则它就是当前最好的行动方案。下图给出了计算倒推值的示例。



倒推值的计算

- 例　一字棋游戏。设有如图4–19(a)所示的九个空格，由A，B二人对弈，轮到谁走棋谁就往空格上放一只自己的棋子，谁先使自己的棋子构成“三子成一线”谁就取得了胜利。

图19　一字棋

- 设A的棋子用“a″表示，B的棋子用“b″表示。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P是A必胜的棋局，则e(P)=+∞。
  - (2)若P是B必胜的棋局，则e(P)=-∞。
  - (3)若P是胜负未定的棋局，
    - 则e(P)=e(+P)-e(-P)

- 其中e(+P)表示棋局P上有可能使a成为三子成一线的数目；e(-P)表示棋局P上有可能使b成为三子成一线的数目。例如，对于图19(b)所示的棋局，则e(P)=6-4=2
- 另外，我们假定具有对称性的两个棋局算作一个棋局。还假定A先走棋，我们站在A的立场上。

下图给出了A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A来说最好的一着棋是S_3，因为S_3比S_1和S_2有较大的倒推值。

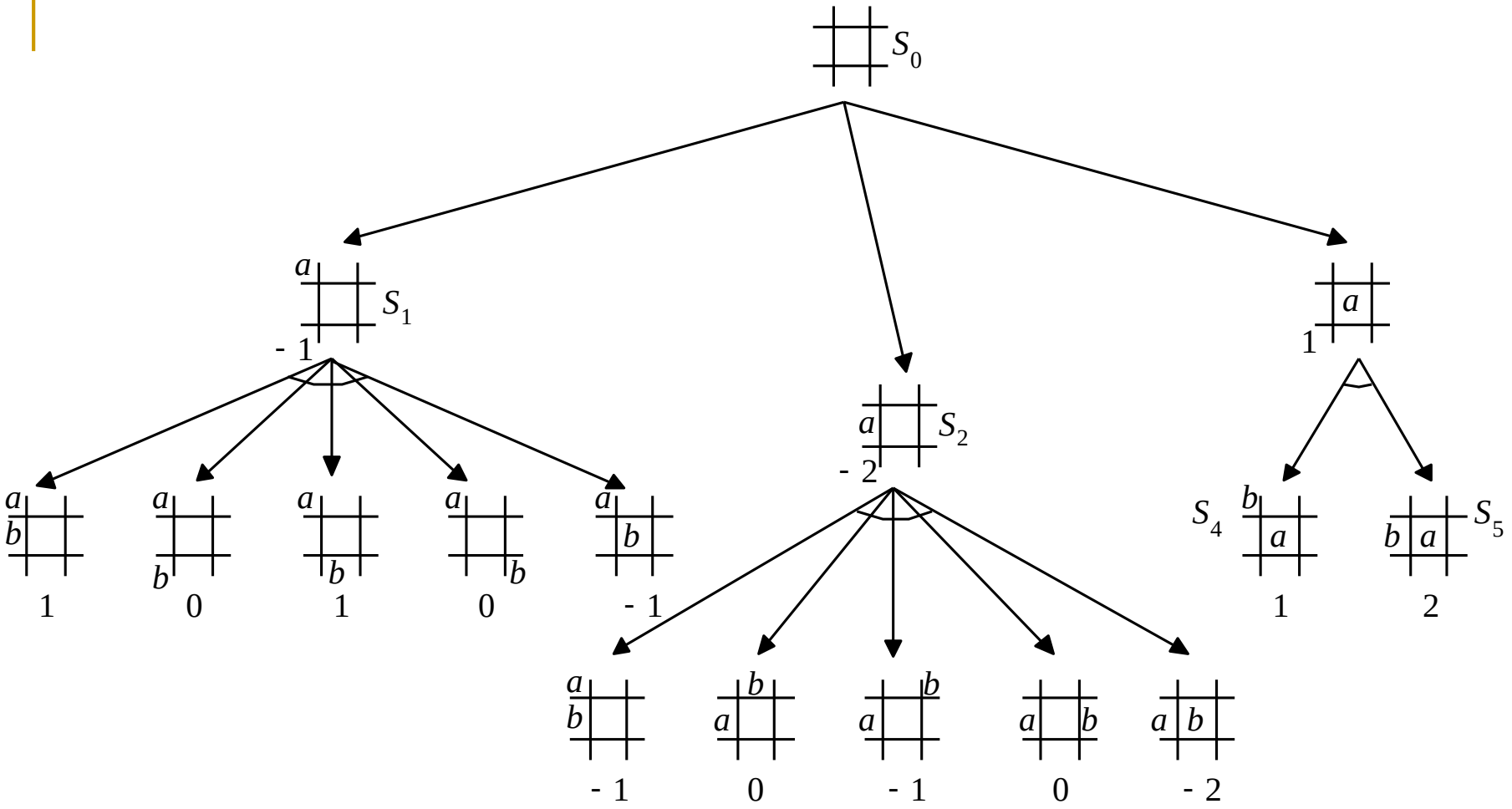


图 一字棋极小极大搜索

- α–β剪枝技术
- 上述的极小极大分析法,实际是先生成一棵博弈树,然后再计算其倒推值。这样做的缺点是效率较低。于是,人们又在极小极大分析法的基础上,提出了α–β剪枝技术。

- 具体的剪枝方法如下：
- (1)对于一个与节点MIN,若能估计出其倒推值的上确界β,并且这个β值不大于MIN的父节点(一定是或节点)的估计倒推值的下确界α,即α≥β,则就不必再扩展该MIN节点的其余子节点了(因为这些节点的估值对MIN父节点的倒推值已无任何影响了)。这一过程称为α剪枝。

- (2)对于一个或节点MAX,若能估计出其倒推值的下确界α,并且这个α值不小于MAX的父节点(一定是与节点)的估计倒推值的上确界β,即α≥β,则就不必再扩展该MAX节点的其余子节点了(因为这些节点的估值对MAX父节点的倒推值已无任何影响了)。这一过程称为β剪枝。

回溯法

- **学习要点**
- 理解回溯法的深度优先搜索策略。
- 掌握用回溯法解题的算法框架
- （1）子集树算法框架
- （2）排列树算法框架

- 通过应用范例学习回溯法的设计策略。
- （1）n后问题；
- 2）图的m着色问题
- （3）哈密尔顿回路问题
- （4）0-1背包问题；
- （5）装载问题；
- （6）批处理作业调度；
- （7）符号三角形问题
- （8）最大团问题；
- （9）圆排列问题
- （10）电路板排列问题
- （11）连续邮资问题
- （12）旅行售货员问题

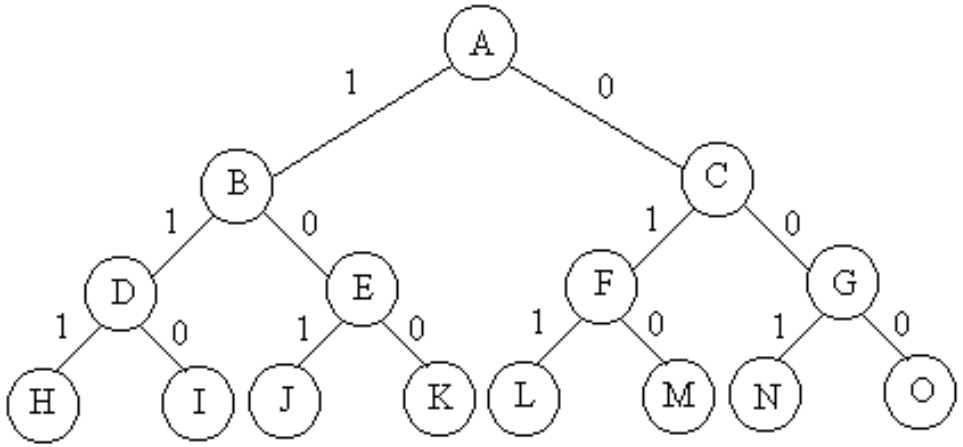
回溯法

- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。
- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式(x1,x2,…,xn)的形式。
- 显约束：对分量x_i的取值限定。

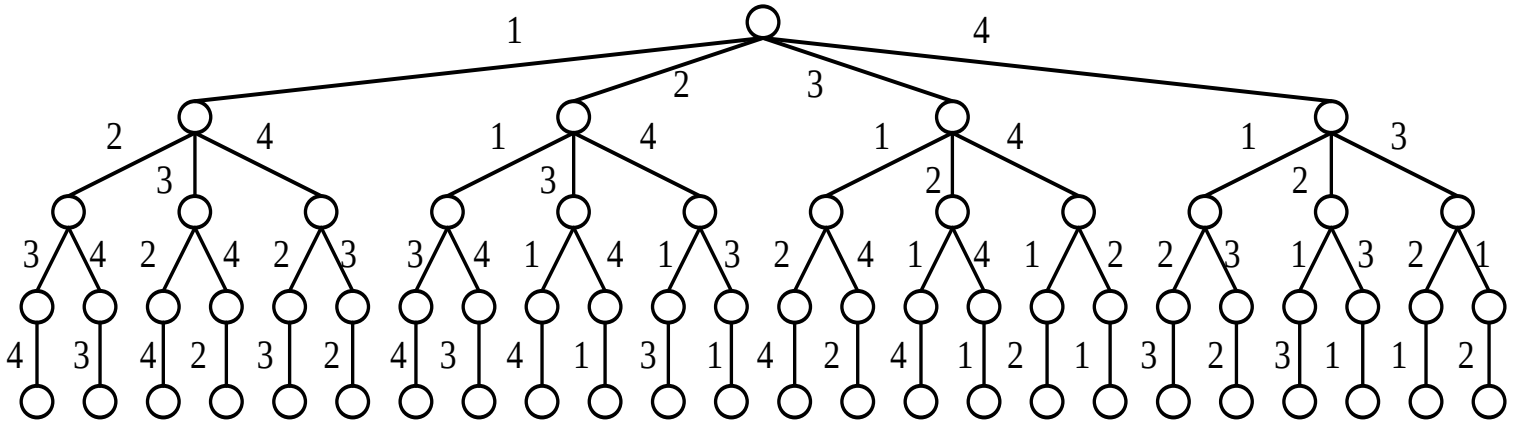
注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



n=3时的0-1背包问题用完全二叉树表示的解空间

问题的解空间

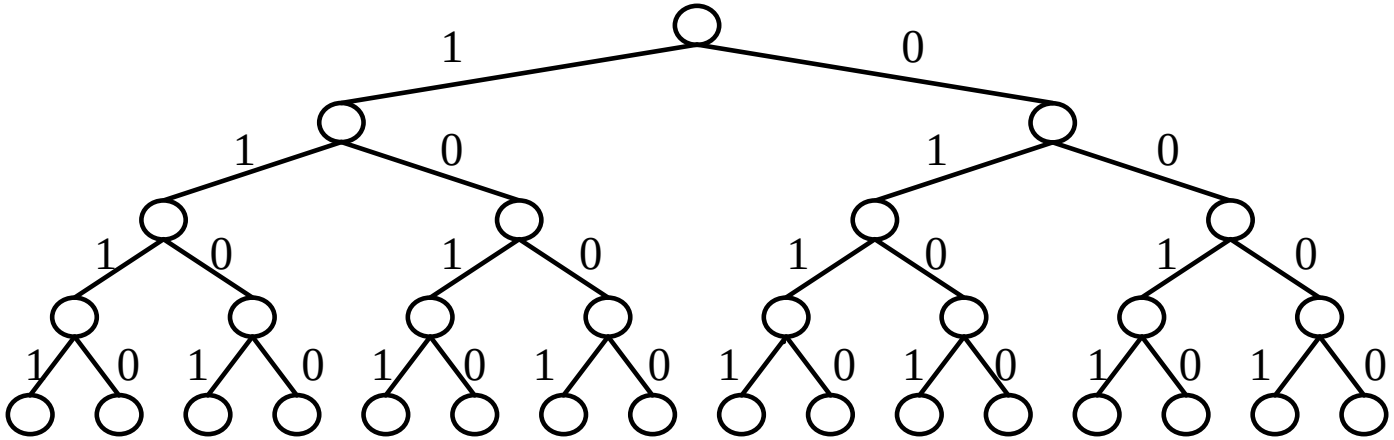
- 解空间压缩:n=4货郎担问题的解空间是？



- n=4时货郎担问题的状态空间树

状态空间树

- 状态空间树：问题解空间的树形表式



- n=4时背包问题的状态空间树

状态空间树的动态搜索

有些问题，只要可行解，不需要最优解，例如八后问题和图的着色问题

- 可行解和最优解
  - ✓ 可行解：满足约束条件的解，解空间中的一个子集
  - ✓ 最优解：使目标函数取极值（极大或极小）的可行解，一个或少数几个
- 例：货郎担问题，n^n有种可能解。（N-1)!种可行解，只有一个或几个解是最优解。
- 例：背包问题，2^n有种可能解，有些是可行解，只有一个或几个是最优解。

生成问题状态的基本方法

- 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：
  - 一个扩展结点变成死结点前，一直是扩展结点
- 回溯法：
  - 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。**具有限界函数的深度优先生成法称为回溯法**

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。

用回溯法解题的一个显著特征: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。

递归回溯

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

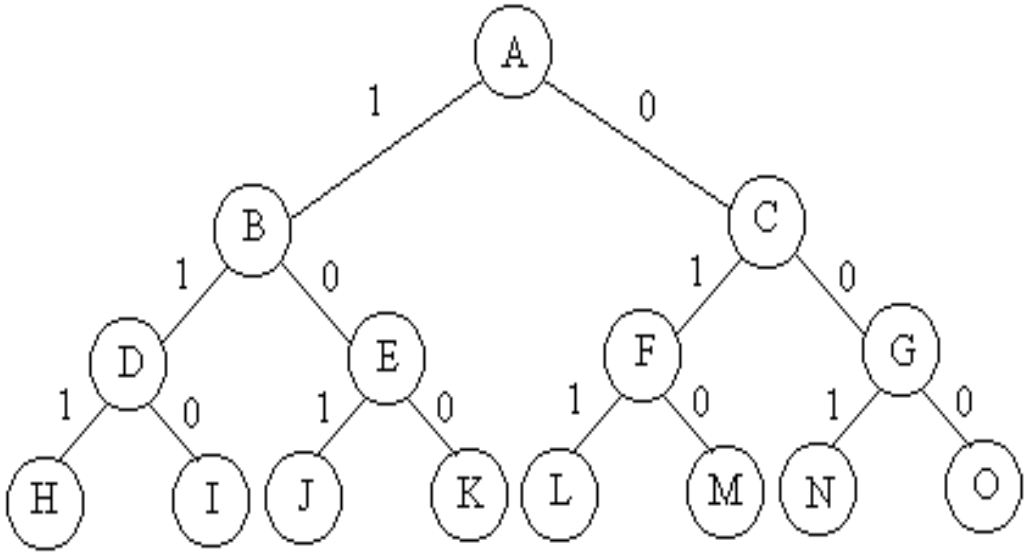
```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t))
                backtrack(t+1);
        }
}
```

迭代回溯

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack ()
{ int t=1;
  while (t>0) {
    if (f(n,t)<=g(n,t))
      for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x); else t++;}
        }
    else t--;
    }
}
```

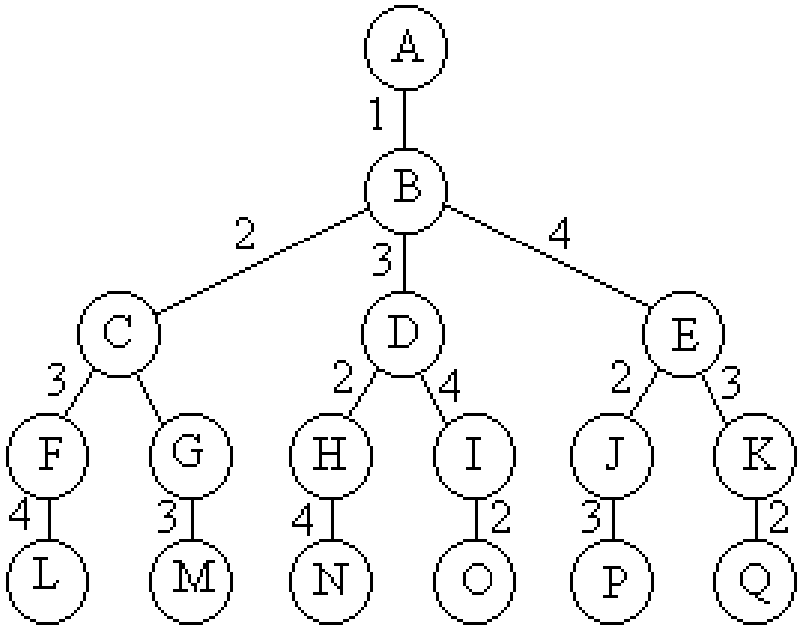
子集树与排列树



遍历子集树需$O(2^n)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
  else
     for (int i=0;i<=1;i++)
     {
        x[t]=i;
        if (legal(t))
           backtrack(t+1);
     }
}
```

子集树与排列树



遍历排列树需要O(n!)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++)
       {
        swap(x[t], x[i]);
        if (legal(t))
         backtrack(t+1);
          swap(x[t], x[i]);
       }
}
```

装载问题

有一批共n个集装箱要装上2艘载重量分别为c_1和c_2的轮船，其中集装箱i的重量为w_i，且

问：$\sum_{i=1}^{n} w_i \le c_1 + c_2$ 装载体积不受限制的情况下，将装箱装上轮船。

装载问题要求确定是否有一个合理的装载方案可将这批集装箱装上这2艘轮船。如果有，找出一种装载方案。

如c1=c2=50 w={10,40,40}

W={20,40,40}

装载问题

容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近c_1。由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\max \sum_{i=1}^{n} w_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

用回溯法设计解装载问题的$O(2^n)$计算时间算法。在某些情况下该算法优于动态规划算法。

装载问题

- 解空间：
  - 子集树
- 可行性约束函数
- （选择当前元素）：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 上界函数
- （不选择当前元素）：

当前载重量cw+剩余集装箱的重量r≤当前最优载重量bestw

装载问题

```
void backtrack (int i)
{// 搜索第i层结点
   if (i > n) // 到达叶结点
   更新最优解bestw;return;
   if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];
   }
    backtrack(i + 1);
}
```

装载问题

```
void backtrack (int i)
{// 搜索第i层结点
   if (i > n) // 到达叶结点
   更新最优解bestx,bestw;return;
   r -= w[i];
   if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];}
   if (cw + r > bestw) {
      x[i] = 0; // 搜索右子树
      backtrack(i + 1);     }
   r += w[i];
}
```

批处理作业调度

✓给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。每个作业必须先由机器1处理，然后由机器2处理。作业J_i需要机器j的处理时间为t_{ji}。

✓对于一个确定的作业调度，设F_{ji}是作业i在机器j上完成处理的时间。所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。

✓批处理作业调度问题要求对于给定的n个作业，制定最佳作业调度方案，使其完成时间和达到最小。

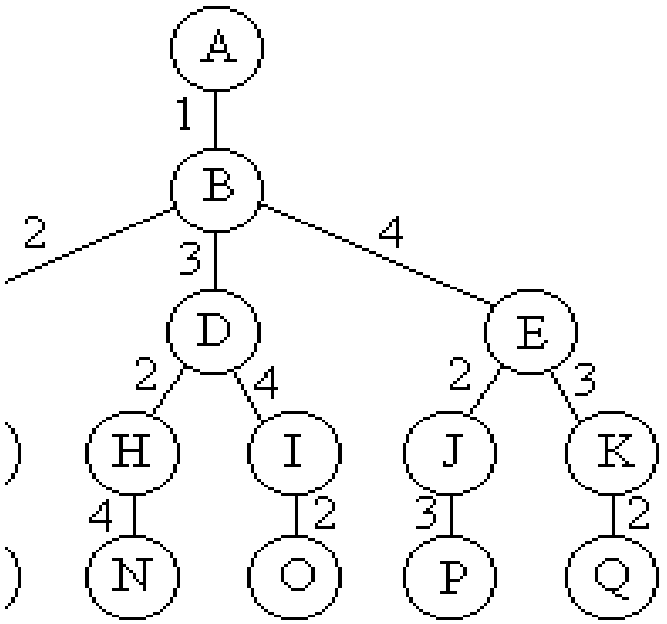
批处理作业调度

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

这3个作业的6种可能的调度方案是
1,2,3；1,3,2；2,1,3；2,3,1；3,1,2；3,2,1；
它们所相应的完成时间和分别是19，18，20，21，19，19。
最佳调度方案是1,3,2，其完成时间和为18。

批处理作业调度

- 解空间：排列树




```
class Flowshop {
 friend Flow(int**, int, int []);
void Backtrack(int i);
  int  **M,//作业所需处理时间
  *x,     // 当前作业调度
  *bestx, //当前最优作业调度
  *f2,    // 机器2完成处理时间
  f1,    // 机器1完成处理时间
  f,     // 完成时间和
  bestf,    // 当前最优值
  n;   // 作业数};
```

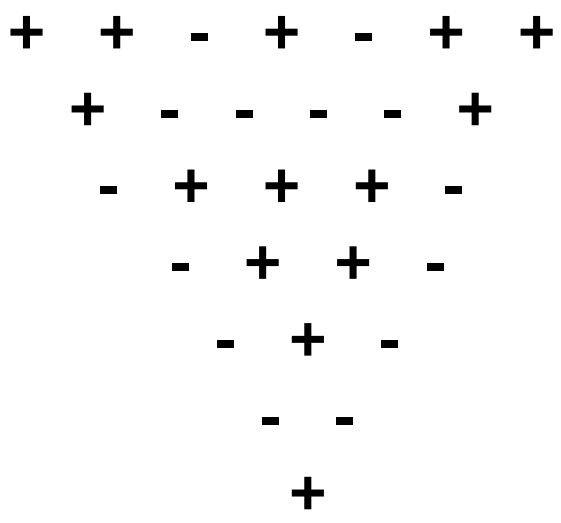
批处理作业调度

```
void Flowshop::Backtrack(int i)
{
   if (i > n) {
      for (int j = 1; j <= n; j++)
        bestx[j] = x[j];
        bestf = f;
      }
   else
```

```
    for (int j = i; j <= n; j++) {
        f1+=M[x[j]][1];
        f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
        f+=f2[i];
        if (f < bestf) {
            Swap(x[i], x[j]);
            Backtrack(i+1);
            Swap(x[i], x[j]);
          }
          f1- =M[x[j]][1];
          f- =f2[i];
    }
}
```

符号三角形问题

下图是由14个“+”和14个“-”组成的符号三角形。2个同号下面都是“+”，2个异号下面都是“-”。



在一般情况下，符号三角形的第一行有n个符号。符号三角形问题要求对于给定的n，计算有多少个不同的符号三角形，使其所含的“+”和“-”的个数相同。

符号三角形问题

✓解向量：

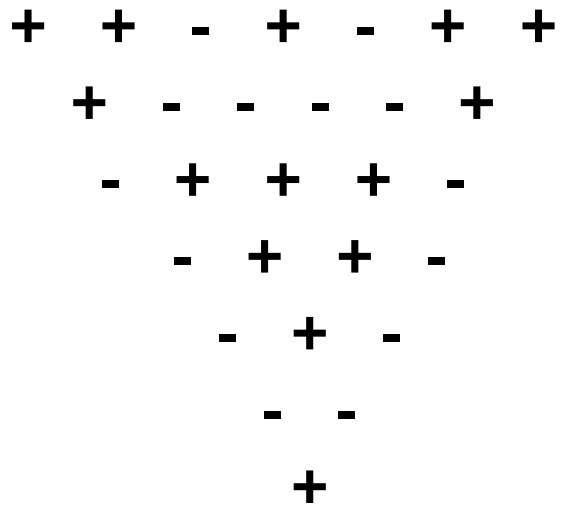
用n元组x[1:n]表示符号三角形的第一行。

✓可行性约束函数：

当前符号三角形所包含的“+"个数与“-"个数均不超过n*(n+1)/4

✓无解的判断：

n*(n+1)/2为奇数



符号三角形问题

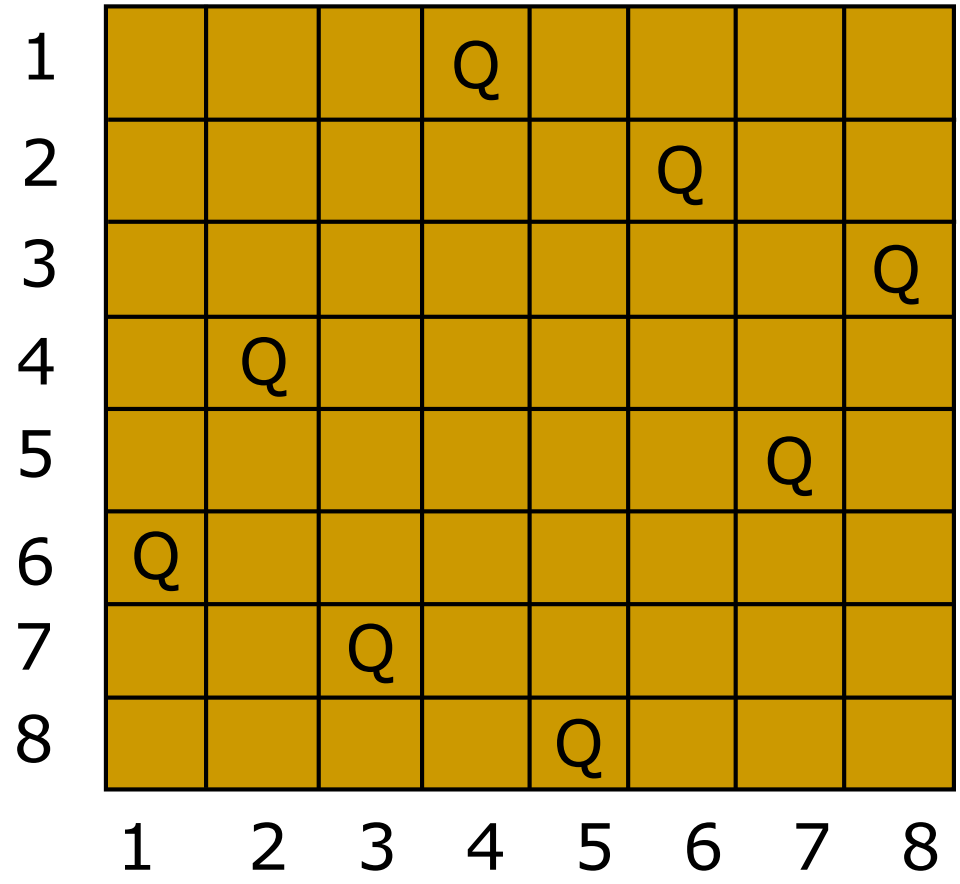
```
void Triangle::Backtrack(int t)
{
  if ((count>half)||(t*(t-1)/2-count>half)) return;
  if (t>n) sum++;
  else  for (int i=0;i<2;i++) {
        p[1][t]=i;
        count+=i;
        for (int j=2;j<=t;j++) {
          p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
          count+=p[j][t-j+1];
        }
      Backtrack(t+1);
      for (int j=2;j<=t;j++)count-=p[j][t-j+1];
      count-=i;
    }
}
```

复杂度分析

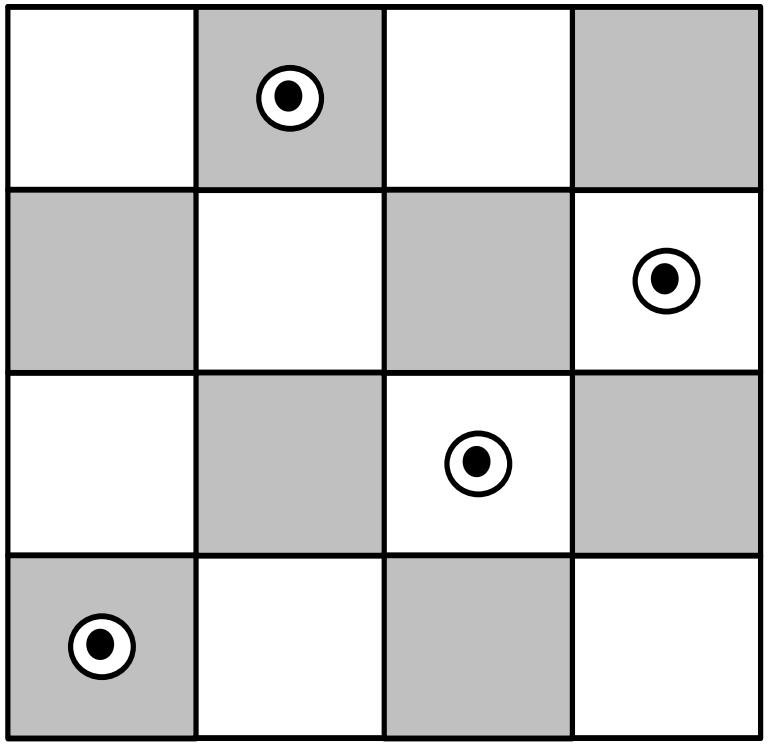
计算可行性约束需要$O(n)$时间，在最坏情况下有$O(2^n)$个结点需要计算可行性约束，故解符号三角形问题的回溯算法所需的计算时间为$O(n2^n)$。

n后问题

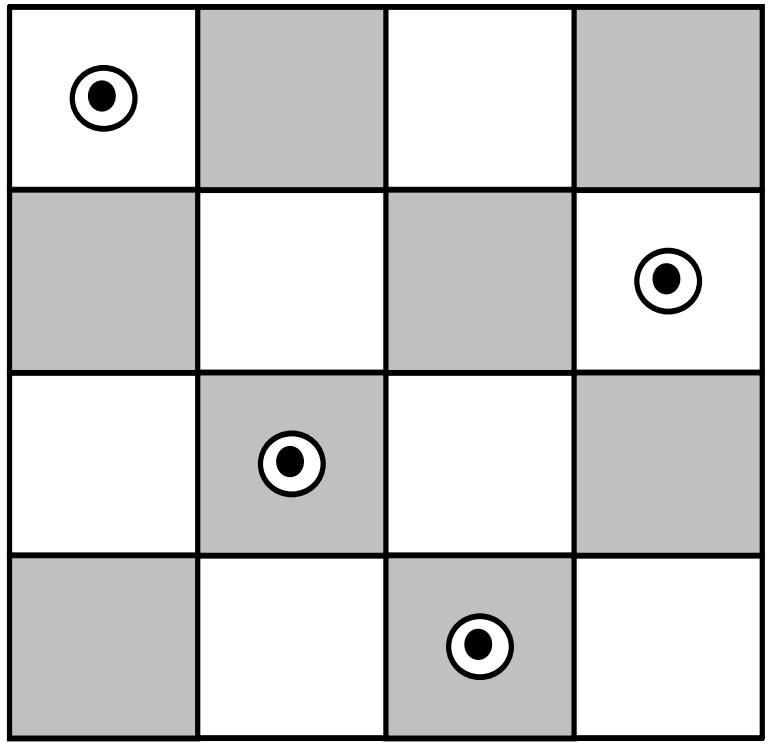
在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。



四后问题的求解过程



（a）



(b)

n后问题

- 解向量：$(x_1, x_2, \cdots, x_n)$
- 显约束：
- $x_i=1,2,\cdots,n$
- 隐约束：

  1)不同列：$x_i \neq x_j$

  2)不处于同一正、反对角线：$|i-j| \neq |x_i-x_j|$

n后问题

```
void Queen::Backtrack(int t)
{  if (t>n) sum++;
    else  for (int i=1;i<=n;i++) {
        x[t]=i;
        if (Place(t)) Backtrack(t+1);
      }
 }
```

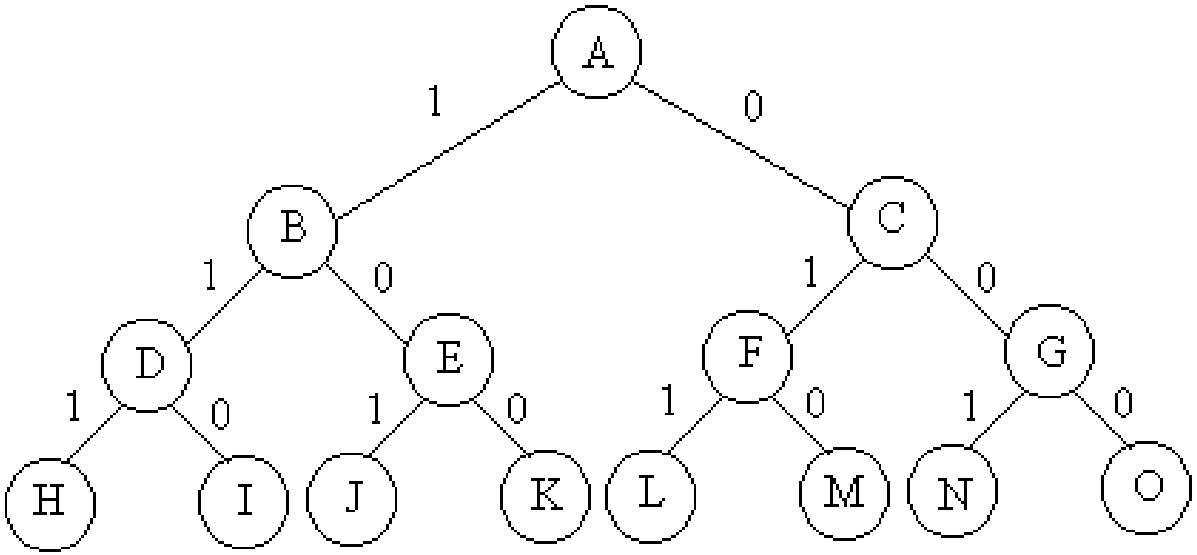
```
bool Queen::Place(int k)
{  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k]))
     return false;
  return true;
}
```

0/1背包问题

- 不需把背包的载重量划分为等分、物体的重量是背包载重量等分的整数倍的限制。

0-1背包问题

- 解空间：
  - 子集树
- 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 上界函数：



0-1背包问题

```
void backtrack (int i)
{// 搜索第i层结点
   if (i > n)  // 到达叶结点
   更新最优解bestx,bestp;return;
if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      cp+=p[i]
      backtrack(i + 1);
      cw -= w[i];  cp-=p[i];
      }
   if (bound(i+1)> bestp) {
      x[i] = 0;  // 搜索右子树
      backtrack(i + 1);     }
}
```

0-1背包问题

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i)
{// 计算上界
   Typew cleft = c - cw;  // 剩余容量
   Typep b = cp;
   // 以物品单位重量价值递减序装入物品
   while (i <= n && w[i] <= cleft) {
      cleft -= w[i];
      b += p[i];
      i++;
      }
   // 装满背包
   if (i <= n) b += p[i]/w[i] * cleft;
   return b;
}
```

最大团问题

✓给定无向图$G=(V, E)$。

✓如果$U \subseteq V$，且对任意$u, v \in U$有$(u, v) \in E$，则称U是G的完全子图。

✓G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。

✓G的最大团是指G中所含顶点数最多的团。

最大团问题

✓给定无向图$G=(V, E)$。
如果$U \subseteq V$且对任意$u, v \in U$有$(u, v) \notin E$，则称U是G的空子图。

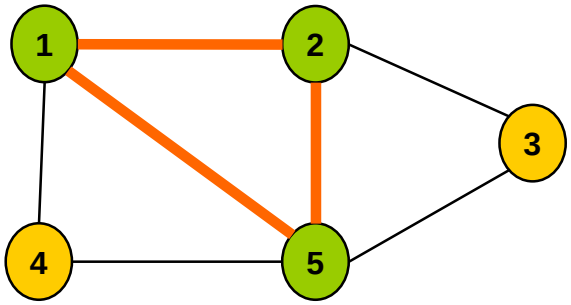
✓G的空子图U是G的独立集当且仅当U不包含在G的更大的空子图中。

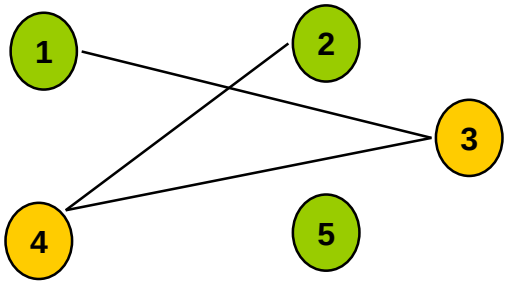
✓G的最大独立集是G中所含顶点数最多的独立集。

最大团问题

对于任一无向图$G=(V, E)$其补图$\bar{G}=(V1, E1)$定义为：$V1=V$，且$(u, v)\in E1$当且仅当$(u, v)\notin E$。

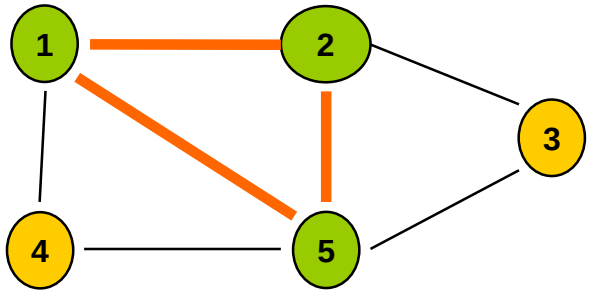
U是G的最大团当且仅当U是G的最大独立集。



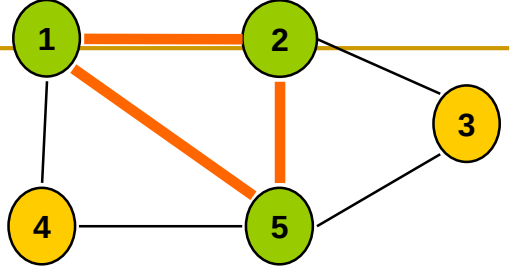


最大团问题

- 解空间：子集树
- 可行性约束函数：顶点i到已选入的顶点集中每一个顶点都有边相连。
- 上界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。



最大团问题




```
void Clique::Backtrack(int i)
{// 计算最大团
   if (i > n) {// 到达叶结点
      for (int j = 1; j <= n; j++) bestx[j] = x[j];
      bestn = cn;   return;}
   int OK = 1; // 检查顶点 i 与当前团的连接
   for (int j = 1; j < i; j++)
      if (x[j] && a[i][j] == 0) {// i与j不相连
         OK = 0;  break;}
   if (OK) {// 进入左子树
      x[i] = 1;  cn++;
      Backtrack(i+1);
      x[i] = 0; cn--;}
   if (cn + n - i > bestn) {// 进入右子树
      x[i] = 0;    Backtrack(i+1);}
}
```

复杂度分析

最大团问题的回溯算法backtrack所需的计算时间显然为$O(n2^n)$。

进一步改进

- **选择合适的搜索顺序**，可以使得上界函数更有效的发挥作用。例如在搜索之前可以将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。
- 定义Si={v_i,v_{i+1},…,v_n}，依次求出S_n,S_{n-1},…,S_1的解。从而得到一个更精确的上界函数，若cn+S_i<=max则剪枝。同时注意到：从S_{i+1}到S_i，如果找到一个更大的团，那么vi必然属于找到的团，此时有Si=S_{i+1}+1，否则S_i=S_{i+1}。因此只要max的值被更新过，就可以确定已经找到最大值，不必再往下搜索了。

图的m着色问题

给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。

图的m可着色判定问题:是否有一种着色法使G中每条边的2个顶点着不同颜色。

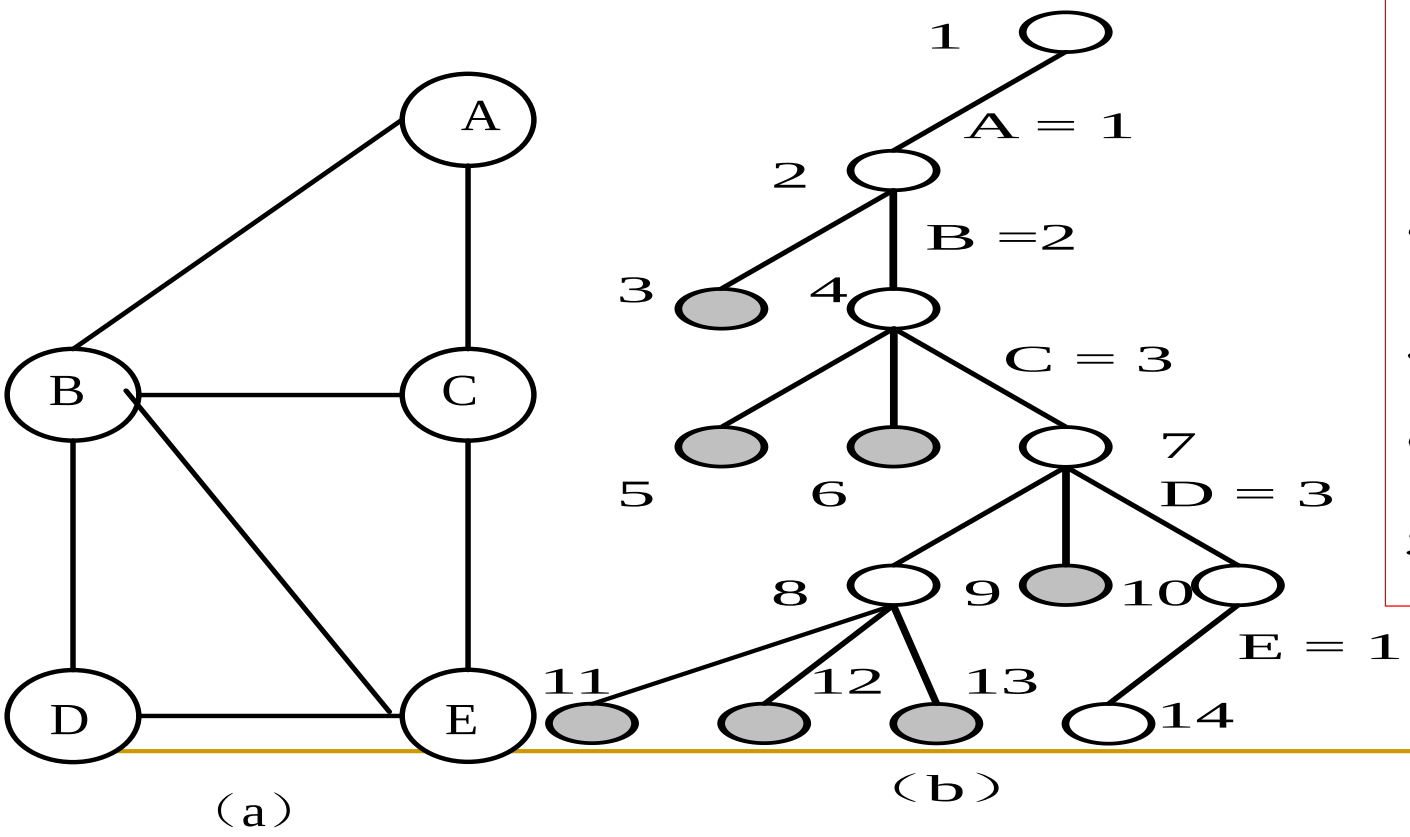
若一个图最少需要m种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数m为该图的色数。

求一个图的色数m的问题称为图的m可着色优化问题。

搜索过程

- 约束方程：x[i]≠x[j]若顶点i与顶点相j邻接
- 回溯法解图三着色的例子

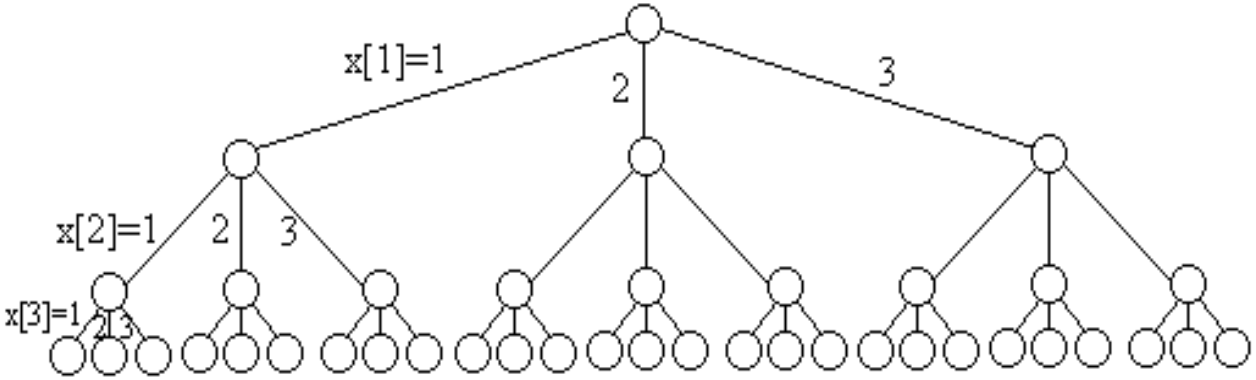
状态空间树结点总数为：1+3+9+27+81+243=364。而在搜索过程中所访问的结点数只有14个。



（a）　（b）

图的m着色问题

- 解向量：$(x_1, x_2, \ldots, x_n)$表示顶点i所着颜色x[i]
- 可行性约束函数：顶点i与已着色的相邻顶点颜色不重复。



图的m着色问题

```
void Color::Backtrack(int t)
{  if (t>n) {
   sum++;
   for (int i=1; i<=n; i++)
     cout << x[i] << ' ';
   cout << endl;
   }
    else for (int i=1;i<=m;i++) {
        x[t]=i;
        if (Ok(t))
           Backtrack(t+1);
      }
}
```

图的m着色问题

```
bool Color::Ok(int k)
{// 检查颜色可用性
  for (int j=1;j<=n;j++)
    if ((a[k][j]==1)&&(x[j]==x[k])) return false;
  return true;
}
```

图的m着色问题

复杂度分析

图m可着色问题的解空间树中内结点个数是$\sum_{i=0}^{n-1} m^i$

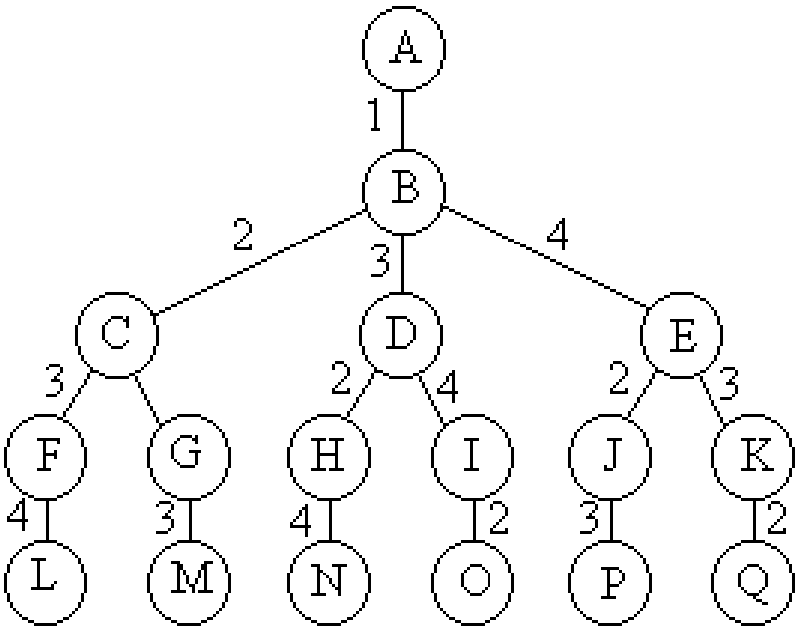
对于每一个内结点，在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时O(mn)。

因此，回溯法总的时间耗费是

$$\sum_{i=0}^{n-1} m^i(mn) = nm(m^n - 1)/(m-1) = O(nm^n)$$

旅行售货员问题

- 解空间：
  - 排列树



遍历排列树需要O(n!)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++)
       {
        swap(x[t], x[i]);
        if (legal(t))
         backtrack(t+1);
          swap(x[t], x[i]);
       }
}
```

旅行售货员问题

- 解空间： 排列树

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
   if (i == n) {
      if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge
&& (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc ==
NoEdge))
      { for (int j = 1; j <= n; j++) bestx[j] = x[j];
        bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
      }
   else {
```

旅行售货员问题

```
else {
    for (int j = i; j <= n; j++)
      // 是否可进入x[j]子树?
      if (a[x[i-1]][x[j]] != NoEdge &&
        (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) {
        // 搜索子树
        Swap(x[i], x[j]);
        cc += a[x[i-1]][x[i]];
        Backtrack(i+1);
        cc -= a[x[i-1]][x[i]];
        Swap(x[i], x[j]);}
    }
}
```

复杂度分析

算法backtrack在最坏情况下可能需要更新当前最优解$O((n-1)!)$次，每次更新bestx需计算时间$O(n)$，从而整个算法的计算时间复杂性为$O(n!)$。

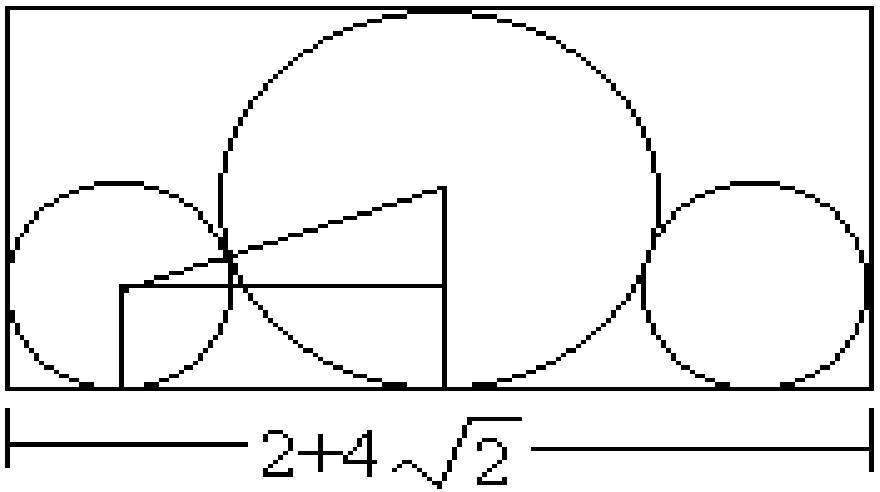
圆排列问题

给定n个大小不等的圆c1,c2,…,cn，现要将这n个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。

圆排列问题:要求从n个圆的所有排列中找出有最小长度的圆排列。

例:当n=3，且所给的3个圆的半径分别为1，1，2时，这3个圆的最小长度的圆排列如图所示。其最小长度为

$$2+4\sqrt{2}$$



```
void Circle::Compute(void)
{// 计算当前圆排列的长度
    float low=0,high=0;
    for (int i=1;i<=n;i++) {
        if (x[i]-r[i]<low) low=x[i]-r[i];
        if (x[i]+r[i]>high) high=x[i]+r[i];
        }
    if (high-low<min) min=high-low;
}
```

```
float Circle::Center(int t)
{// 计算当前所选择圆的圆心横坐标
    float temp=0;
    for (int j=1;j<t;j++) {
        float valuex=x[j]+2.0*sqrt(r[t]*r[j]);
        if (valuex>temp) temp=valuex;
        }
    return temp;
}
```

圆排列问题

```
void Circle::Backtrack(int t)
{
if (t>n) Compute();
else for (int j = t; j <= n; j++)
    { Swap(r[t], r[j]);
    float centerx=Center(t);
    if (centerx+r[t]+r[1]<min){下界约束
       x[t]=centerx;
       Backtrack(t+1);
     }
    Swap(r[t], r[j]);
   }
}
```

圆排列问题

复杂度分析

由于算法backtrack在最坏情况下可能需要计算O(n!)次当前圆排列长度，每次计算需O(n)计算时间，从而整个算法的计算时间复杂性为O((n+1)!)

- 上述算法尚有许多改进的余地。例如，象1,2,…,n-1,n和n,n-1, …,2,1这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。另一方面，如果所给的n个圆中有k个圆有相同的半径，则这k个圆产生的k!个完全相同的圆排列，只计算一个就够了。

连续邮资问题

假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许贴m张邮票。

连续邮资问题:对于给定的n和m的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例如，当n=5和m=4时，面值为(1,3,11,15,32)的5种邮票可以贴出邮资的最大连续邮资区间是1到70。

连续邮资问题

- 解向量：用n元组x[1:n]表示n种不同的邮票面值，并约定它们从小到大排列。x[1]=1是唯一的选择。
- 可行性约束函数：已选定x[1:i-1]，最大连续邮资区间是[1:r]，接下来x[i]的可取值范围是[x[i-1]+1:r+1]。

连续邮资问题

如何确定r的值？

计算X[1:i]的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法。考虑到直接递归的求解复杂度太高，我们不妨尝试计算用不超过m张面值为x[1:i]的邮票贴出邮资k所需的最少邮票数y[k]。通过y[k]可以很快推出r的值。事实上，y[k]可以通过递推在O(n)时间内解决：

连续邮资问题

```
for (int j=0; j<= x[i-2]*(m-1);j++)
        if (y[j]<m)
          for (int k=1;k<=m-y[j];k++)
            if (y[j]+k<y[j+x[i-1]*k])
                  y[j+x[i-1]*k]=y[j]+k;
while (y[r]<maxint) r++;
```

回溯法效率分析

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；

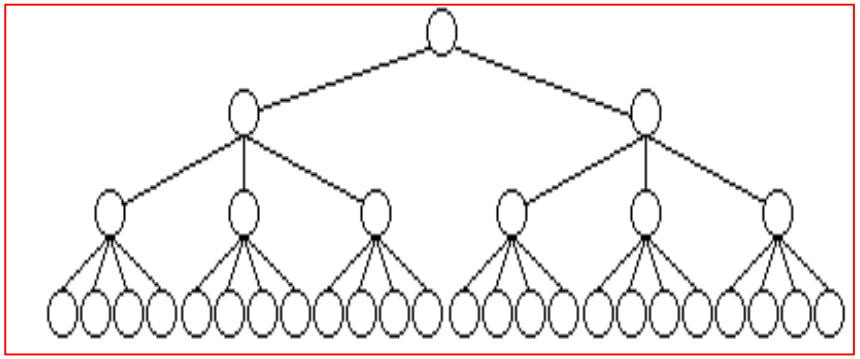
(5)满足约束函数和上界函数约束的所有x[k]的个数。

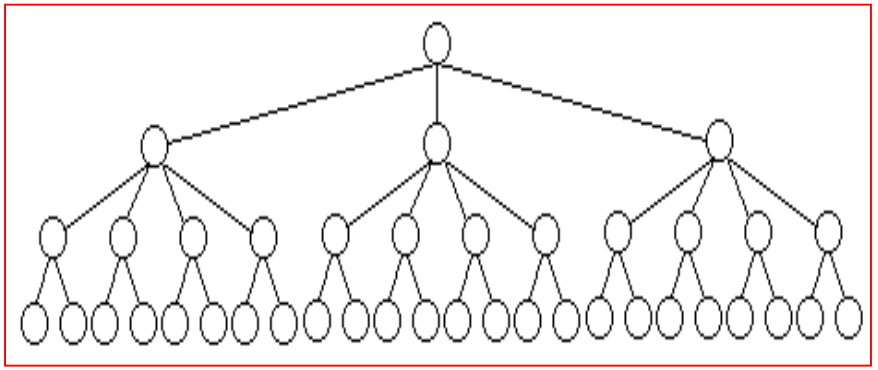
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。

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。

重排原理

对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。**在其它条件相当的前提下，让可取值最少的x[i]优先**。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。





图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

子集和问题

- 问题
  - 给定由n个不同正数组成的集合W=$\{w_i\}$，和正数M，求W中所有和等于M的子集的集合；
  - 例如n = 6, M = 30, W={10, 13, 5, 18, 12, 15}

子集和问题

- 按照回溯法思想，从状态树的根结点出发，做深度优先搜索；
- 为便于计算，将W中的正数按从小到大排序；
- 当在某一状态A下，依次尝试加入和不加入正数w_i，若$\sum A+w_i>M$，则可停止对该结点的搜索；若$\sum A+\sum(w_i...w_n)<M$，则也可停止对该结点的搜索；

有一位探险家用5天的时间徒步横穿A、B两村，两村间是荒无人烟的沙漠，如果一个人只能担负3天的食物和水，那么这个探险家至少雇几个人才能顺利通过沙漠。

A城雇用一人与探险家同带3天食物同行一天，然后被雇人带一天食物返回，并留一天食物给探险家，这样探险家正好有3天的食物继续前行，并于第三天打电话雇B城

人带3天食物出发，第四天会面他们会面，探险家得到一天的食物赴B城。