第五章

回溯法

Backtracking

回溯法概念与基本思想

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）递归回溯

（2）迭代回溯

（3）子集树与排列树

通过算法实例学习回溯法的设计策略

5.1 相关概念

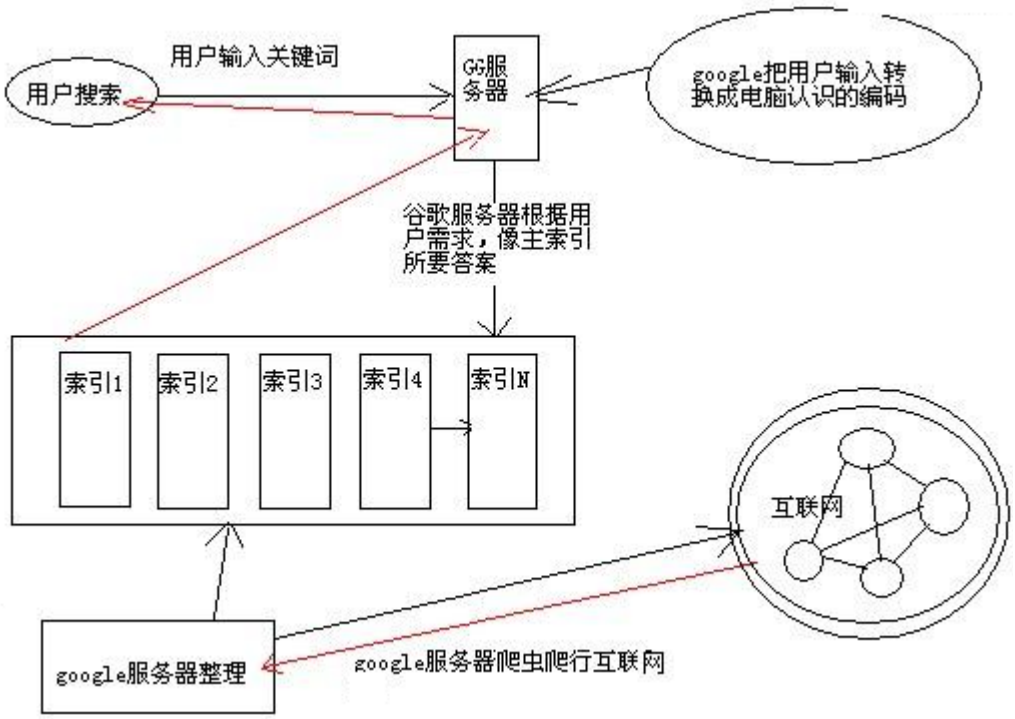
回溯法策略

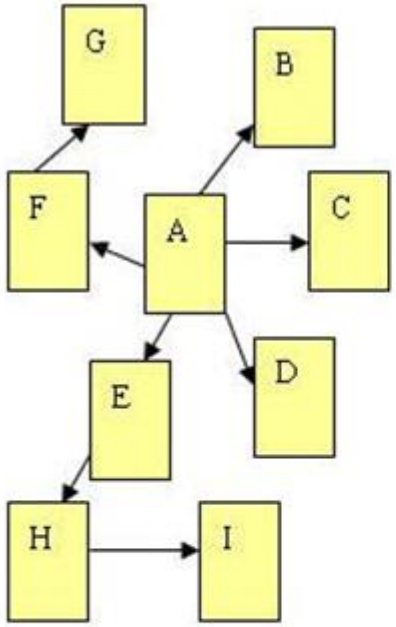
- 回溯（backtracking）是一种系统地搜索问题解答的方法。是一种能避免不必要搜索的穷举式搜索法。该方法适用于解一些组合数相当大的问题。
- 为了实现回溯，首先需要为问题定义一个**解空间**（solution space），这个空间必须至少包含问题的一个解（可能是最优的）。
- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法在包含问题所有解的解空间树中，按**深度优先策略**，从根结点出发搜索解空间树。算法搜索至解空间树的任意一结点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。
- 回溯法又被称为“通用解题法”**怎么理解？**

回溯法指导思想——走不通，就掉头。

5.1 相关概念　　回溯法策略-深度优先

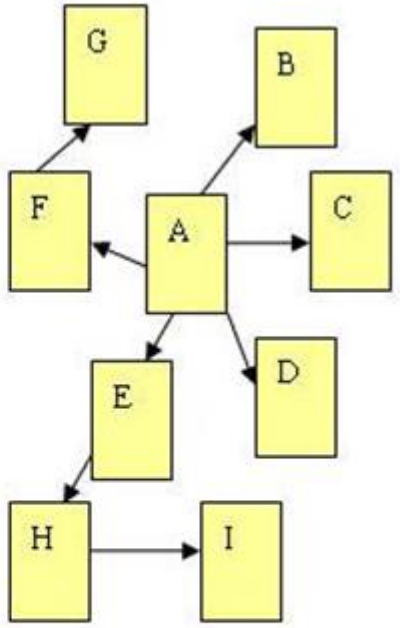
什么是深度优先与广度优先？以搜索引擎中的互联网爬虫为例





广度优先的抓取顺序：
A—B.C.D.E.F—H.G—I

广度优先



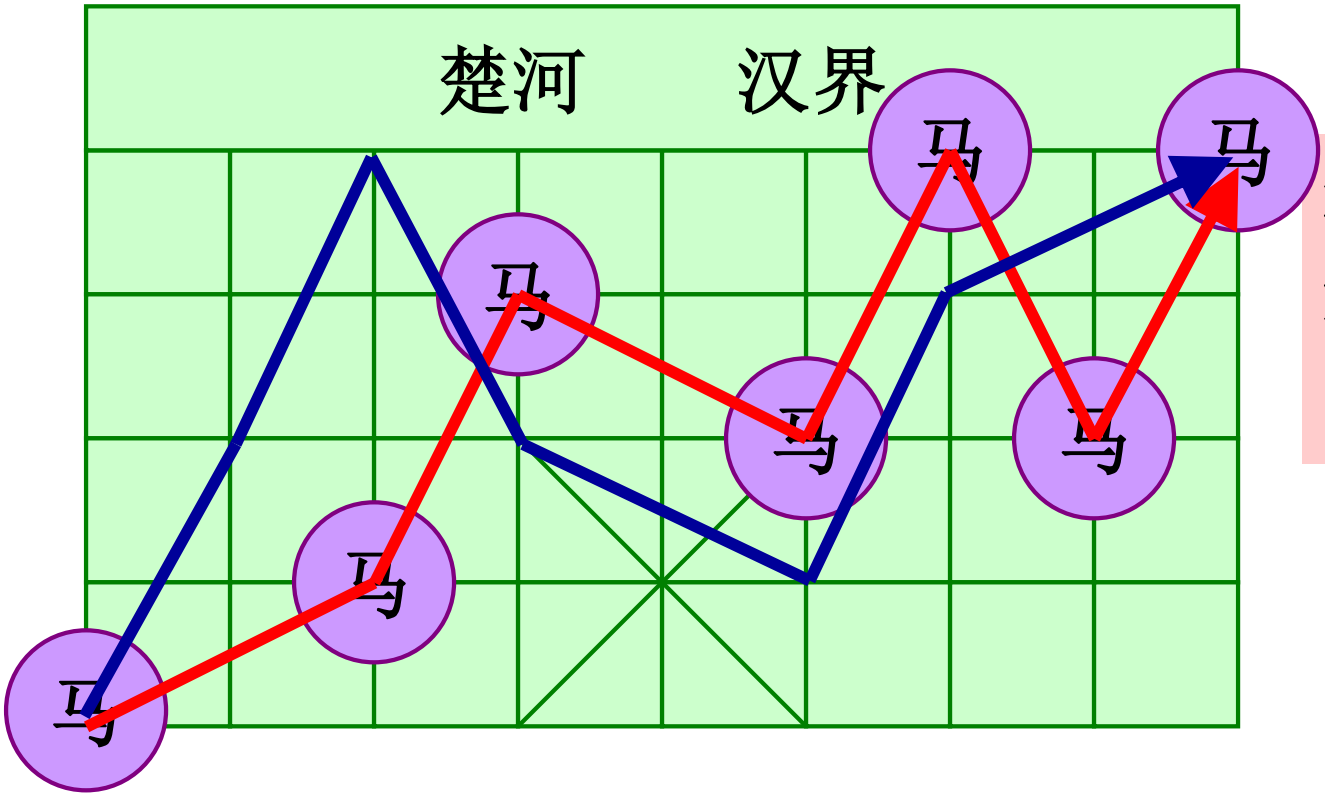
深度优先的抓取顺序：
A—F—G
E—H—I

深度优先

5.1 相关概念

回溯法

- 回溯法核心思想：“试探着走”。如果试得不成功则退回一步，再换一个方法试。反复进行这种试探性选择与返回纠错过程，直到求出问题的解为止。
- 实例：有一只中国象棋的马，在半张棋盘的左下角出发，向右上角跳去。若规定只许向右跳（可上，可下，但不允向左跳），编一个程序，求出从起点A到终点B共有多少种不同跳法。



规定只许向右跳，即可上，可下，但不允向左跳！

5.1 相关概念

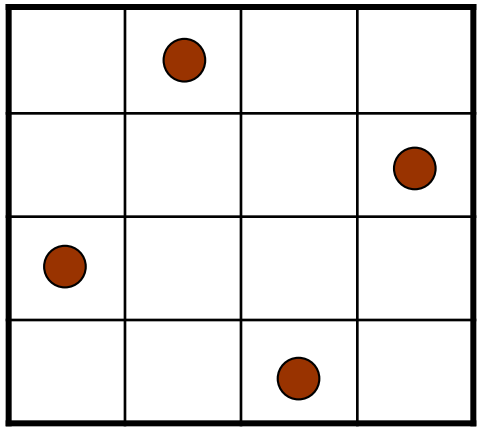
回溯法

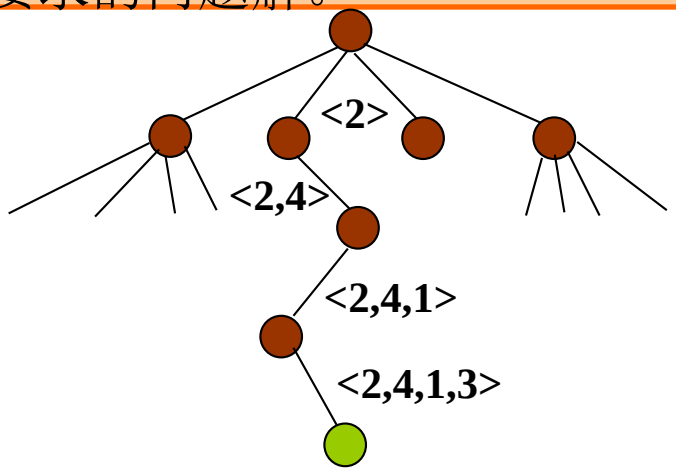
- 在算法设计策略中，**回溯法**是比贪心法和动态规划法更一般的方法。
- 回溯法是一种通过**搜索解空间树**（状态空间树）来求问题的可行解或最优解的方法。**最坏情况？**
- 回溯法通过使用**约束函数**和**限界函数**来压缩需要实际生成的状态空间树的结点数，从而大大节省问题求解时间。

再例：N后问题

在N×N的棋盘中放置N个皇后，使得任何两个皇后之间不能相互攻击，试给出所有的放置方法。

问题所有可行解分布在集合$\{\langle x_1, x_2, \cdots, x_n\rangle \mid 1 \leqslant x_i \leqslant N, 1 \leqslant i \leqslant N\}$（解空间）之中。可将问题解空间表示为一定的结构，通过对解空间的搜索，得到满足要求的问题解。





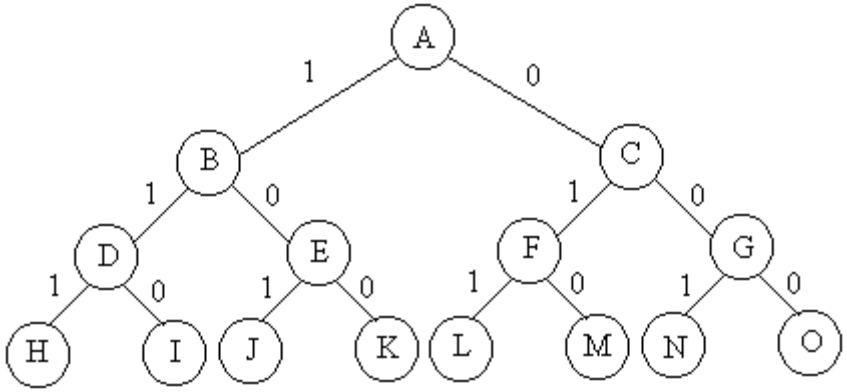
5.1 相关概念

问题的解空间

复杂问题常常有很多的可能解，这些可能解构成了问题的解空间。解空间也就是进行穷举的搜索空间，所以，解空间中应该包括所有的可能解。确定正确的解空间很重要，如果没有确定正确的解空间就开始搜索，可能会增加很多重复解，或者根本就搜索不到正确的解。

问题的解空间一般用解空间树（Solution Space Trees,也称状态空间树）的方式组织，树的根结点位于第1层，表示搜索的初始状态，第2层的结点表示对解向量的第一个分量做出选择后到达的状态，第1层到第2层的边上标出对第一个分量选择的结果，依此类推，从树的根结点到叶子结点的路径就构成了解空间的一个可能解。

n=3时的0-1背包问题用完全二叉树表示的解空间



5.1 相关概念

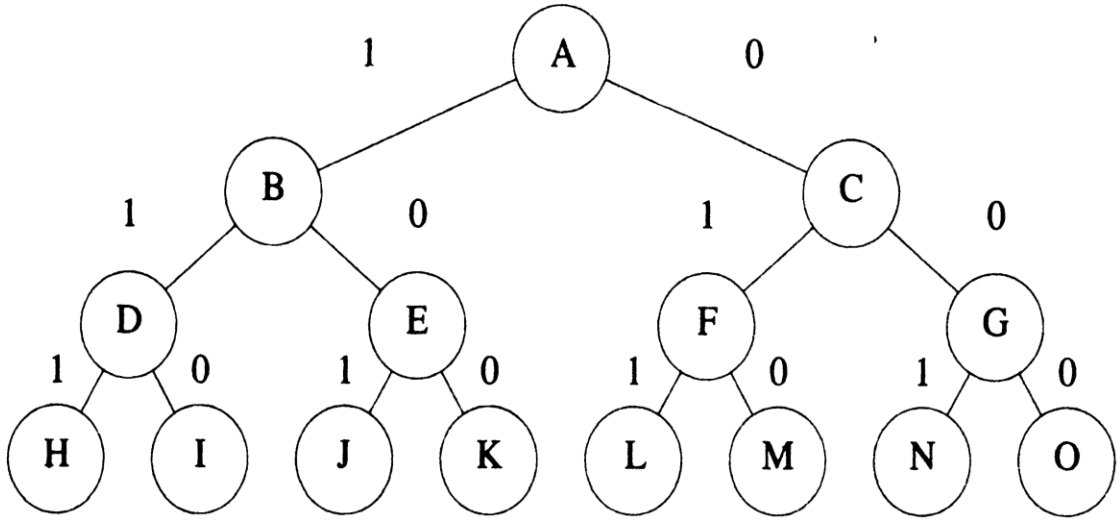
问题的解空间

处理一个复杂的问题，常常会有很多可能解，这些可能解构成了问题的解空间。解空间也就是进行穷举的搜索空间，所以，解空间中应该包括所有的可能解。

举例：0-1背包问题。对于有n=3个物品的0/1背包问题，其可能解的表示：

可能解由一个等长向量$\{x_1, x_2, \ldots, x_n\}$组成，其中$x_i$=1($1 \le i \le n$)表示物品$i$装入背包，$x_i$=0表示物品$i$没有装入背包，当$n$=3时，其解空间是：

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



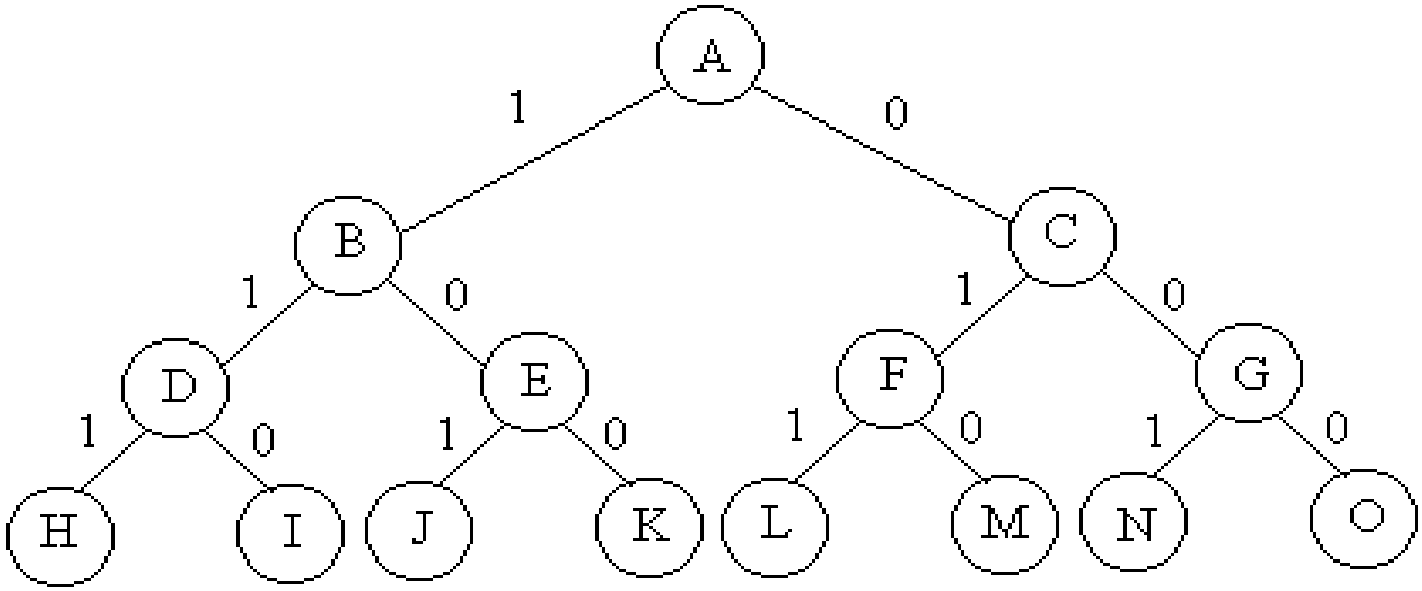
5.2 基本思想

回溯法基本思想及步骤

(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。

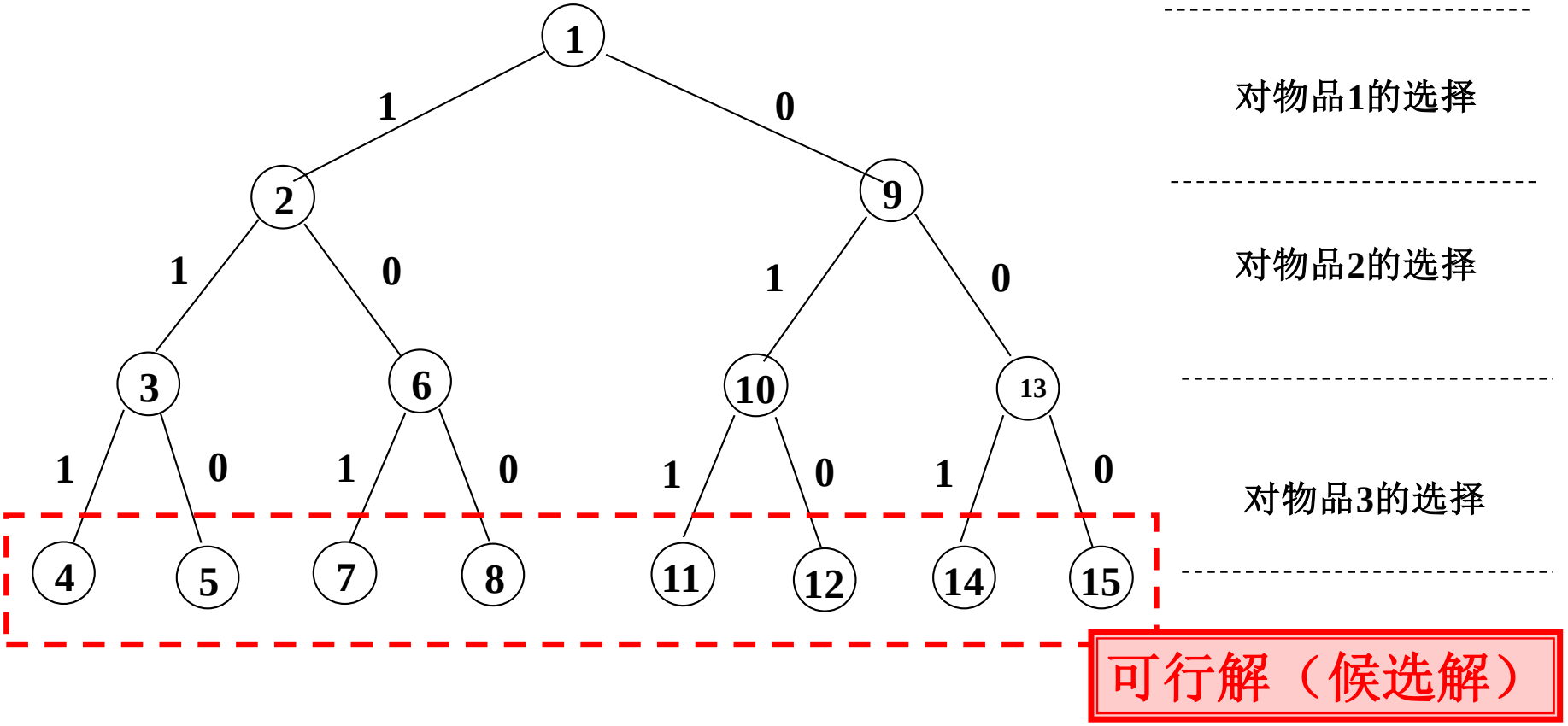
常用剪枝函数：

- **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。**



5.2 基本思想　　解空间结构

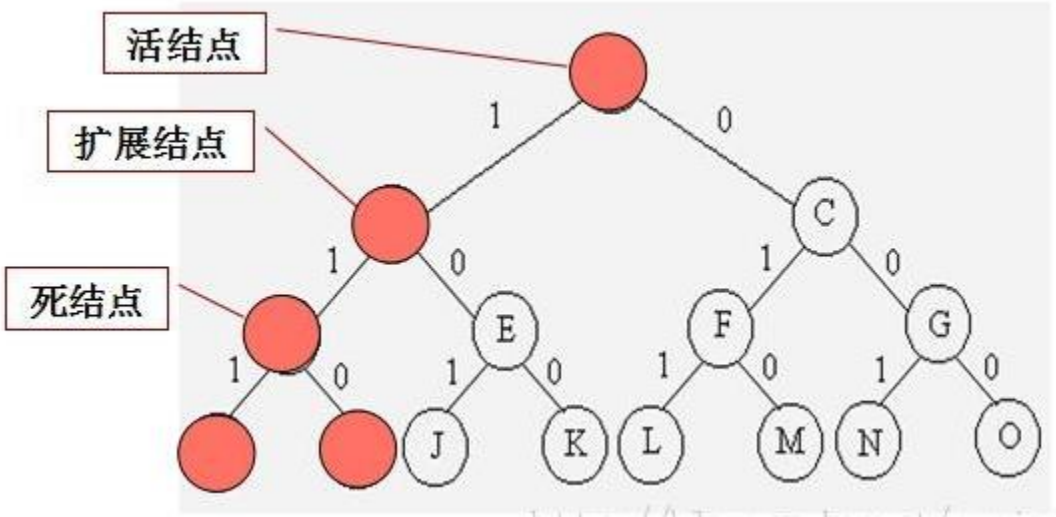
对于n=3的0/1背包问题，其解空间树如下图所示，树中的8个叶子结点分别代表该问题的8个可能解。



对于一个解结构形式为n元组的问题，其可能解构成了问题的解空间，可以用一颗解空间树表示，树的所有叶子结点为可能解。

5.2 基本思想　　剪枝避免无效搜索

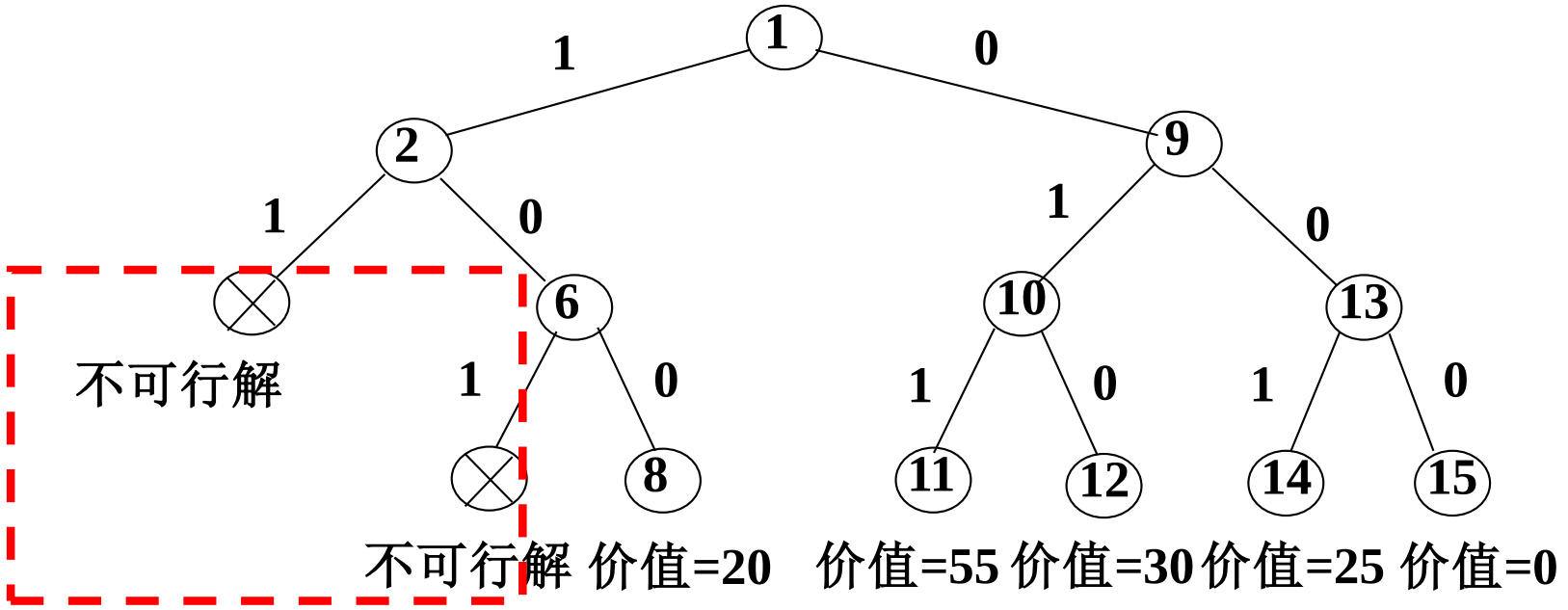
扩展结点: 一个正在产生儿子的结点称为扩展结点
活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
死结点:一个所有儿子已经产生的结点称做死结点



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。

0-1背包实例

例如，对于n=3的0/1背包问题，三个物品的重量为{20, 15, 10}，价值为{20, 30, 25}，背包容量为25，从下图所示的解空间树的根结点开始搜索，搜索过程如下：



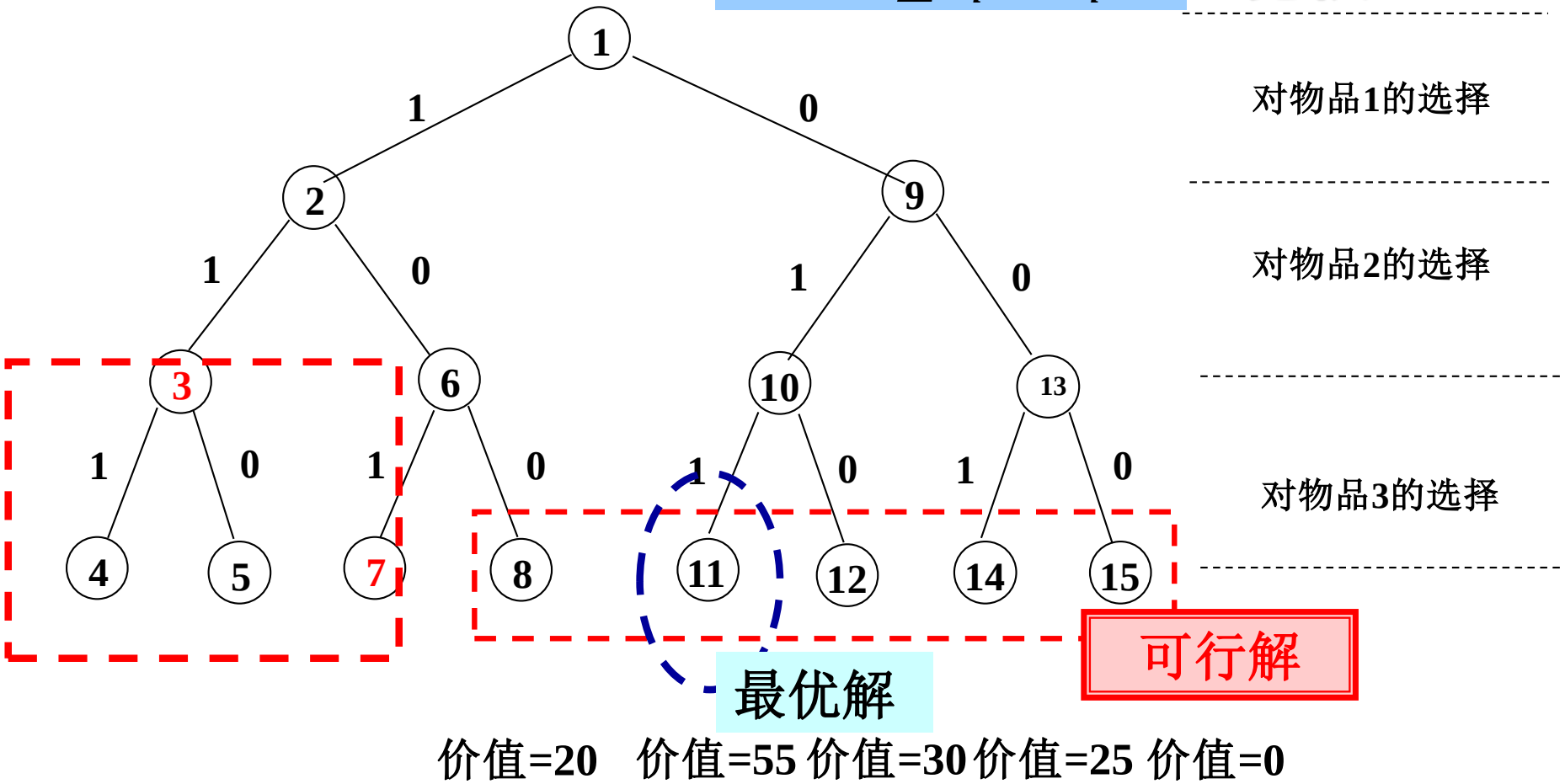
为了避免生成那些不可能产生最佳解的问题状态，要不断地利用**限界函数（约束函数剪枝）**来让那些实际上不可能产生所需解的活结点变成死结点，以减少问题的计算量

5.2 基本思想

0-1背包实例

- 目标函数，也称代价函数（cost function）用来衡量每个可行解的优劣，使目标函数取最大（或最小）值的可行解为问题的最优解。

$$V = \sum v_i \times x_i$$



5.3 实现策略 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。

剪枝？

```
void backtrack (int t)
{
    if (t>n) output(x); // 到达叶子结点，将结果输出
    else
      for (int i=f(n,t);i<=g(n,t);i++) {// 遍历结点t的所有子结点
        x[t]=h(i);
        if (constraint(t)&&bound(t)) backtrack(t+1);
        }
}
```

t-递归深度，即当前扩展结点在解空间树中的深度。
n-用来控制递归深度，即解空间树的高度。当t>n时，算法已搜索到一个叶结点
output(x)对得到的可行解x进行记录或输出处理。
f(n,t)和g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。
h(i)表示在当前扩展结点处x[t]的第i个可选值，x[]表示当前部分解
constraint(t)和bound(t)表示在当前扩展结点处的约束函数和限界函数

5.3 实现策略 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)) //当前节点存在子节点
      for (int i=f(n,t);i<=g(n,t);i++) {
        //遍历当前节点的所有子节点
        x[t]=h(i); //每个子节点的值赋值给x
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x);
          else t++; //没有得到解，继续向下搜索}
        }
    else t--; //不存在子节点，返回上一层
    }}
```

Solution(t)判断当前扩展结点处是否已到问题的一个可行解。返回值为true表示在当前可扩展结点处x[1:t]是问题的一个可行解。若返回值为false则表示在当前扩展结点处x[1:t]只是问题的一个部分解，还需要向纵深方向继续搜索。

f(n,t)和g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。

h(i)表示在当前扩展结点处x[t]的第i个可选值。

5.4 两种解空间树

在用回溯法求解问题时，常常遇到两种典型的解空间树：

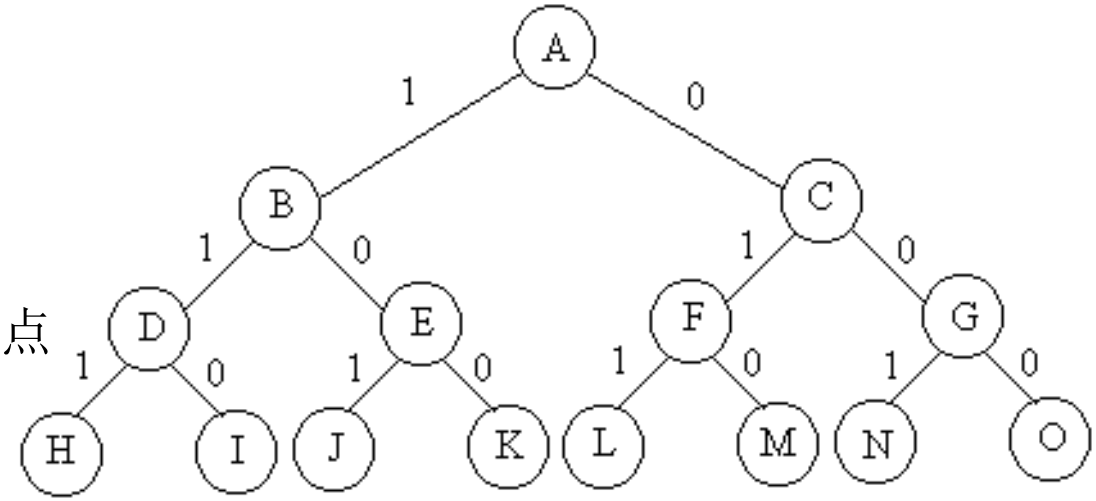
（1）子集树（Subset Trees）：当所给问题是从n个元素的集合中找出满足某种性质的子集时，相应的解空间树称为子集树。在子集树中，$|S_1|=|S_2|=\ldots=|S_n|=c$，即每个结点有相同数目的子树，通常情况下$c=2$，所以，子集树中共有$2^n$个叶子结点，因此，遍历子集树需要$\Omega(2^n)$时间。（如：0/1背包问题）

（2）排列树（Permutation Trees）：当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。在排列树中，通常情况下，$|S_1|=n$，$|S_2|=n-1$，…，$|S_n|=1$，所以，排列树中共有$n!$个叶子结点，因此，遍历排列树需要$\Omega(n!)$时间。（如：旅行售货员问题）

5.4 两种解空间树

子集树　如：0/1背包问题

从n个元素的集合中找出满足某种性质的子集，相应的解空间树称为子集树。子集树通常有2^n个叶结点。遍历子集树需$O(2^n)$计算时间。



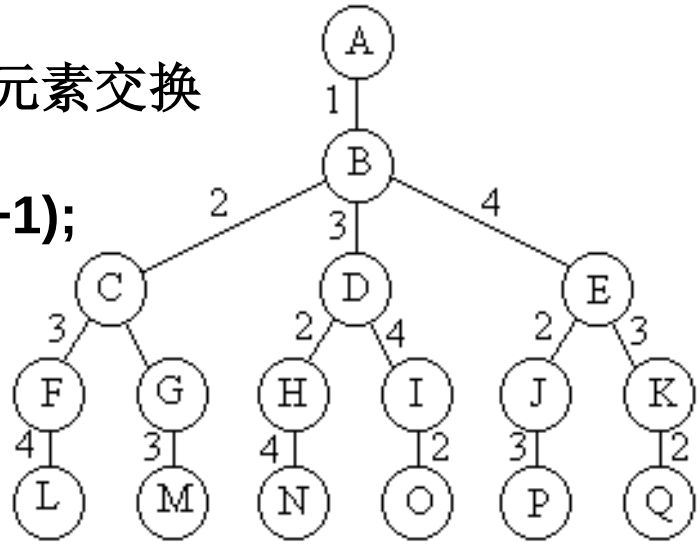

```
void backtrack (int t)
{
  if (t>n) output(x);  // 到达叶子结点
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (constraint(t)&&bound(t)) backtrack(t+1);  // 约束函数
      }
}
```

5.4 两种解空间树

排列树　如：旅行售货员问题

要确定n个元素的满足某种性质的排列时，相应的解空间树称为排列树。排列数通常有n!个叶结点。因此遍历排列树需要0(n!)计算时间。

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        // 更换排列顺序，当前第一个元素与下面元素交换
        swap(x[t], x[i]);
        if (constraint(t)&&bound(t)) backtrack(t+1);
        //搜索完毕后恢复
        swap(x[t], x[i]);
      }
}
```



5.5 回溯法描述 求解过程可表示为在一棵解空间树作深度优先搜索

[解空间树构造]

1.子集树:当解向量为不定长n元组时，树中从根至每一点的路径集合构成解空间.树的每个结点称为一个解状态,有儿子的结点称为可扩展结点,叶结点称为终止结点,若结点v对应解状态$(x_1, x_2, \ldots x_i)$,则其儿子对应扩展的解状态$(x_1, x_2, \ldots x_i, x_{i+1})$.

2.排序树:当解向量为定长n元组时，树中从根至叶结点的路径的集合构成解空间.树的每个叶结点称为一个解状态.

[搜索过程]

搜索按深度优先策略从根开始，当搜索到任一结点时,判断该点是否满足约束条件D(剪枝函数),满足则继续向下深度优先搜索,否则跳过该结点以下的子树(剪枝),向上逐级回溯.

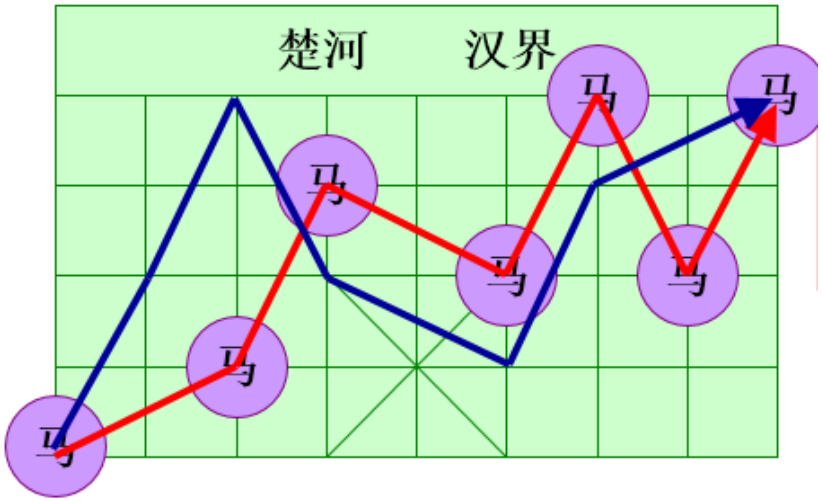
[求解步骤]

1).针对所给问题,定义问题的解空间
2).确定约束条件.
3).以深度优先方式搜索解空间.

回溯法实例：中国象棋马的棋盘遍历问题

在n＊m的棋盘中，马只能走“日”字。马从位置(x,y)处出发，把棋盘的每一格都走一次，且只走一次。找出所有路径。

- 问题1：问题解的搜索空间？
  - 棋盘的规模是n*m，是指行有n条边，列有m条边。
  - 马在棋盘的点上走，所以搜索空间是整个棋盘上的n*m个点。
  - 用n*m的二维数组记录马行走的过程，初值为0表示未经过。

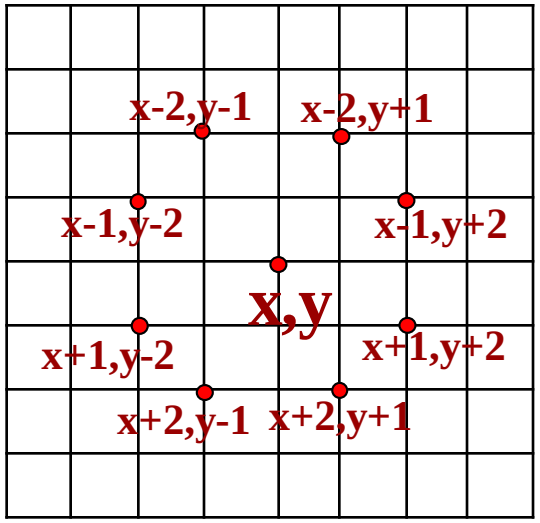


回溯法实例：中国象棋马的棋盘遍历问题

在n＊m的棋盘中，马只能走“日”字。马从位置(x,y)处出发，把棋盘的每一格都走一次，且只走一次。找出所有路径。

- 问题2：在寻找路径过程中，活结点的扩展规则？
  - 对于棋盘上任意一点A(x,y)，有八个扩展方向：

A(x+1,y+2),A(x+2,y+1)
A(x+2,y-1),A(x+1,y-2)
A(x-1,y-2),A(x-2,y-1)
A(x-2,y+1),A(x-1,y+2)



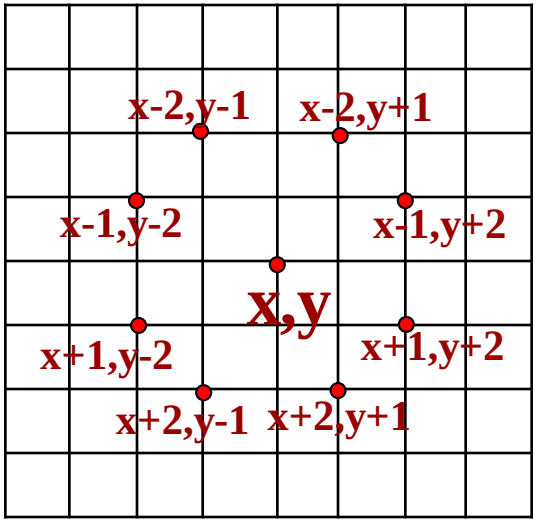
  - 为构造循环体，用数组fx[8]={1,2,2,1,-1,-2,-2,-1}, fy[8]= {2,1,-1,-2,-2,-1,1,2}来模拟马走“日”时下标的变化过程。

回溯法实例：中国象棋马的棋盘遍历问题

在n＊m的棋盘中，马只能走“日”字。马从位置(x,y)处出发，把棋盘的每一格都走一次，且只走一次。找出所有路径。

- 问题3：扩展的约束条件？
  - 不出边界；
  - 每个点只经过一次。

棋盘点对应的数组元素初值为0，对走过的棋盘点的值置为所走步数，起点存储“1”，终点存储“n*m”。



- 函数check，检查当前状态是否合理。

回溯法实例：中国象棋马的棋盘遍历问题

在n＊m的棋盘中，马只能走“日”字。马从位置(x,y)处出发，把棋盘的每一格都走一次，且只走一次。找出所有路径。

- 问题4：搜索解空间？

搜索过程是从任一点（x,y）出发，按深度优先的原则，从8个方向中尝试一个可以走的棋盘点，直到走过棋盘上所有n＊m个点。用递归算法易实现此过程。

注意问题要求找出全部可能的解，就要注意回溯过程的清理现场工作，也就是置当前位置为未经过。

回溯法实例：中国象棋马的棋盘遍历问题

```
int  n=5 , m=4, dep , i, x , y , count;
int  fx[8]={1,2,2,1,-1,-2,-2,-1},fy[8]={2,1,-1,-2,-2,-1,1,2},a[n][m];
main( )
{ count=0;  dep=1;
  print('input x,y');      input(x,y);
  if (y>n  or  x>m  or  x<1  or  y<1)
       { print('x,y error!');   return;}
  for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
         a[i][j]=0;
  a[x][y]=1;
  find(x,y,2);
  if  (count=0 )    print("No answer!");
  else   print("count=",count);
}
```

数据结构定义：

1) 用一个变量dep记录递归深度，也就是走过的点数，当dep=n＊m时，找到一组解。

2) 用n＊m的二维数组a记录马行走的过程，初始值为0表示未经过。搜索完毕后，起点存储的是“1”，终点存储的是 “n＊m”。

回溯法实例：中国象棋马的棋盘遍历问题

```
find(int x,int y,int dep)
{ int  i,xx,yy;
  for (i=1;i<=8;i++)      //加上方向增量,形成新的坐标
  { xx=x+fx[i];  yy=y+fy[i];
    if (check(xx,yy)=1) //判断新坐标是否出界,是否已走过
     { a[xx,yy]=dep;      //走向新的坐标
       if (dep=n*m)  output( );
       else
           find(xx,yy,dep+1);    //从新坐标出发,递归下一层
     }
  }
  a[xx,yy]=0;           //回溯,恢复未走标志
}
```

```
output( )
{     count=count+1;
      print(“换行符”);
        print(“count=”,count);
        for (x=1;x<=n;i++)
        { print(“换行符”);
          for (y=1;y<=m;y++)
              print(a[x,y]);
          }
}
```

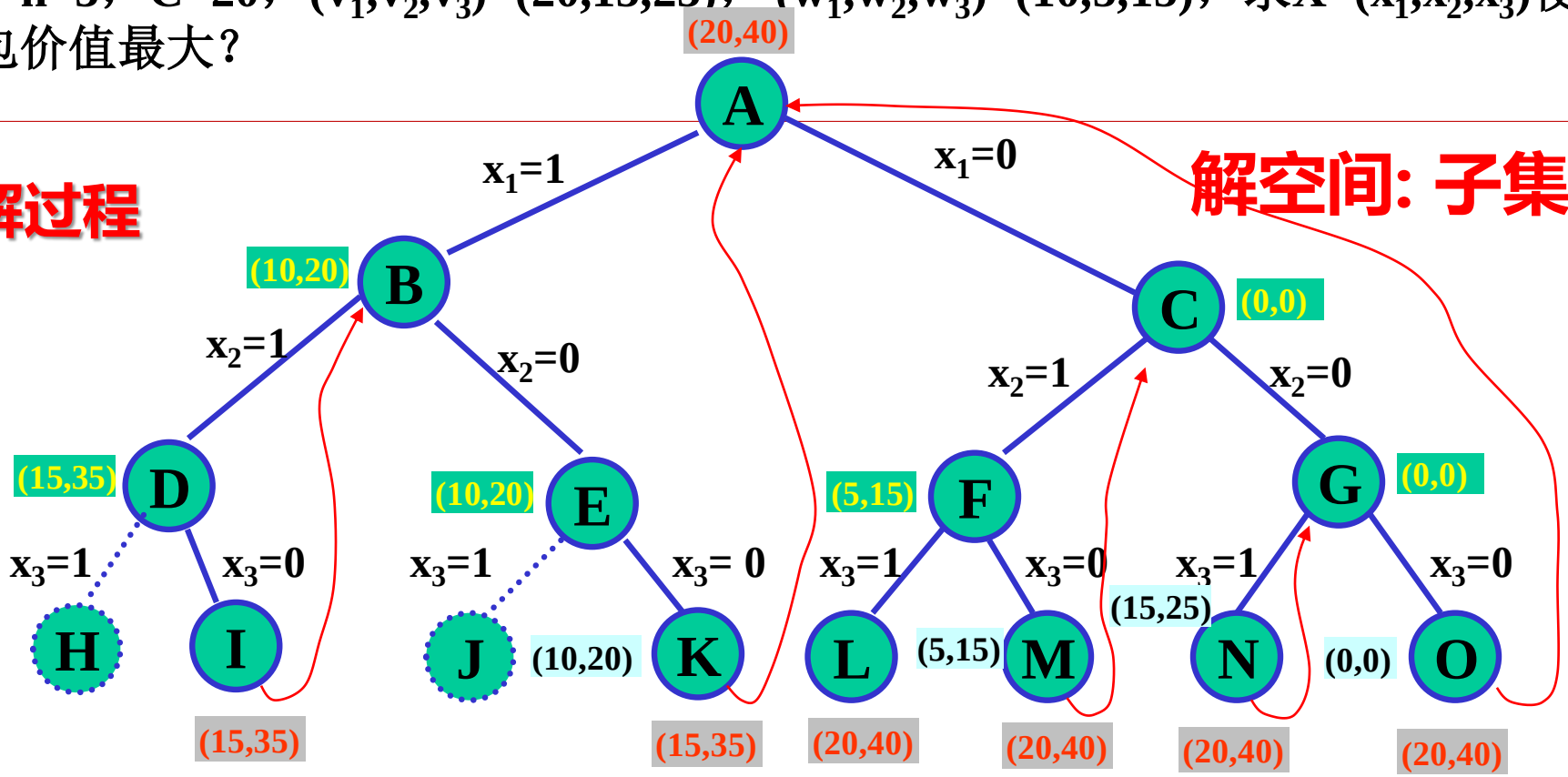
5.6 回溯法解0-1背包问题

设有n个物体和一个背包,物体i的重量为w_i价值为v_i,背包的载荷为 c1, 若将物体$i(1\leq i \leq n,)$装入背包,则有价值为v_i .
目标是找到一个方案,使得能放入背包的物体总价值最高。

例：n=3，C=20，$(v_1,v_2,v_3)=(20,15,25)$，$(w_1,w_2,w_3)=(10,5,15)$，求$X=(x_1,x_2,x_3)$使背包价值最大?

求解过程

解空间: 子集树



当前最优解　中间计算结果　可行解

5.6 回溯法解0-1背包问题

算法描述

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:

当前价值cw+剩余容量可容纳的最大价值<=当前最优价值bestp。

```
Double c;//背包容量
Int n;//物品数
Double [] w;//物品重量数组
Double [] v;//物品价值数组
Double cw;//当前重量
Double cv;//当前价值
Double bestv;//当前最优价值
```

```
double Bound(int i)
{// 计算上界
   double cleft = c - cw;  // 剩余容量
   double bestv = cv;//当前价值
   // 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];
      bestv += v[i];
      i++;
      }
   // 背包有空隙时，装满背包
   if (i <= n) bestv += v[i]/w[i] * cleft;
   return bestv;
}
void BackTrack(int i)
{
    if (i>n) {bestv=cv;return;}//到达叶结点
    if (cw+w[i]<=c)
    {  //进入左子树
       cw+=w[i];cv+=v[i];
       BackTrack(i+1);
       cw-=w[i];cv-=v[i];
    }
    if (bound(i+1)>bestv)
       BackTrack(i+1);//进入右子树
}
```

算法理解？

剪枝？

5.7 *回溯法解装载问题

问题描述:n个集装箱装到2艘载重量分别为c1, c2的货轮, 其中集装箱i的重量为wi且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$. 问题要求找到一个合理的装载方案可将这n个货箱装上这2艘轮船。

例如 当n=3, c1=c2=50, w=[10, 40, 50], 有解;
若w=[20, 40, 40], 问题无解.

若装载问题有解，采用如下策略可得一个最优装载方案:
(1)将第一艘轮船尽可能装满; (2)将剩余的货箱装到第二艘船上。将第一艘船尽可能装满等价于如下0-1背包问题:

$$\max \sum_{i=1}^{n} w_i x_i, \sum_{i=1}^{n} w_i x_i \le c_1,\ x_i \in \{0,1\},\ 1 \le i \le n$$

当 $\sum_{i=1}^{n} w_i = c_1 + c_2$ 时, 问题等价子集和问题;

5.7 *回溯法解装载问题

[算法思路]

用排序树表示解空间,则解为n元向量{x1, ... ,xn }, $x_i \in \{0, 1\}$

约束条件: $\sum_{i=1}^{j} w_i x_i + w_{j+1} \le c1$

由于是最优化问题: $\max \sum_{i=1}^{n} w_i x_i$,可利用此条件进一步剪去不含最优解的子树.

设 bestw: 当前最优载重量,(某个叶节点)

cw = $\sum_{i=1}^{j} w_i x_i$:当前扩展结点的载重量 ;

r = $\sum_{i=j+1}^{n} w_i$:剩余集装箱的重量;

当 cw+r (限界函数)≤bestw 时, 将cw对应的子树剪去。

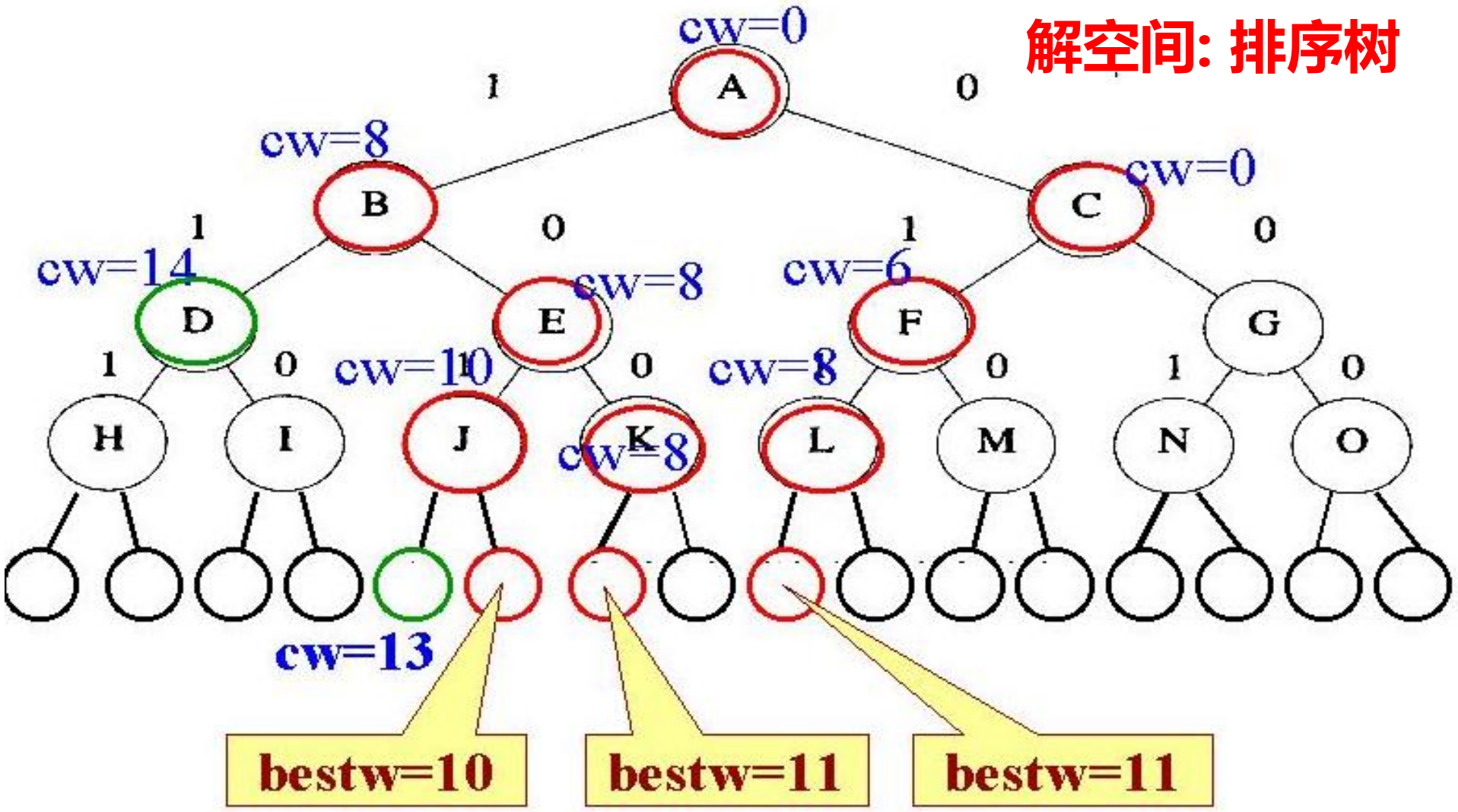
剪枝条件:

$$\sum_{i=1}^{j} w_i x_i + w_{j+1} > c1$$

$$Cw + r \le bestw$$

5.7 *回溯法解装载问题

例如 n=4, c1=12, w=[8, 6, 2, 3]. bestw初值=0;



5.7 *回溯法解装载问题

程序框架

```
void backtrack (int i) // 搜索第i层结点
{
    if (i > n)  { // 到达叶结点
        if (cw>bestw) bestw=cw;        //修正最优值
    }
    if (cw + w[i] <= c) {// 搜索左子树
        cw += w[i];
        backtrack(i + 1);
        cw -= w[i];
    }          //回退
    backtrack(i + 1);      // 搜索右子树
}
```

设cw：是当前载重量
bestw：当前最优载重量

约束函数：

cw + w[i] <= c

5.7 *回溯法解装载问题

剪枝处理—求最优值

- cw：是当前载重量，bestw：当前最优载重量
- r：剩余集装箱的重量

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

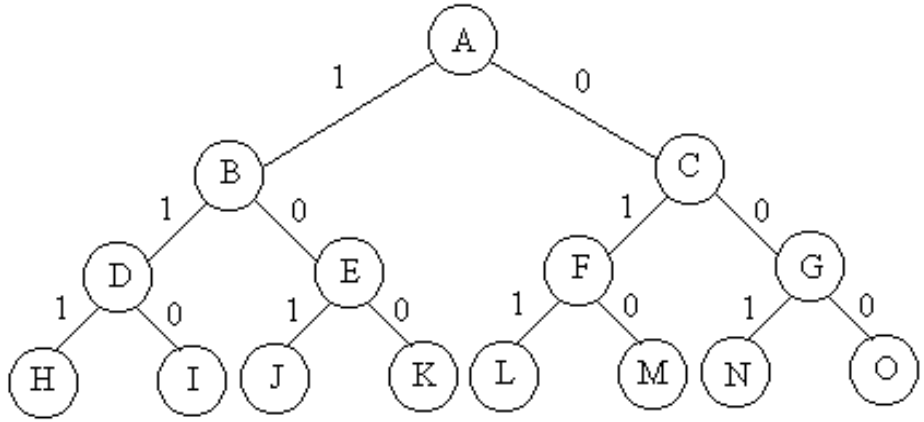
限界函数：cw+r ＞ bestw

5.7 *回溯法解装载问题

求最优解—记录路径

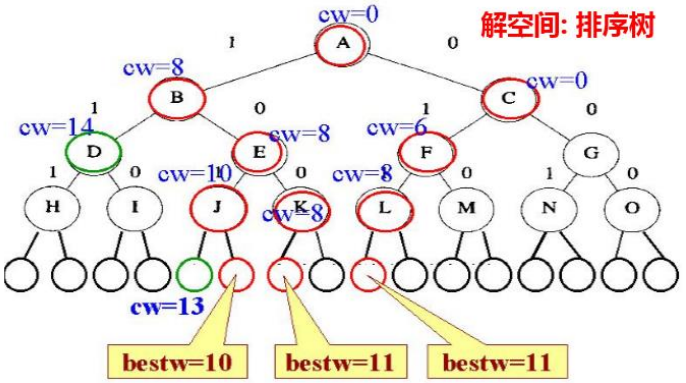
```
void backtrack (int i)  {// 搜索第i层结点
    if (i > n)  // 到达叶结点
    { 更新最优解bestx,bestw;return; }
    r -= w[i];
    if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];      }
    if (cw + r > bestw)  {
      x[i] = 0;  // 搜索右子树
      backtrack(i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

```
if (cw>bestw) {
    for(int j=1; j<=n; j++)
        bestx[j]=x[j];
        bestw=cw;
}
```



5.7 *回溯法解装载问题

装载问题的回溯算法描述




```
template < class Type >
Type Maxloading(type w[], type c,  int n,)
{ loading <Type> X;      //初始化X
   X. w=w;  //集装箱重量数组
   X. c=c;   //第一艘船载重量
   X. n=n;   //集装箱数
   X. bestw=0; //当前最优载重
   X. cw=0;//当前载重量
   X. r=0;   //剩余集装箱重量
   for (int i=1; i<=n; i++)
      X. r +=w[i]
   //计算最优载重量
   X.backtrack(1);
   return X.bestw;
}
```

```
template <clas
void Loading<Type>:: void backtrack (int i)
{// 搜索第i层结点
  if (i > n)  {// 到达叶结点
     if(cw>bestw)bestw=cw;
     return;}
 //搜索子树
  r -= w[i];
  if (cw + w[i] <= c) {// 搜索左子树
     x[i] = 1;//左儿子结点
     cw += w[i];
     backtrack(i + 1);
      cw - = w[i];      }
  if (cw + r > bestw)  {
     x[i] = 0;  // 搜索右子树，右儿子结点
     backtrack(i + 1);     }
  r += w[i];
}
```

剪枝？

算法复杂性： $O(2^n)$