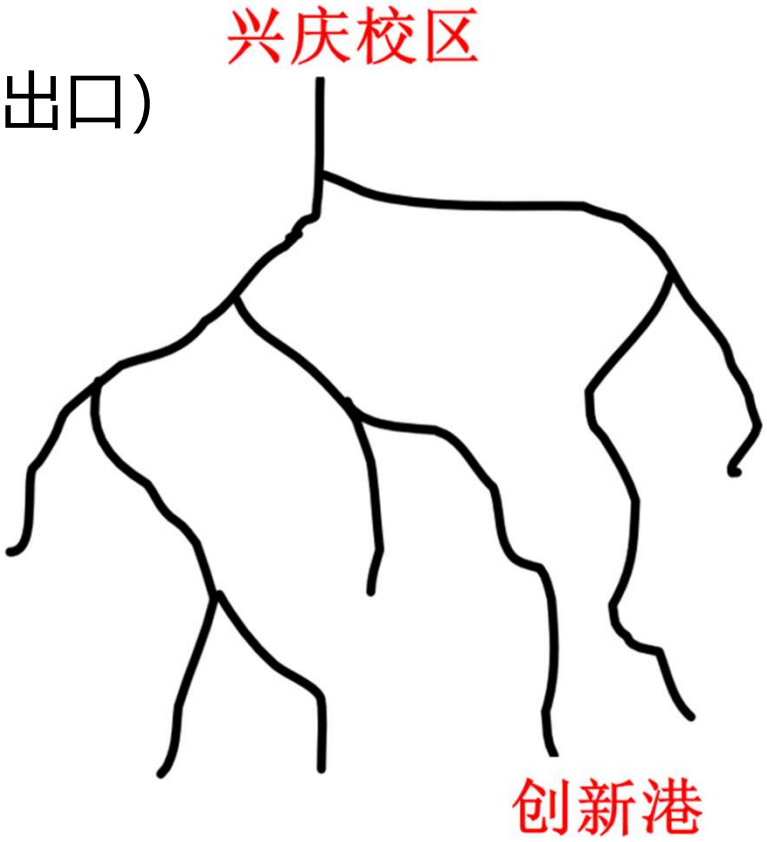
第5章　回溯法

引言

- **实例：**
  - ✓ 兴庆校区（入口）走到创新港（出口）
  - ✓ 若完全不认识路，会怎样走
  - ✓ 类比迷宫



- **两类问题**
  - ✓ 存在性问题（可行解）
  - ✓ 优化问题（最优解）

引言

- 理论上
  - 寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。
- 但是
  - 当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。
  - 若候选解的数量非常大（指数级，大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。

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。

系统性

跳跃性

回溯法的算法框架

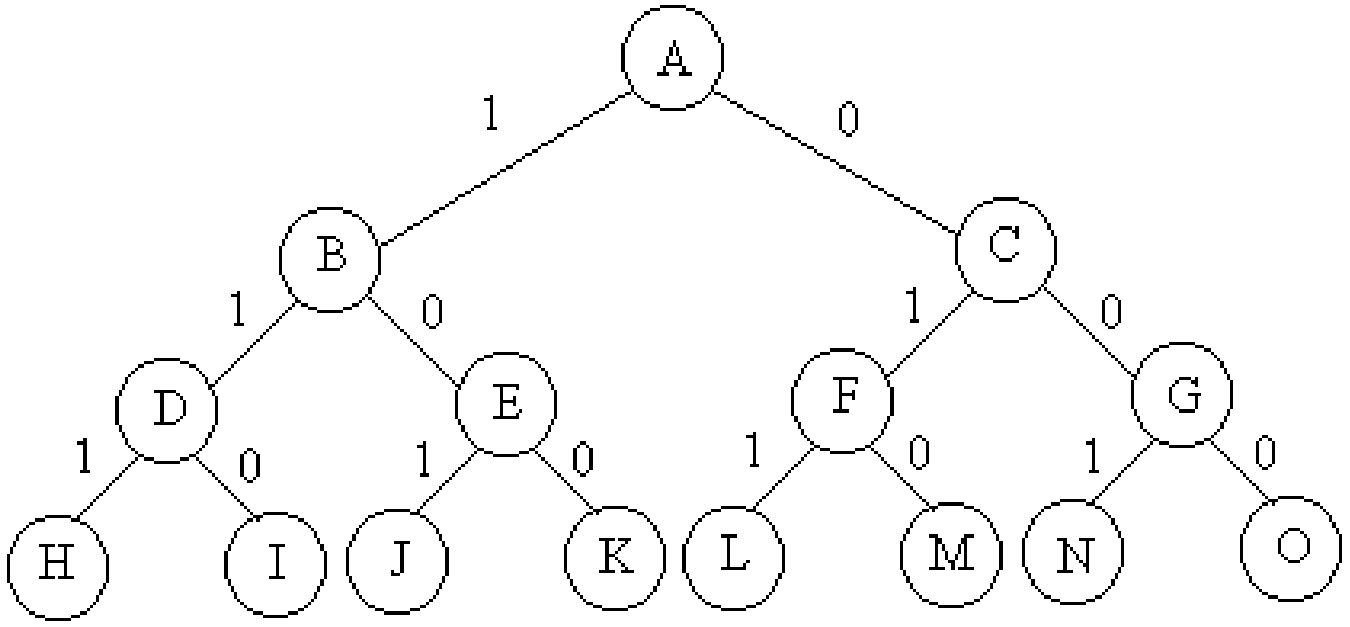
问题的解空间

- 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xi的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

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）。

问题的解空间

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p_i背包的载荷为M, 若将物体$i(1 \le i \le n,)$装入背包,则有价值为p_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
取n=3 , 问题所有可能的解为(解空间):
(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)



n=3时的0-1背包问题用完全二叉树表示的解空间。

生成问题状态的基本方法

问题状态的生成是从开始结点出发，在搜索过程中不断扩展已有的结点来完成。

生成问题状态有两种本质上不同的方法：深度优先生成方法和广度优先生成方法。

- 深度优先的问题状态生成法：对一个结点R，一旦产生了它的一个子结点C，就把C作为当前结点进行扩展。在完成对以C为根的子树的穷尽搜索之后，将R重新变成当前结点，继续生成R的下一个子节点（如果存在）。
- 广度优先的问题状态生成法：在对一个结点扩展产生了它的一个子结点后，继续生成它的下一个子结点，直至所有的子结点都生成。

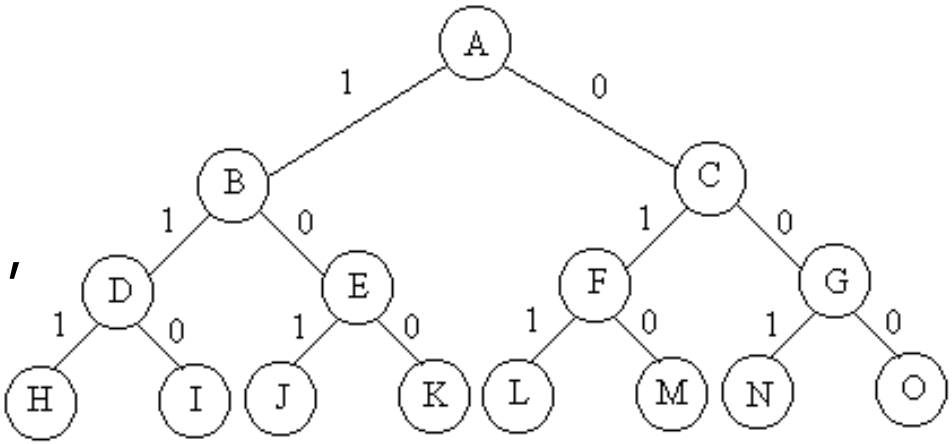
生成问题状态的基本方法

问题状态生成过程中结点的状态：

- 扩展结点：一个正在产生子结点的结点称为扩展结点；
- 活结点：结点已经生成但其子结点还未全部生成的结点称为活结点；
- 死结点：一个所有子结点已经产生的结点称做死结点；

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。

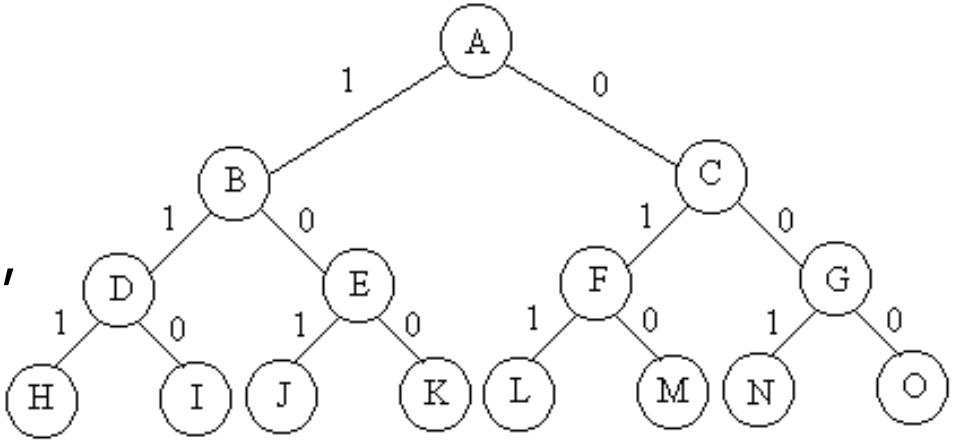
回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 从图中的根节点开始搜索，开始时，根节点A是唯一的活结点，也是当前的扩展结点。在这个扩展结点处，可以沿纵深方向移至B或C；
- 假设选择先移至B。此时A和B是活结点，B成为当前扩展结点。由于选取了w1，故在B处剩余背包容量r=14，价值为45；
- 从B处，可以移至D或E。由于移至D至少需要w2=15的背包容量，而现在剩余背包容量r=14，故移至D导致不可行解，返回B；

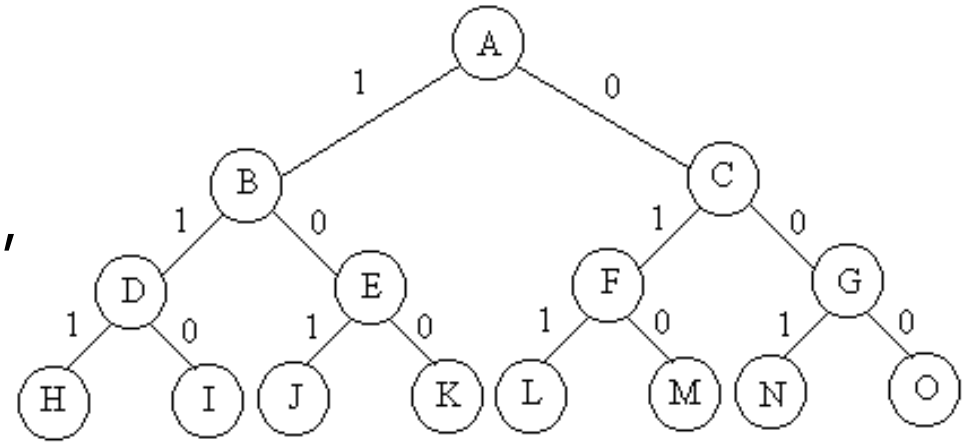
回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 搜索至E处不需要背包容量，因而是可行的。移至E，此时E成为新的扩展结点，A、B和E是活结点，在E处，r=14，价值为45。从E处可以向纵深移至J或K；
- 移向J导致不可行解，于是移向K，K成为新的扩展结点。由于K是叶结点，故得到一个可行解。这个解相应的价值为45。xi的取值由根节点到K的路径唯一确定，即x={1，0，0}。由于在K处已不能再向纵深扩展，所以K成为死结点；

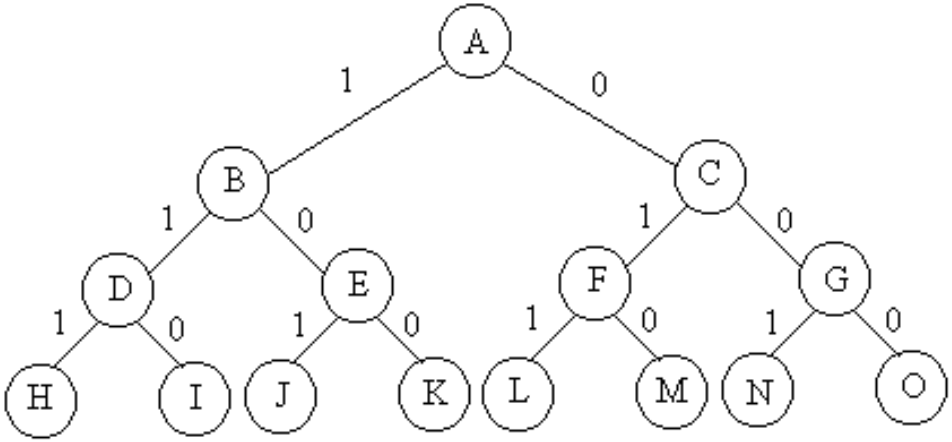
回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 再返回E处，此时E处也没有可扩展的结点，它也成为死结点；
- 接下来返回B，B同样成为死结点；
- 返回A，A再次成为当前扩展节点。A还可继续扩展，到达C。
- 在C处，r=30，价值为0。从C可移向F或G。
- 假设移至F，成为新的扩展结点。结点A、C和F是活结点。此时，r=15，价值为25。
- 从F继续移向L，此时r=0，价值为50。由于L是叶结点，而且是迄今为止价值最高的解，因此记录此可行解。

回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- L不可扩展，又返回到F。
- 按此方式继续搜索遍整个解空间。搜索结束后找到的最优解是相应0-1背包问题的最优解。

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。

回溯法的基本思想

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。

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);
}
```

t：递归的深度

f(n, t),g(n,t): 当前扩展结点的未搜索过的子树的起始编号和终止编号

h(i): 当前扩展结点的第i个可选值

constraint(t): 约束函数，判断x[1:t]是否满足约束条件

bound(t): 限界函数，判断x[1:t]是否有可能得到最优解

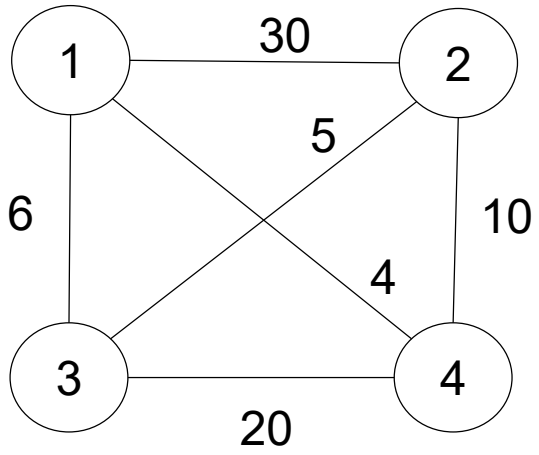
迭代回溯

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) {//还有活结点
        if (f(n,t)<=g(n,t)) //还有未搜索过的子节点
            for (int i=f(n,t);i<=g(n,t);i++)
                x[t]=h(i);
                if (constraint(t)&&bound(t))
                    if (solution(t))  //当前拓展结点处是否已得到问题的可行解
                        output(x);
                    else   t++; //深入搜索子节点
        else t--; //所有子节点搜索完成，返回
}
```

旅行售货员问题

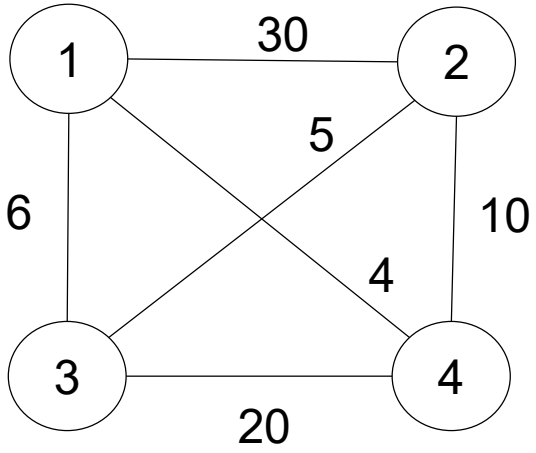
某售货员要到若干城市去推销商品，已知各城市之间的路程。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程最小。

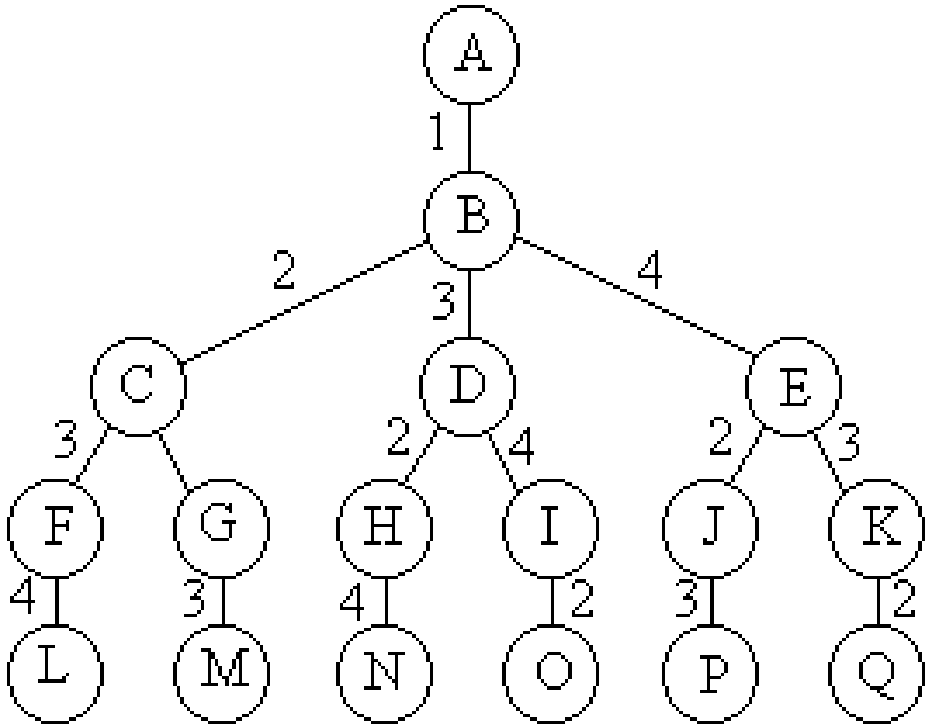


设G=(V,E)是一个带权图。图中的一条周游路线是包括V中的每个顶点在内的一条回路。

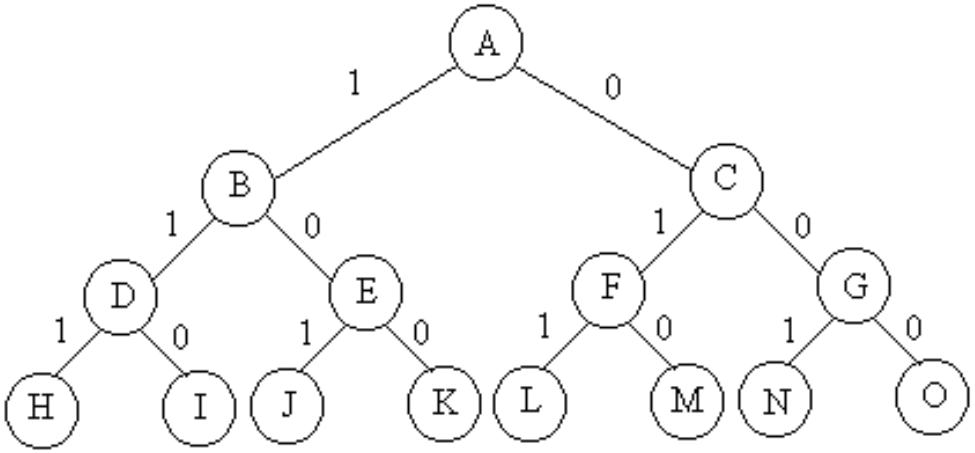
旅行售货员问题

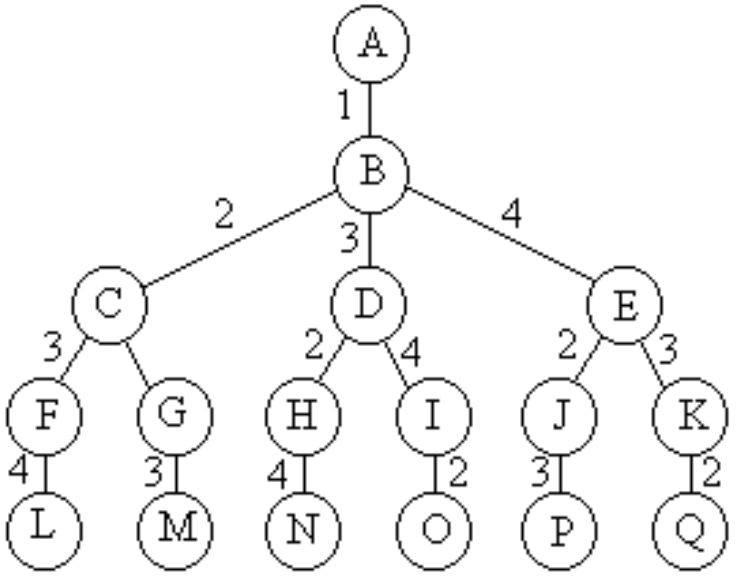
右图是左图例子的解空间树。从根节点A到叶结点L的路径上边的标号组成一条周游路线，如1-2-3-4-1





子集树与排列树

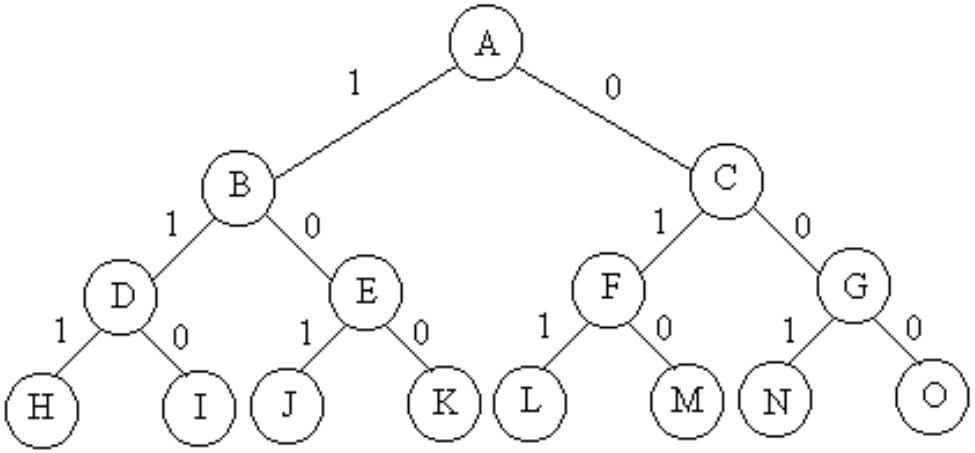


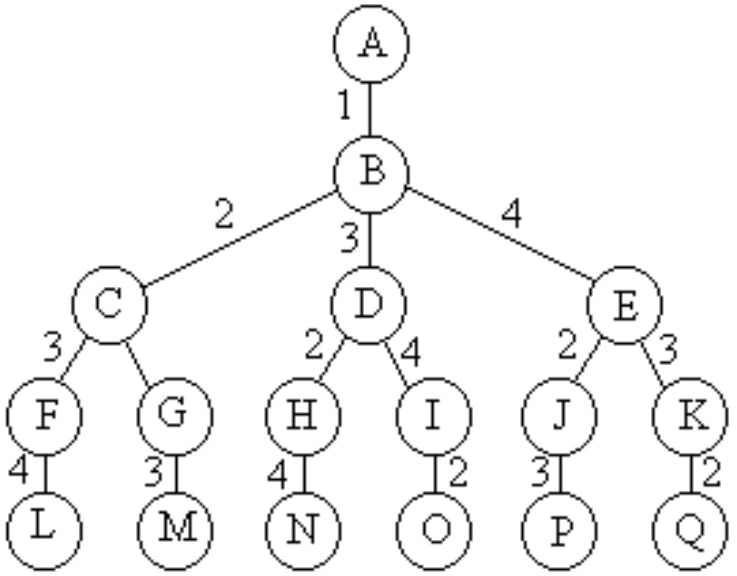


上图中的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树：

- 当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树为子集树（左图）。如0-1背包问题，通常有2^n个叶结点，结点总个数为$2^{n+1}-1$。遍历子集树的任何算法均需$\Omega(2^n)$的计算时间。

子集树与排列树

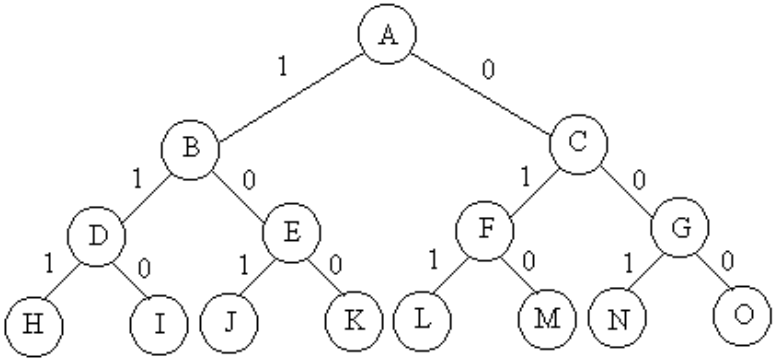


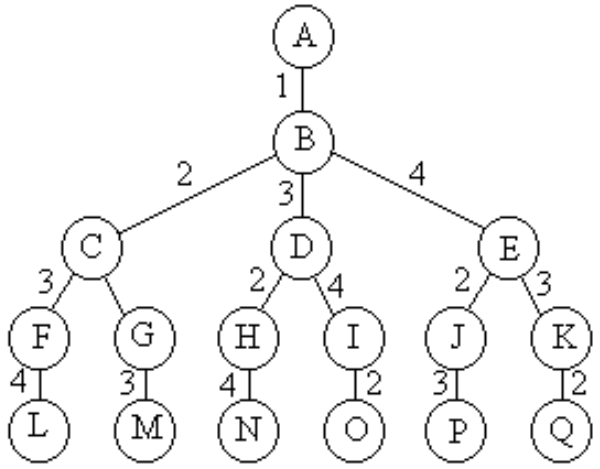


上图中的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树：

- 当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间为排列树（右图）。排列树通常有n！个结点。因此遍历排列树需要Ω(n!)的计算时间。旅行售货员问题的解空间树就是一棵排列树。

子集树与排列树





用回溯法搜索子集树的一般算法：

```
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);
}
```

用回溯法搜索排列树的一般算法：

```
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c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

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

- 例1：n=3，c1=c2=50，w=[10,40,40]，可将集装箱1和集装箱2装上第一艘轮船，将集装箱3装上第二艘轮船。
- 例2： n=3，c1=c2=50，w=[20,40,40]，则无法将这3个集装箱装上轮船。

装载问题

- 当$\sum_{i=1}^{n} w_i = C_1 + C_2$时，装载问题等价于子集和问题：

  给定整数集合S和一个整数t，判定是否存在S的一个子集S’⊆S，使得S’中整数的和为t。

- 当C1=C2且$\sum_{i=1}^{n} w_i = 2C_1$时，装载问题等价于划分问题：给定一个正整数的集合S，是否可以将其分割成两个子集合，使两个子集合的数加起来的和相等。

装载问题

有一批共n个集装箱要装上2艘载重量分别为c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

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

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。

- 首先将第一艘轮船尽可能装满；
- 将剩余的集装箱装上第二艘轮船。

（因为有解，所以能装下，第一艘装越多，第二艘剩余空间越大。）

装载问题

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C1。由此可知，装载问题等价于以下特殊的0-1背包问题。

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

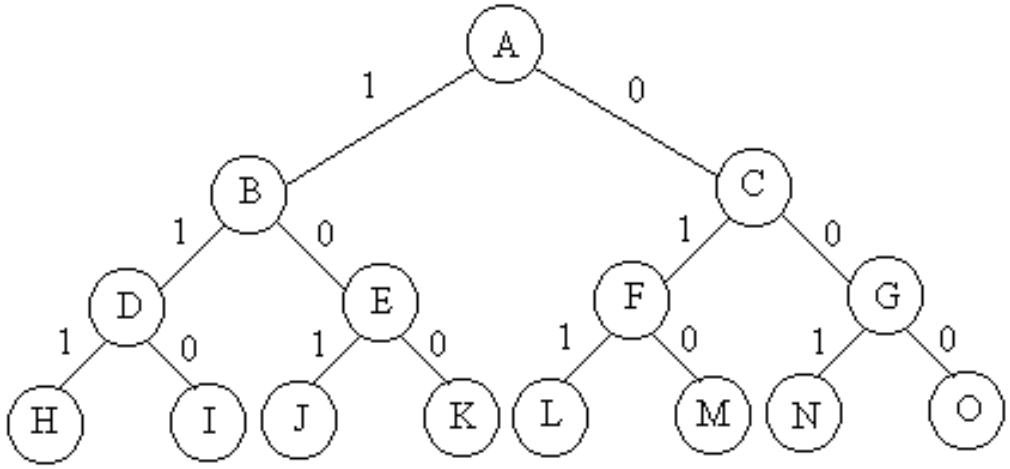
$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

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

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 -- 剪枝



装载问题

当前载重量 cw = $\sum_{j=1}^{i} w_j x_j$

当前最优载重量 bestw

```
private static 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;
        if (cw + w[i] <= c)  // 搜索左子树
            cw += w[i];
            backtrack(i + 1);
            cw - = w[i];
        backtrack(i + 1); // 搜索右子树
}
```

backtrack(1)

装载问题

引入上界函数，剪去不含最优解的子树。

剩余集装箱的重量 $r = \sum_{j=i+1}^{n} w_j$

```
private static void backtrack (int i){// 搜索第i层结点
     if (i > n)  // 到达叶结点
         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];
   }
```

装载问题

记录最优解

```
private static void backtrack (int i){// 搜索第i层结点
      if (i > n)  // 到达叶结点
         for(int j=1;j<=n;j++)
            bestx[j] = x[j];
            bestw = cw;
            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)  // 搜索右子树
         x[i] = 0;
         backtrack(i + 1);
      r += w[i];
}
```

x：记录从根至当前结点的路径

bestx：记录当前最优解