贪心算法

陈长建

计算机科学系

第三次上机实验

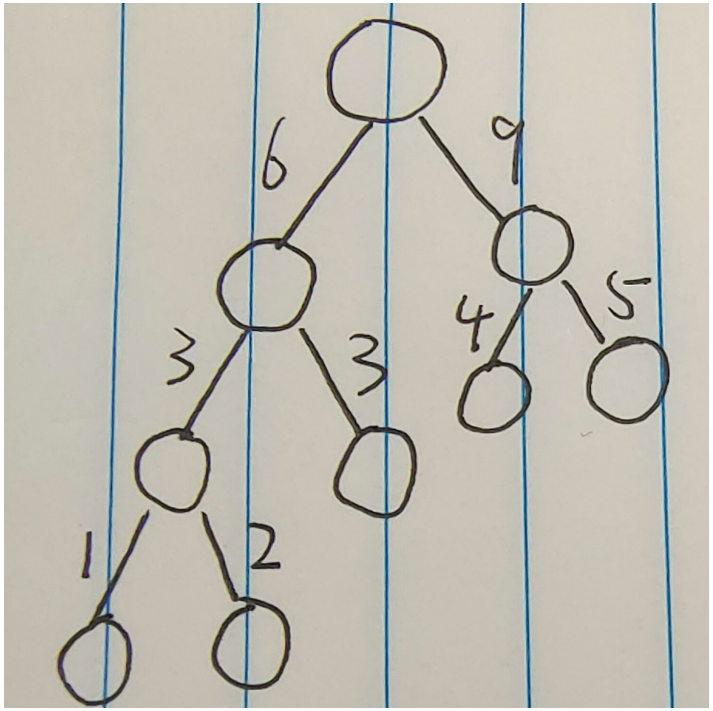
- **院楼103，12月1日**上午8:30-12:00（现场验收）
- 第二次实验离线题（离线准备）
  - 1. 用Dijkstra贪心算法求解单源最短路径问题
  - 2. 收集样本问题（实现题3-15）
  - 3. 字符串比较问题（实现题3-17）
- 在线题
  - acm.hnu.edu.cn

课前问题

- 5个权值{3, 2, 4, 5, 1}构造的哈夫曼树

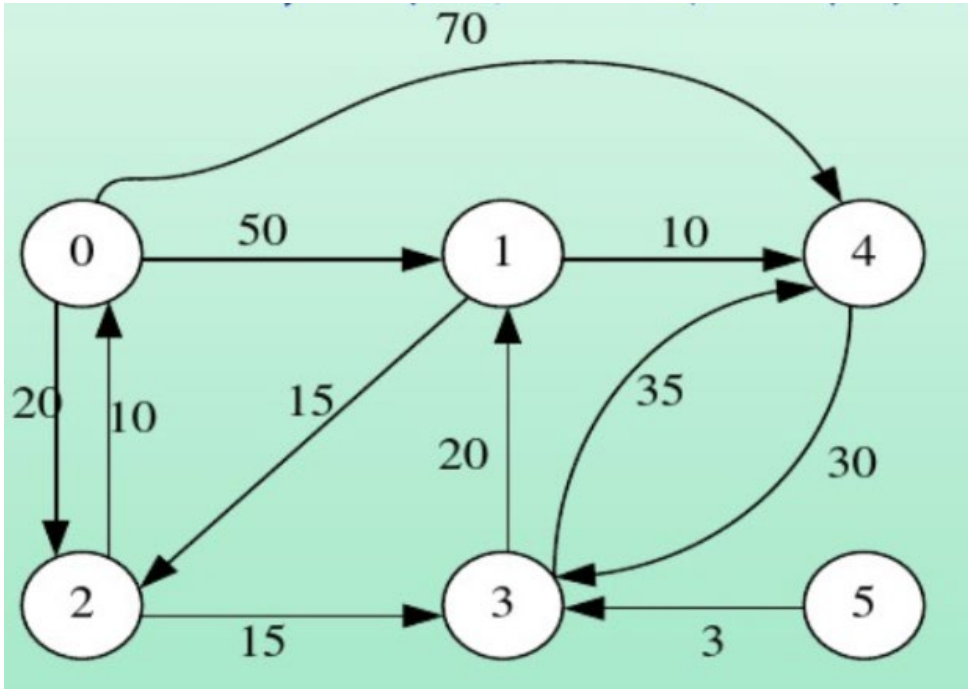
课前问题

- 5个权值{3, 2, 4, 5, 1}构造的哈夫曼树

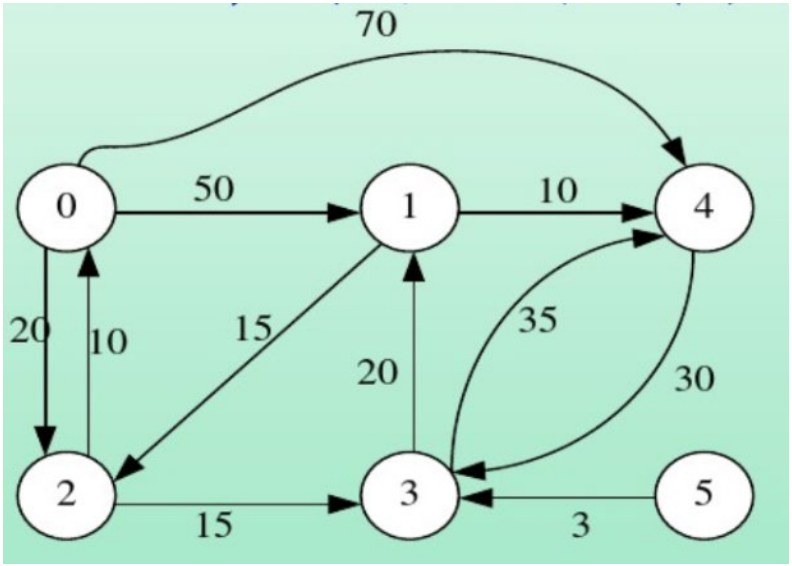


课前问题

- 源点：V0



课前问题



源点	终点	最短路径	路径长度
V0	V1	（V0，V2，V3，V1）	45
	V2	（V0，V2）	10
	V3	（V0，V2，V3）	25
	V4	（V0，V2，V3，V1，V4）	55
	V5	—	∞

例6 - 最小生成树

- 设G=(V,E)是无向带权连通图，即一个网络。
- E中每条边(v,w)的权为c[v][w]。如果G的子图G’是一棵包含G的所有顶点的树，则称G’为G的生成树。
- 生成树上各边权的总和称为该生成树的耗费。在G的所有生成树中，耗费最小的生成树称为G的最小生成树。

应用

- 网络的最小生成树在实际中有广泛应用。
- 例如，在设计通信网络时，用图的顶点表示城市，用边(v,w)的权c[v][w]表示建立城市v和城市w之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。

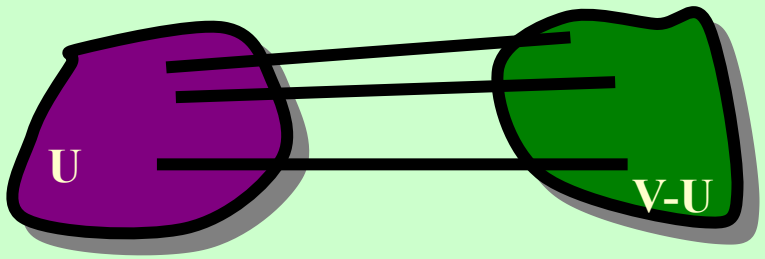
贪心法求解准则

- 将贪心策略用于求解无向连通图的最小代价生成树时，核心问题是需要确定贪心准则。
- 根据最优量度标准，算法的每一步从图中选择一条符合准则的边，共选择n-1条边，构成无向连通图的一棵生成树。
- 贪心法求解的关键：该量度标准必须足够好。它应当保证依据此准则选出n-1条边构成原图的一棵生成树，必定是最小代价生成树。

算法步骤分析

- 设G=(V,E)是带权的连通图，T=(V,S)是图G的最小代价生成树。

```
ESetType SpanningTree(ESetType E,int n)
{ //G=(V,E)为无向图，E是图G的边集，n是图中结点数
   ESetType TE=Ø;    //TE为生成树上边的集合
   int u,v,k=0; EType e;      //e=(u,v)为一条边
   while(k<n-1 && E中尚有未检查的边)
   {   //选择生成树的n-1条边
       e=select(E);  //按最优量度标准选择一条边
       if(TE∪e 不包含回路)  //判定可行性
       {    TE=TE∪e; k++;   }//在生成树边集TE中添加一条边
    }
    return S;
}
```



普里姆（Prim）算法
克鲁斯卡尔（Kruskal）算法

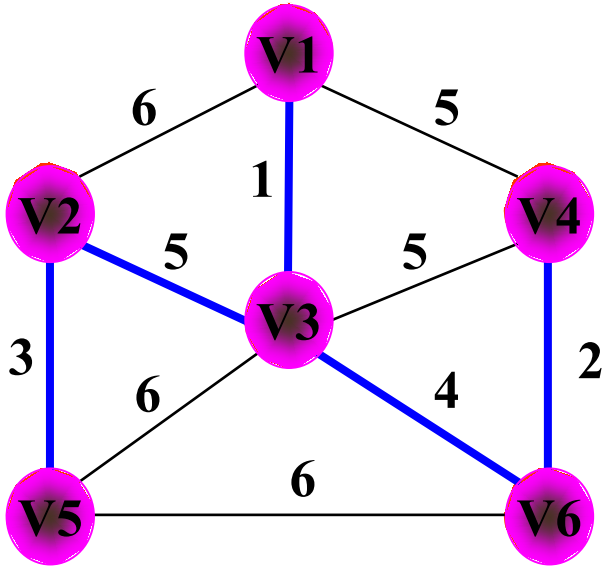
- Kruskal算法的贪心准则：按边代价的非减次序考察E中的边，从中选择一条代价最小的边e=(u,v)。
  - 这种做法使得算法在构造生成树的过程中，当前子图不一定是连通的。
- Prim算法的贪心准则：在保证S所代表的子图是一棵树的前提下选择一条最小代价的边e=(u,v)。

Prim算法的基本步骤

1. 在图G=(V, E) （V表示顶点集合，E表示边集合）中，从集合V中任取一个顶点（例如取顶点v_1）放入集合U中，这时U=$\{v_1\}$，生成树边集合T(E)为空。
2. 寻找与S中顶点相邻（另一顶点在V-U中）权值最小的边的另一顶点v_2，并使v_2加入S。即U=$\{v_1,v_2\}$，同时将该边加入集合T(E)中。
3. 重复2，直到U=V为止。

- 这时T(E)中有n-1条边，T=(U,T(E))就是一棵最小生成树。

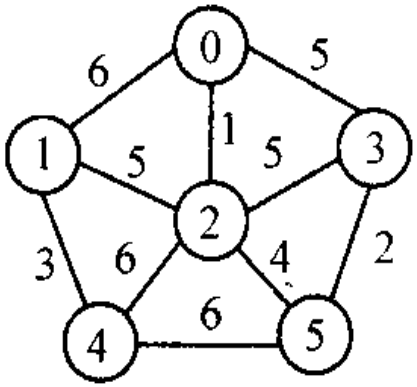
prim算法



算法思想：

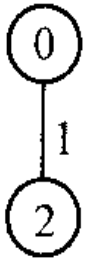
- 设 $N=(V, E)$ 是连通网，TE 是 N 上最小生成树中边的集合。
- 初始令 $U=\{u_0\}, (u_0 \in V)$, $TE=\{\}$。
- 在所有 $u \in U, v \in V\text{-}U$ 的边 $(u, v) \in E$ 中，找一条代价最小的边 (u_0, v_0)。
- 将 (u_0, v_0) 并入集合 TE，同时 v_0 并入 U。
- 重复上述操作直至 $U=V$ 为止，则 $T=(V, TE)$ 为 N 的最小生成树。

Prim算法举例

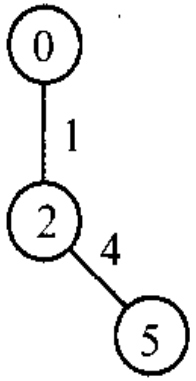


（a）无向图G

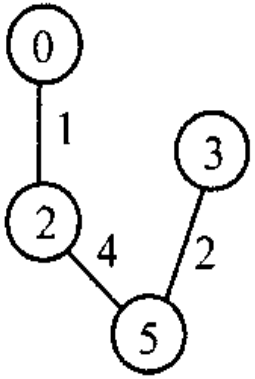
（b）只有源点



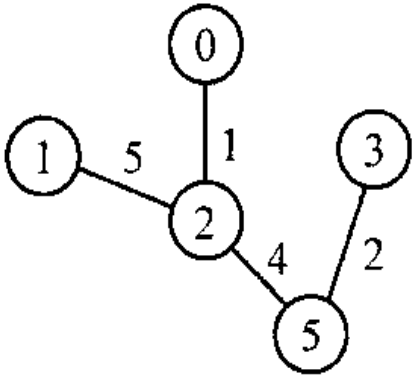
（c）加入第1条边



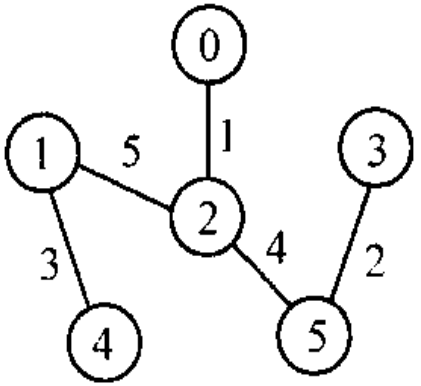
（d）加入第2条边



（e）加入第3条边



（f）加入第4条边

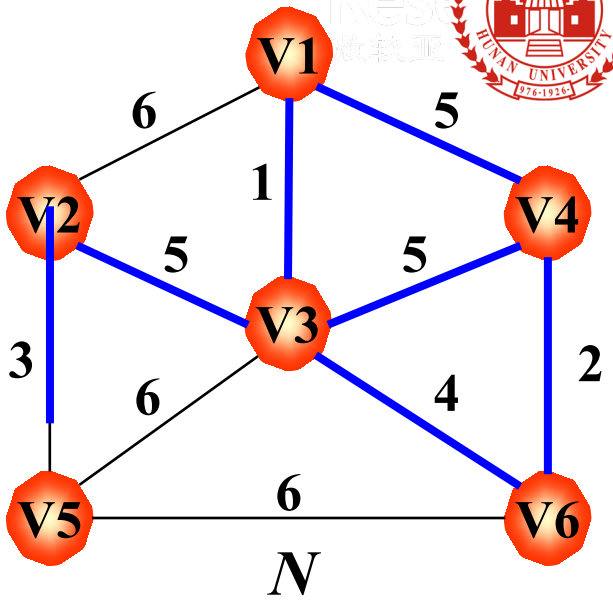


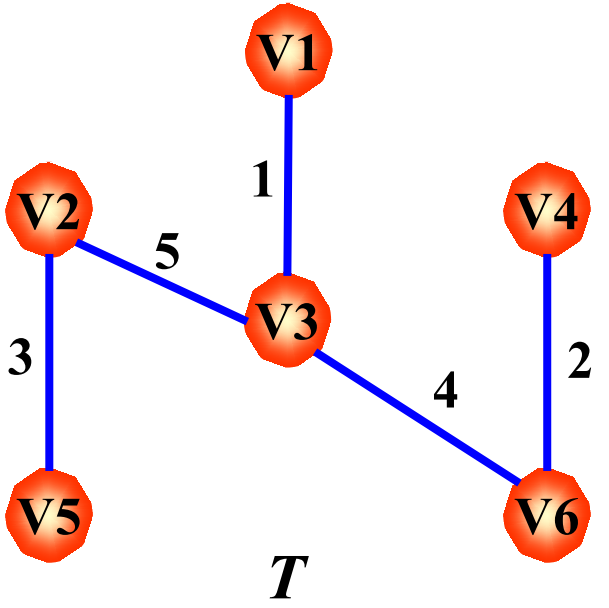
（g）图G的最小代价生成树

Kruskal算法

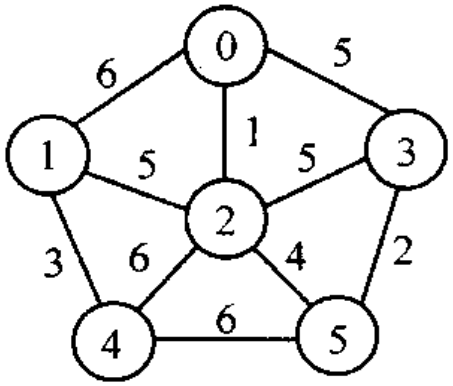
算法思想：

- 设连通网 $N=(V,E)$，令最小生成树初始状态为只有 n 个顶点而无边的非连通图 $T=(V,\{\})$，每个顶点自成一个连通分量。
- 在 E 中选取代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上（即：不能形成环），则将此边加入到 T 中；否则，舍去此边，选取下一条代价最小的边。
- 依此类推，直至 T 中所有顶点都在同一连通分量上为止。

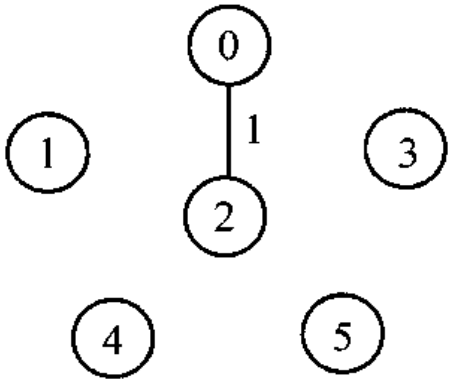




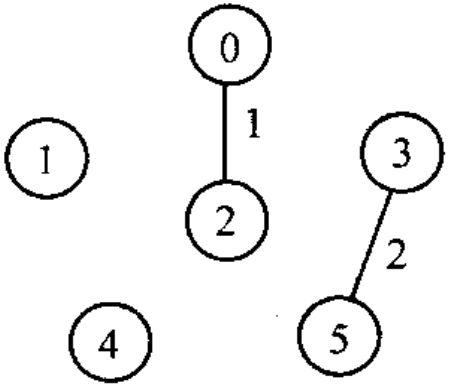
Kruskal算法举例



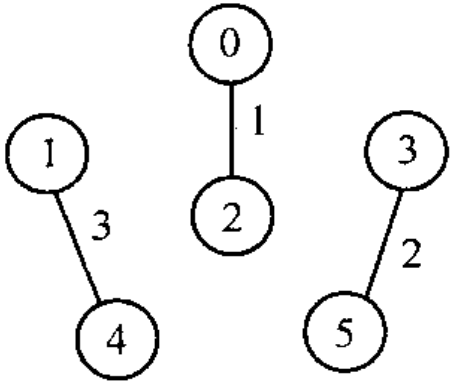
（a）无向图G



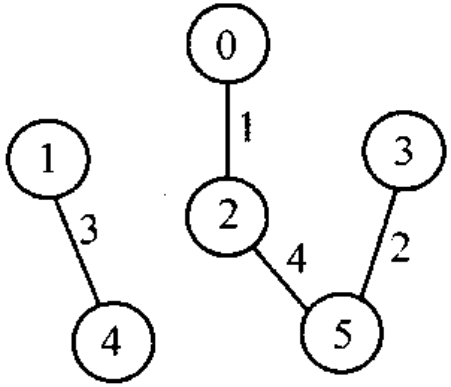
（b）加入第1条边



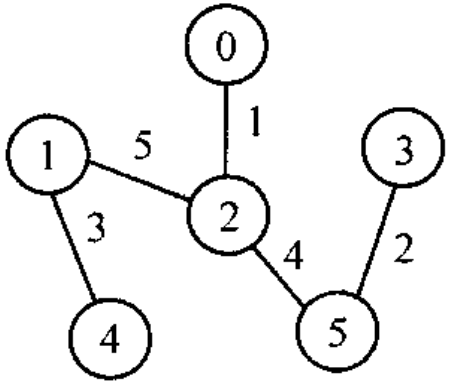
（c）加入第2条边



（d）加入第3条边



（e）加入第4条边



（f）图G的最小代价生成树

时间复杂度

- 如何定义两种算法的时间复杂度?
- Kruskal算法的时间复杂度为O(e*log*e)
- Prim算法的时间复杂度为$O(n^2)$
- 分别适合怎样的应用场合?

算法正确性

- 设图G=(V,E)是一个带权连通图，U是V的一个真子集。若边(u,v) ∈E是所有u∈U，v∈V-U的边中权值最小者，那么一定存在G的一棵最小代价生成树T=(V,TE)，(u,v) ∈TE。
- 这一性质称为MST（minimum spanning tree）性质。

证明：可以用反证法证明。

如果图G的任何一棵最小代价生成树都不包括(u,v)。将(u,v)加到图G的一棵最小代价生成树T中，将形成一条包含边(u,v)的回路，并且在此回路上必定存在另一条不同的边(u',v,)，使得u'∈U，v' ∈V-U。删除边(u',v,)，便可消除回路，并同时得到另一棵生成树T'。

算法正确性

- 设图G=(V,E)是一个带权连通图，U是V的一个真子集。若边(u,v) ∈E是所有u∈U，v∈V-U的边中权值最小者，那么一定存在G的一棵最小代价生成树T=(V,S)，(u,v) ∈S。
- 这一性质称为MST（minimum spanning tree）性质。

因为(u,v)的权值不高于(u',v)，则T'的代价亦不高于T，且T'包含(u,v)，故与假设矛盾。

这一结论是Prim算法和Kruskal算法的理论基础。

无论Prim算法和还是Kruskal算法，每一步选择的边均符合MST，因此必定存在一棵最小代价生成树包含每一步上已经形成的生成树（或者森林），并包含新添加的边。

作业

- 算法分析题5-3 回溯法重写0-1背包
- 算法分析题5-5 旅行商问题（剪枝）
- 算法实现题5-2 最小长度电路板排列问题
- 算法实现题5-7 n色方柱问题

- 提交：12月8日

回溯法

陈长建

计算机科学系

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）递归回溯
- （2）迭代回溯
- （3）子集树算法框架
- （4）排列树算法框架

回溯法

- 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）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。

回溯法的直观印象

- （1）回溯法有“通用的解题法”之称。
- 用它可以系统地搜索一个问题的所有解或最优解。回溯法是一个既带系统性又带有跳跃性的搜索算法。它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发，搜索解空间树。

回溯法的直观印象

- （2）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。

回溯法的直观印象

- （3）回溯法在用来求问题的所有解（或最优解）时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。而在求任一解时，只要搜索到一个解就结束。这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适合于解一些组合数较大的问题。

回溯法：任务描述

- 问题描述：假定问题的解能表示成一个n元组$(x_1,\ldots,x_n)$,其中x_i取自某个有穷集s_i
- 解空间：所有这些n元组的集合构成问题的解空间。假设集合s_i的大小是m_i，可能的元组数为$m=m_1m_2\ldots m_n$

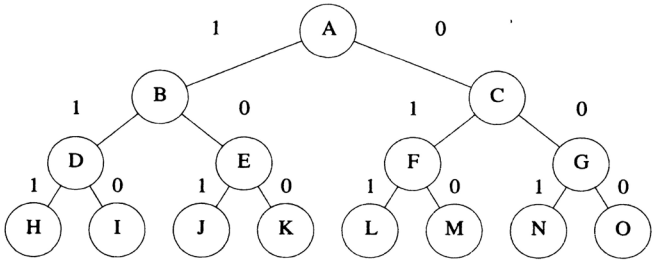
例1 0-1背包问题

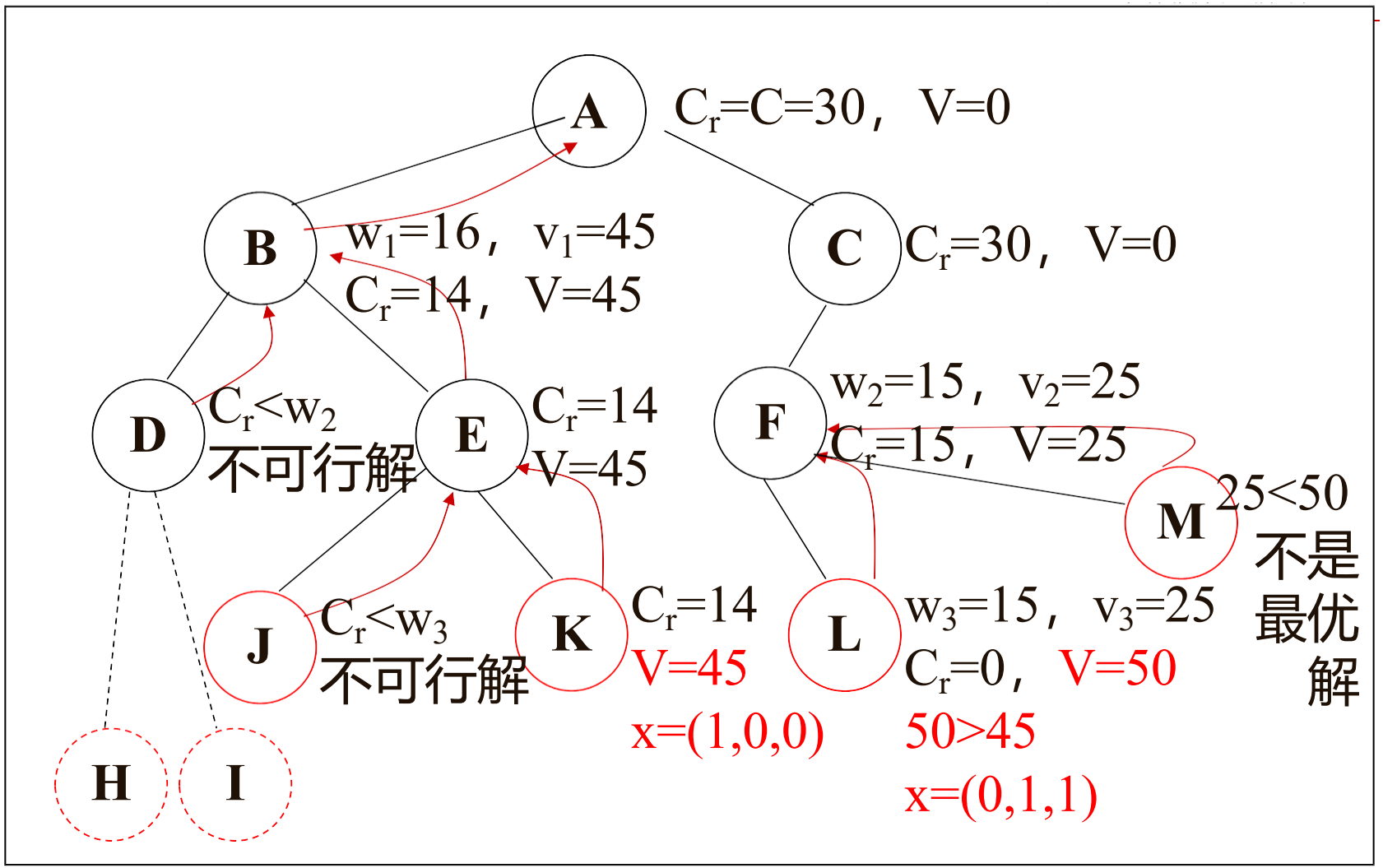
- n=3, C=30, w={16, 15, 15},
- v={45, 25, 25}

- 假如要把它所有可能的解空间遍历出来，要如何表示?

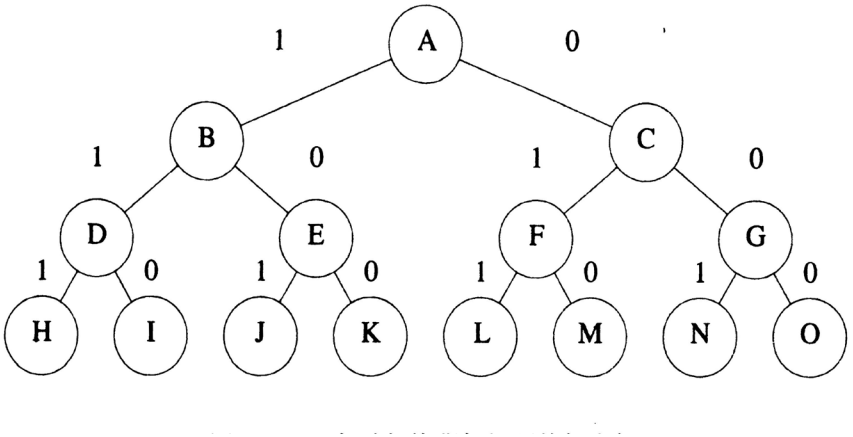
W=[16,15,15], V=[45,25,25],C=30.

例1 0-1背包问题



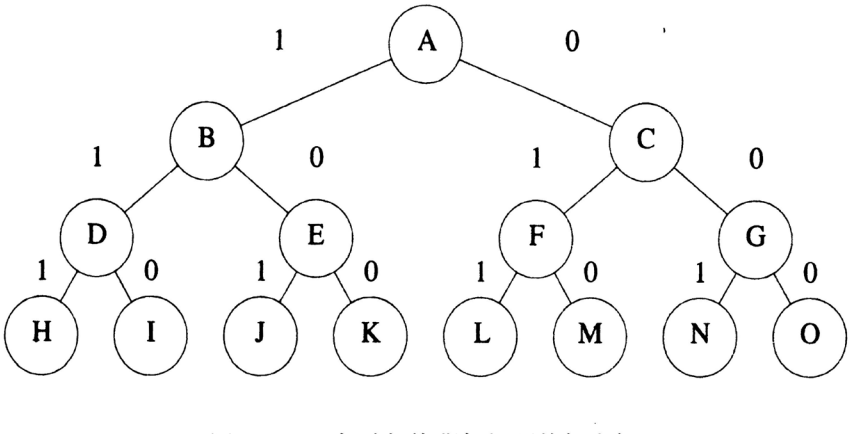


例1 0-1背包问题



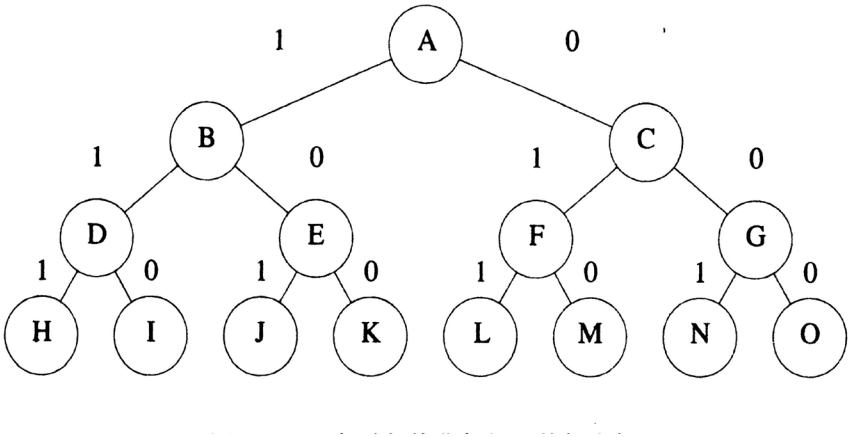
- n=3, C=30, w={16, 15, 15},
- v={45, 25, 25}
- 开始时，C_r=C=30，V=0，A为唯一活结点，也是当前扩展结点
- 扩展A，先到达B结点
  - $C_r=C_r-w_1$=14，$V=V+v_1$=45
  - 此时A、B为活结点，B成为当前扩展结点
  - 扩展B，先到达D
    - $C_r<w_2$，D导致一个不可行解，回溯到B
  - 再扩展B到达E
    - E可行，此时A、B、E是活结点，E成为新的扩展结点
    - 扩展E，先到达J
      - $C_r<w_3$，J导致一个不可行解，回溯到E
    - 再次扩展E到达K
      - 由于K是叶结点，即得到一个可行解x=(1,0,0)，V=45
      - K不可扩展，成为死结点，返回到E
    - E没有可扩展结点，成为死结点，返回到B
  - B没有可扩展结点，成为死结点，返回到A

例1 0-1背包问题



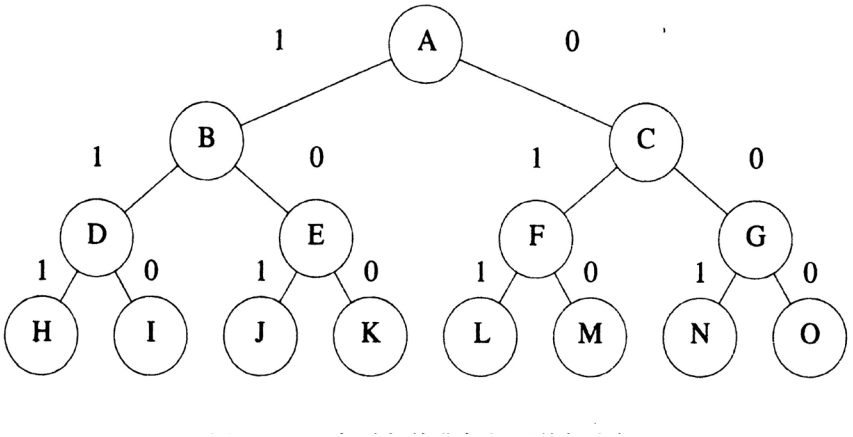
- n=3, C=30, w={16, 15, 15},
- v={45, 25, 25}
- A再次成为扩展结点，扩展A到达C
  - C_r=30，V=0，活结点为A、C，C为当前扩展结点
  - 扩展C，先到达F
    - $C_r=C_r-w_2$=15，$V=V+v_2$=25，此时活结点为A、C、F，F成为当前扩展结点
    - 扩展F，先到达L
      - $C_r=C_r-w_3$=0，$V=V+v_3$=50
      - L是叶结点，且50>45，得到一个可行解x=(0,1,1)，V=50
      - L不可扩展，成为死结点，返回到F
    - 再扩展F到达M
      - M是叶结点，且25<50，不是最优解
      - M不可扩展，成为死结点，返回到F
    - F没有可扩展结点，成为死结点，返回到C

例1 0-1背包问题



- n=3, C=30, w={16, 15, 15},
- v={45, 25, 25}
- A再次成为扩展结点，扩展A到达C
  - C_r=30，V=0，活结点为A、C，C为当前扩展结点
  - 扩展C，先到达F
    - $C_r=C_r-w_2$=15，$V=V+v_2$=25，此时活结点为A、C、F，F成为当前扩展结点
    - 扩展F，先到达L
      - $C_r=C_r-w_3$=0，$V=V+v_3$=50
      - L是叶结点，且50>45，得到一个可行解x=(0,1,1)，V=50
      - L不可扩展，成为死结点，返回到F
    - 再扩展F到达M
      - M是叶结点，且25<50，不是最优解
      - M不可扩展，成为死结点，返回到F
    - F没有可扩展结点，成为死结点，返回到C

例1 0-1背包问题



- n=3, C=30, w={16, 15, 15},
- v={45, 25, 25}
  - 再扩展C到达G
    - C_r=30，V=0，活结点为A、C、G，G为当前扩展结点
    - 扩展G，先到达N，N是叶结点，且25<50，不是最优解，又N不可扩展，返回到G
    - 再扩展G到达O，O是叶结点，且0<50，不是最优解，又O不可扩展，返回到G
    - G没有可扩展结点，成为死结点，返回到C
  - C没有可扩展结点，成为死结点，返回到A
- A没有可扩展结点，成为死结点，算法结束，最优解X=(0,1,1)，最优值V=50

回溯法：任务描述

- 并非解空间中的所有元素都是问题的解
  - 存在性问题：求满足某些条件（约束条件）的一个或全部元组，如果不存在这样的元组算法应返回No。满足约束条件的元组称为问题的可行解（*Feasible Solution*）
  - 优化问题：给定一组约束条件，在满足约束条件的元组中求使某目标函数达到最大（小）值的元组。满足目标函数的元组称为问题的最优解（*Optimal Solution*）

例1’ 0-1背包问题变体

- C=30, w={16, 15, 15}
- 是否存在一种选择，恰好能装满背包?

剪枝函数和回溯法

- 为了提高搜索效率，在搜索过程中使用**约束函数**（constraint function），可以避免无谓地搜索那些不满足约束条件的子树。
- 如果是最优化问题，还可使用**限界函数**（bound function）剪去那些不可能包含最优答案结点的子树。
- 约束函数和限界函数的目的是相同的，都为了剪去不必要搜索的子树，减少问题求解所需实际生成的状态结点数，它们统称为**剪枝函数**（pruning function）。

回溯法与分支限界法的概念

- 使用剪枝函数的深度优先生成状态空间树中结点的求解方法称为**回溯法**（backtracking）；

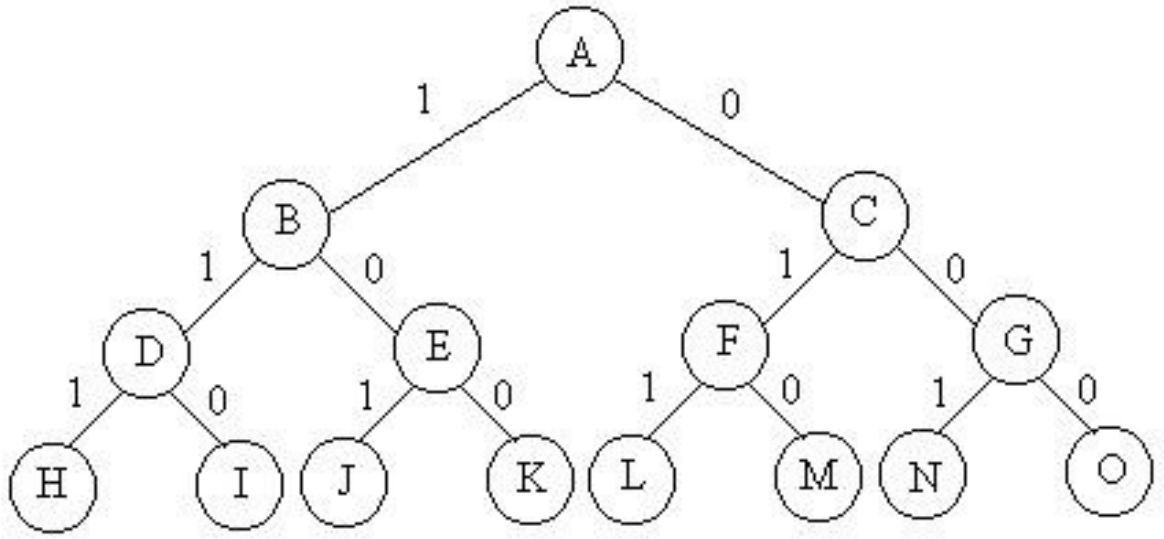
- 广度优先生成结点，并使用剪枝函数的方法称为**分支限界法**（branch-and-bound）。

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$(x_1,x_2,\ldots,x_n)$的形式。
- 显约束：对分量x_i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。

问题的解空间

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）。



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

- 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

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：在一个扩展结点变成死结点之前，它一直是扩展结点

生成问题状态的基本方法

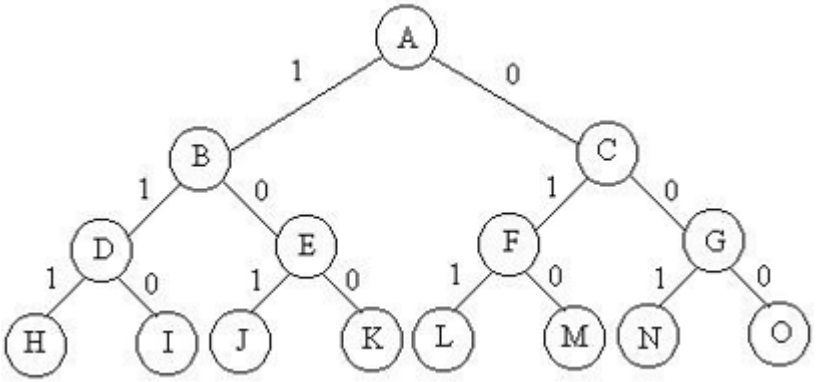
- 回溯法：为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数(bounding function)来处死那些实际上不可能产生所需解的活结点，以减少问题的计算量。

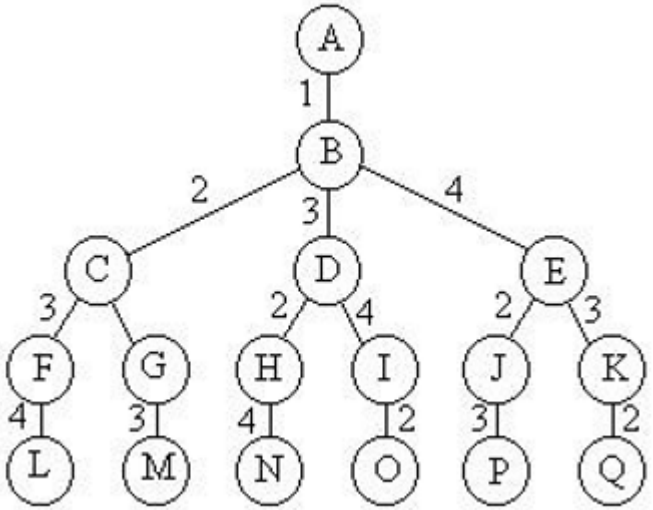
回溯法的解空间树构造

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.排列树: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.

子集树与排列树





回溯法的搜索过程

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.

回溯法的复杂性

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。

回溯法的解题步骤

(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。

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。

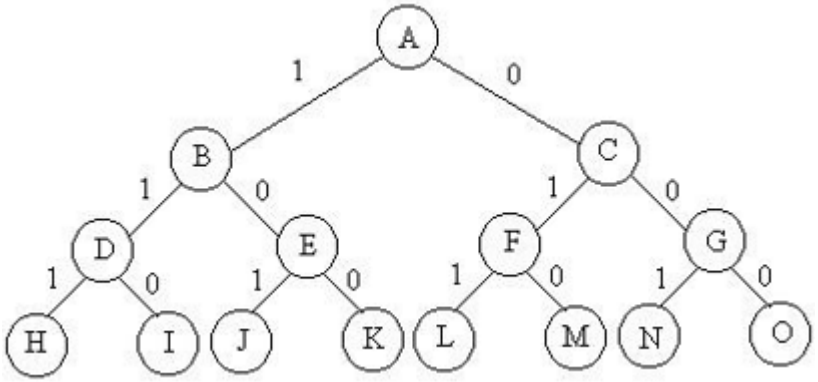
```
void backtrack (int t)   //当前扩展结点在解空间树中的深度
{
    if (t>n) output(x);     //已搜索至叶节点，得到可行解x
    else  //当前扩展结点处未搜索过的子树的起始编号和终止编号
    for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);  //当前扩展结点处x[t]的第i个可选值
        if (constraint(t)&&bound(t)) backtrack(t+1);
    }
}
```

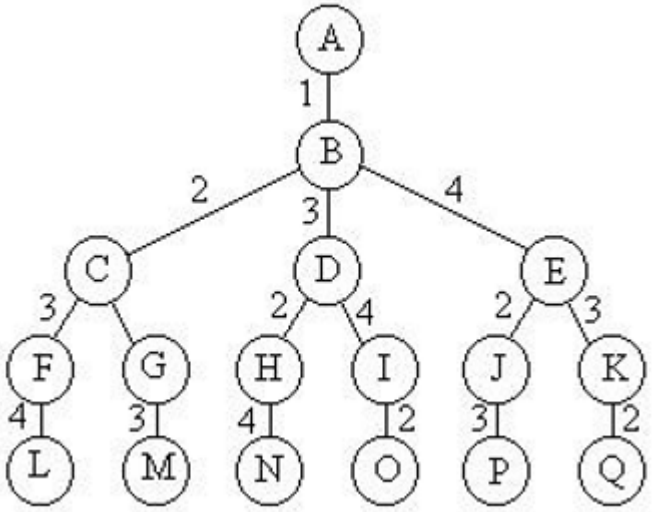
迭代回溯

- 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--; }  //回溯
}
```

子集树与排列树





遍历子集树需$O(2^n)$计算时间

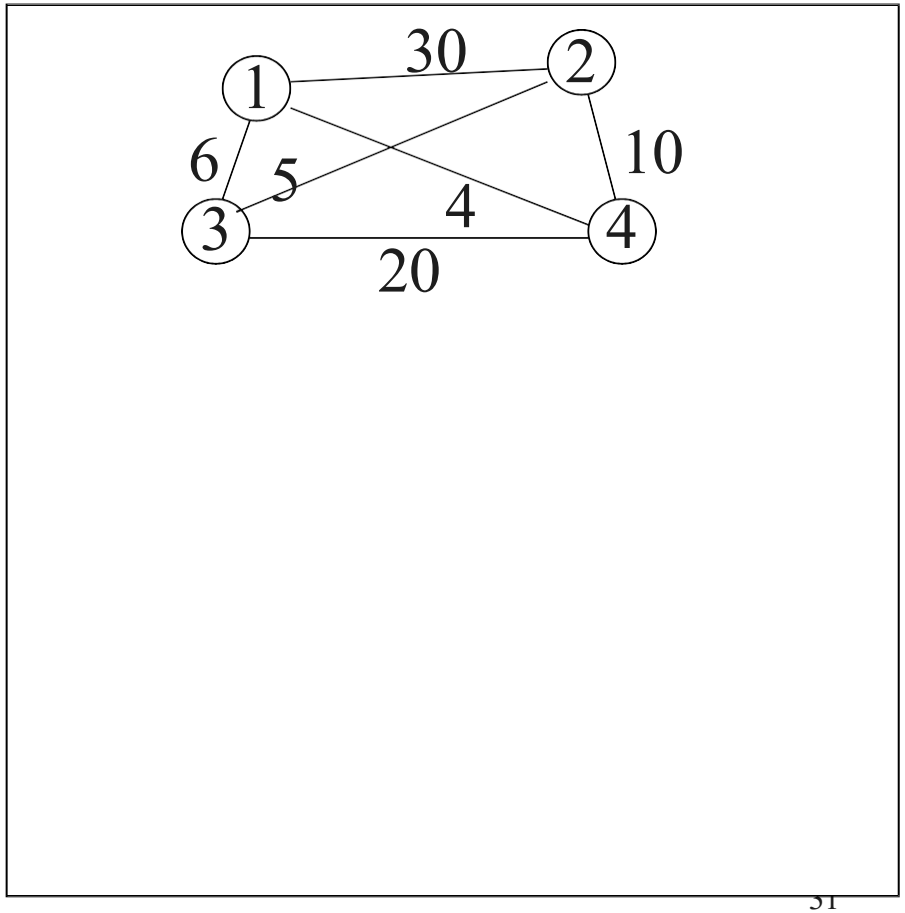
```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (legal(t)) backtrack(t+1); }
}
```

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]); }
}
```

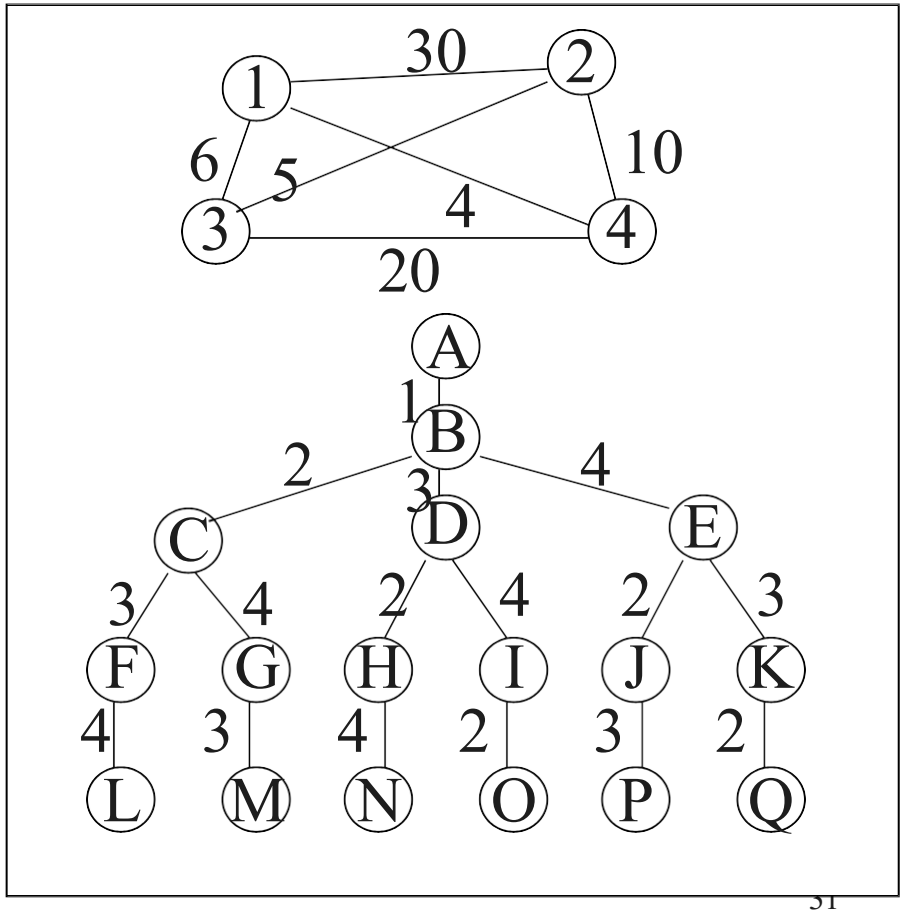
例2 旅行售货员问题

- 问题描述：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**1**出发，经过每个城市一遍，最后回到住地的路线，使总的路程最短。
- 该问题是一个NP完全问题， 有(n-1)!条可选路线。
- 最优解(1,3,2,4,1)，最优值25



例2 旅行售货员问题

- 问题描述：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**1**出发，经过每个城市一遍，最后回到住地的路线，使总的路程最短。
- 该问题是一个NP完全问题， 有(n-1)!条可选路线。
- 最优解(1,3,2,4,1)，最优值25



例3 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 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

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

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第一艘轮船的载重量。

装载问题

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。

例子

- 例如 n=4, c1=12, w=[8, 6, 2, 3].

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)： cw+r (当前载重量+剩余集装箱的重量)
- 在以当前扩展结点为根的子树中任一叶结点所相应的载重量均不超过cw+r

装载问题回溯算法思路

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 ;

$\sum_{i=j+1}^{n} w_j$: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。

装载问题回溯算法思路

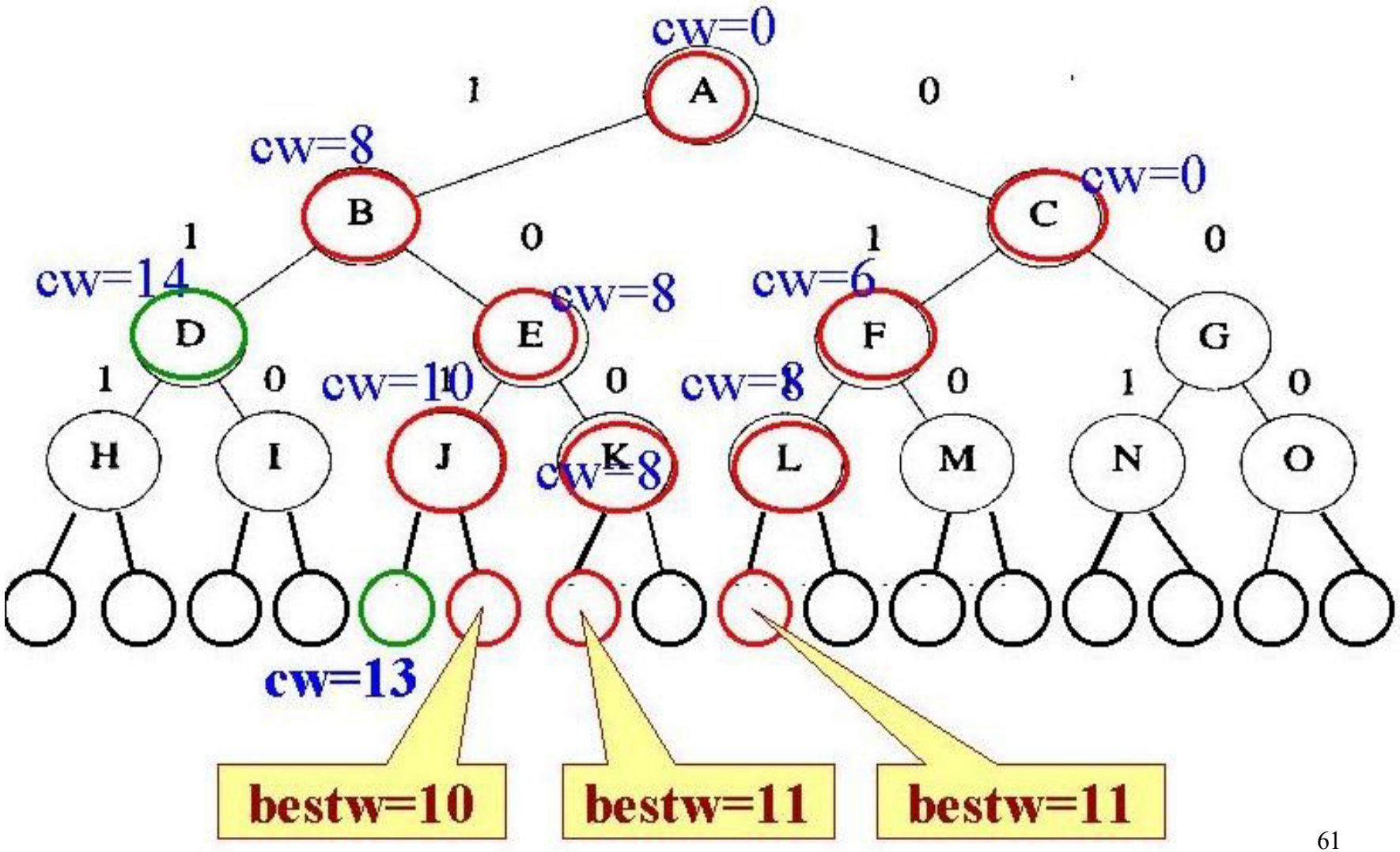
- 限界方法1：设cw为已装重量，如$\mathbf{cw + w_{j+1} > c_1}$则杀死该子结点。
- 限界方法2：设bestw为当前最优装箱重量， r为未装的货箱的总重量，如cw+r<=bestw,则停止展开该结点。
- 两种限界同时使用。

剪枝条件:

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

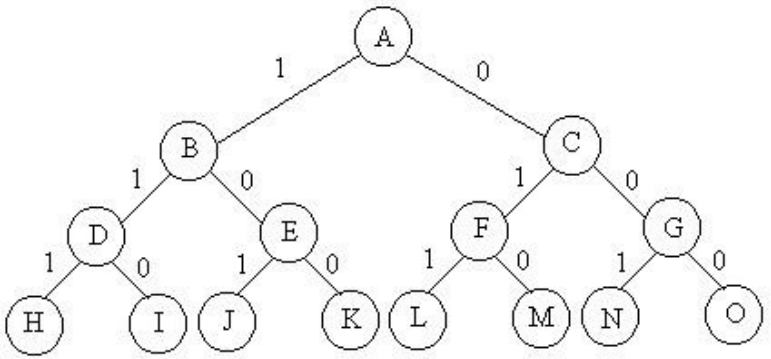
$$cw + r \leq bestw$$

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



装载问题

```
template < class Type >
Type Maxloading(type w[], type c,  int n,) //返回最优装载量
{ 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;   }
```



装载问题

```
template <class Type>
void Loading<Type>:: void backtrack (int i)
{// 搜索第i层结点
   if (i > n)   {// 到达叶结点
        bestw=cw;return;}
 //搜索子树
   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];   }
```

时间复杂度

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。