回溯算法

陈长建

计算机科学系

特别提醒

- 14周进行
  - 算法分析题 3-2、3-3(要求：有ppt代码演示)
  - 算法实现题 3-2、3-11(要求：有ppt和代码演示)
  - 数学之美分主题 2个(要求：有ppt)

    （1）P111 第12章 有限状态机和动态规划、P227 第26章 维特比和他的维特比算法

    （2）第31章 大数据的威力

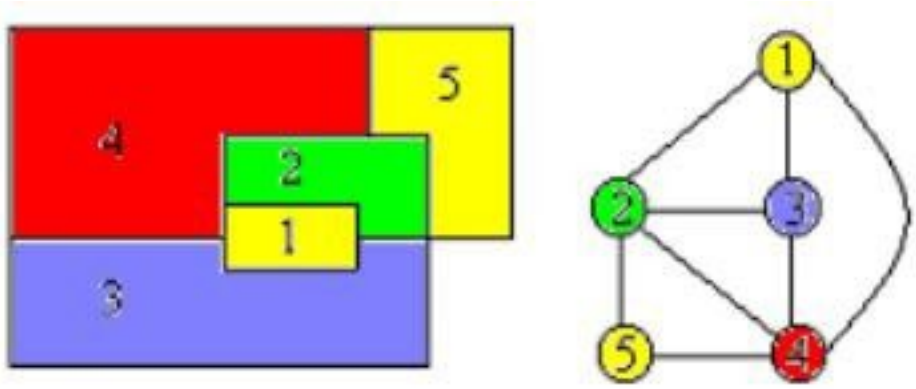
    每组讲解时间：10-12分钟+3-5分钟提问讨论

上节课回顾 - 图的m着色问题

- 图的着色问题是由地图的着色问题引申而来的：用m种颜色为地图着色，使得地图上的每一个区域着一种颜色，且相邻区域颜色不同。

上节课回顾 - 图的m着色问题

- 图着色问题描述为：给定无向连通图G=(V，E)和正整数m，求最小的整数m，使得用m种颜色对G中的顶点着色，使得任意两个相邻顶点着色不同。这个问题是图的m可着色判定问题。
- 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。



上节课回顾 - 图的m着色问题

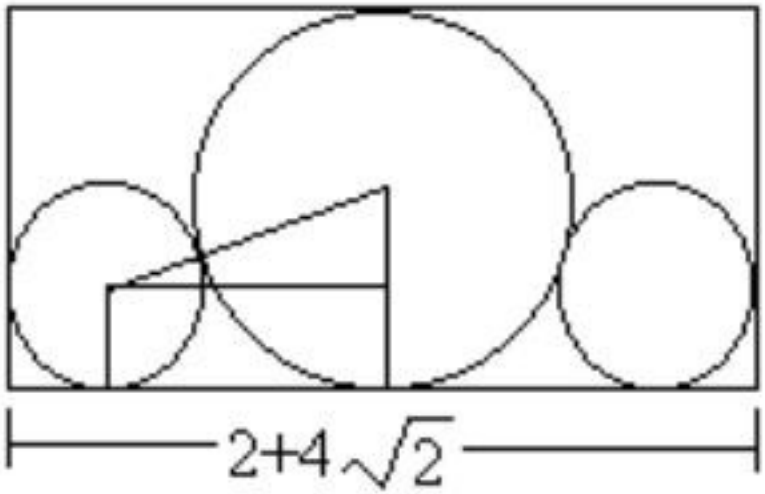
```
GraphColor(int n,int m,int color[],bool c[][5])
{
    int i,k;
    for (i=0; i<n; i++ )                    //将解向量color[n]初始化为0
        color[i]=0;
    k=0;
    while (k>=0)
    {   color[k]=color[k]+1;                //使当前颜色数加1
        while ((color[k]<=m) &&(!ok(color,k,c,n))) //当前颜色是否有效
                                  color[k]=color[k]+1;        //无效，搜索下一个颜色
        if (color[k]<=m)                    //求解完毕，输出解
        {
            if (k==n-1)break;               //是最后的顶点，完成搜索
               else k=k+1;                  //否，处理下一个顶点
        }
        else                                //搜索失败，回溯到前一个顶点
        {
            color[k]=0;
            k=k-1;
        }
    }
}
```

子集树和排列树?

- N皇后
- 图的m着色
- 最大团

圆排列问题

- 给定n个大小不等的圆$c_1,c_2,\ldots,c_n$，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。例如，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$2+4\sqrt{2}$



改进余地?

改进余地?

- 任何一个排列，将其倒序，结果相同

- 如果有k个半径相同的圆，可以合并k!个排列

0-1背包问题 – 再思考

- 0-1背包问题还能否再优化?
- 例:
  - N=4
  - C=10
  - w=[3, 3, 4, 3]
  - v=[40, 60, 50, 60]

0-1背包问题 – 再思考

- 限界函数：当剩余的都放下时，价值都不超过best
  - 怎么来判断?
  - 策略1：剩余价值的和 < best

0-1背包问题 – 再思考

- 限界函数：当剩余的都放下时，价值都不超过best
  - 怎么来判断?
  - 策略1：剩余价值的和 < best　　**估计不准**
  - 例1：当前剩余w={16,15,15}, p={45,25,25}, c=30, best=70

0-1背包问题 – 再思考

- 限界函数：当剩余的都放下时，价值都不超过best
  - 怎么来判断?
  - 策略1：剩余价值的和 < best　　**估计不准**
  - 策略2：尽可能放单位价值高的

0-1背包问题 – 再思考

- 限界函数：当剩余的都放下时，价值都不超过best
  - 怎么来判断?
  - 策略1：剩余价值的和 < best　　**估计不准**
  - 策略2：尽可能放单位价值高的　　**估计不对**
  - 例2：当前剩余w={16,15,15}, p={45,25,25}, c=30, best=49

0-1背包问题 – 再思考

- 限界函数：当剩余的都放下时，价值都不超过best
  - 怎么来判断?
  - 策略1：剩余价值的和 < best　　**估计不准**
  - 策略2：尽可能放单位价值高的　　**估计不对**
  - 策略3: 用部分0-1背包问题的贪心解法估计
    - 例1：当前剩余w={16,15,15}, p={45,25,25}, c=30, best=70
      - 估计bound: 68.3333
    - 例2：当前剩余w={16,15,15}, p={45,25,25}, c=30, best=49

0-1背包问题 – 再思考

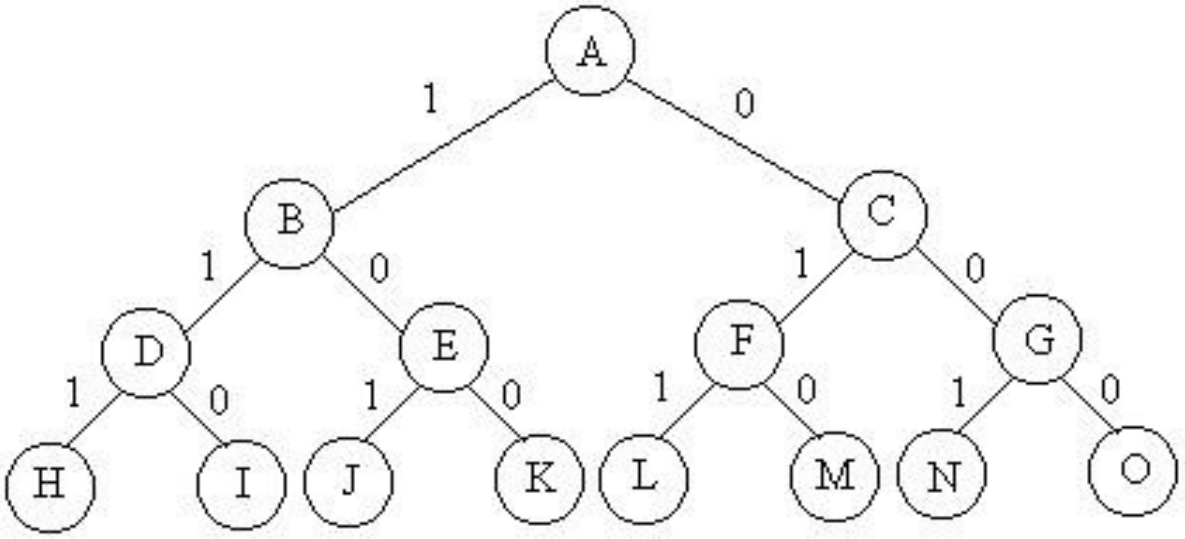
```
while (i <= n && w[i] <= cleft)      // n表示物品总数，cleft为剩余空间
{
        cleft -= w[i];                // w[i]表示i所占空间
        b += p[i];                    // p[i]表示i的价值
        i++;
}
if (i <= n) b += p[i] / w[i] * cleft;   // 装填剩余容量装满背包
return b;                             // b为上界函数
```

0-1背包问题 – 再思考

- 在第四章中,我们用动态规划的方法讨论了背包问题。这里，我们将用回溯的方法求解背包问题。
- 问题的解空间：由各分量 x_i(取值0或1）的2^n个不同的 n元向量组成。与子集和数问题的解空间相同。也用树结构表示（满二叉树）。
- 限界函数：取能产生某些值的上界函数。如果扩展给定活结点和它的任一子孙所导致最好可行解的上界不大于迄今所确定的最好解的值，就可杀死此活结点。
- 0-1背包问题的限界函数：用贪心方法求取上界值。

0-1背包问题

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

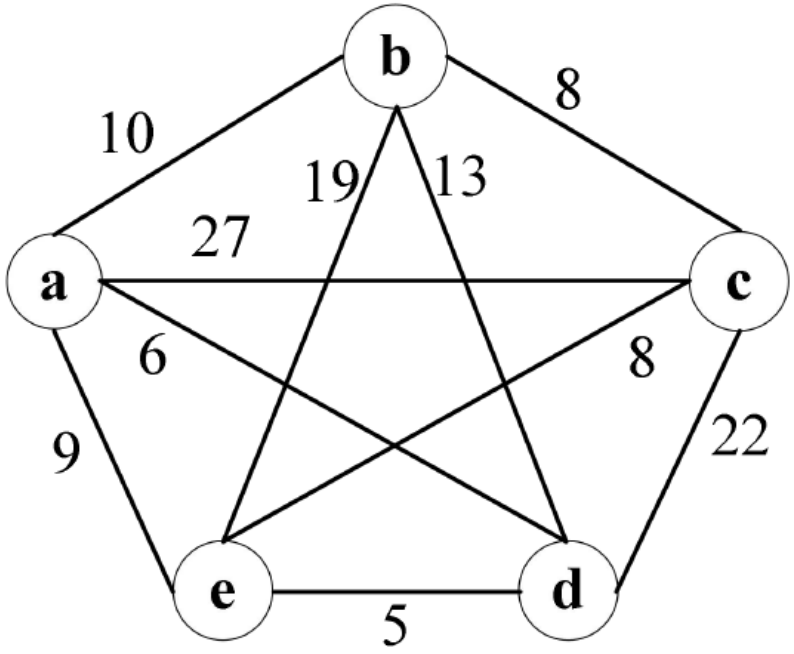


0-1背包问题

```
template<class Typew, class Typep> Typep
Knap<Typew, Typep>::Bound(int i)
{ //  (用贪心方法求取上界值)
  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;//右子树中解的上界
}
```

TSP问题 – 再思考

- 能否再进一步优化?



旅行售货员问题

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
  if (i == n) { //当前扩展结点是排列树的叶结点的父结点
    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 { //当前扩展结点位于排列树的第i-1层
  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]);} }
}
```

回溯法的效率分析

- 本节要点：
- 一、影响回溯算法效率的因素
- 二、重排原理
- 三、回溯法的效率

影响回溯算法效率的因素

- 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。

影响回溯算法效率的因素

- 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。

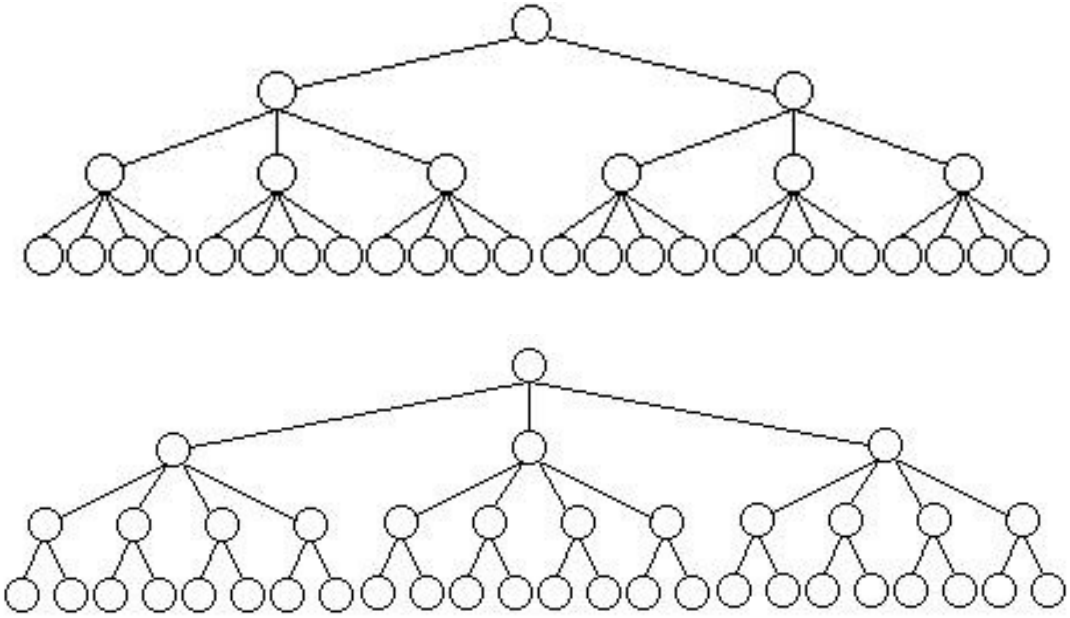
影响回溯算法效率的因素

- 回溯算法的**最坏情况时间复杂度**可达$O(p(n)n!)$（或$O(p(n)2^n)$或$O(p(n)n^n)$），这里$p(n)$是n的多项式，是生成一个结点所需的时间。

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。
- 从下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。

重排原理



(a)

(b)

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。

练习

- 设有n件工作分配给n个人。将工作i分配给第j个人所需的费用为 cij。试设计一个算法，为每一个人都分配1 件不同的工作，并使总费用达到最小。

回溯法小结

- **回溯法要求**问题P的状态能表达为n元组(x_1，x_2,… x_n)，**要求** $x_i \in s_i$，i =1,2,…n，s_i**为有限集**，**对于给定关于**n元组中的分量的一个约束集D，满足D的全部约束条件的所有n元组为问题P的解。
- 从k=1开始构造k元组，如果k元组满足约束，k=k+1，扩展搜索；如果试探了x[k]的所有值，仍不能满足约束，则 k=k-1，回溯到上一层重新选择x[k-1]的值。这种扩展回溯的搜索方法可以表示为状态空间树上的带约束条件的深度优先的搜索。

作业

- 没有

作业

- 没有?

作业

- 1. 算法分析题5-3 回溯法重写0-1背包
- 2. 算法实现题5-7 n色方柱问题
- 3. 算法实现题6-2 最小权顶点覆盖问题
- 4. 算法实现题6-6 n后问题（分支限界法）

分支限界算法

陈长建

计算机科学系

学习要点

- 掌握分支限界法的广度优先搜索策略
- 掌握分支限界法解题的步骤
- 掌握本章经典案例的问题定义、解空间、分支限界法求解过程、剪枝函数及算法复杂度
- 能够对具体问题，设计分支限界法3个步骤的具体做法
- 能够将其它同类问题转换为经典案例问题

分支限界法 VS 回溯法

（1）求解目标

- 回溯法：能够找出解空间树中满足约束条件的所有解
- 分支限界法：
  - 找出满足约束条件的一个解
  - 或是在满足约束条件的解中找出某种意义下的最优解

（2）搜索方式

- 回溯法：深度优先
- 分支限界法：广度优先或以最小耗费优先

分支限界法的基本思想

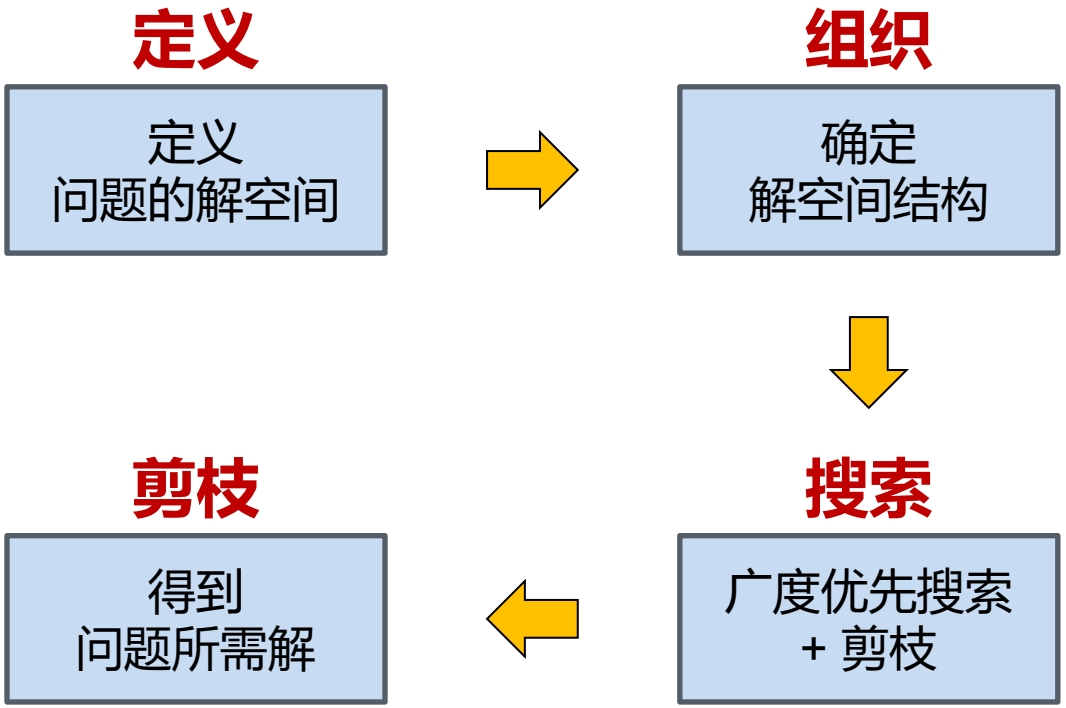
- 分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。
- 每一个活结点只有一次机会成为扩展结点。
  - 活结点一旦成为扩展结点，就一次性产生其所有子结点。
  - 在这些子结点中，导致不可行解或导致非最优解的子结点被舍弃，其余儿子结点被加入活结点表中。
- 从活结点表中取下一结点作为当前扩展结点，进行扩展。
- 直到找到所需的解或活结点表为空时为止。

分支限界法的基本思想

常见的两种分支限界法

- 队列式（FIFO）分支限界法
  - 按照队列先进先出（FIFO）原则选取下一个节点为扩展节点。
- 优先队列式分支限界法
  - 按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

分支限界法的基本步骤



难点：剪枝

(约束+限界)

例1- 0-1背包问题

- 给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?
- 问题三要素：
  - 输入：$C > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$
  - 输出：$(x_1, x_2, \ldots, x_n), x_i \text{ in } \{0, 1\}$
  - 约束：$\sum_{1 \leq i \leq n} w_i x_i \leq C$，$\sum_{1 \leq i \leq n} v_i x_i$ 最大

算法思想

- 对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列。
- 优先队列分支限界法，节点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。

算法思想

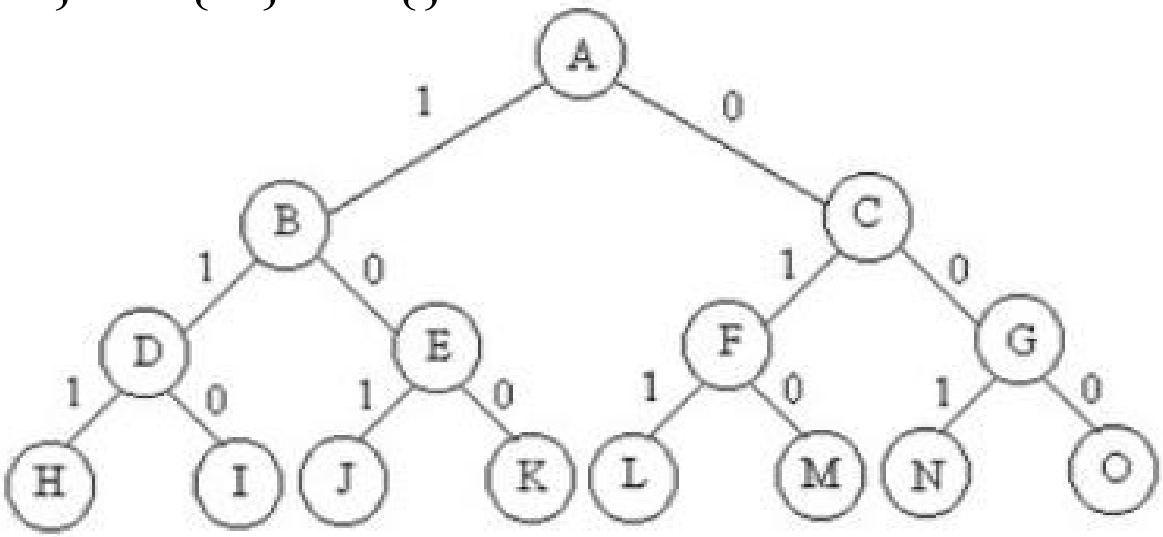
- 子集树
  - 首先检查当前扩展结点的左子结点的可行性。
  - 如果该左子结点是可行结点，则将它加入到子集树和活结点优先队列中。
  - 当前扩展结点的右子结点一定是可行结点，仅当右子结点满足上界约束时才将它加入子集树和活结点优先队列。
  - 当扩展到叶节点时为问题的最优值。

算法思想

- 例: 给定n=3，w={16,15,15}, p={45,25,25}, c=30。
  - 价值大者优先

算法思想

- 例: 给定n=3，w={16,15,15}, p={45,25,25}, c=30。
  - 价值大者优先
  - {}—>{A}—>{B, C}—>{C, D, E}—>{C, E}—>{C, J, K}—>{C}—>{F, G}—>{G, L, M}—>{G, M}—>{G}—>{N, O}—>{O}—>{}



示例代码

```
while (i != n+1) {// 非叶结点
    // 检查当前扩展结点的左儿子结点
    Typew wt = cw + w[i];
    if (wt <= c) {// 左儿子结点为可行结点
      if (cp+p[i] > bestp) bestp = cp+p[i];
      AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}
      up = Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if (up >= bestp) // 右子树可能含最优解
        AddLiveNode(up, cp, cw, false, i+1);
    // 取下一个扩展节点（略）
}
```

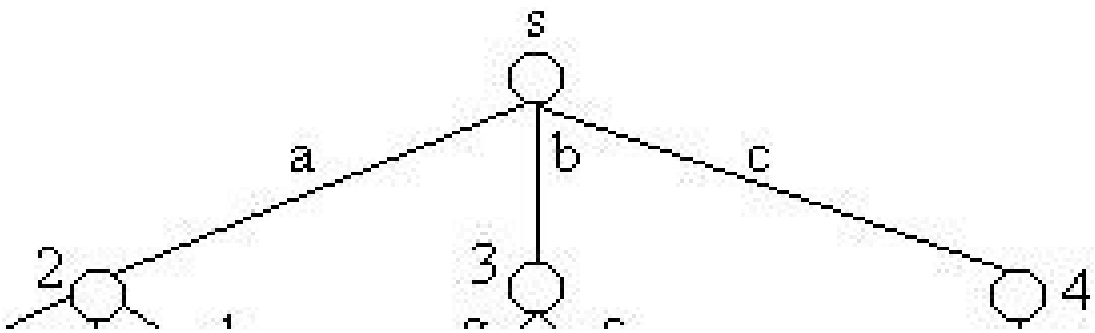
分支限界
搜索过程

示例代码

```
while (i <= n && w[i] <= cleft)      // n表示物品总数，cleft为剩余空间
{
        cleft -= w[i];                // w[i]表示i所占空间
        b += p[i];                    // p[i]表示i的价值
        i++;
}
if (i <= n) b += p[i] / w[i] * cleft;   // 装填剩余容量装满背包
return b;                             // b为上界函数
```

队列式 VS 优先队列式

- 队列式：子节点按生成顺序入队和出队
  - 例如，队列元素：可以有 {s}->{a, b, c}
- 优先队列式：子节点按优先级出队
  - 【若越大越好】{s}->{a, b, c}
  - 【若越小越好】{s}->{c, b, a}



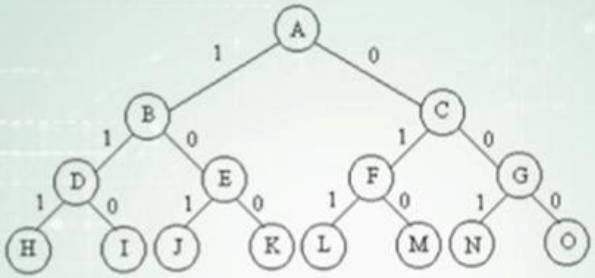
队列式 VS 优先队列式

- 队列式：子节点按生成顺序入队和出队
  - 均匀逐层推进
  - 约束+限界
- 优先队列式：子节点按优先级出队
  - 向最优解的方向快速推进
  - 优先级策略
  - 约束+限界

用限界函数计算优先级

解空间树 VS 搜索树

- 解空间树：不必考虑搜索策略
- 搜索树：考虑搜索策略+剪枝策略



	广度优先搜索	分支限界算法
图搜索视角	遍历整棵完全二叉树	添加剪枝策略，尽量限制搜索无效的分支
枚举视角	全局判定 $[x1, x2, \cdots, xn]$	局部预判 $[x1, ?, ?, \cdots]$ $[x1, x2, ?, \cdots]$

练习

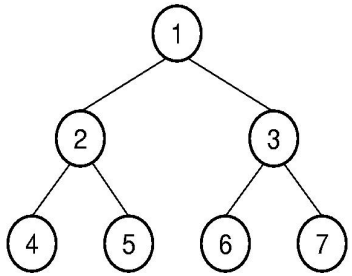
- 给定n=3，w={16,15,15}, p={45,25,25}, c=30。请用队列式分支限界法和优先队列式分支限界法求解该0-1背包问题，并对比其异同。

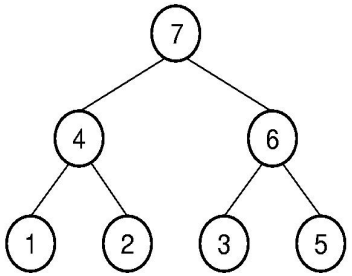
关键数据结构： 堆栈和队列

- 堆栈：先进后出
- 队列：先进先出
- 优先级队列: 顺序入队，高优先级先出队

优先级队列：最大/小堆（heap）

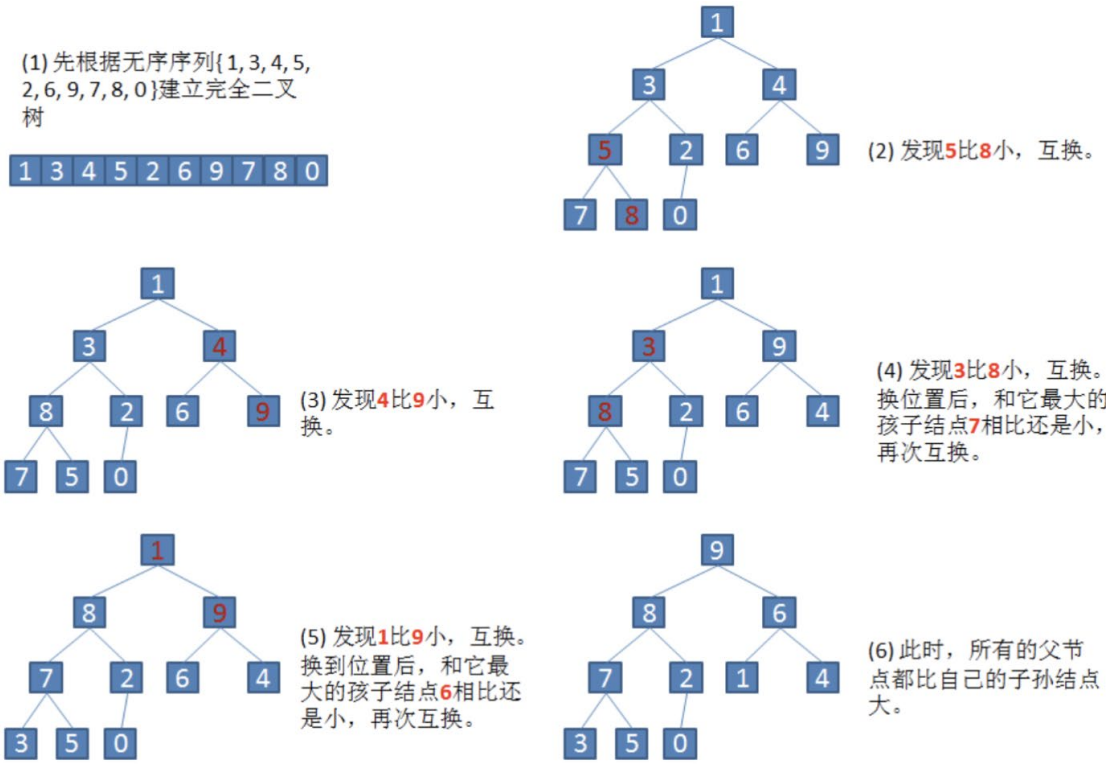
- 堆树或者是一棵空树，或者是具有下列性质的一棵完全二叉树（堆的局部有序特性），堆可以用来表现优先级
  - 最小值堆：每一个结点存储的值都小于或等于其子结点存储的值
  - 最大值堆：每一个结点的值都大于或等于其任意一个子结点存储的值





最大/小堆（heap）

- 堆的复杂度
  - 构建: O(N)
  - 插入: O(logN)
- 链表
  - 构建：O(N)
  - 插入：O(N)



例2-两艘船的装载问题

- 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，且满足

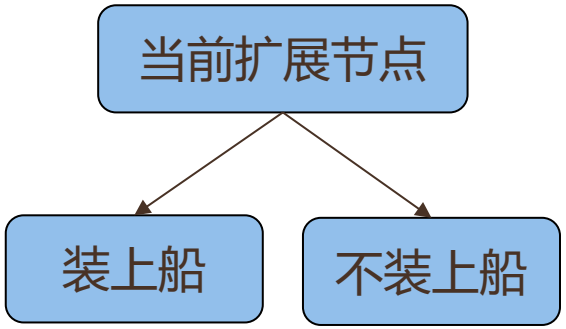
$$\sum_{i=1}^{n} w_i \leq c_1 + c_2$$

- 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。

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。

队列式分支限界法

- 首先检测当前扩展结点的左儿子结点是否为可行结点。若是，加入到活结点队列。再将其右儿子结点加入(一定是可行结点)。2个儿子结点都产生后，舍弃当前扩展结点。



- 非空判断及下一层扩展
  - 活结点队列中的队首元素被取出作为当前扩展结点。队列中每一层结点之后有一尾部标记-1。
  - 当取出的元素是-1时，再判断当前队列是否为空。若非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点。

示例代码

```
while (true) {
    // 检查左儿子结点
    if (Ew + w[i] <= c)       // x[i] = 1
        EnQueue(Q, Ew + w[i], bestw, i, n);
    // 右儿子结点总是可行的
    EnQueue(Q, Ew, bestw, i, n);    // x[i] = 0
    Q.Delete(Ew);             // 取下一扩展结点
    if (Ew == -1) {           // 同层结点尾部
        if (Q.IsEmpty()) return bestw;
        Q.Add(-1);            // 同层结点尾部标志
        Q.Delete(Ew);         // 取下一扩展结点
        i++; }                // 进入下一层
} }
```

算法改进

- 节点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。设bestw是当前最优解；ew是当前扩展结点所相应的重量；r是剩余集装箱的重量。则当ew+r<bestw时，可将其右子树剪去，因为此时若要船装最多集装箱，就应该把此箱装上船。
- 为确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新bestw的值

示例代码

```
// 检查左子结点
Type wt = Ew + w[i];        // 左子结点的重量
    if (wt <= c) {          // 可行结点
        if (wt > bestw) bestw = wt;
        // 加入活结点队列
        if (i < n) Q.Add(wt);
 }

// 检查右子结点
if (Ew + r > bestw && i < n)
    Q.Add(Ew);              // 可能含最优解
    Q.Delete(Ew);           // 取下一扩展结点
```

提前更新bestw

右子节点剪枝

构造最优解

- 为方便构造与最优值相应的最优解，须存储子集树中从活结点到根结点的路径。为此，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志。
- 找到最优值后，可以根据parent回溯到根节点，找到最优解。

```
class QNode
{
   QNode *parent; // 指向父结点的指针
   bool LChild;    // 左儿子标志
   Type weight;    // 结点所相应的载重量
}
```

```
// 构造当前最优解
for (int j = n - 1; j > 0; j--)
{
   bestx[j] = bestE->LChild;
   bestE = bestE->parent;
}
```

优先队列式分支限界法

- 用最大优先队列存储活结点表。活结点x的优先级定义为从根结点到结点x的路径所相应的载重量再加上剩余集装箱的重量之和。
- 优先级最大的活结点成为下一个扩展结点。以结点x为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同。
- 一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。