算法设计与分析

第5章 回溯法

学习要点

- 理解回溯法的深度优先策略
- 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

- 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）最大团问题　　（7）图的m着色问题（8）旅行售货员问题

  （9）圆排列问题（10）连续邮资问题

5.1 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。

5.1 回溯法

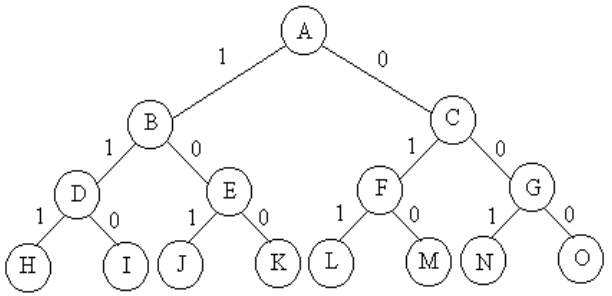
问题的解空间

- 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）。

W=[16,15,15] p=[45,25,25] C=30

- n=3时的0-1背包问题用完全二叉树表示的解空间



5.1 回溯法

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。
- 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）。
- 广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
- 回溯法：为了避免生成那些不可能产生最优解的问题状态，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。

5.1 回溯法

回溯法的基本思路

- 从根开始，以深度优先搜索的方式进行搜索。
- 根结点是活结点并且是当前的扩展结点。在搜索的过程中，当前的扩展结点向纵深方向移向一个新结点，判断该新结点是否满足隐约束。
- 如果满足，则新结点成为活结点，并且成为当前的扩展结点，继续深一层的搜索；
- 如果不满足，则换该新结点的兄弟结点（扩展结点的其它分支）继续搜索；
- 如果新结点没有兄弟结点，或其兄弟结点已全部搜索完毕，则扩展结点成为死结点，搜索回溯到其父结点处继续进行。
- 搜索过程直到找到问题的解或根结点变成死结点为止。

5.1 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。

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。

- 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。

5.1 回溯法

回溯法的基本思想

- 明确搜索范围（定义问题的解空间）

  解的形式：一个n元式(x1,x2,…,xn)

  确定xi的取值范围（显约束），确定解空间的大小。

- 确定解空间的组织结构——树或图
- 搜索解空间（隐约束）

  确定是否能够导致可行解——约束条件

  确定是否能够导致最优解——限界条件

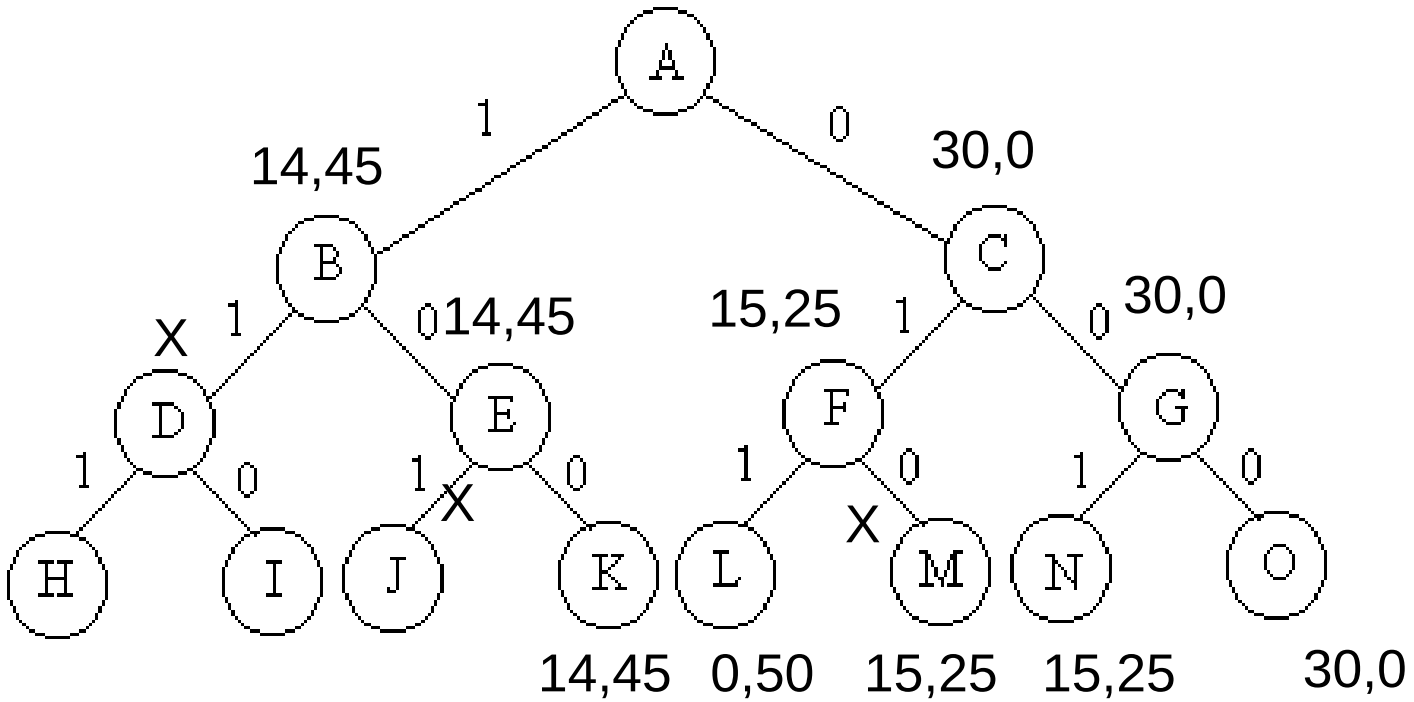
- 用约束函数在扩展结点处剪去不满足约束的子树即：导致不可行解的节点；
- 用限界函数剪去得不到最优解的子树。即：导致非最优解的节点。

5.1 回溯法

回溯法的基本思想

例如：n=3的 0-1 背包问题的回溯法搜索过程。

W=[16,15,15] p=[45,25,25] C=30



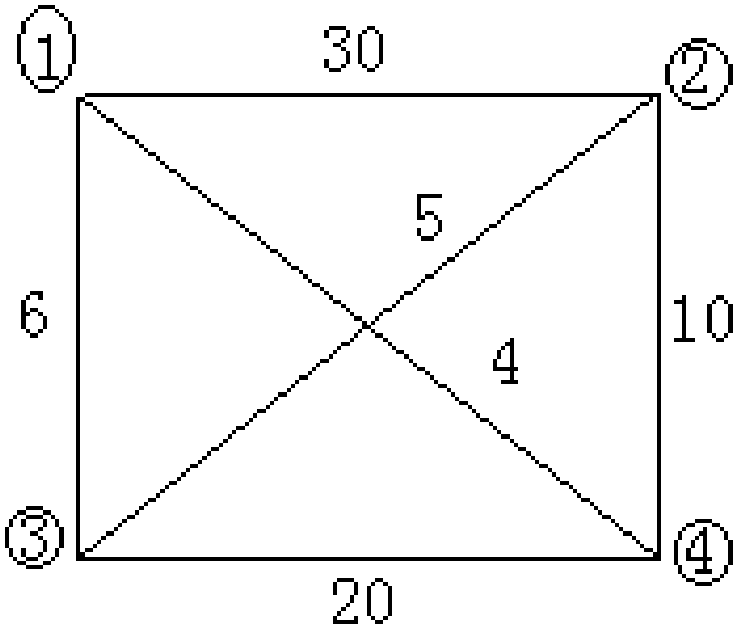
对物品1的选择

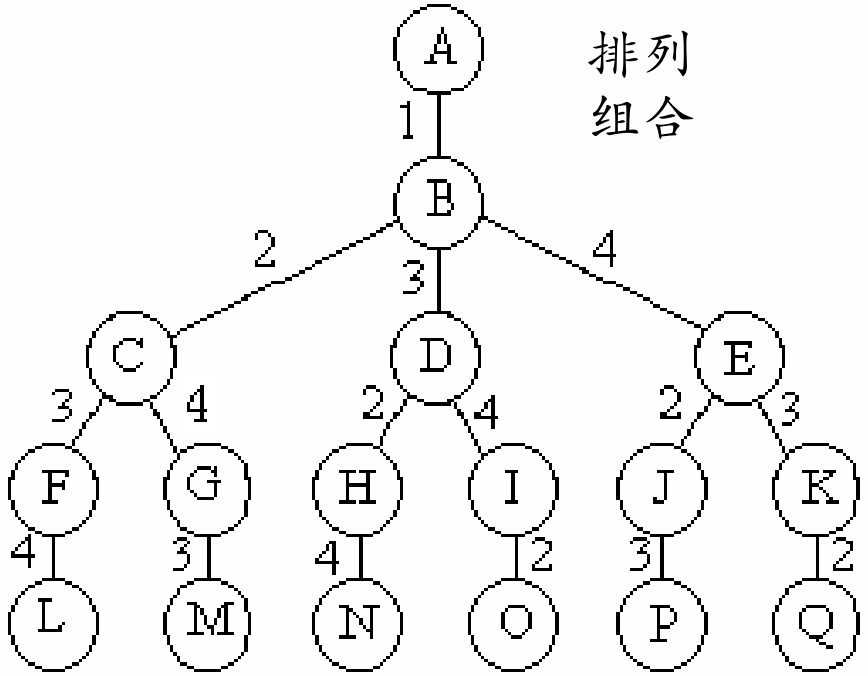
对物品2的选择

对物品3的选择

回溯法的基本思想

例如2：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（旅费），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（总旅费）最小。





5.1 回溯法 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); // 到达叶子结点，将结果输出
     else
       // 遍历结点t的所有子结点，即枚举t所有可能的路径
       for (int i=f(n,t);i<=g(n,t);i++) //检查扩展结点的每个分支
           x[t]=h(i); ;//取当前扩展节点的第i个可选值
           // 如果不满足剪枝条件，则继续遍历，进入下一层
           if (constraint(t)&&bound(t)) backtrack(t+1);
       }
}
```

- 其中f(n,t),g(n,t)表示当前扩展结点处未搜索过的子树的起始标号和终止标号，h(i)表示当前扩展节点处，x[t]第i个可选值。
- constraint(t)和bound(t)是当前扩展结点处的约束函数和限界函数。constraint(t)返回true时，在当前扩展结点x[1:t]取值满足约束条件，否则不满足约束条件，可剪去相应的子树。bound(t)返回的值为true时，在当前扩展结点x[1:t]处取值未使目标函数越界，还需要由backtrack(t+1)对其相应的子树进一步搜索。
- 有时用Legal(t) 表示 constraint(t)&&bound(t)

```
void iterativeBacktrack (){
    int t=1;
    while (t>0) {//有路可走
      if (f(n,t)<=g(n,t)) {
        // 遍历结点t的所有子结点
        for (int i=f(n,t);i<=g(n,t);i++) {
           x[t]=h(i);
          if (constraint(t)&&bound(t)) {
            // 找到问题的解，输出结果
            if (solution(t)) output(x);
            // 未找到，向更深层次遍历
            else t++;}
        }
       else t--; //所有子节点搜索完毕，回溯到上一层的活结点
    }}
```

- 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。

子集树与排列树

- 当所给问题是从n个元素的集合S中找出满足某种性质的子集时，解空间为子集树。

（装载问题、符号三角形问题、0-1背包问题、最大团问题）

- 当所给问题是从n个元素的集合S中找出满足某种性质的排列时，解空间为排列树。

（批处理作业调度、n后问题、旅行售货员问题、圆排列问题、电路板排列问题）

子集树

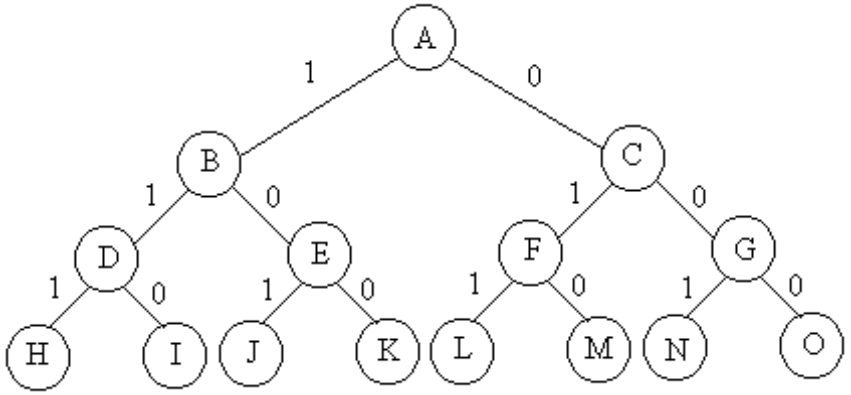
- 当所给的问题是从n个元素组成的集合S中找出满足某种性质的一个子集时，相应的解空间树称为子集树。
- 此类问题解的形式为n元组（x1,x2,…,xn），分量xi(i=1,2,…,n)表示第i个元素是否在要找的子集中，通常有2^n个叶子节点。
- xi的取值为0或1，xi=0表示第i个元素不在要找的子集中；xi=1表示第i个元素在要找的子集中。

排列树

- 当所给的问题是从n个元素的排列中找出满足某种性质的一个排列时，相应的解空间树称为排列树。排列数通常有n!个叶子节点

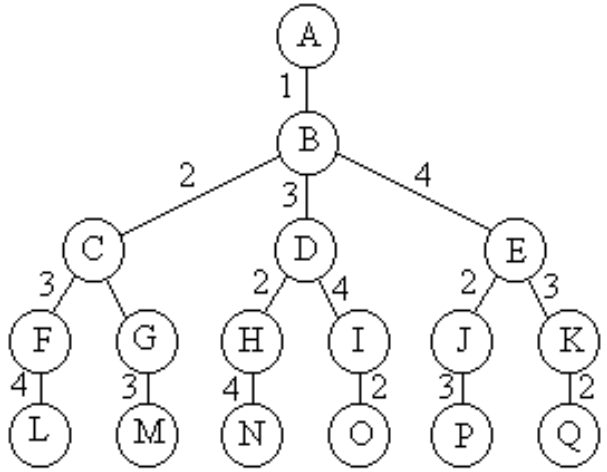
- 此类问题解的形式为n元组（$x_1,x_2,\ldots,x_n$），分量xi(i=1,2,…,n)表示第i个位置的元素是xi。

子集树与排列树



- 遍历子集树需Ω(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++) {//两个分支
        x[t]=i;
        if (legal(t)) backtrack(t+1);
      }
}
```

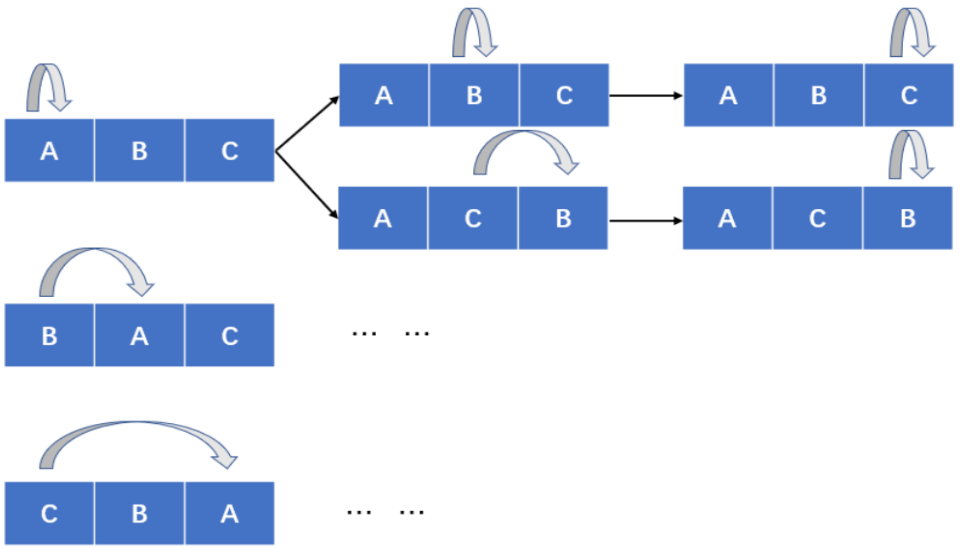


- 遍历排列树需要Ω(n!)计算时间

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

排列树：用交换来实现

给定abc，分别在第一个位置，第二个位置和第三个位置。
第一个空应该填的有三种可能。
当第一个位置的值与第二个位置交换的时候。得到了bac，此时就代表我们选定的第一个位置的字符为b，
接下来的处理其实可以看作：剩下两个位置，剩下两个字符，找出他们的全排列。可以还用刚才的方法，拿第二个位置的值与第二、第三位置分别交换，以此类推，就找到了所有的全排列。



每次交换选定后，递归搜索出他后面字符的全排列后，还需要把该字符交换回去

5.2 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批集装箱装上这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。

(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。

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

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

用回溯法设计解装载问题的$O(2^n)$计算时间算法。

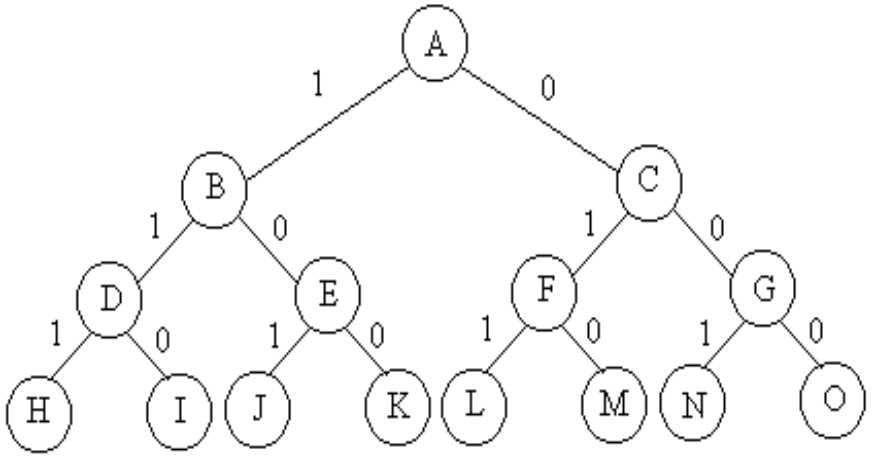
5.2 装载问题

- 解空间：子集树
- 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \le c_1$　　（是否是可行解）
- 限界函数(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。（是否是最优解）

```
static int n;          // 集装箱数
static int w[ ];       // 集装箱重量数组
static int c;          // 第一艘轮船的载重量
static int cw;         // 当前载重量
static int bestw;      // 当前最优载重量
static int r;          // 剩余集装箱重量
static int x[ ];       // 当前解
static int bestx[ ];   // 当前最优解
```

5.2 装载问题

- 对于问题1需要通过判断节点i（集装箱i）的装与不装形成一个二叉树，一条到叶子节点的通路就是一种方案。
- 再利用一变量存储此时重量，以此来记录最优的装载方案。
- 设 bestw：当前最优载重量，
- cw：当前扩展结点Z的载重量；
- r：剩余集装箱的重量；
- w[i]：i节点（i集装箱）的重量；
- 当 cw+w[i]<=c1时 可以放
- 当 cw + r (限界函数) <= bestw时，可将Z的右子树剪去。
  （以Z为根的子树载重量不超过cw+r,因此，均不可能是最优解）
- 当 cw + r (限界函数) > bestw时，可以讨论i节点不放的情况。



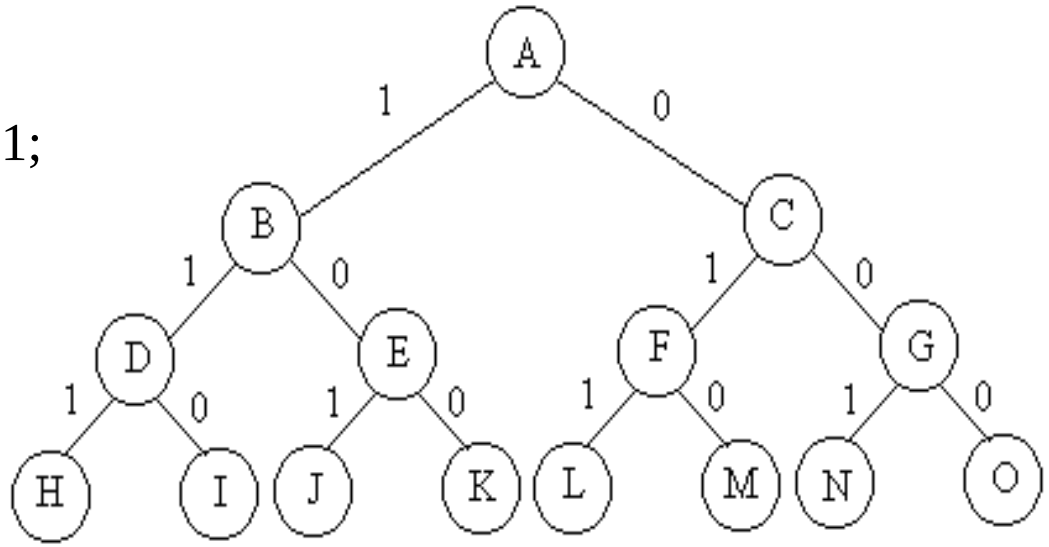

```
static int n;          // 集装箱数
static int w[ ];       // 集装箱重量数组
static int c;          // 第一艘轮船的载重量
static int cw;         // 当前载重量
static int bestw;      // 当前最优载重量
static int r;          // 剩余集装箱重量
static int x[ ];       // 当前解
static int bestx[ ];   // 当前最优解
```

伪代码：

- 用i记录第几个集装箱，判断该集装箱是否装入
- 如果剩余载重量大于该集装箱的重量，先不装入，否则装入。
- 再判断如果先不放该集装箱，再放剩下的集装箱是否可以达成最优装载。
- 利用回溯法，逐一判断集装箱，直到结束。

5.2 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;
       for (int  j=1; j<=n; j++)   bestx[j]=x[j]; // 保存最优解
       bestw=cw;
       return;}
    r -= w[i];
    if (cw + w[i] <= c) {// 如果能放入则搜索左子树x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i]; }
    if (cw + r > bestw)  { //如不能则看搜索
                           // 是否需要搜索右子树 x[i] = 0;
      backtrack(i + 1); }
    r += w[i];
  }
```



5.2 装载问题


```
int maxloading(int ww[ ], int cc, int x[ ])
{
   n=ww.length-1;
   w=ww;
   c=cc;
   bestw=0;
   x=new int[n+1];
   bestx=x;
   // 初始化 r
   for (int i=1; i<=n; i++)
       r+=w[i];
   // 计算最优载重量
   backtrack(1);
   return bestw;
}
```

5.3 批处理作业调度

- 给定n个作业的集合{J1,J2,…,Jn}。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。

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

1,2,3

t_{ji}	机器1	机器2
作业1	2（2）	1（3）
作业2	3（5）	1（6）
作业3	2（7）	3（10）

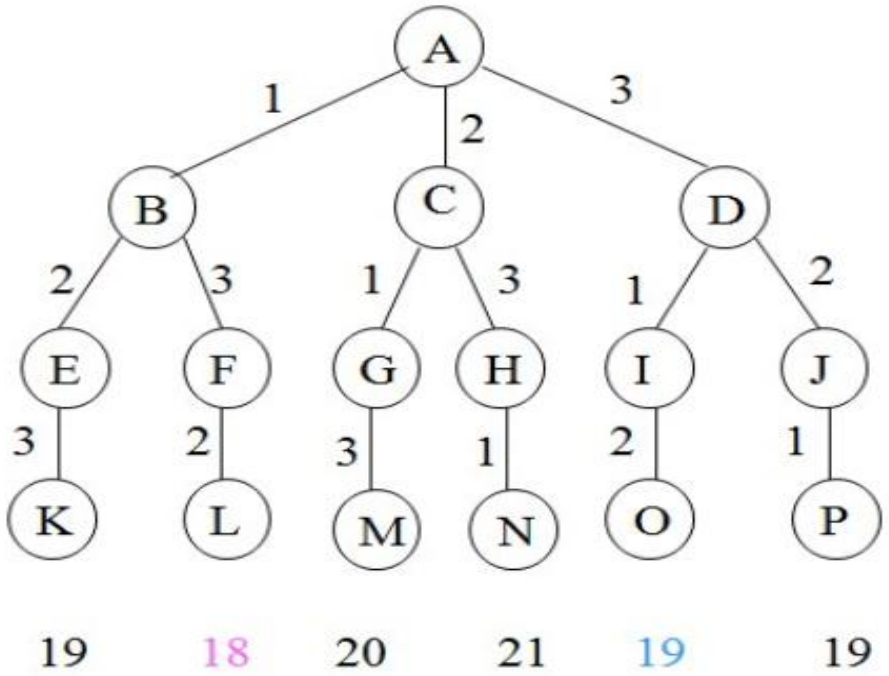
1,3,2

t_{ji}	机器1	机器2
作业1	2（2）	1（3）
作业3	2（4）	3（7）
作业2	3（7）	1（8）

- 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。易见，最佳调度方案是1,3,2，其完成时间和为18。

5.3 批处理作业调度

t_{ji}	机器1	机器2
作业1	2(2)	1(3)
作业2	3(5)	1(6)
作业3	2(7)	3(10)



以1,2,3为例，

- 作业1在机器1上完成的时间为2，在机器2上完成的时间为3
- 作业2在机器1上完成的时间为5，在机器2上完成的时间为6
- 作业3在机器1上完成的时间为7，在机器2上完成的时间为10
- 完成时间和=3+6+10=19

5.3 批处理作业调度

同济大学 TONGJI UNIVERSITY

框架：排列树框架

```
void backtrack(int t){
    if(t>n) output(x);
    else
        for(int i=t;i<=n;i++){//每个结点i都可以放在t这个位置
            swap(x[t],x[i]);
            if(legal(t)) backtrack(t+1)
            swap(x[t],x[i]);
        }
}
```

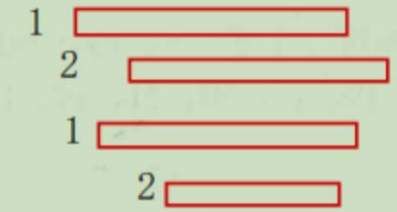
5.3 批处理作业调度

剪枝函数：前i个作业在机器2上的完成时间之和f<当前最优解bestf

```
class Flowshop {
  friend Flow(int**, int, int []);
  private:
    void Backtrack(int i);
    int  **M,    // 各作业所需的处理时间
          *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};
```

t_{ji}	机器1	机器2
作业1	2(2)	1(3)
作业2	3(5)	1(6)
作业3	2(7)	3(10)

机器1第i个任务完成时刻、2第i-1个任务完成时刻对比



机器2第i个任务完成时刻：

前一种：
f2[i-1]+M[x[i]][2]

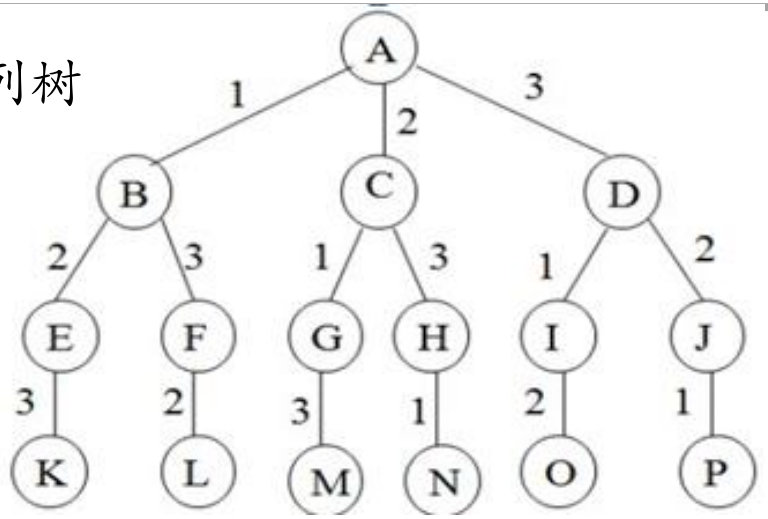
后一种：
f1[i]+M[x[i]][2]

考虑第i个任务在机器2上完成处理的时刻：
①对于机器1上只要考虑第i个任务的情况，所以f1保存
②对于机器2上也只要考虑第i-1个任务完成处理的时刻，但是最终结果是所有的f2之和，用数组保存

5.3 批处理作业调度

```
void Flowshop::Backtrack(int i){
  if (i > n) {//从1层开始，i表示到达的层数
     for (int j = 1; j <= n; j++) //得出一组最优值
       bestx[j] = x[j];
     bestf = f;
     }
  else
    for (int j = i; j <= n; j++) {//j用来指示选择了哪个任务
      f1+=M[x[j]][1];//选择第j个任务来执行
       //从f2[i - 1] 和 f1中选一个大的出来
      f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
      f+=f2[i];
      if (f < bestf) {
        //把选择出的任务j调到当前执行的位置i
         Swap(x[i], x[j]);
         Backtrack(i+1);// 选择下一个任务执行
         Swap(x[i], x[j]); //递归后恢复原样
         }
      f1- =M[x[j]][1];
      f- =f2[i];
    }
```

解空间：排列树




```
class Flowshop {
  friend Flow(int**, int, int []);
  private:
    void Backtrack(int i);
    int  **M,    // 各作业所需的处理时间
          *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};
```

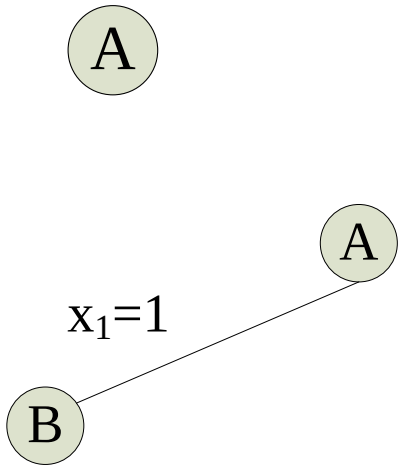
5.3 批处理作业调度

- 问题的解空间
  - 解的形式：(x1,x2,…,xn)
- 解空间的组织结构
  - 排列树，深度为问题的规模
- 搜索解空间
  - 约束条件（无）
  - 限界条件：当前部分排列的作业在机器2上的完成时间和f<当前找到的最优排列所需要的完成时间和bestf

5.3 批处理作业调度

搜索过程

tji	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3



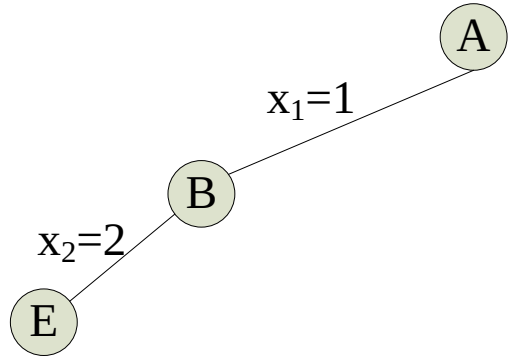
- 从根结点A开始，结点A成为活结点，并且是当前的扩展结点。

- 扩展结点A沿着x1=1的分支扩展，F11=2，F21=3，故f=3，bestf=+∞，f<bestf，限界条件满足。

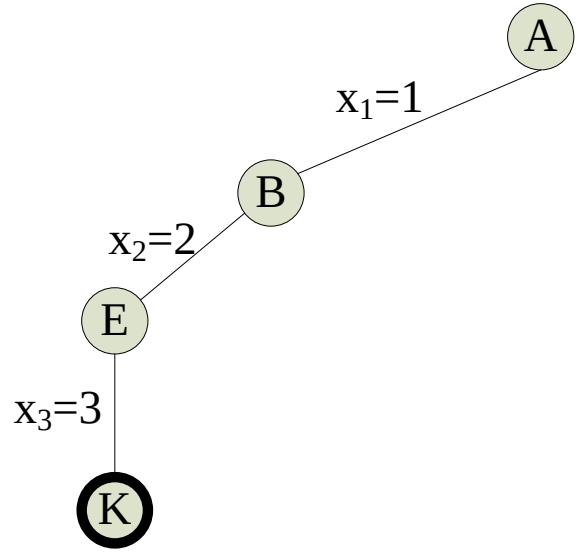
5.3 批处理作业调度

搜索过程

t_{ji}	机器1	机器2
作业1	2（2）	1（3）
作业2	3（5）	1（6）
作业3	2（7）	3（10）



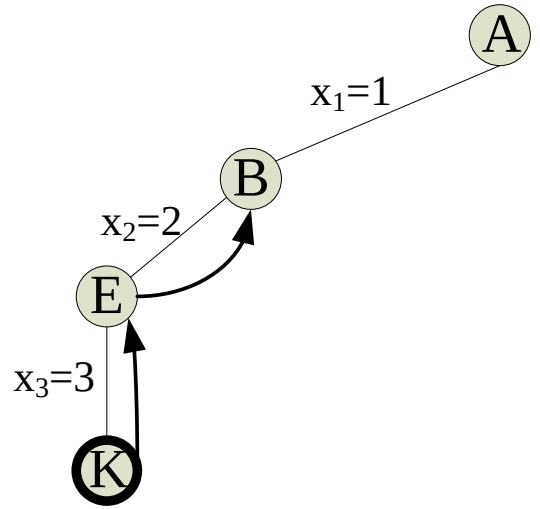
- 扩展结点B沿着x2=2的分支扩展，F12=5，F22=6，故f=F21+F22=9，bestf=+∞，f<bestf，限界条件满足。
- 扩展结点E沿着x3=3的分支扩展，F13=7，F23=10，故f=F21+F22+F23=19, bestf=+∞,f<bestf，限界条件满足，扩展生成的结点K是叶子结点。此时，找到当前最优的一种调度方案（1,2,3），同时修改bestf=19



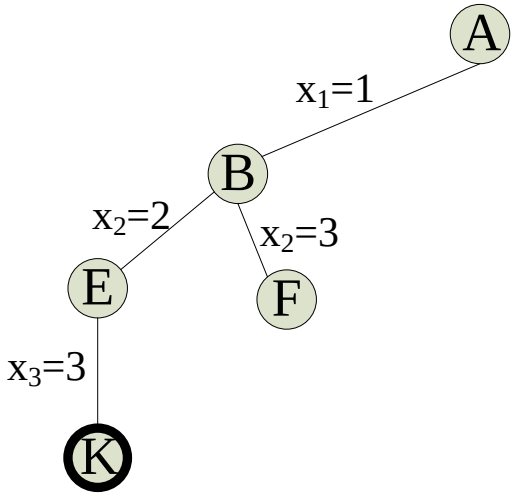
5.3 批处理作业调度

搜索过程

t_{ji}	机器1	机器2
作业1	2（2）	1（3）
作业2	3	1
作业3	2（4）	3（7）



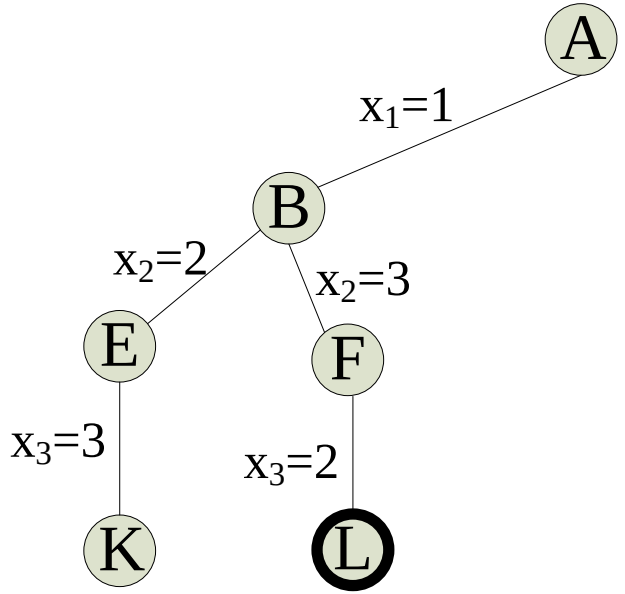
- 叶子结点K不具备扩展能力，开始回溯到活结点E。结点E只有一个分支，且已搜索完毕，因此结点E成为死结点，继续回溯到活结点B。
- 扩展结点B沿着x2=3的分支扩展，f=10，bestf=19，f<bestf，限界条件满足。



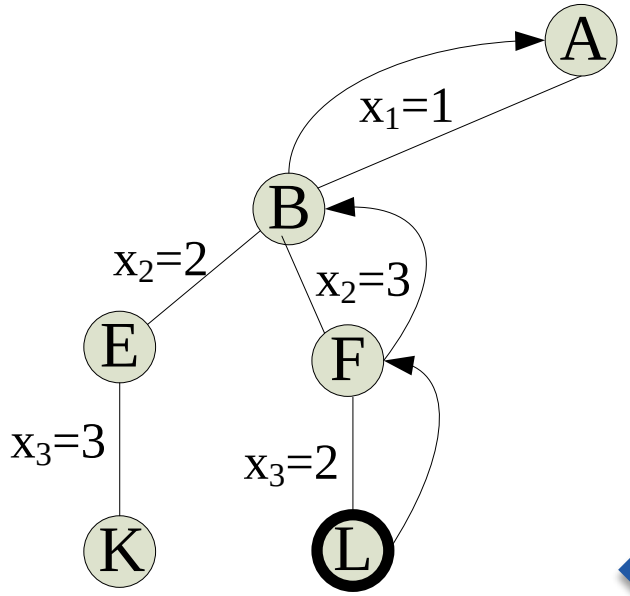
5.3 批处理作业调度

搜索过程

t_{ji}	机器1	机器2
作业1	2	1
作业2	3（7）	1（8）
作业3	2（4）	3（7）



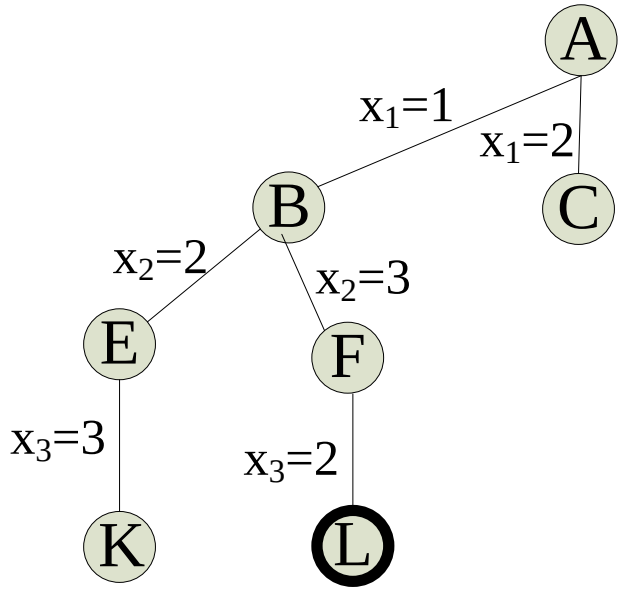
- 扩展结点F沿着x3=2的分支扩展，f=18，bestf=19，f<bestf，限界条件满足，扩展生成的结点L是叶子结点。此时，找到比先前更优的一种调度方案（1,3,2），修改bestf=18。
- 从叶子结点L开始回溯到活结点F。结点F的一个分支已搜索完毕，结点F成为死结点，回溯到活结点B。结点B的两个分支已搜索完毕，回溯到活结点A。



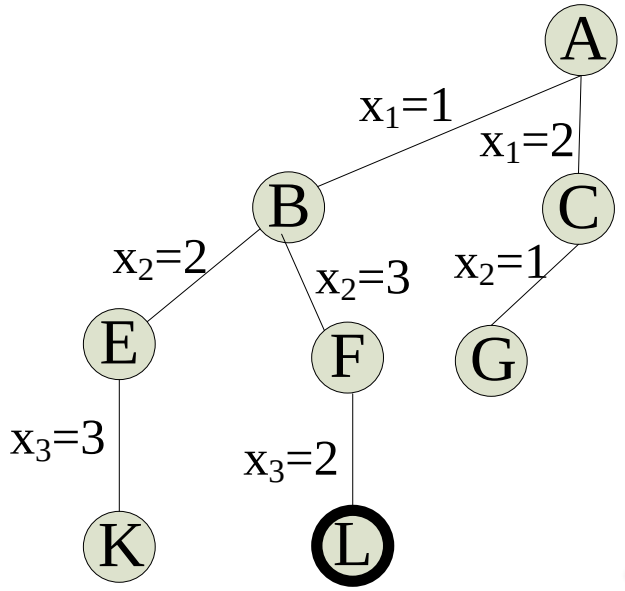
5.3 批处理作业调度

搜索过程

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



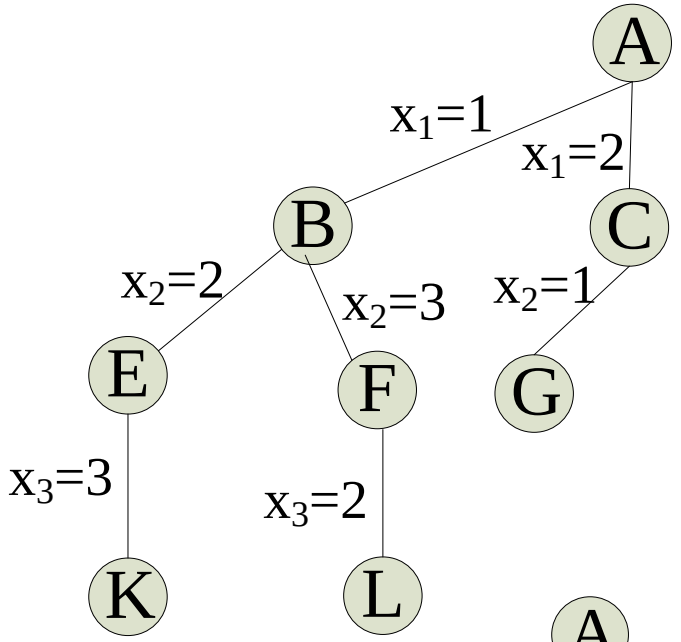
- 扩展结点A沿着x1=2的分支扩展，f=4，bestf=18，f<bestf，限界条件满足。
- 扩展结点C沿着x2=1的分支扩展，f=10，bestf=18，f<bestf，限界条件满足，扩展生成的结点G成为活结点，并且成为当前的扩展结点。



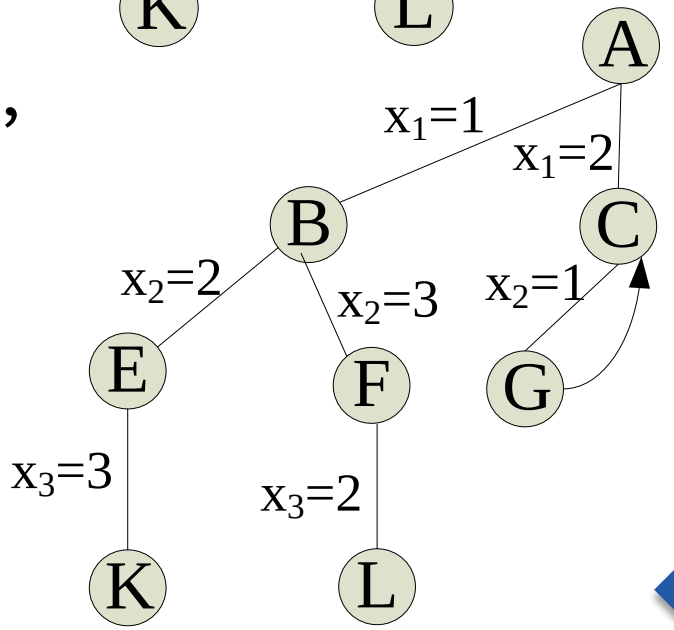
5.3 批处理作业调度

搜索过程

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



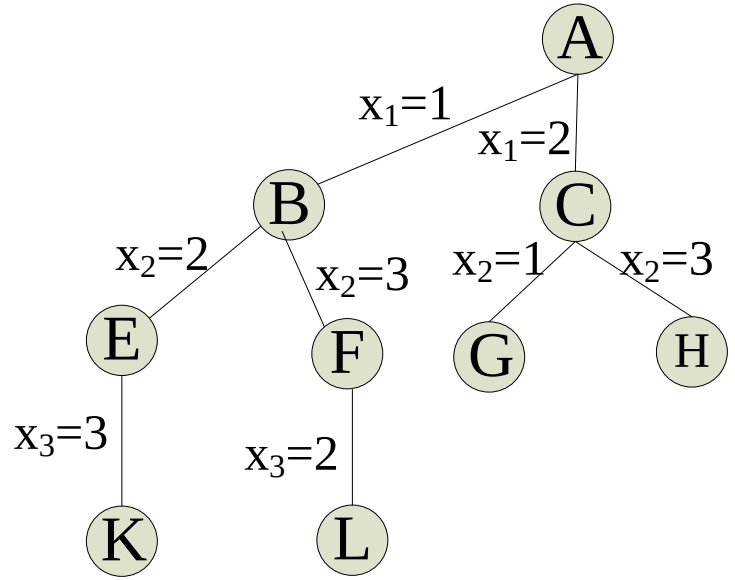
- 扩展结点G沿着x3=3的分支扩展，f=20，bestf=18，f>bestf，限界条件不满足，扩展生成的结点被剪掉。
- 结点G的一个分支搜索完毕，结点G成为死结点，回溯到活结点C。



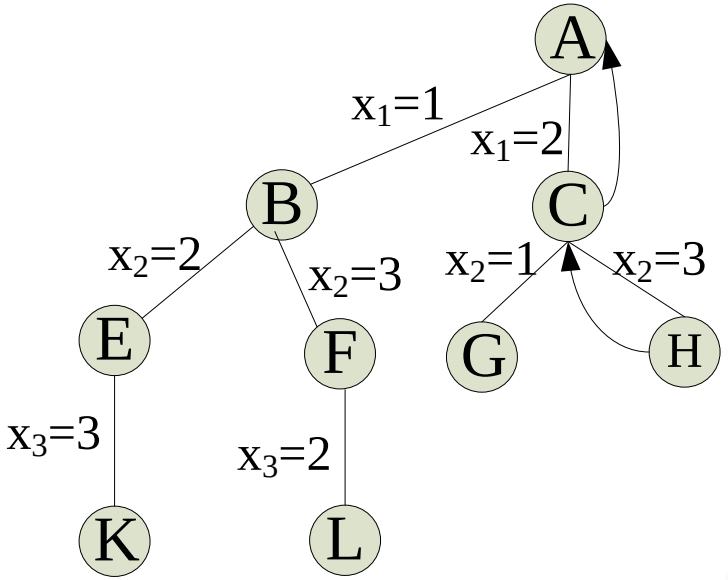
5.3 批处理作业调度

搜索过程

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



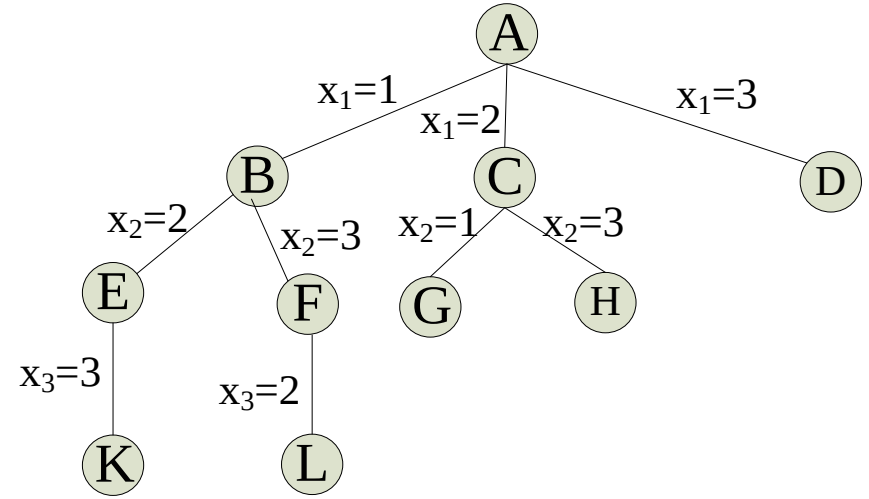
- 扩展结点C沿着x2=3的分支扩展，f=12，bestf=18，f<bestf，限界条件满足。
- 扩展结点H沿着x3=1的分支扩展，f=21，bestf=18，f>bestf，限界条件不满足，扩展生成的结点被剪掉。结点H的一个分支搜索完毕，开始回溯到活结点C。此时，结点C的两个分支已搜索完毕，继续回溯到活结点A。



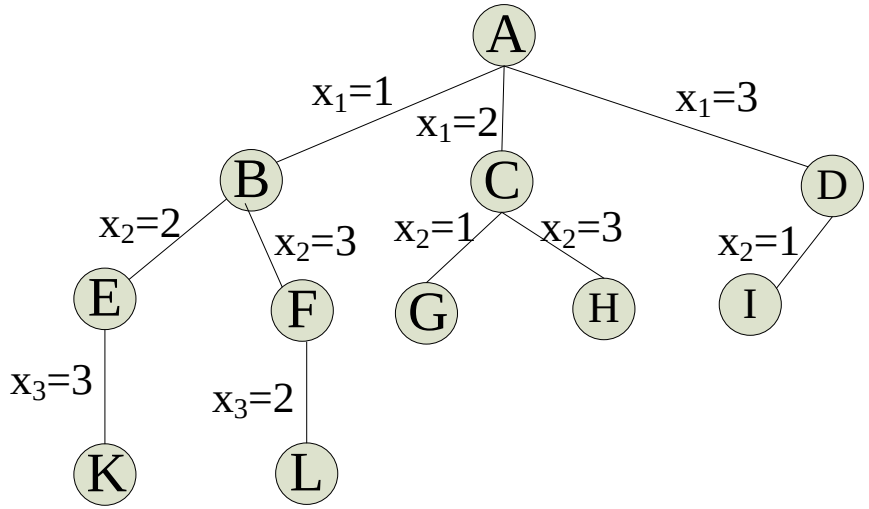
5.3 批处理作业调度

搜索过程

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



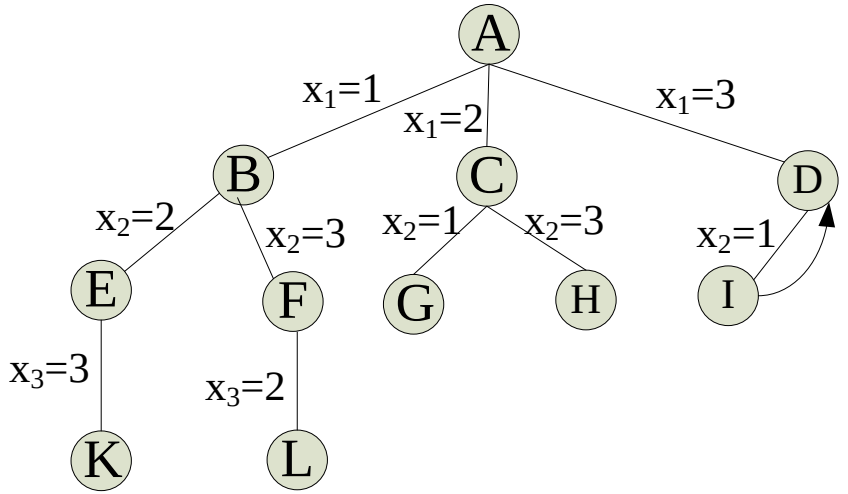
- 扩展结点A沿着x1=3的分支扩展，f=5，bestf=18，f<bestf，限界条件满足。
- 扩展结点D沿着x2=1的分支扩展，f=11，bestf=18，f<bestf，限界条件满足，扩展生成的结点I成为活结点。



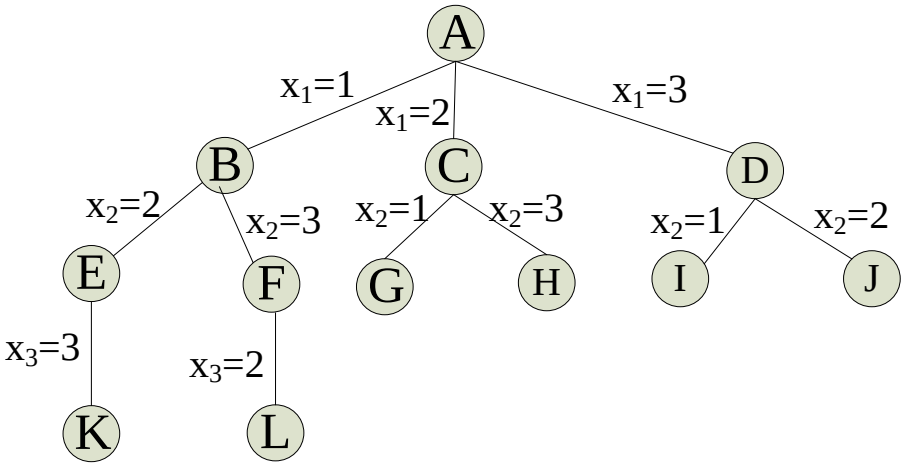
5.3 批处理作业调度

搜索过程

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



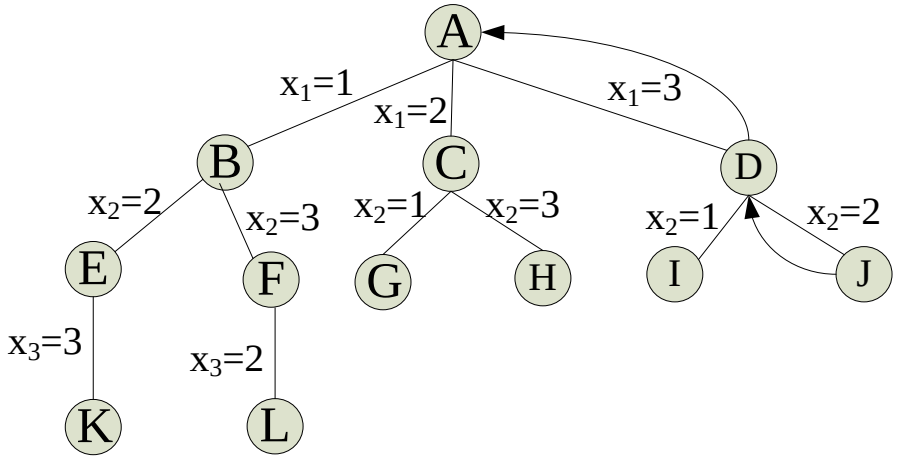
- 扩展结点I沿着x3=2的分支扩展，f=19，bestf=18，f>bestf，限界条件不满足，扩展生成的结点被剪掉，开始回溯到活结点D。
- 扩展结点D沿着x2=2的分支扩展，f=11，bestf=18，f<bestf，限界条件满足，扩展生成的结点J成为活结点。



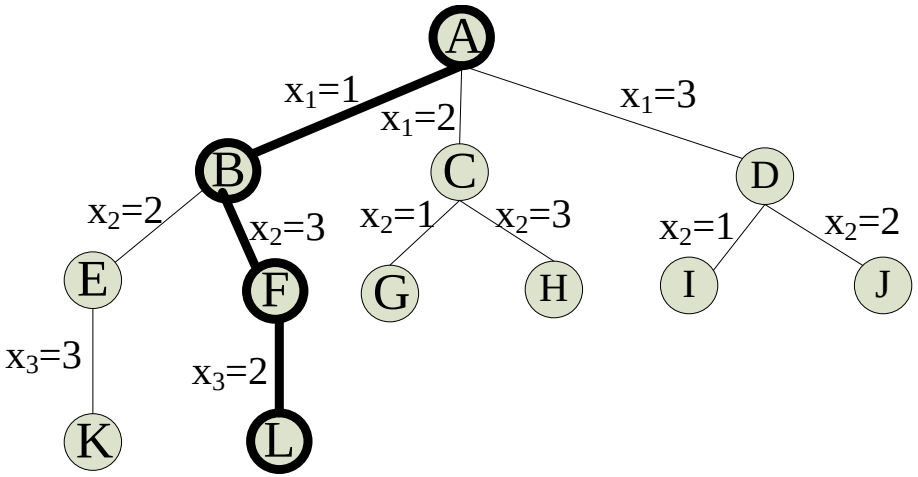
5.3 批处理作业调度

搜索过程

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



- 扩展结点J沿着x3=1的分支扩展，f=19，bestf=18，f>bestf，限界条件不满足，扩展生成的结点被剪掉，开始回溯到活结点D，结点D的两个分支搜索完毕，继续回溯到活结点A。
- 活结点A的三个分支也已搜索完毕，结点A变成死结点，搜索结束。至此，找到的最优的调度方案为从根结点A到叶子结点L的路径（1, 3, 2）。



5.4 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。

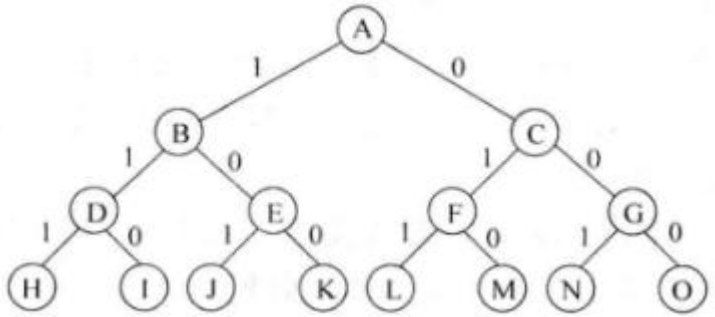
5.4 符号三角形问题

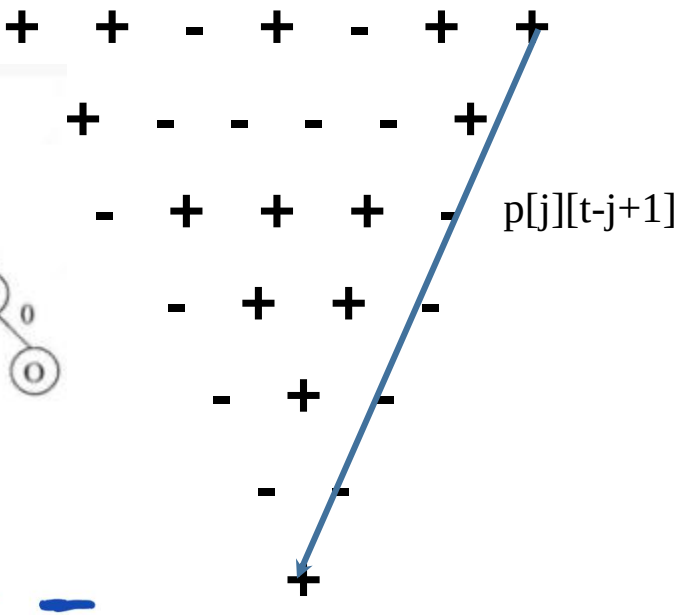
- 在符号三角形的第一行前i个符号x[1:i]确定后，就确定了一个由i(i+1)/2个符号组成的符号三角形。下一步确定x[i+1]的值后，只要在前面已确定的符号三角形的右边加一条边，就可以扩展为x[1:i+1]所相应的符号三角形。最终由x[1:n]所确定的符号三角形中包含“+”号个数与“-”个数同为n(n+1)/4。因此，当前符号三角形所包含的“+”个数与“-”个数均不超过n*(n+1)/4;
- 不断改变第一行每个符号，搜索符合条件的解，可以使用递归回溯，为了便于运算，设+ 为1，- 为0，这样可以使用同或运算符表示符号三角形的关系，++为+即0^0=0, --为+即1^1=0, +-为-即0^1=1, -+为-即1^0=1;
- 因为两种符号个数相同，可以对求解树进行剪枝， 当所有符号总数为奇数时无解，当某种符号超过总数一半时无解。

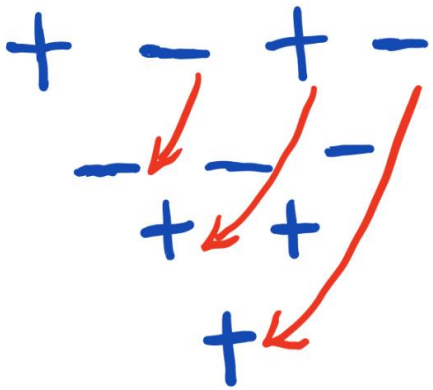
5.4 符号三角形问题

- 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。

- x[i] =1 时，符号三角形的第一行的第i个符号为+
- x[i] =0时，表示符号三角形的第一行的第i个符号位-共有i(i+1)/2个符号组成的符号三角形。
- 确定x[i+1]的值后，只要在前面确定的符号三角形的右边加一条边就扩展为x[1:i+1]所相应的符号三角形。
- 最后三角形中包含的“+”“-”的个数都为i(i+1)/4，因此搜索时，个数不能超过...若超直接可以剪去分枝。
- 当给定的n(n+1)/2为奇数时，也不符合三角形要求。



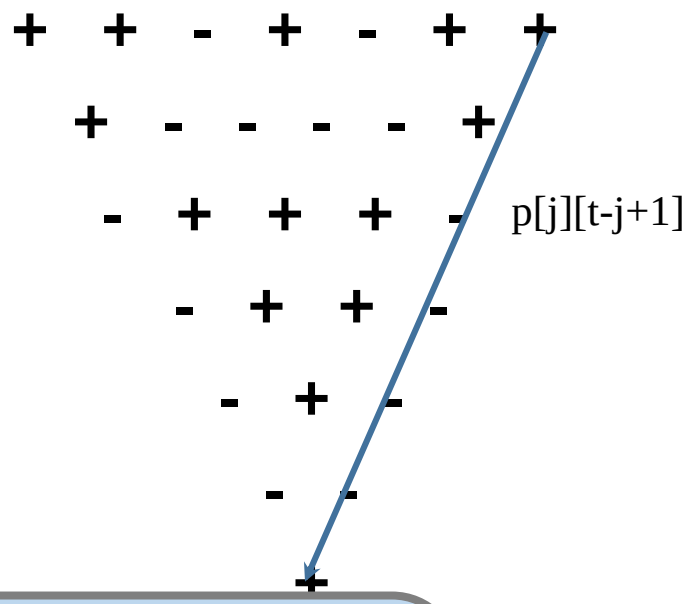




5.4 符号三角形问题

- 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。

```
void Triangle::Backtrack(int t){
  if ((count>half)||(t*(t-1)/2-count>half)) return;//任一符号统计超半数
  if (t>n) sum++; //符号填充完毕，满足条件，已经找到的符号三角形数量sum+1
    else
      for (int i=0;i<2;i++) {
        p[1][t]=i; //第一行第t个符号，加的是+或者-两种情况
        count+=i; //“+”号统计,因为“-"的值是0
        for (int j=2;j<=t;j++) {//当第一行符号>=2时，可以运算出下面边的某些符号，j可代表行号
          p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2]; //通过同或运算下行符号
          count+=p[j][t-j+1];
        }
      Backtrack(t+1); //在第一行增加下一个符号
      for (int j=2;j<=t;j++) //回溯，判断另一种符号情况 像是出栈一样，
                             恢复所有对counter的操作
        count-=p[j][t-j+1];
      count-=i;
    }
```

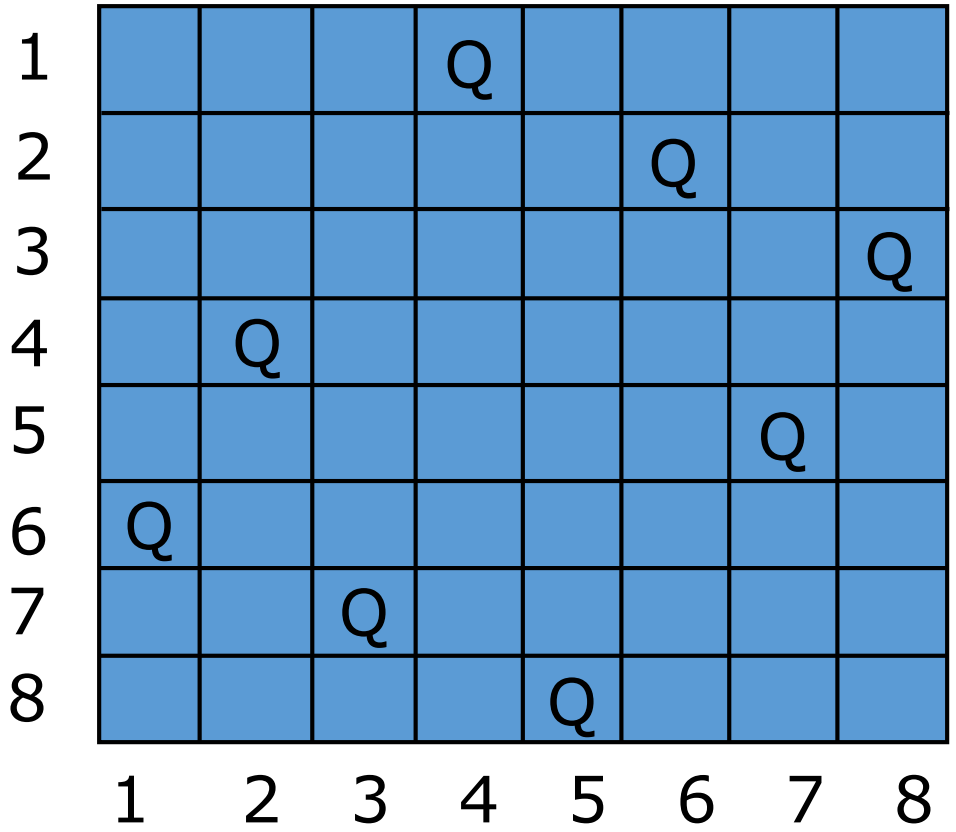


复杂度分析

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$O(n2^n)$。

5.5 n皇后问题

- 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。



5.5 n皇后问题

- 棋盘的每一行上可以而且必须摆放一个皇后，所以，n皇后问题的可能解用一个n元向量X=(x1, …, xi, …, xn)表示，其中，1≤i ≤ n并且1≤xi ≤n，即第i个皇后放在第i行第xi列上。
- 由于两个皇后不能位于同一列上，所以，解向量X必须满足约束条件1：xi≠xj 。

排列树!

5.5 n皇后问题

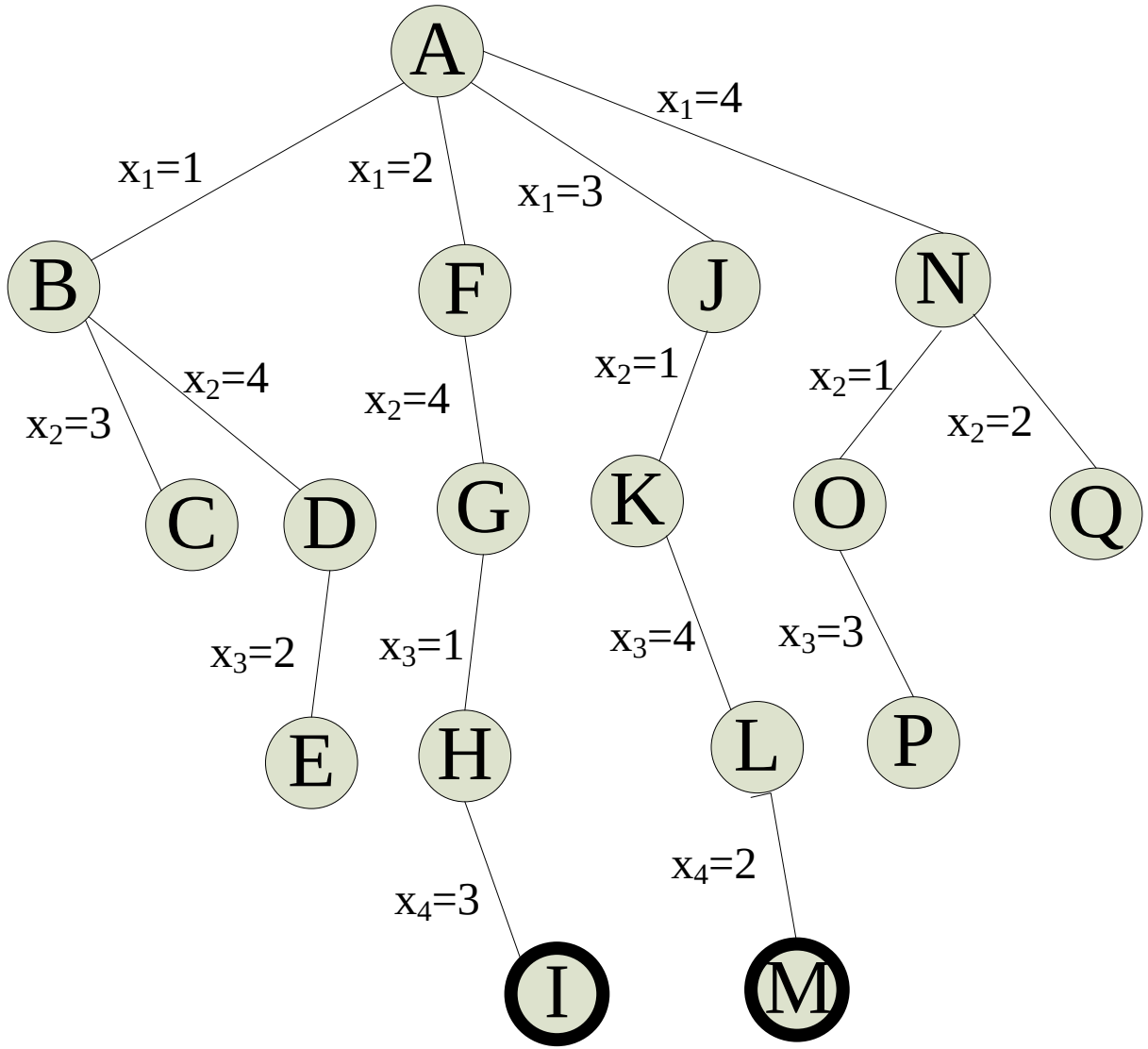
问题的形式化描述

- Si={1,2,…,n}，1≤i ≤ n，且xi≠xj(1≤i,j ≤n ,i≠j)。相应的隐式约束为：对任意1≤i,j ≤n ，当i≠j时，|i-j|≠|xi-xj| 。与此相对应的解空间大小为n!。
- 若两个皇后摆放的位置分别是(i, xi)和(j, xj)，在棋盘上斜率为+1,-1的斜线上，满足条件|i-j|≠|xi-xj|。
- 约束函数的设计：约束函数从隐式约束产生：对1⩽i,j ⩽n ，当i≠j时，要求xi≠xj且|i-j|≠|xi-xj|。

5.5 n皇后问题

- 定义问题的解空间
  - 解的形式：(x1, x2, … , xn)
  - xi的取值范围： xi=1,2, … ,n
- 组织解空间
  - 满n叉树，树的深度为n
- 搜索解空间
  - 约束条件：不同列且不处于同一正、反对角线上：|i-j|≠|xi-xj|
  - 限界条件：无
- 搜索过程（以4皇后问题为例）

5.5 n皇后问题



4皇后问题的搜索树

	√	○	
○			√
√			○
	○	√	

5.5 n皇后问题

- 解向量：(x1, x2, … , xn)
- 显约束：xi=1,2, … ,n
- 隐约束：

（1）不同列：xi≠xj

（2）不处于同一正、反对角线：|i-j|≠|xi-xj|

```
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;
}

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);
     }
 }
```

5.6 0-1背包问题

- 问题描述：给定n种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为W。一个物品要么全部装入背包，要么全部不装入背包，不允许部分装入。装入背包的物品的总重量不超过背包的容量。问应如何选择装入背包的物品，使得装入背包中的物品总价值最大?

- 定义问题的解空间

  解的形式(x1,x2,…,xn)，其中xi=0或1

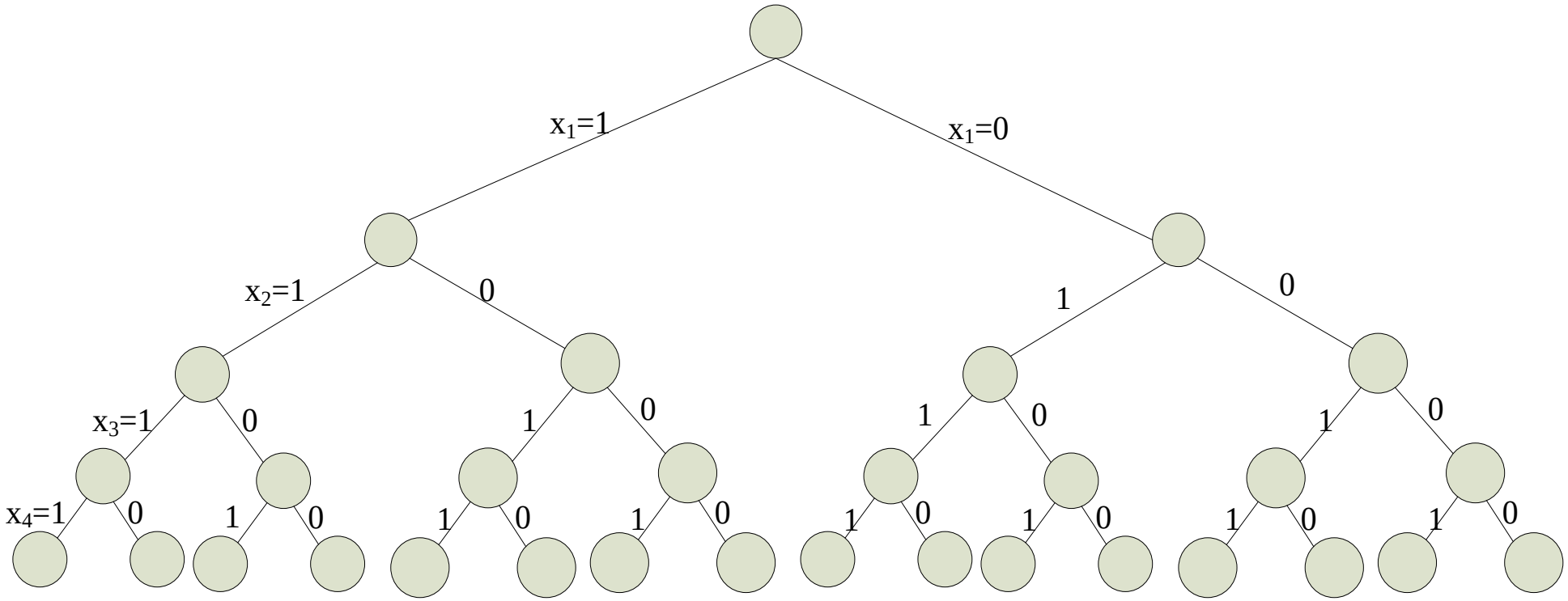
  xi=0:第i个物品不装入背包

  xi=1:第i个物品装入背包

5.6 0-1背包问题

同济大学 TONGJI UNIVERSITY

确定解空间



n=4时的解空间树

5.6 0-1背包问题

搜索解空间

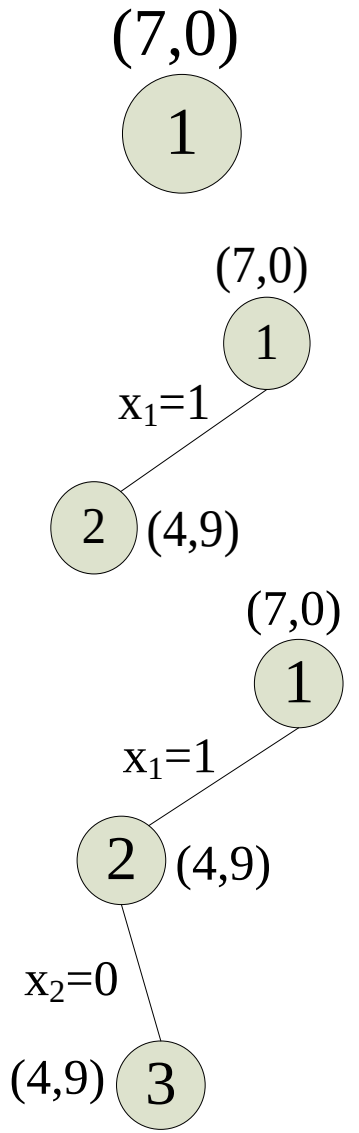
- 约束条件
  - wi≤c’(c’为背包的剩余容量）也就是：$\sum_{i=1}^{n} w_i x_i \le W$
- 限界条件：cp+rp>bestp 其中
  - cp:当前已装入背包的物品的总价值
  - rp:剩余不知道是否装入的物品的总价值
  - bestp： 当前已经找到的最优解的价值
- 搜索过程
  - 以n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4) 为例展示搜索过程

5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 根节点是活节点并且是当前的扩展节点
- 第一个物品的重量为3，3<7,满足约束条件
- 第二个物品的重量为5，5>(7-3)，不满足约束条件
- cp=9，rp=11，bestp=0，cp+rp>bestp, 满足限界条件。



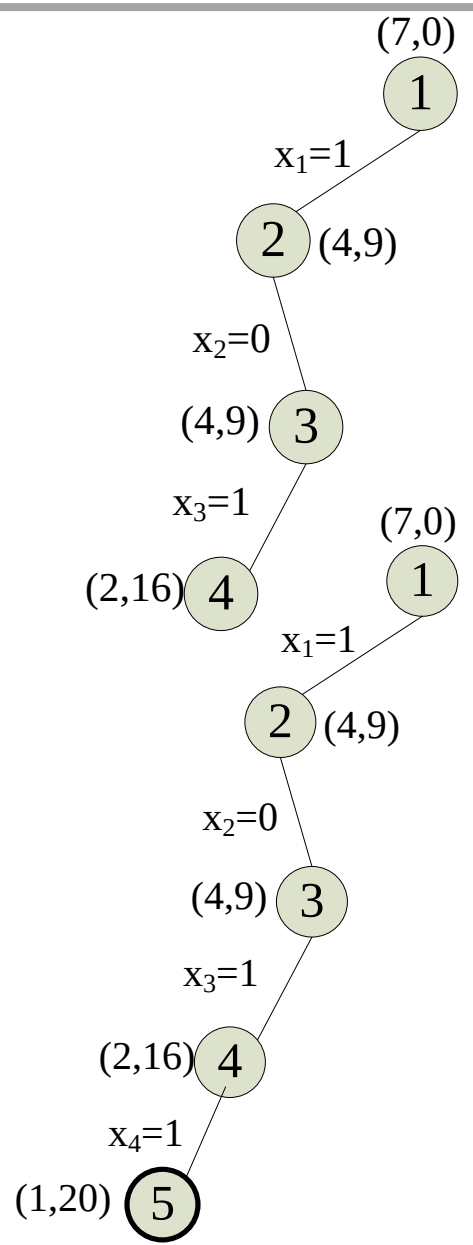
5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 第3个物品的重量是2，背包的剩余容量为4，满足约束条件
- 第4个物品的重量为1，背包的剩余容量为2，满足约束条件。

  现在已经到达叶子，找到当前最优解，bestp=9+7+4=20。

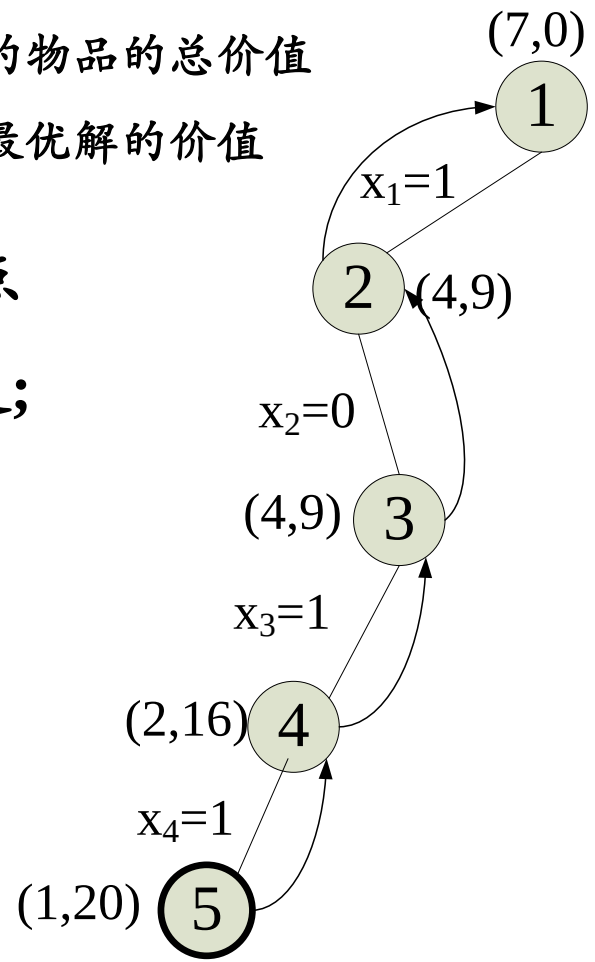


5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- cp:当前已装入背包的物品的总价值
- rp:剩余不知道是否装入的物品的总价值
- bestp：当前已经找到的最优解的价值



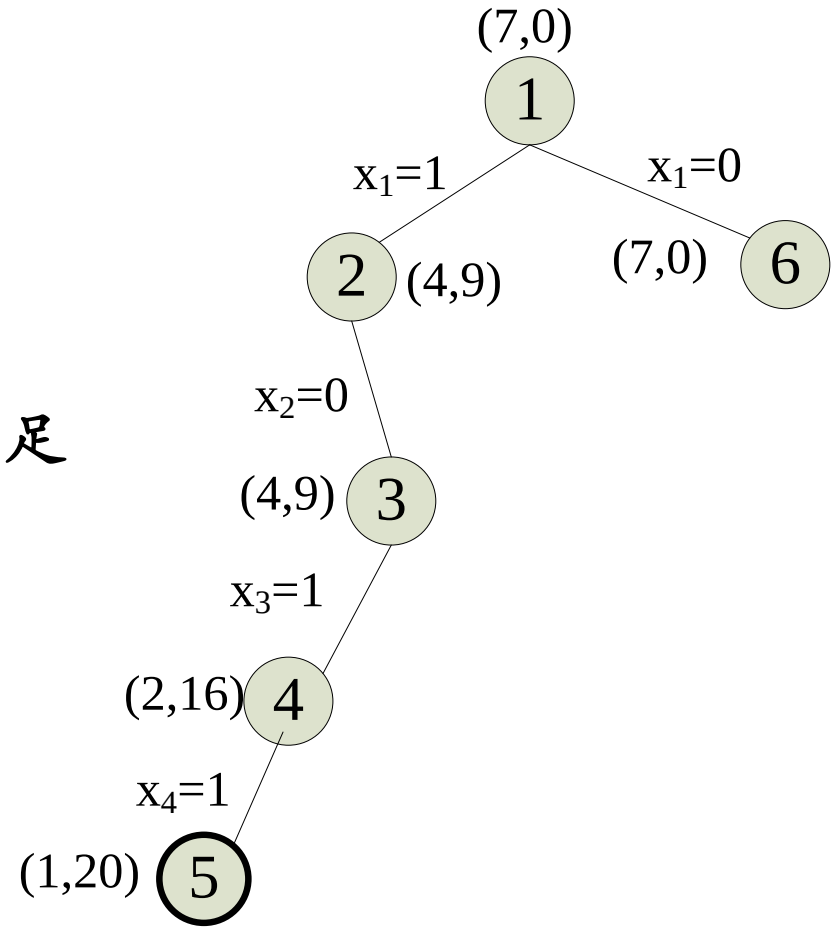
- 此时要回溯到离节点5最近的活节点4，节点4再次成为扩展节点
- 限界条件cp=16，rp=0，bestp=20 cp+rp<bestp，限界条件不满足;
- 回溯到最近的活节点3，节点3再次成为扩展节点
- 限界条件是否满足，cp=9，rp=4(右子树的可能值x3=0,x4=1)，bestp=20，cp+rp<bestp，限界条件不满足
- 回溯到最近的活节点2，继续回溯到节点1

5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 扩展节点1沿着右分支继续扩展，cp=0，rp=21（10+7+4），bestp=20，cp+rp>bestp，限界条件满足

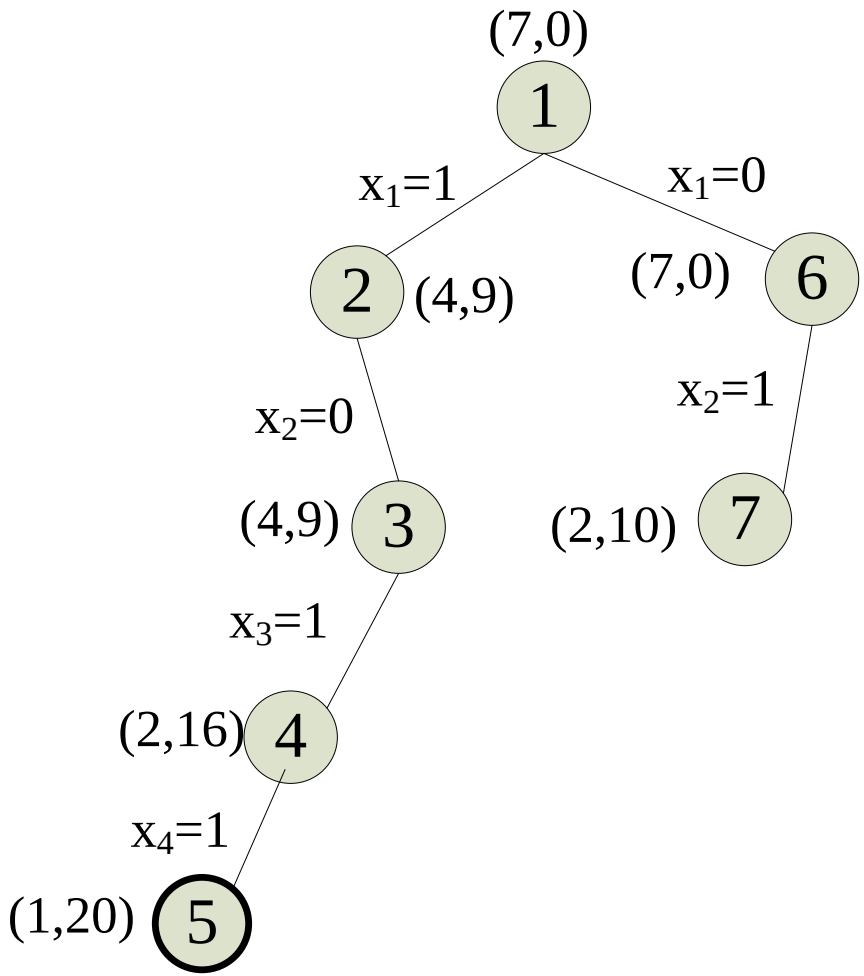


5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 扩展节点6沿着左分支继续扩展，第二个物品的重量为5，5<7，满足约束条件

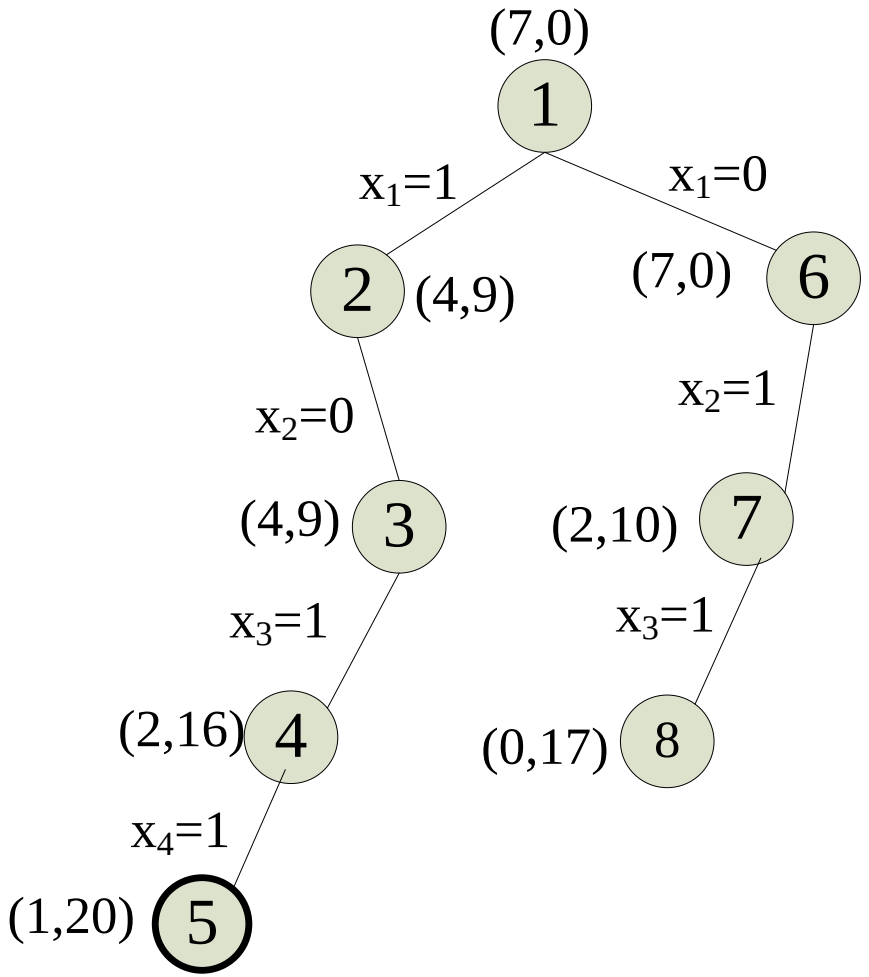


5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 扩展节点7沿着左分支继续扩展，判断约束条件，当前背包剩余容量为2，第3个物品的重量为2，满足约束条件

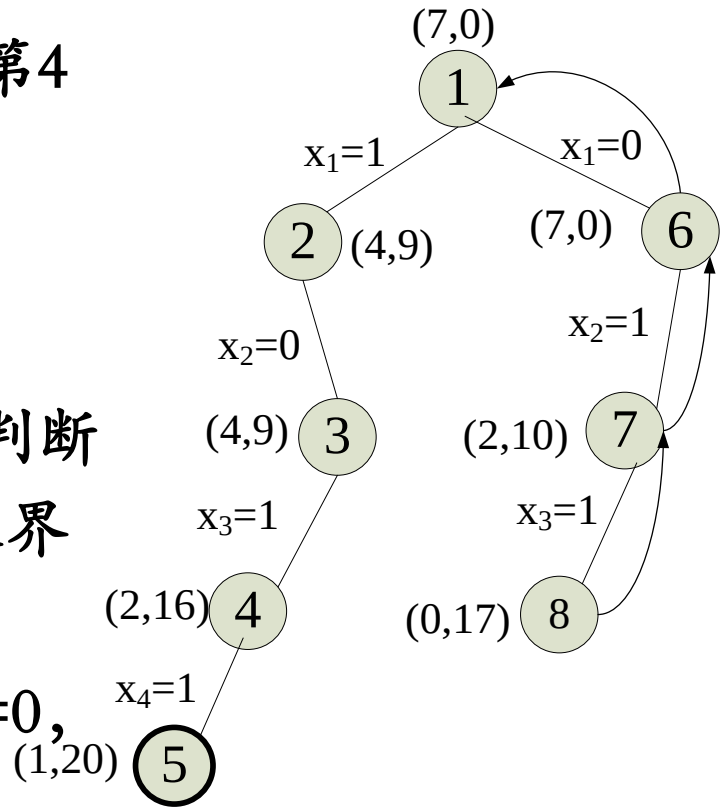


5.6 0-1背包问题

搜索解空间

n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 扩展节点8沿着左分支继续扩展，当前背包剩余容量为0，第4个物品的重量为1，0<1，不满足约束条件
- 沿着扩展节点8的右分支进行扩展，判断限界条件，cp=17，rp=0，bestp=20，cp+rp<bestp，不满足限界条件
- 回溯到最近的活节点7，扩展节点7沿着右分支继续扩展，判断限界条件，当前cp=10，rp=4，bestp=20，cp+rp<bestp，限界条件不满足
- 回溯到活节点6，扩展节点6沿着右分支继续扩展，当前cp=0，rp=11，bestp=20，cp+rp<bestp，限界条件不满足
- 回溯到活节点1，节点1又成为扩展节点。节点1的分支全部搜索完毕，节点1成为死节点，搜索结束。

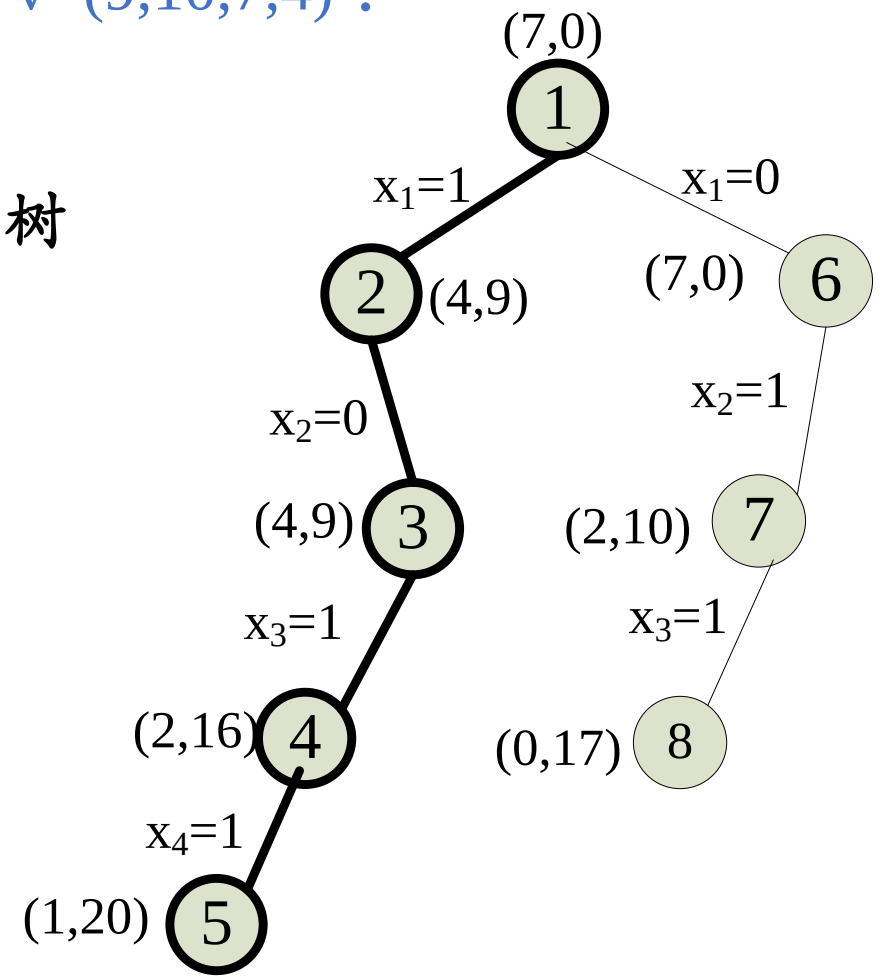


5.6 0-1背包问题

搜索解空间

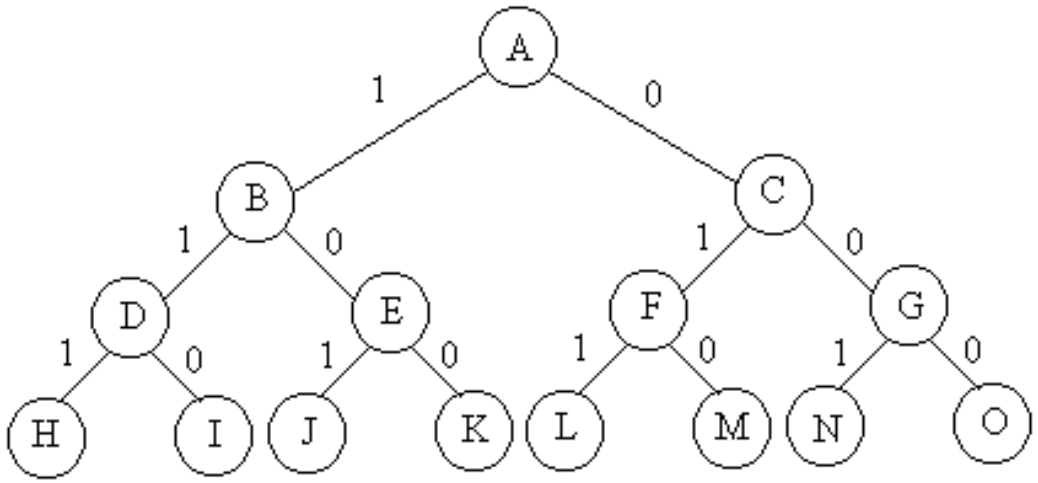
n=4，W=7，w=(3,5,2,1)，v=(9,10,7,4)：

- 示例0-1背包问题的搜索树



5.6 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：




```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i){// 计算上界rp
  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;
}
```

5.6 0-1背包问题

- 复杂性分析
  - 判断约束函数需O(1)，在最坏情况下有2^n-1个左孩子，约束函数耗时最坏为$O(2^n)$。
  - 计算上界限界函数需要O(n)时间，在最坏情况下有2^n-1个右孩子需要计算上界，限界函数耗时最坏为$O(n2^n)$。
  - 0-1背包问题的回溯算法所需的计算时间为$O(2^n)+O(n2^n)= O(n2^n)$。

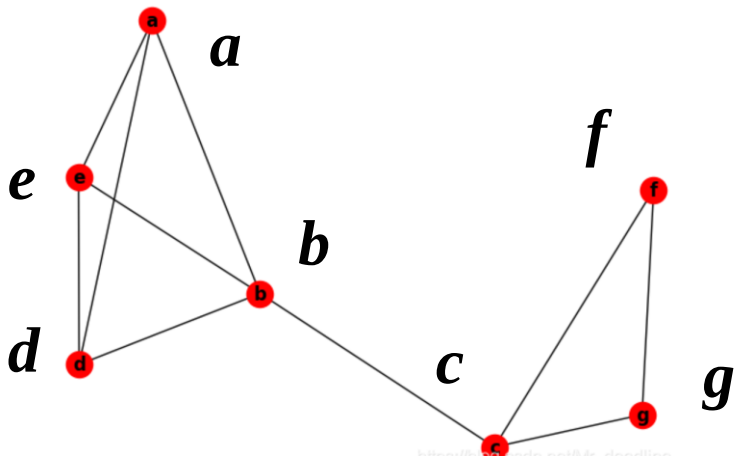
0-1背包问题动态规划解法的复杂度：$O(n2^n)$,
受控跳跃点改进后的方法复杂度$O(\min\{nc, 2^n\})$

```
void Knap::Backtrack(int t)
{
 if(t>n)//到达叶子节点
  {  for(int j=1;j<=n;j++)
          bestx[j]=x[j];
     bestp=cp;
    return;
  }
 if(cw+w[t]<=c) //搜索左子树
  {  x[t]=1;
     cw+=w[t];
     cp+=p[t];
     Backtrack(t+1);
     cw-=w[t];
     cp-=p[t];
    }
  if(Bound(t+1)>bestp)//搜索右子树
     { x[t]=0;
        Backtrack(t+1); }
     }
}
```

5.7 最大团问题

完全图，完全子图，最大完全子图

- 完全图：任意两点都恰有一条边相连的图(任意两点都相邻)。
- 完全子图：满足任意两点都恰有一条边相连的子图。
- 团：不包含在更大的完全子图中。
- 最大完全子图：所有完全子图中顶点数最大的团，即最大团，它的点集模最大。

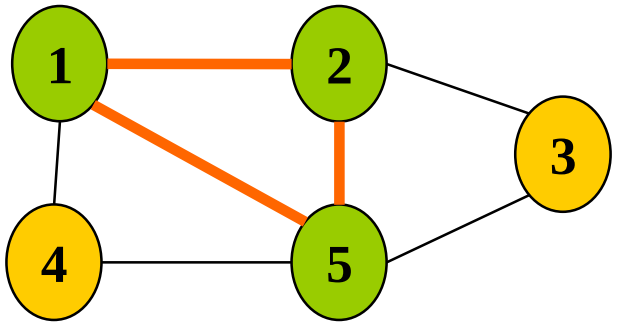


{'a','b','d'}，{'a','e'}，{'c','f','g'}等都是完全子图

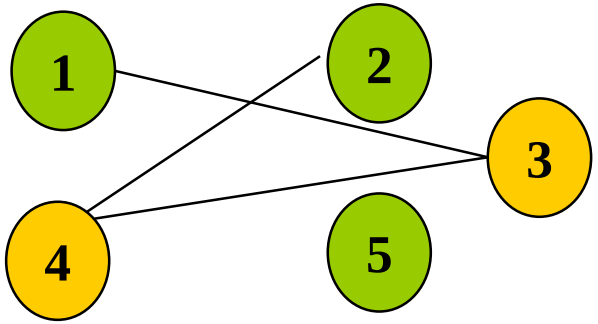
最大完全子图为{'a','b','d','e'}

5.7 最大团问题

- 给定无向图G=(V,E)。如果U⊆V，且对任意u,v∈U有(u,v)∈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。
- 如果U⊆V且对任意u,v∈U有(u,v)∉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。
- 对于任一无向图G=(V, E)其补图G′=(V′, E′)定义为：V′=V，且(u，v)∈E′当且仅当(u,v)∉E。



G

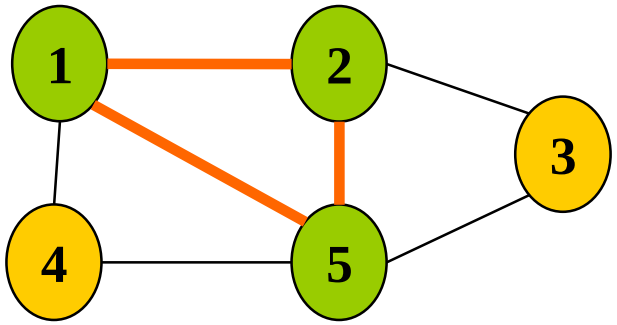


G′

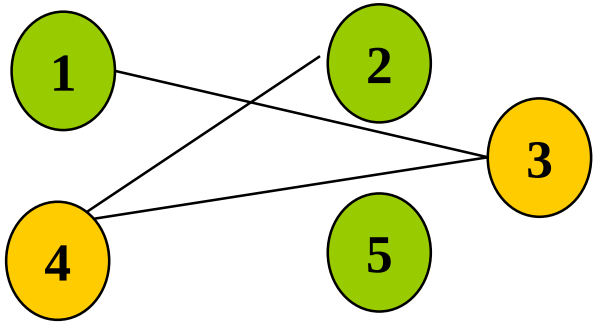
- U是G的最大团当且仅当U是G′的最大独立集

5.7 最大团问题

- 给定无向图G={V, E}，其中V={1,2,3,4,5}，E={(1,2), (1,4), (1,5),(2,3), (2,5), (3,5), (4,5)}。根据最大团定义，子集{1,2}是图G的一个大小为2的完全子图，但不是一个团，因为它包含于G的更大的完全子图{1,2,5}之中。{1,2,5}是G的一个最大团。{1,4,5}和{2,3,5}也是G的最大团。
- 右侧图是无向图G的补图G'。根据最大独立集定义，{2,4}是G的一个空子图，同时也是G的一个最大独立集。
- 虽然{1,2}也是G'的空子图，但它不是G'的独立集，因为它包含在G'的空子图{1,2,5}中。{1,2,5}是G'的最大独立集。{1,4,5}和{2,3,5}也是G'的最大独立集。



G



G'

5.7 最大团问题

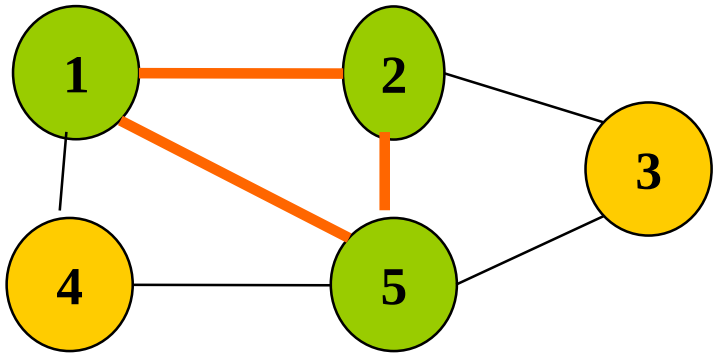
- 问题分析：最大团问题就是要求找出无向图G=(V，E)的n个顶点集合{1,2,3,…,n}的一个子集，这个子集中的任意两个顶点在无向图G中都有边相连，且包含顶点个数是最多的。
- 解空间：子集树
- 约束条件：顶点i到已选入的顶点集合中每一个顶点都有边相连。

判断新节点t是否与已有节点都相连

```
int Place（int t）{
    int OK=1;
    for (int j=1; j<t; j++)
        // 顶点t与顶点j不相连
        if (x[j] && a[t][j]==0){
         OK=0;
         break;
        }
    return OK;
}
```

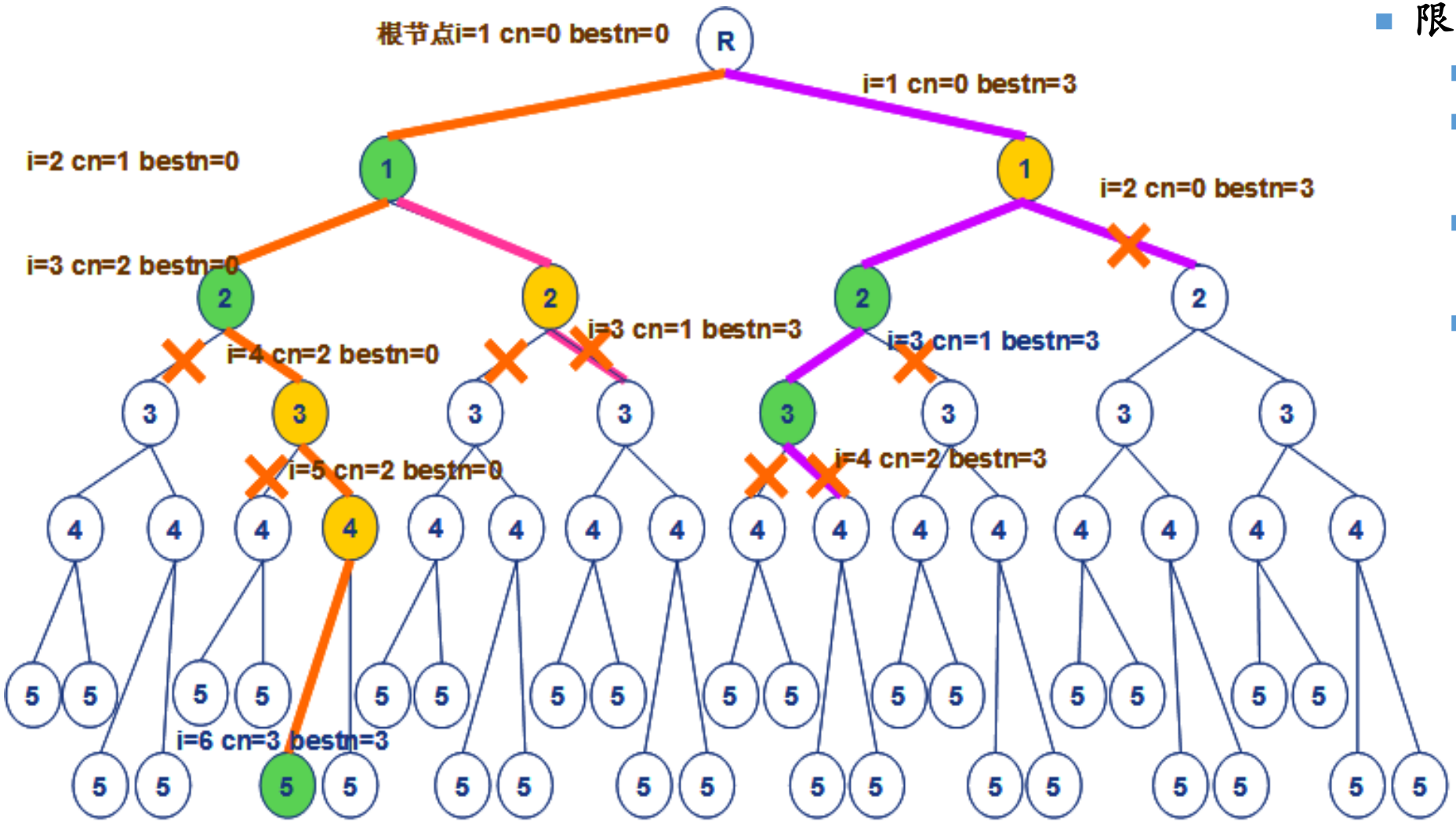
5.7 最大团问题

- 限界条件
  - cn+rn>bestn
  - cn：当前已包含在顶点集中的顶点个数
  - rn：剩余顶点个数(n-i)；
  - bestn：当前已找到的最优解包含的顶点个数

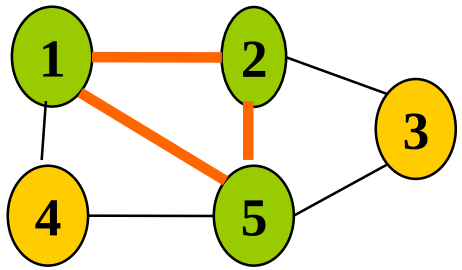


- 搜索过程

5.7 最大团问题



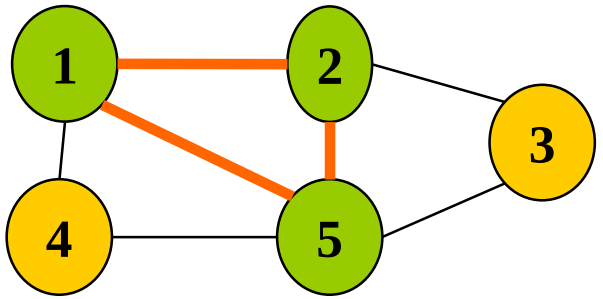
- 限界条件
  - cn+rn>bestn
  - cn：当前已包含在顶点集中的顶点个数
  - rn：剩余顶点个数(n-i)；
  - bestn：当前已找到的最优解包含的顶点个数



思考：为什么没有选出145,235作为最大团？

5.7 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。



复杂度分析

最大团问题的回溯算法backtrack所需的计算时间为$O(n2^n)$。

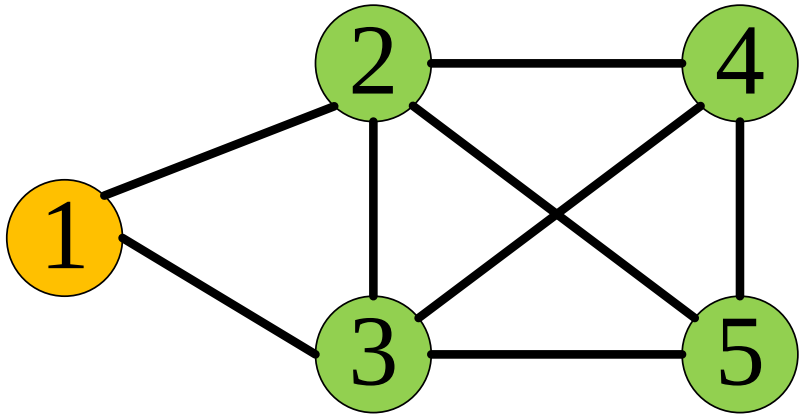
```
void Clique::Backtrack(int i){// 计算最大团
  if (i > n) {// 到达叶节点
    for (int j = 1; j <= n; j++) bestx[j] = x[j];
    bestn = cn;   return;}
  // 检查顶点 i 与当前团的连接
  int OK = 1;
  for (int j = 1; j < i; j++)
    if (x[j] && a[i][j] == 0) {
      // i与j不相连
      OK = 0;  break;}
  if (OK) {// 进入左子树
  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

5.7 最大团问题

- 算法分析
  - 判断约束函数需O(n)，在最坏情况下有2^n-1个左孩子，耗时最坏为$O(n2^n)$。
  - 判断限界函数需要O(1)时间，在最坏情况下有2^n-1个右孩子节点需要判断限界函数，耗时最坏为$O(2^n)$。
  - 最大团问题的回溯算法所需的计算时间为$O(2^n)+O(n2^n)=O(n2^n)$。

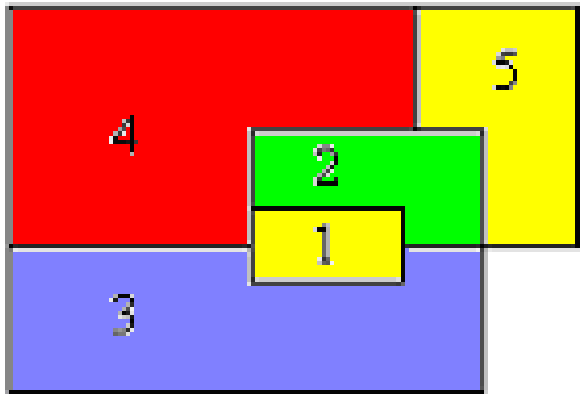
5.7 最大团问题

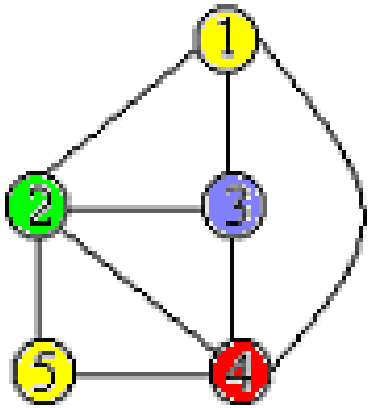
- 搜索过程?



5.8 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。是否有一种着色法使G中每条边的2个顶点着不同颜色。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。





5.8 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。

- 整数m为该图的着色数。求一个图的色数m的问题称为图的m可着色最优化问题。

5.8 图的m着色问题

- 设无向图G=(V, E)采用如下邻接矩阵表示：

$$a[i][j]=\begin{cases}1 & if\ (i,j)\in E\\ 0 & others\end{cases}$$

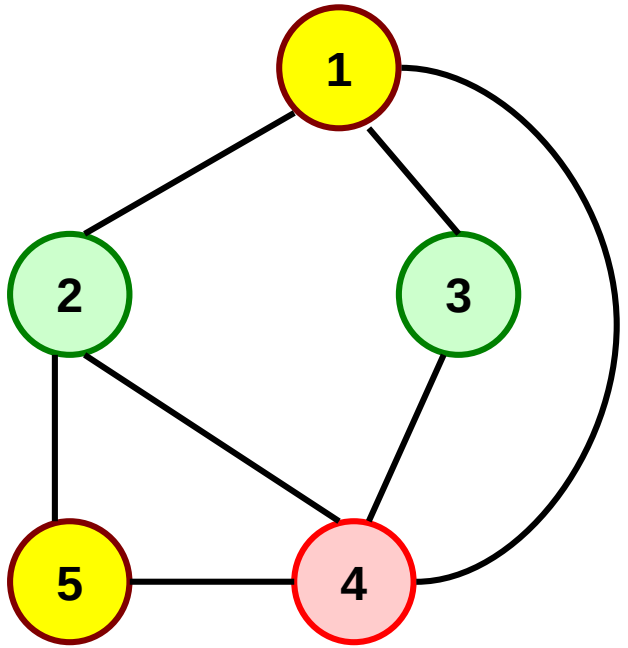
- 由于用m种颜色为无向图G=(V, E)着色，其中，V的顶点个数为n，可以用一个n元组$(x_1, x_2, \cdots, x_n)$来描述图的一种可能着色，其中，$x_i \in \{1, 2, \cdots, m\}(1 \leqslant i \leqslant n)$表示赋予顶点i的颜色，这是显式约束。$x_i=0$表示没有可用的颜色。因此解空间大小为$m^n$。

- 约束函数从隐式约束产生：对所有i和j $(1 \leqslant i,j \leqslant n, i \neq j)$，若a[i][j]=1，则$x_i \neq x_j$。

5.8 图的m着色问题

举例

- 例如， 5元组(1, 2, 2, 3, 1)表示对具有5个顶点的无向图的一种着色，顶点1着颜色1，顶点2着颜色2，顶点3着颜色2，顶点4着颜色3，顶点5着颜色1。
- 如果在n元组中，所有相邻顶点都不着相同颜色，就称此n元组为可行解，否则为无效解。



5.8 图的m着色问题

回溯法求解

- 回溯法求解图着色问题，首先把所有顶点的颜色初始化为0，然后依次为每个顶点着色。
- 在图着色问题的解空间树中：
  - 如果从根节点到当前节点对应一个部分解，也就是所有的颜色指派都没有冲突，则在当前节点处选择第一棵子树继续搜索，也就是为下一个顶点着颜色1，
  - 否则，对当前子树的兄弟子树继续搜索，也就是为当前顶点着下一个颜色，
  - 如果所有m种颜色都已尝试过并且都发生冲突，则回溯到当前节点的父节点处，上一个顶点的颜色被改变，依此类推。

5.8 图的m着色问题

- 解向量：(x1, x2, … , xn)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。

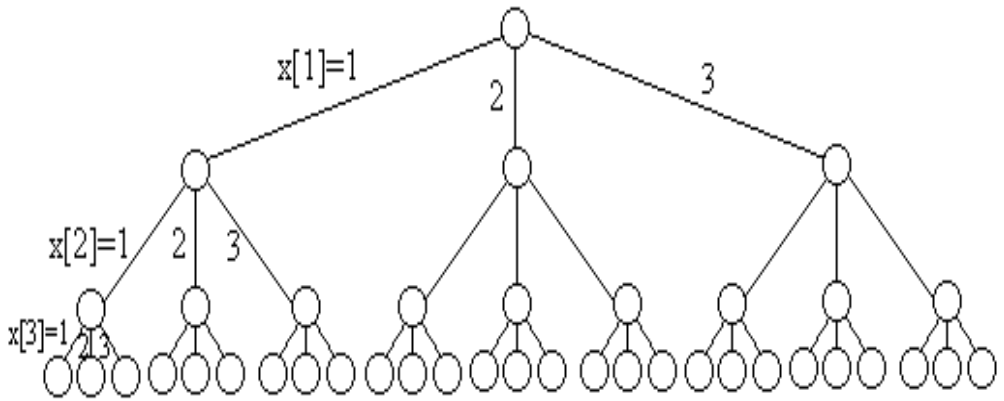
复杂度分析

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

对于每一个内节点，在最坏情况下，用ok检查当前扩展节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

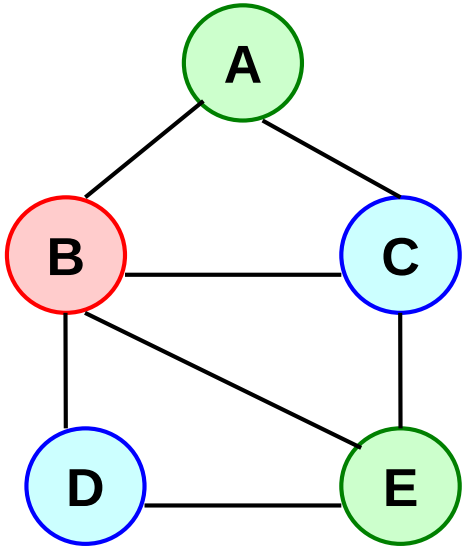
$$\sum_{i=0}^{n-1} m^i(mn) = nm(m^n-1)/(m-1) = O(nm^n)$$

```
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);
       }
}
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;
}
```



5.8 图的m着色问题

- 搜索过程?



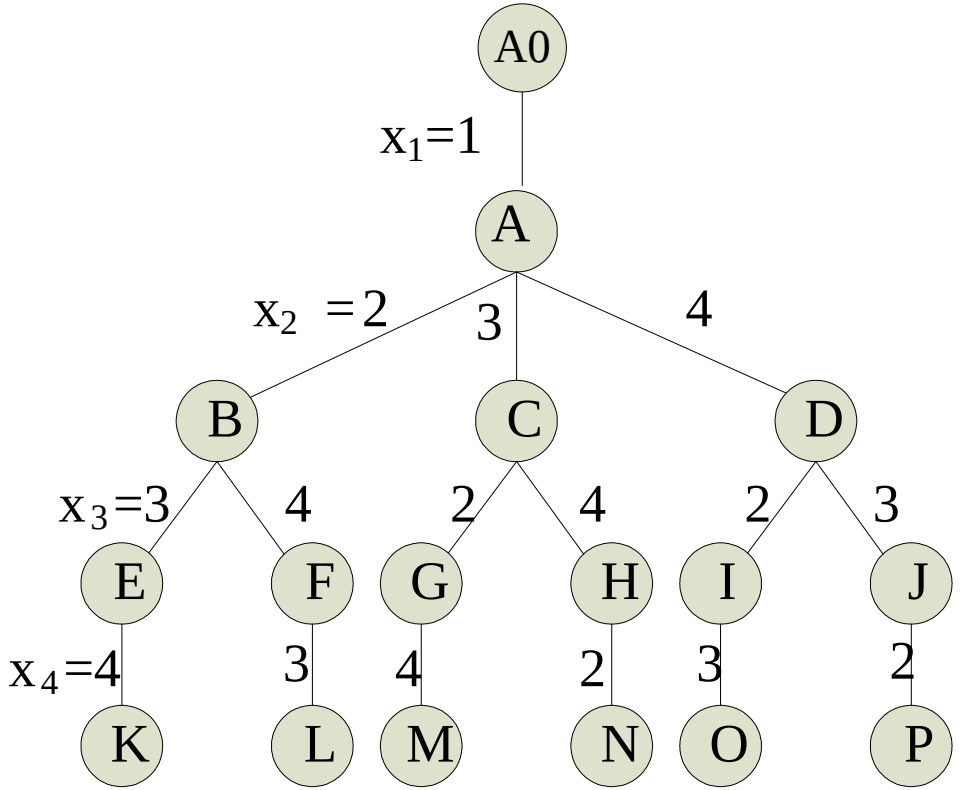
5.9 旅行售货员问题

- 问题描述
  - 设有n个城市组成的交通图，一个售货员从驻地城市出发，到其它城市各一次去推销货物，最后回到驻地城市。假定任意两个城市i，j之间的距离dij (dij=dji)是已知的，问应该怎样选择一条最短（或总旅费最小）的路线?

- 定义问题的解空间
  - 解的形式（x1,x2,…,xn）
  - 令n个城市组成的集合为S={1,2,…,n}，则x1=1，xi∈S-{x1,x2, …,xi-1}，i=2, …,n。

5.9 旅行售货员问题

解空间排列树

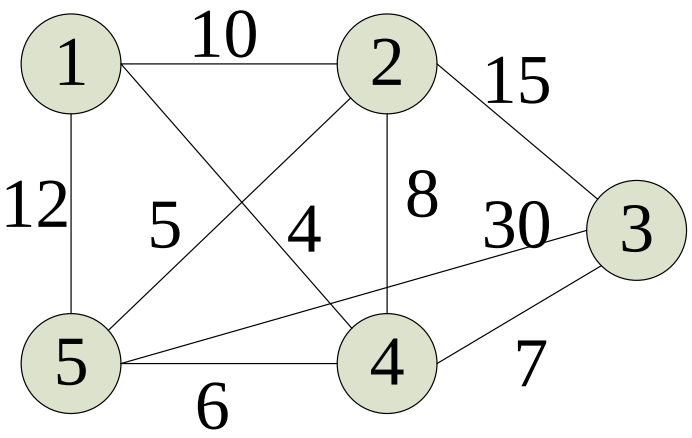


n=4的旅行售货员问题的解空间树

5.9 旅行售货员问题

搜索解空间

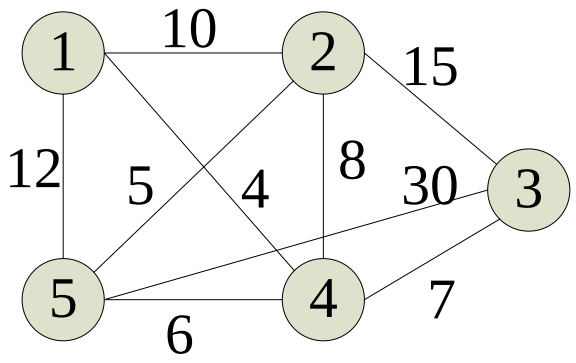
- 设置约束条件；
  - 用二维数组a[ ][ ]存储无向带权图的邻接矩阵，如果a[i][j]≠∞表示城市i和城市j有边相连，能走通。
- 设置限界条件
  - cc<bestc，cc的初始值为0，bestc的初始值为＋∞。
  - cc：当前已走过的城市所用的路径长度
  - bestc：表示当前找到的最短路径的路径长度
- 搜索过程：以右图为例说明

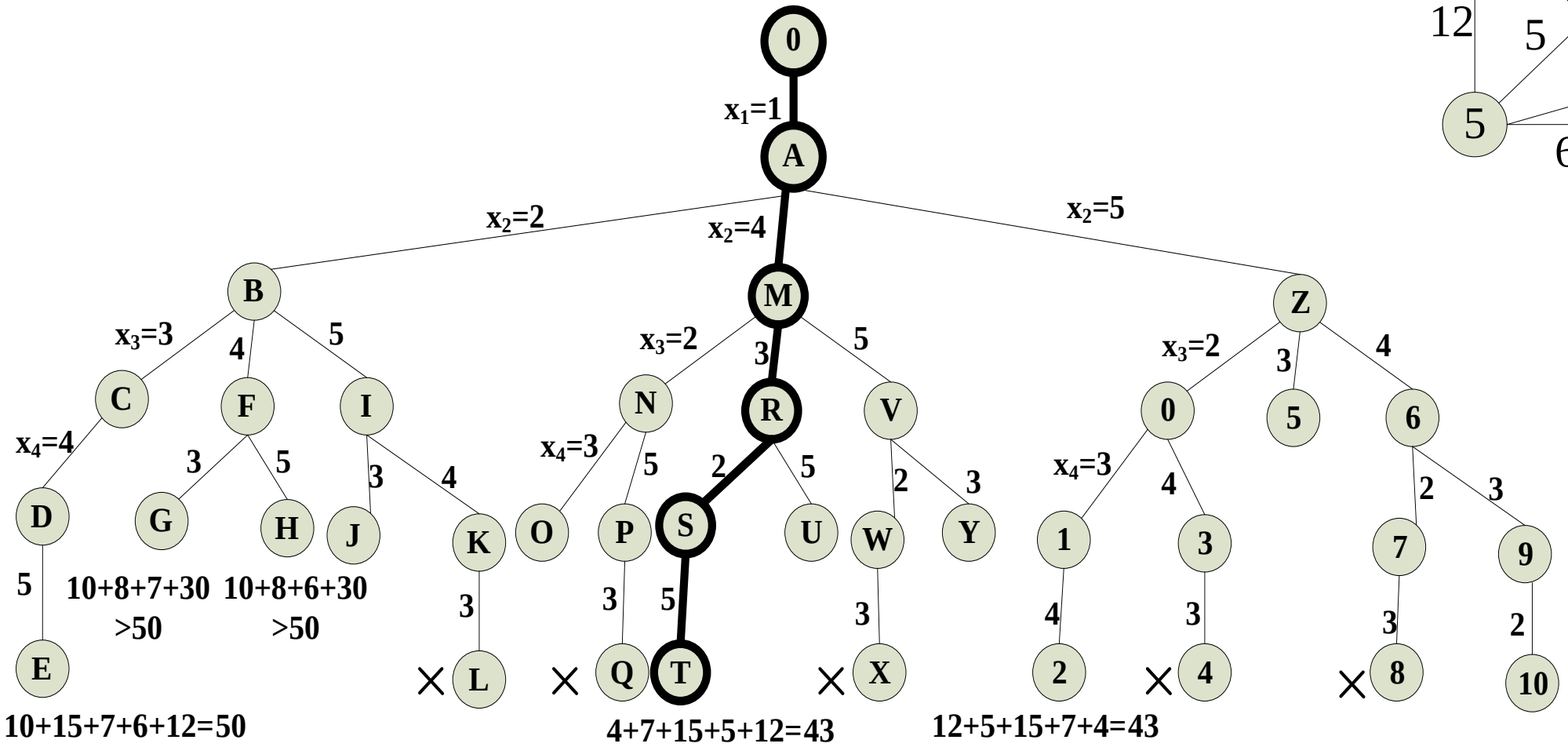


5.9 旅行售货员问题

同济大学 TONGJI UNIVERSITY

搜索树





5.9 旅行售货员问题

- 解空间：排列树

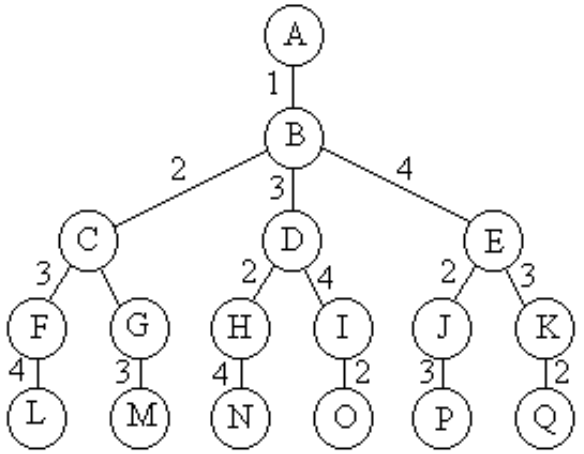
```
template<class Type>
void Traveling<Type>::Backtrack(int i){
    if (i == n) {
        if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
        (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge)) {
            for (int j = 1; j <= n; j++) bestx[j] = x[j];
            bestc = cc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

5.9 旅行售货员问题

- 算法分析
  - 判断限界函数需要O(1)时间，在最坏情况下有1+(n-1)+ (n-1)(n-2)+…+(n-1)(n-2)…2*1 ≤n（n-1）!个节点需要判断限界函数，故耗时O(n!)；
  - 在叶子节点处记录当前最优解需要耗时O(n)，在最坏情况下会搜索到每一个叶子节点，叶子节点有（n-1）!个，故耗时为O(n!)。
  - 旅行售货员问题的回溯算法所需的计算时间为O(n!)+O(n!)= O(n!)。

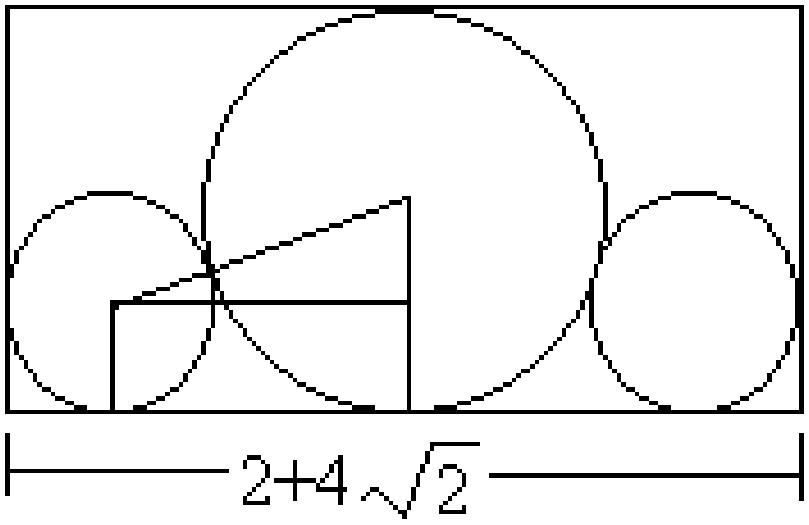
n-1

(n-1)*(n-2)



5.10 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 $2+4\sqrt{2}$



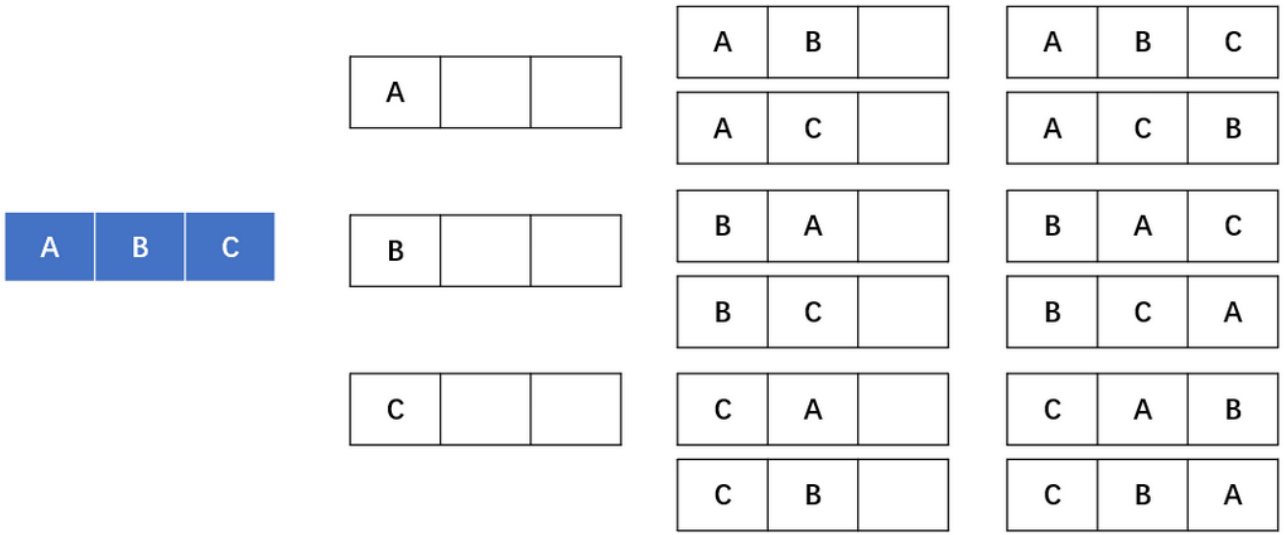
5.10 圆排列问题

- 圆排列问题的解空间是一棵排列树，我们用回溯法在整个排列数中搜索最优解。
- 首先，我们可以把这个问题分解为以下几步：
  - 找出所有圆的排列
  - 计算出每一个排列的长度
  - 选择最小的长度

5.10 圆排列问题

（一）全排列

- 给你一组半径，每个代表一个圆，找出所有的排列方式。这其实就是全排列问题。
- 我们用递归的方法找出全部的排列。
- 比如：给你ABC



5.10 圆排列问题

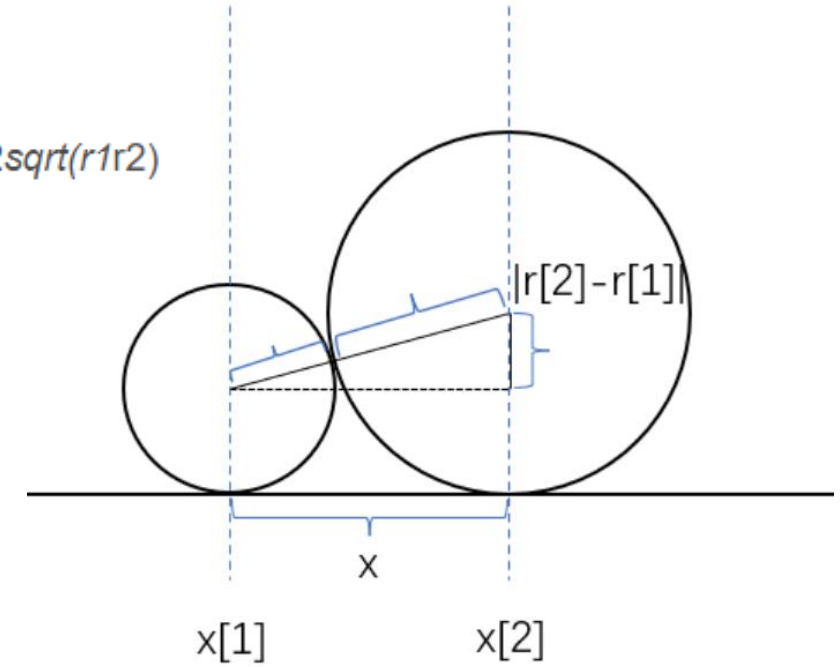
（二）计算一种排列的长度

- 找出了所有的排列，那么只要我们能够计算出一种排列的长度
- 对所有的排列 比较一个最小的即可得到答案
- 那么假设前一个圆的圆心横坐标为x1，后一个是x2，半径是r1和r2
- 那么根据勾股定理即可计算出x：推导出x，

  x2=x1+x

$x^2 = \mathrm{sqrt}((r_1 + r_2)^2 - (r_1 - r_2)^2)$推导出x = 2*sqrt(r1r2)*

- 那么有了这个公式，我们假定第一个圆形的圆心横坐标为0，知道了第一个就能算出第二个，以此类推所有的圆心横坐标就都算出来了。



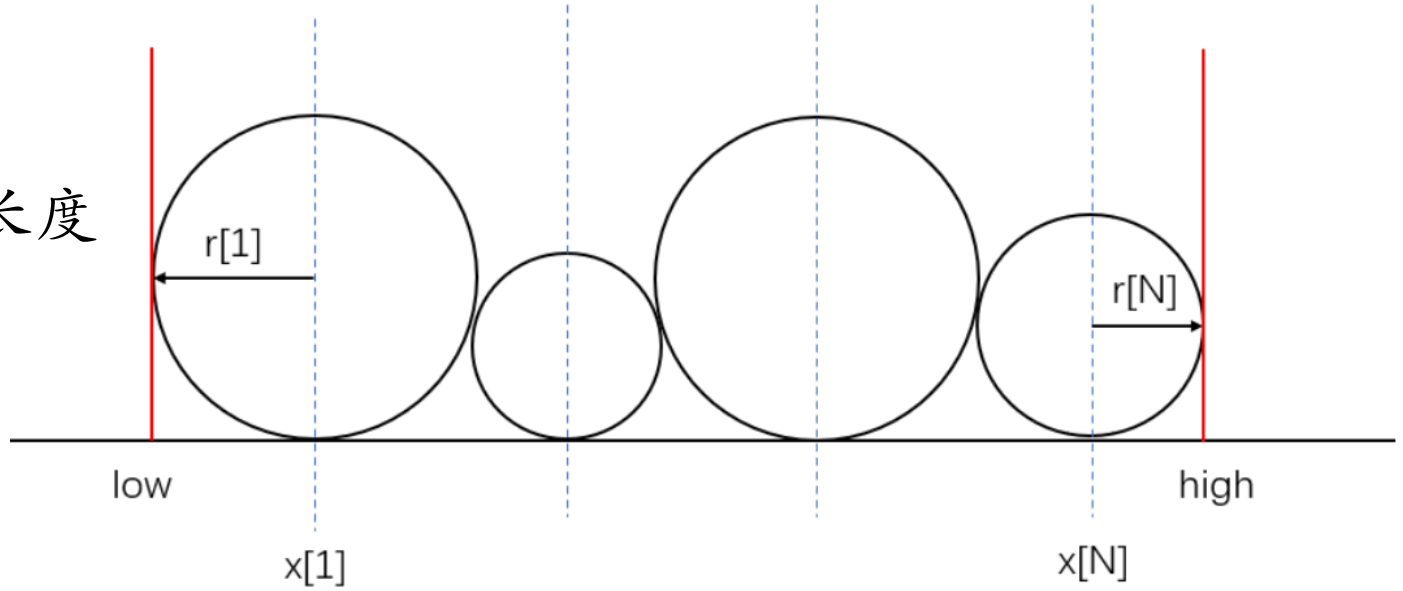
5.10 圆排列问题

（二）计算一种排列的长度

- 有了所有的横坐标，还有该排列所有圆的半径
- 排列的长度：
  - 那么第一个圆的横坐标加上第一个圆的半径就是该排列的最左边low

    最后一个圆的横坐标加上最后一个圆的半径就是该排列的最右边high

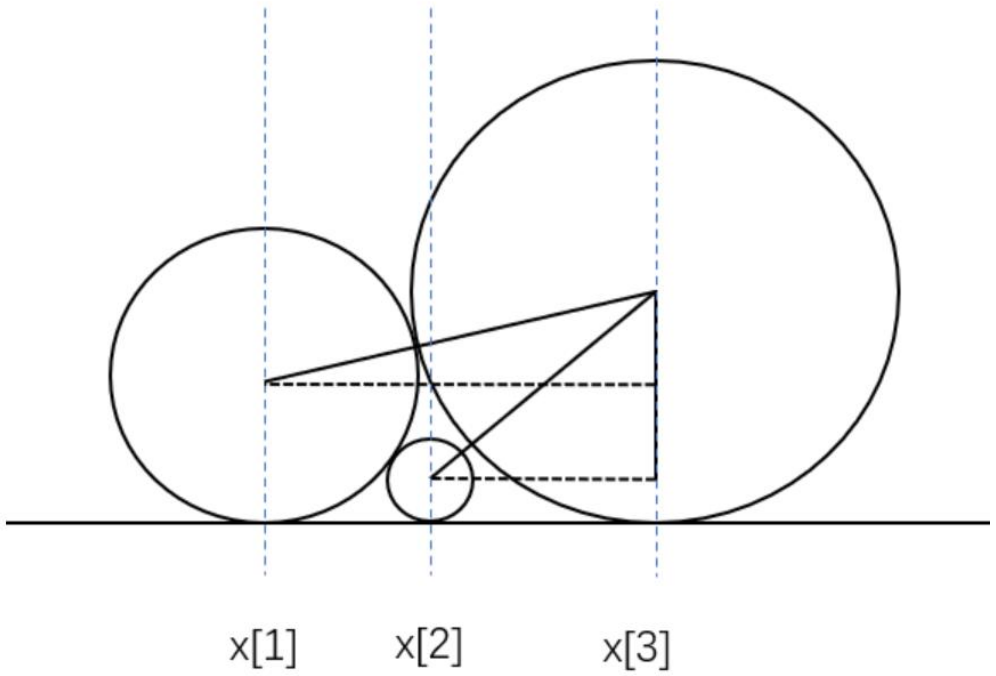
high-low即可得到长度



5.10 圆排列问题

（三）完善算法

- 前面的想法其实还存在一些问题。
- 计算横坐标x的时候，我们使用的公式有一个前提，就是这个圆必须和前一个圆相切，那么实际情况中是相切的嘛?



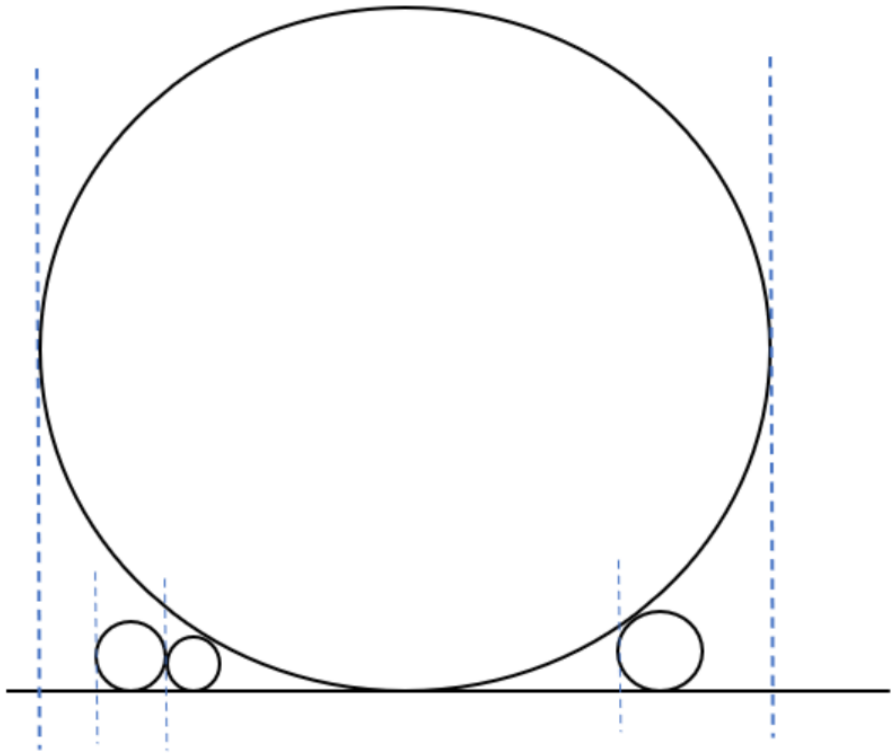
如图，是不一定的。当前圆并不一定与前一个圆相切，有可能和前一个相切，也有可能和前面任意一个圆相切。所以，我们其实需要和前面所有的圆都进行一次计算。

如图，这个圆和前面相邻的那个并不相切，所以你用我们说的那个公式计算的话，会计算出来偏小的结果。所以我们只需要把当前圆与前面所有的圆依次计算，保留最大的值，就是正确的值。

5.10 圆排列问题

（三）完善算法

- 我们计算左右边界的方法是对的吗?
- 左边界并不一定是第一个圆的左边



如图，左边界并不是第一个圆的左边，那么怎么算呢

其实我们有了每个圆的圆心横坐标和半径了，我们只要把每个圆的左边界算出来，取最小的就是整个排列的左边界了

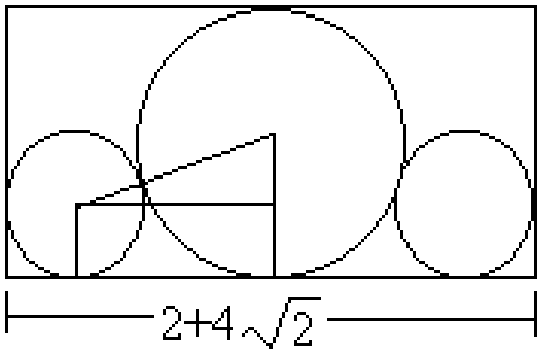
同理，把每个圆的右边界算出来，取最大的，就是整个排列的右边界了

5.10 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]);
            }
}
```

```
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;
}
```

```
 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;
 }
```



5.10 圆排列问题

上述算法尚有许多改进的余地。例如，像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。

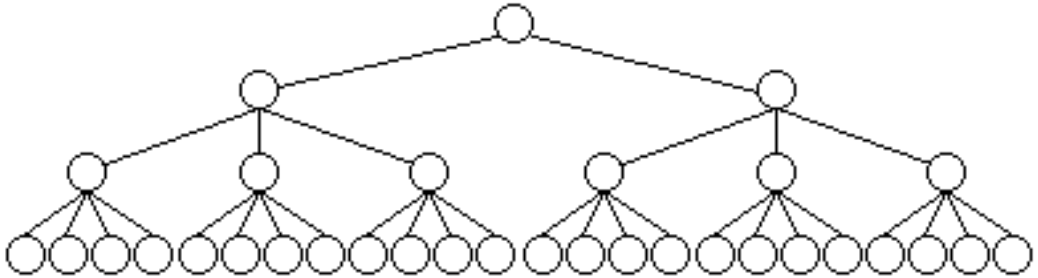
5.12 回溯法效率分析

- 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节点数。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节点数与约束函数计算量之间的折衷。

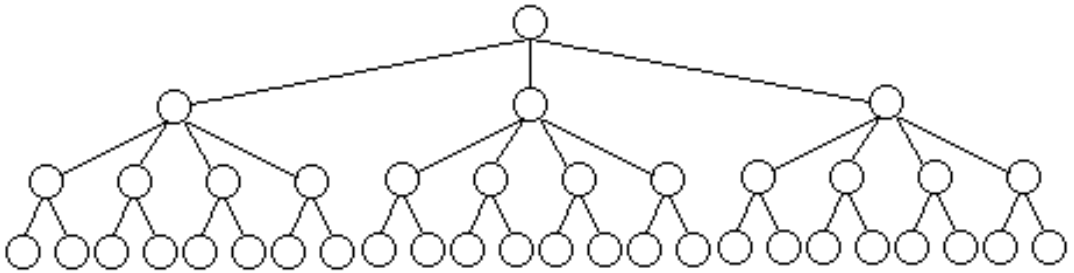
5.12 回溯法效率分析

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。



(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。前者的效果比后者好。