武漢大學 WUHAN UNIVERSITY

回溯法

林 海

Lin.hai@whu.edu.cn

回溯法

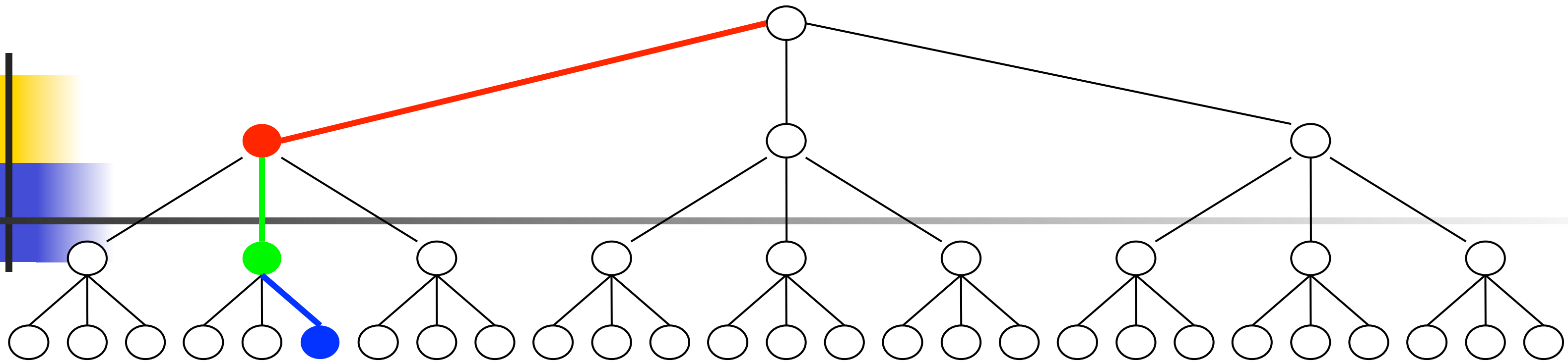
- 组织搜索的一种技术：有组织的穷尽搜索，避免搜索所有可能性
- 适用于那些有潜在大量解，但有限个数的解已经检查过的问题。

引例

- 图的3着色问题
- 给定一个无向图G=(V,E)，及3种颜色{1,2,3}，现要为图的每个顶点着色。每顶点只能着一种颜色，并且要求相邻的顶点具有不同的颜色。

图的3着色问题

- 一个具有n个顶点的图，可用一个n维的向量$(c_1,c_2,\ldots,c_n)$表示一种着色方案，$c_i \in \{1,2,3\}$, $i=1,2,\ldots,n$，共有3^n种可能的着色，可用一棵完全的3叉树表示。
- 下图为有3个顶点的所有可能的着色搜索树，从根到叶节点的每条路径表示一种着色方案。

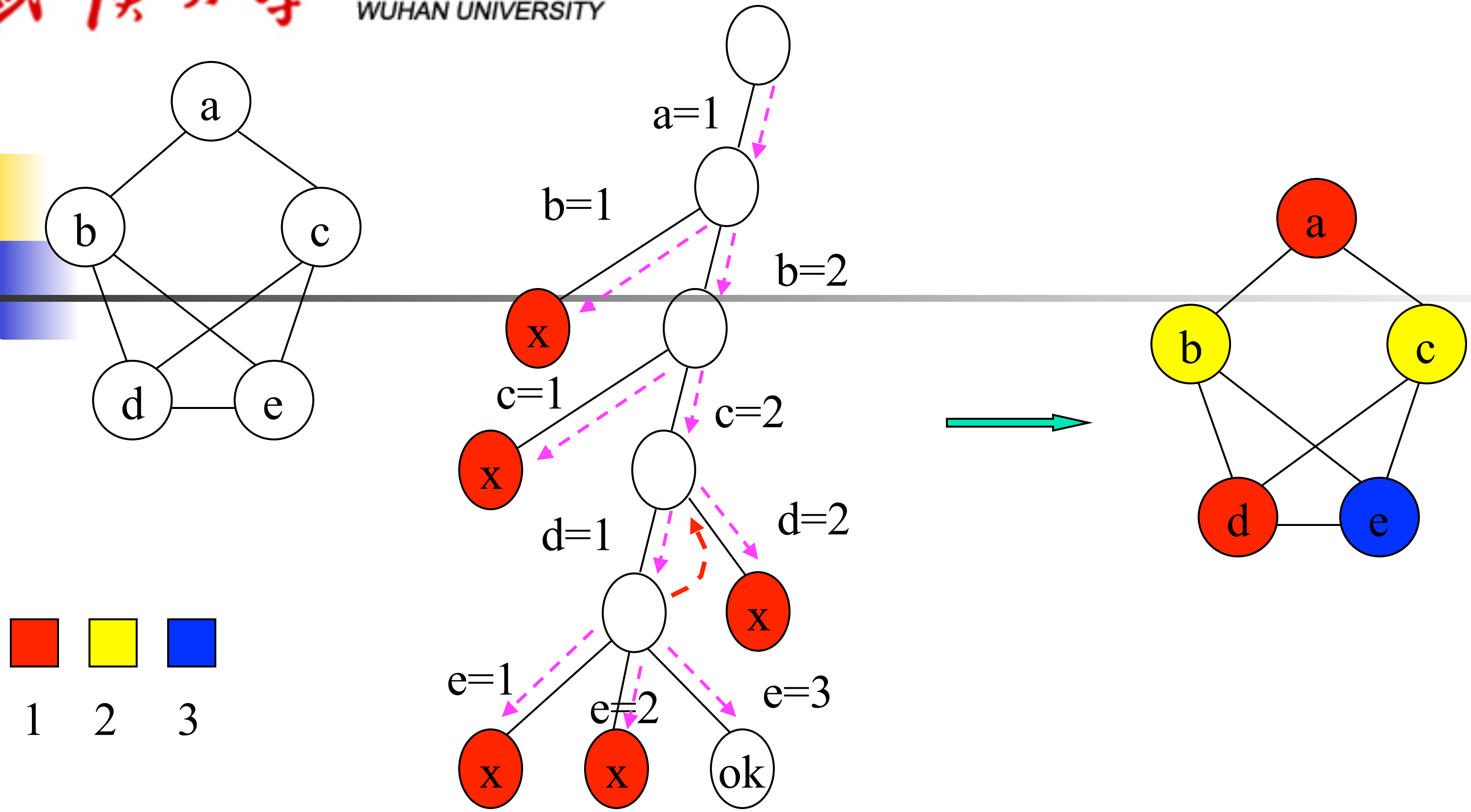


几个概念

- 问题的解向量：问题的解能够表示成一个n维向量$(x_1,x_2,,\ldots,x_n)$的形式。
- 显式约束：对分量x_i的取值限定。
- 隐式约束：为满足问题的解而对**不同分量**之间施加的限定。
- 问题的解空间：对于问题的一个实例，解向量满足显式约束条件的所有n维向量，构成了该问题实例的一个解空间。

回溯法的工作原理

- 回溯法的基本做法是搜索，是一种组织得井井有条，能避免不必要搜索的穷举式搜索法。
- 回溯法按深度优先策略搜索问题的解空间树。算法搜索至解空间树的任意一节点时，先判断该节点是否可能包含问题的解：1) 如果肯定不包含，则跳过这个节点；2) 如果可能包含，进入该子树，继续按深度优先策略搜索；3) 若某节点 i 的所有子节点都不可能包含问题的解，则回溯到 i 的父节点，生成下一个节点，继续搜索。



“合法着色”—全部顶点都已经被着色，且没有两个相邻的顶点是同样的颜色。
“部分着色”—部分顶点还未着色；在已经着色的顶点中，没有两个相邻的顶点是同样的颜色。

观察结论

- 节点是由深度优先搜索方法生成的
- 不需要存储整棵搜索树，只需要存储根到当前活动节点的路径
- 保存颜色指派的踪迹

递归回溯

输入：无向图G=(V,E)

输出：G的顶点的3着色c[1…n]，其中每个c[j]为1,2或3.

```
for k ←1 to n
    c[k] ←0  //no color
end for
flag ←false
graphcolor(1)
if flag then output c
else output “no solution”
```

graphcolor(k)

```
for color=1 to 3
    c[k] ←color
    if c为合法着色   then  flag ←true and exit
    else if c是部分着色  then graphcolor(k+1)
end for
```

迭代回溯

输入：无向图G=(V,E)

输出：G的顶点的3着色c[1…n]，其中每个c[j]为1,2或3.

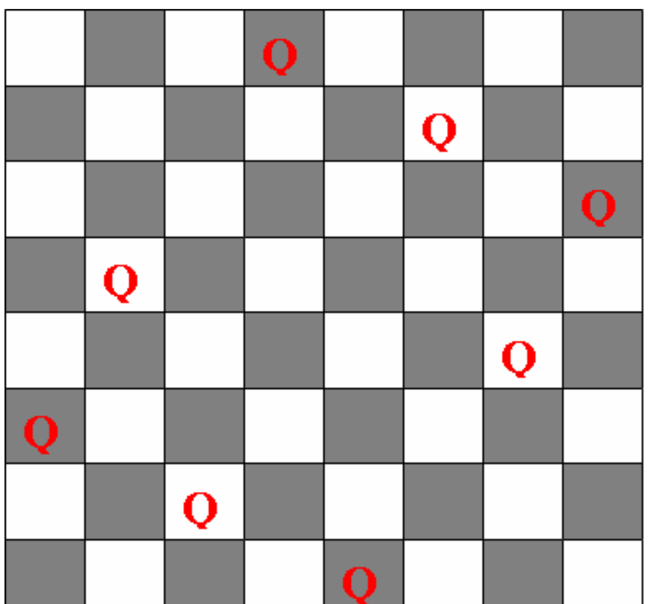
```
1. for k ←1 to n
2.     c[k] ←0
3. end for
4. flag ←false
5. k ←1 //start from v1
6. while k≥1
7.   while c[k]≤2 //为第k个顶点着色
8.        c[k] ←c[k]+1
9.        if c为合法着色then flag ←true and exit
10.       else if c为部分着色then k ←k+1  //准备为下一顶点着色,→7
11.  end while //假设着色既不合法也非部分的，即死节点，试其他色
12.  c[k] ←0 //vk试验了所有颜色均失败, 当前顶点颜色只好归0，回溯
13.  k ←k-1 //回溯到上一个顶点 (配合第6句的k≥1来理解)
14.end while //注意：回溯到上一个顶点后，c[k]+1，即尝试下一种颜色
```

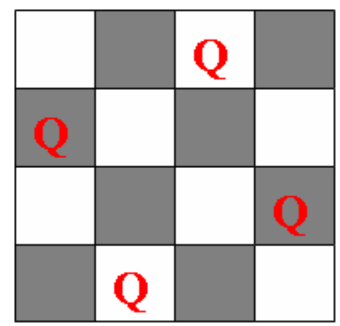
算法复杂度

- 最坏情况生成$O(3^n)$个节点
- 检测每个生成的节点：当前着色合法、部分，需要$O(n)$检查
- 最坏运行时间$O(n3^n)$

皇后问题

- 著名的数学家高斯在1850年提出：在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即：任意两个皇后都不能处于同一行、同一列或同一斜线上，问如何摆放？



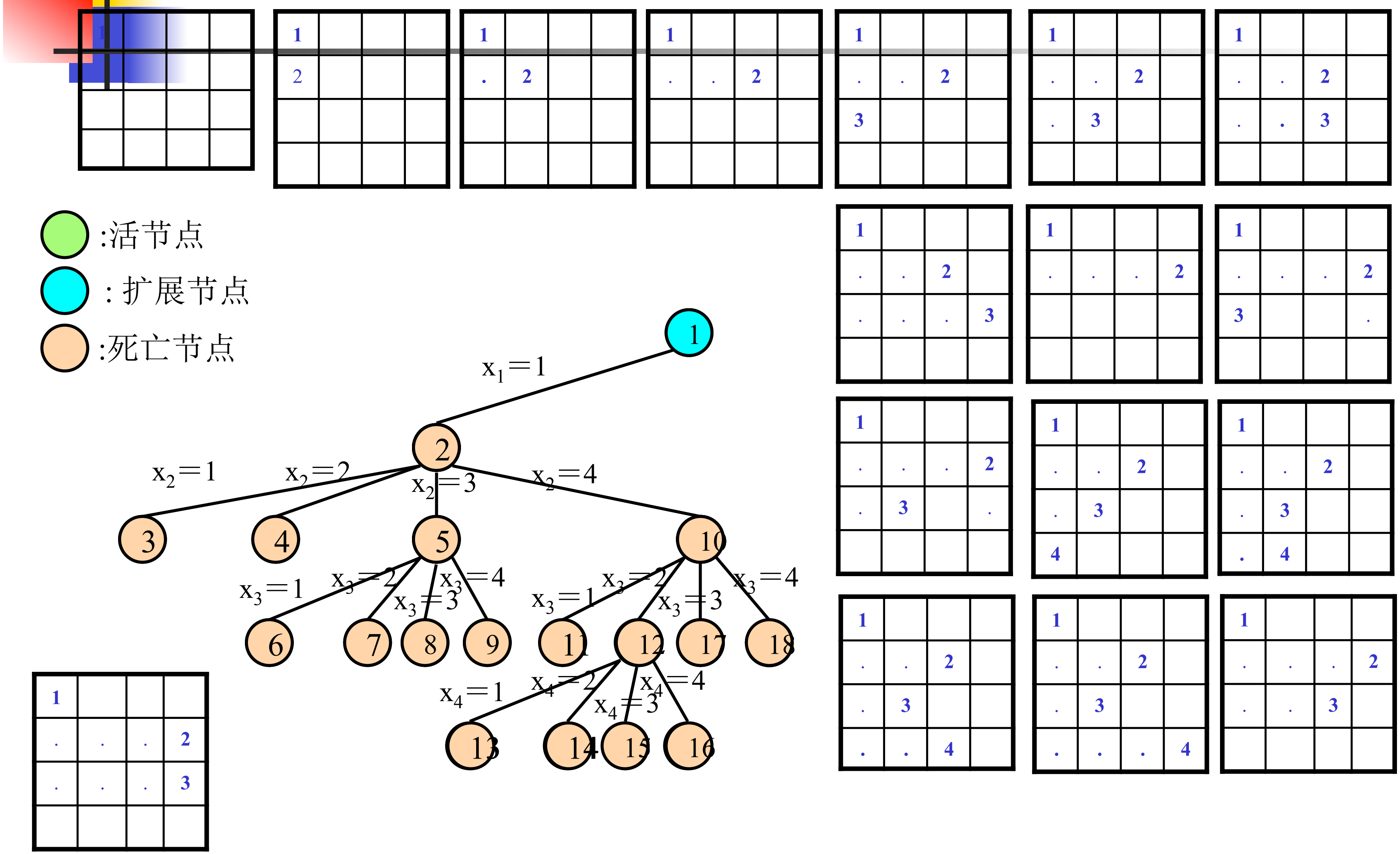


The solution above is (4，6，8，2，7，1，3，5).

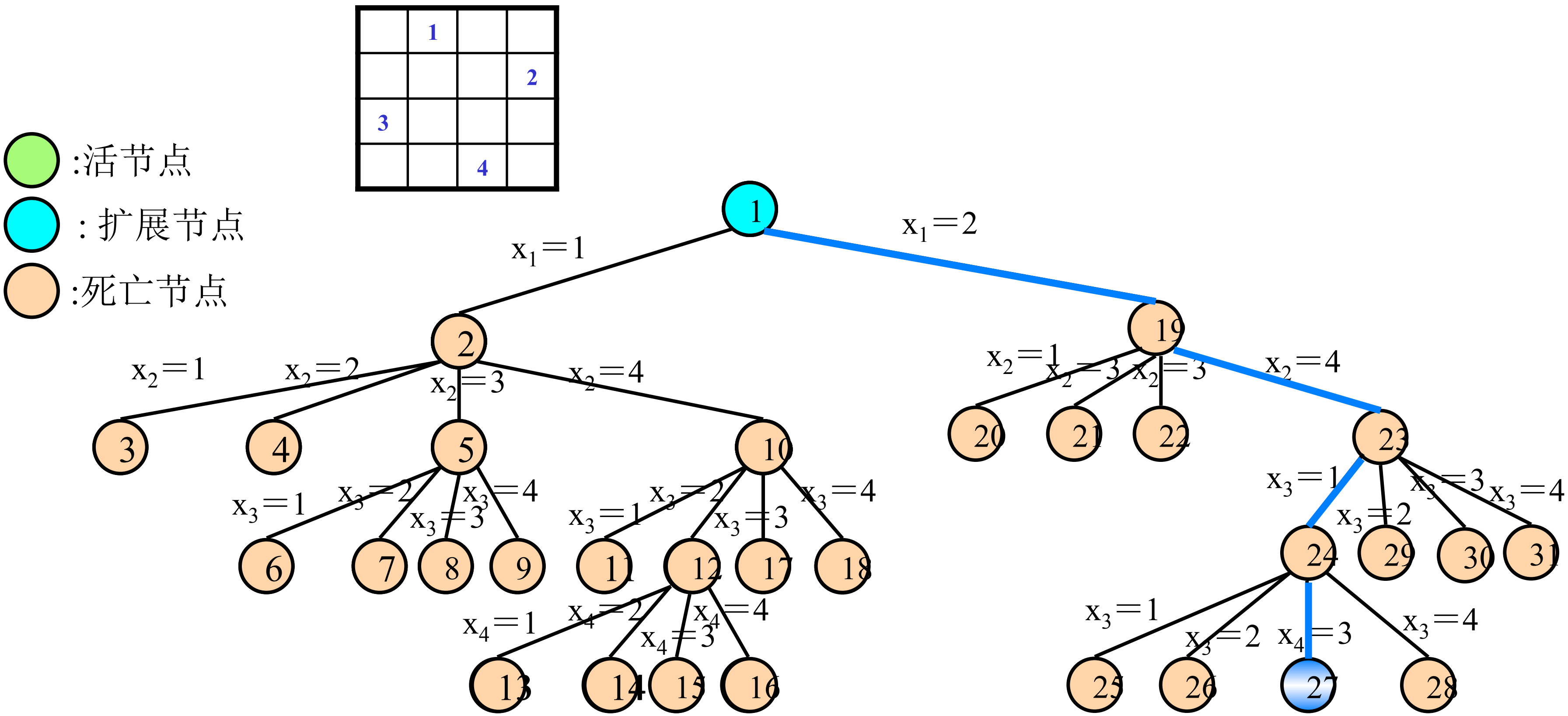
皇后问题

- 考察n皇后问题n=4的情形：解空间有4^4(可减少至4!)种布局, 可用一棵高度为4的完全4叉树表示：树的根对应于没有放置皇后的布局，第一层节点对应于皇后在第一行(列)可能放置列(行)的情况，依此类推。

4后问题

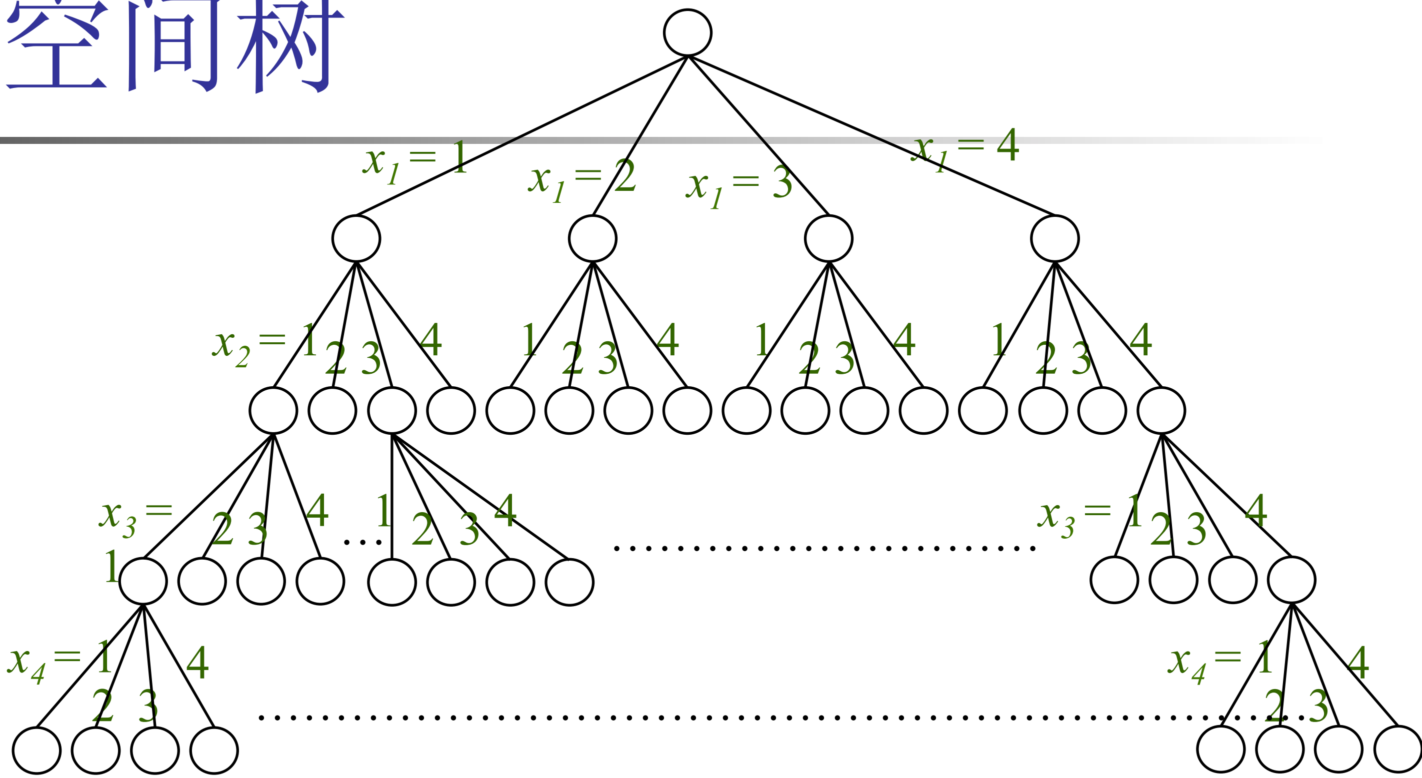

:活节点
: 扩展节点
:死亡节点
$x_1=1$
$x_2=1$
$x_2=2$
$x_2=3$
$x_2=4$
$x_3=1$
$x_3=2$
$x_3=3$
$x_3=4$
$x_4=1$
$x_4=2$
$x_4=3$
$x_4=4$

4后问题



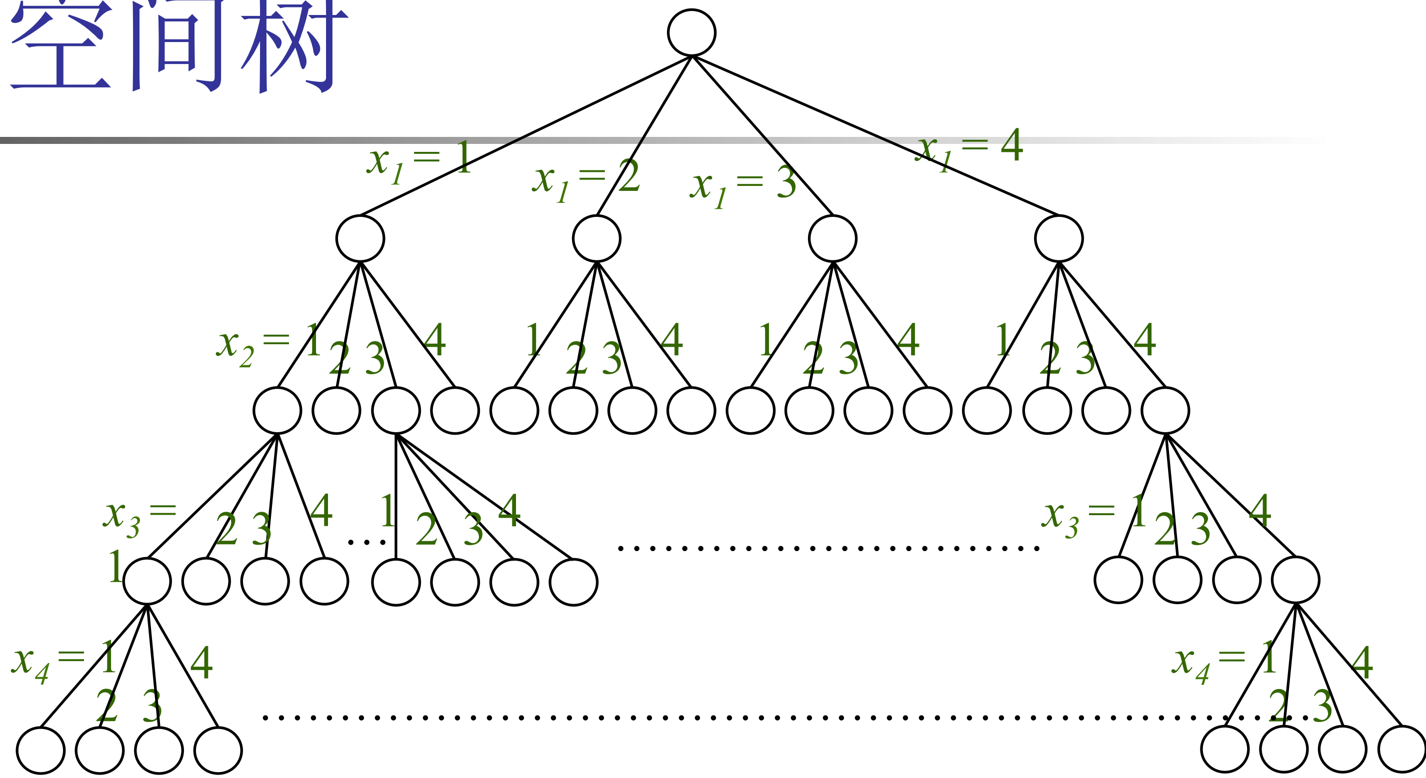
1
2
3
4
:活节点
: 扩展节点
:死亡节点
x1=1
x1=2
x2=1
x2=2
x2=3
x2=4
x3=1
x3=2
x3=3
x3=4
x4=1
x4=2
x4=3
x4=4

状态空间树



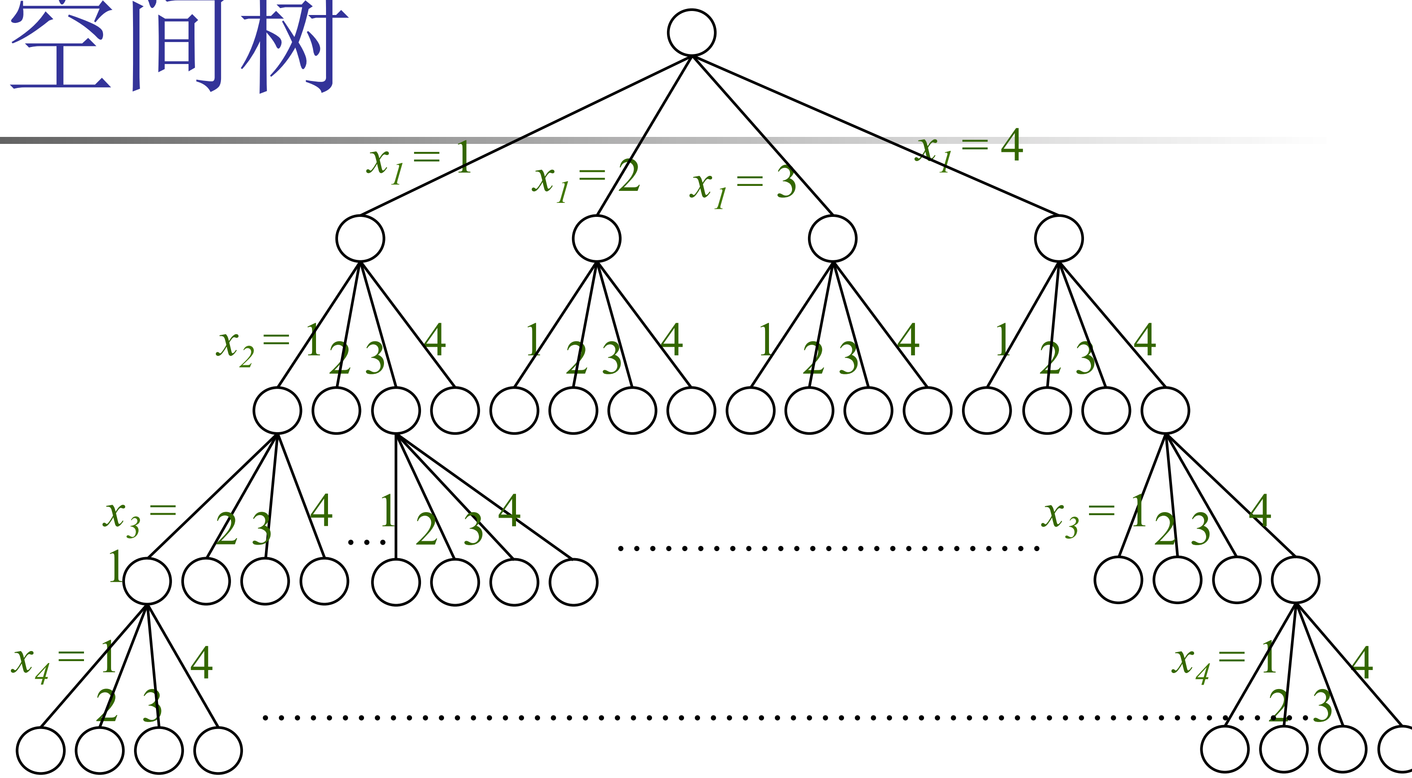
- 对于4后问题状态空间是满4叉树
- 从根节点到叶子节点的每条路径表示一个解决方案
- 叶子节点表示所有的解决方案（4^4）

状态空间树



- 活节点：正常节点
- 扩展节点：当前节点，正在对此节点进行搜索
- 死节点：不可行节点，无需对其下面的节点进行搜索

状态空间树



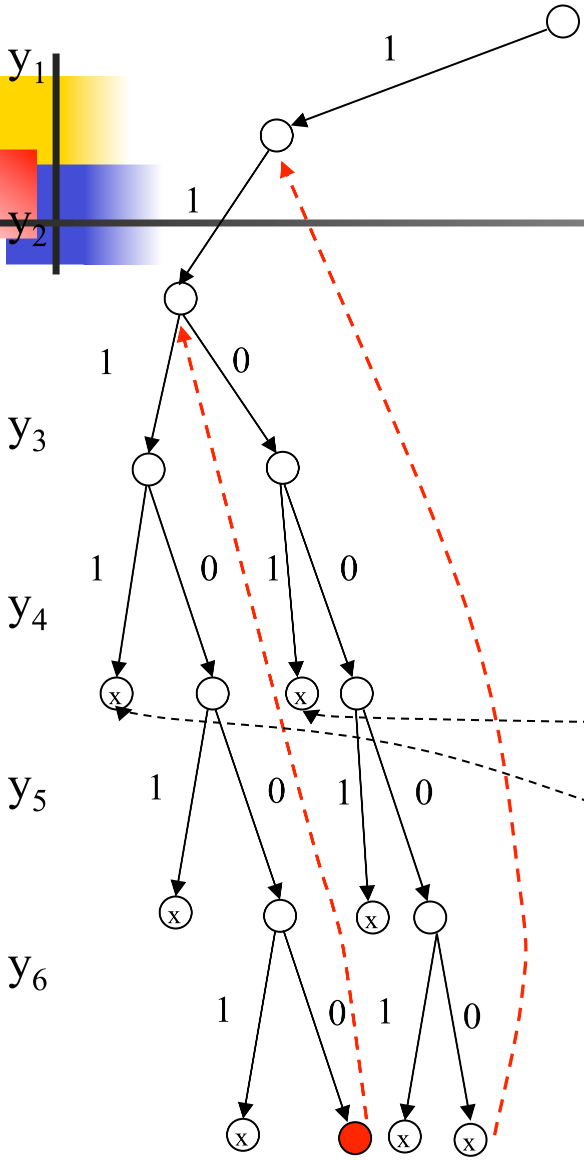
回溯法

- 从根节点（0.1）开始（0.1为活节点也是扩展节点），按照深度优先的方式搜索，如节点（1.1），此时节点1.1都是扩展节点
- 节点1.1再按照深度优先的算法进行搜索，如节点2.1，此节点为不可行节点，成为死节点，退回节点扩展节点1.1，对节点2.2进行搜索，可行。节点1.1为活节点，节点2.2为当前节点
- 重复以上步骤

- 给定一个问题的实例，如果该问题实例的解可以表示成定长的向量形式，那么就可以考虑使用回溯法来解决。
- 如果一个问题的实例，其解是一个变长的向量形式，是否可以使用回溯法？

一个例子

- 考虑如下的划分问题：给定n个整数的集合X＝$\{x_1,x_2,\ldots x_n\}$，整数y，试找到X的一个子集Y，Y中所有元素的和等于y。
- 例如：X=\{10,20,30,40,50,60\}，y=60。那么该问题的合法解有Y=\{10,20,30\}、\{20,40\}、\{60\}。(非定长的向量形式)
- 可以设法将问题的解转化为定长的向量形式：$Y=\{y_1,y_2,y_3,y_4,y_5,y_6\}$
- \{1,1,1,0,0,0\}、\{0,1,0,1,0,0\}、\{0,0,0,0,0,1\}



$X=\{10,20,30,40,50,60\}$， $y=60$

$Y=\{y_1,y_2,y_3,y_4,y_5,y_6\}$

$\{1,1,1,0,0,0\}$、$\{0,1,0,1,0,0\}$、$\{0,0,0,0,1\}$

$10+20+40>60$

$10+20+30+40>60$

通用回溯方法框架

- 本小节描述通用回溯方法的一般框架，可以作为系统搜索的基本框架，在解决实际问题时，修改该基本框架中的相应部分使之适合实际问题即可。
- 对于以下这样一类问题，可以使用回溯法：这类问题的解满足事先定义好的某种约束向量$(x_1,\ldots,x_i)$，这里i是依赖于问题规模n的常量。

通用回溯方法框架

在回溯法中，每个x_i均是属于某个有限集合的，不妨称之为X_i，那么回溯法实质上是按照词典顺序考虑笛卡儿积:

$$X_1 \times X_2 \times \ldots \times X_n$$

中的元素。

- 算法最初从空向量开始，先选择X_1中的最小元素作为x_1；
- 如果(x_1)是部分解，那么选择X_2中的最小元素作为x_2；
- 如果(x_1, x_2)是部分解，则继续考虑X_3中的最小元素作为x_3；
- 否则，考虑X_2中的第二个元素作为x_2。依此类推。

通用回溯方法框架

一般说来，当算法已经检测到部分解$(x_1, x_2,\ldots, x_j)$，需要继续考虑 $v = (x_1, x_2,\ldots, x_j, x_{j+1})$时，有以下几种情形：

1. 若v 表示问题的最终解，算法记录下它作为一个解。如果只需要一个解，算法结束；否则，继续找其它解。
2. 如果$v = (x_1, x_2,\ldots, x_j, x_{j+1})$ 是一个部分解，那么选择集合X_{j+2}中未使用过的最小元素作为x_{j+2}。(向前搜索)
3. 如果v 既非最终解，也非部分解，那么会有以下两种情况：

a. 如果集合X_{j+1}中还有其它未曾使用过的元素，则选择下一个未曾使用过的元素作为x_{j+1}。

b. 如果集合X_{j+1}中没有其它未曾使用过的元素，则回溯，将X_j中未曾使用的下一元素作为x_j，并将x_{j+1}进行重置。如果集合X_j中没有未曾使用的元素，那么继续回溯(同样注意要进行重置)。

回溯法的通用方法

BACKTRACKING

```
BACKTRACKING(n)
 k←1;  （层数）
 while k>0 do
  if there are unchecked X(k), X(k)∈T(X(1),…,X(k-1)) and B(X(1),…,X(k))=true
      then   if (X(1),…,X(k) is an answer）
                  then  print (X(1),…,X(k))
             if (k < n)
                  k←k+1
      else
             k←k-1
```

当前扩展节点是否还有节点没有被访问？
B函数是边界条件

- Construct solution in X(1:n), an answer is output immediately after it is determined.
- T(X(1),…,X(k-1)) : returns all possible X(k) ,given X(1),…,X(k-1).
- B(X(1),…,X(k)): returns whether X(1),…,X(k) satisfies the implicit constraints.

回溯法的通用方法

BACKTRACKING

```
BACKTRACKING(n)
 k←1;  （层数）
 while k>0 do
  if there are unchecked X(k), X(k)∈T(X(1),…,X(k-1)) and B(X(1),…,X(k))=true
      then  if (X(1),…,X(k) is an answer）
                then  print (X(X(1),…,X(k))
            if (k < n)
                k←k+1
      else
            k←k-1
```

此节点是可行解

- Construct solution in X(1:n), an answer is output immediately after it is determined.
- T(X(1),...,X(k-1)) : returns all possible X(k) ,given X(1),…,X(k-1).
- B(X(1),…,X(k)): returns whether X(1),…,X(k) satisfies the implicit constraints.

回溯法的通用方法

BACKTRACKING

```
BACKTRACKING(n)
 k←1;  （层数）
 while k>0 do
  if there are unchecked X(k), X(k)∈T(X(1),…,X(k-1)) and B(X(1),…,X(k))=true
      then   if (X(1),…,X(k) is an answer）
                  then  print (X(1),…,X(k))
             if (k < n)
                  k←k+1
       else
             k←k-1
```

DFS:此节点为扩展节点.

- Construct solution in X(1:n), an answer is output immediately after it is determined.
- T(X(1),...,X(k-1)) : returns all possible X(k) ,given X(1),…,X(k-1).
- B(X(1),…,X(k)): returns whether X(1),…,X(k) satisfies the implicit constraints.

回溯法的通用方法

BACKTRACKING

```
BACKTRACKING(n)
 k←1; （层数）
 while k>0 do
  if there are unchecked X(k), X(k)∈T(X(1),…,X(k-1)) and B(X(1),…,X(k))=true
      then  if (X(1),…,X(k) is an answer）
                then  print (X(1),…,X(k))
            if (k < n)
                k←k+1
      else
            k←k-1
```

所有的节点都已经被访问，不可行，返回

- Construct solution in X(1:n), an answer is output immediately after it is determined.
- T(X(1),...,X(k-1)) : returns all possible X(k) ,given X(1),...,X(k-1).
- B(X(1),...,X(k)): returns whether X(1),...,X(k) satisfies the implicit constraints.

N后的回溯法

NQUEENS

```
NQUEENS(n)
    X(1)←0;k←1
    while k>0 do
        X(k)←X(k)+1
        while X(k) ≤ n and not PLACE(k) do
            X(k)←X(k)+1
        if X(k) ≤ n
            then  if k=n
                      then print (X)
                  else k←k+1;X(k)←0
         else x[k]=0; k←k-1;
```

k : current row;
X(k) : column NO of queen k

Finding next valid column NO using bound function PLACE.

Found or not?

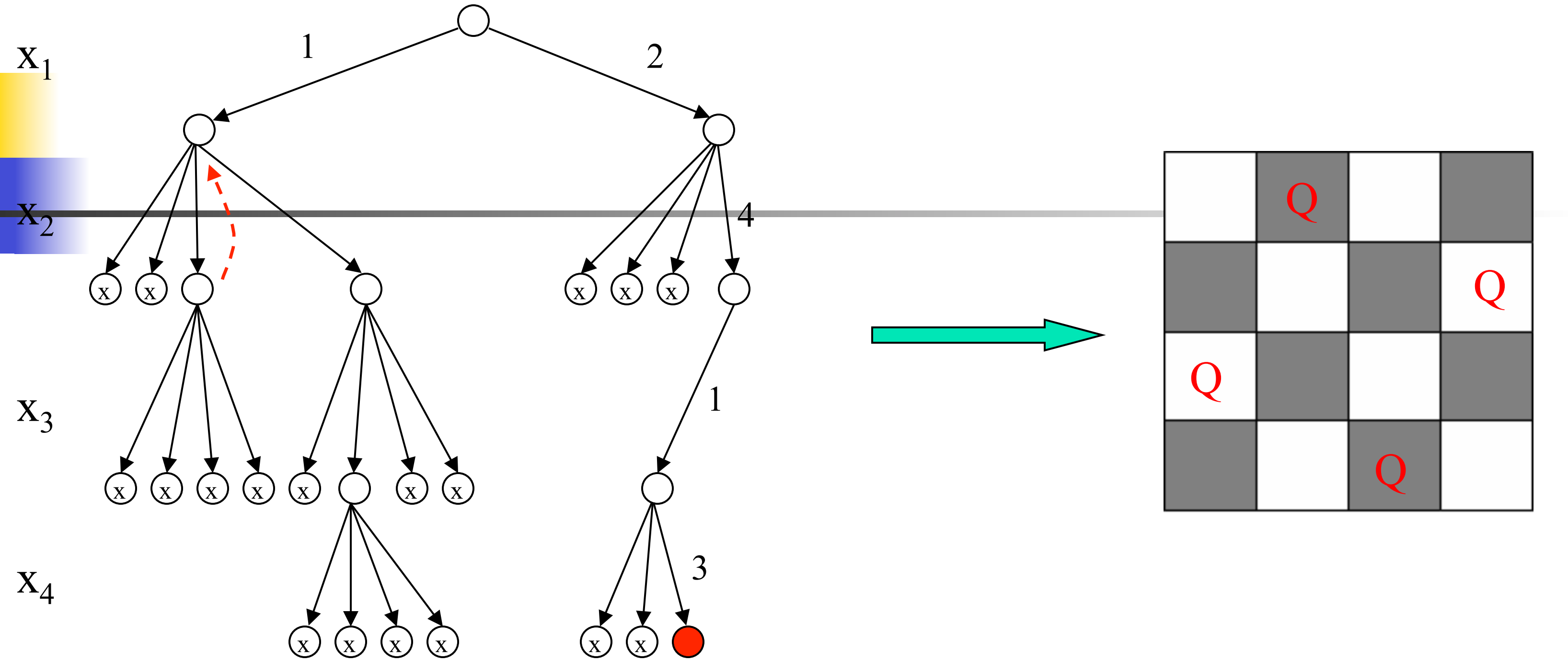
Found! Is it an answer? Print the answer, or shift to the next row and find a place for next queen.

No! Tracking back.

Bound function -- PLACE

NQUEENS

```
PLACE(k)
  i←1
  while i<k do
      if X(i)=X(k) or |X(i)-X(k)| = |i-k|
          then return false
      i←i+1
  return true
```



- 效率问题：蛮力方法候选解的个数是n!；而使用回溯法，候选解的个数是n^n，回溯的效率是否太低了？
- 然而，仔细分析可以发现，回溯法可以极大减少测试次数：
  - 例如，假设前两个皇后一个放1列，一个放2列，蛮力方法仍旧要测试(n −2)!个候选解；而回溯法只要进行一次测试就可以避免剩余无意义的测试。
  - 所以，尽管在最坏情况下要用$O(n^n)$时间来求解，然而大量实际经验表明，它在有效性上远远超过蛮力方法O(n!)。例如在4 皇后问题中，只搜索了341个节点中的27个就找到了解。

回溯法：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?
- 解决方案
  - A solution is defined as an n-元组 $(x_1, x_2, \ldots, x_n)$, where $x_i \in \{0, 1\}$, $1 \le i \le n$. If item *i* is taken, then $x_i = 1$, otherwise $x_i = 0$.
- 0-1背包是找到一个最优解，而不是找一个可行解

0-1背包问题：状态空间树

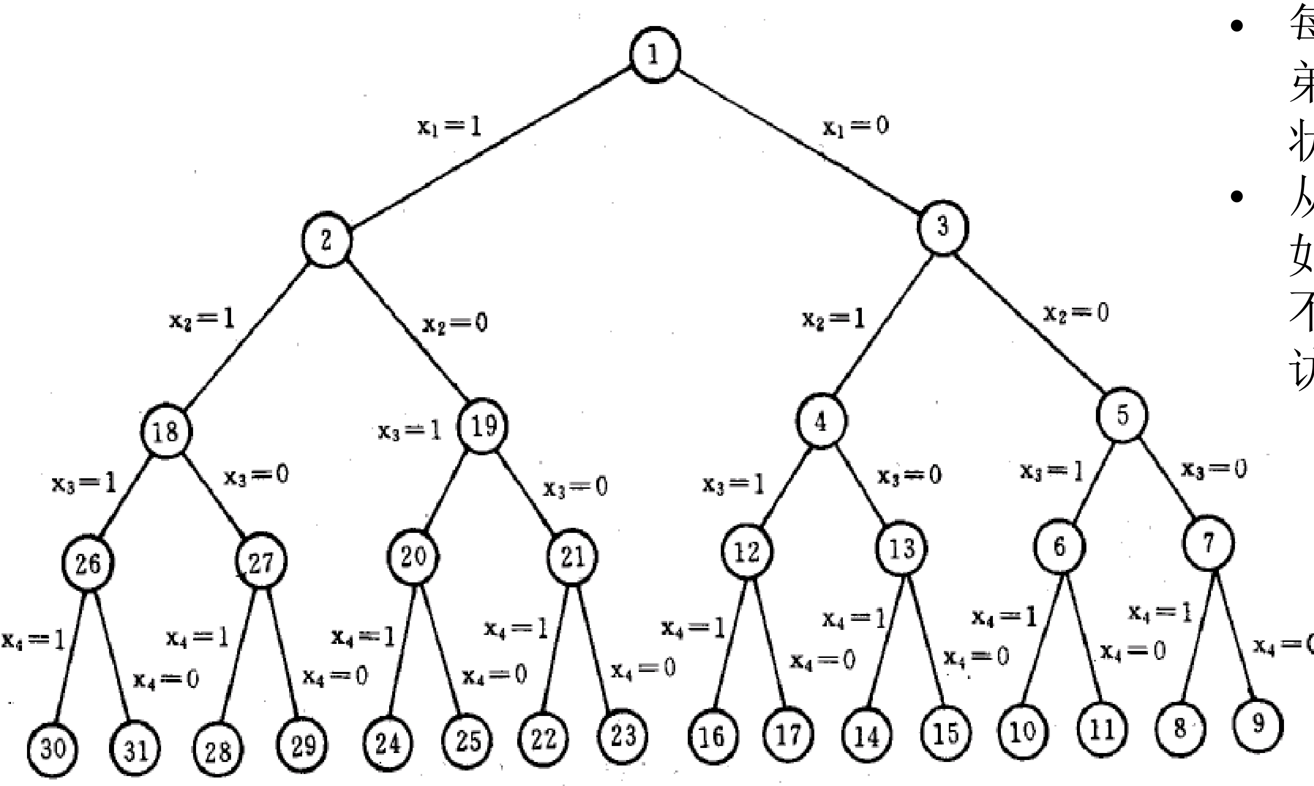


图 6.4

- 每层代表一个物体，兄弟节点表示物体的两种状态，取还是不取
- 从根到叶子依次遍历，如果两个兄弟节点访问不成功，则返回父节点，访问父节点的兄弟节点

武汉大学 WUHAN UNIVERSITY

0-1背包问题：非递归

```
BACKTRACKING(M,W,P,fw,fp,X ) //W weight; P profile; fw, fp: final solution; X
cw←cp←0;k←1;Y=2 //cw总重量，cp总价值
 while k>0
        Y[k]=Y[k]-1;
        if Y[K]==1
            if  cw+W(k)<M; //选取一个有效的Y值
                    cw←cw+W(k);cp←cp+P(k);
            else
                    Y[k]=Y[k]-1//不放此物品
       if Y[k]>=0
              if k=n //k是否已经遍历完毕
                     if (cp>fp) fp←cp;fw←cw;X←Y//记录当前最优解
                      k=k-1;
            else
                      k=k+1;
       else//两个解都已经遍历完毕了
          Y[k]=2;//初始化
          k=k-1;
```

0-1背包问题：非递归2

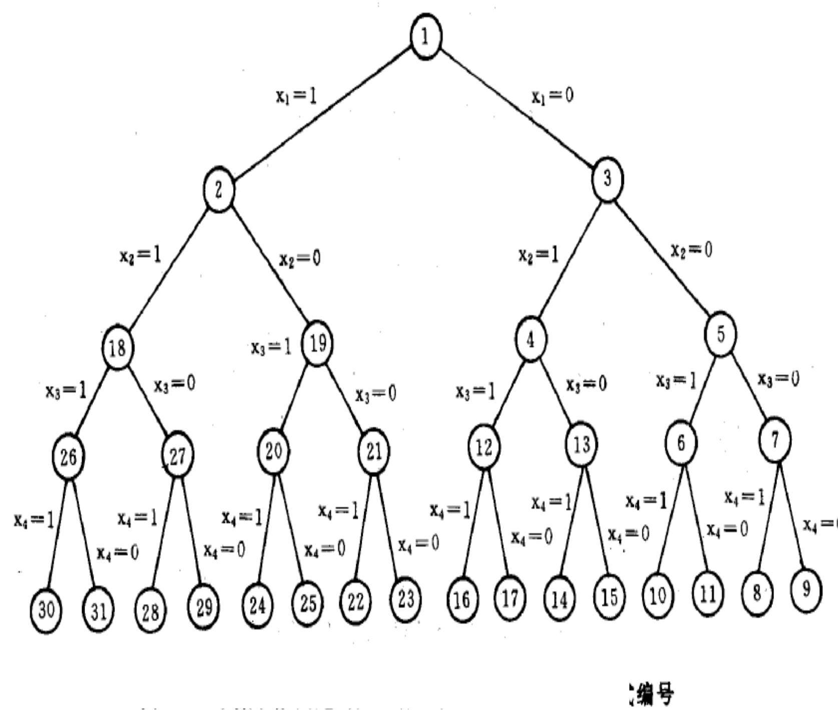
```
BACKTRACKING(M,W,P,fw,fp,X ) //W weight; P profile; fw, fp: final solution; X
cw←cp←0;k←1;
loop
     while k<=n and cw+W(k)<=M
          cw=cw+W(k); cp=cp+P(k); Y[k]=1; k=k+1;
     if  k>n //到达叶子节点
           if (cp>fp)  fp=cp; fn=cn; X=Y; k=n-1;
           while k!=0 and Y[k]=0 //回溯
                           k=k-1;
           Y(k)=0; cw=cw-w(k); cp=cp-P(k);
     else
            Y[k]=0
      if k=0
        return;
      k=k+1;
```

0-1背包问题:递归

```
BACKTRACKING(int i)
  If (i>n)
     If (cp>fp) fw=cw; fp=cp;
      return;
    End
  End
  if (cw+W(i)<=M) //搜索左子树
      cw=cw+W(i);
      cp=cp+P(i);
      BACKTRACKING(i+1)
      cw=cw-W(i);
      cp=cp-P(i)
    End
    BACKTRACKING(i+1) //搜索右子树
```

0-1背包问题：改进

- 在搜索左子树时有判断条件cw+W(i)<=M，但是右子树没有，所以右子树的遍历每次都会执行
- 如果右子树最大可能的填充（上界函数）都比现有最好的解决方案小的话，就可以不用搜索右子树
- 上界函数
    - 物品按照性价比排列（p(i)/w(i)）
    - 当目前遍历物品k，还剩余m容量时，则最大填充为：依次将k+1，k+2，放入，直到不能完整的放整个物品（k+j），则将（k+j）比例放入



0-1背包问题：上界函数

the value of $X(i)$, $1 \le i \le k$ have been determined.
p: profit gained before call
w: weight used before call
k: item being checked just now
M: knapsack capacity
return the up-bound profit value (items are sorted in decreasing order on $P(i)/W(i)$).

```
BOUND( p,w,k,M )
   b←p; c←w
   for i←k+1 to n do
      c←c+W(i)
      if c<M then b←b+P(i)
      else return (b+(1-(c-M)/W(i))*P(i))
  return (b)
```

0-1背包问题：上界函数

Determine up-bound using greedy algorithm: check item k+1~n one by one, place all the item into the knapsack if possible, otherwise place part of it into.

```
BOUND( p,w,k,M )
  b←p; c←w
  for i←k+1 to n do
    c←c+W(i)
    if c<M then b←b+P(i)
      else return (b+(1-(c-M)/W(i))*P(i))
 return (b)
```

0-1背包问题：改进后

M: knapsack capacity
n items, weight and profit of each item are stored in **W(1:n)** and **P(1:n)**, **P(i)/W(i)≥P(i+1)/W(i+1)**;
fw: weight used eventually
fp: profit gained eventually
X: answer vector

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←1;k←k+1
    if k>n then fp←cp;fw←cw;k←n;X←Y
     else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
             k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
   k←k+1
```

0-1背包问题：改进后

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and
          cw←cw+
    if k>n then fp←
    else Y(k)←0
    while BOUND(
          while k≠0 and Y(k)≠1 do
               k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
    k←k+1
```

cw: weight used
cp: profit gained

when fp≠-1, X is the vector that gains profit fp

fw,fp,X: information about the best answer currently found.

cw,cp,Y: corresponding information used when searching.

0-1背包问题：改进后

Travel to left son when possible, store the path in Y.

NOTE: BOUND is not used here.

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←1;k←k+1
    if k>n then fp←cp;fw←cw;k←n;X←Y
     else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
              k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
   k←k+1
```

0-1背包问题：改进后

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←1;k←k+1
    if k>n then fp←cp;fw←cw;k←n;X←Y
    else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
              k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
    k←k+1
```

if k>n, we get that cp>fp(if cp≤fp, the last call to BOUND will stop the traveling to this leaf), save the answer info.

if k≤n, item k can not be placed into the knapsack, so we must travel to right son(assign 0 to Y(k)).

0-1背包问题：改进后

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←
    if k>n then fp←cp;fw←cw;k←n;X←Y
     else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
              k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
   k←k+1
```

Use BOUND to check that whether a better answer is possible, Track back if not.

Track back to the nearest node whose right son has not been generated, generate his right son and check by BOUND.

after while ends, we find a better direction, or the procedure terminates.

0-1背包问题：改进后

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←1;k←k+1
    if k>n then fp←cp;fw←cw;k←n;X←Y
     else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
             k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
   k←k+1
```

Case Study

M=110,n=8

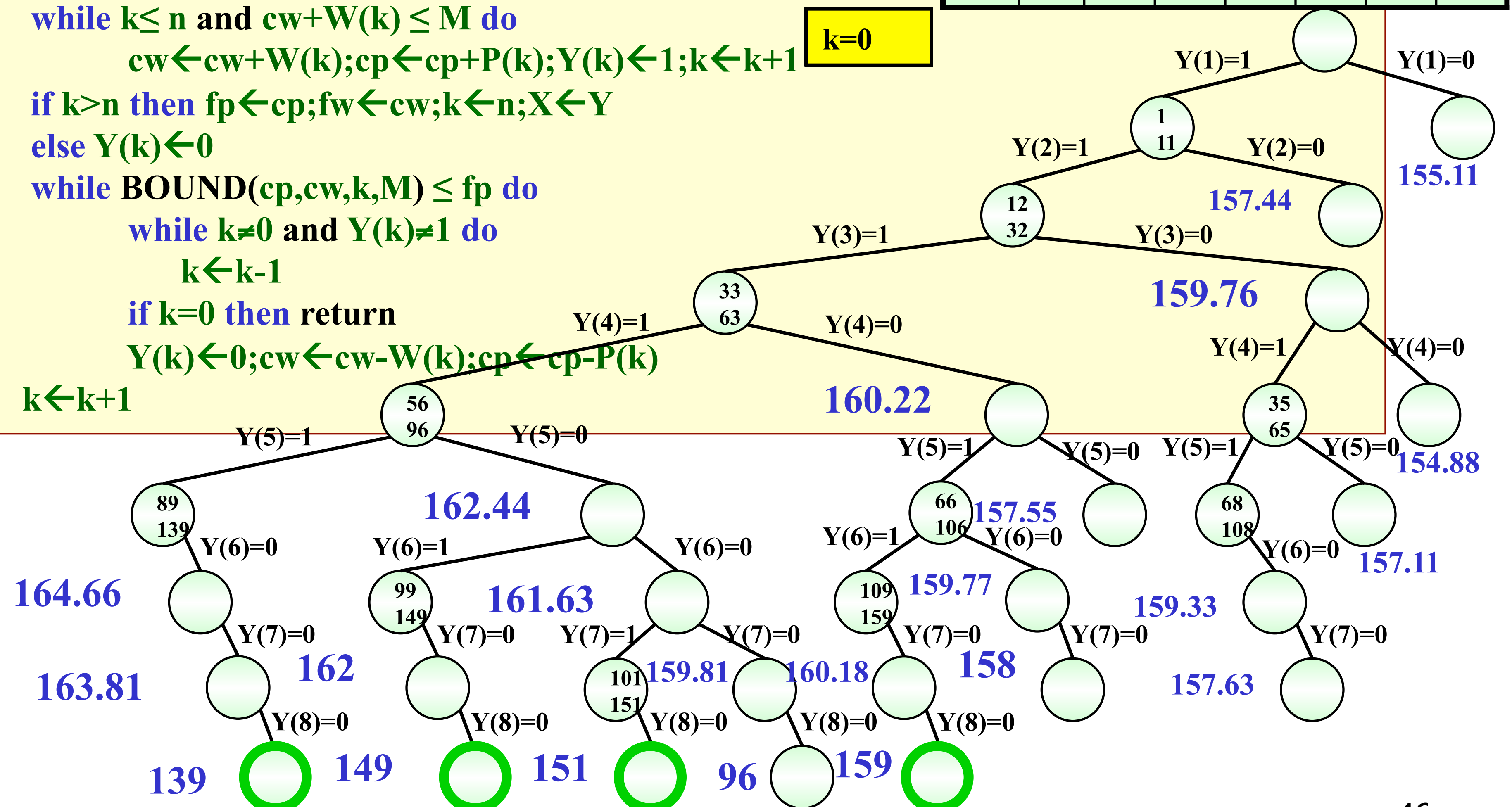
	1	2	3	4	5	6	7	8
P	11	21	31	33	43	53	55	65
W	1	11	21	23	33	43	45	55

fw=109
fp=159

k=0

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)	X(7)	X(8)
1	1	1	0	1	1	0	0

```
BKNAP1( M,n,W,P,fw,fp,X )
  cw←cp←0;k←1;fp←-1
  loop
    while k≤ n and cw+W(k) ≤ M do
          cw←cw+W(k);cp←cp+P(k);Y(k)←1;k←k+1
    if k>n then fp←cp;fw←cw;k←n;X←Y
    else Y(k)←0
    while BOUND(cp,cw,k,M) ≤ fp do
          while k≠0 and Y(k)≠1 do
              k←k-1
          if k=0 then return
          Y(k)←0;cw←cw-W(k);cp←cp-P(k)
  k←k+1
```



Case Study

M=110,n=8

	1	2	3	4	5	6	7	8
P	11	21	31	33	43	53	55	65
W	1	11	21	23	33	43	45	55

generate 35 of $2^9-1=511$ nodes of the state space tree

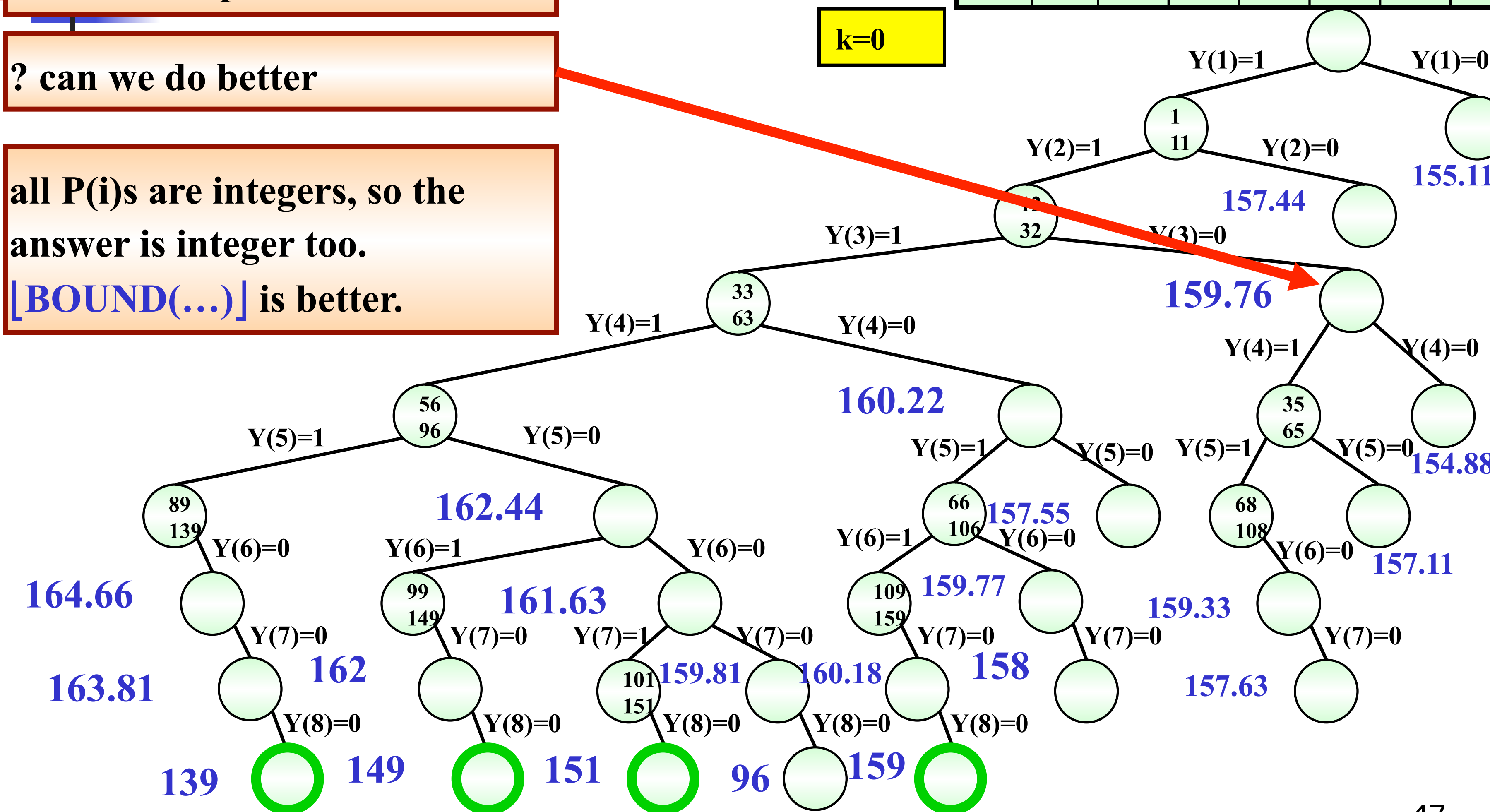
? can we do better

all P(i)s are integers, so the answer is integer too. $\lfloor$BOUND(…)$\rfloor$ is better.

fw=109
fp=159

k=0

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)	X(7)	X(8)
1	1	1	0	1	1	0	0



回溯法：总结

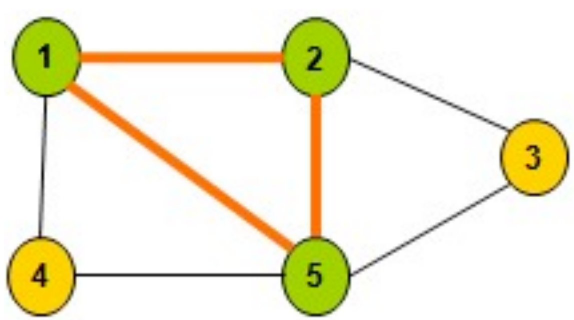
- 3步骤
  - 针对所给问题，定义问题的解空间
    - n后：每行所有的可能n^n，
    - 0-1背包：2^n
  - 确定易于搜索的解空间结构
    - 状态空间树
  - 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索
    - 剪枝函数：约束函数
      - n后：不能同列、对角线
      - 0-1：不能超过总容量
    - 剪枝函数：边界函数
      - 0-1：右子树

最大团问题

- ***问题描述***

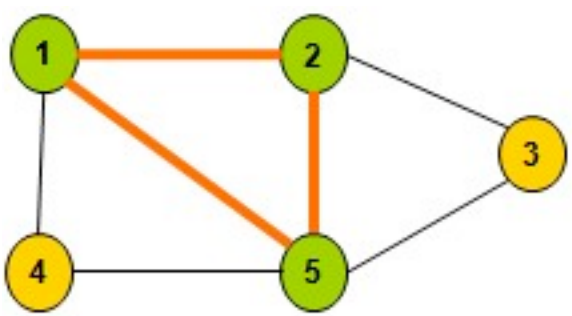
  给定无向图G=(V, E)，其中V是非空集合，称为顶点集；E是V中元素构成的无序二元组的集合，称为边集，无向图中的边均是顶点的无序对，无序对常用圆括号"( )"表示。如果U∈V，且对任意两个顶点u，v∈U有(u, v)∈E，则称U是G的*完全子图(完全图**G**就是指图**G**的每个顶点之间都有连边)*。***G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中***。G的最大团是指G中所含*顶点数最多的团*。

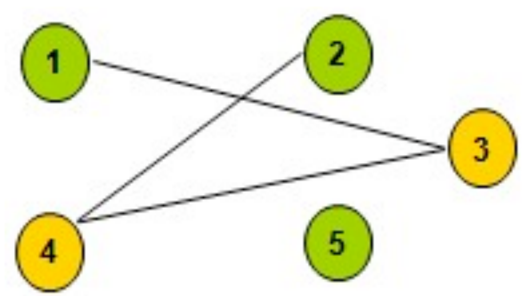
- 集{1，2}是G大小为2的完全子图，这个完全子图不是团，因为它被更大完全子图{1，2，5}包含，{1，2，5}是G的最大团，{1，4，5}和{2，3，5}也是最大团



最大团问题

- 独立集
  - 如果U∈V且对任意u，v∈U有(u, v)不属于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***。
- 补图
  - 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)不∈E。
- 如果U是G的完全子图，则它也是G'的空子图，反之亦然。因此，***G的团与G'的独立集之间存在一一对应的关系***。特殊地，U是G的最大团当且仅当U是G'的最大独立集。





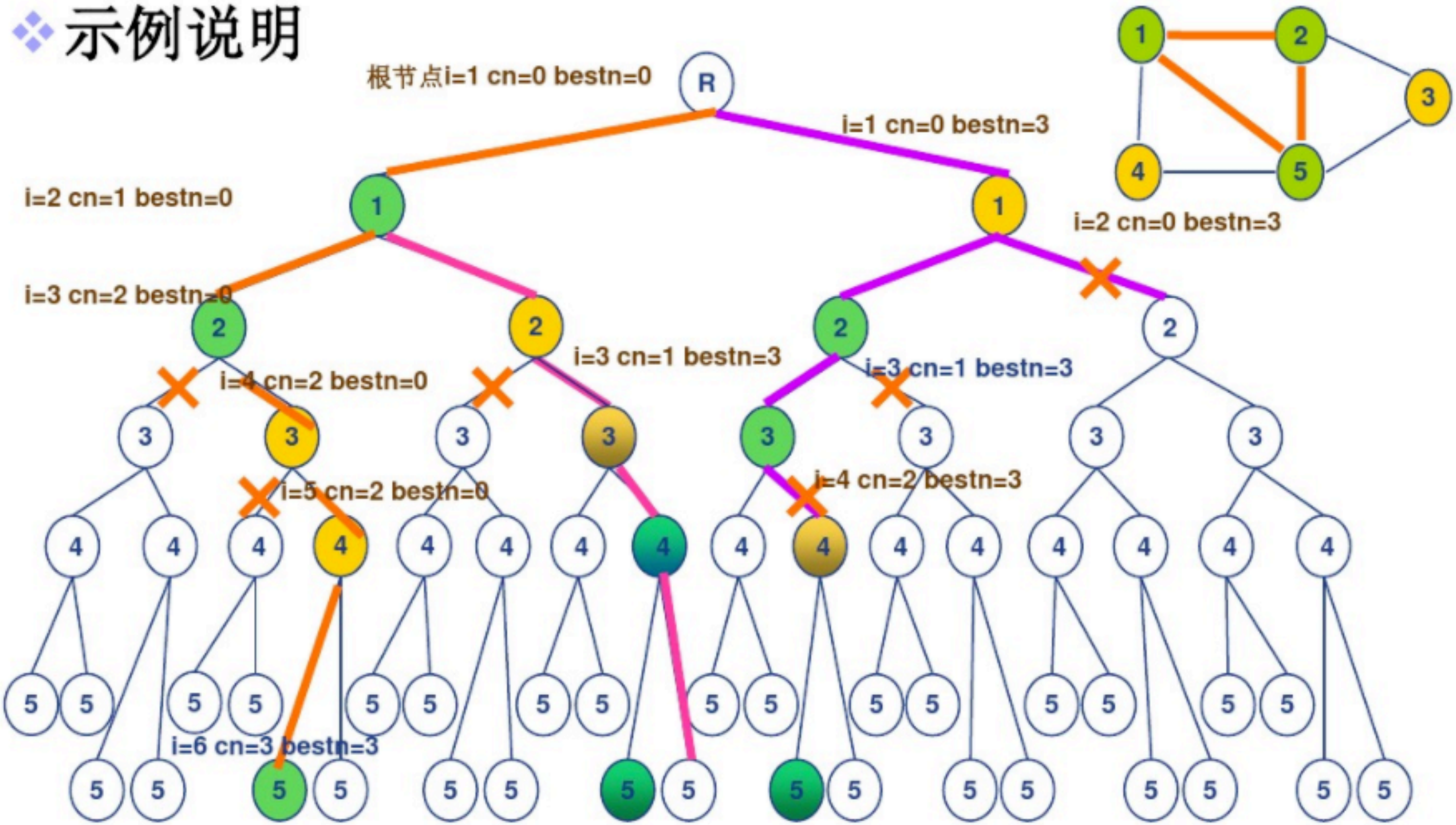
最大团问题：算法设计

无向图G的最大团问题可以看作是图G的顶点集V的子集选取问题。因此可以用子集树表示问题的解空间。设当前扩展节点Z位于解空间树的第i层。在进入左子树前，必须确认从顶点i到已入选的顶点集中每一个顶点都有边相连。在进入右子树之前，必须确认还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。

用邻接矩阵表示图G，n为G的顶点数，cn存储当前团的顶点数，bestn存储最大团的顶点数。cn+n-i为进入右子树的上界函数，当cn+n-i<bestn时，不能在右子树中找到更大的团，可将Z的右节点剪去。

最大团问题：示例说明

- 示例说明



判断顶点是否可入团

为了判断当前顶点加入团之后是否仍是一个团，只需要考虑该顶点和团中顶点是否都有连接。代码如下：

```
for(j=1;j<i;j++)
    if(x[j]&&a[i][j]==0) {ok=0;break;}
```

i表示当前节点，j遍历已经加入的节点，其中已经加入的节点x[j]==1，表示已经加入

剪枝策略

程序中采用了一个比较简单的剪枝策略，即如果剩余未考虑的顶点数加上团中顶点数不大于当前解的顶点数，可停止继续深度搜索，否则继续深度递归，程序代码如下：

```
if(cn+n-i>bestn)
  {
    x[i]=0;//设置标志
    backtrack(i+1);//递归判断下一顶点
  }
```

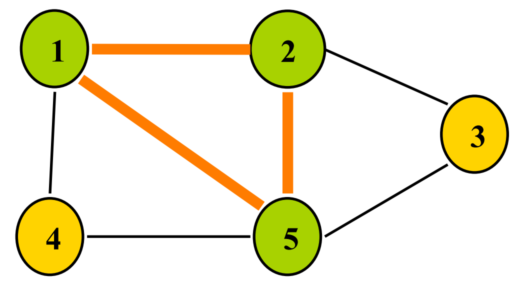
cn表示已经加入团的节点数，n-i表示剩余的节点数
进入右子树表示此节点不被包括在最大团，所以x[i]=0

递归结束条件

当搜索到一个叶结点时，即可停止搜索，此时更新最优解和最优值，程序代码如下：

```
if(i>n)
  {
    for(j=1;j<=n;j++)
      bestx[j]=x[j];
    bestn=cn;
    return;
  }
```

- 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。




```
private static void backtrack(int i)
   {
      if (i > n)   {// 到达叶结点
         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
       boolean ok = true;
      for (int j = 1; j < i; j++)
        if (x[j] == 1 && !a[i][j])    {// i与j不相连
           ok = false;     break;    }
      if (ok)    {// 进入左子树
         x[i] = 1;  cn++;
        backtrack(i + 1);
        cn--;      }
      if (cn + n - i > bestn)      {// 进入右子树
         x[i] = 0;
        backtrack(i + 1);      }   }}
```

时间复杂度

由于最大团问题是一个子集树问题，每个可行叶结点都做一次bestx更新，因此可知算法的时间复杂度为$O(n2^n)$

分支限界法(Branch and Bounds)

- 分支限界法类似于回溯法，也是一种在问题的解空间树T中搜索问题解的算法。
- 求解目标：分支限界法的求解目标则是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在某种意义下的最优解。
- 分支限界法与回溯法的**搜索方式**不同
  - 回溯法：深度优先
  - 分支限界法：广度优先或最小耗费（最大收益）优先

基本思想

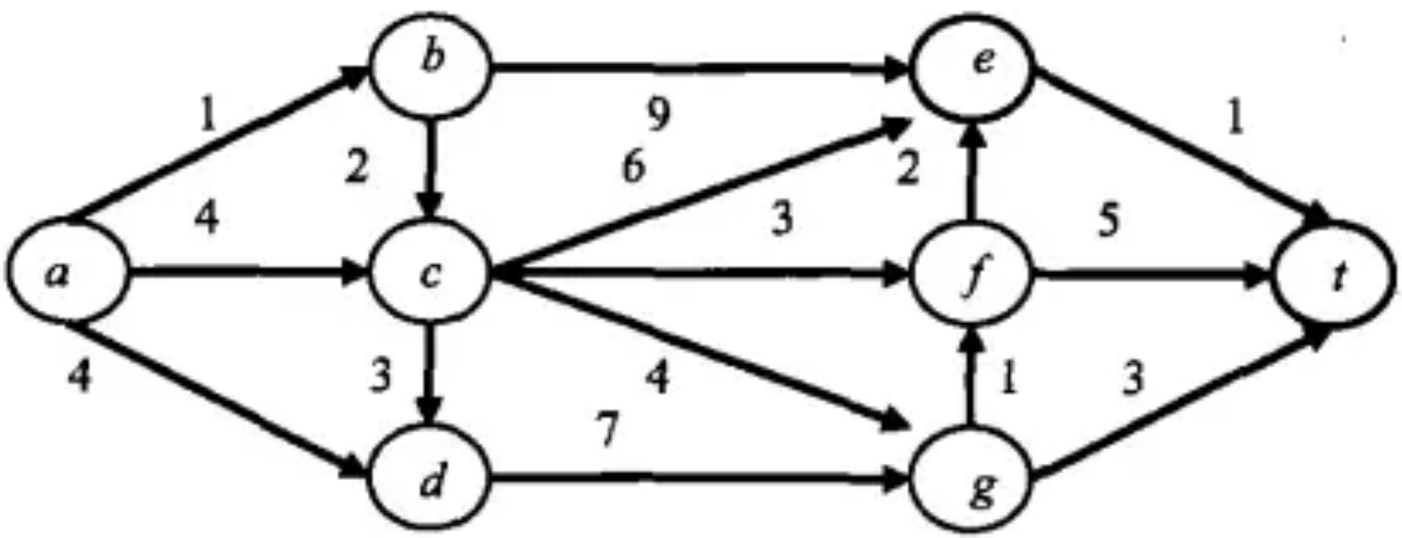
- 分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。
- 在分支限界法中，每一个活节点只有一次机会成为扩展节点。活节点一旦成为扩展节点，就一次性产生其所有儿子节点。在这些儿子节点中，导致不可行解儿子节点被舍弃，其余儿子节点被加入活节点表(队列或是堆，详见后面的例子)中。
- 此后，从活节点表中取下一节点成为当前扩展节点，并重复上述节点扩展过程。这个过程一直持续到找到所需的解或活节点表为空时为止。

常见的两种分支限界法

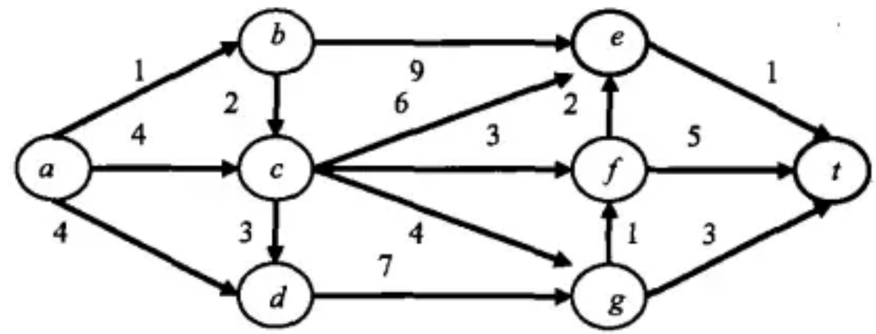
- 从活节点表中选择下一扩展节点的不同方式导致不同的分支限界法：
  - 队列式(FIFO)分支限界法(宽度优先)：按照队列先进先出（FIFO）原则选取下一个节点为扩展节点。
  - 堆式分支限界法(最小耗费或是最大收益优先)：按照堆中规定的优先级选取优先级最高的节点成为当前扩展节点。
    - 最大堆：使用最大堆，体现最大收益优先
    - 最小堆：使用最小堆，体现最小耗费优先

单源最短路径

- 计算节点a到节点t的最短路径



单源最短路径



- 确定下界
  - 经过某节点到t的最短路径的下界为：到此节点的最短路径+此节点的一条最短边

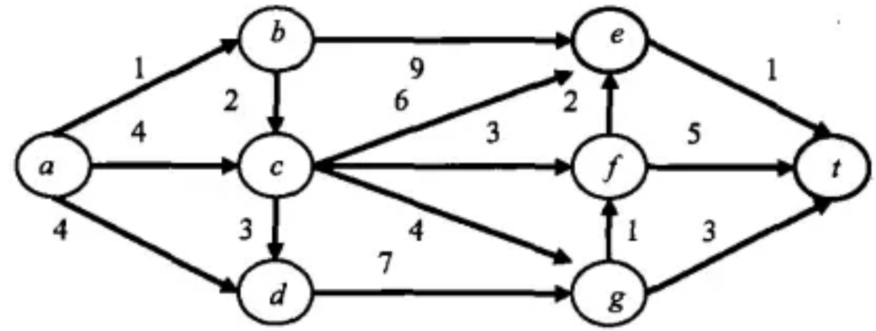
源顶点为a，目标定的为t，把a作为根节点进行搜索：

a->b->t距离的下界为：1+min{2, 9} = 3

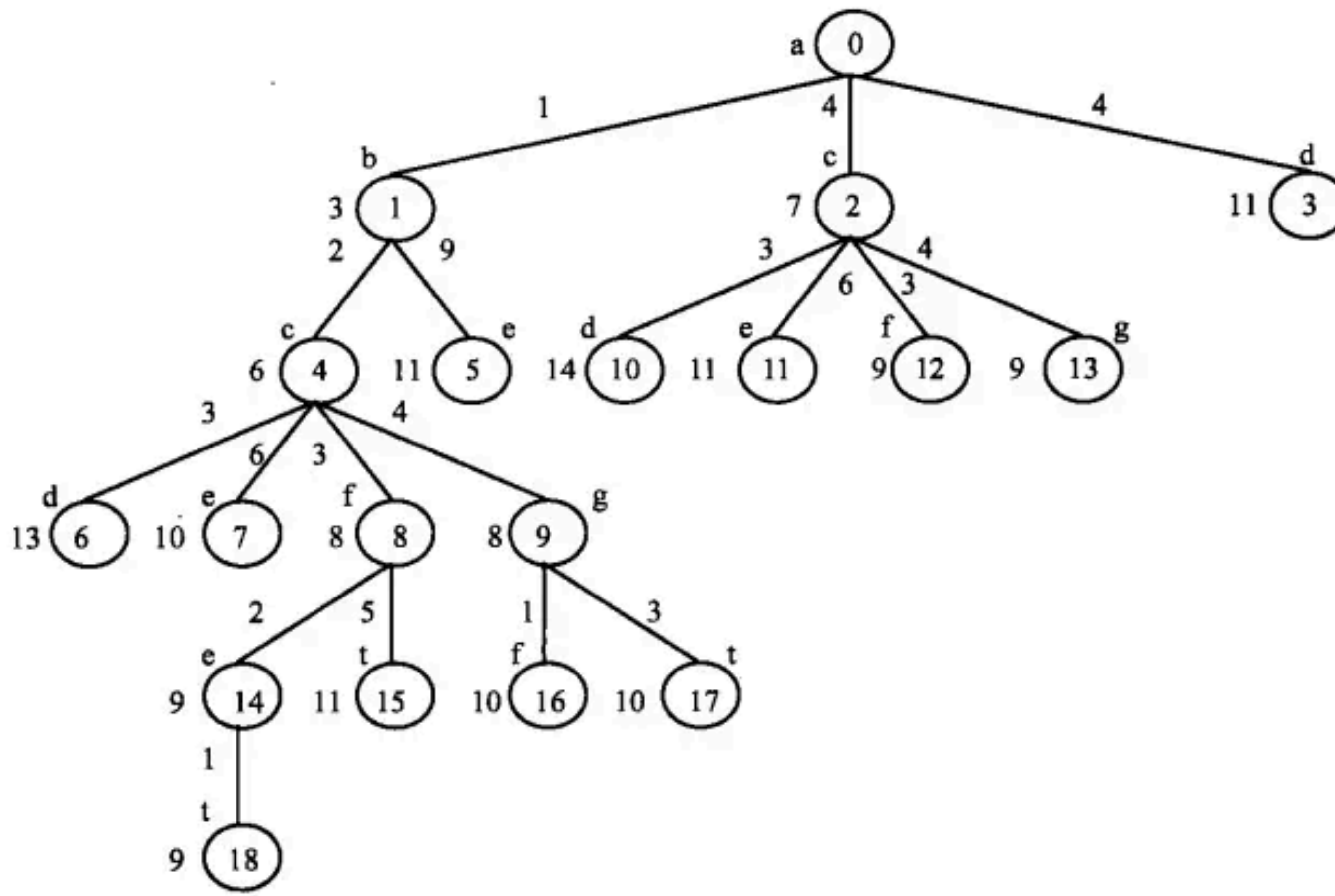
a->c->t距离的下界为：4+min{3,6,3,4} = 7

a->d->t距离的下界为：4+min{7} = 11

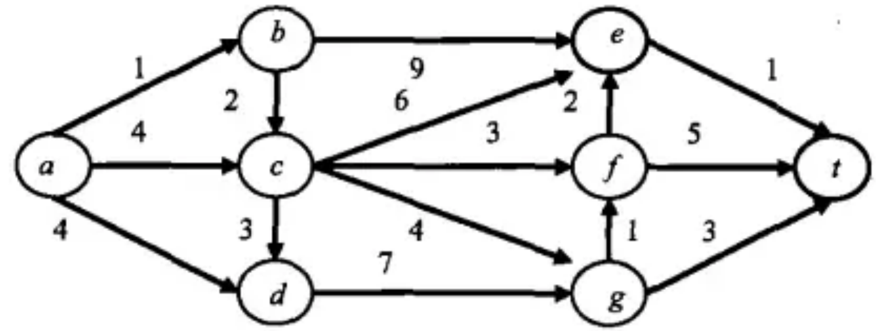
单源最短路径



按照堆进行搜索

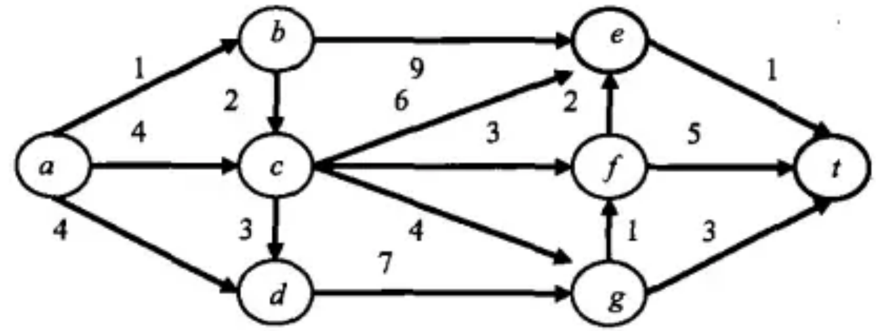


单源最短路径



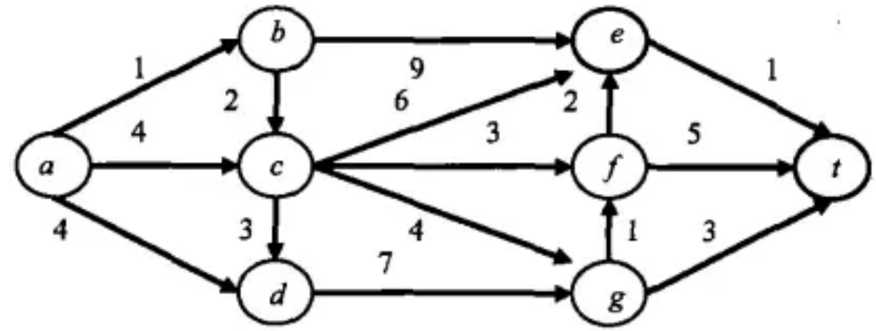
- k=1。
  - 根节点0对应于源点a，有3个邻接顶点b、c、d，其下界为3,7,11，压入堆。
- k=2。
  - 堆中下界3最小，对于的顶点b，也即结点1。从顶点b继续进行搜索。
  - 顶点b的邻接顶点为c和e，其下界为6和11，压入堆。
- k=3。
  - 堆中下界6最小，对应顶点c，也即结点4.从顶点c继续进行搜索。
  - 顶点c邻接顶点d、e、f、g，对应的下界为13,10,8,8，压入堆。
- （**回溯到了k=2**）k=2。
  - 堆中7最小，对应顶点c，也即结点2。从顶点c进行搜索。
  - 顶点c邻接顶点d、e、f、g，对应的下界为14,11,9,9，压入堆。

单源最短路径



- k=4。
  - 堆中8最小，对应顶点f，也即结点8。
  - 顶点f的邻接顶点为e、t，下界分别为9、11，压入栈中。**其中11为一个可行解，将bound置为11.**
- （**回溯到了k=4**）k=4。
  - 堆中8最小，对应顶点g，也即结点9。
  - 顶点g的邻接顶点为f、t，下界都是10。**其中10为一个可行解，将bound置为10.**
- k=5.
  - 堆中9最小，对应顶点e，也即结点14.
  - 顶点e只有一个邻接顶点t，下界为9，从而得到一个可行解，路径长度为9，加入堆中。
    堆中最小的为9，且最后一个结点为t，因此是最优解。

步骤总结



- 初始化
  - 初始化参数，如最优值为∞，
  - 初始化源节点，压入堆中。
- 从堆中取根节点，
  - 如此节点的下界低于当前最优值，则作为当前节点，更新当前节点所有邻节点的下限，压入堆中（可以只压入下限值小于最优值的邻节点），并更新搜索树。
    - 如目的节点在邻节点中，且下限小于最优值，则跟新最优值
  - 否则，算法结束

0-1背包问题

考虑n=4的背包问题（背包承重量为C=10）：

物品	重量	价值	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

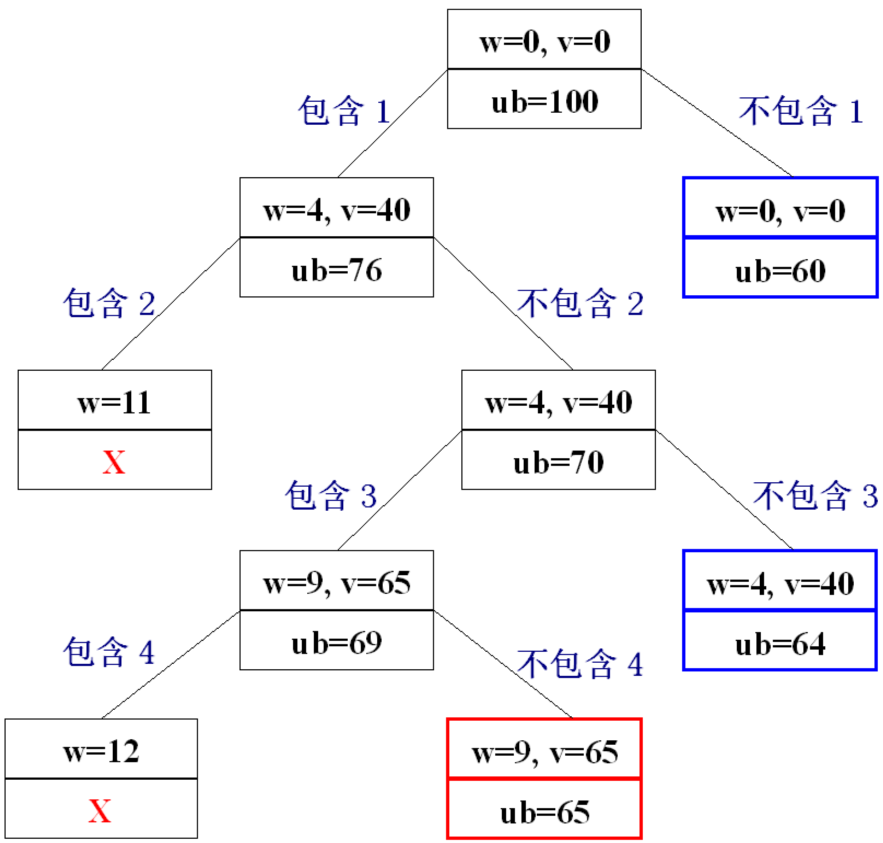
其中物品已经按照“价值-重量比”降序排列，这样会带来处理上的方便。

0-1背包问题

- 为每个搜索节点设置一个上界ub (upper bound)，一个简单方法是：把已经选择的物品总价值v，加上背包剩余承重量C-w与剩下可选择的物品的最高“价值-重量比”的乘积，即$ub = v + (C\text{-}w)\times(v_i/w_i)_{max}$
  - 另外更准确的方法是按照小数背包贪心算法
- 比如，若已装物品1，则价值的上界为40+(10-4)×6=76

物品	重量	价值	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

$$ub = v + (C\text{-}w)\times(v_i/w_i)_{max}$$



分支限界法求解0/1背包问题，其搜索空间如图9.1所示，具体的搜索过程如下：

（1）在根结点1，没有将任何物品装入背包，因此，背包的重量和获得的价值均为0，根据限界函数计算结点1的目标函数值为10×10=100;

（2）在结点2，将物品1装入背包，因此，背包的重量为4，获得的价值为40，目标函数值为40 + (10-4)×6=76，将结点2加入待处理结点表PT（或者堆）中；在结点3，没有将物品1装入背包，因此，背包的重量和获得的价值仍为0，目标函数值为10×6＝60，将结点3加入表PT中；

（3）在表PT中选取目标函数值取得极大的结点2优先进行搜索；

（4）在结点4，将物品2装入背包，因此，背包的重量为11，不满足约束条件，将结点4丢弃；在结点5，没有将物品2装入背包，因此，背包的重量和获得的价值与结点2相同，目标函数值为40＋(10-4)×5=70，将结点5加入表PT中；

（5）在表PT中选取目标函数值取得极大的结点5优先进行搜索；

（6）在结点6，将物品3装入背包，因此，背包的重量为9，获得的价值为65，目标函数值为65 ＋ (10-9)×4=69，将结点6加入表PT中；在结点7，没有将物品3装入背包，因此，背包的重量和获得的价值与结点5相同，目标函数值为40 ＋ (10-4)×4＝64，将结点7加入表PT中；

（7）在表PT中选取目标函数值取得极大的结点6优先进行搜索；

（8）在结点8，将物品4装入背包，因此，背包的重量为12，不满足约束条件，将结点8丢弃；在结点9，没有将物品4装入背包，因此，背包的重量和获得的价值与结点6相同，目标函数值为65;

（9）由于结点9是叶子结点，同时结点9的目标函数值是表PT中的极大值，所以，结点9对应的解即是问题的最优解，搜索结束。

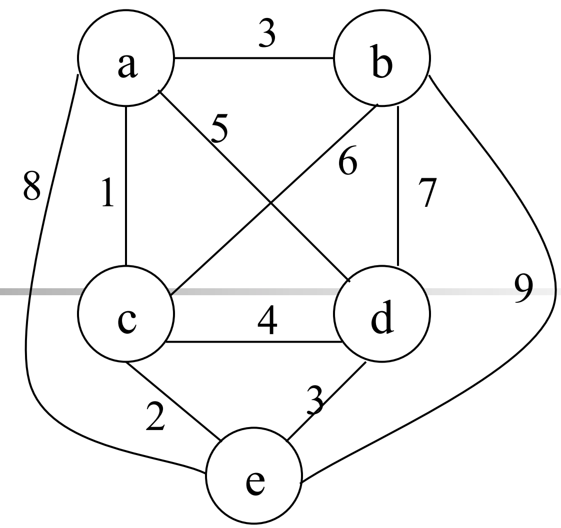
回溯法与分支限界法的对比

方法	对解空间树的搜索方式	存储节点的常用数据结构	节点存储特性	常用应用
回溯法	深度优先搜索	堆栈	活节点的所有可行子节点被遍历后才被从栈中弹出	找出满足约束条件的所有解
分支限界法	广度优先 或 最小消耗(最大效益)优先搜索	队列 或堆	每个节点只有一次成为活节点的机会	找出满足约束条件的一个解或特定意义下的最优解

旅行商问题(TSP)

- 问题描述：一个商人要到n个城市去推销货物，从某城市出发，要求找到一条闭合路径，n个城市都经历一次，最后回到出发点，且使得旅行的花费最小。
- 与背包问题(最大效益)相反，本问题(最小耗费)需要估计每个搜索节点的下界。

一个实例



- 考虑如右图所示的TSP问题。
- 作为演示，附加一个特殊的约束条件：b 在c之前被访问。

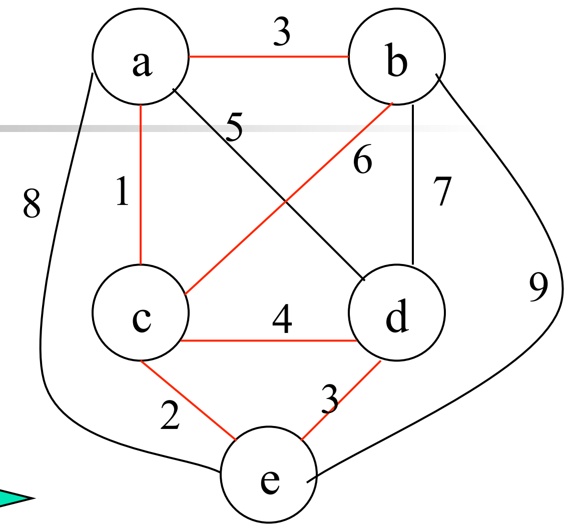
从该节点继续搜索，所获得的收益不会超过ub

0-1背包问题 → 最大收益 → 为搜索节点设置一个上界ub (upper bound) → ub大的节点优先搜索

TSP问题 → 最小耗费 → 为搜索节点设置一个下界lb (lower bound) → lb小的节点优先搜索

从该节点继续搜索，所需要的耗费不会低于lb

为搜索节点设置有效的下界



(1) 简单下界：距离矩阵D的最小元素乘以n即可。

(2) 一个有效的下界：

(2.1) 对每个顶点i，用s_i表示顶点i到最近两个顶点的距离(分别用s_i^1和s_i^2)之和，即$s_i = s_i^1 + s_i^2$; 对s_i求和得到s，然后除以2并向上取整作为搜索节点的下界，即 $lb = \lceil s/2 \rceil$

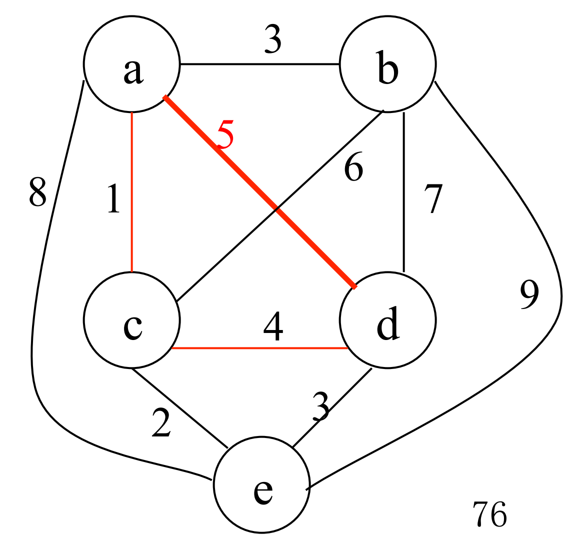
如右图，搜索节点的下界为：

$$lb = \lceil [\underbrace{(1+3)}_{s_a}+\underbrace{(3+6)}_{s_b}+\underbrace{(1+2)}_{s_c}+\underbrace{(3+4)}_{s_d}+\underbrace{(2+3)}_{s_e}]/2 \rceil = 14$$

问题：为什么可以如此设置呢？下页分析。

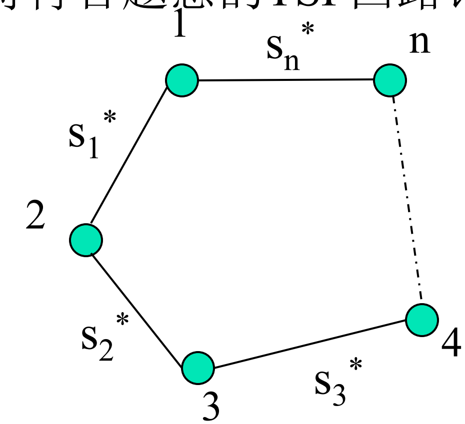
(2.2) 若必须包含某条边，如(a,d): 对于顶点a，其s_a=|(a,d)|+a到其它顶点最短的边；对于顶点d,其s_d= |(a,d)|+d到其它顶点最短的边， 即：

$$lb = \lceil [\underbrace{(1+5)}_{s_a}+\underbrace{(3+6)}_{s_b}+\underbrace{(1+2)}_{s_c}+\underbrace{(3+5)}_{s_d}+\underbrace{(2+3)}_{s_e}]/2 \rceil = 16$$



为什么可以如此设置？

- 给定n个顶点的无向图，边的权值已经给定。将符合题意的TSP回路记为T。
  - 显然，每个顶点在T中只出现一次。
  - 在回路T中的顶点依次编号，1,2,3,…,n
  - 回路T中的边依次记为：$s_1^*, s_2^*, s_3^*, \ldots, s_n^*$



- 显然，我们有：
  - 对于顶点1： $s_1^* + s_n^* >= s_1^1 + s_1^2 = s_1$
  - 对于顶点2： $s_1^* + s_2^* >= s_2^1 + s_2^2 = s_2$
  - …
  - 对于顶点n： $s_{n-1}^* + s_n^* >= s_n^1 + s_n^2 = s_n$

＋

$$2(s_1^* + s_2^* + \ldots + s_n^*) >= s_1 + \ldots + s_n = s$$

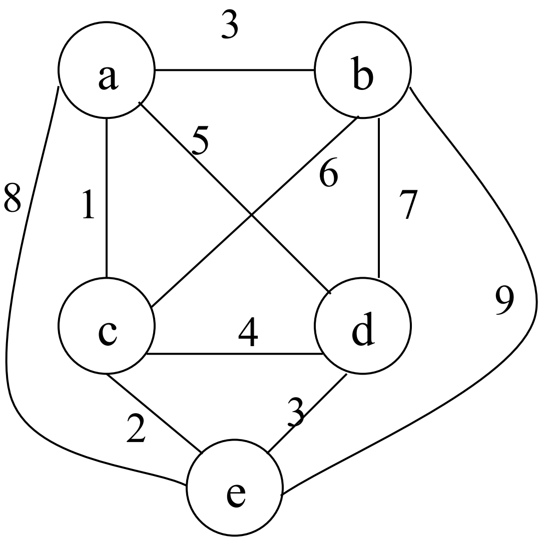
$$(s_1^* + s_2^* + \ldots + s_n^*) >= s/2$$

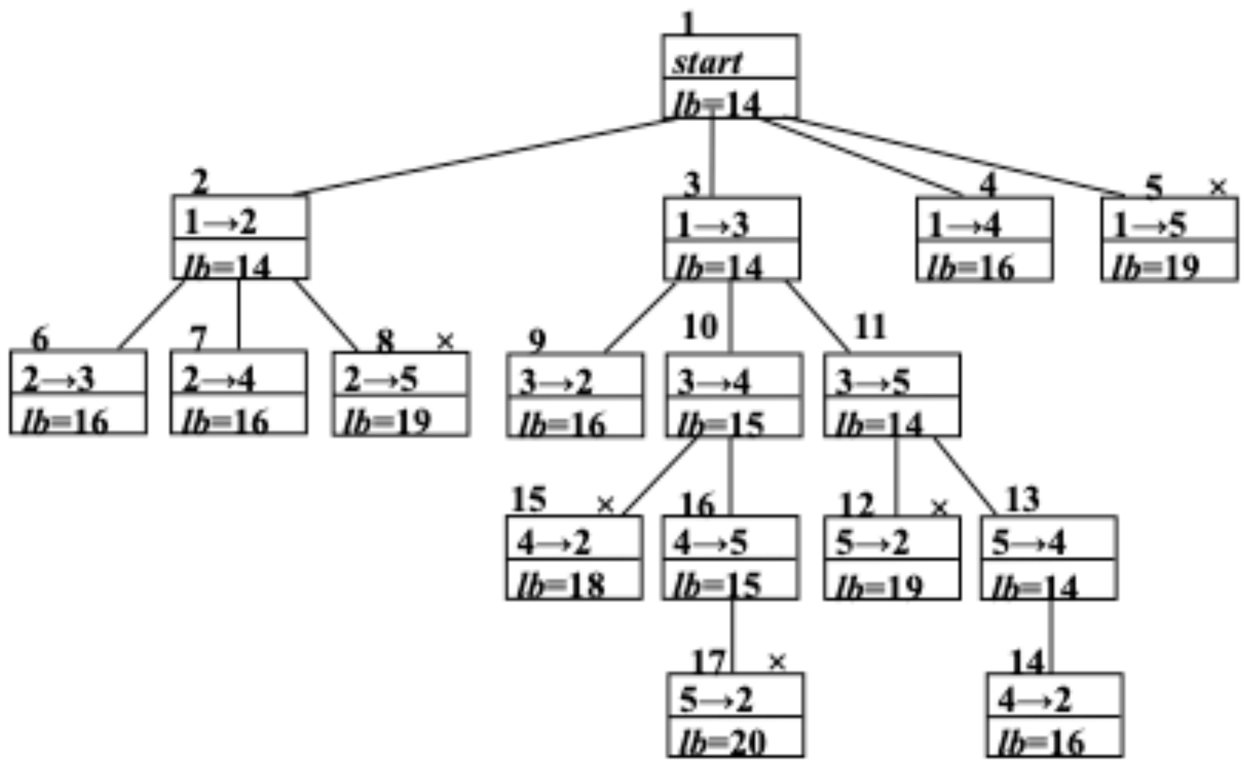
又因为$(s_1^* + s_2^* + \ldots + s_n^*)$是整数，必定有

$$(s_1^* + s_2^* + \ldots + s_n^*) >= \lceil s/2 \rceil$$

所以，可以取$lb = \lceil s/2 \rceil$

搜索树


a
b
c
d
e
3
5
6
8
1
7
9
4
2
3


1 start lb=14
2 1→2 lb=14
3 1→3 lb=14
4 1→4 lb=16
5 × 1→5 lb=19
6 2→3 lb=16
7 2→4 lb=16
8 × 2→5 lb=19
9 3→2 lb=16
10 3→4 lb=15
11 3→5 lb=14
15 × 4→2 lb=18
16 4→5 lb=15
12 × 5→2 lb=19
13 5→4 lb=14
17 × 5→2 lb=20
14 4→2 lb=16

- 为了进一步提高效率可以在TSP问题中确定一个上界

采用贪心法求得近似解为1→3→5→4→2→1，其路径长度为1+2+3+7+3=16，这可以作为TSP问题的上界。

- 这样结合前面得到的下界，得到了目标函数的界[14, 16]。

- 需要强调的是，这个解并不是一个合法的选择（可能没有构成哈密顿回路），它仅仅给出了一个参考下界。

应用分支限界法求解图9.4所示无向图的TSP问题，其搜索空间如图9.5所示，具体的搜索过程如下（加黑表示该路径上已经确定的边）：

（1）在根结点1，根据限界函数计算目标函数的值为 $lb=((1+3)+(3+6)+(1+2)+(3+4)+(2+3))/2=14$；

（2）在结点2，从城市1到城市2，路径长度为3，目标函数的值为 $((1+3)+(3+6)+(1+2)+(3+4)+(2+3))/2=14$，将结点2加入待处理结点表PT中；在结点3，从城市1到城市3，路径长度为1，目标函数的值为 $((1+3)+(3+6)+(1+2)+(3+4)+(2+3))/2=14$，将结点3加入表PT中；在结点4，从城市1到城市4，路径长度为5，目标函数的值为 $((1+5)+(3+6)+(1+2)+(3+5)+(2+3))/2=16$，将结点4加入表PT中；在结点5，从城市1到城市5，路径长度为8，目标函数的值为 $((1+8)+(3+6)+(1+2)+(3+4)+(2+8))/2=19$，超出目标函数的界，将结点5丢弃；

（3）在表PT中选取目标函数值极小的结点2优先进行搜索；
（4）在结点6，从城市2到城市3，目标函数值为
((1+3)+(3+6)+(1+6)+(3+4)+(2+3))/2=16，将结点6加入表PT中；
在结点7，从城市2到城市4，目标函数值为
((1+3)+(3+7)+(1+2)+(3+7)+(2+3))/2=16，将结点7加入表PT中；
在结点8，从城市2到城市5，目标函数值为
((1+3)+(3+9)+(1+2)+(3+4)+(2+9))/2=19，超出目标函数的界，将结点8丢弃；
（5）在表PT中选取目标函数值极小的结点3优先进行搜索；
（6）在结点9，从城市3到城市2，目标函数值为
((1+3)+(3+6)+(1+6)+(3+4)+(2+3))/2=16，将结点9加入表PT中；
在结点10，从城市3到城市4，目标函数值为
((1+3)+(3+6)+(1+4)+(3+4)+(2+3))/2=15，将结点10加入表PT中；
在结点11，从城市3到城市5，目标函数值为
((1+3)+(3+6)+(1+2)+(3+4)+(2+3))/2=14，将结点11加入表PT中；

（7）在表PT中选取目标函数值极小的结点11优先进行搜索；

（8）在结点12，从城市5到城市2，目标函数值为((1+3)+(3+9)+(1+2)+(3+4)+(2+9))/2=19，超出目标函数的界，将结点12丢弃；

在结点13，从城市5到城市4，目标函数值为((1+3)+(3+6)+(1+2)+(3+4)+(2+3))/2=14，将结点13

加入表PT中；

（9）在表PT中选取目标函数值极小的结点13优先进行搜索；

（10）在结点14，从城市4到城市2，目标函数值为((1+3)+(3+7)+(1+2)+(3+7)+(2+3))/2=16，最后从城市2回到城市1，目标函数值为((1+3)+(3+7)+(1+2)+(3+7)+(2+3))/2=16，由于结点14为叶子结点，得到一个可行解，其路径长度为16；

（11）在表PT中选取目标函数值极小的结点10优先进行搜索；

（12）在结点15，从城市4到城市2，目标函数的值为((1+3)+(3+7)+(1+4)+(7+4)+(2+3))/2=18，超出目标函数的界，将结点15丢弃；

在结点16，从城市4到城市5，目标函数值为((1+3)+(3+6)+(1+4)+(3+4)+(2+3))/2=15，将结点16加入表PT中；

（13）在表PT中选取目标函数值极小的结点16优先进行搜索；

（14）在结点17，从城市5到城市2，目标函数的值为((1+3)+(3+9)+(1+4)+(3+4)+(9+3))/2=20，超出目标函数的界，将结点17丢弃；

（15）表PT中目标函数值均为16，且有一个是叶子结点14，所以，结点14对应的解1→3→5→4→2→1即是TSP问题的最优解，搜索过程结束。

(1, 2)14	(1, 3)14	(1, 4)16

(a) 扩展根结点后的状态

(1, 3)14	(1, 4)16	(1, 2, 3)16	(1, 2, 4)16

(b) 扩展结点2后的状态

(1, 4)16	(1, 2, 3)16	(1, 2, 4)16	(1, 3, 2)16	(1, 3, 4)15	(1, 3, 5)14

(c) 扩展结点3后的状态

(1, 4)16	(1, 2, 3)16	(1, 2, 4)16	(1, 3, 2)16	(1, 3, 4)15	(1, 3, 5, 4)14

(d) 扩展结点11后的状态

(1, 4)16	(1, 2, 3)16	(1, 2, 4)16	(1, 3, 2)16	(1, 3, 4)15	(1, 3, 5, 4, 2)16

(e) 扩展结点13后的状态

(1, 4)16	(1, 2, 3)16	(1, 2, 4)16	(1, 3, 2)16	(1, 3, 5, 4, 2)16	(1, 3, 4, 5)15

(f) 扩展结点10后的状态

(1, 4)16	(1, 2, 3)16	(1, 2, 4)16	(1, 3, 2)16	(1, 3, 5, 4, 2)16	(1, 3, 4, 5)15

(g) 扩展结点16后的状态，最优解为1→3→5→4→2→1

图9.6 TSP问题最优解的确定

算法9.1——TSP问题

1．根据限界函数计算目标函数的下界down；采用贪心法得到上界up；
2．将待处理结点表PT初始化为空；
3．for (i=1; i<=n; i++)
x[i]=0;
4．k=1; x[1]=1; //从顶点1出发求解TSP问题
5．while (k>=1)
5.1 i=k+1;
5.2 x[i]=1;
5.3 while (x[i]<=n)
5.3.1 如果路径上顶点不重复，则
5.3.1.1 根据式9.2计算目标函数值lb；
5.3.1.2 if (lb<=up) 将路径上的顶点和lb值存储在表PT中；
5.3.2 x[i]=x[i]+1;
5.4 若i=n且叶子结点的目标函数值在表PT中最小
则将该叶子结点对应的最优解输出；
5.5否则，若i=n，则在表PT中取叶子结点的目标函数值最小的结点lb，
令up=lb，将表PT中目标函数值lb超出up的结点删除；
5.6 k=表PT中lb最小的路径上顶点个数；

任务分配问题

任务分配问题要求把n项任务分配给n个人，每个人完成每项任务的成本不同，要求分配总成本最小的最优分配方案。如图9.10所示是一个任务分配的成本矩阵。

$$
C= \begin{array}{cc} & \begin{array}{cccc} \text{任务1} & \text{任务2} & \text{任务3} & \text{任务4} \end{array} \\ & \begin{pmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{pmatrix} \begin{array}{l} \text{人员}a \\ \text{人员}b \\ \text{人员}c \\ \text{人员}d \end{array} \end{array}
$$

图　任务分配问题的成本矩阵

求最优分配成本的上界和下界

考虑任意一个可行解，例如矩阵中的对角线是一个合法的选择，表示将任务1分配给人员*a*、任务2分配给人员*b*、任务3分配给人员*c*、任务4分配给人员*d*，其成本是9+4+1+4=18；或者应用贪心法求得一个近似解：将任务2分配给人员*a*、任务3分配给人员*b*、任务1分配给人员*c*、任务4分配给人员*d*，其成本是2+3+5+4=14。显然，14是一个更好的上界。

为了获得下界，考虑人员*a*执行所有任务的最小代价是2，人员*b*执行所有任务的最小代价是3，人员*c*执行所有任务的最小代价是1，人员*d*执行所有任务的最小代价是4。因此，将每一行的最小元素加起来就得到解的下界，其成本是2+3+1+4=10。需要强调的是，这个解并不是一个合法的选择（3和1来自于矩阵的同一列），它仅仅给出了一个参考下界，这样，最优值一定是[10, 14]之间的某个值。

设当前已对人员1～i分配了任务，并且获得了成本v，则限界函数可以定义为：

$$lb = v + \sum_{k=i+1}^{n} 第k行的最小值$$

应用分支限界法求解图（87页）所示任务分配问题，对解空间树的搜索如图（93页）所示，具体的搜索过程如下：

（1）在根结点1，没有分配任务，根据限界函数估算目标函数值为2+3+1+4=10；

（2）在结点2，将任务1分配给人员*a*，获得的成本为9，目标函数值为9 + (3+1+4)=17，超出目标函数的界[10, 14]，将结点2丢弃；在结点3，将任务2分配给人员*a*，获得的成本为2，目标函数值为2 + (3+1+4)=10，将结点3加入待处理结点表PT中；在结点4，将任务3分配给人员*a*，获得的成本为7，目标函数值为7 + (3+1+4)=15，超出目标函数的界[10， 14]，将结点4丢弃；在结点5，将任务4分配给人员*a*，获得的成本为8，目标函数值为8 + (3+1+4)=16，超出目标函数的界[10, 14]，将结点5丢弃；

（3）在表PT中选取目标函数值极小的结点3优先进行搜索；

（4）在结点6，将任务1分配给人员*b*，获得的成本为2+6=8，目标函数值为8+(1+4)＝13，将结点6加入表PT中；在结点7，将任务3分配给人员*b*，获得的成本为2+3=5，目标函数值为5+(1+4)＝10，将结点7加入表PT中；在结点8，将任务4分配给人员*b*，获得的成本为2+7=9，目标函数值为9+(1+4)＝14，将结点8加入表PT中；

（5）在表PT中选取目标函数值极小的结点7优先进行搜索；

（6）在结点9，将任务1分配给人员*c*，获得的成本为5+5=10，目标函数值为10+4=14，将结点9加入表PT中；在结点10，将任务4分配给人员*c*，获得的成本为5+8=13，目标函数值为13+4=17，超出目标函数的界[10, 14]，将结点10丢弃；

（7）在表PT中选取目标函数值极小的结点6优先进行搜索；

（8）在结点11，将任务3分配给人员c，获得的成本为8+1=9，目标函数值为9+4=13，将结点11加入表PT中；在结点12，将任务4分配给人员c，获得的成本为8+8=16，目标函数值为16+4=20，超出目标函数的界[10, 14]，将结点12丢弃；

（9）在表PT中选取目标函数值极小的结点11优先进行搜索；

（10）在结点13，将任务4分配给人员d，获得的成本为9+4=13，目标函数值为13，由于结点13是叶子结点，同时结点13的目标函数值是表PT中的极小值，所以，结点13对应的解即是问题的最优解，搜索结束。

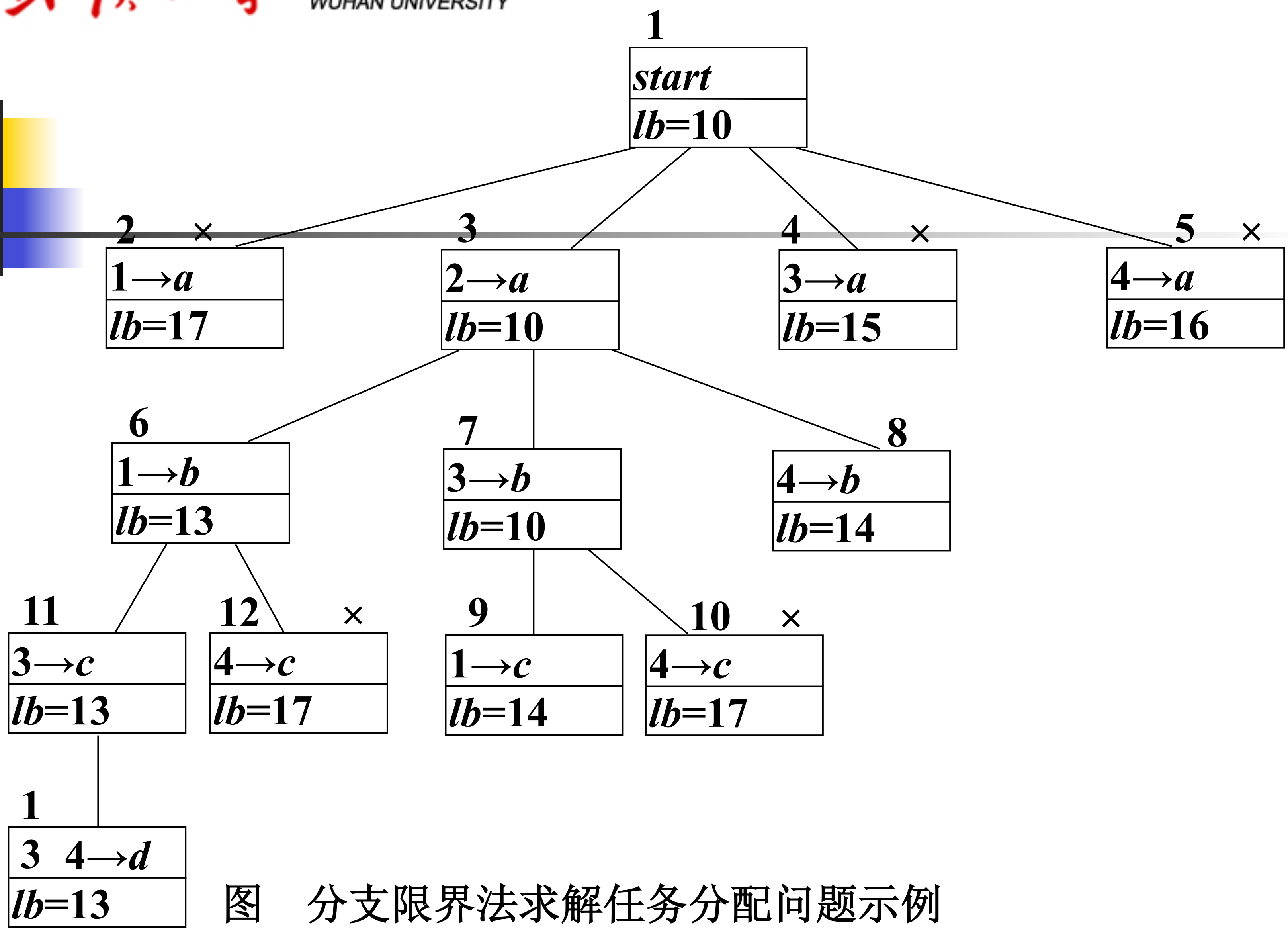


图　分支限界法求解任务分配问题示例

(×表示该结点被丢弃，结点上方的数组表示搜索顺序)

为了在搜索过程中构建搜索经过的树结构，设一个表ST，在表PT中取出最小值结点进行扩充时，将最小值结点存储到表ST中，表PT和表ST的数据结构为(人员i-1分配的任务,<任务k, 人员i>lb)

PT	(0,<2, *a*>10)			
ST				

(a) 扩展根结点后的状态

PT	(2,<1, *b*>13)	(2,<3, *b*>10)	(2,<4, *b*>14)	
ST	(0,<2, *a*>10)			

(b) 扩展结点3后的状态

PT	(2,<1, *b*>13)	(2,<4, *b*>14)	(3,<1, *c*>14)	
ST	(0,<2, *a*>10)	(2,<3, *b*>10)		

(c) 扩展结点7后的状态

PT	(2,<4, *b*>14)	(3,<1, *c*>14)	(1,<3, *c*>13)	
ST	(0,<2, *a*>10)	(2,<3, *b*>10)	(2,<1, *b*>13)	

(d) 扩展结点6后的状态

PT	(2,<4, *b*>14)	(3,<1, *c*>14)	(3,<4, *d*>13)	
ST	(0,<2, *a*>10)	(2,<3, *b*>10)	(2,<1, *b*>13)	(1,<3, *c*>13)

(e) 扩展结点11后的状态，最优解为2→*a* 1→*b* 3→*c* 4→*d*

图9.12 任务分配问题最优解的确定

回溯过程是：

(3,<4, *d*>13)→(1,<3, *c*>13)→(2,<1, *b*>13)→(0,<2, *a*>10) 。

伪代码

法9.3——任务分配问题

1．根据限界函数计算目标函数的下界down；采用贪心法得到上界up；

2．将待处理结点表PT初始化为空；

3．for (i=1; i<=n; i++)

  x[i]=0;

4．k=1; i=0;   //为第k个人分配任务，i为第k-1个人分配的任务

5．while (k>=1)

 5.1 x[k]=1;

 5.2 while (x[k]<=n)

  5.2.1 如果人员k分配任务x[k]不发生冲突，则

   5.2.1.1 根据式9.4计算目标函数值lb;

   5.2.1.2 若lb<=up，则将i,<x[k], k>lb存储在表PT中；

  5.2.2 x[k]=x[k]+1;

 5.3 如果k= =n且叶子结点的lb值在表PT中最小，

  则输出该叶子结点对应的最优解；

 5.4 否则，如果k= =n且表PT中的叶子结点的lb值不是最小，则

  5.4.1 up=表PT中的叶子结点最小的lb值;

  5.4.2 将表PT中超出目标函数界的结点删除；

 5.5 i=表PT中lb最小的结点的x[k]值；

 5.6 k=表PT中lb最小的结点的k值；k++;