算法设计与分析

5. 回溯法

学习要点与要求

- 掌握与理解回溯法的DFS搜索策略与方法

（1）掌握递归回溯

（2）理解迭代回溯编程技巧

- 掌握用回溯法解题的算法框架，了解搜索过程

（1）子集树算法框架

（2）排列树算法框架

- 通过学习典型范例，掌握回溯法的设计策略
- （1）装载问题　（2）批处理作业调度（3）n后问题
- （4）0-1背包问题　（5）最大团问题
- （6）图的m着色问题（7）旅行售货员问题

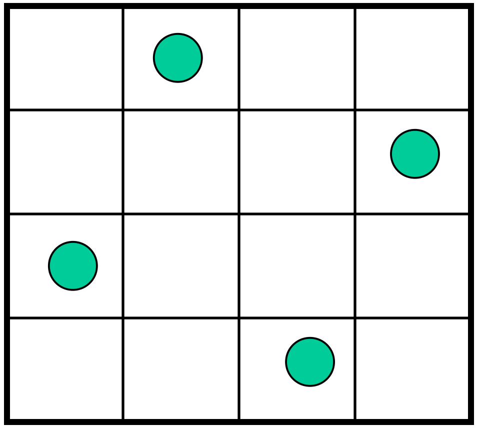
引 言

n皇后问题

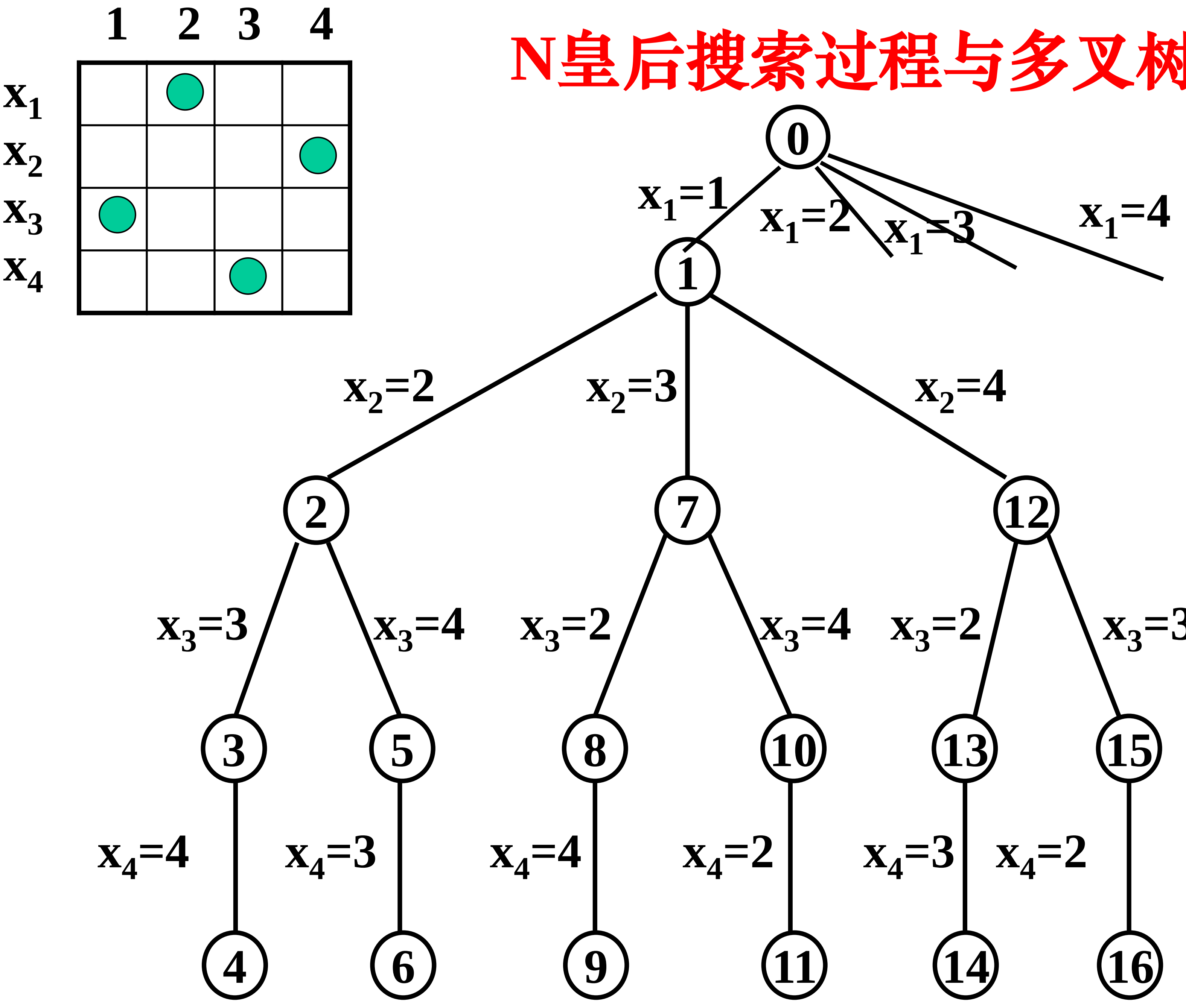
国际象棋中的“皇后”在横向、直向、和斜向都能走步和吃子，问在n×n 格的棋盘上如何摆上n个皇后而使她们都不能相互吃掉?

穷举法:

n=4时，有n！=24中情况



N皇后搜索过程与多叉树



5.1 回溯法的算法框架

问题的解空间（1）

1. 解向量：问题的解用向量表示

$(x_1, x_2, \ldots, x_k)$　其中$k \le n$，n为问题的规模。

2. 约束条件

1）显式约束：对分量x_i的取值的明显限定。

2）隐式约束：为满足问题的解而对不同分量之间施加的约束。

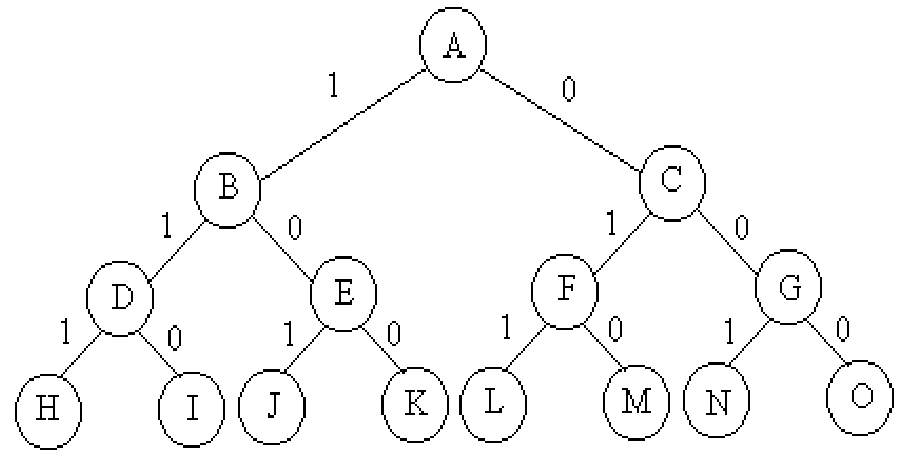
3. 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

问题的解空间（2）

4、状态空间树

用于形象描述解空间的树。

n=3时的0-1背包问题用完全二叉树表示的解空间



5、目标函数与最优解

1. 目标函数：衡量问题解的“优劣”标准。

2. 最优解：使目标函数取极（大/小）值的解。

搜索状态空间树的两种策略

1. 以深度优先方式系统搜索问题的解------回溯法

2. 以广度优先方式搜索问题的解------分支-限界法

- 几种搜索过程中涉及的结点:

  1．扩展结点：一个正在产生儿子的结点称为扩展结点

  2．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

  3．死结点：一个所有儿子已经产生的结点称做死结点

回溯法

1．基本方法：利用限界函数来避免生成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，避免无效搜索。

2．限界函数：用于剪枝

(1)约束函数：某个满足条件的表达式或关系式。

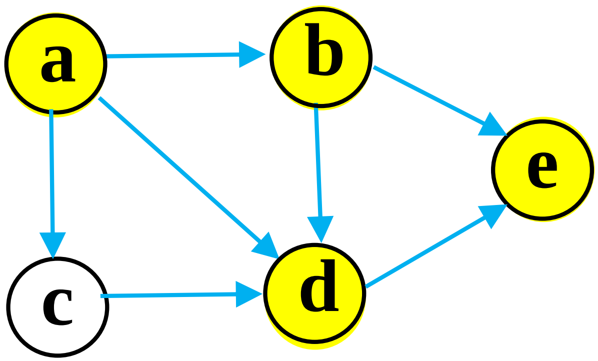
不真时用于在扩展结点处剪去不满足约束的子树；

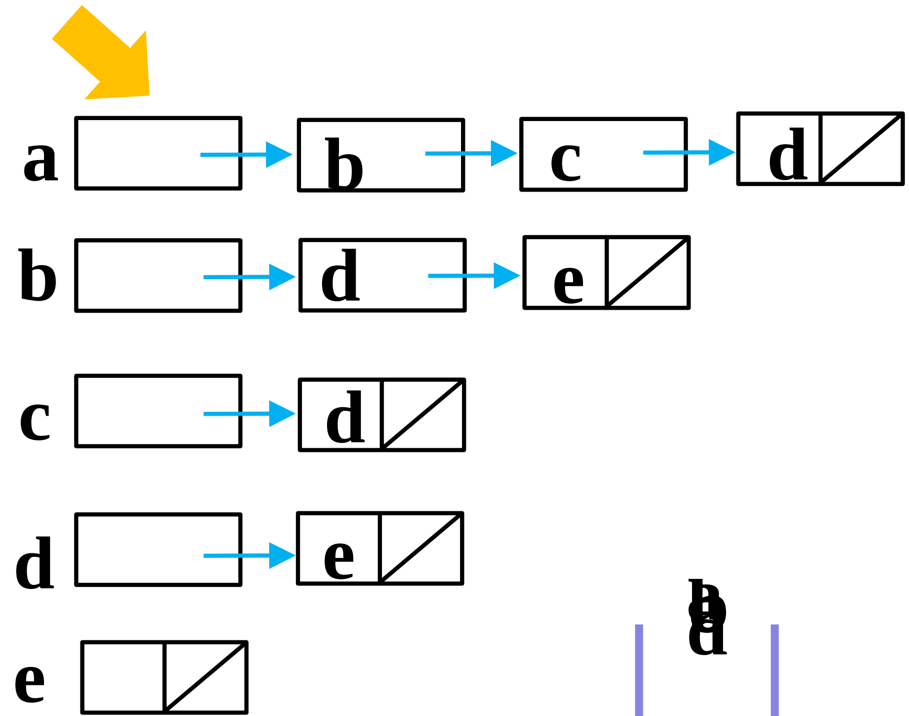
(2)限界函数：某个函数表达式或关系式。

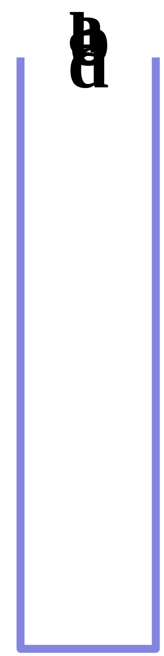
不真时，用于剪去得不到最优解的子树。

3．回溯法：具有限界函数的深度优先搜索方法

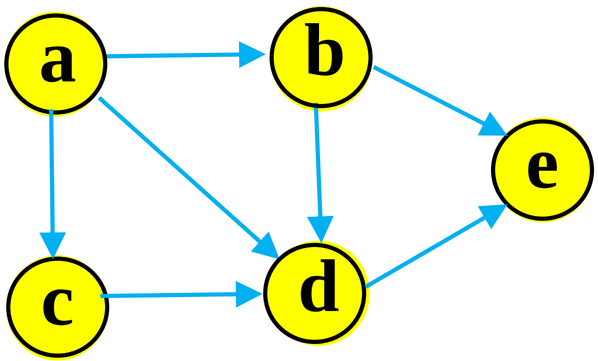
DFS算法回顾

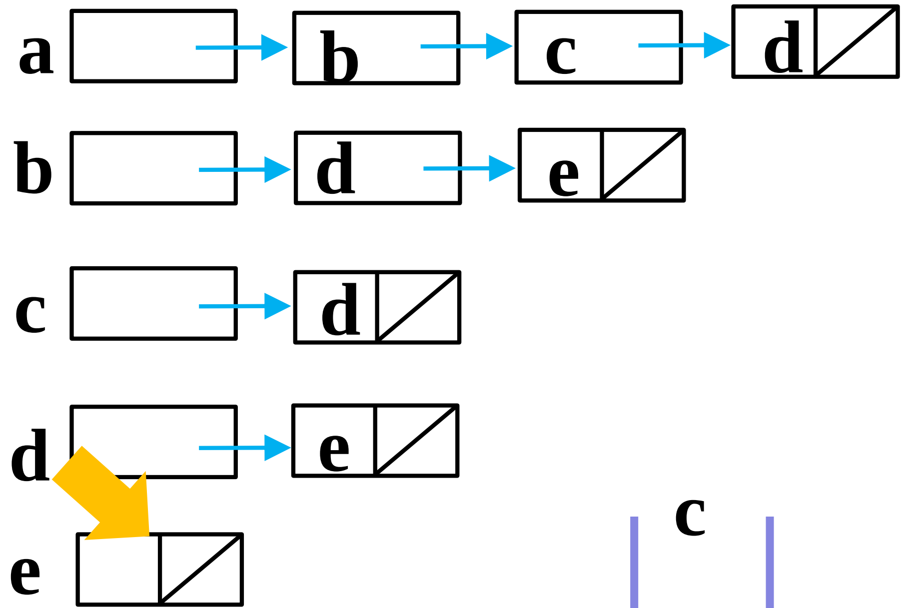


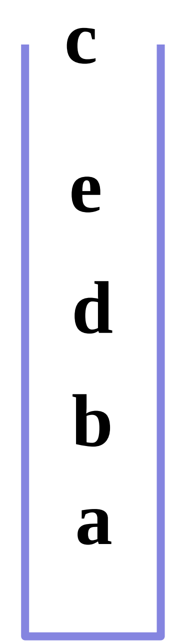




DFS算法回顾





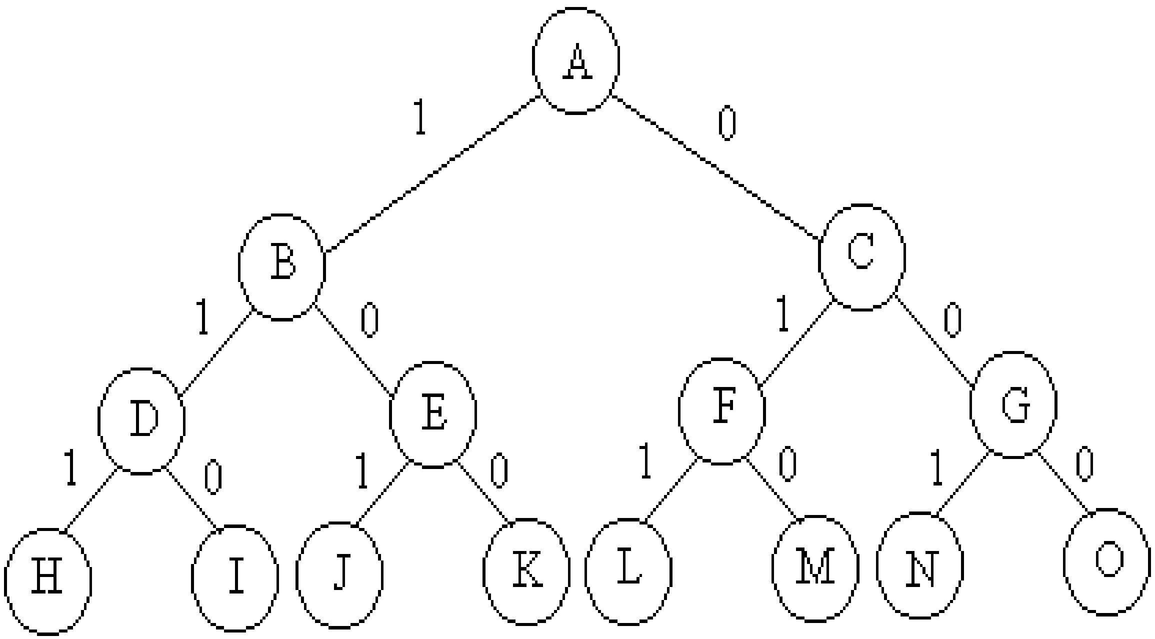


回溯法的基本思想

1. 以深度优先方式搜索解空间。
2. 开始时，根结点为活结点，也是当前的扩展结点。
3. 对扩展结点，寻找儿子结点：
   如找到新结点，新结点成为活结点并成为扩展结点。转3；
   如找不到新结点，当前结点成为死结点，并回退到最近的一个活结点，使它成为扩展结点。转3；
4. 搜索继续进行，直到找到所求的解或解空间中已无活结点时为止。

搜索过程举例--0-1背包问题

- n=3时的0-1背包问题：
  c=30,w=[16,15,15],v=[45,25,25]



回溯法解题步骤

(1)针对所给问题，定义问题的解空间；

(2)确定合适的解空间结构；

(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，直到找到所求的解或解空间中已无活结点时为止。

回溯法形式描述-递归算法

void BACKTRACK(int t) { //n为递归深度,t为当前深度

if (t > n) 返回;

else

while (存在合适的x_t)

{ if ($x_1,\dots,x_t$)是解

输出解$x_1,\dots,x_t$;

BACKTRACK(t + 1);

}

}

另一种方式

for(int i=f(n,t);i<=g(n,t);i++){

x[t]=h(i);

若（x_t满足约束和限界条件）

BACKTRACK(t + 1)

}

回溯法形式描述-非递归算法

迭代回溯

void BACKTRACK(int n) {

int t= 1;

while (t > 0){

if (有x_t，满足 约束C($x_1,\ldots,x_t$) &&限界Bound($x_1,\ldots,x_t$))

{ if ((($x_1,\ldots,x_t$)是问题的一个“解”)

则 输出解($x_1,\ldots,x_t$);

else t= t + 1 //

}

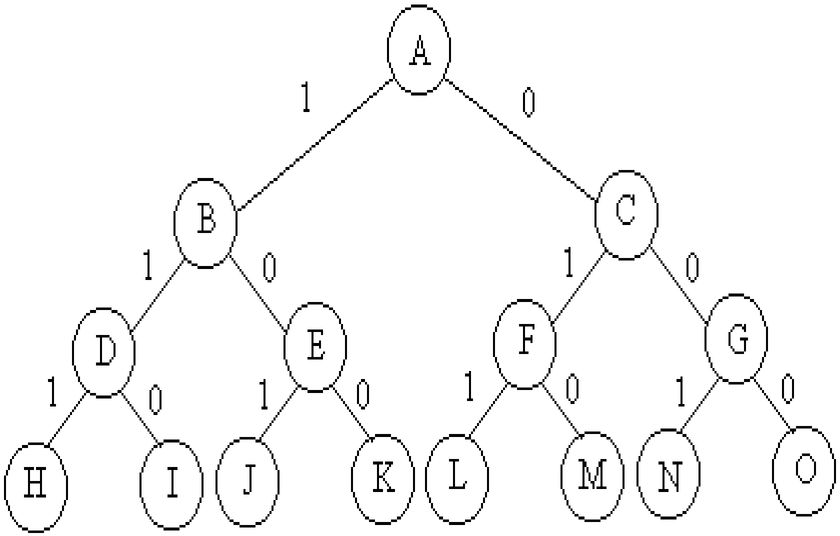
else t= t - 1 ; //

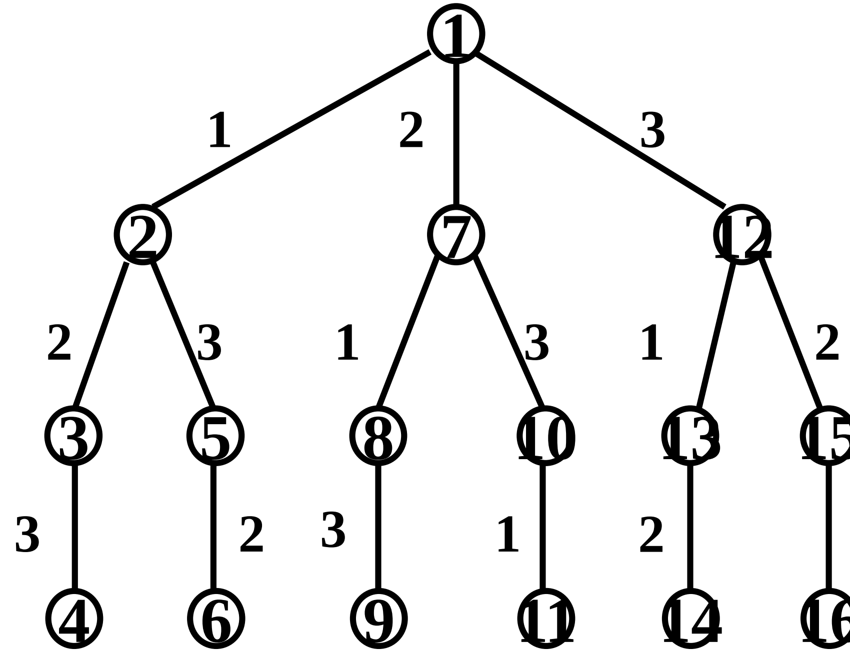
}

}

```
if (f(n,t)<=g(n,t))
    for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);
        if (Constraint(t)&&Bound(t)) {
            if (solution(t)) output(x);
            else t++;
        }
    }
```

子集树与排列树





遍历子集树需O(2^n)计算时间

遍历排列树需要O(n!)计算时间

注意：2^n<< n!

子集树与排列树算法框架-递归形式

子集树

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

  backtrack(t+1);
      }
}
```

排列树

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

legal(t)) **为Constraint**(t)&&**Bound**(t)

例：n=3的排列生成

初始排列x[]=1 2 3
backtrack (1): // t=1
  i=1..3 分别做{ swap(x[t], x[i]);
    做backtrack(2); // t=2
    swap(x[t], x[i]); } 分别讨论

i=1时，开始x[]=1 2 3，t=1,未交换仍为123，做 backtrack(2)后，分别输出123与132
i=2时，x[]=123经交换后为213 做backtrack(2)后，分别输出213与231
i=3时，x[]=123经交换后为321 做backtrack(2)后，分别输出312与321

对排列x[]=x1, x2, x3
backtrack (2): // t=2
  j=2..3做{
    swap(x[2], x[j]);
    backtrack(3); // t=3
    swap(x[2], x[j]);
  }

分别讨论:
j=2时， x[]仍为x1, x2, x3 经backtrack(3)后 x[]仍为x1 x2 x3并输出
j=3时， x[]变为x1 x3 x2 经backtrack(3)后 x[]仍为x1 x3 x2并输出

5.2 装载问题

装载问题描述

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

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

例子

1. n=3, c1=c2=50, w=[10,40,40], 有解
2. n=3, c1=c2=50, w=[20,40,40], 无解

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集合的一个子集，使该子集中集装箱重量之和最接近c_1。
- 装载问题等价于以下特殊的0-1背包问题。但跟第四章的装载问题不同。

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

$$\sum_{i=1}^{n} w_i x_i \le c_1$$

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

装载问题的回溯法

- 解空间：子集树，完全二叉树
- 设定解向量：$(x_1, x_2, \ldots, x_n)$

约束条件

1. 显式约束：$x_i = 0, 1 \quad (i=1, 2, \ldots, n)$

2. 隐式约束：无

约束函数（整体）： $\sum_{i=1}^{n} w_i x_i \le c_1$

程序框架

```
void backtrack (int i) // 搜索第i层结点
{
    if (i > n)  { // 到达叶结点
                                                  //修正最优值
    }
     if (cw + w[i] <= c) {// 搜索左子树

          backtrack(i + 1);

     }          //回退
     backtrack(i + 1); // 搜索右子树
}
```

设cw：是当前载重量
bestw：当前最优载重量

约束函数：

cw + w[i] <= c

程序框架

```
void backtrack (int i) // 搜索第i层结点
{
    if (i > n)  { // 到达叶结点
          if (cw>bestw) bestw=cw; //修正最优值
          return;
    }
    if (cw + w[i] <= c) {// 搜索左子树
          cw += w[i];
          backtrack(i + 1);
          cw -= w[i];
    }        //回退
    backtrack(i + 1); // 搜索右子树
}
```

设cw：是当前载重量
bestw：当前最优载重量

约束函数：

$cw + w[i] <= c$

剪枝处理—求最优值

- cw：是当前载重量，bestw：当前最优载重量
- r：剩余集装箱的重量，限界函数：cw+r > bestw

```
void backtrack (int i)  {// 搜索第i层结点
      if (i > n) { // 到达叶结点
          if (cw>bestw)  bestw=cw;
         return;}

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

   }
```

剪枝处理—求最优值

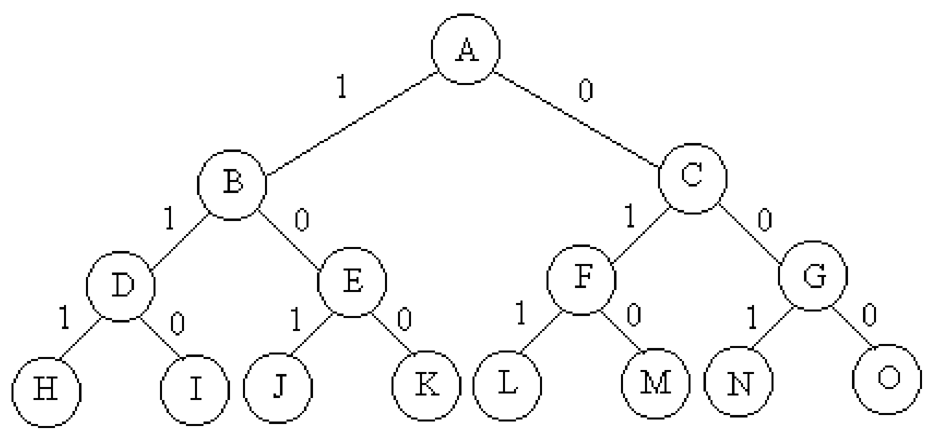
- cw：是当前载重量，bestw：当前最优载重量
- r：剩余集装箱的重量，限界函数：cw+r > bestw

```
void backtrack (int i)  {// 搜索第i层结点
      if (i > n) { // 到达叶结点
          if (cw>bestw)  bestw=cw;
         return;}
       r -= w[i];
       if (cw + w[i] <= c) {// 搜索左子树
           cw += w[i];
           backtrack(i + 1);
           cw -= w[i];      }
       if (cw + r > bestw     )  { // 搜索右子树
           backtrack(i + 1);   }
           r += w[i];
   }
```

求最优解—记录路径

```
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;return; }
    r -= w[i];
    if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];      }
    if (cw + r > bestw)  {
      x[i] = 0;  // 搜索右子树
      backtrack(i + 1);      }
    r += w[i];
  }
```

```
if (cw>bestw) {
   for(int j=1; j<=n; j++)
         bestx[j]=x[j];
         bestw=cw;
}
```



迭代回溯

- **将回溯法表示成非递归的形式**
- **算法Maxloading所需的计算时间仍为$O(2^n)$。**

- **优化：修改递归回溯程序，使所需的计算时间仍为$O(2^n)$。**

5.3 批处理作业调度

批处理作业调度问题描述

- 给定n个作业的集合$J=\{J_1,J_2,\ldots,J_n\}$。每个作业必须先由机器M1处理，然后由机器M2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M2上完成处理的时间和称为该作业调度的完成时间和。

$$f=\sum_{i=1}^{n} F_{2i}$$

- 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。

实例

t_{ji}	M1	M2
J_1	2	1
J_2	3	1
J_3	2	3

3

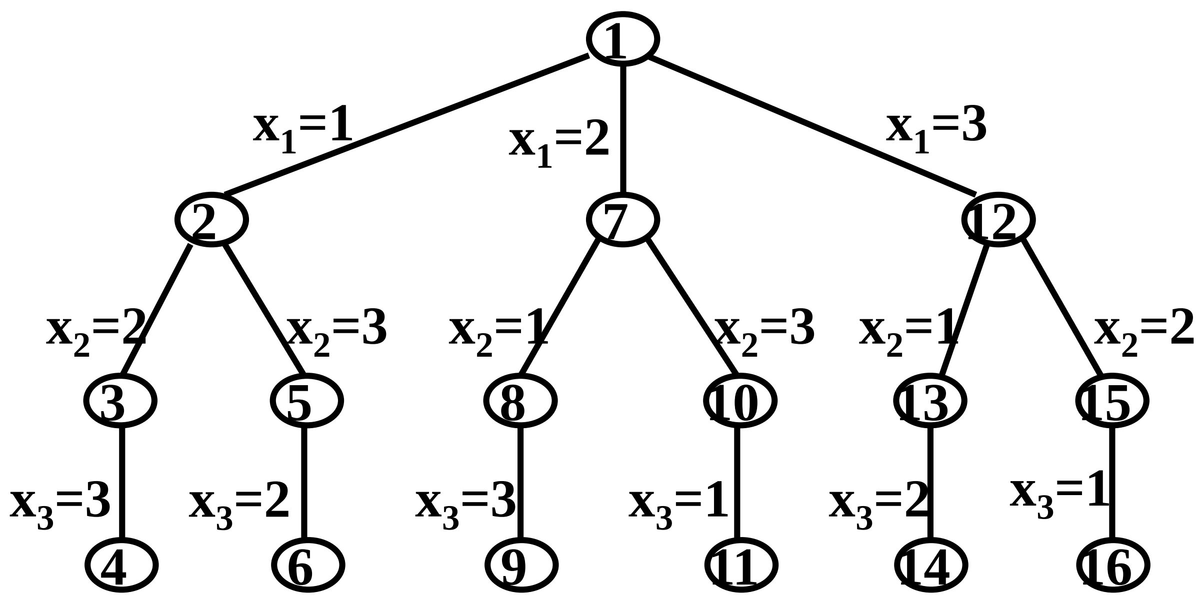
6

- 共有3！=6种调度方案，例
- 调度123 f=3+6+10=19
- 调度132 f=3+7+8=18
- 调度213 f=4+6+10=20

最佳调度方案是1,3,2，其完成时间和为18。

算法设计

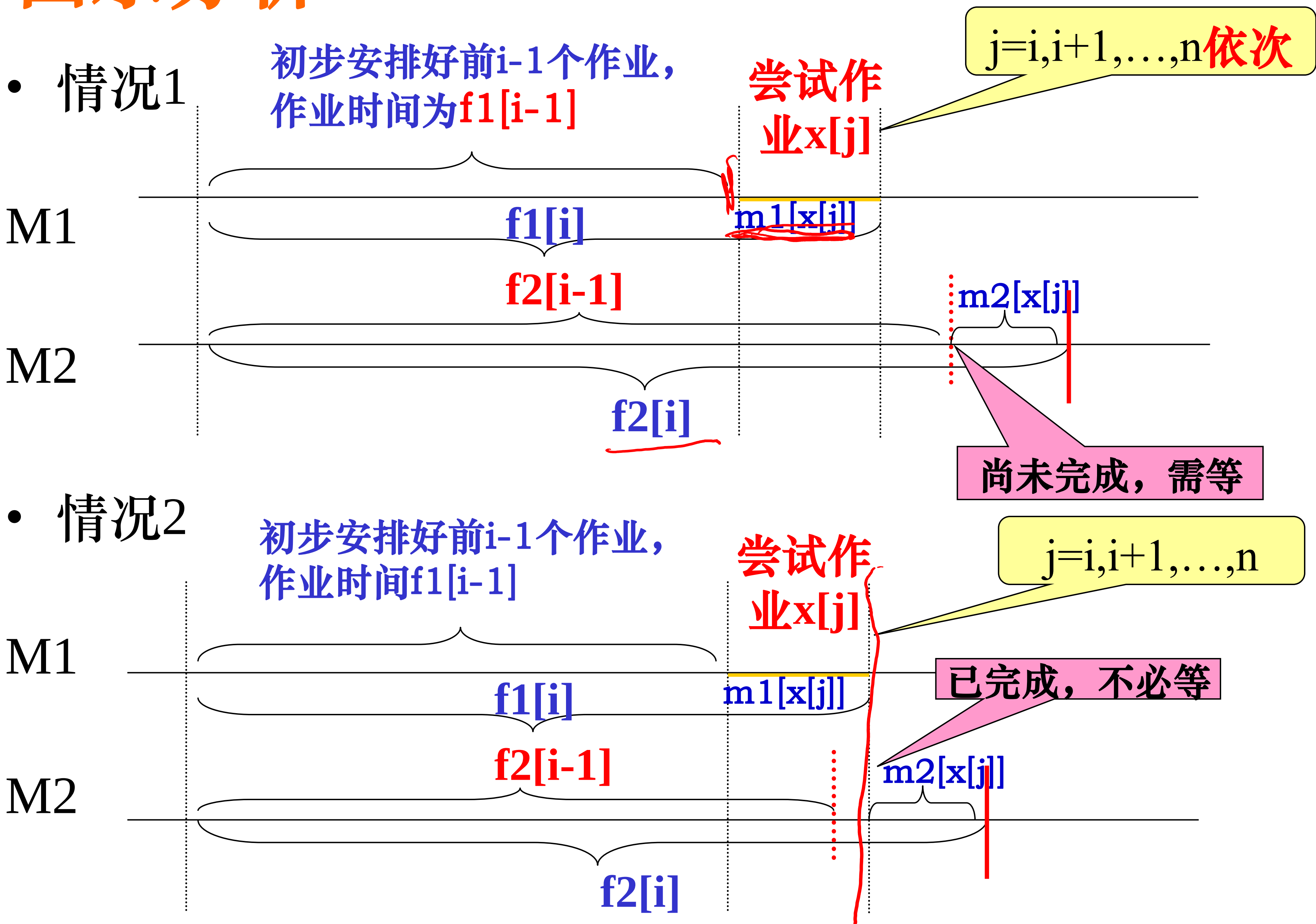
- 设x[1..n]是n个作业，解空间为排列树
- 需从排列树中找出一个解（即作业排列）
- 设bestx是返回的最佳作业调度，bestf是当前最小完成时间



算法设计

- 尝试将x[1]（取值1、2、…、n）先排入作业队列，计算完成x[1]所需的M1上的作业时间f1[1]，M2上做完所需的时间f2[1] 。选择一个使f2[1] 最小的安排。
- 假定现在已安排好前i-1个作业，M1、M2上作业时间为f1[i-1]、 f2[i-1] 。
- 现在考虑第i个作业。在没有安排的作业中任选一个进行考察，即取 x[j]。

图示分析



• 情况1
初步安排好前i-1个作业，
作业时间为f1[i-1]
尝试作业x[j]
j=i,i+1,…,n依次
M1
f1[i]
m1[x[j]]
f2[i-1]
m2[x[j]]
M2
f2[i]
尚未完成，需等
• 情况2
初步安排好前i-1个作业，
作业时间f1[i-1]
尝试作业x[j]
j=i,i+1,…,n
M1
f1[i]
m1[x[j]]
已完成，不必等
f2[i-1]
m2[x[j]]
M2
f2[i]

f2[i]的计算

f1= f1 + m[x[j]][1];

f2[i]=((f2[i-1]>f1)? f2[i-1]:f1)
+m[x[j]][2]

对j=i,i+1,…,n**依次尝试**

算法框架

```
void backtrack (int i){
  if (i>n) output(x);  //记录或修正解
    else
      for (int j=i;j<=n;j++) {
        //此处做一些必要的工作，如计算f1,f2[i]
        //当前完成时间f
         swap(x[i], x[j]);
         if (legal(i)) //若合法、满足限界
             backtrack(i+1);
         swap(x[i], x[j]);
      }
      //完成回退操作
}
```

算法实现

```
void backtrack(int i){
  if (i > n) {
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];  bestf = f;
    }
   else
    for (int j = i; j <= n; j++) {
       f1+=m[x[j]][1];
       f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)
    +m[x[j]][2];
       f+=f2[i];
       if (f < bestf) {
          swap(x,i,j);
          backtrack(i+1);
          swap(x,i,j);
       }
       f1-=m[x[j]][1];    f-=f2[i];
    }
}
```

有关的变量定义，可用类定义

```
int  n,      // 作业数
f1,  // 机器1完成处理时间
 f,   // 完成时间和
bestf;  // 当前最优值
int **m;  // 各作业所需的处理时间
int *x;   // 当前作业调度
int *bestx;  // 当前最优作业调度
int *f2;  // 机器2完成处理
```

限界函数： f= $\sum_{k=1}^{i} f_2(k) < bestf$

初始化、调用、算法复杂性

- 初始作业队列1234…n，bestf=无穷大
- 调用Backtrack(1)
- 时间复杂性O(n!)

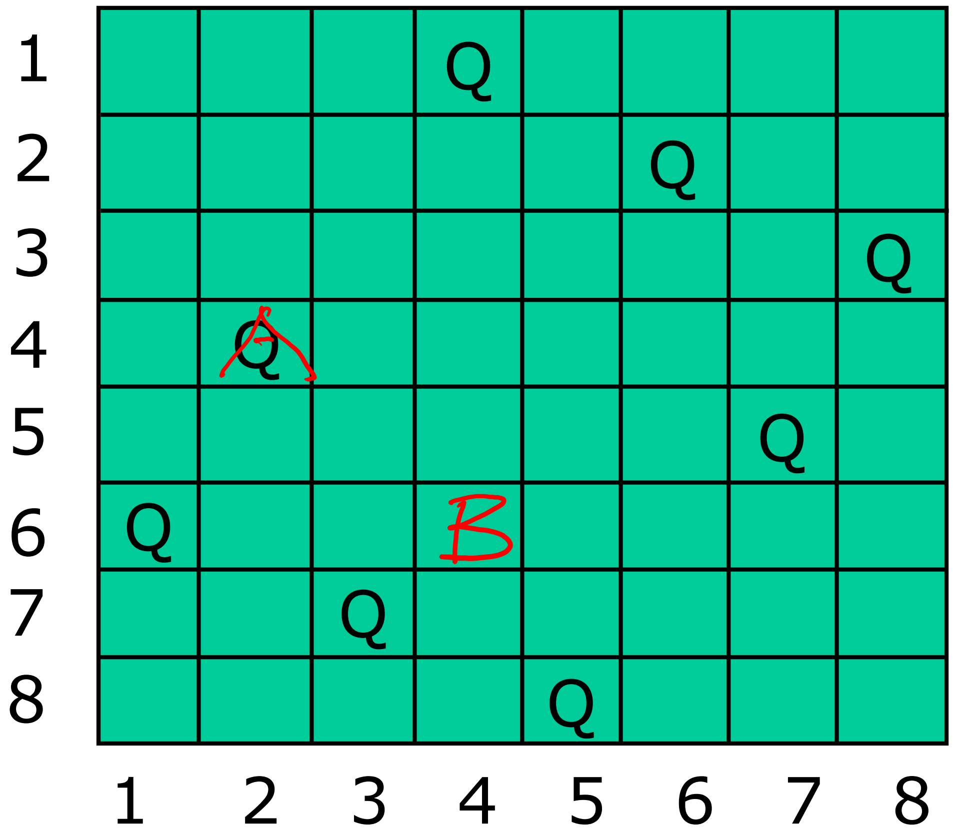
练习：确定批处理作业最佳作业调度方案，使其完成时间和达到最小。

t_{ji}	M1	M2
J_1	5	7
J_2	10	5
J_3	9	7
J_4	5	8

5.5 n后问题

n皇后问题描述

- 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，皇后可以攻击与它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。



算法分析

1、设定解向量：$(x_1, x_2, \ldots, x_n)$，采用排列树

2、约束条件

（1）显式约束：$x_i = 1, 2, \ldots, n \quad (i = 1,2, \ldots, n)$

（2）隐式约束

1）“不同列”：$x_i \neq x_j \quad (i,j = 1, 2, \ldots ,n;\ i \neq j)$

2)不处于同一正、反对角线：$|i-j| \neq |x_i - x_j|$

可放k个皇后条件

```
bool 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;
 }
```

递归回溯-统计解总数

- n：皇后个数
- x：当前解
- sum：当前已找到的可行方案数

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

调用：backtrack(1);

本问题：排列树
约束条件：由place(t)控制

约束函数：
place(t)

迭代回溯

```
void   N-queens(n){
   x[1]= 0;   k = 1;      //k为层次
   while  (k > 0)   {
       x[k] = x[k] + 1;
       while(  (x[k] <= n) &&!place(k))  //当前位置不合适
               x[k]= x[k] + 1;                  //尝试下一个位置
         if  (x[k]  <=  n)
               if  (k == n) sum++;
               else {     //设置禁止放置皇后的“标志”；
                     k= k + 1;    x[k]= 0  }
         else k --;    }
}
```

练习：给定n，输出几组解

统计解数

这是迭代回溯法的很好例子，应学好

5.6 0-1背包问题

算法分析

- 0-1背包问题属于子集选取问题
- 解空间：子集树
- 可行性约束函数： $\sum_{i=1}^{n} w_i x_i \le c$

- 限界函数估计（即上界函数，仿一般背包问题）：
  以物品单位重量价值递减序装入物品；整个装不下时，再装该物体的一部分，而把背包装满，求得可能的最大价值。

例：n=4,c=7,v=[9,10,7,4], w=[3,5,2,1]

单位重量价值：9/3, 10/5, 7/2, 4/1

降序排列后，先装物品4（重=1），再装物品3（重=2），
物品1（重=3），再装物品2的一部分（重=1），

最大价值最多为22。

上界（或限界）函数计算

```
double bound(int i)   {// 计算上界
     double cleft = c - cw;   // 剩余容量
     double bnd = cp;         // cp：当前价值
     while (i <= n &&  w[i] <= cleft) {
           // 以物品单位重量价值递减序装入物品
       cleft -= w[i];
       bnd += p[i];
       i++;
     }     // 背包有空隙时，装满背包
     if (i <= n &&  w[i] > cleft)  bnd += p[i] / w[i] * cleft;
     return bnd;
  }
```

回溯程序Backtrack

c：背包容量　　n：物品数　　w：物品重量数组

p：物品价值数组　　cw：当前重量　　cp：当前价值

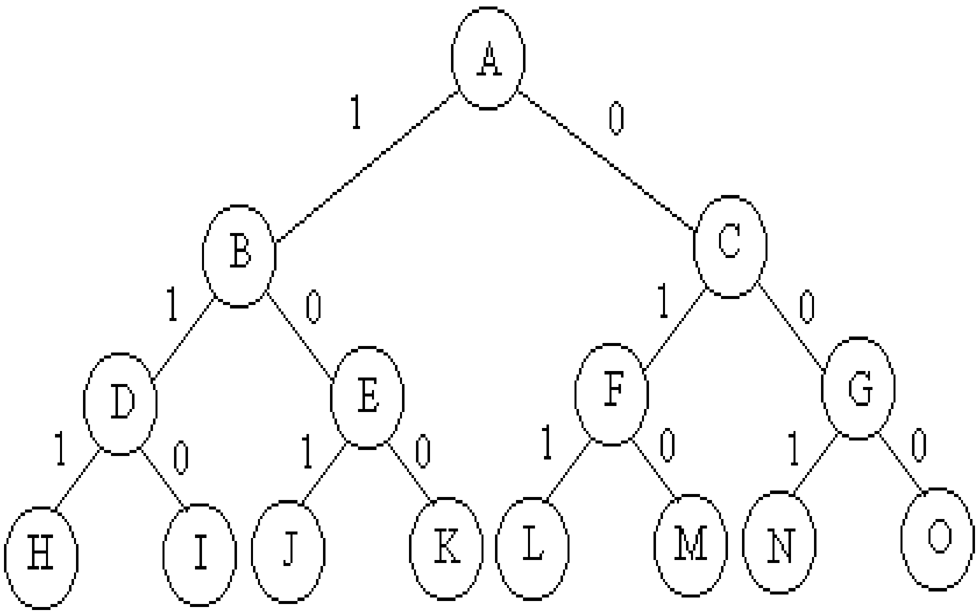
bestp：当前最优价值

```
void Backtrack(int i){
    if (i > n) {
        bestp = cp;   return; }
    if (cw + w[i] <= c) {          //x[i] = 1
        cw + = w[i];     cp + = p[i];
        Backtrack(i + 1);
        cw - = w[i];
        cp - = p[i]; }
    if (Bound(i + 1) > bestp)         // x[i] = 0，否则剪去右枝
        Backtrack(i + 1);
}
```

约束函数：
cw + w[i]

限界函数：
Bound(i + 1)

0-1背包问题的搜索树



- **在操作时可采取剪枝技术:**

  在Bound(i + 1) > bestp前提下，当cp+r<=bestp时，剪去右枝

5.7 最大团问题

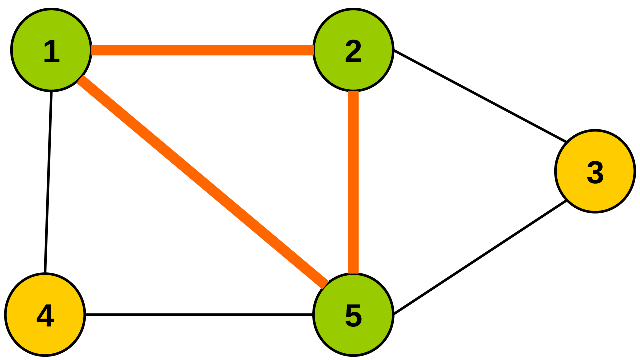
几个概念(1)

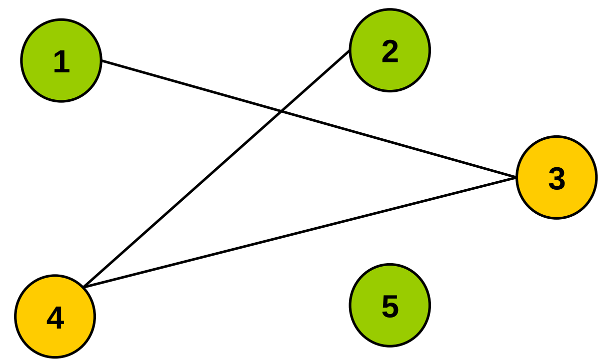
- 给定无向图G=(V，E)。
- 完全子图：如果U⊆V，且对任意u，v∈U有(u，v)∈E，则称U是G的完全子图。
- 团：G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中
- 最大团：G的最大团是指G中所含顶点数最多的团。
- 空子图：如果U⊆V且对任意u，v∈U有(u，v)∉E，则称U是G的空子图。即不含边的子图。

几个概念(2)

- **独立集：** 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。
- **最大独立集：** 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_1，E_1)定义为：V_1=V，且(u，v)$\in E_1$当且仅当(u，v)$\notin$E

举 例





补图

- **团：{1，2，5}{2，3，5}{1，4，5}**

一些关系

U是G的完全子图，当且仅当，U是$\overline{G}$的空子图。

U是G的团，当且仅当，U是$\overline{G}$的独立集。

U是G的最大团，当且仅当，U是$\overline{G}$的最大独立集。

算法分析

- 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。--相连性
- 限界函数：取cn + n – i，即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。

记

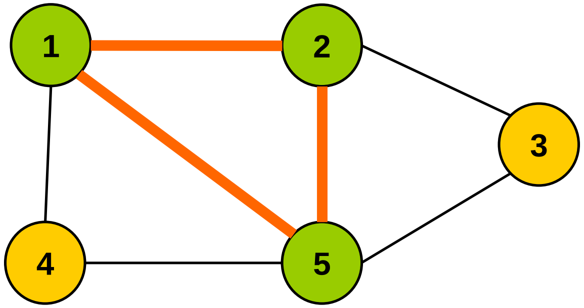
n：图的顶点数

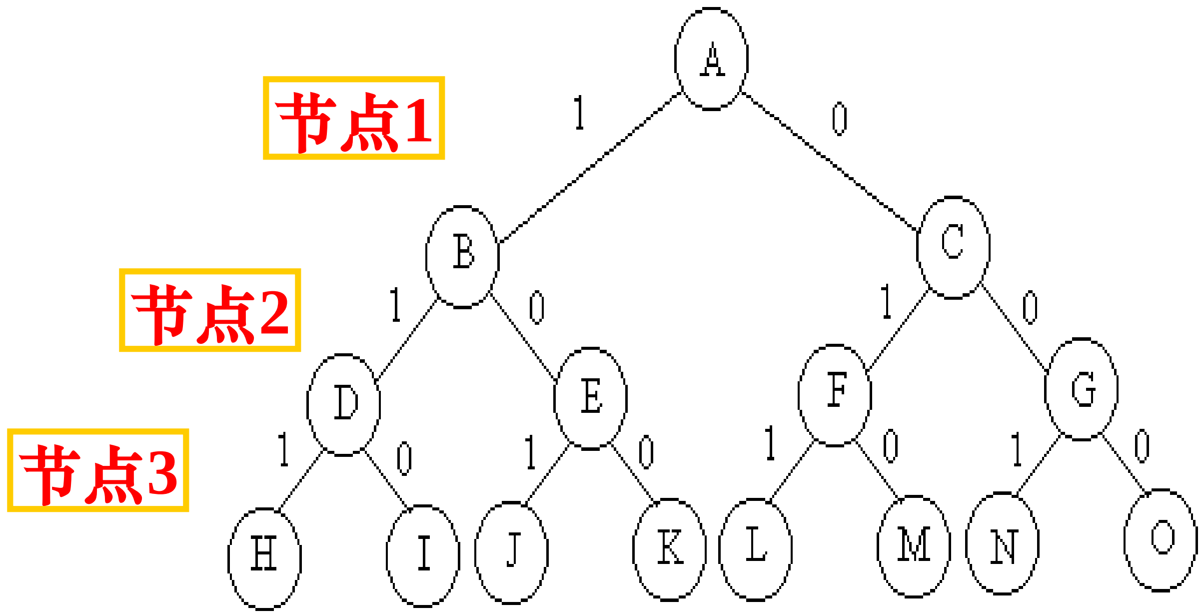
x：当前解

bestx：当前最优解

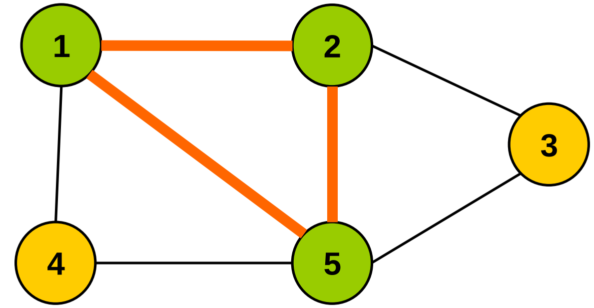
cn：当前顶点数

bestn：当前最大顶点数


1
2
3
4
5


节点1
节点2
节点3
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
1
0

程序




```
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
    int ok = 1;
    for (int j = 1; j < i; j++)          //欲扩展节点i
      if (x[j] == 1 && !a[i][j])    {//考察： i与前面的j是否相连
        ok = 0;     break;    }          // i与前面的j不相连，舍弃i
    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);      }
  }
```

限界函数

最大团问题复杂性

复杂度分析

最大团问题的回溯算法backtrack所需的计算时间显然为$O(n2^n)$。

- **算法改进设想：**

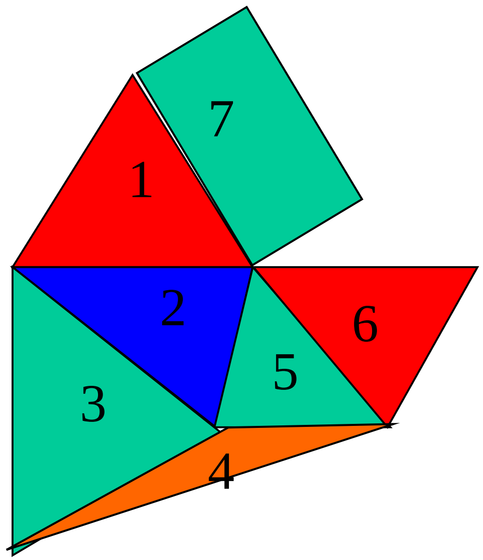
  选择合适的搜索顺序，可以使得上界函数更有效地发挥作用。如在搜索之前可将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。

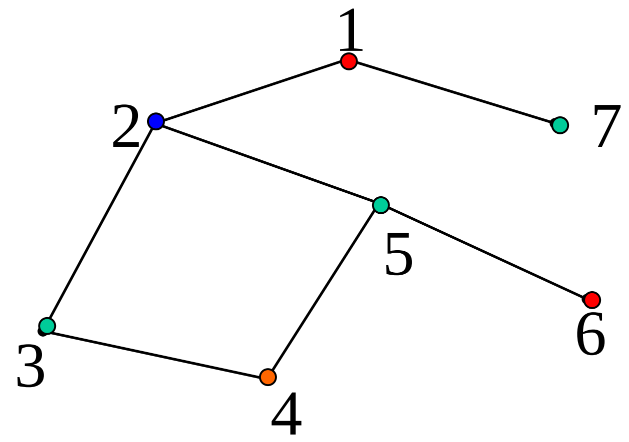
5.8 图的m着色问题

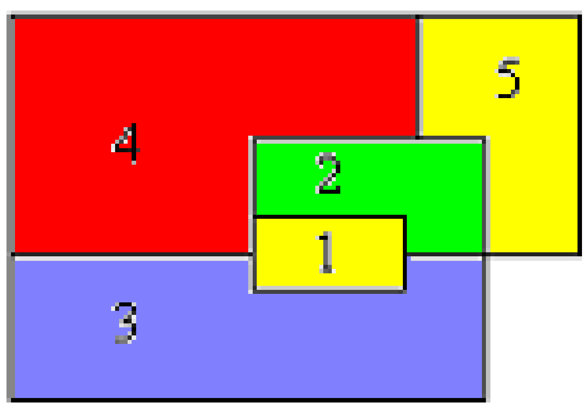
问题描述

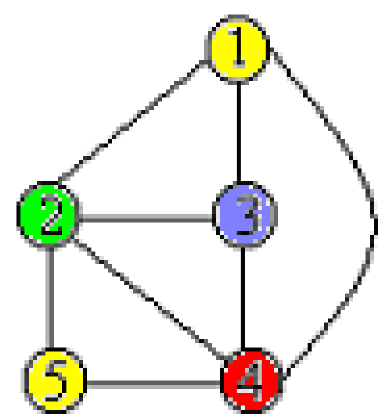
- 给定无向连通图G和m种不同的颜色。
- **顶点着色：**用m种颜色为图G的各顶点着色，每个顶点着一种颜色。每条边的2个顶点着不同颜色。
- **边着色：**用这些颜色为图G的各边着色，每边着一种颜色。与每顶点关联的边着不同颜色。
- **面着色：**用这些颜色为图G的各面着色，每个面着一种颜色。相邻的2个面着不同颜色。

顶点着色与面着色









问题描述

- **图的m着色问题**：是否有一种着色法使G中每条边的2个顶点着不同颜色。
- **图的 m可着色优化问题**：求一个图的色数m的问题称为图的m可着色优化问题。
- **图的四色问题**：用至多四种颜色可对一个图进行面着色。（1976年解决）

算法分析

- 解向量：$(x_1, x_2, \ldots, x_n)$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。
- n：图的顶点数
- m：可用颜色数
- x：当前解
- sum：当前已找到的可m着色方案数

跟n皇后问题类似

```
void backtrack(int t)
  {
    if (t>n) {sum++;输出解}
    else
      for (int i=1;i<=m;i++) {
        x[t]=i;
        if (ok(t)) backtrack(t+1);
        }
  }
```

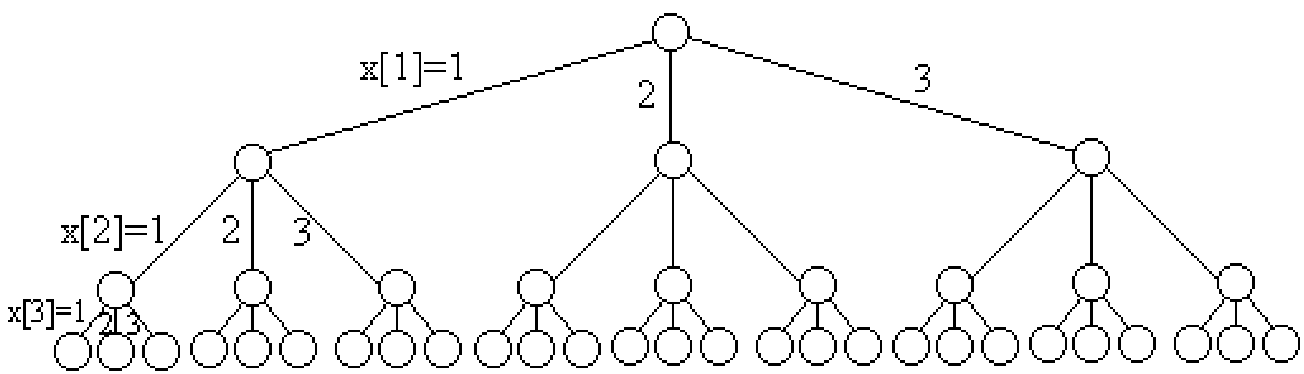
```
bool ok(int k)
  {// 检查颜色可用性
    for (int j=1;j<=n;j++)
      if (a[k][j] && (x[j]==x[k]))
        return false;
    return true;
  }
```

思考：用迭代回溯如何实现?

复杂度分析

图m可着色问题的解空间树中内结点个数是 $\sum_{i=0}^{n-1} m^i$ **对于每一个内结点，在最坏情况下，用ok检查当前扩展结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**

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



思考

图的m着色问题与图的最大团问题有何关系，你能否利用这个关系改进最大团问题的上界?