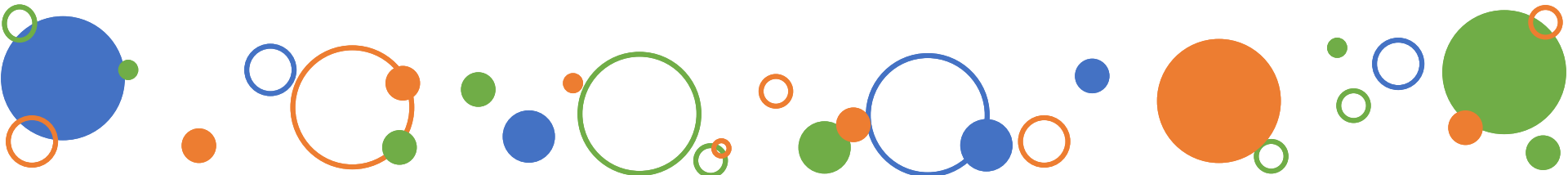
DiCA

Data Intelligence & Computing Art Laboratory

算法分析

刘权辉

2024 春




SICHUAN UNIVERSITY
1896
DiCA
Data Intelligence & Computing Art Laboratory

第五章：回溯算法

学习要点

- 理解回溯法的深度优先搜索策略。
- 掌握用回溯法解题的算法框架
  - 递归回溯
  - 迭代回溯
  - 子集树算法框架
  - 排列树算法框架
- 通过应用范例学习回溯算法设计策略：
  - 装载问题；
  - 批处理作业调度；
  - 旅行售货员问题；
  - N后问题；
  - 0-1背包问题；
  - 图着色问题；
  - 最大团问题

引言

- 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。回溯法有“通用的解题法”之称。
- 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。
- 即：将问题的解组织成一颗解空间树，在这个颗树上进行搜索得到问题解。

引言

- 回溯法求问题的所有解时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。
- 回溯法求问题的一个解时，只要搜索到问题的一个解就可结束。

引言

- 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。

- 这种以深度优先方式系统搜索问题解的算法称为回溯法。

问题的解空间

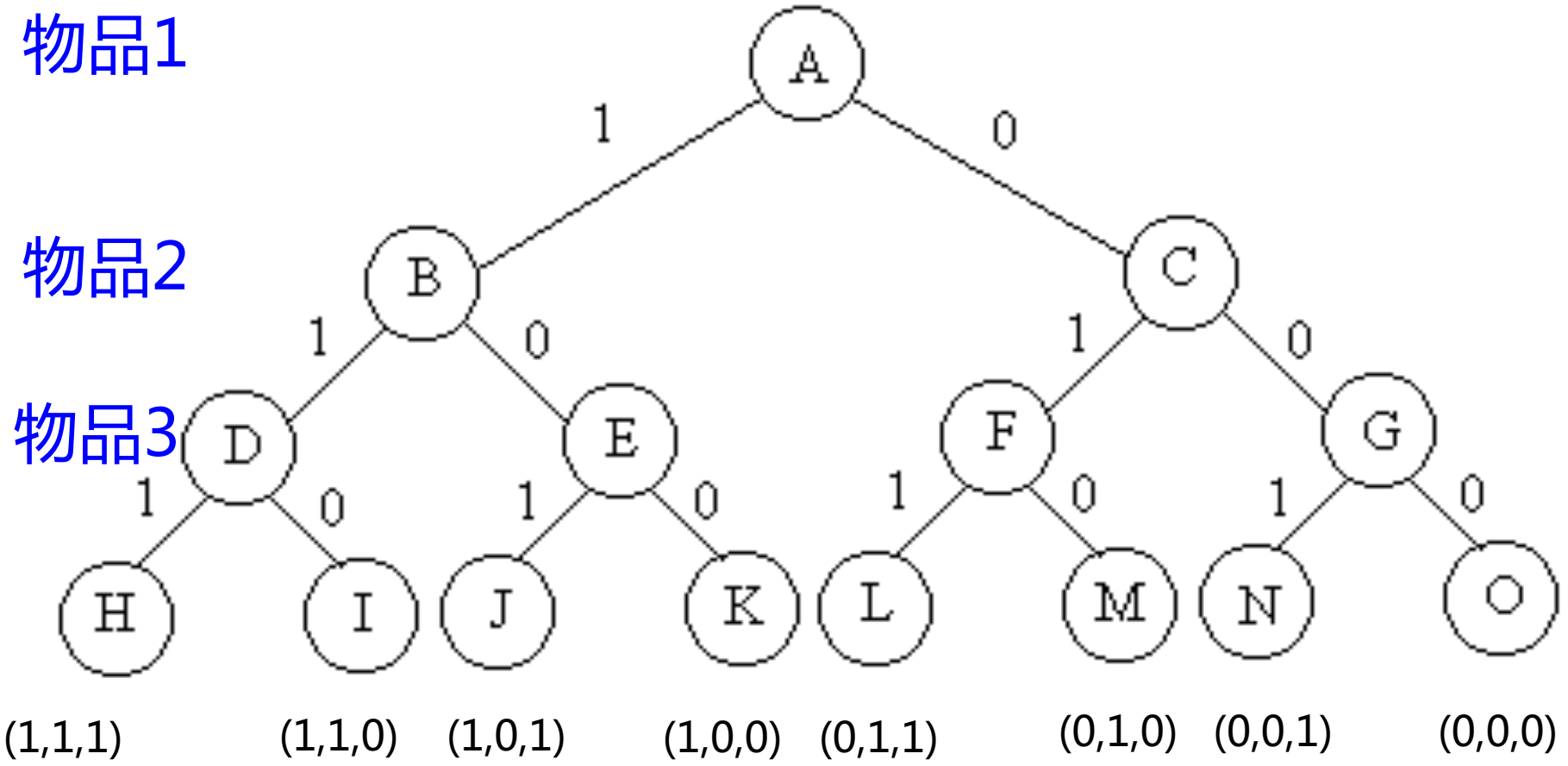
口 概念

- 问题的解向量：回溯法希望一个问题的解能够表示成一个 n 元式 $(x_1, x_2, \ldots, x_n)$ 的形式。
- 显约束：对分量 x_i 的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

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）。

问题的解空间

例：n=3时0-1背包问题用完全二叉树表示的解空间



概念

- 扩展结点：一个正在产生儿子的结点称为扩展结点；
- 活结点：一个自身已生成但其儿子还没有全部生成的节点称做活结点；
- 死结点：一个所有儿子已经产生的结点称做死结点。

生成问题状态的基本方法

问题状态生成的方法

- 深度优先的问题状态生成法：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）；

- 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；

口 问题状态生成的方法

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

回溯法基本步骤

1）针对所给问题，定义问题的解空间；

2）确定易于搜索的解空间结构；

3）以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

回溯法的基本思想

- 确定了解空间的组织结构后，回溯法就从开始根结点出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。
- 在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。
- 回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已没有活结点时为止。

回溯法的基本思想

- 确定了解空间的组织结构后，回溯法就从开始根结点出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。
- 在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。
- 回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已没有活结点时为止。

回溯法的基本思想

- 确定了解空间的组织结构后，回溯法就从开始根结点出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。
- 在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。
- 回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已没有活结点时为止。

回溯法的基本思想

- 确定了解空间的组织结构后，回溯法就从开始根结点出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。
- 在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。
- 回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已没有活结点时为止。

回溯法的基本思想

回溯法的特征与空间复杂度

- 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)!)的内存空间。

回溯法的基本思想

回溯法的特征与空间复杂度

- 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)!)的内存空间。

回溯法的基本思想

回溯法的特征与空间复杂度

- 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)!)的内存空间。

常用剪枝函数：

- 用约束函数在扩展结点处剪去不满足约束的子树；
- 用限界函数剪去得不到最优解的子树。

递归回溯

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
   if (t>n) output(x);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

t为递归深度：当前扩展结点在解空间树中的深度。

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);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

t>n：已搜索到叶结点

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);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

f(n,t)，g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号

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);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

h(i)表示当前扩展结点处x[t]的第i个可选值

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);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

返回true表示：在当前扩展结点处x[1：t]的取值满足问题的约束条件，否则剪掉相应的子树。

递归回溯

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
   if (t>n) output(x);
   else
   {
      for (int i=f(n,t);i<=g(n,t); i++)
      {
         x[t]=h(i);
         if ( constraint(t) && bound(t) )
            backtrack(t+1);
      }
   }
}
```

返回true表示：在当前扩展结点处x[1：t]的取值未使目标函数越界，否则剪掉相应的子树。

迭代回溯（了解）

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

```
void iterativeBacktrack (){
    int t=1;
    while (t>0){
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++){
                x[t]=h(i);
                if (constraint(t)&&bound(t)){
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

未达到叶节点

迭代回溯（了解）

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

```
void iterativeBacktrack (){
    int t=1;
    while (t>0){
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++){
                x[t]=h(i);
                if (constraint(t)&&bound(t)){
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

f(n,t) 和 g(n,t) 分别表示在当前扩展结点处未搜索过的子树的 起始编号和终止编号。当起始编号小于终止编号时，进入子树

迭代回溯（了解）

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

```
void iterativeBacktrack (){
    int t=1;
    while (t>0){
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++){
                x[t]=h(i);
                if (constraint(t)&&bound(t)){
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

h(i)表示在当前扩展结点处x[t]的第i个可选值。

迭代回溯（了解）

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

```
void iterativeBacktrack (){
    int t=1;
    while (t>0){
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++){
                x[t]=h(i);
                if ( constraint(t)&&bound(t) ){
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

在当前扩展节点处满足约束函数和限界函数

迭代回溯（了解）

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

```
void iterativeBacktrack (){
    int t=1;
    while (t>0){
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++){
                x[t]=h(i);
                if ( constraint(t)&&bound(t) ){
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

判断在当前扩展结点处是否已得到问题的可行解，
1)返回true表示：在当前扩展结点处x[1:t]是问题的可行解；
2)否则表示x[1:t]只是问题的部分解，需进一步求解。

概念

- 子集树：当所给的问题是从n个元素的集合S中找出S满足某种性质的子树时，相应的解空间树称为子集树。

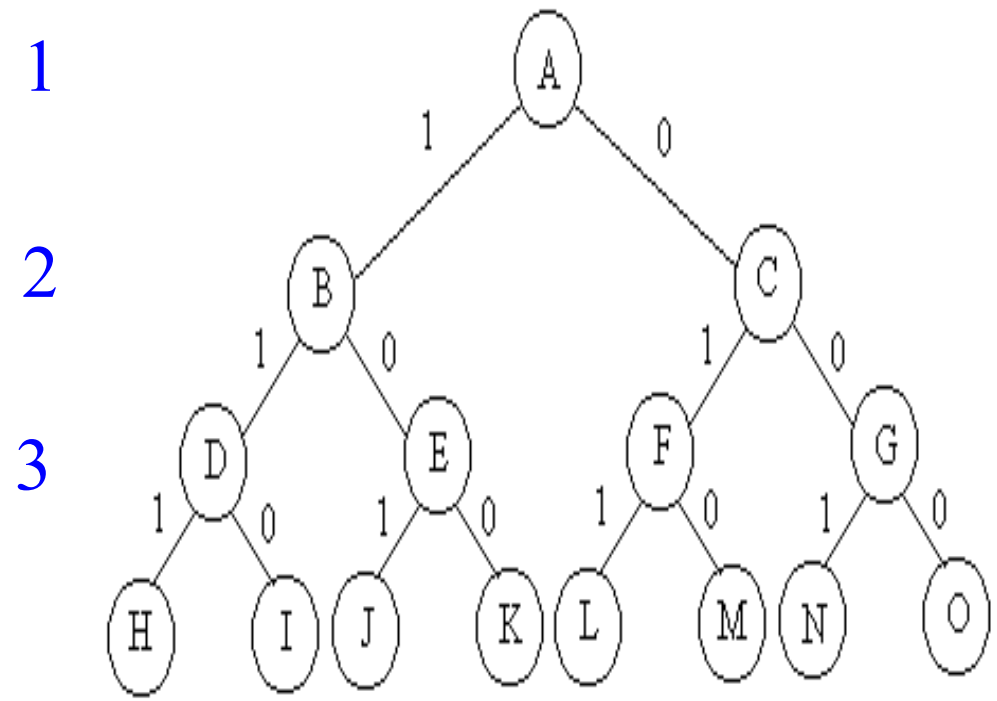
- 例如：0-1背包问题所对应的解空间是一颗子集树，这类子集树通常有2^n个叶子结点，其结点总数为2^{n+1}-1,遍历子集树需要$\Omega(2^n)$。

子集树

遍历子集树需 $\Omega(2^n)$ 计算时间

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

01背包问题



概念

- 排列树：当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有 n! 个叶子结点，因此遍历排列树的需要 Ω(n!)。

- 例如：旅行售货员问题的解空间是一颗排列树。

补充：旅行售货员问题

问题描述

- TSP问题是指旅行家要旅行n个城市，要求各个城市经历且仅经历一次然后回到出发城市，并要求所走的路程最短
- 各个城市间的距离可以用代价矩阵来表示

$$C=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix} \end{array}$$

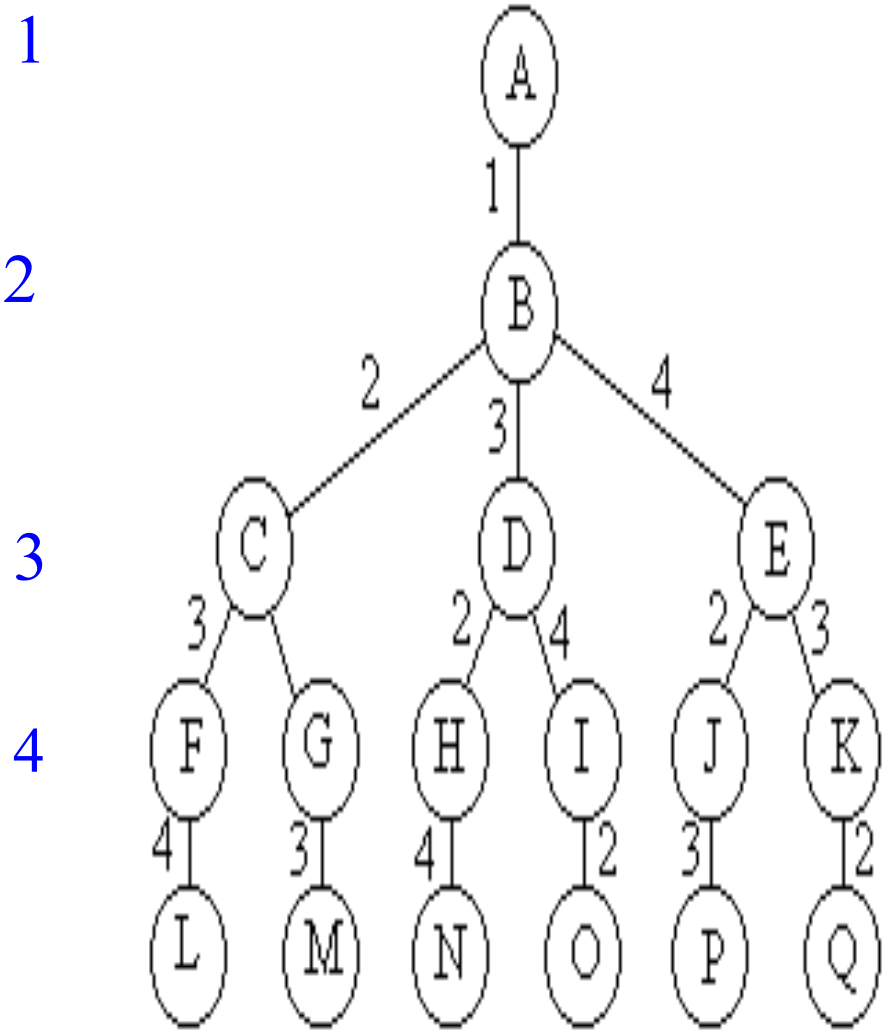
带权图的代价矩阵

- 解空间由n个城市的全排列构成

排列树

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++)
        {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t+1);
            swap(x[t], x[i]);
        }
}
```



注：先将数组X初始化为单位排列：（1，2，3...，n）


四川大学
SICHUAN UNIVERSITY
1896
DiCA
Data Intelligence & Computing Art Laboratory

5.1 装载问题

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n个集装箱装上这2艘轮船。如果有，找出一种装载方案。

口 分析：

- 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。

口 分析：

- 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c_1。
- 由此可知，装载问题等价于以下特殊的0-1背包问题：

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

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

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

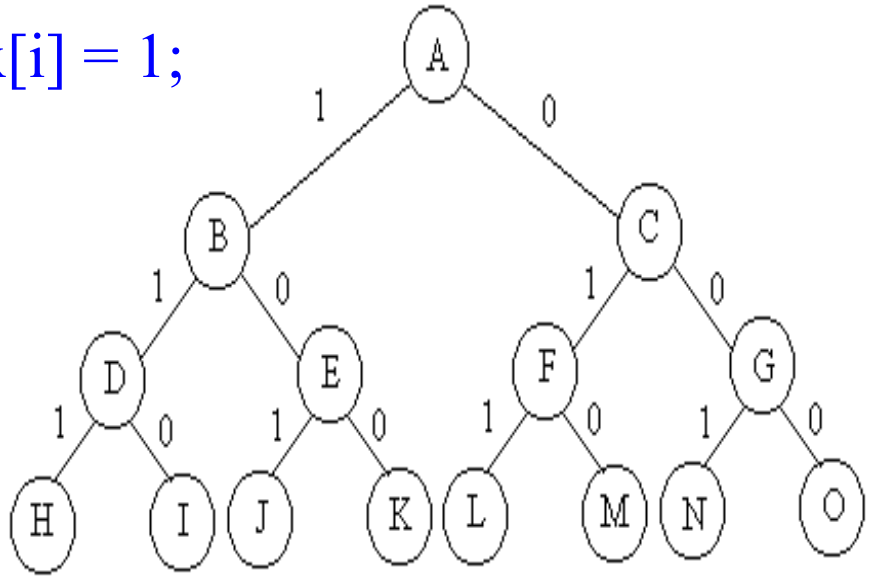
注意：用回溯法设计解装载问题的$O(2^n)$计算时间算法。在某些情况下该算法优于动态规划算法。

装载问题

分析：

- 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 当前载重量：cw；当前最优载重量 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,x[i] = 1;
        cw += w[i];
        backtrack(i + 1);
        cw -= w[i];}
    else{          // 搜索右子树:x[i] = 0;
        backtrack(i + 1);}
}
```



DiCA

Data Intelligence & Computing Art Laboratory

5.2 批处理作业调度

批处理作业调度

问题描述：

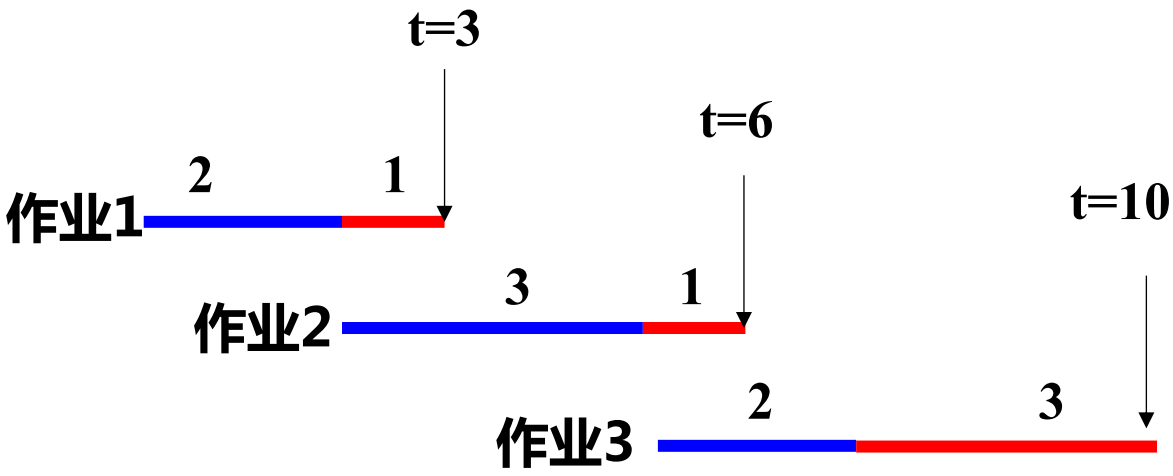
- 给定n个作业的集合 $\{j_1, j_2, \ldots, j_n\}$。每个作业必须先由机器1处理，然后由机器2处理。作业j_i需要机器j的处理时间为t_{ji}。
- 对于一个确定的作业调度，设F_{ji}是作业i在机器j上完成处理的时间。所有作业在机器2上完成处理的时间和(包括等待时间)称为该作业调度的完成时间和：$f = \sum_{i=1}^{n} F_{2i}$

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。

批处理作业调度

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

作业调度为1,2,3时： 3+6+10=19



口 分析：

- 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。

批处理作业调度

分析：

- 批处理作业调度问题要从n个作业的所有排列中找出有最小完成时间和的作业调度，所以批处理作业调度问题的解空间是一棵排列树；

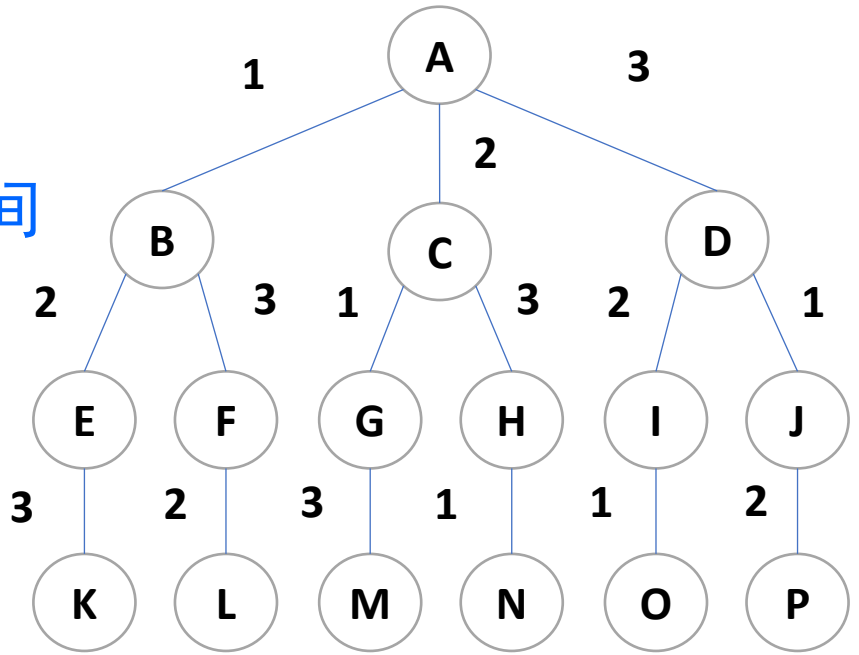
- 限界函数：当前最小完成时间和（上界约束）。

批处理作业调度

口 实现

M[i][1]:作业i需要机器1的处理时间
M[i][2]:作业i需要机器2的处理时间

```
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};
```



批处理作业调度

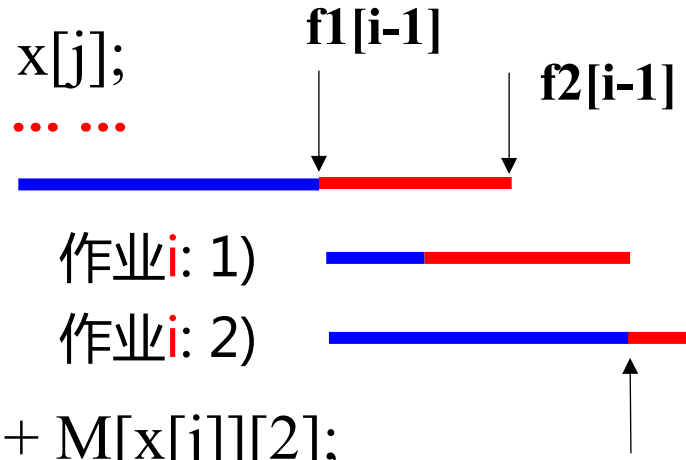
```
void Flowshop::Backtrack(int i){
    if (i > n) {
        for (int j = 1; j <= n; j++)      bestx[j] = x[j];
        bestf = f; }
    else {
        for (int j = i; j <= n; j++) {
            f1 += M[x[j]][1];
            f2[i] = ( (f2[i-1] > f1)  ? f2[i-1]:f1 ) + M[x[j]][2];
            f += f2[i];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

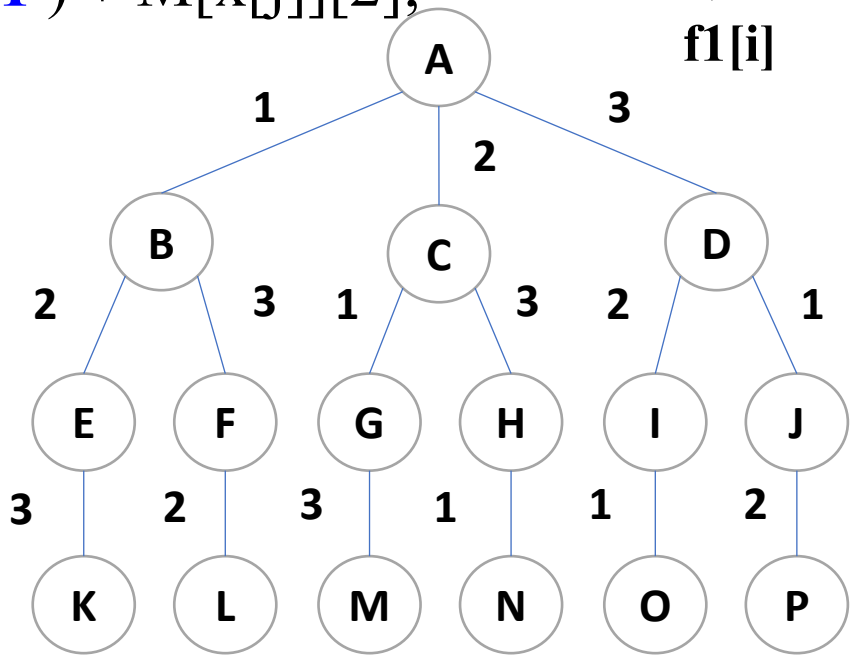
到达叶节点

批处理作业调度

非叶节点，位于排列树第i-1层

```
void Flowshop::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], x[j]);
        Backtrack(i+1);
        Swap(x[i], x[j]); }
      f1 -= M[x[j]][1];
      f -= f2[i];
    }  }
```





批处理作业调度

```
void Flowshop::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], x[j]);
        Backtrack(i+1);
        Swap(x[i], x[j]); }
      f1 -= M[x[j]][1];
      f -= f2[i];
    }  }
```

限界剪枝

批处理作业调度

```
void Flowshop::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], x[j]);
        Backtrack(i+1);
        Swap(x[i], x[j]); }
      f1 -= M[x[j]][1];
      f -= f2[i];
    }  }
```

时间复杂度分析：

- 对于排列树，最坏情况下时间复杂度为O(n!)


四川大学
1896
SICHUAN UNIVERSITY
DiCA
Data Intelligence & Computing Art Laboratory

5.3 旅行售货员问题

旅行售货员问题

问题描述

- TSP问题是指旅行家要旅行n个城市，要求各个城市经历且仅经历一次然后回到出发城市，并要求所走的路程最短

- 各个城市间的距离可以用代价矩阵来表示

$$C=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix} \end{array}$$

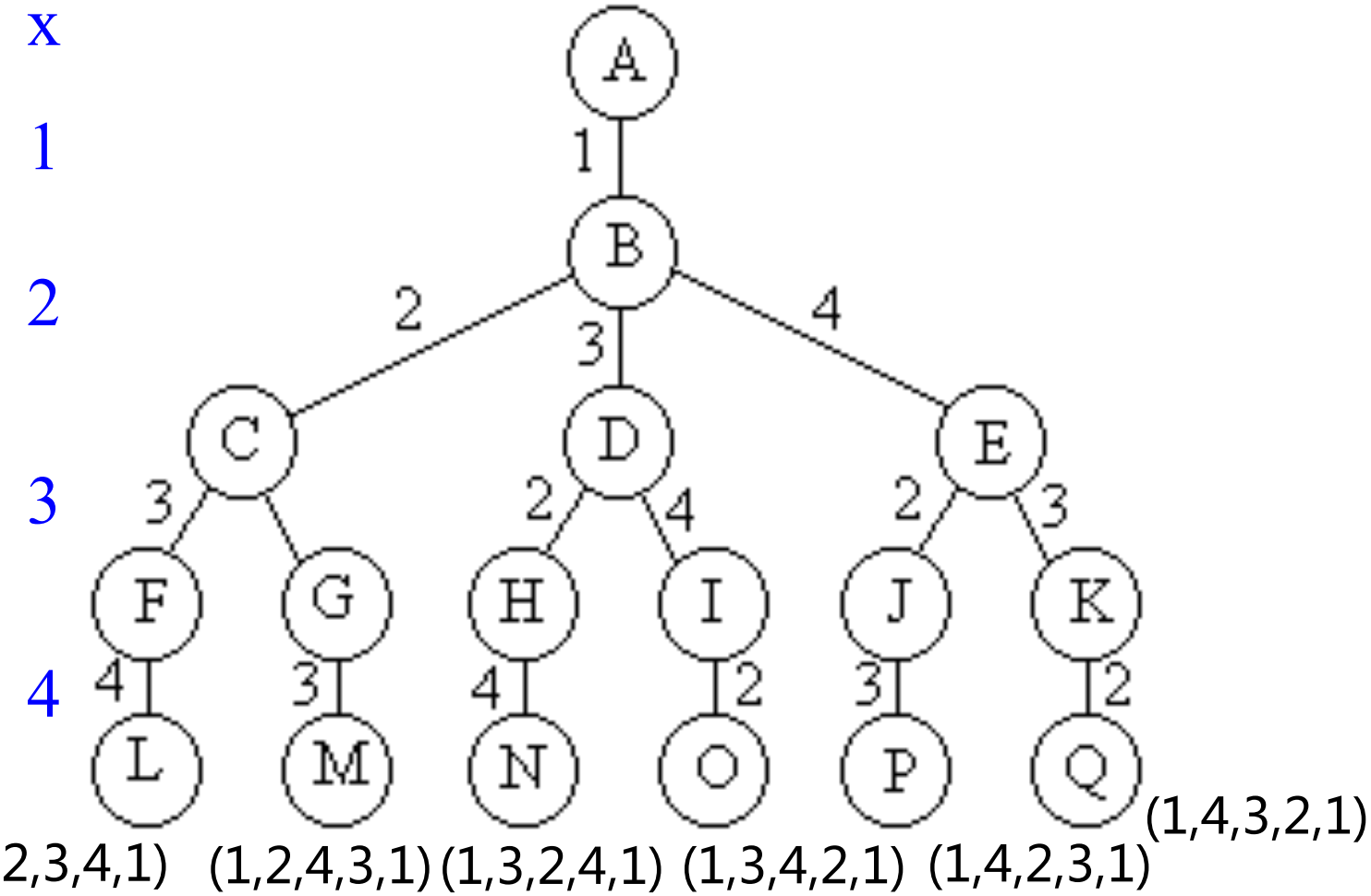
带权图的代价矩阵

- 解空间由n个城市的全排列构成

旅行售货员问题

分析

- 解空间由n个城市的全排列构成



实现

```
template <classTyp e>
class Traveling {
   friend Type TSP（int **, int [], int, Type）;
   private:
      void Backtrack（int i）;
      int n,                    //图G的顶点数
        *x,                     //当前解
     *bestx;                    //当前最优解
    Type **a,                   //图G的邻接矩阵
        cc,                     //当前费用
      bestc,                    //当前最优值
     NoEdge;                    //无边标记
}
```

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     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]);}
    }
}
```

i=n时,当前扩展结点是排列树的叶结点的父节点

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     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]);}
    }
}
```

是否存在一条从顶点x[n-1]到顶点x[n]的边和一条从顶点x[n]到顶点1的边。

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     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]);}
    }
}
```

判断这条回路的费用是否优于已找到的当前最优回路的费用bestc

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     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]);}
    }
}
```

i < n 时，当前扩展结点位于排列树的第 i -1层。

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     if (a[x[i-1]][x[j]] != NoEdge && (cc + a[x[i-1]][x[j]] < 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]);}
    }
}
```

存在从顶点x[i-1]到顶点x[j]的边时，x[1:i]构成图G的一条路径，

旅行售货员问题 (实现)

```
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++) {        //是否进入子树
     if (a[x[i-1]][x[j]] != NoEdge && (cc + a[x[i-1]][x[j]] < 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]);}
    }
}
```

X[1:i]的费用小于当前最优值时算法进入排列树的第 i 层

旅行售货员问题

时间复杂度分析：

- 如果不考虑更新bestx所需的计算时间，则Backtrack需要 O((n-1)!) 计算时间。
- 由于算法Backtrack在最坏情况下可能需要更新当前最优解 O((n-1)!) 次，每次更新bestx需 O(n) 计算时间，从而整个算法的计算时间复杂性为 O(n!)。


四川大学
1896
SICHUAN UNIVERSITY


DiCA
Data Intelligence & Computing Art Laboratory

5.4 0-1背包问题

0-1背包问题

- **问题描述**：略
- **分析：**
  - 0-1背包问题也是一个子集选取问题（子集树）。
  - 例：有3个不可分割的商品，其重量与价值分别如下图示。若背包容量为30公斤（C=30），请利用回溯法策略找出最佳解。

Item	重量 (W)	价值(P)
1	20	$40
2	15	$25
3	15	$25

0-1背包问题

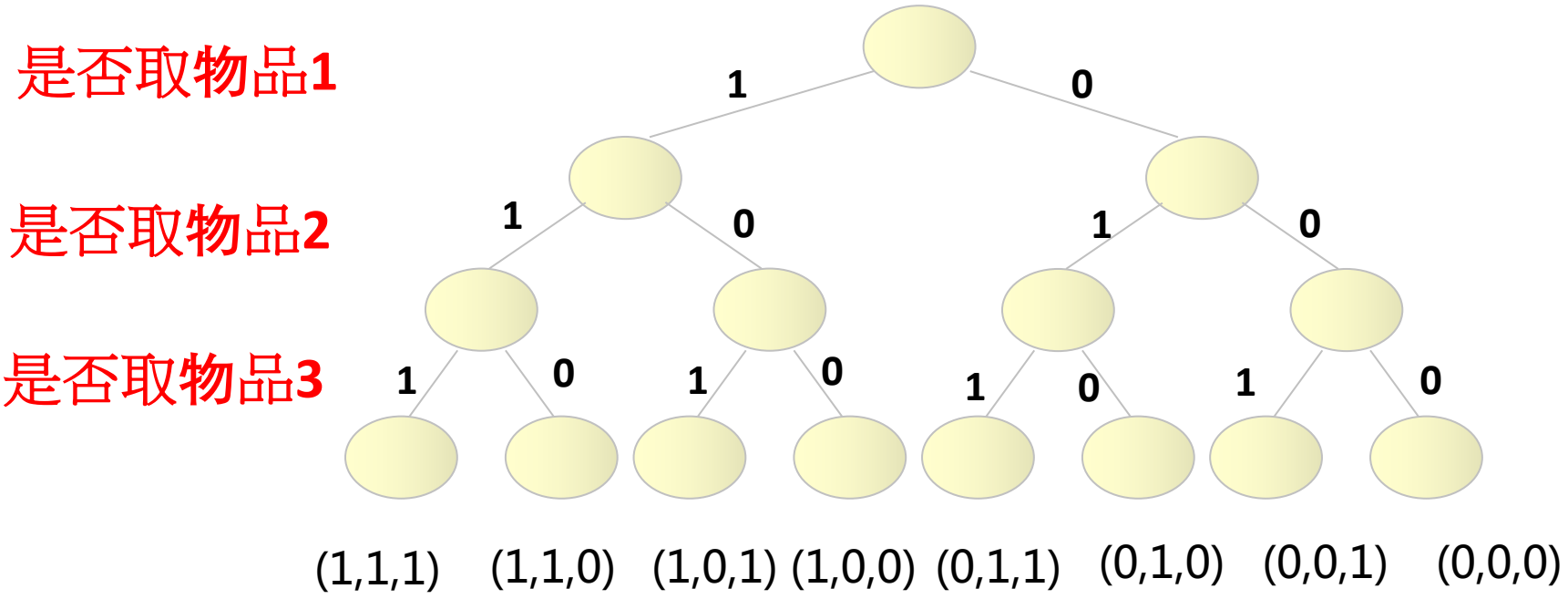
口 分析：

- 边界函数为：$\sum w_i * x_i \leq C$, 其中：

  1）w_i是商品i的重量 (整数变量)

  2）x_i#是商品i是否被装入 (布尔变量)

- 三个商品的0-1背包问题的子集树如下：



0-1背包问题

Item	重量 (W)	价值(P)
1	20	$40
2	15	$25
3	15	$25

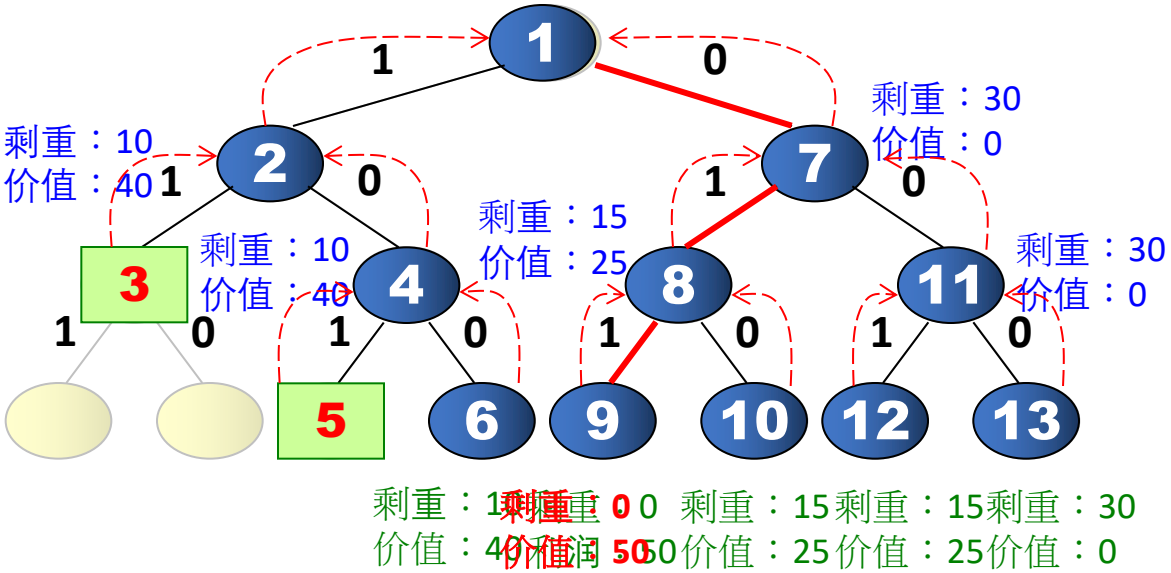
分析：

- 边界函数为：$\sum w_i * x_i \leq C$, 其中：
  1）w_i是商品i的重量 (整数变量)
  2）x_i#是商品i是否被装入 (布尔变量)
- 三个商品的0-1背包问题的子集树如下：　C=30

是否取商品1

是否取商品2

是否取商品3



0-1背包问题

- **实现：**

  算法描述略

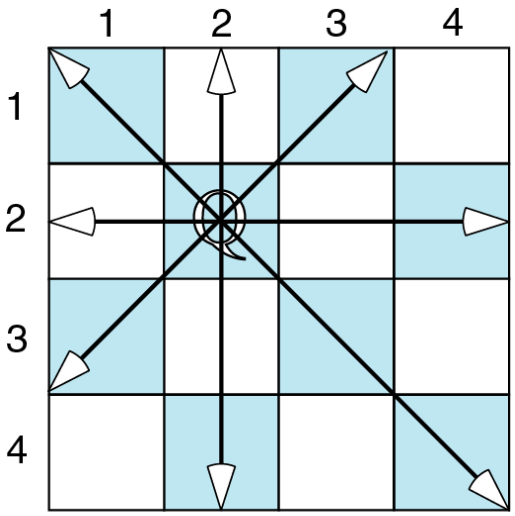

四川大学
1896
SICHUAN UNIVERSITY
DiCA
Data Intelligence & Computing Art Laboratory

5.5 N后问题

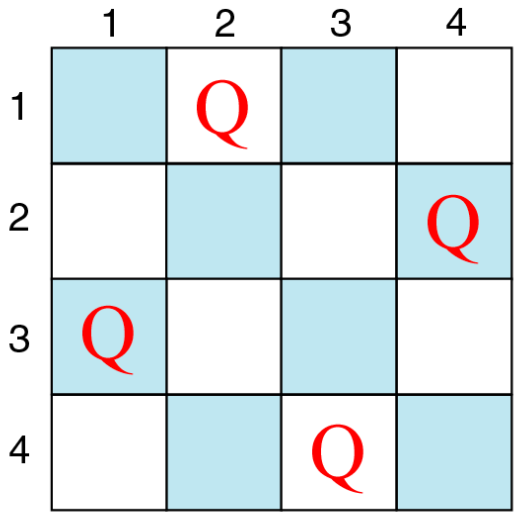
N后问题

问题描述：

- N后问题是指 “如何将n个国际象棋中的皇后棋子摆放在一个具有n列n行的棋盘中，但是这n个棋子彼此不会吃掉对方，也就是这n个皇后棋子彼此不允许在同一列、同一行或是同一个对角线上”。



(a) Queen capture rules



(b) First four queens solution

N后问题

- **分析**：
  - 方法1：
    - ✓ N个皇后需放置于不同列上；
    - ✓ 第一列可摆放的位置有n个，第二列可摆放的位置有n个，第三列可摆放的位置有n个，...，第n列可摆放的位置有n个；
    - ✓ n*n*n* ... *n = n^n 。

N后问题

分析：

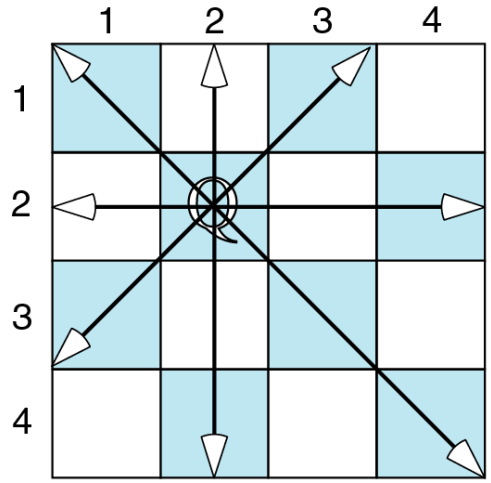
- 方法2：
  - ✓ n个皇后需放置于不同列、不同行的位置上；
  - ✓ 第一列可摆放的位置有n个，第二列可摆放的位置有n-1个，第三列可摆放的位置有n-2个，…，第n列可摆放的位置有1个；
  - ✓ n * (n-1) * (n-2) * … * 1 = n!；

显然此种方法的解空间更小

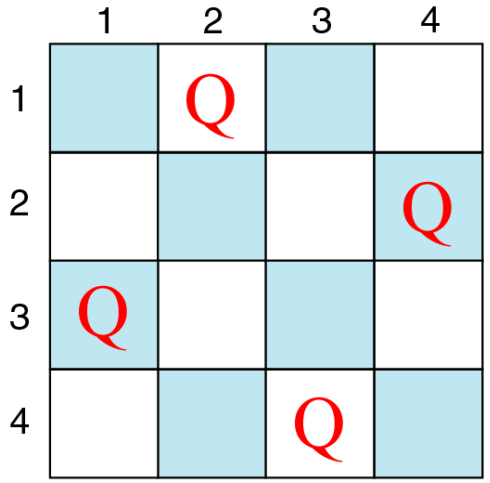
N后问题

例：四皇后问题

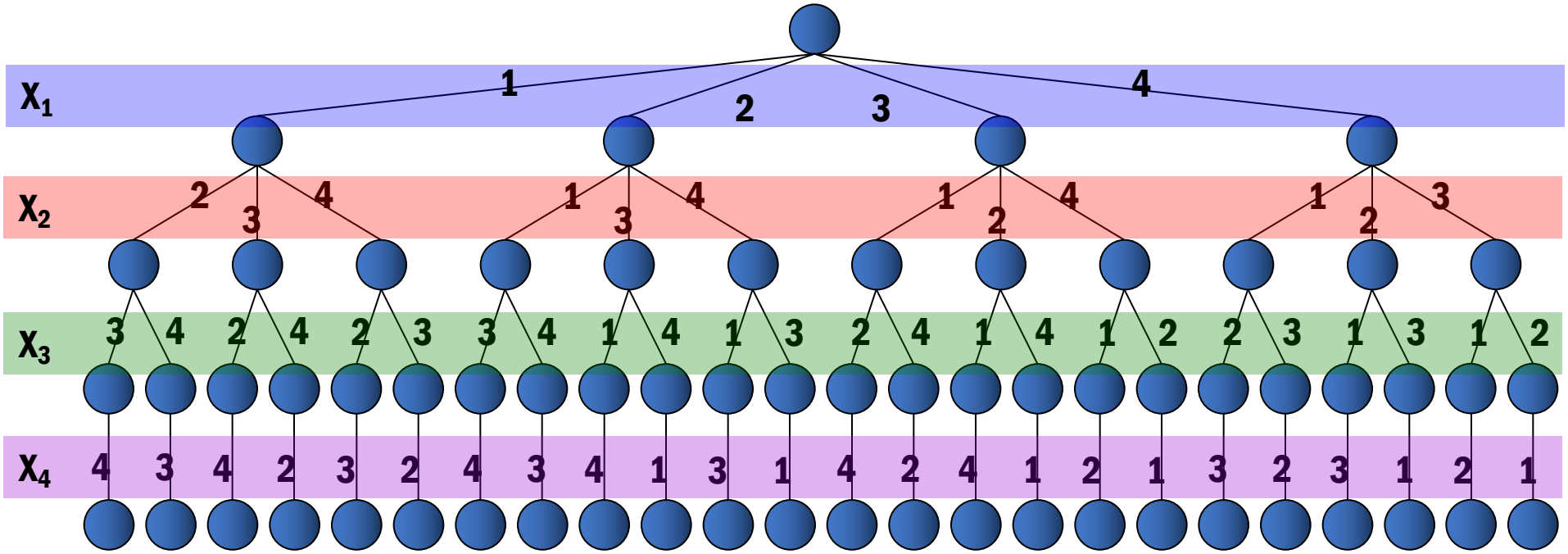
- x_i表示在第i列的皇后所在行的位置，则解空间树如下：



(a) Queen capture rules



(b) First four queens solution



例：四皇后问题

- 约束函数 - 判断皇后是否互吃
- 假设Q1皇后所在的位置是(a, xa)，Q2皇后所在的位置是(b, xb)，下列位置均会造成两个皇后互吃：

① 如果b =a ，则表示Q1皇后和Q2皇后在同一列;

② 如果xb =xa ，则表示Q1皇后和Q2皇后在同一行;

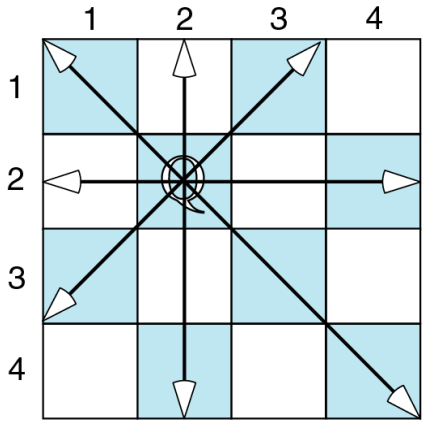
③ 如果 (xb- xa) =b – a，则表示Q1皇后和Q2皇后均在右斜45°线;

④ 如果 (xb- xa) = -(b – a)，则表示Q1皇后和Q2皇后均在左斜45°线。

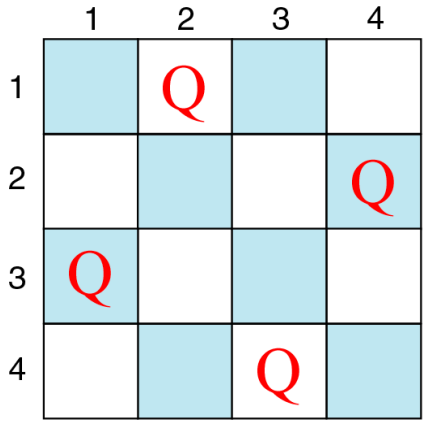
N后问题

例：四皇后问题

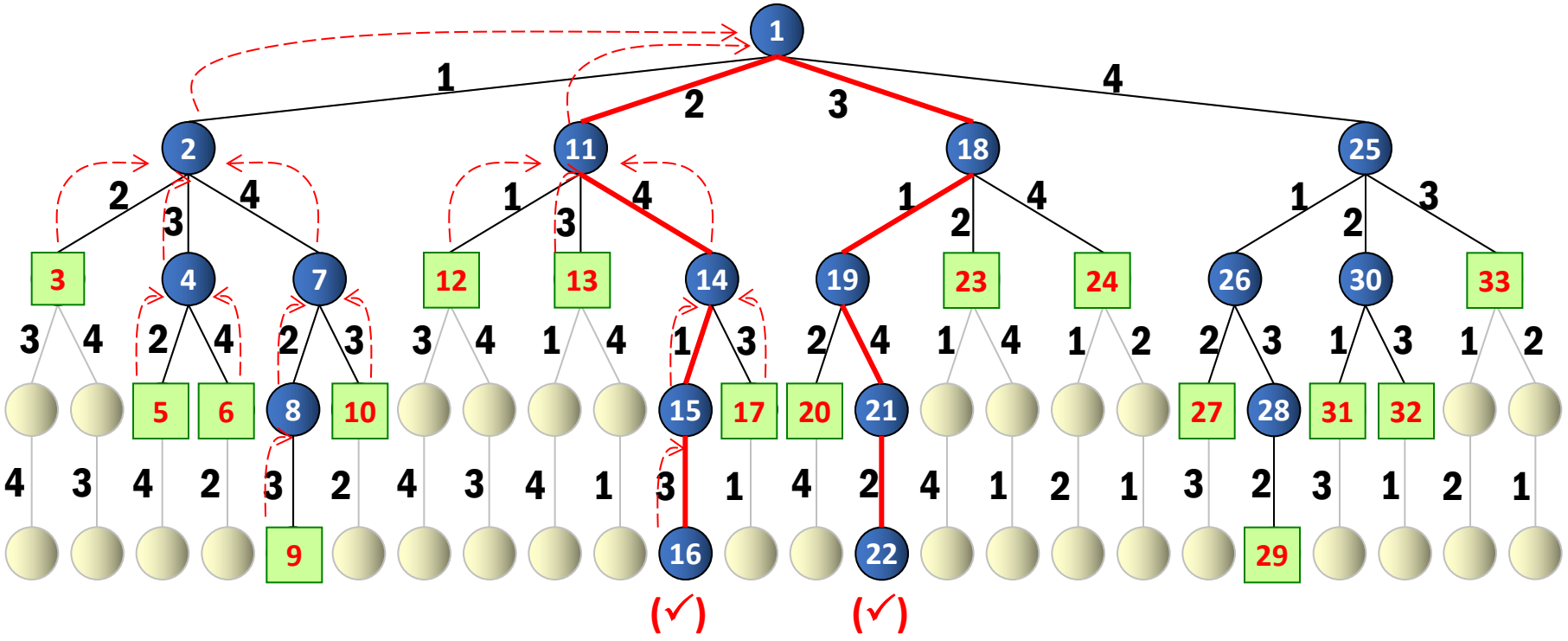
- 搜索示意图如下：



(a) Queen capture rules



(b) First four queens solution



实现

```
int nQueen（int n）    {
   Queen X;      //初始化X
   X．n＝n;
   X．sum＝0;
   int *p＝new int［n+1］;
   for（int i＝0; i＜＝n; i++）
      p［i］＝0;
   X．x＝p;
   X．Backtrack（1）;
   delete［］p;
   return X．sum;
}
```

N后问题

```
class Queen {
    friend int nQueen(int);
    private:
        bool Place(int k);
        void Backtrack(int t);
        int n,    //皇后个数
          *x;    //当前解
    long sum;    //当前已找到的可行方案数
};
```

x[i]表示皇后i放在棋盘的第i列的第x[i]行

N后问题

约束函数

```
bool Queen::Place(int k){
  for(int j=1; j<k; j++) //与前面k-1个皇后比较
    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); } }
```

设皇后位置分别是(i,j)和(k, l)，需要|i-k|!=|j-l|成立，即不在一条斜线上

N后问题

```
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); } }
```

x[i]互不相同，即皇后不同行不同列

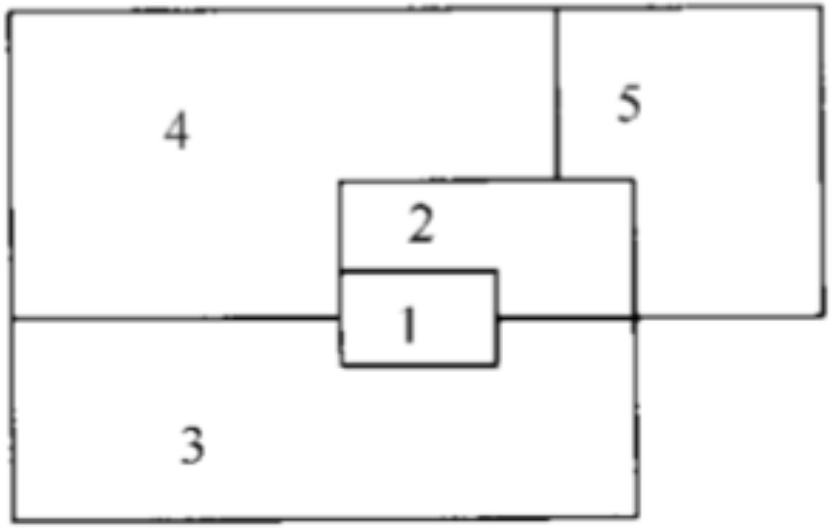
t>n 到达叶节点，得到一组可行解


四川大学
SICHUAN UNIVERSITY
1896
DiCA
Data Intelligence & Computing Art Laboratory

5.6 图着色问题

问题描述

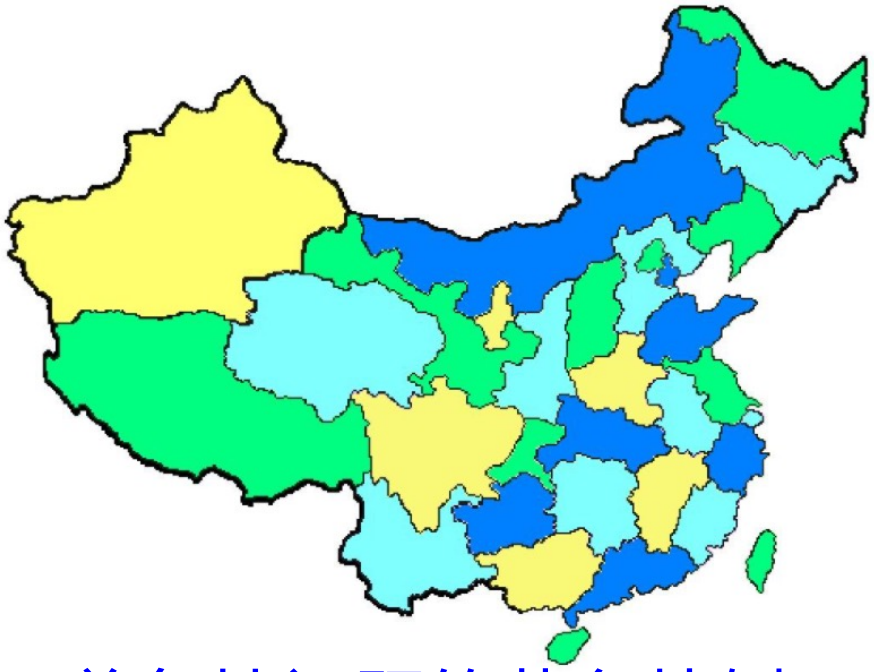
- 已知一个图G和m>0种颜色，在只准使用这m种颜色对G的结点着色的情况下，是否能使图中任何相邻的两个结点都具有不同的颜色呢？这个问题称为m-着色判定问题;



图着色问题

问题描述

- m着色最优化问题是求对图G着色的最小整数m。这个整数称为图G的色数;

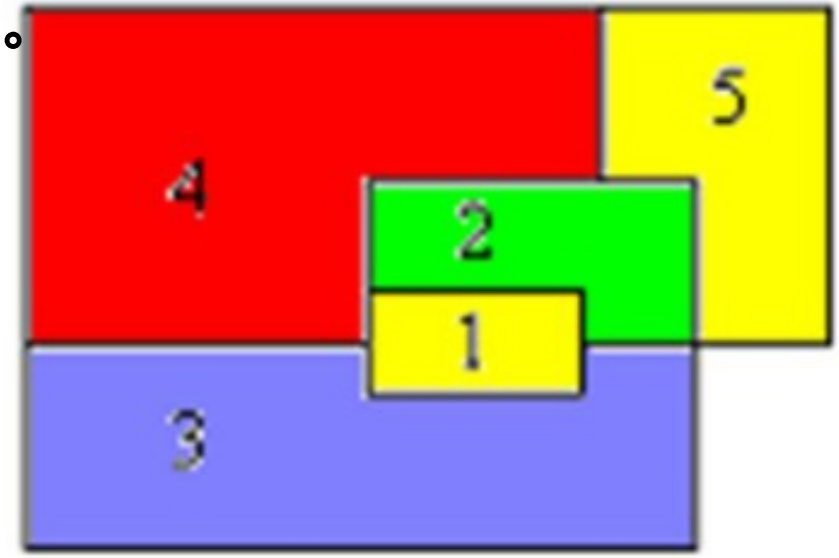


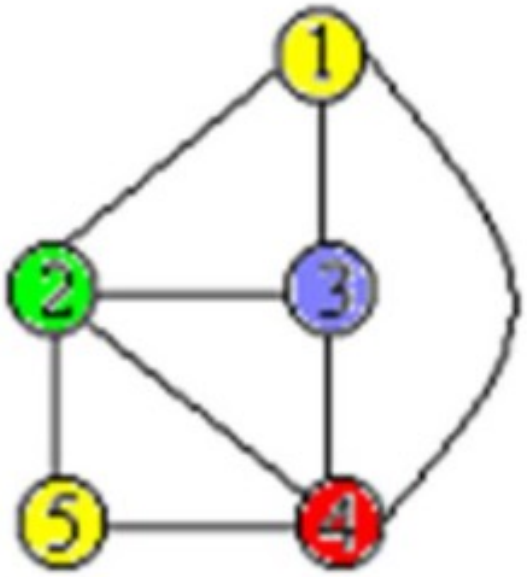
- 图着色的研究是从m-可着色性问题的著名特例——四色问题开始的。四色问题要求证明平面或球面上的任何地图的所有区域都至多可用四种颜色来着色，并使任何两个有一段公共边界的相邻区域没有相同的颜色;

图着色问题

问题描述

- 四色问题可以转换成对一平面图的4-着色判定问题;
- 将地图的每一个区域变成一个结点，若两个区域相邻，则相应的结点用一条边连接起来。
- 下图显示了一幅有5个区域的地图以及与该地图相应的平面图。





问题描述

- 多年来虽然已经证明用5种颜色足以对任一幅地图着色，但是一直找不到一定要求多于4种颜色的地图。直到1976年这个问题才由科学家利用电子计算机的帮助得以解决。他们证明了4种颜色足以对任何地图着色。
- 我们所考虑的不只是由地图产生的图，而是所有的图。讨论在至多使用m种颜色的情况下，可对一给定的图着色的所有不同的方法。

图着色问题

问题分析：用图的邻接矩阵 a 表示无向连通图 G=(V, E)；

① 若 (i, j) 属于图 G=(V,E)的边集 E ，则a[i][j]=1 ，否则 a[i][j]=0；

② 整数 1 ， 2 ，… ， m 用来表示 m 种不同颜色;

③ 顶点 i 所着的颜色用x[i]表示;

④ 数组 x[1:n] 是问题的解向量;

⑤ 那么，问题的解空间可表示为一棵高度为 n+1 的完全 m 叉树, 解空间树的第 i （ $1 \le i \le n$ ）层中每一结点都有 m 个儿子，每个儿子相应于 x[i] 的 m 个可能的着色之一;

⑥ 第 n +1层结点均为叶结点。

图着色问题

口 例：

- n=3, m=3时的解空间树

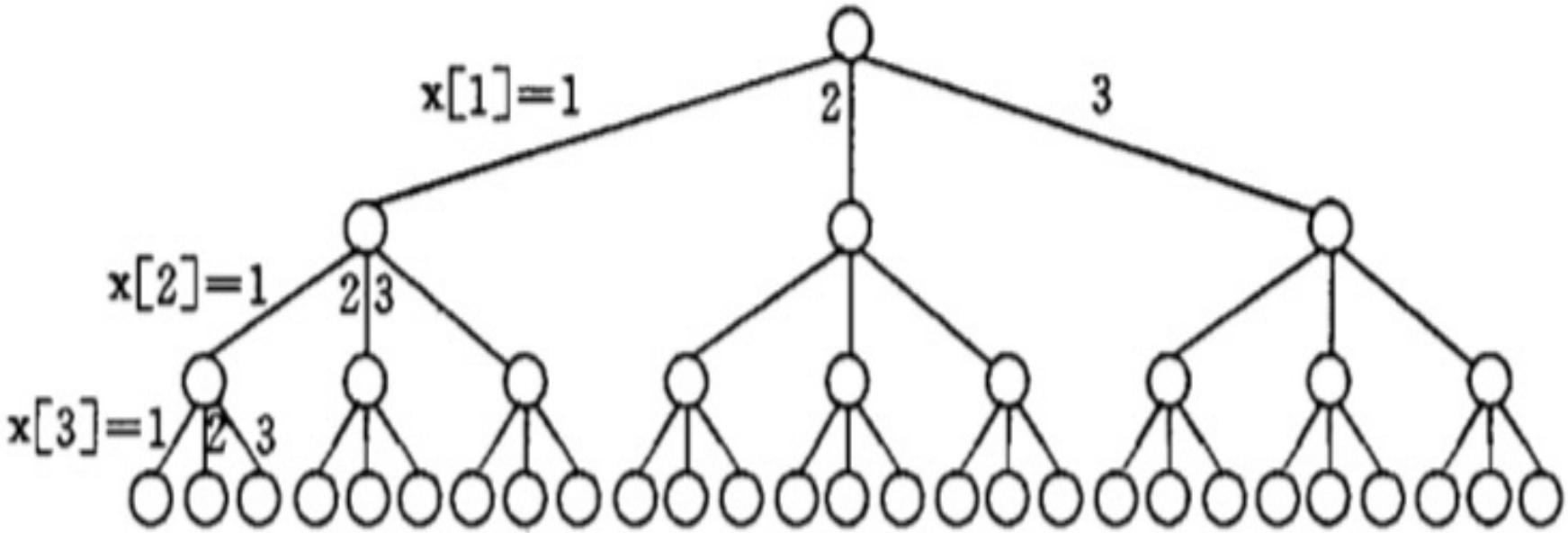


图 5-7　$n=3$ 和 $m=3$ 时的解空间树

约束函数

- 判断互相连接的两个节点的颜色是否相同

实现

```
class Color {
    friend int mColoring（int，int，int **）；
    private：
        bool Ok（int k）；
        void Backtrack（int t）；
        int n，     //图的顶点数
            m，     //可用颜色数
          **a；     //图的邻接矩阵
           *x，     //当前解（某个点用某种颜色着色）
    long sum；     //当前已找到的可m着色方案数
}；
```

实现

```
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；
 }
```

图着色问题

实现

```
void Color：：Backtrack（int t） {
    if（t>n）{  //到达叶子节点
        sum++;
        for（int i=1; i<=n; i++）
            cout << x［i］ <<";
        cout << endl;    }
    else
    {
        for（int i=1;  i<= m;  i++）
        {     //测试每种颜色
            x［t］=i;
            if ( OK(t) )
                Backtrack（t+1）;
            x［t］=0;
        }
    } }
```

图着色问题

□ 时间复杂度分析：

- 图m可着色问题的回溯算法的计算时间上界可以通过计算解空间树中内结点个数来估计；
- 图m可着色问题的解空间树中内结点个数是$\Sigma_{i=0}^{n-1} m^i$；
- 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的可用性需耗时 O(mn)；
- 因此，回溯法总的时间耗费是

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

5.7 最大团问题

最大团问题

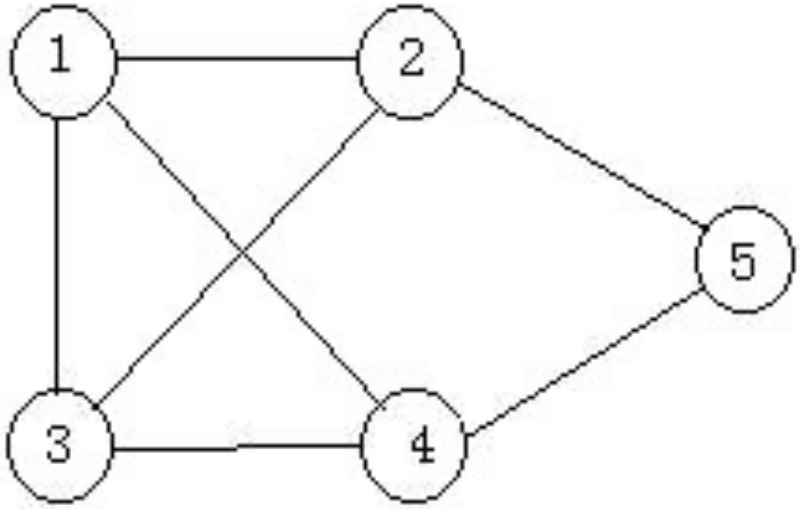
问题描述

- 给定一个无向图 G=(V, E), 如果 U 是 V 的子集，且对任意 u,v ∈U, (u,v)一定是边，且有(u,v)∈E ，则称 U 是 G 的一个完全子图（U是完全子图的必要条件：U为完全图）；
- G 的完全子图 U 是 G 的一个团当且仅当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——最大完全子图。

最大团问题

口例：下图所示无向图 G 中：

- 子集{1，2} 是 G 的大小为2的完全子图；
- {1，2 ，5} 不是完全子图（因为它当中的1 ，5 之间没有边）
- {1，2，3} 是 G 的大小为3的完全子图，它包含了{1 ，2} ，所以，{1 ，2} 这个完全子图不是团，{1 ，2 ，3} 和{1 ，3 ，4} 是G 的最大团。



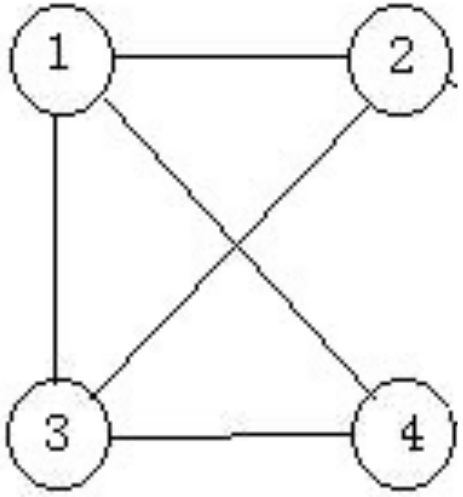
无向图G，顶点数n=5

最大团问题

口 分析

- 将图G的n个顶点依次编号1,2,…,n, 并将图用邻接矩阵表示;
- 定义：int g[n][n]; 用二维数组g存储图的邻接矩阵,图G的邻接矩阵可表示为数组g：

```
g={ 0, 1, 1, 1,
    1, 0, 1, 0,
    1, 1, 0, 1,
    1, 0, 1, 0, };
```



无向图G

最大团问题

口 分析

- 我们用 x_i=0 表示不选第i号顶点，x_i=1表示选第i号顶点，其中，i=1,2,…, n 。则此问题的解可表示为n元组（x_1，x_2，…，x_n ），$x_i \in \{0,1\}$, $1 \le i \le n$；

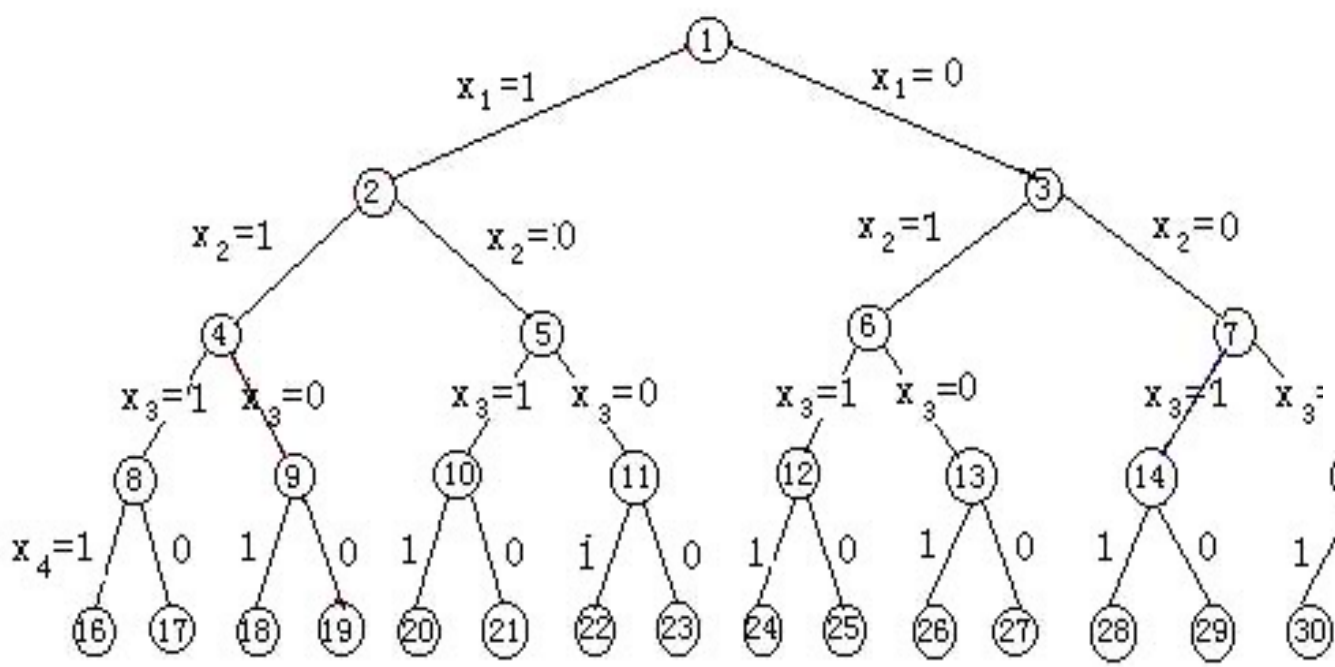
- 因此，可将最大团问题的视为图G顶点集V的子集选取问题，解空间树为一颗子集树;

- 在不受约束的情况下，所有可能的解共有 2^n 个（为解空间树的叶子数目），解空间可构造成一棵深度为 n+1 的满二叉树，见下页

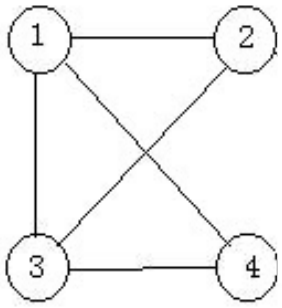
最大团问题

口 分析

- 第k层即代表第k个节点
- 约束函数：当前备选的节点x_k加入后，原图仍够成一个完全子图——任意两个点间存在边，即x_k与x_1,…，x_{k-1}均存在边。



n=4，图的最大团问题的解空间树（定长解）



无向图G

最大团问题

实现

- 定义变量

```
int g[n][n];      //存储图的邻接矩阵
int bestn;        //存储当前最优解（最大团）的
                  最大顶点数
int nown;         //存储当前寻找的团的顶点数
int x[n+1];       // 存储当前解
int bestx[n+1];   // 存储最优解
```

- 主函数

```
void MaxClique( void ) {
    int i;
    bestn=0;
    nown=0;
    Backtrack(1);
    printf("\n 最大团的顶点数： %d\n" , bestn );
    for (i=1; i<=n; i++)
      if (bestx[i]==1)   printf("%d, ", i );
   }
```

最大团问题——实现

```
void Backtrack(int k){
    int j, OK;
    if(k>n) {    //叶子节点
        for (j=1; j<=n; j++)
            bestx[j]=x[j];
        bestn = nown;
        return; }
    OK=1;
    //判断能否选择节点k
    for (j=1; j<k; j++)
      if (x[j]==1 && g[k-1][j-1]==0)
      {
            Ok=0;
            break;
      }
// 当顶点 k 能加入团中时
.....
}
```

```
// 当顶点 k 能加入团中时
if (OK){
        x[k]=1;          //进入左子树
        nown=nown+1;     //团的顶点数增 1
        Backtrack(k+1);  //继续递归处理
        x[k]=0;          //回溯，恢复原来的值
        nown=nown-1;
}
//当不要顶点k时，判断是否仍有解
if (nown+n-k > bestn ) {
        x[k]=0;              //进入右子树
        Backtrack(k+1); //继续递归处理
 }
    return;
```

最大团问题——实现

```
void Backtrack(int k){
    int j, OK;
    if(k>n) {    //叶子节点
        for (j=1; j<=n; j++)
            bestx[j]=x[j];
        bestn = nown;
        return;  }
    OK=1;
    //判断能否选择节点k
    for (j=1; j<k; j++)
      if (x[j]==1 && g[k-1][j-1]==0)
      {
          Ok=0;
          break;
      }
// 当顶点 k 能加入团中时
.....
}
```

```
// 当顶点 k 能加入团中时
if (OK){
        x[k]=1;          //进入左子树
        nown=nown+1;     //团的顶点数增 1
        Backtrack(k+1);  //继续递归处理
        x[k]=0;          //回溯，恢复原来的值
        nown=nown-1;
}
//当不要顶点k时，判断是否仍有解
if (nown+n-k > bestn ) {
        x[k]=0;              //进入右子树
        Backtrack(k+1); //继续递归处理
 }
    return;
```

最大团问题——实现

```
void Backtrack(int k){
    int j, OK;
    if(k>n) {    //叶子节点
        for (j=1; j<=n; j++)
            bestx[j]=x[j];
        bestn = nown;
        return;  }
    OK=1;
    //判断能否选择节点k
    for (j=1; j<k; j++)
      if (x[j]==1 && g[k-1][j-1]==0)
      {
            Ok=0;
             break;
      }
// 当顶点 k 能加入团中时
.....
}
```

```
// 当顶点 k 能加入团中时
if (OK){
        x[k]=1;          //进入左子树
        nown=nown+1;     //团的顶点数增 1
        Backtrack(k+1);  //继续递归处理
        x[k]=0;          //回溯，恢复原来的值
        nown=nown-1;
}
//当不要顶点k时，判断是否仍有解
if (nown+n-k > bestn ) {
        x[k]=0;              //进入右子树
        Backtrack(k+1); //继续递归处理
 }
    return;
```

最大团问题——实现

```
void Backtrack(int k){
    int j, OK;
    if(k>n) {    //叶子节点
        for (j=1; j<=n; j++)
            bestx[j]=x[j];
        bestn = nown;
        return;  }
    OK=1;
    //判断能否选择节点k
    for (j=1; j<k; j++)
      if (x[j]==1 && g[k-1][j-1]==0)
      {
            Ok=0;
             break;
      }
// 当顶点 k 能加入团中时
.....
}
```

```
// 当顶点 k 能加入团中时
if (OK){
        x[k]=1;          //进入左子树
        nown=nown+1;     //团的顶点数增 1
        Backtrack(k+1);  //继续递归处理
        x[k]=0;          //回溯，恢复原来的值
        nown=nown-1;
}
//当不要顶点k时，判断是否仍有解
if (nown+n-k > bestn ) {
        x[k]=0;              //进入右子树
        Backtrack(k+1); //继续递归处理
 }
    return;
```

时间复杂度分析：

- 显然，由于解空间树有2^n个叶节点，所以时间复杂度为$O(n2^n)$


SICHUAN UNIVERSITY
1896


DiCA
Data Intelligence & Computing Art Laboratory

End