中国科学技术大学 计算机科学与技术系
University of Science and Technology of China
Department of Computer Science and Technology

算法设计与分析

Design and Analysis of Algorithms

主讲人　徐云

Fall 2018, USTC

第2章(补充) 回溯法

2.1 方法概述

2.2 树和图的遍历

2.3 n后问题

2.4 排列生成问题

2.5 TSP问题

2.6 0-1背包

2.1 方法概述

- 搜索算法介绍
- 回溯算法
- 相关术语
- 回溯算法的基本步骤
- 实例分析

搜索算法介绍（1）

- 搜索算法

(1)穷举搜索(*Exhaustive Search*)

(2)盲目搜索(*Blind or Brute-Force Search*)

- 深度优先(*DFS*)或回溯搜索(*Backtracking*);
- 广度优先搜索(*BFS*);
- 迭代加深搜索(*Iterative Deepening*);
- 分枝限界法(*Branch & Bound*);
- 博弈树搜索(*α-β Search*)

(3)启发式搜索(*Heuristic Search*)

- A*算法和最佳优先(*Best-First Search*)
- 迭代加深的A*算法
- B*,AO*,SSS*等算法
- Local Search, GA等算法

搜索算法介绍（2）

- 搜索空间的三种表示
  - 表序表示：搜索对象用线性表数据结构表示；
  - 显式图表示：搜索对象在搜索前就用图(树)的数据结构表示；
  - 隐式图表示：除了初始结点，其他结点在搜索过程中动态生成. 缘于搜索空间大，难以全部存储.

搜索算法介绍（3）

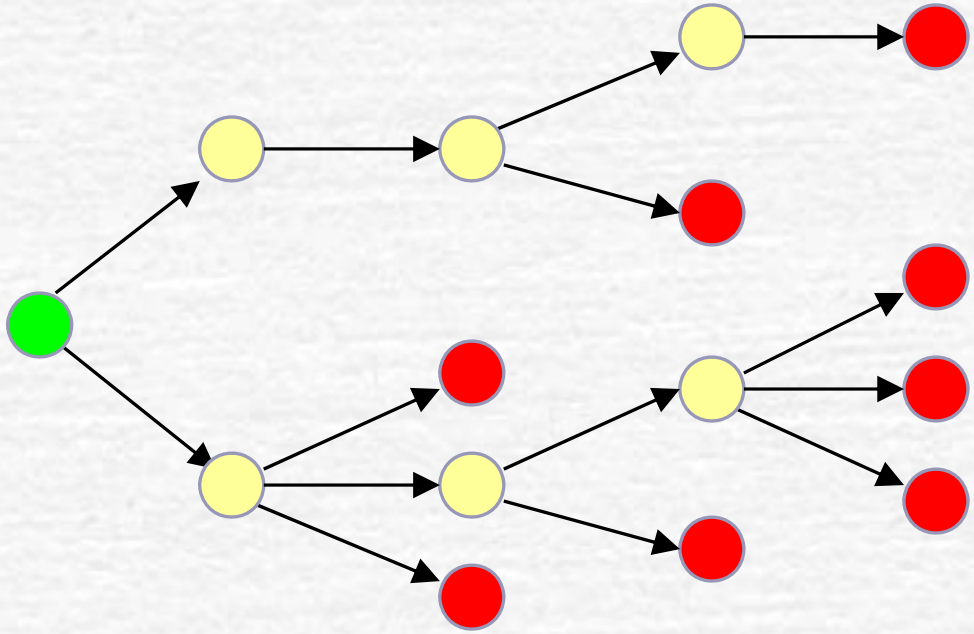
- 提高搜索效率的思考：随机搜索
  - 上世纪70年代中期开始，国外一些学者致力于研究随机搜索求解困难的组合问题，将随机过程引入搜索；
  - 选择规则是随机地从可选结点中取一个，从而可以从统计角度分析搜索的平均性能；
  - 随机搜索的一个成功例子：判定一个很大的数是不是素数，获得了第一个多项式时间的算法；

回溯算法

- 回溯法是一个既带有系统性又带有跳跃性的搜索算法。
- 它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。——系统性
- 算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以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，继续按深度优先的策略进行搜索。——跳跃性
- 这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题。

相关术语（1）

解空间树：



有三种结点：

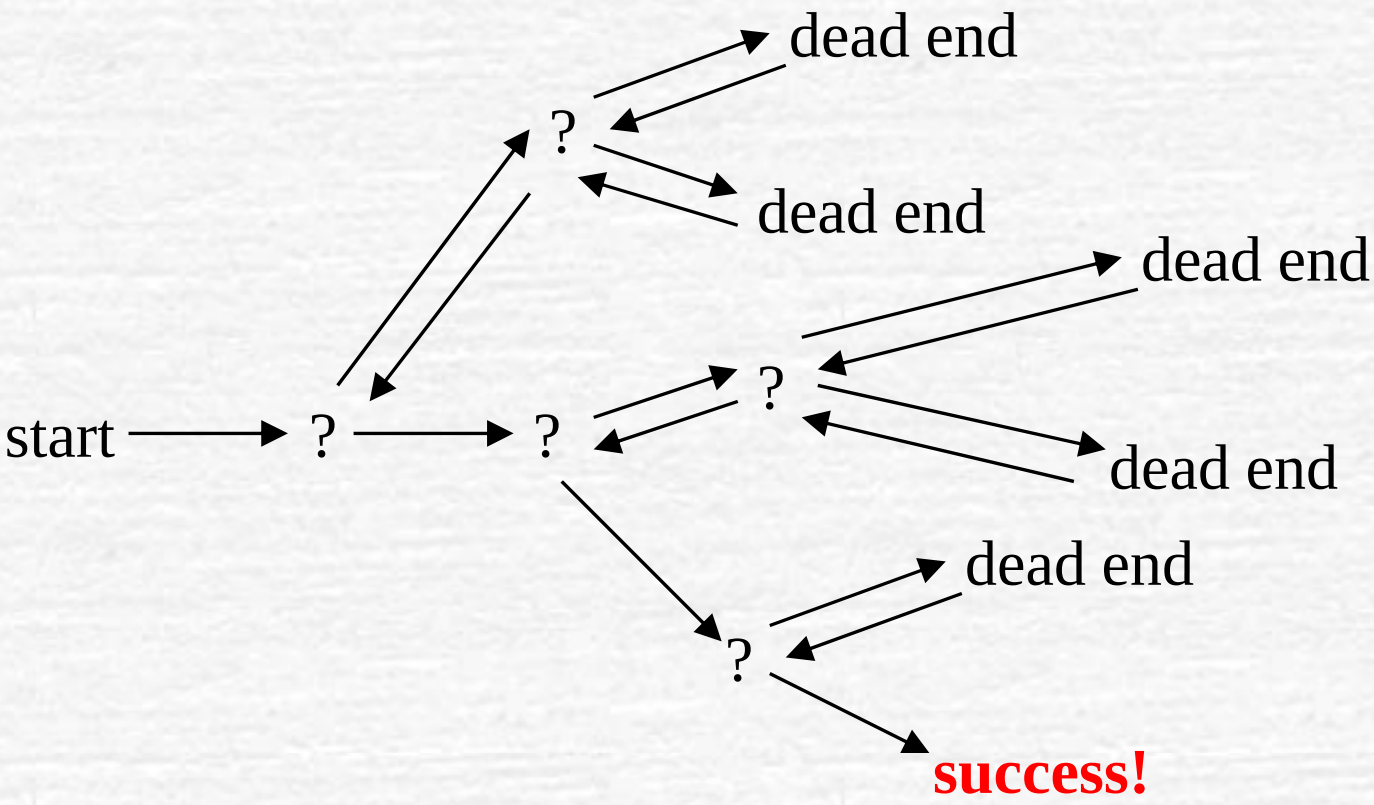
- 根结点（搜索的起点）
- 中间结点（非终端结点）
- 叶结点（终端结点），叶结点为解向量

搜索过程就是找一个或一些特别的叶结点。

相关术语（2）

- 搜索从开始结点（根结点）出发，以DFS搜索整个解空间。
- 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向纵深方向移至一个新结点，并成为一个新的活结点，也成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向扩展，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；直至找到一个解或全部解。

相关术语（3）



算法的基本步骤

(1) 针对问题，定义问题的解空间（对解进行编码）；

(2) 确定易于搜索的解空间组织结构（按树或图组织解）；

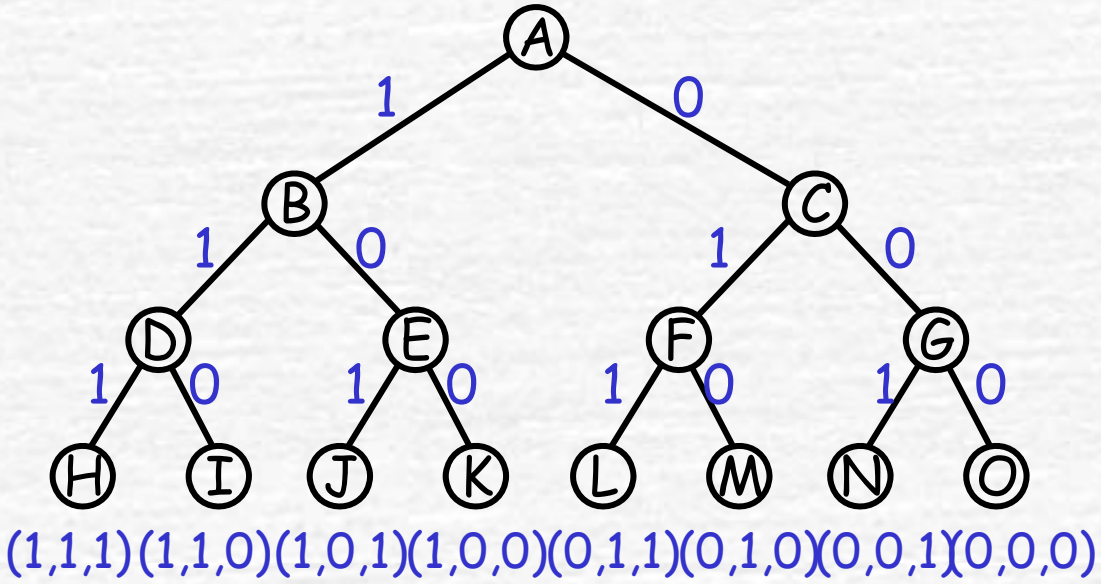
(3) 以深度优先方式搜索解空间，搜索过程中裁减掉死结点的子树提高搜索效率。

实例分析（1）

- 例1[0-1背包]：n=3, w=(16,15,15), v=(45,25,25), c=30

(1)定义解空间：X={(0,0,0),(0,0,1),(0,1,0),…,(1,1,0),(1,1,1)}

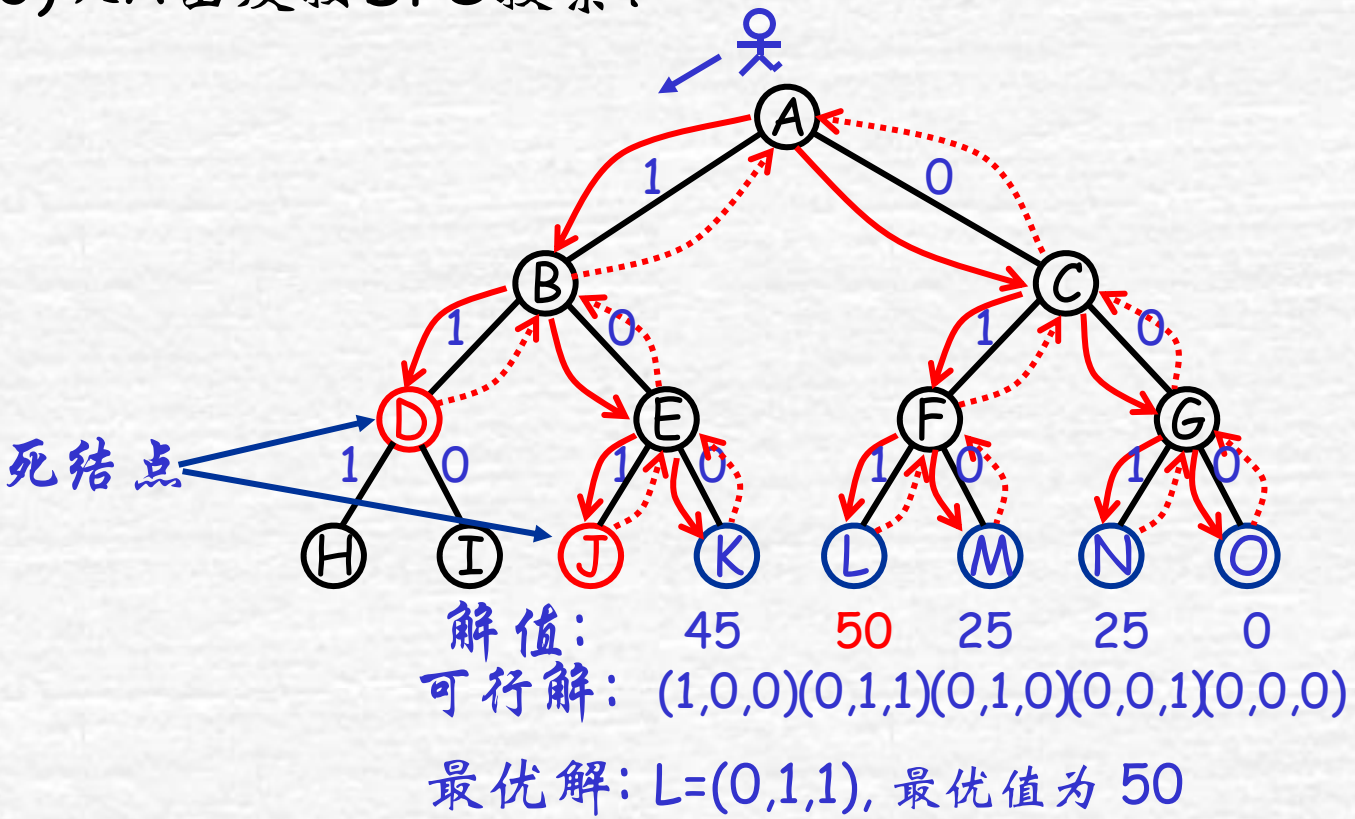
(2)构造解空间树：



实例分析（2）

- 例1(cont.)： n=3, w=(16,15,15), v=(45,25,25), c=30

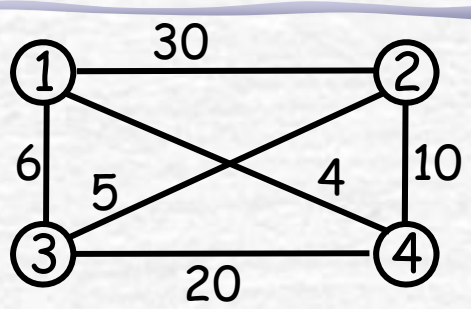
(3)从A出发按DFS搜索：



最优解：L=(0,1,1)，最优值为 50

实例分析（3）

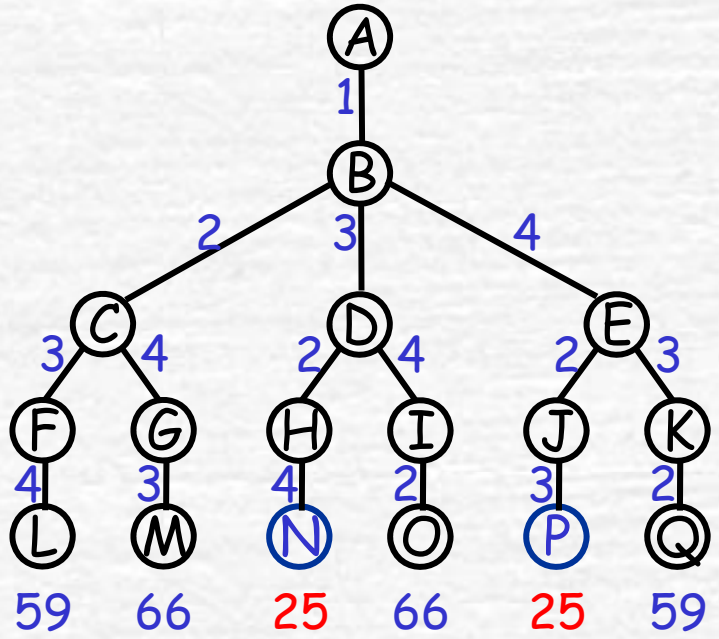
- 例2[TSP问题]



(1)定义解空间：X={12341,12431,13241,13421,14231,14321}

(2)构造解空间树：

(3)从A出发按DFS搜索整棵树：



最优解：13241，14231

成　本：25

实例分析（4）

- 注解：

（1）提高回溯法效率的二种方法

①用约束函数剪去不满足约束的子树；

②用限界函数剪去不能得到最优解的子树。

（2）二类常见的解空间树

①子集树：如0-1背包，叶结点数2^n，总结点数2^{n+1}，遍历时间为$\Omega(2^n)$；

②排列树：如TSP问题，叶结点数n!，遍历时间为$\Omega(n!)$。

第2章(补充) 回溯法

2.1 递归设计技术

2.2 树和图的遍历

2.3 n后问题

2.4 排列生成问题

2.5 TSP问题

2.6 0-1背包

2.2 树和图的遍历

- 二叉树的遍历
  - 先序遍历
  - 中序遍历
  - 后序遍历
  - 按层次遍历
- 图的遍历
  - 深度优先遍历
  - 广度优先遍历
  - 双向广度优先遍历

二叉树的遍历（1）

- 二叉链表的存储结构

```
typedef struct BiTnode {
  ElemType data;
  struct BiTnode *lchild, *rchild;
}*BiTree;
```

- 先序遍历

```
Preorder(BiTree T)
{//递归程序
   if T!=nil then
   {  visit(T);
      Preorder(T->lchild);
      Preorder(T->rchild);
    }
}
```

```
Preorder(BiTree T)
{//非递归程序
    if T=nil then return;
    inistack(S); push(S, T);
    while(!empty(S))  do
    {  BiTree p=pop(S);
       while(p!=nil)  do
       { visit(p); push(S, p->rchild);
          p=p->lchild;
       }
    }
}
```

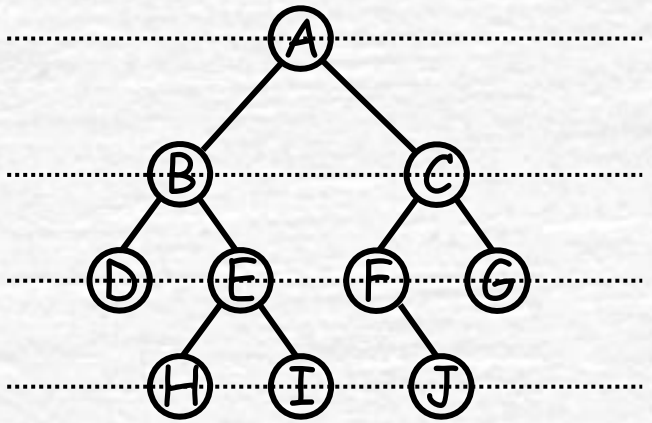
注：中序遍历（略）

后序遍历（略）

二叉树的遍历（2）

- 按层次遍历

```
BFSorder(BiTree T)
{
   if  T=nil then return;
   iniqueue(Q); enqueeu(Q, T);
   while(!empty(Q)) do
   {   BiTree p=Dequeue(Q);
       visit(p);
       if p->lchild != nil then
           enqueue(Q, p->lchild);
       if(p->rchild!=NIL)
           enqueue(Q, p->rchild);
    }
 }
```

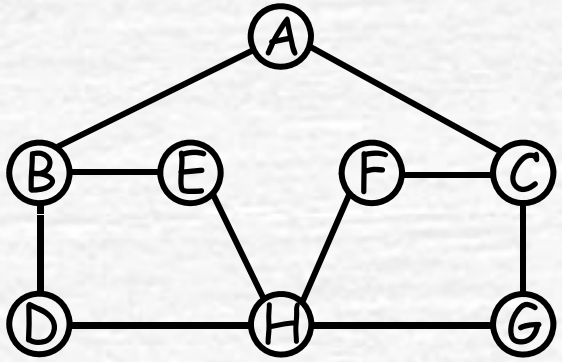


BFS遍历：ABCDEFGHIJ

图的遍历（1）

- 深度优先搜索（DFS）

```
DFS(v0)
{
   v0.visited=True;
   while(所有与v0邻接的顶点v and !v.visited ) do
      DFS(v);
   return;
}
```

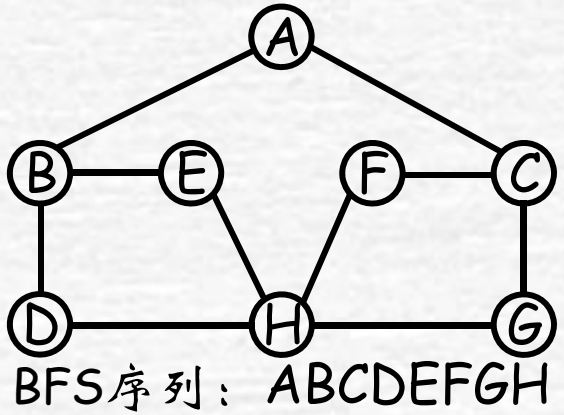


DFS序列：ABDHEFCG

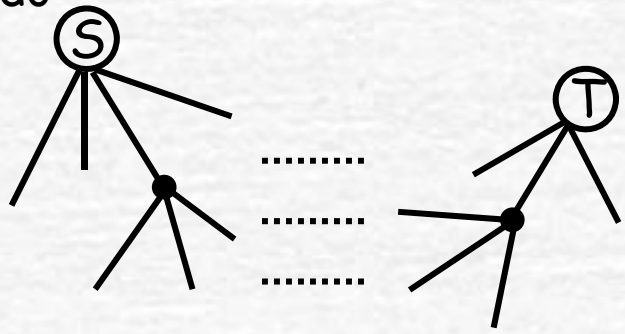
图的遍历（2）

- 广度优先搜索（BFS）

```
BFS(v0)
{  iniqueue(Q); enqueue(Q, v0);
   while( !empty(Q) )  do
   {   p=dequeue(Q);
       p.visited=True;
       while(所有与p邻接的顶点v && !v.visited ) do
           enqueue(Q, v);
   }
}
```



BFS序列：ABCDEFGH



- 双向广度优先搜索

对于迷宫等问题(S为入口,T为出口)，可以采用双向广度优先搜索。S正向搜索，T反向搜索，当两个方向搜索在某层上生成同一结点时，即找到一条路径。算法比单向搜索的结点数少得多。

第2章(补充) 回溯法

2.1 递归设计技术

2.2 树和图的遍历

2.3 n后问题

2.4 排列生成问题

2.5 TSP问题

2.6 0-1背包

2.3 n后问题

- 4皇后问题
  - 问题描述
  - 解表示
  - 搜索求解
- n后问题的算法
- 回溯算法的一般框架

4皇后（1）

- 问题描述

在4×4棋盘上放上4个皇后，使皇后彼此不受攻击。不受攻击的条件是彼此不在同行(列)、斜线上。求出全部的放法。

- 解表示

– 解编码：(x_1, x_2, x_3, x_4)4元组，x_i表示皇后i放在i行上的列号，如(3,1,2,4)

– 解空间：$\{(x_1, x_2, x_3, x_4) | x_i \in S, i=1\sim4\}$ $S=\{1,2,3,4\}$

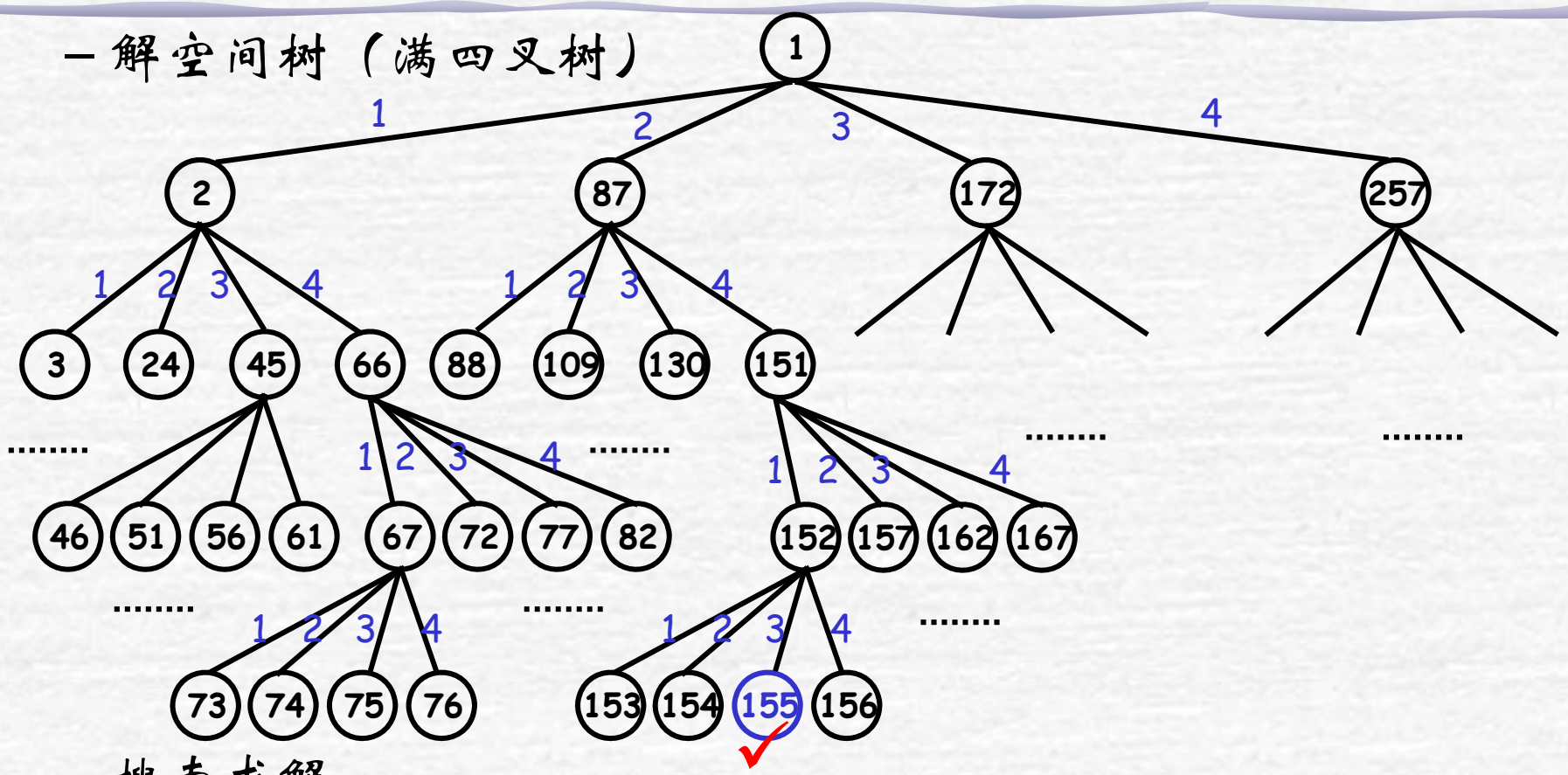
可行解满足：

显约束： $x_i \in S$, $i=1\sim4$

隐约束($i \neq j$)：
$$\begin{cases} x_i \neq x_j & \text{(不在同一列)} \\ |x_i - x_j| \neq |i-j| & \text{(不在同一斜线)} \end{cases}$$

4皇后（2）

- 解空间树（满四叉树）



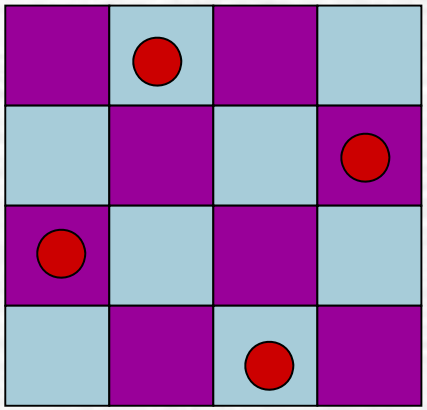
- 搜索求解：

从①起按DFS搜索，搜索时应满足隐约束，搜索到叶结点输出解：(2,4,1,3)，(3,1,4,2)

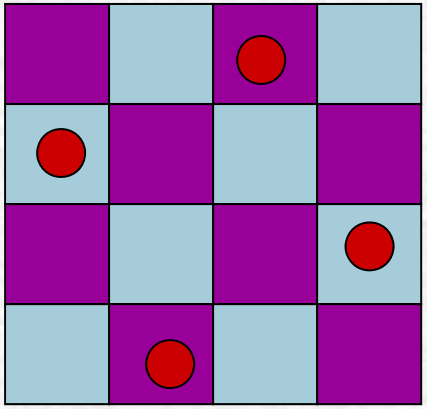
4皇后（3）

输出解：

(2,4,1,3)



(3,1,4,2)



n后问题的算法

- 递归算法

```
NQueen(int k)
{//由第K层向第K+1层扩展，确定x[k]的值
    if k>n then printf(x[1], … ,x[n]);     //搜索到叶结点输出解
    else
       for i=1 to n do
       {   x[k]=i;
           if placetest(k) then NQueen(k+1);
        }
 }
Placetest(int k)
{//检查x[k]位置是否合法
    for i=1 to k-1 do
        if ( x[i]=x[k] or abs(x[i]-x[k])=abs(i-k) ) then return false;
    return true;
 }
```

注：求解时执行NQueen(1)

回溯算法的一般框架（1）

- 子集树回溯算法

```
Backtrack(int t)  //搜索到树的第t层
{//由第t层向第t+1层扩展，确定x[t]的值
    if  t>n then output(x); //叶结点是可行解，输出解
    else
        while( all Xt)  do    // Xt为所有x[t]的合法取值集
        {    x[t]= Xt中第i个值;
             if( Constraint(t) and Bound(t) )
                 Backtrack(t+1);
        }
}
```

执行时： Backtrack(1)   //从1扩展并回溯

回溯算法的一般框架（2）

- 排列树回溯算法

```
Backtrack(int t)  //搜索到树的第t层
{//由第t层向第t+1层扩展，确定x[t]的值
    if t>n then output(x); //叶结点是可行解，输出解
    else
        for i=t to n do
        {    swap(x[t], x[i]);
             if( Constraint(t) and Bound(t) )
                 Backtrack(t+1);
             swap(x[t], x[i]);
         }
}
```

第2章(补充) 回溯法

2.1 递归设计技术

2.2 树和图的遍历

2.3 n后问题

2.4 排列生成问题

2.5 TSP问题

2.6 0-1背包

2.4 排列生成问题

- 问题定义
- 解空间树
- 回溯算法
- 回溯过程的验证

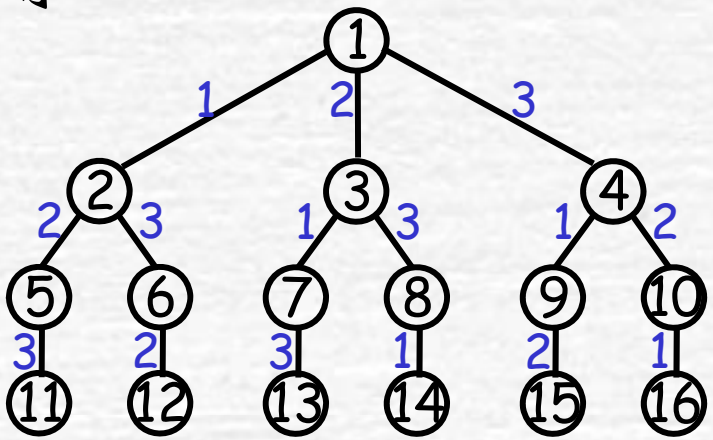
问题定义和解空间树

- 问题定义

  给定正整数n，生成1,2,…,n所有排列。

- 解空间树（排列树）

  当n=3时



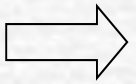
回溯算法和验证

- 回溯算法

```
Backtrack(int t)
{
   if t>n then output(x);
   else
     for i=t to n do
     {   swap(x[t], x[i]);
         Backtrack(t+1);
         swap(x[t], x[i]);
     }
}
main(int n)
{
    for i=1 to n do  x[i]=i;
    Backtrack(1);
}
```

- 对n=3的执行情况验证：

```
– Backtrack(1)
  Backtrack(2)
  x[1]↔x[2]
  Backtrack(2)
  x[1]↔x[2]
  x[1]↔x[3]
  Backtrack(2)
  x[1]↔x[3]
– Backtrack(2)
  Backtrack(3)
  x[2]↔x[3]
  Backtrack(3)
  x[2]↔x[3]
– Backtrack(3)
  Backtrack(4)
– Backtrack(4)
  output(x)
```



```
123
132
213
231
321
312
```

第2章(补充) 回溯法

- 2.1 递归设计技术
- 2.2 树和图的遍历
- 2.3 n后问题
- 2.4 排列生成问题
- 2.5 TSP问题
- 2.6 0-1背包

2.5 TSP问题

- 基本思想
- 回溯算法

基本思想

利用排列生成问题的回溯算法Backtrack(2)，对x[]={1,2,…,n}的x[2..n]进行全排列，则(x[1],x[2]),(x[2],x[3]),…,(x[n],x[1])构成一个回路。在全排列算法的基础上，进行路径计算保存以及进行限界剪枝。

回溯算法（1）

```
main(int n)
{//主程序
    a[n][n]; x[n]={1,2,…,n}; bestx[]; cc=0.0;
    bestv=∞;   //bestx保存当前最佳路径，bestv保存当前最优值
    input(a);          //输入邻接矩阵
    TSPBacktrack(2);
    output(bestv, bestx[]);
}
```

回溯算法（2）

```
TSPBacktrack(int i)
{//cc记录(x[1],x[2]), … ,(x[i-1],x[i])的距离和
   if i>n then    //搜索到叶结点，输出可行解与当前最优解比较
   {  if ( cc+a[x[n]][1]<bestv or bestv=∞ )  then
      {   bestv=cc+a[x[n]][1];
          for j=1 to n do   bestx[j]=x[j];
      }
   }
   else {
      for j=i to n do
         if ( cc+a[x[i-1]][x[j]]<bestv or bestv=∞ ) then  //限界裁剪子树
         {   swap(x[i], x[j]);
             cc+=a[x[i-1]][x[i]];
             TSPBacktrack(i+1);
             cc-=a[x[i-1]][x[i]];
             swap(x[i], x[j]);
         }
   }
}
```

第2章(补充) 回溯法

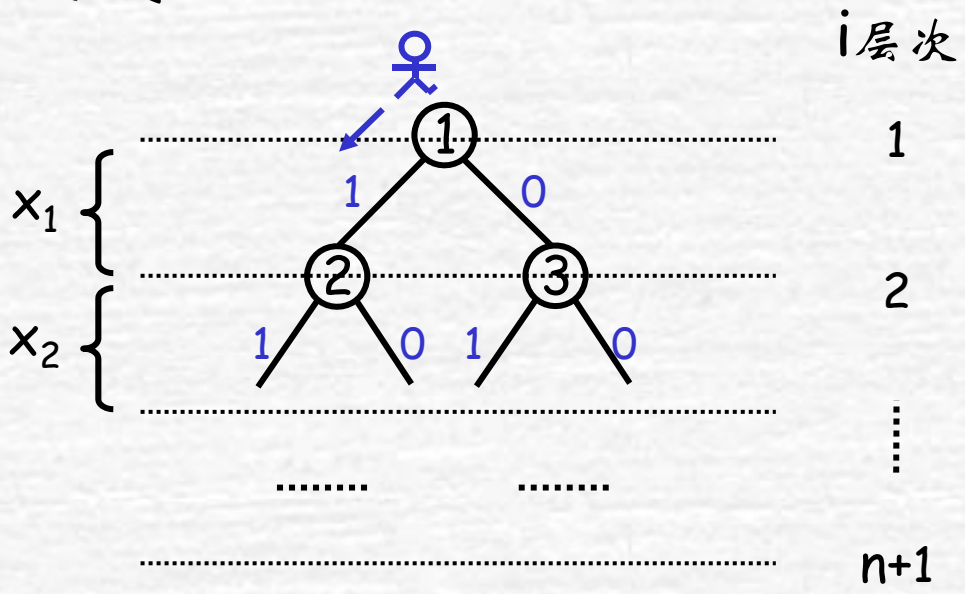
- 2.1 递归设计技术
- 2.2 树和图的遍历
- 2.3 n后问题
- 2.4 排列生成问题
- 2.5 TSP问题
- 2.6 0-1背包

2.6 0-1背包

- 问题描述（skipped）
- 解表示和解空间
- 解空间树
- 无限界函数的算法
- 有限界函数的算法

0-1背包（1）

- 问题描述（略）
- 解表示和解空间：$\{(x_1,x_2,\ldots,x_n) \mid x_i \in \{0,1\}, i=1\sim n\}$
- 解空间树



0-1背包（2）

- 无限界函数的算法

```
KnapBacktrack(int i)
{//cw当前背包重量，cv当前背包价值，bestv当前最优价值
    if i>n then {      //搜索到可行解
       bestv=(bestv<cv)?cv:bestv;
       output(x); }
    else { if  cv + av[i] < bestv: //剪枝    av是剩下的xi的v[xi]的和
           return
       if cw+w[i]<=c  then {  //走左子树
          x[i]=1; cw+=w[i]; cv+=v[i];
          KnapBacktrack(i+1);
          cw-=w[i]; cv-=v[i];
        }  //以下走右子树
       {  x[i]=0;
          KnapBacktrack(i+1);
        }
    }
}
```

```
main(float c, int n, float w[],
        float v[], int x[])
{ //主程序
    float cw=0.0, cv=0.0, bestv=0.0;
    KnapBacktrack(1);
}
```

0-1背包（3）

- 有限界函数的算法
  - 基本思想
    - ☞设r是当前扩展结点z的右子树(或左子树)的价值上界，如果cv+r≤bestv时，则可以裁剪掉右子树(或左子树)。
    - ☞一种简单的确定z的左、右子树最优值上界的方法（设z为第k层结点）：

      左子树上界= $_{i=k}\sum_{i=n}v_i$，右子树上界=$_{i=k+1}\sum_{i=n}v_i$，
  - 求经扩展结点z的可行解价值上界的方法
    - ☞计算至扩展结点的当前背包价值

      已知x_i, i=1~k-1，当前背包价值cv=$_{i=1}\sum_{i=k-1}v_ix_i$
    - ☞最后，

      经z左子树的可行解价值上界＝cv+左子树上界

      经z右子树的可行解价值上界＝cv+右子树上界
  - 算法（略）

End of SChap2